Axioms for sets

To define sets we use:
- Constant symbol $\emptyset$, which refers to the empty set
- Unary predicate $\text{Set}$, which is true of sets
- Binary predicates $x \in s$ and $s_1 \subseteq s_2$
- Binary function are $s_1 \cup s_2$, $s_1 \cap s_2$, and $\{ x|s \}$, which refers to the set resulting from adjoining element $x$ to set $s$

The only sets are the empty set and those made by adjoining something to a set:
- $\forall s: \text{Set}(s) \iff s = \emptyset \lor \exists x, s_2: \text{Set}(s_2) \land s = \{ x|s_2 \}$.
- The empty set has no elements adjoined into it:
  $\neg \exists x, s: \{ x|s \} = \emptyset$.

- Adjoining an element already in the set has no effect:
  $\forall x, s: x \in s \iff s = \{ x|s \}$.

- The only members of a set are those elements that were adjoined into it:
  $\forall x, s: x \in s \iff [\exists y, s_2: (s = \{ y|s_2 \}) \land (x = y \lor x \in s_2)]$.

- Set inclusion:
  $\forall s_1, s_2: s_1 \subseteq s_2 \iff (\forall x: x \in s_1 \iff x \in s_2)$.

- Two set are equal iff each is a subset of the other:
  $\forall s_1, s_2: (s_1 = s_2) \iff (s_1 \subseteq s_2 \land s_2 \subseteq s_1)$.
  $\forall x, s_1, s_2: x \in (s_1 \setminus s_2) \iff (x \in s_1 \land x \notin s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cup s_2) \iff (x \in s_1 \lor x \in s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cap s_2) \iff (x \in s_1 \land x \in s_2)$.  
  $\forall x, s_1, s_2: x \in s_2 \iff (x \in s_1 \lor x \in s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \setminus s_2) \iff (x \in s_1 \land x \notin s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cup s_2) \iff (x \in s_1 \lor x \in s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cap s_2) \iff (x \in s_1 \land x \in s_2)$.  
  $\forall x, s_1, s_2: x \in s_2 \iff (x \in s_1 \lor x \in s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \setminus s_2) \iff (x \in s_1 \land x \notin s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cup s_2) \iff (x \in s_1 \lor x \in s_2)$.  
  $\forall x, s_1, s_2: x \in (s_1 \cap s_2) \iff (x \in s_1 \land x \in s_2)$.
9 INFERENCE IN FIRST-ORDER LOGIC

- By eliminating the quantifiers from the sentences of predicate logic we reduce them to sentences of proposition logic and can turn to familiar inference rules.
- We may substitute the variable \( v \) in an universally quantified sentence \( \forall v : \alpha \) with any ground term, a term without variables.
- Substitution \( \theta \) is a binding list – set of variable/term pairs – in which each variable is given the ground term replacing it.
- Let \( \alpha(\theta) \) denote the result of applying the substitution \( \theta \) to the sentence \( \alpha \).
- Universal instantiation to eliminate the quantifier is the inference:
  \[
  \forall v : \alpha \\
  \alpha(v/g),
  \]
  for any variable \( v \) and ground term \( g \).

- In existential instantiation the variable \( v \) is replaced by a Skolem constant \( k \) a symbol that does not appear elsewhere in the KB.
  \[
  \exists v : \alpha \\
  \alpha(v/k),
  \]
  E.g., from the sentence \( \exists x : \text{Father}(John) = x \) we can infer the instantiation \( \text{Father}(John) = F_1 \), where \( F_1 \) is a new constant.
  Eliminating existential quantifiers by replacing the variables with Skolem constants and universal quantifiers with the set of all possible instantiations turns the KB essentially propositional.
  Ground atomic sentences such as \( \text{Student}(John) \) and \( \text{Mother}(John, Mary) \) must be viewed as proposition symbols.
  Therefore, we can apply any complete propositional inference algorithm to obtain first-order conclusions.
• When the KB includes a function symbol, the set of possible ground term substitutions is infinite
  \( \text{Father(Father(Father(John)))} \)
• By the famous theorem due to Herbrand (1930) if a sentence is entailed by the original, first-order KB, then there is a proof involving just a finite subset of the propositionalized knowledge base.
• Thus, nested functions can be handled in the order of increasing depth without losing the possibility to prove any entailed sentence.
• Inference is hence complete.
• Analogy with the halting problem for Turing machines however shows that the problem is undecidable.
• More exactly: the problem is semidecidable, the sketched approach comes up with a proof for entailed sentences.

9.2 Unification and Lifting

• Inference in propositional logic is obviously too inefficient.
• Writing out all variable bindings seems to be futile.
• When there is a substitution \( \theta \) s.t. \( p_i'(\theta) = p_i(\theta) \), for all \( i \), where \( p_i' \) and \( p_i \) are atomic sentences as well as \( q \), we can use the Generalized Modus Ponens (GMP):
  \[
  p_1', \ldots, p_n', (p_1 \land \ldots \land p_n \Rightarrow q) \\
  q(\theta)
  \]
• For example, from the fact \( \text{Student(John)} \) and sentences
  \( \forall x: \text{EagerToLearn}(x) \) and
  \( \forall y: \text{Student}(y) \land \text{EagerToLearn}(y) \Rightarrow \text{Thesis_2013}(y) \)
we can infer \( \text{Thesis_2013(John)} \) because of the substitution \( \{ y/\text{John}, x/\text{John} \} \).
• Generalized Modus Ponens is a sound inference rule
• Similarly as GMP can be lifted from propositional logic to first-order logic, also forward chaining, backward chaining, and the resolution algorithm can be lifted

• A key component of all first-order inference algorithms is unification
• The unification algorithm \textit{Unify} takes two sentences and returns a unifier for them if one exists:
  \begin{align*}
  \text{Unify}(p, q) &= \emptyset, \text{ s.t. } p(\emptyset) = q(\emptyset) \\
  \text{otherwise unification fails}
  \end{align*}

\begin{align*}
\text{Unify}(\text{Knows(John, x)}, \text{Knows(John, Jane)}) &= \{ x/\text{Jane} \} \\
\text{Unify}(\text{Knows(John, x)}, \text{Knows(y, Bill)}) &= \{ x/\text{Bill}, y/\text{John} \} \\
\text{Unify}(\text{Knows(John, x)}, \text{Knows(y, Mother(y))}) &= \{ y/\text{John}, x/\text{Mother(John)} \} \\
\text{Unify}(\text{Knows(John, x)}, \text{Knows(x, Eliza)}) &= \text{fail}
\end{align*}

• The last unification fails because \( x \) cannot take on the values \( \text{John} \) and \( \text{Eliza} \) simultaneously
• Because variables are universally quantified, \( \text{Knows(x, Eliza)} \) means that everyone knows \( \text{Eliza} \)
• In that sense, we should be able to infer that \( \text{John knows Eliza} \)
• The problem above arises only because the two sentences happen to use the same variable name
• The variable names of different sentences have no bearing (prior to unification) and one can standardize them apart
• There can be more than one unifier, which of them to return?
• The unification algorithm is required to return the (unique) most general unifier
• The fewer restrictions (bindings to constants) the unifier places, the more general it is

\[
\text{Unify( Knows(John, x), Knows(y, z) )}
\]
\[
\{ y/John, z/x \}
\]
\[
\{ y/John, x/John, z/John \}
\]

9.3 Forward Chaining

• As before, let us consider knowledge bases in Horn normal form
• A definite clause either is atomic or is an implication whose body is a conjunction of positive literals and whose head is a single positive literal

\[
\text{Student(John)}
\]
\[
\text{EagerToLearn(x)}
\]
\[
\text{Student(y) \land EagerToLearn(y) \Rightarrow Thesis\_2013(y)}
\]

• Unlike propositional literals, first-order literals can include variables
• The variables are assumed to be universally quantified
As in propositional logic we start from facts, and by applying Generalized Modus Ponens are able to do forward chaining inference.

One needs to take care that a "new" fact is not just a renaming of a known fact.

\[ Likes(x, \text{Candy}) \]
\[ Likes(y, \text{Candy}) \]

Since every inference is just an application of Generalized Modus Ponens, forward chaining is a sound inference algorithm. It is also complete in the sense that it answers every query whose answers are entailed by any knowledge base of definite clauses.

---

**Datalog**

In a Datalog knowledge base the definite clauses contain no function symbols at all. In this case we can easily prove the completeness of inference.

Let in the knowledge base:
- \( p \) be the number of predicates,
- \( k \) be the maximum arity of predicates (= the number of arguments), and
- \( n \) the number of constants.

There can be no more than \( p n^k \) distinct ground facts.
So after this many iterations the algorithm must have reached a fixed point, where new inferences are not possible.
• In Datalog a polynomial number of steps is enough to generate all entailments
• For general definite clauses we have to appeal to Herbrand’s theorem to establish that the algorithm will find a proof
• If the query has no answer (is not entailed by the KB), forward chaining may fail to terminate in some cases
• E.g., if the KB includes the Peano axioms, then forward chaining adds facts

NatNum(S(0)).
NatNum(S(S(0))).
NatNum(S(S(S(0)))).
...

• Entailment with definite clauses is semidecidable

9.4 Backward Chaining

• In predicate logic backward chaining explores the bodies of those rules whose head unifies with the goal
• Each conjunct in the body recursively becomes a goal
• When the goal unifies with a known fact – a clause with a head but no body – no new (sub)goals are added to the stack and the goal is solved
• Depth-first search algorithm
• The returned substitution is composed from the substitutions needed to solve all intermediate stages (subgoals)
• Inference in Prolog is based on backward chaining
9.4.2 Logic programming

- Prolog, Alain Colmerauer 1972
- Program = a knowledge base expressed as definite clauses
- Queries to the knowledge base
  - **Closed world assumption**: we assume \( \neg \psi \) to be true if sentence \( \psi \) is not entailed by the knowledge base
- Syntax:
  - Capital characters denote variables,
  - Small character stand for constants,
  - The head of the rule precedes the body,
  - Instead of implication use :–,
  - Comma stand for conjunction,
  - Period ends a sentence

\[
\text{thesis}_2013(X) :– \text{student}(X), \text{eager}_\text{to}_\text{learn}(X).
\]

- Prolog has a lot of syntactic sugar, e.g., for lists and arithmetics

Prolog program \texttt{append(X, Y, Z)} succeeds if list \( Z \) is the result of appending (catenating) lists \( X \) and \( Y \)

\[
\begin{align*}
\text{append}([\ ] , Y, Y). \\
\text{append([A|X], Y, [A|Z]) :– append(X, Y, Z).}
\end{align*}
\]

- Query: \texttt{append([1], [2], Z)-fw}
  \[
  Z=[1,2]
  \]
- We can also ask the query
  \[
  \text{append}(A, B, [1,2])?\]
  Appending what two lists gives the list \([1,2]\) ?
- As the answer we get back all possible substitutions
  \[
  \begin{align*}
  A=&[ ] \quad &B=&[1,2] \\
  A=&[1] \quad &B=&[2] \\
  A=&[1,2] \quad &B=&[ ]
  \end{align*}
  \]
The clauses in a Prolog program are tried in the order in which they are written in the knowledge base. Also the conjuncts in the body of the clause are examined in order (from left to right). There is a set of built-in functions for arithmetic, which need not be inferred further. E.g., $X \text{ is } 4+3 \rightarrow X=7$.

For instance I/O is taken care of using built-in predicates that have side effect when executed. Negation as failure:

$$\text{alive}(X) : - \text{ not } \text{dead}(X).$$

"Everybody is alive if not provably dead"

The negation in Prolog does not correspond to the negation of logic (using the closed world assumption)

$$\text{single\_student}(X) : -$$
  $$\text{not } \text{married}(X), \text{student}(X).$$
  $$\text{student}(\text{peter}).$$
  $$\text{married}(\text{john}).$$

By the closed world assumption, $X=\text{peter}$ is a solution to the program. The execution of the program, however, fails because when $X=\text{john}$ the first predicate of the body fails. If the conjuncts in the body were inverted, it would succeed.
• An equality goal succeeds if the two terms are unifiable
  • E.g., \( X+Y=2+3 \Rightarrow X=2, \ Y=3 \)

• Prolog omits some necessary checks in connection of variable bindings \( \Rightarrow \) Inference is not sound
• These are seldom a problem
• Depth-first search can lead to infinite loops (= incomplete)
  \[
  \text{path}(X,Z) :\text{-} \text{path}(X,Y), \text{link}(Y,Z).
  \]
  \[
  \text{path}(X,Z) :\text{-} \text{link}(X,Z).
  \]
• Careful programming, however, lets us escape such problems
  \[
  \text{path}(X,Z) :\text{-} \text{link}(X,Z).
  \]
  \[
  \text{path}(X,Z) :\text{-} \text{path}(X,Y), \text{link}(Y,Z).
  \]

• Anonymous variable _
  \[
  \text{member}(X,[X|\_]).
  \]
  \[
  \text{member}(X,[_|Y]) :\text{-} \text{member}(X,Y).
  \]
• Works just fine
  \[
  \text{member}(d,[a,b,c,d,e,f,g])?
  \]
  yes
  \[
  \text{member}(2,[3,a,4,f])?
  \]
  no
• But queries
  \[
  \text{member}(a,X)\
  \text{member}(a,[a,b,r,a,c,a,d,a,b,r,a])?
  \]
do not necessarily give the intended answers
We can explicitly prune the execution of Prolog programs by cutting.

Negation using cut

\[
\text{not } X :- X, !, \text{ fail.}
\]
\[
\text{not } X.
\]

\text{fail} causes the program to fail.

At the point of a cut all bindings that have been made since starting to examine the rule are fixed.

For the conjuncts in the body preceding the cut, no new solutions are searched for.

Neither does one examine other rules having the same head.

Prolog may come up with the same answer through several inference paths.

Then the same answer is returned more than once:

\[
\text{minimum}(X, Y, X) :- X \leq Y.
\]
\[
\text{minimum}(X, Y, Y) :- X \geq Y.
\]

Both rules yield the same answer for the query \text{minimum}(2, 2, M) ?

One must be careful in using cut for optimizing inference:

\[
\text{minimum}(X, Y, X) :- X \leq Y, !.
\]
\[
\text{minimum}(X, Y, Y).
\]

This program is erroneous, for instance \text{minimum}(2, 8, 8) holds according to it.
The key question of Prolog and logic programming obviously is efficiency of execution. Prolog implementations use a wide variety of enhancement techniques. For example, instead of generating all possible solutions for a subgoal before examining the next subgoal, a Prolog interpreter is content (so far) with just one. Similarly, variable binding is at each instant unique; only when the search runs into a dead end, can backing up to a choice point lead to unbinding of variables. A stack of history, called the trail, needs to be maintained to keep track of all variable bindings.

9.5 Resolution

Kurt Gödel’s completeness theorem (1930) for first-order logic: any entailed sentence has a finite proof

\[ T \models \varphi \iff T \vdash \varphi \]

It was not until Robinson’s (1965) resolution algorithm that a practical proof procedure was found. Gödel’s more famous result is the incompleteness theorem: a logical system that includes the principle of induction, is necessary incomplete. There are sentences that are entailed, but have no finite proof. This holds in particular for number theory, which thus cannot be axiomatized.
• For resolution, we need to convert the sentences to CNF
  • E.g., “Everyone who loves all animals is loved by someone”
    \[ \forall x: [\forall y: \text{Animal}(y) \implies \text{Loves}(x, y)] \implies [\exists y: \text{Loves}(y, x)]. \]

  • Eliminate implications
    \[ \forall x: [\neg \forall y: \neg \text{Animal}(y) \lor \text{Loves}(x, y)] \lor [\exists y: \text{Loves}(y, x)]. \]

  • Move negation inwards
    \[ \forall x: [\exists y: \text{Animal}(y) \land \neg \text{Loves}(x, y)] \lor [\exists y: \text{Loves}(y, x)]. \]

  • Standardize variables
    \[ \forall x: [\exists y: \text{Animal}(y) \land \neg \text{Loves}(x, y)] \lor [\exists z: \text{Loves}(z, x)]. \]

• Skolemization
  \[ \forall x: [\text{Animal}(F(x)) \land \neg \text{Loves}(x, F(x))] \lor \text{Loves}(G(z), x). \]

• Drop universal quantifiers
  \[ [\text{Animal}(F(x)) \land \neg \text{Loves}(x, F(x))] \lor \text{Loves}(G(z), x). \]

• Distribute \lor over \land
  \[ [\text{Animal}(F(x)) \lor \text{Loves}(G(z), x)] \land \\
    [\neg \text{Loves}(x, F(x)) \lor \text{Loves}(G(z), x)]. \]

• The end result is quite hard to comprehend, but it doesn’t matter, because the translation procedure is easily automated
First-order literals are complementary if one unifies with the negation of the other. Thus the binary resolution rule is:

\[
\frac{l_1 \lor \ldots \lor l_k \lor m}{(l_1 \lor \ldots \lor l_{i-1} \lor l_{i+1} \lor \ldots \lor l_k)(\theta)}
\]

where \(\text{Unify}(l_i, \neg m) = \theta\).

For example, we can resolve

\([\text{Animal}(F(x)) \lor \text{Loves}(G(x), x)]\) and \([\neg \text{Loves}(u, v) \lor \neg \text{Kills}(u, v)]\)

by eliminating the complementary literals

\(\text{Loves}(G(x), x)\) and \(\neg \text{Loves}(u, v)\)

with unifier \(\theta = \{ u/G(x), v/x \}\) to produce the resolvent clause

\([\text{Animal}(F(x)) \lor \neg \text{Kills}(G(x), x)]\).

Resolution is a complete inference rule also for predicate logic in the sense that we can check (not generate) all logical consequences of the knowledge base \(\text{KB}\).

\(\text{KB} \models a\) is proved by showing that

\(\text{KB} \land \neg a\)

is unsatisfiable through a proof by refutation.
• Theorem provers (automated reasoners) accept full first-order logic, whereas most logic programming languages handle only Horn clauses
  • For example Prolog intertwines logic and control
  • In most theorem provers, the syntactic form chosen for the sentences does not affect the result
  • Application areas: verification of software and hardware
  • In mathematics theorem provers have a high standing nowadays: they have come up with novel mathematical results
  • For instance, in 1996 a version of well-known Otter was the first to prove (eight days of computation) that the axioms proposed by Herbert Robbins in 1933 really define Boolean algebra

13 QUANTIFYING UNCERTAINTY

• In practice agents almost never have full access to the whole truth of their environment and, therefore, must act under uncertainty
  • A logical agent may fail to acquire certain knowledge that it would require
  • If the agent cannot conclude that any particular course of action achieves its goal, then it will be unable to act
  • Conditional planning can overcome uncertainty to some extent, but it does not resolve it
  • An agent based solely on logics cannot choose rational actions in an uncertain environment
Logical knowledge representation requires rules without exceptions.

In practice, we can typically at best provide some degree of belief for a proposition.

In dealing with degrees of belief we will use probability theory.

Probability 0 corresponds to an unequivocal belief that the sentence is false and, respectively, 1 to an unequivocal belief that the sentence is true.

Probabilities in between correspond to intermediate degrees of belief in the truth of the sentence, not on its relative truth.

Utilities that have been weighted with probabilities give the agent a chance of acting rationally by preferring the action that yields the highest expected utility.

Principle of Maximum Expected Utility (MEU)

13.2 Basic Probability Notation

Probabilistic assertions are about possible worlds just like logical assertions.

However, they talk about how probable the various worlds are.

The set of all possible worlds $\Omega$ is called the sample space.

The possible worlds are mutually exclusive and exhaustive.

If we roll two (indistinguishable) dice, there are 36 worlds to consider: $(1,1), (1,2), \ldots, (6,6)$.

A fully specified probability model associates a numerical probability $P(\omega)$ with each possible world.

The basic axioms of probability theory say that:

1. every possible world has a probability between 0 and 1
   
   
   
   

2. the total probability of the set of possible worlds is 1
   
   
   

\[ \sum_{\omega \in \Omega} P(\omega) = 1 \]
Probabilistic assertions and queries are not usually about particular possible worlds, but about sets of them.

- For example, we might be interested in the cases where the two dice add up to 11.
- In probability theory these sets are called events.
- In AI the sets are always described by propositions in a formal language.
- The probability associated with a proposition is defined to be the sum of the probabilities of the worlds in which it holds.
- For any proposition $\phi$,

$$P(\phi) = \sum_{\omega \in \phi} P(\omega)$$

Prior and posterior probability

- Rolling fair dice, we have

$$P(\text{Total} = 11) = P((5,6)) + P((6,5)) = \frac{1}{36} + \frac{1}{36} = \frac{1}{18}$$

- Probability such as $P(\text{Total} = 11)$ is called unconditional or prior probability.
- $P(\alpha)$ is the degree of belief accorded to proposition $\alpha$ in the absence of any other information.
- Once the agent has obtained some evidence, we have to switch to using conditional (posterior) probabilities.

$$P(\text{doubles} | \text{Die}_1 = 5)$$

- $P(\text{cavity}) = 0.2$ is interesting when visiting a dentist for regular checkup, but $P(\text{cavity} | \text{toothache}) = 0.6$ matters when visiting the dentist because of a toothache.
• We can express conditional probabilities in terms of unconditional probabilities:

\[ P(a|b) = \frac{P(a \land b)}{P(b)} \]

whenever \( P(b) > 0 \)

• E.g.,

\[ P(\text{doubles} \mid \text{Die}_1 = 5) = \frac{P(\text{doubles} \land \text{Die}_1 = 5)}{P(\text{Die}_1 = 5)} \]

• Rewriting the definition of conditional probability yields the product rule

\[ P(a \land b) = P(a \mid b) P(b) \]

• We can, of course, have the rule the other way around

\[ P(a \land b) = P(b \mid a) P(a) \]

• A random variable refers to a part of the world, whose status is initially unknown

• Random variables play a role similar to proposition symbols in propositional logic

• E.g., Cavity might refer whether the lower left wisdom tooth has a cavity

• The domain of a random variable may be of type
  - Boolean: we write \( \text{Cavity} = \text{true} \Rightarrow \text{cavity} \) and \( \text{Cavity} = \text{false} \Rightarrow \neg \text{cavity} \);
  - discrete: e.g., Weather might have the domain \( \{ \text{sunny, rain, cloudy, snow} \} \);
  - continuous: then one usually examines the cumulative distribution function; e.g., \( X \leq 4.02 \)