Computational Formal Semantics Notes: Part 1

Adrian Brasoveanu*

October 14, 2013

Contents

1 Syntax of propositional logic

We load the module that defines the syntax of propositional (technically speaking, sentential) logic.

```
ghci 1> :l PropLsyn
```

Note that this module imports the *Data.List* module:

(1) **import** *Data.List*

The *PropLsyn* module defines the recursive type of sentential formulas *Form*. The definition of *Form* includes the expected recursive value constructors:

- negation

```
ghci 2> :t Ng
Ng :: Form → Form
```

- conjunction (of a list of formulas, not just 2 formulas)

*Code based on *Computational Semantics with Functional Programming* by Jan van Eijck & Christina Unger, http://www.computational-semantics.eu.

```
ghci 3> :t Cnj
Cnj :: [Form] → Form
```

- disjunction (of a list of formulas)

```
ghci 4> :t Dsj
Dsj :: [Form] → Form
```

These formula constructors cover most of the definition of the syntax of sentential logic, which is particularly simple:

(2) The syntax of propositional logic:

```
data Form = P String | Ng Form | Cnj [Form] | Dsj [Form] deriving Eq
```

We have a value constructor for atomic formulas *P*, which takes a string, i.e., a formula name, as its only argument and returns an atomic formula with that name as the result:

```
ghci 5> :t P
P :: String → Form
```

In adition, we have the *Ng*, *Cnj* and *Dsj* constructors we already looked at.

Finally, we derive *Eq* for our *Form* type since we want to be able to determine when two formulas are identical *as syntactic objects*.

We manually derive the instace of *Show* for formulas so that we can display them in a more familiar format.

The recursive definition of the *show* function might seem a bit complicated – just remember that this is really just used for pretty-printing. The actual definition of the syntax of propositional logic is particularly straightforward and concise.

Nonetheless, make sure you understand the recursive definition of the *show* function for our *Form* data type, which is provided below for convenience:

(3)
```
instance Show Form where
  show (P name) = name
  show (Ng f) = "~ " ++ show f
  show (Cnj fs) = "(" ++ intercalate " /\\ " (map show fs) ++ ")"
  show (Dsj fs) = "(" ++ intercalate " \\/ " (map show fs) ++ ")"
```

The two example formulas below are predefined in the *PropLsyn* module as shown below:

(4)
```
form1,form2 :: Form
form1 = Cnj [P "p",Ng (P "p")]
form2 = Dsj [P "p1",P "p2",P "p3",P "p4"]
```

```
ghci 6> form1
(p ∧ ~p)
```

```
ghci 7> form2
    (p1 ∨ p2 ∨ p3 ∨ p4)
```

We can define more formulas, both atomic and non-atomic:

```
ghci 8> let form3 = P "jake likes chocolate"
```

```
ghci 9> form3
    jake likes chocolate
```

```
ghci 10> let form4 = P "sam likes vanilla"
```

```
ghci 11> form4
    sam likes vanilla
```

```
ghci 12> let form5 = Ng form3
```

```
ghci 13> form5
    ∼jake likes chocolate
```

```
ghci 14> let form6 = Ng $ Dsj [form3,form4]
```

```
ghci 15> form6
    ∼(jake likes chocolate ∨ sam likes vanilla)
```

```
ghci 16> let form7 = Ng $ Cnj [form5,form6]
```

```
ghci 17> form7
    ∼(∼jake likes chocolate ∧ ∼(jake likes chocolate ∨ sam likes vanilla))
```

2 Semantics of propositional logic

We load the module that defines the semantics for our propositional logic system:

```
ghci 18> :l PropLsem
```

Note that this module imports the syntax module *PropLsyn* right at the very beginning (in addition to the *Data.List* module):

(5) **import** *Data.List*
import *PropLsyn*

Every time a module is (re)loaded in `ghci`, the namespace is completely erased and restarted – so we redefine the formulas we introduced above:

```
ghci 19> let {form3 = P "jake likes chocolate";
              form4 = P "sam likes vanilla";
              form5 = Ng form3;
              form6 = Ng $ Dsj [form3,form4];
              form7 = Ng $ Cnj [form5,form6] }
```

We want to assign semantic values, i.e., *True* or *False*, to formulas like these. Such formulas have a semantic value relative to a model that assigns semantic values (i.e., truth values) to the atomic propositions in these formulas.

Propositional logic is so simple that we can actually generate all possible models for any given formula and then evaluate that formula relative to any / all of those models.

To be able to generate a model for an arbitrary propositional formula, we first need to have access to the atomic propositions that it contains.

So we have a helper function that extracts the names of all the atomic propositions that occur as sub-formulas in any input formula:

(6)
$$\begin{aligned}
&propNames :: Form \to [String] \\
&propNames\ (P\ name) = [name] \\
&propNames\ (Ng\ f) = propNames\ f \\
&propNames\ (Cnj\ fs) = sort \circ nub \circ concat\ \$\ map\ propNames\ fs \\
&propNames\ (Dsj\ fs) = sort \circ nub \circ concat\ \$\ map\ propNames\ fs
\end{aligned}$$

For example, consider the formula below:

```
ghci 20> form5
    ~jake likes chocolate
```

It contains only one atomic proposition, namely:

```
ghci 21> propNames form5
    ["jake likes chocolate"]
```

We can do the same for more complicated formulas, e.g.:

ghci 22> *form7*
$\sim(\sim$*jake likes chocolate* $\wedge\ \sim($*jake likes chocolate* $\vee$ *sam likes vanilla*$))$

ghci 23> *propNames form7*
`["jake likes chocolate","sam likes vanilla"]`

Given a list of atom names, we can generate all the possible models, a.k.a. valuations, for that list:

- every single atom name can be paired with either *True* or *False*;
- a model / valuation is a list of (name, truth value) pairs such that the name of each atom occurs in exactly one pair;
- to generate all models / valuations for a list of atomic proposition names, we simply need to generate all the possible lists of (name, truth value) pairs of this kind.

And this is exactly what the recursive *genVals* function does:

(7) $genVals :: [String] \to [[(String, Bool)]]$
$genVals\ [\,] = [[\,]]$
$genVals\ (name : names) = map\ ((name, True):)\ (genVals\ names) +\!\!+$
$map\ ((name, False):)\ (genVals\ names)$

Take a minute and make sure you understand the definition of *genVals* properly. Do you see why *map* is used and what it does?

We can now generate all the possible models / valuations for any formula by first extracting the list of atomic propositions in that formula and then generating all the models for that list of atoms:

(8) $allVals :: Form \to [[(String, Bool)]]$
$allVals = genVals \circ propNames$

For example:

ghci 24> *form5*
$\sim$*jake likes chocolate*

ghci 25> *allVals form5*
`[[("jake likes chocolate",True)],[("jake likes chocolate",False)]]`

And here's another example:

ghci 26> *form7*
$\sim(\sim$*jake likes chocolate* $\wedge\ \sim($*jake likes chocolate* $\vee$ *sam likes vanilla*$))$

ghci 27> *allVals form7*
`[[("jake likes chocolate",True),("sam likes vanilla",True)],`
`[("jake likes chocolate",True),("sam likes vanilla",False)],`
`[("jake likes chocolate",False),("sam likes vanilla",True)],`
`[("jake likes chocolate",False),("sam likes vanilla",False)]]`

Let's take *form7* and choose one model:

```
ghci 28> let models_form7 = allVals form7
```

```
ghci 29> models_form7
    [[("jake likes chocolate",True),("sam likes vanilla",True)],
    [("jake likes chocolate",True),("sam likes vanilla",False)],
    [("jake likes chocolate",False),("sam likes vanilla",True)],
    [("jake likes chocolate",False),("sam likes vanilla",False)]]
```

```
ghci 30> let model3_form7 = models_form7 !! 2
```

```
ghci 31> model3_form7
    [("jake likes chocolate",False),("sam likes vanilla",True)]
```

We can now evaluate *form7* relative to this model:

```
ghci 32> eval model3_form7 form7
    True
```

The recursive definition of the *eval* function is actually the definition of the semantics of our propositional logic system:

(9) The semantics of propositional logic:

```
eval :: [(String, Bool)] → Form → Bool
eval [] (P name) = error ("no info about " ++ show name)
eval ((atom, value) : xs) (P name)
    | name ≡ atom = value
    | otherwise = eval xs (P name)
eval xs (Ng f) = ¬ (eval xs f)
eval xs (Cnj fs) = all (eval xs) fs
eval xs (Dsj fs) = any (eval xs) fs
```

Thus, the recursive definition of truth for our propositional logic system is given by the interpretation function *eval*, which takes a model of type $[(String, Bool)]$ and a formula (of type *Form*) as arguments and returns a truth value (of type *Bool*).

The first thing to note about the definition of *eval* is how closely it tracks the definition of the syntax of our propositional logic system, in particular, how the clauses pattern match against the four value constructors *P* (which builds atomic propositions), *Ng* (negative formulas), *Cnj* (conjunctions) and *Dsj* (disjunctions).

Since the recursive interpretation function *eval* takes two arguments – a model and a formula – we have two edge conditions. Both of them look at the edge / base condition for formulas, i.e., both of them are about atomic formulas of the form *P name*.

Let's start with the second edge condition, namely:

(10) $eval\ ((atom, value) : xs)\ (P\ name)$
$\quad\mid name \equiv atom = value$
$\quad\mid otherwise = eval\ xs\ (P\ name)$

This condition assumes that our model is a non-empty list of (atom name, truth value) pairs. We extract the head of this list, i.e., the initial $(atom, value)$ pair and then check two cases:

- if that pair happens to evaluate the atomic proposition we're trying to evaluate, i.e., if $name \equiv atom$, then the atomic proposition receives the semantic value that the model assigns to it, namely *value*;
- but if the head $(atom, value)$ pair of our model does not assign a semantic value to our atomic proposition $P\ name$, we keep looking for the semantic value of this atomic proposition in the rest of our model – hence the recursive call $eval\ xs\ (P\ name)$;

We keep looking for our atomic proposition $P\ name$ in the model from left to right, one $(atom, value)$ pair at a time. If our model is incomplete and does not provide a semantic value for that atomic proposition (either because the model is empty to begin with or just because that particular atomic proposition is not included in the model), we will keep discarding pairs from our model until we reach the empty model $[\,]$.

At that point, the first edge condition kicks in and throws an error that lets us know which atomic proposition was not included in the model:

(11) $eval\ [\,]\ (P\ name) = error\ ($`"no info about "` $+\!\!+\ show\ name)$

For example, evaluating relative to the null model gives an error:

ghci 33> *form7*
$\sim(\sim jake\ likes\ chocolate \land \sim(jake\ likes\ chocolate \lor sam\ likes\ vanilla))$

ghci 34> $eval\ [\,]\ form7$
$***Exception: no\ info\ about$ `"jake likes chocolate"`

And we also get an error if we have info only about some of the atomic propositions in our formula:

ghci 35> $eval\ [($`"jake likes chocolate"`$, False)]\ form7$
$***Exception: no\ info\ about$ `"sam likes vanilla"`

The remaining three recursive clauses in the definition of *eval* deal with non-atomic propositions, which can be formed with any of the three value constructors *Ng*, *Cnj* and *Dsj*. We assign semantic values to these formulas in the expected way and in the process, we leverage the fact that Haskell makes available several Boolean operators.

We'll examine these final three clauses in turn.

First, if we have a negative formula, i.e., a formula of the form $Ng\ f$, its semantic value is obtained by negating the semantic value of its immediate subformula f:

(12) $eval\ xs\ (Ng\ f) = \neg\ (eval\ xs\ f)$

We obtain the semantic value of the immediate subformula by recursively calling *eval* with the same model *xs*, i.e., $eval\ xs\ f$. We negate that truth value with the operator $\neg$ that Haskell provides:

```
ghci 36> :i ¬
¬ :: Bool → Bool   -- Defined in 'GHC.Classes'
```

```
ghci 37> :! hoogle   -- info not
Prelude not :: Bool -> Bool

Boolean "not"

From package base not :: Bool -> Bool
```

If we have a conjunction of a list of formulas, its semantic value is *True* if all the subformulas in the list evaluate to *True*, otherwise the conjunction is *False*:

(13) $eval\ xs\ (Cnj\ fs) = all\ (eval\ xs)\ fs$

We implement this by recursively calling *eval* with the same model *xs* on all the subformulas in the list of *fs*, and checking that they are all *True* with the operator *all* that Haskell provides:

```
ghci 38> :i all
all :: (a → Bool) → [a] → Bool   -- Defined in 'GHC.List'
```

```
ghci 39> :! hoogle   -- info all
Prelude all :: (a -> Bool) -> [a] -> Bool

Applied to a predicate and a list, all determines if all elements of the list satisfy the predicate. For the result to be True, the list must be finite; False, however, results from a False value for the predicate applied to an element at a finite index of a finite or infinite list.

From package base all :: (a -> Bool) -> [a] -> Bool
```

Finally, if we have a disjunction of a list of formulas, its semantic value is *True* if any subformula in the list evaluates to *True*, otherwise the disjunction is *False*:

(14) $eval\ xs\ (Dsj\ fs) = any\ (eval\ xs)\ fs$

Just as for conjunctions, we implement this by recursively calling *eval* with the same model *xs* on all the subformulas in the list of *fs*, and checking that at least one of them is *True* with the operator *any* that Haskell provides:

```
ghci 40> :i any
any :: (a → Bool) → [a] → Bool   -- Defined in 'GHC.List'
```

```
ghci 41> :! hoogle   -- info any
Prelude any :: (a -> Bool) -> [a] -> Bool

Applied to a predicate and a list, any determines if any element of the list satisfies the predicate. For the result to be False, the list must be finite; True, however, results from a True value for the predicate applied to an element at a finite index of a finite or infinite list.

From package base any :: (a -> Bool) -> [a] -> Bool
```

We can now understand in detail how *form7*, for example, is evaluated relative to a model:

```
ghci 42> form7
    ∼(∼jake likes chocolate ∧ ∼(jake likes chocolate ∨ sam likes vanilla))
```

```
ghci 43> model3_form7
    [("jake likes chocolate",False),("sam likes vanilla",True)]
```

```
ghci 44> eval model3_form7 form7
    True
```

3 Tautologies, satisfiability and contradictions

Because we can generate all models for a formula, we can easily check if the formula is a tautology (true in any model), satisfiable (true in at least one model) or a contradiction (true in no model / unsatisfiable).

The definition of the *tautology* predicate is provided below:

(15) $tautology :: Form \to Bool$
$tautology\ f = all\ (\lambda v \to eval\ v\ f)\ (allVals\ f)$

tautology is a function of type $Form \to Bool$: it takes a formula f as its argument and it returns *True* if that formula is a tautology and *False* otherwise. We check this by:

- generating the list of all the models for the formula f with $allVals\ f$;
- checking that for each model / valuation v in this list, when we evaluate f relative to v, we get *True*; this is achieved by the function $\lambda v \to eval\ v\ f$, together with the operator *all*.

And here are two examples:

```
ghci 45> tautology form7
    False
```

```
ghci 46> tautology $ Dsj [form7,Ng form7]
    True
```

The definition of the *satisfiable* predicate is exactly like the definition of *tautology* except the operator *all* is replaced with *any*:

(16) $satisfiable :: Form \to Bool$
$satisfiable\ f = any\ (\lambda v \to eval\ v\ f)\ (allVals\ f)$

And the *contradiction* predicate is defined as being unsatisfiable:

(17) $contradiction :: Form \to Bool$
$contradiction = \neg \circ satisfiable$

Here are several examples:

```
ghci 47> satisfiable form7
True
```

```
ghci 48> satisfiable $ Dsj [form7, Ng form7]
True
```

```
ghci 49> satisfiable $ Cnj [form7, Ng form7]
False
```

```
ghci 50> contradiction form7
False
```

```
ghci 51> contradiction $ Dsj [form7, Ng form7]
False
```

```
ghci 52> contradiction $ Cnj [form7, Ng form7]
True
```

4 Checking for entailment between two formulas

We can also test if a formula entails another formula, i.e., if there is a subset relation between the models that satisfy them.

The way we chose to define this *implies* predicate is in terms of *contradiction* / unsatisfiability: a formula φ entails a formula ψ iff conjoining φ and $\neg\psi$ is a contradiction.

(18)
```
implies :: Form → Form → Bool
implies f1 f2 = contradiction (Cnj [f1, Ng f2])
```

Optional homework: why do we choose to define entailment like this rather than directly in terms of a subset relation between the sets of models that satisfy the two formulas? Hint: what kind of models do we generate if one of the formulas contains an atomic proposition that the other doesn't, e.g., we check if φ entails $\varphi \vee \psi$?

And here are a couple of examples:

```
ghci 53> implies form7 (Dsj [form7, Ng form7])
True
```

```
ghci 54> implies form7 (Cnj [form7, Ng form7])
False
```

5 Context Set update

Finally, we can model the update contributed by each formula relative to a Stalnaker-style Context Set, i.e., relative to the set of models / valuations that are our current candidates for the actual model.

We define a function *update* that takes as arguments:

- the current Context Set, i.e., the set of models that we currently think are live candidates for the actual model (real-world situation); this is a list of models, and since each model is a list of $(name, value)$ pairs, Context Sets are lists of lists of pairs, i.e., they are of type $[[(String, Bool)]]$;
- the formula we are updating with, i.e., the formula containing the new factual information we just learned.

The *update* function then returns a smaller Context Set containing only the models in the input Context Set that satisfy the formula we're updating with:

(19) $update :: [[(String, Bool)]] \to Form \to [[(String, Bool)]]$
$update\ vals\ f = [v \mid v \leftarrow vals, eval\ v\ f]$

For example:

```
ghci 55> models_form7
    [[("jake likes chocolate",True),("sam likes vanilla",True)],
    [("jake likes chocolate",True),("sam likes vanilla",False)],
    [("jake likes chocolate",False),("sam likes vanilla",True)],
    [("jake likes chocolate",False),("sam likes vanilla",False)]]
```

```
ghci 56> update models_form7 form7
    [[("jake likes chocolate",True),("sam likes vanilla",True)],
    [("jake likes chocolate",True),("sam likes vanilla",False)],
    [("jake likes chocolate",False),("sam likes vanilla",True)]]
```

```
ghci 57> length $ models_form7
    4
```

```
ghci 58> length $ update models_form7 form7
    3
```

Here's another example in which we update a singleton Context Set:

```
ghci 59> [model3_form7]
    [[("jake likes chocolate",False),("sam likes vanilla",True)]]
```

```
ghci 60> update [model3_form7] form7
    [[("jake likes chocolate",False),("sam likes vanilla",True)]]
```

And here's a final example in which we try to update the empty Context Set:

```
ghci 61> update [] form7
    []
```