Formal Specification and Verification

First-order logic (Part 1)

15.05.2012

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Mathematical foundations

Formal logic:

- Syntax: a formal language (formula expressing facts)
- Semantics: to define the meaning of the language, that is which facts are valid)
- Deductive system: made of axioms and inference rules to formaly derive theorems, that is facts that are provable

Last time

Propositional classical logic

- Syntax
- Semantics

  Models, Validity, and Satisfiability; Entailment and Equivalence
- Checking Unsatisfiability

  Truth tables

  "Rewriting" using equivalences

  Proof systems: clausal/non-clausal

  - non-clausal: Hilbert calculus

    sequent calculus

  - clausal: Resolution; DPLL (translation to CNF needed)

  - Binary Decision Diagrams

Limitations of Propositional Logic

- Fixed, finite number of objects

  Cannot express: let G be group with arbitrary number of elements

- No functions or relations with arguments

  Can express: finite function/relation table p_{ij}

  Cannot express: properties of function/relation on all arguments, e.g., $+$ is associative

- Static interpretation

  Programs change value of their variables, e.g., via assignment, call, etc.

  Propositional formulas look at one single interpretation at a time

Beyond the Limitations of Propositional Logic

- First order logic

  (+ functions)

- Temporal logic

  (+ computations)

- Dynamic logic

  (+ computations + functions)

Part 2: First-Order Logic

First-order logic

- formalizes fundamental mathematical concepts
- is expressive (Turing-complete)
- is not too expressive
  (e. g. not axiomatizable: natural numbers, uncountable sets)
- has a rich structure of decidable fragments
- has a rich model and proof theory

First-order logic is also called (first-order) predicate logic.

2.1 Syntax

Syntax:

- non-logical symbols (domain-specific)
  $\Rightarrow$ terms, atomic formulas
- logical symbols (domain-independent)
  $\Rightarrow$ Boolean combinations, quantifiers

Signature

A signature

$$\Sigma = (\Omega, \Pi),$$

fixes an alphabet of non-logical symbols, where

- Ω is a set of function symbols f with arity $n \geq 0$, written f/n,
- Π is a set of predicate symbols p with arity $m \geq 0$, written p/m.

If $n = 0$ then f is also called a constant (symbol).
If $m = 0$ then p is also called a propositional variable.
We use letters P, Q, R, S, to denote propositional variables.

Signature

Refined concept for practical applications:
many-sorted signatures (corresponds to simple type systems in programming languages).

Most results established for one-sorted signatures extend in a natural way to many-sorted signatures.

Many-sorted Signature

A many-sorted signature

$$\Sigma = (S, \Omega, \Pi),$$

fixes an alphabet of non-logical symbols, where

- S is a set of sorts,
- Ω is a set of function symbols f with arity $a(f) = s_1 \dots s_n \to s$,
- Π is a set of predicate symbols p with arity $a(p) = s_1 \dots s_m$

where $s_1, \dots, s_n, s_m, s$ are sorts.

Variables

Predicate logic admits the formulation of abstract, schematic assertions. (Object) variables are the technical tool for schematization.

We assume that

$$X$$

is a given countably infinite set of symbols which we use for (the denotation of) variables.

Variables

Predicate logic admits the formulation of abstract, schematic assertions. (Object) variables are the technical tool for schematization.

We assume that

$$X$$

is a given countably infinite set of symbols which we use for (the denotation of) variables.

Many-sorted case:

We assume that for every sort $s \in S$, X_s is a given countably infinite set of symbols which we use for (the denotation of) variables of sort s.

Terms

Terms over Σ (resp., Σ-terms) are formed according to these syntactic rules:

$$
\begin{array}{llll}
t, u, v & ::= & x & , x \in X & \text{(variable)} \\
 & \mid & f(t_1, ..., t_n) & , f/n \in \Omega & \text{(functional term)}
\end{array}
$$

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X).

A term not containing any variable is called a ground term.

By T_Σ we denote the set of Σ-ground terms.

Terms

Terms over Σ (resp., Σ-terms) are formed according to these syntactic rules:

$$
\begin{array}{llll}
t, u, v & ::= & x & , x \in X & \text{(variable)} \\
 & | & f(t_1, ..., t_n) & , f/n \in \Omega & \text{(functional term)}
\end{array}
$$

By $\mathrm{T}_\Sigma(X)$ we denote the set of Σ-terms (over X).
A term not containing any variable is called a ground term.
By T_Σ we denote the set of Σ-ground terms.

Many-sorted case:

a variable $x \in X_s$ is a term of sort s

if $a(f) = s_1 \ldots s_n \rightarrow s$, and t_i are terms of sort s_i, $i = 1, \ldots, n$ then $f(t_1, ..., t_n)$ is a term of sort s.

Terms

In other words, terms are formal expressions with well-balanced brackets which we may also view as marked, ordered trees.

The markings are function symbols or variables.

The nodes correspond to the subterms of the term.

A node v that is marked with a function symbol f of arity n has exactly n subtrees representing the n immediate subterms of v.

Atoms

Atoms (also called atomic formulas) over Σ are formed according to this syntax:

$$
\begin{array}{llll}
A, B & ::= & p(t_1, \ldots, t_m) & ,\ p/m \in \Pi \\
\Big[& \mid & (t \approx t') & \text{(equation)} \quad \Big]
\end{array}
$$

Whenever we admit equations as atomic formulas we are in the realm of first-order logic with equality. Admitting equality does not really increase the expressiveness of first-order logic, (cf. exercises). But deductive systems where equality is treated specifically can be much more efficient.

Atoms

Atoms (also called atomic formulas) over Σ are formed according to this syntax:

$$\begin{array}{llll} A, B & ::= & p(t_1, \ldots, t_m) & , \; p/m \in \Pi \\ \Big[& \mid & (t \approx t') & \text{(equation)} \quad \Big] \end{array}$$

Whenever we admit equations as atomic formulas we are in the realm of first-order logic with equality. Admitting equality does not really increase the expressiveness of first-order logic, (cf. exercises). But deductive systems where equality is treated specifically can be much more efficient.

Many-sorted case:

If $a(p) = s_1 \ldots s_m$, we require that t_i is a term of sort s_i for $i = 1, \ldots, m$.

Literals

$$
\begin{array}{lcll}
L & ::= & A & \text{(positive literal)} \\
 & \mid & \neg A & \text{(negative literal)}
\end{array}
$$

Clauses

$$
\begin{array}{rcll}
C, D & ::= & \bot & \text{(empty clause)} \\
 & | & L_1 \vee \ldots \vee L_k, \quad k \geq 1 & \text{(non-empty clause)}
\end{array}
$$

General First-Order Formulas

$F_\Sigma(X)$ is the set of first-order formulas over Σ defined as follows:

$$
\begin{array}{llll}
F, G, H & ::= & \bot & \text{(falsum)} \\
& \mid & \top & \text{(verum)} \\
& \mid & A & \text{(atomic formula)} \\
& \mid & \neg F & \text{(negation)} \\
& \mid & (F \wedge G) & \text{(conjunction)} \\
& \mid & (F \vee G) & \text{(disjunction)} \\
& \mid & (F \rightarrow G) & \text{(implication)} \\
& \mid & (F \leftrightarrow G) & \text{(equivalence)} \\
& \mid & \forall x F & \text{(universal quantification)} \\
& \mid & \exists x F & \text{(existential quantification)}
\end{array}
$$

Notational Conventions

We omit brackets according to the following rules:

- $\neg \quad >_p \quad \wedge \quad >_p \quad \vee \quad >_p \quad \rightarrow \quad >_p \quad \leftrightarrow$
  (binding precedences)
- $\vee$ and $\wedge$ are associative and commutative
- $\rightarrow$ is right-associative

$Qx_1, \ldots, x_n\, F$ abbreviates $Qx_1 \ldots Qx_n\, F$.

Notational Conventions

We use infix-, prefix-, postfix-, or mixfix-notation with the usual operator precedences.

Examples:

$$
\begin{array}{ccc}
s + t * u & \text{for} & +(s, *(t, u)) \\
s * u \leq t + v & \text{for} & \leq(*(s, u), +(t, v)) \\
-s & \text{for} & -(s) \\
0 & \text{for} & 0()
\end{array}
$$

Example: Peano Arithmetic

Signature:

$$\Sigma_{PA} = (\Omega_{PA},\ \Pi_{PA})$$
$$\Omega_{PA} = \{0/0,\ +/2,\ */2,\ s/1\}$$
$$\Pi_{PA} = \{\leq/2,\ </2\}$$

$+$, $*$, $<$, $\leq$ infix; $* >_p + >_p < >_p \leq$

Examples of formulas over this signature are:

$$\forall x, y(x \leq y \leftrightarrow \exists z(x + z \approx y))$$
$$\exists x \forall y(x + y \approx y)$$
$$\forall x, y(x * s(y) \approx x * y + x)$$
$$\forall x, y(s(x) \approx s(y) \rightarrow x \approx y)$$
$$\forall x \exists y(x < y \land \neg \exists z(x < z \land z < y))$$

Remarks About the Example

We observe that the symbols $\leq$, $<$, 0, s are redundant as they can be defined in first-order logic with equality just with the help of $+$. The first formula defines $\leq$, while the second defines zero. The last formula, respectively, defines s.

Eliminating the existential quantifiers by Skolemization (cf. below) reintroduces the "redundant" symbols.

Consequently there is a *trade-off* between the complexity of the quantification structure and the complexity of the signature.

Example: Specifying LISP lists

Signature:

$\Sigma_{\mathsf{Lists}} = (\Omega_{\mathsf{Lists}}, \Pi_{\mathsf{Lists}})$

$\Omega_{\mathsf{Lists}} = \{\mathsf{car}/1, \mathsf{cdr}/1, \mathsf{cons}/2\}$

$\Pi_{\mathsf{Lists}} = \emptyset$

Examples of formulae:

$\forall x, y \;\; \mathsf{car}(\mathsf{cons}(x, y)) \approx x$

$\forall x, y \;\; \mathsf{cdr}(\mathsf{cons}(x, y)) \approx y$

$\forall x \quad \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) \approx x$

Many-sorted signatures

Example:

Signature

$S = \{\text{array}, \text{index}, \text{element}\}$ set of sorts

$\Omega = \{\text{read}, \text{write}\}$

$$a(\text{read}) = \text{array} \times \text{index} \rightarrow \text{element}$$

$$a(\text{write}) = \text{array} \times \text{index} \times \text{element} \rightarrow \text{array}$$

$\Pi = \emptyset$

$X = \{X_s \mid s \in S\}$

Examples of formulae:

$$\forall x : \text{array} \;\; \forall i : \text{index} \;\; \forall j : \text{index} \;\; (i \approx j \rightarrow \text{write}(x, i, \text{read}(x, j)) \approx x)$$

$$\forall x : \text{array} \; \forall y : \text{array} \;\; (x \approx y \leftrightarrow \forall i : \text{index} \;\; (\text{read}(x, i) \approx \text{read}(y, i)))$$

Bound and Free Variables

In QxF, $Q \in \{\exists, \forall\}$, we call F the scope of the quantifier Qx.
An *occurrence* of a variable x is called bound, if it is inside the scope of a quantifier Qx.
Any other occurrence of a variable is called free.

Formulas without free variables are also called closed formulas or sentential forms.

Formulas without variables are called ground.

Bound and Free Variables

Example:

$$\forall y \quad (\overbrace{\forall x \quad \overbrace{p(x)}^{scope} \quad \rightarrow \quad q(x, y)}^{scope})$$

The occurrence of y is bound, as is the first occurrence of x. The second occurrence of x is a free occurrence.

Substitutions

Substitution is a fundamental operation on terms and formulas that occurs in all inference systems for first-order logic.

In general, substitutions are mappings

$$\sigma : X \rightarrow \mathsf{T}_\Sigma(X)$$

such that the domain of σ, that is, the set

$$dom(\sigma) = \{x \in X \mid \sigma(x) \neq x\},$$

is finite. The set of variables introduced by σ, that is, the set of variables occurring in one of the terms $\sigma(x)$, with $x \in dom(\sigma)$, is denoted by $codom(\sigma)$.

Substitutions

Substitutions are often written as $[s_1/x_1, \ldots, s_n/x_n]$, with x_i pairwise distinct, and then denote the mapping

$$[s_1/x_1, \ldots, s_n/x_n](y) = \begin{cases} s_i, & \text{if } y = x_i \\ y, & \text{otherwise} \end{cases}$$

We also write $x\sigma$ for $\sigma(x)$.

The modification of a substitution σ at x is defined as follows:

$$\sigma[x \mapsto t](y) = \begin{cases} t, & \text{if } y = x \\ \sigma(y), & \text{otherwise} \end{cases}$$

Why Substitution is Complicated

We define the application of a substitution σ to a term t or formula F by structural induction over the syntactic structure of t or F by the equations depicted on the next page.

In the presence of quantification it is surprisingly complex:
We need to make sure that the (free) variables in the codomain of σ are not *captured* upon placing them into the scope of a quantifier Qy, hence the bound variable must be renamed into a "fresh", that is, previously unused, variable z.

Application of a Substitution

"Homomorphic" extension of σ to terms and formulas:

$$
\begin{aligned}
f(s_1, \ldots, s_n)\sigma &= f(s_1\sigma, \ldots, s_n\sigma)\\
\bot\sigma &= \bot\\
\top\sigma &= \top\\
p(s_1, \ldots, s_n)\sigma &= p(s_1\sigma, \ldots, s_n\sigma)\\
(u \approx v)\sigma &= (u\sigma \approx v\sigma)\\
\neg F\sigma &= \neg(F\sigma)\\
(F \rho G)\sigma &= (F\sigma \, \rho \, G\sigma)\ ; \quad \text{for each binary connective } \rho\\
(Qx\, F)\sigma &= Qz\,(F\,\sigma[x \mapsto z])\ ; \quad \text{with } z \text{ a fresh variable}
\end{aligned}
$$

2.2 Semantics

To give semantics to a logical system means to define a notion of truth for the formulas. The concept of truth that we will now define for first-order logic goes back to Tarski.

As in the propositional case, we use a two-valued logic with truth values "true" and "false" denoted by 1 and 0, respectively.

Structures

A Σ-algebra (also called Σ-interpretation or Σ-structure) is a triple

$$\mathcal{A} = (U,\ (f_{\mathcal{A}} : U^n \rightarrow U)_{f/n \in \Omega},\ (p_{\mathcal{A}} \subseteq U^m)_{p/m \in \Pi})$$

where $U \neq \emptyset$ is a set, called the universe of $\mathcal{A}$.

Normally, by abuse of notation, we will have $\mathcal{A}$ denote both the algebra and its universe.

By $\Sigma - \mathrm{Alg}$ we denote the class of all Σ-algebras.

Many-sorted Structures

A many-sorted Σ-algebra (also called Σ-interpretation or Σ-structure), where $\Sigma = (S, \Omega, \Pi)$ is a triple

$$\mathcal{A} = (\{U_s\}_{s \in S},\ (f_{\mathcal{A}} : U_{s_1} \times \ldots \times U_{s_n} \rightarrow U_s)_{\substack{f \in \Omega, \\ a(f) = s_1 \ldots s_n \rightarrow s}}\ (p_{\mathcal{A}} : U_{s_1} \times \ldots \times U_{s_m} \rightarrow \{0, 1\})_{\substack{p \in \Pi \\ a(p) = s_1 \ldots s_m}})$$

where $U \neq \emptyset$ is a set, called the universe of $\mathcal{A}$.

Assignments

A variable has no intrinsic meaning. The meaning of a variable has to be defined externally (explicitly or implicitly in a given context) by an assignment.

A (variable) assignment, also called a valuation (over a given Σ-algebra $\mathcal{A}$), is a map $\beta : X \rightarrow \mathcal{A}$.

Assignments

A variable has no intrinsic meaning. The meaning of a variable has to be defined externally (explicitly or implicitly in a given context) by an assignment.

A (variable) assignment, also called a valuation (over a given Σ-algebra $\mathcal{A}$), is a map $\beta : X \to \mathcal{A}$.

Many-sorted case:

$\beta = \{\beta_s\}_{s \in S}, \beta_s : X_s \to U_s$

Value of a Term in $\mathcal{A}$ with Respect to β

By structural induction we define

$$\mathcal{A}(\beta) : \mathsf{T}_\Sigma(X) \to \mathcal{A}$$

as follows:

$$\begin{aligned} \mathcal{A}(\beta)(x) &= \beta(x), \qquad & x \in X \\ \mathcal{A}(\beta)(f(s_1, \ldots, s_n)) &= f_{\mathcal{A}}(\mathcal{A}(\beta)(s_1), \ldots, \mathcal{A}(\beta)(s_n)), \qquad & f/n \in \Omega \end{aligned}$$

Value of a Term in $\mathcal{A}$ with Respect to β

In the scope of a quantifier we need to evaluate terms with respect to modified assignments. To that end, let $\beta[x \mapsto a] : X \to \mathcal{A}$, for $x \in X$ and $a \in \mathcal{A}$, denote the assignment

$$\beta[x \mapsto a](y) := \begin{cases} a & \text{if } x = y \\ \beta(y) & \text{otherwise} \end{cases}$$

Truth Value of a Formula in $\mathcal{A}$ with Respect to β

$\mathcal{A}(\beta) : \mathsf{F}_\Sigma(X) \to \{0, 1\}$ is defined inductively as follows:

$$
\begin{aligned}
\mathcal{A}(\beta)(\bot) &= 0 \\
\mathcal{A}(\beta)(\top) &= 1 \\
\mathcal{A}(\beta)(p(s_1, \dots, s_n)) &= p_\mathcal{A}(\mathcal{A}(\beta)(s_1), \dots, \mathcal{A}(\beta)(s_n)) \\
\mathcal{A}(\beta)(s \approx t) = 1 \quad &\Leftrightarrow \quad \mathcal{A}(\beta)(s) = \mathcal{A}(\beta)(t) \\
\mathcal{A}(\beta)(\neg F) = 1 \quad &\Leftrightarrow \quad \mathcal{A}(\beta)(F) = 0 \\
\mathcal{A}(\beta)(F \rho G) &= \mathsf{B}_\rho(\mathcal{A}(\beta)(F), \mathcal{A}(\beta)(G)) \\
& \text{ with } \mathsf{B}_\rho \text{ the Boolean function associated with } \rho \\
\mathcal{A}(\beta)(\forall x F) &= \min_{a \in U} \{\mathcal{A}(\beta[x \mapsto a])(F)\} \\
\mathcal{A}(\beta)(\exists x F) &= \max_{a \in U} \{\mathcal{A}(\beta[x \mapsto a])(F)\}
\end{aligned}
$$

Example

The "Standard" Interpretation for Peano Arithmetic:

$$
\begin{aligned}
U_{\mathbb{N}} &= \{0, 1, 2, \ldots\} \\
0_{\mathbb{N}} &= 0 \\
s_{\mathbb{N}} : U_{\mathbb{N}} \to U_{\mathbb{N}} & \quad s_{\mathbb{N}}(n) = n + 1 \\
+_{\mathbb{N}} : U_{\mathbb{N}}^2 \to U_{\mathbb{N}} & \quad +_{\mathbb{N}}(n, m) = n + m \\
*_{\mathbb{N}} : U_{\mathbb{N}}^2 \to U_{\mathbb{N}} & \quad *_{\mathbb{N}}(n, m) = n * m \\
\leq_{\mathbb{N}} : U_{\mathbb{N}}^2 \to \{0, 1\} & \quad \leq_{\mathbb{N}} (n, m) = 1 \text{ iff } n \text{ less than or equal to } m \\
<_{\mathbb{N}} : U_{\mathbb{N}}^2 \to \{0, 1\} & \quad \leq_{\mathbb{N}} (n, m) = 1 \text{ iff } n \text{ less than } m
\end{aligned}
$$

Note that $\mathbb{N}$ is just one out of many possible Σ_{PA}-interpretations.

Example

Values over $\mathbb{N}$ for Sample Terms and Formulas:

Under the assignment $\beta : x \mapsto 1, y \mapsto 3$ we obtain

$$\begin{array}{lcl} \mathbb{N}(\beta)(s(x) + s(0)) & = & 3 \\ \mathbb{N}(\beta)(x + y \approx s(y)) & = & 1 \\ \mathbb{N}(\beta)(\forall x, y(x + y \approx y + x)) & = & 1 \\ \mathbb{N}(\beta)(\forall z\ z \leq y) & = & 0 \\ \mathbb{N}(\beta)(\forall x \exists y\ x < y) & = & 1 \end{array}$$

2.3 Models, Validity, and Satisfiability

F is valid in $\mathcal{A}$ under assignment β:

$$\mathcal{A}, \beta \models F \quad :\Leftrightarrow \quad \mathcal{A}(\beta)(F) = 1$$

F is valid in $\mathcal{A}$ ($\mathcal{A}$ is a model of F):

$$\mathcal{A} \models F \quad :\Leftrightarrow \quad \mathcal{A}, \beta \models F, \text{ for all } \beta \in X \to U_{\mathcal{A}}$$

F is valid (or is a tautology):

$$\models F \quad :\Leftrightarrow \quad \mathcal{A} \models F, \text{ for all } \mathcal{A} \in \Sigma\text{-alg}$$

F is called satisfiable iff there exist $\mathcal{A}$ and β such that $\mathcal{A}, \beta \models F$.
Otherwise F is called unsatisfiable.

Entailment and Equivalence

F entails (implies) G (or G is a consequence of F), written $F \models G$

$:\Leftrightarrow$ for all $\mathcal{A} \in \Sigma$-alg and $\beta \in X \to U_{\mathcal{A}}$,
whenever $\mathcal{A}, \beta \models F$ then $\mathcal{A}, \beta \models G$.

F and G are called equivalent

$:\Leftrightarrow$ for all $\mathcal{A} \in \Sigma$-alg und $\beta \in X \to U_{\mathcal{A}}$ we have
$\mathcal{A}, \beta \models F \quad \Leftrightarrow \quad \mathcal{A}, \beta \models G$.

Entailment and Equivalence

Proposition 2.6:
F entails G iff $(F \rightarrow G)$ is valid

Proposition 2.7:
F and G are equivalent iff $(F \leftrightarrow G)$ is valid.

Extension to sets of formulas N in the "natural way", e.g., $N \models F$

$:\Leftrightarrow$ for all $\mathcal{A} \in \Sigma$-alg and $\beta \in X \rightarrow U_{\mathcal{A}}$:
if $\mathcal{A}, \beta \models G$, for all $G \in N$, then $\mathcal{A}, \beta \models F$.

Validity vs. Unsatisfiability

Validity and unsatisfiability are just two sides of the same medal as explained by the following proposition.

Proposition 2.8:

$$F \text{ valid} \quad \Leftrightarrow \quad \neg F \text{ unsatisfiable}$$

Hence in order to design a theorem prover (validity checker) it is sufficient to design a checker for unsatisfiability.

Q: In a similar way, entailment $N \models F$ can be reduced to unsatisfiability. How?