



Carleton
UNIVERSITY

Computational Logic and Automated Reasoning

Chapter 0: A Warm-Up Appetizer

Leopoldo Bertossi

Carleton University

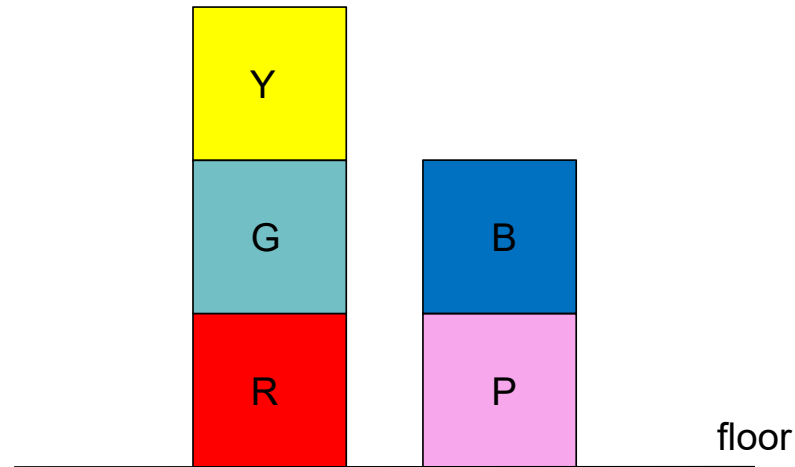
Ottawa, Canada

The Context

- In general terms, this is a course on logic in computer science (CS) and computing
- Logic as investigated in math and CS
- Logic is at the very root of CS
- Logic is an area of computer science
- Logic is at the basis and in the practice of several areas of CS
 - AI, SE, hardware/chip design/verification, programming, theory, data management (relational DBs are logic at work), etc.
- We are interested in logic for data management, knowledge representation (KR), and (logic) programming
- Logic-based KR is about:

- Representing knowledge in computers in symbolic form, and
- Using that knowledge in/by computer systems, mainly through some form of symbolic reasoning
- KR is an important and large area of AI
- Logic-based KR systems can be extended with representation of uncertainty
- Newer areas: Probabilistic logic programming, statistical relational learning (and representations), probabilistic ontologies, etc.
- Logic-based KR systems can be combined with other forms of KR, e.g. neural-network-based systems (or deep-learning systems)
- They can be used to do re-engineering of deep-learning systems, making explicit and declarative the knowledge they “encode”, which otherwise stays hidden in a black-box; e.g. producing rules (e.g. a NN trained to do entity-resolution)

Computational Representations and Models



This is our external, outside reality

How can we:

- Represent it in the computer?
- Model it in mathematical terms?

Maybe we can do both simultaneously ...

A model is a simplified, abstract representation or description of something

- We keep what is essential, relevant, ... (which is application dependent)
- We leave outside what is irrelevant or contingent (e.g. we may not care if a block above has a stain)

Want a computational representation that we can process and use

Computers are good at storing and processing symbolic representations

Why not starting with a **relational database**?

Need to identify **relevant high-level concepts/entities, properties, relationships**

- A two-argument predicate (or property): *OnTopOf*(\cdot, \cdot)
- A one-argument predicate: *Block*(\cdot)
- A two-argument predicate: *ColorOf*(\cdot, \cdot)
- A two-argument predicate: *LeftOf*(\cdot, \cdot)
- And we use symbolic names for the individuals outside; let's keep the same (not mandatory): *Y, ..., floor, ...*

These ingredients form the **schema of the database** (no data so far ...)

Now the **instance of the database**, in terms of **material relations**:

OnTopOf		
	Y	G
	G	R
	R	floor
	etc.	

ColorOf		
	Y	yellow
	G	green
	R	red
	etc.	

Block	
	Y
	G
	R
	B
	P

LeftOf		
	R	P

We can pose queries to the DB, e.g.

- What is on top of what?
- What is on top of something that is a block?

(We need a language to express these queries, and can be processed and answered by the DB)

We might create other relations, with material data

Sometimes it is better to **define new relationships** as needed, e.g. for a particular session with the DB

Virtual relations ...

For this we also need an appropriate language

1. Define the one-argument predicate (or property) to be satisfied by the objects that are on top of something or have something on top and that are not blocks (the floor should qualify)

Introduce a new predicate $NoBlock(\cdot)$, no data for it, but instead its definition:

$$\forall x(NoBlock(x) \leftrightarrow (\exists y(OnTopOf(x, y) \vee OnTopOf(y, x)) \wedge \neg Block(x)))$$

- A symbolic definition that uses the basic elements of the schema PLUS logical symbols
- It is a definition of the kind we find in math and science in general: the new thing (LHS) is being defined via an “iff” (possibly implicit) in terms of things we already know about (RHS)

E.g. math:

A function between two vector spaces is **linear** iff Etc.

A matrix is **invertible** iff Etc.

- The underlined part says that object x appears in table *OnTopOf*, which bounds the extension for the complement (of *Block*)
- This is a formula written in the language of **relational calculus**, which is a particular case of **predicate logic**
- Relational DBs can handle this definition
- In DB parlance, we are defining **a view**, a virtual relation that can be used in queries, etc.
- A relational DB can handle this definition; and a system like Prover9 (or Otter) can reason with it

2. As a possible alternative to the item right above, define now:

$$\forall x(NoBlock(x) \leftrightarrow \neg Block(x))$$

- In principle fine, now even the colors, e.g. *yellow*, will satisfy this predicate, something that may not be intended
- Now the complement *Block* is unbounded: whatever is outside the table *Block* qualifies
- Relational databases cannot handle “absolute complements” like this (of a relation in this case)

Relational DBs handle only “relative complements”, i.e. set-differences of tables (material or virtual)

3. Define a new relation $Above(\cdot, \cdot)$:

What is in $OnTopOf$ should make it into it ...

$$Above(x, y) \leftarrow OnTopOf(x, y) \quad (*)$$

With this Y is above G , ... but there should be more, e.g. Y above $floor$

$$Above(x, y) \leftarrow OnTopOf(x, z) \wedge OnTopOf(z, y)$$

z is a join variable capturing the transitive relationship ...

Not very elegant ... and not general enough: we might stack new blocks on top and this definition would fall short ...

Better: $Above(x, y) \leftarrow Above(x, z) \wedge OnTopOf(z, y) \quad (**)$

- A **recursive definition**! Given by (*) and (**), the former being the base case
- With value R as value for variable z we get Y above the $floor$
Exercise: Obtain that Y is above the $floor$

3. Define a new relation $Above(\cdot, \cdot)$:

What is in $OnTopOf$ should make it into it ...

$$Above(x, y) \leftarrow OnTopOf(x, y) \quad (*)$$

With this Y is above G , ... but there should be more, e.g. Y above *floor*

$$Above(x, y) \leftarrow OnTopOf(x, z) \wedge OnTopOf(z, y)$$

z is a join variable capturing the transitive relationship ...

Not very elegant ... and not general enough: we might stack new blocks on top and this definition would fall short ...

Better: $Above(x, y) \leftarrow Above(x, z) \wedge OnTopOf(z, y) \quad (**)$

- A **recursive definition**! Given by (*) and (**), the former being the base case

In the definition the predicate on the LHS calls itself (on a simpler case); it is defined in terms of itself

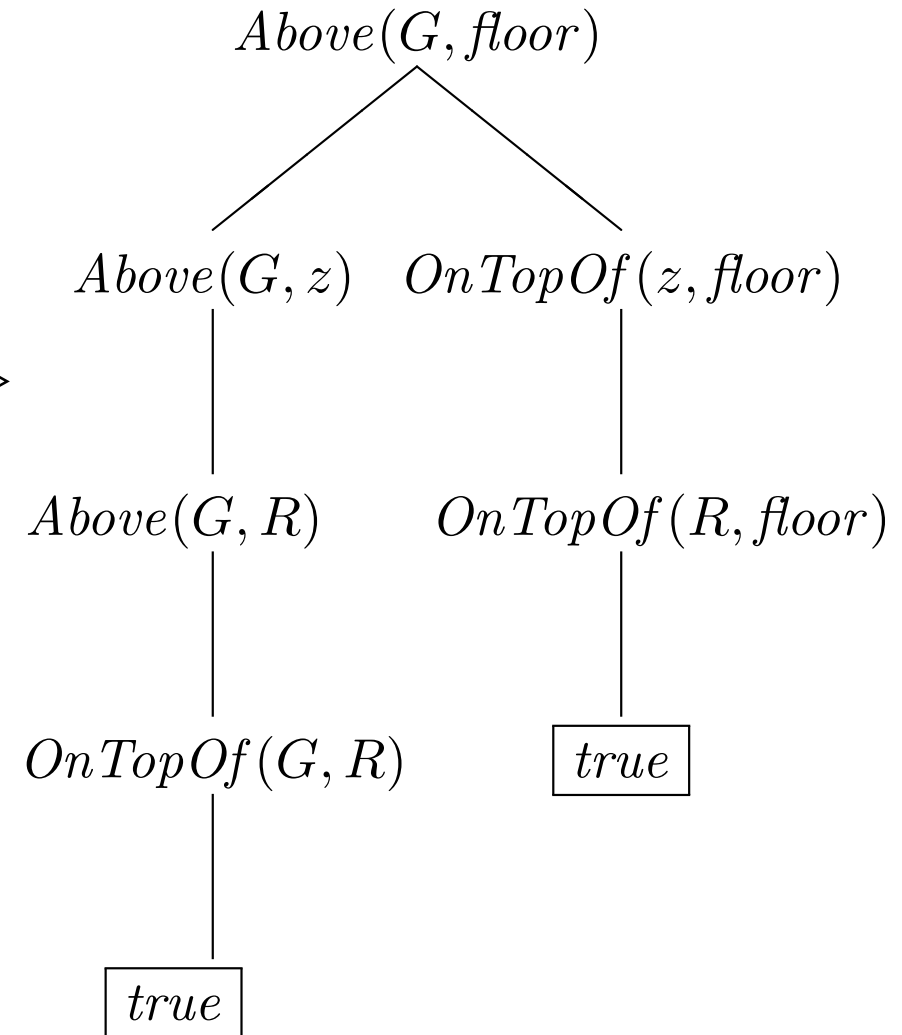
- With value R as value for variable z we get Y above the *floor*
- Exercise: Obtain that Y is above the *floor*
- *Above* becomes the **transitive closure** of relation *OnTopOf*¹
- The definition above is not in predicate logic (or relational calculus)
- It still can be handled by relational DBs (a recent addition)
- The language above is **Datalog**
- Systems like Prolog, DLV, etc. can (correctly) reason with a definition like this, but not Prover9 (which is designed for classical reasoning in predicate logic)
- (It can be proved that classical predicate logic cannot specify the transitive closure of a binary relation;² it does not provide recursion or anything similarly expressive)

¹The TC of binary relation R is the minimal (wrt. set containment) transitive relation T than contains R

²First proved in the context of relational DBs. Cf. A. V. Aho and J. D. Ullman. "Universality of Data Retrieval Languages". In Proc. ACM Symp. on Principles of Programming Languages, pp. 110-117, 1979

A computation that $Above(G, floor)$ is true using backward reasoning with the definition of $Above$:³ We start from the root of the proof-tree

Here ----->
we (successfully) tried
the value R for z
(try other values)



³Prolog reasons like this

Exercise: Since the table *LeftOf* above is incomplete, add Datalog rules to define its extended version, capturing in general the relationship “being left of”.

E.g. Y should end up being to the left of P (or of any potential block to the right of P)

Do some computation as above

4. We can define “objects” as those appearing in *OnTopOf*:

$$\forall x(Obj(x) \leftrightarrow \exists y(OnTopOf(x, y) \vee OnTopOf(y, x)))$$

In Datalog or Prolog the same definition would be with two rules:

$$Obj(x) \leftarrow OnTopOf(x, y)$$

$$Obj(x) \leftarrow OnTopOf(y, x)$$

Adding **a rule about objects that are movable by default?**

$$\forall x(Obj(x) \wedge \neg Abnormal(x) \rightarrow Movable(x)) \quad (***)$$

“Normally objects are movable”

A sort of commonsense piece of knowledge, as in *“normally birds fly”*

And we can add **a table explicitly listing objects that are abnormal:**

<i>Abnormal</i>	
	floor P

(maybe P it is abnormally glued to the floor)

Is block G movable?

Not if we use “classical logical reasoning”: no “classical” evidence that G is not abnormal ...

No way to obtain -classically- that $Abnormal(G)$ is false (and then $\neg Abnormal(G)$ true)

Appealing to “commonsense reasoning” we can get what we want:

- There is **no positive evidence** that G is abnormal
- Assuming that it is not abnormal does not contradict anything else
- So, **under the commonsense assumption that G is not abnormal**, we obtain from (***) that it is movable

Very much in the spirit of concluding that “*Tweety flies*” from the default rule saying that “*Normally birds fly*” and “*Tweety is a bird*”

- Rule (***) could be transformed into a view definition in relational DBs

As it is, it is expressed in predicate logic

- But the use we made of it does not belong to relational calculus or predicate logic
- The negation in it was interpreted and used in a non-classical way
- To make the distinction, rule (***) would normally be written as
$$\forall x(Obj(x) \wedge not Abnormal(x) \rightarrow Movable(x))$$
- With this “commonsense” interpretation, it cannot be handled by a system like Prover9, which does only classical reasoning
- But it can be used as intended with Prolog or DLV



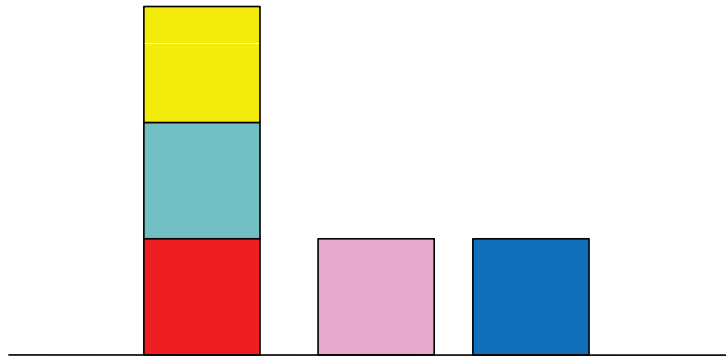
Chapter 1: Models and Representation

(Introduction)

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

A Simple Blocks World



An external reality

5 colored blocks and the floor

How to represent or model it?

- A model is a simplified, abstract description/representation of a reality, phenomenon, ...

We concentrate on what is relevant (for us), leaving aside what is irrelevant or contingent

- What is relevant here?

There are certain objects, some on top of others ...

- **How to model this?**

Several alternatives ...¹

¹On representation and theories: Patrick Suppes. "Representation and Invariance of Scientific Structures". CSLI Publications, 2002

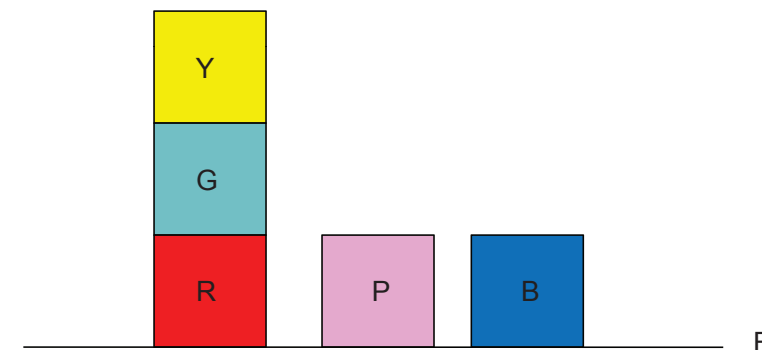
A. Set-Theoretic Structures

- We are used to do this in Math ...
- We use a **set-theoretic structure**, with a **universe U** and a **binary relation On**
- Easier if we give names to the objects (or their math representations), as we always do in Math ...

In set-theoretic terms ...

$$U = \left\{ \text{yellow square}, \text{teal square}, \text{red square}, \text{pink square}, \text{blue square}, \text{---} \right\}$$

$$On = \left\{ (\text{teal square}, \text{red square}), (\text{yellow square}, \text{teal square}), \dots, (\text{blue square}, \text{---}) \right\}$$



$$U = \{ Y, G, R, P, B, F \}$$

$$On = \{ (Y, G), \dots, (B, F) \}$$

- We have a structure: $\mathcal{B} = \langle U, O_n \rangle$
 - A non-empty universe or domain U
 - A binary relation O_n on U : $O_n \subseteq U \times U$
- Not very different from other structures we find in Math, e.g.
 - $\mathcal{N} = \langle \mathbb{N}, < \rangle$
 With $\mathbb{N} = \{0, 1, 2, \dots\}$, and $< = \{(0, 1), (0, 2), \dots, (1, 2), \dots\}$
 - $\mathcal{Z} = \langle \mathbb{Z}, +, \times \rangle$
 - Vector spaces, of the form $\mathcal{V} = \langle V, S, +^V, +^S, \times^S, \cdot^{V,S} \rangle$
 With a vector domain, a scalar domain, operations on them, and one on both ...
- Is the structure \mathcal{B} above a good representation?

Mathematically it is fine ...

From the application point of view, it depends ...

More detail?

- What about $\mathcal{B}' = \langle U, On, LeftOf \rangle$?

With $LeftOf = \{(P, B), (R, P), (G, P), \dots\}$, a binary relation

Also fine ...

B. Propositional Languages

- We choose a language to talk about (describe) the outside reality
- The most basic languages provided by mathematical logic are those of “propositional logic”
- Our language \mathcal{L} is built from **propositional variables**

They **denote -are names for- basic statements** (basic sentences) about the described domain

1. $YonG, GonR, RonF, RleftOfP, \dots, BonF$

For: “Yellow block is on green block”, ..., “Red block if left of purple block”, ...
(We could use p, q, r, \dots as propositional variables)

2. Why not these?: $RonG, PleftOfG, \dots$

For: “Red block is on green block”, “Purple block if left of green block”,
... Perfectly fine ...

- Our choice, including the level of granularity

Normally propositional variables denote statements that would not (cannot) be decomposed into sub-statements

Propositional variables become atomic (symbolic) sentences

- We can build more **complex, composite symbolic propositions** using propositional variables and **logical connectives**

$YonG \wedge GonR, YonG \vee BonF, \neg YonG, \neg RonY$

$YonG \wedge RonG, (RleftOfP \wedge PleftOfB) \rightarrow RleftOfB$

- These are just symbolic sentences, **no truth or falsity** status assigned to them ... yet ...

- The outside reality \mathcal{O} determines a **truth valuation** (assignment) $\sigma_{\mathcal{O}}$

It assigns truth values, T, F or $0, 1$, to symbolic sentences

- **It starts** by assigning value 1 to those propositional variables that are (we consider to be) true of (in) \mathcal{O} , and 0 to those that are false

True ones in red and false ones in green below

YonG, GonR, RonF, RleftOfP, BonF, RonG, PleftOfG, ...

$\sigma_{\mathcal{O}} : YonG \mapsto 1, \dots, \sigma_{\mathcal{O}} : RonG \mapsto 0$

- **It continues** by assigning truth values to complex symbolic sentences (still under $\sigma_{\mathcal{O}}$)

As usual according to the truth table of logical connectives:

YonG \wedge GonR, YonG \vee BonF, \neg YonG, \neg RonY

YonG \wedge RonG, (RleftOfP \wedge PleftOfB) \rightarrow RleftOfB

- Complex formulas take truth values under $\sigma_{\mathcal{O}}$ on the basis of the truth values for propositional variables and the truth tables of the logical connectives: For φ, ψ arbitrary formulas,

φ	ψ	$\neg\varphi$	$(\varphi \wedge \psi)$	$(\varphi \vee \psi)$	$(\varphi \rightarrow \psi)$	$(\varphi \leftrightarrow \psi)$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

This general table assigns the semantics (meaning) to the logical connectives and truth values to composite formulas

Exercise: Verify using this table that the formula $(RleftOfP \wedge PleftOfB) \rightarrow RleftOfB$ takes the value 1 under $\sigma_{\mathcal{O}}$

Similarly, check that $(PleftOfR \wedge PleftOfB) \rightarrow RleftOfB$ takes value 1

(the last two rows in the table are the relevant ones to consider in this case)

- So, what then is the representation of \mathcal{O} above? Alternatives:
 - Represented by the propositional knowledge base that contains *all* the symbolic sentences that are true in it (i.e. take value 1 under $\sigma_{\mathcal{O}}$)
 May be too large and difficult to build
 - All the propositional variables that are true in \mathcal{O} (or, equivalently, that are made true by $\sigma_{\mathcal{O}}$) Idem ...
 - We keep *some* propositional variables that are true plus *some* general, composite formulas
 Other formulas that are entailed (implied) by these become *implicitly* part of the representation
 The most common and practical approach
 Requires storage of propositional variables and reasoning

In all cases, a representation based on symbolic logic ... propositional logic

C. Predicate-Logic Languages

- Again, we use a symbolic language, more expressive than the propositional above
- Very much like choosing a relational database schema
- We have come up with some predicates, and names for (some) individuals or objects in the domain

Our choice depending on what we want to express or capture

- Introduce names for domain individuals: $\#y, \#r, \#g, \#p, \#f$

For: yellow block, ..., floor

No name for the blue box (our choice)

- Predicates: $OnTop(\cdot, \cdot), LOf(\cdot, \cdot), Block(\cdot)$

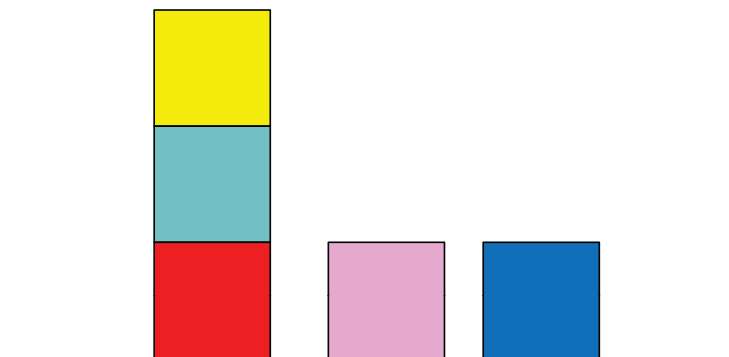
Binary, binary, unary, resp., i.e. 2-argument, and 1-argument

- We can start “saying” things, symbolically:

$OnTop(\#g, \#r), OnTop(\#y, \#g),$

$OnTop(\#r, \#f), OnTop(\#p, \#f),$

$LOf(\#r, \#p), Block(\#y), \dots$



These become “atomic” symbolic sentences: they cannot be decomposed any further (into more basic symbolic statements)

- We do not have to use the names for individuals we had “outside” the language, nor the relation names (but we could)
- It is usual (but not mandatory) to include the binary predicate symbol of equality: =

We could say, e.g. $\#y = \#y$

- We can bring into this picture the “logical elements” of the language, and produce **more complex, composite symbolic sentences**:

$$\neg \#y = \#r \quad (\text{abbreviated: } \#y \neq \#r)$$

$$\neg \text{OnTop}(\#r, \#g), \quad \text{OnTop}(\#y, \#g) \vee \text{OnTop}(\#r, \#g)$$

$$\exists x(\text{Block}(x) \wedge \text{OnTop}(x, \#f) \wedge x \neq \#y \wedge x \neq \#g \wedge x \neq \#r \wedge x \neq \#p \wedge x \neq \#f)$$

$$\forall z(\neg \text{OnTop}(z, z))$$

$$\forall x \forall y \forall z(\text{OnTop}(x, y) \wedge \text{LOf}(y, z) \rightarrow \text{LOf}(x, z))$$

$$\forall x \forall y \forall z(\text{LOf}(x, y) \wedge \text{LOf}(y, z) \rightarrow \text{LOf}(x, z))$$

- These sentences are all “true” in relation to the outside reality \mathcal{O}
- This is a more expressive language: the last four sentences above cannot be expressed in propositional logic

- We can represent \mathcal{O} through the set of symbolic sentences that are true “in it”

Or a knowledge base that is just a subset of true sentences: the others might be logically entailed ...

- Since we have a symbolic language that is independent from \mathcal{O} , we can make symbolic statements that are false in \mathcal{O} :

$OnTop(\#r, \#g)$, $\exists z OnTop(z, \#y)$, etc.

They would not be included in the representation of \mathcal{O}

- We have to be more careful about the notion of “truth”

We are assuming that the outside reality \mathcal{O} is a reliable **interpretation** (domain) for the language

Actually, we will not use an outside reality like \mathcal{O} to interpret the language, but a **structure** that represents \mathcal{O} , as in A., and is compatible with the symbolic language

- Very much as the relationship between a relational database schema (and its associated symbolic languages, such as SQL, relational calculus/algebra) and the database (instance)

The DB instance acts as an interpretation structure for the language

The structure in A. could be represented as a relational DB instance

- More precisely, we use the **interpretation structure**:

$$\bar{\mathcal{B}} = \left\langle \underbrace{\{\square, \square, \square, \square, \square, -\}}_U, \underbrace{\{(\square, \square), (\square, \square), (\square, -), (\square, -), (\square, -)\}}_{On}, \right.$$

$$\underbrace{\{(\square, \square), (\square, \square), (\square, \square), (\square, \square), (\square, \square), (\square, \square), (\square, \square)\}}_{LeftOf},$$

$$\underbrace{\{\square, \square, \square, \square, \square\}}_{Blocks}, \underbrace{\{\square, \square, \square, \square, -\}}_{\text{interpretations for symbolic names (not blue block)}} \rangle$$

Or:

$$\bar{B} = \langle \{Y, G, R, B, P, F\}, \{(Y, G), \dots\}, \{(P, B), \dots\}, \{Y, G, R, B, P\}, Y, G, R, P, F \rangle$$

- Notice the blue block belongs to the domain, but there is not name for it in the symbolic language (this is fine)
- Notice the “correspondence” between the non-logical elements of the language and the components of the set-theoretic structure (that represents \mathcal{O} and makes it possible to interpret the language)

$$\mathcal{S} = \{ \textit{OnTop}, \textit{LOf}, \textit{Block}, \#y, \#r, \#g, \#p, \#f \}$$

$$\bar{B} = \langle U, \textit{On}, \textit{LeftOf}, \textit{Blocks}, Y, R, G, P, F \rangle$$

- As here above, predicate names in the symbolic language may not be the same relation names in the structure, but they are in correspondence

This correspondence will allow us to “define” truth of logical formulas in (compatible) interpretation structures



**Chapter 2: Automated Reasoning, Theorem Proving,
Mathematical Theories and Mathematical Logic
(Introduction)**

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

Can a Computer Do the Same?

Consider the following **argument**:

If Superman were able and willing to prevent evil, he would do so. If Superman were unable to prevent evil, he would be impotent; if he were unwilling to prevent evil, he would be malevolent. Superman does not prevent evil. If Superman exists, he is neither impotent nor malevolent. Therefore, Superman does not exist.

Can we prove that this argument is correct?

Or that “*Superman does not exist*” follows from the premises?

Actually, we can *mathematically or logically prove* the validity of this argument

Could a computer do the same?

Or do more complex proofs?

Killing Superman

If Superman were able and willing to prevent evil, he would do so. If Superman were unable to prevent evil, he would be impotent; if he were unwilling to prevent evil, he would be malevolent. Superman does not prevent evil. If Superman exists, he is neither impotent nor malevolent. Therefore, Superman does not exist.

- The validity of this argument has to do with the “logical structure” of the information involved
- To make this structure more clear, we get rid of contingent details and introduce names to denote “basic propositions” in this argument
- The following “propositional variables” denote the “atomic sentences or statements”

a: Superman is able to prevent evil

m: Superman is malevolent

w: Superman is willing to prevent evil

p: Superman prevents evil

i: Superman is impotent

e: Superman exists

- The main propositions in the argument can be represented using these basic propositions and the “logical connectives”:

$$\varphi_0: (a \wedge w) \rightarrow p$$

$$\varphi_1: (\neg a \rightarrow i) \wedge (\neg w \rightarrow m)$$

$$\varphi_2: \neg p$$

$$\varphi_3: e \rightarrow (\neg i \wedge \neg m)$$

- The whole argument can be represented by the formula:

$$\psi: (\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \varphi_3) \rightarrow \neg e$$

- Proving that this is a valid argument amounts to proving that it is always true, no matter if the atomic sentences are true or false
- So, we could try all the combinations of “truth values” for the atomic sentences

It is possible to check that no matter what truth value a, w, i, m, p, e take, ψ always takes the truth value 1

A bit cumbersome ...? Too many combinations to check ...?¹ (2^6)

- What about reasoning as we do when we do mathematical proofs?
- For example, try to “derive” $\neg e$ by backward reasoning, going back to the hypothesis ...

¹Checking if a formula is always true by checking its truth table is an exponential time algorithm in the number of propositional variables in the formula (or the length of the formula). It is not known if there is a polynomial time algorithm to solve this problem.

Contrapositive of φ_3 : $\neg(\neg i \wedge \neg m) \rightarrow \neg e$

Try to prove: $\neg(\neg i \wedge \neg m)$, i.e. $i \vee m$ (*)

Now from φ_1 we get: $\neg a \rightarrow i$

Then, instead of proving (*), it is good enough to prove $\neg a \vee m$
($\neg a$ replaced for i)

Again from φ_1 we get: $\neg w \rightarrow m$

Then, it is good enough to prove $\neg a \vee \neg w$ (same idea), i.e. $\neg(a \wedge w)$

From φ_0 we get $\neg p \rightarrow \neg(a \wedge w)$ (contrapositive)

We need to prove: $\neg p$

But it is given directly by φ_2 We are done!

- Any other way to do this proof?

- What about a “proof by contradiction”?
- We assume that what we want to prove is not true, then false, i.e. its negation is true

We add $\neg\neg e$, i.e. e to our set of hypothesis and we try to derive “a contradiction”, i.e. a sentence that is always false

From φ_3 we obtain: $\neg i \wedge \neg m$, i.e. $\neg i$ and $\neg m$ (**)

Now from φ_1 we obtain both $\neg a \rightarrow i$ and $\neg w \rightarrow m$, whose “contrapositives” are $\neg i \rightarrow a$ and $\neg m \rightarrow w$, resp.

Combining them with those in (**), we obtain: a and w , i.e. $a \wedge w$

From φ_0 we obtain: p

This combined with φ_2 gives the contradiction: $p \wedge \neg p$ **Done!**

- What about having a computer doing this proof?
- Let us try with the Otter theorem prover ...

Input:

```
set(binary_res).  
  
formula_list(usable).  
(a & w) -> p.  
(-a -> i) & (-w -> m).  
-p.  
e -> (-i & -m).  
end_of_list.  
  
formula_list(sos).  
e.  
end_of_list.
```

Run with: C:\> otter < superman.in > sup.out

Output: (extract)

----- Otter 3.3, August 2003 -----

The process was started by a Windows user on a Windows machine,
Wed Aug 22 16:09:15 2007 The command was "otter".

```
set(binary_res).
  dependent: set(factor).
  dependent: set(unit_deletion).
```

```
formula_list(usable).
a&w->p.      (-a->i)& (-w->m).      -p.      e-> -i& -m.
end_of_list.
```

-----> usable clasifies to:

```
list(usable).
1 [] -a|-w|p.
2 [] a|i.
3 [] w|m.
4 [] -p.
5 [] -e| -i.
6 [] -e| -m.
end_of_list.
```

```
formula_list(sos).  
e.  
end_of_list.
```

```
-----> sos clasifies to:
```

```
list(sos).  
7 [] e.  
end_of_list.
```

```
===== end of input processing =====
```

===== start of search =====

given clause #1: (wt=1) 7 [] e. ** KEPT (pick-wt=1): 8
[binary,7.1,6.1] -m. ** KEPT (pick-wt=1): 9 [binary,7.1,5.1]
-i. 8 back subsumes 6. 9 back subsumes 5.

given clause #2: (wt=1) 8 [binary,7.1,6.1] -m. ** KEPT
(pick-wt=1): 10 [binary,8.1,3.2] w. 10 back subsumes 3.

given clause #3: (wt=1) 9 [binary,7.1,5.1] -i. ** KEPT
(pick-wt=1): 11 [binary,9.1,2.2] a. 11 back subsumes 2.

given clause #4: (wt=1) 10 [binary,8.1,3.2] t. ** KEPT
(pick-wt=0): 12 [binary,10.1,1.2,unit_del,11,4] \$F.

-----> EMPTY CLAUSE at 0.00 sec ----->
12 [binary,10.1,1.2,unit_del,11,4] \$F.

Length of proof is 4. Level of proof is 2.

The clean proof is provided:²

```
----- PROOF -----  
  
1 [] -a| -w|p.  
2 [] a|i.  
3 [] w|m.  
4 [] -p.  
5 [] -e| -i.  
6 [] -e| -m.  
7 [] e.  
8 [binary,7.1,6.1] -m.  
9 [binary,7.1,5.1] -i.  
10 [binary,8.1,3.2] w.  
11 [binary,9.1,2.2] a.  
12 [binary,10.1,1.2,unit_del,11,4] $F.  
  
----- end of proof -----
```

²Sometimes the proofs themselves, apart from knowing that they exist, can be relevant: useful information can be extracted from a proof.

Search stopped by max_proofs option.

==== end of search =====

----- statistics -----

clauses given 4 clauses generated 5

binary_res generated 5

----- times (seconds) -----

user CPU time 0.20 (0 hr, 0 min, 0 sec)

That finishes the proof of the theorem.

Process 0 finished Wed Aug 22 16:09:15 2007

Otter reasons by contradiction, targeting the particular contradictory formula $\$F$.

The .in and .out files for proofs with Otter can be found all at:

<http://www.scs.carleton.ca/~bertossi/complog/progfiles>

Remarks:

- The contradiction “ $\exists F .$ ”, that Otter reaches does not depend on the vocabulary of the input specification
- Compare with the contradiction $p \wedge \neg p$ obtained in slide 7, which does depend on the specification)
- Having a fixed, universal contradiction allows for the design of heuristics and mechanisms that always target the same formula (as opposed to a moving target)

Exercise: Redo the Superman example with Prover9, Otter's successor (the input file is a bit different)



Good News!

- A mathematical problem had been open, without answer, since the 1930s
- The conjecture (believed, but not proven):

Every Robbins Algebra is a Boolean Algebra

- There was neither a proof nor a refutation, despite efforts of many good mathematicians
- In December 1996, Robbins' conjecture was proved by means of a computer program

The news appeared in the New York Times, and was rapidly broadcasted on internet

Times

With Major Math Proof, Brute Computers Show Flash of Reasoning Power

The achievement would have been called creative if a human had done it.

By GINA KOLATA

Computers are whizzes when it comes to the grunt work of mathematics. But for creative and elegant solutions to hard mathematical problems, nothing has been able to beat the human mind. That is, perhaps, until now.

A computer program written by researchers at Argonne National Laboratory in Illinois has come up with a major mathematical proof that would have been called creative if a human had thought of it. In doing so, the computer has, for the first time, got a foothold into pure mathematics, a field described by its practitioners as more of an art form than a science. And the implications, some say, are profound, showing just how powerful computers can be at reasoning itself, at mimicking the flashes of logical insight or even genius that have characterized the best human minds.

Computers have found proofs of mathematical conjectures before, of course, but those conjectures were easy to prove. The difference this time is that the computer has solved a conjecture that stumped some of the best mathematicians for 60 years. And it did so with a program that was designed to reason, not to solve a specific problem. In that sense, the program is very different from chess-playing computer programs, for example, which are intended to solve just one problem: the moves of a chess game.

"It's a sign of power, of reasoning power," said Dr. Larry Wos, the supervisor of the computer reasoning project at Argonne. And with this result, obtained by a

colleague, Dr. William McCune, he said, "We've taken a quantum leap forward."

Dr. Wos predicts that the result may mark the beginning of the end for mathematics research as it is now practiced, eventually freeing mathematicians to focus on discovering new conjectures, and leaving the proof to computers.

But the result also may challenge the very notion of creative thinking, raising the possibility that computers could take a parallel path to reach the same conclusions as great human thinkers. Or it may be that since no one has any idea how humans think, the magnificent bursts of creativity that spring apparently full-blown from the minds of geniuses are actually a result of hidden, computer-like drudge work in the unconscious recesses of the brain.

Dr. Stanley Burris, a mathematician at the University of Waterloo in Canada, said that the result was "the first sort of real breakthrough in automated theorem proving," and that it did seem to be different in kind from what went before. It shows, he said, that "it's a very thin line between the mechanical and the creative and it may disappear."

Dr. Robert Boyer, a computer scientist at the University of Texas in Austin, hedged. "I think it's the most remarkable result in automated theorem proving in 30 years," he said, and "clearly a form of computer thinking." But, he added, "I don't want to make too much of that." It's best, he said, to think of a computer as "just another colleague, one that is sometimes helpful, but often not."

Dr. McCune's proof concerns a conjecture that is the very epitome of pure mathematics. "It has no applications," Dr. McCune said. His computer program proved that a set of three equations is equivalent to a Boolean algebra, that set of rules, familiar to generations of high school students, that govern unions and complements and intersections among

Continued on Page B10

t for 2
idea for
auts, a
n cycle,
s gravity and
ms on the
o provide
strengthen
es and bones.

ace Cycle
tion by Istvan Banyai

nding nicely to
uestion. But all
dicine business
emics — fairly
a countermea-
it will work for

B12

- For the proof of the *Robbins' Conjecture* an “automated reasoner” was used

Essentially, OTTER (Argonne National Laboratory)

- The conjecture -now Theorem- was proven using a **general purpose computational program, implemented to prove general theorems**

Not designed to prove a theorem in particular

That is, like humans beings that use general reasoning capabilities to prove theorems

- We could attribute “intelligence”, “creativity”, etc. to this program; features usually attributed to human beings

- The mathematics involved in the proof of this theorem was done by the computer

Not by a human being supported by the brute force of a computer

(As happened before with the “4-color conjecture”)

(C.f. Larry Wos. “Programs That Offer Fast, Flawless, Logical Reasoning”.

Communications of the ACM, 1998, 41(6):87-95.)

[The conjecture that any map can be painted with 4 colors was open for maaaany years, until the early 70s, when it was proved to be true.

It was done by reducing the proof to checking a few thousand cases, which was successfully done by a computer.

Some maps can be colored with 3 colors, others cannot. It is not known if this property can be checked by means of a general algorithm that runs in polynomial time in the size of the map.]

- What Was Done?

The prover established, in essence, that:

The **Equations for Robbins' Algebras**:

$$x \vee y = y \vee x$$

$$(x \vee y) \vee z = x \vee (y \vee z)$$

$$\neg(\neg(x \vee y) \vee \neg(x \vee \neg y)) = x$$

Imply the **Equations for Boolean Algebras**:

$$x \vee y = y \vee x$$

$$(x \vee y) \vee z = x \vee (y \vee z)$$

$$\neg(\neg x \vee y) \vee \neg(\neg x \vee \neg y) = x$$

- That is, the prover showed to be capable of: performing **equational reasoning**, and doing proofs from mathematical theories

Reasoning in/from (Mathematical) Theories

- Scientists usually come up with “theories” about specific areas of their domain of study
- In general, after a certain phenomenon has been studied, usually including:
 - intuition,
 - reasoning,
 - observation, and
 - experimentation,a scientist proposes some **basic principles or laws** that explain the phenomenon
- Other “true” facts/laws about the subject should be consequences (or be explained) by those -hopefully few- basic principles

- So, the theory consists of the basic principles, aka. *axioms*, and the propositions that are logical consequences of those axioms
- Analyzing the theory by concentrating on a few principles, that can be tested and locally investigated, has advantages:
 - simplicity and economy in understanding and representation
 - better organization and structure of knowledge
 - makes easier analysis of:
 - consistency: is the set of principles free of contradictions?
 - redundancy: is any of the principles already implied by the others?
 - independence: is one particular principle not implied by the others, nor its negation?

Example: Geometry: First “axiomatization” of geometry by Euclid around 300 B.C.

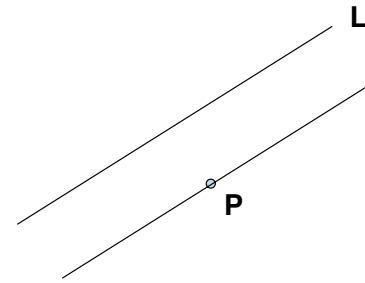
In “The Elements”, Euclid gave five postulates (axioms) for geometry, from which the other true geometrical statements (the theorems of geometry) should follow

Until the 19th century mathematicians were struggling to determine if Euclid’s 5th postulate was independent from the other first four

People thought it was less fundamental than the others and could be proved from them

The 5th postulate turned out to be independent!

Given a line L and a point P outside L ,
exactly one line can be drawn through P
that is parallel to L



How can one prove an independence result like this?

What does it mean?

If 1. - 5. are the five postulates, that the last one is independent from the others means that:

- 1. - 4. $\not\Rightarrow$ 5.
- 1. - 4. $\not\Rightarrow$ \neg 5.

To establish that, e.g. the 5th postulate is not a consequence of the first four postulates, one has to create a **mathematical structure** (world, universe) where the first four postulates are true, but not the fifth

Similarly for the second non-implication

That is, two “counterexamples” have to be exhibited:

- one where 1. -4. are true, but not 5.
- another where 1. - 4. are true, and 5. is true

The latter shows that 5. is **consistent with** 1. - 4.

All this was established long ago ... (\sim 1830)

So, given 1. - 4., we are free to adopt either 5. or its negation (or nothing)

Not adopting 5. means we have a non-Euclidean geometry!

Such a strange universe exhibits a non-Euclidean geometry, where, e.g. infinitely many lines parallel to L can be drawn through P

A pure mathematical development that found applications almost 100 years later, in the context of relativity theory!

- In scientific (mathematical) reasoning, counterexample finding is almost as important as doing proofs

A different, companion activity to inference

- There are automated reasoning systems for counterexample construction!

Counterexample finding has many applications, in particular in computing

[One can automate the verification of properties of software system formal (symbolic) specifications. Automated tools can create counterexamples showing that certain properties do not hold (i.e. do not follow, as expected or intended, from the specification).]

Exercise: An **equivalence relation** on a set can be seen as a structure $\langle A, R \rangle$, where A is a non-empty set, and $R \subseteq A \times A$ is a binary relation with the following **properties**:

1. For every $a \in A$: $(a, a) \in R$ (reflexivity)
2. For every $a, b \in A$: $(a, b) \in R \Rightarrow (b, a) \in R$ (symmetry)
3. For every $a, b, c \in A$: $(a, b) \in R$ and $(b, c) \in R \Rightarrow (a, c) \in R$ (transitivity)

(a) Show a concrete finite equivalence relation on a finite set

(b) Prove from 1. - 3. the following theorem of the theory of equivalence relations:

“For every $a, b, c \in A$: $(a, c) \in R$ and $(b, c) \in R \Rightarrow (a, b) \in R$ ”

(c) Show by means of a finite example (structure) that: 1. & 2. $\not\Rightarrow$ 3.

That is, a counterexample that proves the **independence** of 3. from 1. and 2.

(d) Show by means of a finite example (structure) that: 1. & 2. $\not\Rightarrow \neg 3$.

That is, a counterexample that proves the **consistency** of 3. with 1. and 2.

($\neg 3$. means: “There are $a, b, c \in A$ with: $(a, b) \in R$ and $(b, c) \in R$ and $(a, c) \notin R$ ”)

- Otter and its successor Prover9 have as a relative a (finite) counterexample builder: Mace4
- Anyway, there are now many theories around, in science and mathematics

Example: Physical theories: “relativity theory”, “quantum theory”, “string theory”, ...

Example: More recent mathematical theories: number theory, vector spaces, set theory, group theory, boolean algebras, ...

- Now we will briefly visit two mathematical theories

We are not interested in them *per se* (in this course)

We just want to use them for raising some logical and computational issues ...

Example: The Theory of Boolean Algebras

A Boolean algebra (BA) is a (set-theoretic) structure $\langle B, (\cdot)'\!, \sqcup, \sqcap, \hat{0}, \hat{1} \rangle$, where

- B is a non empty set (its domain or universe)
- $(\cdot)'\! : B \longrightarrow B$ (a unary operation, the complement)
- $\sqcap, \sqcup : B \times B \longrightarrow B$ (two binary operations)
- $\hat{0}, \hat{1} \in B$ (two distinguished elements),

with the following properties:

BA1 Commutativity For all $a, b \in B$:

$$a \sqcup b = b \sqcup a \quad a \sqcap b = b \sqcap a$$

BA2 Associativity

$$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$$

$$(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$$

BA3 Distributivity

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$$
$$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

BA4 Neutral elements

$$a \sqcup \hat{0} = a \quad a \sqcap \hat{1} = a$$

BA5 “Complement”

$$a \sqcup a' = \hat{1} \quad a \sqcap a' = \hat{0}$$

The theory of BAs is a set of propositions: the axioms above plus the **theorems**, i.e. the propositions that can be mathematically proved from the axioms

Above, we use complete abstract symbols above, e.g. \sqcap, \sqcup, \dots to emphasize the generality and abstract nature of the axioms

However, they are reminiscent of the usual symbols \cap, \cup, \dots or \wedge, \vee, \dots that we find in concrete boolean algebras, like those formed by sets, by propositional formulas, by digital circuits, etc.

Actually, the axioms above could be better understood at the light of a very common, particular, and concrete Boolean algebra, namely the structure $\langle P(U), (\cdot)^c, \cup, \cap, \emptyset, U \rangle$, where U is a set, $P(U)$ is the power set of U , and $(\cdot)^c$ is “complement taking”

Every theorem of the theory of Boolean algebras has to be derived from the axioms above alone

It is not enough to check its truth in concrete, specific Boolean algebras

Theorem 1: (Idempotency) For all a : $a \sqcup a = a$

Remarks:

- Theorem says that in every BA and for every element a in its domain the property holds
- But it has to be *proved from the axioms only*, to be sure that it holds in every possible BA
- It is not enough (in general) to prove that it holds in a particular BA

Proof:³

$$\begin{aligned} a \sqcup \hat{0} &= a && \text{By BA4} \\ a \sqcup (a \sqcap a') &= a && \text{By BA5} \\ (a \sqcup a) \sqcap (a \sqcup a') &= a && \text{By BA3} \\ (a \sqcup a) \sqcap \hat{1} &= a && \text{By BA5} \\ (a \sqcup a) &= a && \text{By BA4} \end{aligned}$$

³On the RHS we find explanations for the individual steps; they could be used for “proof checking”. This is another important mathematical activity, with huge impact on computing; a task that that can be automated. Otter’s relative “Ivy” is a proof checker, among others. This a “direct proof” that leads directly from the hypothesis (axioms) to the statement being proved. So, not a proof by contradiction.

We used:

- Axioms of BAs
- Logical-mathematical reasoning, in particular reasoning with equality:
 - replacement of equal terms in equal expressions (equations in this case) produces equal expressions
 - transitivity of equality: If $a = b$ and $b = c$, then $a = c$
 - symmetry of equality: If $a = b$, then $b = a$

These “reasoning rules with equality” are not part of the axioms

They live somewhere else ... In our mind? In our “logic”?

“In our logic for mathematical reasoning?”

Does a computer have it?

- We picked an “arbitrary, but generic” a for the proof (not $\hat{0}$ or $\hat{1}$)
To obtain a universal statement
“Logic” at play again? Similar remarks can be made ...

The theorem becomes **true in every** BA (i.e. every mathematical structure that satisfies the BA axioms)

The power of the combination of:

- abstraction: detecting and specifying the axioms, of what is relevant and in common
- mathematical reasoning!!

Example: The Theory of Groups

Examples of groups, i.e. (set-theoretic) structures $\langle G, \circ, e \rangle$ with a set (domain or universe) of individuals G , a binary operation \circ , and a distinguished element e , that satisfy the three axioms, can be found frequently in mathematics, physics, computer science, ...

Group theory has three axioms:

1. For every g : $g \circ e = g$
(e is a neutral element for the binary operation \circ)
2. For every g_1, g_2, g_3 : $g_1 \circ (g_2 \circ g_3) = (g_1 \circ g_2) \circ g_3$
(operation \circ is associative)
3. For every g there is g' such that $g \circ g' = e$
(every element has an inverse element wrt operation \circ)

Typical examples are:

- The structure $\langle \mathbb{Z}, +, 0 \rangle$ of integer numbers with addition
- The structure $\langle \mathbb{Q}^{>0}, \times, 1 \rangle$ of positive rational numbers with product
- The structure $\langle \mathbf{P}_3, \circ, I \rangle$, where \mathbf{P}_3 has as elements all the permutations of three objects, say

$$\begin{pmatrix} abc \\ abc \end{pmatrix}, \begin{pmatrix} abc \\ acb \end{pmatrix}, \begin{pmatrix} abc \\ cab \end{pmatrix}, \begin{pmatrix} abc \\ cba \end{pmatrix}, \begin{pmatrix} abc \\ bac \end{pmatrix}, \begin{pmatrix} abc \\ bca \end{pmatrix}$$

The operation \circ is the composition of permutations, and I is the identity permutation (the first one)

New propositions can be deduced (or proved) from the axioms; they become the “theorems” of group theory, and elements of the theory

They are true in all structures where the three axioms are true, i.e. in all groups

One such theorem is the following:

T: If every element is its own inverse, then the operation is commutative

More symbolically:

If for every g , $g \circ g = e$, then for every g_1, g_2 it holds: $g_1 \circ g_2 = g_2 \circ g_1$

This proposition can be proved from the three axioms (coming ...)

Notice that the commutativity property of groups is independent from the three given axioms

It does not follow from them, and a counterexample to the universal statement that “every group is commutative” is the permutation group⁴

$$\text{E.g. } \begin{pmatrix} abc \\ acb \end{pmatrix} \circ \begin{pmatrix} abc \\ cab \end{pmatrix} \neq \begin{pmatrix} abc \\ cab \end{pmatrix} \circ \begin{pmatrix} abc \\ acb \end{pmatrix}$$

⁴Since T does follow from the axioms of group theory, T is true in every group, in particular in P_3 , which is non-commutative. So, the antecedent of T must be false there (find a counterexample!).

The non-commutativity does not follow either: A (commutative) counterexample is the group $\langle \mathbb{Z}, +, 0 \rangle$

So, given the three axioms, we may want to add the commutativity property as a new axiom, obtaining the “theory of commutative groups”

This theory is bound to be an extension of (larger than) the “theory of groups”, because every commutative group \mathcal{G} is also a group

Then \mathcal{G} satisfies all the properties of the theory of groups, and more, e.g. commutativity

This is general, logical property of mathematical reasoning and axiomatization

Monotonicity: New knowledge (e.g. axiom) added to a mathematical theory can only make the theory grow in terms of new consequences (or theorems); and we never have to retract previous theorems!

What about making the computer do the proof the theorem T ?

Since it does symbolic processing, we have to express the axioms and the theorem (to be) in precise syntactic terms (according to logic and the specific theorem prover at hand)

First in usual logical terms:

1. $\forall x \forall y \forall z (x \circ (y \circ z) = (x \circ y) \circ z)$
2. $\forall x (x \circ e = x)$
3. $\forall x \exists y (x \circ y = e)$

We want to prove: $\forall x (x \circ x = e) \rightarrow \forall y \forall z (y \circ z = z \circ y)$

We use Otter again; this is the input file

```

set(binary_res). set(para_from).

formula_list(usable).
% Properties of operation o(x,y).
% o(x,y) is associative:

all x y z (o(o(x,y),z)=o(x,o(y,z))).

% the neutral element:

all x (o(x,e)=x).

% there are inverse elements:

all x exists y (o(x,y)=e).

end_of_list.

formula_list(sos).
% Proof is by contradiction: Assume the negation of the
% theorem we want to prove.

-((all x (o(x,x)=e) -> (all x y (o(x,y)=o(y,x)))).

end_of_list.

```

That is, the input knowledge base for Otter consists of the “usable” and “sos” lists together

Putting $\neg T$ in “sos” gives to it a special status

Otter transforms the input formulas into new formulas that can be handled by the deductive system

The transformation leaves all the variables **implicitly universally quantified**

There are no existentially quantified variables (neither explicit nor implicit)

```
-----> usable clausifies to:
```

```
list(usable).  
1 [] o(o(x,y),z)=o(x,o(y,z)).  
2 [] o(x,e)=x.  
3 [] o(x,$f1(x))=e.  
end_of_list.
```

```
-----> sos clausifies to:
```

```
list(sos).  
4 [] o(x,x)=e.  
5 [] o($c2,$c1)!=o($c1,$c2).  
end_of_list.  
===== end of input processing =====
```

Otter eliminates existential quantifiers by introducing **fresh, auxiliary, symbolic functions or constants**, as witnesses of the “existing” values

For example the “clausified” formula [3] above comes from the original axiom $\forall x \exists y \circ(x, y) = e$

This formula is saying that y is a function of x , but the function itself is unknown (and possibly different for each group)

So, a symbolic function $f_1(x)$ is introduced, obtaining $(\forall x) \circ(x, f_1(y)) = e$

For an additional example, consider now the formula: $\exists y \forall x \circ(x, y) = e$

In this case, Otter would replace y by a new constant, say c , producing the formula $(\forall x) \circ(x, c) = e$

This is the reason for the constants $\$c1$, $\$c2$ in 4 and 5 above (we will come back to this ...)

Actually formulas [4] and [5] clearly come from:

$$\forall x(x \circ x = e) \rightarrow \forall y \forall z(y \circ z = z \circ y): \quad (*)$$

[4]: The antecedent of the implication is added as a new hypothesis: It is first assumed that every elements is the inverse of itself, and

[5]: The consequent of the implication is negated; accordingly the assumption is made that there are specific, (unknown) elements (counterexamples) for which the commutativity does not hold; say $\$c1,$
 $\$c2$

This is the way we would try to do the proof as well ...

Otter and Prover9 apply a symbolic transformation of formula [4], including the introduction of fresh constants as “witnesses” for existential values

Formulas [4] and [5] are obtained in pure syntactic manner by negating the whole formula (*) (as we always do with consequences to be proved by contradiction):

$$\neg(\forall x(x \circ x = e) \rightarrow \forall y\forall z(y \circ z = z \circ y)) \equiv$$

$$\neg(\neg\forall x(x \circ x = e) \vee \forall y\forall z(y \circ z = z \circ y)) \equiv$$

$$\forall x(x \circ x = e) \wedge \neg\forall y\forall z(y \circ z = z \circ y) \equiv$$

$$\forall x(x \circ x = e) \wedge \exists y\exists z(y \circ z \neq z \circ y)$$

The conjunction can be split into two formulas, namely [4] and [5]

Here, the symbol \equiv denotes logical equivalence of formulas; it is not a logical symbol, not a symbol at the same level of the symbolic axioms, but a “meta-symbol” or a “higher-level” symbol (more on this later)

The pruned proof returned by Otter (approx. 2 secs.):

```
----- PROOF -----
1 [] o(o(x,y),z)=o(x,o(y,z)).
2 [] o(x,e)=x.
3 [] o(x,$f1(x))=e.
4 [] o(x,x)=e.
5 [] o($c2,$c1)!=o($c1,$c2).
11 [para_from,4.1.2,4.1.2] o(x,x)=o(y,y).
12 [para_from,4.1.2,3.1.2] o(x,$f1(x))=o(y,y).
13 [para_from,4.1.2,2.1.1.2] o(x,o(y,y))=x.
19 [para_from,13.1.1,1.1.2] o(o(x,y),y)=x.
23 [para_from,19.1.1,2.1.1] x=o(x,e).
27 [para_from,23.1.2,11.1.1] e=o(x,x).
30 [para_from,27.1.2,19.1.1.1] o(e,x)=x.
31 [para_from,30.1.1,3.1.1] $f1(e)=e.
37 [para_from,31.1.2,3.1.2] o(x,$f1(x))=$f1(e).
39 [para_from,31.1.2,30.1.1.1] o($f1(e),x)=x.
135 [para_from,37.1.2,39.1.1.1] o(o(x,$f1(x)),y)=y.
1108 [para_from,12.1.1,19.1.1.1] o(o(x,x),$f1(y))=y.
4664 [para_from,1108.1.1,135.1.1.1] o(o(x,x),y)=y.
4671 [para_from,4664.1.1,1.1.1] x=o(y,o(y,x)).
4677 [para_from,4671.1.2,19.1.1.1] o(x,o(y,x))=y.
4682 [para_from,4677.1.1,4671.1.2.2] o(x,y)=o(y,x).
4683 [binary,4682.1,5.1] $F.
----- end of proof -----
```

Otter is able to reason with equality through the “paramodulation” deduction rule, mentioned on the LHS

Exercise: Use Prover9 to do the proof in slide 31

Remarks:

- Notice that a proof by Otter is also a symbolic object: a finite list of formulas that ends in $\%F$.

And every formula is an axiom (with empty justification) or a formula obtained from previous formulas in the list (those mentioned in the justification box on the LHS) by means of a deductive step (that we haven't seen yet)

- By inspecting the justifications in a proof, it is possible to detect which of the axioms was actually used in a proof (this can be useful information)
- We could have done this proof ourselves
- It is nice to see a computer (re)doing these proofs

It should be surprising too

- Can automated theorem proving (ATP) do something new?

We saw that there are theorems that have been proved for the first time by an ATP, not by a human being

Conjectures that have been open for decades, such as Robbins's Conjecture

Actually by an ATP that is a close relative of Otter

- ATP has many applications in computer science, beyond proving mathematical theorems:
 - Software and hardware verification (from the formal specifications):
Prove that a system designed according to a formal, symbolic specification has the intended properties
 - System verification, e.g. termination, liveness, ...
 - Program synthesis from specifications
 - Automated planning
One proves that *there is* a goal state of the world, with the desired properties, that can be reached by a sequence of actions (the plan)
The proof itself is more important than the fact that there is a proof: the plan can be extracted from the proof
 - Verification of security and communication protocols
 - Proofs of integrity constraints in databases
Prove from the specification of the dynamics of a database that it will never reach an inconsistent state (the ICs become “invariants” that have to stay true along a run)
 - Artificial Intelligence (AI), in general

- Robotics
- Knowledge representation (KR)
- Agent systems
- Semantic web

Ontologies (essentially knowledge bases written in predicate logic) are used as semantic layers wrapping web sites, repositories, services, ..., describing the available resources

For automated integration and interoperability

- Ontological engineering

What Is a Theorem Prover Doing? How and Why?

- A (general-purpose) automated theorem prover (ATP) is doing mathematical reasoning through symbolic manipulation of symbolic formulas Really?

How do we know it is doing something like what humans do (when proving theorems)?

If yes, how is it possible?

- Mathematical reasoning (by humans) is a form of “classical logical reasoning”

There are also forms of “non-classical reasoning” (by humans), e.g. common-sense reasoning

- How was it possible to implement in computers the capability of doing classical (mathematical) reasoning?
- Through many efforts, among which there are some crucial ones:
 - Understanding (classical) logical (mathematical) reasoning by humans
 - Investigating this activity
 - Proposing theories about this activity
 - Modeling this activity
 - **Mathematically modeling** this activity
As a symbolic formulae manipulation process
 - Presenting and capturing this process as a computational process

- More specifically, many more specific questions had to be posed and answered:
 1. What is a mathematical proof?
At least to recognize one when done by a computer or a human
 2. What are the admissible deductive steps in a mathematical proof?
 3. What is “the logic” behind deductive/mathematical reasoning?
 4. How to express the logic in terms that can be represented and processed in/by a computer?
In symbolic terms, for representation and manipulation
Through a mechanical and deterministic processes
Transforming symbolic representations into symbolic representations
The final proofs are purely symbolic objects (“legal” lists of symbolic formulas)

5. To implement the symbolic proof mechanisms in a computer, how do we come up with:

- Fast and simple deductive steps
- Strategies, heuristics to search for a proof (or generate one)

While choosing and combining formulas

Choosing and applying deductive steps

Generating new formulas along the way

(Humans are also confronted to alternative steps when they do proofs)

One thing is to have the “laws” of the logical game (of symbolic classical/mathematical reasoning),

Another is using them properly ...

6. What mathematical results guarantee that the symbolic process captures “the logics” of mathematical reasoning?

That the symbolic process is *strong and reliable enough*?

- Everything that is proven through the symbolic processes is indeed something that is a mathematical consequence in the usual sense
- Furthermore and hopefully: that every theorem in the usual sense is provable in the symbolic sense
(The symbolic proof may not be known/found, but that is another problem)

These important and fundamental properties can actually be proved, mathematically!

7. Addressing these last two questions (and others) requires the formalization and investigation of **the semantics of propositional and predicate logics**

This has to do with the:

- **Meaning and interpretations** of symbolic languages
- Notion of **truth** of symbolic statements
- Notion of **logical consequence**: A consequence has to be true whenever the hypothesis (premises) are true

A central notion in mathematics and in logic

It characterizes what (logically) follows from what

All this brings us outside the purely symbolic setting ...

- Again, addressing all these questions requires making the human activity of mathematical reasoning an object of mathematical (scientific) inquiry
- Requires converting mathematics (at least its proofs and notion of mathematical consequence) an object of scientific/mathematical investigation
- *Mathematical Logic* emerges as a mathematical discipline whose object of study is the logic of (human) mathematical reasoning

Logical-mathematical reasoning becomes a subject of mathematical investigation

The application of mathematics to the study of itself!

Metamathematics! The mathematics of mathematics ...

At the same time an investigation of the foundations of mathematics

- Mathematical logic appears much earlier than the inception of computers

Earlier than the emergence of mathematical models of computation

Actually, the latter are heavily influenced by questions in mathematical logic

And developed my mathematical logicians ... (Goedel, Turing, Church, Kleene, Post, ...)

More Specific Questions in Mathematical Logic

1. What are “the laws” that govern logical reasoning?

2. When is a proof correct?

For example, if we have that both assertions “ A implies B ” and A are true, can we conclude that B is true?

3. Using the same “logical laws”, we could develop and implement “proof checkers” that verify the correctness or legality of proofs

A mathematical proof given in symbolic terms as on page 44 could be verified by an automated system

This is a different task from producing proofs

Proof verification turns out to be an important problem today in other areas of algorithms and complexity

C.f. Sanjeev Arora: “How NP got a new definition: a survey of probabilistically checkable proofs”.

CoRR cs.CC/0304038 (2003)

<http://arxiv.org/pdf/cs/0304038v1.pdf>

4. Can we use a purely symbolical language to express (mathematical) statements?
5. Can we capture logical reasoning in pure symbolic terms, i.e. as symbolic manipulation of formulas (symbolic statements)?

If yes, this opens for computers the possibility of doing logical reasoning

Not that new ... The power of high-school algebra resides in the symbolic manipulation of formulas

When we start from formulas that are true (in the numeric domain), we obtain correct or true numerical conclusions

Algebra:

$$\begin{array}{r} x + y = 2 \\ -y + 2x = 3 \\ \hline 3x = 5 \end{array}$$

Logic:

$$\begin{array}{r} p \rightarrow q \\ p \\ \hline q \end{array}$$

$$\begin{array}{r} (p \rightarrow (r \vee q)) \\ (p \wedge \neg r) \\ \hline q \end{array}$$

6. What is the **syntax** of those languages?

7. When is a logical formula valid?

More generally, when is a logical argument valid?

“All men are mortal. Socrates is a man. Then, Socrates is mortal.”

8. Can we determine in purely symbolic terms if an argument is valid?

It is the “syntactic structure” of the argument in 3. that makes it valid

We could have used other predicates, e.g. used *“All grad students are students”, etc.*, and the syntactic structure is the same

Syntactically, the argument is of the form

$$(\forall x(A(x) \rightarrow B(x)) \wedge A(s)) \rightarrow B(s)$$

9. When can we say that a statement is a logical consequence of a body of knowledge?

Is *B* a logical consequence of “*A implies B*” and *A*?

10. In a theorem, what is the relationship between the set of hypothesis and the conclusion (the thesis)?

Whenever the hypothesis are all true, the conclusion is also true

11. When is a statement true, false?

True or false where?

We saw before that they are true or false in **mathematical structures**, which represent “worlds”

12. What is the **semantics**, i.e. meaning of those formal statements?

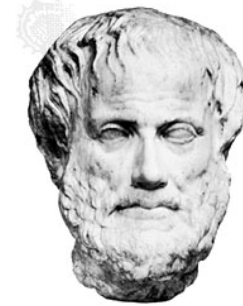
13. What is the relationship between the syntax and semantics of the symbolic languages

This is important: whatever the computer does (syntactically) can be understood and assessed in semantic terms

How we Got Here

- Aristotle (4th century BC)

Identified and studied certain patterns of valid reasoning, the syllogisms



Aristotle, marble bust with a restored nose, Roman copy of a Greek original, last quarter of the 4th century BC, in the Kunsthistorisches Museum, Vienna.

Courtesy of the Kunsthistorisches Museum, Vienna

All humans are mortal. All Greeks are humans. Thus, all Greeks are mortals.

More abstractly: *All A's are B's. All B's are C's. Thus, all A's are C's.*

- Ramon Lull (≈ 1235-1316)

<http://www.scs.carleton.ca/~bertossi/complog/material/lull.pdf>

- Gottfried-Wilhelm Leibniz
(1646-1716)

Among many contributions, he thought and wrote about the mechanization of reasoning



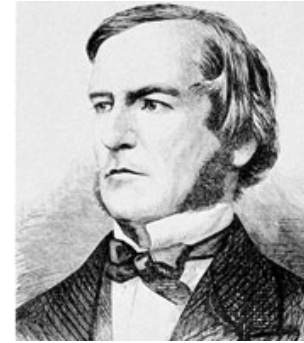
Gottfried Wilhelm Leibniz , coloured stipple engraving .
The Granger Collection , New York City

“It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used ... For if praise is given to the men who have determined the number of regular solids ... how much better will it be to bring under mathematical laws human reasoning, which is the most excellent and useful thing we have.”

- George Boole (1815-1864)

Propositional logic

Captures logical reasoning in algebraic terms



George Boole, engraving .

Courtesy of the trustees of the British Museum ; photograph , J.R. Freeman & Co. Ltd.

The laws/operations that govern logical formula manipulation have algebraic properties \mapsto **Boolean algebra**

A propositional argument such as

$((manSocrates \rightarrow mortalSocrates) \wedge manSocrates) \rightarrow mortalSocrates$

becomes valid

Main work (1854): *“An Investigation into the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities”*

- Gottlob Frege (1848-1925)

Founder of modern mathematical logic, predicate logic, symbolic logic, formal or symbolic deductive systems

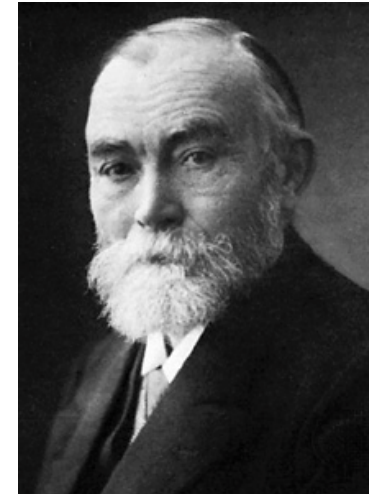
A predicate-logic argument like

$$(\forall x(Man(x) \rightarrow Mortal(x)) \wedge Man(socrates)) \rightarrow Mortal(socrates)$$

becomes valid

In contrast with propositional logic, we have predicates, quantifiers, variables, ... We can talk about properties in general, not only instantiated on particular individuals

An extension of propositional logic



Frege

By courtesy of the Universitätsbibliothek , Jena , E. Ger.

- Some other important names we will find along the way:

Bertrand Russell, David Hilbert, Kurt Goedel, Alfred Tarski,
Alan Turing, ...

- Some of them lead us also to the inception of computer science!

Together with other pioneers: Alonso Church, Emil Post, Steve Kleene,
etc.

Computer science has its roots in mathematical logic ...

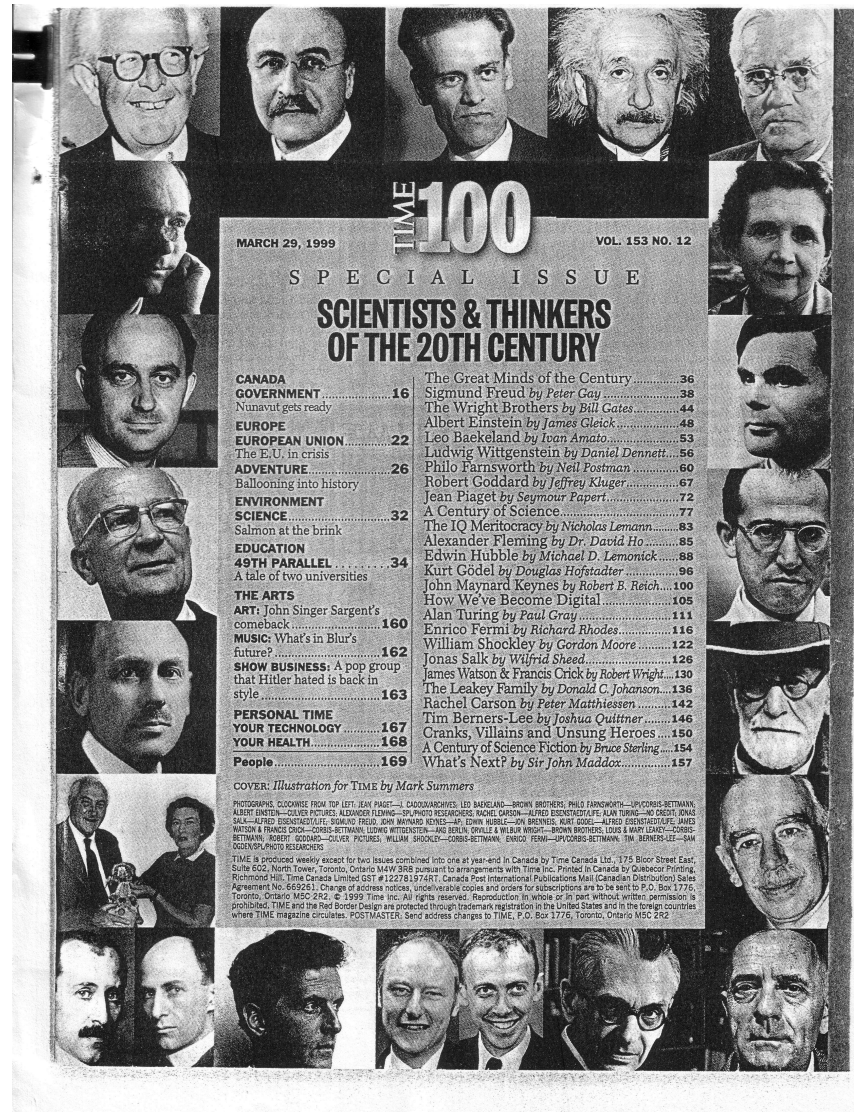
A direct path: Leibnitz → Boole → Frege → Hilbert → Goedel →
Turing → John von Neumann⁵ → ...

See my talk:

<http://people.scs.carleton.ca/~bertossi/talks/hilbTur14.pdf>

⁵Thomas Haigh & Mark Priestley. “Where Code Comes From: Architectures of Automatic Control from Babbage to Algol”. Communications of the ACM, 2016, 59(1):39-44.

Goedel, Turing, Von Neumann €



- George Dyson. “Turing’s Cathedral”. Pantheon Books, 2012.
- Martin Davis. “Engines of Logic”. W. W. Norton and Co., 2000. (also published as: “The Universal Computer: The Road from Leibniz to Turing”. CRC Press, 2012.)



**Chapter 3: Review of Classical Propositional Logic
(and a bit more)**

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

Introduction

- Propositional logic can be used to represent knowledge
- Given a domain, one chooses symbolic **propositional variables**, e.g. p, q, r, s, \dots to *denote* atomic propositions about the domain of discourse

Example: Courses at a university

p : “*Student takes 1805*”

q : “*Student takes Advanced Programming*”

r : “*Student takes Analysis of Algorithms*”, etc.

These are indivisible, **atomic propositions**

- More complex propositions can be represented (constructed) using propositional variables and logical connectives

- This has to do with the *syntax* of the symbolic languages of propositional logic

1. “*If student takes Analysis of Algorithms, then it takes 1805*” can be represented by: $(r \rightarrow p)$

2. “*Student takes Analysis of Algorithms or Advanced Programming*”:
 $(q \vee r)$

3. “*Student takes Analysis of Algorithms or Advanced Programming, but not both*”: $(q \underline{\vee} r)$ (or $(q \oplus r)$)

4. “*Taking 1805 is a necessary condition to take Analysis of Algorithms*”: $(r \rightarrow p)$

5. “*Taking 1805 is a sufficient condition to take Analysis of Algorithms*”: $(p \rightarrow r)$

We are using and modeling “necessary” and “sufficient” conditions as we use them in mathematical statements

6. “Analysis of Algorithms is *not* taken without taking 1805”:

$$(\neg p \rightarrow \neg r)$$

The *contrapositive* of 4.

7. “If 1805 is taken when Advanced Programming is taken, and Advanced Programming is taken, then 1805 is taken”:

$$((q \rightarrow p) \wedge q) \rightarrow p$$

Example: Binary circuit $C(x_1, x_2, x_3, x_4, x_5)$, with 5 binary entries and one binary output

Output value is 1 exactly when one of the inputs is 1, and 0, otherwise

x_1	x_2	x_3	x_4	x_5	output
1	0	0	0	0	1
1	1	0	1	1	0
·	·	·	·	·	·
·	·	·	·	·	·
·	·	·	·	·	·
0	0	0	1	1	0
0	0	0	0	1	1

(the input/output relation)

Want to “reconstruct” C using *not*, *and*, *or* gates from this functional table

Equivalently, we want a **propositional formula** with **logical connectives** \neg, \wedge, \vee that takes the same truth values as the circuit

We want a propositional formula φ belonging to the **propositional language** $L(\{x_1, x_2, x_3, x_4, x_5\})$

$L(\{x_1, x_2, x_3, x_4, x_5\})$ is constructed on the basis of the set of **propositional variables** x_1, x_2, x_3, x_4, x_5 and the **logical connectives**

x_i is propositional variable to denote the claim “input x_i takes the value 1”

Also called **atomic formula** (cannot be decomposed into more basic formulas)

This propositional formula works: $\varphi: (x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5) \vee$

$$(\neg x_1 \wedge x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge \neg x_5) \vee \dots \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge x_5)$$

It represents C in that it takes the truth value 1 exactly when C takes the output value 1, i.e. for the same input values (otherwise both take the value 0)

A concrete circuit C can be built from it (do it!)

A formula can be seen as a symbolic proposition, that can be basic or complex

It follows a precise syntax (not any arbitrary string of characters is an admissible formula)

- Each x_i or $\neg x_i$ is a **literal**, i.e. an atomic formula or the negation of an atomic formula
- The propositional variables x_i take exactly one of the values 1 or 0 (true or false, resp.)
- $\neg x_i$ takes the value 1 iff x_i takes the value 0
- Each **conjunction** (\wedge), e.g. $(x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4 \wedge x_5)$ takes the value 1 iff each of the literals in it takes the value 1
- A **disjunction** (\vee) takes the value 1 iff at least one of the disjuncts takes the value 1

The last four claims do not belong to the syntax of the language, but to the **semantics**

- Formula φ above is in **disjunctive normal form** (DNF): a disjunction of conjunctions of literals
- Another “canonical” formula format is the **conjunctive normal form** (CNF): a conjunction of disjunctions of literals

E.g. $(p \vee \neg q \vee r) \wedge (\neg p \vee \neg p \vee s \vee \neg r)$ is in CNF

- **Propositional formulas are symbolic objects, finite strings of symbols, constructed according to precise syntactic rules** (coming ...)

On Propositional and Other Logics

- Many knowledge representation (KR) formalisms and mechanisms are based on **classical logic** and its extensions
- Symbolic logic offers:
 - **Declarative languages** to represent knowledge: We describe how things are in a domain, not how to compute things
 - Languages that have a precise syntax
 - Symbolic deductive methodologies that can be used for retrieving knowledge that is implicit in the representation
 - Computational implementations of those deductive methodologies
 - Languages that have a clear and well-studied *semantics*

- The **role of the semantics** of a particular logic (and of a particular language of that logic) consists in determining:
 - The possible **meanings** and **interpretations** of the symbolic propositions or formulas
 - The **truth values** of the formulas
 - What is a valid **logical consequence** of what

What are the **valid conclusions** that can be drawn from a set of formulas

- Once this is done, the following questions appear:
 - Is there a purely symbolic deductive mechanism that corresponds to that semantics?
 - Are there computational implementations of the the symbolic deductive mechanisms?

- Every time we invent a logic we should aim at developing it following the “program” presented in these last two slides, namely:

1. Syntax

- 1.1. Grammar of the language

- 1.2. Symbolic deductive reasoning

2. Semantics

- 2.1. Interpretations and models of formulas (and sets thereof)

- 2.2. Logical consequence

3. Correspondence between 1.2 and 2.2.

4. Computational implementation of 1.2. that respects 3.

Syntax of Propositional Logic (official)

- We start from a domain dependent set P of propositional variables

Using the elements of P , logical symbols, and punctuation symbols we develop a full symbolic language $L(P)$

Example: Set of propositional variables:

$$P = \{socratesIsMortal, socratesIsMan\}$$

They **denote** the basic propositions with which we construct more complex formal propositions (or formulas)

They are symbolic names for those propositions

Definition: The formulas of the propositional language $L(P)$ based on P are those strings obtained after a finite number of applications of the following **syntactic rules**

- If $\varphi \in P$, then $\varphi \in L(P)$; it is an atomic formula
- If φ is formula, then also $\neg\varphi$ is a formula
- If φ, ψ are formulas, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$, $(\varphi \leftrightarrow \psi)$ are formulas
- This is a **grammar** to build formulas
- Formulas are being defined **by induction on the structure of the formula**

- Having an arbitrarily complex formula (or a pair thereof), the grammar tells us how to produce a more complex formula in one step
- Similar to inductive (or recursive) definitions in arithmetic:

$$\begin{aligned} \mathit{fact}(n + 1) &:= (n + 1) \times \mathit{fact}(n); \\ \mathit{fact}(0) &= 1 \end{aligned}$$

The induction principle for natural numbers allows us to prove that there is a single function on natural numbers that satisfies the recursive definition of the factorial

- A recursive program based on the rules above can be written to recognize legitimate formulas of $L(P)$

Or to generate formulas

Example: (cont.) The following are formulas of the **propositional language** $L(P)$:

- $socratesIsMortal$ (atomic)
- $(socratesIsMortal \wedge socratesIsMan)$
- $\neg socratesIsMan$
- $((socratesIsMan \rightarrow socratesIsMortal) \wedge socratesIsMan) \rightarrow socratesIsMortal$
- $\neg(socratesIsMan \wedge \neg socratesIsMortal)$

Is $socratesIsMortal \wedge \neg$ a formula of $L(P)$?

It should be possible to prove it, either way

How?

- The inductive definition of formulas allows us to:
 - Define concepts, objects, etc. over formulas of a propositional language
 - Prove that all formulas of a propositional language have a certain property

This is done by *induction on the structure of a formula*

So as we do with natural numbers, using induction on natural numbers

For example, we can prove from the syntactic definition of formula that “every formula has the same number of left and right parenthesis” (a necessary, but not sufficient condition to be a formula)

From this follows that $socratesIsMortal \wedge) \neg$ is not a formula

Semantics of Propositional Logic

- Our treatment of the semantics of propositional languages has been intuitive and informal
- Now we will give to logical connectives their official (and intended) interpretations, for the first time

Propositional formulas do not have a meaning, an interpretation or a truth value a priori

- The semantics of propositional languages is very simple (the language is simple after all)
- We **interpret** formulas by means of **valuations** or **truth assignments**, which are functions from the set of propositional variables to the set of truth values $\{0, 1\}$: $\sigma : P \longrightarrow \{0, 1\}$

- Truth assignments are mathematical functions that represent an external reality in very simple terms
- For the examples above, different possible valuations:

- $\sigma_1 : \{x_1, x_2, x_3, x_4, x_5\} \rightarrow \{0, 1\}$ (among 2^5 possible valuations)

$$x_1 \mapsto 1, \quad x_2 \mapsto 0, \quad x_3 \mapsto 0, \quad x_4 \mapsto 0, \quad x_5 \mapsto 1$$

(x_1 's intended meaning (or denotation for): "Gate x_1 takes input 1", which can be true or false)

$$\sigma_2 : \{x_1, x_2, x_3, x_4, x_5\} \rightarrow \{0, 1\}$$

$$x_1 \mapsto 1, \quad x_2 \mapsto 1, \quad x_3 \mapsto 0, \quad x_4 \mapsto 1, \quad x_5 \mapsto 0$$

- $\sigma_1 : \{socratesIsMortal, socratesIsMan\} \rightarrow \{0, 1\}$

$$socratesIsMortal \mapsto 1, \quad socratesIsMan \mapsto 0$$

$$\sigma_2 : \{socratesIsMortal, socratesIsMan\} \rightarrow \{0, 1\}$$

$$socratesIsMortal \mapsto 0, \quad socratesIsMan \mapsto 0$$

- According to the semantics of classical propositional logic, *under a truth valuation*, every atomic formula takes one and only one of the two possible truth values (0 or 1)
- Intuitively, each atomic formula **denotes** a basic, undecomposable, complete sentence

Giving to it the value 1 (0) means that the denoted proposition is (considered to be) true (false)

- A valuation is just a simple function as interpretation, in comparison with the set-theoretic structures used for predicate logic

σ_1	truth value
x_1	1
x_2	0
x_3	0
x_4	0
x_5	1

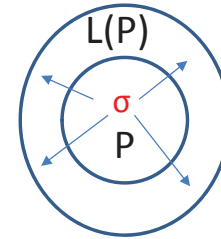
Or

σ_1	true atomic flas.
	x_1
	x_5

(the other 3 false by default, as in relational DBs)

Sometimes denoted: $\sigma_1 = \{x_1, x_5\}$

- Truth valuations will be extended to all the propositional formulas



- More complex formulas take a truth value 1 or 0 on the basis of the values assigned to the atomic formulas and the **semantics of the logical connectives** $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$

This is sanctioned in (by) this **official table**: (φ, ψ are arbitrary formulas)

φ	ψ	$\neg\varphi$	$(\varphi \wedge \psi)$	$(\varphi \vee \psi)$	$(\varphi \rightarrow \psi)$	$(\varphi \leftrightarrow \psi)$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

- This is an inductive definition (on the structure of the formulas)

Given a valuation $\sigma : P \longrightarrow \{0, 1\}$ (the base case), σ is inductively extended to all of $L(P)$

- Now every formula of $L(P)$ receives exactly one of the two truth values under a valuation σ
 - In non-classical propositional logics this may not hold, e.g. there are “multi-valued logics”, with more than two truth values, e.g. *true*, *false*, *undetermined*
 - Or the same formula may be both true and false, etc.
- Negation has a “classical” semantics

For $\neg\varphi$ to be true, φ has to be false

Lack of “evidence” for φ is not enough (as in some other logics)

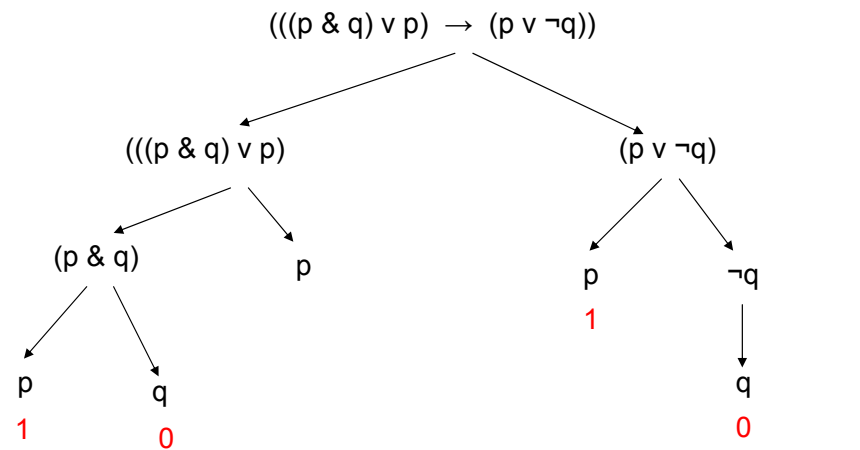
In classical logic, negation is a “strong negation” (very demanding)

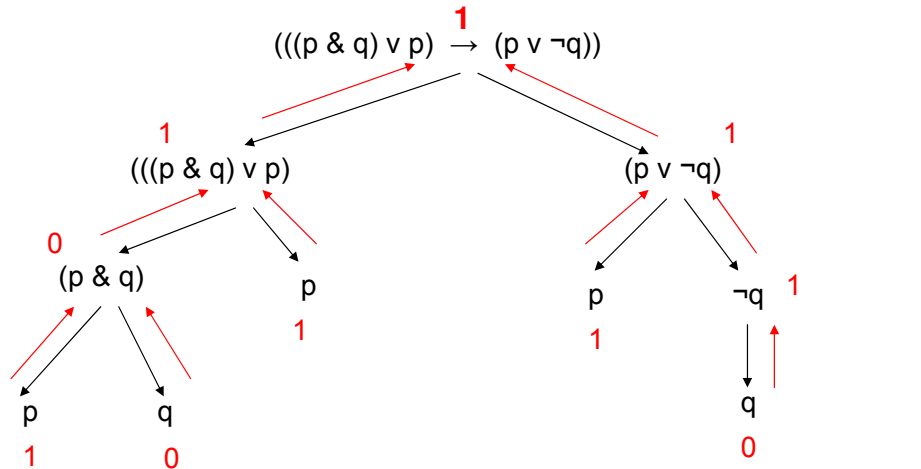
Example: Consider $P = \{p, q\}$, and $\sigma: p \mapsto 1, q \mapsto 0$

Which is the truth value of $((p \wedge q) \vee p) \rightarrow (p \vee \neg q)$ under valuation σ ?

We evaluate its truth value **recursively**

The (sub)formula tree:





- Valuation σ *satisfies the formula*; or *the formula is true wrt* σ when it takes value 1 under σ

Notation: $\sigma \models ((p \wedge q) \vee p) \rightarrow (p \vee \neg q)$

We also say that “ σ is a model of φ ”, a semantic statement

(More, precisely “is a model that makes φ true”)

- The **notion of satisfaction is compositional**: the truth value $\sigma(\varphi)$ of a formula φ depends on the truth values of the subformulas of φ

Some notions associated to formula satisfaction:

- The formulas $p \wedge (p \rightarrow q) \wedge (p \rightarrow \neg q)$, and $p \wedge \neg p$ are **contradictions**, i.e. they are false under every valuation

p	q	$\neg q$	$p \rightarrow q$	$p \rightarrow \neg q$	$p \wedge (p \rightarrow q) \wedge (p \rightarrow \neg q)$
1	1	0	1	0	0
1	0	1	0	1	0
0	1	0	1	1	0
0	0	1	1	1	0

- The formula $(p \rightarrow q) \wedge (p \rightarrow \neg q)$ is **satisfiable** (or consistent), i.e. there is a valuation that makes it true

p	q	$\neg q$	$p \rightarrow q$	$p \rightarrow \neg q$	$(p \rightarrow q) \wedge (p \rightarrow \neg q)$
1	1	0	1	0	0
1	0	1	0	1	0
0	1	0	1	1	1
0	0	1	1	1	1

- The formula

$$\left(\left(\text{socratesIsMan} \rightarrow \text{socratesIsMortal} \right) \wedge \text{socratesIsMan} \right) \rightarrow \text{socratesIsMortal}$$

is **valid** or a **tautology**, i.e. **it is always true** for every σ , i.e. no matter what truth values are taken by the propositional variables

This can be verified using a **truth table**

p	q	$p \rightarrow q$	$(p \rightarrow q) \wedge p$	$((p \rightarrow q) \wedge p) \rightarrow q$
1	1	1	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	1

Exercise: Consider the argument

If Joe fails to submit the a project in course CS414, then he fails the course. If Joe fails CS414, then he cannot graduate. Hence, if Joe graduates, he must have submitted a project.

Is it a valid argument?

Exercise: The same with the argument

If X is greater that zero, then if Y is zero, then Z is zero. Variable Y is zero. Hence, either X is greater than zero or Z is zero.

Next two slides are personal reading

Exercise: modeling valid patterns of reasoning

- **Proofs by contradiction** follow the same logical pattern

Usually, theorems are of the form:

$$\textit{HYPOTHESIS} (H) \Rightarrow \textit{THESIS} (T) \quad (\alpha)$$

The proof can be direct (start from H and reach T) or

By contradiction: we try to prove

$$H \textit{ and not } T \Rightarrow F \quad (\beta)$$

where F is a contradiction (an always false proposition)

If we establish (β) , we know that (α) is true

Because $(\beta) \Rightarrow (\alpha)$ is true

That is, in terms of propositional logic,

$$((H \wedge \neg T) \rightarrow F) \rightarrow (H \rightarrow T)$$

is always true, a tautology (check it!)

- **Proofs by cases:** again we want to establish a theorem of the form (α)

We consider two (or more) cases C_1, C_2 for H that cover all the possibilities

$$\frac{\begin{array}{c} (H \wedge C_1) \rightarrow T \\ (H \wedge C_2) \rightarrow T \\ C_1 \vee C_2 \end{array}}{H \rightarrow T}$$

Can be justified:

$$((H \wedge C_1) \rightarrow T) \wedge ((H \wedge C_2) \rightarrow T) \wedge (C_1 \vee C_2) \longrightarrow (H \rightarrow T)$$

is a tautology (check!)

- The formulas
 - $(socratesIsMan \rightarrow socratesIsMortal)$, and
 - $(\neg socratesIsMan \vee socratesIsMortal)$

are **logically equivalent**, i.e. they take the same truth values for every truth assignment σ

p	q	$\neg p$	$(\neg p \vee q)$	$(p \rightarrow q)$
1	1	0	1	1
1	0	0	1	1
0	1	1	1	1
0	0	1	0	0

This is the same as saying that

$$((p \rightarrow q) \leftrightarrow (\neg p \vee q))$$

is a tautology (do it!)

We write $(p \rightarrow q) \equiv (\neg p \vee q)$

- The symbol for logical equivalence (\equiv) is a symbol of the **metalanguage** used to talk about objects of the symbolic language (the object language)

That is $(p \rightarrow q) \equiv (\neg p \vee q)$ is not a propositional formula of language $L(\{p, q\})$

$(p \rightarrow q) \equiv (\neg p \vee q)$ tells us that $((p \rightarrow q) \leftrightarrow (\neg p \vee q))$ (which is a propositional formula of $L(\{p, q\})$) is a tautology

Some useful “logical equivalences”:

- For arbitrary formulas φ, ψ, χ of a propositional language $L(P)$, it holds (check them with truth tables):
 - $(\varphi \rightarrow \psi) \equiv (\neg\varphi \vee \psi)$ (we had this one)
 - $\neg\neg\varphi \equiv \varphi$
 - $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$
 - $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$
 - $(\varphi \leftrightarrow \psi) \equiv ((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$
 - $(\varphi \wedge (\psi \vee \chi)) \equiv ((\varphi \wedge \psi) \vee (\varphi \wedge \chi))$
 - $(\varphi \vee (\psi \wedge \chi)) \equiv ((\varphi \vee \psi) \wedge (\varphi \vee \chi))$
 - $(\varphi \rightarrow \psi) \equiv (\neg\psi \rightarrow \neg\varphi)$ (contrapositive)
 - $(\varphi \leftrightarrow \psi) \equiv (\neg\varphi \leftrightarrow \neg\psi)$ (contrapositive)
- Using these equivalences we could express every logical connective in terms of, e.g. \neg and \vee , obtaining logically equivalent formulas

Example:

$$\begin{aligned}(p \wedge (\neg q \rightarrow s)) &\equiv (p \wedge (\neg\neg q \vee s)) \\ &\equiv (p \wedge (q \vee s)) \equiv \neg(\neg p \vee \neg(q \vee s))\end{aligned}$$

- We say that $\{\neg, \vee\}$ is a “functionally complete set of connectives”
- We can also use those logical equivalences as “algebraic laws”

They allow us to symbolically manipulate formulas in order to obtain new logically equivalent formulas (as in the previous example)

Exercise: (a) Transform the formula in DNF in slide 6 into an equivalent one in CNF

(b) What is the computational complexity of the transformation process?

Some computational decision problems:

A natural computational problem is to **decide** if a formula is satisfiable

Using truth tables (cf. page 34) gives us a decision algorithm

Deciding if a formula is satisfiable using that method may be difficult

If the number of propositional variables in the formula is n , we have to check 2^n valuations in the worst case

The number of steps for checking satisfiability using truth tables in this way is **exponential** in the size of the formula (in the worst case)

Are there more efficient decision algorithms?

The most important open problem in computer science consists in determining if there is a polynomial time algorithm for deciding satisfiability of propositional formulas, i.e. for solving SAT

Nobody knows; and people have been trying to prove either direction for several decades, and systematically since, at least, 1971

Steve Cook's Conjecture (1971): There is no polynomial time algorithm for deciding if propositional formulas are satisfiable or not

Why so prominent a problem?

Why apparently so difficult?

Computational Decision Problems:

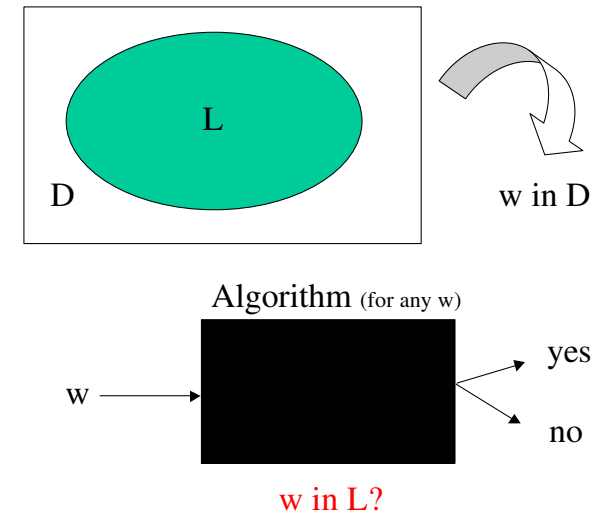
D is a domain of (symbolic) words
(problem instances)

L a subset of D

We want a **general** algorithm to
decide membership of L

D could be a propositional
language $L(P)$

L the subset of its satisfiable
formulas



If such algorithm exists, we say:
“ L is decidable” (undecidable, otherwise)

Not every problem of a computational nature is computationally solvable

There is no algorithm (or a program in Smalltalk) to decide if arbitrary programs in Smalltalk terminate

We say that this decision problem is **undecidable** or **unsolvable**

This comes from the celebre “**Turing machine’s halting problem**”:

There is not Turing machine (TM) to decide if an arbitrary TM (represented by a binary code) terminates or not (Alan Turing \approx 1936)

Computation has intrinsic limitations ...

SAT is decidable (solvable), but apparently it’s a hard problem ...

Related decision problems:

Checking if a formula is a tautology and checking if it is a contradiction do not seem to be easier problems:

- φ is a contradiction iff φ is not satisfiable
- φ is a tautology iff $\neg\varphi$ is not satisfiable
- φ is satisfiable iff φ is not a contradiction iff $\neg\varphi$ is not a tautology
- φ and ψ are logically equivalent iff $(\varphi \leftrightarrow \psi)$ is a tautology

These problems are computationally related ...

It is not surprising that these problems around propositional logic (PL) are difficult

PL is still quite expressive and can be used to encode combinatorial problems in other areas of math, computing, etc.

Another difficult decision problem: 3-GC

- Deciding if a map (arbitrary) can be colored with 3 colors (without adjacent countries sharing a color)
- Same status as SAT ... Not know if there is a general polynomial-time (PTIME) decision algorithm for 3-GC
- Actually, they can be reduced to each other (in PTIME)

In particular, given a map, \mathcal{M} , it is possible to compute (fast) a set $\Sigma(\mathcal{M})$ of propositional formulas such that:

\mathcal{M} can be colored with 3 colors iff $\Sigma(\mathcal{M})$ is satisfiable

(a reduction from 3-GC to SAT)

Exercise: Use Prover9 to play with different submaps of Southamerica, with 2 and 3 colors, via the propositional reduction



- SAT and 3-GC belong to the same class of decision problems

That of **NP-Complete** problems, the most difficult in the complexity class **NP** of decision problems

SAT, 3-GC \in **NP**, and they are complete for the class

NP contains the problems solvable in non-deterministic PTIME

Equivalently, those whose positive solutions can be verified in PTIME

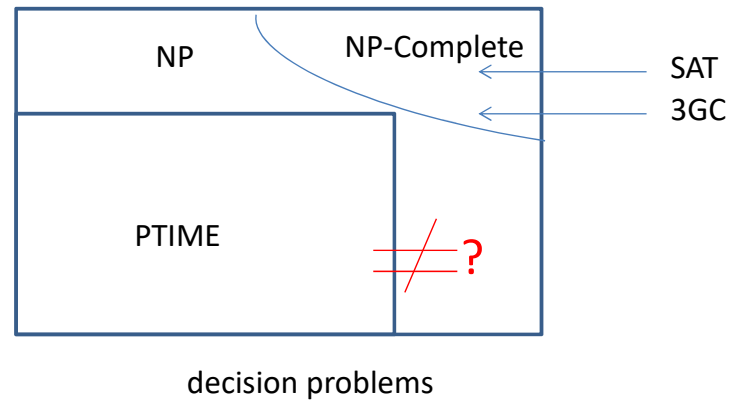
- There are thousands of important and practical decision problems that are NP-Complete

All of them with the same open status wrt. solvability in PTIME

- If the question is answered for one of them, the same answer automatically applies to all of them (they are all reducible to each other in PTIME)

- SAT and 3GC are in NP, because any positive instance w , i.e. that gets answer “yes” to the question “does $w \in L?$ ”, has a short certificate of membership that can be verified in PTIME
 - If φ is a propositional formula, and σ is a valuation that makes φ true (the certificate), verifying $\sigma \models \varphi$ can be done easily (and the size of σ is “short” wrt. the size of φ , definitely not exponential)
 - If G is a map that is colorable with 3 colors, and c is such a coloring, verifying that c is a proper coloring for G is easy
- More technically, a problem L is NP-Complete if: (a) $L \in \text{NP}$; (b) every problem $L' \in \text{NP}$ can be reduced in PTIME to L (and hence solved as an instance for L)

For this to happen, L has to be among the most difficult problems in NP



- Conjecture: $PTIME \subsetneq NP$

If this is true, the NP-Complete problems will be outside PTIME



Example: An undirected graph $\mathcal{G} = \langle V, E \rangle$, with set of vertices $V = \{1, 2, 3, 4\}$ and set of edges $E = \{\{1, 3\}, \{1, 2\}, \{2, 4\}\}$

(no simple loops, i.e. $\{v\} \notin E$ for every vertex v , no multiple edges)

We want to express properties of graphs like this in a propositional language

For example, that “the graph has a vertex of degree 2”

The **degree of a vertex** is the number of vertices that are directly connected to it

Introduce propositional variables p_{ij} , $1 \leq i \leq 4$, $i < j \leq 4$, with the intended meaning that “vertices i and j are connected by an edge”

In a particular graph \mathcal{G} the p_{ij} become true or false, that is every graph determines a valuation $\sigma_{\mathcal{G}}$

An outside reality being described by the symbolic language naturally becomes a domain of interpretation, i.e. a truth valuation, e.g. for the given graph, $\sigma_{\mathcal{G}}(p_{12}) = 1$, $\sigma_{\mathcal{G}}(p_{34}) = 0$

We build a boolean function (propositional formula) φ_{2d} that becomes true of a graph \mathcal{G} , i.e. under valuation $\sigma_{\mathcal{G}}$, if and only if \mathcal{G} has a vertex of degree 2

The following formula of the propositional language $L(\{p_{ij} \mid 1 \leq i \leq 4, i < j \leq 4\})$ works:

Vertex 1: $(p_{12} \wedge p_{13} \wedge \neg p_{14}) \vee (p_{12} \wedge p_{14} \wedge \neg p_{13}) \vee (p_{14} \wedge p_{13} \wedge \neg p_{12})$ ∨

Vertex 2: essentially the same ∨

Vertex 3: idem ∨

Vertex 4: idem

BTW this formula φ_{2d} is in “disjunctive normal form” (DNF): a disjunction of conjunctions of literals

It is useful to have formulas in normal forms, e.g. for some reasoning and computational tasks

The language $L(\{p_{ij} \mid 1 \leq i \leq 4, i < j \leq 4\})$ can be used to describe general properties of undirected graphs with 4 nodes and no simple loops; and also to describe any particular graph of this kind (not only \mathcal{G} above)

Proposition: Each propositional formula is logically equivalent to both a formula in DNF and CNF (conjunction of disjunctions of literals)

This can be proved in general by induction on formulas

Here the “trick” is doing induction simultaneously for the two claims

Example: $(\neg(p \rightarrow \neg q) \leftrightarrow (q \wedge (\neg r \rightarrow s)))$

Transformation into CNF:

1. $(\neg(p \rightarrow \neg q) \rightarrow (q \wedge (\neg r \rightarrow s))) \wedge$
 $((q \wedge (\neg r \rightarrow s)) \rightarrow \neg(p \rightarrow \neg q))$
2. $\neg(\neg(p \rightarrow \neg q)) \vee (q \wedge (\neg r \rightarrow s)) \wedge$
 $\neg(q \wedge (\neg r \rightarrow s)) \vee (\neg(p \rightarrow \neg q))$
3. $(p \rightarrow \neg q) \vee (q \wedge (\neg r \rightarrow s)) \wedge$
 $(\neg q \vee \neg(\neg r \rightarrow s)) \vee (\neg(p \rightarrow \neg q))$
4. $(\neg p \vee \neg q) \vee (q \wedge (\neg \neg r \vee s)) \wedge$
 $(\neg q \vee \neg(\neg \neg r \vee s)) \vee (\neg(\neg p \vee \neg q))$

5. $(\neg p \vee \neg q) \vee (q \wedge (r \vee s)) \wedge$
 $(\neg q \vee \neg(r \vee s)) \vee (\neg(\neg p \vee \neg q))$
6. $(\neg p \vee \neg q \vee q) \wedge (\neg p \vee \neg q \vee (r \vee s)) \wedge$
 $(\neg q \vee (\neg r \wedge \neg s)) \vee ((\neg\neg p \wedge \neg\neg q))$
7. $(\neg p \vee \neg q \vee q) \wedge (\neg p \vee \neg q \vee r \vee s) \wedge$
 $((\neg q \vee \neg r) \wedge (\neg q \vee \neg s)) \vee ((p \wedge q))$
8. $(\neg p \vee \neg q \vee q) \wedge (\neg p \vee \neg q \vee r \vee s) \wedge$
 $((\neg q \vee \neg r) \vee (p \wedge q)) \wedge ((\neg q \vee \neg s) \vee (p \wedge q))$
9. $(\neg p \vee \neg q \vee q) \wedge (\neg p \vee \neg q \vee r \vee s) \wedge$
 $(\neg q \vee \neg r \vee p) \wedge (\neg q \vee \neg r \vee q) \wedge (\neg q \vee \neg s \vee p) \wedge (\neg q \vee \neg s \vee q)$
10. $(\neg p \vee \neg q \vee r \vee s) \wedge (\neg q \vee \neg r \vee p) \wedge (\neg q \vee \neg s \vee p)$

CNF: Conjunction of disjunctions of literals!

Otter and Prover9 do this transformation before any reasoning task

Propositional logic can be used to represent knowledge

Notice that a valuation $\sigma: P \rightarrow \{0, 1\}$ can be seen as a **model**

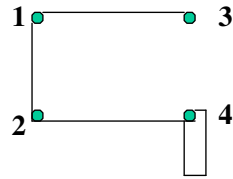
It is a model in the usual sense

It corresponds to a simplified representation (description) of a reality, that captures what is essential to it

The very simple model is given in terms of:

- **Names for basic propositions** (that are relevant about that reality)
- An indication of which of them are **true** and which of them are **false**

Example: The external reality is, e.g. an undirected graph



To build the logical model, we choose a set of propositional variables

$$P = \{p_{11}, p_{12}, p_{13}, p_{14}, p_{22}, p_{23}, p_{24}, p_{33}, p_{34}, p_{44}\},$$

that contains names for relevant basic propositions about the concrete graph

The graph is modeled by the specific valuation $\sigma^* : P \rightarrow \{0, 1\}$:

- $\sigma^*(p_{12}) = \sigma^*(p_{13}) = \sigma^*(p_{24}) = \sigma^*(p_{44}) = 1$
- $\sigma^*(p_{14}) = \sigma^*(p_{23}) = \sigma^*(p_{34}) = \sigma^*(p_{11}) = \sigma^*(p_{22}) = \sigma^*(p_{33}) = 0$

In this case, **the model can be captured at the symbolic level** by the formula

$$\varphi: p_{12} \wedge p_{13} \wedge p_{24} \wedge p_{44} \wedge \neg p_{14} \wedge \neg p_{23} \wedge \neg p_{34} \wedge \neg p_{11} \wedge \neg p_{22} \wedge \neg p_{33}$$

Or, equivalently, by set of formulas

$$\Sigma = \{p_{12}, p_{13}, p_{24}, p_{44}, \neg p_{14}, \neg p_{23}, \neg p_{34}, \neg p_{11}, \neg p_{22}, \neg p_{33}\},$$

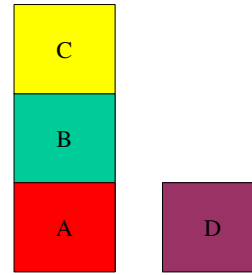
all of which have to be made simultaneously true

Every valuation σ that satisfies φ (or Σ) coincides with σ^*

Example: The formula in page 60 is true under an assignment σ iff the set of formulas $\Sigma = \{\neg p \vee \neg q \vee r \vee s, \neg q \vee \neg r \vee p, \neg q \vee \neg s \vee p\}$ is true under σ

We can say the formula is logically equivalent to the set of formulas ...

Exercise: Pick any external reality you may want to describe, and model it as a valuation for a propositional language



Exercise: Produce a language of propositional logic to describe the blocks world \mathcal{W} in the figure above. You should be able to express in a non-atomic manner things like “block C is to the left of block D ”. More precisely:

- (a) Indicate the propositional variables.
- (b) The world in the figure induces a truth valuation for the language.
Give the truth values for the propositional variables.
There must be true and false propositional variables (you may have to revisit item 1 accordingly).
- (c) Give examples of: a true atomic formula, a false atomic formula, a true non-atomic formula, a false non-atomic formula (all in relation to \mathcal{W} , of course).
- (d) Give a description (a knowledge-base) Σ in your language that does not explicitly contain, but has as a logical consequence, that “block C is to the left of block D ”.
(Your may have to revise item 1, etc., if necessary)

Semantics: Sets of Formulas

Consider a valuation $\sigma: P \rightarrow \{0, 1\}$ and a set of formulas $\Sigma \subseteq L(P)$

σ is a **model of (satisfies, makes true)** Σ , denoted $\sigma \models \Sigma$, if σ makes true all the formulas in Σ :

$$\sigma \models \Sigma \quad :\Leftrightarrow \quad \sigma \models \varphi \quad \text{for all } \varphi \in \Sigma \quad (*)$$

(a relationship between a semantic and a symbolic object)

We usually deal with sets of formulas: Theories, knowledge bases, ontologies, etc. are sets of formulas

Notice that the expression (*) is not a symbolic formula of the object language, but one of the metalanguage, the usual mathematical language; hence the “ \Leftrightarrow ” (“ \leftrightarrow ” is reserved for the object language)

A mathematical definition like that in (*) is nothing but an abbreviation, a shorthand (the new notion or object on the LHS of the implication) for something stated in terms of things that have already been defined (the RHS)

Example: Consider $\Sigma = \{p, \neg q, (p \vee r), (\neg q \vee \neg s)\}$

The semantics of Σ is given by its **models**, i.e. by the (set of) valuations that make it true (i.e. all the formulas in it true)

Valuation	p	q	r	s	Σ
σ_1	1	0	1	1	1
σ_2	1	0	0	1	1
σ_3	1	0	1	0	1
σ_4	1	0	0	0	1

For example: $\sigma_1 \models \Sigma$

Here Σ is **satisfiable or consistent**: it has models, i.e. valuations that make it true

But $\Sigma' = \{(p \rightarrow q), p, \neg q\}$ is **inconsistent**, it has no models

A particular case:

If $\Sigma \subseteq L(P)$ is the empty set of formulas, \emptyset , then every valuation σ over P satisfies Σ

We have to verify that, for every valuation σ over P , $\sigma \models \emptyset$

This is equivalent to: For every valuation σ over P , it holds: For every formula $\varphi \in L(P)$:

$$\varphi \in \emptyset \implies \sigma \models \varphi$$

This implication is trivially true, because the antecedent is false

(Similar to the proof that the empty set is a subset of every set)

Some Remarks:

- Why “model of a set of formulas”?

A set of formulas Σ can be seen as a linguistic description or specification (spec)

A valuation σ that satisfies Σ can be seen as a possible materialization, a concrete witness, of that description

Like a model of a ship or of an airplane ... they can be models of a blueprint (the spec)

- A spec Σ may be uncertain and have several different models, e.g. if the spec says “it has 4 wheels”, a model could be a car, or a shopping cart, etc.
- If we ask a car dealer about “a fast, small, red car”, there is plenty of room for uncertainty: many cars satisfy this specification

We make our choice more accurate (narrowing down the spectrum of choices) by imposing extra requirements (conditions), e.g. “has to be a sedan”, etc.

- We can narrow down the number of potential models by adding knowledge (new conditions) into Σ (examples below)
- The less models we have for Σ , the more meaning (semantics) we capture

There is less **uncertainty** wrt the knowledge base

In the previous example, there is certainty about the truth status of p , but uncertainty about the truth status of s : it is true in some models of Σ , and false in others

However, it is certain wrt Σ that $(q \rightarrow r)$ is true, because it is true in all models of Σ

This implicit and certain knowledge will be considered later to be a “logical consequence” of Σ

This implicit certain knowledge is the one we want to establish by automated reasoning, without considering all possible valuations

Example: Consider $P = \{p, q\}$, and $\Sigma_1 = \{p\}$ has two models:

(a) $\sigma_1: p \mapsto 1, q \mapsto 0$; and (b) $\sigma_2: p \mapsto 1, q \mapsto 1$

There is uncertainty wrt q at the light of the knowledge base Σ

Whereas for $\Sigma_2 = \{p, \neg q\}$, there is only one model, namely σ_1 , and there is less uncertainty

- We can capture (more) semantics by imposing **semantics constraints**, i.e. extra formulas that have to be true, which has the effect of filtering out some of the originally admissible models

We could have the constraint saying that “it is not possible for r and s to be both simultaneously true”, i.e. we impose the formula

$$\psi: \neg(r \wedge s):$$

	Valuation	p	q	r	s	Σ	$\Sigma \cup \{\psi\}$
×	σ_1	1	0	1	1	1	0
✓	σ_2	1	0	0	1	1	1
✓	σ_3	1	0	1	0	1	1
✓	σ_4	1	0	0	0	1	1

Like imposing integrity constraints (aka. semantic constraints) on databases; with them some undesirable instances (states) of the DB are discarded ...

Characterizing Models

- It is common practice to express a valuation σ for a propositional language $L(P)$ as the set of propositional variables that it makes true

The others left outside are considered to be false under σ

Example: If $P = \{p, q, r, s, t\}$, and $\sigma(p) = \sigma(r) = \sigma(t) = 1$, and $\sigma(q) = \sigma(s) = 0$, then σ is identified with the set $\{p, r, t\}$ (cf. page 29)

We can even write $\sigma = \{p, r, t\}$

- Given a set of formulas $\Sigma \subseteq L(P)$, its models or satisfying valuations are also expressed similarly

Example: For $\Sigma = \{(p \wedge q) \vee r\}$, its models can be written as:

$$\{p, q\}, \{p, q, r\}, \{r\}, \{r, q\}, \{r, p\}$$

- It is common to **compare models** of a set of formulas

This is important for capturing certain forms of non-classical reasoning (more on this coming)

One singles out certain **intended or preferred models** of a set of formulas

Example: (cont.) For the set of models Σ :

$$Mod(\Sigma) = \{\{p, q\}, \{p, q, r\}, \{r\}, \{r, q\}, \{r, p\}\}$$

Some of the models are “minimal”, under different notions of minimality

- Minimal models under set inclusion, i.e. models that have no proper submodel, are: $MinMod_{\subseteq}(\Sigma) = \{\{p, q\}, \{r\}\}$
- Minimal models under cardinality, i.e. models with a minimum number of true variables: $MinMod_{card}(\Sigma) = \{\{r\}\}$

Exercise: Find the models and minimal models (under set inclusion and cardinality) of the following sets of formulas:

- $\Sigma = \{r \wedge \neg s \rightarrow p \vee q, q \wedge \neg p \rightarrow r\}$
- $\Sigma = \{r \wedge \neg s \rightarrow p \vee q, q \wedge \neg p \rightarrow r, r\}$

Remember that a model makes true all the formulas in the set

Logical Consequence

Example: Consider Σ containing:

1. $(p \vee \neg(q \wedge r)) \rightarrow (s \vee u)$
2. $\neg r$
3. $\neg s$

Is there any implicit knowledge we can obtain, deduce from Σ ?

What are the **logical consequences** from Σ ?

Look at the models of Σ : (4 out of 2^5 valuations)

	p	q	r	s	u	Σ
σ_1	1	1	0	0	1	1
σ_2	1	0	0	0	1	1
σ_3	0	1	0	0	1	1
σ_4	0	0	0	0	1	1

Notice that **every time** Σ is true, also u is true

Whenever **all** the formulas in Σ are simultaneously true, also formula u becomes true

In other words, **for every** truth assignment

$$\sigma : \{p, q, r, s, u\} \rightarrow \{0, 1\}$$

that makes every formula in Σ true, it holds $\sigma(u) = 1$

We say that u is a logical consequence of Σ

In symbols:

$$\{(p \vee \neg(q \wedge r)) \rightarrow (s \vee u), \neg r, \neg s\} \models u$$

(a relationship between two symbolic objects)

Definition: For $\Sigma \subseteq L(P)$ and $\varphi \in L(P)$

$$\Sigma \models \varphi \quad :\Leftrightarrow \quad \text{for every valuation } \sigma, \\ \sigma \models \Sigma \quad \Longrightarrow \quad \sigma \models \varphi$$

(Σ may be infinite)

In principle, we had to check 2^5 possible valuations!

Can we do better?

Typical mathematical reasoning at the semantic level?

Let σ be an arbitrary assignment that makes Σ true; it holds

- $\sigma(\neg r) = 1$ Then $\sigma(r) = 0$
- $\sigma(\neg s) = 1$ Then $\sigma(s) = 0$
- $\sigma((p \vee \neg(q \wedge r)) \rightarrow (s \vee u)) = 1$ (*)

Now: $\sigma(r) = 0$ implies $\sigma((q \wedge r)) = 0$ implies $\sigma(\neg(q \wedge r)) = 1$ implies

$$\sigma((p \vee \neg(q \wedge r)) = 1 \quad (**)$$

On the other side, $\sigma(s) = 0$ implies

$$\sigma(s \vee u) = \sigma(u) \quad (***)$$

From (*) and (**) we obtain: $\sigma(s \vee u) = 1$

With (***) we obtain $\sigma(u) = 1$

Was it necessary to carry the generic σ around in this proof?

Can't we do the same by purely symbolic manipulation of formulas, without appealing to explicit valuations?

Is there a **purely symbolic process** to get the same?

That can be implemented in computer?

Example:

$\{(socratesIsMan \rightarrow socratesIsMortal), socratesIsMan\}$

$\models socratesIsMortal$

p	q	$(p \rightarrow q)$
1	1	1
1	0	0
0	1	1
0	0	1

$\{(socratesIsMan \rightarrow socratesIsMortal), socratesIsMortal\}$

$\not\models socratesIsMan$

p	q	$(p \rightarrow q)$
1	1	1
1	0	0
0	1	1
0	0	1

✗ (a counterexample for \models)

Example: (Superman strikes back) We established semantically that:

$$\{(a \wedge w) \rightarrow p, (\neg a \rightarrow i) \wedge (\neg w \rightarrow m), \neg p, \\ e \rightarrow (\neg i \wedge \neg m)\} \models \neg e$$

Example: Show that

$$\{(p \vee \neg(q \wedge r)) \rightarrow (s \vee u), \neg r, \neg s\} \not\models q$$

That is, q is not a logical consequence of Σ

We need to exhibit a **counterexample**, i.e. a concrete valuation σ^* , such that:

- $\sigma^* \models \Sigma$, and
- $\sigma^* \not\models q$

The valuations σ_2 and σ_4 on page 77 are counterexamples

Remarks:

- A particular case: $\Sigma = \emptyset$, the empty set of formulas

It holds: For every formula $\varphi \in L(P)$:

$$\emptyset \models \varphi \text{ iff } \varphi \text{ is a tautology (or valid)}$$

That is, φ is a tautology iff it holds without hypothesis/conditions

The proof is immediate from the fact that, for every valuation σ over P , $\sigma \models \emptyset$ (seen before)

Notation: Instead of $\emptyset \models \varphi$, we simply write $\models \varphi$

E.g. $\models (p \vee \neg p)$, $\models (((p \rightarrow q) \wedge p) \rightarrow q)$

But $\not\models (((p \rightarrow q) \wedge q) \rightarrow p)$

- Deciding $\Sigma \models \varphi$ may be computationally difficult

At least as difficult as checking “tautology”

In fact:

If $\Sigma = \{\varphi_1, \dots, \varphi_n\}$, deciding $\Sigma \models \varphi$ is equivalent to deciding if $(\varphi_1 \wedge \dots \wedge \varphi_n) \rightarrow \varphi$ is a tautology

E.g. $\{(p \rightarrow q), p\} \models q \iff \models ((p \rightarrow q) \wedge p) \rightarrow q$

- A note of caution!

Notice that the symbol \models is overloaded; it's used to denote two different notions

- $\sigma \models \varphi$: a relationship between a semantic object (σ) and a syntactic object (φ)
- $\Sigma \models \varphi$: a relationship between two syntactic objects, a set of formulas and a formula

We could see the latter as a generalization of the former:

$\Sigma \models \varphi$ means that $\sigma \models \varphi$ holds for every model σ of Σ , i.e. φ is true in the set of models of Σ

- Beware!

- For a valuation σ and a formula φ (of the corresponding language) **it always holds**

$$\sigma \models \varphi \quad \text{or} \quad \sigma \models \neg\varphi$$

That is, a valuation always makes a formula true or its negation true (but not both)

That is, the truth value of a formula is always determined with respect to a valuation

- Whereas, **it may hold**: $\Sigma \not\models \varphi$ and $\Sigma \not\models \neg\varphi$

An example can be obtained from page 77: there are models for q and models for $\neg q$, so neither can be a logical consequence

$$\Sigma \not\models q \quad \text{and} \quad \Sigma \not\models \neg q$$

There are models of Σ where q is false, and models of Σ where $\neg q$ is false

q (and so $\neg q$) is undetermined wrt. Σ , a knowledge base that exhibits uncertainty as witnessed by the presence of several models

We can say that the truth value of q (and $\neg q$) is undetermined wrt. the knowledge base Σ

In this case, the knowledge base Σ is **incomplete**: this set of formulas (or theory) does not determine the truth value of every formula of its language

Example: $P = \{p, q, r\}$ and $\Sigma = \{p, \neg q, r\}$

Every formula φ of $L(P)$ has its truth value determined wrt. Σ

That is, it holds either $\Sigma \models \varphi$ or $\Sigma \models \neg\varphi$

Why? Σ encodes a whole and single valuation, say σ , that makes it true,
namely: $\sigma(p) = \sigma(r) = 1, \sigma(q) = 0$

Σ is a **complete** set of formulas: it determines complete knowledge (as expressible in its language) A very special case!

- Summarizing:
 - A valuation σ for $L(P)$ is a **full description** of a reality
All the basic statements have a truth value
Hence all the formulas have a truth value
 - A set of formulas Σ of $L(P)$ may be only a **partial description** of a reality
Only certain statements (basic or complex) are assumed to be true
So, some other statements may be left undetermined, uncertain at the light of Σ

Some Properties of the Semantics

- There are properties that make this logic “classical”

They are important for the design and implementation of symbolic reasoning

- All of them can be easily proved (really!) from the previous definitions

A. A Two-valued Logic

- For propositional $\varphi \in L(P)$ and valuation $\sigma : P \rightarrow \{0, 1\}$, σ assigns exactly one of the truth values 0 or 1 to φ

A distinctive feature of classical logic: only two truth values, and one (and only one) of them is always taken by a formula in a valuation

- In some “non-classical” logics (multi/many-valued logics) we may have other truth values, e.g. $\{0, \mathbf{u}, 1\}$, with \mathbf{u} for “unknown”

B. Inconsistency and Trivialization of Reasoning

- If $\Sigma \subseteq L(P)$ is inconsistent, i.e. not satisfiable, then, for every $\varphi \in L(P)$ it holds, $\Sigma \models \varphi$

Why?

It is trivially true that, for every valuation σ , it holds

$$\sigma \models \Sigma \implies \sigma \models \varphi \quad (\text{the antecedent is false})$$

- Everything becomes true at the light of an inconsistent knowledge base
- There are non-classical logics that allow for non-trivial reasoning in the presence of inconsistency (paraconsistent logics)

They can encapsulate inconsistencies, still doing relevant reasoning somewhere else in the context of the knowledge base

C. Monotonicity of Logical Consequence

- Intuitively, if we extend our set of hypothesis, we never lose previous conclusions

Example: We had

$$\{(socratesIsMan \rightarrow socratesIsMortal), socratesIsMan\} \models socratesIsMortal$$

Then it also holds:

$$\{(socratesIsMan \rightarrow socratesIsMortal), socratesIsMan, socratesLovedCrete\} \models socratesIsMortal$$

- More formally, for $\Sigma \subseteq \Sigma'$:

$$\text{If } \Sigma \models \varphi, \text{ then also } \Sigma' \models \varphi$$

(notice that commonsense reasoning is essentially non-monotonic, so different logics are required to model it)

D. Logical Consequence vs. Inconsistency

- A crucial property for automated reasoning

Example: For $\Sigma = \{(p \rightarrow q), p\}$ we had $\Sigma \models q$

Notice that there is no valuation that makes the set of formulas

$$\{(p \rightarrow q), p, \neg q\} \quad (+)$$

true, i.e. that makes every formula in it true

It is impossible for a valuation to make both Σ and $\neg q$ true

Because q is true in every model of Σ

(+) turns out to be **inconsistent** (or unsatisfiable)

p	q	$(p \rightarrow q)$	Σ	$\neg q$	$\Sigma \cup \{\neg q\}$
1	1	1	1	0	0
1	0	0	0	1	0
0	1	1	0	0	0
0	0	1	0	1	0

- In general:

$$\Sigma \models \varphi \quad \text{iff} \quad \Sigma \cup \{\neg\varphi\} \text{ is inconsistent}$$

- Logical consequence (usually what people want to establish) can be reduced to (in)consistency checking

The latter is usually “easier” to implement

E. A Fixed Contradiction

- Σ is inconsistent iff $\Sigma \models F$

Here F is a **fixed** formula that is always false (a contradiction),
e.g. $(p \wedge \neg p)$

- Also a crucial property for automated reasoning

Exercise: Prove the “meta-theorem of deduction” (a theorem about theorems):

$$\Sigma \cup \{\varphi\} \models \psi \Leftrightarrow \Sigma \models (\varphi \rightarrow \psi)$$



Chapter 4: Propositional Symbolic Deductive Reasoning
(and applications)

Leopoldo Bertossi

Carleton University
School of Computer Science
Ottawa, Canada

Towards Automated Reasoning

- We build upon the semantic properties of logical consequence
- A **high-level methodology** for implementing automated reasoning:
 1. One usually wants to prove that $\Sigma \models \varphi$ (*)
 2. Transform this problem into proving that $\Sigma \cup \{\neg\varphi\}$ is inconsistent
 3. Pick a **fixed and “convenient” contradiction** F
(the same all the time)
 4. Try to prove that $\Sigma \cup \{\neg\varphi\} \models F$
Actually, the **symbolic counterpart** of this semantic claim
 5. The automated reasoner can be designed to reach the **fixed target** F , as opposed to a changing target φ in (*)

- We will develop this program in full ...
- There are other several symbolic approaches

Example: Prove that

$$\{(p \vee \neg(q \wedge r)) \rightarrow (s \vee u), \neg r, \neg s\} \models u$$

We have to prove that

$$\{(p \vee \neg(q \wedge r)) \rightarrow (s \vee u), \neg r, \neg s, \neg u\}$$

is inconsistent

First notice that

$$\begin{aligned} ((p \vee \neg(q \wedge r)) \rightarrow (s \vee u)) &\equiv ((p \vee \neg q \vee \neg r) \rightarrow (s \vee u)) \\ &\equiv (\neg(p \vee \neg q \vee \neg r) \vee (s \vee u)) \equiv ((\neg p \wedge q \wedge r) \vee (s \vee u)) \end{aligned}$$

We have to prove the inconsistency of

$$\{((\neg p \wedge q \wedge r) \vee (s \vee u)), \neg r, \neg s, \neg u\}$$

Resolution-Based Automated Reasoning

- We will present in detail one methodology for symbolic deduction for propositional logic: resolution
- “Resolution” is at the basis of many automated reasoning systems and logic programming languages, e.g. Otter, Prolog, ...

It can be applied to formulas that are clauses

- A clause is a disjunction of literals, e.g. $(\neg p \vee r \vee \neg q)$
- **We know:** Each propositional formula is logically equivalent to a formula in conjunctive normal form (CNF), i.e. to a conjunction of clauses

- The transformation can be obtained by using the following **logical equivalences**:

- $\neg\neg\varphi \equiv \varphi$

- $\neg(\varphi \vee \psi) \equiv (\neg\varphi \wedge \neg\psi)$ (De Morgan's law)

- $\neg(\varphi \wedge \psi) \equiv (\neg\varphi \vee \neg\psi)$ (De Morgan's law)

- $\varphi \wedge (\psi \vee \chi) \equiv (\varphi \wedge \psi) \vee (\varphi \wedge \chi)$ (distributivity)

- $\varphi \vee (\psi \wedge \chi) \equiv (\varphi \vee \psi) \wedge (\varphi \vee \chi)$ (distributivity)

- $(\varphi \wedge \psi) \equiv (\psi \wedge \varphi)$ (commutativity, etc.)

- $(\varphi \rightarrow \psi) \equiv (\neg\varphi \vee \psi)$

- $(\varphi \leftrightarrow \psi) \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$

Example: We obtained that

$$(\neg(p \rightarrow \neg q)) \longleftrightarrow ((q \wedge (\neg r \rightarrow s)))$$

is logically equivalent to a formula in CNF:

$$(\neg p \vee \neg q \vee r \vee s) \wedge (\neg q \vee \neg r \vee p) \wedge (\neg q \vee \neg s \vee p)$$

Each of the disjuncts is a **clause**, i.e. a formula that is a disjunction of literals

- That is, a clause is a formula of the form $(l_1 \vee \dots \vee l_k)$, where each l_i is atomic or negation of atomic

For example, $(p \vee \neg p)$, p , $\neg p$, $(\neg p \vee r \vee s)$ are clauses (assuming p, q, r, s are propositional variables)

- Then, every propositional formula is equivalent to a conjunction of clauses (just compute the equivalent formula in CNF)

- Equivalently: Every propositional formula φ can be transformed into a logically equivalent set of clauses \mathcal{C}

$$(\neg(p \rightarrow \neg q)) \longleftrightarrow ((q \wedge (\neg r \rightarrow s)) \longmapsto$$

$$\mathcal{C} := \{(\neg p \vee \neg q \vee r \vee s), (\neg q \vee \neg r \vee p), (\neg q \vee \neg s \vee p)\}$$

- A valuation σ makes φ true iff σ makes all the clauses in \mathcal{C} (simultaneously) true
- A knowledge base written in propositional logic can always be transformed into a logically equivalent knowledge base consisting of clauses only

Until further notice we will be working with clauses only

Example: In page 4, we had to establish the inconsistency of

$$\{((\neg p \wedge q \wedge r) \vee (s \vee u)), \neg r, \neg s, \neg u\}$$

Equivalently, the inconsistency of the set of clauses:

$$\{\neg p \vee s \vee u, q \vee s \vee u, r \vee s \vee u, \neg r, \neg s, \neg u\}$$

Easy: assume σ makes these formulas true

Then, $\sigma(r \vee s \vee u) = 1$ and $\sigma(\neg u) = 1$

Then, $\sigma(r \vee s) = 1$ But $\sigma(\neg s) = 1$

Then, $\sigma(r) = 1$ But $\sigma(\neg r) = 1$ A contradiction!

Do we need to carry σ around?

Basically a formula simplification process: **cancelation of complementary literals, until nothing was left!**

- Notation:

Sometimes we write **clauses as sets of literals**, e.g. instead of $(\neg p \vee \neg q \vee r \vee s)$, we write $\{\neg p, \neg q, r, s\}$

- Beware:

If a clause is written as a set, a valuation makes it true if there is **at least one literal** in it that becomes true (as opposed to all literals in it being true)

E.g. the **clause** $\{\neg p, \neg q, r, s\}$ is made true by the valuation σ with $\sigma(p) = \sigma(q) = \sigma(r) = 1, \sigma(s) = 0$

- A particular clause: **the empty clause** $\{\}$ (or \square)

No valuation makes \square true (there is no literal in it that can be made true), i.e. it is a contradiction! Always false!

In fact: for every valuation σ it holds:

$$\sigma \models \square \iff \text{there is literal } l, \text{ such that } \underline{l \in \square} \text{ and } \sigma \models l$$

The RHS is false

- Can this be the particular contradiction F we wanted?
- It could be reached by a deductive methodology that simplifies clauses, getting rid of literals ...

A Deduction Rule: Resolution

- Below C, C_1, C_2, \dots denote clauses

$$\frac{C_1 \cup \{l\} \quad C_2 \cup \{\bar{l}\}}{(C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})}$$

- $C_1 \cup \{l\}$ and $C_2 \cup \{\bar{l}\}$ are clauses
- l and \bar{l} are **complementary literals** (they are the negation of each other)
- Those two are canceled and a new clause is generated, the **resolvent**
- This is a **deduction rule**: we pass from two clauses to a third clause
- Only one pair of literals is canceled at a time

Example:

$$\frac{\begin{array}{c} \{\neg q, s\} \cup \{\neg q\} \\ \{\neg t, s\} \cup \{q\} \end{array}}{(\{\neg q, s\} \setminus \{\neg q\}) \cup (\{\neg t, s\} \setminus \{q\})}$$

So, the resolvent is $\{s, \neg t\}$

Which is the same as:

$$\frac{\begin{array}{c} \{\cancel{\neg q}, s\} \cup \{\cancel{\neg q}\} \\ \{\neg t, s\} \cup \{\cancel{q}\} \end{array}}{\{s, \neg t\}}$$

Example:

$$\frac{\begin{array}{l} \{q, s\} \cup \{\neg r\} \\ \{\neg t, q\} \cup \{r\} \end{array}}{\{q, s, \neg t\}}$$

or:

$$\frac{\begin{array}{l} \{q, s, \neg r\} \\ \{\neg t, q, r\} \end{array}}{\{q, s, \neg t\}}$$

or

$$\frac{\begin{array}{l} q \vee s \vee \neg r \\ \neg t \vee q \vee r \end{array}}{q \vee s \vee \neg t}$$

Example:

$$\frac{q \vee s \vee p \vee q}{\neg t \vee q \vee \neg p}$$

$$q \vee s \vee \neg t$$

Example:

$$\frac{p}{\neg p}$$

$$\square$$

The ideal case!

Example:

$$\frac{\{\neg p, q\}}{\{p, \neg q\}}$$

$$\square \quad \text{???? NO!}$$

The two clauses together are logically equivalent to the formula $(p \rightarrow q) \wedge (q \rightarrow p)$, ie. to $(p \leftrightarrow q)$, which is not a contradiction

A **correct deduction rule should “preserve truth”**, i.e. the resolvent should be (actually is) a logical consequence of the parent clauses

If \square was a logical consequence of the two clauses, then $p \leftrightarrow q$ would be inconsistent, i.e. a contradiction; but $p \leftrightarrow q$ is satisfiable

We should get the empty clause only if we start from an inconsistent set of clauses

The application of the rule above is wrong; only one pair of complementary literals can be eliminated, obtaining each of

$$\frac{\begin{array}{l} \{\neg p, q\} \\ \{p, \neg q\} \end{array}}{\{p, \neg p\}}$$

$$\frac{\begin{array}{l} \{\neg p, q\} \\ \{p, \neg q\} \end{array}}{\{q, \neg q\}}$$

Example: $p \vee \neg p??$

$$\frac{\begin{array}{c} \{\neg p, p\} \\ \{p, \neg p\} \end{array}}{\{p, \neg p\}}$$

Nothing new ...

Truth is preserved, because the clause $\{p, \neg p\}$ is a tautology

The parent clauses are not needed

Example: Prove that:

$$\{((p \rightarrow q) \rightarrow r), (r \rightarrow p)\} \models (p \wedge (q \rightarrow r))$$

Alternatively we can prove that:

$$\{((p \rightarrow q) \rightarrow r), (r \rightarrow p), \neg(p \wedge (q \rightarrow r))\} \quad (*)$$

is inconsistent

Apply resolution: Start by transforming $(*)$ into an equivalent set of clauses;
i.e. go through the CNF:

- $((p \rightarrow q) \rightarrow r) \equiv \neg(\neg p \vee q) \vee r \equiv (p \wedge \neg q) \vee r \equiv (p \vee r) \wedge (\neg q \vee r)$
- $(r \rightarrow p) \equiv (\neg r \vee p)$

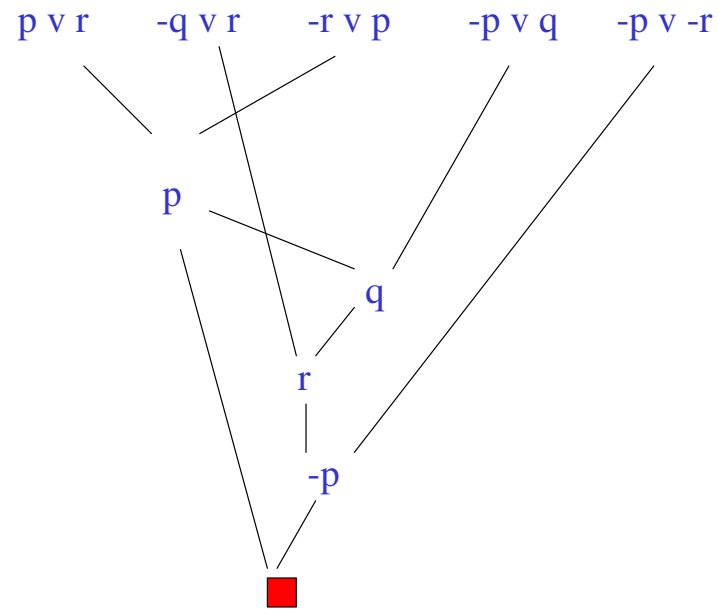
- $$\begin{aligned} \neg(p \wedge (q \rightarrow r)) &\equiv (\neg p \vee \neg(\neg q \vee r)) \equiv (\neg p \vee (q \wedge \neg r)) \\ &\equiv (\neg p \vee q) \wedge (\neg p \vee \neg r) \end{aligned}$$

We get the following set of clauses:

$$\mathcal{C} = \{p \vee r, \neg q \vee r, \neg r \vee p, \neg p \vee q, \neg p \vee \neg r\}$$

which is inconsistent iff $(*)$ is inconsistent

Now we try to establish the inconsistency of \mathcal{C}



The search tree shows a successful derivation of the empty clause: a **resolution-based refutation tree**

We have an implicit search tree

There may be different ways to reach the empty clause

A successful proof can be presented as a **refutation** (a formal proof that reaches a contradiction)

- (1) $(p \vee r)$ (hypothesis)
- (2) $(\neg q \vee r)$ (hypothesis)
- (3) $(\neg r \vee p)$ (hypothesis)
- (4) $(\neg p \vee q)$ (hypothesis)
- (5) $(\neg p \vee \neg r)$ (hypothesis)
- (6) p (resolution with (1) and (3))
- (7) q (resolution with (4) and (6))
- (8) r (resolution with (2) and (7))
- (9) $\neg p$ (resolution with (5) and (8))
- (10) \square (resolution with (6) and (9))

So, a **resolution-based refutation** from a set of clauses \mathcal{C} is

- a finite sequence of clauses,
- ending in the empty clause \square , such that
- every clause in it belongs to \mathcal{C} or is obtained as a resolvent from two previous clauses in the sequence

Useful information can be obtained from a refutation, e.g. the clauses that participate in an inconsistency (or do not participate)

From this point of view, it may be useful to compute several (or all) refutations

That is, the whole search space should be explored

Some systems give more flexibility than others ...

Exercise: Use Prover9 to play with the example in page 11

Example: Consider the following statements

“If Superman were able and willing to prevent evil, he would do so. If Superman were unable to prevent evil, then he would be impotent; and if he were unwilling to prevent evil, he would be malevolent. If Superman exists, he is neither evil nor impotent. Superman does not prevent evil.”

We want represent this body of knowledge in a propositional language; and prove from it that Superman does not exist

As before, we introduce natural propositional variables

a : Superman is able to prevent evil

w : Superman is willing to prevent evil

i : Superman is impotent

m : Superman is malevolent

p : Superman prevents evil

e : Superman exists

The knowledge base written in a propositional language:

$$a \wedge w \rightarrow p$$

$$(\neg a \rightarrow i) \wedge (\neg w \rightarrow m)$$

$$\neg p$$

$$e \rightarrow \neg i \wedge \neg m$$

In equivalent clausal form:

1. $\neg a \vee \neg w \vee p$
2. $a \vee i$
3. $w \vee m$
4. $\neg p$
5. $\neg e \vee \neg i$
6. $\neg e \vee \neg m$

If we want to prove $\neg e$, we add its negation to this list of clauses:

7. e

Let us try to obtain a contradiction, which shows that the set of formulas 1.-7. is inconsistent

Resolution of 7. and 6.:

8. $\neg m$

Resolution of 7. and 5.:

9. $\neg i$

Resolution of 8. and 3.:

10. w

Resolution of 9. and 2.:

11. a

Resolution of 10. and 1.:

12. $\neg a \vee p$

Resolution of 12. and 11.:

13. p

Resolution of 13. and 4.:

14. \square

A clause (original or intermediate) can be used several times

Properties of Resolution-Based Refutations

Is all this symbolic processing of formulas justified?

1. The **resolution rule is sound**: The resolvent is a logical consequence of its parent clauses, i.e. if a clause C is obtained by applying resolution to clauses C_1, C_2 , then $\{C_1, C_2\} \models C$

In other words, the resolution rule preserves truth

2. **Resolution-based refutations are sound**: if the empty clause can be obtained by a finite sequence of applications of the resolution rule starting from a set of clauses \mathcal{C} , then \mathcal{C} is inconsistent

We can trust resolution-based refutations, they do not sanction as inconsistent anything that is not inconsistent

3. The methodology of resolution-based refutations is complete:

If a set of clauses \mathcal{C} is inconsistent, then there is a refutation from \mathcal{C} based on resolution

If the set of clauses is inconsistent, this can be established using resolution

The deductive system is powerful; all inconsistencies (that were defined in semantic terms) can be reached by resolution (a syntactic, symbolic, formal methodology)

4. Inconsistency, a crucial semantic concept, can be captured at a purely symbolic level!

Resolution based refutations have been implemented in several systems, e.g. Otter, Prover9, Prolog

Theorem: (completeness) If \mathcal{C} is an inconsistent set of clauses, then there is a refutation (ending in \square) from \mathcal{C}

Finally, we immediately obtain, putting all together:

Theorem: (correctness of resolution) For \mathcal{C} a set of propositional clauses

\mathcal{C} is inconsistent iff there is a resolution-based refutation from \mathcal{C}

Notice: The completeness theorem does not say **how** to find the existing refutation

It just says there is a refutation somewhere there ...

Finding it is a different problem: **We need additional search techniques, heuristics, ...**

Implementations have built-in heuristics; some of them can be explicitly specified, e.g. Otter (also weights can be assigned to clauses)

The Proofs: (next 9 slides are non-mandatory reading)

Theorem: The resolution rule is sound, i.e. the resolvent is logical consequence of its parent clauses:

Proof: Assume the application of the rule is

$$\frac{C_1 \cup \{l\} \quad C_2 \cup \{\bar{l}\}}{(C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})}$$

that $\sigma \models C_1 \cup \{l\}$ and $\sigma \models C_2 \cup \{\bar{l}\}$

To prove: $\sigma \models (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})$

This is easy: consider all the different alternatives for σ to satisfy the parent clauses

In all the cases, σ also satisfies the resolvent

In fact, there are different cases:

- $\sigma \models (C_1 \cup \{l\})$ because $\sigma \models l'$, with $l' \in C_1, l' \neq l$
In this case, $l' \in (C_1 \setminus \{l\}) \cup (C_2 \setminus \{\bar{l}\})$, and then σ satisfies the resolvent
- $\sigma \models (C_1 \cup \{l\})$ because $\sigma \models l$
In this case, $\sigma \not\models \bar{l}$, and there must be $l' \in C_2$, such that $\sigma \models l'$
That l' is in the resolvent
So, σ satisfies the resolvent

Theorem: (soundness) Resolution-based refutations are sound

I.e. if \square is obtained by a finite chain of applications of resolution starting from a set of clauses \mathcal{C} , then \mathcal{C} is inconsistent

Proof: It is good enough to prove that in any finite sequence of clauses C_1, \dots, C_n obtained using clauses in \mathcal{C} and the resolution rule applied to clauses that come before in the sequence, it holds $\mathcal{C} \models C_n$

This can be proved by induction on n (this is arithmetical induction, not on formulas), using the previous theorem and the transitivity of logical consequence

In particular, for a finite sequence ending in \square , it will hold $\mathcal{C} \models \square$; and this possible only when \mathcal{C} is always false, i.e. inconsistent (otherwise a valuation σ making \mathcal{C} true, would be forced to make \square true, which is not possible)

Theorem: (completeness) If \mathcal{C} is an inconsistent set of clauses, then there is a refutation (ending in \square) from \mathcal{C}

Proof: We will assume that the unsatisfiable set of clauses \mathcal{C} is finite¹

To prove: There is a resolution-based refutation tree (i.e. that has a \square as the root)

We can try to *construct* it

How?

Inductively ... I.e. by induction

Induction on what?

By induction on the (finite) number of propositional variables in \mathcal{C}

Since \mathcal{C} is finite, a finite number of propositional variables appear in it

¹It is possible to prove the theorem in general. Actually, the so-called “compactness theorem” of classical logic tells us that an arbitrary set of formulas is inconsistent iff there is a finite subset of it that is inconsistent.

More precisely, we prove that

“For every n , if \mathcal{C} is an inconsistent set of clauses with n propositional variables, then there is a resolution-based refutation tree from \mathcal{C} ”

- Base case: $n = 0$

The only possibility is $\mathcal{C} = \{\square\}$; and we are done

- Inductive step: Assume \mathcal{C} , inconsistent, has $n + 1$ propositional variables

We have to reduce this case to the case for n , to apply the inductive hypothesis
How?

We have to get rid of one propositional variable

Produce one or more sets of clauses, with n propositional variables; and they should be inconsistent

By IH there are refutation trees for them

Combine them to produce a tree for \mathcal{C}

Let's see ...

We pick up one of the variables in \mathcal{C} , say the literal u , and define

$$\mathcal{C}(u) := \{C \setminus \{\bar{u}\} \mid C \in \mathcal{C} \text{ and } u \notin C\}$$

$$\mathcal{C}(\bar{u}) := \{C \setminus \{u\} \mid C \in \mathcal{C} \text{ and } \bar{u} \notin C\}$$

u, \bar{u} do not appear in these two sets of clauses; so they have n propositional variables (we can forget clauses of \mathcal{C} that contain both u, \bar{u} , they are tautological and do not contribute to inconsistencies)

We can apply the induction hypothesis, because it is possible to prove that $\mathcal{C}(u), \mathcal{C}(\bar{u})$ are inconsistent when \mathcal{C} is inconsistent (do it!)²

²We can safely assume that the inconsistency of \mathcal{C} is not due to the presence of the two clauses $\{u\}, \{\bar{u}\}$; this case give immediately a refutation tree

Etc. Better show the rest of the proof with an example ...

Example: (for the inductive step)

$$\mathcal{C} = \{p \vee r, \neg q \vee r, \neg r \vee p, \neg p \vee q, \neg p \vee \neg r\}$$

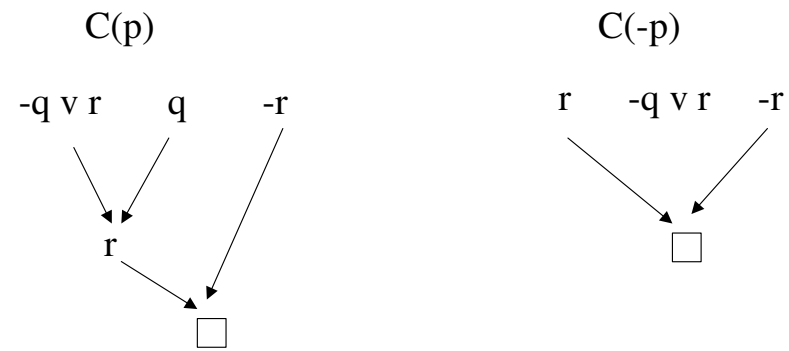
(inconsistent, check it)

Choose, say p :

$$\mathcal{C}(p) = \{\neg q \vee r, q, \neg r\} \quad \mathcal{C}(\neg p) = \{r, \neg q \vee r, \neg r\}$$

(no $p, \neg p$ now, and both still inconsistent)

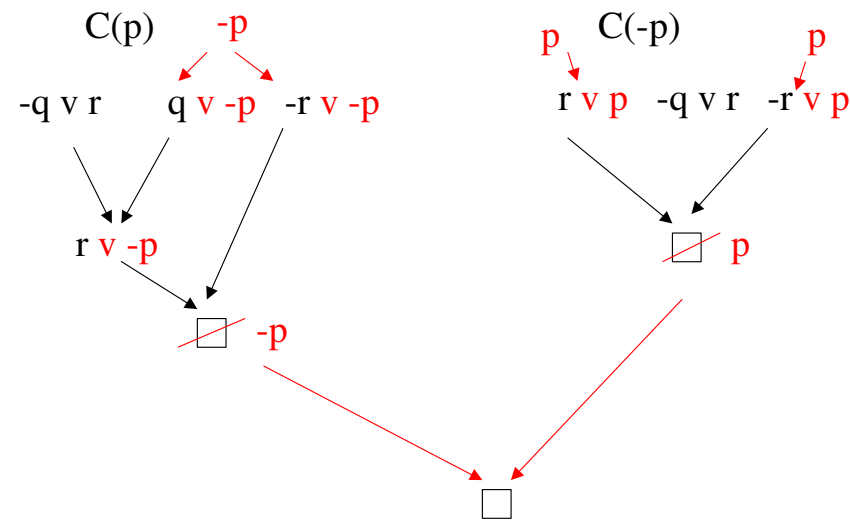
There are refutation trees for both sets:



We have to combine them somehow to obtain a refutation tree for \mathcal{C}

Furthermore, the clauses in both trees do not belong to the original \mathcal{C}

We have to reinsert $p, \neg p$ at the top and propagate them downwards ...



We have seen how to use induction in a constructive way, to obtain a refutation tree

Actually, we have an inductive/recursive procedure for constructing refutation trees (when they exist, i.e. when the given set of clauses is inconsistent)

Induction can be a powerful technique for designing algorithms, cf. Udi Manber, "Using Induction to Design Algorithms". Communications of the ACM, 1988, 31(11):1300-1313.



Carleton
UNIVERSITY

Chapter 5: Propositional Logic Programming

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

Some History and Motivation

Automated theorem proving had been around for some time, but there were efficiency problems

1965: J.A. Robinson introduces the **resolution rule for first-order predicate logic**, as a combination of propositional resolution and **unification**; more efficient and a breakthrough ...

1971 - 1972: A. Colmerauer implements first PROLOG (PROgramming in LOGic), that uses Robinson's resolution and backtracking as search mechanism

R. Kowalski proposes the idea of using logic (logic programming) to specify or describe a domain; and computing from the description: the **declarative programming paradigm**

- C.f. R.Kowalski. “Logic for Problem Solving”. North-Holland, 1981 (new edition in 2014)

A computational problem is solved by:

- Describing a domain in logic (knowledge specification)
- Using an underlying, general control mechanism to handle formulas and inferences

In consequence, **the programmer’s task becomes the description of a domain; and the actual computation is left to an internal control mechanism**

Kowalski's Equation:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

- “Logic” refers to the domain description; and
- “Control”, to the mechanisms for selecting and combining formulas; and performing inferences

The equation captures the essence of the declarative programming paradigm

If the control mechanism is fixed and implemented, programming becomes using a logic-based language for domain descriptions and expression of goals, i.e. of **what** we want computed

(as opposed to imperative programming, where we need to say **how** to compute things)

(Early, classical) logic programming is usually based on:

- A **particular class of clauses** as logical formulas for domain descriptions (to write programs)
- An inference mechanism based on **resolution**
- **Specific and fixed search strategy and control mechanism** for handling clauses

PROLOG is only ONE particular -and early- way to capture and implement this idea (with good things, but some important limitations)

(unless explicitly stated, in the following **we are not** talking about PROLOG in particular)

Definite Propositional Logic Programming

A logic program consists of **Horn clauses** (HC), a particular syntactic subclass of clauses

A HC is a clause of the form $l_1 \vee l_2 \vee \dots \vee l_n$ where **at most** one of the **literals** l_i is positive

Example: $\neg q \vee \neg r \vee p$ is Horn ($\neg r \vee p \vee s$ is not)

If the only positive literal is, say l_1 , the clause can be written as:

$l_1 \leftarrow \bar{l}_2 \wedge \dots \wedge \bar{l}_n$ (the \bar{l}_j are the complementary literals)

Example: $p \leftarrow q \wedge r$

The conjunctions on the RHS -in the **body**- are written as comas, and contain only positive literals; the LHS is the **head** of the clause

Example: $p \leftarrow q, r$

Examples:

- $p \leftarrow q, r$ equiv. to $\neg q \vee \neg r \vee p$
- $p \leftarrow p, s$ a **recursive** clause
- $p \leftarrow q, s$ clauses with mutually **recursive** definitions
 $q \leftarrow p, t$
- $p (\leftarrow)$ a **fact** (inconditional)
- $\leftarrow q$ a clause with empty head, equiv. to $\neg q$
- $\leftarrow q, r$ equiv. to $\neg(q \wedge r)$
- \leftarrow the empty clause \square

Resolution with HCs: As before

$$\begin{array}{l} p \leftarrow q, \underline{r}, s \\ r \leftarrow t, d \\ \hline p \leftarrow q, \underline{t}, d, s \end{array}$$

We could also get (the logically equivalent):

$$p \leftarrow t, q, d, s,$$

but in some systems, e.g. PROLOG, keeping the order is crucial

The representation of a clause starting from the head is inspired by a **procedural interpretation** of clauses:

$$p \leftarrow q, r, s$$

indicates that the procedure p is defined in terms of the sub-procedures q, r, s

The rule is taken from the head (the invoked procedure), and in order to satisfy or run it, one goes to the body to see which sub-procedures have to be satisfied or run in their turn

In the example, to satisfy (run) r , one has to find a clause that **defines** it, i.e. a clause with head r

If the goal (to satisfy) is p , then the subgoals q, r, s are generated (**backward propagation**, from head to body)

A (definite) logic program P consists of a (usually ordered) list of **positive** Horn clauses, i.e. with non-empty head

Queries or goals appear as **negative** Horn clauses, i.e. with empty head

Goals are added to the program, to trigger computation (they are not part of it)

Not new: If we want to establish (prove, query) a positive atom q , we add $\neg q$ to the KB, in this case P , as $\leftarrow q$

Next, one tries to derive the empty clause

$$P \stackrel{??}{\models} q \quad \longmapsto \quad P \cup \{\leftarrow q\} \stackrel{??}{\rightsquigarrow}_{\text{refutation}} \square$$

We are not interested in LP as a general programming language (as Prolog is), but as a declarative formalism for KR and reasoning

Example: Program P_1

$p \leftarrow q, s$

$s \leftarrow t$

t

q

A proof of p : The goal $\leftarrow p$ is added to P_1 ; more precisely, one starts doing resolutions with it:

1. $\leftarrow p$ (negation of p)
2. $p \leftarrow q, s$ (definition of p)
3. $\leftarrow q, s$ (resolution with 1. and 2.)
4. $s \leftarrow t$ (definition of s)
5. $\leftarrow q, t$ (resolution with 3. and 4.)
6. $t \leftarrow$ (definition of t)
7. $\leftarrow q$ (resolution with 5. and 6.)
8. $q \leftarrow$ (definition of q)
9. \leftarrow (resolution with 7. and 8.)

We can conclude $P_1 \models p$ (classically)

What if there was in the program also the clause $p \leftarrow u$?

That is, we have the two clauses: $p \leftarrow q, s$ and $p \leftarrow u$

The two together are logically equivalent to: $p \leftarrow ((q \wedge s) \vee u)$

This is not a clause, not allowed in the program, but shows that p is defined by an implicit disjunction

Each of the clauses with p in the head is a partial definition of p

Which one would be choose?

Another point: Which of the generated subgoals has to be satisfied first?

E.g. in 3.?

The underlying control mechanism of an implementation determines all this

PROLOG is a particular implementation, with a fixed control mechanism

Programs of this kind are called **definite**, because the clauses in them have one single atom in the head, then they represent **definite knowledge**

As opposed to indefinite (or uncertain) knowledge that would be obtained by using, e.g. **disjunctive clauses**, e.g. $p \vee q \leftarrow r, s$

Propositional PROLOG

PROLOG Strategy: Depth-first search with backtracking, top-down and backwards reasoning

Clauses are ordered as they appear in the program

Subgoals in a clause body are ordered from left to right

1. Start with the goal $\leftarrow p$ (top-down refutation)
2. Choose the first clause with head p
3. Try to reach the empty clause
4. If not possible, choose the next applicable clause
5. The subgoals are being proved from left to right
6. Subgoals at the right of the one being proved are postponed while the current subgoal generates its own sub-subgoals, etc.

Example: Program P_2

$p :- q, s.$

$p :- u.$

$s :- t.$

$s :- d.$

$q :- v.$

$d :- w.$ (“:-” stands for “ \leftarrow ” in Prolog, and other systems)

$v.$

$u.$

Proof of p ? Is p true? (a query posed as “:-? $p.$ ”)

Prolog answers: **Yes!**, because there is a successful refutation (below)

Then, $P_2 \models p$

PROLOG’s strategy is compatible, more precisely, **sound** wrt. classical logic

We have both:

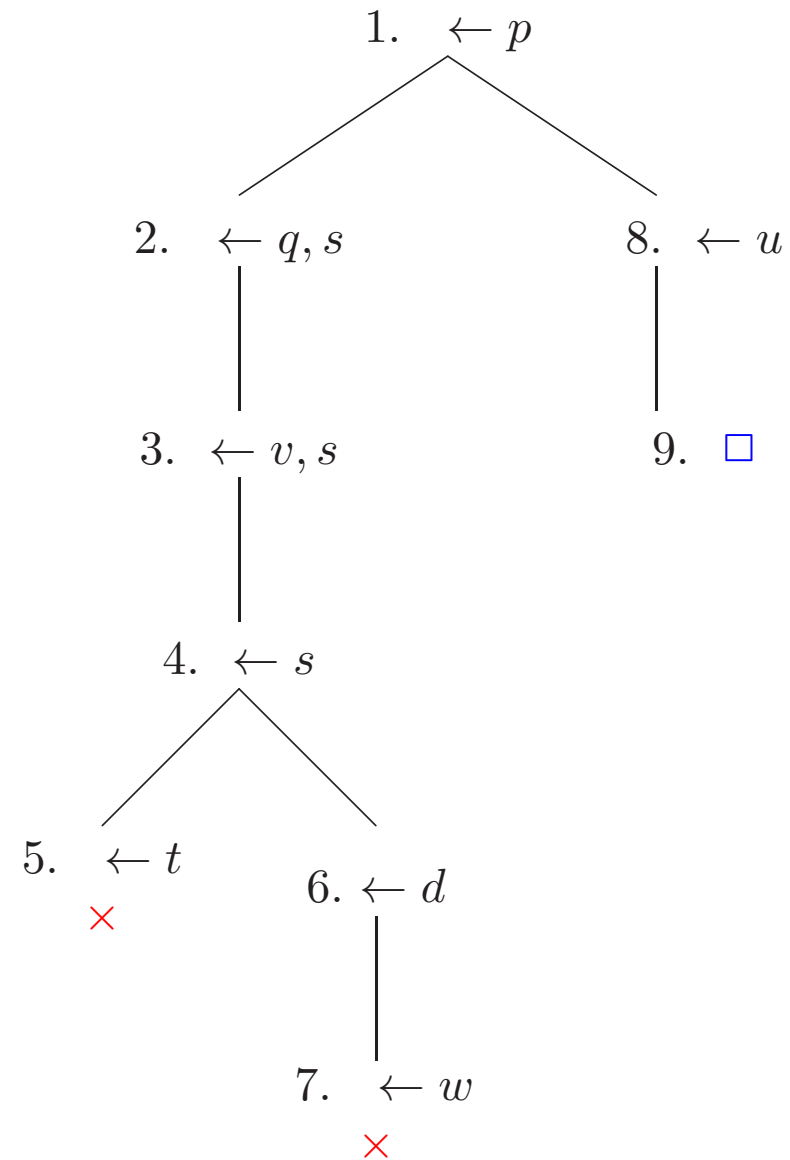
$P_2 \models p$ (classically)

and $P_2 \vdash_{Prolog} p$

(i.e. syntactically provable
by Prolog)

It is common to use the symbol \vdash to
denote a form of inference based on
purely symbolic, deductive methods

Here, the \vdash_{Prolog} emphasizes that it is
Prolog's deductive method



1. $\leftarrow p$
2. $p \leftarrow q, s$
3. $\leftarrow q, s$
4. $q \leftarrow v$
5. $\leftarrow v, s$
6. v
7. $\leftarrow s$
8. $s \leftarrow t$
9. $\leftarrow t$ (\times) (backtracking to 7.)
10. $s \leftarrow d$
11. $\leftarrow d$
12. $d \leftarrow w$
13. $\leftarrow w$ (\times) (backtracking to 2.)
14. $p \leftarrow u$
15. $\leftarrow u$
16. u

17. \leftarrow

A clean refutation, as those returned by Otter, would contain that associated to the right-most subtree

Notice that a breadth-first development/exploration of the (implicit) tree would have returned the answer sooner

Prolog uses depth-first search though

Example: PROLOG's strategy may lead to infinite "proofs"

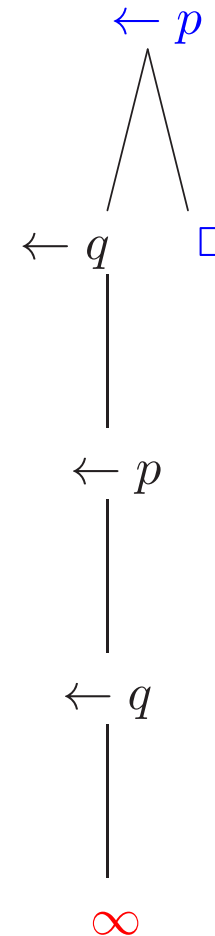
Even in cases where, with other strategies, we could have success

P_3 : $p \leftarrow q$
 $q \leftarrow p$
 $p \leftarrow$

The empty clause on the RHS will never be reached

Here, $P_3 \models p$,
but $P_3 \not\vdash_{Prolog} p$

That is, **PROLOG is incomplete wrt resolution-based refutations**, due to the particular and fixed search strategy



Example: Sometimes we may **finitely fail**

Program P_4 :

$p := q, s.$

$q := u.$

$s := t.$

$s := d.$

$q := v.$

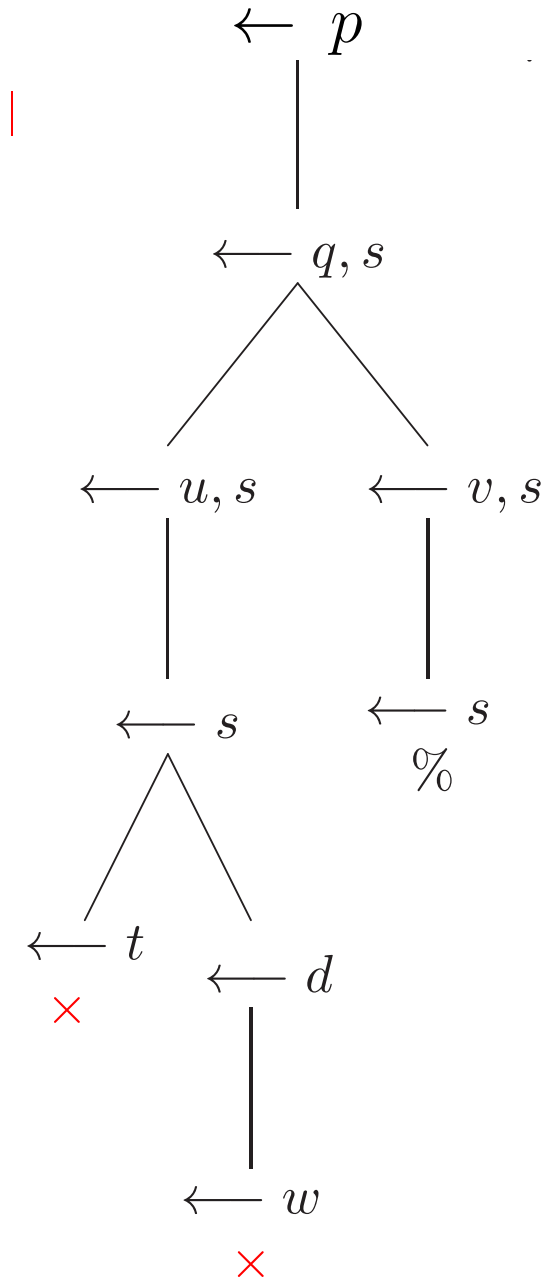
$d := w.$

$v.$

$u.$

All the possibilities are explored and **we fail** (we do not reach \square) in a **finite and exhaustive** search process

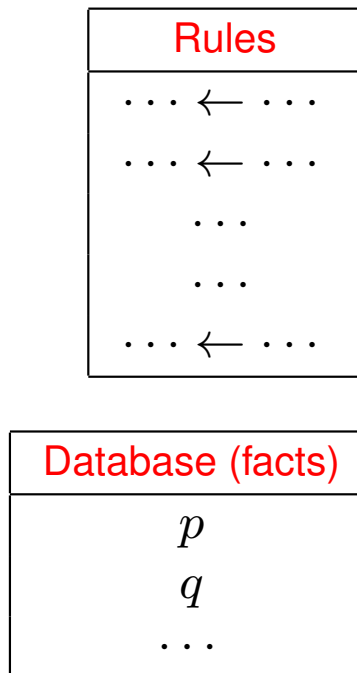
Here we can safely say: $P_4 \not\equiv p$



Non-Monotonic Negation and Definite Programs

So far, LPs are positive (no negative literals in head or body, and no clauses with empty head)

So, a KB represented in LP is as follows



Usually the rules are in a layer on top of a (possibly real relational) database storing the facts

In a definite program, there is no explicit negation

- Since there is no negative literal in the head, **it is not possible to prove a negative literal** from such a positive LP **with the reasoning mechanism we have so far**
- In particular, it is not possible to obtain from such an LP both a literal and its complement, i.e. no inconsistencies are generated
- More generally, it can be easily shown that a **definite LP is always consistent**

(The truth assignment that makes every variable true makes the whole LP true)
- Inconsistencies can only appear when the goal, say $\leftarrow q$, is added to the program, but q is still positive; so, if we prove q , it is still a positive conclusion

If we want to prove $\neg p$ (p an atom) from a definite LP P , we may think of adding $\neg\neg p$ to P

We get $P' = P \cup \{p \leftarrow\}$, which is a definite program again, and then, consistent, and then no refutations leading to \square are possible ...

Example: Consider program P (clearly consistent)

$p \leftarrow q$

$q \leftarrow r$

s

Want to obtain $\neg p$: adding its negation $\neg\neg p$ to P gives program P'

$p \leftarrow q$

$q \leftarrow r$

s

p It is consistent: make every atom true

So, $\neg p$ is not a consequence of P

If we want to obtain negative information, we need to extend our representation language and reasoning mechanism ...

How???

Actually, not an unfamiliar situation

Can we prove (obtain) **negative information from a relational database?**

E.g. **Is true that there is no direct flight from Ottawa to Hong-Kong?**

Answer: Yes! No direct flight because we **search inside** trying to find if there is a flight (the corresponding positive query) and we **do not find** any

A **procedural support** for the answer; the **Closed World Assumption (CWA)** at work

Example: Query: $Q(x) : \exists y(R(x, y) \wedge \neg S(y))$

R	A	B
	a	b
	e	c

S	B
	c

Answer: $x = a$ Because b not found in S !

Not this answer with the DBs open:

R	A	B
	a	b
	e	c

S	B
	c

Now we have a definite LP instead of a RDB

Now the mechanism is **not simply look inside** (query) the DB, but **try to prove** the corresponding positive fact (using resolution-based refutations)

If we **finitely and exhaustively fail**, we declare the negative statement true

In order to obtain negative knowledge:

- We need an **extended mechanism**
- **Not based on classical logic**, we need to go beyond ...
- **Negation as Failure** (NAF): Can be seen as an extension of the CWA, involving exhaustive reasoning instead of exhaustive direct inspection

Given a definite LP P and a **negative atom** $\neg p$, we want to obtain $\neg p$ as a (non-classical) consequence of P

The query $\leftarrow \neg p$ gets answer *true* (Yes) iff, when trying to prove $\leftarrow p$, we **finitely fail** (using resolution-based refutations)

That is, **the search space is completely, exhaustively and finitely explored, and no success (\square) is reached** (as in page 18)

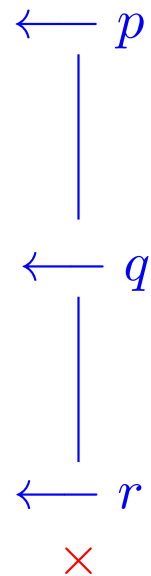
In the case of, e.g., PROLOG the fixed search strategy we saw is also part of the procedure

Example: Program P_5 :

$$\begin{array}{l} p \leftarrow q \\ q \leftarrow r \end{array}$$

$:-? p$ **NO!** (obtained as before, finite failure, see below)

$:-? \neg p$ **YES!** (this is new)



We can say, $\neg p$ is a consequence of P_5 with resolution-based refutations **plus** NAF

Denoted: $P_5 \vdash_{NAF} \neg p$ However

$\neg p$ is not a classical logical consequence of P_5 : $P_5 \not\models \neg p$

We are departing from classical logic ...

Example: From program P_4 we can get $\neg p$ using NAF:

$$P_4 \vdash_{NAF} \neg p$$

Because we finitely fail trying to establish p ...

Example: $P_2 \not\vdash_{NAF} \neg p$

Because, p can be proved from P_2 using resolution-based refutations

Example: $P_3 \not\vdash_{NAF} \neg p$

Because, p can be proved from P_3 using resolution-based refutations

The fact that PROLOG fails to establish p does not count; it is **not a finite failure**: $P_3 \not\vdash_{Prolog,NAF} \neg p$

Remarks:

- In classical terms: $P_5 \not\models p$ and $P_5 \not\models \neg p$
I.e. p (and $\neg p$) is **undetermined** wrt P_5
However, P_5 as a LP with NAF, sanctions $\neg p$ as true
That is, in the presence of **incomplete knowledge about p** (and $\neg p$), $\neg p$ is sanctioned as true; a sort of **commonsense assumption or conclusion**
A sort of application of a CWA (but now involving reasoning)
Obviously this is not classical logic, nor a classical consequence ...
- **We have a negation that is different from classical negation!!**

- Classical negation is **strong**: only the negation of something false becomes true

But, classically, p is not false wrt P_5 , only undetermined

We have a **weak negation** (less demanding condition)

- Better change notation, usually **weak negation is denoted by** *not*
- The new notion of logical consequence \vdash_{NAF} is **non-monotonic**: If we add new information into the KB (the LP), then previous conclusions may have to be retracted

Similar to the application of CWA in RDBs, that is also non-monotonic

Example: Program $P'_4 := P_4 \cup \{w \leftarrow\}$:

Before: $P_4 \vdash_{NAF} \text{not } p$

Now: $P'_4 \not\vdash_{NAF} \text{not } p$

NAF is a **procedural negation**

So far it only appears in the “user goals” (i.e. top-goals), i.e. in $:- \text{not } p$

Why not to use it in the goals in bodies (sub-goals)?

We add NAF in the bodies of program rules, increasing the expressive power of the LP language

Evaluation mechanism?

As before ...

If in the execution from P we reach a negative goal of the form $not\ p$ in a body, say

$$\longleftarrow \underline{not\ p}, q, r, not\ s$$

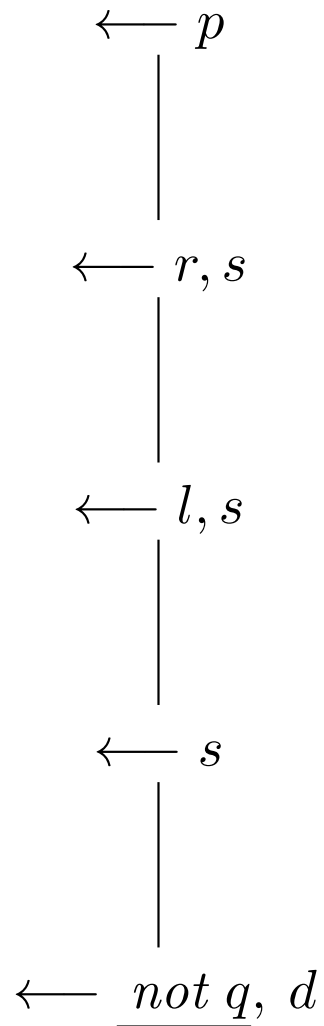
- The processing of $not\ p$ is suspended
- One tries to prove p from P
 - If one finitely fails with p , the subgoal $not\ p$ succeeds, and we move to

$$\longleftarrow \underline{q}, r, not\ s$$

- Otherwise, if one succeeds proving p , then the subgoal $not\ p$ fails, and the whole goal (that is a conjunction)

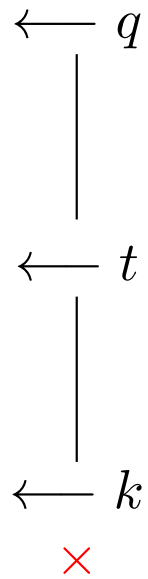
$$\longleftarrow not\ p, q, r, not\ s \qquad \text{also fails}$$

Example: P_6 :



$p \leftarrow r, s$
 $s \leftarrow \text{not } q, d$
 $r \leftarrow l$
 $q \leftarrow t$
 $t \leftarrow k$
 l
 d
 $:-? p$

One goes off this proof, leaving it pending,
trying to establish q



(finite failure with q)

We return to the main proof

Since the negative goal succeeds:

← d
□ *Yes!*

Some Commonsense Knowledge using LP

We can use LPs with weak negation to represent knowledge involving **defaults** **rules** and do non-monotonic reasoning

Example: We have “*Normally birds fly*” and “*Ostriches and penguins are abnormal*” and “*Tweety is a bird*”

Does Tweety fly?

It should, given that there is no evidence that it is abnormal

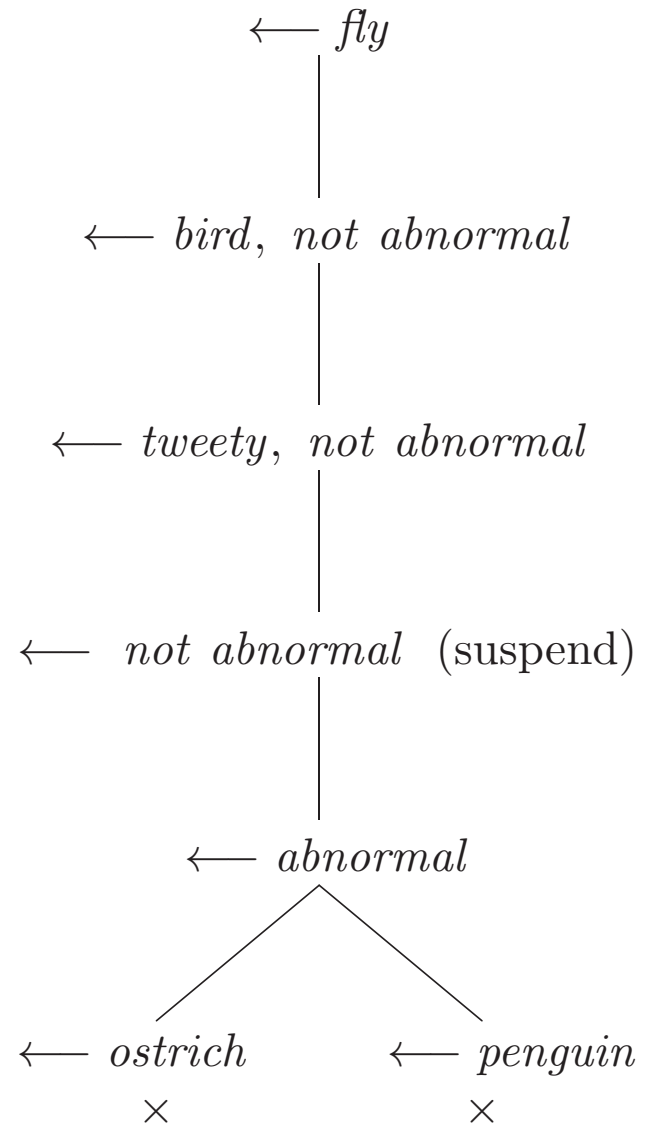
An ornithological LP program P_O :

$$\begin{aligned} fly &\leftarrow bird, \text{ not } abnormal. \\ abnormal &\leftarrow ostrich. \\ abnormal &\leftarrow penguin. \\ bird &\leftarrow tweety. \\ tweety. \end{aligned}$$

The first rule represents a default rule, a commonsense rule

$:-? fly$ *Yes!*

A **non-monotonic conclusion**: If we add $ostrich \leftarrow tweety$, we no longer obtain fly , and previous conclusion has to be retracted



Some Propositional Prolog

- Beware: Logic programming (LP) cannot be identified with Prolog

Prolog is just one approach and (families of) implementation(s) of LP

- The addition of negation to Prolog (and other things) takes us outside classical logic

“Prolog’s logic” becomes, in particular, non-monotonic

- Full Prolog is Turing-complete, i.e. it is a universal programming language

Actually, it is possible to program (simulate) a Turing machine using Prolog

- We are interested in Prolog as a knowledge representation paradigm

Staying “as close as possible” to classical logic ...



Propositional Prolog: Dynamic Predicates



This refers to Sicstus Prolog, but is rather general

Certain predicates can be changed during the execution, which is useful for testing, and exploring alternatives

An Initial Example:

```
:- dynamic q/0.
p:-q.
q:-r.

% Example of Execution

| ?- asserta(q).
yes
| ?- asserta(r).
yes
| ?- asserta(p).
{PERMISSION ERROR: asserta(user:p) - cannot assert static user:p/0}
    % p has to be dynamic or undefined (i.e. in no head)
```

```
| ?- listing(q).  
q.                % q placed at the beginning  
q :- r.  
yes  
  
| ?- listing(r).  
r.  
yes  
  
| ?- p.  
yes  
| ?- retract(q).  
yes  
| ?- assertz(q).  
yes  
| ?- listing(q).  
q :- r.  
q.                % q placed at the end  
yes  
  
| ?- retract(r).  
yes
```

Some problems with Asserta/1 and Assertz/1

```
:- dynamic p/0,q/0.
```

```
p:-q.
```

```
q:-p.
```

```
| ?- p.
```

```
1 1 Call: p ?
```

```
2 2 Call: q ?
```

```
3 3 Call: p ?
```

```
4 4 Call: q ?
```

```
5 5 Call: p ?
```

```
6 6 Call: q ?
```

```
7 7 Call: p ?
```

```
8 8 Call: q ? % an infinite loop
```

```
9 9 Call: p ? a
```

```
{Execution aborted}
```

```
| ?- assertz(q).
```

```
yes
```

```
| ?- p.
```

```
1 1 Call: p ?
```

```
2 2 Call: q ?
```

```
3 3 Call: p ?
```

```
4 4 Call: q ?
```

```
5 5 Call: p ?
6 6 Call: q ?
7 7 Call: p ?
8 8 Call: q ?
9 9 Call: p ? a
{Execution aborted}
```

```
| ?- listing(q).
q :- p.
q.
yes
```

```
| ?- retract(q).
yes
```

```
| ?- asserta(q).
yes
```

```
| ?- p.
1 1 Call: p ?
2 2 Call: q ?
2 2 Exit: q ?
1 1 Exit: p ?
```

```
yes
```

Another Example:

```
/* Begin of Prolog program, stored in file ejel.pl */

:- dynamic a/0,b/0,p/0,q/0,r/0,s/0.

p:-q,r,s.
q:-a.
r:-b.
s:-a,b.
a.
b.
/* End of Prolog program */

/* loading the program */

SICStus 3 #3: Thu Nov 30 11:19:40
| ?- [ejel].
{consulting /tmp_mnt/home/ing/dcc/invest/marenas/latex/tmp/ejel.pl...}
{/tmp_mnt/home/ing/dcc/invest/marenas/latex/tmp/ejel.pl consulted,
 10 msec 9808 bytes}

yes

/* end of load */

/* debugging */
```

```
| ?- trace.  
{The debugger will first creep -- showing everything (trace)}
```

```
yes
```

```
{trace}
```

```
| ?- p.
```

```
1 1 Call: p ?  
2 2 Call: q ?  
3 3 Call: a ?  
3 3 Exit: a ?  
2 2 Exit: q ?  
4 2 Call: r ?  
5 3 Call: b ?  
5 3 Exit: b ?  
4 2 Exit: r ?  
6 2 Call: s ?  
7 3 Call: a ?  
7 3 Exit: a ?  
8 3 Call: b ?  
8 3 Exit: b ?  
6 2 Exit: s ?  
1 1 Exit: p ?
```

```
yes
```

```
{trace}
```

```
/* end of debugging */
```

:- dynamic p/0,q/0,r/0,s/0,t/0,w/0.

p:-q,r.

q:-s,t.

q:-w.

r:-t.

r:-s,w.

s.

w.

| ?- p.

1 1 Call: p ?

2 2 Call: q ?

3 3 Call: s ?

3 3 Exit: s ?

4 3 Call: t ?

4 3 Fail: t ?

3 3 Redo: s ?

3 3 Fail: s ?

3 3 Call: w ?

3 3 Exit: w ?

2 2 Exit: q ?

4 2 Call: r ?

5 3 Call: t ?

5 3 Fail: t ?

5 3 Call: s ?


```
5 3 Exit: s ?
6 3 Call: w ?
6 3 Exit: w ?
4 2 Exit: r ?
1 1 Exit: p ?
```

yes

```
:- dynamic p/0,q/0,r/0,s/0,t/0,v/0,w/0.
```

```
p:-q,r.
```

```
q:-s,t.
```

```
q:-w.
```

```
r:-t.
```

```
r:-s,w,v.
```

```
s.
```

```
w.
```

```
| ?- p.
```

```
1 1 Call: p ?
```

```
2 2 Call: q ?
```

```
3 3 Call: s ?
```

```
3 3 Exit: s ?
```

```
4 3 Call: t ?
```

```
4 3 Fail: t ?
```

```
3 3 Redo: s ?
```

```
3 3 Fail: s ?
```

3 3 Call: w ?
3 3 Exit: w ?
2 2 Exit: q ?
4 2 Call: r ?
5 3 Call: t ?
5 3 Fail: t ?
5 3 Call: s ?
5 3 Exit: s ?
6 3 Call: w ?
6 3 Exit: w ?
7 3 Call: v ?
7 3 Fail: v ?
6 3 Redo: w ?
6 3 Fail: w ?
5 3 Redo: s ?
5 3 Fail: s ?
4 2 Fail: r ?
2 2 Redo: q ?
3 3 Redo: w ?
3 3 Fail: w ?
2 2 Fail: q ?
1 1 Fail: p ?

no

:- dynamic p/0,q/0,r/0,s/0,t/0,v/0,w/0.

p:-q,r.
q:-s,t.
q:-w.
r:-t.
r:-s,w,\+ v.
s.
w.

| ?- p.
1 1 Call: p ?
2 2 Call: q ?
3 3 Call: s ?
3 3 Exit: s ?
4 3 Call: t ?
4 3 Fail: t ?
3 3 Redo: s ?
3 3 Fail: s ?
3 3 Call: w ?
3 3 Exit: w ?
2 2 Exit: q ?
4 2 Call: r ?
5 3 Call: t ?
5 3 Fail: t ?
5 3 Call: s ?
5 3 Exit: s ?
6 3 Call: w ?
6 3 Exit: w ?

```
7 3 Call: v ?
7 3 Fail: v ?
4 2 Exit: r ?
1 1 Exit: p ?
```

yes

```
:- dynamic a/0,p/0,q/0,r/0,s/0,t/0,v/0,w/0.
```

```
a:- p.
a:- \+ p,s.
p:- q,r.
q:- s,t.
q:- w.
r:- t.
r:- s,w,v.
s.
w.
```

```
| ?- a.
1 1 Call: a ?
2 2 Call: p ? s
2 2 Fail: p ?
2 2 Call: p ? s
2 2 Fail: p ?
2 2 Call: s ?
2 2 Exit: s ?
```

```

1 1 Exit: a ?

yes

p:-q,r.
q:-s,t.
q:-w.
r:-t.
r:-s,w.
s.
w.

| ?- p.
1 1 Call: p ?
2 2 Call: q ?
3 3 Call: s ?
3 3 Exit: s ?
4 3 Call: t ?
4 3 Exception: t ?
2 2 Exception: q ?
1 1 Exception: p ?
{EXISTENCE ERROR: t: procedure user:t/0 does not exist}
{trace}

:- dynamic t/0.

p:-q,r.

```

q:-s,t.
q:-w.
r:-t.
r:-s,w.
s.
w.

| ?- p.
1 1 Call: p ?
2 2 Call: q ?
3 3 Call: s ?
3 3 Exit: s ?
4 3 Call: t ?
4 3 Fail: t ?
3 3 Redo: s ?
3 3 Fail: s ?
3 3 Call: w ?
3 3 Exit: w ?
2 2 Exit: q ?
4 2 Call: r ? s
4 2 Exit: r ?
1 1 Exit: p ?

yes

A First Modeling Example:

```
:- dynamic rained_last_night/0, sprinkler_was_on/0.
```

```
wet-lawn:- rained_last_night.
```

```
wet-lawn:- sprinkler_was_on.
```

```
wet_street:- rained_last_night.
```

```
wet_and_shiny_lawn:- wet-lawn.
```

```
wet_shoes:- wet-lawn.
```

```
| ?- wet-lawn.
```

```
no
```

```
| ?- wet_street.
```

```
no
```

```
| ?- wet_and_shiny_lawn.
```

```
no
```

```
| ?- asserta(sprinkler_was_on).
```

```
yes
```

```
| ?- wet-lawn.
```

```
yes
```

```
| ?- wet_street.
```

```
no
```

```
| ?- wet_and_shiny_lawn.
```

```
yes
```

```
| ?- asserta(rained_last_night).  
yes  
| ?- wet_street.  
yes  
| ?- wet_and_shiny_lawn.  
yes  
| ?- wet_shoes.  
yes
```


Birds Normally Fly:

```
:- dynamic ostrich_tweety/0, penguin_tweety/0,  
   canario_tweety/0, abnormal_tweety/0.
```

```
flies_tweety:- bird_tweety, \+ abnormal_tweety.
```

```
abnormal_tweety:- ostrich_tweety.
```

```
bird_tweety.
```

```
| ?- flies_tweety.  
yes  
| ?- abnormal_tweety.  
no  
| ?- asserta(ostrich_tweety).  
yes  
| ?- abnormal_tweety.  
yes  
| ?- flies_tweety.  
no  
| ?- retract(ostrich_tweety).  
no  
| ?- retract(ostrich_tweety).  
yes  
| ?- asserta(penguin_tweety).
```

```
yes
| ?- flies_tweety.
yes
| ?- abnormal_tweety.
no
| ?- assertz((abnormal_tweety:-penguin_tweety)).
yes
| ?- flies_tweety.
no
| ?- abnormal_tweety.
yes
```



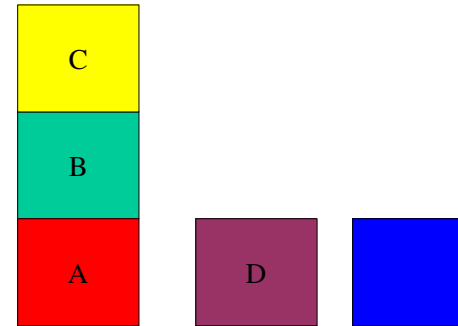
Chapter 6: Introduction to Predicate Logic
Representation and Semantics

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

Why?

- We saw that propositional logic has limited expressive power in comparison with predicate logic



- Description of this blocks world in propositional logic?

Propositional variables (and true basic propositions):

BonA, ConB, DonGround, AleftOfD, ...

General statements about properties of objects?

“Every object that is on top of a block is not on the ground”

Defining new or extending old properties?

“A first object is to the left of a second object if it is on top of a third object that is to the left of the second”

Can we **conclude** that B is to the left of D ?

Only if there is an explicit statement of all possible particular implications applied to concrete objects, e.g.

$$A_{\text{leftOf}}D \wedge B_{\text{on}}A \rightarrow B_{\text{leftOf}}D \quad \textit{etc.}$$

- A more **classical example**:

“All men are mortal. Socrates is a man. Then Socrates is mortal”

Cannot be expressed in a natural way in propositional logic as a finite and **logically valid argument (formula)**

- Or *“All rational numbers that are not smaller than $\sqrt{2}$ are greater than $\sqrt{2}$ ”*

We do not want an (uncountably) infinite conjunction of particular implications ...

- We need languages that allow us to talk about **properties** (predicates) of individuals in our domain of discourse and to **quantify** over those individuals

Syntax of First-Order Predicate Logic

- First-order predicate logic: FOPL
- Introduce **names to denote individuals** (some of) in the domain, e.g. A, B, C, D, ...

There may be individuals -with or without a name in the domain- that do not have a name in the formal language

- We are distinguishing between the symbolic name that denotes an object and the object (or its non-symbolic, meta-level name)

For simplicity, we omit the underline

It is the symbolic name what matters (for the moment) ...

- **Predicates** (symbolic): $On(\cdot, \cdot)$, $LeftOf(\cdot, \cdot)$, $Color(\cdot, \cdot)$, $Block(\cdot)$
- **Symbolic sentences**: $Block(A)$, $On(B, A)$, $LeftOf(A, D)$, ...

More complex ones:

$$\forall x \forall y \forall z (LeftOf(x, y) \wedge On(z, x) \rightarrow LeftOf(z, y))$$

“for all ...”

$$\forall x (\exists y Block(y) \wedge On(x, y) \rightarrow \neg On(x, ground))$$

“for every object, if there is a block”

- We can have **symbolic functions**: $color(\cdot)$, $weight(\cdot)$
- More sentences: $color(C) = yellow$, $color(green) = green$,
 $weight(A) = 5$, $weight(yellow) = 0$, ...

- More generally, the symbolic language is based on the following **alphabet**:
 - Names (or **constants**) to **denote** some distinguished elements of the domain: $A, B, C, D, ground, green, yellow, 0, 1, 2, \dots$
 - **Predicates**: $On(\cdot, \cdot), LeftOf(\cdot, \cdot), Color(\cdot, \cdot), Block(\cdot)$
 - **Function symbols**: $color(\cdot), weight(\cdot)$
 - **Propositional connectives**: $\neg, \wedge, \vee, \rightarrow, \dots$
 - **Parenthesis**: $), ($
 - **Equality**: $=$ (binary predicate, not always available)
 - **Variables**: x, y, z, \dots (countable infinite list, to refer to **individuals**)
 - **Quantifiers**: \forall, \exists

- **What is a formula?** First, formulas talk about objects, and the symbolic objects are the **terms**
- **Terms are basically as in algebra**, where they appear in equations (formulas) and can be composed using numbers, variables and operations, as in: $(2 \cdot x + z) - 5 \cdot y = 0$ (terms: $x, 2, 2 \cdot x, 2 \cdot x + z, \dots$)

In our case, the terms are:

1. Each constant is a term: $A, yellow, \dots$
2. Each variable is a term: x, y, z, \dots
3. If t is a term (arbitrary, no matter how it was obtained or constructed), then $colour(t)$ is a term, and $weight(t)$ is a term

An inductive definition of term!!!

- Examples of terms: $A, x, color(B), color(color(D)), weight(color(5)), \dots$

- **Formulas are symbolic propositions** and can be defined similarly, by **induction**
 1. Predicates applied to terms produce **atomic formulas**:
 $On(B, A), LeftOf(x, D), On(color(A), y), \dots$
 2. Equalities between terms are atomic formulas:
 $color(C) = yellow, weight(z) = weight(ground), \dots$
 3. If φ is a formula, then $\neg\varphi$ is a formula
 4. If φ, ψ are formulas, then propositional combinations of them are formulas (with $\wedge, \vee, \rightarrow, \dots$)
 5. If φ is a formula and x is a variable, then $\forall x\varphi$ and $\exists x\varphi$ are formulas

- (Legal or well-formed) formulas are those obtained by applying a **finite** number of times the rules above

Another way to put it:

The set (language) of formulas is the smallest set (under set inclusion) of finite words over the alphabet above that is closed under the rules 1. - 5.

A minimization principle at work ...

- From the inductive definition: FOPL essentially extends (the syntax of) propositional logic

Example: $weight(z) = weight(ground)$ is a formula, then
 $\exists z(weight(z) = weight(ground))$ is also a formula

Also: $\exists x(weight(z) = weight(ground))$ and
 $\forall z(weight(z) = weight(ground))$

Among the formulas, the **sentences** are those that do not have **free variables**, and they represent closed statements

A free variable in a formula is one that appears outside the scope of a quantifier

$weight(z) = weight(ground)$ is not a sentence, because z is free, whereas $\forall z(weight(z) = weight(ground))$ is a sentence

$\exists z(weight(z) = weight(y))$ and $(\exists y On(y, A)) \wedge color(y) = blue$ are not a sentences, because y appears free (and z is **bound**)

Exercise: Define **by induction** on formulas φ :

- The set of variables appearing in a formula
- The set of quantified variables in a formula
- The set of free variables appearing in a formula

I.e. the set $FV(\varphi)$ of free variables of formula φ

E.g. $FV(\exists x(P(x) \wedge R(y))) = \{y\}$

- From this define in precise terms the notion of sentence

Easy: $FV(\varphi) = \emptyset$

- The length of a formula

This is all similar to defining by recursion the length of a list:

1. $length([]) := 0$ (base case, the empty list)
2. $length([a|L]) := 1 + length(L)$ (a : an element of the base alphabet; L : a list)

$[a|L]$ is the list obtained by appending symbol a as a prefix to L

Semantics of FOPL

- All this is symbolic
- What about **truth** of formulas?

This is part of the **semantics**

- We need to **interpret** symbols, terms, formulas Where?
- We use **semantic structures** that stay in **correspondence** with the symbolic language

Structures can be seen as **models** of the domain of discourse, i.e. as (simplified) abstractions that capture its relevant aspects

- Think of set-theoretic structures as those studied in mathematics, e.g. graphs, order relations on sets, vector spaces, groups, numerical structures, ...

- Structures are represented in set-theoretic terms

$$\mathfrak{S} = \langle U, R^U, \dots, f^U, \dots, c^U, \dots \rangle$$

Consisting of:

- **Domain** (or **universe**): a non-empty set

- **Relations** between (domain) elements

- **Functions** that send elements to elements

(functions are assumed to be *total* in the domain, i.e. defined for every possible argument)

- **Distinguished elements**

The semantics of FOPL will be presented by means of examples ...

Example: A structure \mathcal{B} representing our blocks world

- Domain $\mathcal{B} = \{A, B, C, D, \text{"the blue block"}, \dots, \text{green}, \text{yellow}, \dots, \text{ground}, 0, 1, \dots\}$

- Relations:

$$\text{Block}^{\mathcal{B}} = \{A, B, C, D, \text{"the blue block"}, \dots\} \quad (\text{unary})$$

$$\text{On}^{\mathcal{B}} = \{(A, \text{ground}), (B, A), (C, B), (\text{"the blue block"}, D), (D, \text{ground}), \dots\} \quad (\text{binary})$$

$$\text{LeftOf}^{\mathcal{B}} = \{(A, D), (B, D), (C, D), (A, \text{"the blue block"}), \dots\}$$

- Functions:

$$\text{colour}^{\mathcal{B}}: A \mapsto \text{red}, B \mapsto \text{green}, C \mapsto \text{yellow}, D \mapsto \text{purple}, \\ \text{yellow} \mapsto \text{yellow}, 0, 1, 2, \dots \mapsto \text{black}$$

$$\text{weight}^{\mathcal{B}}: A \mapsto 5, \dots, \text{yellow} \mapsto 0, \dots, 0, 1, 2, 3, \dots \mapsto 0$$

- Distinguished individuals: $A, B, C, D, \text{yellow}, \dots$
(not the blue block, we do not have a symbolic name for it)

Example: Consider a language of predicate logic for talking about algebraic properties of real numbers

We start from the **set of symbols** (aka. **signature**, schema in DBs)

$$\mathcal{S}^a = \{<(\cdot, \cdot), +(\cdot, \cdot), \times(\cdot, \cdot), \bar{0}, \bar{1}\} \quad (\text{there are fixed arities})$$

We may assume here that the equality, $=$, is also there, as a part of the logic, for free

In contrast to other predicate names, the interpretation of “ $=$ ” is fixed, by the logic: it has to behave as equality (more coming)

We can build some formulas:

1. $<(\times(y, +(\bar{1}, \bar{1})), z)$ (informally $y \times (\bar{1} + \bar{1}) < z$)
2. $\exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})$
3. $((\bar{1} + \bar{1}) + \bar{1}) + \bar{1} = (\bar{1} + \bar{1}) \times (\bar{1} + \bar{1})$
4. $\exists u(u \times u = \bar{1} + \bar{1})$
5. $\forall x \forall y(x + y = y + x)$
6. $\exists w(w \times w + \bar{1} = \bar{0})$

The symbolic language can be interpreted in the **structure of real numbers**:

$$\mathfrak{R} = \langle \mathbb{R}, <^{\mathbb{R}}, +^{\mathbb{R}}, \times^{\mathbb{R}}, 0, 1 \rangle$$

Only **0** and **1** are distinguished elements (our choice), and we have names for them only

But there are other elements in the domain \mathbb{R} of the structure

Are the formulas above true in \mathfrak{R} ?

There is a definition of truth in formalized languages by Alfred Tarski, based on “theory of correspondence”, 1933

Notice below that the definition of formula satisfaction is inductive/recursive on the structure of the formula

And for this, also “compositional”

The first step is **interpreting the symbols of \mathcal{S}^a in the structure \mathfrak{R}**

- Predicate $<$ is interpreted as the binary relation $<^{\mathbb{R}} = \{(0, 1), (1, 2), (-5, 4), (3.3, 5.8), (\sqrt{2}, \pi), \dots\}$
- Function symbols $+$, \times are interpreted as the binary functions addition $(+^{\mathbb{R}})$ and multiplication $(\times^{\mathbb{R}})$ in \mathbb{R} , resp.
- $\bar{0}$ and $\bar{1}$ are interpreted as the real numbers 0 and 1 , resp.

Now we can ask again: **Are the formulas above true?**

Let's check them ...

Some of them have free variables, e.g. in $y \times (\bar{1} + \bar{1}) < z$

Truth depends on the values of those variables too

We have to give an account of them too ...

1. We need an **assignment** of values in the domain to the variables

There are many possible assignments, e.g.

$\theta_1: x \mapsto 2, y \mapsto 4, z \mapsto 9.5, \dots$

\mathcal{R} and θ_1 make the formula true, denoted

$$\mathcal{R} \models (y \times (\bar{1} + \bar{1}) < z)[\theta_1]$$

Because in \mathcal{R} : $4 \times (1 + 1) < 9.5$

Read: *“The structure \mathcal{R} makes the formula $y \times (\bar{1} + \bar{1}) < z$ true when the variables are interpreted according to θ_1 ”*

As before, “ \models ” denotes the satisfaction of a formula by a structure

Other assignments:

$\theta_2: x \mapsto 0, y \mapsto 4, z \mapsto 9.5, \dots$ (true)

$\theta_3: x \mapsto 2, y \mapsto 8.2, z \mapsto 9, \dots$ (false)

$$\mathfrak{R} \not\models (y \times (\bar{1} + \bar{1}) < z)[\theta_3]$$

Value for x does not matter, but only the values for free variables in the formula

- With an interpretation structure \mathfrak{S} for the symbols in \mathcal{S} and an assignment θ (of values in the domain to the variables), we can interpret any symbolic term (easily done by induction on terms)

2. $\exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})$ has a free variable, y

So, truth still depends upon its value

But not on the values assigned to auxiliary variables used to quantify only, in this case z

Consider $\theta_4: x \mapsto 2, y \mapsto 4, z \mapsto 3.5, \dots$

The formula becomes true: Because **there is a value** for z in \mathbb{R} (the domain), e.g. $z \mapsto 100$, that, together with the values that θ_4 assigns to the **other variables**, makes $(y \times (\bar{1} + \bar{1}) < z + \bar{0})$ true

That is, there is a value, e.g. $100 \in \mathbb{R}$, such that the **new assignment** $\theta_4(\frac{z}{100})$ that coincides with θ_4 for every variable, except for z (that gets value 100), makes the non-quantified formula true, i.e.

$$\mathfrak{R} \models \exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})[\theta_4]$$

Because

$$\mathfrak{R} \models (y \times (\bar{1} + \bar{1}) < z + \bar{0})[\theta_4(\frac{z}{100})]$$

Because in \mathfrak{R} : $4 \times^{\mathbb{R}} (1 +^{\mathbb{R}} 1) <^{\mathbb{R}} 100 +^{\mathbb{R}} 0$

Notice the interpretation of the term $y \times (\bar{1} + \bar{1})$ above, as $4 \times^{\mathbb{R}} (1 +^{\mathbb{R}} 1)$

Notice the **recursive (inductive) definition** of the interpretation of a term

Notice the **recursive (inductive) definition of truth** of a formula:

The truth of $\exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})$ wrt. \mathfrak{R}, θ_4 is **reduced** to the truth of the simpler formula $y \times (\bar{1} + \bar{1}) < z + \bar{0}$ wrt. \mathfrak{R}, θ'_4 , where θ'_4 is a local perturbation of θ_4 at z

Instead of $\mathfrak{R} \models \exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})[\theta_4]$,

we can equivalently write (this can be proved):

$$\mathfrak{R} \models \exists z(y \times (\bar{1} + \bar{1}) < z + \bar{0})[4]$$

In the square brackets we indicate only the values for the free variables, in this case y ; those are the relevant values

- The definition of truth is **compositional** in the sense that the truth of a formula is determined by the truth of its subformulas

3. The formula $(\bar{1} + \bar{1}) + \bar{1} = (\bar{1} + \bar{1}) \times (\bar{1} + \bar{1})$ is clearly true

We do not need any assignments in this case

$$\mathfrak{R} \models ((\bar{1} + \bar{1}) + \bar{1}) = (\bar{1} + \bar{1}) \times (\bar{1} + \bar{1})$$

The **equality symbol** $=$ in the formal language is interpreted as the equality relation in the domain; i.e. as the “diagonal” in $\mathbb{R} \times \mathbb{R}$:

$$=^{\mathbb{R}} := \{(a, a) \mid a \in \mathbb{R}\}$$

Usually this interpretation is fixed, **given by the logic**, not like other relations symbols, whose interpretations are variable

4. Formula $\exists u(u \times u = \bar{1} + \bar{1})$ is true

Because there is a value in \mathbb{R} for variable u , namely $\sqrt{2}$, such that
 $\mathfrak{R} \models (u \times u = \bar{1} + \bar{1})[\sqrt{2}]$ (recursion/induction again)

Then, $\mathfrak{R} \models \exists u(u \times u = \bar{1} + \bar{1})$

5. Formula $\forall x \forall y(x + y = y + x)$ is true, because **for every values** $a, b \in \mathbb{R}$ for variables x, y , it holds (and again ...)

$$\mathfrak{R} \models (x + y = y + x)[a, b]$$

More precisely, for every assignment θ and values $a, b \in \mathbb{R}$:

$$\mathfrak{R} \models (x + y = y + x)\left[\theta\left(\frac{x}{a} \frac{y}{b}\right)\right]$$

6. $\mathfrak{R} \not\models \exists w(w \times w + \bar{1} = \bar{0})$

Exercise: Model our class (lecture room) as a set-theoretic structure

Introduce a corresponding language $L(S)$ of predicate logic based on a set of symbols S to talk about that structure

Include in S unary and binary predicates and function symbols

State some formulas

List some true and false formulas and sentences, evaluating their truth using the definition given above

- Remark:

The formulas of the arithmetical language $L(S^a)$ above can be interpreted in any structure that is **compatible** with the language

It does not have to be arithmetical, not even “numerical”

- All we need is a structure $\mathfrak{A} = \langle A, <^A, +^A, \times^A, 0^A, 1^A \rangle$, with:
 - A is a non-empty domain
 - $<^A \subseteq A^2$ is a binary relation on A
 - $+^A, \times^A : A^2 \rightarrow A$ are two binary, total operations on A
 - $0^A, 1^A \in A$ are two distinguished elements
- For some of those formulas we also need a valuation from variables to the domain A (when there are free variables)

Example: $\mathfrak{A} = \langle A, <^A, +^A, \times^A, elephant, mouse \rangle$, with

- A is the set of names of mammals
- $name1 <^A name2$ iff typically the size of (mammal with name) $name1$ is smaller than the size of $name2$
- $+^A: (name1, name2) \mapsto name1$
- $\times^A: (name1, name2) \mapsto elephant$

That the formulas are true or false in those arbitrary, but compatible structures is another issue (check with this structure, using a valuation!)

Other numerical structures naturally offer themselves to interpret those formulas

Exercise: Check the truth or falsity of the formulas above in the structures that follow

Whenever necessary, invent your own valuation (or assignment) θ

- $\mathfrak{N} = \langle \mathbb{N}, <^{\mathbb{N}}, +^{\mathbb{N}}, \times^{\mathbb{N}}, 0^{\mathbb{N}}, 1^{\mathbb{N}} \rangle$

(invent yourself a valuation or several to try ...)

- $\mathfrak{C} = \langle \mathbb{C}, <^{\mathbb{C}}, +^{\mathbb{C}}, \times^{\mathbb{C}}, 0^{\mathbb{C}}, 1^{\mathbb{C}} \rangle$, where $<^{\mathbb{C}}$ can be defined, e.g. by $z_1 <^{\mathbb{C}} z_2$ iff $|z_1| < |z_2|$

(there is no natural order in \mathbb{C} , so we are just inventing a binary relation)

Exercise: Are the following sentences true or false in the structure \mathfrak{B} that models the blocks world?

Decide intuitively, **but also** using the precise definition of truth introduced a *propos* the structure of real numbers

1. $\exists x (Block(x) \wedge colour(x) = blue)$
2. $\forall x \neg \exists y On(y, x)$
3. $\neg \forall x \neg \exists y On(y, x)$
4. $\exists x \exists y \neg On(y, x)$
5. $\forall x (\exists y On(x, y) \rightarrow Block(x))$
6. $\exists y (Block(y) \wedge \neg \exists z (Block(z) \wedge LeftOf(y, z)))$
7. $\forall x \forall y (colour(x) = colour(y) \rightarrow x = y)$

Example: Propositional logic works fine for graphs with a fixed and small number of vertices

Not for arbitrary finite graphs

We can use the more expressive first-order predicate logic

Graphs are structures of the form $\mathfrak{G} = \langle V, E \rangle$, where V is a finite set and $E \subseteq V \times V$ is a binary relation on V (directed in this case)

Introduce a binary predicate symbol $R(\cdot, \cdot)$ to denote the binary relation E , and the binary symbol $=$ to denote equality

In the language $L(\{R\})$ of first-order predicate logic we may specify several properties of graphs:

1. No self-loops: $\forall x \neg R(x, x)$

This can be imposed as a requirement on graphs, an extra condition

2. The edge relationship is symmetric:

$\forall x \forall y (R(x, y) \rightarrow R(y, x))$ (a way to handle non-directed graphs)

3. We could also ignore the direction by **defining** a new predicate

$\forall x \forall y (C(x, y) \iff R(x, y) \vee R(y, x))$, using it instead of R

(a new predicate symbol introduced in the language with its definition)

On the LHS is the new, defined predicate is introduced; on the RHS side we have already available relations (definitions in math are of this kind; they can be seen as abbreviations/shorthands, and could be eliminated)

It is like defining a [view in relational DBs](#)

In this, case on top of the extensional table R

Now one can prove (or obtain as logical consequence) that C is symmetric (do it!)

4. There is a vertex of **outdegree** 2:

$$\exists x \exists y \exists z (R(x, y) \wedge R(x, z) \wedge \neg y = z \wedge \forall w ((\neg w = y \wedge \neg w = z) \rightarrow \neg R(x, w)))$$

If we replace here R by C (keeping its definition), the condition is now on the degree in general (in- or outdegree)

Notice that equality allows us to express some simple **cardinality conditions** (next two examples are related to this point)

They are useful in many areas, e.g. relational DBs, to define keys

When we interpret languages of predicate logic, the equality symbol $=$ is usually interpreted as the equality relation in the corresponding domain

Equality has a semantics fixed by the logic, while the symbol R can be interpreted as any binary relation on V

In item 4. above, $w = y$ and $R(x, y)$ are atomic formulas; and $w = y$, $R(x, y)$, $\neg y = z$, $\neg R(x, w)$ are **literals**

Example: The Relational Database Connection

- The database **schema** determines the symbolic language; it corresponds to the signature in logic

A database has to be compatible with the schema (making it “an instance” for the schema)

As in predicate logic: the DB has to be a set-theoretic structure compatible with the language

- A relational database can be seen as an interpretation structure \mathcal{D} , with a (possibly infinite) domain U and certain **finite** relations defined on it

<i>Salaries</i>	<i>Name</i>	<i>Salary</i>	<i>Positions</i>	<i>Name</i>	<i>Position</i>
	<i>J.Page</i>	5,000		<i>J.Page</i>	manager
	<i>V.Smith</i>	3,000		<i>V.Smith</i>	secretary
	<i>M.Stowe</i>	7,000		<i>M.Stowe</i>	manager
	<i>K.Stein</i>	4,000		<i>K.Stein</i>	accountant

- The **active domain** consists of the elements of U that appear in the tables

- A query $Q(\bar{x})$ can be seen as formula of predicate logic with free variables \bar{x} (a formula of **relational calculus**)
- The values for those variables when the database satisfies the query are the **answers to the query** from the database

<i>Salaries</i>	<i>Name</i>	<i>Salary</i>	<i>Positions</i>	<i>Name</i>	<i>Position</i>
	<i>J.Page</i>	5,000		<i>J.Page</i>	manager
	<i>V.Smith</i>	3,000		<i>V.Smith</i>	secretary
	<i>M.Stowe</i>	7,000		<i>M.Stowe</i>	manager
	<i>K.Stein</i>	4,000		<i>K.Stein</i>	accountant

```
SELECT Salary, Position
FROM Salaries, Positions
WHERE Salaries.Name = Position.Name AND Position = 'manager'
```

(the condition involves a join, plus a selection)

$Q(x, y): \exists z(Salaries(z, x) \wedge Positions(z, y) \wedge y = manager)$

(the join captured through the variable in common, z)

$\mathcal{D} \models Q[5,000, manager]$

$\mathcal{D} \models Q[7,000, manager]$

(the brackets contain the values for the variables that make the formula true, i.e. the answers from the database)

- A DBMS evaluates this query in a compositional way going through the recursive definition of truth of a query

- We can also express functional dependencies (FDs), e.g. of *Position* upon *Name*:

$$\forall x \forall y (Positions(x, y) \wedge Positions(x, z) \rightarrow y = z)$$

Actually, a “key constraint”: “Every employee has at most one position”

And also other integrity constraints (ICs)

- ICs are sentences (no free variables) that the database has to satisfy, i.e. make true

Being sentences, ICs are true or false in a given instance

- Notice that the query above and the FD can be evaluated or checked by inspecting elements inside the active domain, as opposed to other values in U

That is, the query and the FD are *safe* or *domain-independent*

- This query: $Q(x, y): \neg Salaries(x, y)$, although a proper formula of FO predicate logic, is not safe, and would not be accepted by relational DBMS

Evaluating it requires looking for values outside the tables

- Formulas can be used to **define views** (on top of existing predicates or tables) in the same way as in item 3. in page 3

For example, the definition of the virtual table *Top* showing employees who make more than 3K:

$$\forall x(Top(x) \leftrightarrow \exists y(Salaries(x, y) \wedge y > 3K))$$

And can be used in other queries as if there were tables:

$$Q'(x, y): Top(x) \wedge Positions(x, y)$$

for positions of employees with top salaries

Example: In every structure $\mathfrak{A} = \langle A, \dots \rangle$ the formula

$$\varphi_{\geq 3}: \exists x_1 x_2 x_3 (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3)$$

is true iff A , the universe of the structure, has at least 3 elements

$x_i \neq x_j$ is an abbreviation for $\neg x_i = x_j$

Similarly, one can define $\varphi_{\geq n}$ for every $n \in \mathbb{N}, n \geq 1$

Then, we can express “the structure has exactly (at most) n elements” by

$$\varphi_{\geq n} \wedge \neg \varphi_{\geq n+1} (\neg \varphi_{\geq n+1})$$

Notice that it is not clear how to express with a **finite** number of formulas (or, equivalently, with a single formula) that the structure has an infinite (or finite) number of elements

Actually, it can be proved that it is impossible; a result on the limited expressive power of first-order predicate logic

Example: We may specify by means of the following axioms 1. - 3. that a binary relation is an **equivalence relation**

1. $\forall x R(x, x)$ (reflexivity)
2. $\forall x \forall y (R(x, y) \rightarrow R(y, x))$ (symmetry)
3. $\forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$ (transitivity)

These formulas are interpreted in **structures** that are compatible with the language $L(\{R(\cdot, \cdot)\})$ of first-order predicate logic based on the set of symbols $\{R(\cdot, \cdot)\}$

They are of the form: $\mathfrak{A} = \langle A, R^A \rangle$

- A is a non-empty set, the *domain* or *universe* of the structure
- $R^A \subseteq A \times A$, is a binary relation on A
- Implicit in the language there is the equality symbol, $=$, with interpretation $=^A \subseteq A \times A$, the “diagonal of A ”, i.e.

$$=^A := \{(a, a) \mid a \in A\}$$

That is, the symbol is interpreted as a logical symbol, with its intended meaning (and is usually left implicit)

For example, $\langle \mathbb{N}, < \rangle$ is a possible interpretation:

- The usual order relation $<$ is the interpretation of the symbol R
- Formulas 1. & 2. are false in this structure: $<$ is not reflexive, nor symmetric; but it is transitive

Another interpretation: $\langle \mathbb{N}, R^{\mathbb{N}} \rangle$, with:

$$R^{\mathbb{N}} = \{(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7), \\ \dots \text{ (the diagonal) }, (2, 4), (4, 6), (4, 2), (6, 4), (2, 6), (6, 2)\}$$

Now formulas 1. - 3. are all true

Some Concepts Revisited

In predicate logic some old stuff reappears:

Valid formulas:

A formula φ is **valid** if **for every structure** \mathcal{A} compatible with its language and **every assignment** θ of values in the structure domain to variables, the formula becomes true, i.e. $\mathcal{A} \models \varphi[\theta]$

If φ is a sentence (no free variables in it), we can forget the assignment

Since predicate logic extends propositional logic, it inherits, now as valid formulas, all the “instances of tautologies”

E.g. $\forall x P(x) \vee \neg \forall x P(x)$ is an **instance of a tautology** from propositional logic, and becomes a valid formula

But there are also some “new” valid formulas, that do not come from propositional logic, e.g. $\forall x P(x) \rightarrow P(c)$ and $P(c) \rightarrow \exists y P(y)$

Logical consequence:

Given a set of **sentences** Σ and a **sentence** φ over the same language, we say that φ is a **logical consequence** of Σ iff, for every structure that makes all the sentences in Σ true, also φ becomes true, denoted $\Sigma \models \varphi$

$$\{\forall x(Man(x) \rightarrow Mortal(x)), Man(socrates)\} \models Mortal(socrates)$$

Example: Consider the set of sentences

$$\Sigma = \{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x)), \\ \forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))\}$$

Prove that $\Sigma \models \forall x \forall y (\exists z (R(x, z) \wedge R(y, z)) \rightarrow R(x, y))$ (*)

At this point we only have the semantic definition of logical consequence

Proving this directly, in semantic terms, is what we do in math all the time

The proof: Let $\mathfrak{A} = \langle A, R^A \rangle$ be an **arbitrary structure compatible with the language**; i.e. A is a non-empty set (domain), and $R^A \subseteq A \times A$, is a binary relation on A

Assume that $\mathcal{A} \models \Sigma$; then R^A is a reflexive, symmetric and transitive binary relation on A

We have to prove that $\mathcal{A} \models \varphi$, where φ is the sentence on the RHS in (*)

Let $a, b \in A$, for which there is $c \in A$ with: $(a, c), (b, c) \in R^A$

By the symmetry of R^A , it holds $(a, c), (c, b) \in R^A$

By the transitivity of R^A , it holds $(a, b) \in R^A$

Exercise: Do a symbolic proof of the logical consequence above using Otter or Prover9

Example: If Σ^G is the set of sentences corresponding to the three axioms for group theory (cf. chapter 1), we already established (semantically and with Otter) that:

$$\Sigma^G \models (\forall x(x \circ x = e) \rightarrow \forall y \forall z(y \circ z = z \circ y))$$

Example: Similarly, if Σ^{BA} is the set of axioms BA1-BA5 for boolean algebras (cf. chapter 1), we established that:

$$\Sigma^{BA} \models \forall x(x \sqcup x = x)$$

Example: Prove that that transitivity is needed to prove (*) above:

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \not\models \forall x \forall y (\exists z (R(x, z) \wedge R(y, z)) \rightarrow R(x, y))$$

Since the notion of logical consequence is a universal statement, in this case, to prove it, we have to exhibit a **counterexample**, i.e. a concrete structure that makes the LHS true, but false the RHS

$$\mathcal{A}_0 = \langle \{0, 1, 2\}, \{(0, 0), (1, 1), (2, 2), (0, 1), (1, 0), (0, 2), (2, 0)\} \rangle$$

This result also shows that **the property of transitivity is not implied by the properties of reflexivity and symmetry** (otherwise we would have obtained the RHS as a logical consequence above)

That is, the same counterexample also shows -directly- that:

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \not\models \forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$$

Notice that establishing

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \not\models \forall x \forall y (\exists z (R(x, z) \wedge R(y, z)) \rightarrow R(x, y))$$

is not the same as establishing that

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \models \neg \forall x \forall y (\exists z (R(x, z) \wedge R(y, z)) \rightarrow R(x, y))$$

Actually, one can prove that: (do it!)

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \not\models \neg \forall x \forall y (\exists z (R(x, z) \wedge R(y, z)) \rightarrow R(x, y))$$

Similarly, one can prove that:

$$\{\forall x R(x, x), \forall x \forall y (R(x, y) \rightarrow R(y, x))\} \not\models \neg \forall x \forall y \forall z (R(x, z) \wedge R(z, y)) \rightarrow R(x, y)$$

In other words, transitivity is **independent** from the other two axioms for equivalence relations (ERs)

Exercise: Prove the independence of each of the axioms for ERs from the other two

Example: We established (cf. chapter 1) that commutativity is independent from the axioms for group theory

$$\Sigma^G \not\models \forall y \forall z (y \circ z = z \circ y)$$

$$\Sigma^G \not\models \neg \forall y \forall z (y \circ z = z \circ y)$$

Remark: We have seen that for a set of sentences Σ and a sentence φ (in the same language), **it may hold:**

- $\Sigma \not\models \varphi$
- $\Sigma \not\models \neg \varphi$

We say the theory axiomatized by Σ is **incomplete**

However, for a structure \mathfrak{A} compatible with the language **it always holds:**

Either $\mathfrak{A} \models \varphi$ or $\mathfrak{A} \models \neg \varphi$ (no uncertainty wrt. a fixed structure)

Logical equivalence:

Two formulas φ, ψ are **logically equivalent** if for every valuation and assignment (compatible with their language in common), the two formulas take the same truth value:

$$\forall x P(x) \rightarrow \forall y Q(y) \equiv \neg \forall x P(x) \vee \forall y Q(y)$$

$$\forall x \forall y (P(x) \rightarrow Q(y)) \equiv \forall x \forall y (\neg P(x) \vee Q(y))$$

$$\forall x \forall y (P(x) \vee \neg Q(y)) \equiv \forall x \forall y \neg (\neg P(x) \wedge Q(y))$$

Any new equivalences, not inherited from propositional logic?

- $\neg\forall x\varphi(x) \equiv \exists x\neg\varphi(x)$ $\neg\exists x\varphi(x) \equiv \forall x\neg\varphi(x)$
 $\forall x\varphi(x) \equiv \forall y\varphi(y)$ On the RHS, y replaces x in φ **but** when y does not appear free on the LHS

(Notice: $\forall xR(x, y) \not\equiv \forall yR(y, y)$, the same with \exists)

Quantified (bound) variables can always be replaced by fresh variables

- $\forall x(\varphi(x) \wedge \psi(x)) \equiv \forall x\varphi(x) \wedge \forall x\psi(x)$
 $\exists x(\varphi(x) \vee \psi(x)) \equiv \exists x\varphi(x) \vee \exists x\psi(x)$
 $\forall x(\varphi(x) \vee \psi) \equiv \forall x\varphi(x) \vee \psi$

If x does not appear (free) in ψ

$$\exists x(\varphi(x) \wedge \psi) \equiv \exists x\varphi(x) \wedge \psi$$

If x does not appear (free) in ψ

The last four equivalences (and their restrictions) are very important

Example: $\forall x(\exists y F(y, x) \rightarrow \exists z U(x, z))$ is equivalent to
 $\forall x \forall y \exists z (F(y, x) \rightarrow U(x, z))$

The second formula is in **prenex normal form**, i.e. a prefix of quantifiers followed by a quantifier free formula

$$\begin{aligned} \forall x(\exists y F(y, x) \rightarrow \exists z U(x, z)) &\equiv \forall x(\neg \exists y F(x, y) \vee \exists z U(x, z)) \\ &\equiv \forall x(\forall y \neg F(x, y) \vee \exists z U(x, z)) \equiv \forall x \forall y (\neg F(x, y) \vee \exists z U(x, z)) \\ &\equiv \forall x \forall y \exists z (\neg F(x, y) \vee U(x, z)) \end{aligned}$$

Since the inner formula is in CNF, this formula is in **prenex CNF**

In particular, negation appears only in negative literals, i.e. negations of atomic formulas

Finally $\equiv \forall x \forall y \exists z (F(x, y) \rightarrow U(x, z))$

Exercise: Verify that

- $\forall x(\varphi(x) \vee \psi(x)) \not\equiv \forall x\varphi(x) \vee \forall x\psi(x)$
- $\exists x(\varphi(x) \wedge \psi(x)) \not\equiv \exists x\varphi(x) \wedge \exists x\psi(x)$

Hint: In each case, a **counterexample** must be found, i.e. a pair of concrete formulas φ, ψ and a concrete **structure** where the two formulas do not take the same truth values, e.g. elaborate on the following pairs of formulas:

$$\forall x(Odd(x) \vee Even(x)), (\forall xOdd(x) \vee \forall xEven(x))$$

$$\exists x(Even(x) \wedge Odd(x)), (\exists xEven(x) \wedge \exists xOdd(x))$$

Exercise: Transform $\forall x\exists y(P(x, y) \rightarrow (\neg\exists z\exists yR(z, y) \wedge \neg\forall xS(x)))$ into an equivalent formula in **prenex conjunctive normal form**

Remark: (decidability) We have seen that for propositional logic, all the decision problems that follow are solvable (or decidable), but hard:

Deciding if: (a) A formula is satisfiable, (b) A formula is tautology;
(c) Two formulas are logically equivalent; (d) a formula is a logical consequence of a set of (finite) formulas

Truth tables can be used for all this, but we do not have anything like truth tables in predicate logic (infinitely many structures to check, with infinite domains)

Actually, for predicate logic they are all unsolvable (undecidable)

Reason: First-Order (FO) Predicate Logic is more expressive, actually sufficiently expressive to specify a Turing Machine, its execution, and terminating condition

(In particular, a predicate can be used to simulate the transition function, and the existential quantifier can be used to claim the existence of a halting state)

This was first proved by Church and by Turing, using each his own model of computation (1935/6)

In the case of Turing, the Turing machines; in the case of Church, the Lambda Calculus, which is the basis for functional programming languages, e.g. LISP, Scheme, ML, ...

The logical specification is valid (as a formula) iff the machine halts ... So, if we can decide “validity” we can decide “termination” of the TM, which is not possible

Still FO predicate logic has provably a good balance of expressive power and good computational properties (not everything is lost with the undecidability above)

If we want decidability, we have to work with well-behaved, proper fragments of predicate logic, e.g. “descriptions logics” (DLs), which are used in ontologies, semantic web, conceptual models (e.g. ER and UML models), etc.

On 1st Order vs. 2nd Order

We presented the syntax and semantics of **first-order** predicate logic

This means that relations and properties and quantifications apply to **individuals** of the domain or universe of discourse

For example, in $\forall x \exists y On(x, y)$ the quantifiers refers to individuals of the domain, in this case, blocks and locations

Furthermore, the property *On* applies to this same kind of individuals

However, a 2nd order (SO) language may allow to built formulas like

$$\forall X \exists x \exists y (x \in X \wedge y \in X \wedge x \neq y \wedge colour(x) = colour(y))$$

“**every set of individuals** contains two individuals with the same colour”

Usually written: $\forall X \exists x \exists y (X(x) \wedge X(y) \wedge x \neq y \wedge colour(x) = colour(y))$

In SO languages we find quantifiers and variables for **sets of individuals** (and for relations, properties, functions)

Example: The induction principle (IP) for natural numbers

$$\forall X[(X(0) \wedge \forall x(X(x) \rightarrow X(x+1))) \rightarrow \forall x X(x)]$$

has a **SO quantification over subsets of the domain**

It has a minimization effect: The domain contains exactly the numbers 0, 1, 1+1, 1+1+1, ...

For comparison, this is first-order property of natural numbers:

$$\forall x \exists y (x = (1+1) \times y \vee x = (1+1) \times y + 1)$$

The above SO quantification is necessary to capture the structure of natural numbers

It can be proved that IP above cannot be, logically-equivalently, replaced by a (not even infinite) set of FO axioms

SO predicate logic is provably more expressive than FO predicate logic



Chapter 7: Symbolic Reasoning in Predicate Logic

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

FO Resolution

Now **clauses** are sentences that are **implicitly universally quantified** disjunctions of literals

$$\neg R(x, y) \vee P(a, z) \vee \neg Q(b)$$

$$\neg S(f(x), c) \vee R(w, v)$$

Resolution is a deduction rule that derives a new clause from two other clauses

Can we cancel $\neg R(x, y)$ and $R(w, v)$ above?

As before, we use to write FO clauses as sets of literals, e.g.

$$\{\neg R(x, y), P(a, z), \neg Q(b)\}$$

Example: (idea) Axioms 1. - 3. for equivalence relations:

1. $\forall x R(x, x)$ (reflexivity)
2. $\forall x \forall y (R(x, y) \rightarrow R(y, x))$ (symmetry)
3. $\forall x \forall y \forall z (R(x, y) \wedge R(y, z) \rightarrow R(x, z))$ (transitivity)

Add the facts: 4. $R(a, b)$, 5. $R(b, c)$

Let us prove in a **symbolic manner** that $R(a, c)$ follows from the set of formulas 1. - 5.

Assume it does not, i.e. we add to the clausal versions of 1. - 5. the formula:

$$6. \neg R(a, c) \tag{6.}$$

$$\neg R(x, y) \vee \neg R(y, z) \vee R(x, z) \tag{3.}$$

In the second clause the implicit universal quantifiers on x, y, z allow them to take any symbolic values

We may cancel 6. with the third literal in 3., after having transformed $R(x, z)$ into (having unified $R(x, z)$ with) the complementary literal of $\neg R(a, c)$ by giving to x, z the values a, c , resp.

We obtain the clause: $\neg R(a, y) \vee \neg R(y, c)$ (7.)

Now, we cancel the first literal with 4. after having unified y with b ; obtaining: $\neg R(b, c)$ (8.)

Finally, we cancel this literal with formula 5., obtaining the empty clause

We have proved (if this symbolic manipulation is sound) that $R(a, c)$ is a logical consequence of the set of clauses corresponding to 1. – 5.

(We can see from the proof (refutation) that axioms 1. and 2. are not needed)

Example:

$$\frac{R(x, g(x)) \vee Q(x) \quad \neg R(f(z), y)}{?}$$

To apply resolution we need to have two complementary literals, we do not have them here

To achieve this, apply the **unification** process, that makes two atoms syntactically the same by **substituting variables by terms**

A substitution θ that achieves this is a **unifier**

There may be many of them, but we always choose a **most general unifier (mgu)**, one that is less restrictive than others; and then leaves more room for further unifications

If we had (unnecessarily) made $y := a$ in page 3, instead of (7.) above, we would have obtained $\neg R(a, a) \vee \neg R(a, c)$, and we would have been stuck

A possible unifier $\theta: x \mapsto f(c), y \mapsto g(f(c)), z \mapsto c$

With $\theta: x \mapsto f(c), y \mapsto g(f(c)), z \mapsto c$:

$$R(x, g(x))\theta = R(f(c), g(f(c))), \quad Q(x)\theta = Q(f(c))$$
$$\neg R(f(z), y)\theta = \neg R(f(c), g(f(c)))$$

Then

$$\frac{(R(x, g(x)) \vee Q(x))\theta \quad \neg R(f(z), y)\theta}{Q(f(c))}$$

Which is fine, but θ is not an mgu; instead this is an mgu $\theta': x \mapsto f(z), y \mapsto g(f(z))$

Now

$$\frac{(R(x, g(x)) \vee Q(x))\theta' \quad \neg R(f(z), y)\theta'}{Q(f(z))}$$

This leaves the variable z free, and can be used for new resolutions (which may be needed to reach the empty clause)

We will always apply resolution with an mgu

The resolution rule is sound with any unifier, but in complete proofs we need to use the mgus in order to have completeness

... because several resolutions may be necessary, which requires to have the variables as “free” as possible, to take any values that may be needed

There are fast algorithms to find mgus or determine if none exists; systems like Otter, Prover9, Prolog, ... have them implemented

It is usually the case that in a KB no pair of clauses have variables in common, this makes the unification easier (and possible)

This can always be achieved, because clauses have all the variables implicitly universally quantified

So, the variables in any of them can be replaced by fresh variables

We will assume the clauses in a KB do not share variables; and the resolvents get fresh variables

For example, the (implicitly universally quantified) clauses

$$\neg R(x, y) \vee P(a, z) \vee \neg Q(b)$$
$$\neg S(f(x), c) \vee R(w, v)$$

in the KB, can be replaced by the equivalent

$$\neg R(x, y) \vee P(a, z) \vee \neg Q(b)$$
$$\neg S(f(u), c) \vee R(w, v)$$

One says: “variables are standardized apart”

Officially: The FO resolution rule:

$$\frac{C_1 \quad C_2}{[(C_1 \setminus S_1) \cup (C_2 \setminus S_2)]\theta}$$

C_1, C_2 are clauses, $S_1 \subseteq C_1, S_2 \subseteq C_2$, θ is an mgu of $S_1 \cup \overline{S_2}$ (i.e. it collapses to a single literal), and $\overline{S_2}$ is formed by the complementary literals of those in S_2

The example above in these terms: (with $S_1 = \{R(x, g(x))\}, S_2 = \{\neg R(f(z), y)\}$)

$$\frac{\begin{array}{c} \{Q(x), R(x, g(x))\} \\ \{\neg R(f(z), y)\} \end{array}}{[(\{Q(x), R(x, g(x))\} \setminus \{R(x, g(x))\}) \cup (\{\neg R(f(z), y)\} \setminus \{\neg R(f(z), y)\})]\theta'}$$

i.e. the resolvent is $Q(f(z))$, that with a fresh variable may be $Q(f(u))$ (θ' as in page 6) (here $(S_1 \cup \overline{S_2})\theta' = \{R(f(z), g(f(z)))\}$)

FO resolution works with clauses

What if the KB is not in clausal form?

Where do we get clauses from?

In propositional logic it was easy: compute first the equivalent CNF; **now we have quantifiers**

Example: $\exists y \forall x (\forall u Q(x, u, z) \rightarrow \forall z P(y, u, z)) \equiv$

$$\exists y \forall x (\neg \forall u Q(x, u, z) \vee \forall z P(y, u, z)) \equiv$$

$$\exists y \forall x (\exists u \neg Q(x, u, z) \vee \forall z P(y, u, z)) \equiv$$

$$\exists y \forall x (\exists v \neg Q(x, v, z) \vee \forall z P(y, u, z)) \equiv \quad (!)$$

$$\exists y \forall x \exists v (\neg Q(x, v, z) \vee \forall z P(y, u, z)) \equiv$$

$$\exists y \forall x \exists v (\neg Q(x, v, z) \vee \forall w P(y, u, w)) \equiv$$

$$\exists y \forall x \exists v \forall w (\neg Q(x, v, z) \vee P(y, u, w)) \quad (*) \quad (\text{in prenex CNF})$$

We want universal quantifiers only

Replace the first existential quantifier in the last formula by a **fresh constant**, a **Skolem constant** that represents the (undetermined) existing object:

$$\forall x \exists v \forall w (\neg Q(x, v, z) \vee P(c, u, w))$$

Do the same with the second existential quantifier, but this time the v depends on x , so

$$\forall x \forall w (\neg Q(x, f(x), z) \vee P(c, u, w)) \quad (**)$$

(f is a fresh **Skolem function**)

Finally, we leave the universal quantifiers implicit, tacit, obtaining the clause $\neg Q(x, f(x), z) \vee P(c, u, w)$

There is no claim about the logical equivalence between (*) and (**), actually they have different vocabularies

However, they are **equi-consistent**

(This is a general theorem about the transformation above of a KB in FO predicate logic into a set of clauses)

It can be proved that (*) is (in)consistent iff (**) is (in)consistent

This is not difficult to prove in general: If one of them has a model, it is possible to obtain from it a model for the other

This **equi-consistency** result is good enough for our purposes, due to the reduction of logical consequence to inconsistency

Remarks:

- It can be proved that, at a each resolution step, any mgu will do, i.e. they are essentially the same
- A prefix of the form $\forall x \forall y \exists z \dots$ will introduce a fresh Skolem function $f(x, y)$ for z
- The order of quantifier extraction (cf. (!) on page 10) does not matter (for equi-consistency)

Example: Prove that

$$\forall x(\neg P(x, a) \rightarrow \exists y(P(x, y) \wedge P(y, x))) \models \\ \exists x \exists y(P(x, a) \wedge P(x, y) \wedge P(y, x))$$

We have to show that the set \mathcal{F} formed by the sentences

- (a) $\forall x(\neg P(x, a) \rightarrow \exists y(P(x, y) \wedge P(y, x)))$,
- (b) $\neg \exists x \exists y(P(x, a) \wedge P(x, y) \wedge P(y, x))$ is inconsistent

Formulas in \mathcal{F} have to be taken into clauses:

- (a) $\forall x(P(x, a) \vee \exists y(P(x, y) \wedge P(y, x)))$
 $\forall x \exists y(P(x, a) \vee (P(x, y) \wedge P(y, x)))$
 $\forall x(P(x, a) \vee (P(x, f(x)) \wedge P(f(x), x)))$
 $\forall x((P(x, a) \vee P(x, f(x))) \wedge (P(x, a) \vee P(f(x), x)))$
- Two clauses:** $P(x, a) \vee P(x, f(x)), \quad P(y, a) \vee P(f(y), y)$
(we separate variables apart)

$$\begin{aligned}
\text{(b)} \quad & \neg \exists x \exists y (P(x, a) \wedge P(x, y) \wedge P(y, x)) \\
& \forall x \forall y \neg (P(x, a) \wedge P(x, y) \wedge P(y, x)) \\
& \forall x \forall y (\neg P(x, a) \vee \neg P(x, y) \vee \neg P(y, x))
\end{aligned}$$

One clause: $\neg P(z, a) \vee \neg P(z, u) \vee \neg P(u, z)$

(fresh variables)

Now from $\mathcal{C} = \{P(x, a) \vee P(x, f(x)), P(y, a) \vee P(f(y), y),$
 $\neg P(z, a) \vee \neg P(z, u) \vee \neg P(u, z)\}$

we try to reach the empty clause

(here, $S_1 = \{P(x, a)\}$,
 $S_2 = \{\neg P(z, a),$
 $\neg P(z, u), \neg P(u, z)\}$)

(In contrast to propositional resolution,
several different atoms can canceled in one shot)

$$\frac{P(x, a) \vee P(x, f(x)) \quad \neg P(z, a) \vee \neg P(z, u) \vee \neg P(u, z)}{P(x, f(x))\theta}$$

$$P(a, f(a))$$

$$(\theta = \{\frac{x}{a}, \frac{z}{a}, \frac{u}{a}\})$$

$$\frac{P(y, a) \vee P(f(y), y) \quad \neg P(z, a) \vee \neg P(z, u), \neg P(u, z)}{P(f(a), a)}$$

$$(\theta = \{\frac{y}{a}, \frac{z}{a}, \frac{u}{a}\})$$

$$\frac{P(f(a), a) \quad \neg P(z, a) \vee \neg P(z, u), \neg P(u, z)}{\neg P(a, f(a))}$$

$$(\theta = \{\frac{z}{f(a)}, \frac{u}{a}\})$$

$$\frac{\neg P(a, f(a)) \quad P(a, f(a))}{\square}$$

In this example we did not use mgus, but this does not compromise soundness

Theorem: (soundness and completeness)

A set \mathcal{C} of FO clauses is inconsistent iff there is a FO resolution-based refutation from \mathcal{C} (resolution with mgus)

Remark:

- This theorem tells us that there is a perfect correspondence between the semantics (captured by inconsistency, in the sense that \mathcal{C} has no models) and syntactic resolution-based refutations
- For FO predicate logic there are other sound and complete deductive systems that are not based on resolution and refutations

They also allow for direct proofs

- It can be proved that SO predicate logic can have only (and has) sound deductive systems

So, there is nothing like a complete resolution-based refutation system for SO predicate logic

What about equality?

We gave to equality (symbol) a special semantics, fixed by the logic

How is the deductive methodology (system) supposed to know that?

It could treat it as any binary predicate; but the semantics would be missed ...

Two possibilities:

1. **Axiomatize the equality**, saying what are its properties and add that specification (those axioms) to the KB that requires reasoning with $=$

The axioms have to capture that it is an equivalence relation, and that “properties” are preserved when replacing equals by equals

It is good enough to concentrate on atomic properties (by induction on formulas this extended to general properties as expressed by the FO language)

Example: These are the axioms for $=$ if the language also contains a predicate $R(\cdot, \cdot)$ and a function symbol $f(\cdot)$

- $\forall x(x = x)$ (reflexivity)
- $\forall x\forall y(x = y \rightarrow y = x)$ (symmetry)
- $\forall x\forall y\forall z((x = y) \wedge (y = z) \rightarrow x = z)$ (transitivity)
- $\forall x\forall y\forall u\forall v((R(x, y) \wedge x = u \wedge y = v) \rightarrow R(u, v))$
 $\forall x\forall y(x = y \rightarrow f(x) = f(y))$
(substitution or replacement by equals)

Exercise: Generate the equality axioms for a language of predicate logic based on the symbols $\{P(\cdot), Q(\cdot, \cdot, \cdot), g(\cdot, \cdot)\}$

Exercise: Reconsider the proof with Otter for group theory of chapter 1

The proof used “=” and paramodulation

(a) Introduce a new binary predicate $E(\cdot, \cdot)$, and formulate the axioms and the theorem using E instead of “=”

(b) Express the equality axioms in terms of E (“=” should not appear anywhere)

(c) Prove the theorem with Otter, without paramodulation, but now using the (rewritten) axioms for groups and equality

2. Introduce a **new, additional deduction rule**, special for handling equality at the symbolic level: the **paramodulation rule**

$$\frac{\begin{array}{c} \{L(t)\} \cup C_1 \\ \{r = s\} \cup C_2 \end{array}}{(\{L(r)\} \cup C_1 \cup C_2)\theta}$$

C_1, C_2 are clauses, L is a literal, r, s, t are terms, θ is a mgu of t and s ; $L(r)$ obtained by replacing t by r in L

Idea: replace terms by equal terms in literals

Examples:

$$\frac{\begin{array}{c} \textit{superman} = \textit{clarkKent} \\ \textit{Flies}(\textit{superman}) \end{array}}{\textit{Flies}(\textit{clarkKent})}$$

$$\frac{\begin{array}{c} R(f(y)) \\ (x = z) \vee \neg Q(z) \end{array}}{(R(x) \vee \neg Q(z)) \frac{z}{f(y)}} \\ R(x) \vee \neg Q(f(y))$$

Prover9 does skolemization, "clausification", FO resolution and paramodulation

Some FO Otter

Example: Map of Saskatchewan, British Columbia, Alberta and Northwest Territories:



Is it possible to paint this map with blue and green?

Can we use Otter to we answer this question?

FO specification Σ of the problem:

Predicates: $area(x)$ (for territories or provinces), $neighbors(x, y)$,
 $paint(x, y)$

Sentences in Σ :

- Every area must be painted

$$\forall x (area(x) \rightarrow (paint(x, green) \vee paint(x, blue)))$$

- Every area is painted with at most one color

$$\forall x (\neg paint(x, blue) \vee \neg paint(x, green))$$

- Neighboring areas can't be painted of the same color

$$\forall x y z ((paint(x, y) \wedge neighbors(x, z)) \rightarrow \neg paint(z, y))$$

- The facts: $neighbors(BC, NT)$, $area(BC)$, etc.

Alternatives?

If all the areas have neighbors, we can eliminate the “area” facts, and add the rules:

$$\forall x y (neighbors(x, y) \rightarrow area(x))$$

$$\forall x y (neighbors(x, y) \rightarrow area(y))$$

We can also avoid entering symmetric facts, e.g. *neighbors(BC, NT)* **and** *neighbors(NT, BC)*, by adding the rule:

$$\forall x y (neighbors(x, y) \rightarrow neighbors(y, x))$$

The **specification in OTTER** is as follows:

```

formula_list(usable).
%facts
neighbors(BC,NT).  neighbors(BC,AB).  neighbors(AB,NT).
neighbors(AB,SK).  neighbors(NT,SK).

%neighbors symmetry
all x all y ( neighbors(x,y) -> neighbors(y,x) ).

%area
all x y ( neighbors(x,y) -> area(x) ).
all x y ( neighbors(x,y) -> area(y) ).

%Every area must be painted of a color
all x (area(x) -> (painted(x,green) | painted(x,blue))).

%Every area is painted of at most one color
all x ( -painted(x,green) | -painted(x,blue) ).

%Neighboring areas can't be painted of the same color
all x y z ( ( painted(x,y) & neighbors(x,z) ) ->
              -painted(z,y) ).
end_of_list.

```

OTTER transforms all the formulas into clauses:

```
list(usable) .  
1 [] neighbors(BC,NT) .  
2 [] neighbors(BC,AB) .  
3 [] neighbors(AB,NT) .  
4 [] neighbors(AB,SK) .  
5 [] neighbors(NT,SK) .  
6 [] -neighbors(x,y) | neighbors(y,x) .  
7 [] -neighbors(x,y) | area(x) .  
8 [] -neighbors(x,y) | area(y) .  
9 [] -area(x) | painted(x,green) | painted(x,blue) .  
10 [] -painted(x,green) | -painted(x,blue) .  
11 [] -painted(x,y) | -neighbors(x,z) | -painted(z,y) .  
end_of_list .
```

OTTER tries to **reach a contradiction** from the formulas stored in the `usable` and `SOS` lists, starting from the formulas in the latter

What should we put into the `SOS` list?

For the map not the colorable with the two colors, its colorability and the specification above must be mutually inconsistent

So, put into `SOS` that the map is colorable, e.g. through the sentence:

$$\textit{painted}(BC, \textit{green}) \vee \textit{painted}(BC, \textit{blue})$$

Then, our `SOS` list is:

```
list(sos).  
painted(BC,green) | painted(BC,blue).  
end_of_list.
```

If we run this in OTTER, we get the following proof:

```

----- PROOF -----
1 [] neighbors(BC,NT) .
2 [] neighbors(BC,AB) .
3 [] neighbors(AB,NT) .
8 [] -neighbors(x,y) | area(y) .
9 [] -area(x) | painted(x,green) | painted(x,blue) .
10 [] -painted(x,green) | -painted(x,blue) .
11 [] -painted(x,y) | -neighbors(x,z) | -painted(z,y) .
12 [] painted(BC,green) | painted(BC,blue) .

14 [binary,12.1,11.1] painted(BC,blue) | -neighbors(BC,x) | -painted(x,green) .

16 [binary,12.2,11.1] painted(BC,green) | -neighbors(BC,x) | -painted(x,blue) .
25 [binary,14.2,2.1] painted(BC,blue) | -painted(AB,green) .
26 [binary,14.2,1.1] painted(BC,blue) | -painted(NT,green) .
31 [binary,25.2,9.2] painted(BC,blue) | -area(AB) | painted(AB,blue) .
35 [binary,26.2,9.2] painted(BC,blue) | -area(NT) | painted(NT,blue) .
40 [binary,31.2,8.2] painted(BC,blue) | painted(AB,blue) | -neighbors(x,AB) .
46 [binary,35.2,8.2] painted(BC,blue) | painted(NT,blue) | -neighbors(x,NT) .
78 [binary,16.2,2.1] painted(BC,green) | -painted(AB,blue) .
79 [binary,16.2,1.1] painted(BC,green) | -painted(NT,blue) .
86 [binary,78.2,9.3] painted(BC,green) | -area(AB) | painted(AB,green) .
91 [binary,79.1,10.1] -painted(NT,blue) | -painted(BC,blue) .
92 [binary,79.2,9.3] painted(BC,green) | -area(NT) | painted(NT,green) .
113 [binary,86.2,8.2] painted(BC,green) | painted(AB,green) | -neighbors(x,AB) .
124 [binary,92.2,8.2] painted(BC,green) | painted(NT,green) | -neighbors(x,NT) .
153 [binary,40.3,2.1] painted(BC,blue) | painted(AB,blue) .
159 [binary,153.2,11.1] painted(BC,blue) | -neighbors(AB,x) | -painted(x,blue) .

```

```

175 [binary,46.3,3.1] painted(BC,blue)|painted(NT,blue).
180 [binary,175.1,10.2] painted(NT,blue)| -painted(BC,green).
201 [binary,113.3,2.1] painted(BC,green)|painted(AB,green).
202 [binary,201.1,180.2] painted(AB,green)|painted(NT,blue).
226 [binary,124.3,3.1] painted(BC,green)|painted(NT,green).
227 [binary,226.1,180.2] painted(NT,green)|painted(NT,blue).
253 [binary,159.2,3.1] painted(BC,blue)| -painted(NT,blue).
275 [binary,253.1,91.2] -painted(NT,blue).
281 [binary,275.1,227.2] painted(NT,green).
282 [binary,275.1,202.2] painted(AB,green).
287 [binary,281.1,11.3] -painted(x,green)| -neighbors(x,NT).
293 [binary,287.1,282.1] -neighbors(AB,NT).
294 [binary,293.1,3.1] $F.
----- end of proof -----

```

Clause 14, for example, is obtained by unifying x and y in clause 11 with BC and $green$ respectively of clause 12.

Since we got the empty clause:

⇒ The map is **not colorable** with **green** and **blue**

Exercise:

- Run the previous specification with OTTER
- What if we hadn't obtained the empty clause with the contradiction using OTTER? Is that a proof that the map is two colorable?
(Hint: Try to prove that
 $(\textit{painted}(BC, \textit{green}) \wedge \textit{painted}(BC, \textit{blue}))$
is a logical consequence of Σ using OTTER)

Paramodulation in Otter

The deduction rule:

$$\frac{\begin{array}{l} \{L(t)\} \cup C_1 \\ \{r = s\} \cup C_2 \end{array}}{(\{L'\} \cup C_1 \cup C_2)\theta}$$

C_1, C_2 are clauses, $L(r)$ is a literal, r, s, t are terms, θ is the mgu of t and s ; L' obtained by replacing t by r in L

Paramodulation is activated with one of the following flags:

- `set(para_into)`: will apply into a formula of the SOS list an equality from the usable list
- `set(para_from)`: will apply an equality from the SOS list to a rule of the usable list

The next example shows one case in which paramodulation is needed with the flag `set(para_into)`:

```
set(para_into).
```

```
formula_list(usable).
```

```
all x (f(x)=x).
```

```
-p(a,b).
```

```
end_of_list.
```

```
formula_list(sos).
```

```
p(f(a),b).
```

```
end_of_list.
```

The proof returned by OTTER is:

```
----- PROOF -----  
1 [] f(x)=x.  
2 [] -p(a,b).  
3 [] p(f(a),b).  
4 [para_into,3.1.1,1.1.1] p(a,b).  
5 [binary,4.1,2.1] $F.  
----- end of proof -----
```

The next example shows a case where paramodulation is needed with the flag set (para_from) :

```
set(para_from).
formula_list(usable).
p(f(a),b).
~p(a,b).
end_of_list.
```

```
formula_list(sos).
all x (f(x)=x).
end_of_list.
```

This example has the same set of formulas of the previous example, but the equality is now in the SOS list

The proof returned by OTTER is:

```
----- PROOF -----
1 [] p(f(a),b).
2 [] ~p(a,b).
3 [] f(x)=x.
5 [para_from,3.1.1,1.1.1] p(a,b).
6 [binary,5.1,2.1] $F.
----- end of proof -----
```

If we change the flag to `set (para_from)` in this example to `set (para_into)`, OTTER will not be able to find a contradiction, but it will return:

```
Search stopped because sos empty
```

The same happens in the previous example

Run these two examples!

Now, we can use equality in our specifications, and write an alternative specification for the 2-colorability map problem:

```

set(binary_res).      clear(unit_deletion).  clear(factor).  set(para_from).
formula_list(usable).
color(green).        color(blue).          neighbors(BC,NT).  neighbors(BC,AB).
neighbors(AB,NT).    neighbors(AB,SK).  neighbors(NT,SK).

%The only colors are blue and green
all x (color(x) -> (x=green |x=blue)).

%neighbors symmetry and area definition
all x all y ( neighbors(x,y) -> neighbors(y,x) ).
all x y (neighbors(x,y) -> area(x)).    all x y (neighbors(x,y) -> area(y)).

%Every area must be painted of a color
all x exists y (color(y) & (area(x) -> painted(x,y))).

%Every area is painted of at most one color
all x y z ( (painted(x,y) & color(y) & color(z) & y != z) -> -painted(x,z) ).

%Neighboring areas can't be painted of the same color
all x y z ( ( painted(x,y) & neighbors(x,z) ) -> -painted(z,y) ).
end_of_list.

```

Exercise:

- Show that for this specification you can also prove that the map is not 2-colorable
- Check what happens if the flag `set (para_into)` is used
- Do you see any advantages in this specification?



Chapter 8: Introduction to FO Logic Programming

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

FO Logic Programming

As before:

- FO Horn clauses; and with empty head for goals
- FO Resolution
- Similar search strategy, e.g. in Prolog it is top-down proofs with depth first search with backtracking
- Extension with NAF

What's new?

Predicates and **Variables!**

- In clauses
- In queries (goals)
- In combination with NAF (!)

Do we gain anything wrt FOL?

What follows below applies to general definite logic programs, and Prolog, in particular

Example: Consider the definite logic program (DLP) Π defining predicates P and S :

$$\begin{aligned} P(x) &\leftarrow Q(x) && (*) \\ S(x) &\leftarrow P(x) \\ Q(a). \quad S(b). \end{aligned}$$

Let's compute the "contents" of predicate P , i.e. only those ground P -atoms (ground atomic formulas with predicate P) that the program sanctions as "true"

We do this by checking candidate atoms as usual so far, i.e. top-down with backward propagation, i.e. making a head rule true **only when** the body is true:

Goal: $P(a)$? Yes!
Goal: $P(b)$? No!

Goal: $P(a)$? Yes!

Goal: $P(b)$? No! (**)

Wrt. (**), it is clear that $\Pi \not\models P(b)$, but nor $\Pi \not\models \neg P(b)$ (we know now classically negative atom can be entailed by a DLP)

So, $P(b)$ is “classically” (i.e. with FO predicate logic) undetermined, but the DLP “semantics” is sanctioning it as false, i.e. a form of CWA is applied

(Opening the ground for the introduction of the non-monotonic negation *not*)

The DLP and the FO semantics depart at this point ...

Under the former, we obtain as contents for predicate P **exactly** $\{P(a)\}$

The DLP semantics (and underlying evaluation mechanism) interprets the definition of P , given in DLP by the unidirectional rule (*), as a classical bi-directional implication in FO logic: $\forall x(P(x) : \longleftrightarrow Q(x))$

P gets a **minimal** extension under Π and its DLP procedural semantics

Similarly, the classical FO interpretations of predicates S and Q that corresponds to their DLP interpretations are:

$$\forall x(S(x) :\longleftrightarrow (P(x) \vee x = b))$$

$$\forall x(Q(x) :\longleftrightarrow x = a)$$

It is when we use:

Goal: $P(b)$? *No!*

and introduce NAF, to make $\text{not } P(b)$ true, that we start departing from classical FO logic

Example: Consider the program Π defining predicate P :

$$P(x) \leftarrow Q(x)$$

$$P(x) \leftarrow S(x) \quad \text{plus the facts: } Q(a), S(b), R(c)$$

There are two rules defining P , i.e. two cases or an implicit disjunction

For reasons as for the previous example, the implicit definition of P according to Π under the DLP semantics corresponds in classical terms to:

$$\forall x (P(x) : \longleftrightarrow (Q(x) \vee S(x)))$$

The extension for P is $\{a, b\}$

We **do not get** $P(c)$, although the instantiated rule $P(c) \leftarrow Q(c)$ is still classically true when $P(c)$ is true and $Q(c)$ false

For $P(c)$ to be true, it needs the support of (a justification via) $Q(c)$ or $S(c)$, which is not there ...

In the next chapter we will revisit semantic considerations of DLPs

In these examples, in order to obtain a classical reinterpretation of the DLP semantics, we applied what is called the “Clark’s predicate completion” of the predicates in the program

The idea is to consider “semi-definitions” given by one-directional rules in DLP, capturing “sufficient conditions”, into bi-directional sentences that capture “necessary and sufficient conditions” (which is what we do in classical and math reasoning)

Clark’s completion can be safely applied to programs without recursion as those above

Exercise: Consider Σ :

$$P(x) \leftarrow Q(x)$$

$$S(x) \leftarrow P(x)$$

$$Q(a)$$

$$S(b)$$

- (a) Run it in Prolog to see if $P(a)$ and $P(b)$ is true or false
- (b) Use the KB above in Otter/Prover9 to try to establish the same
(notice that these ATPs work in classical logic)

Example: (the blocks world revisited)

Want to **define** in the FO language the relation of “being above”, e.g. B and C are above A

First attempt:

$$\forall x \forall y (\exists z (Above(x, z) \wedge On(z, y)) \rightarrow Above(x, y)) \quad (*)$$

We add to this “definition”:

$$\forall x \forall y (On(x, y) \rightarrow Above(x, y)) \quad (**)$$

This definition is “almost good”, but does not completely capture the intended relation

The pairs in $Above$ should be “exactly those” that can be “obtained” applying the two formulas, i.e. that **but nothing more**

The relation *Above* should be **transitive closure** of the relation *On*, i.e. the **minimal relation** that contains *On* and is transitive

This “minimization” of extension cannot be captured by FOL!

It can be proved that the transitive closure of an arbitrary binary relation cannot be defined in FO logic

Exercise: Find models of the combination of (*) and (**) where *Above* is not the transitive closure of *On*

However, DLP systems like Prolog have the “minimization” above built-in in their semantics and evaluation strategy (more on this in next chapter)

(Making them depart from FO predicate logic)

Revisiting (*) and (**), notice that

$$\forall x \forall y (\exists z (Above(x, z) \wedge On(z, y)) \rightarrow Above(x, y)) \equiv$$

$$\forall x \forall y (\neg \exists z (Above(x, z) \wedge On(z, y)) \vee Above(x, y)) \equiv$$

$$\forall x \forall y (\forall z \neg (Above(x, z) \wedge On(z, y)) \vee Above(x, y)) \equiv$$

$$\forall x \forall y \forall z (\neg (Above(x, z) \wedge On(z, y)) \vee Above(x, y)) \equiv$$

$$\forall x \forall y \forall z ((Above(x, z) \wedge On(z, y)) \rightarrow Above(x, y)) \equiv$$

$$Above(x, y) \leftarrow Above(x, z), On(z, y)$$

A Horn clause (with implicit universal quantification)!

(Notice that variables in the body that are not in the head, correspond to existential variables in the body, but not explicit anymore)

So, the “definition” of *Above* above is equivalent to the pair of clauses

$Above(x, y) \leftarrow Above(x, z), On(z, y)$

$Above(x, y) \leftarrow On(x, y)$ (“base case” for the recursion)

These two clauses together with the LP mechanism/semantics for computing answers provide a recursive definition of relation *Above*!!

The underlying computation mechanism takes care of the “minimization”: (cf. page 5)

In general, recursive definitions require an implicit or explicit minimization principle or mechanism for them to work

(Recursive definitions of functions on natural numbers work due the induction principle)

Due to the backwards processing of clauses: a head can be made true **only if** the body becomes true (a sort of minimization mechanism)

Example:

1. $On(B, A)$
2. $On(C, B)$
3. $On(A, floor)$
4. $On(D, floor)$
5. $Above(x, y) \leftarrow On(x, y)$
6. $Above(x, y) \leftarrow Above(x, z), On(z, y)$

Goal (query): $\leftarrow Above(x, A)$

It is added to the program and is asking for the objects above A

Notice:

$$\leftarrow Above(x, A) \equiv \forall x(\neg Above(x, A)) \equiv \neg \exists x Above(x, A)$$

That is, **the goal is equivalent to stating that there is nothing above A**

This will be in contradiction with the program (the KB) if there is something above A

A refutation will provide the values for x that participate in the contradiction; i.e. those things above A

Those values are the counterexamples to the previous universal claim in red above

With Prolog's strategy, this is the sequence of goals generated

$\leftarrow \textit{Above}(x, A)$

top goal

$\leftarrow \textit{On}(x, A)$

resolution with 5.

□

resolution with 1. and $x = B$

an answer

more answers? backtracking ...

$\leftarrow \textit{Above}(x, z), \textit{On}(z, A)$

resolution with 6.

$\leftarrow \textit{On}(x, z), \textit{On}(z, A)$

resolution with 5. (***)

$\leftarrow \textit{On}(A, A)$

resolution with 1. and $x = B, z = A$ ×

$\leftarrow \textit{On}(B, A)$

resolution of (***) and 2. with

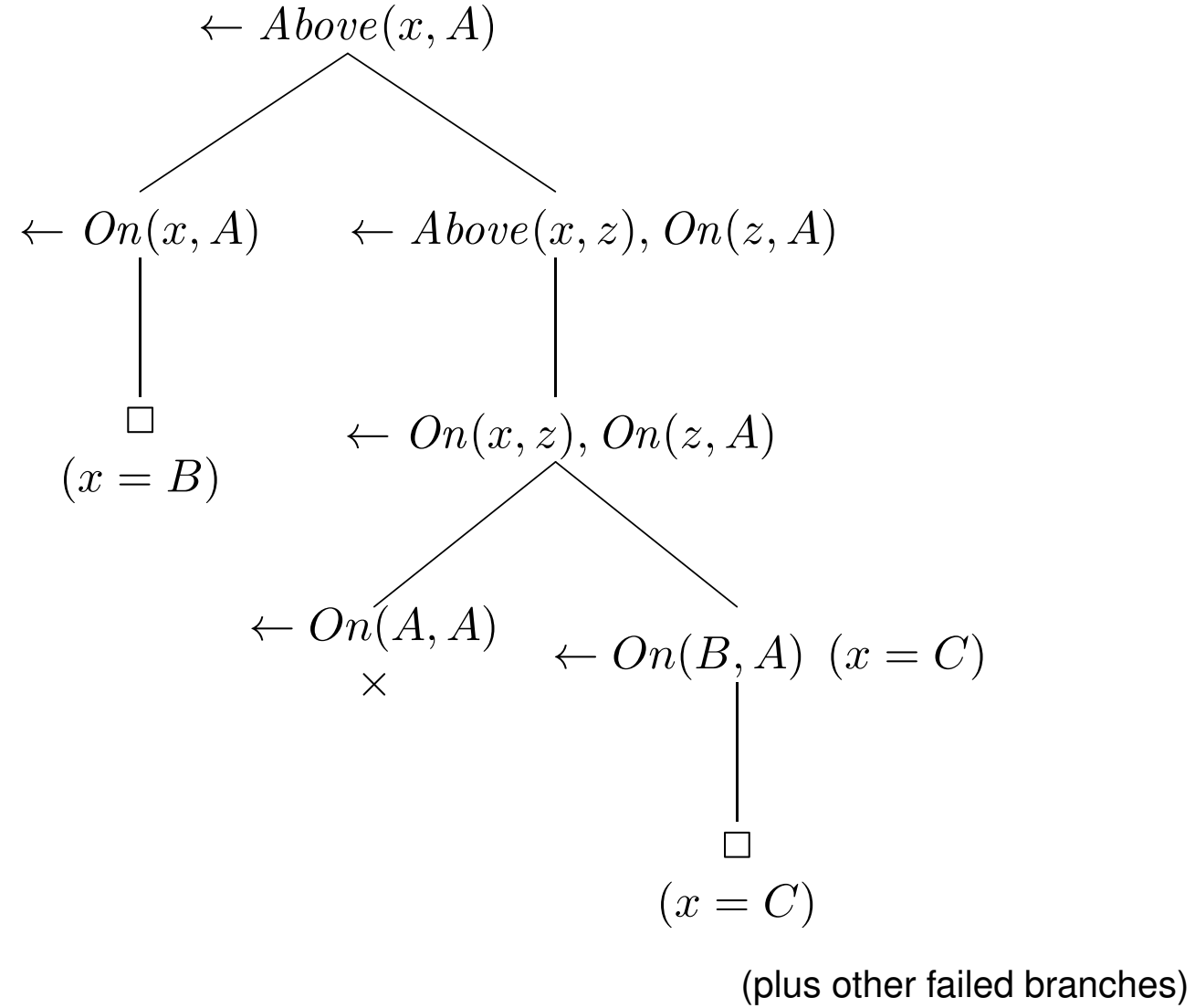
$x = C, z = B$

□

resolution with 1. and $x = C$;

another answer

No more answers, the search tree is exhaustively and finitely explored



With the goal $\leftarrow \textit{Above}(x, y)$ **all and only** the tuples in the transitive closure of *On* are computed

What we could not do in FOL ...

Exercise: Define in LP the extension of the *LeftOf* relation; and compute with it

Bottom-line, take-home message:

- FO predicate logic provably does not allow to express (define) the transitive closure (TC) of a binary relation
- Relational calculus (the FO predicate logic-based query language for relational DBs) and relational algebra, its algebraic counterpart, do not allow to define the TC of a two-column table
- LP (say à la Prolog) and Datalog (the relational DB counterpart) do allow to express the TC
- SQL3, which borrows from Datalog, allow for recursive view definitions; and TC can be expressed

Exercise: With the KB 1. - 6. in page 13

- (a) Run it in Prolog to see if $Above(C, floor)$ is true or false
- (b) Run it in Prolog to see if $Above(A, D)$ is true or false
- (c) Try to use the KB above with Otter/Prover9 to try to establish the same as in (a) and (b)
- (d) Try to use the KB above with Otter/Prover9 to try to establish the only *Above*-relationships are: $(A, floor), (B, A), (C, B), (B, floor), (C, floor), (C, A), (D, floor)$

Example: (another of transitive closure in Prolog)

The transitive closure TR of a binary relation R can be computed

```
r(a,b) .    r(b,c) .    r(c,d) .    r(d,e) .  
tcr(X,Y) :- r(X,Y) .  
tcr(X,Y) :- tcr(X,Z) , r(Z,Y) .
```

A session with the goal `:- tcr(a,X)`

The points to whom a is connected by transitivity ...

```
[bertossi@bertos ~/Programas]$ pl Welcome to SWI-Prolog
(Version 4.0.10) Copyright (c) 1990-2000 University of
Amsterdam. Copy policy: GPL-2 (see www.gnu.org)
```

```
For help, use ?- help(Topic). or ?- apropos(Word).
```

```
1 ?- ['tc.pl'].
```

```
% tc.pl compiled 0.00 sec, 1,004 bytes
```

```
Yes 2 ?- trace, tcr(a,X).
```

```
Call: (7) tcr(a, _G241) ? y
```

```
ERROR: Unknown option (h for help)
```

```
Call: (7) tcr(a, _G241) ?
```

```
Call: (8) r(a, _G241) ?
```

```
Exit: (8) r(a, b) ?
```

```
Exit: (7) tcr(a, b) ?
```

```
X = b
```

```
Yes [debug] 3 ?- trace, tcr(a,X).
```

```
Call: (7) tcr(a, _G415) ?
```

```
Call: (8) r(a, _G415) ?
```

```
Exit: (8) r(a, b) ?
```

```
Exit: (7) tcr(a, b) ?
```

```
X = b ;
```

```
Fail: (8) r(a, _G415) ?
```

Redo: (7) tcr(a, _G415) ?
Call: (8) tcr(a, _G482) ?
Call: (9) r(a, _G482) ?
Exit: (9) r(a, b) ?
Exit: (8) tcr(a, b) ?
Call: (8) r(b, _G415) ?
Exit: (8) r(b, c) ?
Exit: (7) tcr(a, c) ?

X = c ;

Fail: (8) r(b, _G415) ?
Fail: (9) r(a, _G482) ?
Redo: (8) tcr(a, _G482) ?
Call: (9) tcr(a, _G482) ?
Call: (10) r(a, _G482) ?
Exit: (10) r(a, b) ?
Exit: (9) tcr(a, b) ?
Call: (9) r(b, _G482) ?
Exit: (9) r(b, c) ?
Exit: (8) tcr(a, c) ?
Call: (8) r(c, _G415) ?
Exit: (8) r(c, d) ?
Exit: (7) tcr(a, d) ?

X = d

Yes

All the answers were found; every time an answer is given, with “;” we indicate we want more; to be obtained using backtracking

Within the same session we now want all points in the transitive closure

```
[debug] 4 ?- trace, tcr(X,Y).  
  Call: (7) tcr(_G414, _G415) ?  
  Call: (8) r(_G414, _G415) ?  
  Exit: (8) r(a, b) ?  
  Exit: (7) tcr(a, b) ?
```

```
X = a Y = b ;  
  Redo: (8) r(_G414, _G415) ?  
  Exit: (8) r(b, c) ?  
  Exit: (7) tcr(b, c) ?
```

```
X = b Y = c ;  
  Redo: (8) r(_G414, _G415) ?  
  Exit: (8) r(c, d) ?  
  Exit: (7) tcr(c, d) ?
```

```
X = c Y = d ;  
  Redo: (8) r(_G414, _G415) ?  
  Exit: (8) r(d, e) ?  
  Exit: (7) tcr(d, e) ?
```

```
X = d Y = e ;  
  Fail: (8) r(_G414, _G415) ?  
  Redo: (7) tcr(_G414, _G415) ?  
  Call: (8) tcr(_G414, _G498) ?  
  Call: (9) r(_G414, _G498) ?  
  Exit: (9) r(a, b) ?  
  Exit: (8) tcr(a, b) ?  
  Call: (8) r(b, _G415) ?  
  Exit: (8) r(b, c) ?  
  Exit: (7) tcr(a, c) ?
```

```
X = a Y = c ;  
  Fail: (8) r(b, _G415) ?  
  Redo: (9) r(_G414, _G498) ?  
  Exit: (9) r(b, c) ?  
  Exit: (8) tcr(b, c) ?  
  Call: (8) r(c, _G415) ?  
  Exit: (8) r(c, d) ?  
  Exit: (7) tcr(b, d) ?
```

```
X = b Y = d
```

```
Yes [debug] 5 ?- halt. [bertossi@bertos ~/Programas]$ exit
```

```
Process shell finished
```

Here we stopped the search (by not typing a semi-colon), and exited from the program by answering `halt.` to the prompt

Exercise: Given a binary predicate $R(\cdot, \cdot)$, the **symmetric closure** of R can be defined in FO predicate logic by the formula:

$$\forall x \forall y (S(x, y) \leftrightarrow (R(x, y) \vee R(y, x))) \quad (*)$$

1. Assume the following facts $R(a, b)$

Use FO Prover9 with (*) to prove that $S(a, b)$ and $S(b, a)$ hold

2. Using only (*) and FO Prover9, prove that S is indeed symmetric, i.e

$$\forall x \forall y (S(x, y) \rightarrow S(y, x))$$

3. Define the symmetric closure S above in FO Prolog

Beware that the limited syntax of Prolog does not allow you to directly represent all the formulas you can represent in Prover9

4. Use FO Prolog to solve item 1. above

5. Can you use FO Prolog to solve item 2. above?

You need to find a way around ...

NAF in top goals and subgoals:

So far, classical “positive” logic extended with recursion

Example: $P(x) \leftarrow Q(x), \text{ not } R(x)$
 $Q(x) \leftarrow S(x)$
 $S(b)$
 $R(a)$

Goal:

$\leftarrow P(x)$

$\leftarrow Q(x), \text{ not } R(x)$

$\leftarrow S(x), \text{ not } R(x)$

$\leftarrow \text{ not } R(b) \quad (*) \quad (\text{suspended})$

$\leftarrow R(b) \quad (\text{fail})$

Then, (*) becomes true, and so the top goal

Answer: *Yes!*, with $x = b$

Example:

$Flies(x) \leftarrow Bird(x), \text{ not } Abnormal(x)$
 $Bird(x) \leftarrow Canary(x)$
 $Bird(x) \leftarrow Hammingbird(x)$
 $Bird(x) \leftarrow Emu(x)$
 $Abnormal(x) \leftarrow Emu(x), \text{ not } Abnormal_1(x)$
 $Abnormal_1(x) \leftarrow SuperEmu(x)$
 $Emu(x) \leftarrow SuperEmu(x)$
 $Canary(tweety)$

Goal: $\leftarrow Flies(tweety)$ **Yes!**

Here we have representation of

- Defaults
- Exceptions; actually, **nested** exceptions (hierarchy)

Non-monotonic: Adding $Emu(tweety)$ answer becomes **No!**

Problems with combination of variables and NAF

Example:

$P(x) \leftarrow \text{not } R(x), Q(x)$

$Q(x) \leftarrow S(x)$

$S(b)$

$R(a)$

Goal: $\leftarrow P(x)$ It calls $\leftarrow \text{not } R(x), Q(x)$ (*)

Not clear how to evaluate the **uninstantiated subgoal** $\text{not } R(x)$

In the previous examples the variable affected by *not* in the program was always instantiated when called

(Other LP paradigms, e.g. ASP, do not have this problems)

Some solutions, among others:

1. Change the order in (*): $\leftarrow Q(x), \text{ not } R(x)$
2. Delay the evaluation of negated atoms with variables until they are instantiated

These two solutions work if rules are **safe**, i.e. every variable in a negated atom (in a body) appears in a positive literal in the same body

3. Introduce a “domain predicate” and an extension for it, e.g.

$$P(x) \leftarrow \text{dom}(x), \text{ not } R(x), Q(x)$$

$$Q(x) \leftarrow S(x)$$

$$S(b)$$

$$R(a)$$

$$\text{dom}(a), \text{dom}(b), \dots$$

There are other LP paradigms that overcome these problems (cf. later)

Remark: The program rule $P(x, y) \leftarrow Q(x, y), \text{ not } S(x, y)$ can be seen as a view definition, of the difference $Q \setminus S$ in relational algebra (RA) of tables Q and S

In RA there is not “complement” operation

A rule $P(x, y) \leftarrow \text{not } S(x, y)$ would be like defining the complement of table S

Not a safe rule, and does not make sense in Prolog, or Datalog, its relational DB counterpart (coming in next chapter)

Exercise: Consider a relational referential constraint:

$$\forall x(R(x) \rightarrow \exists yS(x, y))$$

- (a) Define in relational calculus the “violation view” $V(x)$ for the IC, i.e. the one that contains the values for x in R that do not appear in S
- (b) Do the same in Prolog^{not} using safe rules

Example: Consider a relational referential constraint on a DB:

$$\forall x(R(x) \rightarrow \exists yS(x, y))$$

(a) We can define in relational calculus the “violation view” $V(x)$ for the IC, i.e. the one that contains the values for x in R that do not appear in S

By the query: $Q(x): R(x) \wedge \neg\exists yS(x, y)$

(b) Do the same in Prolog^{not} using safe rules: the database tuples as facts of the program plus

$V(x) \leftarrow R(x), \text{ “not } S(x, y)\text{”}$ Not clear meaning, not safe!

Better: $V(x) \leftarrow R(x), \text{ not } C(x)$ plus

$C(x) \leftarrow S(x, y)$ All safe now!

4. Another issue with Prolog

What about the “program”?

Male(x) ← Person(x), not Female(x)

Female(x) ← Person(x), not Male(x)

Person(a)

There is **recursion via negation**

(a) Try to determine *a*'s gender by hand, à la Prolog

(b) Try to use Prolog for (a)

You will see Prolog cannot handle this

We will see other LP paradigms in the following chapters ...

Prolog as a Programming Language

Prolog is a “Turing-complete” programming language

Any Turing machine can be simulated by a Prolog program

As a consequence, deciding if Prolog programs terminate is unsolvable

Prolog can be used as a full (declarative) programming language

Notice that we can introduce and specify functions

Just for the gist, we can:

- Define the natural numbers and arithmetic operations
(Prolog has them as built-in operations, but it is possible to define them from scratch)
- Do list processing, and take it from there (idem)

Example: (defining natural numbers and basic arithmetic in Prolog)

We can use the “free data-type constructors” 0 and $s(\cdot)$ (meant to be the successor function)

We will obtain that the natural number are exactly (no less and no more than):
 $0, s(0), s(s(0)), s(s(s(0))), \dots$

For this, in Math we use the induction principle (with its implicit minimality principle)

Here, we have recursion instead (and the implicit minimality provided by the evaluation methodology)

Defining natural numbers and verifying that one is ...

```
natural(0).
natural(s(X)) :- natural(X).

| ?- trace.

yes
{trace}
| ?- natural(s(s(s(0)))).
  1  1  Call: natural(s(s(s(0)))) ?
  2  2  Call: natural(s(s(0))) ?
  3  3  Call: natural(s(0)) ?
  4  4  Call: natural(0) ?
  4  4  Exit: natural(0) ?
  3  3  Exit: natural(s(0)) ?
  2  2  Exit: natural(s(s(0))) ?
  1  1  Exit: natural(s(s(s(0)))) ?

yes
{trace}
```


Finding natural numbers ... (with backtracking)

```
| ?- natural(X).  
  1  1  Call: natural(_69) ?  
  1  1  Exit: natural(0) ?  
  
X = 0 ? ;  
  1  1  Redo: natural(0) ?  
  2  2  Call: natural(_319) ?  
  2  2  Exit: natural(0) ?  
  1  1  Exit: natural(s(0)) ?
```

X = s(0) ?

yes

Now we can define the sum by appealing to the usual recursive definition:

$$\begin{aligned} \text{sum}(n, 0) &:= n \\ \text{sum}(s(n), m) &:= s(\text{sum}(n, m)) \end{aligned}$$

(also a ternary predicate can be used, with last argument storing the sum)

Defining sum and multiplication by recursion ...

```
% sum(?X,?Y,?Z).
% Z is the sum of X and Y.

sum(0,Y,Y).
sum(s(X),Y,s(Z)) :- sum(X,Y,Z).

% mult(+X,+Y,?Z).
% Z is the multiplication of X and Y.

mult(0,_,0).
mult(s(X),Y,Z) :- mult(X,Y,Z1), sum(Y,Z1,Z).

| ?- sum(s(s(0)),s(s(s(0))),N).
  1 1 Call: sum(s(s(0)),s(s(s(0))),_121) ?
  2 2 Call: sum(s(0),s(s(s(0))),_436) ?
  3 3 Call: sum(0,s(s(s(0))),_653) ?
  3 3 Exit: sum(0,s(s(s(0))),s(s(s(0)))) ?
  2 2 Exit: sum(s(0),s(s(s(0))),s(s(s(s(0)))) ?
  1 1 Exit: sum(s(s(0)),s(s(s(0))),s(s(s(s(s(0)))) ?

N = s(s(s(s(s(0)))) ?
```

yes

```

| ?- mult(s(s(0)),s(s(0)),N).
  1 1 Call: mult(s(s(0)),s(s(0)),_113) ?
  2 2 Call: mult(s(0),s(s(0)),_419) ?
  3 3 Call: mult(0,s(s(0)),_637) ?
  3 3 Exit: mult(0,s(s(0)),0) ?
  4 3 Call: sum(s(s(0)),0,_419) ? s
  4 3 Exit: sum(s(s(0)),0,s(s(0))) ?
  2 2 Exit: mult(s(0),s(s(0)),s(s(0))) ?
  7 2 Call: sum(s(s(0)),s(s(0)),_113) ? s
  7 2 Exit: sum(s(s(0)),s(s(0)),s(s(s(s(0)))) ?
  1 1 Exit: mult(s(s(0)),s(s(0)),s(s(s(s(0)))) ?

```

N = s(s(s(s(0)))) ?

yes

Example: (defining operations with lists)

(Prolog has a library with definitions for them, but we can start from scratch; it is also possible to start from an even lower level, defining the list data type)

```
% member(X,L): defines X as an element of list L.
```

```
member(X, [X|_]).
```

```
member(X, [_|L]) :- member(X,L).
```

```
| ?- member(c, [a,b,c,d]).
```

```
1 1 Call: member(c, [a,b,c,d]) ?
```

```
1 1 Call: member(c, [a,b,c,d]) ?
```

```
2 2 Call: member(c, [b,c,d]) ?
```

```
3 3 Call: member(c, [c,d]) ?
```

```
3 3 Exit: member(c, [c,d]) ?
```

```
2 2 Exit: member(c, [b,c,d]) ?
```

```
1 1 Exit: member(c, [a,b,c,d]) ?
```

```
| ?- member(X, [a,b,c,d]).
  1 1 Call: member(_69, [a,b,c,d]) ?
  1 1 Exit: member(a, [a,b,c,d]) ?
```

```
X = a ? ;
  1 1 Redo: member(a, [a,b,c,d]) ?
  2 2 Call: member(_69, [b,c,d]) ?
  2 2 Exit: member(b, [b,c,d]) ?
  1 1 Exit: member(b, [a,b,c,d]) ?
```

```
X = b ?
```

```
% append(L1,L2,L): L is the concatenation of lists L1 and L2.
```

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

```
| ?- append([a,b],[c,d,e],L).
  1 1 Call: append([a,b],[c,d,e],_107) ?
  2 2 Call: append([b],[c,d,e],_382) ?
  3 3 Call: append([], [c,d,e],_587) ?
  3 3 Exit: append([], [c,d,e], [c,d,e]) ?
  2 2 Exit: append([b], [c,d,e], [b,c,d,e]) ?
  1 1 Exit: append([a,b], [c,d,e], [a,b,c,d,e]) ?
```

```
L = [a,b,c,d,e] ?
```

```

| ?- append([a,b],X,[a,b,c,d,e]).
  1 1 Call: append([a,b],_85,[a,b,c,d,e]) ?
  2 2 Call: append([b],_85,[b,c,d,e]) ?
  3 3 Call: append([],_85,[c,d,e]) ?
  3 3 Exit: append([], [c,d,e], [c,d,e]) ?
  2 2 Exit: append([b], [c,d,e], [b,c,d,e]) ?
  1 1 Exit: append([a,b], [c,d,e], [a,b,c,d,e]) ?

```

```

X = [c,d,e] ?

```

```

% prefix(L1,L2): defines L1 as a prefix of list L2.

```

```

prefix([],_).
prefix([X|L1],[X|L2]) :- prefix(L1,L2).

```

```

% suffix(L1,L2): defines L1 as a suffix of list L2.

```

```

suffix(L,L).
suffix(L,[_|L1]) :- suffix(L,L1).

```

```

% sublist(L1,L2): defines L1 as a sublist of L2.

```

```

sublista(L1,L2) :- prefix(L3,L2), suffix(L1,L3).

```

```
% length(L,N): N is the length of list L.
```

```
length([],0).
```

```
length(_|L,N) :- length(L,M), N is M + 1.
```

```
| ?- length([a,b,c],N).
```

```
1 1 Call: length([a,b,c],_91) ?
```

```
2 2 Call: length([b,c],_359) ?
```

```
3 3 Call: length([c],_566) ?
```

```
4 4 Call: length([],_772) ?
```

```
4 4 Exit: length([],0) ?
```

```
5 4 Call: _566 is 0+1 ?
```

```
5 4 Exit: 1 is 0+1 ?
```

```
3 3 Exit: length([c],1) ?
```

```
6 3 Call: _359 is 1+1 ?
```

```
6 3 Exit: 2 is 1+1 ?
```

```
2 2 Exit: length([b,c],2) ?
```

```
7 2 Call: _91 is 2+1 ?
```

```
7 2 Exit: 3 is 2+1 ?
```

```
1 1 Exit: length([a,b,c],3) ?
```

```
N = 3 ?
```

Exercise: Write propositional formulas over a set of propositional variables $V = \{p1, \dots, p10\}$ as lists, e.g. $((p1 \vee \neg p3) \wedge p4)$ becomes the list $[(, (, p1, \vee, \neg, p3,), \wedge, p4,)]$

(a) Define a binary predicate $WF(L, A)$ that defines formula L (as a list) as a well-formed (i.e. legal) formula

For example, it should be: $WF([(, (, p1, \vee, \neg, p3,), \wedge, p4,)], ok)$, but $WF([(, (, p1, \vee], no)$

Hint: Remember the inductive definition of propositional formulas

(b) Use Prolog to define the length of a formula

(c) Use Prolog to define a predicate that returns in an output argument the list of propositional variables in the formula, e.g. it should be:

$PV([(, (, p1, \vee, \neg, p3,), \wedge, p4,)], [p1, p3, p4])$

Exercise: Instead of using lists, do the same as in the previous exercise, but now representing formulas as terms of Boolean algebra in prefix notation, e.g. $((p1 \vee \neg p3) \wedge p4)$ becomes: $\wedge(\vee(p1, \neg(p3)), p4)$

Example: (specifying QuickSort in Prolog)

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

% quicksort(+L,-LO).
% Predicate returns in LO the result of ordering list L.
% IMPORTANT: L has no duplicates.

quicksort([X],[X]).
quicksort(L,R):- pivot(L,P), two_lists(L,P,L1,L2), quicksort(L1,RL1),
                 quicksort(L2,RL2), append(RL1,RL2,R).

% pivot(+L,?P).
% X1 is the smallest element of L and X2 the greatest of L, this predicate
% returns in P the average of these two numbers.

pivot(L,X):- min_list(L,X1), max_list(L,X2), X is (X1+X2)/2.

% two_lists(+L,+P,?L1,?L2).
% Given a list L with numbers and a number P, this predicate returns in
% L1 the lists with all elements of L that are smaller than or equal to
% P, and in L2 the elements that are greater than P.

two_lists([],_,[],[]).
two_lists([X|R],P,[X|R1],L2) :- X=<P, two_lists(R,P,R1,L2).

two_lists([X|R],P,L1,[X|R2]) :- X>P, two_lists(R,P,L1,R2).
```

```

| ?- quicksort([3,2,7,6,1],X).
  1  1  Call: quicksort([3,2,7,6,1],_103) ?
  2  2  Call: pivot([3,2,7,6,1],_395) ?
  3  3  Call: min_list([3,2,7,6,1],_612) ?
  3  3  Exit: min_list([3,2,7,6,1],1) ?
  4  3  Call: max_list([3,2,7,6,1],_606) ?
  4  3  Exit: max_list([3,2,7,6,1],7) ?
  5  3  Call: _395 is(1+7)/2 ?
  5  3  Exit: 4.0 is(1+7)/2 ?
  2  2  Exit: pivot([3,2,7,6,1],4.0) ?
  6  2  Call: two_lists([3,2,7,6,1],4.0,_388,_389) ?
  7  3  Call: 3=<4.0 ?
  7  3  Exit: 3=<4.0 ?
  8  3  Call: two_lists([2,7,6,1],4.0,_2185,_389) ?
  9  4  Call: 2=<4.0 ?
  9  4  Exit: 2=<4.0 ?
 10  4  Call: two_lists([7,6,1],4.0,_2797,_389) ?
 11  5  Call: 7=<4.0 ?
 11  5  Fail: 7=<4.0 ?
 11  5  Call: 7>4.0 ?
 11  5  Exit: 7>4.0 ?
 12  5  Call: two_lists([6,1],4.0,_2797,_3408) ?
 13  6  Call: 6=<4.0 ?
 13  6  Fail: 6=<4.0 ?
 13  6  Call: 6>4.0 ?
 13  6  Exit: 6>4.0 ?
 14  6  Call: two_lists([1],4.0,_2797,_4018) ?

```

```

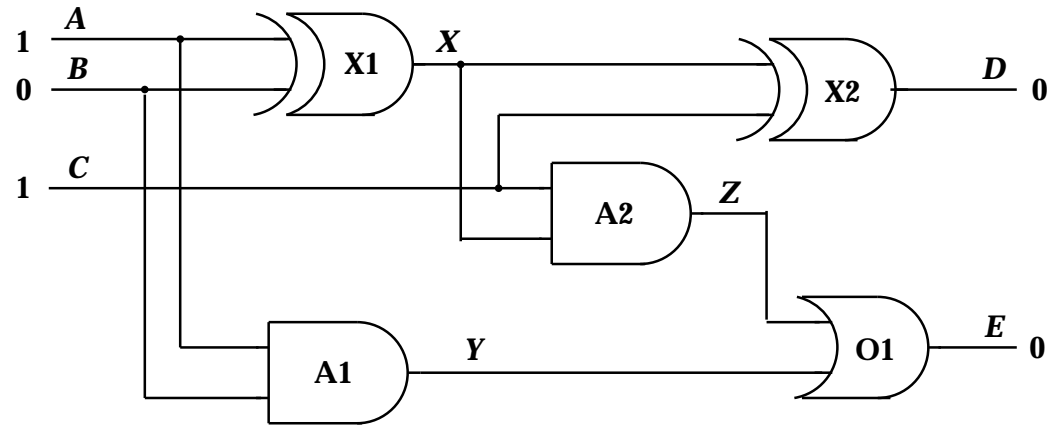
15 7 Call: 1=<4.0 ?
15 7 Exit: 1=<4.0 ?
16 7 Call: two_lists([],4.0,_4627,_4018) ?
16 7 Exit: two_lists([],4.0,[],[]) ?
14 6 Exit: two_lists([1],4.0,[1],[]) ?
12 5 Exit: two_lists([6,1],4.0,[1],[6]) ?
10 4 Exit: two_lists([7,6,1],4.0,[1],[7,6]) ?
8 3 Exit: two_lists([2,7,6,1],4.0,[2,1],[7,6]) ?
6 2 Exit: two_lists([3,2,7,6,1],4.0,[3,2,1],[7,6]) ?
17 2 Call: quicksort([3,2,1],_381) ?
18 3 Call: pivot([3,2,1],_6519) ? s
18 3 Exit: pivot([3,2,1],2.0) ?
22 3 Call: two_lists([3,2,1],2.0,_6512,_6513) ? s
22 3 Exit: two_lists([3,2,1],2.0,[2,1],[3]) ?
29 3 Call: quicksort([2,1],_6505) ?
30 4 Call: pivot([2,1],_7808) ? s
30 4 Exit: pivot([2,1],1.5) ?
34 4 Call: two_lists([2,1],1.5,_7801,_7802) ? s
34 4 Exit: two_lists([2,1],1.5,[1],[2]) ?
39 4 Call: quicksort([1],_7794) ?
39 4 Exit: quicksort([1],[1]) ?
40 4 Call: quicksort([2],_7788) ?
40 4 Exit: quicksort([2],[2]) ?
41 4 Call: append([1],[2],_6505) ?
41 4 Exit: append([1],[2],[1,2]) ?
29 3 Exit: quicksort([2,1],[1,2]) ?
42 3 Call: quicksort([3],_6499) ?

```

```
42 3  Exit: quicksort([3],[3]) ?
43 3  Call: append([1,2],[3],_381) ?
43 3  Exit: append([1,2],[3],[1,2,3]) ?
17 2  Exit: quicksort([3,2,1],[1,2,3]) ?
44 2  Call: quicksort([7,6],_375) ?
45 3  Call: pivot([7,6],_11171) ? s
45 3  Exit: pivot([7,6],6.5) ?
49 3  Call: two_lists([7,6],6.5,_11164,_11165) ? s
49 3  Exit: two_lists([7,6],6.5,[6],[7]) ?
54 3  Call: quicksort([6],_11157) ?
54 3  Exit: quicksort([6],[6]) ?
55 3  Call: quicksort([7],_11151) ?
55 3  Exit: quicksort([7],[7]) ?
56 3  Call: append([6],[7],_375) ?
56 3  Exit: append([6],[7],[6,7]) ?
44 2  Exit: quicksort([7,6],[6,7]) ?
57 2  Call: append([1,2,3],[6,7],_103) ?
57 2  Exit: append([1,2,3],[6,7],[1,2,3,6,7]) ?
1 1  Exit: quicksort([3,2,7,6,1],[1,2,3,6,7]) ?
```

X = [1,2,3,6,7] ?

A Diagnosis Example in Prolog



The Complete Adder (why “adder”?)

(this methodology is different from that seen in class)

```
/* A model-based diagnosis example. It relies on logic
   programming with NAF, which is denoted in SICSTUS
   Prolog by \+ */
```

```
/* Model of the Adder: */
```

```

/* The states (ok or abnormal) of the gates are represented
   by a term (functor) "state" which indicates with each one
   of its arguments the state of one of the gates */

/* Capitalized letters are variables, and lower case letters
   are constants */

/* Predicates and functors start all with lower case letters */

adder(state(X1,X2,A1,A2,O1),in(In1,In2,In3),out(Out1,Out2)) :-
    xorgate(X1,In1,In2,A),
    xorgate(X2,In3,A,Out1),
    andgate(A1,In1,In2,B),
    andgate(A2,In3,A,C),
    orgate(O1,B,C,Out2).

/* This clause captures the input/output relationships
   between gates */

/* Variables X1,X2,A1,A2,O1 will get values "ab" or "ok" */

/* Description of Normal Behaviour of the adder is given by the
   corresponding Boolean functions */

/* E.g. a normal xorgate behaves as the usual boolean function
   xor, etc. */

```

```

xorgate(ok, In1, In2, Out) :- xor(In1, In2, Out).
andgate(ok, In1, In2, Out) :- and(In1, In2, Out).
orgate(ok, In1, In2, Out) :- or(In1, In2, Out).

/* Description of usual Boolean functions: */
xor(1, 1, 0).
xor(1, 0, 1).
xor(0, 1, 1).
xor(0, 0, 0).

and(1, 1, 1).
and(1, 0, 0).
and(0, 1, 0).
and(0, 0, 0).

or(1, 1, 1).
or(1, 0, 1).
or(0, 1, 1).
or(0, 0, 0).

/* We need a model of abnormal behaviour. In general, we may
   specify as abnormal any behaviour that is not normal; in
   this sense it is a weak model of failure */

xorgate(ab, In1, In2, Out) :- \+ xor(In1, In2, Out).

andgate(ab, In1, In2, Out) :- \+ and(In1, In2, Out).

orgate(ab, In1, In2, Out) :- \+ or(In1, In2, Out).

```


(These non-safe bodies right above should not be a problem if they are called instantiated all the time, with values coming from the heads)

A run in search for a diagnosis when input is $A = 1, B = 0, C = 1$, and output is $D = 1, E = 0$

With $\leftarrow \text{adder}(\text{State}, \text{in}(1, 0, 1), \text{out}(1, 0))$, that uses variable *State*, we ask about the state of the circuit's components

```
bertossi@malloco [1] prolog SICStus 2.1 #6: Fri Nov 6 11:27:56 CDT
1992
```

```
| ?- ['diagnostico.pro']. {consulting
/tmp_mnt/dcc/ing/dcc/profes/bertossi/programas/diagnostico.pro...}
{/tmp_mnt/dcc/ing/dcc/profes/bertossi/programas/diagnostico.pro
consulted, 70 msec 3936 bytes}
```

```
yes | ?- adder(State, in(1,0,1), out(1,0)).
```

```
State = state(ok,ab,ok,ok,ab) ? ;
```

```
no | ?- halt.
```

```
bertossi@malloco [2]
```

This answer means that the gates are: $X1 = ok, X2 = ab, A1 = ok,$
 $A2 = ok, O1 = ab$, and this is the only answer



**Chapter 9: Semantics of Definite Programs, Datalog and its
Extensions**

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

Semantics of Definite Programs

- We haven't said much so far about the semantics of definite programs
- Even less about their extensions with weak, non-monotonic negation
- We have rather seen evaluation mechanisms: top-down, resolution, NAF (which is procedural)
- We want a “**model-theoretic**” semantics:

Given a program, we want to specify or **characterize what are the intended set-theoretic structures the program is about** (or describing)
- We will start with the semantics of definite programs, and set-theoretic structures of a very particular kind
- Extensions for other kinds of programs, e.g. with non-monotonic negation will be presented too (but mainly in the next chapter)

Herbrand Structures for FO Predicate Logic

Example: Consider the FO language $L(S)$ for predicate logic determined by the set of symbols $S = \{R(\cdot, \cdot), =, a, b, c\}$

We have seen that interpretation structures are of the form

$$\mathfrak{A} = \langle A, R^A, =^A, a^A, b^A, c^A \rangle$$

$$(A \neq \emptyset, R^A \subseteq A \times A, a^A, \dots \in A, \text{ and } =^A \text{ the diagonal of } A)$$

From now on we will consider **structures of a particular kind** for a given language

The so-called “Herbrand structures” (HSs), which are of a “syntactic nature”, directly determined by the language at hand

Herbrand structures for the language $L(S)$ based on $S = \{R(\cdot, \cdot), =, a, b, c\}$

A HS: $\mathcal{I}_1 = \langle H, I_1, a, b, c \rangle$ with:

1. $H = \{a, b, c\}$ is the **Herbrand universe**, whose elements are all the **ground terms** of the language, i.e. those terms that do not contain variables

The Herbrand universe is a **symbolic domain**, determined directly by the language

2. $I_1 := \{R(a, a), R(b, b), R(c, c), R(a, b), R(b, c)\}$ is the interpretation for R (i.e. $(a, a), (b, b), \dots \in R^{I_1}$)

Formed by a set of **ground atomic formulas** of the language (with predicates other than $=$)

I_1 contains all the **ground atoms that are true** (the others are false, as with the CWA; e.g. $R(a, c)$ is false)

3. Symbol “=” is interpreted as **syntactic equality**:

$a = a, b = b, c = c$, but **not** $a = c$, etc.

4. a, b, c (as elements of H) are the interpretations for themselves (as constants of the language)

For a given language, all HSs have the same universe H

They differ only wrt. 2. above, i.e. the ground atoms that they designate as true

Another HS: $\mathcal{I}_2 = \langle H, I_2, a, b, c \rangle$, defined as above, but now:

$I_2 := \{R(a, a), R(b, b), R(c, c), R(a, b), R(b, c), R(b, a), R(c, b),$
 $R(a, c), R(c, a)\}$

Herbrand structures are semantic interpretation structures that are directly determined by the syntax of the language

Similar to relational databases, which serve as interpretations for their own schema

Like a relational database, seen as symbolic KB, that seems to be talking directly about the objects mentioned in the database

For example, I_2 could be seen as a relational table showing exactly what is true of R :

R	A	B
	a	a
	b	b
	c	c
	a	b
	b	c

It is natural to consider Herbrand interpretations, because the sentences “seem” to be talking about the objects mentioned in them a, b, c (and not about their names only)

It can be proved that for many purposes it is good enough to restrict the semantics to HSs, especially for universal KBs, such as definite logic programs

We will do so in the rest of this chapter

Example: (cont.)

$$\mathcal{I}_1 \not\models \forall x \forall y (R(x, y) \rightarrow R(y, x))$$

$$\mathcal{I}_2 \models \forall x \forall y (R(x, y) \rightarrow R(y, x))$$

\mathcal{I}_2 is a **Herbrand model** of the sentence

Example: FO language based on $S = \{R(\cdot, \cdot), P(\cdot), f(\cdot), a, b\}$

All HSs have the same infinite Herbrand universe formed by all ground terms

$$H = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), \dots\}$$

In all of them, the constants a, b are interpreted by themselves

In all of them equality of terms is interpreted as **syntactic equality**, i.e. identity as strings, e.g.

$a = a, f(a) = f(a), \dots$ are true, but

$a = b, a = f(b), f(a) = f(b), \dots$ are false

(In non-Herbrand interpretation structures, constants a and b could be interpreted as the same element in the domain; but not in HSs ...)

The function symbol f is interpreted in the obvious and fixed way: the term \mathbf{t} ($\in H$) is mapped to the new term $f(\mathbf{t})$ (also $\in H$)



As a consequence, different Herbrand structures for a language may differ only in the interpretations of predicates (here, R and/or P)

For a fixed language, each HS can be represented as (and identified with) a subset of the **Herbrand base** (HB): the **collection of all possible (ground) atomic sentences** based on R and P

Here: $HB = \{R(a, a), R(a, b), R(b, a), R(b, b), R(f(a), a), \dots, P(a), P(b), P(f(a)), P(f^2(a)), P(f(b)), \dots\}$

For a particular HS, the predicates R, P are interpreted by choosing a subset of HB

Those atoms caught in the subset are the only true ones among those atomic sentences, e.g.

$$I_1 := \{R(a, b), R(a, f(a)), R(b, f(a)), P(a), P(f^3(b))\} \subseteq HB$$

It determines the HS $\langle H, I_1, f, a, b \rangle$

Similarly: $I_2 := \{R(f^3(a), f^2(b)), R(b, b), P(f^5(b))\}$

It determines the HS $\langle H, I_2, f, a, b \rangle$

The only distinctive part of a HS is the chosen subset of the HB

So, we identify the HS with the corresponding subset of the HB

E.g. we say “the HS I_1 ...”

Herbrand structures are important in databases, deductive databases, logic programming, automated reasoning, etc.

HSs are used to give semantics to many different logic-based KR languages

We will do so in the following

Semantics of Definite Logic Programs (revisited)

Remember the first class of logic programs we had: Positive Horn clauses, no negation

We gave top-down, backward-propagation evaluation methodologies for them (not necessarily Prolog's methodology)

Something like a specific “operational semantics”

Now we give a precise, “model-theoretic” or “model-based” semantics to them

In terms of the intended models of the specification (programs)

By definition, what is true wrt. the program is what is true in the intended models of the program

This idea can be applied to all kinds of logic-based specifications

Example: Program Π

$$\text{path}(x, z) \leftarrow \text{arc}(x, y), \text{path}(y, z)$$
$$\text{path}(x, x) \leftarrow$$
$$\text{arc}(b, c) \leftarrow$$

What is the **semantics** of Π ?

What world is Π describing?

Is there an **intended model** for Π ?

We concentrate on the Herbrand models (HMs) of Π

A **Herbrand structure** for Π :

- Has the **Herbrand universe** H (for Π) as its domain:
 $H = \{b, c\}$
- Interpretation of predicates determined by a subset of the **Herbrand Base** (HB) or Π

Here, $HB(\Pi) = \{arc(b, b), arc(c, c), arc(b, c), arc(c, b),$
 $path(b, b), path(c, c), path(b, c), path(c, b)\}$

Contains all the possible ground atomic propositions that can be potentially true

- The terms of the language are interpreted by themselves (seen above)

A **Herbrand model** (HM) for Π is a HS for Π that makes **all** the rules (clauses) in Π true

More precisely, we consider (at least conceptually) all the **ground instantiations** of the program rules on H

$$\begin{aligned} \textit{path}(b, b) &\leftarrow \textit{arc}(b, b), \textit{path}(b, b) \\ \textit{path}(b, c) &\leftarrow \textit{arc}(b, b), \textit{path}(b, c) \\ &\dots \leftarrow \dots \\ \textit{path}(b, b) &\leftarrow \\ \textit{path}(c, c) &\leftarrow \\ \textit{arc}(b, c) &\leftarrow \end{aligned}$$

For a HS, say $M_2 = \{\textit{arc}(b, c), \textit{path}(b, b), \textit{path}(c, c), \textit{path}(b, c)\}$, a ground rule, say $\textit{path}(b, c) \leftarrow \textit{arc}(b, c), \textit{path}(c, c)$ is true (or satisfied by M_2) iff when all the atoms in the body are true, i.e. belong to M_2 , then also the head belongs to M_2

M_2 does make this rule true: $M_2 \models (\textit{path}(b, c) \leftarrow \textit{arc}(b, c), \textit{path}(c, c))$

Similarly: $M_2 \models (\textit{path}(b, c) \leftarrow \textit{arc}(b, b), \textit{path}(b, c))$

- An HM of the program (a subset of HB):

$$M_1 = \{arc(b, b), arc(c, c), arc(b, c), arc(c, b), path(b, b), path(c, c), path(b, c), path(c, b)\}$$

- Another HM

$$M_2 = \{arc(b, c), path(b, b), path(c, c), path(b, c)\}$$

- Yet another HM

$$M_3 = \{arc(b, c), path(b, b), path(c, c), path(b, c), path(c, b)\}$$

(last atom not “justified” by first rule, but still that implication is true)

- A HS for Π that is **not an HM** of Π

$$M_4 = \{arc(b, c), path(c, c), path(c, b)\}$$

For a fixed language (or program), HSs, as subsets of HB , can be compared by set inclusion

\subseteq is a partial order in the class of subsets of HB

The same applies to HMs, i.e. HSs that are also models of Π

In the example above, M_2 is a **minimal** HM (of Π)

M_2 is an HM of Π and no proper subset of M_2 is an HM of Π

Example: Consider the definite program

$$\begin{aligned} R(x) &\leftarrow P(x). \\ P(a). \\ Q(b). \end{aligned}$$

The following set of ground tuples

$$\mathcal{M}_1 = \{P(a), Q(b), R(a), R(b)\}$$

describes (represents) a possible state of the world; a possible set of extensions for the “extensional” relations (here, P and Q are explicitly given) and “intentional” relations (here, R , a defined view)

It says that those tuples in \mathcal{M}_1 are true and nothing else, e.g. that $P(a)$ is true, but not $P(b)$

This is **model** of the program, because it makes all the rules above true (check this!)

There are models and non-models:

Models

- $\{P(a), Q(b), R(a), R(b), P(b)\}$
- $\{P(a), Q(b), R(a), R(b)\}$
- $\mathcal{M}_0 = \{P(a), Q(b), R(a)\}$
- ...

Non-models

- $\{P(a), R(a), R(b), P(b)\}$
- $\{P(a), Q(b), R(b)\}$
- $\{Q(b), R(a), R(b)\}$
- ...

\mathcal{M}_1 is a model, but in it $R(b)$ is unnecessarily true

$R(a)$ is forced to be true (if we want to satisfy the rules), but not $R(b)$

Instead, \mathcal{M}_0 contains exactly what is necessary to make the program true

It is (set-theoretically) contained in any other model of the program

\mathcal{M}_0 gives the meaning (semantics) to the program

More generally: We are comparing models (sets of ground atoms) by set inclusion, which is a partial order

A partial order may have none, one or several minimal elements

However:

Theorem: A definite program Π has exactly **one minimal HM**, denoted $\underline{M}(\Pi)$

By definition, the semantics of Π is given by $\underline{M}(\Pi)$

What is true wrt to Π is what is true in $\underline{M}(\Pi)$

In particular, for a ground atom A :

A is true wrt Π (denoted $\Pi \models_{mm} A$) $:\iff A \in \underline{M}(\Pi)$

Connection with what we had before?

More generally: We are comparing models (sets of ground atoms) by set inclusion, which is a partial order

A partial order may have none, one or several minimal elements

However:

Theorem: A definite program Π has exactly **one minimal HM**, denoted $\underline{M}(\Pi)$

By definition, the semantics of Π is given by $\underline{M}(\Pi)$

What is true wrt to Π is what is true in $\underline{M}(\Pi)$

In particular, for a ground atom A :

A is true wrt Π (denoted $\Pi \models_{mm} A$) $:\iff A \in \underline{M}(\Pi)$

Connection with what we had before?

Theorem: The ground atoms $A \in \underline{M}(\Pi)$ are exactly those that can be proved by resolution-based refutations from Π

The model-theoretic semantics and the procedural (deductive) semantics coincide (for definite programs, in the sense of the theorem)!

Remember that Prolog with its evaluation strategy was also minimal in terms of what it considered to be true

Now we are seeing the same from a different perspective

Any way to compute the minimal model of the program other than via top-down, backward-chaining refutation?

What about bottom-up with forward-chaining?

Example: Program Π

$$\begin{aligned}r(x) &\leftarrow p(x) \\p(x) &\leftarrow q(x, y) \\q(a, a) &\leftarrow \\q(a, b) &\leftarrow\end{aligned}$$

Ground instantiation Π_H

of Π (on the H-universe):

We are left with an essentially propositional program that we can use for all purposes instead of the original one

$$\begin{aligned}r(a) &\leftarrow p(a) \\r(b) &\leftarrow p(b) \\p(a) &\leftarrow q(a, a) \\p(a) &\leftarrow q(a, b) \\p(b) &\leftarrow q(b, b) \\p(b) &\leftarrow q(b, a) \\q(a, a) &\leftarrow \\q(a, b) &\leftarrow\end{aligned}$$

The minimal model $\underline{M}(\Pi)$ of the original program can be obtained bottom-up, by propagating the facts through the rules, from right to left (forward-propagation), iteratively:

First step: $q(a, a), q(a, b) \in \underline{M}(\Pi)$

Second step: $p(a) \in \underline{M}(\Pi)$

Third step: $r(a) \in \underline{M}(\Pi)$

A fix-point has been reached; nothing new is obtained

$$\underline{M}(\Pi) = \{q(a, a), q(a, b), p(a), r(a)\}$$

This is general, even with recursion: **The minimal model of a definite program can be obtained as the fix-point of the bottom-up evaluation we just described**

(Producing the ground instantiation first is not necessary; actually not done for this purpose)

Datalog

Datalog programs are definite LPs without function symbols

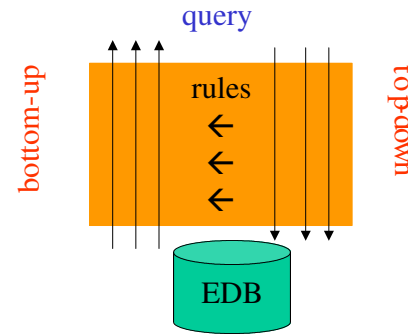
Datalog is used as a language to express queries and define view to/on a relational database (RDB)

Datalog is used in RDBs to define new predicates (views), possibly recursive ones, on top of the RDB, which is the **extensional database** (EDB) that provides the facts for the program

It is meant to be used with large sets of facts (i.e. tuples in the underlying relational tables)

(In “definite logic programming” we may have few facts and large sets of rules)

Datalog uses bottom-up evaluation



Definite logic programming uses top-down evaluation

In general, bottom-up evaluation is better (faster) for database applications, where data may be massive

Top-down with backtracking is too slow for those applications

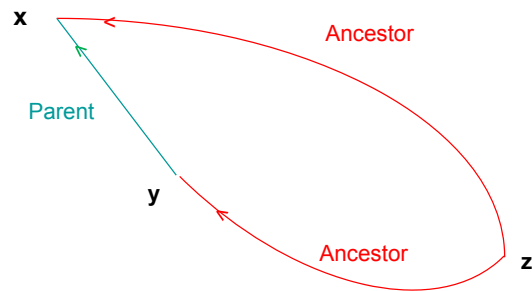
Example:

$$Q(x) \leftarrow \textit{Ancestor}(aa, x) \quad (1)$$

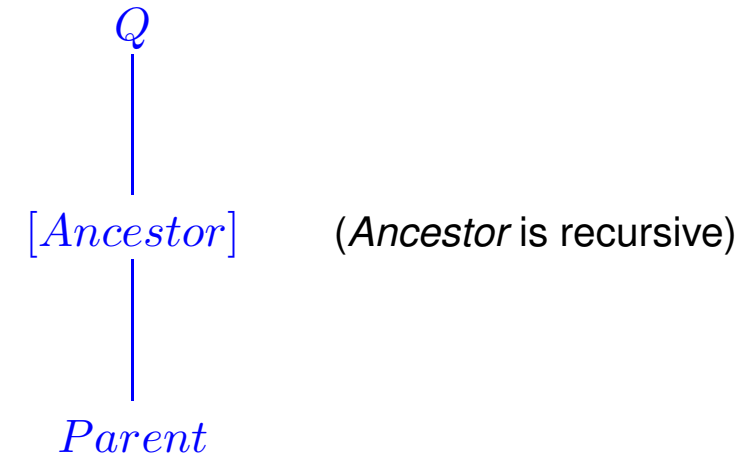
$$\textit{Ancestor}(x, y) \leftarrow \textit{Parent}(x, y) \quad (2)$$

$$\textit{Ancestor}(x, y) \leftarrow \textit{Parent}(x, z), \textit{Ancestor}(z, y) \quad (3)$$

Facts: $\textit{Parent}(a, aa), \textit{Parent}(a, ab), \textit{Parent}(aa, aaa),$
 $\textit{Parent}(aa, aab), \textit{Parent}(aaa, aaaa), \textit{Parent}(c, ca)$



Dependency Graph:



Like having a selection/projection query (the first rule) on top of a recursively defined view; in its turn defined on top of a relational database

Computation:

1. Initialize *Ancestor* and Q (query answer collector predicate) as empty:

$$\textit{Ancestor} = \emptyset \quad Q = \emptyset$$

2. View *Ancestor* needs to be computed

$$\text{First } \textit{Ancestor} = \emptyset$$

Apply rule (2) once, obtaining by forward propagation:

$$\textit{Ancestor} = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca)\}$$

This is a **partial** computation of the view

3. Apply (3) with the tuples obtained in the previous step as input for the right-hand side, and propagate to the head

That is, perform the join $Parent \bowtie \text{“Ancestor”}$, where “Ancestor” is only a partial version of the real (final) *Ancestor*

Newly generated tuples for *Ancestor*:

$$(a, aaa), (a, aab), (aa, aaaa)$$

New state:

$$Ancestor = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaa)\}$$

4. Since new tuples were generated wrt 1., apply rule (3) again, with the partial extension for *Ancestor* as input, from right to left (forwards)

Newly generated tuples for *Ancestor*:

$$\underline{(a, aaa), (a, aab), (aa, aaaa)}, (a, aaaa)$$

The underlined tuples were recomputed!

New state: $Ancestor = \{(a, aa), (a, ab), (aa, aaa), (aa, aab), (aaa, aaaa), (c, ca), (a, aaa), (a, aab), (aa, aaaa), (a, aaaa)\}$

5. Since new tuples were generated, apply rule (3) once more

Generated tuples for *Ancestor*: $(a, aaa), (a, aab), (aa, aaaa), (a, aaaa)$

6. **No new tuples were obtained** (redundant recomputation!);

same state

Block for *Ancestor* is completely computed

7. Now compute the extension of Q applying its defining rule (a selection followed by a projection)

Generated tuples: $Q = \{aaa, aaaa, aab\}$

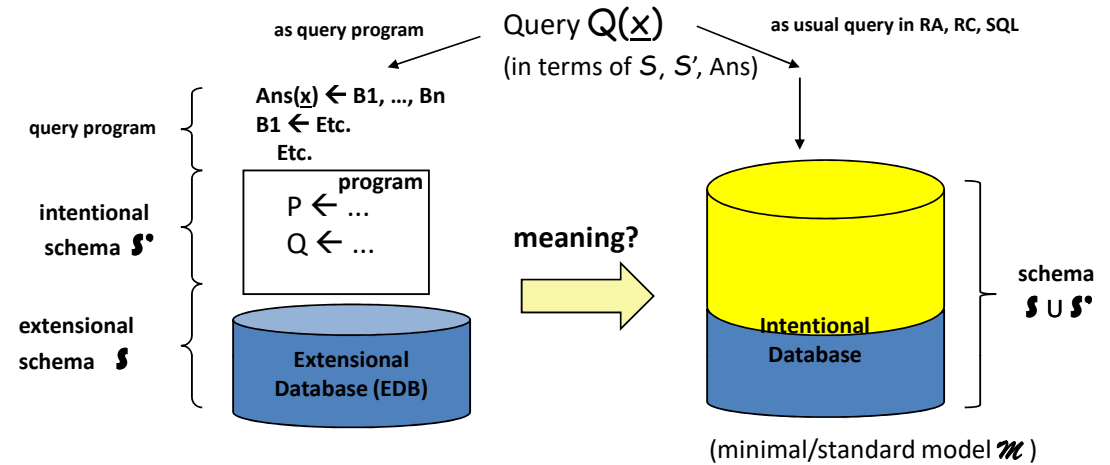
Like computing the minimal model for the rules (2)-(3) plus the facts, materializing it, and posing the top query (1) to the minimal model

The same minimal model can be used for different top queries

The minimal model can be computed in polynomial time in the size of the EDB
 i.e. in **data complexity** (i.e. varying the EDB, but keeping the program fixed)

Like creating a new database that extends EDB and querying it as usual

Alternatively, one can pose queries to the new database, keeping it virtual, without materializing it first and posing a usual query next



Instead, one can pose the query directly on top of the program that defines the extension, as an extra, top layer

Propagating upwards data that are relevant and necessary to answer the query

In the example above all data are moved upwards, to the top query level

There is recomputation of tuples and computation of “irrelevant” tuples:

- The top selection may discard at the end too many computed tuples
- In the previous example, all computations before last step (7.) were done without considering the parameter *aa* in the query

Many tuples were carried to the upper level and then filtered out: too much useless computation

Top-down mechanisms are more sensitive to the parameters (values), the predicates, and the structure of the query

In general, top-down mechanisms are more “focused” on the relevant data, but less efficient for DB applications

With bottom-up evaluation we get all the answers at once

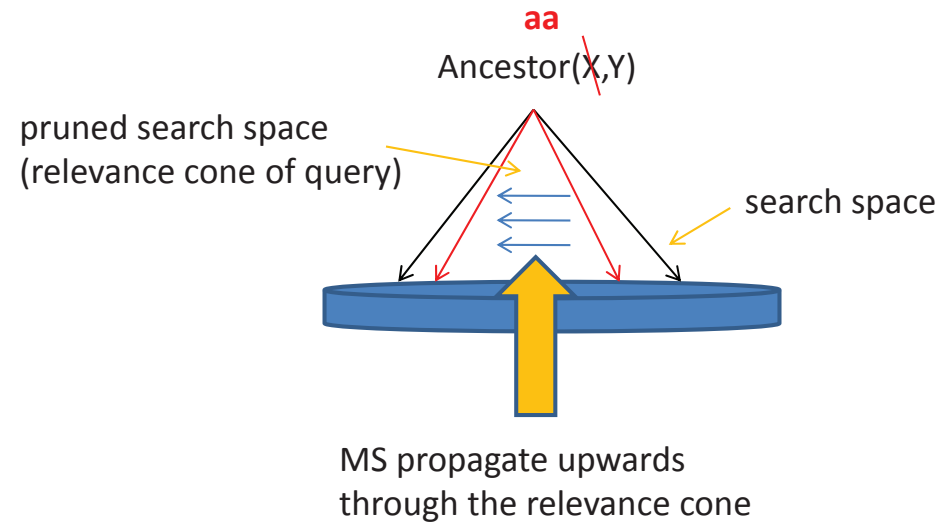
Idea: Get the best of the two worlds

The **Magic Sets Method** (MSM) is an optimization methodology for (query evaluation with) Datalog programs

It makes the computation more sensitive/suited to the parameters in the query, and the relations and rules that are relevant to the query

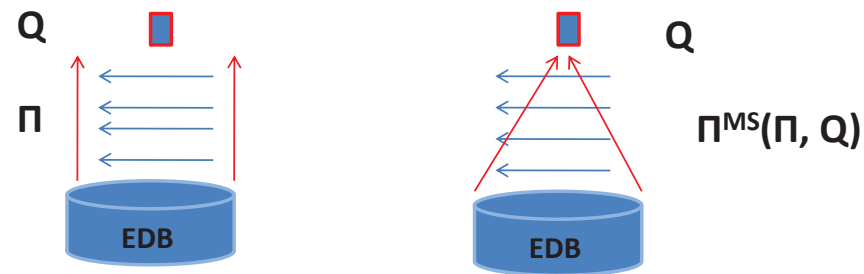
Only “the relevant data” for the particular query in the extensional DB are considered

MSM is a bottom-up query evaluation mechanism that simulates a top-down approach (in terms of the data that are used and how)



Using the query, **syntactically rewrite** the program in to a new one, the associated *magic program* that can be better evaluated bottom-up

$$\pi, Q \longmapsto \pi^{MS}(\pi, Q) \text{ (magic program)}$$



The new program has additional predicates to impose more and adequate conditions (related to the structure and parameters in the query)

The new program is used for bottom-up evaluation as before, but through a narrower “cone”

In different forms and with different limitations, MSM are implemented in DBMS

Recursion in SQL3

Datalog is built-in in the newer standards of SQL

In particular, SQL3 allows to define recursive views

In a RDBMS using SQL3, we want to answer queries like

Example:

1. Relation *ParentOf(parent,child)*

Query: Find all ancestors of Mary

2. “Explosion of Parts”

Relations: *PartOf(part#,subpart#)*, *Cost(part#,price)*

Query: What is the total cost of part #123?

Each part consists of subparts, each subpart of subsubparts, etc.

Command `WITH` in SQL3:

To create “temporary” views; think of rules in a Datalog program

```
WITH R1 AS (query),  
      R2 AS (query),  
      ... ,  
      Rn AS (query)
```

`<query>` (involving R1, R2, ..., Rn and other relations)

Idea:

1. Compute R1, R2, ..., Rn in temporary relations
2. Evaluate the query that involves R1, ..., Rn and other relations
3. Destroy R1, R2, ..., Rn

It is also possible to specify schemas for the R_i 's, their attribute names

```
WITH R1 (A1, A2, ..., Am) AS (query), ...
```

Example: Students applying to both Berkeley and Santa Cruz

```
WITH      Berk AS
          (SELECT ID, date
           FROM Apply
           WHERE location = "Berkeley"),
          SC AS
          (SELECT ID, date
           FROM Apply
           WHERE location = "SC")
SELECT    ID, Berk.date AS Bdate,
          SC.date AS SCdate
FROM      Berk, SC
WHERE     Berk.ID = SC.ID
```

Assertion `WITH` can be seen as a “temporary view definition” for syntactical convenience

- **BUT:** the R_i 's can be recursive or mutually recursive
- They need the keyword `RECURSIVE`
- The recursion is achieved through the `UNION` of the base case plus the properly recursive case

In the case of Datalog, this is achieved by means of a rule for the base case, and another for the recursive case (hence a union)

Example: Ancestors of Mary given the table ParentOf (parent, child) ?

```
WITH RECURSIVE Ancestor(anc, desc) AS
  ((SELECT parent AS anc, child AS desc
    FROM ParentOf)
  UNION
  (SELECT Ancestor.anc,
    ParentOf.child AS desc
    FROM Ancestor, ParentOf
    WHERE Ancestor.desc = ParentOf.parent))
SELECT  anc
FROM    Ancestor
WHERE   desc = "Mary";
```

Notice that the definition contains the base case, and the properly recursive case

The union corresponds to the two rules used to define the new predicate

Example: Total cost of part #123 from PartOf (part#, subpart#)
and Cost (part#, price)?

```
WITH RECURSIVE AllParts AS
      ( (SELECT * FROM PartOf)
        UNION
        (SELECT A1.part#, A2.part#
         FROM AllParts A1, AllParts A2
         WHERE A1.subpart# = A2.part#) )
SELECT sum(Cost.price)
FROM AllParts, Cost
WHERE AllParts.part# = 123
      AND AllParts.subpart# = Cost.part#
```

Disjunctive, Negation-Free Programs

We now consider more expressive FO logic programs containing (non-Horn) clauses of the form

$$A_1 \vee \dots \vee A_n \longleftarrow B_1, \dots, B_m$$

where the A_i, B_j are **atoms** (implicit universal quantifiers in front)

Example:

$$P(x) \vee Q(x) \longleftarrow R(x), S(x, y).$$

$$R(x) \longleftarrow U(x), V(x).$$

$$S(x, y) \longleftarrow U(x), V(y).$$

$$U(a). U(c). V(b). V(a).$$

Herbrand Universe: $H = \{a, b, c\}$

Herbrand Base: $HB =$ all possible ground atoms

The notion of Herbrand model is almost as before: when the body becomes true, at least one of the disjuncts in the head has to be true

Now, **two minimal (Herbrand) models**:

$$M_1 = \{U(a), U(c), V(b), V(a), S(a, a), S(a, b), S(c, a), S(c, b), R(a), P(a)\}$$

$$M_2 = \{U(a), U(c), V(b), V(a), S(a, a), S(a, b), S(c, a), S(c, b), R(a), Q(a)\}$$

Disjunctive programs may have more than one minimal model

In general, there will be as many branching points to build (minimal) models as disjunctions in heads

Under this “minimal model semantics”, disjunctions in heads are exclusive (unless otherwise implied by other rules)

(The program: $P(x) \vee R(x) \leftarrow T(x)$. $P(x) \leftarrow T(x)$. $R(x) \leftarrow T(x)$. $T(a)$. is forced to make both $P(a)$ and $R(a)$ true)

What are the logical consequences of such a program?

Cautious, certain or skeptical semantics: What is true in all minimal models

Here, $R(a)$ is certainly true, but not $P(a)$

Brave or possible semantics: What is true in some minimal model

$P(a), Q(a)$ are possibly true (but not certainly true)

Normal Programs

Now programs with **one atom in the head**, **weak negation** (*not*) in the body

Rules may be of the form:

$$A \leftarrow A_1, \dots, A_m, \textit{not} A_{m+1}, \dots, \textit{not} A_n$$

where the A, A_i are atoms

We want a **general**, declarative, model-based semantics

This negation is not the “procedural” NAF we saw in top-down, resolution-based refutations (as in Prolog)

(In some cases the not above can be interpreted and handled as that NAF, but not always)

We keep using HSs, and let's revisit the notion of (Herbrand) model of a program

For the program Π with rules as above, assume that

$$A \leftarrow A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_n$$

is a ground instantiation of one of Π 's rules, and $M \subseteq HB(\Pi)$ is a HS

M makes the rule true if:

Whenever $A_1, \dots, A_m \in M$, and none of A_{m+1}, \dots, A_n belong to M ,
then $A \in M$

M is a (Herbrand) model of Π if it makes true all the ground instantiations of rules in Π

Example: $\Pi: P(x) \leftarrow Q(x), \text{not } R(x).$
 $Q(a). Q(b). R(b).$

Ground program: $\Pi^g: P(a) \leftarrow Q(a), \text{not } R(a).$
 $P(b) \leftarrow Q(b), \text{not } R(b).$
 $Q(a). Q(b). R(b).$

For $M_1 = \{Q(a), Q(b), R(b), P(a), P(b)\}$, $M_1 \models \Pi$

For $M_2 = \{Q(a), Q(b), R(b), P(a)\}$, $M_2 \models \Pi$

For $M_3 = \{Q(a), Q(b), R(b), R(a)\}$, $M_3 \models \Pi$

For $M_4 = \{Q(a), Q(b), R(b), P(b)\}$, $M_4 \not\models \Pi$

Notice: M_2 and M_3 are (mutually incomparable) minimal models

What is (are) the intended model(s) of a program like this?

Example: Normal program Π

$$D(x) \leftarrow Q(x), \text{ not } R(x)$$

$$R(x) \leftarrow S(x)$$

$$R(x) \leftarrow H(x), \text{ not } S(x)$$

$$S(x) \leftarrow T(x, y), \text{ not } U(y)$$

$$S(x) \leftarrow U(x)$$

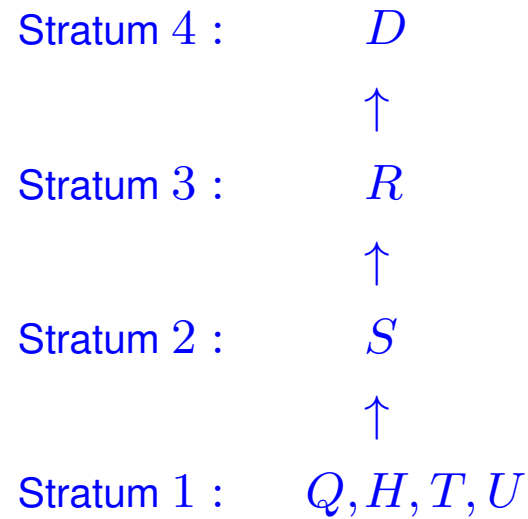
$$Q(a). Q(b). Q(c). H(b). T(a, b).$$

This program (so is the previous one) is **stratified**: there is natural hierarchy of predicates wrt. their definitions; they can be placed in **strata**

In a stratified program each predicate can be completely computed without calling via negation other predicates that are not at a lower stratum

In other words, if a predicate P is placed in a stratum i and its definition calls (in the body) a subgoal $\text{not } Q(c)$, then Q is placed at stratum k with $k < i$

We define (and compute) a model bottom-up guided by the stratification:



Compute their extensions
bottom-up, fully computing
each stratum before moving
to the next one above

1. $M_1 = \{Q(a), Q(b), Q(c), H(b), T(a, b)\}$
2. $M_2 = \{S(a)\}$
3. $M_3 = \{R(a), R(b)\}$
4. $M_4 = \{D(c)\}$

We get only one model in this way: $M = M_1 \cup M_2 \cup M_3 \cup M_4$

Notice that the evaluation of each rule body of in the program happens on top of what has been collected so far at lower strata

Each rule evaluation becomes query evaluation that can be performed by means of relational algebra (or calculus)

For example, the evaluation of rule

$$S(x) \longleftarrow T(x, y), \text{ not } U(y)$$

on M_1 becomes the evaluation of:

$$S(x) \longleftarrow T(x, y)$$

on M_1 (no data for U)

This becomes a projection query, on the first attribute of T

The evaluation of the second rule defining R becomes the evaluation of the relational set difference $H \setminus S$ on $M_1 \cup M_2$

Finally, the evaluation of $\textit{not } R(x)$ in the body of the first rule (for D) becomes verifying if:

$$R(a) \in M_1 \cup M_2 \cup M_3, \quad R(b) \in M_1 \cup M_2 \cup M_3, \\ R(c) \in M_1 \cup M_2 \cup M_3$$

Only the latter becomes false (and then $\textit{not } R(c)$ true), because $R(c)$ is not explicitly in $M_1 \cup M_2 \cup M_3$

That is, \textit{not} as a sort of local CWA for relational databases

Again, in some sense, a form of NAF: failure to find the atom in the model so far

Hence a weak (or non-monotonic) negation (see next slide)

The motivation for defining stratified programs is about characterizing a class of programs with negation that are well-behaved in terms of semantics and its computation

Those programs do not have recursive definitions involving negation

It is not about not having both negation and recursion, but only the good combinations

This program is still stratified; it mixes recursion and negation, but in a stratified manner: every negative call to a predicate has the latter fully computed already

$$P(x) \leftarrow R(x), \text{not } Q(x)$$
$$Q(x) \leftarrow R(x)$$
$$R(x) \leftarrow Q(x), \text{not } S(x)$$
$$S(a)$$

The semantics of stratified programs is given by the (single) **standard model**

The one computed bottom-up as above, which turns out to be a minimal model (possibly among others)

So, no proper subset of the standard model is a model of the program

This semantics is non-monotonic, i.e. adding new facts may invalidate previous consequences (a good reason to use *not* instead of classical negation \neg)

Example: (ex. page 45) $\Pi^g : P(a) \leftarrow Q(a), \text{not } R(a).$
 $P(b) \leftarrow Q(b), \text{not } R(b).$
 $Q(a). Q(b). R(b).$

$M_2 = \{Q(a), Q(b), R(b), P(a)\}$ is the standard model; and $\Pi \models_{\text{st-mod}} P(a)$

Now add $R(a)$ to the facts

With new standard model $M' = \{Q(a), Q(b), R(b), R(a)\}$: $\Pi' \not\models_{\text{st-mod}} P(a)$

So as the minimal model of a Datalog program, the standard model can be computed in polynomial time in the size of the extensional database at the bottom level

The SQL standard -and commercial relational DBMSs- allow for recursive view definitions, but only stratified negation in recursive definitions

That is, relational DB technology (or the current SQL standard) does not go beyond stratified view definitions (programs)

View definitions with disjunctive heads not allowed

Stratified programs extend an underlying, extensional, relational database in a single manner, upwards

Example: For the last program above, we want to find the tuples (or values) in D who do not have a companion in the second argument of T

A query posed to the DB to catch violations of a referential integrity constraint; expressed in relational calculus as:

$$Q(x): D(x) \wedge \neg \exists z T(x, z)$$

It cannot be expressed like this as a program, to be used as a top layer of a combined program

It has to be transformed by first choosing a predicate to collect the answers, say $Ans(x)$, which has to be defined by rules, in this case:

$$Ans(x) \leftarrow D(x), \text{ not } Aux(x)$$

$$Aux(x) \leftarrow T(x, y)$$

The auxiliary Aux predicate is introduced to avoid having a non-safe rule

“Query program” is combined with the one on page 46, producing a new stratified program

Datalog Properly Extends Relational Algebra

All the operations of RA (and relational calculus) can be (declaratively) expressed in Datalog with stratified negation:

- Projection $\Pi_A R(A, B): Ans(x) \leftarrow R(x, y)$
- Join $R(A, B) \bowtie_B S(B, C): Ans(x, y, z) \leftarrow R(x, y), S(y, z)$
- Selection $\sigma_{A=a} R(A, B): Ans(a, x) \leftarrow R(a, x)$
- Union $R(A, B) \cup S(A, B):$
 $Ans(x, y) \leftarrow R(x, y)$
 $Ans(x, y) \leftarrow S(x, y)$
- Set Difference: $R(A, B) \setminus S(A, B):$
 $Ans(x, y) \leftarrow R(x, y), \textit{not } S(x, y)$

And in addition, we have recursion! (not available in RA or RC)

Aggregation

(Stratified) Datalog can be extended in different ways ...

One of the extensions includes **aggregation**

For example, the following Datalog rule with aggregation defines an intensional predicate with aggregation in an argument

$$P(x, y, \text{sum}(z)) \leftarrow T(x, y, w, z)$$

This is **aggregation with group-by** over the extensional relation T

T	A	B	C	D
	a	b	c1	5
	a	b	c2	7
	a	d	c3	4
	a	d	c4	2
	b	e	c5	3

P	A	B	D
	a	b	12
	a	d	6
	b	e	3

Exercise: Given a table of authors with their publications, and the number of citations of the latter, that the author has H-Index h means that h is the largest number such that the author has at least h publications with at least h citations

Publications	Author	Title	Citations
	john	aaaabb	6
	john	aaabbb	8
	mary	acabb	5
	john	caaabb	100
	john	aaaacb	2
	peter	acaabb	9

john's H-Index is $h = 3$

Write a Datalog program with aggregation that computes an author's H-Index from any instance as above, in a defined predicate *HIndex(Author, Number)*

Beyond Stratified Programs?

Do we want or need to go beyond stratified programs?

We saw some problems with resolution/NAF for some classes of normal programs, and we want to overcome them

In particular, we saw that there are problems with **recursion via (NAF-based) negation**

WE confront also new issues when trying to give a model-theoretic semantics to programs without stratified negation

Example: A non-stratified program:

$P(x) \leftarrow R(x), \text{ not } Q(x)$

$Q(x) \leftarrow P(x)$

$Q(a). \quad R(b).$

Example: $P(x) \leftarrow R(x), \text{ not } Q(x)$
 $Q(x) \leftarrow P(x)$
 $Q(a), R(b)$

What if we use resolution with NAF to obtain P 's contents?

Goal:

$\leftarrow P(x)$
 $\leftarrow R(x), \text{ not } Q(x)$
 $\leftarrow \text{ not } Q(b)$
 $\leftarrow Q(b)$
 $\leftarrow P(b)$
 $\leftarrow R(b), \text{ not } Q(b)$
 $\leftarrow \text{ not } Q(b)$ ETC.

An **unstratified program**, with a **recursive definition involving negation**: P defined in terms of P via the negation of Q

Q is not fully defined/computed before entering the recursion

Exercise: Try to use Prolog to run this program

What if we use bottom-up evaluation as with stratified programs to obtain a reasonable model and P 's contents?

$Q(b)$
↑ with second rule
 $P(b)$
↑ with first rule, because $Q(b)$ is false
 $R(b), Q(a)$

We obtain: $M = \{R(b), Q(a), P(b), Q(b)\}$, where the “justification” for $P(b)$ is not longer true (the absence of $Q(b)$)

M is a model (every ground rule is true), it has unjustified (unsupported) atoms, but is **not minimal**

The proper subset $M' = \{R(b), Q(a), Q(b)\}$ is a minimal model, which is also weird as a model: $Q(b)$ is unjustified

Stratified programs (the good ones for this kind of evaluation) do not exhibit these undesirable behaviors

In them recursion and negation may coexist, but no recursion involves a negation of a predicate that is being defined by the same recursion (the one above is not stratified)

Exercise: Prove by contradiction that the program on page 58 is not stratified (unstratified), i.e. there is no possible stratification

Need for a Semantics for Unstratified Programs

Unstratified programs (UPs) are very important in practice of KR and have a higher expressive power than stratified ones

However, it is not clear (for the moment) what is the semantics of an unstratified program (we'll come back)

Unstratified normal programs can have more than one intended model

Unstratified normal programs can be used to express (specify) combinatorial problems in the class NP , in particular, NP -complete ones

Example: 3-graph coloring (3-GC): (slight modification of program in chapter 1)

We could attempt to use the following rules (part of the program only):

$color(x, green) \leftarrow country(x), \text{ not } color(x, blue), \text{ not } color(x, red)$
 $color(x, blue) \leftarrow country(x), \text{ not } color(x, green), \text{ not } color(x, red)$
 $color(x, red) \leftarrow country(x), \text{ not } color(x, blue), \text{ not } color(x, green)$

Not stratified! (there is recursion via negation)

Each intended model should be a solution to the combinatorial problem (if there is a solution)

In this case, a particular 3-coloring (if they exist)

That an intended model does not exist means no solution to the computational problem

3-GC is NP -complete, so as (propositional) SAT, TSP, HC, ...

Notice that stratified programs or any polynomial-time computable program cannot represent a problem like this (unless $P = NP$)

So, what are the intended models of an unstratified normal program?

We identified problems for these programs, related to the combination of variables and negation, recursion and negation, ...

What is the intended meaning, i.e. the intended model(s) of the unstratified program on page 57? Should $P(b)$, $Q(b)$ be true, false?

What about the semantics for the 3-GC program?

So, we need a general, model-based semantics for unstratified normal programs

Hopefully one that extends the semantics for the “good” classes of programs (i.e. plain Datalog programs, stratified normal programs)



Chapter 10: Stable Model Semantics of LPs

and

Answer-Set Programming

Leopoldo Bertossi

**Carleton University
School of Computer Science
Ottawa, Canada**

The Stable Model Semantics

- We give a **new, model-based semantics to normal programs**

Done by characterizing the collection of intended models

- Programs may have no negation, stratified negation, or unstratified negation

Weak or non-monotonic negation (not necessarily the procedural NAF)

Example: What is the semantics of this program Π ?

$P(a) \leftarrow \text{not } P(b)$ An unstratified ground program

What are its intended models? Not clear, as seen in previous chapter

- We **need a new semantics** for this kind of programs

Hopefully we will reobtain the good old semantics for negation-free and stratified programs

- It is the **stable model semantics** (or more generally, **answer set semantics**) (Gelfond & Lifschitz, 1988)

It can be applied to a normal program Π ; stratified or unstratified

Π can be already a ground program as above or have variables, in which case it is instantiated first

- Let $S \subseteq HB(\Pi)$, i.e. a subset of the program's Herbrand Base

S is a set of assumptions, and a candidate to be a (stable) model of Π

A “guess” that will be accepted if properly supported by Π

- For S to be an intended model, its atoms have to be properly justified by Π

S will be a **stable model of Π** if it can be justified on the basis Π

More precisely, if assuming S , we can recover S via Π

- We make S pass the following **test**:
 1. Pass from Π to Π_H , the ground instantiation of Π
 2. Construct a new ground program Π_H^S , depending on S as follows:
 - (a) Delete from Π_H every rule that has a subgoal *not* A in the body, with $A \in S$

Intuitively: We are assuming A to be true, then *not* A is false, then the whole body is false, and nothing can be concluded with that rule, it is useless
 - (b) From the remaining rules, delete the negative subgoals

Intuitively: Those rules are left because the negative subgoals are true, and since they are true, we can eliminate them as conditions in bodies (because they hold)
 3. We are left with a **ground definite program** Π_H^S (no negation)

4. Compute $\underline{M}(\Pi_H^S)$, the minimal model of the definite program
 5. If $\underline{M}(\Pi_H^S) = S$, we say that S is a **stable model** of Π
- Intuitively, we started with S (as an assumption) and we recovered it, it was stable wrt. to the Π -guided process described above; it is self-justified

Example: Program Π $P(a) \leftarrow \text{not } P(b)$ (nothing to ground, already ground)

Consider $S = \{P(a)\}$

Here: $P(b) \notin S$, then $\text{not } P(b)$ is satisfied in S and can be eliminated from the body

We obtain $\Pi^S: P(a) \leftarrow$, a definite program

Its minimal model is $\{P(a)\}$, that is equal to S

S is a stable model of the original program

Notice that Π is unstratified (there is recursion via negation), but has a stable model, actually only one in this case

Exercise: For the program above, verify that:

(a) The empty set $\{\}$ (as a subset of the HB) is not a model

(b) $\{P(a)\}$ is a model

That is, it makes all the implications of the program true

For $S \subseteq HB(\Pi)$, by definition: *not* $P(a)$ is true in S iff $P(a) \notin S$

(c) $\{P(a)\}$ is a minimal model, that is, no proper subset is a model

(d) $\{P(b)\}$ is a model

(e) $\{P(b)\}$ is a minimal model

(f) $\{P(b)\}$ is not a stable model

So, there are minimal models that are not stable

Example: Program Π (unstratified)

$P(x) \leftarrow Q(x, y), \text{ not } P(y). \quad Q(a, b).$

Π_H : (ground instantiation) **Candidate** $S = \{P(b)\}$

$P(a) \leftarrow Q(a, a), \text{ not } P(a)$

$P(a) \leftarrow Q(a, b), \text{ not } P(b) \quad \times$

$P(b) \leftarrow Q(b, a), \text{ not } P(a)$

$P(b) \leftarrow Q(b, b), \text{ not } P(b) \quad \times \quad Q(a, b).$

$\Pi_H^S :$ $P(a) \leftarrow Q(a, a)$

$P(b) \leftarrow Q(b, a) \quad Q(a, b)$

Minimal model of Π_H^S is $\{Q(a, b)\} \neq S$, then S is not a stable model

Now consider: $S = \{Q(a, b), P(a)\}$

$P(a) \leftarrow Q(a, a), \underline{\text{not } P(a)}$ \times

$P(a) \leftarrow Q(a, b), \text{not } P(b)$

$P(b) \leftarrow Q(b, a), \underline{\text{not } P(a)}$ \times

$P(b) \leftarrow Q(b, b), \text{not } P(b)$ $Q(a, b).$

$\Pi_H^S :$ $P(a) \leftarrow Q(a, b)$

$P(b) \leftarrow Q(b, b)$ $Q(a, b)$

Minimal model of Π_H^S is $\{Q(a, b), P(a)\} = S$

S is a stable model of Π

Example: Program Π : $male(x) \leftarrow person(x), not\ female(x)$
 $female(x) \leftarrow person(x), not\ male(x)$
 $person(a).$

If $S_1 = \{person(a), male(a)\}$, then Π^{S_1} is

$male(a) \leftarrow person(a)$
 $person(a).$

Then, S_1 is a stable model

$S_2 = \{person(a), female(a)\}$ is also a stable model of Π

There may be more than one stable model for a program! (Check them!)

Again, Π is unstratified

Exercise: For the program $P(a) \leftarrow \text{not } P(a)$, prove that:

(a) $\{\}$ is not a model

(b) The program has no stable models Hint: Consider two cases for an $S \subseteq HB(\Pi)$: (a) $q \notin S$; (b) $q \in S$

We say that **the program is inconsistent** (under the stable model semantics)

So, normal programs may have no stable models

(c) Actually it has no (H-)models

Exercise: Find and verify the stable models of the program

$P(a) \leftarrow \text{not } P(b). \quad P(b) \leftarrow \text{not } P(a).$

Example: Program Π

$$\begin{aligned} & \textit{even}(0). \\ & \textit{even}(x) \leftarrow \textit{not even}(s(x)) \end{aligned}$$

$$H = \{0, s(0), s(s(0)), s(s(s(0))), \dots\}$$

Π_H :

$$\begin{aligned} & \textit{even}(0). \\ & \textit{even}(0) \leftarrow \textit{not even}(s(0)) \\ & \textit{even}(s(0)) \leftarrow \textit{not even}(s(s(0))) \\ & \dots \end{aligned}$$

$S = \{ \overset{\dots}{\textit{even}(0)}, \overset{\dots}{\textit{even}(s(s(0)))}, \textit{even}(s(s(s(s(0))))), \dots \}$ is the only stable model

Π_H^S :

$$\begin{aligned} & \textit{even}(0). \\ & \textit{even}(0). \\ & \textit{even}(s(s(0))). \quad \text{Etc.} \end{aligned}$$

$$\underline{M}(\Pi_H^S) = \{ \textit{even}(0), \textit{even}(s(s(0))), \textit{even}(s(s(s(s(0))))), \dots \}$$

Some Results and Notions:

- Every stable model of Π is a Herbrand model in the usual sense

In them, *not* is interpreted as “not belonging to the model”

- **A stable model is always a minimal model** (i.e. no proper subset of it is a model of the program)
- **A normal program may have several stable models**
- Several (stable) models may determine the semantics of a program:

What is true of (wrt.) the program is what is true in all its stable models

- If there are several stable models for a program, it means that some atoms are left undetermined (those that are true in some of them, but false in others)

Example: The program on page 10 leaves every *male*-atom and *female*-atom undetermined, uncertain

However, it is certain that $person(a)$

However, it is still certain that $male(a) \vee female(a)$

Exercise: Find the stable models for the unstratified program shown as example at the end of the previous chapter:

$P(x) \leftarrow R(x), \text{ not } Q(x)$

$Q(x) \leftarrow P(x)$

$Q(a), R(b)$

Exercise: Show using any of the example programs above that the use of negation in the stable model semantics is indeed non-monotonic

Hint: You can show that by adding a new fact to the EDB, you may lose a previous certain consequence, i.e. something true in all stable models

- We have given a declarative, model-based semantics to a wider class of programs (with or without negation), even non-stratified
- **Notions of entailment** from a program under the **stable model semantics** of a normal program?
 - **Skeptical, certain or cautious semantics**: What is true of a program is what is true of **all** stable models of the program
In the previous example, $person(a)$ is skeptically true, but not $male(a)$
 - **Brave or possible semantics**: What is true of a program is what is true of **some** stable model of the program
In the previous example, $male(a)$ is possibly (or bravely) true

- One can prove: If Π is definite or stratified-normal, then it has a unique stable model

This stable model coincides with the minimal model for definite programs, and the standard model for stratified normal programs!



- Then, the stable model semantics extends the ones we had for the “good” classes before!

In particular, in those cases, the unique stable model can be computed by means of an bottom-up iterative process, in polynomial time in the size of the underlying database (set of facts)

- We also obtain right away that the stable model semantics is non-monotonic (already shown for stratified programs)

Exercise: Check that the standard model for the stratified program at the end of the previous chapter (pages 45 and 51), obtained by upward computation, is a stable model

Complexity Considerations:

- Given a normal program Π , the problems of **deciding if**:
 - a given set of atoms M is a stable model (the model checking problem)
 - the program has a stable model (i.e. the program is consistent)
 - an atom is a skeptical consequence 
 - an atom is a brave consequence 

have all rather high complexity; **at least NP-hard** for the last three in the size of the underlying extensional data (i.e. in data complexity)

- Computing a stable model is also hard ... 

- However, with unstratified programs with stable model semantics we gain expressive power

And we can solve problems that are intrinsically complex and require such an expressive power (and complexity of program evaluation)

- In particular, computational implementations for the stable model semantics are being used to find solutions to combinatorial optimization and decision problems

NP-complete problems can be represented by unstratified programs, and the stable models correspond to their solutions (see below)

In those applications finding one model is usually good enough



A Useful “Extension”: Program Constraints

- Program constraints (PCs) can be added to a normal program:

$$\leftarrow A_1, \dots, A_m, \textit{not } A_{m+1}, \dots, \textit{not } A_k,$$

with A_i atoms, can be added to a program Π

- Intuitively: It is not possible that the body becomes true (for any values for the variables) ... A prohibition!
- It has the effect of filtering or eliminating the stable models that make the body true

Example: Program Π

$$\begin{aligned} ug(x) &\leftarrow stud(x), \textit{not } grad(x) \\ grad(x) &\leftarrow stud(x), \textit{not } ug(x) \\ stud(mary) &\leftarrow \\ &\leftarrow ug(x) \end{aligned}$$

Without the PC, two stable models: $\{stud(mary), ug(mary)\}$ and $\{stud(mary), grad(mary)\}$; with the PC, only the second one

- A program constraint

$$\leftarrow A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_k,$$

can be represented as a normal rule:

(Notice: unstratified!)

$$q \leftarrow A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_k, \text{not } q,$$

q is a fresh propositional atom; appearing nowhere else in the program

- If in a stable model $A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_k$ becomes true, then also $q \leftarrow \text{not } q$ becomes true in that model

This is not possible (cf. page 11)

So, in no stable model of the program $A_1, \dots, A_m, \text{not } A_{m+1}, \dots, \text{not } A_k$ will be true

Example: (cont.) $q \leftarrow ug(x), not\ q$

If a candidate S to be a stable model $ug(x)$ becomes true (for any value of x), also $q \leftarrow not\ q$ has to be true

- A program constraint (PC) has the effect of filtering the stable models of Π (without the PC) where the body of the PC becomes true
- PCs are an extra source of unstratified negation
- In data management applications, PCs are useful to capture common integrity constraints, e.g. denial or check constraints:

$\leftarrow Employee(x, janitor), InBoard(x, yes)$

“No janitors in the company board”

- The rules of the program may specify (and perform) different data management tasks

... but we want to make sure the possible alternative states of the database, represented by the programs' stable models, are consistent (wrt. the PCs)

... the PCs play that role

Example: Consistency of DB wrt. a referential constraint: *“the values in the second argument of table $R(A, B)$ must appear in the first column of table $S(B, C)$ ”*

$\leftarrow R(x, y), \text{ not } S(y, z)$ (not safe)

Better: $\leftarrow R(x, y), \text{ not } S'(y)$ Plus the rule: $S'(y) \leftarrow S(y, z)$

Now functional dependency FD: $A \rightarrow B$ on R : (i.e. B functionally depends on A)

$\leftarrow R(x, y), R(x, z), y \neq z$

Example: (3-GC revisited)

$color(x, green) \leftarrow country(x), \text{ not } color(x, blue), \text{ not } color(x, red)$

$color(x, blue) \leftarrow country(x), \text{ not } color(x, green), \text{ not } color(x, red)$

$color(x, red) \leftarrow country(x), \text{ not } color(x, blue), \text{ not } color(x, green)$

$\leftarrow edge(x, y), color(x, z), color(y, z)$

“No neighboring countries can share the same color”

The same program can be used with any instance of the problem, i.e. concrete graph, represented through the facts

- Normal programs with stable model semantics can be used to solve hard (and easy) combinatorial problems

Determining if there are stable models, and computing one (in the positive case) is good enough (the brave semantics can be used)

- ASPs are used to implicitly specify by means of a general program $\Pi^{\mathcal{G}}$ all the solutions of a general, usually combinatorial problem \mathcal{P}

Given a specific instance, I , for problem \mathcal{P} , as the extensional instance (set of facts), E , for the program, the stable models of the combined specification $\Pi^{\mathcal{G}} \cup E$ become the solutions of \mathcal{P} for instance I

- The solutions can be computed using a general implementation of the stable models semantics, e.g. DLV

The Choice Operator: A Useful “Extension”

- A program Π may have a “choice rule” of the form:

$$P(x, y, z) \leftarrow Q(\dots, x, \dots), S(\dots, y, \dots), R(\dots, z, \dots), \textit{Choice}((x, y), z) \quad (*)$$

- Intuitively, for each combination of values (x, y) , **non-deterministically choose a unique value** for z and put the trio (x, y, z) into P
(with those values satisfying the other conditions in the body)

Example: Consider the relational table that does not satisfy the functional dependency $AB \rightarrow C$

R	A	B	C
	a	b	c
	a	c	e
	a	b	d

Repairing R :

$$R'(x, y, z) \leftarrow R(x, y, z), \textit{choice}((x, y), z).$$

$R(a, b, c)$. Etc.

Two stable models (then two repairs): $\{R'(a, b, c), R'(a, c, e), R(a, b, c), \dots\}$
 $\{R'(a, b, d), R'(a, c, e), R(a, b, c), \dots\}$

- Different choices for z appear in different stable models
- Why stable models? Where is negation?
- Can the “choice operator” be replaced by regular rules, i.e. eliminated and redefined?
- No essential need for the *choice* operator: (by nice and useful having it!)

Replace the choice rule (*) by:

$$P(x, y, z) \leftarrow Q(\dots, x, \dots), S(\dots, y, \dots), R(\dots, z, \dots), \textit{Chosen}(x, y, z),$$

And define the *Chosen* predicate with two extra rules:

$$\textit{Chosen}(x, y, z) \leftarrow Q(\dots, x, \dots), S(\dots, y, \dots), R(\dots, z, \dots), \textit{not DiffChoice}(x, y, z)$$

$$\textit{DiffChoice}(x, y, z) \leftarrow \textit{Chosen}(x, y, z'), z' \neq z \quad \text{Unstratified!}$$

The last two rules ensure that, for every pair of values (x, y) that satisfies the body, the predicate $\textit{Chosen}(x, y, z)$ satisfies the functional dependency: $xy \rightarrow z$

- In (*) the choice operator can (or must) be replaced by the new predicate *Chosen* that forces the expected functional dependency

Some systems do not support the choice operator, so *Chosen* has to be used instead

- The *choice* operator is an extra source of unstratified negation

Exercise: In the example above, replace the *choice* operator by its corresponding and concrete *Chosen* predicate (adding its definition, of course); and compute the repairs of the DB with DLV

Exercise: Give a general program to solve the “Hamiltonian Cycle” problem

Disjunctive Stable Model Semantics

- Now we admit rules of the form

$$B_1 \vee \cdots \vee B_k \longleftarrow A_1, \cdots, A_m, \textit{not } A_{m+1}, \cdots, \textit{not } A_n$$

with B_j, A_i atoms, and *not* weak negation

E.g. $P(x) \vee T(x) \longleftarrow R(x), Q(y), \textit{not } T(y), \textit{not } Q(x)$

- Now we have the **disjunctive stable model semantics** (Gelfond & Lifschitz, 1991)
- Similarly to normal programs: Start from a (candidate) set of atoms S
 1. Pass from Π to Π_H (grounding process)
 2. Construct a new ground program Π_H^S as follows:
 - (a) Delete from Π_H every rule that has a subgoal *not* A in the body, with $A \in S$
 - (b) From the remaining rules, delete the negative subgoals

3. We are left with a **ground disjunctive positive program** Π_H^S (no negation)

4. Consider $MinMod(\Pi_H^S)$, the set of minimal H-models of Π_H^S

If $S \in MinMod(\Pi_H^S)$, we say that S is a **stable model** of Π

- As before, it turns out that a stable model of Π is a minimal H-model of Π

Example: $p \vee q \leftarrow s, \text{ not } p$
 $s \leftarrow$

For $S_1 = \{s, q\}$, the residual program is Π^{S_1} :

$p \vee q \leftarrow s$
 $s \leftarrow$

$MinMod(\Pi^{S_1}) = \{\{s, p\}, \{s, q\}\}$

$S_1 \in MinMod(\Pi^{S_1})$: it is a stable model

$S_2 = \{s, p\}$?

Residual program is Π^{S_2} : $s \leftarrow$

$MinMod(\Pi^{S_2}) = \{\{s\}\}$

$S_2 \notin MinMod(\Pi^{S_2})$: not a stable model

Example: (3-GC revisited) This could be the program:

$$\begin{aligned} & \text{color}(x, \text{green}) \vee \text{color}(x, \text{blue}) \vee \text{color}(x, \text{red}) \leftarrow \text{country}(x) \\ & \leftarrow \text{edge}(x, y), \text{color}(x, z), \text{color}(y, z) \end{aligned}$$

No neighboring countries can share the same color ...

Unstratified negation is hidden in the program constraint



- Disjunctive programs with negation in certain syntactic classes of programs can be rewritten as normal programs, i.e. without disjunction

This is the case of the program above

Compare with program on page **23**

Remarks:

- Disjunctive logic programs with stable model semantics are very expressive

More than (non-disjunctive) normal programs with stable model semantics (under some complexity-theoretic assumptions)

So, some disjunctive programs cannot be expressed (equivalently, i.e. with the same stable models) as normal programs

- There are several implementations of the stable and disjunctive stable model semantics: DLV, SModels, ...
- There are other semantics for (disjunctive) normal programs

Most prominently, the “**well-founded semantics**”, which is related to stable model semantics and has nice computational features

(Van Gelder, Ross, Schlipf; 1988, 1991), (Leone, Rullo, Scarcello; 1997)



The Answer Set Programming Paradigm

- Answer set programming is a new declarative programming paradigm
- “Answer sets”, as opposed to “stable models”, means that we:
 - Describe worlds by means of *sets of classical literals*, e.g.

$$\mathcal{M} = \{P(a), P(b), \neg P(c), Q(d), \neg Q(a)\}$$

There is incomplete knowledge (partial models, no CWA at this level):
 $P(a), P(b), Q(d)$ are true, $P(c), Q(a)$ are false, and we do not know about $P(d), Q(b), Q(c)$

- Also accept classical literals (instead of only atoms), possibly affected by weak negation, e.g.

$$(\neg)A(x) \leftarrow B(x, y), \neg S(y), \text{not } P(x), \text{not } \neg M(x, y)$$

That is, classical and weak negation may coexist: **weak negation**
in front of classical literals

(at this point CWA can be applied with weak negation)

These are usually called **“extended logic programs”**

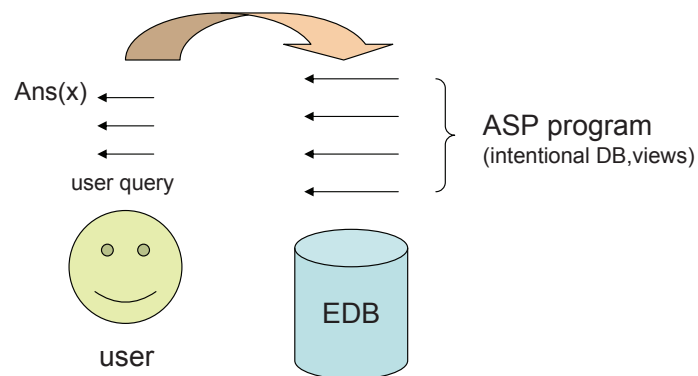
- Semantics coming later on, in terms of “answer sets” ...
- ASPs extend disjunctive programs with stable model semantics
- DLV supports *extended disjunctive logic programs with answer set semantics*
- Answer set programs (ASPs) have been successfully used in different areas and problems (Cf. Brewka, G., Eiter, T. and Truszczyński, M. Answer Set Programming at a Glance. Comm. of the ACM, 2011, 54(12), pp. 93-103)
- Some applications areas:

1. Obviously, logic-based **knowledge representation** in general
 In particular, for representation of commonsense knowledge and commonsense reasoning
2. Solution of hard combinatorial problems (seen above ...)
3. Many applications in **data management**

After all ASP extends Datalog

The program goes on top of the (relational) database

It can be used to define complex views and express complex queries



The stable models of EDB + ASP program are extended with query atoms $Ans(a)$

Extended Normal Logic Programs (Answer-Sets)

- Now we will make weak and classical negations coexist:

$$L \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_n$$

L, L_i are **literals** of the form A or $\neg A$, with A is an atom, and \neg classical (strong) negation

Examples: $D(x) \leftarrow R(x), \neg Q(y), \text{not } E(x), \text{not } \neg Q(x)$

$$\neg D(x) \leftarrow R(x), \neg Q(y), \text{not } E(x), \text{not } \neg Q(x)$$

- No weak negation (*not*) in the heads (but there are extensions)
- We gain expressivity wrt normal programs
- Instead of stable model semantics, we have the **answer sets semantics**, which is similar and extends the former

(Gelfond & Lifschitz, 1991)

- Instead of stable models of a program Π , we will have **answer sets**
- They are special sets of **ground literals**, taken from the “**Extended Herbrand Base**” (EHB)

\overline{HB} contains **all the ground literals** (not only ground atoms)

Example: A candidate set $S = \{R(a), \neg Q(c), \neg D(a), E(d)\}$

S is a the set of true assumptions about the world (we know of)

S says that $R(a)$, $E(d)$ are true, and $Q(c)$, $D(a)$ are false, and the other ground literals are undetermined, e.g. $R(b)$

It does not represent a “complete structure or interpretation”, but only a “partial interpretation”: we do not know if $R(b)$ is true or not ...

$S' = \{\dots, R(a), \dots, \neg R(a), \dots\}$ is an **inconsistent** set of assumptions

- Remember that for non-extended programs, without \neg , a set of ground atoms represents what is true of the world, and whatever is not in it is considered to be false (hence a complete structure)

- **Semantics of extended programs?** First the positive case

(A) As before, thinking of a test for candidate models, we **start with “positive” extended programs**, i.e. without *not* (but possibly with \neg)

For an **extended (ground) program Π without weak negation, *not***, an **answer set** is defined as a minimal set of ground literals S , such that:

1. For every ground rule $L_0 \leftarrow L_1, \dots, L_m$: If $L_1, \dots, L_m \in S$, then $L_0 \in S$
2. If a pair of **complementary literals**, say $A, \neg A$, belong to S , then $S = \overline{HB}$ (from a contradiction everything follows) (\leftarrow **this is new!**)

To capture strong (classical) negation, by making reasoning trivial when a classical contradiction is found or generated

- As with definite programs, a program of this kind has a unique answer set

It can be computed bottom-up with forward propagation as before

Any ground literal $\neg P(a)$ is (can be) treated as a positive instantiation of a new predicate \bar{P} , not related to P (except for item 2. above)

It is the minimal answer set $\underline{M}(\Pi)$

(In correspondence with the minimal model of the positive program with \bar{P} instead of $\neg P$)

Example: An extended ground program without *not*

$$\begin{array}{l} D(a) \leftarrow R(a) \\ D(b) \leftarrow R(b) \\ D(c) \leftarrow R(c) \\ R(c) \leftarrow \neg U(c) \\ E(a, c) \leftarrow U(a) \\ E(b, c) \leftarrow U(b) \\ E(c, c) \leftarrow U(c) \\ U(a). U(c). \quad V(b). V(a). \end{array}$$

Minimal answer set Π_H :

$$\underline{M}(\Pi_H) = \{U(a), U(c), V(b), V(a), E(c, c), E(a, c)\}$$

Example: An extended ground program without *not*

$$\begin{array}{lcl} P(a) & \leftarrow & R(a) \\ \neg P(a) & \leftarrow & R(b) \\ R(b) & \leftarrow & R(c) \\ & & R(a). \quad R(c). \end{array}$$

Any answer set must contain $R(a)$ and $R(c)$, and then also $P(a)$ and $\neg P(a)$

So, the only answer set S is the whole extended HB:

$$\{P(a), \neg P(a), R(a), R(b), R(c), \neg R(a), \neg R(b), \neg R(c), P(b), P(c), \neg P(b), \neg P(c)\}$$

(B) **Now extended normal programs Π** : they can have *not*

Similarly to the stable model semantics

A **set of ground literals S** is an **answer set** of Π if it passes the following test:

1. Produce the full ground instantiation Π_H of Π on the Herbrand universe H
2. Delete every clause from Π_H that contains in the body a subgoal of the form *not* L , where L is a literal and $L \in S$
3. Delete from every remaining rule in Π_H the subgoals of the form *not* L , where L is a literal and $L \notin S$
4. We are left with a “definite” ground program Π_H^S containing classical literals, but no weak negation
E.g. clauses of the form $p \leftarrow q, \neg r$ or $\neg p \leftarrow q, \neg s$ (since this new program is ground, it can be seen as a propositional program)
5. Compute its answer set $\underline{M}(\Pi_H^S)$ exactly as for definite programs, with a caveat ...
Now by definition $\underline{M}(\Pi_H^S)$ contains all the ground literals if it happens to contain a pair of complementary literals, say $Q(a), \neg Q(a)$ (\leftarrow this is new!)
6. $S = \underline{M}(\Pi_H^S)$

- An answer set may not be a complete model (structure): not necessarily every ground atom has a truth value

In contrast, stable models (of \neg -free) programs are complete models

Example:

$$\begin{aligned} D(x) &\longleftarrow R(x), \text{ not } \neg E(x, y). \\ R(x) &\longleftarrow \neg U(x), \text{ not } V(x). \\ E(x, y) &\longleftarrow U(x), \text{ not } V(y). \\ U(a). U(c). & \quad V(b). V(a). \end{aligned}$$

$$H = \{a, b, c\}$$

The extended Herbrand Base is:

$$\overline{HB} = \{D(a), D(b), \dots, E(a, b), \dots, \neg D(a), \neg D(b), \dots, \neg E(a, b), \dots\}$$

Π_H obtained by replacing all elements of H for variables in Π , e.g.

$$D(a) \longleftarrow R(a), \text{ not } \neg E(a, b).$$

Etc.

Candidate: $S = \{D(a), D(b), R(b), E(a, b), \neg E(a, a), \neg V(c), U(a), U(c), V(a), V(b)\} \subseteq \overline{HB}$

Program Π_H^S after the pruning steps:

$$\begin{array}{l} D(a) \longleftarrow R(a) \\ D(b) \longleftarrow R(b) \\ D(c) \longleftarrow R(c) \\ R(c) \longleftarrow \neg U(c) \\ E(a, c) \longleftarrow U(a) \\ E(b, c) \longleftarrow U(b) \\ E(c, c) \longleftarrow U(c) \\ U(a). U(c). \quad V(b). V(a). \end{array}$$

Minimal model Π_H^S :

$$\underline{M}(\Pi_H^S) = \{U(a), U(c), V(b), V(a), E(c, c), E(a, c)\} \neq S$$

S is not an answer set

(find answer sets!)

The minimal model can be computed as before, treating the whole $\neg U(c)$ as an atom, and the rule can be applied only if $\neg U(c)$ has been explicitly obtained (no CWA applied to $U(c)$ here)

- The answer sets semantics for ELP extends the stable models semantics for normal programs, i.e. if the program is \neg -free, then the answer sets are the stable models

Exercise: Find answer set and non-answer sets for

$$\begin{array}{l} P(x) \longleftarrow \neg R(x), \text{ not } \neg S(x, y). \\ \neg R(x) \longleftarrow \neg U(x), \text{ not } V(x). \\ S(x, y) \longleftarrow U(x), \text{ not } V(y). \\ U(a). U(c). \qquad V(b). V(a). \end{array}$$

- Once more, notice that given and processing candidates to answer sets, e.g. $S = \{R(a), \neg Q(c), \neg D(a), E(d)\}$, its negative ground literals can be treated as atoms
- Actually, it is possible to replace the negative literal by a new positive atom, introducing a new predicate, e.g. the one above becomes $\{R(a), \overline{Q}(c), \overline{D}(a), E(d)\}$

- Similarly, program rules can be transformed to get a \neg -free program, e.g.

$$D(x) \leftarrow R(x), \neg Q(y), \text{not } E(x), \text{not } \neg Q(x)$$

becomes $D(x) \leftarrow R(x), \overline{Q}(y), \text{not } E(x), \text{not } \overline{Q}(x)$

- However, the new rule (program constraint) is added to capture the connection

$$\leftarrow Q(x), \overline{Q}(x)$$

Prohibiting that $Q(x), \overline{Q}(x)$ are simultaneously true (for any x)

- Thus, extended logic programs can be reduced to normal logic programs
- With ELPs we “gain” expressivity, e.g. we can **express the CWA**:

$$\neg D(x) \leftarrow \text{not } D(x)$$

If $D(a)$ is not in the DB, then it is declared false, i.e. $\neg D(a)$ becomes true

Disjunctive Extended Logic Programs

- Now we admit rules of the form

$$B_1 \vee \cdots \vee B_k \longleftarrow A_1, \cdots, A_m, \textit{not } A_{m+1}, \cdots, \textit{not } A_n$$

with B_j, A_i are **classical literals** and *not* is weak negation

E.g. $P(x) \vee \neg T(x) \longleftarrow R(x), \neg Q(y), \textit{not } S(x), \textit{not } \neg Q(x)$

- As before we have an answer set semantics (Gelfond & Lifschitz, 1991)
- Now we will work with sets of ground literals; and so are the candidates S to be answer sets, i.e. of the form $S = \{p, f, k, \neg l, \neg d\}$ (ground literals can be seen in this way)

- We do not talk of “answer models”, because a set of literals, in particular, answer sets, do not necessarily correspond to a (complete) structure

E.g. in $S = \{p, f, k, \neg l, \neg d\}$, the atoms explicitly indicated here are determined wrt their true value: p, f, k are true, and l, d are false, but, say m (if appearing in the program) is undetermined in the sense that neither m nor $\neg m$ appears in S

- A set of literals is **consistent** if it does not contain two literals of the form $p, \neg p$

If this happens, it is extended to the whole sets of literals (everything becomes true)

- The test for S is similar:
 - Fully instantiate the program
 - Pruning process as for ELPs
 - Now we get a **ground disjunctive extended program without *not***, i.e. with clauses of the form

$$p \vee \neg q \longleftarrow s, t, u, \neg v, \neg w$$

- **The resulting disjunctive program may have several minimal models**
- S is an answer set if it coincides with **one of the minimal models** of the program in the previous step

Example: Consider

$$\begin{aligned}P(x) \vee \neg Q(x) &\longleftarrow \neg R(x), \text{not } S(x, y) \\ \neg R(x) &\longleftarrow \text{not } \neg U(x), V(x) \\ S(x, y) &\longleftarrow U(x), V(y) \\ U(a). \quad V(b).\end{aligned}$$

and the candidate $S = \{U(a), V(b), S(a, b), \neg R(b), P(b)\}$

The instantiated program contains the following rules (apart from the original facts $U(a), V(b)$):

$$\begin{aligned}P(a) \vee \neg Q(a) &\longleftarrow \neg R(a), \text{not } S(a, a) \\ P(a) \vee \neg Q(a) &\longleftarrow \neg R(a), \text{not } S(a, b) \\ P(b) \vee \neg Q(b) &\longleftarrow \neg R(b), \text{not } S(b, a) \\ P(b) \vee \neg Q(b) &\longleftarrow \neg R(b), \text{not } S(b, b)\end{aligned}$$

$$\begin{aligned}
\neg R(a) &\longleftarrow \text{not } \neg U(a), V(a) \\
\neg R(b) &\longleftarrow \text{not } \neg U(b), V(b) \\
S(a, a) &\longleftarrow U(a), V(a) \\
S(b, b) &\longleftarrow U(b), V(b) \\
S(a, b) &\longleftarrow U(a), V(b) \\
S(b, a) &\longleftarrow U(b), V(a)
\end{aligned}$$

After the pruning process we are left with Π^S consisting of $U(a), V(b)$ plus

$$\begin{aligned}
P(a) \vee \neg Q(a) &\longleftarrow \neg R(a) \\
P(b) \vee \neg Q(b) &\longleftarrow \neg R(b) \\
\neg R(a) &\longleftarrow V(a) \\
\neg R(b) &\longleftarrow V(b)
\end{aligned}$$

$$\begin{aligned}
S(a, a) &\longleftarrow U(a), V(a) \\
S(b, b) &\longleftarrow U(b), V(b) \\
S(a, b) &\longleftarrow U(a), V(b) \\
S(b, a) &\longleftarrow U(b), V(a)
\end{aligned}$$

A minimal model for this program is precisely S (whatever is not in S is considered to be false when evaluating if it is a model)

Thus, S is a stable model of the original program

Exercise:

$$\begin{aligned}
P(x) \vee \neg Q(x) &\longleftarrow \neg R(x), \text{not } S(x, y) \\
R(x) &\longleftarrow \text{not } \neg U(x), V(x) \\
S(x, y) &\longleftarrow U(x), V(y) \\
U(a). \quad U(c). \quad V(b). \quad V(a).
\end{aligned}$$

Find answer and non-answer sets ...

- If the disjunctive program has weak negation *not*, but no classical negation \neg , then we call it a **disjunctive normal program**

The answer sets are called stable models

They are real models, and for the “normal”, non-disjunctive, case they provably coincide with the stable models we had before

Thus, we obtain as a special case a stable model semantics for disjunctive normal programs

Exercise:

$$P(x) \vee Q(x) \longleftarrow R(x), \text{ not } S(x, y)$$

$$R(x) \longleftarrow \text{ not } U(x), V(x)$$

$$S(x, y) \longleftarrow U(x), V(y)$$

$$U(a). \quad U(c). \quad V(b). \quad V(a).$$

Find stable models for this disjunctive normal program

□

- As for (disjunctive) normal programs, it is also possible to use **program constraints**; now of the form

$$\leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_k,$$

with L_i now **classical literals**, can be added to a program Π

They play the same role as program constraints for programs without classical negation

Similarly, program constraints can be encoded as usual rules, e.g.

$$q \leftarrow L_1, \dots, L_m, \text{not } L_{m+1}, \dots, \text{not } L_k, \text{not } q,$$

with q a fresh (propositional) atom

Program constraints have the effect of filtering the answer sets of Π where the body becomes true

The DLV ASP System

An implementation of a deductive database system

Computes the answer sets (intended models), and stable models for non-extended programs, in particular, of **(extended) disjunctive normal programs**

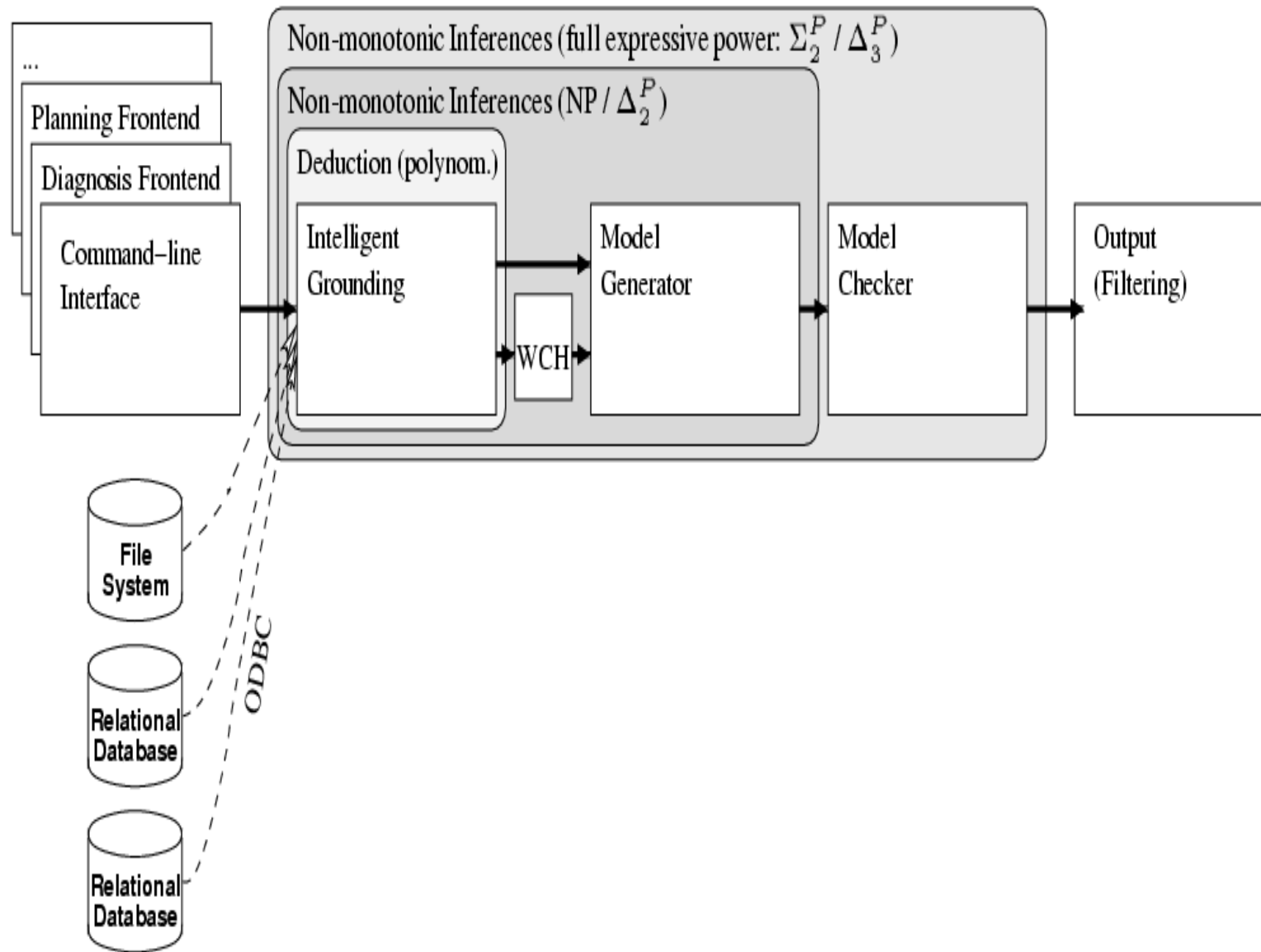
There may be head disjunctions and classical negation coexisting with weak negation

$$p(X) \vee Q(a) \text{ :- } s(X, Y), \text{ not } -r(Y) .$$

It can do query answering under the answer-set (stable-model) semantics

Programs may include disjunction, weak negation (“not” above) and strong (or classical) negation (“-” above)

It offers nice and useful interfaces to commercial relational DBMSs



- Naively grounding the original program will create a huge ground program, and processing it will be extremely costly

Better ground as needed, "intelligently"

- Guesses for models are not chosen randomly, and models are not built from scratch

It is possible to prove that all the stable models have a large intersection that can be computed in polynomial time

Individual models are guessed (generated) pivoting from that core, and then put to the test (checked)

To run DLV write the following command in a command window:

> *DLV -stats input_file > output_file*

Option *-stats* prints on screen the statistics of the run

In output file one gets the answer sets of the program given in the input file

- :- stands for ←
- Every rule ends with a dot
- A constant is a string of letters, numbers and underscores, beginning with a lowercase letter or number:

a1, 1, 9862, aBc1, c__

- A variable is a string of letter, numbers and underscores that begins with an uppercase:

A, V2f, Vi_X3

- Predicate symbols begin with a letter and may contain letters, underscores and digits: *ord, oBp, r2D2, E_mc2*

- *not* is weak negation; and “-” or “~” are strong (classical) negation
- The literals in the body are separated by commas; and in the head, by \vee

Example: Input file for the program $p(a) \leftarrow \text{not } p(b)$ is:

```
%File: ex1.dlv  
p(a) :- not p(b) .
```

The program `ex1.dlv` is run using the command:

```
> DLV -stats ex1.dlv >ex1.out
```

The output is automatically stored in the text file `ex1.out`:

```
DLV [build BEN/Jun 26 2003 gcc 2.95.3-10 (cygwin special)]  
{p(a)}
```

The statistics of the run are shown in the command window:

Residual Instantiation

Rules	:	1
Constraints	:	0
Weak Constraints	:	0
Structural Size	:	2
Maximum recursion level	:	0
Choice Points	:	0
Answer sets printed	:	1
Time for first answer set	:	0.016000
		(including instantiation)
Time for all answer sets	:	0.016000
		(including instantiation)
Instantiation time	:	0.000000
Model Generator time	:	0.015000
Model Checker time	:	0.000000
Total Model Check time	:	0.000000
Partial Model Check time	:	0.000000

Example: The Senators are organizing a party, and they want to invite all their friends, and the friends of their friends, and the friends of the friends of the friends, etc. However, they want to make sure no Maple Leaf player is invited to the party

```
player(mats,mapleLeaf).
player(bryan,mapleLeaf).
player(owen,mapleLeaf).
player(gary,mapleLeaf).
player(joe,mapleLeaf).
player(darcy,mapleLeaf).
player(marian,senators).
player(daniel,senators).
player(martin,senators).
player(jason,senators).
player(radek,senators).
player(wade,senators).
friend(wade,kevin).
friend(bryan,carolina).
friend(daniel,carolina).
friend(cathy,bryan).
friend(daniel,monica).
friend(radek,natalia).
friend(radek,cristal).
friend(cristal,isabel).
friend(radek,mary).
friend(mary,jason).
friend(ann,radek).
friend(radek,cristina).
```



```
friend(X,Y) :- friend(Y,X).

transitive_friend(X,Y) :- friend(X,Y).
transitive_friend(X,Y) :- transitive_friend(X,Z),
                           friend(Z,Y).

list(X) :- player(X,senators).
list(X) :- player(Y,senators),
           transitive_friend(Y,X).

-welcome(X) :- player(X,mapleLeaf).

invited(X) :- list(X), not -welcome(X).
```

Since the program is stratified, with no disjunction, we get only one stable model:

```
{player(mats,mapleLeaf), ... , player(marian,senators),
..., friend(bryan,carolina), ..., friend(carolina,daniel),
..., transitive_friend(bryan,daniel), ... ,
transitive_friend(natalia,jason), ... , list(bryan),
list(marian), list(daniel), list(martin), list(jason),
list(radek), list(wade), list(kevin), list(carolina),
list(cathy), list(monica), list(natalia), list(cristal),
list(isabel), list(mary), list(ann), list(cristina),
-welcome(mats),-welcome(bryan), -welcome(owen),
-welcome(gary), -welcome(joe), -welcome(darcy), invited(marian),
invited(daniel),invited(martin), invited(jason), invited(radek),
invited(wade), invited(kevin), invited(carolina), invited(cathy),
invited(monica),invited(natalia),invited(cristal),invited(isabel),
invited(mary),invited(ann), invited(cristina)}
```

Constraints in DLV:

DLV can handle denial constraints (rules with no head)

$$\leftarrow P(X), \text{not } Q(X).$$

This restriction says that for all the values of X we cannot have $P(X)$ and not have $Q(X)$

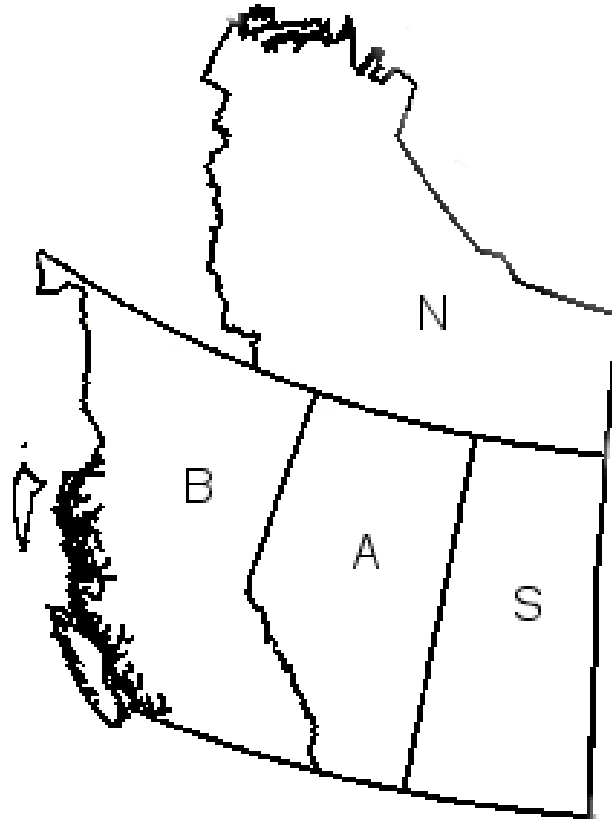
Only stable models that satisfy those program constraints are returned by the program

Program constraints, like integrity constraints in databases, capture more of the semantics of the application

By pruning out undesired models (so as ICs exclude certain DB states)

Example: Is the map of Alberta (A), Saskatchewan (S), British Columbia (B) and Northwest Territories (N) three colorable?

An instance of a NP-complete combinatorial problem ...



Input File:

```

[commandchars=\\\{\}]
% Extensional DB (EDB)
edge(a,s). edge(a,b). edge(s,n). edge(a,n).
edge(b,n).

% Intensional DB (IDB)
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).

colored(X,red) v colored(X,green) v colored(X,blue)
                    :- node(X).                (*)
                    :- edge(X,Y), colored(X,C), colored(Y,C).

```

- In general, a disjunctive rule, like (*), unless forced otherwise by the other program rules, interprets a disjunction in an exclusive manner (by minimality)
- There are 5 possible ways to paint the map as shown by the following stable models

One stable model per solution of the 3-coloring problem:

```

%[fontsize=\large]
{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,red), colored(s,green),
colored(b,green), colored(n,blue)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,green), colored(s,red),
colored(b,red), colored(n,blue)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,red), colored(s,blue),
colored(b,blue), colored(n,green)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,blue), colored(s,red),
colored(b,red), colored(n,green)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,blue), colored(s,green),
colored(b,green), colored(n,red)}

{edge(a,s), edge(a,b), edge(a,n), edge(s,n), edge(b,n), node(a),
node(s), node(b), node(n), colored(a,green), colored(s,blue),
colored(b,blue), colored(n,red)}

```

- This program does not seem to contain weak negation

Actually, there is one hidden in the program constraint (PC)

- If we want to transform a PC, say: $\leftarrow B(\bar{x})$ into a regular headed-rule, we need something like: $q \leftarrow B(\bar{x}), \text{ not } q$

Here, q a fresh, propositional atom

If in a Herbrand structure $B(\bar{a})$ becomes true for some \bar{a} , then in it $q \leftarrow \text{ not } q$ will hold

No model can make this hold, so the structure cannot be a model (the same effect as discarding it via the original PC)

- This particular program falls in a nice syntactic class of disjunctive programs with negation (head-cycle-free programs)

For it, the rule (*) can be replaced -in a standard manner- by three non-disjunctive rules (cf. slide 23)

```
colored(X,red) :- node(X), not colored(X,green),  
                not colored(X,blue).
```

ETC.

The new program has the same stable models as the original one

Not every program allows this transformation

- If we try to paint the map with only two colors, we would get no stable model

The meaning of not having stable models is that the set of rules of the program is inconsistent, and then any formula is a consequence of it

- What is the meaning of an empty stable model?

Example: Program $p(X) :- q(X).$

DLV will return one empty stable model: $\{\}$, which means that no atom is a consequence of the program

Safe Rules in DLV: Each rule in a DLV program has to be **safe** in order to avoid operations on infinite predicates:

- Variable in the head of a rule has to be in a positive literal in the body of the same rule

$path(X, X) \leftarrow$ is not safe
 $path(X, X) \leftarrow dom(X)$ is safe

- Variables in a *not*-negated literal, must also be in a positive literal in the body

$flies(X) \leftarrow bird(X), not\ ping(X)$ is safe
 $\neg P(X) \leftarrow R(X), not\ T(X, Y)$ is not safe

However, the second one can be written as:

$\neg P(X) \leftarrow R(X), not\ S(X).$
 $S(X) \leftarrow T(X, Y).$

- Variables in a built-in comparison predicate, must also appear in a positive literal in the body

$P(X, Y) \leftarrow R(X), X < Y$ is not safe

$P(Z) \leftarrow R(Z, X), S(Z, Y), X < Y$ is safe

Example: Consider a student database with two tables
 $Student(ID, Name)$ and $Courses(ID, Course, Status)$

The status can be pass or fail

We want to retrieve the names of the students that haven't failed courses

Consider the following rules? Do they give the expected answers?
Are they safe? (test them with DLV)

1. $Ans(N) \leftarrow St(ID, N), not C(ID, C, failed).$

It doesn't give the right semantic nor is safe

2. $Ans(N) \leftarrow St(ID, N), not aux(ID, failed).$
 $aux(ID, S) \leftarrow C(ID, C, S).$

It is safe, but it doesn't give the right semantics

3. $Ans(N) \leftarrow St(ID, N), not aux(ID).$
 $aux(ID) \leftarrow C(ID, C, failed).$

It is safe and gives the right semantics

Queries in DLV: We have seen how to generate the stable models (answer sets)

To determine if an atom is a logical consequence of a program we may consider two different semantics: cautious and brave

You can use them explicitly in DLV by running one of the following commands:

```
> DLV -brave test1  
> DLV -cautious test1
```

The query is written at end of input file; it can be a conjunction of ground literals (no variables, but see later)

```
not P(a) ?      Q(a,b), -S(b) ?      not ~a, not b ?
```

The predicates in the query have to be IDB predicates

Example: A non-stratified program:

```
% EDB
  s(a,b).
  s(e,e).
% IDB
  p(X) :- s(X,Y), not t(X).
  t(X) :- s(X,Y), not p(Y).
```

The stable models obtained from DLV are:

```
{s(a,b), s(e,e), t(a), p(e)}
{s(a,b), s(e,e), t(a), t(e)}
```

The following table has the answers for different queries and semantics given by DLV

Query	Semantic	Answer
t(a)?	brave	t(a) is bravely true, evidenced by {s(a,b), s(e,e), t(a), p(e)}
t(a)?	cautious	t(a) is cautiously true
t(e)?	brave	t(e) is bravely true, evidenced by {s(a,b), s(e,e), t(a), t(e)}
t(e)?	cautious	t(e) is cautiously false, evidenced by {s(a,b), s(e,e), t(a), p(e)}
-t(f)?	brave	-t(f) is bravely false
not t(f)?	brave	not t(f) is bravely true, evidenced by {s(a,b), s(e,e), t(a), p(e)}

If we want to pose complex queries? For example, give all the possible values of the first attribute of predicate s

We can use an auxiliary predicate Ans , and add a new rule to the program:

$$Ans(X) \leftarrow s(X, Y).$$

After that, we run DLV to generate all the stable models, that now include extensions for the Ans predicate

If we added that rule to the program in the previous example and ran it in DLV, we would get:

```
{s(a,e), s(e,e), Ans(a), Ans(e), p(e), p(a)}  
{s(a,e), s(e,e), Ans(a), Ans(e), t(a), t(e)}
```

The the cautions and brave answers to the query are a and e

Latest versions of DLV accept some forms of SQL3 queries ...