Computational Formal Semantics Notes: Part 3

Adrian Brasoveanu*

November 14, 2013

Contents

1 The syntax of our english fragment

```
ghci 1> :l EF1syn
```

We have proper names:

```
ghci 2> :t ALICE
ALICE :: NP
```

We also have determiners and CNs that we can put together to form NPs:

```
ghci 3> :t Every
Every :: DET
```

```
ghci 4> :t Boy
Boy :: CN
```

```
ghci 5> :t NP1 Every Boy
NP1 Every Boy :: NP
```

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

```
ghci 6> :t NP1 A Sword
NP1 A Sword :: NP
```

We can also add adjectives:

```
ghci 7> :t Fake
Fake :: ADJ
```

```
ghci 8> :t RCN3 Fake Sword
RCN3 Fake Sword :: RCN
```

```
ghci 9> :t NP2 No (RCN3 Fake Sword)
NP2 No (RCN3 Fake Sword) :: NP
```

We also have intransitive verbs that form VPs directly, transitive verbs and ditransitive verbs:

```
ghci 10> :t Laughed
Laughed :: VP
```

```
ghci 11> :t VP1 Helped (NP1 Every Boy)
VP1 Helped (NP1 Every Boy) :: VP
```

```
ghci 12> :t VP2 Gave (NP1 Every Boy) (NP1 A Sword)
VP2 Gave (NP1 Every Boy) (NP1 A Sword) :: VP
```

We can form sentences with these NPs and VPs:

```
ghci 13> :t Sent (NP1 No Girl) Laughed
Sent (NP1 No Girl) Laughed :: Sent
```

```
ghci 14> :t Sent (NP1 No Girl) (VP1 Helped (NP1 Every Boy))
Sent (NP1 No Girl) (VP1 Helped (NP1 Every Boy)) :: Sent
```

```
ghci 15> :t Sent (NP1 No Girl) (VP2 Gave (NP1 Every Boy) (NP1 A Sword))
Sent (NP1 No Girl) (VP2 Gave (NP1 Every Boy) (NP1 A Sword)) :: Sent
```

We have relative clauses in our fragment, both with a subject gap ...

```
ghci 16> :t NP2 Every (RCN1 Boy That Laughed)
NP2 Every (RCN1 Boy That Laughed) :: NP
```

```
ghci 17> :t Sent (NP2 Every (RCN1 Boy That Laughed)) Smiled
Sent (NP2 Every (RCN1 Boy That Laughed)) Smiled :: Sent
```

...and with a direct object gap

```
ghci 18> :t NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)
NP2 Every (RCN2 Boy That (NP1 A Girl) Loved) :: NP
```

```
ghci 19> :t Sent (NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)) Smiled
Sent (NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)) Smiled :: Sent
```

2 Semantics, part 1: English-to-FOL translation

```
ghci 20> :l EF1sem
```

We define an indirect interpretation for our fragment of English, i.e., we define a (compositional) translation function from the sentences in our Eng. fragment into the first-order logic we defined previously. The Eng. sentences indirectly receive an interpretation: their meaning is the meaning of the FOL formulas they get translated into.

Our translation involves higher-order lambda terms exactly as it does in Montague's PTQ. However, those terms do not have an interpretation, the only model that we have is the FO model we had for FOL.

We don't have to do things this way, we could generalize our FOL model and assign interpretations to all the intermediate, higher-order lambda terms that are produced by our translation function. This is in fact what we will do soon, but it's instructive to do things both ways.

The translation function is compositionally defined based on the above syntax of the Eng. fragment. We have a translation function associated with every syntactic category, that is, for every syntactic tree whose mother node is of that syntactic category, its translation is a function of:

i. the translations of its immediate daughters

ii. the way they are syntactically put together

(this is the textbook definition of compositionality)

Our FOL syntax has 2 basic types of expressions: terms and formulas.

```
ghci 21> :i Term
data Term = Var Name Index | Struct Name [Term]   -- Defined at PredLsyn.hs:7:6 instance Eq Term – Defined at
```

```
ghci 22> :i Formula
  data Formula = Atom Name [Term] | Eq Term Term | Neg Formula | Impl Formula Formula |
  Equi Formula Formula | Conj [Formula] | Disj [Formula] | Forall Term Formula |
  Exists Term Formula   -- Defined at PredLsyn.hs:24:6 instance Eq Formula – Defined at PredLsyn.hs:24:195 insta
```

Therefore, all the translations of Eng. expressions will be terms, formulas or higher-order functions over the domains of FOL terms and formulas. In extensional Montague semantics, we have two basic types e (for entities) and t (for truth values) based on which we define arbitrary higher-order functional types. In the translations we define here, *Term* basically functions as type e and *Formula* is basically type t.

Let's take a CN, e.g., *Boy*: its Montagovian translation has type (et), a function from entities to truth values. In our implementation, its translation will be a function from *Terms* to *Formulas*:

```
ghci 23> :t lfCN Boy
  lfCN Boy :: Term → Formula
```

And the translation function itself, i.e., *lfCN*, will be a function from *CNs* to *Term* → *Formula* functions:

```
ghci 24> :t lfCN
  lfCN :: CN → Term → Formula
```

Incidentally, you can get info about where all these are defined like so:

```
ghci 25> :i lfCN Boy
  lfCN :: CN → Term → Formula   -- Defined at EF1sem.hs:23:1 data CN = ... Boy ... – Defined at EF1syn.hs:8:22
```

For example, the definition of *lfCN* in the *EF1sem* (English Fragment 1 Semantics) module is:

```
lfCN :: CN → Term → Formula
...
lfCN Boy = λt → Atom "boy" [t]
...
```

The formulas on the rhs are FOL formulas and their interpretation is defined in the *PredLsem* module as follows:

```
eval :: Eq a ⇒
  [a]        →
  Interp a   →
  Assignment a →
  Formula → Bool
eval domain i = eval' where
  eval' g (Atom str vs) = i str (map g vs)
  ...
```

That is, we take the string `"boy"` and see what semantic value is assigned to it by the interpretation i. We take the term t in the singleton list $[t]$ that is the argument of `"boy"` and see what semantic value t is assigned by the variable assignment g. Finally, we check whether the semantic value of t is in the semantic value of `"boy"`.

We translate proper names as the Montagovian lifts of the corresponding FOL constant, i.e., their translation is of quantifier type:

ghci 26> $:t\ lfNP\ ALICE$
$lfNP\ ALICE :: (Term \to Formula) \to Formula$

$lfNP\ ALICE = \lambda p \to p\ (Struct\ \texttt{"Alice"}\ [\,])$

Determiners and NP headed by determiners have translations of the expected Montague-style types:

ghci 27> $:t\ lfDET\ Every$
$lfDET\ Every :: (Term \to Formula) \to (Term \to Formula) \to Formula$

ghci 28> $:t\ lfNP\ \$\ NP1\ Every\ Boy$
$lfNP\ \$\ NP1\ Every\ Boy :: (Term \to Formula) \to Formula$

ghci 29> $:t\ lfNP\ \$\ NP1\ A\ Sword$
$lfNP\ \$\ NP1\ A\ Sword :: (Term \to Formula) \to Formula$

The translations for VPs containing intrasitive, transitive and ditransive verbs also have the expected Montagovian form. For example:

ghci 30> $:t\ lfVP\ Laughed$
$lfVP\ Laughed :: Term \to Formula$

ghci 31> $:t\ lfVP\ \$\ VP1\ Helped\ (NP1\ Every\ Boy)$
$lfVP\ \$\ VP1\ Helped\ (NP1\ Every\ Boy) :: Term \to Formula$

ghci 32> $:t\ lfVP\ \$\ VP2\ Gave\ (NP1\ Every\ Boy)\ (NP1\ A\ Sword)$
$lfVP\ \$\ VP2\ Gave\ (NP1\ Every\ Boy)\ (NP1\ A\ Sword) :: Term \to Formula$

To avoid accidental binding of variables when we translate quantificational determiners, we are always careful to introduce a fresh variable – we do this by defining three helper functions:

i. *bInLFs*, which identifies the indices on the variables already present in the formulas we want to quantify over

ii. *freshIndex*, which produces a variable index that is different from any of the indices in an arbitrary list of indices

iii. *fresh*, which takes a list of predicates (functions from *Terms* to *Formulas*), and returns an fresh index, i.e., a variable index different from any of the current variable indices

We can now translate full sentences:

```
ghci 33> :t lfSent $ Sent (NP1 No Girl) Laughed
lfSent $ Sent (NP1 No Girl) Laughed :: Formula
```

```
ghci 34> :t lfSent $ Sent (NP1 No Girl) (VP1 Helped (NP1 Every Boy))
lfSent $ Sent (NP1 No Girl) (VP1 Helped (NP1 Every Boy)) :: Formula
```

```
ghci 35> :t lfSent $ Sent (NP1 No Girl) (VP2 Gave (NP1 Every Boy) (NP1 A Sword))
lfSent $ Sent (NP1 No Girl) (VP2 Gave (NP1 Every Boy) (NP1 A Sword)) :: Formula
```

Finally, restrictive relative clauses with a subject or object gap are also translated in the expected Montagovian way:

```
ghci 36> :t lfRCN $ RCN1 Boy That Laughed
lfRCN $ RCN1 Boy That Laughed :: Term → Formula
```

```
ghci 37> :t lfNP $ NP2 Every (RCN1 Boy That Laughed)
lfNP $ NP2 Every (RCN1 Boy That Laughed) :: (Term → Formula) → Formula
```

```
ghci 38> :t lfSent $ Sent (NP2 Every (RCN1 Boy That Laughed)) Smiled
lfSent $ Sent (NP2 Every (RCN1 Boy That Laughed)) Smiled :: Formula
```

```
ghci 39> :t lfRCN $ RCN2 Boy That (NP1 A Girl) Loved
lfRCN $ RCN2 Boy That (NP1 A Girl) Loved :: Term → Formula
```

```
ghci 40> :t lfNP $ NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)
lfNP $ NP2 Every (RCN2 Boy That (NP1 A Girl) Loved) :: (Term → Formula) → Formula
```

```
ghci 41> :t lfSent $ Sent (NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)) Smiled
lfSent $ Sent (NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)) Smiled :: Formula
```

3 Semantics, part 2: Evaluation in a model (model checking)

```
ghci 42> :l EF1sem
```

- the set of boys in the model is {*LittleMook*, *Atreyu*}

- the set of girls in the model is $\{SnowWhite, Alice, Dorothy, Goldilocks\}$
- the set of love-pairs in the model is $\{(Atreyu, Ellie), (Bob, SnowWhite), (Remmy, SnowWhite), (SnowWhite, LittleMook)\}$
- the set of smilers in the model is $\{Alice, Bob, Cyrus, Dorothy, Ellie, Fred, Goldilocks, LittleMook\}$

Therefore, *Every boy that a girl loved smiled* is true b/c *LittleMook* is the only boy loved by a girl and *LittleMook* is in the set of smilers:

```
ghci 43> eval entities int0 ass0
           (lfSent $ Sent (NP2 Every (RCN2 Boy That (NP1 A Girl) Loved)) Smiled)
    True
```

And *No boy that a girl loved smiled* is false:

```
ghci 44> eval entities int0 ass0
           (lfSent $ Sent (NP2 No (RCN2 Boy That (NP1 A Girl) Loved)) Smiled)
    False
```

An example using name constants:

```
ghci 45> evl entities int0 fint0 ass0 (lfSent $ Sent SNOWWHITE (VP1 Loved LITTLEMOOK))
    True
```