Logic

Jan van Eijck

CWI, Amsterdam and Uil-OTS, Utrecht

jve@cwi.nl

June 11, 2008

Abstract

We look at some key concepts from propositional logic: valuation, satisfaction, tautology, contradiction, logical consequence.
Next, we turn to predicate logic, and study and implement the truth definition for predicate logical formulas.

Module Declaration

```
module Logic

where
import List
import Char
import FormContent
```

We import `FormContent` because we want to be able to refer to our example implementation of a model for predicate logic.

Propositions

```
data Form = Prop String
          | Not Form
          | Conj [Form]
          | Disj [Form]
          deriving (Eq,Ord)
```

Some Examples

```
p1 = Prop "p1" ; p2 = Prop "p2" ; p3 = Prop "p3"
q1 = Prop "q1" ; q2 = Prop "q2" ; q3 = Prop "q3"
r1 = Prop "r1" ; r2 = Prop "r2" ; r3 = Prop "r3"

frm1 = Disj [p1, Not p1]
frm2 = Conj [p1, Not p1]
```

Showing Propositions

```
instance Show Form where
  show (Prop name) = name
  show (Not f) = '~' : '(' : show f ++ ")"
  show (Conj fs) = '&' : show fs
  show (Disj fs) = 'v' : show fs
```

Collecting the Variables from a Formula

This was part of your homework:

```
collectVars :: Form -> [String]
collectVars (Prop name) = [name]
collectVars (Not f) = collectVars f
collectVars (Conj fs) =
  (nub.concat) (map collectVars fs)
collectVars (Disj fs) =
  (nub.concat) (map collectVars fs)
```

Valuations

A valuation for a propositional formula is a map from its variable names to the booleans.

The list of all valuations for a propositional formula:

```
allVals :: Form -> [[(String,Bool)]]
allVals f = cv (collectVars f)
  where
   cv [] = [[]]
   cv (v:vs) = map ((v,True):) (cv vs)
               ++
               map ((v,False):) (cv vs)
```

Evaluation of Propositional Formulas

What is a reasonable convention for the truth of `Conj []`?

A conjunction is true if all its conjuncts are true. This requirement is trivially fulfilled in case there are no conjuncts. So `Conj []` is always true.

What is a reasonable convention for the truth of `Disj []`?

A disjunction is true if at least one disjunct is true. This requirement is never fulfilled in case there are no disjuncts. So `Disj []` is always false.

In the implementation we can use `all` for conjunctions and `any` for disjunctions.

```
eval :: [(String,Bool)] -> Form -> Bool
eval [] (Prop c) = error ("no info about " ++ c)
eval ((x,b):xs) (Prop c)
      | c == x    = b
      | otherwise = eval xs (Prop c)
eval xs (Not f)   = not (eval xs f)
eval xs (Conj fs) = all (eval xs) fs
eval xs (Disj fs) = any (eval xs) fs
```

Tautologies, Contradictions

Tautologies are formulas that evaluate to `True` for every valuation:

```
tautology :: Form -> Bool
tautology f =
   all (\ v -> eval v f) (allVals f)
```

Contradictions are formulas that evaluate to `True` for no valuation:

```
contradiction :: Form -> Bool
contradiction f =
   not (any  (\ v -> eval v f) (allVals f))
```

Propositional Satisfiability

A propositional formula is called satisfiable if there is a propositional valuation that makes the formula true.

Clearly, a formula is satisfiable if it is not a contradiction.

Or we can give a direct implementation:

```
satisfiable :: Form -> Bool
satisfiable  f =
   any (\ v -> eval v f) (allVals f)
```

Propositional Consequence

A propositional formula F_1 implies another formula F_1 if every valuation that satisfies F_1 also satisfies F_2.

From this definition:

F_1 implies F_2 iff $F_1 \wedge \neg F_2$ is a contradiction.

Implementation:

```
implies :: Form -> Form -> Bool
implies f1 f2 = contradiction (Conj [f1, Not f2])
```

Predicate Logic

Predicate logic is an extension of propositional logic with **structured basic propositions** and **quantifications**.

- A structured basic proposition consists of an n-ary predicate followed by n proper names or variables.
- A universally quantified formula consists of the symbol $\forall$ followed by a variable followed by a formula.
- An existentially quantified formula consists of the symbol $\exists$ followed by a variable followed by a formula.
- Other ingredients as in propositional logic.

Other names for predicate logic are first order logic or first order predicate logic. 'First order' indicates that the quantification is over entities (objects of the first order).

Formal Syntax for Predicate Logic

$$
\begin{aligned}
d &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
n &::= d \mid dn \\
\mathsf{var} &::= v_n \\
\mathsf{const} &::= c_n \\
\mathsf{term} &::= \mathsf{var} \mid \mathsf{const} \\
F &::= P_n(\mathsf{term}, \ldots, \mathsf{term}) \mid \neg F \mid (F \wedge F) \mid (F \vee F) \mid \forall \mathsf{var} F \mid \exists \mathsf{var} F
\end{aligned}
$$

Reformulation Using Lists

$$
\begin{aligned}
d &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
n &::= d \mid dn \\
\mathsf{var} &::= v_n \\
\mathsf{const} &::= c_n \\
\mathsf{term} &::= \mathsf{var} \mid \mathsf{const} \\
F &::= P_n[\mathsf{term}] \mid \neg F \mid \wedge[F] \mid \vee[F] \mid \forall \mathsf{var} F \mid \exists \mathsf{var} F
\end{aligned}
$$

A Datatype for Predicate Logical Formulas

For convenience, we leave out the constants.

```
type Name = String
data Var  = Var Name deriving (Eq,Ord)
```

Showing variables:

```
instance Show Var where
  show (Var name)  = name
```

Example variables:

```
x, y, z :: Var
x  = Var "x"
y  = Var "y"
z  = Var "z"
```

Formulas

```
data Frm = Pred String [Var]
         | Eql Var Var
         | Neg Frm
         | Impl Frm Frm
         | Equiv Frm Frm
         | Cnj [Frm]
         | Dsj [Frm]
         | Forall Var Frm
         | Exists Var Frm
     deriving (Eq,Ord)
```

Showing Formulas

```
instance Show Frm where
  show (Pred str []) = str
  show (Pred str vs) = str ++ concat [ show vs ]
  show (Eql v1 v2)   = show v1 ++ "==" ++ show v2
  show (Neg form)    = '~': (show form)
  show (Impl f1 f2) =
     "(" ++ show f1 ++ "==>" ++ show f2 ++ ")"
  show (Equiv f1 f2) =
     "(" ++ show f1 ++ "<=>" ++ show f2 ++ ")"
  show (Cnj []) =  "true"
  show (Cnj fs) =  "conj" ++ concat [ show fs ]
  show (Dsj []) =  "false"
  show (Dsj fs) =  "disj" ++ concat [ show fs ]
```

```
show (Forall v f) =
   "A" ++  show v ++ (' ' : show f)
show (Exists v f) =
   "E" ++  show v ++ (' ' : show f)
```

```
form0 = Pred "R" [x,y]
form1 = Pred "R" [x,x]
form2 = Exists x form1
reflF = Forall x form1
symmF = Forall x
         (Forall y
           (Impl (Pred "R" [x,y])
                 (Pred "R" [y,x])))
transF = Forall x
         (Forall y
           (Forall z
             (Impl (Cnj [Pred "R" [x,y],
                         Pred "R" [y,z]])
                        (Pred "R" [x,z]))))
```

```
Logic> form0
R[x,y]
Logic> form1
R[x,x]
Logic> form2
Ex R[x,x]
Logic> reflF
Ax R[x,x]
Logic> symmF
Ax Ay (R[x,y]==>R[y,x])
Logic> transF
Ax Ay Az (conj[R[x,y],R[y,z]]==>R[x,z])
```

Evaluation in a Model

Ingredients: interpretations and variable maps.

- An interpretation is a function from predicate names to appropriate relations in the model.
- A variable map is a function from variables to entities in the model.

Because predicates take lists of variables, we need interpretation functions of type `String -> [Entity] -> Bool`.

Semantics of Predicate Logic

A structure $M = (D, I)$ consisting of a non-empty domain D with an interpretation function for the predicate symbols of the language is called a model for the language.

Let $s : V \rightarrow D$ be a valuation function for the variables V of the language.

We use $s(v|d)$ for the valuation that is like s except for the fact that v gets value d (where s might have assigned a different value).

Example: Let $D = \{0, 1, 2, 3\}$, and let $V = \{v_0, v_1, v_2, v_3\}$.

Let s be given by $s(v_0) = 0, s(v_1) = 0, s(v_2) = 2, s(v_3) = 3$.

What is $s(v_1|1)$?

$M \models_s F$

State the cases where model $M = (D, I)$ and valuation s satisfy a formula F.

We use the notation $M \models_s F$ for this notion.

$M \models_s Pv$	if	$s(v) \in I(P)$
$M \models_s R(v_1, v_2)$	if	$(s(v_1), s(v_2)) \in I(R)$
$M \models_s S(v_1, v_2, v_3)$	if	$(s(v_1), s(v_2), s(v_3)) \in I(S)$
$M \models_s v_1 = v_2$	if	$s(v_1) = s(v_2)$
$M \models_s \neg F$	if	it is not the case that $M \models_s F$.
$M \models_s (F_1 \wedge F_2)$	if	$M \models_s F_1$ and $M \models_s F_2$
$M \models_s (F_1 \vee F_2)$	if	$M \models_s F_1$ or $M \models_s F_2$
$M \models_s \forall v F$	if	for all $d \in D$ it holds that $M \models_{s(v\|d)} F$
$M \models_s \exists v F$	if	for at least one $d \in D$ it holds that $M \models_{s(v\|d)} F$

Reformulation using Lists

$M \models_s P[v_1, \ldots, v_n]$	if	$[s(v_1), \ldots, s(v_n)] \in I(P)$
$M \models_s v_1 = v_2$	if	$s(v_1) = s(v_2)$
$M \models_s \neg F$	if	it is not the case that $M \models_s F$.
$M \models_s \wedge[F_1, \ldots, F_n]$	if	$M \models_s F_1$ and $\ldots$ and $M \models_s F_n$
$M \models_s \vee[F_1, \ldots, F_n]$	if	$M \models_s F_1$ or $\ldots$ or $M \models_s F_n$
$M \models_s \forall v F$	if	for all $d \in D$ it holds that $M \models_{s(v\|d)} F$
$M \models_s \exists v F$	if	for at least one $d \in D$ it holds that $M \models_{s(v\|d)} F$

Implementation

Straightforward, once we have captured the notion $s(v|d)$.

```
change :: (Var -> a) -> Var -> a -> Var -> a
change s x d = \ v -> if x == v then d else s v
```

Now `change s x d` is the implementation of $s(x|d)$.

As an example, here is a definition of the assignment that maps every variable to object `A`.

```
ass0 :: Var -> Entity
ass0 = \ v -> A
```

The function that is like `ass0`, except for the fact that `y` gets mapped to `B`, is given by:

```
ass1 :: Var -> Entity
ass1 = change ass0 y B
```

Assumptions about the domain of evaluation

- We will assume that tests for equality are possible on the domain. In Haskell terminology: the type `a` of our domain should in the class `Eq`.
- We will assume that the domain can be enumerated, and we will use the enumerated domain as an argument to the evaluation function.

Note that our example domain of entities `Entity` satisfies these requirements.

The domain of non-negative integers satisfies the requirements as well.

The requirements are not enough to guarantee termination of the evaluation function for all possible arguments.

Type of predicate logical evaluation function

```
evl :: Eq a =>
 [a]                           -- domain of discourse
  -> (String -> [a] -> Bool)   -- interpretation
  -> (Var -> a)                -- valuation
  -> Frm                       -- formula
  -> Bool                      -- outcome
```

```
evl dom i s (Pred str vs) = i str (map s vs)
evl dom i s (Eql v1 v2)   = (s v1) == (s v2)
evl dom i s (Neg f)       = not (evl dom i s f)
evl dom i s (Impl f1 f2)  =
   not ((evl dom i s f1) && not (evl dom i s f2))
evl dom i s (Equiv f1 f2)  =
   (evl dom i s f1) == (evl dom i s f2)
evl dom i s (Cnj fs)     = all (evl dom i s) fs
evl dom i s (Dsj fs)     = any (evl dom i s) fs
evl dom i s (Forall v f)  =
   all (\d -> evl dom i (change s v d) f) dom
evl dom i s (Exists v f)  =
   any (\d -> evl dom i (change s v d) f) dom
```

Example Interpretation Function

```
int0 :: String -> [Entity] -> Bool
int0 "P" = \ [x] -> laugh x
int0 "Q" = \ [x] -> smile x
int0 "R" = \ [x,y] -> love (x,y)
int0 "S" = \ [x,y] -> hate (x,y)
```

Note the use of lambda abstraction to convert types `Entity -> Bool` and `(Entity,Entity) -> Bool` to `[Entity] -> Bool`.

We can check the claim that various formulas make about the Entity domain, when R gets interpreted as the love relation, as follows:

```
Logic> filter love [(x,y) | x <- entities, y <- entities]
[(A,J),(B,J),(B,M),(J,J),(J,M),(M,J)]
Logic> map ass1 [x,y,z]
[A,B,A]

Logic> evl entities int0 ass1 form0
False
Logic> evl entities int0 ass1 form1
False
Logic> evl entities int0 ass1 form2
True
Logic> evl entities int0 ass1 reflF
False
```

Other interpretations

Let the domain of discourse be the integers, and let `int1` be the following interpretation for the predicates `"P"`, `"Q"`, `"R"` and `"S"`.

```
int1 :: String -> [Integer] -> Bool
int1 "P" = \ [x] -> even x
int1 "Q" = \ [x] -> x > 0
int1 "R" = \ [x,y] -> x < y
int1 "S" = \ [x,y] -> x <= y
```

```
ass2 :: Var -> Integer
ass2 v = if v == x then 1
         else if v == y then 2 else 0
```

What do the following queries express?

```
Logic> evl [0..] int1 ass2 form0
True
Logic> evl [0..] int1 ass2 form1
False
Logic> evl [0..] int1 ass2 form2
{Interrupted!}

Logic> evl [0..] int1 ass2 reflF
False
Logic> evl [0..] int1 ass2 symmF
False
Logic> evl [0..] int1 ass2 transF
{Interrupted!}
```

Decidability Questions

A yes/no question is decidable if there exists a method that tells us in a finite amount of time whether the answer to that question is 'yes' or 'no'.

"Is propositional formula F satisfiable" (is there a propositional valuation that makes F true) is a decidable question, for arbitrary formulas F. The truth table method is a decision method. The function `satisfiable` defined above always terminates.

"Is predicate logical formula F satisfiable" (are there a model and a variable assignment that make F true) is not a decidable question. There is no analogue to the truth table method for predicate logic.

A crucial difference between propositional logic and predicate logic is the number of relevant situations.

The number of relevant valuations for a propositional formula is always

finite: if n is the number of different proposition letters that occur in F, then the truth table for F will have 2^n rows, for there are 2^n different ways of evaluating n variables, and each valuation corresponds to a row in the truth table.

In the case of predicate logic, the number of possible models for a formula is infinite. There is no analogue to the truth table method for predicate logic.

There is the semantic tableau method (see the course textbook), but this is not a decision method: there are formulas for which the method does not terminate.