First Order Logic as a Specification Language

Jan van Eijck

Specification and Testing, Week 3, 2012

First Order Logic: Syntax

Assume a set of function symbols is given, and let f range over function symbols.

Assume a set of predicate symbols is given, and let P range over predicate symbols.

$$
\begin{array}{rcl}
t & ::= & x \mid f(t_1, \ldots, t_n) \\
\varphi & ::= & P(t_1, \ldots, t_n) \mid t_1 = t_2 \\
 & & \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\varphi \leftrightarrow \varphi) \\
 & & \mid (\forall x \varphi) \mid (\exists x \varphi)
\end{array}
$$

Important Syntactic Notions

- Free and bound variable occurrences in a formula φ.
- Substitution of a term t for all free occurrences of variable x in φ. Notation $\varphi[t/x]$.
- t is free for x in φ.
- Alphabetic variant of a formula φ.

Know Yourself!

- Do you know the basic concepts of first order logic: language, freedom and bondage, models, truth definition?
- Are you able to read and understand formulas of first order logic?
- If the answer to one or both of these questions is no, you should consult:

  `http://www.logicinaction.org/docs/ch4.pdf`

Models

- Universes (or: domains)
- Interpretations for relation symbols
- Interpretations for function symbols
- Notion of a lookup-table or environment for a universe.
- Question: Why does something like the truth table method for propositional logic fail for first order predicate logic?

Truth Definition

$$M \models_g \varphi$$

- φ is true in M under variable assignment g.
- Here $M = (D, I)$. D is the domain, I is the interpretation function.
- The variable assignment g is a function of type $X \to D$, where X is the set of variables of the language.
- If t is a term, then $[\![t]\!]_I^g$ is the value of t in M, given variable assignment g.

Values for Terms

1. $[\![v]\!]_I^g := g(v)$

2. $[\![f(t_1, \ldots, t_n)]\!]_I^g := f_I([\![t_1]\!]_I^g, \ldots, [\![t_n]\!]_I^g)$

Truth of Formulas

$$
\begin{array}{lll}
M \models_g Pt_1 \cdots t_n & \text{iff} & ([\![t_1]\!]^g_I, \ldots, [\![t_n]\!]^g_I) \in P_I \\
M \models_g t_1 = t_2 & \text{iff} & [\![t_1]\!]^g_I = [\![t_2]\!]^g_I \\
M \models_g \neg\varphi & \text{iff} & \text{it is not the case that } M \models_g \varphi. \\
M \models_g \varphi_1 \wedge \varphi_2 & \text{iff} & M \models_g \varphi_1 \text{ and } M \models_g \varphi_2 \\
M \models_g \varphi_1 \vee \varphi_2 & \text{iff} & M \models_g \varphi_1 \text{ or } M \models_g \varphi_2 \\
M \models_g \varphi_1 \longrightarrow \varphi_2 & \text{iff} & M \models_g \varphi_1 \text{ implies } M \models_g \varphi_2 \\
M \models_g \varphi_1 \leftrightarrow \varphi_2 & \text{iff} & M \models_g \varphi_1 \text{ if and only if } M \models_g \varphi_2 \\
M \models_g \forall v\varphi & \text{iff} & \text{for all } d \in D \text{ it holds that } M \models_{g[v:=d]} \varphi \\
M \models_g \exists v\varphi & \text{iff} & \text{for at least one } d \in D \text{ it holds that } M \models_{g[v:=d]} \varphi
\end{array}
$$

Valid Consequence

1. A formula ψ logically follows from a formula φ (alternatively, φ logically implies ψ) if every model plus assignment which makes φ true also makes ψ true.

2. The notation for 'φ logically implies ψ' is $\varphi \models \psi$.

3. In $\varphi \models \psi$, the formula φ is called the premiss, ψ the conclusion.

4. We can also allow more than one premiss.

Which if the following are true?

1. $\forall x \forall y (Rxy \Rightarrow Ryx), Rab \models Rba$
2. $\forall x \forall y (Rxy \Rightarrow Ryx), Rab \models Raa$
3. $\forall x \forall y \forall z ((Rxy \wedge Ryz) \Rightarrow Rxz), Rab, Rac \models Rbc,$
4. $\forall x \forall y \forall z ((Rxy \wedge Ryz) \Rightarrow Rxz), Rab, Rbc \models Rac,$
5. $\forall x \forall y \forall z ((Rxy \wedge Ryz) \Rightarrow Rxz), Rab, \neg Rac \models \neg Rbc,$
6. $\forall x \forall y \forall z ((Rxy \wedge Ryz) \Rightarrow Rxz), \forall x \forall y (Rxy \Rightarrow Ryx), Rab \models Raa.$

Module Declaration

```
module Week3

where

import Data.List
```

Syntax of First Order Logic in Haskell: Terms

```
type Name = String
data Term = V Name | F Name [Term] deriving (Eq,Ord)

instance Show Term where
  show (V name)    = name
  show (F name []) = name
  show (F name ts) = name ++ show ts
```

Operations on Terms (1): Finding the Variables

```
x, y, z :: Term
x = V "x"
y = V "y"
z = V "z"

varsInTerm :: Term -> [Name]
varsInTerm (V name)       = [name]
varsInTerm (F _ ts) = varsInTerms ts where
  varsInTerms :: [Term] -> [Name]
  varsInTerms = nub . concat . map varsInTerm
```

Operations on Terms (2): Substitution

```
subst :: Name -> Term -> Term -> Term
subst name t (V name') =
  if name == name' then t else (V name')
subst name t (F name' ts) =
      F name' (map (subst name t) ts)
```

Example: $fxxxx^{x}_{fxxxx}$

```
*Week3> subst "x" (F "f" [x,x,x,x]) (F "f" [x,x,x,x])
f[f[x,x,x,x],f[x,x,x,x],f[x,x,x,x],f[x,x,x,x]]
```

Syntax of First Order Logic in Haskell: Formulas

```
data Formula = Atom Name [Term]
               | Eq Term Term
               | Neg  Formula
               | Impl Formula Formula
               | Equi Formula Formula
               | Conj [Formula]
               | Disj [Formula]
               | Forall Name Formula
               | Exists Name Formula
               deriving (Eq,Ord)
```

Syntax of First Order Logic in Haskell: Formulas (2)

```
instance Show Formula where
  show (Atom s [])    = s
  show (Atom s xs)    = s ++ show xs
  show (Eq t1 t2)     = show t1 ++ "==" ++ show t2
  show (Neg form)     = '~' : (show form)
  show (Impl f1 f2)   = "(" ++ show f1 ++ "==>"
                            ++ show f2 ++ ")"
  show (Equi f1 f2)   = "(" ++ show f1 ++ "<=>"
                            ++ show f2 ++ ")"
  show (Conj [])      = "true"
  show (Conj fs)      = "conj" ++ show fs
  show (Disj [])      = "false"
  show (Disj fs)      = "disj" ++ show fs
  show (Forall v f)   = "A " ++  v ++ (' ' : show f)
  show (Exists v f)   = "E " ++  v ++ (' ' : show f)
```

Example Formulas

```
r = Atom "R"

formula1 = Forall "x" (r [x,x])
formula2 = Forall "x"
             (Forall "y"
               (Impl (r [x,y]) (r [y,x])))
```

```
*Week3> formula1
A x R[x,x]
*Week3> formula2
A x A y (R[x,y]==>R[y,x])
```

Models for First Order Logic in Haskell

Several ways to represent a first order model. Here is one way:

- Domains as lists. Type `[a]`.
- Relations as characteristic functions for lists. Type `[a] -> Bool`.
- Functions as maps from lists to objects. Type `[a] -> a`.

Symbol Interpretation, Variable Lookup, Term Interpretation

Interpretations for relation symbols and function symbols.

```
type Rint a = Name -> [a] -> Bool
type Fint a = Name -> [a] -> a
```

Lookup function for variables.

```
type Lookup a = Name -> a
```

Interpretation of $[\![t]\!]^g_I$, where I is the interpretation of function symbols.

```
termVal :: Lookup a -> Fint a -> Term -> a
termVal g i (V name) = g name
termVal g i (F name ts) =
   i name (map (termVal g i) ts)
```

Definition of $g[v := d]$

```
changeLookup :: Lookup a -> Name -> a -> Lookup a
changeLookup g v d = \
    v' -> if v == v' then d else g v'
```

Implementation of Truth Definition

```
evalFOL :: Eq a =>
   [a] -> Lookup a -> Fint a -> Rint a -> Formula -> Bool
evalFOL domain g f i = evalFOL' g where
  evalFOL' g (Atom name ts) = i name (map (termVal g f) ts)
  evalFOL' g (Eq t1 t2) = termVal g f t1 == termVal g f t2
  evalFOL' g (Neg form) = not (evalFOL' g form)
  evalFOL' g (Impl f1 f2) = not
                   (evalFOL' g f1 && not (evalFOL' g f2))
  evalFOL' g (Equi f1 f2) = evalFOL' g f1 == evalFOL' g f2
  evalFOL' g (Conj fs)    = and (map (evalFOL' g) fs)
  evalFOL' g (Disj fs)    = or  (map (evalFOL' g) fs)
  evalFOL' g (Forall v form) =
    all (\ d -> evalFOL' (changeLookup g v d) form) domain
  evalFOL' g (Exists v form) =
    any (\ d -> evalFOL' (changeLookup g v d) form) domain
```

Example Interpretation Functions

```
f :: Fint Int
f "z" []    = 0
f "s" [i]   = succ i
f "p" [i,j] = i + j
f "t" [i,j] = i * j

i :: Rint Int
i "R" [i,j] = i < j
```

What Will Happen?

```
zero = F "z" []

frm1 = Exists "x" (r [zero,x])
frm2 = Exists "x" (Exists "y" (r [x,y]))
frm3 = Forall "x" (Exists "y" (r [x,y]))

evalFOL1 = evalFOL [0..] (\ v -> 0) f i frm1
evalFOL2 = evalFOL [0..] (\ v -> 0) f i frm2
evalFOL3 = evalFOL [0..] (\ v -> 0) f i frm3
evalFOL4 = evalFOL [0..] (\ v -> 0) f i (Neg frm3)
```