Logic: overview

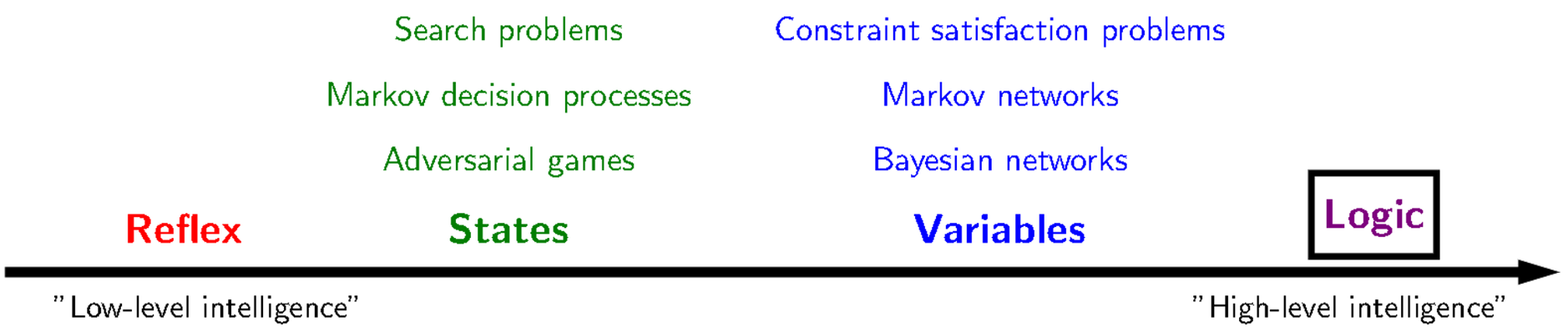
answer in chat

Question

If $X_1 + X_2 = 10$ and $X_1 - X_2 = 4$, what is X_1?

- Think about how you solved this problem. You could treat it as a CSP with variables X_1 and X_2, and search through the set of candidate solutions, checking the constraints.
- However, more likely, you just added the two equations, divided both sides by 2 to easily find out that $X_1 = 7$. This is the power of **logical inference**, where we apply a set of truth-preserving rules to arrive at the answer. This is in contrast to what is called **model checking** (for reasons that will become clear), which tries to directly find assignments.
- We'll see that logical inference allows you to perform very powerful manipulations in a very compact way. This allows us to vastly increase the representational power of our models.

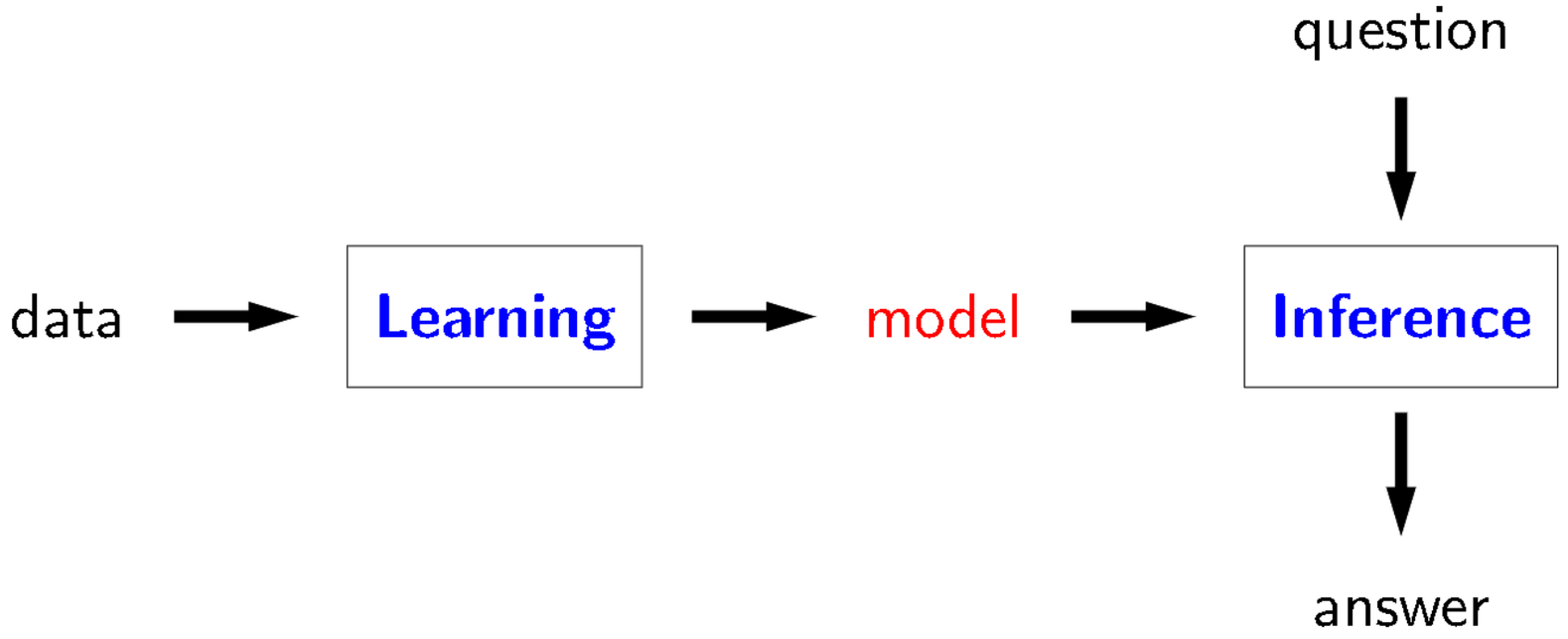
Course plan



Machine learning

- We are at the last stage of our journey through the AI topics of this course: logic. Before launching in, let's take a moment to reflect.

Taking a step back



Examples: search problems, MDPs, games, CSPs, Bayesian networks

- For each topic (e.g., MDPs) that we've studied, we followed the modeling-inference-learning paradigm: We take some data, feed it into a learning algorithm to produce a model with tuned parameters. Then we take this model and use it to perform inference (turning questions into answers).
- For search problems, the question is "what is the minimum cost path?" Inference algorithms such as DFS, UCS or A* produced the minimum cost path. Learning algorithms such as the structured Perceptron filled in the action costs based on data (minimum cost paths).
- For MDPs and games, the question is "what is the maximum value policy?" Inference algorithms such as value iteration or minimax produced this. Learning algorithms such as Q-learning or TD learning allow you to work when we don't know the transitions and rewards.
- For CSPs, the question is "what is the maximum weight assignment?" Inference algorithms such as backtracking search, beam search, or variable elimination find such an assignment. We did not discuss learning algorithms here, but something similar to the structured Perceptron works.
- For Bayesian networks, the question is "what is the probability of a query given evidence?" Inference algorithms such as Gibbs sampling and particle filtering compute these probabilistic inference queries. Learning: if we don't know the local conditional distributions, we can learn them using maximum likelihood.
- We can think of learning as induction, where we need to generalize, and inference as deduction, where it's about computing the best predicted answer under the model.

Modeling paradigms

State-based models: search problems, MDPs, games

Applications: route finding, game playing, etc.

Think in terms of **states, actions, and costs**

Variable-based models: CSPs, Bayesian networks

Applications: scheduling, tracking, medical diagnosis, etc.

Think in terms of **variables and factors**

Logic-based models: propositional logic, first-order logic

Applications: theorem proving, verification, reasoning

Think in terms of **logical formulas and inference rules**

- Each topic corresponded to a modeling paradigm. The way the modeling paradigm is set up influences the way we approach a problem.
- In state-based models, we thought about inference as finding minimum cost paths in a graph. This leads us to think in terms of states, actions, and costs.
- In variable-based models, we thought about inference as finding maximum weight assignments or computing conditional probabilities. There we thought about variables and factors.
- Now, we will talk about logic-based models, where inference is applying a set of rules. For these models, we will think in terms of logical formulas and inference rules.

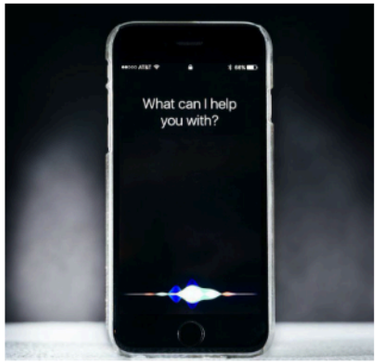
A historical note

- Logic was dominant paradigm in AI before 1990s

- Problem 1: deterministic, didn't handle **uncertainty** (probability addresses this)
- Problem 2: rule-based, didn't allow fine tuning from **data** (machine learning addresses this)
- Strength: provides **expressiveness** in a compact way

- Historically, in AI, logic was the dominant paradigm before the 1990s, but this tradition fell out of favor with the rise of probability and machine learning.
- There were two reasons for this: First, logic as an inference mechanism was brittle and did not handle uncertainty, whereas probability offered a coherent framework for dealing with uncertainty.
- Second, people built rule-based systems which were tedious and did not scale up, whereas machine learning automated much of the fine-tuning of a system by using data.
- However, there is one strength of logic which has not quite yet been recouped by existing probability and machine learning methods, and that is the expressivity of the model.

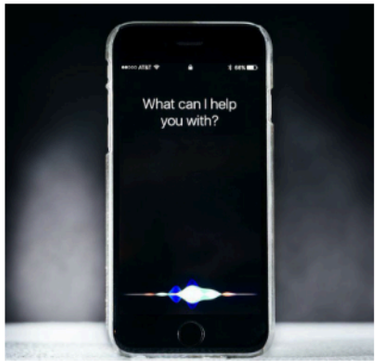
Motivation: smart personal assistant



- How can we motivate logic-based models? We will take a little bit of a detour and think about an AI grand challenge: building smart personal assistants.
- Today, we have systems like Apple's Siri, Microsoft's Cortana, Amazon's Alexa, and Google Assistant.

Motivation: smart personal assistant

Tell information



Ask questions

Use natural language!

[demo: `python nli.py`]

Need to:

- Digest **heterogenous** information
- Reason **deeply** with that information

- We would like to have more intelligent assistants such as Data from Star Trek. What is the functionality that's missing in between?
- At an abstract level, one fundamental thing a good personal assistant should be able to do is to take in information from people and be able to answer questions that require drawing inferences from the facts.
- In some sense, telling the system information is like machine learning, but it feels like a very different form of learning than seeing 10M images and their labels or 10M sentences and their translations. The type of information we get here is both more heterogenous, more abstract, and the expectation is that we process it more deeply (we don't want to have to tell our personal assistant 100 times that we prefer morning meetings).
- And how do we interact with our personal assistants? Let's use natural language, the very tool that was built for communication!

Natural language

Example:

- A **dime** is better than a **nickel**.
- A **nickel** is better than a **penny**.
- Therefore, a **dime** is better than a **penny**.

Example:

- A **penny** is better than **nothing**.
- **Nothing** is better than **world peace**.
- Therefore, a **penny** is better than **world peace**???

Natural language is slippery...

- But natural language is tricky, because it is replete with ambiguities and vagueness. And drawing inferences using natural languages can be quite slippery. Of course, some concepts are genuinely vague and slippery, and natural language is as good as it gets, but that still leaves open the question of how a computer would handle those cases.

Language

Language *is a mechanism for expression.*

Natural languages (informal):

English: *Two divides even numbers.*

German: *Zwei dividieren geraden zahlen.*

Programming languages (formal):

Python:
```
def even(x):  return x % 2 == 0
```

C++:
```
bool even(int x) { return x % 2 == 0; }
```

Logical languages (formal):

First-order-logic: $\forall x.\text{Even}(x) \rightarrow \text{Divides}(x, 2)$

- Let's think about language a bit deeply. What does it really buy you? Primarily, language is this wonderful human creation that allows us to express and communicate complex ideas and thoughts.
- We have mostly been talking about natural languages such as English and German. But as you all know, there are programming languages as well, which allow one to express computation formally so that a computer can understand it.
- This lecture is mostly about logical languages such as propositional logic and first-order logic. These are formal languages, but are a more suitable way of capturing declarative knowledge rather than concrete procedures, and are better connected with natural language.

Two goals of a logic language

- **Represent** knowledge about the world

- **Reason** with that knowledge

- Some of you already know about logic, but it's important to keep the AI goal in mind: We want to use it to represent knowledge, and we want to be able to reason (or do inference) with that knowledge.
- Finally, we need to keep in mind that our goal is to get computers to use logic automatically, not for you to do it. This means that we need to think very mechanistically.

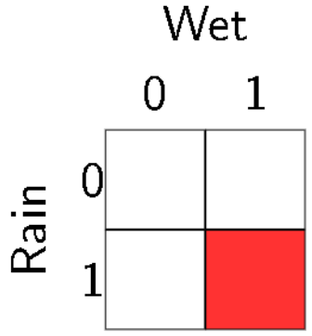
Ingredients of a logic

Syntax: defines a set of valid **formulas** (Formulas)

Example: Rain $\wedge$ Wet

Semantics: for each formula, specify a set of **models** (assignments / configurations of the world)

Example:



Inference rules: given f, what new formulas g can be added that are guaranteed to follow $(\frac{f}{g})$?

Example: from Rain $\wedge$ Wet, derive Rain

- The **syntax** defines a set of valid formulas, which are things which are grammatical to say in the language.
- **Semantics** usually doesn't receive much attention if you have a casual exposure to logic, but this is really the important piece that makes logic rigorous. Formally, semantics specifies the meaning of a formula, which in our setting is a set of configurations of the world in which the formula holds. This is what we care about in the end.
- But in order to get there, it's helpful to operate directly on the syntax using a set of **inference rules**. For example, if I tell you that it's raining and wet, then you should be able to conclude that it is also raining (obviously) without even explicitly mentioning semantics. Most of the time when people do logic casually, they are really just applying inference rules.

Syntax versus semantics

Syntax: what are valid expressions in the language?

Semantics: what do these expressions mean?

Different syntax, same semantics (5):

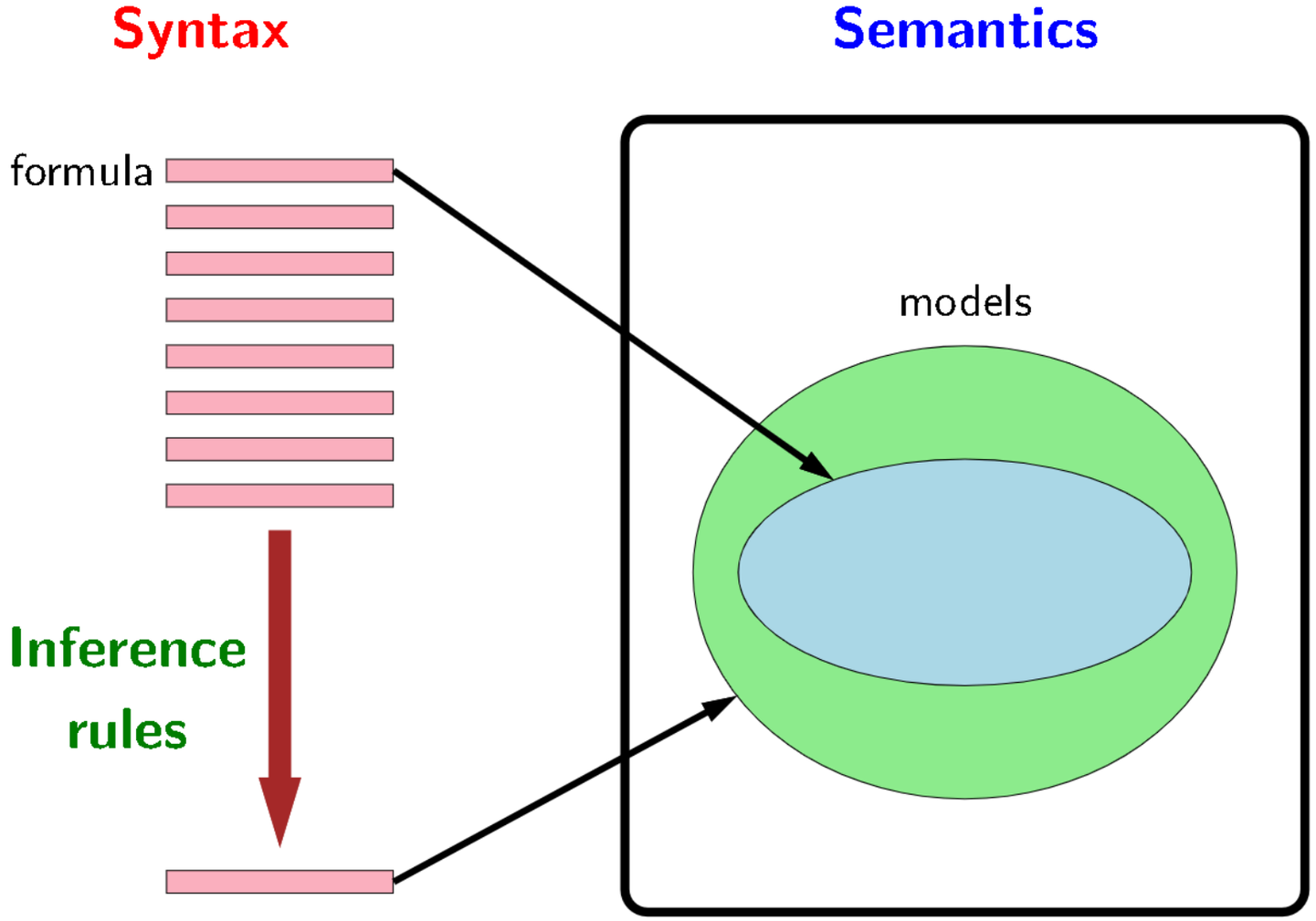
$$2 + 3 \Leftrightarrow 3 + 2$$

Same syntax, different semantics (1 versus 1.5):

$$3 \ / \ 2 \text{ (Python 2.7)} \not\Leftrightarrow 3 \ / \ 2 \text{ (Python 3)}$$

- Just to hammer in the point that syntax and semantics are different, consider two examples from programming languages.
- First, the formula `2 + 3` and `3 + 2` are superficially different (a syntactic notion), but they have the same semantics (5).
- Second, the formula `3 / 2` means something different depending on which language. In Python 2.7, the semantics is 1 (integer division), and in Python 3 the semantics is 1.5 (floating point division).

Propositional logic



Logics

- **Propositional logic with only Horn clauses**
- **Propositional logic**
- Modal logic
- **First-order logic with only Horn clauses**
- **First-order logic**
- Second-order logic
- ...

Key idea: tradeoff

Balance **expressivity** and **computational efficiency**.

- There are many different logical languages, just like there are programming languages. Whereas most programming languages have the expressive power (all Turing complete), logical languages exhibit a larger spectrum of expressivity.
- The bolded items are the ones we will discuss in this class.

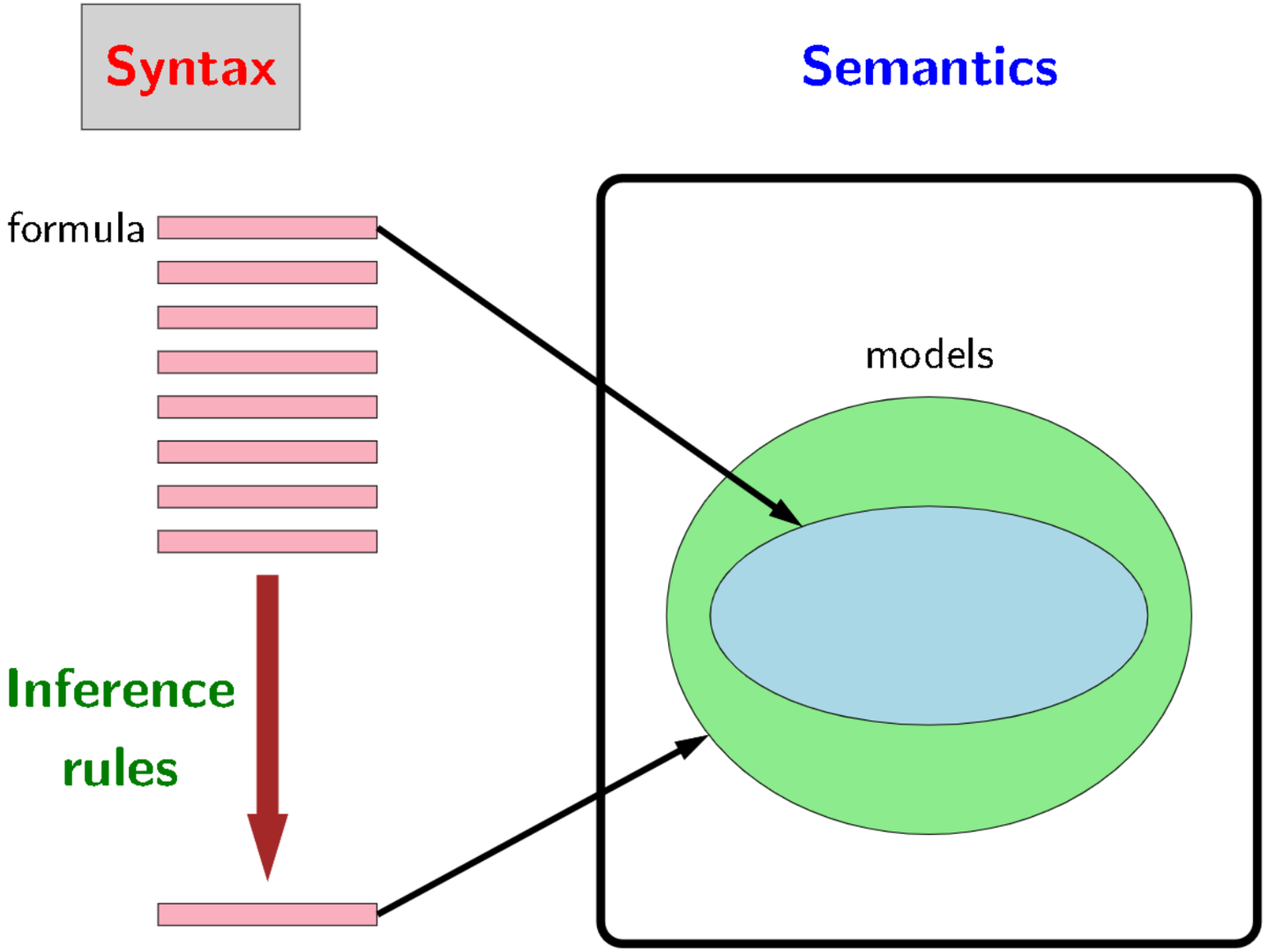
Roadmap

Modeling	Inference
Propositional Logic Syntax	Inference Rules
Propositional Logic Semantics	Propositional modus ponens
First-order Logic	Propositional resolution
	First-order modus ponens
	First-order resolution

- Here are the rest of the modules under the logic unit.
- We will start by talking about the core elemenst of logics: syntax, semantics, and inference rules. We will start by defining syntax and semantics for propositional logic.
- We will then discuss a set of inference rules for propostional logic including modus ponens and resolution. We will discuss soundness and correctness of these inference algorithms.
- We then describe a more expressive logic, i.e., first order logic. We will go over its syntax and semantics, and then extend the notions of modus ponens and resolution to first order logic using unification and substitution.

Logic: propositional logic syntax

Propositional logic



- We begin with the syntax of propositional logic: what are the allowable formulas?

Syntax of propositional logic

Propositional symbols (atomic formulas): A, B, C

Logical connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Build up formulas recursively—if f and g are formulas, so are the following:

- Negation: $\neg f$
- Conjunction: $f \wedge g$
- Disjunction: $f \vee g$
- Implication: $f \rightarrow g$
- Biconditional: $f \leftrightarrow g$

- The building blocks of the syntax are the propositional symbols and connectives. The set of propositional symbols can be anything (e.g., A, Wet, etc.), but the set of connectives is fixed to these five.
- All the propositional symbols are **atomic formulas** (also called atoms). We can **recursively** create larger formulas by combining smaller formulas using connectives.

Syntax of propositional logic

- Formula: A
- Formula: $\neg A$
- Formula: $\neg B \rightarrow C$
- Formula: $\neg A \wedge (\neg B \rightarrow C) \vee (\neg B \vee D)$
- Formula: $\neg\neg A$
- Non-formula: $A\neg B$
- Non-formula: $A + B$

- Here are some examples of valid and invalid propositional formulas.

Syntax of propositional logic

Key idea: syntax provides symbols

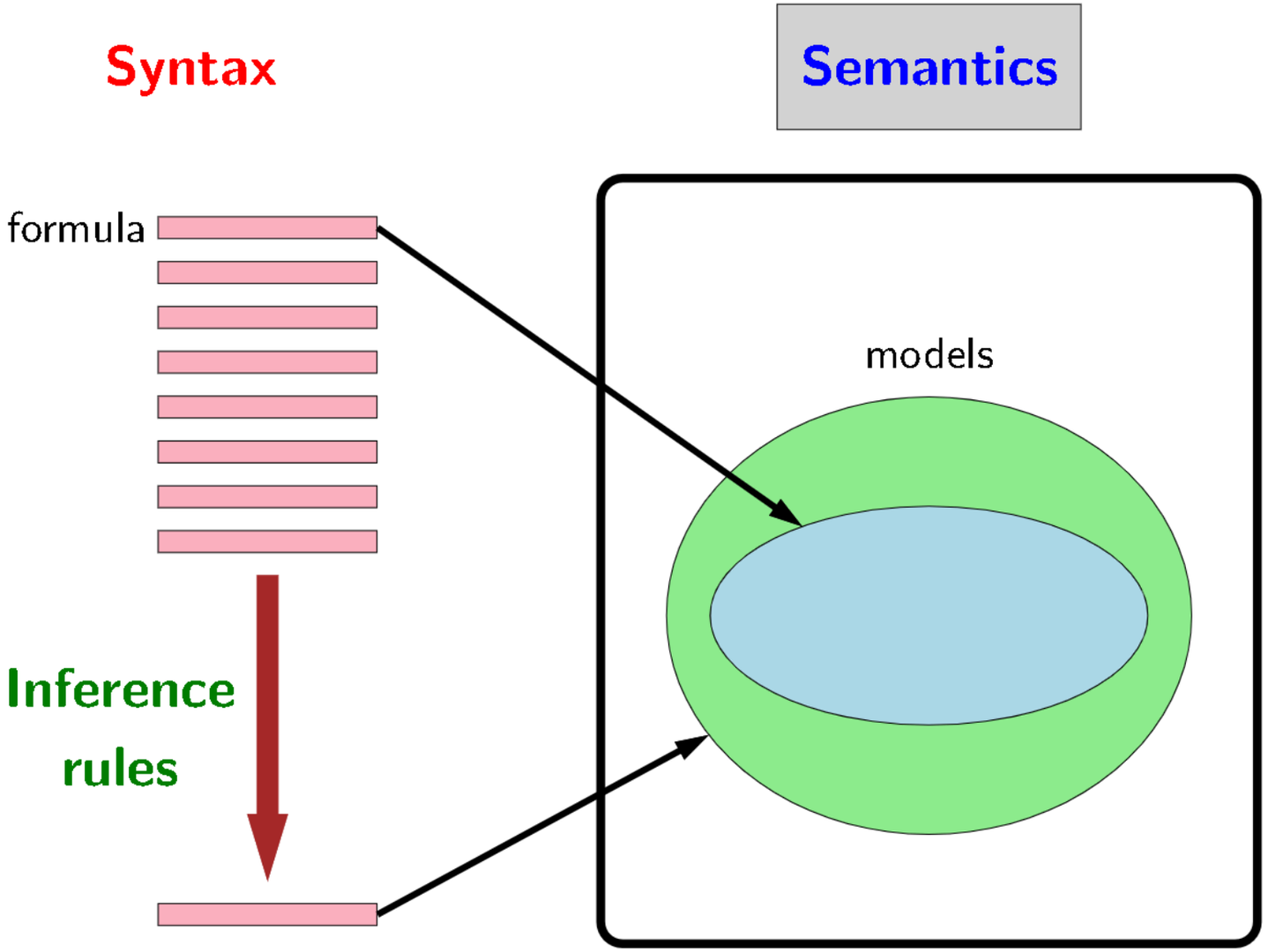
Formulas by themselves are just symbols (syntax).

No meaning yet (semantics)!

- It's important to remember that whenever we talk about syntax, we're just talking about symbols; we're not actually talking about what they mean — that's the role of semantics. Of course it will be difficult to ignore the semantics for propositional logic completely because you already have a working knowledge of what the symbols mean.

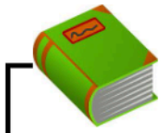
Logic: propositional logic semantics

Propositional logic



- Having defined the syntax of propositional logic, let's talk about their semantics or meaning.

Model



Definition: model

A **model** w in propositional logic is an **assignment** of truth values to propositional symbols.

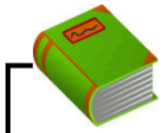
Example:

- 3 propositional symbols: A, B, C
- $2^3 = 8$ possible models w:

$$\{A:0, B:0, C:0\}$$
$$\{A:0, B:0, C:1\}$$
$$\{A:0, B:1, C:0\}$$
$$\{A:0, B:1, C:1\}$$
$$\{A:1, B:0, C:0\}$$
$$\{A:1, B:0, C:1\}$$
$$\{A:1, B:1, C:0\}$$
$$\{A:1, B:1, C:1\}$$

- In logic, the word **model** has a special meaning, quite distinct from the way we've been using it in the class (quite an unfortunate collision). A model (in the logical sense) represents a possible state of affairs in the world. In propositional logic, this is an assignment that specifies a truth value (true or false) for each propositional symbol.

Interpretation function



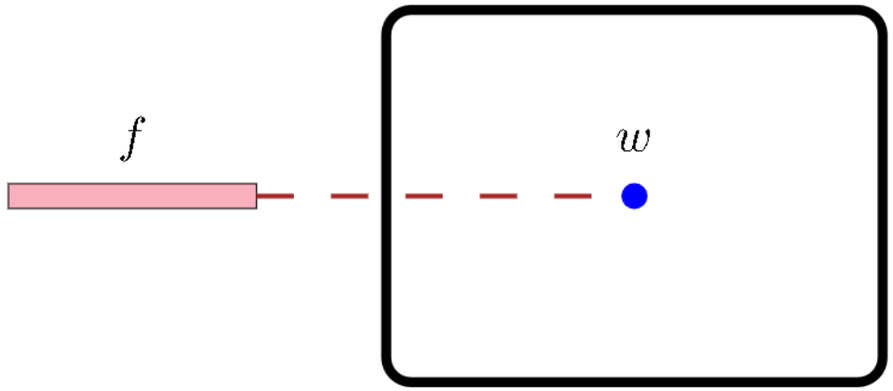
Definition: interpretation function

Let f be a formula.

Let w be a model.

An **interpretation function** $\mathcal{I}(f, w)$ returns:

- true (1) (say that w satisfies f)
- false (0) (say that w does not satisfy f)



- The semantics is given by an **interpretation function**, which takes a formula f and a model w, and returns whether w satisfies f. In other words, is f true in w?
- For example, if f represents "it is Wednesday" and w corresponds to right now, then $\mathcal{I}(f, w) = 1$. If w corresponded to yesterday, then $\mathcal{I}(f, w) = 0$.

Interpretation function: definition

Base case:

- For a propositional symbol p (e.g., A, B, C): $\mathcal{I}(p, w) = w(p)$

Recursive case:

- For any two formulas f and g, define:

$\mathcal{I}(f, w)$	$\mathcal{I}(g, w)$	$\mathcal{I}(\neg f, w)$	$\mathcal{I}(f \wedge g, w)$	$\mathcal{I}(f \vee g, w)$	$\mathcal{I}(f \rightarrow g, w)$	$\mathcal{I}(f \leftrightarrow g, w)$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

- The interpretation function is defined recursively, where the cases neatly parallel the definition of the syntax.
- Formally, for propositional logic, the interpretation function is fully defined as follows. In the base case, the interpretation of a propositional symbol p is just gotten by looking p up in the model w. For every possible value of $(\mathcal{I}(f, w), \mathcal{I}(g, w))$, we specify the interpretation of the combination of f and g.

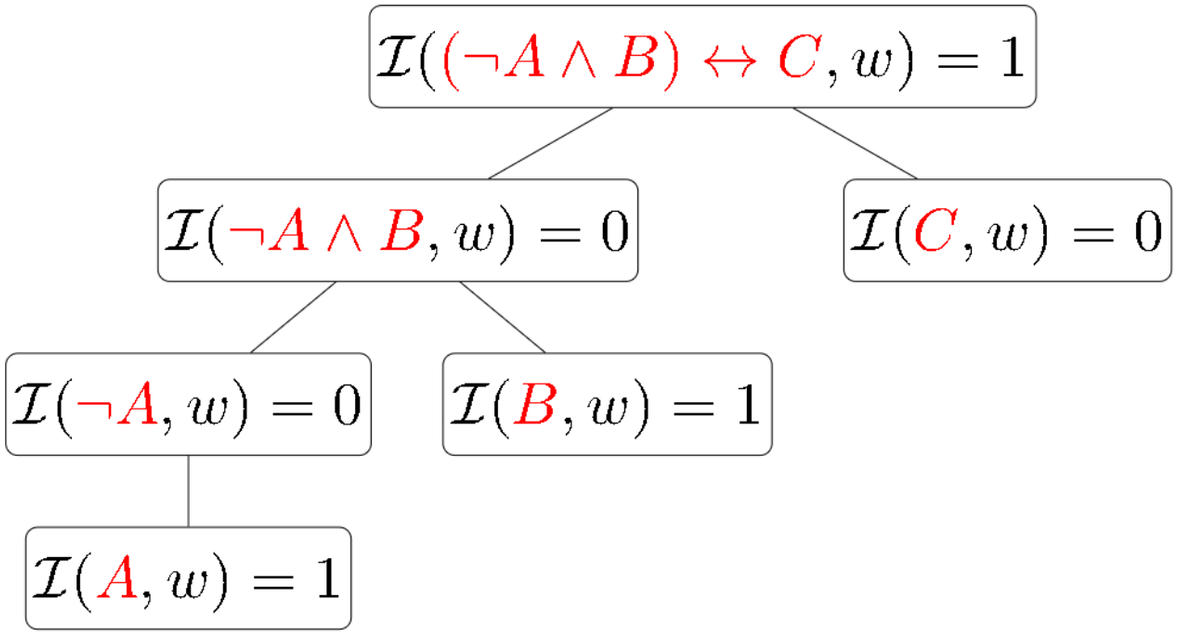
Interpretation function: example

Example: interpretation function

Formula: $f = (\neg A \wedge B) \leftrightarrow C$

Model: $w = \{A : 1, B : 1, C : 0\}$

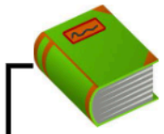
Interpretation:



- For example, given the formula, we break down the formula into parts, recursively compute the truth value of the parts, and then finally combines these truth values based on the connective.

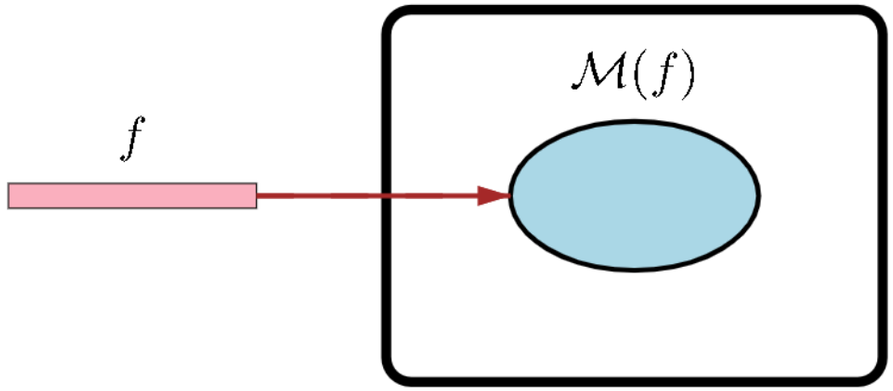
Formula represents a set of models

So far: each formula f and model w has an interpretation $\mathcal{I}(f, w) \in \{0, 1\}$



Definition: models

Let $\mathcal{M}(f)$ be the set of **models** w for which $\mathcal{I}(f, w) = 1$.



- So far, we've focused on relating a single model. A more useful but equivalent way to think about semantics is to think about the formula $\mathcal{M}(f)$ as **a set of models** — those for which $\mathcal{I}(f, w) = 1$.

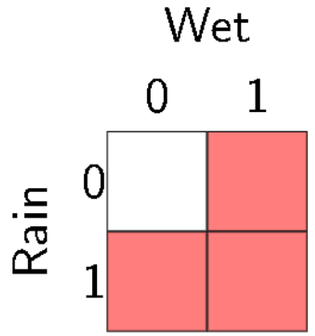
Models: example

Formula:

$$f = \text{Rain} \vee \text{Wet}$$

Models:

$\mathcal{M}(f) =$

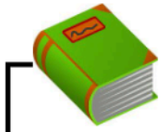


Key idea: compact representation

A **formula** *compactly* represents a set of **models**.

- In this example, there are four models for which the formula holds, as one can easily verify. From the point of view of $\mathcal{M}$, a formula's main job is to define a set of models.
- Recall that a model is a possible configuration of the world. So a formula like "it is raining" will pick out all the hypothetical configurations of the world where it's raining; in some of these configurations, it will be Wednesday; in others, it won't.

Knowledge base



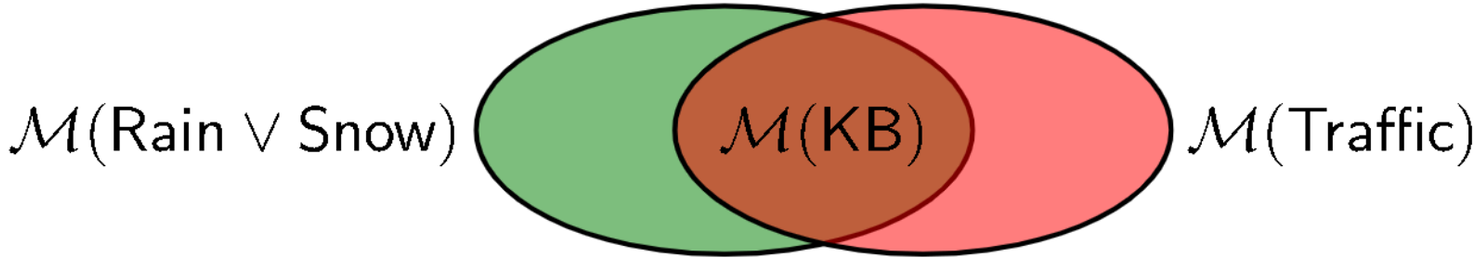
Definition: Knowledge base

A **knowledge base** KB is a set of formulas representing their conjunction / intersection:

$$\mathcal{M}(\text{KB}) = \bigcap_{f \in \text{KB}} \mathcal{M}(f).$$

Intuition: KB specifies constraints on the world. $\mathcal{M}(\text{KB})$ is the set of all worlds satisfying those constraints.

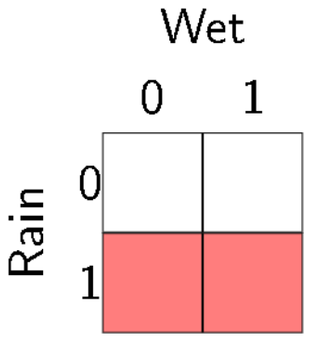
Let $\text{KB} = \{\text{Rain} \vee \text{Snow}, \text{Traffic}\}$.



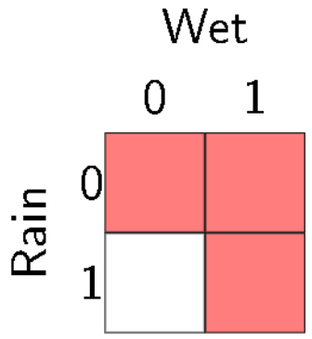
- If you take a set of formulas, you get a **knowledge base**. Each knowledge base defines a set of models — exactly those which are satisfiable by all the formulas in the knowledge base.
- Think of each formula as a fact that you know, and the **knowledge** is just the collection of those facts. Each fact narrows down the space of possible models, so the more facts you have, the fewer models you have.

Knowledge base: example

$\mathcal{M}(\text{Rain})$

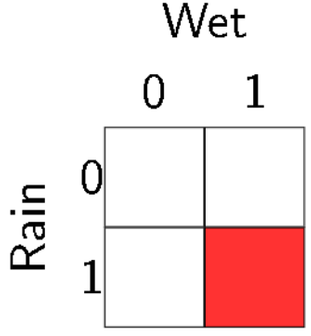


$\mathcal{M}(\text{Rain} \to \text{Wet})$



Intersection:

$\mathcal{M}(\{\text{Rain}, \text{Rain} \to \text{Wet}\})$



- As a concrete example, consider the two formulas Rain and Rain $\rightarrow$ Wet. If you know both of these facts, then the set of models is constrained to those where it is raining and wet.

Adding to the knowledge base

Adding more formulas to the knowledge base:

$$\text{KB} \longrightarrow \text{KB} \cup \{f\}$$

Shrinks the set of models:

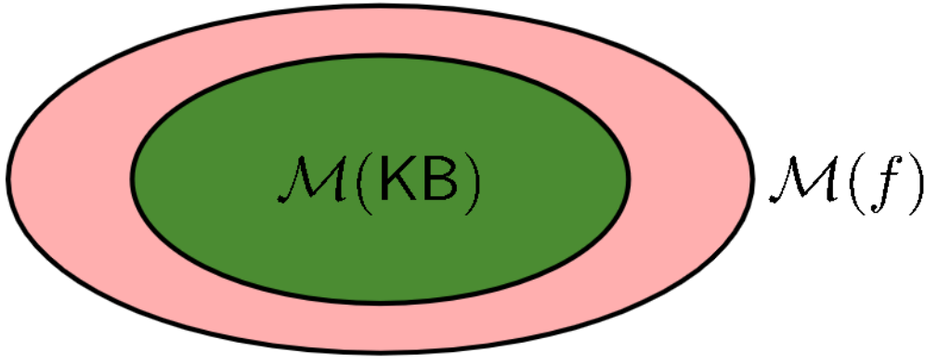
$$\mathcal{M}(\text{KB}) \longrightarrow \mathcal{M}(\text{KB}) \cap \mathcal{M}(f)$$

How much does $\mathcal{M}(\text{KB})$ shrink?

[whiteboard]

- We should think about a knowledge base as carving out a set of models. Over time, we will add additional formulas to the knowledge base, thereby winnowing down the set of models.
- Intuitively, adding a formula to the knowledge base imposes yet another constraint on our world, which naturally decreases the set of possible worlds.
- Thus, as the number of formulas in the knowledge base gets larger, the set of models gets smaller.
- A central question is how much f shrinks the set of models. There are three cases of importance.

Entailment



Intuition: f added no information/constraints (it was already known).

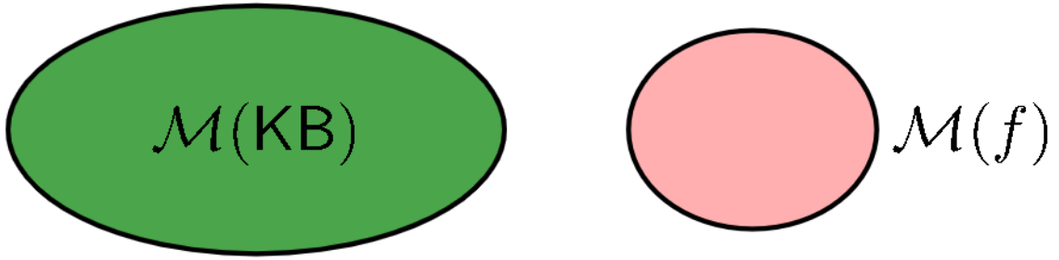
Definition: entailment

KB entails f (written KB $\models f$) iff $\mathcal{M}(\text{KB}) \subseteq \mathcal{M}(f)$.

Example: Rain $\wedge$ Snow $\models$ Snow

- The first case is if the set of models of f is a superset of the models of KB, then f adds no information. We say that KB **entails** f.

Contradiction



Intuition: f contradicts what we know (captured in KB).

Definition: contradiction

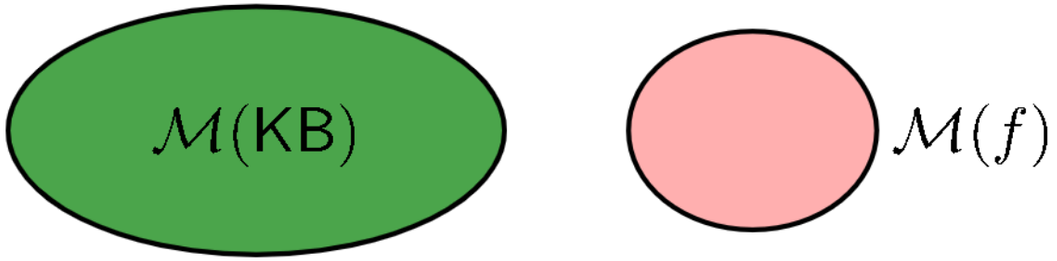
KB contradicts f iff $\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \emptyset$.

Example: Rain $\wedge$ Snow contradicts $\neg$Snow

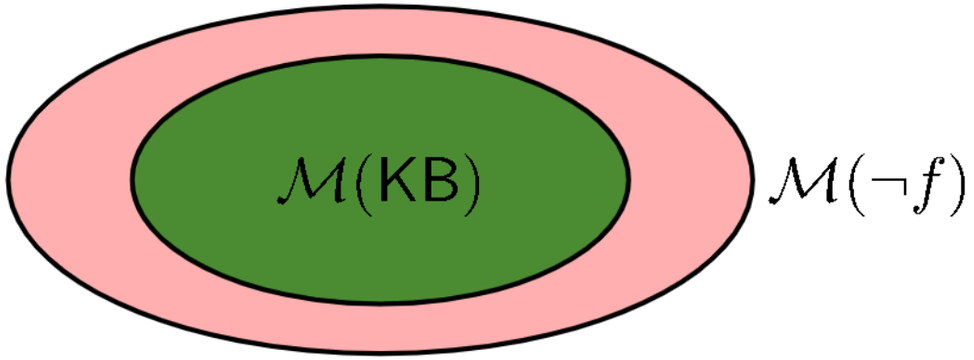
- The second case is if the set of models defined by f is completely disjoint from those of KB. Then we say that the KB and f **contradict** each other. If we believe KB, then we cannot possibly believe f.

Contradiction and entailment

Contradiction:



Entailment:

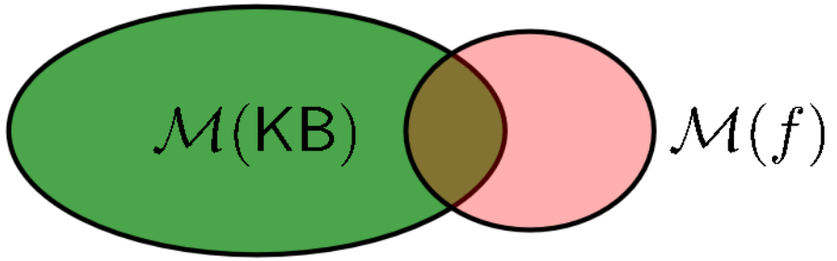


Proposition: contradiction and entailment

KB **contradicts** f iff KB **entails** $\neg f$.

- There is a useful connection between entailment and contradiction. If f is contradictory, then its negation $(\neg f)$ is entailed, and vice-versa.
- You can see this because the models $\mathcal{M}(f)$ and $\mathcal{M}(\neg f)$ partition the space of models.

Contingency



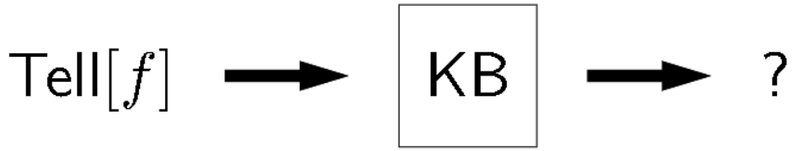
Intuition: f adds non-trivial information to KB

$$\emptyset \subsetneq \mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \subsetneq \mathcal{M}(\text{KB})$$

Example: Rain and Snow

- In the third case, we have a non-trivial overlap between the models of KB and f. We say in this case that f is **contingent**; f could be satisfied or not satisfied depending on the model.

Tell operation



Tell: *It is raining.*

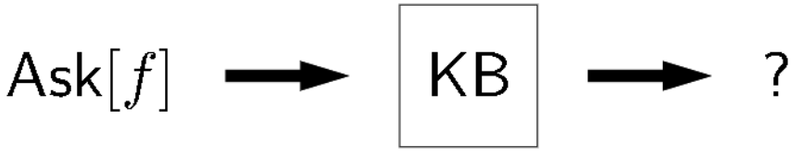
Tell[Rain]

Possible responses:

- Already knew that: entailment (KB $\models f$)
- Don't believe that: contradiction (KB $\models \neg f$)
- Learned something new (update KB): contingent

- Having defined the three possible relationships that a new formula f can have with respect to a knowledge base KB, let's try to determine the appropriate response that a system should have.
- Suppose we tell the system that it is raining ($f = \text{Rain}$). If f is entailed, then we should reply that we already knew that. If f contradicts the knowledge base, then we should reply that we don't believe that. If f is contingent, then this is the interesting case, where we have non-trivially restricted the set of models, so we reply that we've learned something new.

Ask operation



Ask: *Is it raining?*

Ask[Rain]

Possible responses:

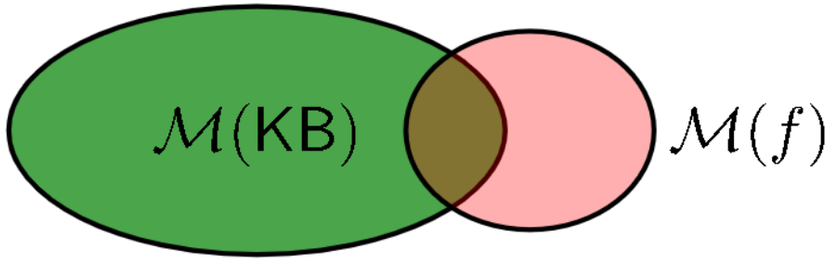
- Yes: entailment (KB $\models f$)
- No: contradiction (KB $\models \neg f$)
- I don't know: contingent

- Suppose now that we ask the system a question: is it raining? If f is entailed, then we should reply with a definitive yes. If f contradicts the knowledge base, then we should reply with a definitive no. If f is contingent, then we should just confess that we don't know.

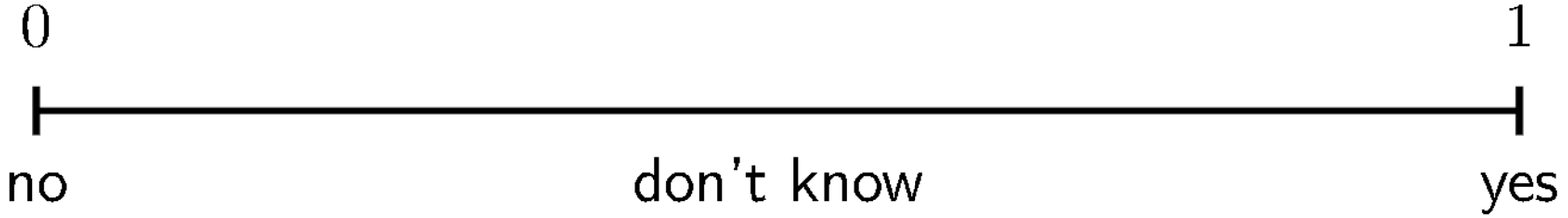
Digression: probabilistic generalization

Bayesian network: distribution over assignments (models)

w	$\mathbb{P}(W = w)$
{ A: 0, B: 0, C: 0 }	0.3
{ A: 0, B: 0, C: 1 }	0.1
...	...

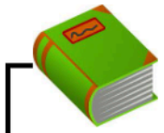


$$\mathbb{P}(f \mid \text{KB}) = \frac{\sum_{w \in \mathcal{M}(\text{KB} \cup \{f\})} \mathbb{P}(W = w)}{\sum_{w \in \mathcal{M}(\text{KB})} \mathbb{P}(W = w)}$$



- Note that logic captures uncertainty in a very crude way. We can't say that we're almost sure or not very sure or not sure at all.
- Probability can help here. Remember that a Bayesian network (or more generally a factor graph) defines a distribution over assignments to the variables in the Bayesian network. Then we could ask questions such as: conditioned on having a cough but not itchy eyes, what's the probability of having a cold?
- Recall that in propositional logic, models are just assignments to propositional symbols. So we can think of KB as the evidence that we're conditioning on, and f as the query.

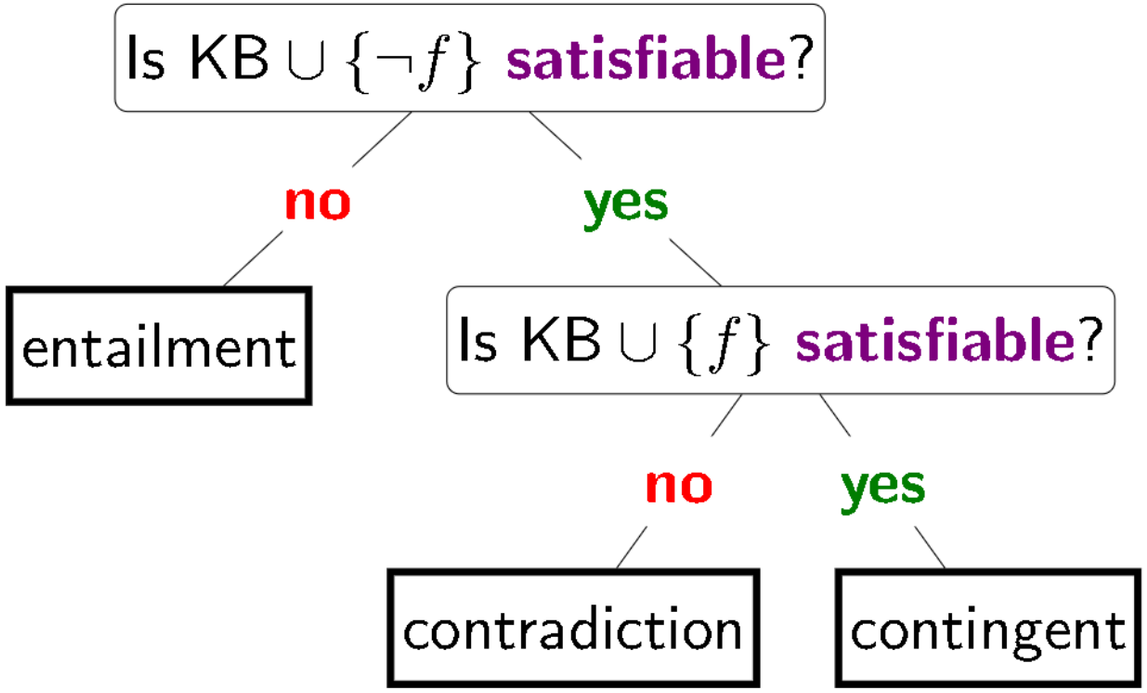
Satisfiability



Definition: satisfiability

A knowledge base KB is **satisfiable** if $\mathcal{M}(\text{KB}) \neq \emptyset$.

Reduce Ask$[f]$ and Tell$[f]$ to satisfiability:



- Now let's return to pure logic land again. How can we go about actually checking entailment, contradiction, and contingency? One useful concept to rule them all is **satisfiability**.
- Recall that we said a particular model w satisfies f if the interpretation function returns true $\mathcal{I}(f, w) = 1$. We can say that a formula f by itself is satisfiable if there is some model that satisfies f. Finally, a knowledge base (which is no more than just the conjunction of its formulas) is satisfiable if there is some model that satisfies all the formulas $f \in \text{KB}$.
- With this definition in hand, we can implement Ask$[f]$ and Tell$[f]$ as follows:
- First, we check if $\text{KB} \cup \{\neg f\}$ is satisfiable. If the answer is no, that means the models of $\neg f$ and KB don't intersect (in other words, the two contradict each other). Recall that this is equivalent to saying that KB entails f.
- Otherwise, we need to do another test: check whether $\text{KB} \cup \{f\}$ is satisfiable. If the answer is no here, then KB and f are contradictory. Otherwise, we have that both f and $\neg f$ are compatible with KB, so the result is contingent.

Model checking

Checking satisfiability (SAT) in propositional logic is special case of solving CSPs!

Mapping:

propositional symbol	$\Rightarrow$	variable
formula	$\Rightarrow$	constraint
model	$\Leftarrow$	assignment

- Now we have reduced the problem of working with knowledge bases to checking satisfiability. The bad news is that this is an (actually, the canonical) NP-complete problem, so there are no efficient algorithms in general.
- The good news is that people try to solve the problem anyway, and we actually have pretty good SAT solvers these days. In terms of this class, this problem is just a CSP, if we convert the terminology: Each propositional symbol becomes a variable and each formula is a constraint. We can then solve the CSP, which produces an assignment, or in logic-speak, a model.

Model checking

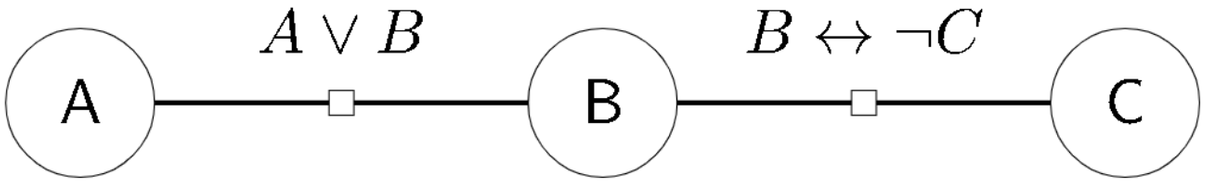
Example: model checking

KB $= \{A \vee B, B \leftrightarrow \neg C\}$

Propositional symbols (CSP variables):

$$\{A, B, C\}$$

CSP:

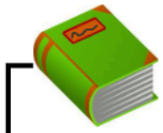


Consistent assignment (satisfying model):

$$\{A : 1, B : 0, C : 1\}$$

- As an example, consider a knowledge base that has two formulas and three variables. Then the CSP is shown. Solving the CSP produces a consistent assignment (if one exists), which is a model that satisfies KB.
- Note that in the knowledge base tell/ask application, we don't technically need the satisfying assignment. An assignment would only offer a counterexample certifying that the answer **isn't** entailment or contradiction. This is an important point: entailment and contradiction is a claim about all models, not about the existence of a model.

Model checking



Definition: model checking

Input: knowledge base KB

Output: exists satisfying model ($\mathcal{M}(\text{KB}) \neq \emptyset$)?

Popular algorithms:

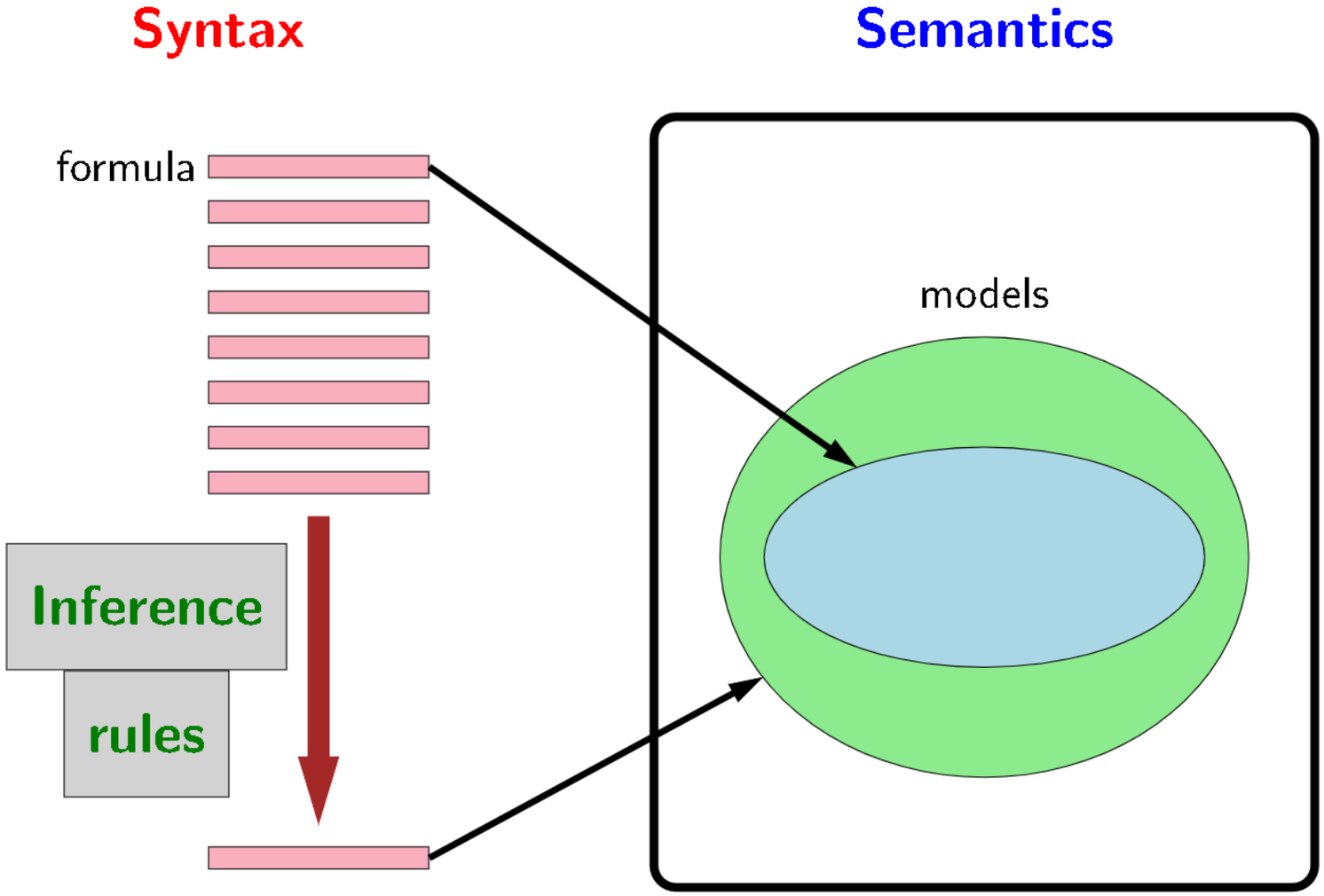
- DPLL (backtracking search + pruning)
- WalkSat (randomized local search)

Next: Can we exploit the fact that factors are formulas?

- Checking satisfiability of a knowledge base is called **model checking**. For propositional logic, there are several algorithms that work quite well which are based on the algorithms we saw for solving CSPs (backtracking search and local search).
- However, can we do a bit better? Our CSP factors are not arbitrary — they are logic formulas, and recall that formulas are defined recursively and have some compositional structure. Let's see how to exploit this.

Logic: inference rules

Propositional logic



- So far, we have used formulas, via semantics, to define sets of models. And all our reasoning on formulas has been through these models (e.g., reduction to satisfiability). Inference rules allow us to do reasoning on the formulas themselves without ever instantiating the models.
- This can be quite powerful. If you have a huge KB with lots of formulas and propositional symbols, sometimes you can draw a conclusion without instantiating the full model checking problem. This will be very important when we move to first-order logic, where the models can be infinite, and so model checking would be infeasible.

Inference rules

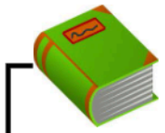
Example of making an inference:

It is raining. (Rain)

If it is raining, then it is wet. (Rain $\rightarrow$ Wet)

Therefore, it is wet. (Wet)

$$\frac{\text{Rain}, \quad \text{Rain} \rightarrow \text{Wet}}{\text{Wet}} \qquad \frac{(\text{premises})}{(\text{conclusion})}$$



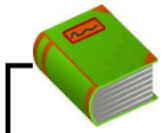
Definition: Modus ponens inference rule

For any propositional symbols p and q:

$$\frac{p, \quad p \rightarrow q}{q}$$

- The idea of making an inference should be quite intuitive to you. The classic example is **modus ponens**, which captures the if-then reasoning pattern.

Inference framework



Definition: inference rule

If $f_1, \ldots, f_k, g$ are formulas, then the following is an **inference rule**:

$$\frac{f_1, \quad \ldots \quad , f_k}{g}$$

Key idea: inference rules

Rules operate directly on **syntax**, not on **semantics**.

- In general, an inference rule has a set of premises and a conclusion. The rule says that if the premises are in the KB, then you can add the conclusion to the KB.
- We haven't yet specified whether this is a valid thing to do, but it is a thing to do. Remember, syntax is just about symbol pushing; it is only by linking to models that we have notions of truth and meaning (semantics).

Inference algorithm

Algorithm: forward inference

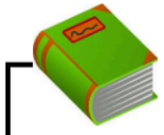
Input: set of inference rules Rules.

Repeat until no changes to KB:

  Choose set of formulas $f_1, \ldots, f_k \in$ KB.

  If matching rule $\frac{f_1, \quad \ldots \quad, f_k}{g}$ exists:

    Add g to KB.



Definition: derivation

KB **derives/proves** f (KB $\vdash f$) iff f eventually gets added to KB.

- Given a set of inference rules (e.g., modus ponens), we can just keep on trying to apply rules. Those rules generate new formulas which get added to the knowledge base, and those formulas might then be premises of other rules, which in turn generate more formulas, etc.
- We say that the KB derives or proves a formula f if by blindly applying rules, we can eventually add f to the KB.

Inference example

Example: Modus ponens inference

Starting point:

$$\text{KB} = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}\}$$

Apply modus ponens to Rain and Rain $\rightarrow$ Wet:

$$\text{KB} = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}, \text{Wet}\}$$

Apply modus ponens to Wet and Wet $\rightarrow$ Slippery:

$$\text{KB} = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}, \text{Wet}, \text{Slippery}\}$$

Converged.

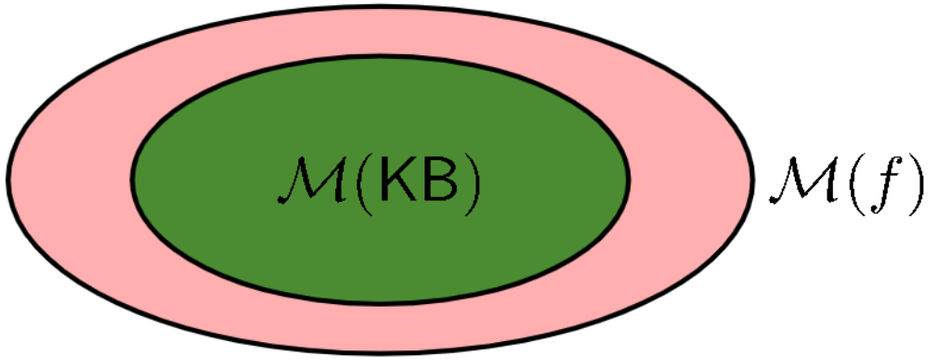
Can't derive some formulas: $\neg\text{Wet}$, $\text{Rain} \rightarrow \text{Slippery}$

- Here is an example where we've applied modus ponens twice. Note that Wet and Slippery are derived by the KB.
- But there are some formulas which cannot be derived. Some of these underivable formulas will look bad anyway ($\neg$Wet), but others will seem reasonable (Rain $\rightarrow$ Slippery).

Desiderata for inference rules

Semantics

Interpretation defines **entailed/true** formulas: $\text{KB} \models f$:



Syntax:

Inference rules **derive** formulas: $\text{KB} \vdash f$

How does $\{f : \text{KB} \models f\}$ relate to $\{f : \text{KB} \vdash f\}$?

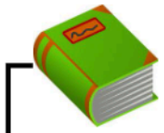
- We can apply inference rules all day long, but now we desperately need some guidance on whether a set of inference rules is doing anything remotely sensible.
- For this, we turn to semantics, which gives an objective notion of truth. Recall that the semantics provides us with $\mathcal{M}$, the set of satisfiable models for each formula f or knowledge base. This defines a set of formulas $\{f : \mathsf{KB} \models f\}$ which are defined to be true.
- On the other hand, inference rules also gives us a mechanism for generating a set of formulas, just by repeated application. This defines another set of formulas $\{f : \mathsf{KB} \vdash f\}$.

Truth

$$\{f : \mathsf{KB} \models f\}$$

- Imagine a glass that represents the set of possible formulas entailed by the KB (these are necessarily true).
- By applying inference rules, we are filling up the glass with water.

Soundness



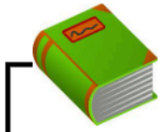
Definition: soundness

A set of inference rules Rules is sound if:

$$\{f : \text{KB} \vdash f\} \subseteq \{f : \text{KB} \models f\}$$

- We say that a set of inference rules is **sound** if using those inference rules, we never overflow the glass: the set of derived formulas is a subset of the set of true/entailed formulas.

Completeness



Definition: completeness

A set of inference rules Rules is complete if:

$$\{f : \text{KB} \vdash f\} \supseteq \{f : \text{KB} \models f\}$$

- We say that a set of inference rules is **complete** if using those inference rules, we fill up the glass to the brim (and possibly go over): the set of derived formulas is a superset of the set of true/entailed formulas.

Soundness and completeness

The truth, the whole truth, and nothing but the truth.

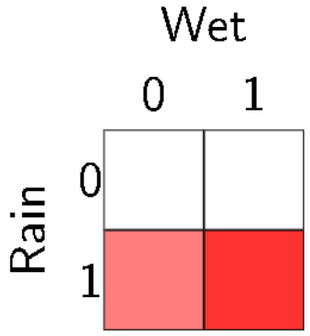
- **Soundness**: nothing but the truth
- **Completeness**: whole truth

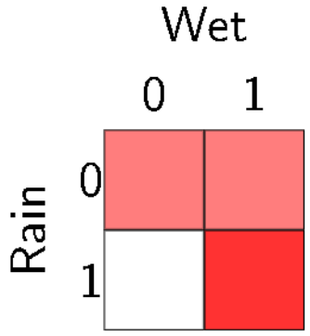
- A slogan to keep in mind is the oath given in a sworn testimony.

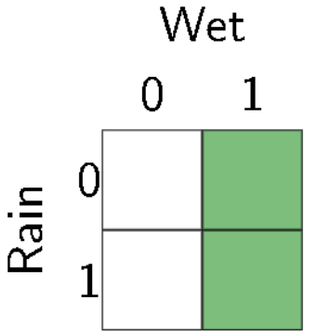
Soundness: example

Is $\dfrac{\text{Rain}, \quad \text{Rain} \to \text{Wet}}{\text{Wet}}$ (Modus ponens) sound?

$$\mathcal{M}(\text{Rain}) \quad \cap \quad \mathcal{M}(\text{Rain} \to \text{Wet}) \quad \subseteq ? \quad \mathcal{M}(\text{Wet})$$







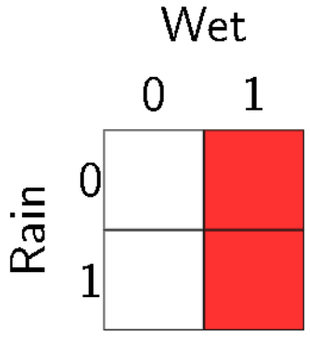
Sound!

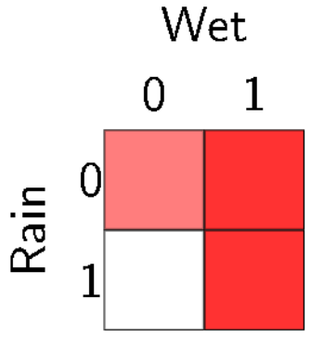
- To check the soundness of a set of rules, it suffices to focus on one rule at a time.
- Take the modus ponens rule, for instance. We can derive Wet using modus ponens. To check entailment, we map all the formulas into semantics-land (the set of satisfiable models). Because the models of Wet is a superset of the intersection of models of Rain and Rain $\rightarrow$ Wet (remember that the models in the KB are an intersection of the models of each formula), we can conclude that Wet is also entailed. If we had other formulas in the KB, that would reduce both sides of $\subseteq$ by the same amount and won't affect the fact that the relation holds. Therefore, this rule is sound.
- Note, we use Wet and Rain to make the example more colorful, but this argument works for arbitrary propositional symbols.

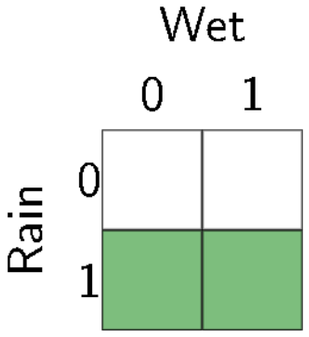
Soundness: example

Is $\dfrac{\text{Wet}, \quad \text{Rain} \rightarrow \text{Wet}}{\text{Rain}}$ sound?

$\mathcal{M}(\text{Wet}) \quad \cap \quad \mathcal{M}(\text{Rain} \rightarrow \text{Wet}) \quad \subseteq? \quad \mathcal{M}(\text{Rain})$







Unsound!

- Here is another example: given Wet and Rain $\rightarrow$ Wet, can we infer Rain? To check it, we mechanically construct the models for the premises and conclusion. Here, the intersection of the models in the premise are not a subset, then the rule is unsound.
- Indeed, backward reasoning is faulty. Note that we can actually do a bit of backward reasoning using Bayesian networks, since we don't have to commit to 0 or 1 for the truth value.

Completeness: example

Recall completeness: inference rules derive all entailed formulas (f such that KB $\models f$)

Example: Modus ponens is incomplete

Setup:

$$\text{KB} = \{\text{Rain}, \text{Rain} \vee \text{Snow} \rightarrow \text{Wet}\}$$

$$f = \text{Wet}$$

$$\text{Rules} = \{\frac{f, \quad f \rightarrow g}{g}\} \text{ (Modus ponens)}$$

Semantically: KB $\models f$ (f is entailed).

Syntactically: KB $\nvdash f$ (can't derive f).

Incomplete!

- Completeness is trickier, and here is a simple example that shows that modus ponens alone is not complete, since it can't derive Wet, when semantically, Wet is true!

Fixing completeness

Option 1: Restrict the allowed set of formulas

propositional logic

propositional logic with only Horn clauses

Option 2: Use more powerful inference rules

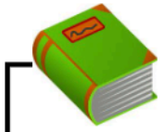
Modus ponens

resolution

- At this point, there are two ways to fix completeness. First, we can restrict the set of allowed formulas, making the water glass smaller in hopes that modus ponens will be able to fill that smaller glass.
- Second, we can use more powerful inference rules, pouring more vigorously into the same glass in hopes that this will be able to fill the glass; we'll look at one such rule, resolution, in the next lecture.

Logic: modus ponens with Horn clauses

Definite clauses



Definition: Definite clause

A **definite clause** has the following form:

$$(p_1 \wedge \cdots \wedge p_k) \rightarrow q$$

where $p_1, \ldots, p_k, q$ are propositional symbols.

Intuition: if $p_1, \ldots, p_k$ hold, then q holds.

Example: $(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$

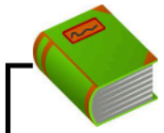
Example: Traffic

Non-example: $\neg$Traffic

Non-example: $(\text{Rain} \wedge \text{Snow}) \rightarrow (\text{Traffic} \vee \text{Peaceful})$

- First we will choose to restrict the allowed set of formulas. Towards that end, let's define a **definite clause** as a formula that says, if a conjunction of propositional symbols holds, then some other propositional symbol q holds. Note that this is a formula, not to be confused with an inference rule.

Horn clauses



Definition: Horn clause

A **Horn clause** is either:

- a definite clause $(p_1 \wedge \cdots \wedge p_k \rightarrow q)$
- a goal clause $(p_1 \wedge \cdots \wedge p_k \rightarrow \text{false})$

Example (definite): $(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$

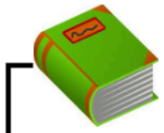
Example (goal): $\text{Traffic} \wedge \text{Accident} \rightarrow \text{false}$

equivalent: $\neg(\text{Traffic} \wedge \text{Accident})$

- A **Horn clause** is basically a definite clause, but includes another type of clause called a **goal clause**, which is the conjunction of a bunch of propositional symbols implying false. The form of the goal clause might seem a bit strange, but the way to interpret it is simply that it's the negation of the conjunction.

Modus ponens

Inference rule:



Definition: Modus ponens

$$\frac{p_1, \quad \cdots \quad , p_k, \quad (p_1 \wedge \cdots \wedge p_k) \rightarrow q}{q}$$

Example:

Example: Modus ponens

$$\frac{\text{Wet}, \quad \text{Weekday}, \quad \text{Wet} \wedge \text{Weekday} \rightarrow \text{Traffic}}{\text{Traffic}}$$

- Recall the Modus ponens rule from before. We simply have generalized it to arbitrary number of premises.

Completeness of modus ponens

Theorem: Modus ponens on Horn clauses

Modus ponens is **complete** with respect to Horn clauses:

- Suppose KB contains only Horn clauses and p is an entailed propositional symbol.
- Then applying modus ponens will derive p.

Upshot:

$$\text{KB} \models p \text{ (entailment) is the same as KB} \vdash p \text{ (derivation)!}$$

- There's a theorem that says that modus ponens is complete on Horn clauses. This means that any propositional symbol that is entailed can be derived by modus ponens too, provided that all the formulas in the KB are Horn clauses.
- We already proved that modus ponens is sound, and now we have that it is complete (for Horn clauses). The upshot of this is that entailment (a semantic notion, what we care about) and being able to derive a formula (a syntactic notion, what we do with inference) are equivalent!

Example: Modus ponens

KB

Rain

Weekday

Rain $\rightarrow$ Wet

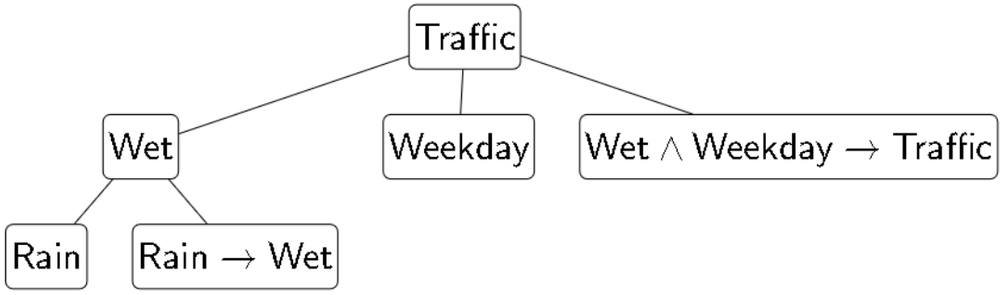
Wet $\wedge$ Weekday $\rightarrow$ Traffic

Traffic $\wedge$ Careless $\rightarrow$ Accident

Definition: Modus ponens

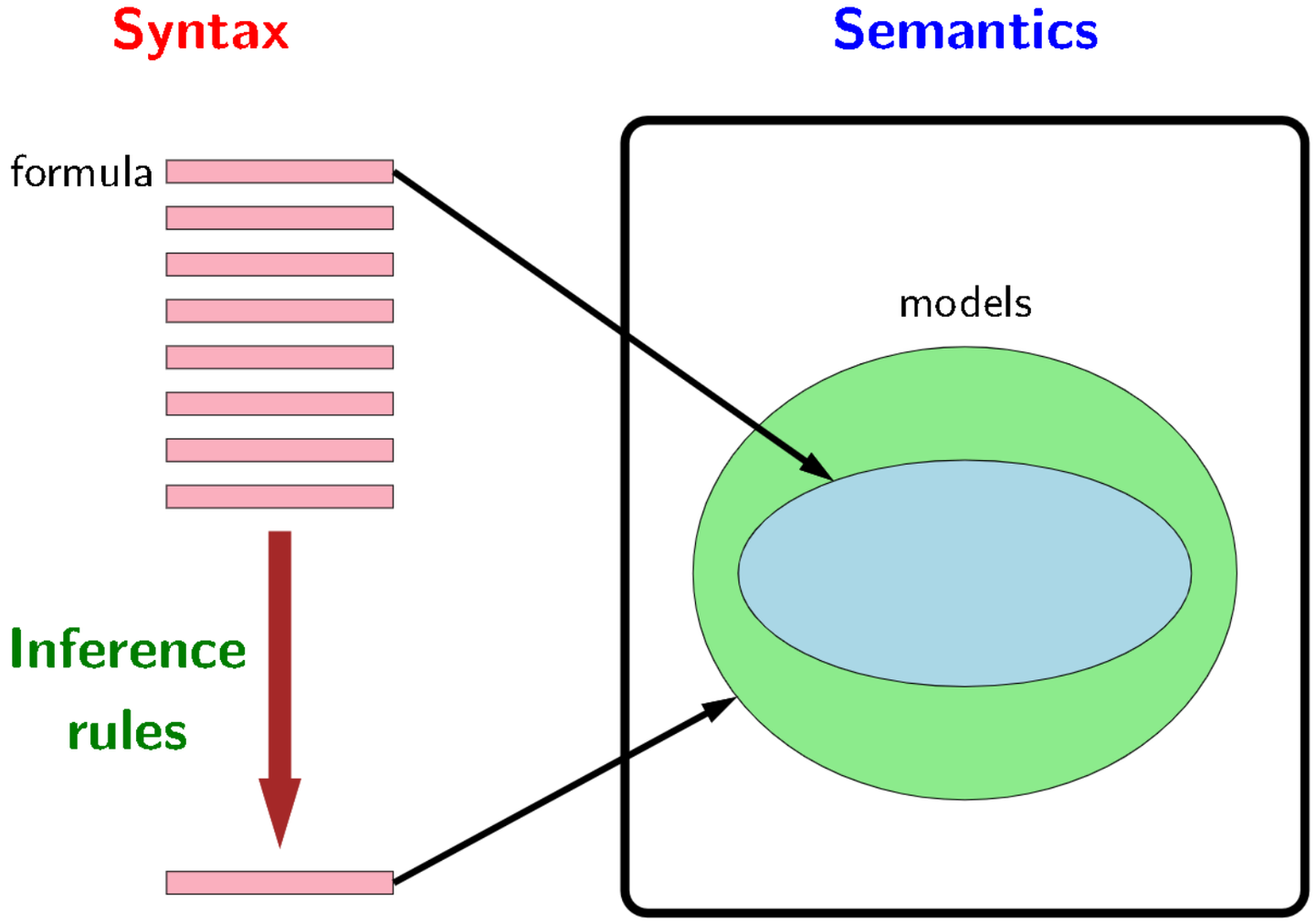
$$\frac{p_1, \quad \cdots \quad , p_k, \quad (p_1 \wedge \cdots \wedge p_k) \rightarrow q}{q}$$

Question: KB $\models$ Traffic $\Leftrightarrow$ KB $\vdash$ Traffic



- Let's see modus ponens on Horn clauses in action. Suppose we have the given KB consisting of only Horn clauses (in fact, these are all definite clauses), and we wish to ask whether the KB entails Traffic.
- We can construct a **derivation**, a tree where the root formula (e.g., Traffic) was derived using inference rules.
- The leaves are the original formulas in the KB, and each internal node corresponds to a formula which is produced by applying an inference rule (e.g., modus ponens) with the children as premises.
- If a symbol is used as the premise in two different rules, then it would have two parents, resulting in a DAG.

Summary



Logic: resolution

Review: tradeoffs

Formulas allowed	**Inference rule**	**Complete?**
Propositional logic	modus ponens	no
Propositional logic (only Horn clauses)	modus ponens	yes
Propositional logic	**resolution**	yes

- We saw that if our logical language was restricted to Horn clauses, then modus ponens alone was sufficient for completeness. For general propositional logic, modus ponens is insufficient.
- In this lecture, we'll see that a more powerful inference rule, **resolution**, is complete for all of propositional logic.

Horn clauses and disjunction

Written with implication	**Written with disjunction**
$A \rightarrow C$	$\neg A \vee C$
$A \wedge B \rightarrow C$	$\neg A \vee \neg B \vee C$

- **Literal**: either p or $\neg p$, where p is a propositional symbol
- **Clause**: disjunction of literals
- **Horn clauses**: at most one positive literal

Modus ponens (rewritten):

$$\frac{A, \quad \neg A \vee C}{C}$$

- Intuition: cancel out A and $\neg A$

- Modus ponens can only deal with Horn clauses, so let's see why Horn clauses are limiting. We can equivalently write implication using negation and disjunction. Then it's clear that Horn clauses are just disjunctions of literals where there is at most one positive literal and zero or more negative literals. The negative literals correspond to the propositional symbols on the left side of the implication, and the positive literal corresponds to the propositional symbol on the right side of the implication.
- If we rewrite modus ponens, we can see a "canceling out" intuition emerging. To make the intuition a bit more explicit, remember that, to respect soundness, we require $\{A, \neg A \vee C\} \models C$; this is equivalent to: if $A \wedge (\neg A \vee C)$ is true, then C is also true. This is clearly the case.
- But modus ponens cannot operate on general clauses.

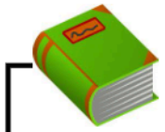
Resolution [Robinson, 1965]

General clauses have any number of literals:

$$\neg A \vee B \vee \neg C \vee D \vee \neg E \vee F$$

Example: resolution inference rule

$$\frac{\text{Rain} \vee \text{Snow}, \quad \neg\text{Snow} \vee \text{Traffic}}{\text{Rain} \vee \text{Traffic}}$$



Definition: resolution inference rule

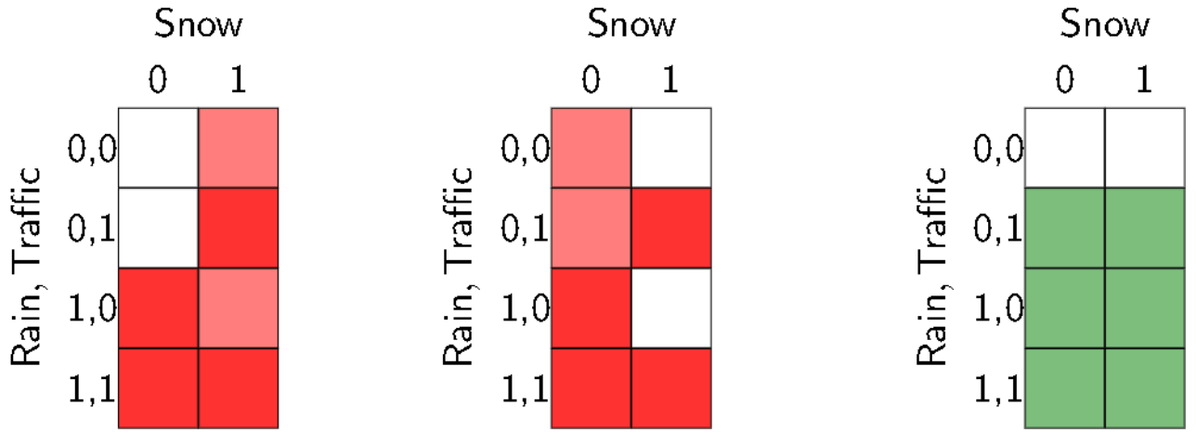
$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg p \vee g_1 \vee \cdots \vee g_m}{f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m}$$

- Let's try to generalize modus ponens by allowing it to work on general clauses. This generalized inference rule is called **resolution**, which was invented in 1965 by John Alan Robinson.
- The idea behind resolution is that it takes two general clauses, where one of them has some propositional symbol p and the other clause has its negation $\neg p$, and simply takes the disjunction of the two clauses with p and $\neg p$ removed. Here, $f_1, \ldots, f_n, g_1, \ldots, g_m$ are arbitrary literals.

Soundness of resolution

$$\frac{\text{Rain} \vee \text{Snow}, \quad \neg\text{Snow} \vee \text{Traffic}}{\text{Rain} \vee \text{Traffic}} \text{ (resolution rule)}$$

$\mathcal{M}(\text{Rain} \vee \text{Snow}) \cap \mathcal{M}(\neg\text{Snow} \vee \text{Traffic}) \subseteq ? \mathcal{M}(\text{Rain} \vee \text{Traffic})$

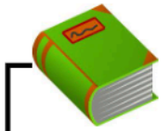


Sound!

- Why is resolution logically sound? We can verify the soundness of resolution by checking its semantic interpretation. Indeed, the intersection of the models of f and g is a subset of models of $f \vee g$.

Conjunctive normal form

So far: resolution only works on clauses...but that's enough!



Definition: conjunctive normal form (CNF)

A **CNF formula** is a conjunction of clauses.

Example: $(A \vee B \vee \neg C) \wedge (\neg B \vee D)$

Equivalent: knowledge base where each formula is a clause

Proposition: conversion to CNF

Every formula f in propositional logic can be converted into an equivalent CNF formula f':

$$\mathcal{M}(f) = \mathcal{M}(f')$$

- But so far, we've only considered clauses, which are disjunctions of literals. Surely this can't be all of propositional logic... But it turns out it actually is in the following sense.
- A conjunction of clauses is called a CNF formula, and every formula in propositional logic can be converted into an equivalent CNF. Given a CNF formula, we can toss each of its clauses into the knowledge base.
- But why can every formula be put in CNF?

Conversion to CNF: example

Initial formula:

$$(\text{Summer} \rightarrow \text{Snow}) \rightarrow \text{Bizzare}$$

Remove implication ($\rightarrow$):

$$\neg(\neg\text{Summer} \vee \text{Snow}) \vee \text{Bizzare}$$

Push negation ($\neg$) inwards (de Morgan):

$$(\neg\neg\text{Summer} \wedge \neg\text{Snow}) \vee \text{Bizzare}$$

Remove double negation:

$$(\text{Summer} \wedge \neg\text{Snow}) \vee \text{Bizzare}$$

Distribute $\vee$ over $\wedge$:

$$(\text{Summer} \vee \text{Bizzare}) \wedge (\neg\text{Snow} \vee \text{Bizzare})$$

- The answer is by construction. There is a six-step procedure that takes any propositional formula and turns it into CNF. Here is an example of how it works (only four of the six steps apply here).

Conversion to CNF: general

Conversion rules:

- Eliminate $\leftrightarrow$: $\frac{f \leftrightarrow g}{(f \rightarrow g) \wedge (g \rightarrow f)}$
- Eliminate $\rightarrow$: $\frac{f \rightarrow g}{\neg f \vee g}$
- Move $\neg$ inwards: $\frac{\neg(f \wedge g)}{\neg f \vee \neg g}$
- Move $\neg$ inwards: $\frac{\neg(f \vee g)}{\neg f \wedge \neg g}$
- Eliminate double negation: $\frac{\neg\neg f}{f}$
- Distribute $\vee$ over $\wedge$: $\frac{f \vee (g \wedge h)}{(f \vee g) \wedge (f \vee h)}$

- Here are the general rules that convert any formula to CNF. First, we try to reduce everything to negation, conjunction, and disjunction.
- Next, we try to push negation inwards so that they sit on the propositional symbols (forming literals). Note that when negation gets pushed inside, it flips conjunction to disjunction, and vice-versa.
- Finally, we distribute so that the conjunctions are on the outside, and the disjunctions are on the inside.
- Note that each of these operations preserves the semantics of the logical form (remember there are many formula that map to the same set of models). This is in contrast with most inference rules, where the conclusion is more general than the conjunction of the premises.
- Also, when we apply a CNF rewrite rule, we replace the old formula with the new one, so there is no blow-up in the number of formulas. This is in contrast to applying general inference rules. An analogy: conversion to CNF does simplification in the context of full inference, just like AC-3 does simplification in the context of backtracking search.

Resolution algorithm

Recall: relationship between entailment and contradiction (basically "proof by contradiction")

$$\text{KB} \models f \quad \Longleftrightarrow \quad \text{KB} \cup \{\neg f\} \text{ is unsatisfiable}$$

Algorithm: resolution-based inference

- Add $\neg f$ into KB.
- Convert all formulas into **CNF**.
- Repeatedly apply **resolution** rule.
- Return entailment iff derive false.

- After we have converted all the formulas to CNF, we can repeatedly apply the resolution rule. But what is the final target?
- Recall that both testing for entailment and contradiction boil down to checking satisfiability. Resolution can be used to do this very thing. If we ever apply a resolution rule (e.g., to premises A and $\neg A$) and we derive false (which represents a contradiction), then the set of formulas in the knowledge base is unsatisfiable.
- If we are unable to derive false, that means the knowledge base is satisfiable because resolution is complete. However, unlike in model checking, we don't actually produce a concrete model that satisfies the KB.

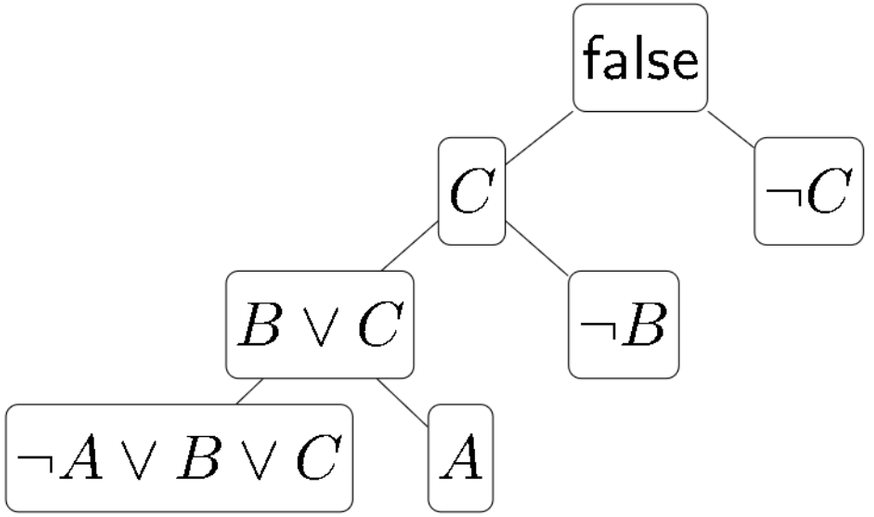
Resolution: example

$$\text{KB}' = \{A \to (B \vee C), A, \neg B, \neg C\}$$

Convert to CNF:

$$\text{KB}' = \{\neg A \vee B \vee C, A, \neg B, \neg C\}$$

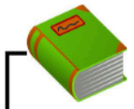
Repeatedly apply **resolution** rule:



Conclusion: KB **entails** f

- Here's an example of taking a knowledge base, converting it into CNF, and applying resolution. In this case, we derive false, which means that the original knowledge base was unsatisfiable.

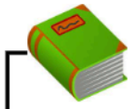
Time complexity



Definition: modus ponens inference rule

$$\frac{p_1, \quad \cdots \quad , p_k, \quad (p_1 \wedge \cdots \wedge p_k) \rightarrow q}{q}$$

- Each rule application adds clause with **one** propositional symbol $\Rightarrow$ linear time



Definition: resolution inference rule

$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg p \vee g_1 \vee \cdots \vee g_m}{f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m}$$

- Each rule application adds clause with **many** propositional symbols $\Rightarrow$ exponential time

- There we have it — a sound and complete inference procedure for all of propositional logic (although we didn't prove completeness). But what do we have to pay computationally for this increase?
- If we only have to apply modus ponens, each propositional symbol can only get added once, so with the appropriate algorithm (forward chaining), we can apply all necessary modus ponens rules in linear time.
- But with resolution, we can end up adding clauses with many propositional symbols, and possibly any subset of them! Therefore, this can take exponential time.

Summary

Horn clauses	**any clauses**
modus ponens	resolution
linear time	exponential time
less expressive	more expressive

- To summarize, we can either content ourselves with the limited expressivity of Horn clauses and obtain an efficient inference procedure (via modus ponens).
- If we wanted the expressivity of full propositional logic, then we need to use resolution and thus pay more.

Logic: first-order logic

Limitations of propositional logic

Alice and Bob both know arithmetic.

$\text{AliceKnowsArithmetic} \wedge \text{BobKnowsArithmetic}$

All students know arithmetic.

$\text{AliceIsStudent} \rightarrow \text{AliceKnowsArithmetic}$

$\text{BobIsStudent} \rightarrow \text{BobKnowsArithmetic}$

$\ldots$

Every even integer greater than 2 is the sum of two primes.

???

- If the goal of logic is to be able to express facts in the world in a compact way, let us ask ourselves if propositional logic is enough.
- Some facts can be expressed in propositional logic, but it is very clunky, having to instantiate many different formulas. Others simply can't be expressed at all, because we would need to use an infinite number of formulas.

Limitations of propositional logic

All students know arithmetic.

AliceIsStudent $\rightarrow$ AliceKnowsArithmetic

BobIsStudent $\rightarrow$ BobKnowsArithmetic

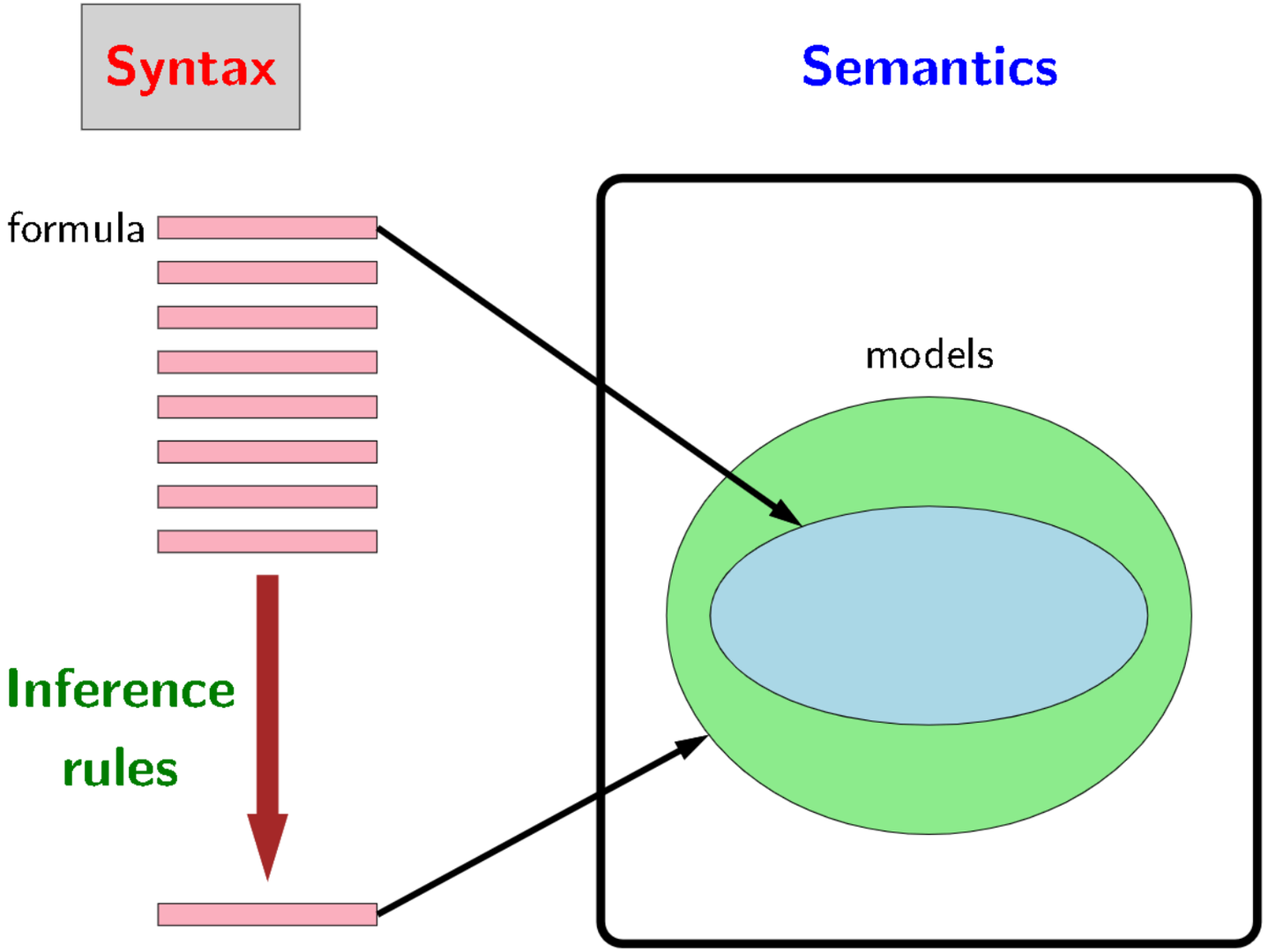
...

Propositional logic is very clunky. What's missing?

- Objects and predicates: propositions (e.g., AliceKnowsArithmetic) have more internal structure (alice, Knows, arithmetic)
- Quantifiers and variables: *all* is a quantifier which applies to each person, don't want to enumerate them all...

- What's missing? The key conceptual observation is that the world is not just a bunch of atomic facts, but that each fact is actually made out of **objects** and **predicates** on those objects.
- Once facts are decomposed in this way, we can use **quantifiers** and **variables** to implicitly define a huge (and possibly infinite) number of facts with one compact formula. Again, where logic excels is the ability to represent complex things via simple means.

First-order logic



- We will now introduce **first-order logic**, which will address the representational limitations of propositional logic.
- Remember to define a logic, we need to talk about its syntax, its semantics (interpretation function), and finally inference rules that we can use to operate on the syntax.

First-order logic: examples

Alice and Bob both know arithmetic.

$$\mathsf{Knows}(\mathsf{alice}, \mathsf{arithmetic}) \wedge \mathsf{Knows}(\mathsf{bob}, \mathsf{arithmetic})$$

All students know arithmetic.

$$\forall x \, \mathsf{Student}(x) \rightarrow \mathsf{Knows}(x, \mathsf{arithmetic})$$

- Before formally defining things, let's look at two examples. First-order logic is basically propositional logic with a few more symbols.

Syntax of first-order logic

Terms (refer to objects):

- Constant symbol (e.g., arithmetic)
- Variable (e.g., x)
- Function of terms (e.g., $\text{Sum}(3, x)$)

Formulas (refer to truth values):

- Atomic formulas (atoms): predicate applied to terms (e.g., $\text{Knows}(x, \text{arithmetic})$)
- Connectives applied to formulas (e.g., $\text{Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$)
- Quantifiers applied to formulas (e.g., $\forall x\, \text{Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$)

- In propositional logic, everything was a formula (or a connective). In first-order logic, there are two types of beasts: terms and formulas. There are three types of terms: constant symbols (which refer to specific objects), variables (which refer to some unspecified object to be determined by quantifiers), and functions (which is a function applied to a set of arguments which are themselves terms).
- Given the terms, we can form atomic formulas, which are the analogue of propositional symbols, but with internal structure (e.g., terms).
- From this point, we can apply the same connectives on these atomic formulas, as we applied to propositional symbols in propositional logic. At this level, first-order logic looks very much like propositional logic.
- Finally, to make use of the fact that atomic formulas have internal structure, we have **quantifiers**, which are really the whole point of first-order logic!

Quantifiers

Universal quantification ($\forall$):

Think conjunction: $\forall x\, P(x)$ is like $P(A) \wedge P(B) \wedge \cdots$

Existential quantification ($\exists$):

Think disjunction: $\exists x\, P(x)$ is like $P(A) \vee P(B) \vee \cdots$

Some properties:

- $\neg \forall x\, P(x)$ equivalent to $\exists x\, \neg P(x)$
- $\forall x\, \exists y\, \mathsf{Knows}(x, y)$ different from $\exists y\, \forall x\, \mathsf{Knows}(x, y)$

- There are two types of quantifiers: universal and existential. These are basically glorified ways of doing conjunction and disjunction, respectively.
- For crude intuition, we can think of conjunction and disjunction as very nice syntactic sugar, which can be rolled out into something that looks more like propositional logic. But quantifiers aren't just sugar, and it is important that they be compact, for sometimes the variable being quantified over can take on an infinite number of objects.
- That being said, the conjunction and disjunction intuition suffices for day-to-day guidance. For example, it should be intuitive that pushing the negation inside a universal quantifier (conjunction) turns it into a existential (disjunction), which was the case for propositional logic (by de Morgan's laws). Also, one cannot interchange universal and existential quantifiers any more than one can swap conjunction and disjunction in propositional logic.

Natural language quantifiers

Universal quantification ($\forall$):

Every student knows arithmetic.

$\forall x\, \mathsf{Student}(x) \rightarrow \mathsf{Knows}(x, \mathsf{arithmetic})$

Existential quantification ($\exists$):

Some student knows arithmetic.

$\exists x\, \mathsf{Student}(x) \wedge \mathsf{Knows}(x, \mathsf{arithmetic})$

Note the different connectives!

- Universal and existential quantifiers naturally correspond to the words *every* and *some*, respectively. But when converting English to formal logic, one must exercise caution.
- *Every* can be thought of as taking two arguments P and Q (e.g., *student* and *knows arithmetic*). The connective between P and Q is an implication (not conjunction, which is a common mistake). This makes sense because when we talk about every P, we are only restricting our attention to objects x for which $P(x)$ is true. Implication does exactly that.
- On the other hand, the connective for existential quantification is conjunction, because we're asking for an object x such that $P(x)$ and $Q(x)$ both hold.

Some examples of first-order logic

There is some course that every student has taken.

$$\exists y\, \mathsf{Course}(y) \wedge [\forall x\, \mathsf{Student}(x) \rightarrow \mathsf{Takes}(x, y)]$$

Every even integer greater than 2 is the sum of two primes.

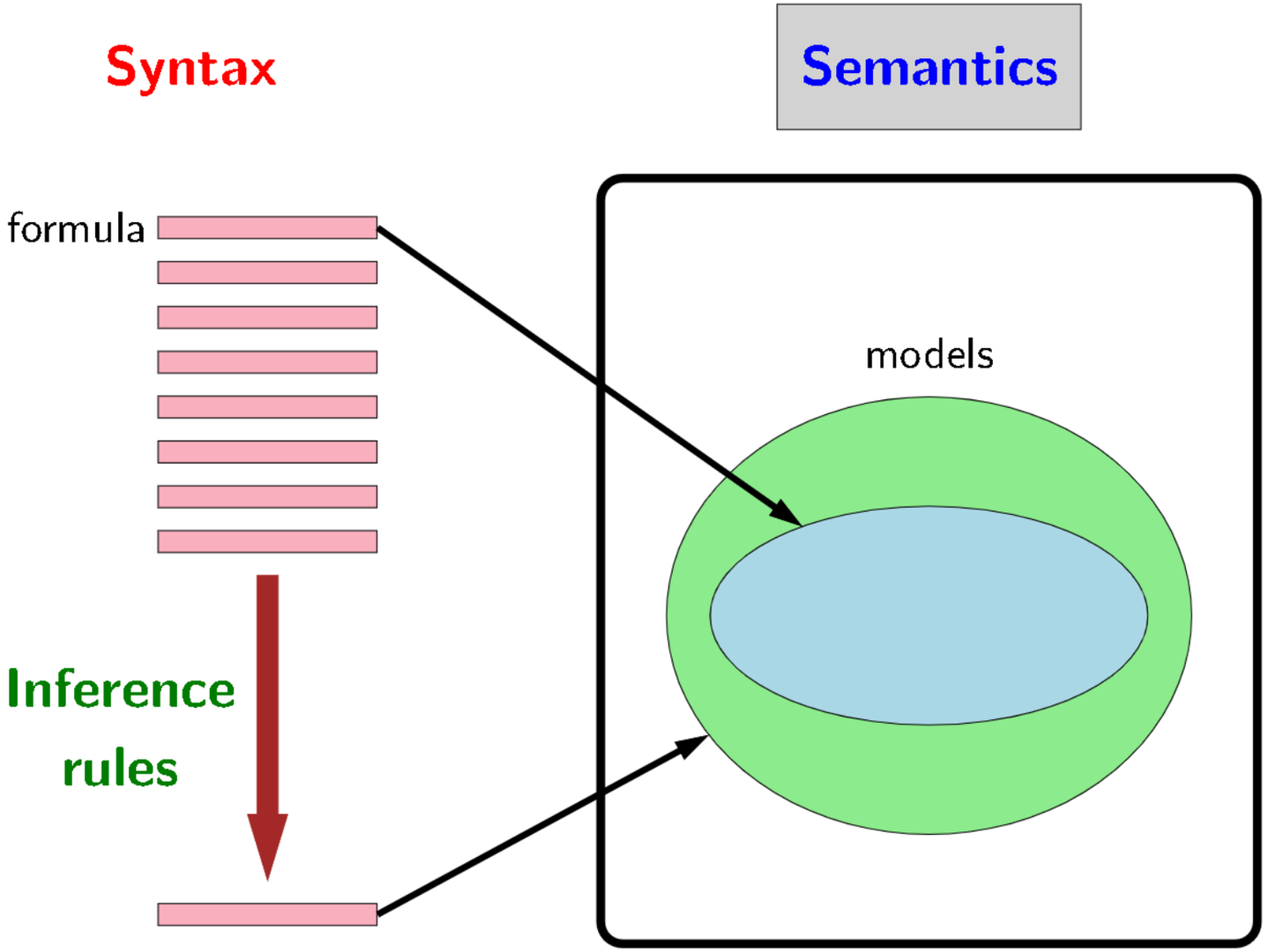
$$\forall x\, \mathsf{EvenInt}(x) \wedge \mathsf{Greater}(x, 2) \rightarrow \exists y\, \exists z\, \mathsf{Equals}(x, \mathsf{Sum}(y, z)) \wedge \mathsf{Prime}(y) \wedge \mathsf{Prime}(z)$$

If a student takes a course and the course covers a concept, then the student knows that concept.

$$\forall x\, \forall y\, \forall z\, (\mathsf{Student}(x) \wedge \mathsf{Takes}(x, y) \wedge \mathsf{Course}(y) \wedge \mathsf{Covers}(y, z)) \rightarrow \mathsf{Knows}(x, z)$$

- Let's do some more examples of converting natural language to first-order logic. Remember the connectives associated with existential and universal quantification!
- Note that some English words such as *a* can trigger both universal or existential quantification, depending on context. In *A student took CS221*, we have existential quantification, but in *if a student takes CS221, ...*, we have universal quantification.
- Formal logic clears up the ambiguities associated with natural language.

First-order logic



- So far, we've only presented the syntax of first-order logic, although we've actually given quite a bit of intuition about what the formulas mean. After all, it's hard to talk about the syntax without at least a hint of semantics for motivation.
- Now let's talk about the formal semantics of first-order logic.

Models in first-order logic

Recall a model represents a possible situation in the world.

Propositional logic: Model w maps propositional symbols to truth values.

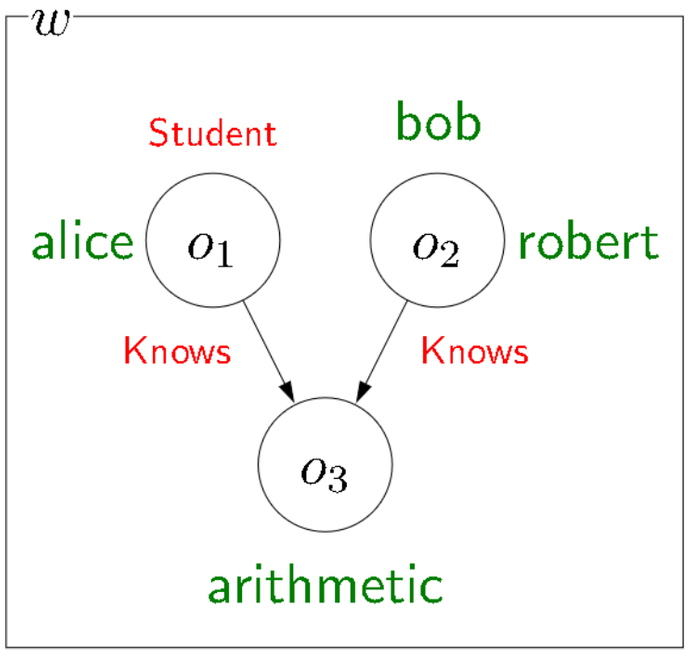
$$w = \{\text{AliceKnowsArithmetic} : 1, \text{BobKnowsArithmetic} : 0\}$$

First-order logic: ?

- Recall that a model in propositional logic was just an assignment of truth values to propositional symbols.
- A natural candidate for a model in first-order logic would then be an assignment of truth values to grounded atomic formula (those formulas whose terms are constants as opposed to variables). This is almost right, but doesn't talk about the relationship between constant symbols.

Graph representation of a model

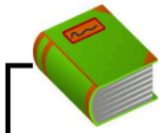
If only have unary and binary predicates, a model w can be represented as a directed graph:



- Nodes are objects, labeled with constant symbols
- Directed edges are binary predicates, labeled with predicate symbols; unary predicates are additional node labels

- A better way to think about a first-order model is that there are a number of objects in the world ($o_1, o_2, \ldots$); think of these as nodes in a graph. Then we have predicates between these objects. Predicates that take two arguments can be visualized as labeled edges between objects. Predicates that take one argument can be visualized as node labels (but these are not so important).
- So far, the objects are unnamed. We can access individual objects directly using constant symbols, which are labels on the nodes.

Models in first-order logic



Definition: model in first-order logic

A model w in first-order logic maps:

- constant symbols to objects

$w(\text{alice}) = o_1, w(\text{bob}) = o_2, w(\text{arithmetic}) = o_3$

- predicate symbols to tuples of objects

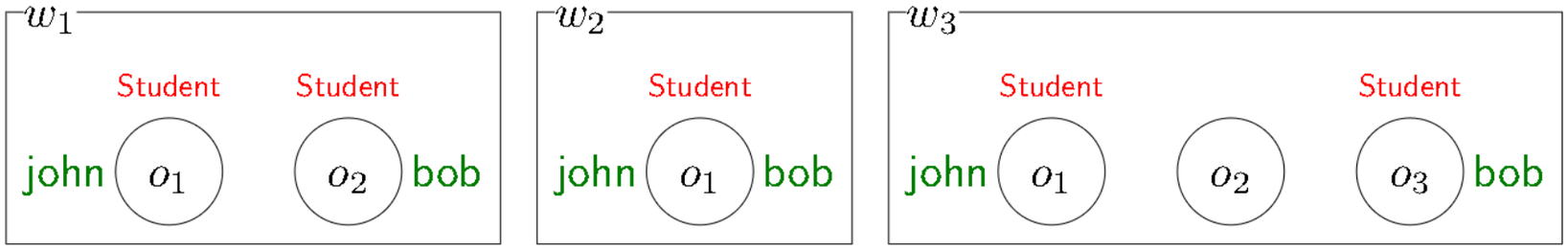
$w(\text{Knows}) = \{(o_1, o_3), (o_2, o_3), \ldots\}$

- Formally, a first-order model w maps constant symbols to objects and predicate symbols to tuples of objects (2 for binary predicates).

A restriction on models

John and Bob are students.

$$\text{Student}(\text{john}) \wedge \text{Student}(\text{bob})$$



- Unique names assumption: Each object has **at most one** constant symbol. This rules out w_2.
- Domain closure: Each object has **at least one** constant symbol. This rules out w_3.

Point:

constant symbol ⟷ object

- Note that by default, two constant symbols can refer to the same object, and there can be objects which no constant symbols refer to. This can make life somewhat confusing. Fortunately, there are two assumptions that people sometimes make to simplify things.
- The unique names assumption says that there's at most one way to refer to an object via a constant symbol. Domain closure says there's at least one. Together, they imply that there is a one-to-one relationship between constant symbols in syntax-land and objects in semantics-land.

Propositionalization

If one-to-one mapping between constant symbols and objects (**unique names** and **domain closure**),

first-order logic is syntactic sugar for propositional logic:

Knowledge base in first-order logic

$$\text{Student}(\text{alice}) \wedge \text{Student}(\text{bob})$$
$$\forall x\, \text{Student}(x) \rightarrow \text{Person}(x)$$
$$\exists x\, \text{Student}(x) \wedge \text{Creative}(x)$$

Knowledge base in propositional logic

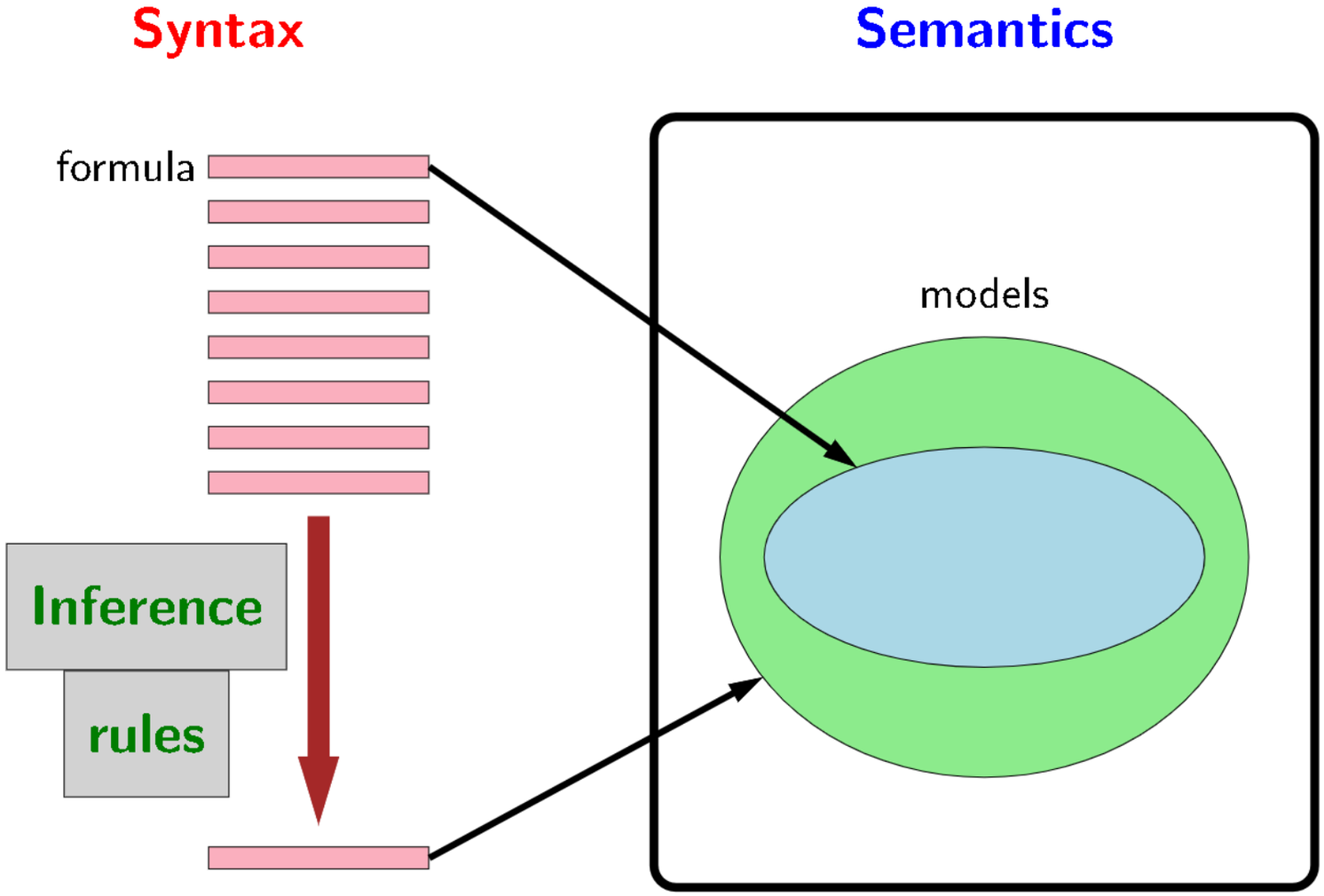
$$\text{Student}\text{alice} \wedge \text{Student}\text{bob}$$
$$(\text{Student}\text{alice} \rightarrow \text{Person}\text{alice}) \wedge (\text{Student}\text{bob} \rightarrow \text{Person}\text{bob})$$
$$(\text{Student}\text{alice} \wedge \text{Creative}\text{alice}) \vee (\text{Student}\text{bob} \wedge \text{Creative}\text{bob})$$

Point: use any inference algorithm for propositional logic!

- If a one-to-one mapping really exists, then we can **propositionalize** all our formulas, which basically unrolls all the quantifiers into explicit conjunctions and disjunctions.
- The upshot of this conversion, is that we're back to propositional logic, and we know how to do inference in propositional logic (either using model checking or by applying inference rules). Of course, propositionalization could be quite expensive and not the most efficient thing to do.

Logic: first-order modus ponens

First-order logic

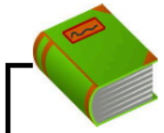


- Now we look at inference rules which can make first-order inference much more efficient. The key is to do everything implicitly and avoid propositionalization; again the whole spirit of logic is to do things compactly and implicitly.

Definite clauses

$$\forall x \, \forall y \, \forall z \, (\text{Takes}(x, y) \land \text{Covers}(y, z)) \rightarrow \text{Knows}(x, z)$$

Note: if propositionalize, get one formula for each value to (x, y, z), e.g., $(\text{alice}, \text{cs221}, \text{mdp})$



Definition: definite clause (first-order logic)

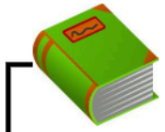
A definite clause has the following form:

$$\forall x_1 \cdots \forall x_n \, (a_1 \land \cdots \land a_k) \rightarrow b$$

for variables $x_1, \ldots, x_n$ and atomic formulas $a_1, \ldots, a_k, b$ (which contain those variables).

- Like our development of inference in propositional logic, we will first talk about first-order logic restricted to Horn clauses, in which case a first-order version of modus ponens will be sound and complete. After that, we'll see how resolution allows to handle all of first-order logic.
- We start by generalizing definite clauses from propositional logic to first-order logic. The only difference is that we now have universal quantifiers sitting at the beginning of the definite clause. This makes sense since universal quantification is associated with implication, and one can check that if one propositionalizes a first-order definite clause, one obtains a set (conjunction) of multiple propositional definite clauses.

Modus ponens (first attempt)



Definition: modus ponens (first-order logic)

$$\frac{a_1, \ldots, a_k \quad \forall x_1 \cdots \forall x_n (a_1 \wedge \cdots \wedge a_k) \rightarrow b}{b}$$

Setup:

Given $P(\text{alice})$ and $\forall x\, P(x) \rightarrow Q(x)$.

Problem:

Can't infer $Q(\text{alice})$ because $P(x)$ and $P(\text{alice})$ don't match!

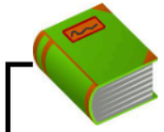
Solution: substitution and unification

- If we try to write down the modus ponens rule, we would fail.
- As a simple example, suppose we are given $P(\text{alice})$ and $\forall x\, P(x) \to Q(x)$. We would naturally want to derive $Q(\text{alice})$. But notice that we can't apply modus ponens because $P(\text{alice})$ and $P(x)$ don't match!
- Recall that we're in syntax-land, which means that these formulas are just symbols. Inference rules don't have access to the semantics of the constants and variables — it is just a pattern matcher. So we have to be very methodical.
- To develop a mechanism to match variables and constants, we will introduce two concepts, substitution and unification for this purpose.

Substitution

$\text{Subst}[\{x/\text{alice}\}, P(x)] = P(\text{alice})$

$\text{Subst}[\{x/\text{alice}, y/z\}, P(x) \wedge K(x, y)] = P(\text{alice}) \wedge K(\text{alice}, z)$



Definition: Substitution

A substitution θ is a mapping from variables to terms.

$\text{Subst}[\theta, f]$ returns the result of performing substitution θ on f.

- The first step is substitution, which applies a search-and-replace operation on a formula or term.
- We won't define Subst$[\theta, f]$ formally, but from the examples, it should be clear what Subst does.
- Technical note: if θ contains variable substitutions x/alice we only apply the substitution to the free variables in f, which are the variables not bound by quantification (e.g., x in $\exists y, P(x, y)$). Later, we'll see how CNF formulas allow us to remove all the quantifiers.

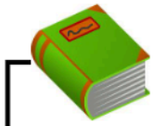
Unification

$\text{Unify}[\text{Knows}(\text{alice}, \text{arithmetic}), \text{Knows}(x, \text{arithmetic})] = \{x/\text{alice}\}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(x, z)] = \{x/\text{alice}, y/z\}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(\text{bob}, z)] = \text{fail}$

$\text{Unify}[\text{Knows}(\text{alice}, y), \text{Knows}(x, F(x))] = \{x/\text{alice}, y/F(\text{alice})\}$



Definition: Unification

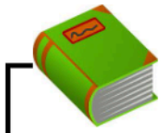
Unification takes two formulas f and g and returns a substitution θ which is the most general unifier:

$\text{Unify}[f, g] = \theta$ such that $\text{Subst}[\theta, f] = \text{Subst}[\theta, g]$

or "fail" if no such θ exists.

- Substitution can be used to make two formulas identical, and unification is the way to find the least committal substitution we can find to achieve this.
- Unification, like substitution, can be implemented recursively. The implementation details are not the most exciting, but it's useful to get some intuition from the examples.

Modus ponens



Definition: modus ponens (first-order logic)

$$\frac{a'_1, \ldots, a'_k \quad \forall x_1 \cdots \forall x_n (a_1 \wedge \cdots \wedge a_k) \rightarrow b}{b'}$$

Get most general unifier θ on premises:

- $\theta = \text{Unify}[a'_1 \wedge \cdots \wedge a'_k, a_1 \wedge \cdots \wedge a_k]$

Apply θ to conclusion:

- $\text{Subst}[\theta, b] = b'$

- Having defined substitution and unification, we are in position to finally define the modus ponens rule for first-order logic. Instead of performing a exact match, we instead perform a unification, which generates a substitution θ. Using θ, we can generate the conclusion b' on the fly.
- Note the significance here: the rule $a_1 \wedge \cdots \wedge a_k \rightarrow b$ can be used in a myriad ways, but Unify identifies the appropriate substitution, so that it can be applied to the conclusion.

Modus ponens example

Example: modus ponens in first-order logic

Premises:

$\text{Takes}(\text{alice}, \text{cs221})$

$\text{Covers}(\text{cs221}, \text{mdp})$

$\forall x \, \forall y \, \forall z \, \text{Takes}(x, y) \land \text{Covers}(y, z) \rightarrow \text{Knows}(x, z)$

Conclusion:

$\theta = \{x/\text{alice}, y/\text{cs221}, z/\text{mdp}\}$

Derive $\text{Knows}(\text{alice}, \text{mdp})$

- Here's a simple example of modus ponens in action. We bind x, y, z to appropriate objects (constant symbols), which is used to generate the conclusion Knows(alice, mdp).

Complexity

$$\forall x \, \forall y \, \forall z \, P(x, y, z)$$

- Each application of Modus ponens produces an atomic formula.
- If no function symbols, number of atomic formulas is at most

$$(\text{num-constant-symbols})^{(\text{maximum-predicate-arity})}$$

- If there are function symbols (e.g., F), then infinite...

$$Q(a) \quad Q(F(a)) \quad Q(F(F(a))) \quad Q(F(F(F(a)))) \quad \cdots$$

- In propositional logic, modus ponens was considered efficient, since in the worst case, we generate each propositional symbol.
- In first-order logic, though, we typically have many more atomic formulas in place of propositional symbols, which leads to a potentially exponentially number of atomic formulas, or worse, with function symbols, there might be an infinite set of atomic formulas.

Complexity

Theorem: completeness

Modus ponens is complete for first-order logic with only Horn clauses.

Theorem: semi-decidability

First-order logic (even restricted to only Horn clauses) is **semi-decidable**.

- If KB $\models f$, forward inference on complete inference rules will prove f in finite time.
- If KB $\not\models f$, no algorithm can show this in finite time.

- We can show that modus ponens is complete with respect to Horn clauses, which means that every true formula has an actual finite derivation.
- However, this doesn't mean that we can just run modus ponens and be done with it, for first-order logic even restricted to Horn clauses is semi-decidable, which means that if a formula is entailed, then we will be able to derive it, but if it is not entailed, then we don't even know when to stop the algorithm — quite troubling!
- With propositional logic, there were a finite number of propositional symbols, but now the number of atomic formulas can be infinite (the culprit is function symbols).
- Though we have hit a theoretical barrier, life goes on and we can still run modus ponens inference to get a one-sided answer. Next, we will move to working with full first-order logic.

Logic: resolution

Resolution

Recall: First-order logic includes non-Horn clauses

$$\forall x\, \mathsf{Student}(x) \rightarrow \exists y\, \mathsf{Knows}(x, y)$$

High-level strategy (same as in propositional logic):

- Convert all formulas to CNF
- Repeatedly apply resolution rule

- To go beyond Horn clauses, we will develop a single resolution rule which is sound and complete.
- The high-level strategy is the same as propositional logic: convert to CNF and apply resolution.

Conversion to CNF

Input:

$$\forall x\,(\forall y\,\mathsf{Animal}(y) \rightarrow \mathsf{Loves}(x,y)) \rightarrow \exists y\,\mathsf{Loves}(y,x)$$

Output:

$$(\mathsf{Animal}(Y(x)) \vee \mathsf{Loves}(Z(x),x)) \wedge (\neg\mathsf{Loves}(x,Y(x)) \vee \mathsf{Loves}(Z(x),x))$$

New to first-order logic:

- All variables (e.g., x) have universal quantifiers by default
- Introduce **Skolem functions** (e.g., $Y(x)$) to represent existential quantified variables

- Consider the logical formula corresponding to *Everyone who loves all animals is loved by someone*. The slide shows the desired output, which looks like a CNF formula in propositional logic, but there are two differences: there are variables (e.g., x) and functions of variables (e.g., $Y(x)$). The variables are assumed to be universally quantified over, and the functions are called **Skolem functions** and stand for a property of the variable.

Conversion to CNF (part 1)

Anyone who likes all animals is liked by someone.

Input:

$$\forall x\,(\forall y\,\mathsf{Animal}(y) \rightarrow \mathsf{Loves}(x,y)) \rightarrow \exists y\,\mathsf{Loves}(y,x)$$

Eliminate implications (old):

$$\forall x\,\neg(\forall y\,\neg\mathsf{Animal}(y) \vee \mathsf{Loves}(x,y)) \vee \exists y\,\mathsf{Loves}(y,x)$$

Push $\neg$ inwards, eliminate double negation (old):

$$\forall x\,(\exists y\,\mathsf{Animal}(y) \wedge \neg\mathsf{Loves}(x,y)) \vee \exists y\,\mathsf{Loves}(y,x)$$

Standardize variables (**new**):

$$\forall x\,(\exists y\,\mathsf{Animal}(y) \wedge \neg\mathsf{Loves}(x,y)) \vee \exists z\,\mathsf{Loves}(z,x)$$

- We start by eliminating implications, pushing negation inside, and eliminating double negation, which is all old.
- The first thing new to first-order logic is standardization of variables. Note that in $\exists x\, P(x) \wedge \exists x\, Q(x)$, there are two instances of x whose scopes don't overlap. To make this clearer, we will convert this into $\exists x\, P(x) \wedge \exists y\, Q(y)$. This sets the stage for when we will drop the quantifiers on the variables.

Conversion to CNF (part 2)

$\forall x\,(\exists y\,\mathsf{Animal}(y) \wedge \neg\mathsf{Loves}(x,y)) \vee \exists z\,\mathsf{Loves}(z,x)$

Replace existentially quantified variables with Skolem functions (**new**):

$\forall x\,[\mathsf{Animal}(Y(x)) \wedge \neg\mathsf{Loves}(x,Y(x))] \vee \mathsf{Loves}(Z(x),x)$

Distribute $\vee$ over $\wedge$ (old):

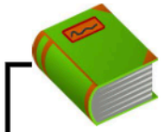
$\forall x\,[\mathsf{Animal}(Y(x)) \vee \mathsf{Loves}(Z(x),x)] \wedge [\neg\mathsf{Loves}(x,Y(x)) \vee \mathsf{Loves}(Z(x),x)]$

Remove universal quantifiers (**new**):

$[\mathsf{Animal}(Y(x)) \vee \mathsf{Loves}(Z(x),x)] \wedge [\neg\mathsf{Loves}(x,Y(x)) \vee \mathsf{Loves}(Z(x),x)]$

- The next step is to remove existential variables by replacing them with Skolem functions. This is perhaps the most non-trivial part of the process. Consider the formula: $\forall x \exists y \, P(x, y)$. Here, y is existentially quantified and depends on x. So we can mark this dependence explicitly by setting $y = Y(x)$. Then the formula becomes $\forall x \, P(x, Y(x))$. You can even think of the function Y as being existentially quantified over outside the $\forall x$.
- Next, we distribute disjunction over conjunction as before.
- Finally, we simply drop all universal quantifiers. Because those are the only quantifiers left, there is no ambiguity.
- The final CNF formula can be difficult to interpret, but we can be assured that the final formula captures exactly the same information as the original formula.

Resolution



Definition: resolution rule (first-order logic)

$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg q \vee g_1 \vee \cdots \vee g_m}{\text{Subst}[\theta, f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m]}$$

where $\theta = \text{Unify}[p, q]$.

Example: resolution

$$\frac{\text{Animal}(Y(x)) \vee \text{Loves}(Z(x), x), \quad \neg\text{Loves}(u, v) \vee \text{Feeds}(u, v)}{\text{Animal}(Y(x)) \vee \text{Feeds}(Z(x), x)}$$

Substitution: $\theta = \{u/Z(x), v/x\}$.

- After convering all formulas to CNF, then we can apply the resolution rule, which is generalized to first-order logic. This means that instead of doing exact matching of a literal p, we unify atomic formulas p and q, and then apply the resulting substitution θ on the conclusion.

Logic: recap

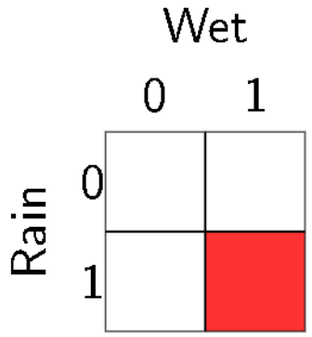
Review: ingredients of a logic

Syntax: defines a set of valid **formulas** (Formulas)

Example: Rain $\wedge$ Wet

Semantics: for each formula f, specify a set of **models** $\mathcal{M}(f)$ (assignments / configurations of the world)

Example:



Inference rules: given KB, what new formulas f can be derived?

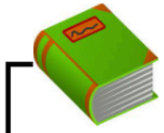
Example: from Rain $\wedge$ Wet, derive Rain

- Logic provides a formal language to talk about the world.
- The valid sentences in the language are the logical formulas, which live in syntax-land.
- In semantics-land, a model represents a possible configuration of the world. An interpretation function connects syntax and semantics. Specifically, it defines, for each formula f, a set of models $\mathcal{M}(f)$.

Review: inference algorithm

Inference algorithm:

KB —(repeatedly apply inference rules)→ f



Definition: modus ponens inference rule

$$\frac{p_1, \quad \cdots \quad , p_k, \quad (p_1 \wedge \cdots \wedge p_k) \rightarrow q}{q}$$

Desiderata: soundness and completeness

entailment ($\text{KB} \models f$)

derivation ($\text{KB} \vdash f$)

- A knowledge base is a set of formulas we know to be true. Semantically the KB represents the conjunction of the formulas.
- The central goal of logic is inference: to figure out whether a query formula is entailed by, contradictory with, or contingent on the KB (these are semantic notions defined by the interpretation function).
- The unique thing about having a logical language is that we can also perform inference directly on syntax by applying **inference rules**, rather than always appealing to semantics (and performing model checking there).
- We would like the inference algorithm to be both sound (not derive any false formulas) and complete (derive all true formulas). Soundness is easy to check, completeness is harder.

Review: formulas

Propositional logic: any legal combination of symbols

$$(\text{Rain} \wedge \text{Snow}) \rightarrow (\text{Traffic} \vee \text{Peaceful}) \wedge \text{Wet}$$

Propositional logic with only Horn clauses: restricted

$$(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$$

- Whether a set of inference rules is complete depends on what the formulas are. Last time, we looked at two logical languages: propositional logic and propositional logic restricted to Horn clauses (essentially formulas that look like $p_1 \wedge \cdots \wedge p_k \rightarrow q$), which intuitively can only derive positive information.

Review: tradeoffs

Formulas allowed	**Inference rule**	**Complete?**
Propositional logic	modus ponens	no
Propositional logic (only Horn clauses)	modus ponens	yes
Propositional logic	resolution	yes

- We saw that if our logical language was restricted to Horn clauses, then modus ponens alone was sufficient for completeness. For general propositional logic, modus ponens is insufficient.
- In this lecture, we'll see that a more powerful inference rule, **resolution**, is complete for all of propositional logic.

Summary

Propositional logic	**First-order logic**
model checking	n/a
$\Leftarrow$ propositionalization	
modus ponens (Horn clauses)	modus ponens++ (Horn clauses)
resolution (general)	resolution++ (general)

++: unification and substitution

Key idea: variables in first-order logic

Variables yield compact knowledge representations.

- To summarize, we have presented propositional logic and first-order logic. When there is a one-to-one mapping between constant symbols and objects, we can propositionalize, thereby converting first-order logic into propositional logic. This is needed if we want to use model checking to do inference.
- For inference based on syntactic derivations, there is a neat parallel between using modus ponens for Horn clauses and resolution for general formulas (after conversion to CNF). In the first-order logic case, things are more complex because we have to use unification and substitution to do matching of formulas.
- The main idea in first-order logic is the use of variables (not to be confused with the variables in variable-based models, which are mere propositional symbols from the point of view of logic), coupled with quantifiers.
- Propositional formulas allow us to express large complex sets of models compactly using a small piece of propositional syntax. Variables in first-order logic in essence takes this idea one more step forward, allowing us to effectively express large complex propositional formulas compactly using a small piece of first-order syntax.
- Note that variables in first-order logic are not same as the variables in variable-based models (CSPs). CSPs variables correspond to atomic formula and denote truth values. First-order logic variables denote objects.