Lecture 2
Modelling in Datalog

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Introduce you to the main concepts around *modelling* and how these are concretely realised in the *Datalog* language.

Learning Outcomes

By the end of this lecture you will be able to:

1. Describe concepts such as *consistency*, *consequence*, and *database semantics*
2. Recall and recognise the syntax for Datalog
3. Define what a *fact* and *rule* are with respect to Datalog
4. Explain the meaning of *interpretation* in the formal sense

What is a Model?

What is a Model?

What is a Model?

Definition of Model

A simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions.

Let's Build Some Models

A red block is on top of a blue block.
There are 5 blocks in total.

Let's Build Some Models

A red block is on top of a blue block.
There are 5 blocks in total.

Domain

Some part of the world about which we wish to express some knowledge

Let's Build Some Models

A green block is on top of a blue block

Let's Build Some Models

A green block is on top of a blue block

Closed World Assumption

The only things that are true are the things that are stated (or derived)

Let's Build Some Models

A green block is on top of a blue block.
Smaller blocks can go on bigger blocks.

Let's Build Some Models

A green block is on top of a blue block.
Smaller blocks can go on bigger blocks.

Domain Closure Assumption

All elements of the domain are explicitly mentioned.

Let's Build Some Models

A green block is on top of a blue block.
Smaller blocks can go on bigger blocks
There is a red block.

Let's Build Some Models

A red block is on top of a blue block.
A blue block is on top of a red block.

Let's Build Some Models

A red block is on top of a blue block.
A blue block is on top of a red block.

Consistency

A model is consistent if there is at least one world that it models, otherwise it is inconsistent.

Let's Build Some Models

Block A is on top of a white block.
Block B is on top of a white block.
There are two yellow blocks.

Let's Build Some Models

Block A is on top of a white block.
Block B is on top of a white block.
There are two yellow blocks.

Unique Names Assumption

Things with different names are necessarily unique.

Database Semantics

Together the following assumptions describe Database Semantics. These are not true of logic in general and when they are assumed they change the semantics (the meaning) of a statement. Later (in a few weeks) we will show how they can be written directly in first-order logic.

Closed World Assumption

The only things that are true are the things that are stated (or derived)

Domain Closure Assumption

All elements of the domain are explicitly mentioned.

Unique Names Assumption

Things with different names are necessarily unique.

Getting Formal: Objects and Relations

We assume finite sets of object symbols $\mathcal{O}$ and relation symbols $\mathcal{R}$

A relation symbol $r \in \mathcal{R}$ has an arity given by $arr(r)$

(the arity of a relation is the number of arguments it applies to)

Getting Formal: Objects and Relations

We assume finite sets of object symbols $\mathcal{O}$ and relation symbols $\mathcal{R}$

A relation symbol $r \in \mathcal{R}$ has an arity given by $arr(r)$

(the arity of a relation is the number of arguments it applies to)

A fact is of the form $r(o_1, \ldots, o_{arr(r)})$ given $r \in \mathcal{R}$ and $o_i \in \mathcal{O}$

e.g. a relation symbol applied to the necessary number of objects

It is a convention that we write these symbols using lowercase

Relations are Tables. . .

. . . assuming Database Semantics

id	*name*	*course*
1	Adam	French
2	Raj	Fashion
3	Vlad	Music
4	Husain	Architecture
5	Sara	Engineering

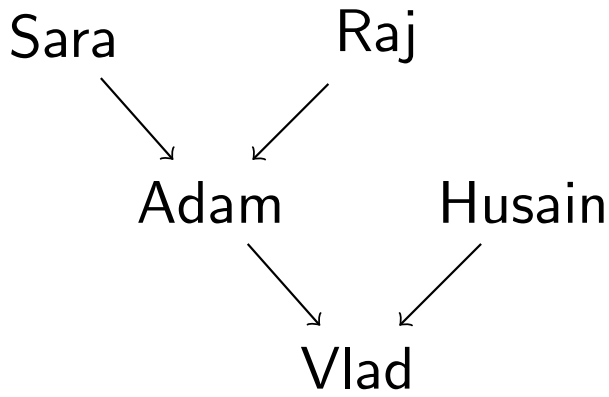
student(1, Adam, French)
student(2, Raj, Fashion)
student(3, Vlad, Music)
student(4, Husain, Architecture)
student(5, Sara, Engineering)

Datalog assumes Database Semantics

Relations are Trees/Graphs. . .

. . . but need additional rules to make $\rightarrow$ transitive, antisymmetric etc

child	*parent*
Adam	Sara
Adam	Raj
Vlad	Adam
Vlad	Husain



We'll revisit this later.

Properties

We often call unary properties (with arity 1) properties

For example, red(car), lecturer(giles), expensive(yacht).

Properties

We often call unary properties (with arity 1) properties

For example, red(car), lecturer(giles), expensive(yacht).

This places red(car), green(car), and fast(car) at the same conceptual level

Maybe colour should be a relationship between cars and colours e.g. color(car, red) and color(car, green)

Properties

We often call unary properties (with arity 1) properties

For example, red(car), lecturer(giles), expensive(yacht).

This places red(car), green(car), and fast(car) at the same conceptual level

Maybe colour should be a relationship between cars and colours e.g. color(car, red) and color(car, green)

Perhaps it is useful to treat the notion of properties uniformly e.g. hasProperty(car, red), hasProperty(yacht, expensive)

Properties

We often call unary properties (with arity 1) properties

For example, red(car), lecturer(giles), expensive(yacht).

This places red(car), green(car), and fast(car) at the same conceptual level

Maybe colour should be a relationship between cars and colours e.g. color(car, red) and color(car, green)

Perhaps it is useful to treat the notion of properties uniformly e.g. hasProperty(car, red), hasProperty(yacht, expensive)

The way we model things matters but there is not one best approach. Like database design, it is a bit of an art-form and there is good practice.

Although next time we will see how it can impact on reasoning

Rules

Define how new facts can be inferred from old facts

A rule is of the form $f_1, \ldots, f_n \Rightarrow f_{n+1}$ where f_i are *facts* ($n \geq 0$)

The meaning (semantics) is *if all the facts on the left (body) are true then the fact on the right (head) is true*

Rules

Define how new facts can be inferred from old facts

A rule is of the form $f_1, \ldots, f_n \Rightarrow f_{n+1}$ where f_i are *facts* ($n \geq 0$)

The meaning (semantics) is *if all the facts on the left (body) are true then the fact on the right (head) is true*

Facts in rules can contain variables (placeholders) in place of objects

Restriction: Let $var(f)$ be the variables in fact f then

$$var(f_{n+1}) \subseteq var(f_1) \cup \ldots \cup var(f_n)$$

e.g. the variables in the rule's conclusion must appear in the premises

Such rules can be seen as *templates* for ground (variable-free) rules. Due to domain closure a rule has a finite number of ground instances.

Facts and Rules Logically

If you are familiar with predicate/first-order logic then

Facts are predicates

Rules are definite clauses e.g. universally quantified disjunctions containing *exactly* one positive literal

The variable occurrence restriction is quite artificial and we'll see why it's useful next lecture

We'll return to this later in the course

Recursive Rules

It is a convention to write variables starting with an Uppercase

It is common to use rules to give recursively defined relations e.g.

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

(this is why Datalog is relational algebra + recursion)

Extensional and Intensional Relations

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

Example from Monday

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

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

Let's Model Something

Let's Model Something

Datalog has quite a few restrictions that are frustrating but disappear later with Prolog or full first-order logic.

Some things that seem like restrictions at first are circumvented by the database semantics.

By the Way: Datalog Syntax

You will probably see rules written as

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

That's proper Datalog... I'm using logical notation to get you used to it for later. The above is the syntax you'll see in Prolog.

Semantics

Let us call a collection of facts and rules a knowledge base

What is the *meaning* of a knowledge base i.e. its semantics?

What do we mean by this?
What are the set of facts that follow from the knowledge base.

For this we are going to turn to the heavyweight pursuit of *model theory*

Interpretations

An interpretation $\mathcal{I}$ is a function that assigns meaning (truth) to symbols, and hence the facts and rules that contain them

An interpretation $\mathcal{I}$ gives a truth value to each fact built from $\mathcal{O}$ and $\mathcal{R}$. If $\mathcal{I}$ applied to fact f is *true* then we say $\mathcal{I}$ satisfies f.

Interpretations

An interpretation $\mathcal{I}$ is a function that assigns meaning (truth) to symbols, and hence the facts and rules that contain them

An interpretation $\mathcal{I}$ gives a truth value to each fact built from $\mathcal{O}$ and $\mathcal{R}$. If $\mathcal{I}$ applied to fact f is *true* then we say $\mathcal{I}$ satisfies f.

A substitution is a *map* (function with finite domain) from variables to objects. We can apply a substitution to a fact containing variables to produce a new fact e.g. $\{X \mapsto a\}(f(X, Y)) = f(a, Y)$.

An interpretation $\mathcal{I}$ satisfies a rule $f_1, \ldots, f_n \Rightarrow f_{n+1}$ if for every substitution σ whenever $\mathcal{I}$ satisfies $\sigma(f_1), \ldots \sigma(f_n)$ it also satisfies $\sigma(f_{n+1})$.

Interpretations are different Realities

If our knowledge base is

$$\text{red}(\text{block1}) \qquad \text{blue}(\text{block2}) \qquad \text{onTop}(\text{red}, \text{blue})$$

then we can have multiple interpretations of the knowledge base

Interpretations are different Realities

If our knowledge base is

$$\text{red}(\text{block1}) \qquad \text{blue}(\text{block2}) \qquad \text{onTop}(\text{red}, \text{blue})$$

then we can have multiple interpretations of the knowledge base

Unless we have the closed world assumption

In which case, there is one minimal interpretation

Note that an interpretation can be defined exactly by the facts true in it

Consistency and Consequence

An interpretation satisfies a knowledge base if it satisfies all facts and clauses in the knowledge base.

A knowledge base is consistent if there is at least one interpretation that satisfies it.

A fact f is a consequence of the knowledge base if every interpretation that satisfies the knowledge base also satisfies f.

If a fact f is a consequence of a knowledge base $\mathcal{KB}$ we write $\mathcal{KB} \models f$

Aside: Symbols are just Symbols

$\text{rich}(\text{giles})$ $\quad$ $\text{rich}(X) \Rightarrow \text{happy}(X)$

Aside: Symbols are just Symbols

$$\text{rich}(\text{giles}) \qquad \text{rich}(X) \Rightarrow \text{happy}(X) \qquad \text{happy}(\text{giles})$$

Aside: Symbols are just Symbols

$\text{rich}(\text{giles})$ $\quad$ $\text{rich}(X) \Rightarrow \text{happy}(X)$ $\quad$ $\text{happy}(\text{giles})$

$\text{fruit}(\text{giles})$ $\quad$ $\text{fruit}(X) \Rightarrow \text{dancing}(X)$

Aside: Symbols are just Symbols

$\text{rich}(\text{giles})$ $\quad$ $\text{rich}(X) \Rightarrow \text{happy}(X)$ $\quad$ $\text{happy}(\text{giles})$

$\text{fruit}(\text{giles})$ $\quad$ $\text{fruit}(X) \Rightarrow \text{dancing}(X)$ $\quad$ $\text{dancing}(\text{giles})$

Aside: Symbols are just Symbols

$$\text{rich}(\text{giles}) \qquad \text{rich}(X) \Rightarrow \text{happy}(X) \qquad \text{happy}(\text{giles})$$

$$\text{fruit}(\text{giles}) \qquad \text{fruit}(X) \Rightarrow \text{dancing}(X) \qquad \text{dancing}(\text{giles})$$

The consequences are equivalently valid. The symbols don't care that they don't make sense. You have to make sure the knowledge base is *sensible*.

Summary

Warnings:

- I have been a little lazy with some bits that we will revisit properly later when considering reasoning in full first-order logic
- I'm treating Datalog relatively logically (rather than programatically) so have ignored some 'features'
- I'm avoiding the word *model* in the logical sense on purpose at the moment

The idea is to introduce you to the basic concepts in a simple setting

Next time: we now know what a knowledge base is, but how do we use it? We will look at what a query is and what the answer to a query is, and how to compute answers.

Lecture 3 Reasoning in Datalog

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Introduce you to the main concepts around *quering a knowledge base* and how these are concretely realised in the *Datalog* language.

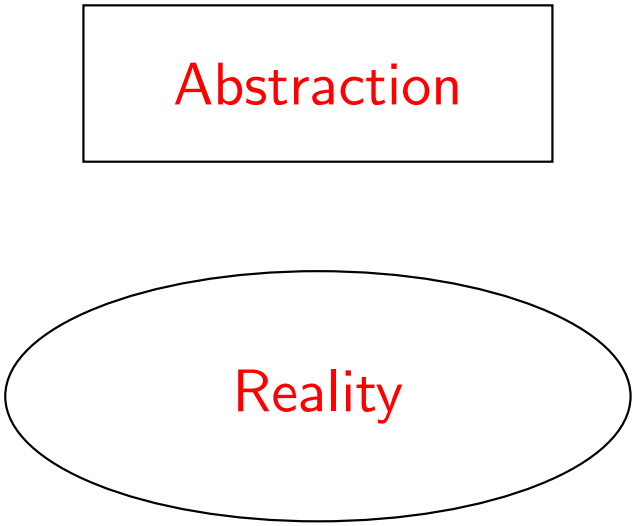
Learning Outcomes

By the end of this lecture you will be able to:

1. Describe what it means to *query* a knowledge base
2. Define *matching* and compute matching substitutions
3. Apply the *forward chaining* algorithm to find consequences of a knowledge base
4. Explain certain optimisations of the algorithm

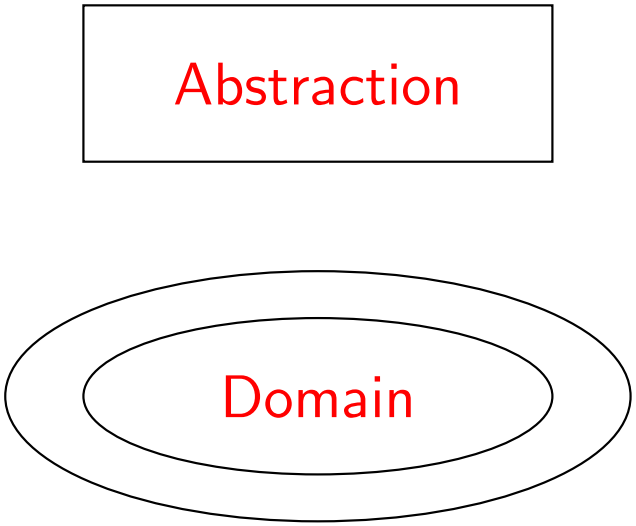
Recap

In General



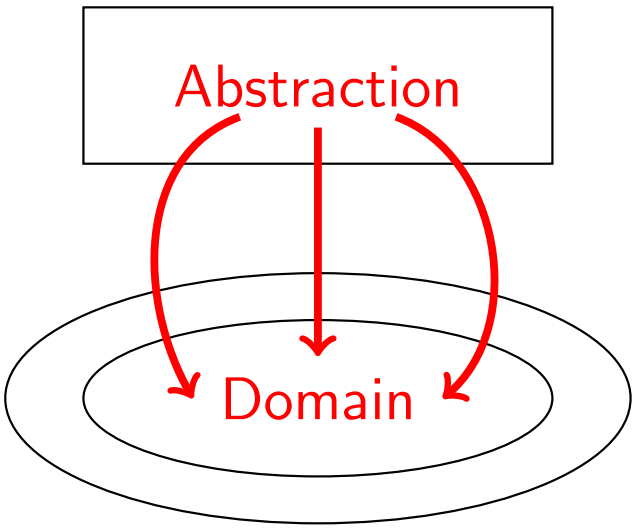
Recap

In General



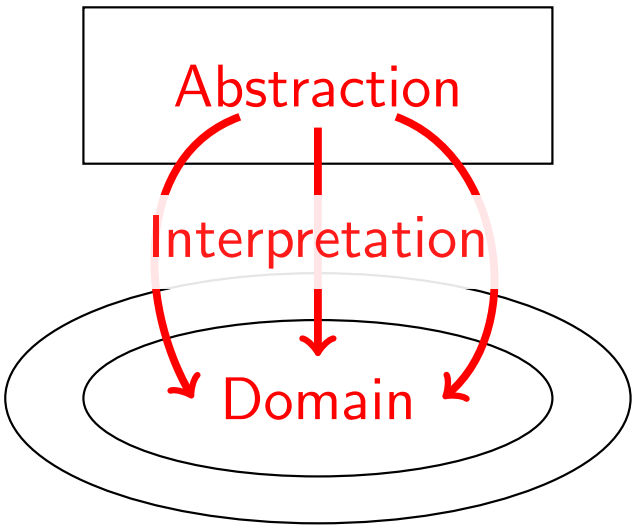
Recap

In General



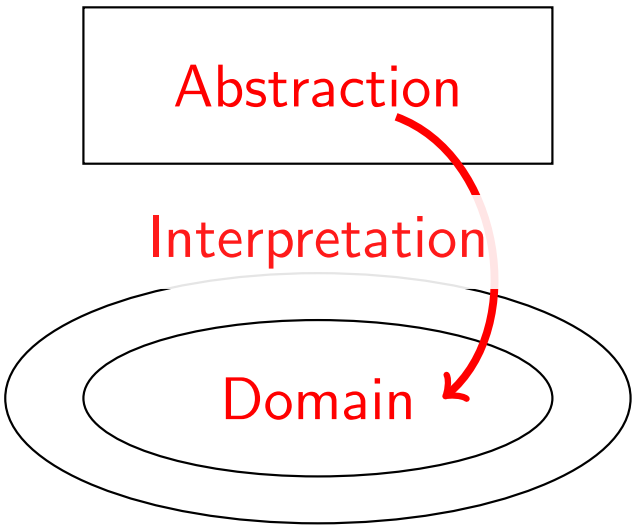
Recap

In General



Recap

In General

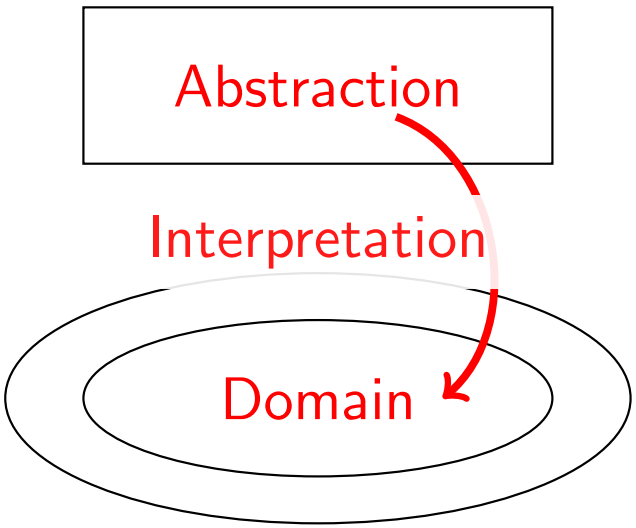


Database Semantics

- Closed World
- Domain Closure
- Unique Names

Recap

In General



Database Semantics

- Closed World
- Domain Closure
- Unique Names

Datalog

Has Database Semantics

Fact: concrete relationship between objects
e.g. $\mathsf{loves}(\mathsf{giles}, \mathsf{cheese})$

Rule: $\underbrace{\mathsf{loves}(X, Y), \mathsf{has}(X, Y)}_{body} \Rightarrow \underbrace{\mathsf{happy}(X)}_{head}$

Knowledge Base $\mathcal{KB}$: set of facts and rules

Fact f is a consequence of $\mathcal{KB}$
If all interpretations satisfying $\mathcal{KB}$ satisfy f
written $\mathcal{KB} \models f$

Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why?

Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)

Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)

Checking if f is a consequence of $\mathcal{KB}$ is decidable.

- An interpretation can be defined by the facts true in it
- Due to database semantics, $\mathcal{KB}$ has a single minimal interpretation $\mathcal{M}$ satisfying it. If a fact is satisfied by this it is a consequence of $\mathcal{KB}$
- The set of all facts built from $\mathcal{O}$ and $\mathcal{R}$ is finite, call this $\mathcal{A}$
- Clearly $\mathcal{M} \subseteq \mathcal{A}$; we can search all subsets of $\mathcal{A}$

Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)

Checking if f is a consequence of $\mathcal{KB}$ is decidable.

- An interpretation can be defined by the facts true in it
- Due to database semantics, $\mathcal{KB}$ has a single minimal interpretation $\mathcal{M}$ satisfying it. If a fact is satisfied by this it is a consequence of $\mathcal{KB}$
- The set of all facts built from $\mathcal{O}$ and $\mathcal{R}$ is finite, call this $\mathcal{A}$
- Clearly $\mathcal{M} \subseteq \mathcal{A}$; we can search all subsets of $\mathcal{A}$

Finally, can a Datalog knowledge base be inconsistent?

Properties of Datalog

Given a knowledge base there are a finite number of consequences

Why? Each rule has a finite number of instances (finite new facts)

Checking if f is a consequence of $\mathcal{KB}$ is decidable.

- An interpretation can be defined by the facts true in it
- Due to database semantics, $\mathcal{KB}$ has a single minimal interpretation $\mathcal{M}$ satisfying it. If a fact is satisfied by this it is a consequence of $\mathcal{KB}$
- The set of all facts built from $\mathcal{O}$ and $\mathcal{R}$ is finite, call this $\mathcal{A}$
- Clearly $\mathcal{M} \subseteq \mathcal{A}$; we can search all subsets of $\mathcal{A}$

Finally, can a Datalog knowledge base be inconsistent? No, the set of all consequences always exists and defines a satisfying interpretation.

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

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

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

The Semantics of Queries

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Computing the Set of Consequences

Given our initial set of facts $\mathcal{F}_0$ we want to add *new* consequences until we reach a fixed-point

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

Define the *next* set of facts as follows

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

This reaches a fixed point when $\mathcal{F}_j = \mathcal{F}_{j+1}$

As there are finite consequences this will terminate

Computing the Set of Consequences

Given our initial set of facts $\mathcal{F}_0$ we want to add *new* consequences until we reach a fixed-point

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

Define the *next* set of facts as follows

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

This reaches a fixed point when $\mathcal{F}_j = \mathcal{F}_{j+1}$

As there are finite consequences this will terminate

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

Matching

A fact is ground if it does not contain variables

Given a fact f_1 and a ground fact f_2 we say f_2 matches f_1 if there exists a substitution σ such that $f_2 = \sigma(f_1)$.

Examples:

Ground fact f_2 matches	fact f_1 using	substitution σ
happy(giles)	happy(X)	$\{X \mapsto \text{giles}\}$
loves(giles, cheese)	loves(X, cheese)	$\{X \mapsto \text{giles}\}$
loves(giles, cheese)	loves(X, Y)	$\{X \mapsto \text{giles}, Y \mapsto \text{cheese}\}$
happy(giles)	happy(giles)	$\{\}$

Note that loves(giles, cheese) does not match with loves(X, X).

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** $\sigma(args_1[i]) \neq args_2[i])$ **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.

Matching is an instance of unification, which we will meet later. In unification both sides can contain variables.

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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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$\}$

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** $\sigma(args_1[i]) \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

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

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

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** $\sigma(args_1[i]) \neq args_2[i])$ **then**
    - **return** $\perp$
- **end**
- **return** σ

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

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

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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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** $\sigma(args_1[i]) \neq args_2[i])$ **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
- $\sigma(args_1[0]) = \{X \mapsto$ bob$\}(X) =$ 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** $\sigma(args_1[i]) \neq args_2[i])$ **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, 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
```

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
```

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

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

```
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 $= \{\emptyset\}$

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

Matching A Rule Body

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

```
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

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

```
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 $= \emptyset$
$\sigma_1 = \emptyset$

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

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
```

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

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

$\text{new} = \left\{ \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \right\}$

$\sigma_1 = \emptyset$

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

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

Matching A Rule Body

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

```
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 $=$

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

$\sigma_1 = \emptyset$

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

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
```

matches $= \{\emptyset\}$

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

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

Matching A Rule Body

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

```
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 $= \left\{ \{X \mapsto \text{giles}, Y \mapsto \text{mark}\} \right\}$

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

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, 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 $=$

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

$\sigma_1 = \emptyset$

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

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

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

```
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
```

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

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

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

$\sigma_1 = \emptyset$

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

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

Matching A Rule Body

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

```
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
```

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

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

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

$\sigma_1 = \emptyset$

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

Matching A Rule Body

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

```
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
```

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

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

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

Matching A Rule Body

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

```
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
```

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

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

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

Matching A Rule Body

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

```
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
```

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

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

$\text{new} = \emptyset$

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

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 we do something cleverer

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \bot$ **then** $ans.add(\sigma)$
- **return** ans

Forward Chaining Algorithm

Compute the next set of consequences whilst there are new consequences. Search all consequences for facts matching the query.

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

- $\mathcal{F} = \emptyset$; $new = \mathcal{F}_0$
- **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** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$
    - **end**
  - **end**
- **while** $new \neq \emptyset$
- $ans = \emptyset$
- **for** $f \in \mathcal{F}$ **do** $\sigma = \text{match}(q, f, \emptyset)$; **if** $\sigma \neq \perp$ **then** $ans.add(\sigma)$
- **return** ans

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. We will see such a data structure for *unification* towards the end of the course

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

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.

Summary

Queries are facts possibly containing variables

To answer queries we can compute all consequences and check these

We can use forward chaining to compute consequences

This relies on matching, which can be tricky to implement efficiently

Next time: Prolog!

Lecture 4: Prolog

COMP24412: Symbolic AI

Martin Riener

School of Computer Science, University of Manchester, UK

February 2019

Overview

Outline

Prolog

- Programmation en logique - “Programming in logic”
- Declarative programming language:
  describe solution, not how to get there
- Based on automated theorem proving in FOL (SLD Resolution)
  easier to reason about
- Super-set of Datalog
- Turing-complete

History

- ∼1972: Colmerauer and Roussel define language, first implementation
- 1977: Warren writes first compiler (DEC-10 Prolog)
- 1983: Warren abstract machine (WAM)
- 1995: Becomes ISO/IEC 13211-1 standard
- 2000: Latest standard so far ISO/IEC 13211-2

Common Implementations

- Ciao Prolog
- Eclipse
- GNU Prolog
- IF Prolog
- Sicstus Prolog*
- SWI Prolog*
- XSB Prolog
- YAP Prolog

* available during exercise classes

Fields of use

- Prototyping
- Constraint Solving, Logistics
- Parsing, Natural Language Processing
- Search Problems with non-deterministic decisions

Use in Industry

- Query Engine of IBM Watson

  `https://www.theregister.co.uk/2009/04/27/ibm_watson_jeopardy?page=2`

- Clarissa (NASA): speech guided navigations through maintainence procedures on ISS

  `https://ti.arc.nasa.gov/tech/cas/user-centered-technologies/clarissa`

- $\sim 1/3$ of flight bookings in Europe handled by a Prolog system

  `https://www.sics.se/projects/sicstus-prolog-leading-prolog-technology`

Anatomy of a Prolog Program

- Program = Facts + Rules
- Query: "Is this fact derivable from the program?"
- Queries may contain variables
- Answer substitution:
  "Which variable assignments are necessary to derive the query?"
- Queries often have multiple answers!

Outline

Prolog terms

- Constant: basic object
- Variable: can be replaced by another term
- Predicate:
  - Rules define predicates
  - Never appear inside another term
  - There is no return value!

Prolog terms

- Predefined predicates:
  - `X = Y`: true if LHS and RHS are equal
    There is no assignment!
  - `dif(X,Y)`: true if LHS and RHS are different (not ISO)
- Function:
  - always appears inside a predicate
  - comparable to datastructures
  - There is no return value!

Prolog terms

```
band_song_date(rihanna, Song,date(Y,M,D))
```

- rihanna : Constant term
  Starts with a lower-case letter
- 'Twist and shout': Quoted constant term
- Song : Variable
  Starts with an upper-case letter
- _: Anonymous variable
  We will not be informed about assignments of this variable
- band_song_date/3 : Predicate (of arity 3) Starts with a lower-case letter
- date/3: function (of arity 3)

Queries

- Suppose we have a database of bands and their songs
- "Is Rihanna's Diamonds in the database?"

```
?- band_song(rihanna, diamonds).
false.
```

Queries

- "Do we know about any song by Rihanna?"

```
?- band_song(rihanna, Song).
false.
```

Queries

- "Is there anything in the database?"

```
?- band_song(Band, Song).
Band = beatles,
Song = 'While_my_guitar_gently_weeps' ;
Band = beatles,
Song = 'Twist_and_shout' ;
Band = beatles,
Song = 'Love_me_do'
% ....
```

Queries

- "There's surely more than The Beatles?"

```
?- dif(Band, beatles), band_song(Band, Song).
Band = 'Isley Brothers',
Song = 'Twist and shout' ;
Band = iggy,
Song = 'The passenger' ;
Band = banshees,
Song = 'The passenger' .
% ....
```

Queries

- "Which versions of The Passenger are there?"

```
?- Song = 'The passenger', band_song(Band, Song).
Song = 'The passenger',
Band = iggy ;
Song = 'The passenger',
Band = banshees ;
Song = 'The passenger',
Band = bauhaus.
```

Outline

Anatomy of a query

```
 ?- dif(Band, beatles), band_song(Band, Song).
Band = 'Isley Brothers',
Song = 'Twist and shout';
Band = iggy,
Song = 'The passenger' a
```

- ?- Query prompt
- , Conjunction of queries
- . End of query
- Band = 'Isley Brothers', Song = 'Twist and shout': Answer substitution
- ; User input (next answer)
- a User input (abort)

Turning a query into a rule

- Query

```
?- dif(Band, beatles), band_song(Band, Song).
```

- Rule

```
nobeatles_song(Band, Song) :-
  dif(Band, beatles),
  band_song(Band, Song).
```

Anatomy of a rule

```
head(X,Y,Z) :-
  goal1(X,A),
  goal2(Y,B),
  goal3(A,B,Z).
```

- "Derive head **if** goal1 **and** goal2 **and** goal3 are derivable."
- Predicate logic formula:

$$\forall X, Y, Z, A, B.$$
$$goal1(X, A) \wedge goal2(Y, B) \wedge goal3(A, B, Z)$$
$$\rightarrow head(X, Y, Z)$$

Facts

A fact is always true:

```
coldplace(siberia) :-
  true.
```

Easier to write:

```
coldplace(siberia).
```

Outline

Substitutions

Substitution:

- Maps finitely many variables to terms
- All other variables are mapped to themselves
- Apply $\sigma = \{$`X=car, Y=house`$\}$ to `owns(lucia,X)` and obtain `owns(lucia,car)`
- Apply σ to `owns(lucia,Z)` and obtain `owns(lucia,Z)`
- Substitutions can be composed:
  $\tau = \{$`pair(X,Y)`$\}$
  $\tau\sigma = \{$ `pair(car,house)` $\}$

Unification

```
?- X=1, X=2.
false.
```

Why?

Unification

```
?- X=1, X=2.
false.
```

Why?

- assign X = 1:
  1=1, 1=2.

Unification

```
?- X=1, X=2.
false.
```

Why?

- assign X = 1:
  1=1, 1=2.
- assign X = 2:
  1=2, 2=2.

Are these terms unifiable?

- `contains(X, milk) = contains(capuccino, Y)`

Are these terms unifiable?

- `contains(X, milk) = contains(capuccino, Y)`

  yes
- `contains(X, house) = contains(house, X)`

Are these terms unifiable?

- `contains(X, milk) = contains(capuccino, Y)`
  yes
- `contains(X, house) = contains(house, X)`
  yes
- `contains(X, milk) = contains(capuccino, X)`

Are these terms unifiable?

- `contains(X, milk) = contains(capuccino, Y)`
  yes
- `contains(X, house) = contains(house, X)`
  yes
- `contains(X, milk) = contains(capuccino, X)`
  no
- `climate(X) = climate(Y)`
  yes

Unification Problem

Unification Problem

Given a set of term equalities $s_1 = t_1, \ldots, s_n = t_n$, is there a unifying substitution σ such that for each equation s=t, sσ and tσ are the same terms?
If yes, which one?

Unification Problem

Unification Problem

Given a set of term equalities $s_1 = t_1, \ldots, s_n = t_n$, is there a unifying substitution σ such that for each equation s=t, sσ and tσ are the same terms?
If yes, which one?

Unifiers for the problems before:

- `X=cappucino, Y=milk`
- `X=house, Y=X`
- not unifiable
- `X=Y`

Unification Rules

Transformation rules:

- trivial: $t = t, P$
  delete $t = t$, solve P
- orient: $t = X, P$
  move variable to LHS: $X = t, P$
- decomposition: $f(s1, \ldots, sn) = f(t1, \ldots, tn), P$
  solve $s1 = t1, \ldots, sn = tn, P$
- variable elimination: $X = t, P$
  replace all occurrences of X in P with t, solve P

Unification Rules

Solved form:

- P contains only equations $X = a, Y = b, \ldots$
- No Transformation rule can be applied

Failure cases:

- name clash (constants): $c = d$
- name clash (functions): $f(A, B) = g(X, Y)$
- occurs check: $X = f(X)$
  (X occurs nested inside RHS term)

Unification: examples

- $contains(X, milk) = contains(capuccino, Y)$
  - Decompose: $X = capuccino, milk = Y$
  - Orient: $X = capuccino, Y = milk$
  - Solved!

Unification: examples

- $contains(X, house) = contains(house, X)$
  - Decompose: $X = house, house = X$
  - Eliminate X: $house = house$
  - Remove trivial
  - Solved!

Unification: examples

- $contains(X, milk) = contains(capuccino, X)$
  - Decompose: $X = capuccino, milk = X$
  - Eliminate X: $milk = capuccino$
  - Constant clash
  - Failure!

Most general unifiers

Problem: $f(X) = f(Y)$ has infinitely many unifiers:

```
X=a, Y=a
X=b, Y=b
X=c, Y=c
...
X=f(a), Y=f(a)
X=g(a), Y=g(a)
...
X=Y
```

Most general unifiers

More general substitutions

Let σ and τ be substitutions. If there exists a non-trivial substitution λ such that $\sigma\lambda = \tau$ then σ is *more general* than τ.

Most general unifiers

More general substitutions

Let σ and τ be substitutions. If there exists a non-trivial substitution λ such that $\sigma\lambda = \tau$ then σ is *more general* than τ.

Most general unifier

A unifier is a *most general* substitution if there is no other unifier that is more general.

Most general unifiers

More general substitutions

Let σ and τ be substitutions. If there exists a non-trivial substitution λ such that $\sigma\lambda = \tau$ then σ is *more general* than τ.

Most general unifier

A unifier is a *most general* substitution if there is no other unifier that is more general.

- $\{$X=Y$\}$ is more general than $X = a$ (take $\lambda = \{$Y=a$\}$)
- $\{$X=b$\}$ neither more nor less general than $\{$X=a$\}$

Most general unifiers

More general substitutions

Let σ and τ be substitutions. If there exists a non-trivial substitution λ such that $\sigma\lambda = \tau$ then σ is *more general* than τ.

Most general unifier

A unifier is a *most general* substitution if there is no other unifier that is more general.

Applying the unification algorithm presented, we always obtain the most general unifier (up to renaming of variables).

Summary

- Prolog is a Turing complete, logic based programming language
- Queries to Prolog program yields a sequence of answer substitutions
- Answers are found by backward chaining, trying the rules in order of appearance
- Suitable rules to apply are found via unification
- Functions allow the expression of arbitrary large terms, e.g. lists

That's all for today!

Lecture 5: Prolog Programming Techniques

COMP24412: Symbolic AI

Martin Riener

School of Computer Science, University of Manchester, UK

February 2019

What happened so far

- Prolog is a Turing complete, logic based programming language
- Queries to Prolog program yields a sequence of answer substitutions
- Answers are found by backward chaining, trying the rules in order of appearance
- Suitable rules to apply are found via unification
- Functions allow the expression of arbitrary large terms, e.g. lists

Overview

Outline

Finding good predicate names

- Programs are written to a file (queries happen at the prompt)
- All rules defining the same predicate must appear in succession
- Programs are loaded via `consult('program.pl').`
  Careful: editors might mistake the extension for Perl

Finding good predicate names

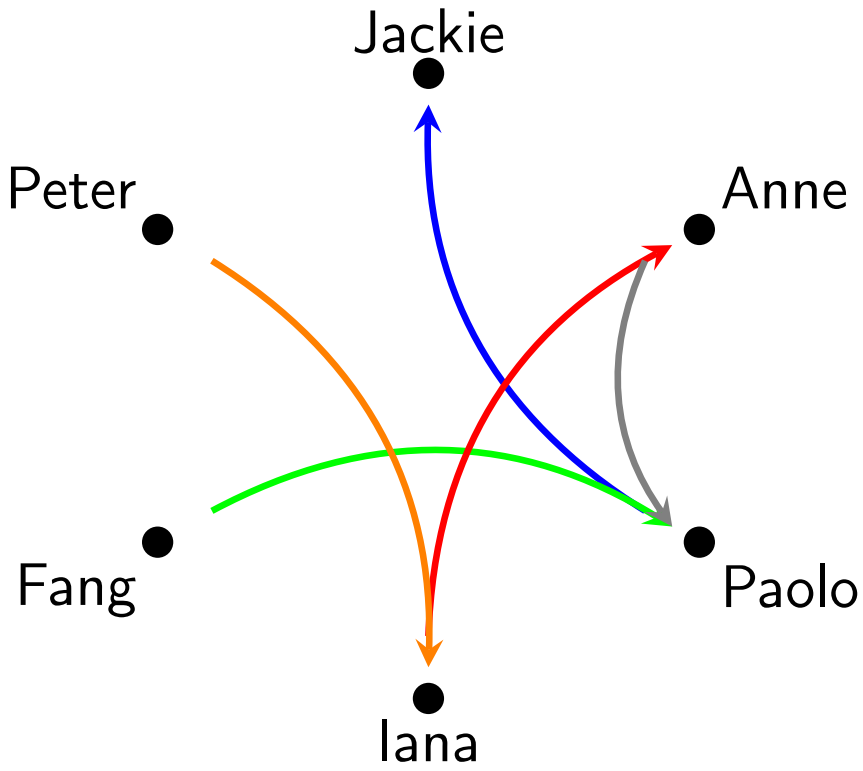
- Programs are written to a file (queries happen at the prompt)
- All rules defining the same predicate must appear in succession
- Programs are loaded via `consult('program.pl').`
  Careful: editors might mistake the extension for Perl

- Predicates describe relations, don't assume a direction of evaluation:
  Compare `find(car, List)` to `member_of(car, List)`
- Use short phrases for each argument, use `_` to seperate them:
  Compare `flight(X,Y,Z)` to `flightno_from_to(X,Y,Z)`

Recursively defined predicates: inductive reasoning

- Formulate simplest facts
- Find rules that extend smaller terms to (slightly) larger terms

A ball game



Representing the graph

```
child_throwsto(peter, iana).
child_throwsto(iana, anne).
child_throwsto(fang, paolo).
child_throwsto(paolo, jackie).
child_throwsto(anne, paolo).
```

Tossing the ball around

```
ballfrom_reaches(From,To) :-
    child_throwsto(From,To).
```

Tossing the ball around

```
ballfrom_reaches(From,To) :-
    child_throwsto(From,To).
ballfrom_reaches(From,To) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches(Neighbour,To).
```

Tossing the ball around

Why not?

```
ballfrom_reaches(From,To) :-
    ballfrom_reaches(From,Neighbour),
    ballfrom_reaches(Neighbour,To).
```

- No constraint on the solution in the first recursion step
- Leads to an infinite recursion

Outline

Prolog's execution mechanism

- Prolog applies backwards reasoning (start with the query)
- Multiple goals are derived left-to-right
- Search for unifiers with rule heads, top-to-bottom
- Head unifies:
  - try to derive the goals of the rule body
  - continue fulfilling the original goal
- No head unifies: backtrack and try next rule!

A simple query

```
?- ballfrom_reaches(iana,paolo).
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true ;
false.
```

A simple query

```
?- ballfrom_reaches(iana,paolo).
true ;
false.
```

What is happening?

Step-by-step execution

goal(s)

```
ballfrom_reaches(iana,paolo)
```

trying rule

```
ballfrom_reaches(From,To) :-
      child_throwsto(From,To).
```

Step-by-step execution

goal(s)

```
ballfrom_reaches(iana,paolo)
```

trying rule

```
ballfrom_reaches(From,To) :-
      child_throwsto(From,To).
```

instance: `From=iana, To=paolo`

```
ballfrom_reaches(iana,paolo) :-
      child_throwsto(iana,paolo).
```

Step-by-step execution

goal(s)

`child_throwsto(iana,paolo)`

---

trying rule

`child_throwsto(peter, iana).`

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`
trying rule
`child_throwsto(peter, iana).`
instance: `iana` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)

`child_throwsto(iana,paolo)`

---

trying rule

`child_throwsto(iana, anne).`

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`

trying rule

`child_throwsto(iana, anne).`

instance: `paolo` $\neq$ anne – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`
trying rule
`child_throwsto(fang, paolo).`
instance: `iana` $\neq$ `fang` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`
trying rule
`child_throwsto(paolo, jackie).`
instance: `iana` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,paolo)`
trying rule
`child_throwsto(anne, paolo).`
instance: `iana` $\neq$ `anne` – backtrack!
no more `child_throwstos` – backtrack!

Step-by-step execution

goal(s)

`child_throwsto(iana,paolo)`

---

trying rule

`child_throwsto(anne, paolo).`

Step-by-step execution

goal(s)

```
ballfrom_reaches(iana,paolo)
```

trying rule

```
ballfrom_reaches(From,To) :-
      child_throwsto(From,Neighbour),
      ballfrom_reaches(Neighbour,To).
```

Step-by-step execution

goal(s)

```
ballfrom_reaches(iana,paolo)
```

trying rule

```
ballfrom_reaches(From,To) :-
      child_throwsto(From,Neighbour),
      ballfrom_reaches(Neighbour,To).
```

instance: From=iana, To=paolo

```
ballfrom_reaches(iana,paolo) :-
      child_throwsto(iana,Neighbour),
      ballfrom_reaches(Neighbour,paolo).
```

Step-by-step execution

goal(s)

```
child_throwsto(iana,Neighbour),
ballfrom_reaches(Neighbour,paolo)
```

trying rule

```
child_throwsto(peter, iana).
```

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`
trying rule
instance: `iana` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)

```
child_throwsto(iana,Neighbour),
ballfrom_reaches(Neighbour,paolo)
```

trying rule

```
child_throwsto(iana, anne).
```

instance: Neighbour=anne – next goal!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`
trying rule
`child_throwsto(peter, iana).`
instance: `anne` $\neq$ `peter` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`
trying rule
`child_throwsto(iana, anne).`
instance: `anne` $\neq$ `iana` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`
trying rule
`child_throwsto(fang, paolo).`
instance: anne $\neq$ fang – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`
trying rule
`child_throwsto(paolo, jackie).`
instance: `anne` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)

`child_throwsto(anne,paolo)`

---

trying rule

`child_throwsto(anne, paolo).`

instance: no substitution needed

Step-by-step execution

goal(s)

---

no goals! tell the user!

Step-by-step execution

goal(s)

---

user wants more solutions, backtrack!

Step-by-step execution

goal(s)
`child_throwsto(anne,paolo)`
trying rule
no more `child_throwstos` – backtrack!

Step-by-step execution

goal(s)

`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`

---

trying rule
`child_throwsto(fang, paolo).`
instance: `iana` $\neq$ `fang` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`
trying rule
instance: `iana` $\neq$ `paolo` – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),` `ballfrom_reaches(Neighbour,paolo)`
trying rule instance: `iana` $\neq$ anne – backtrack!

Step-by-step execution

goal(s)
`child_throwsto(iana,Neighbour),`
`ballfrom_reaches(Neighbour,paolo)`
trying rule
etc. etc. etc.

Step-by-step execution

goal(s)

---

all paths exhausted, report false!

Outline

Can we make the steps visible?

- Introduce an additional argument:

```
ballfrom_reaches_via(From,To,direct(To)) :-
    child_throwsto(From,To).
ballfrom_reaches_via(From,To,next(Neighbour,Others)) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches_via(Neighbour,To,Others).
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(iana,paolo, Path).
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(iana,paolo, Path).
Path = next(anne, direct(paolo)) ;
false.
```

Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(From,To, next(A,next(B,direct(C)))).
```

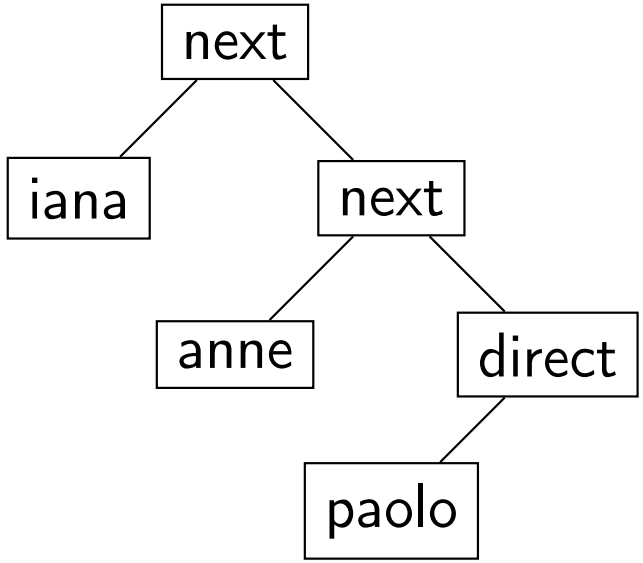
Can we make the steps visible?

- Query:

```
?- ballfrom_reaches_via(From,To, next(A,next(B,direct(C)))).
From = peter,
To = C, C = paolo,
A = iana,
B = anne ;
From = iana,
To = C, C = jackie,
A = anne,
B = paolo ;
false.
```

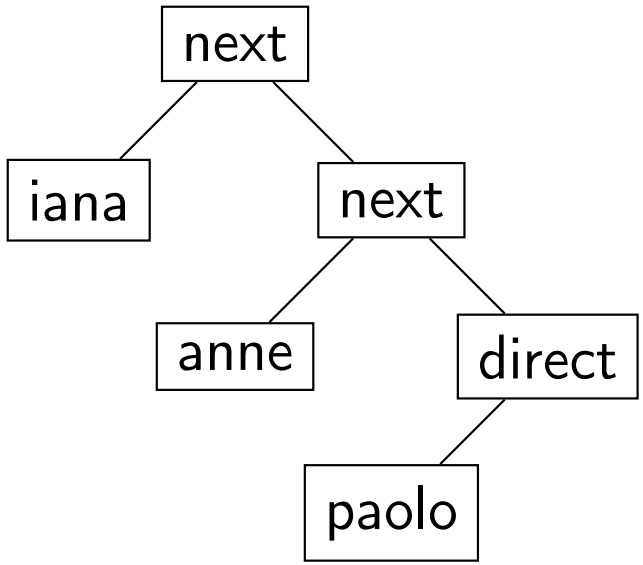
Data-structures: Lists

- We used `next/2` and `direct/1` to track paths
- Term graph of `next(iana, next(anne,direct(paolo)))`:



Data-structures: Lists

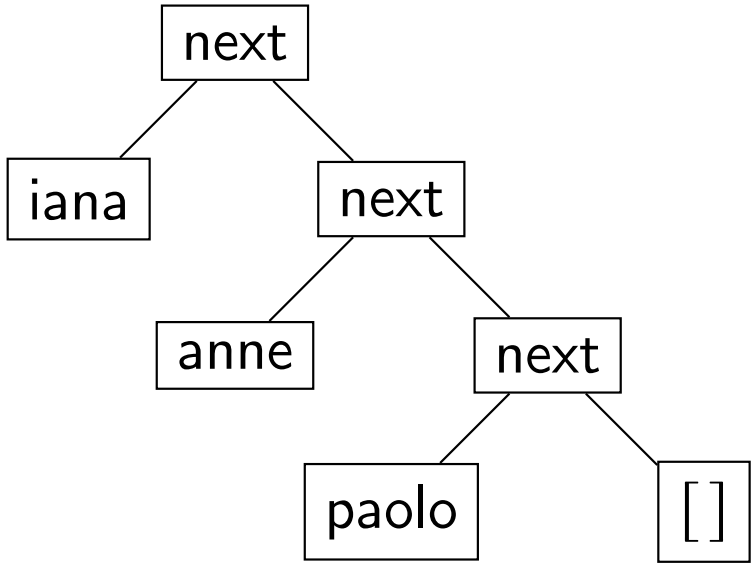
- We used `next/2` and `direct/1` to track paths
- Term graph of `next(iana, next(anne,direct(paolo)))`:



- What about an empty path?

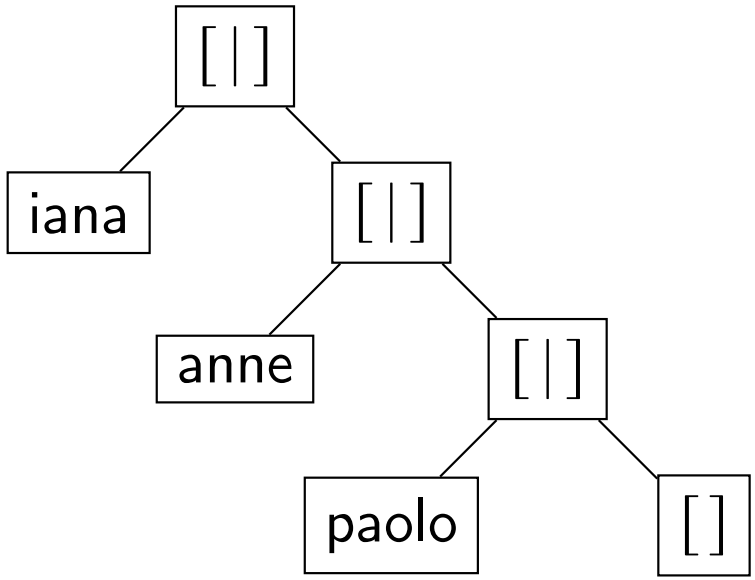
Data-structures: Lists

- Replace `direct/1` with `[]/0`
- Term graph of `next(iana, next(anne,next(paolo,[])))` :



Data-structures: Lists

- Rename `next/2` to `[|]/2`,
- Term graph of `[iana | [anne | [paolo | [] ]]]` :



Data-structures: Lists

- Structure built over []/0 and [|]/2: linked list
- Datatype:
  - "The empty list is a list"
    ```
    isa_list([]).
    ```
  - "Any head prepended to a tail list is a list"
    ```
    isa_list([Head | Tail]) :-
          isa_list(Tail).
    ```
- Special list notation:
  - No need to append to empty list:
    ```
    [Head | []] = [Head]
    ```
  - Use , to avoid writing the tail in squarebrackets:
    ```
    [X | [Y | Tail] ] = [X, Y | Tail]
    ```

Properties of linked lists

- Access to head: $O(1)$
  `List = [Head | _ ]`
- Access to tail: $O(1)$
  `List = [_ | Tail ]`
- Traversal: $O(n)$
  `member_of(X,List)`

Outline

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[Head|_Tail]) :- % X is member of a list
      X = Head.              % if X is the head of the list
```

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

- Find recursive case(s):
  "Given a smaller term, how to extend it to a larger one?"

Recursively defined predicates: member_of/2

- Task:
  Create a predicate `member_of(X,List)` that is true whenever X is an element of list $List$.
- Find base case(s):

```
member_of(X,[X|_Tail]). % directly unify in head
```

- Find recursive case(s):
  "Given a smaller term, how to extend it to a larger one?"

```
member_of(X,[_Head|Tail]) :- % X is member of the list
    member_of(X, Tail).      % if X is member of the tail
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

- Find recursive case(s):

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

- Find recursive case(s):

```
nonmember_of(X,[Head|Tail]) :-
    % fill in constraint
    nonmember_of(X, Tail). % X is not in the tail
```

Recursively defined predicates: nonmember_of/2

- Task:
  Create a predicate `nonmember_of(X,List)` that is true whenever X is *not* an element of list $List$.
- Find base case(s):

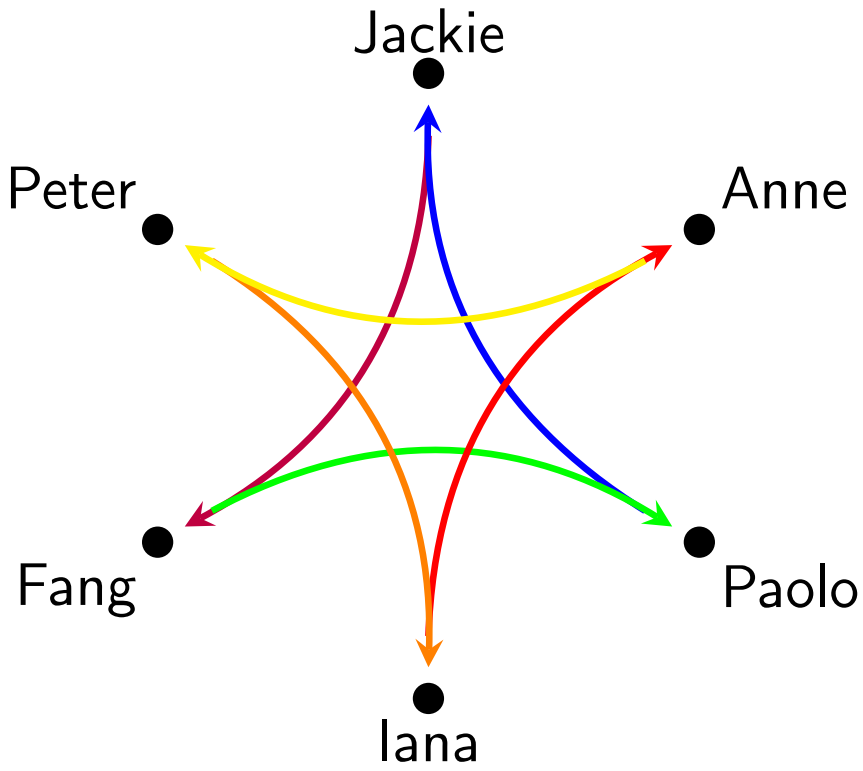
```
nonmember_of(_X,[]).  % Any element is not in the empty list
```

- Find recursive case(s):

```
nonmember_of(X,[Head|Tail]) :-
    dif(X,Head),           % X is different from the head
    nonmember_of(X, Tail). % X is not in the tail
```

Outline

More ball games



Representing the graph

```
child_throwsto(anne, peter).
child_throwsto(peter, iana).
child_throwsto(iana, anne).
child_throwsto(jackie, fang).
child_throwsto(fang, paolo).
child_throwsto(paolo, jackie).
```

Tossing the ball around

```
ballfrom_reaches_lvia(From,To,[To]) :-
    child_throwsto(From,To).
ballfrom_reaches_lvia(From,To,[Neighbour|Path]) :-
    child_throwsto(From,Neighbour),
    ballfrom_reaches_lvia(Neighbour,To,Path).
```

Some Queries

```
?- ballfrom_reaches_lvia(jackie, paolo,Path).
Path = [fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo, jackie, fang, paolo] ;
Path = [fang, paolo, jackie, fang, paolo, jackie, fang, paolo, jackie|...] ;
```

- Does it ever stop?

Some Queries

- Does it ever stop?
- Try to append `, false.` to the query:

```
?- ballfrom_reaches_lvia(jackie, paolo,Path), false.
% Hit Ctrl+C
Action (h for help) ? abort
% Execution Aborted
```

Some Queries

- Does it ever stop?
- The `,false` enforces backtracking, visiting all possible answers
- Finite solution space: interpreter returns `false`
  Remember: $P \wedge \bot \rightarrow \bot$ for any P in FOL
- Infinite answer sequence: infinite derivation (non-termination)
- Infinite set of answers leads to non-termination

Non-termination spreads

- Consider

```
ballfrom_reaches2(From, To) :-
    ballfrom_reaches_lvia(From, To, _Path).
```

  - Even worse:

```
?- ballfrom_reaches2(jackie, anne).
% unreachable, but explores infinitely long paths (non-termination)
```

Non-termination spreads

- Consider

```
ballfrom_reaches2(From, To) :-
    ballfrom_reaches_lvia(From, To, _Path).
```

- Execution:

```
?- ballfrom_reaches2(jackie, paolo).
true ;
true ;
true ;
true    % abort

?- ballfrom_reaches2(jackie, paolo), false.
% non-termination
```

  - Even worse:

```
?- ballfrom_reaches2(jackie, anne).
% unreachable, but explores infinitely long paths (non-termination)
```

Reordering goals can change termination behaviour

```
?- Xs = [something], isa_list(Xs).
Xs = [something].

?- isa_list(Xs), Xs = [something].
Xs = [something] ;
% loops enumerating all lists
```

Consequences of non-termination

- A recursive goal without instantiated variables always loops
- Putting always terminating predicates as early goals often helps (Restricts what gets passed to recursive goals)
- Properties of pure, monotonic Prolog programs (no non-logical elements, no negation)
  - Generalizing a non-terminating rule / query can not lead to termination
    e.g.: `isa_list([1,2,3|Xs])` to `isa_list(Xs)`
  - Instatiating a rule query can improve termination
    e.g. `isa_list([1,2,3|Xs])` to `isa_list([1,2,3|notalist])`
  - Removing goals from a rule can only increase the number of solutions

Outline

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    % ...
```

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    % ...
aballfrom_reaches_lvia_acc(From,To,[Neighbour | Others], Acc) :-
    child_throwsto(From,Neighbour),
    % ...
    aballfrom_reaches_lvia_acc(Neighbour,To,Others, [Neighbour|Acc]).
```

Improving the termination behaviour of the ball game

- Queries cannot terminate with an infinite set of paths
- Idea: consider only acyclic paths (fintely many for finite graphs)
- Add additional argument to pass history to recursive goals

```
aballfrom_reaches_lvia_acc(From,To,[To],Acc) :-
    child_throwsto(From,To),
    nonmember_of(To, Acc).
aballfrom_reaches_lvia_acc(From,To,[Neighbour | Others], Acc) :-
    child_throwsto(From,Neighbour),
    nonmember_of(From, Acc),
    aballfrom_reaches_lvia_acc(Neighbour,To,Others, [Neighbour|Acc]).
```

Hiding the accumulator

- This is still too general:

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, Acc).
Path = [fang, paolo],
Acc = [] ;
Path = [fang, paolo],
Acc = [_1204],
dif(_1204, paolo),
dif(_1204, jackie) ;
% ...
```

Hiding the accumulator

- This is still too general:

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, Acc).
Path = [fang, paolo],
Acc = [] ;
Path = [fang, paolo],
Acc = [_1204],
dif(_1204, paolo),
dif(_1204, jackie) ;
% ...
```

- Accumulator can have an arbitrary tail – start with an empty Acc

Hiding the accumulator

- Accumulator can have an arbitrary tail – start with an empty Acc

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, []).
Path = [fang, paolo] ;
false.

?- aballfrom_reaches_lvia_acc(jackie, anne, Path, []).
false.
```

Hiding the accumulator

- Accumulator can have an arbitrary tail – start with an empty Acc

```
?- aballfrom_reaches_lvia_acc(jackie, paolo, Path, []).
Path = [fang, paolo] ;
false.

?- aballfrom_reaches_lvia_acc(jackie, anne, Path, []).
false.
```

- Hide the accumulator from the user

```
aballfrom_reaches_lvia(From, To, Neighbour) :-
    aballfrom_reaches_lvia_acc(From, To, Neighbour, []).
```

Hiding the accumulator

- The new predicate always terminates:

```
?- aballfrom_reaches_lvia(From, To, Path), false.
false.
```

Summary

- Prolog's execution order weakens logical properties
- Swapping the order of goals does not influence the set of solutions but termination properties may change
- Appending `false` is a simple check for termination
- Narrowing answer set is an easy way to improve termination properties
- Accumulators pass information about the current goals to recursive goals

That's all for today!

Lecture 6 Prolog Programming Techniques

COMP24412: Symbolic AI

Martin Riener

School of Computer Science, University of Manchester, UK

February 2019

What happened so far

- We learned how Prolog executes derivations
- Simple predicates like transitive closure can lead to non-termination
- Thinking about the set of answers / number of substitutions may explain non-termination
- Sometimes reordering goals helps
- Sometimes we reformulate the problem

Overview

Outline

Arithmetic

- Unification is not computation:

```
?- X = 1+(2*3).
X = 1+(2*3).
```

- `is/2` relates ground arithmetic expressions to evaluation:

```
?- X is 1+(2*3).
X = 7.
```

- Other predicates: <,>, =<, >=, =:=

Arithmetic

- Problem:

```
?- 7 is 1+(X*Y).
ERROR: is/2: Arguments are not sufficiently instantiated
 ?- X < 10.
ERROR: </2: Arguments are not sufficiently instantiated
```

- Computation must be possible when goal is encountered
- Declarative properties (commutativity of goals) are lost:

```
?- X < 10, X=1.
ERROR: </2: Arguments are not sufficiently instantiated
?- X=1, X < 10.
X = 1.
```

Outline

Finite Domain Constraints – CLP(FD)

- Reasoning with constraints over integer domain:

```
?- 7 #= 1+(X*Y).
X in -6.. -1\/1..6,
X*Y#=6,
Y in -6.. -1\/1..6.
```

"The equation holds under the conditions:
$X \in \{-6 \ldots -1\} \cup \{1 \ldots 6\}$
$Y \in \{-6 \ldots -1\} \cup \{1 \ldots 6\}$
$X * Y = 6$"

Finite Domain Constraints – CLP(FD)

- Concrete substitutions require enumeration of variables:

```
?- 7 #= 1+(X*Y), labeling([], [X,Y]).
X = -6,
Y = -1 ;
% ...
```

Finite Domain Constraints – CLP(FD)

- Arithmetic Relations: #=, #\= #<,#>, #=<, #>=
- Domain Relations:
  - `Var in Lower .. Upper`: $X \in \{Lower \ldots Upper\}$
    `Lower`, `Upper` must be numbers or `inf` / `sup`
  - `[A,B,C] ins Lower .. Upper`: like `in` but for lists of variables
- Labeling: `label/1` expects a list of variables to label

Example: Magic Squares

Problem

Given a 3x3 square of fields, assign each of the numbers 1 to 9 to the fields such that the sums of each row, the sums of each column and the sums of the diagonals amount to the same value.

Example: Magic Squares

```
:- use_module(library(clpfd)).

rows_sum([A1,A2,A3,B1,B2,B3,C1,C2,C3], Sum):-
    A1+A2+A3 #= Sum,
    B1+B2+B3 #= Sum,
    C1+C2+C3 #= Sum.

cols_sum([A1,A2,A3,B1,B2,B3,C1,C2,C3], Sum):-
    A1+B1+C1 #= Sum,
    A2+B2+C2 #= Sum,
    A3+B3+C3 #= Sum.
```

Example: Magic Squares

```
diag_sum([A1,_A2,A3,_B1,B2,_B3,C1,_C2,C3], Sum):-
    A1+B2+C3 #= Sum,
    A3+B2+C1 #= Sum.

magicsquare(Sum,[A1,A2,A3],[B1,B2,B3],[C1,C2,C3]) :-
    % define domain variables
    Zs = [A1,A2,A3,B1,B2,B3,C1,C2,C3],
    Zs ins 1..9,
    % core predicates
    all_distinct(Zs),
    rows_sum(Zs, Sum),
    cols_sum(Zs, Sum),
    diag_sum(Zs, Sum),
    % labeling
    label([Sum|Zs]).
```

The structure of a CLP(FD) program

- Define a list Zs of finite domain variables to label
- Set the domain for the variables
- Add constraints on core predicates
  Core predicates do not label on their own!
- Finally: Label Zs

Why label only in the end?

- CLP(FD) maintains a set of constraints
- Adding new constraints allows constraint propagation:

```
?- X in 2..6, X #> 3.
X in 4..6.
```

- Labeling grounds the constraint, barely any propagation.
  Compare

```
?- time((X in 1..1000, label([X]), X #> 950, false)).
% 68,119 inferences, 0.006 CPU in 0.006 seconds (99% CPU, 11166330 Lips)
false.
```

  to

```
?- time((X in 1..1000, X #> 950, label([X]), false)).
% 3,527 inferences, 0.001 CPU in 0.001 seconds (94% CPU, 4958534 Lips)
false.
```

Why label at all?

- Constraints are not guaranteed to be satisfiable
- Only labeling guarantees their satisfiability
- Example:

```
?- X #< Y, Y #< X.
Y#=<X+ -1,
X#=<Y+ -1.
```

but there are not X,Y s.t. $X < Y \wedge Y < X$.

Labeling strategies

- `labeling/2`:
  Like `label/1` but first argument has list of options
  - Variable selection:
    `leftmost` (order of appearance), `ff` (order by domain size),
    `ffc` (ff, prefer by number of occurrences),
    `min` (order by smallest lower bound),
    `max` (order by largest upper bound),
  - Value order
    `up` (ascending order), `down` (descending order)
  - Branching strategy:
    `step` (distinguish equal / different from picked value),
    `enum` (distinguish all possible values at the same time),
    `bisect` (divide search space along middle point)

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Calculate Sum:
    Formula for $n \times n$: $Sum = \frac{n^3+n}{2}$
    15 for n=3

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Calculate Sum:
    Formula for $n \times n$: $Sum = \frac{n^3+n}{2}$
    15 for n=3

```
?- time((magicsquare(N, R1, R2, R3),false)).
% 1,523,920 inferences, 0.151 CPU in 0.152 seconds
false.

?- time((magicsquare(15, R1, R2, R3),false)).
% 341,873 inferences, 0.049 CPU in 0.050 seconds
false.
```

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Add symmetry breaking constraints:

2	7	6
9	5	1
4	3	8

original

4	3	8
9	5	1
2	7	6

horizontally flipped
A1 < C1

6	7	2
1	5	9
8	3	4

vertically flipped
A1 < A3

2	9	4
7	5	1
6	3	8

diagonally flipped
B1 < A2

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Add symmetry breaking constraints:

```
symmetries([A1,A2,A3,B1,_B2,_B3,C1,_C2,_C3]) :-
    A1 #< A3,
    A1 #< C1,
    A2 #< B1.
```

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Add symmetry breaking constraints:

```
symm_magicsquare(N, [A1,A2,A3],[B1,B2,B3],[C1,C2,C3]) :-
    % ...
    symmetries(Zs),
    % ...
```

Improving performance

- Try different labeling strategies
- Reduce the number of solutions!
  Magic squares example:
  - Add symmetry breaking constraints:

```
?- time((magicsquare(15, R1, R2, R3),false)).
% 341,873 inferences, 0.032 CPU in 0.032 seconds (100% CPU, 10587328
false.

?- time((symm_magicsquare(15, R1, R2, R3),false)).
% 96,382 inferences, 0.010 CPU in 0.010 seconds (100% CPU, 9494353 Li
false.
```

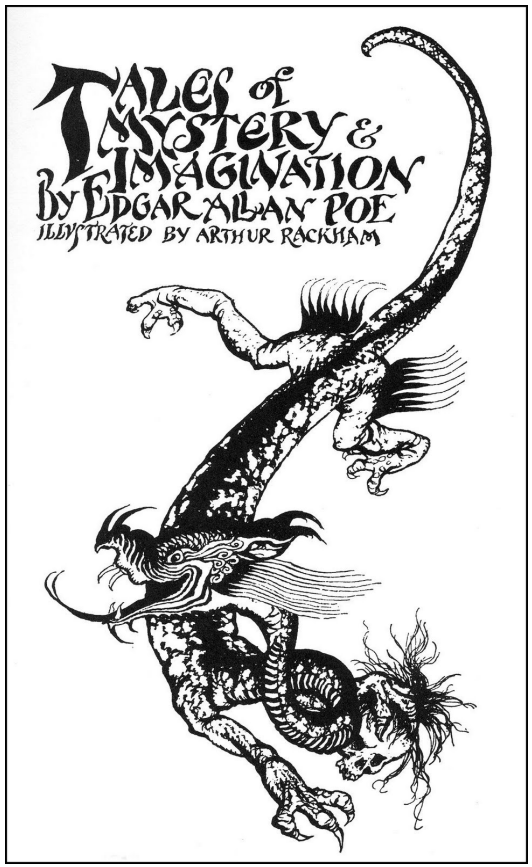
Improving performance

- Try different labeling strategies
- Reduce the number of solutions!

Lesson

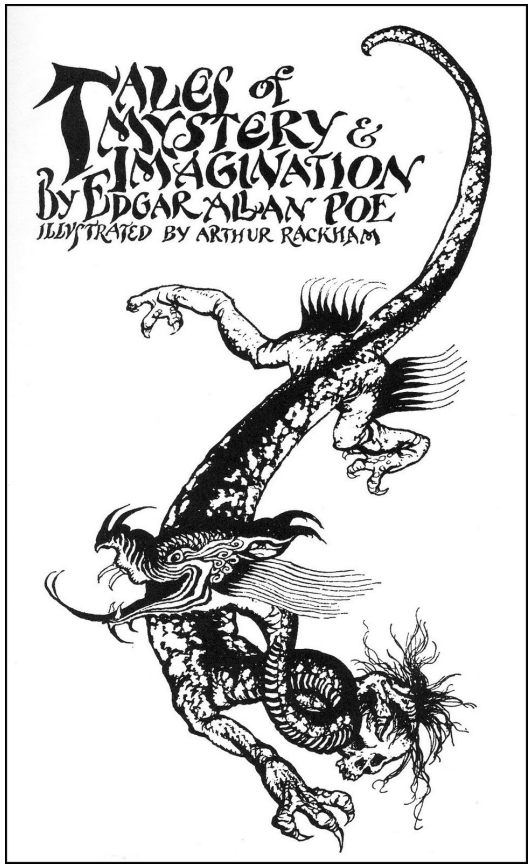
A little thinking makes the program $> 10\times$ faster!

Non-logical / Meta-logical Predicates



Hic sunt dragones!

Non-logical / Meta-logical Predicates



You need to be aware of non-logical predicates but you need **not** be skilled in their use.

Non-logical Predicates

Some predicates usually destroy the declarative properties of Prolog

- Cut
- Negation-as-failure
- If-then-else
- Input / Output

Non-logical Predicates

Some predicates usually destroy the declarative properties of Prolog

- Cut
- Negation-as-failure
- If-then-else
- Input / Output

. . . but you will encounter them in practice.

This lecture will only explain them and how to avoid them. If you want to learn about them properly, read *R. O'Keefe: The Craft of Prolog*.

Outline

Meta-logical Predicates

- Meta-logical predicates go beyond the expressitivity of FOL:
  - Using terms as predicates
  - Querying if a term is a variable / atom / ground etc.
  - Generating the list of all solutions of a predicate
- Meta-logical predicates may destroy the declarative meaning of a program

Meta-Calls

- We can create terms programmatically with `=../2`:

```
?- P =.. [isa_list, X].
P = isa_list(X).
```

- We can use such a term as a goal:

```
?- P =.. [isa_list, X], call(P).
P = isa_list([]),
X = [] ;
P = isa_list([_4302]),
X = [_4302] ;
% ... just like calling ?- isa_list(X). directly.
```

Typechecks

- `var/1`: true if argument is a variable
- `nonvar/1`: true if argument is not a variable
- `atom/1`: true if argument is a constant (no variable, no function)
- `ground/1`: true if argument does not contain variables
- `==/2`: true if arguments are identical (no unification!)
- `\==/2`: true if terms are not identical (no unification!)
- `\=/2`: true if **no** substitution makes the terms equal

Typechecks

- What are they useful for?
  - Program transformation of Prolog programs,
  - Writing your own unification predicate
  - Writing a Prolog interpreter in Prolog

Typechecks

- What are they useful for?
  - Program transformation of Prolog programs,
  - Writing your own unification predicate
  - Writing a Prolog interpreter in Prolog
- What makes them problematic? – They destroy commutativity of conjunction!

```
?- var(X), X = something.
X = something.

?- X = something, var(X).
false.
```

Aggregating all solutions of a predicate

- when `setof(Pattern, Goal, List)` succeeds, `List` contains each answer substitution to `Goal` applied to `Pattern`
  Variables in `Pattern` and `Goal` must not occur elsewhere!

Aggregating all solutions of a predicate

- when `setof(Pattern, Goal, List)` succeeds, `List` contains each answer substitution to `Goal` applied to `Pattern`
  Variables in `Pattern` and `Goal` must not occur elsewhere!
- Example: create a list of all elements of $\{1,2,3\} \times \{2,3,5\}$

```
?- setof(X-Y, ( member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Xs = [1-2, 1-3, 1-5, 2-2, 2-3, 2-5, 3-2, 3-3, ... - ...].
```

Aggregating all solutions of a predicate

- Variables that do not occur in the pattern lead to backtracking:

```
?- setof(X, ( member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Y = 2,
Xs = [1, 2, 3] ;
Y = 3,
Xs = [1, 2, 3] ;
Y = 5,
Xs = [1, 2, 3].
```

Aggregating all solutions of a predicate

- If we want to ignore the value of `Y`, we have to add an existential quantifier `Goal`:

```
?- setof(X, Y ^ (  member_of(X, [1,2,1,3]), member_of(Y, [3,2,5])), Xs).
Xs = [1, 2, 3].
```

Aggregating all solutions of a predicate

- Only useful for terminating predicates!

```
?- setof(X, isa_list(X), Xs).
% does not terminate, exhausts memory really fast
```

Outline

Cut

- The cut operator ! cuts off derivation branches

Cut

- The cut operator ! cuts off derivation branches
- Example:

```
nondet(a,c).
nondet(b,d).

nocut(X) :-
    nondet(X, _).
nocut(X) :-
    nondet(_, X).
```

Cut

- The cut operator ! cuts off derivation branches
- Example:

```
nondet(a,c).
nondet(b,d).

% extract first argument of nondet/2
withcut(Y) :-
    !,          % never backtrack past this point
    nondet(Y, _).
% extract second argument of nondet/2
withcut(Y) :-
    !,          % never backtrack past this point
    nondet(_, Y).
```

Cut

- The cut operator ! cuts off derivation branches
- Example:

```
nondet(a,c).
nondet(b,d).

% extract second argument of nondet/2
withcut2(Y) :-
    !,          % never backtrack past this point
    nondet(_, Y).
% extract first argument of nondet/2
withcut2(Y) :-
    !,          % never backtrack past this point
    nondet(Y, _).
```

Cut

- The cut operator ! cuts off derivation branches
- Comparison:

```
?- nocut(X).
X = a ;
X = b ;
X = c ;
X = d.

?- withcut(X).
X = a ;
X = b.

?- withcut2(X).
X = c ;
X = d.
```

Cut

- The cut operator ! cuts off derivation branches
- nocut/1 has not the same solution set as withcut/1 and withcut2/1
- The order of rules changed the set of solutions between withcut/1 and withcut2/1!
- **Red** cuts change the solutions of a program,
  Green cuts prune derivations but keep the solution set intact,
  Blue cuts are green cuts that the compiler should optimize automatically

Cut

- Why use it then?

Cut

- Why use it then?
  - Speed up a program
    Writing green cuts is difficult – correctness over speed!
  - Needed to implement negation-as-failure and if-then-else

Why cuts are problematic

Problem

Implement a predicate `max/3` such that the third argument is equivalent to the maximum of the first two arguments.

Why cuts are problematic

Solution without cuts:

```
max(X,Y,X) :-
    X >= Y.
max(X,Y,Y) :-
    X < Y.
```

Why cuts are problematic

Solution with a blue cut:

```
max_blue(X,Y,X) :-
    X >= Y,
    !.
max_blue(X,Y,Y) :-
    X < Y.
```

The two branches are mutually exclusive

Why cuts are problematic

Temptation: Let's remove the second guard!
If $X \not\geqslant Y$ then $X < Y$ must hold, after all...

```
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).
```

Why cuts are problematic

```
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).
```

```
?- max_red(9,0,0).
true.
```

this is not proper mathematics. . .

Why cuts are problematic

```
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).
```

```
?- max_red(9,0,0).
true.
```

this is not proper mathematics. . . what happened?

- `max_red(9,0,0)` and rule head `max_red(X,Y,X)` are not unifiable!
- Prolog immediately tries the second rule but we deleted the guard

Why cuts are problematic

```
max_red(X,Y,X) :-
    X >= Y,
    !.
max_red(X,Y,Y).
```

```
?- max_red(9,0,0).
true.
```

this is not proper mathematics. . . what happened?

- `max_red(9,0,0)` and rule head `max_red(X,Y,X)` are not unifiable!
- Prolog immediately tries the second rule but we deleted the guard
- This particular predicate can be fixed by making it steadfast – see chapter 3.11 in *The Craft of Prolog*.

Why cuts are problematic

Lesson 1: Using cuts for efficiency is error prone

As long as we can achieve magnitudes of speedups by cleverly restating the problem, why use cuts?

Lesson 2: Cut is rarely neccessary

In most cases, we can get by without cut. In the rare cases we need it, there are slightly safer predicates.

Negation as failure

- Inference rule: if we can not derive `pred` then conclude $\neg$`pred`.
- Implementation (uses cut):

```
\+(Goal) :-
    call(Goal),  % call to Goal
    !.           % we have derived Goal, cut the other branch
    false.       % ... and fail
\+(_Goal) :-
    true.        % we could not derive Goal, succeed
```

Negation as failure and the closed world assumption

- Consider the following program:

```
continent(antarctica).
continent(america).
continent(asia).
continent(australia).
continent(europe).

land(X) :-  % if X is a continent, X is on land
    continent(X).

water(X) :- % if X is not a continent, X is in the ocean
    \+ land(X).
```

Negation as failure and the closed world assumption

```
?- land(X).
X = antarctica ;
X = america ;
X = asia ;
X = australia ;
X = europe.
```

so far, so good!

Negation as failure and the closed world assumption

```
?- water(pacific_ocean).
true.

?- water(saturn).
true.

?- water(minnie_mouse).
true.
```

it's getting stranger. . . but that's due to the closed world assumption

Negation as failure and the closed world assumption

```
?- water(X).
false.
```

. . . There is no water?

Negation as failure and the closed world assumption

```
?- water(X).
false.
```

. . . There is no water?

- According to our definition, whenever `land(X)` succeeds, `water(X` fails.
  This is not classical logic! In FOL we have $p(t) \rightarrow \exists x\, p(x)$!

Negation as failure and the closed world assumption

- Negation by failure coincides with FOL if
  - the query is ground
  - the negated goal terminates
- Everything else is tricky

Summary

- The built-in predicates are fast but only compute
- Constraint logic programming over finite domains provides declarative integer arithmetic
- CLP(FD) predicates...
  - assign variables to domain
  - compose constraints with core predicates
  - need to label the variables to find the solutions
- Non-logical predicates...
  - destroy the declarative reading of Prolog
  - are useful for special cases (negation-as-failure, if-then-else, meta-programming)
  - should only be used when absolutely neccessary

That's all for today!

Organisational Reminders

There is a Blackboard Forum for questions related to course material/labs.

There is a Prolog Help Session at 1pm in 1.10 today.

Lecture 7 First-Order Logic

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Introduce you to the general language of first-order logic including its syntax, semantics, and how it relates to Datalog and Prolog.

Learning Outcomes

By the end of this lecture you will be able to:

1. Write formulas in first-order logic modelling real-world situations and describe their meaning
2. Recall the notion of an *interpretation* in first-order logic and apply it to determine the truth of a first-order logic
3. Give examples of why first-order logic is more expressive than Datalog or Prolog

Recap

Representation and Reasoning

Representation	Reasoning	Properties
Datalog: Facts and Rules Database Semantics Rules: $f_1 \wedge \ldots \wedge f_2 \Rightarrow h$ vars in head must be in body function-free, negation-free	**Forward Chaining** Compute all consequences Answer atomic queries Deductive Database	Finite Terminating Single Interpretation
Prolog: extend Datalog with Functions e.g. lists Lift variable restriction Non-logic part	**Backward Chaining** Conjunctive queries Use rules to reduce goal to subgoals (backtracking)	Possibly Infinite Turing-complete

Recap

Representation and Reasoning

Representation	Reasoning	Properties
Datalog: Facts and Rules Database Semantics Rules: $f_1 \wedge \ldots \wedge f_2 \Rightarrow h$ vars in head must be in body function-free, negation-free	**Forward Chaining** Compute all consequences Answer atomic queries Deductive Database	Finite Terminating Single Interpretation
Prolog: extend Datalog with Functions e.g. lists Lift variable restriction Non-logic part	**Backward Chaining** Conjunctive queries Use rules to reduce goal to subgoals (backtracking)	Possibly Infinite Turing-complete
First-order logic No Database Semantics General formulas	**Proof Search** Saturation-based Lots of heuristics	Semi-decidable

Today

(Revising) First-Order (Predicate) Logic - defining formulas

What formulas mean (intuitively)

What formulas mean (formally)

Relating this to things we've seen

First-order logic: Syntax

Definition (Signature)

The signature $\Sigma = \langle \mathcal{F}, \mathcal{P}, \text{arr} \rangle$ of a first-order formula consists of a disjoint sets of function and predicate symbols $\mathcal{F}$ and $\mathcal{P}$, and a function $\text{arr} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}_0$ giving the arity (number of parameters) for each symbol.

First-order logic: Syntax

Definition (Signature)

The signature $\Sigma = \langle \mathcal{F}, \mathcal{P}, \text{arr} \rangle$ of a first-order formula consists of a disjoint sets of function and predicate symbols $\mathcal{F}$ and $\mathcal{P}$, and a function $\text{arr} : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}_0$ giving the arity (number of parameters) for each symbol.

Definition (Term - similar to Prolog)

A variable is a term. A constant symbol ($c \in \mathcal{F}$ s.t. $\text{arr}(c) = 0$) is a term. If $f \in \mathcal{F}$ and $\text{arr}(f) = n > 0$ then $f(t_1, \ldots t_n)$ is a term if each t_i is a term.

First-order logic: Syntax

Definition (Signature)

The signature $\Sigma = \langle \mathcal{F}, \mathcal{P}, \text{arr} \rangle$ of a first-order formula consists of a disjoint sets of function and predicate symbols $\mathcal{F}$ and $\mathcal{P}$, and a function $\text{arr} : \mathcal{F} \cup \mathcal{P} \rightarrow \mathbb{N}_0$ giving the arity (number of parameters) for each symbol.

Definition (Term - similar to Prolog)

A variable is a term. A constant symbol ($c \in \mathcal{F}$ s.t. $\text{arr}(c) = 0$) is a term. If $f \in \mathcal{F}$ and $\text{arr}(f) = n > 0$ then $f(t_1, \ldots t_n)$ is a term if each t_i is a term.

Terms are similar to Prolog in syntax but not semantics. In first-order logic terms do not automatically have an interpretation as algebraic datatypes.

E.g. it is possible that $f(a, b) = a$, or $add(one, zero) = one$.

First-order logic: Syntax

Definition (Atoms)

A proposition ($p \in \mathcal{P}$ s.t. $\text{arr}(p) = 0$) is an atom. If t_1 and t_2 are terms then $t_1 \neq t_2$ is an atom. If $p \in \mathcal{P}$ and $\text{arr}(p) = n > 0$ then $p(t_1, \ldots t_n)$ is an atom if each t_i is a term.

First-order logic: Syntax

Definition (Atoms)

A proposition ($p \in \mathcal{P}$ s.t. $\text{arr}(p) = 0$) is an atom. If t_1 and t_2 are terms then $t_1 \neq t_2$ is an atom. If $p \in \mathcal{P}$ and $\text{arr}(p) = n > 0$ then $p(t_1, \ldots t_n)$ is an atom if each t_i is a term.

Definition (Formula)

All atoms are formulas. The boolean value true is a formula. If ϕ_1 and ϕ_2 are formulae then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, and $\forall x.\phi_1$.

First-order logic: Syntax

Definition (Atoms)

A proposition ($p \in \mathcal{P}$ s.t. $\text{arr}(p) = 0$) is an atom. If t_1 and t_2 are terms then $t_1 \neq t_2$ is an atom. If $p \in \mathcal{P}$ and $\text{arr}(p) = n > 0$ then $p(t_1, \ldots t_n)$ is an atom if each t_i is a term.

Definition (Formula)

All atoms are formulas. The boolean value true is a formula. If ϕ_1 and ϕ_2 are formulae then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, and $\forall x.\phi_1$.

We define false $= \neg$true, $\phi_1 \vee \phi_2 = \neg(\phi_1 \wedge \phi_2)$, $\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2 = ((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$ and $\exists x.\phi = \neg(\forall x.\neg\phi)$.

First-order logic: Syntax

Definition (Atoms)

A proposition ($p \in \mathcal{P}$ s.t. $\text{arr}(p) = 0$) is an atom. If t_1 and t_2 are terms then $t_1 \neq t_2$ is an atom. If $p \in \mathcal{P}$ and $\text{arr}(p) = n > 0$ then $p(t_1, \ldots t_n)$ is an atom if each t_i is a term.

Definition (Formula)

All atoms are formulas. The boolean value true is a formula. If ϕ_1 and ϕ_2 are formulae then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, and $\forall x.\phi_1$.

We define false $= \neg$true, $\phi_1 \vee \phi_2 = \neg(\phi_1 \wedge \phi_2)$, $\phi_1 \rightarrow \phi_2 = \neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2 = ((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$ and $\exists x.\phi = \neg(\forall x.\neg\phi)$.

In terms of precedence, unary symbols bind tighter than binary symbols. Otherwise, I use parenthesis to disambiguate.

How to Read Formulas

$=$	equals
$\neg$	not
$\wedge$	and
$\vee$	or
$\rightarrow$	implies, or sometimes if *left* then *right*
$\leftrightarrow$	equivalent, bi-implication, if-and-only-if
$\forall$	(for) all, (for) every
$\exists$	exists, some

I will write $\neq$ (read 'not equal') instead of $\neg(t_1 = t_2)$ as it is nicer.

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human
Every red car is cool
Every bird that is not a penguin or an ostrich can fly
If something is either fruit or cheese then it is food
Every car is either petrol or diesel
No car is both petrol and diesel

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	`human(X) :- man(X).`
Every red car is cool	
Every bird that is not a penguin or an ostrich can fly	
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	
Every bird that is not a penguin or an ostrich can fly	
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	`cool(X) :- red(X), car(X).`
Every bird that is not a penguin or an ostrich can fly	
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	`fly(X) :- bird(X), dif(X,ostrich), dif(X,penguin.`
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	`food(X) :- fruit(X).` `food(X) :- cheese(X).`
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$(\forall x.fruit(x) \rightarrow food(x)) \wedge (\forall x.cheese(x) \rightarrow food(x))$
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	Cannot be expressed in Prolog
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	$\forall x.car(x) \rightarrow \begin{pmatrix} petrol(x) \vee \\ diesel(x) \end{pmatrix}$
No car is both petrol and diesel	

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	$\forall x.car(x) \rightarrow \begin{pmatrix} petrol(x) \vee \\ diesel(x) \end{pmatrix}$
No car is both petrol and diesel	`petrol(X) :-` `not(diesel(X)).`

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	$\forall x.car(x) \rightarrow \begin{pmatrix} petrol(x) \vee \\ diesel(x) \end{pmatrix}$
No car is both petrol and diesel	$\forall x.\neg(petrol(x) \wedge diesel(x))$

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \left(\begin{array}{l} fruit(x) \vee \\ cheese(x) \end{array} \right) \rightarrow fruit(x)$
Every car is either petrol or diesel	$\forall x.car(x) \rightarrow \left(\begin{array}{l} petrol(x) \vee \\ diesel(x) \end{array} \right)$
No car is both petrol and diesel	$\neg \exists x.(petrol(x) \wedge diesel(x))$

Examples of Universal Quantification

We should be used to Universal quantification by now, it is what we have been implicitly using in rules.

Every man is human	$\forall x.(man(x) \rightarrow human(x))$
Every red car is cool	$\forall x.((red(x) \wedge car(x)) \rightarrow cool(x))$
Every bird that is not a penguin or an ostrich can fly	$\forall x.(bird(x) \wedge x \neq penguin \wedge x \neq ostrich) \rightarrow fly(x)$
If something is either fruit or cheese then it is food	$\forall x. \begin{pmatrix} fruit(x) \vee \\ cheese(x) \end{pmatrix} \rightarrow fruit(x)$
Every car is either petrol or diesel	$\forall x.car(x) \rightarrow \begin{pmatrix} petrol(x) \vee \\ diesel(x) \end{pmatrix}$
No car is both petrol and diesel	$\neg\exists x.(petrol(x) \wedge diesel(x))$

Remember: Every Yacht that I own is made of gold.

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man
There are at least two men
Everybody has a mother

If two different people have parents who are siblings then they are cousins

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man	$\exists x.(man(x) \wedge human(x))$
There are at least two men	
Everybody has a mother	

If two different people have parents who are siblings then they are cousins

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man	$\exists x.(man(x) \wedge human(x))$
There are at least two men	$\exists x.\exists y.(man(x) \wedge man(y) \wedge x \neq y)$
Everybody has a mother	

If two different people have parents who are siblings then they are cousins

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man	$\exists x.(man(x) \wedge human(x))$
There are at least two men	$\exists x.\exists y.(man(x) \wedge man(y) \wedge x \neq y)$
Everybody has a mother	$\forall x.\exists y.(person(x) \rightarrow mother\_of(y, x))$

If two different people have parents who are siblings then they are cousins

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man	$\exists x.(man(x) \wedge human(x))$
There are at least two men	$\exists x.\exists y.(man(x) \wedge man(y) \wedge x \neq y)$
Everybody has a mother	$\forall x.\exists y.(person(x) \rightarrow mother\_of(y, x))$

If two different people have parents who are siblings then they are cousins

$$\forall x, y.(x \neq y \wedge (\exists u, v. \begin{pmatrix} parent(u, x) \wedge \\ parent(v, y) \wedge \\ sibling(u, v) \end{pmatrix})) \rightarrow cousins(x, y)$$

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

There is a human man	$\exists x.(man(x) \wedge human(x))$
There are at least two men	$\exists x.\exists y.(man(x) \wedge man(y) \wedge x \neq y)$
Everybody has a mother	$\forall x.\exists y.(person(x) \rightarrow mother\_of(y, x))$

If two different people have parents who are siblings then they are cousins

$$\forall x, y, u, v.(x \neq y \wedge \begin{pmatrix} parent(u, x) \wedge \\ parent(v, y) \wedge \\ sibling(u, v) \end{pmatrix}) \rightarrow cousins(x, y)$$

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

Examples of Existential Quantification

We don't have existential quantification in Prolog as it is negated universal quantification.

Not quite true.

When we ask a query `brother(X,zeus).` we are asking if there exists an object for `X` that makes the fact true.

We'll return to that point later.

Examples of Existential Quantification

Existential quantification over an empty domain is false.

Existential quantification over a finite domain is finite disjunction

$$(\forall x.(x = a \vee x = b)) \rightarrow ((\exists x.p(x)) \leftrightarrow (p(a) \vee p(b)))$$

Which Formula?

Everybody is either a Manchester United Supporter or a Manchester City Supporter and cannot be both.

1. $\forall p.(person(p) \rightarrow (manU(p) \vee manC(p)))$
2. $\forall p.(person(p) \rightarrow ((manU(p) \vee manC(p)) \wedge \neg(manU(p) \wedge manC(p))))$
3. $\exists p_1, p_2.(person(p_1) \wedge person(p_2) \wedge p_1 \neq p_2 \wedge manU(p_1) \wedge manC(p_2))$
4. $\forall p.(person(p) \rightarrow (\exists s.(supports(p, s) \wedge (s = manU \vee s = manC))))$

Which Formula?

Everybody is either a Manchester United Supporter or a Manchester City Supporter and cannot be both.

1. $\forall p.(person(p) \rightarrow (manU(p) \vee manC(p)))$
2. $\forall p.(person(p) \rightarrow ((manU(p) \vee manC(p)) \wedge \neg(manU(p) \wedge manC(p))))$
3. $\exists p_1, p_2.(person(p_1) \wedge person(p_2) \wedge p_1 \neq p_2 \wedge manU(p_1) \wedge manC(p_2))$
4. $\forall p.(person(p) \rightarrow (\exists s.(supports(p, s) \wedge (s = manU \vee s = manC))))$

Which Formula?

Everybody is either a Manchester United Supporter or a Manchester City Supporter and cannot be both.

1. $\forall p.(person(p) \rightarrow (manU(p) \vee manC(p))) \wedge (manU(p) \leftrightarrow \neg manC(p))$
2. $\forall p.(person(p) \rightarrow ((manU(p) \vee manC(p)) \wedge \neg(manU(p) \wedge manC(p))))$
3. $\exists p_1, p_2.(person(p_1) \wedge person(p_2) \wedge p_1 \neq p_2 \wedge manU(p_1) \wedge manC(p_2))$
4. $\forall p.(person(p) \rightarrow (\exists s.(supports(p, s) \wedge (s = manU \vee s = manC))))$

Which Formula?

Everybody is either a Manchester United Supporter or a Manchester City Supporter and cannot be both.

1. $\forall p.(person(p) \rightarrow (manU(p) \vee manC(p))) \wedge (manU(p) \leftrightarrow \neg manC(p))$
2. $\forall p.(person(p) \rightarrow ((manU(p) \vee manC(p)) \wedge \neg(manU(p) \wedge manC(p))))$
3. $\exists p_1, p_2.(person(p_1) \wedge person(p_2) \wedge p_1 \neq p_2 \wedge manU(p_1) \wedge manC(p_2))$
4. $\forall p.(person(p) \rightarrow (\exists s.(supports(p, s) \wedge (s = manU \vee s = manC))))$
5. $\forall p.(person(p) \rightarrow \begin{pmatrix} supports(p) = manU \vee \\ supports(p) = manC \end{pmatrix}) \wedge manU \neq manC$

Which Formula?

There is a book that if I read it I will score the best mark in all of my exams.

1. $\forall e.\exists b.(read(b) \wedge (take(e) \rightarrow best(e)))$
2. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow best(e)))$
3. $(\exists b.read(b)) \wedge (\forall e.(take(e) \rightarrow best(e)))$
4. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow \forall p.(takes(p, e) \rightarrow better(e, p))))$

Which Formula?

There is a book that if I read it I will score the best mark in all of my exams.

1. $\forall e.\exists b.(read(b) \wedge (take(e) \rightarrow best(e)))$
2. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow best(e)))$
3. $(\exists b.read(b)) \wedge (\forall e.(take(e) \rightarrow best(e)))$
4. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow \forall p.(takes(p, e) \rightarrow better(e, p))))$

Which Formula?

There is a book that if I read it I will score the best mark in all of my exams.

1. $\forall e.\exists b.(read(b) \wedge (take(e) \rightarrow best(e)))$
2. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow best(e)))$
3. $(\exists b.read(b)) \wedge (\forall e.(take(e) \rightarrow best(e)))$
4. $\exists b.(read(b) \rightarrow \forall e.(take(e) \rightarrow \forall p.(takes(p, e) \rightarrow better(e, p))))$

Why First-Order Logic

The order of a logic is related to the kind of things one can quantify over.

Propositional logic is 'zero' ordered as one cannot quantify.

Extensions of propositional logic such as QBF or PLFD are useful for modelling and allow for efficient reasoning but do not increase expressive power.

First-order logic allows one to quantify over individuals

Second-order logic allows one to quantify over sets of individuals or equivalently predicts over individuals.

And so on.

Free Variables, Sentences and Closure

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

E.g. x and y are free in $(\forall y.p(x, y)) \wedge (\exists x.p(x, y))$

Free Variables, Sentences and Closure

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

E.g. x and y are free in $(\forall y.p(x, y)) \wedge (\exists x.p(x, y))$

Free Variables, Sentences and Closure

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

E.g. x and y are free in $(\forall y.p(x, y)) \wedge (\exists x.p(x, y))$

It's a bit confusing as x and y occur in as bound and unbound. Later we rewrite formulas to avoid this (rectification).

Free Variables, Sentences and Closure

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

E.g. x and y are free in $(\forall y.p(x,y)) \wedge (\exists x.p(x,y))$

It's a bit confusing as x and y occur in as bound and unbound. Later we rewrite formulas to avoid this (rectification).

If ϕ has free variables X we might write it $\phi[X]$.
We write $\phi[V]$ for the formula $\phi[X]$ where X is replaced by V.

Free Variables, Sentences and Closure

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

E.g. x and y are free in $(\forall y.p(x,y)) \wedge (\exists x.p(x,y))$

It's a bit confusing as x and y occur in as bound and unbound. Later we rewrite formulas to avoid this (rectification).

If ϕ has free variables X we might write it $\phi[X]$.
We write $\phi[V]$ for the formula $\phi[X]$ where X is replaced by V.

A formula is a sentence if it does not contain any free variables

The universal closure of $\phi[X]$ is $\forall X.\phi[X]$

Interpretation

(We already met this general notion in Lecture 3)

An Interpretation allows us to assign a truth value to every sentence.

Let $\langle \mathcal{D}, \mathcal{I} \rangle$ be a structure such that $\mathcal{I}$ is an interpretation over a non-empty (possibly infinite) domain $\mathcal{D}$. We often leave $\mathcal{D}$ implicit and refer directly to $\mathcal{I}$.

The map $\mathcal{I}$ maps

- Every constant symbol to an element of $\mathcal{D}$
- Every function symbol of arity n to a function in $\mathcal{D}^n \to \mathcal{D}$
- Every proposition symbol to a truth value in $\mathbb{B}$
- Every predicate symbol of arity n to a function in $\mathcal{D}^n \to \mathbb{B}$

Interpretation of Atoms

We can then lift interpretations to non-constant terms recursively as

$$\mathcal{I}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$$

(variables are ignored as we only consider ground terms)

We can lift interpretations to non-propositional atoms similarly e.g.

$$\begin{array}{rcl} \mathcal{I}(t_1 = t_2) & = & \mathcal{I}(t_1) = \mathcal{I}(t_2) \\ \mathcal{I}(p(t_1, \ldots, t_n)) & = & \mathcal{I}(p)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \end{array}$$

Every atom is interpreted as a truth value.

Interpretation of Atoms

We can then lift interpretations to non-constant terms recursively as

$$\mathcal{I}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$$

(variables are ignored as we only consider ground terms)

We can lift interpretations to non-propositional atoms similarly e.g.

$$\begin{aligned} \mathcal{I}(t_1 = t_2) &= \mathcal{I}(t_1) = \mathcal{I}(t_2) \\ \mathcal{I}(p(t_1, \ldots, t_n)) &= \mathcal{I}(p)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \end{aligned}$$

Every atom is interpreted as a truth value.

This mixes three kinds of equality (be careful, I'm being lazy).

Interpreting Equality

Usually FOL is introduced without equality and then extended.

I am introducing FOL with equality directly.

Usually in the extension the set of models is restricted to normal models that interpret equality as equality and such that all domain elements are distinct with respect to equality. We will enforce this straight away here.

Interpreting Equality

Usually FOL is introduced without equality and then extended.

I am introducing FOL with equality directly.

Usually in the extension the set of models is restricted to normal models that interpret equality as equality and such that all domain elements are distinct with respect to equality. We will enforce this straight away here.

We do not *need* to make equality directly part of the language. Adding

$$\forall x.x = x \qquad \forall x, y.(x = y \rightarrow y = x) \qquad \forall x, y, z.((x = y \wedge y = z) \rightarrow x = z)$$
$$\forall x_1, \ldots x_n, y_1, \ldots y_n.((x_1 = y_1 \wedge \ldots x_n = y_n) \rightarrow f(x_1, \ldots x_n) = f(y_1, \ldots, y_n))$$
$$\forall x_1, \ldots x_n, y_1, \ldots y_n.((x_1 = y_1 \wedge \ldots x_n = y_n) \rightarrow p(x_1, \ldots x_n) \leftrightarrow p(y_1, \ldots, y_n))$$

(for all functions f and predicates p) forces $=$ to behave as above.

Interpreting Equality

Usually FOL is introduced without equality and then extended.

I am introducing FOL with equality directly.

Usually in the extension the set of models is restricted to normal models that interpret equality as equality and such that all domain elements are distinct with respect to equality. We will enforce this straight away here.

We do not *need* to make equality directly part of the language. Adding

$$\forall x.x = x \qquad \forall x, y.(x = y \rightarrow y = x) \qquad \forall x, y, z.((x = y \wedge y = z) \rightarrow x = z)$$
$$\forall x_1, \ldots x_n, y_1, \ldots y_n.((x_1 = y_1 \wedge \ldots x_n = y_n) \rightarrow f(x_1, \ldots x_n) = f(y_1, \ldots, y_n))$$
$$\forall x_1, \ldots x_n, y_1, \ldots y_n.((x_1 = y_1 \wedge \ldots x_n = y_n) \rightarrow p(x_1, \ldots x_n) \leftrightarrow p(y_1, \ldots, y_n))$$

(for all functions f and predicates p) forces $=$ to behave as above.

This general approach of restricting the interpretations of interest with respect to some theory is also used in arithmetic (see end of my part).

Interpretation of Formulas

Finally we interpret formulas

$\mathcal{I}(\mathit{true})$	is always true
$\mathcal{I}(\neg\phi)$	*iff* $\mathcal{I}(\phi)$ is not true
$\mathcal{I}(\phi_1 \wedge \phi_2)$	*iff* both $\mathcal{I}(\phi_1)$ and $\mathcal{I}(\phi_2)$ are true
$\mathcal{I}(\forall x.\phi[x])$	*iff* for every $d \in \mathcal{D}$ we have that $\mathcal{I}(\phi[d])$ is true

Hopefully the first 3 are straightforward. For $\forall$ we take each element d of the domain and see if the quantified formula holds if we replace x by d.

We could extend this for the derived operators but do not need to.

Big Aside

In the next slide I am going to use the syntax

$$\lambda x.\ (\textit{expression using } x)$$

to represent an anonymous function that takes something (x) as input and returns the result of evaluating the expression on that input.

This notation is borrowed from the λ-calculus, a model of computation that is central to computer science but not currently taught in our Undergraduate syllabus. I suggest you at least read the Wikipedia page at some point!

In any case, it is used to represent functions and that's all you need to know for the next slide.

A Formula Can Have Many Interpretations

$$parent(giles, mark)$$
$$man(giles)$$
$$\forall x.((man(x) \wedge \exists y.parent(x, y)) \rightarrow father(x))$$

We need to interpret $parent, giles, mark, man, father$. Fix $\mathcal{D} = \{1, 2\}$.

A Formula Can Have Many Interpretations

$$parent(giles, mark)$$
$$man(giles)$$
$$\forall x.((man(x) \wedge \exists y.parent(x, y)) \rightarrow father(x))$$

We need to interpret $parent, giles, mark, man, father$. Fix $\mathcal{D} = \{1, 2\}$.

Giles is Mark. Everything is true.

$$\mathcal{I}(giles) = 1$$
$$\mathcal{I}(mark) = 1$$
$$\mathcal{I}(parent) = (\lambda x, y.(true))$$
$$\mathcal{I}(man) = (\lambda x.(true))$$
$$\mathcal{I}(father) = (\lambda x.(true))$$

A Formula Can Have Many Interpretations

$$parent(giles, mark)$$
$$man(giles), mark \neq giles$$
$$\forall x.((man(x) \wedge \exists y.parent(x, y)) \rightarrow father(x))$$

We need to interpret $parent, giles, mark, man, father$. Fix $\mathcal{D} = \{1, 2\}$.

Giles and Mark different. Everything is true.

$$\mathcal{I}(giles) = 1$$
$$\mathcal{I}(mark) = 2$$
$$\mathcal{I}(parent) = (\lambda x, y.(true))$$
$$\mathcal{I}(man) = (\lambda x.(true))$$
$$\mathcal{I}(father) = (\lambda x.(true))$$

A Formula Can Have Many Interpretations

$$parent(giles, mark)$$
$$man(giles), mark \neq giles$$
$$\forall x.((man(x) \wedge \exists y.parent(x, y)) \rightarrow father(x))$$

We need to interpret $parent, giles, mark, man, father$. Fix $\mathcal{D} = \{1, 2\}$.

Giles and Mark different. Least is true.

$$\mathcal{I}(giles) = 1$$
$$\mathcal{I}(mark) = 2$$
$$\mathcal{I}(parent) = (\lambda x, y.(x = 1 \wedge y = 2)))$$
$$\mathcal{I}(man) = (\lambda x.(x = 2))$$
$$\mathcal{I}(father) = (\lambda x.(x = 2))$$

A Formula Can Have Many Interpretations

$$parent(giles, mark)$$
$$man(giles), mark \neq giles$$
$$\forall x.((man(x) \wedge \exists y.parent(x, y)) \rightarrow father(x))$$

We need to interpret $parent, giles, mark, man, father$. Fix $\mathcal{D} = \{1, 2\}$.

Giles and Mark different. A bit more is true.

$$\mathcal{I}(giles) = 1$$
$$\mathcal{I}(mark) = 2$$
$$\mathcal{I}(parent) = (\lambda x, y.(x = 1 \wedge y = 2)))$$
$$\mathcal{I}(man) = (\lambda x.(x = 2 \vee x = 1))$$
$$\mathcal{I}(father) = (\lambda x.(x = 2))$$

Summary

First-Order Logic has explicit universal and existential quantification and allows arbitrary boolean structure in formulas.

The semantics of a formula is defined in terms of interpretations.

A FOL formula can have many models. It can also have a single model (up to renaming of domain constants) or no models.

Lecture 8 First-Order Logic
Models and Reasoning with Clauses

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Explore what it means for an interpretation to be a model of a formula and see how we can reason with formulas in clausal form

Learning Outcomes

By the end of this lecture you will be able to:

1. Identify when an interpretation is a model of a formula
2. Give examples of why first-order logic is more expressive than Datalog or Prolog
3. Explain the meaning of the open and closed world assumptions in terms of interpretations
4. Recall the resolution rule and how it applies to sets of clauses to solve reasoning problems

Recap

Datalog: closed-world, function-free, rules, matching

Prolog: closed-world, functions, rules, unification

First-order logic: open-world, functions, formulas

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$$p(x, y) \leftrightarrow \exists z.(r(x, z) \wedge r(z, y))$$
$$(\forall x.\exists y.p(y))$$
$$(\forall x.p(x, y)) \wedge (\exists x.p(x, y))$$

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$$p(x, y) \leftrightarrow \exists z.(r(x, z) \wedge r(z, y)) \qquad \{x, y\}$$
$$(\forall x.\exists y.p(y))$$
$$(\forall x.p(x, y)) \wedge (\exists x.p(x, y))$$

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y)) \qquad \{x,y\}$$
$$(\forall x.\exists y.p(y))$$
$$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$$

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y))$	$\{x,y\}$
$(\forall x.\exists y.p(y))$	$\{\}$
$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$	

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y))$	$\{x,y\}$
$(\forall x.\exists y.p(y))$	$\{\}$
$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$	

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y))$	$\{x,y\}$
$(\forall x.\exists y.p(y))$	$\{\}$
$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$	$\{y\}$

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y))$	$\{x,y\}$
$(\forall x.\exists y.p(y))$	$\{\}$
$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$	$\{y\}$

If ϕ has free variables X we might write it $\phi[X]$.
We write $\phi[V]$ for the formula $\phi[X]$ where X is replaced by V.

Free Variables, Sentences and Closure (Recap)

The free variables of a formula f are those not captured by a quantifer. Otherwise they are bound variables.

What are the free variables in the following:

$p(x,y) \leftrightarrow \exists z.(r(x,z) \wedge r(z,y))$	$\{x,y\}$
$(\forall x.\exists y.p(y))$	$\{\}$
$(\forall x.p(x,y)) \wedge (\exists x.p(x,y))$	$\{y\}$

If ϕ has free variables X we might write it $\phi[X]$.
We write $\phi[V]$ for the formula $\phi[X]$ where X is replaced by V.

A formula is a sentence if it does not contain any free variables

The universal closure of $\phi[X]$ is $\forall X.\phi[X]$ (similarly for existential)

Interpretation (Recap)

Let $\langle \mathcal{D}, \mathcal{I} \rangle$ be a structure such that $\mathcal{I}$ is an interpretation over a non-empty (possibly infinite) domain $\mathcal{D}$.

The map $\mathcal{I}$ maps

- Every constant symbol to an element of $\mathcal{D}$
- Every function symbol of arity n to a function in $\mathcal{D}^n \to \mathcal{D}$
- Every proposition symbol to a truth value in $\mathbb{B}$
- Every predicate symbol of arity n to a function in $\mathcal{D}^n \to \mathbb{B}$

We lift interpretations to non-constant terms and atoms recursively as

$$\begin{array}{rcl} \mathcal{I}(f(t_1, \ldots, t_n)) & = & \mathcal{I}(f)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \\ \mathcal{I}(t_1 = t_2) & = & \mathcal{I}(t_1) = \mathcal{I}(t_2) \\ \mathcal{I}(p(t_1, \ldots, t_n)) & = & \mathcal{I}(p)(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \end{array}$$

Interpretation of Formulas (Recap)

Finally we interpret formulas

$\mathcal{I}(true)$	is always true
$\mathcal{I}(\neg\phi)$	*iff* $\mathcal{I}(\phi)$ is not true
$\mathcal{I}(\phi_1 \wedge \phi_2)$	*iff* both $\mathcal{I}(\phi_1)$ and $\mathcal{I}(\phi_2)$ are true
$\mathcal{I}(\forall x.\phi[x])$	*iff* for every $d \in \mathcal{D}$ we have that $\mathcal{I}(\phi[d])$ is true

Recall - a formula can have many consistent interpretations

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

Is this a Model?

$$p(a, b) \wedge p(b, a)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Is this a Model?

$$p(a, b) \wedge p(b, a)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Yes

Is this a Model?

$$p(a,b) \wedge p(b,a) \wedge \neg p(a,a)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Is this a Model?

$$p(a,b) \wedge p(b,a) \wedge \neg p(a,a)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

No

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

$\mathcal{I}(\forall x.\phi[x])$ *iff* for every $d \in \mathcal{D}$ we have that $\mathcal{I}(\phi[d])$ is true

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

$\mathcal{I}(\forall x.\forall y.p(x,y))$ *iff* for every $d \in \mathcal{D}$ we have that $\mathcal{I}(\forall y.p(d,y))$ is true

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

$\mathcal{I}(\forall y.p(1,y))$ is true and $\mathcal{I}(\forall y.p(2,y))$ is true

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

$\mathcal{I}(p(1,1))$ is true, $\mathcal{I}(p(1,2))$ is true, $\mathcal{I}(p(2,1))$ is true, $\mathcal{I}(p(2,2))$ is true

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

$\mathcal{I}(p(1,1))$ is true, $\mathcal{I}(p(1,2))$ is true, $\mathcal{I}(p(2,1))$ is true, $\mathcal{I}(p(2,2))$ is true

Is this a Model?

$$\forall x.\forall y.p(x,y)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

No

Is this a Model?

$$\exists x, y, z.(x \neq y \wedge x \neq z \wedge y \neq z)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Is this a Model?

$$\exists x, y, z.(x \neq y \wedge x \neq z \wedge y \neq z)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

No

Is this a Model?

$$\exists x, y, z.(x \neq y \wedge x \neq z \wedge y \neq z)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(c) & = & 3 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Is this a Model?

$$\exists x, y, z.(x \neq y \wedge x \neq z \wedge y \neq z)$$

$$\begin{array}{lcl} \mathcal{I}(a) & = & 1 \\ \mathcal{I}(b) & = & 2 \\ \mathcal{I}(c) & = & 3 \\ \mathcal{I}(p) & = & \{(1,2),(2,1),(1,1)\} \end{array}$$

Yes

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \end{array}$$

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{rcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \end{array}$$

No

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 4 \end{array}$$

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 4 \end{array}$$

No

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 2 \end{array}$$

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 2 \end{array}$$

Yes

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\forall x, y.(suc(x) = suc(y) \rightarrow x = y)$$

$$\begin{array}{rcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 2 \end{array}$$

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\forall x, y.(suc(x) = suc(y) \rightarrow x = y)$$

$$\begin{array}{rcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 2 \end{array}$$

No

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\forall x, y.(suc(x) = suc(y) \rightarrow x = y)$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 4 \\ \mathcal{I}(suc(4)) & = & 5 \\ \ldots & & \end{array}$$

Is this a Model?

$$\forall x.(x \neq suc(x)) \wedge \forall x.\exists y.(suc(x) = y) \wedge \neg\exists x.(zero = suc(x))$$

$$\forall x, y.(suc(x) = suc(y) \rightarrow x = y)$$

$$\begin{array}{lcl} \mathcal{I}(zero) & = & 1 \\ \mathcal{I}(suc(1)) & = & 2 \\ \mathcal{I}(suc(2)) & = & 3 \\ \mathcal{I}(suc(3)) & = & 4 \\ \mathcal{I}(suc(4)) & = & 5 \\ \ldots & & \end{array}$$

Yes, it's infinite

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

An interpretation satisfies a formula if it satisfies its universal closure.

A formula is satisfiable or consistent if it has a model.

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

An interpretation satisfies a formula if it satisfies its universal closure.

A formula is satisfiable or consistent if it has a model.

A formula is valid if *every* interpretation satisfies it.

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

An interpretation satisfies a formula if it satisfies its universal closure.

A formula is satisfiable or consistent if it has a model.

A formula is valid if *every* interpretation satisfies it.

A formula unsatisfiable or inconsistent if it has no models.

Validity vs Satisfiability/Consistency (Formally)

An interpretation satisfies a sentence if the sentence evaluates to true in it. We say that the interpretation is a model of that sentence

An interpretation satisfies a formula if it satisfies its universal closure.

A formula is satisfiable or consistent if it has a model.

A formula is valid if *every* interpretation satisfies it.

A formula unsatisfiable or inconsistent if it has no models.

Try writing down a consistent, a valid, and an inconsistent formula.

Entailment

We lift these notions to sets of formulas by interpreting them as conjunctions e.g. we can talk about a consistent set of formulas.

Entailment

We lift these notions to sets of formulas by interpreting them as conjunctions e.g. we can talk about a consistent set of formulas.

A formula ϕ is entailed by a set of formulas Γ if every model of Γ is also a model of ϕ, we write this

$$\Gamma \models \phi$$

also read as f is a consequence of Γ.

Entailment

We lift these notions to sets of formulas by interpreting them as conjunctions e.g. we can talk about a consistent set of formulas.

A formula ϕ is entailed by a set of formulas Γ if every model of Γ is also a model of ϕ, we write this

$$\Gamma \models \phi$$

also read as f is a consequence of Γ.

This is equivalent to $\Gamma \rightarrow \phi$ being valid.

Entailment

We lift these notions to sets of formulas by interpreting them as conjunctions e.g. we can talk about a consistent set of formulas.

A formula ϕ is entailed by a set of formulas Γ if every model of Γ is also a model of ϕ, we write this

$$\Gamma \models \phi$$

also read as f is a consequence of Γ.

This is equivalent to $\Gamma \rightarrow \phi$ being valid.

What if Γ is inconsistent?

Demonstrating Validity, Consistency, and Entailment

When there can be many interpretations the notion of truth can change.

Let us take an example

$$man(aristotle) \qquad human(cleopatra) \qquad \forall x.(man(x) \rightarrow human(x))$$

There exists a model where cleopatra is a man.

In every model aristotle is human.

Demonstrating Validity, Consistency, and Entailment

When there can be many interpretations the notion of truth can change.

Let us take an example

$$man(aristotle) \qquad human(cleopatra) \qquad \forall x.(man(x) \rightarrow human(x))$$

There exists a model where cleopatra is a man.

The set $\Gamma \cup man(cleopatra)$ is satisfiable or consistent

In every model aristotle is human.

Demonstrating Validity, Consistency, and Entailment

When there can be many interpretations the notion of truth can change.

Let us take an example

$$man(aristotle) \qquad human(cleopatra) \qquad \forall x.(man(x) \rightarrow human(x))$$

There exists a model where cleopatra is a man.

The set $\Gamma \cup man(cleopatra)$ is satisfiable or consistent

In every model aristotle is human.

The statement $man(aristotle)$ is entailed by Γ

Demonstrating Validity, Consistency, and Entailment

When there can be many interpretations the notion of truth can change.

Let us take an example

$$man(aristotle) \qquad human(cleopatra) \qquad \forall x.(man(x) \rightarrow human(x))$$

There exists a model where cleopatra is a man.

The set $\Gamma \cup man(cleopatra)$ is satisfiable or consistent

In every model aristotle is human.

The statement $man(aristotle)$ is entailed by Γ
The formula $\Gamma \rightarrow man(aristotle)$ is valid

Relation Between (In)Consistency and Validity

Let Γ be a set (conjunction) of formulas and ϕ be a sentence

The models of ϕ are exactly those that are not the models of $\neg\phi$

If Γ is inconsistent then $\Gamma \models \mathit{false}$.

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

Relation Between (In)Consistency and Validity

Let Γ be a set (conjunction) of formulas and ϕ be a sentence

The models of ϕ are exactly those that are not the models of $\neg\phi$

If Γ is inconsistent then $\Gamma \models \mathit{false}$.

If Γ is inconsistent then it has no models, *false* has no models, all models of Γ are also models of *false*.

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

Relation Between (In)Consistency and Validity

Let Γ be a set (conjunction) of formulas and ϕ be a sentence

The models of ϕ are exactly those that are not the models of $\neg\phi$

If Γ is inconsistent then $\Gamma \models \mathit{false}$.

If Γ is inconsistent then it has no models, *false* has no models, all models of Γ are also models of *false*.

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

If ϕ is true in all models of Γ then $\neg\phi$ must be true in no models of Γ

No Database Semantics

Notice we have not assumed database semantics

Can model

- Domain Closure Assumption by explicitly referring to the domain and its closure e.g.

$$\forall x.(\mathit{colour}(x) \rightarrow (x = \mathit{red} \vee x = \mathit{blue} \vee \ldots)$$

- Unique Names Assumption by explicitly stating this, although this only works for explicitly named things. Due to the open world interpretation, not everything needs to be explicitly named.

The Closed World Assumption is difficult to model and attempts to do so are not very friendly.

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 (due to negation-as-failure).

Negation in an Open World

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

Negation is used to restrict what can be true.

For example,

$$\forall x, y.((parent(x, y) \rightarrow \neg parent(y, x))$$

e.g. the *parent* relation is asymmetric.

Or simply

$$\neg loves(giles, marmite)$$

Knowledge Base Queries as Entailment

A (purely logical) knowledge base can be turned into a FOL formula by universally closing rules and conjoining all rules and facts.

A query can be turned into a FOL formula by existential closure e.g. query $\phi[X]$ becomes $\exists X.\phi[X]$

Given knowledge base Γ and a query ϕ both in FOL form, if the query is true then necessarily

$$\Gamma \models \phi$$

but the converse may not hold (if assuming database semantics).

Any Formula can be a 'Query' in FOL

In Prolog we are restricted by the kinds of queries we could ask.

```
car(X) :- hasWheels(X), hasEngine(X).
supercar(X) :- car(X), reallyFast(X).
```

Do all supercars have wheels?

Any Formula can be a 'Query' in FOL

In Prolog we are restricted by the kinds of queries we could ask.

```
car(X) :- hasWheels(X), hasEngine(X).
supercar(X) :- car(X), reallyFast(X).
```

Do all supercars have wheels?

Is the rule `supercar(X) :- hasWheels(X)` entailed?

Any Formula can be a 'Query' in FOL

In Prolog we are restricted by the kinds of queries we could ask.

```
car(X) :- hasWheels(X), hasEngine(X).
supercar(X) :- car(X), reallyFast(X).
```

Do all supercars have wheels?

Is the rule `supercar(X) :- hasWheels(X)` entailed?

$$\forall x.((wheels(x) \wedge engine(x)) \rightarrow car(x))$$
$$\forall x.((car(x) \wedge fast(x)) \rightarrow supercar(x)) \models \forall x.(wheels(x) \rightarrow supercar(x))$$

Differences with Datalog/Prolog

Expressiveness:

- FOL can use negation
- FOL can use existential quantification
- FOL can have multiple facts in the head (e.g. pure disjunction)
- FOL allows arbitrary 'queries'

However:

- Prolog is a programming language with many non-logical parts. It is Turing-complete
- Prolog and Datalog have Database Semantics, which can be helpful from a modelling perspective

How do we reason in FOL?

We had forward chaining in Datalog and backward chaining in Prolog

Forward chaining worked by generating consequences

Backward chaining worked by subgoal reduction

We have similar parallels for FOL reasoning. I'm going to focus on forward-ish techniques

Reasoning with Implications

If someone is rich then they are happy
I am rich

Reasoning with Implications

If someone is rich then they are happy
I am rich

Therefore, I am happy

Reasoning with Implications

$rich \rightarrow happy$
$rich$

$happy$

Reasoning with Implications

$rich \rightarrow happy$
$rich$

$happy$

This is captured by the well-known Modus Ponens rule

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

It's what we were applying in forward chaining.

Reasoning with Implications

$rich \rightarrow happy$
$rich$

$happy$

This is captured by the well-known Modus Ponens rule

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

It's what we were applying in forward chaining. Actually this is.

Reasoning with Implications

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

$happy(giles)$

This is captured by the well-known Modus Ponens rule

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

It's what we were applying in forward chaining. Actually this is.

Modus Ponens

More generally it is written

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

where mgu stands for most general unifier - recall unification from Prolog.

A unifier is any unifying substitution and most general means any other unifier is a special case.

Modus Ponens

More generally it is written

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

where mgu stands for most general unifier - recall unification from Prolog.

A unifier is any unifying substitution and most general means any other unifier is a special case.

Does this rule make sense from what we know about models?

Modus Ponens

More generally it is written

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

where mgu stands for most general unifier - recall unification from Prolog.

A unifier is any unifying substitution and most general means any other unifier is a special case.

Does this rule make sense from what we know about models?

To be a sound inference every model of the premises should be a model of the conclusion

Modus Ponens

More generally it is written

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

where mgu stands for most general unifier - recall unification from Prolog.

A unifier is any unifying substitution and most general means any other unifier is a special case.

Does this rule make sense from what we know about models?

To be a sound inference every model of the premises should be a model of the conclusion

Modus Ponens

More generally it is written

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

where mgu stands for most general unifier - recall unification from Prolog.

A unifier is any unifying substitution and most general means any other unifier is a special case.

Does this rule make sense from what we know about models?

To be a sound inference every model of the premises should be a model of the conclusion

Reasoning with Implications

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

Therefore, ?

Reasoning with Implications

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

Therefore, ?

Reasoning with Implications

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

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

Reasoning with 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 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 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 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

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 what we have in Prolog.

If $m = 1$ then a clause is definite - this is what we have in Datalog.

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.

What about Equality?

$$\begin{array}{ll} \neg rich(father(x)) \vee happy(x) & \\ rich(david) & \models happy(giles) \\ father(giles) = david & \end{array}$$

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

Paramodulation

The paramodulation rule lifts the idea behind resolution to equality

$$\frac{C \vee s = t \qquad l[u] \vee D}{(l[t] \vee C \vee D)\theta} \quad \theta = \mathrm{mgu}(s, u)$$

where u is not a variable.

For example

$$\frac{father(giles) = david \qquad \neg rich(father(x)) \vee happy(x)}{\neg rich(david) \vee happy(giles)}$$

where $u = father(x)$.

Lecture 9 First-Order Logic Reasoning and Transformation to Clausal Form

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Look at the saturation-based setting and the first step of transforming to clausal form

Learning Outcomes

By the end of this lecture you will be able to:

1. Recall the saturation loop and the main steps in it
2. Describe the clausal transformation pipeline
3. Transform general first-order formulas into clausal form
4. Identify points where this transformation could be optimised
5. Recall how to run Vampire to perform resolution and clausification

Unification (Recap)

In first-order logic every two terms have a most general unifier

A substitution σ is a unifier for two terms t_1 and t_2 if $\sigma(t1) = \sigma(t_2)$

I avoid giving a very formal definition of more general but a unifier σ_1 is more general than σ_2 if $\sigma_2(t)$ is always an instance of $\sigma_1(t)$

There is a relatively straightforward algorithm for generating most general unifiers called Robinsons Algorithm (does the obvious thing) but in reality we compute unifiers using special data structures.

Clauses and Resolution (Recap)

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.

Resolution works on clauses

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

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.

What about Equality?

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

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

What about Equality?

$$\neg rich(father(x)) \vee happy(x)$$
$$rich(david)$$
$$father(giles) = david$$
$$\neg happy(giles)$$
$$\neg rich(father(giles))$$

$father(x)$ and $david$ do not unify.

Paramodulation

The paramodulation rule lifts the idea behind resolution to equality

$$\frac{C \vee s = t \qquad l[u] \vee D}{(l[t] \vee C \vee D)\theta} \quad \theta = \mathrm{mgu}(s, u)$$

where u is not a variable.

Paramodulation

The paramodulation rule lifts the idea behind resolution to equality

$$\frac{C \vee s = t \qquad l[u] \vee D}{(l[t] \vee C \vee D)\theta} \quad \theta = \mathrm{mgu}(s, u)$$

where u is not a variable.

For example

$$\frac{father(giles) = david \qquad \neg rich(father(x)) \vee happy(x)}{\neg rich(david) \vee happy(giles)}$$

where $u = father(x)$ and therefore $\theta = \{x \mapsto giles\}$.

Special case: unit equalities

The demodulation rule works with unit equalities

$$\frac{s = t \qquad l[u] \vee D}{(l[t] \vee D)\theta} \quad \theta = \text{matches}(s, u)$$

Note that u must be an instance of s.

Why is this special? The premise $l[u] \vee D$ becomes redundant. We'll find out what that means properly next time but it means we can remove it.

Notice that we could apply this in either direction - this is going to lead to redundancy and later we will see that we should (where possible) order equalities so that we rewrite more complicated things with simpler things.

Equality Resolution

We have another special case, is the following ever true?

$$\forall x.f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Equality Resolution

We have another special case, is the following ever true?

$$\forall x.f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Functions in first-order logic are always total

Equality Resolution

We have another special case, is the following ever true?

$$\forall x.f(x) \neq f(a)$$

e.g. for every input to f the result is not equal to applying f to a.

Functions in first-order logic are always total

So, no. We have a rule called equality resolution

$$\frac{s \neq t \vee C}{C\theta} \quad \theta = \text{mgu}(s, t)$$

e.g. if we can make two terms equal then any disequality is false.

More Examples Using Vampire

Vampire is a first-order theorem prover

It works on Clauses and implements resolution (and lots of other rules)

It takes problems in either TPTP or STM-LIB format

```
cnf(one,axiom, ~rich(X) | happy(X)).
cnf(two,axiom, rich(giles)).
cnf(three, negated_conjecture, ~happy(giles)).
```

The above is TPTP format already in conjunctive normal form.

Saturation Based Reasoning

To decide $\Gamma \models \varphi$ we are going to follow the below steps

1. Let $NotDone = \Gamma \cup \{\neg\varphi\}$
2. Transform *NotDone* into clauses
3. Let *Done* be an empty set of clauses
4. Select a clause c from *NotDone*
5. Perform all inferences (e.g. resolution) between c and all clauses in *Done* putting any *new* children in *NotDone*
6. Move c to *Done*
7. If one of the children was *false* then **return valid**
8. If *NotDone* is empty stop otherwise go to 4
9. **return not valid**

Saturation Based Reasoning

To decide $\Gamma \models \varphi$ we are going to follow the below steps

1. Let $NotDone = \Gamma \cup \{\neg\varphi\}$
2. Transform *NotDone* into clauses
3. Let *Done* be an empty set of clauses
4. Select a clause c from *NotDone*
5. Perform all inferences (e.g. resolution) between c and all clauses in *Done* putting any *new* children in *NotDone*
6. Move c to *Done*
7. If one of the children was *false* then **return valid**
8. If *NotDone* is empty stop otherwise go to 4
9. **return not valid**

Transformation to Clausal Form

To go from general formula to set of clauses we're going to go through the following steps

1. Rectify the formula
2. Transform to *Negation Normal Form*
3. Eliminate quantifiers
4. Transform into conjunctive normal form

At any stage we might simplify using rules about *true* and *false*, e.g. $\mathit{false} \wedge \phi = \mathit{false}$, and remove tautologies, e.g. $p \vee p$.

Rectification

A formula is rectified if each quantifier binds a different variable and all bound variables are distinct from free ones.

To rectify a formula we identify any name clashes, pick one and rename it consistently. It is important to work out which variables are in the scope of a quantifier e.g.

$$\forall x.(p(x) \vee \exists x.r(x)) \qquad \textit{becomes} \qquad \forall x(p(x) \vee \exists y.r(y))$$

To automate this we typically rename bound variables starting with x_0 etc.

Negation Normal Form

Apply rules in a completely deterministic syntactically-guided way

$$
\begin{aligned}
\neg(F_1 \wedge \ldots \wedge F_n) &\Rightarrow \neg F_1 \vee \ldots \vee \neg F_n \\
\neg(F_1 \vee \ldots \vee F_n) &\Rightarrow \neg F_1 \wedge \ldots \wedge \neg F_n \\
F_1 \rightarrow F_2 &\Rightarrow \neg F_1 \vee F_2 \\
\neg\neg F &\Rightarrow F \\
\neg\forall x_1, \ldots, x_n F &\Rightarrow \exists x_1, \ldots, x_n \neg F \\
\neg\exists x_1, \ldots, x_n F &\Rightarrow \forall x_1, \ldots, x_n \neg F \\
\neg(F_1 \leftrightarrow F_2) &\Rightarrow F_1 \otimes F_2 \\
\neg(F_1 \otimes F_2) &\Rightarrow F_1 \leftrightarrow F_2 \\
F_1 \leftrightarrow F_2 &\Rightarrow (F_1 \rightarrow F_2) \wedge (F_2 \rightarrow F_1); \\
F_1 \otimes F_2 &\Rightarrow (F_1 \vee F_2) \wedge (\neg F_1 \vee \neg F_2).
\end{aligned}
$$

Where $\otimes$ is exclusive or. Can get an exponential increase in size.

Examples

$$(\forall x.p(x)) \rightarrow (\exists x.p(x))$$

$$(\forall x.(p(x) \vee q(x)) \leftrightarrow (\neg\exists x.(\neg p(x) \wedge \neg q(x)))$$

$$\forall x, y, z.((f(x) = y \wedge f(x) = z) \rightarrow y = z)$$

$$\forall x.((\exists y.p(x, y)) \rightarrow q(x)) \wedge p(a, b) \wedge \neg\exists x.q(x)$$

Also, which of the above statements are valid or inconsistent?

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.((pizza(x) \wedge pizza(y) \wedge x \neq y) \rightarrow better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\exists x.\forall y.(\neg pizza(x) \vee \neg pizza(y) \vee x = y \vee better(x, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$\exists x.\exists y.\exists z.(x \neq y \wedge lives\_in(x, z) \wedge lives\_in(y, z))$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\exists x.\phi[x])$ is true then there must be some domain constant d such that $\mathcal{I}(\phi[d])$ is true.

Let a be a fresh constant symbol (a new name for the object that has to exist). As it is fresh we can freely let $\mathcal{I}(a) = d$.
(This requires the Axiom of Choice)

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.\exists y.loves(x, y)$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Existential Quantifiers

There is a best kind of pizza
$\forall y.(\neg pizza(a) \vee \neg pizza(y) \vee a = y \vee better(a, y))$

There are two different people who live in the same house
$a \neq b \wedge lives\_in(a, c) \wedge lives\_in(b, c)$

Everybody loves somebody
$\forall x.loves(x, f(x))$

Let $\mathcal{I}$ be some interpretation. If $\mathcal{I}(\forall x.\exists y.\phi[x, y])$ is true then for every domain constant d_1 there must be some other domain constant d_2 such that $\mathcal{I}(\phi[d_1, d_2])$ is true. But how do we find d_2 given d_1?

Let f be a fresh function symbol whose interpretation can be made to *select* the necessary domain constant.

Dealing with Universal Quantifiers

Forget about them

Skolemisation

This process is called Skolemisation and the new symbols we introduce are called Skolem constants or Skolem functions.

The rules are simply

$$\begin{aligned} \forall x_1, \ldots, x_n F &\Rightarrow F \\ \exists x_1, \ldots, x_n F &\Rightarrow F\{x_1 \mapsto f_1(y_1, \ldots, y_m), \ldots, x_n \mapsto f_n(y_1, \ldots, y_m)\}, \end{aligned}$$

where f_i are fresh function symbols of the correct arrity and $y_1, \ldots y_m$ are the free variables of F e.g. they are the things universally quantified in the larger scope.

Remember that Skolem constants/functions act as witnesses for something that we know has to exist for the formula to be true.

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

No - the first one does not need to interpret a

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Yes - there's no negation, just make everything true

Validity or Satisfiability Preserving?

Do these two formulas have the same models?

$$\exists x.\forall y.p(x,y) \qquad \forall y.p(a,y)$$

No - the first one does not need to interpret a

Do the formulas both have models?

Yes - there's no negation, just make everything true

When we introduce new symbols we preserve satisfiability (the existence of models) not validity (the same models).

However, most transformations are stronger than this and preserve models on the initial signature.

CNF Transformation

Now the only connectives left should be $\wedge$ and $\vee$ and we need to push the $\vee$ symbols under the $\wedge$ symbols

The associated rule is

$$(A_1 \wedge \ldots \wedge A_m) \vee B_1 \vee \ldots \vee B_n \quad \Rightarrow \quad \begin{array}{ll} (A_1 \vee B_1 \vee \ldots \vee B_n) & \wedge \\ \ldots & \wedge \\ (A_m \vee B_1 \vee \ldots \vee B_n). & \end{array}$$

This can lead to exponential growth.

Looking at Vampire's clausification

Run

```
./vampire --mode clausify problem
```

on problem file `problem`

It will sometimes do a lot more than what we've discussed above.

Vampire performs lots of optimisations e.g. naming subformulas, removing pure symbols, or unused definitions.

Examples

$$(\forall x.p(x)) \rightarrow (\exists x.p(x))$$

$$(\forall x.(p(x) \vee q(x)) \leftrightarrow (\neg\exists x.(\neg p(x) \wedge \neg q(x)))$$

$$\forall x, y, z.((f(x) = y \wedge f(x) = z) \rightarrow y = z)$$

$$\forall x.((\exists y.p(x, y)) \rightarrow q(x)) \wedge p(a, b) \wedge \neg\exists x.q(x)$$

Also, which of the above statements are valid or inconsistent?

Optimisation (subformula naming)

To go from general formula to set of clauses we're going to go through the following steps

1. Rectify the formula
2. Transform to *Equivalence Negation Form*
3. Apply *naming* of subformulas
4. Transform to *Negation Normal Form*
5. Eliminate quantifiers
6. Transform into conjunctive normal form

Equivalence Negation Form

Push negations in but preserve equivalences

$$
\begin{array}{rcl}
\neg(F_1 \wedge \ldots \wedge F_n) & \Rightarrow & \neg F_1 \vee \ldots \vee \neg F_n \\
\neg(F_1 \vee \ldots \vee F_n) & \Rightarrow & \neg F_1 \wedge \ldots \wedge \neg F_n \\
F_1 \rightarrow F_2 & \Rightarrow & \neg F_1 \vee F_2 \\
\neg\neg F & \Rightarrow & F \\
\neg\forall x_1, \ldots, x_n F & \Rightarrow & \exists x_1, \ldots, x_n \neg F \\
\neg\exists x_1, \ldots, x_n F & \Rightarrow & \forall x_1, \ldots, x_n \neg F \\
\neg(F_1 \leftrightarrow F_2) & \Rightarrow & F_1 \otimes F_2 \\
\neg(F_1 \otimes F_2) & \Rightarrow & F_1 \leftrightarrow F_2
\end{array}
$$

Only get a linear increase in size.

Subformula Naming

We want to get to a conjunction of disjunctions but this process can 'blow up' in general e.g.

$$p(x,y) \leftrightarrow (q(x) \leftrightarrow (p(y,y) \leftrightarrow q(y))),$$

is equivalent to

$$\begin{array}{c}
p(x,y) \vee \neg q(y) \vee \neg p(y,y) \vee \neg q(x)) \\
p(x,y) \vee q(y) \vee p(y,y) \vee \neg q(x)) \\
p(x,y) \vee p(y,y) \vee \neg q(y) \vee q(x)) \\
p(x,y) \vee q(y) \vee \neg p(y,y) \vee q(x)) \\
q(x) \vee \neg q(y) \vee \neg p(y,y) \vee \neg p(x,y)) \\
q(x) \vee q(y) \vee p(y,y) \vee \neg p(x,y)) \\
p(y,y) \vee \neg q(y) \vee \neg q(x) \vee \neg p(x,y)) \\
q(y) \vee \neg p(y,y) \vee \neg q(x) \vee \neg p(x,y))
\end{array}$$

Subformula Naming

We can replace

$$p(x,y) \leftrightarrow (q(x) \leftrightarrow (p(y,y) \leftrightarrow q(y))),$$

by

$$p(x,y) \leftrightarrow (q(x) \leftrightarrow n(y));$$
$$n(y) \leftrightarrow (p(y,y) \leftrightarrow q(y)).$$

to get the same number of clauses but each clause is simpler (better for reasoning).

In the case when the subformula $F(x_1, \ldots, x_k)$ has only positive occurrences in G, one can use the axiom $n(x_1, \ldots, x_k) \rightarrow F(x_1, \ldots, x_k)$ instead of $n(x_1, \ldots, x_k) \leftrightarrow F(x_1, \ldots, x_k)$. This will lead to fewer clauses.

Subformula Naming

Assigning a name n to $F_2 \leftrightarrow F_3$ yields two formulas

$$F_1 \leftrightarrow n;$$
$$n \leftrightarrow (F_2 \leftrightarrow F_3),$$

where the second formula has the same structure as the original formula $F_1 \leftrightarrow (F_2 \leftrightarrow F_3)$.

When to Name Subformulas?

Vampire uses a heuristic that that estimates how many clauses a subformula will produce and names that subformula if that number is above a certain threshold.

Naming can not increase the number of clauses introduced but does not always reduce.

The idea is the same as the optimised structural transformation in the propositional case but we don't always apply it as the cost on reasoning is much higher here.

Lecture 10 First-Order Logic Saturation-Based Reasoning

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Give the idea behind the completeness of resolution, introduce ordered resolution, and discuss clause selection

Learning Outcomes

By the end of this lecture you will be able to:

1. State what it means for an inference system to be (refutationally) complete
2. Describe the general idea behind the model construction approach
3. Describe, with examples, a fair clause selection approach
4. Apply the given clause algorithm with resolution (etc) to a set of clauses

First-Order Logic Stuff

Syntax (propositional logic with predicates and quantifiers)

Semantics in terms of models

Clausal representation

Reasoning with Clauses using Resolution

Reasoning with Equality with Paramodulation (and Equality Resolution)

Transformation to Clausal Form

Today

Ground resolution is sound and complete

The completeness argument allows us to optimise its application

We can lift it to first-order resolution

Need fairness, get given clause algorithm

Ground Resolution: Soundness

We consider the ground case. Reminder:

$$\frac{l \vee C \qquad \neg l \vee D}{C \vee D}$$

where l is a ground atom and C, D are ground clauses.

This rule is sound, we only derive true things.

For any model $\mathcal{M}$ if $\mathcal{M} \models l \vee C$ and $\mathcal{M} \models \neg l \vee D$ then $\mathcal{M} \models C \vee D$.

Two cases

1. $\mathcal{M} \models l$ and therefore $\mathcal{M} \models D$
2. $\mathcal{M} \models \neg l$ and therefore $\mathcal{M} \models C$

Note that $l \vee \neg l$ is a tautology.

Ground Resolution: Completeness

We consider the ground case. Reminder:

$$\frac{l \vee C \qquad \neg l \vee D}{C \vee D}$$

where l is a ground atom and C, D are ground clauses.

This rule is refutationally complete, if it is unsat we can show it.

Let N be a set of ground clauses and N^* be the set saturated with respect to the above rule. Then $N \models \mathit{false}$ if and only if $\mathit{false} \in N^*$.

If direction by soundness of resolution.

Only if direction by constructing a model of N from N^* if $\mathit{false} \notin N^*$.

Ordering Clauses

A partial ordering (irreflexive, transitive) $\succ$ is well-founded if there exist no infinite chains $a_0 \succ a_1 \succ a_2 \succ \ldots$

Assume a well-founded partial order $\succ$ on ground atoms. We could use a simple 'dictionary' order, which would also be total.

First, lift to literals such that $\neg l \succ l$ for every atom l

Now, lift to clauses: $C \succ D$ if for every l in D/C there is a $l' \succ l$ in C/D

Example, given $p \succ q \succ r$

$$p \vee q \succ p \succ \neg q \vee r \succ q \vee r$$

Observations on Clause Ordering

$\succ$ on clauses is total and well-founded

Let $\max(C)$ be the maximal literal in C, this exists and is unique

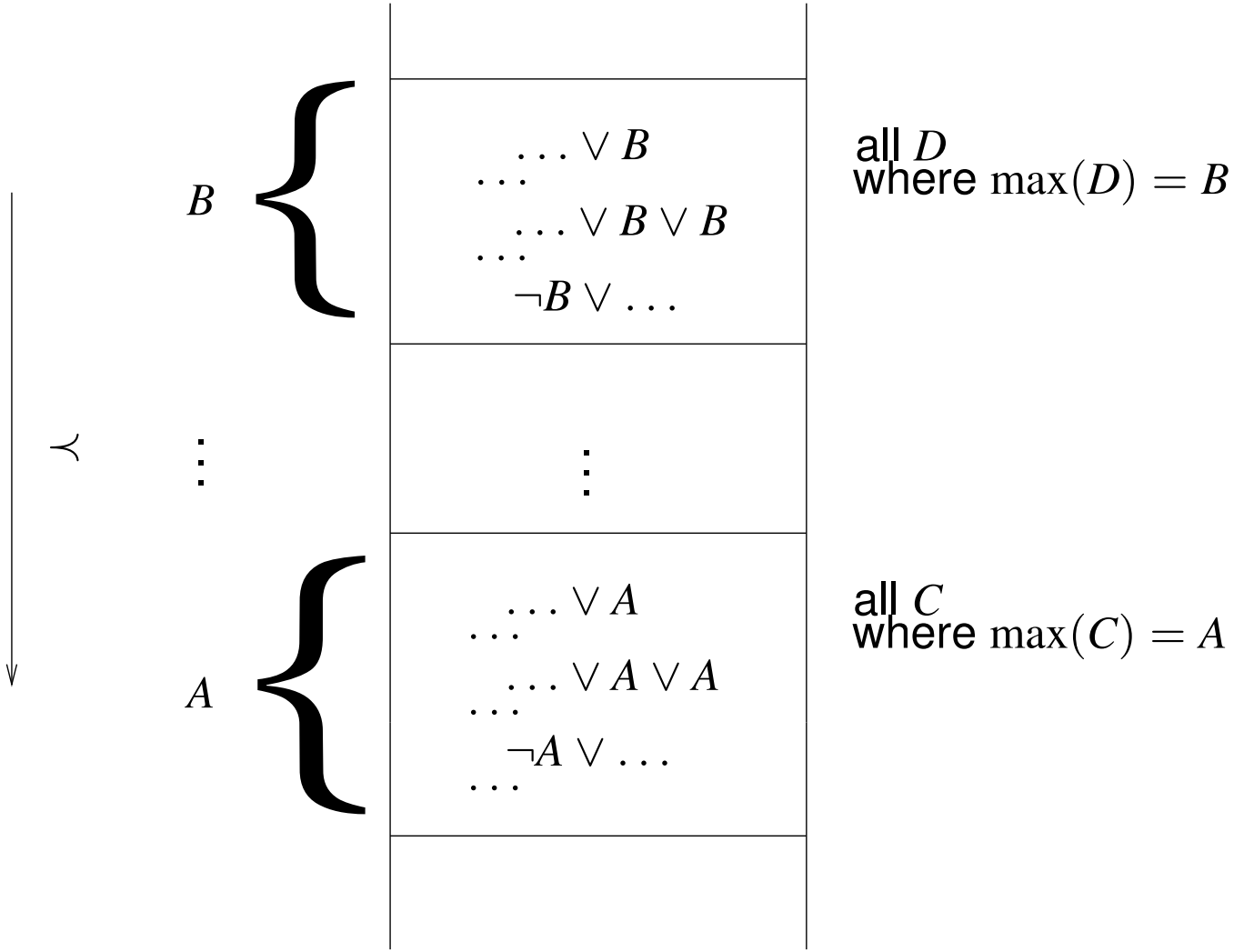
If $\max(C) \succ \max(D)$ then $C \succ D$

If $\max(C) = \max(D)$ but $\max(C)$ is neg and $\max(D)$ pos then $C \succ D$

This gives a stratification of clause sets by maximal literal

Stratified Clause Sets

Let $A \succ B$. Clause sets are then stratified in this form:



Model Construction

Idea:

- Build up an interpretation $\mathcal{M}$ incrementally
- Look at clauses from smallest to largest
- If C is already true in I then carry on
- Otherwise, make the maximal literal in C true in I

We use $\mathcal{M}_C$ for the interpretation after processing C and Δ_C for the new clauses produced by C. Then

$$\begin{aligned} \mathcal{M}_C &= \bigcup_{C \succ D} \Delta_D \\ \Delta_C &= \begin{cases} \{I\} & \text{if } \mathcal{M}_C \not\models C \text{ and } I = \max(C) \text{ and } I \text{ is pos} \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

If $\Delta_C = \{I\}$ we say that C is produces I and C is productive

Example

Let $p_5 \succ p_4 \succ p_3 \succ p_2 \succ p_1 \succ p_0$

clauses	$\mathcal{M}_C$	Δ_C	Remark
$\neg p_0$	$\emptyset$	$\emptyset$	true in $\mathcal{M}_C$
$p_0 \vee p_1$	$\emptyset$	$\{p_1\}$	
$p_1 \vee p_2$	$\{p_1\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_1 \vee p_2$	$\{p_1\}$	$\{p_2\}$	
$\neg p_1 \vee p_3 \vee p_0$	$\{p_1, p_2\}$	$\{p_3\}$	
$\neg p_1 \vee p_4 \vee p_3 \vee p_0$	$\{p_1, p_2, p_3\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_1 \vee \neg p_4 \vee p_3$	$\{p_1, p_2, p_3\}$	$\emptyset$	true in $\mathcal{M}_C$
$\neg p_4 \vee p_5$	$\{p_1, p_2, p_3\}$	$\{p_5\}$	

So $\mathcal{M} = \{p_1, p_2, p_3, p_5\}$

Observations

We use these next:

If $C = \neg l \vee C'$ then C does not produce l and no $D \succ C'$ produces l

Therefore, if l in $\mathcal{M}$ then some smaller clause most produce l

If C is productive then $\Delta_C \models C$, hence $\mathcal{M} \models C$

Model Existence

Let $\mathcal{M}$ be the model constructed from N^* where $\mathit{false} \notin N^*$.
We have $\mathcal{M} \models N^*$.

Proof by contradiction. Suppose $\mathcal{M} \not\models N^*$.

There must be a smallest C in N^* s.t. $\mathcal{M} \not\models C$

C is not productive, hence $l = \max(C)$ is negative

$C = \neg l \vee C'$, hence $\mathcal{M} \not\models C'$ and $\mathcal{M} \models l$

As $\mathcal{M} \models l$ there is some $D = l \vee D'$, $C \succ D$ s.t. D produces l

So $\mathcal{M} \not\models D'$ (as D produces l)

By resolution, $C' \vee D' \in N^*$ and $\mathcal{M} \not\models C' \vee D'$

but $C \succ C' \vee D'$ thus C is not the smallest such clause

Compactness

Compactness of propositional logic follows.

A set of propositional formulas N is unsatisfiable if and only if there is a finite subset of N that is unsatisfiable.

The if part is non-trivial.

If N is unsatisfiable then N^* is unsatisfiable, thus $\mathit{false} \in N^*$

There must be a finite number of resolution steps required to derive *false*

Let P be the clauses in the resolution proof and $M = P \cap N$

M is finite, M is unsatisfiable and $M \subseteq N$

Ordered Resolution

Given the previous model construction we can observe that certain inferences can be excluded and the model construction still works.

If we resolve on either a maximal or negative literal then we do all inferences required by model construction.

This gives us ordered resolution:

$$\frac{l_1 \vee C \qquad \neg l_1 \vee D}{(C \vee D)\theta}$$

where l_1 is maximal in $l_1 \vee C$.

We're also allowed to arbitrarily select at least one negative literal in a clause and restrict inferences to the selected literals.

Less Work

Two choices of inference.

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

Less Work

One choice.

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

Lifting

Now let N be a set of non-ground clauses.

Let $G_\Sigma(N)$ be the grounding of N using Σ

If N is saturated wrt non-ground resolution then $G_\Sigma(N)$ is saturated wrt ground resolution

We can apply the model construction with $G_\Sigma(N)$ and the result is a model of N

Compactness lifts in a similar way

We can do something similar with equality but much more work (and requires replacing paramodulation with something else)

Missing Rule

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{\theta(C \vee l_1)}\ \theta = \mathrm{mgu}(l_1, l_2)$$

Missing Rule

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{\theta(C \vee l_1)}\ \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Missing Rule

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{\theta(C \vee l_1)}\ \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

Missing Rule

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{\theta(C \vee l_1)}\ \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$4 \qquad p(u) \vee p(f(w)) \qquad (1, 2)$$

Missing Rule

Usually we also have the (positive) factoring rule

$$\frac{C \vee l_1 \vee l_2}{\theta(C \vee l_1)}\ \theta = \mathrm{mgu}(l_1, l_2)$$

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$\begin{array}{lll} 4 & p(u) \vee p(f(w)) & (1,2) \\ 5 & p(u) \vee \neg p(f(f(u))) & (1,3) \end{array}$$

Missing Rule

Usually we also have the (positive) factoring rule

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

which is required in some cases

$$\begin{array}{ll} 1 & p(u) \vee p(f(u)) \\ 2 & \neg p(v) \vee p(f(w)) \\ 3 & \neg p(x) \vee \neg p(f(x)) \end{array}$$

Resolvents

$$\begin{array}{lll} 4 & p(u) \vee p(f(w)) & (1,2) \\ 5 & p(u) \vee \neg p(f(f(u))) & (1,3) \end{array}$$

We're only going to get clauses with 2 literals.
However, we can factor (4) to $p(f(w))$
Resolving with (3) gives $\neg p(f(f(z)))$ then with $p(f(w))$ gives *false*

Fairness: Clause Selection

The above view is static, it assumes we have the saturated set

In reality we need to generate it but what if we infinitely delay performing an inference?

We lose the partial decidability

A saturation process is fair if no clause is delayed infinitely often

Two fair clause selection strategies:

- First-in first-out
- Smallest (in number of symbols) first
  (there are a finite number of terms with at most k symbols)

Given Clause Algorithm

input: *Init*: set of clauses;
var *active*, *passive*, *unprocessed*: set of clauses;
var *given*, *new*: clause;
active := $\emptyset$; *unprocessed* := *Init*;
loop
  while *unprocessed* $\neq \emptyset$
    new :=*pop*(*unprocessed*);
    if *new* $= \square$ **then return** *unsatisfiable*;
    add *new* to *passive*
  if *passive* $= \emptyset$ **then return** *satisfiable* or *unknown*
  given := *select*(*passive*); *(* clause selection *)*
  move *given* from *passive* to *active*;
  unprocessed :=*infer*(*given*, *active*); *(* generating inferences *)*

Complete Example

$$\left\{ \begin{array}{c} \forall x.(happy(x) \leftrightarrow \exists y.(loves(x,y))) \\ \forall x.(rich(x) \leftarrow loves(X, money)) \\ rich(giles) \end{array} \right\} \models happy(giles)$$

Lecture 11 Demonstrating FOL Reasoning

COMP24412: Symbolic AI

Giles Reger

March 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Look at things required to make the previous theory work (in Vampire)

Learning Outcomes

By the end of this lecture you will be able to:

1. Order terms and literals using KBO
2. Apply the given clause algorithm
3. (For the lab) run Vampire

First-Order Logic Stuff

Syntax (propositional logic with predicates and quantifiers)

Semantics in terms of models

Clausal representation and reasoning with Clauses using Resolution

Reasoning with Equality with Paramodulation (and Equality Resolution)

Transformation to Clausal Form

Completeness via Model Construction

Ordered Resolution and clause selection (more today)

Important: naming conventions

On Friday I was asked how you can tell what are variables and what are constants

I will always use letters from the end of the alphabet (u,v,w,x,y,z) for variables and letters from the start of the alphabet (a,b,c,d) for constants. I'll also tend to use f,g,h for function symbols, p,q,r for predicates, and s,t for terms. I will sometimes use *sk* for Skolem constants/functions.

If you prefer you can use capital letters X,Y for variables as in Prolog.

Note that in inference rules we have *meta-variables* e.g. they are templates

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

The Hidden Rule

Are these two clauses consistent?

$$p(x, a) \qquad \neg p(b, x)$$

Trying to resolve them requires us to unify $p(x, a)$ with $p(b, x)$, can we?

But they represent $(\forall x.p(x, a)) \wedge (\forall x.p(b, x))$, which is unsat

Equivalent to $(\forall x.p(x, a)) \wedge (\forall y.p(b, y))$ i.e. clauses $p(x, a)$ and $\neg p(b, y)$

Hidden rule: Naming Apart.
When unifying literals from two different clauses you should first rename variables so that the literals do not share literals.

Sometimes I do this implicitly e.g. in $p(x)$ and $\neg p(x)$ we should rename to $p(x)$ and $\neg p(y)$ and then apply $\{x \mapsto y\}$ but that's tedious.

What Non-Ground Clauses Mean

It is helpful to get an intuition for this. What does

$$p(x, y) \rightarrow q(x, z)$$

mean?

We can think of non-ground clauses as representing all instances of that clause. This is probably infinite.

When we reason with non-ground clauses we reason with the those sets.

We usually need to instantiation the ground-clause a bit first so that we are reasoning with the sets that we want. All non-ground inference rules can be split into two phases: instantiation and then 'ground' reasoning.

Ordered Resolution

On Friday I introduced ordered ground resolution

Lifting this to the non-ground case requires us to be a bit more explicit about orderings

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a, g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a well-founded total ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total?

Ground Term Ordering

Let the number of (function or variable) symbols in a term be its weight given by a function w e.g. $w(f(a, g(x))) = 4$.

Let $\succ_S$ be an ordering on function symbols. We define a well-founded total ordering on ground terms such that $s \succ_G t$ if

1. $w(s) > w(t)$, or
2. $w(s) = w(t)$ and $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ and either
   - $f \succ_S g$, or
   - $s_i \succ_G t_i$ for some i and for all $j < i$, $s_j = t_j$

Why is this total?

Examples (given $f \succ_S g \succ_S h \succ_G a \succ_G b$):

$$f(a) \succ_G a \qquad f(a) \succ_G h(a) \qquad g(a, f(a)) \succ_G g(a, f(b))$$

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial?

Non-Ground Term Ordering

We lift $\succ_G$ to non-ground terms s and t such that $s \succ_N t$ if

1. For each variable x, the number of x in s is $\geq$ that in t, and
2. Either $s \succ_G t$ or $t = x$ and $s = f^n(x)$ for $x > 0$

Why do we need (1)? Consider $f(g(a), x)$ and $f(x, x)$.
Why is this partial? Consider $f(x)$ and $f(b)$ where $a \succ_S b \succ_S c$

Examples (given $f \succ_S g \succ_S a$):

$$f(x) \succ_N g(x) \qquad f(x) \succ_N x \qquad g(x, f(y)) \succ_N g(x, a)$$

What about $g(x, y)$ and $g(y, x)$?

This ordering is $\succ_{KBO}$ the Knuth-Bendix Ordering (KBO) used in Vampire

Ordered Resolution

$\succ_{KBO}$ lifts to predicates directly (extend $\succ_S$ to predicate symbols).

Lift $\succ_{KBO}$ to literals: $\neg l \succ l$ for every atom l, treat $=$ as biggest predicate.

Need a slightly different characterisation of ordered resolution using selection functions - we perform resolution on selected literals.

Ordered Resolution is then

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

where l_1 and $\neg l_2$ are selected.

A selection function is well-behaved if it either selects (i) at least one negative literal, or (ii) all maximal literals. This ensures completeness.

Fairness: Clause Selection

Firstly, we want to ensure that every clause is eventually selected

Secondly, we may want to select 'good' clauses earlier

A saturation process is fair if no clause is delayed infinitely often

Two fair clause selection strategies:

- First-in first-out
- Smallest (in number of symbols, e.g. weight) first
  (there are a finite number of terms with at most k symbols)

Vampire chooses between the two with a given ratio

Given Clause Algorithm

input: *Init*: set of clauses;
var *active*, *passive*, *unprocessed*: set of clauses;
var *given*, *new*: clause;
active := $\emptyset$; *unprocessed* := *Init*;
loop
  while *unprocessed* $\neq \emptyset$
    new :=*pop*(*unprocessed*);
    if *new* $= \square$ **then return** *unsatisfiable*;
    add *new* to *passive*
  if *passive* $= \emptyset$ **then return** *satisfiable* or *unknown*
  given := *select*(*passive*); *(* clause selection *)*
  move *given* from *passive* to *active*;
  unprocessed :=*infer*(*given*, *active*); *(* generating inferences *)*

Example 1

$$\left\{ \begin{array}{c} \forall x.(happy(x) \leftrightarrow \exists y.(loves(x,y))) \\ \forall x.(rich(x) \rightarrow loves(x, money)) \\ rich(giles) \end{array} \right\} \models happy(giles)$$

Example 2

$$\left\{ \begin{array}{c} \forall x.(require(x) \rightarrow require(depend(x))) \\ depend(a) = b \\ depend(b) = c \\ require(a) \end{array} \right\} \models require(c)$$

Vampire

Automated theorem prover for first-order logic

Implements everything we've talked about (and more)

Very efficient/powerful (wins lots of competitions)

Used as a back-box solver in lots of other things (in academia and industry)

Available from `https://vprover.github.io`

You have to use Vampire in lab 2b

TPTP

Language for describing first-order formulas in ASCII format.

See `http://tptp.cs.miami.edu/~tptp/`

For example

```
fof(one,axiom, ![X] : (happy(X) <=> (?[Y] : loves(X,Y)))).
fof(two,axiom, ![X] : (rich(X) => loves(X,money))).
fof(three,axiom, rich(giles)).
fof(goal,conjecture, happy(giles)).
```

Important - `axiom` for knowledge base, `conjecture` for goal and Vampire will do the negation for you.

Run Vampire

Vampire is a command-line tool

Run using

```
./vampire <options> <file>
```

I recommend setting certain options to make Vampire behave more similarly to the things defined in this course

```
-updr off -fde none -nm 0 -av off -fsr off -s 1 -sa discount
```

First 3 options are preprocessing optimisations, last 4 are proof search options.

Vampire

Lots of extra bits not in this course:

- Superposition (ordered paramodulation)
- Simplifications (saturation up to redundancy)
- Clause splitting
- Finite model building
- Arithmetic and datatype reasoning
- Lots and lots of proof search heuristics

If you're interested, especially if you'd like to do a 3rd year project in this area, then come and chat to me about it

Lecture 12 Beyond (and Beneath) FOL

COMP24412: Symbolic AI

Giles Reger

March 2019

Aim and Learning Outcomes

The aim of this lecture is to:

Understand how FOL relates to other knowledge representation formalisms

Learning Outcomes

By the end of this lecture you will be able to:

1. Recall that adding arithmetic to FOL makes it fully undecidable
2. Recall a number of different formalisms and describe the general way in which they are related to FOL (e.g. more or less expressive)

Today

Beyond First-Order Logic

- Adding arithmetic
- Higher-order logic

Beneath First-Order Logic

- Fragments of FOL
- Description Logic
- Modal Logic

Arithmetic is Useful

People modelling in first-order logic usually assume that they have access to arithmetic. However, it does not come for free.

There are different ways of encoding arithmetic in first-order logic but the most general are undecidable.

Often full arithmetic is not required and it is better to encode orderings or counting in some other way.

Different Kinds of Arithmetic

Presburger Arithmetic has symbols $0, succ, +, =$ defined by some axioms

$$\begin{aligned} &0 \neq x + 1 \\ &(x + 1 = y + 1) \rightarrow x = y \\ &(x + 0) = x \\ &x + (y + 1) = (x + y) + 1 \end{aligned}$$

and *induction* e.g. for every formula $\phi[n]$

$$(\phi[0] \wedge \forall x.(\phi[x] \rightarrow \phi[x + 1])) \rightarrow (\forall y.\phi[y])$$

but induction is not finitely axiomitisable - we cannot represent arithmetic in first order logic.

However, by itself Presburg arithmetic is decidable.

Different Kinds of Arithmetic

Peano Arithmetic has symbols $0, succ, +, \times, =, \leq$ defined by some axioms

$$\forall x.(x \neq succ(x))$$
$$\forall x, y.(succ(x) = succ(y) \rightarrow x = y)$$
$$\forall x.(x + 0 = x)$$
$$\forall x, y.(x + succ(y) = succ(x + y))$$
$$\forall x.(x \times 0 = 0)$$
$$\forall x, y.(x \times succ(y) = x + (x \times y))$$
$$\forall x, y.(x \leq y \leftrightarrow \exists z.(x + z = y))$$

and *induction* e.g. for every formula $\phi[n]$

$$(\phi[0] \wedge \forall x.(\phi[x] \rightarrow \phi[succ(x)])) \rightarrow (\forall y.\phi[y])$$

which, again is not finitely axiomitisable.

Peano arithmetic is incomplete and undecidable.

Validity Modulo a Theory

Let $\Sigma_{\mathcal{T}}$ be an interpreted signature e.g. $+, \times, \geq, 1, 2, 3, \ldots$

Let a theory $\mathcal{T}$ over $\Sigma_{\mathcal{T}}$ be a class of interpretations that fix some interpretation for $\Sigma_{\mathcal{T}}$. Often this class is singular e.g. we only interpret $+$ in one way.

An interpretation is consistent with $\mathcal{T}$ if it is consistent on $\Sigma_{\mathcal{T}}$

A formula is consistent modulo $\mathcal{T}$ if it has a model consistent with $\mathcal{T}$. It is valid if it is true in all interpretations consistent with $\mathcal{T}$.

If we saturate then we can build a model, but it is not guaranteed that this model is consistent with $\mathcal{T}$. We cannot use model construction argument.

Reasoning with Arithmetic

Often we just use Integer Arithmetic with constants $0, 1, 2, 3, \ldots$ that interprets $+, -, \times, =\leq$ etc directly on these. When we add division we get an infinite set of models where division by 0 can be interepreted arbitrarily. Clearly, we can do as much as in Peano arithmetic.

Just add some things that we know are true e.g.

$$x + 0 = x$$
$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$x + 1 > x$$

In Vampire we do other clever reasoning tricks (evaluation of ground things, calling out to ground decision procedures for sub-problems etc).

There are tools called Satisfiability Modulo Theories (SMT) solvers that work differently from Vampire by model building for quantifier-free problems with theories.

Quantifying over Functions/Predicates/Sets

There are some things we cannot say in first-order logic, for example the induction schema we saw earlier where we want to quantify over all predicates in the language.

Something we cannot express in first-order logic is reachability (the reason why is quite complex) but we can in higher-order logic

$$\forall P(\forall x, y, z \left(\begin{array}{l} P(x,x) \wedge \\ (P(x,y) \wedge P(z,y) \rightarrow P(x,z)) \wedge \\ (R(x,y) \rightarrow P(x,y)) \end{array} \right) \rightarrow P(u,v))$$

In program specification/verification we want to reason over objects representing programs and requirements (predicates on states).

Higher-order Logic

In first-order logic we have variables representing individuals and we can quantify over them.

In second-order logic we have variables representing sets of individuals (or functions on individuals) and we can quantify over them.

In third-order logic we have variables representing sets of sets of individuals...

We call second-order logic and above higher-order logic

λ-calculus

This is a calculus of functions. The standard building-block is the anonymous function $\lambda x.E[x]$ which can be read as a function that takes a value for x and evaluates E with x replaced by that value (similar to functions we're familiar with).

We can then have functions that take other functions as arguments and can return functions.

Functions can then be applied to each other e.g.

$$(\lambda x.\lambda y.xy)(\lambda x.x) \rightarrow_\beta (\lambda y.(\lambda x.x)y) \rightarrow_\eta \lambda x.x$$

This is a big and interesting topic but we don't have any more space here.

It is a useful term language for higher-order logic as λ-terms witness function variables

Reasoning in Higher-order Logic

Higher-order logic is clearly undecidable

Standard methods for reasoning in it are either...

Interactive e.g. using a proof assistant to allow a human to make reasoning steps (and suggesting possibly good reasoning steps)

or

Approximate by a translation to first-order logic that preserves inconsistency (this is what we do in Vampire)

Less Expressive/More Efficient

Find things that are decidable but still usefully expressive:

- Propositional logic (QBF, PLFD)
- Well-behaved First-order fragments
- (Some) Description Logics
- (Some) Modal Logics

Some 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.

The following logics often target these fragments to ensure decidability.

Description Logic

A family of logics that are usually decidable. They are used for describing ontologies. The terminology is different from what we're used to.

In description logic we separate facts in the $\mathcal{A}$-Box and rules in the $\mathcal{T}$-Box

Individuals belong to Concepts and may be related by Roles.

Concepts are sets of elements (unary predicates)

Roles relate two individuals (binary relations/predicates)

Complex concepts are logical combinations of concepts/roles

Facts assert individuals belong to concepts or roles

Rules capture relationships between complex concepts

Description Logic: Concepts

The most basic description logic

(In $\mathcal{ALC}$) Complex concepts:

- $A \sqcap B$: things that are A and B
- $A \sqcup B$: things that are A or B
- $\neg A$: things that are not A
- $\exists r.C$ things that are related by r to things that are C
- $\forall r.C$ things where all r related things are C

Examples:

Somebody that has a human child	$\exists$hasChild.Human
Somebody that only drinks beer	$\forall$drinks.Beer
Somebody that is either French or knows somebody who is	French $\sqcup \exists$knows.French

Rules and Reasoning

Rules are of the form $C \sqsubseteq D$ e.g. everything that is a C is also a D

$C \equiv D$ is the same as $C \sqsubseteq D$ and $D \sqsubseteq C$

e.g. $\text{Father} \equiv \text{Man} \sqcap \exists \text{hasChild.Human}$

An ontology is a set of facts and rules ($\mathcal{A}$-box and $\mathcal{T}$-box)

The semantics are defined in terms of interpretations (should be familiar)

Standard reasoning problems include

- Is an ontology consistent
- Is an individual in a concept (entailment)
- Is one concept subsumed by another (entailment)

Embedding in FOL

Introduce translation function t_x that maps into FOL formula with free x

Concepts map directly to FOL predicts, $t_x(A) = A(x)$

Complex Concepts map to logical combinations e.g.

$$t_x(\exists r.C) = \exists y.r(x, y) \wedge t_y(C)$$

The resulting FOL formulas are in the two-variable fragment and the guarded fragment.

This is for the simplest description logic $\mathcal{ALC}$. There are lots of more complicated description logics with extra features where the translation is less straightforward.

Propositional Modal Logic

It would be nice to be able to not only talk about what is true but when it is true (when in a general sense)

Modal logic allows us to do this. In English a *modal* qualifies a statement.

In modal logic we typically have two modal operators $\Diamond$ and $\Box$

Traditionally $\Diamond P$ means *Possibly* P whereas $\Box P$ means *Necessarily* P

For example,

$$\neg\Diamond win \rightarrow \Box\neg win$$
$$\Box(rain \wedge wind) \rightarrow \Box rain$$
$$(\Diamond rain \wedge \Diamond wind) \rightarrow \Diamond(rain \vee wind)$$

Semantics and Flavours

The semantics of modal logic is given by something called a Kripke structure, which is really just a graph. We have a relation R between worlds where different propositions are true in each world. $\Box$ then means *in all worlds adjacent by* R and $\Diamond$ means *in some worlds adjacent by* r.

We get different kinds of modal logic depending on how we control r, or equivalently which axioms about $\Box$ and $\Diamond$ we assume.

For example, reflexivity of R or $\Box A \rightarrow A$, and transitivity of R or $\Box A \rightarrow \Box\Box A$ gives us temporal logic where $\Box$ means all futures and $\Diamond$ means some

Other popular flavours in AI (particularly agent-based reasoning) are epistemic logic where modalities correspond to knowledge and doxastic logic where they correspond to belief.

Embedding in FOL

We use the adjacency relation R and a predicate $holds(x, y)$ that is true if x is true in world y

We can then encoding the meaning of modal formulas e.g.

$$holds(\Box p, u) \leftrightarrow \forall v.(R(u, v) \rightarrow holds(p, v))$$

The satisfiability/validity of a modal formula is the existential/universal closure of the resulting translation

We also need to add the axioms e.g. reflexivity and transitivity of R

We actually have to do quite a bit of extra work to get things into a decidable fragment, but we can

Decidability does not necessarily mean more efficient

A decision procedure is good because it will terminate in finite time, this does not mean that time is short.

Many fragments/logics also have strong complexity bounds on specific reasoning problems, which can help. But these are upper bounds.

In practice, it might be that a less efficient method, or an incomplete one, solves particular instances of problems faster.

This is generally our experience with Vampire.

Finally, Knowledge Engineering?

I have told you about lots of different formalisms and how to reason in some of them.

You have had to model different domains in some of these formalisms but there has been no general method for this.

At a high level *knowledge engineering* consists of the following steps:

1. Identify what the knowledge will be used for
2. Find the knowledge (where is it/who knows it)
3. Decide on a vocabulary e.g. names for predicates/functions etc
4. Decide on formalism to use
5. Encode general/domain knowledge
6. Encode the specific problem
7. Use the knowledge base (pose queries)
8. Debug the knowledge base

Summary of Part 1

Datalog: syntax, model-based semantics, matching, forward chaining

Prolog: more syntax, unification, backward chaining

First-order logic: syntax, model-based semantics, clausal representation/translation, resolution and paramodulation, completeness argument, orderings and ordered resolution, given clause algorithm

Relation to other logics (today)