Logic & Computation
Distributed Systems / Paradigms
Sistemi Distribuiti / Paradigmi

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2016/2017
1 Logic

2 Logic Programming

3 Distributed Logic Programming
Next in Line...
What is Logic?

Logic

http://www.britannica.com/topic/logic

**Logic**, the study of correct reasoning, especially as it involves the drawing of inferences.

- logic studies the way we draw conclusions and express ourselves, and deals with how to formalise it [Metakides and Nerode, 1996a]
- it dates back to Aristotle, for whom logic is the instrument for human knowledge [Rijk, 2002]
Computational Logic

Logic in Computer Science

- **computational logic** (CL) [Lloyd, 1990] is *logic in computer science*
- it concerns all uses of logic in computer science: to compute, to represent computation, to reason about computation
- it is rooted in logic, applied math, artificial intelligence, computer science
Early history

- after a long lack of interest, at the end of the 17th century, Gottfried Leibniz (quite isolatedly) observed that if ideas are concepts like numbers, they should be possibly represented and manipulated in the same way as we do with numbers—clearly bridging logic and math for the first time [Levesque, 2012]

- a formalism for mathematics is typically credited to George Boole, starting form the middle of the 19th century [Boole, 2009]: along with Augustus De Morgan, they started extending logic to become a tool for the study of the foundations of mathematics

- logic finally regain centrality in Western thought with Gottlob Frege, introducing the first formal language for logic and mathematics for grounding mathematical reasoning on a sound basis [Frege, 1971]
The rise of mathematical logic

- starting from a paradox in Cantor’s theory of numbers, *Russell’s paradox* (the set of all sets that do not belong to themselves) exposed the logical *inconsistencies* at the foundations of mathematics.
- this made clear that only a rigorous formalisation of mathematics could lead to well-founded mathematical reasoning and theories.
- around 1920, David *Hilbert’s program* aimed at addressing the issue—to be soon undermined by Gödel’s Incompleteness Theorems [*Gödel, 1931*].
- the acme of those efforts resulted in the “Principia Mathematica” [*Whitehead and Russell, 1927*], a complete logic formalisation of all mathematics, finally treated as a whole *axiomatic system*. 
Syntax vs. semantics: examples

- “the sum of numbers two and one is three” is a true proposition in any formalisation of arithmetics, whatever symbols we use—the semantic side of truth.

- If “the Snark was a Boojum” [Carroll, 1876] then “something is a Boojum” is a true proposition whatever the meaning of the sentence—the syntax side of truth.
Two notions of truth for a sentence

According to the above examples, we have two ways to set the truth of a sentence, respectively

- by considering a sentence as a sequence of symbols, and manipulating them independently of their meaning
- by interpreting symbols in the sentence, and considering the truth of the meaning of the sentence
Interpretation and logic entailment

- **Alfred Tarski** [Tarski and Tarski, 1994] formalised the notion of **logical interpretation** as the relation between the *symbols* (syntax) and the elements of the *domain of discourse* (semantics).
- He defined **logical entailment** (or, **logical consequence**) precisely: when a *conclusion* is true in every interpretation making all *premises* true, then premises logically entail the conclusion—or, the conclusion is a logical consequence of the premises.
after Tarski’s work, logic is typically discussed as a two-sided story

[Levesque, 2012]

- a *syntactic side* involving *axioms* and *rules of inference*—sometimes called a *proof theory*
- a *semantic side* involving *interpretations* and *truth*—sometimes called a *model theory*
- with logical *soundness* and *completeness* theorems relating the two sides
Truth V

logic axioms

no interpretation

interpretation

syntax
(symbols)

semantics
(meanings)

theorem proof

logic entailment
Truth VI

- Logic axioms
- No interpretation
- Interpretation
- Syntax (symbols)
- Semantics (meanings)
- Soundness
- Theorem proof
- Logic entailment
- Completeness
Computation vs. Deduction I

The difference between computation and deduction [Pfenning, 2007]

- To compute we start from a given expression and, according to a fixed set of rules (the program) generate a result. For example, $15 + 26 \rightarrow (1 + 2 + 1)1 \rightarrow (3 + 1)1 \rightarrow 41$.
- To deduce we start from a conjecture and, according to a fixed set of rules (the axioms and inference rules), try to construct a proof of the conjecture.
- So computation is mechanical and requires no ingenuity, while deduction is a creative process. For example, $a^n + b^n \neq c^n$ for $n > 2$, . . . , 357 years of hard work . . . , QED.
How do computation and deduction relate? [Pfenning, 2007]

- in some restricted areas they can be unified—e.g., Boolean algebras
  http://www.britannica.com/topic/Boolean-algebra

- more generally, even if we follow a well-defined set of formal rules, we know that not everything we can reason about is mechanically computable—given the fundamental undecidability results [Gödel, 1931]
Yet, somehow computation $\iff$ deduction [Pfenning, 2007]

1. computation can be seen as a limited form of deduction since it establishes theorems—e.g., $15 + 26 = 41$ is both the result of a computation, and a theorem of arithmetic.

2. deduction can be considered a form of computation if we fix a strategy for proof search, removing the guesswork (and the possibility of employing ingenuity) from the deductive process.

This latter idea is the foundation of logic programming. Logic program computation proceeds by proof search according to a fixed strategy. By knowing what this strategy is, we can implement particular algorithms in logic, and execute the algorithms by proof search.
## Judgements [Martin-Löf, 1996]

- **a judgement** is an *object of knowledge*—knowing a judgement comes from the *evidence* we have for it.
- The most common judgement is *A true*—that is, given a proposition *A*, *A* is true.
  - Most of logic programming deals with *truth of propositions*.
- Other judgements are:
  - *A false* — that is, *A* is false — for *true negation*.
  - *A true at t* — that is, *A* is true at time *t* — the subject of *temporal logic*.
  - *K knows A* — that is, *K* knows that *A* is true — the subject of *epistemic logic*. 

---

Andrea Omicini (DISI, Univ. Bologna)  
P1 – Logic & Computation  
A.Y. 2016/2017  
17 / 85
**Proof through deduction** [Pfenning, 2007]

- the most interesting evidence here is a **proof**
- \( J, J_1, \ldots, J_n \) be judgements, \( R \) an **inference rule**: a **deduction** is a **proof** that could be (equivalently) read as
  - if \( J_1 \) and \( \ldots \) and \( J_n \), then we can conclude \( J \) by virtue of rule \( R \)
  - given \( J_1 \) and \( \ldots \) and \( J_n \), we can **prove** \( J \) by rule \( R \)
  - \( J \) can be **deduced/inferred** from \( J_1 \) and \( \ldots \) and \( J_n \) by means of rule \( R \)
where \( J_1, \ldots, J_n \) are the **premises**, and \( J \) is the **conclusion**
- formally, we write the deduction as

\[
\frac{J_1, \ldots, J_n}{J} R
\]
Logic

Deduction III

Inference chain

- if we know $J_1, \ldots, J_n$ and $\frac{J_1, \ldots, J_n}{J} R$ then we can infer $J$
- how do we know, say, $J_1$?
  - we may know $J_{1,1}, \ldots, J_{1,n}$ and $\frac{J_{1,1}, \ldots, J_{1,n}}{J_1} R_1$, then infer $J_1$
- thus we may build an inference chain to prove $J$—if possible
Proof tree

- given the structure of a deduction, the data structure covering all the possible inference chains for a judgement \( J \) is a tree—a proof tree
  - whose root is the conclusion \( J \)
  - whose edges are deductions labelled by an inference rule
  - where premises are child nodes, conclusions are parent nodes

\[
\begin{align*}
\overline{\ldots R_{1,1}, \ldots} & \quad \overline{\ldots R_{1,n_1}} \\
\overline{J_{1,1}, \ldots, J_{1,n_1}} & \quad \overline{R_1 \ldots} \\
\overline{J_1} & \quad \overline{J_n} \\
\overline{\ldots R_{n,1}, \ldots} & \quad \overline{\ldots R_{n,n_n}} \\
\end{align*}
\]

- \( J \) is the root of the tree
- \( J_1, \ldots J_n \) are nodes of the tree
and so are $J_{1,1}, \ldots$ 

- $R, R_1, \ldots, R_n$ are edges of the tree 
  - and so are $R_{1,1}, \ldots$

- $J_1, \ldots J_n$ are parent nodes of node $J$—which is then their child node 
  - and so are $J_{1,1}, \ldots J_{1,n}$ for node $J_1, \ldots$
What are the *leaves* of the proof tree?

- where do we start with our deductions?
- is there any judgements that can be said to be true without proof?
- this is the role of *axioms*
- axioms can be used as premises in a proof tree with no need of proof

**Axioms**

http://www.britannica.com/topic/axiom

*Axiom*, in logic, an *indemonstrable* first principle, rule, or maxim, that has found general acceptance or is thought worthy of common acceptance whether by virtue of a claim to *intrinsic* merit or on the basis of an appeal to *self-evidence*.
How do we explore the proof tree?

- where do we start with our proofs?
- should we start from what we know, and try to prove our conjecture, deduction after deduction?
- or, should we start from the conjecture, find the deductions that prove it, and try to prove the premises, recursively?
- or, again, should we just try to build the proof tree in any other way, then explore it with some other strategy?
Forward Chaining

From the axioms forward to the conjecture

- forward-reasoning search
  - we start from what we know (initially, the axioms)
  - we exploit inference rules to deduce new judgements (theorems)
  - we add new evidence to the old one
  - recursively
  - until we obtain our conjecture

- traversing the proof tree from the leaves down to the root

\[
\frac{\cdots J_1,1, \cdots, \cdots J_1,n_1, \cdots}{\cdots R_1,1, \cdots, \cdots R_1,n_1} \quad \frac{\cdots J_n,1, \cdots, \cdots J_n,n_n, \cdots}{\cdots R_n,1, \cdots, \cdots R_n,n_n} \quad R
\]
Backward Chaining

From the conjecture backward to the axioms

- **goal-directed** search
  - we start from what we need to prove (initially, the conjecture)
  - we exploit inference rules to find the judgements which it depends upon
  - we find new judgements to the prove
  - recursively
  - until we end depending only from the axioms
- traversing the proof tree from the root up to the leaves
Computing search strategies

- both strategies seems to fit computation, but
  - would this be a deterministic computation?
  - would it converge anyway?
  - would it actually terminate?
  - most importantly, here: could it be a parallel / concurrent / distributed computation?
Proof Search & Computation II

Preliminary answers

- many inference rules could apply at the same time: potentially non-deterministic computation
- sequential computation could take the wrong path: potentially diverging computation
- proof trees might be infinite: potentially non-terminating computation
- given an inference rule, all premises for the given conclusion could be, e.g., proven in parallel
- given a conclusion, all the applicable inference rules could be, e.g., tried concurrently
- given a set of axioms, the same conjecture could be proven, e.g., using different strategies at the same time by exploiting many distributed devices
Logic Programming

Next in Line...
Origins I

Early history [Apt, 2005]

- automatic deduction of theorems
- *first-order logic* (FOL) by Frege, Peano and Russell
- computation as deduction by Gödel and Herbrand
- *resolution* principle by Robinson [Robinson, 1965], along with *unification*

The key issue

- resolution by Robinson
  - allowed proof of FOL theorem made it possible to compute with logic
  - not yet to see logic as a full computational framework
- from computable logic to logic as a programming language something was still missing
The procedural interpretation of Horn clauses

- by defining logic programs as collections of Horn clauses
- by restricting Robinson’s principle accordingly
- Kowalski showed how a logical implication could be amenable of both a *declarative* and a *procedural* implication [Kowalski, 1974]
- thus providing the *foundations* for a logic programming language
- Prolog, by Colmerauer in Marseille, came along in 1973

*There is no question that Prolog is essentially a theorem prover à la Robinson. Our contribution was to transform that theorem prover into a programming language.* [Colmerauer and Roussel, 1996]
Three fundamental features [Apt, 2005]

<table>
<thead>
<tr>
<th>terms</th>
<th>Computing takes place over the domain of all terms defined over a “universal” alphabet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgu</td>
<td>Values are assigned to variables by means of automatically-generated substitutions, called most general unifiers. These values may contain variables, called logical variables.</td>
</tr>
<tr>
<td>backtracking</td>
<td>The control is provided by a single mechanism: automatic backtracking.</td>
</tr>
</tbody>
</table>
Declarative programming

- according to Aristotle, **declarative** is a sentence that can be said either *true* or *false* [Rijk, 2002]

→ **declarative programming** means first of all programming through (true) sentences, which declare *what* to compute—the **meaning**

- **procedural programming** is instead programming through *operational statements*, which determine *how* to compute—the **method**

- e.g., in object-oriented languages, classes and interfaces are defined declaratively, whereas methods are defined procedurally

- logic programming is amenable of either a **declarative** or an **operational interpretation**, and the two corresponding **semantics match** [Kowalski, 1974]
Declarative programming: features and issues [Apt, 2005]

- logic programs can be seen as executable specifications
  - the logic programmer is concerned on what to compute
  - how to compute (control) is delegated to the underlying (logic programming) machinery

  ! sometimes this could lead to inefficiency

- logic programming languages can be seen as formalisms for either executable code or knowledge representation

→ languages for artificial intelligence
Interactive programming

- the model behind the notion of computation as deduction natively supports the idea of writing a logic program, then interact with the logic machinery by means of multiple queries, or, by asking for multiple solutions

- logic languages intrinsically support the interactive style of programming and computing

while this will be evident in the lab session, it should be already clear how such a feature could be useful in distributed systems, supporting novel notions such as LPaaS (Logic Programming as a Service [Calegari et al., 2016])
Atomic actions [Apt, 2005]

- logic programming is a different paradigm for programming languages since it is ruled by different principles w.r.t. the other sorts of programming languages.
  - atomic actions are equations between terms
  - executed by means of the unification process trying to solve them
  - unification assigns values to variables
  - values can be arbitrary terms—in fact, there is just one sort of variable, ranging over the set of all terms

- so, in order to understand logic programming as a computational paradigm, we first need to understand its basic units of computation
Terms: Definition

- a variable is a term
- a functor (or, function symbol) with arity 0 is called a constant, and is a term
- if \( f \) is a functor of arity \( n \), and \( t_1, \ldots, t_n \) are \( n \) terms, then \( f(t_1, \ldots, t_n) \) is a term
**Terms: Examples**

- Let's say that $X$, $Y$ are variables, $a$, $b$ constants (or, functors of arity 0), $f$, $g$ functors of arity 3, 2 respectively. Then
  - $a$, $b$, $X$, and $Y$ are proper terms
  - $f(a, b, a)$ and $g(X, Y)$ are proper terms
  - $f(a, X, g(Y, b))$ is a proper term
- Variables and constant are atomic terms, terms built out of proper functors are structured terms. Then
  - $a$, $b$, $X$, and $Y$ are atomic terms
  - $f(a, b, a)$, $g(X, Y)$ and $f(a, X, g(Y, b))$ are structured terms
    - In the structured term $f(a, X, g(Y, b))$, $f$ is the functor symbol of arity 3, whereas $a$, $X$, $g(Y, b)$ are the three subterms
a *recursive* definition, leading to a recursive data structure—a *tree*
- e.g., structured term $f(a, X, g(Y, b))$ maps onto tree

```
          f
         /   \
        a     g
       /     /  \ 
      X     Y   b
```

- fundamental in mathematical logic, terms are *essential in computer science*, too: e.g., they capture both arithmetic expressions and strings
- *no specific alphabet* is assumed—*universal alphabet* for all terms
- *no meaning* is a *a priori* attached to symbols, in particular to functors—e.g., $+$ is just a functor, not associated *a priori* with the plus sign of arithmetic
- → *no types*
Basic Units of Computation V

Terms: Semantics

? if they have no predefined meaning, no type, how we to represent the application domain in a logic program?

- In principle, every term of a logic program can be associated to an entity of the *domain of discourse* through a pre-interpretation.
- It is a *conceptual mapping* from the set of all *ground terms* (terms without variables) – also called the *Herbrand Universe* – and the elements of the domain of discourse.
- E.g., even though symbol 1 is not a priori associated with its common arithmetic value, it could be explicitly pre-interpreted as such.
Variables

- they start working as non initialised—unlike many other language we know
- their value range over the set of all possible terms
- since variable are terms, and their values are terms as well, their assignments are called substitutions
Substitution

- substitutions are mapping from variables to their values (terms)
  - excluding variables mapped to themselves
- variables in a substitution are *initialised*
- substitutions represent a meaningful part of the state of a logic programming machinery
- notation

\[ \{X_1/t_1, \ldots, X_n/t_n\} \]

denotes a substitution *binding* variable \(X_i\) to term \(t_i\), for \(1 \leq i \leq n\)
substitutions can be used for evaluation. The process of evaluation is called an application of a substitution to a term. Each variable in a substitution is replaced by the corresponding term. For example, applying substitution \{X/g(a, Y), Y/b\} to term \(f(a, X, g(Y, b))\) results in the term \(f(a, g(a, b), g(b, b))\).
Equality, equations, and unifiers

- *equation* between terms is the basic operation in logic programming notation

\[ t = t' \]

denotes the equation making terms \( t \) and \( t' \) equal

- a substitution making two terms equal is called *unifier*

for instance,

- given terms \( g(a, Y) \) and \( g(X, Z) \), substitutions \( \{X/a, Y/b, Z/b\} \), \( \{X/a, Y/a, Z/a\} \), \( \{X/a, Y/Z\} \) are all unifiers for equation \( g(a, Y) = g(X, Z) \)

- given terms \( g(a, Y) \) and \( g(b, Z) \), no substitution exists that is a unifier for equation \( g(a, Y) = g(b, Z) \)
Basic Units of Computation X

Unification

- in general, in logic programming equation means unification
- intuitively, given two well-formed terms, they unify according to the following simple rules
  - two (uninstantiated) variables \( X, Y \) unify with substitution \( X/Y \)
  - two constants unify if only if they are the same constant
  - two structured terms if only if they have the same functor and arity, and their subterms recursively unify
- unification is decidable [Robinson, 1965]
- and can be computed using the efficient algorithm by Martelli & Montanari [Martelli and Montanari, 1982]
Most General Unifier (MGU)

- the *least constraining* unifier is called the *most general unifier* (MGU)
- for instance,
  - given terms $g(a, Y)$ and $g(X, Z)$, substitution $\{X/a, Y/Z\}$ is more general / less constraining than substitutions $\{X/a, Y/b, Z/b\}$, $\{X/a, Y/a, Z/a\}$
- MGU is essentially the solution to the basic equation of logic programming
Basic question

- Since logic programs compute over the truth values of sentences, how do we write sentences?
- We know how to denote the elements of the domain of discourse, not how to talk about them.
- Sentences, in logic, are typically called *propositions*.
Logic Formulae II

Predicate and atoms

- predicates can be used to write propositions in logic programming
- if $p$ is a predicate symbol of arity $n$, $t_1, \ldots, t_n$ are terms, then
  \[ p(t_1, \ldots, t_n) \]
  is an atom
- atoms represent elementary propositions in logic programming
- if $A$ is an atom, then
  \[ A \]
  is a logic formula, stating that $A$ is true
Logic Formulae III

Negation and literals

- **negation** makes it possible to deal with false propositions
- if $A$ is an atom, then
  - **negation** $\neg A$ (read: not $A$) is a logic formula, stating that $A$ is false
  - **literals** $A$, $\neg A$ are *literals*
Logical connectives

- Literals can be combined through *logical connectives* to build articulated *logic formulae*.

- If $A$, $B$ are literals, then
  - **Conjunction** $A \land B$ (read: $A$ and $B$) is a logic formula, stating that both $A$ and $B$ are true.
  - **Disjunction** $A \lor B$ (read: $A$ or $B$) is a logic formula, stating that either $A$ or $B$ are true.
  - **Implication** $A \rightarrow B$ (read: $A$ implies $B$) is a logic formula, stating that if $A$ is true then $B$ is true.
  - **Equivalence** $A \leftrightarrow B$ (read: $A$ is equivalent to $B$) is a logic formula, stating that $A$ is true if and only if $B$ is true.
Logic Programs I

Logic clause

- A **logic clause** is a (finite) disjunction of literals [Console et al., 1997]
- If $A_1, \ldots, A_n, B_1 \ldots, B_m$ are atoms, containing variables $X_1, \ldots, X_k$, then

$$\forall X_1, \ldots, X_k (A_1 \lor \ldots \lor A_n \lor \neg B_1 \lor \ldots \lor \neg B_m)$$

is a logic clause, which is *logically equivalent* to

$$\forall X_1, \ldots, X_k ((A_1 \lor \ldots \lor A_n) \leftarrow (B_1 \land \ldots \land B_m))$$

usually written simply as

$$A_1, \ldots, A_n \leftarrow B_1, \ldots, B_m$$

- A **clausal normal form** (CNF) is a *conjunction of clauses*
Logic Programs II

Definite clauses

- A definite clause, has just one positive literal \( (n = 1) \)
  \[ A \leftarrow B_1, \ldots, B_m \]

- A unitary clause, is a definite clause with no negative literal \( (m = 0, n = 1) \)
  \[ A \leftarrow \]

- A definite goal is a definite clause with no positive literal \( (n = 0) \)
  \[ \leftarrow B_1, \ldots, B_m \]

Horn clauses

- A Horn clause is either a definite clause or a definite goal \( (n = 1 \text{ or } n = 0) \)
Logic Programs III

Logic program

- in a logic program
  - a definite clause is called a rule
  - a unitary clause is a fact
  - a definite goal is just a goal

- a logic program is a CNF of Horn clauses
  - so, it is a conjunction of rules and facts (and goals)

... a logic program is a conjunction of Horn clauses... waitbutwhy???
**Resolution principle**

- Robinson’s resolution principle works for general clauses [Robinson, 1965]
  - given a CNF $H$ and a formula $F$, it shows that it is possible to compute (by contradiction) whether $H$ logically entails $F$
  - however, it does not provide a proof strategy for a full-fledged logic programming language
- Kowalski showed that this could be obtained by restricting logic programs to CNF of *Horn clauses*, and re-casting Robinson’s principle accordingly [Kowalski, 1974]
  - given a CNF $H$ and a formula $F$, it shows that it is possible to compute (by contradiction) whether $H$ logically entails $F$
  - so-called SLD-resolution principle [Nilsson and Maluszynski, 1995]
Declarative vs. procedural interpretation

- a definite clause $A \leftarrow B_1, \ldots, B_m$ is amenable of either a *declarative* or a *procedural interpretation*

  **declarative interpretation**  $A$ is true if $B_1, \ldots, B_m$ are true

  **procedural interpretation**  to prove $A$, prove $B_1, \ldots, B_m$

- the two interpretations coincide [Kowalski, 1974]

- logic programming languages such as Prolog are the only ones for which this property holds [Metakides and Nerode, 1996b]
Robinson’s principle proceed by contradiction, trying to prove a formula $F$ false against CNF $H$, succeeding if this fails.
- technically, proving that $H \cup \neg F$ is not satisfiable.

proving an atom $G$ in logic programming amounts at proving $\neg G$ against logic program $P$.
- technically, proving goal $\leftarrow G$ on $P$.

computation in logic programming proceeds by **proving goals**.
- resolution leads to *backward chaining*—from goal back to axioms.
SLD resolution informally

- to prove a goal $G$ w.r.t. program $P$, the resolution principle for logic programming proceeds according to the procedural interpretation.
- so, first we look for one clause $A ← B_1, \ldots, B_n$ in $P$ whose head $A$ unifies with $G$.
- if the most general unifier of $G$ and $A$ is $\theta$ ($\text{mgu}(G, A) = \theta$), then the proof of $G$ succeeds if we can further prove $B_1\theta, \ldots, B_n\theta$—where $B_i\theta$ represents the application of the mgu $\theta$ to $B_i$.

  ! the application of $\theta$ to clause $A ← B_1, \ldots, B_n$ specialises the clause to the specific atom we need to proof—that is, our current goal.

  ! resolution proceed recursively with the proof of subgoals $B_1\theta, \ldots, B_n\theta$.

→ in general, the computational state of the SLD resolution include a (possibly empty) conjunction of atom (goals) $G_1, \ldots, G_n$ to be proven—the current goal of the proof.
SLD Resolution: how it ends—if it does

- when the current goal is empty, the proof (called *SLD derivation*) ends as a *successful* one—SLD refutation
- when the current goal is not empty, a *selection rule* $R$ is used to select the subgoal to prove (one if the execution is sequential)
- if the selected goal matches no head of the clauses in the program, the proof *fails*
- if the current goal never gets emptied, but there is always a clause whose head matches the selected subgoal, the SLD derivation *does not terminate*
SLD resolution: inference rule

\[
\begin{align*}
\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m & \quad B_0 \leftarrow B_1, \ldots, , B_n \quad (A_1, \ldots, A_{i-1}, B_1, \ldots, , B_n, A_{i+1}, \ldots, A_m)\theta
\end{align*}
\]

- \( A_1, \ldots, A_m \) are atomic formulas
  - \( \leftarrow A_1, \ldots, A_m \) is the list / set / conjunction of the subgoals to prove
- \( B_0 \leftarrow B_1, \ldots, , B_n \) is a definite clause in program \( P \) \((n \geq 0)\)
  - suitably renamed (that is, with new and uniques variable names) to avoid name clashes
- there is an \( A_i \) unifying with \( B_0 \) such that \( mgu(A_i, B_0) = \theta \)
Non-determinism of SLD resolution

- More than one clause could unify (through its head) with our current goal: we could choose either one of them for the resolution step.

- More than one goal could be subject to proof at the same time (as for $B_1\theta, \ldots, B_n\theta$): we could proceed by choosing either one of them—through a selection rule.

- The choice does not affect correctness of the resolution, so we could choose non-deterministically.

- How to exploit either or-nondeterminism or and-nondeterminism, or both, determines how the automatic resolution process explores the proof tree.

- Also, different computational models (sequential, parallel, concurrent) could be exploited to explore the proof tree—e.g., more clauses with a unifying head could be used for goal proof at the same time, either parallel or concurrently.
A simple logic program

\[
\begin{align*}
\text{parent}(\text{joey}, \text{luca}) \\
\text{parent}(\text{joey}, \text{simone}) \\
\text{parent}(\text{lino}, \text{joey}) \\
\text{parent}(\text{mirella}, \text{joey}) \\
\text{grandparent}(X, Z) \leftarrow \text{parent}(X, Y), \text{parent}(Y, Z)
\end{align*}
\]
An Example II

Declarative interpretation

- four facts are expressed by means of predicate parent/2
  - four propositions that are considered true with no need of proof—our axioms
  - a possible interpretation is that, e.g., joey is a parent of luca—just one of the many, even though the most intuitive for English speakers
- one rule is expressed by means of predicate grandparent/2
  - since it is the short form for
    \[ \forall X, Y, Z, \text{grandparent}(X, Z) \leftarrow \text{parent}(X, Y), \text{parent}(Y, Z) \]
    it means that formula grandparent(X, Z) holds if both parent(X, Y) and parent(Y, Z) are true, whatever the values of X, Y, Z
  - so, it can be used to prove the truth of, e.g., formula grandparent(lino, luca) since both parent(joey, luca) and parent(lino, joey) are true since they are facts in the logic program
  - independently of the possible interpretations
An Example III

Procedural interpretation

- Two procedure are defined: `parent/2` and `grandparent/2`
- Two (procedure) calls can be executed correspondingly—goals of the form:
  - `← parent(? , ?)`
  - `← grandparent(? , ?)`

  With any sort of term in the place of the `?`
  - For instance, `← grandparent(lino, luca)`

  To compute `parent/2` we can use the four facts, non-deterministically.

  To compute `grandparent/2` we can use the rule, first matching the rule head, then proceeding by calling the two subprocedures, via the two subgoals of the form `parent/2`
  - For instance, to compute `← grandparent(lino, luca)` we will compute subgoals `← parent(lino, Y)` and `← parent(Y, luca)`.\)
An Example IV

Possible goals

- `grandparent(lino, luca)` succeeds—one refutation, no *computed substitution*
- `grandparent(lino, joey)` fails—no refutations
- `grandparent(lino, X)` succeeds twice—two refutations, two different computed substitutions
  - `X/luca`
  - `X/simone`
- `grandparent(X, simone)` succeeds twice—two refutations, two different computed substitutions
  - `X/lino`
  - `X/mirella`
- `grandparent(X, Y)` succeeds four times—four refutations, four different computed substitutions
  - `X/lino, Y/luca`
  - `X/lino, Y/simone`
  - `X/mirella, Y/luca`
  - `X/mirella, Y/simone`

Remarks

From the example we get some early hints about some benefits of logic programming:

- **Multiple uses of the single program**
  - The simple program above can be used to test the family relations between known people, or, to compute them.
- **Input / output parameters**
  - Needs not to be defined a priori.
- Knowledge-based programming
  - Arbitrarily complex relations expressed as FOL facts represent the core of a logic program.
- Knowledge representation is straightforward in the logic programming formalism—with FOL language for rule-based systems.
Logic & Logic Programming: Overall Picture

- Logic axioms
  - No interpretation
  - Interpretation
- Syntax (symbols)
- Semantics (meanings)
- Theorem proof
- Logic entailment
- Soundness
- Completeness

Axiomatic-deductive theories
Automatic theorem proof

Resolution for general clauses
SLD resolution for Horn clauses

Logic programming
Why Logic Languages?

- since the early days of logic programming, features like
  - high-level languages
  - non-determinism
  - referential transparency
  made the potential for exploitation of parallelism in the execution of logic programs clearly emerge [Gupta et al., 2001]
- at the same time, the articulation of logic computation, with the frequent occurrence of
  - heavy computational load
  - non-trivial computational structures
  made the computational techniques developed for logic languages relevant for the general scenario of concurrent / parallel computation
Why Prolog?

Or- & and-parallelism

- the first reason why Prolog is of interest to distributed systems is that its search mechanism allows for parallel evaluation
  - the proof tree could be explored parallel / concurrently
  - even more, the potential computational complexity of the proof tree actually encourages parallel exploration
- basically, Prolog supports two sorts of parallelism
  - and-parallelism
  - or-parallelism

that suit well for distributed computing
the first source of parallelism in Prolog comes from the fact that the proof can adopt any matching clauses in the current logic theory

exploiting don’t-know non-determinism

e.g., when a subgoal $G$ matches both $A_1$ and $A_2$, and two clauses

- $A_1 \leftarrow B_{1,1}, \ldots, B_{1,m_1}$
- $A_2 \leftarrow B_{2,1}, \ldots, B_{2,m_2}$

belongs to the Prolog theory, then both can be used to compute a possible demonstration at the same time

in general, multiple clauses whose head matches the current subgoal could be used for proof simultaneously

exploration of the proof tree could then proceed in parallel
Implicit parallelism

- in principle, a standard logic program can be executed in parallel by exploiting or-parallelism
- no need for special *explicit* programming constructs for parallel computation
- basically, as another consequence of the Kowalski’s Principle

\[
	ext{Programs} = \text{Logic} + \text{Control} \quad \text{[Kowalski, 1979]}
\]

Example

- *Aurora* [Lusk et al., 1990] is an or-parallel implementation of the full Prolog language
- aimed at shared-memory multiprocessors
- based on SICStus Prolog [http://sicstus.sics.se]
Or-Parallelism III

Issues

failure essentially refers to any parallel stream of computation, global failure requires coordination

backtracking loses some meaning as a mechanism for sequential exploration of the proof tree
Solving more subgoals at a time

- the second source of parallelism in LP comes from the fact that the body of a Horn clause is a *conjunction* of atoms
- e.g., when a two subgoals $G_1$ and $G_2$ have to be solved, they could be in principle be solved in parallel rather than sequentially
  - thus generating two (independent?) computations
- in general, any *conjunction of goals* could be potentially demonstrated *simultaneously*
  - if *independent* from each other
  - which basically means, if they do not share variables
And-Parallelism II

Issues

**Independence** it is not necessarily easy to spot out whether a subset of the subgoals in the current goal are independent of each other.

**Detection** who is going to detect independency, the programmer or the compiler?

**Constructs** *implicit vs. explicit* and-parallelism [Gupta et al., 2001]

**Shared Variables** as communication channels among parallelly-processed subgoals
Example

- Concurrent Prolog [Shapiro, 1989] uses shared variables for communication / synchronisation.
- Shared variables work as the communication / synchronisation channel between concurrent processes.

While still in the field of distributed computing, those approaches start moving logic languages towards distributed systems.
Concurrent logic processes

- in concurrent systems, processes can be represented by logic processes
  - logic agents in the acceptance of coordination models [Ciancarini, 1996]
- there, concurrent / distributed systems are first of all collection of logic engines running in a concurrent / distributed way
  [Robertson, 2004, Brogi and Ciancarini, 1991]
- finally, moving LP towards distributed systems
Logic-based Coordination

Logic tuple spaces

- the space of interaction is built around \textit{logic tuple spaces}
  
  [Ciancarini, 1994, Denti et al., 1996]

- allowing for heterogeneous distributed processes to be coordinated

- blackboards and programmable logic tuple spaces promote logic-based coordination of distributed systems

  [Brogi and Ciancarini, 1991, Omicini and Denti, 2001]
Distributed logic engines

- IoT and CPS scenarios mandates for distributed & situated *micro-intelligence*
  - huge numbers of small unit of computation
  - situated within a spatially-distributed environment
  - promoting the local exploitation of high-level symbolic languages
  - with inferential capabilities

- distributed Prolog engines in pervasive scenarios [Denti et al., 2013]

- Logic Programming as a Service (LPaaS) could promote Prolog integration within standard distributed systems [Calegari et al., 2016]

- a *logic-based middleware* is nowadays a feasible technology target
Summary

- logic languages like Prolog are a perfect fit for *distributed computing* techniques
- logic-based *distributed system* can be built by exploiting logic languages for *distributed components*
  - processes
  - agents
  - logic engines
  
as well as for their *coordination*


References

The birth of Prolog.

Programmazione logica e Prolog.
UTET Libreria.

Logic tuple spaces for the coordination of heterogeneous agents.

tuProlog: Making Prolog ubiquitous.
ALP Newsletter.

On the Foundations of Geometry and Formal Theories of Arithmetic.
Yale University Press.


References VI


Multi-agent coordination as Distributed Logic Programming.

A machine-oriented logic based on the resolution principle.

The family of concurrent logic programming languages.

Introduction to Logic and to the Methodology of the Deductive Sciences, volume 24 of Oxford Logic Guides.
Oxford University Press.

Principia Mathematica.
Cambridge University Press.