Logic & Computation

Digital Culture

Andrea Omicini
andrea.omicini@unibo.it

Master in Business Management / Bologna Business School

Academic Year 2017/2018

1. Logic
2. Programming with Logic
3. Prolog

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

Logic & Math I

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]
  - crediting Ramon Llull's *Ars Generalis* (1308) as an inspiration
- 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]

Logic & Math II

The rise of mathematical logic

- starting form 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*

Truth I

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

Truth II

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

Truth III

Interpretation and logic entailment

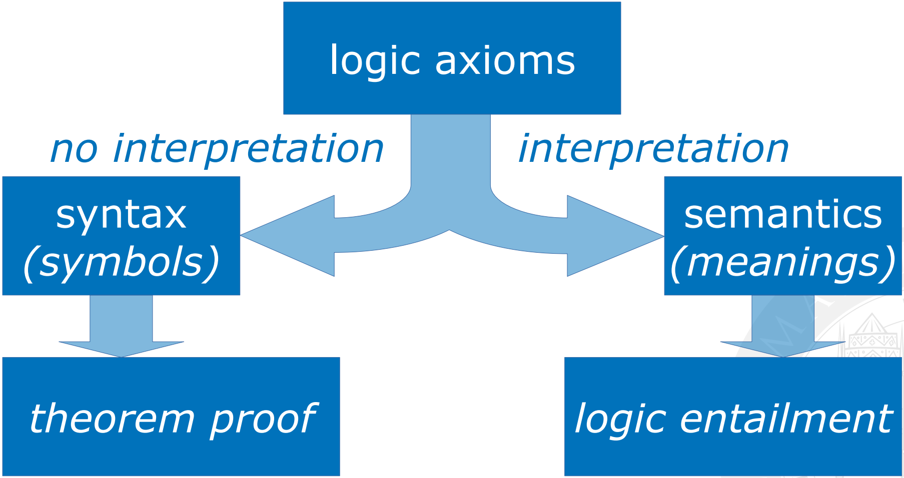
- Alfred Tarski [Tarski and Tarski, 1994] formalised the notion of logical interpretation as the relation between the *symbols* (syntax) and the elements of the *domain of discourse* (semantics)
- he defined logical entailment (or, *logical consequence*) precisely: when a *conclusion* is true in every interpretation making all *premises* true, then premises logically entail the conclusion—or, the conclusion is a logical consequence of the premises

Truth IV

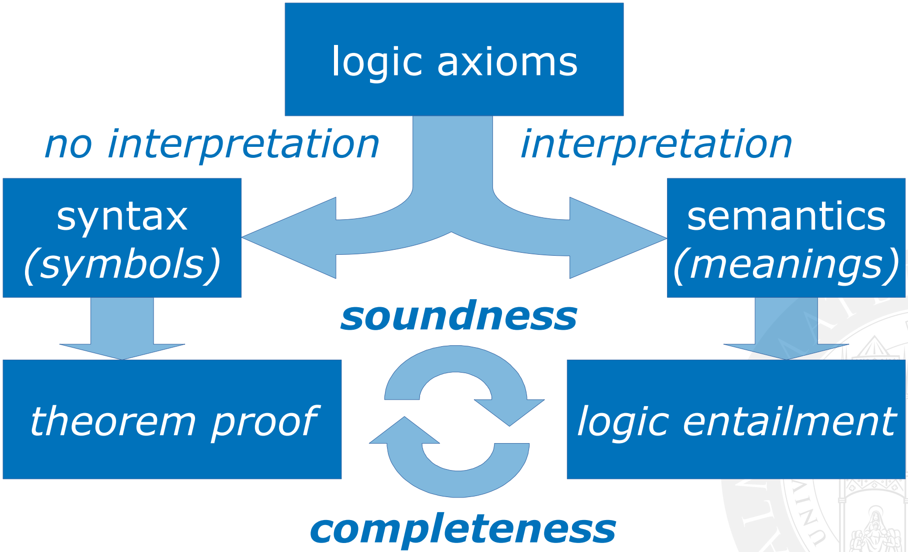
Interpretation and logic entailment

- 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. For example, we conjecture that* $a^n + b^n \neq c^n$ *for* $n > 2$
- *So computation is mechanical and requires no ingenuity, while deduction is a creative process. For example, proving that* $a^n + b^n \neq c^n$ *for* $n > 2$ *basically took 357 years of hard creative work.*

Computation vs. Deduction II

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]

Computation vs. Deduction III

Yet, somehow *computation* $\rightleftharpoons$ *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.

Deduction I

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*

Deduction II

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 ... and J_n, then we can conclude J by virtue of rule R
  - given J_1 and ... and J_n, we can *prove* J by rule R
  - J can be *deduced*/*inferred* from J_1 and ... 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

$$\frac{J_1, \ldots, J_n}{J} R$$

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 I

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

$$\dfrac{\dfrac{\dfrac{\dots}{J_{1,1}}R_{1,1},\ \dots,\ \dfrac{\dots}{J_{1,n_1}}R_{1,n_1}}{J_1}R_1 \ \dots\ \dfrac{\dfrac{\dots}{J_{n,1}}R_{n,1},\ \dots,\ \dfrac{\dots}{J_{n,n_n}}R_{n,n_n}}{J_n}R_n}{J}R$$

- J is the root of the tree
- $J_1, \dots J_n$ are *nodes* of the tree

Proof II

- and so are $J_{1,1}$, ...
- R, $R_1, \ldots, R_n$ are *edges* of the tree
  - and so are $R_{1,1}$, ...
- $J_1, \ldots J_n$ 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, ...

Axioms

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.*

Proof Search

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

$$\dfrac{\dfrac{\dfrac{\dots}{J_{1,1}}R_{1,1},\ \dots,\ \dfrac{\dots}{J_{1,n_1}}R_{1,n_1}}{J_1}R_1 \ \dots\ \dfrac{\dfrac{\dots}{J_{n,1}}R_{n,1},\ \dots,\ \dfrac{\dots}{J_{n,n_n}}R_{n,n_n}}{J_n}R_n}{J}R$$

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

$$\cfrac{\cfrac{\cfrac{\cdots}{J_{1,1}}R_{1,1},\ldots,\cfrac{\cdots}{J_{1,n_1}}R_{1,n_1}}{J_1}R_1 \;\ldots\; \cfrac{\cfrac{\cdots}{J_{n,1}}R_{n,1},\ldots,\cfrac{\cdots}{J_{n,n_n}}R_{n,n_n}}{J_n}R_n}{J}R$$

Proof Search & Computation I

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

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

Origins II

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]

Essentials I

Three fundamental features [Apt, 2005]

terms *Computing takes place over the domain of all terms defined over a "universal" alphabet.*

mgu *Values are assigned to variables by means of automatically-generated substitutions, called most general unifiers. These values may contain variables, called logical variables.*

backtracking *The control is provided by a single mechanism: automatic backtracking.*

Other Features I

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]

Other Features II

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

Other Features III

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., 2017]

Basic Units of Computation I

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

Basic Units of Computation II

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

Basic Units of Computation III

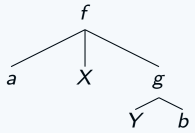
Terms: Examples

- 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*

Basic Units of Computation IV

Terms: Remarks

- a *recursive* definition, leading to a recursive data structure—a tree
  - e.g., structured term $f(a, X, g(Y, b))$ maps onto tree



- 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

$\rightarrow$ 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

Basic Units of Computation VI

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

Basic Units of Computation VII

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 $1 \leq i \leq n$

Basic Units of Computation VIII

Evaluation

- 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
  - e.g., applying substitution $\{X/g(a, Y), Y/b\}$ to term $f(a, X, g(Y, b))$ results in the term $f(a, g(a, b), g(b, b))$

Basic Units of Computation IX

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]

Basic Units of Computation XI

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

Logic Formulae I

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

  atoms 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*

Logic Formulae IV

Logical connectives

- literals can be combined through *logical connectives* to build articulated *logic formulae*
- if A, B are literals, then

conjunction $A \wedge B$ (read: A and B) is a logic formula, stating that both A and B are true

disjunction $A \vee B$ (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

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

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

$$\forall X_1, \ldots, X_k ((A_1 \vee \ldots \vee A_n) \leftarrow (B_1 \wedge \ldots \wedge B_m))$$

  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 ($n = 1$ 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*???

Goals & Proofs I

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]

Goals & Proofs II

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]

Goals & Proofs III

Proving goals

- 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

Goals & Proofs IV

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 $A \leftarrow B_1, \ldots, B_n$ in P whose head A unifies with G
- if the most general unifier of G and A is θ ($mgu(G, A) = \theta$), 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* θ to B_i

! the application of θ to clause $A \leftarrow B_1, \ldots, B_n$ 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

Goals & Proofs V

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 $\mathbb{R}$ 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*

Goals & Proofs VI

SLD resolution: inference rule

$$\frac{\leftarrow A_1, \ldots, A_{i-1}, A_i, A_{i+1}, \ldots, A_m \qquad B_0 \leftarrow B_1, \ldots, , B_n}{\leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, , B_n, A_{i+1}, \ldots, A_m)\theta}$$

- $A_1, \ldots, A_m$ 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 $mgu(A_i, B_0) = \theta$

Goals & Proofs VII

Non-determinism of SLD resolution

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

and 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 do 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

An Example I

A simple logic program

$parent(joey, luca)$
$parent(joey, simone)$
$parent(lino, joey)$
$parent(mirella, joey)$

$grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$

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

    $$\forall X, Y, Z, grandparent(X, Z) \leftarrow parent(X, Y), parent(Y, Z)$$

    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
  - $\leftarrow parent(?,?)$
  - $\leftarrow grandparent(?,?)$

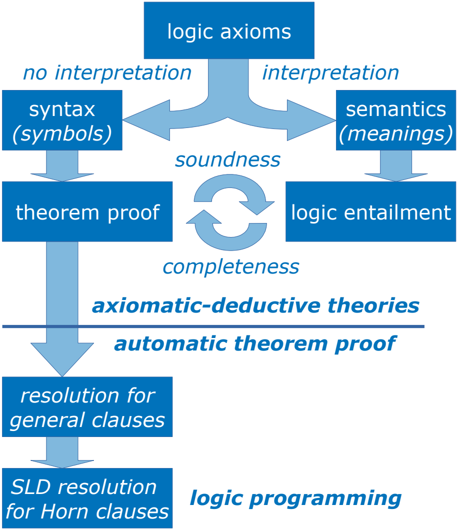
  with any sort of term in the place of the ?
  - for instance, $\leftarrow grandparent(lino, luca)$
- to compute $parent/2$ we can use the four facts, non-deterministically
- to compute $grandparent/2$ we can use the rule, first matching the rule head, then proceeding by calling the two subprocedures, via the two *subgoals* of the form $parent/2$
  - for instance, to compute $\leftarrow grandparent(lino, luca)$ we will compute subgoals $\leftarrow parent(lino, Y)$ and $\leftarrow parent(Y, luca)$

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
  - $X/lino$, $Y/luca$
  - $X/lino$, $Y/simone$
  - $X/mirella$, $Y/luca$
  - $X/mirella$, $Y/simone$

Logic & Logic Programming: Overall Picture



Next in Line...

Prolog Syntax I

Prolog terms

variables alphanumeric strings starting with either an *uppercase* letter or an *underscore*

- underscore alone is the *anonymous variable*—sort of *don't care* variable
- underscore followed by a string is a normal variable during resolution, but it does not need to be exposed in the computed substitution

functors alphanumeric strings starting with a *lowercase* letter

- holds for both proper functors and constants

terms are built recursively out of functors and variables as in logic programming

→ e.g., `term`, `Var`, `f(X)`, `p(Y,f(a))` are *Prolog terms*

→ e.g., `term`, `var`, `f(a)`, `p(x,y)` are *Prolog ground terms*

Prolog Syntax II

Prolog atoms

predicates alphanumeric strings starting with a *lowercase* letter

- the same as functors

atoms are built applying predicates to terms as in logic programming

→ e.g., `predicate`, `f(X)`, `p(Y,f(a))` are *Prolog atoms*

→ e.g., `predicate`, `f(a)`, `p(x,y)` are *Prolog ground atoms*

Towards meta-programmings

- `parent(lino,joey)` – out of context – could represent either a ground atom or a ground term
- is this an issue or a feature?
- it paves the way towards *meta-programming*

Prolog Syntax III

Prolog clauses

clause a Horn clause of the form `A :- B1, ..., Bn.`

- where `A`, `B1, ..., Bn` are Prolog atoms
- `A` is the head of the clause
- `B1, ..., Bn` is the body of the clause
- `:-` denotes logic implication
- `.` is the terminator

fact a clause with no body `A.` ($n = 0$)

rule a clause with at least one atom in the body
`A :- B1, ..., Bn.` ($n > 0$)

goal a clause with no head and at least one atom in the body
`:- B1, ..., Bn.` ($n > 0$)

- often written as `?- B1, ..., Bn.`

Prolog Syntax IV

Prolog program

program: a sequence of Prolog clauses

interpreted as a *conjunction* of clauses

logic theory: constituting a *logic theory* made of Horn clauses written according the Prolog syntax

Prolog Execution I

Aim of a Prolog computation

- given a Prolog program P and the goal `?- p(t1,t2,...,tm)` (also called *query*)
- if `X1,X2,...,Xn` are the variables in terms `t1,t2,...,tm`
- the meaning of the goal is to query P and find whether there are some values for `X1,X2,...,Xn` that make `p(t1,t2,...,tm)` true

→ thus, the aim of the Prolog computation is to find a substitution $\sigma = \mathtt{X1/s1,X2/s2,...,Xn/sn}$ such that $P \models \mathtt{p}(\mathtt{t1}, \mathtt{t2}, \ldots, \mathtt{tm})\sigma$

Prolog Execution II

Prolog search strategy

- as a logic programming language, Prolog adopts SLD resolution
- as a search strategy, Prolog applies resolution in a strictly linear fashion
  - *goals* are replaced *left-to-right*, sequentially
  - *clauses* are considered in *top-to-bottom* order
  - *subgoals* are considered *immediately* once set up

→ resulting in a *depth-first* search strategy

Prolog Execution III

Prolog backtracking

- in order to achieve completeness, Prolog saves choicepoints for any possible alternative still to be explored
- and goes back to the nearest choice point available in case of failure
- exploiting automatic backtracking

Focus on. . .

Example 1: The parent.pl Knowledge Base I

```
parent(joey,luca).
parent(joey,simone).
parent(lino,joey).
parent(mirella,joey).
```

A logic theory

- a simple logic *program*
- with four *ground facts*
- representing one sort of relation between elements of the *domain of discourse*

? is there anything we can do with this program?

?? can we *compute* anything?

Example 1: The `parent.pl` Knowledge Base II

Constants & predicates

- `joey`, `luca`, `simone`, `lino` and `mirella` are *constant* used in the program as *ground terms* to denote the element of the domain
- `parent` is the *predicate* used in the program to talk about the domain of discourse—`parent/2` says that `parent` is the *predicate symbol* with *arity* 2

Example 1: The `parent.pl` Knowledge Base III

Goals

- since the only predicate in the program is `parent/2`, we cannot prove anything else, in principle—except for tautologies, or built-in Prolog predicates
- possible goals
  1. `:- parent(joey,luca).`
  2. `:- parent(joey,lino).`
  3. `:- parent(joey,X).`
  4. `:- parent(X,joey).`
  5. `:- parent(X,Y).`

Let us try the above *queries* in tuProlog

tuProlog in Short I

What is tuProlog?

- tuProlog is a light-weight Prolog system for distributed applications and infrastructures [Denti et al., 2001]
- intentionally designed around a minimal core
- to be either statically or dynamically *configured* by loading/unloading libraries of predicates
- tuProlog natively supports multi-paradigm programming [Denti et al., 2005], providing a clean, seamless integration model between Prolog and mainstream object-oriented languages

tuProlog in Short II

Where is tuProlog?

UniBo `http://tuprolog.unibo.it`

BitBucket `http://bitbucket.org/tuprologteam/tuprolog`

downloads `http://bitbucket.org/tuprologteam/tuprolog/downloads`

tuProlog in Short III

What to download?

From `http://bitbucket.org/tuprologteam/tuprolog/downloads`

v. 3.2.2.0 `http://bitbucket.org/tuprologteam/tuprolog/downloads/2p-3.2.2.0.zip`

guide `http://bitbucket.org/tuprologteam/tuprolog/downloads/tuprolog-guide-3.2.pdf`

Let us try the above *queries* in tuProlog

Prolog Interactive Programming in tuProlog I

Starting tuProlog CLI

- after downloading the tuProlog ZIP file, unzip it, and position your command line in the 2p-3.2.2.0/bin/ directory
- then invoke the tuProlog command line interface (CLI) with

```
java -cp 2p.jar alice.tuprologx.ide.CUIConsole
```

- to see if everything is working, invoke query

```
?- write('Hello, World!').
```

  and see what happens

Prolog Interactive Programming in tuProlog II

- first, load a simple program with the query

```
?- set_theory("parent(joey,luca). parent(joey,simone).
               parent(lino,joey). parent(mirella,joey).").
```

- then try the following queries:

1. `?- parent(joey,luca).`
2. `?- parent(joey,lino).`
3. `?- parent(joey,X).`
4. `?- parent(X,joey).`
5. `?- parent(X,Y).`

  and see what happens, by responding with

  - ⟨return⟩ to accept the answer
  - ; to have more answers

Prolog Interactive Programming in tuProlog III

Remarks on interaction

1. success
2. failure
3. computed substitution
4. unification
5. backtracking

all no input / output parameters: no *direction* required for arguments in principle thanks to unification

tuProlog GUI

Starting tuProlog GUI

- in the same `2p-3.2.2.0/bin/` directory as before, invoke the tuProlog graphical user interface (GUI) with

  ```
  java -cp 2p.jar alice.tuprologx.ide.GUILauncher
  ```

- to see if everything is working, invoke query

  ```
  ?- write('Hello, World!').
  ```

  and see what happens
- redo `parent.pl` experiments with GUI
  - load `parent.pl`
  - retry the queries discussed in the CLI experiments

Example 2: grandparent.pl I

Adding a rule

- load grandparent.pl
- there, rule

  ```
  grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
  ```

  is added to the knowledge base
- now the logic program is a collection of *facts* and *rules*

! it is a so-called *universal rule*

Example 2: `grandparent.pl` II

Test the program

- now try the following queries:
  1. `?- grandparent(lino,luca).`
  2. `?- grandparent(lino,joey).`
  3. `?- grandparent(joey,X).`
  4. `?- grandparent(lino,X).`
  5. `?- grandparent(X,Y).`

  and discuss all the results

Example 3: sibling.pl I

Adding another rule

- load `sibling.pl`
- there, rule

  `sibling(Y,Z) :- parent(X,Y), parent(X,Z).`

  is added to the previous logic theory
- all the previous theorems are true: all previous computations are the same
- just adding new theorems based on a new rule

! operator `\=/2` represent an *explicit* computation over terms
  - succeeding when the two arguments are terms that do not unify
  - all other computations over terms till now were *implicitly* driven by goal unification

Example 3: `sibling.pl` II

Test the program

- now try the following queries:
  1. `?- sibling(simone,luca).`
  2. `?- sibling(lino,joey).`
  3. `?- sibling(luca,X).`
  4. `?- sibling(lino,X).`
  5. `?- sibling(X,Y).`

  and discuss all the results

Focus on. . .

Types in LP

No types in LP

- in pure LP 3 and + are just symbols, with no specific meaning attached
- and, in *pure Prolog* as well

? how do we deal, e.g., with natural numbers?

! Giuseppe Peano shows us the way

Natural Numbers in Logic I

Peano axioms

http://www.britannica.com/topic/Peano-axioms

> *Peano axioms, also known as Peano's postulates, in number theory, five axioms introduced in 1889 by Italian mathematician Giuseppe Peano. Like the axioms for geometry devised by Greek mathematician Euclid (c. 300 bce), the Peano axioms were meant to provide a rigorous foundation for the natural numbers (0, 1, 2, 3, ...) used in arithmetic, number theory, and set theory. In particular, the Peano axioms enable an infinite set to be generated by a finite set of symbols and rules.*

Natural Numbers in Logic II

The five Peano axioms are

1. zero is a natural number
2. every natural number has a successor in the natural numbers
3. zero is not the successor of any natural number
4. if the successor of two natural numbers is the same, then the two original numbers are the same
5. if a set contains zero and the successor of every number is in the set, then the set contains the natural numbers

Natural Numbers in LP I

How do we represent natural numbers in LP?

- in LP
  - the *Herbrand Universe* represents the elements of the domain of discourse
  - the *Herbrand Base* represents the elementary propositions over the domain of discourse

$\rightarrow$ in order to represent natural numbers in LP, we use
  - *constants* and *functors* for natural numbers
  - *predicates* for relations such as $n \in \mathbb{N}$

Natural Numbers in LP II

Constants & functors

- our domain of discourse is $\mathbb{N}$, the set of *natural numbers*
- our Herbrand Universe needs to express the elements of $\mathbb{N}$
- from Peano axioms, we understand that
  - zero is essential
    - `z` we use `z` as the *constant* symbol for *zero*
    - ! it is not something belonging to the language: it is just our *(pre-)interpretation* of symbol `z` for our current purposes
  - the notion of successor is essential, too
    - `s/1` we use `s/1` (`s` with arity 1) as the *functor* symbol for *successor*
    - → if `N` is a natural number, then `s(N)` is another natural number—precisely, the *successor* of `N`
    - *recursive data structure*: the successor of `s(N)` is `s(s(N))`, which is another natural number

e.g. in our pre-interpretation, `z` is zero, `s(z)` is one, `s(s(z))` is two, ...

Natural Numbers in LP III

Predicates

- we focus on the simple relation $n \in \mathbb{N}$—that is, n is a natural number

`nat/1` we use `nat/1` (`nat` with arity 1) as the *predicate* symbol for such a relation
  - if `N` is a variable, atom `nat(N)` represent the proposition "`N` is a natural number"
  - according to our current *interpretation*

e.g. in our interpretation, atom `nat(s(s(z)))` means that 'two is a natural number'

Natural Numbers in Prolog I

A first program

```
nat(z).          % zero is a natural number
nat(s(z)).       % one is a natural number
nat(s(s(z))).    % two is a natural number
...
```

- remarks
  - infinite number of clauses (facts) for infinite numbers
  - ! not exactly in the spirit of axiomatic systems
  - in particular, of Peano's one
  - we just used *facts*
  - *rules* are what triggers *intensional representation*, with universal variables
  - *finite* representation for (potentially) *infinite* propositions

Natural Numbers in Prolog II

A second program: recursive rules

```
nat(z).                   % zero is a natural number
nat(s(N)) :- nat(N).      % if N is a natural number
                          % its successor is a natural number
```

- remarks
  - one fact and one rule for infinite numbers
  - a *recursive* rule
- goals
  1. `?- nat(z).`
  2. `?- nat(0).`
  3. `?- nat(s(s(s(s(s(s(z))))))).`
  4. `?- nat(N).`
  5. `?- nat(s(N)).`

Natural Numbers in Prolog III

- remarks
  - z is zero, 0 is not zero
  - recursive data structures are powerful and expressive, yet unreadable for humans: which is why Prolog is actually *impure* LP
  - LP allows for *generation* of data—-e.g., the set of natural numbers, or the set of positive numbers
  - *unlimited* number of SLD *refutations*: infinite branch of the proof tree
  - *order of clauses* matters in Prolog: what if we exchange the fact with the rule?

Computing with Naturals in Prolog I

Sum

```
plus(z, N, N).            % zero plus N is N
plus(s(M), N, s(P))       % if M plus N is P, then
   :- plus(M, N, P).      % adding N to the successor of M
                          % returns the successor of P
```

- remarks
  - based on the same representation of $\mathbb{N}$ used above
  - `plus/3` actually represents an equation: `plus(X,Y,Z)` means that $X + Y = Z$

Computing with Naturals in Prolog II

- goals
  1. `?- plus(s(z),s(s(z)),S).`
  2. `?- plus(0,s(0),S).`
  3. `?- plus(X,s(s(z)),s(s(s(s(s(s(z))))))).`
  4. `?- plus(s(s(s(s(z)))),Y,s(s(s(s(s(s(z))))))).`
  5. `?- plus(X,Y,s(s(s(s(s(s(z)))))))`
  6. `?- plus(X,s(s(z)),Y), plus(X,Y,s(s(s(s(s(s(z))))))).`

Computing with Naturals in Prolog III

- remarks
  - again, `z` is zero, `0` is not zero
  - no predefined input/output parameters for `plus/3` procedure
  - → atoms represent equations
  - generation of possible solutions
  - equations can be combined in systems of equations

Focus on. . .

1. Logic
2. Programming with Logic
3. Prolog
   - Knowledge Representation & Inference
   - Numbers & Recursion
   - Lists

Lists in Prolog I

Lists

- list are defined via two constructors

  nil the empty list, containing no elements

  cons the constructor *cons*, taking an element H and a list T, and generating the list $cons(H, T)$

e.g. $cons(a, cons(b, cons(c, nil)))$ would represent list a, b, c

- typical *recursive data structures*
- used to represent *sequences* of any sort

Lists in Prolog II

Prolog lists

- in Prolog, list are defined via two analogous constructors
  - `[]` represents the empty list, containing no elements—a *constant*
  - `.` stands for *cons*, taking an element `H` and a list `T`, and generating the list `.(H,T)`—a *functor* of arity 2
- Prolog *sequence notation* simplifies writing lists
  - `.(H,T)` can be written as `[H|T]`
  - `.(H,.(H',T'))` can be written as `[H,H'|T']`
  - there, empty list can be omitted

e.g. `[a,b,c]` would represent list a, b, c in Prolog, where

- a is the head of the list
- `[b,c]` is the tail of the list
- $mgu([\mathtt{a},\mathtt{b},\mathtt{c}],[\mathtt{H}|\mathtt{T}]) = \{\mathtt{H}/\mathtt{a},\mathtt{T}/[\mathtt{b},\mathtt{c}]\}$

Computing with Lists I

Recursion

- being recursive data structures, lists are typically handled by recursive rules
- which incidentally is also the *only* way to handle repeated operations over sequences in Prolog, where there is nothing like a *cycle* programming construct

Recursion scheme in Prolog

- since Prolog search strategy is depth-first
- in particular, with clauses used orderly, top-down
- *termination* is handled with a fact, typically coming *before* the recursive rule
- as already seen in the cases of `num/1` and `plus/3` above

Typical Example: `member` I

`member/2`

Checking whether the first argument is a term that is a member of the list in the second argument

```
member(X,[X|Xs]).
member(X,[Y|Ys]) :-  member(X,Ys).
```

- goals
  1. `?- member(b,[a,b,c])`
  2. `?- member(X,[a,b,c]).`
  3. `?- member(g(X),[f(a),g(b),f(c),g(d)]).`
  4. `?- member(z,[XT]).—`

Typical Example: `member` II

- remarks
  - search strategy: left to right through the list
  - devising out all the members of the list
  - conditional membership—given a certain computed substitution
  - generation of lists

References I

Apt, K. R. (2005).
The logic programming paradigm and Prolog.
In Mitchell, J. C., editor, *Concepts in Programming Languages*, chapter 15, pages 475–508. Cambridge University Press, Cambridge, UK.

Boole, G. ([1847] 2009).
The Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning.
Cambridge Library Collection – Mathematics. Cambridge University Press, reissue edition.

Buchanan, B. G. and Shortliffe, E. H. (1984).
Rule Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.
The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Longman Publishing Co., Inc.

Calegari, R., Denti, E., Mariani, S., and Omicini, A. (2017).
Logic Programming as a Service (LPaaS): Intelligence for the IoT.
In Fortino, G., Zhou, M., Lukszo, Z., Vasilakos, A. V., Basile, F., Palau, C., Liotta, A., Fanti, M. P., Guerrieri, A., and Vinci, A., editors, *2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*, pages 72–77. IEEE.

References II

Carroll, L. (1876).
The Hunting of the Snark.
Macmillan Publishers.

Colmerauer, A. and Roussel, P. (1996).
The birth of Prolog.
In Bergin Jr., T. J. and Gibson Jr., R. G., editors, *History of programming languages—II*, volume II, pages 331–367. ACM, New York, NY, USA.

Console, L., Lamma, E., Mello, P., and Milano, M. (1997).
Programmazione logica e Prolog.
UTET Libreria.

Denti, E., Omicini, A., and Ricci, A. (2001).
tuProlog: A light-weight Prolog for Internet applications and infrastructures.
In Ramakrishnan, I., editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg.
3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

References III

Denti, E., Omicini, A., and Ricci, A. (2005).
Multi-paradigm Java-Prolog integration in tuProlog.
Science of Computer Programming, 57(2):217–250.

Frege, G. ([1903] 1971).
On the Foundations of Geometry and Formal Theories of Arithmetic.
Yale University Press.

Gödel, K. (1931).
Über formal unentscheidbare sätze der Principia Mathematica und verwandter Systeme I.
Monatshefte für Mathematik und Physik, 38(1):173–198.

Kowalski, R. A. (1974).
Predicate logic as programming language.
In *Information Processing 74 – Proceedings of the 1974 IFIP Congress*, pages 569–574.
North-Holland Publishing Company.

Levesque, H. J. (2012).
Thinking as Computation: A First Course.
MIT Press.

References IV

Lloyd, J. W., editor (1990).
Computational Logic Symposium. Proceedings, Brussels, November 13/14, 1990.
ESPRIT Basic Research Series. Springer Berlin Heidelberg, Berlin, Heidelberg.

Martelli, A. and Montanari, U. (1982).
An efficient unification algorithm.
ACM Transactions on Programming Languages and Systems, 4(2):258–282.

Martin-Löf, P. (1996).
On the meanings of the logical constants and the justifications of the logical laws.
Nordic Journal of Philosophical Logic, 1(1):11–60.

Metakides, G. and Nerode, A. (1996a).
Foreword.
In [Metakides and Nerode, 1996b], pages vii–ix.

Metakides, G. and Nerode, A. (1996b).
Principles of Logic and Logic Programming, volume 13 of *Studies in Computer Science and Artificial Intelligence*.
Elsevier.

References V

Nilsson, U. and Maluszynski, J. (1995).
Logic, Programming and Prolog.
John Wiley & Sons, Inc., New York, NY, USA.

Pfenning, F. (2007).
Logic programming.
`http://www.cs.cmu.edu/~fp/courses/lp/lectures/lp-all.pdf`.

Rijk, L. M. D. (2002).
Aristotle: Semantics and Ontology. Volume I: General Introduction. The Works on Logic, volume 91 of *Philosophia Antiqua*.
Brill Academic Publishers.

Robinson, J. A. (1965).
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1):23–41.

Tarski, A. and Tarski, J. (1994).
Introduction to Logic and to the Methodology of the Deductive Sciences, volume 24 of *Oxford Logic Guides*.
Oxford University Press.

References VI

Whitehead, A. N. and Russell, B. (1927).
Principia Mathematica.
Cambridge University Press.

Logic & Computation

Digital Culture

Andrea Omicini
andrea.omicini@unibo.it

Master in Business Management / Bologna Business School

Academic Year 2017/2018