Computational Logic

Recall of First-Order Logic

Damiano Zanardini

UPM European Master in Computational Logic (EMCL)
School of Computer Science
Technical University of Madrid
damiano@fi.upm.es

Academic Year 2008/2009

Syntax of a first-order language

Sidenotes

- we try to introduce the ideas of this course requiring as little *formal logic* background as possible (since you are studying it at the same time)
- we start directly from *first-order logic* (sometimes referred to as *predicate logic*); *propositional logic* is a special, much simpler case

PROPOSITIONAL LOGIC (only propositions, no variables) " $\subset$ " FIRST-ORDER LOGIC

What we need to talk about logic: thinking formally!

- be able to deal with symbols
- do not be misled by names or notation
- true is not reasonable nor well-formed! $\rightsquigarrow$ implication
- practice: (1) formalize informal ideas (2) give informal meaning to formulæ
- never forget the relation between syntax and semantics

Syntax of a first-order language

An *alphabet* $\mathcal{A}$ consists of

- *variable* symbols: x, y, z, w, x', ...
- *function* symbols: $f(\_)$, $g(\_,\_)$, ... ($arity$ = no. of args: $f/1$, $g/2$)
- *predicate* symbols: $p(\_)$, $q(\_,\_)$, ..., $=/2$
- *connectives*: $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$
- *quantifiers*: $\forall$, $\exists$

- *constants* (a, b, c, ...) = 0-arity functions
- *propositions* = 0-arity predicates

Metalanguage

F and G denote arbitrary formulæ

Syntax of a first-order language

Example: alphabet

- variables: x, y, z
- constants (0-ary functions): a, b, c, tom, 0, 1
- functions: $f/1$, $g/2$
- predicates: $p/0$, $q/2$, $cat/1$, $+/3$

Terms

- a variable is a term
- if $t_1, ..., t_n$ are terms and f is an n-ary ($n \geq 0$, thus including constants) function symbol, then $f(t_1, ..., t_n)$ is a term

Examples

- correct: a, $f(tom)$, $f(f(f(x)))$, $g(1, f(a))$, $a()$?
- incorrect: $a(1)$, $f(2)$, $g(g, g(1))$, $a + y$, $g(0c)$, $f(1, +(a, f(z), c)$

Syntax of a first-order language

Example: alphabet

- variables: x, y, z
- constants (0-ary functions): a, b, c, tom, 0, 1
- functions: $f/1$, $g/2$
- predicates: $p/0$, $q/2$, $cat/1$, $+/3$

Atoms

- if $t_1, ..., t_n$ are terms and p is an n-ary ($n \geq 0$) predicate symbol, then $p(t_1, ..., t_n)$ is an atom

Examples

- correct: p, $q(a, f(1))$, $cat(g(x, y))$, $+(a, f(z), c)$
- incorrect: $q(p, p)$, $cat(a, 1)$, $f(q(a, a))$, $q(0, , z)$

Syntax of a first-order language

Example: alphabet

- variables: x, y, z
- constants (0-ary functions): a, b, c, tom, 0, 1
- functions: $f/1$, $g/2$
- predicates: $p/0$, $q/2$, $cat/1$, $+/3$

Formulæ

- an atom is a formula
- if F and G are formulæ, and x is a variable, then $\neg F$, $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$, $\forall x F$ and $\exists x F$ are also formulæ

Examples

- correct: $(p \rightarrow \neg q(a, f(x)))$, $(\neg cat(a) \wedge (\forall x p \vee \exists y cat(y)))$
- incorrect: $p\wedge$, $\exists z f(1)$, $\forall p$, $q(a, b) \leftrightarrow tom$

Syntax of a first-order language

Literals

A *literal* is an atom or the negation of an atom

- $p, \quad \neg p, \quad q(a, f(1)), \quad \neg q(a, f(1)), \quad cat(g(x, y)), \quad \neg cat(g(x, y))$

Precedence

- parentheses give an *order* between operators, but can make a formula quite unreadable
- we can use an order of precedence between operators in order to remove some parentheses without introducing *ambiguity*

$$((p \wedge \neg q) \rightarrow (p \vee r)) \quad \rightsquigarrow \quad p \wedge \neg q \rightarrow p \vee r \quad \rightsquigarrow \quad \begin{array}{c} p \wedge (\neg q \rightarrow p) \vee r \\ ((p \wedge \neg q) \rightarrow p) \vee r \\ (p \wedge \neg q) \rightarrow (p \vee r) \\ p \wedge (\neg q \rightarrow p \vee r) \end{array}$$

Syntax of a first-order language

Literals

A *literal* is an atom or the negation of an atom

- $p, \quad \neg p, \quad q(a, f(1)), \quad \neg q(a, f(1)), \quad cat(g(x, y)), \quad \neg cat(g(x, y))$

Precedence

- parentheses give an *order* between operators, but can make a formula quite unreadable
- we can use an order of precedence between operators in order to remove some parentheses without introducing *ambiguity*

$$((p \wedge \neg q) \rightarrow (p \vee r)) \quad \rightsquigarrow \quad p \wedge \neg q \rightarrow p \vee r \quad \rightsquigarrow \quad \begin{array}{c} p \wedge (\neg q \rightarrow p) \vee r \\ ((p \wedge \neg q) \rightarrow p) \vee r \\ (p \wedge \neg q) \rightarrow (p \vee r) \\ p \wedge (\neg q \rightarrow p \vee r) \end{array}$$

- $\{\neg, \forall, \exists\}$ higher precedence than $\{\wedge, \vee\}$
  $\{\wedge, \vee\}$ higher precedence than $\{\rightarrow, \leftrightarrow\}$

Syntax of a first-order language

Free and bounded variables

- an occurrence of the variable x is *bounded* in F if it is in the scope of a quantifier $\forall x$ or $\exists x$ $\quad \rightsquigarrow \quad$ $\exists x(p(1, x, y) \wedge q(x)) \wedge q(x)$
- otherwise, it is said to be *free* $\quad \rightsquigarrow \quad$ $\exists x(p(1, x, y) \wedge q(x)) \wedge q(x)$
- a formula if *closed* if it contains no free occurrences

Substitution

- $F(x)$ denotes a formula where x occurs free somewhere
- $F(x/t)$ denotes a formula where every free occurrence of x has been replaced by a term t, provided x does not occur free in the scope of any $\forall y$ or $\exists y$ for y occurring in t

$$F \equiv s(x) \wedge (\forall y(p(x) \rightarrow q(y))) \qquad\qquad F(x/f(y, y)) \text{ cannot be done}$$

- important: $\forall x p(x)$ is the same as $\forall y p(y)$, and this can be generalized

Syntax of a first-order language

Alternative notation

$\wedge$	&	$\rightarrow$	$\Rightarrow$, $\supset$
$\leftrightarrow$	$\Leftrightarrow$, $\equiv$	$\exists xF$	$\exists x.\ F$
$\forall xF$	$\forall x.\ F$	p, q, r	P, Q, R

More than first-order

- *second-order logic*: it allows quantification over functions and predicates (e.g., *mathematical induction*)

$$\forall p(p(0) \wedge \forall k(p(k) \rightarrow p(s(k))) \rightarrow \forall np(n))$$

- *higher-order logic* allows quantification over functions and predicates of any order (as in *functional programming*)