# Non-monotonic Reasoning

## Common Sense

# Monotonicity

A formalism is **monotonic** when the addition of new knowledge can only increase the number of conclusions

If $\Gamma \subseteq \Delta$ and $\Gamma \vdash A$ then $\Delta \vdash A$, or also $Cn(\Gamma) \subseteq Cn(\Delta)$

where $Cn(KB)$ denotes the set of logical consequences of the $KB$.

# Non-monotonic Reasoning

Conversely, in a non-monotonic formalism, the addition of new information can invalidate some of the previously derivable conclusions

Non-monotonicity is a property of the formalism, rather than of the reasoning

**Incomplete Representation** (lack of information)

**Common Sense Reasoning**

# Negative Information

$flight(c_1, c_2)$ indicates a connection between two cities

$\forall xyz.(flight(x, y) \wedge flight(y, z) \Rightarrow flight(x, z))$

In classical logic we cannot formally derive that two cities are not connected

Assumption: when we cannot formally derive the existence of a connection, that connection doesn't exist

For example, if we cannot derive $flight(Roma, Orte)$,
we assume $\neg flight(Roma, Orte)$

Non-monotonicity: if we add $flight(Roma, Orte)$ we cannot derive any longer that $\neg flight(Roma, Orte)$ holds

# Universal and General Assertions

Violins have 4 cords

- **Universal**: Assertive properties that hold for all the instances
- **General**: Properties that generally hold

Violins have 4 cords, but when they have some trouble

Non-monotonicity: If a violin "looses a cord," it remains a violin with three cords

# Exceptions

Birds fly:
$\forall x.bird(x) \Rightarrow flies(x)$

In this form the rule doesn't allow for exceptions, but kiwis do not fly

$\forall x.bird(x) \wedge \neg kiwi(x) \Rightarrow flies(x)$

In addition, to kiwis there are several other exceptions:

$\forall x.bird(x) \wedge \neg kiwi(x) \wedge ... \Rightarrow flies(x)$

In general, not all exceptions are known

# Closed World Reasoning

Basic Idea:

There are many more false things than true things. If something is true and relevant, it has been put into the KB. So, if something is not present in the KB, it can reasonably be assumed to be false.

- Closed World Assumption
- Negation as Failure

# Closed World Assumption

Closed World Assumption: CWA (Reiter '78).

$$CLOSURE(KB) = KB \cup \{\neg L | L \text{ is a positive literal and } KB \not\models L\}$$

$$CLOSURE(KB) = KB^{+}$$

$KB^{+}$ contains all the sentences of the form $\neg flight(Roma, Orte)$

Answers to queries are obtained from $KB^{+}$ instead of $KB$

# Consistency of the CWA

CWA is consistent on definite $KB$s

If $KB$ contains **incomplete** knowledge:

$KB \models A \vee B$, but $KB \not\models A$ and $KB \not\models B$

$KB^+$ contains both $\neg A$ and $\neg B$, hence we have an inconsistency

# Generalized CWA

GCWA

$KB^* = KB \cup \{\neg L | L$ is a positive literal
and there is no positive clause $C$
such that $KB \models L \vee C$ and $KB \not\models C\}$

Further restrictions are possible (CCWA, ECWA).

# Negation as failure

Negation as (finite) failure ($P$ is a KB and $A$ an atom)

*If from $P$ we do not prove $A$ then from $P$ we deduce $\neg A$*

# SLDNF = SLD-resolution + NF

SLD (Selective Linear Resolution) + NF (Negation as Finite Failure):
an atom $\neg A$ succeeds if the derivation tree corresponding to the goal $A$ is finite and its leaves are all failure leaves.

Consistent (with the various characterizations of negation), but incomplete.

Conditions for completeness are very stringent:

- instantiated variables in negated atoms
- constraints on the program structure

Answer Set Programming extends SLDNF, computing the answer on the basis of the stable model semantics

# Negation as Failure in PROLOG

In prolog negation is realized as **failure** in the search.

If `X` is a list of atoms, `not(X)` is true if the interpreter cannot find a proof for `X`.

```
student(bill).
student(joe).
married(joe).

unmarried_student(X) :- not(married(X)), student(X).
? unmarried_student(bill/joe).
```

# Termination of negation

The negation mechanism of Prolog is neither correct nor complete: it depends on the ordering of evaluation of clauses

The termination of `not(X)` depends on the termination of `X`:
- If `X` terminates, `not(X)` terminates.
- If `X` has infinite solutions, `not(X)` terminates if a success node is found before an infinite branch is encountered.

```
married(gino, remberta).
married(X,Y):-married(Y,X).

? not(married(remberta, gino)).

? not(married(remberta, giovanni)).
```

# Problems with negation

```
unmarried_student(X):-not(married(X)),student(X).
...
?- unmarried_student(X).

unmarried_student1(X) :- student(X), not(married(X)).
student(bill).
student(joe).
married(joe).
?- unmarried_student1(X).
```

**Sufficient** condition for correctness:
variables in negated atoms must be instantiated before they are executed