Lecture 3 Reasoning in Datalog

COMP24412: Symbolic AI

Giles Reger

February 2019

Aim and Learning Outcomes

The aim of this lecture is to:

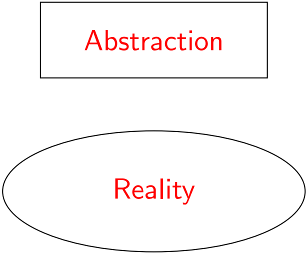
Introduce you to the main concepts around *quering a knowledge base* and how these are concretely realised in the *Datalog* language.

Learning Outcomes

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

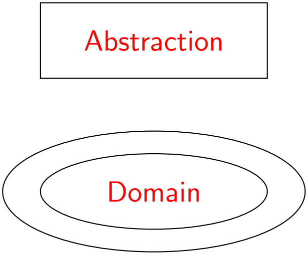
1. Describe what it means to *query* a knowledge base
2. Define *matching* and compute matching substitutions
3. Apply the *forward chaining* algorithm to find consequences of a knowledge base
4. Explain certain optimisations of the algorithm

In General



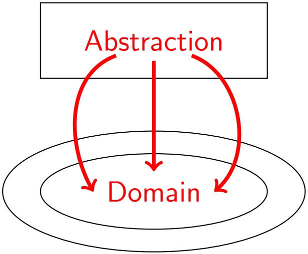
Recap

In General

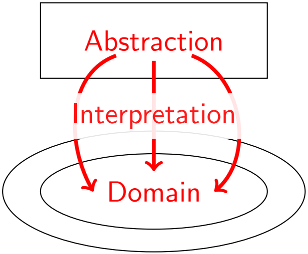


Recap

In General

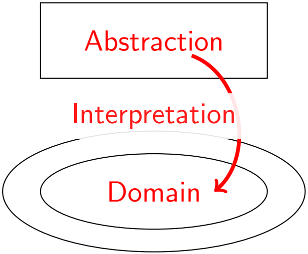


In General



Recap

In General

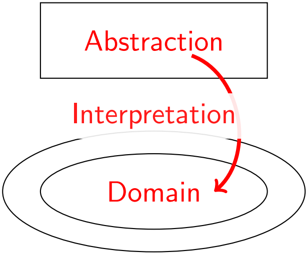


Database Semantics

- 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. loves(giles, cheese)

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

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 $\text{ancestor}(\text{giles}, \text{adam})$ true?

Or, more interestingly, they can contain variables e.g. give me all ancestors of giles or more formally all X such that $\text{ancestor}(\text{giles}, \text{X})$ 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

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

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
$\mathsf{happy}(\mathsf{giles})$	$\mathsf{happy}(X)$	$\{X \mapsto \mathsf{giles}\}$
$\mathsf{loves}(\mathsf{giles}, \mathsf{cheese})$	$\mathsf{loves}(X, \mathsf{cheese})$	$\{X \mapsto \mathsf{giles}\}$
$\mathsf{loves}(\mathsf{giles}, \mathsf{cheese})$	$\mathsf{loves}(X, Y)$	$\{X \mapsto \mathsf{giles}, Y \mapsto \mathsf{cheese}\}$
$\mathsf{happy}(\mathsf{giles})$	$\mathsf{happy}(\mathsf{giles})$	$\{\}$

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** $\bot$;
 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 $\bot$
 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** $\bot$;
- **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** $\bot$
- **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 *name₁ and name₂ 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$\}$)

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

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

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

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

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$\}$
- $args_1[0] = X$
- $args_2[0] =$ 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), $\{X \mapsto \text{giles}\}$)

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

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$\}$)

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

- $args_1[1] = Y$
- $args_2[1] = \text{mark}$

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), $\{X \mapsto \text{giles}\}$)

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

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

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

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

Computing Matching Substitutions

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

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

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

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

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

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

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

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

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

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

Computing Matching Substitutions

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

  if *$name_1$ and $name_2$ are different* **then return** $\bot$;
  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 $\bot$
  end
  return σ

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

Matching A Rule Body

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

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

Matching A Rule Body

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

def match(*body*, $\mathcal{F}$)**:**

```
matches = {∅}
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*, $\mathcal{F}$*)*:
  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 $= \{\emptyset\}$

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 $\sigma_2 \neq \bot$ **then** new.add(σ_2);
      end
    end
    matches $=$ new
  end
  return *matches*

matches $= \{\emptyset\}$

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

Matching A Rule Body

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

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

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

Matching A Rule Body

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

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

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

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

$\text{new} =$

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

$\sigma_1 = \emptyset$

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

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

Matching A Rule Body

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

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

matches $= \{\emptyset\}$

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

new $=$

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

$\sigma_1 = \emptyset$

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

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
```

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

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

$\text{new} =$

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \perp$

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*, $\mathcal{F}$):
    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 $=$
$\{\ \{X \mapsto \text{giles}, Y \mapsto \text{mark}\}\ \}$

$\sigma_1 = \emptyset$
$f_2 = \text{man(giles)}$
$\sigma_2 = \perp$

Matching A Rule Body

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

def match(*body*, $\mathcal{F}$)**:**
    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 $\sigma_2 \neq \bot$ **then** new.add(σ_2);
            end
        end
        matches $=$ new
    end
    return *matches*

matches $= \{\emptyset\}$

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

new $=$

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

$\sigma_1 = \emptyset$

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

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*, $\mathcal{F}$)**:**
    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 $\sigma_2 \neq \bot$ **then** new.add(σ_2);
            end
        end
        matches $=$ new
    end
    return *matches*

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

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

Matching A Rule Body

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

def match(*body*, $\mathcal{F}$)**:**

  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 $\sigma_2 \neq \bot$ **then** new.add(σ_2);
      end
    end
    matches $=$ new
  end
  return *matches*

matches $= \{\emptyset\}$

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

new $=$

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

$\sigma_1 = \emptyset$

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

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

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

matches $= \{\emptyset\}$

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

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

$\sigma_1 = \emptyset$

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

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*, $\mathcal{F}$)**:**
  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 $\sigma_2 \neq \bot$ **then** new.add(σ_2);
      end
    end
    matches = new
  end
  return *matches*

matches $= \{\emptyset\}$

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

new $=$

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

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
```

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

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

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

$\sigma_1 = \emptyset$

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

$\sigma_2 = \bot$

Matching A Rule Body

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

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** $\sigma_2 \neq \bot$ **then** new.add(σ_2);
    - **end**
  - **end**
  - matches $=$ new
- **end**
- **return** *matches*

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

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

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)**:**

$\quad \mathcal{F} = \emptyset$; $new = \mathcal{F}_0$

$\quad$ **do**

$\quad\quad \mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$

$\quad\quad$ **for** $body \Rightarrow head \in \mathcal{RU}$ **do**

$\quad\quad\quad$ **for** $\sigma \in \mathrm{match}(body, \mathcal{F})$ **do**

$\quad\quad\quad\quad$ **if** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$

$\quad\quad\quad$ **end**

$\quad\quad$ **end**

$\quad$ **while** $new \neq \emptyset$

$\quad ans = \emptyset$

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

$\quad$ **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 $\sigma \in \mathrm{match}(body, \mathcal{F})$ **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** $\sigma = \mathrm{match}(q, f, \emptyset)$; **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

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** $\sigma \in \mathrm{match}(body, \mathcal{F})$ **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** $\sigma = \mathrm{match}(q, f, \emptyset)$; **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)**:**

$\quad \mathcal{F} = \emptyset$; $new = \mathcal{F}_0$

$\quad$ **do**

$\quad\quad \mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$

$\quad\quad$ **for** $body \Rightarrow head \in \mathcal{RU}$ **do**

$\quad\quad\quad$ **for** $\sigma \in \mathrm{match}(body, \mathcal{F})$ **do**

$\quad\quad\quad\quad$ **if** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$

$\quad\quad\quad$ **end**

$\quad\quad$ **end**

$\quad$ **while** $new \neq \emptyset$

$\quad ans = \emptyset$

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

$\quad$ **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)**:**

$\quad \mathcal{F} = \emptyset$; $new = \mathcal{F}_0$

$\quad$ **do**

$\quad\quad \mathcal{F} = \mathcal{F} \cup new$; $new = \emptyset$

$\quad\quad$ **for** $body \Rightarrow head \in \mathcal{RU}$ **do**

$\quad\quad\quad$ **for** $\sigma \in \text{match}(body, \mathcal{F})$ **do**

$\quad\quad\quad\quad$ **if** $\sigma(head) \notin \mathcal{F}$ **then** $new.add(\sigma(head))$

$\quad\quad\quad$ **end**

$\quad\quad$ **end**

$\quad$ **while** $new \neq \emptyset$

$\quad ans = \emptyset$

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

$\quad$ **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** $\sigma \in \mathrm{match}(body, \mathcal{F})$ **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** $\sigma = \mathrm{match}(q, f, \emptyset)$; **if** $\sigma \neq \bot$ **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!