An Inference Engine with a Natural Language Interface

Jan van Eijck, CWI

GLLC-16 Workshop on Generalized Quantifiers, 6 March 2009

Overview

- We present a natural language engine for talking about classes.
- The example is taken from Jan van Eijck and Christina Unger, Computational Semantics with Functional Programming, Cambridge University Press 2009 (to appear).
- http://www.cwi.nl/~jve/cs/
- Demo
- A tentative connection with cognitive realities.

The Simplest Natural Language Engine You Can Get

Questions and Statements (PN for plural nouns):

$$
\begin{array}{rcl}
Q & ::= & \text{Are all PN PN?} \\
 & | & \text{Are no PN PN?} \\
 & | & \text{Are any PN PN?} \\
 & | & \text{Are any PN not PN?} \\
 & | & \text{What about PN?}
\end{array}
$$

$$
\begin{array}{rcl}
S & ::= & \text{All PN are PN.} \\
 & | & \text{No PN are PN.} \\
 & | & \text{Some PN are PN.} \\
 & | & \text{Some PN are not PN.}
\end{array}
$$

The Simplest Knowledge Base You Can Get

The two relations we are going to model in the knowledge base are that of inclusion $\subseteq$ and that of non-inclusion $\not\subseteq$.

'all A are B' $\rightsquigarrow A \subseteq B$

'no A are B' $\rightsquigarrow A \subseteq \overline{B}$

'some A are not B' $\rightsquigarrow A \not\subseteq B$

'some A are B' $\rightsquigarrow A \not\subseteq \overline{B}$ (equivalently: $A \cap B \neq \emptyset$).

A **knowledge base** is a list of triples

$$(\text{Class}_1, \text{Class}_2, \text{Boolean})$$

where $(A, B, \top)$ expresses that $A \subseteq B$,

and $(A, B, \bot)$ expresses that $A \not\subseteq B$.

Rules of the Inference Engine

Let $\widetilde{A}$ be given by: if A is of the form $\overline{C}$ then $\widetilde{A} = C$, otherwise $\widetilde{A} = \overline{A}$. Let $A \Longrightarrow B$ express $A \subseteq B$. Let $A \not\Longrightarrow B$ express $A \not\subseteq B$.

Computing the subset relation from the knowledge base:

$$\frac{A \Longrightarrow B}{\widetilde{B} \Longrightarrow \widetilde{A}} \qquad \frac{A \Longrightarrow B \qquad B \Longrightarrow C}{A \Longrightarrow C}$$

Computing the non-subset relation from the knowledge base:

$$\frac{A \not\Longrightarrow B}{\widetilde{B} \not\Longrightarrow \widetilde{A}} \qquad \frac{A \Longleftarrow B \quad B \not\Longrightarrow C \qquad C \Longleftarrow D}{A \not\Longrightarrow D}$$

Reflexivity and existential import:

$$\frac{}{A \Longrightarrow A} \qquad A \text{ not of the form } \overline{C}\,\frac{}{A \not\Longrightarrow \widetilde{A}}$$

Implementation (in Haskell)

We will need list and character processing, and we want to read natural language sentences from a file, so we import the I/O-module `System.IO`.

```
import List
import Char
import System.IO
```

In our Haskell implementation we can use `[(a,a)]` for relations.

```
type Rel a = [(a,a)]
```

If $R \subseteq A^2$ and $x \in A$, then $xR := \{y \mid (x, y) \in R\}$.

```
rSection :: Eq a => a -> Rel a -> [a]
rSection x r = [ y | (z,y) <- r, x == z ]
```

`Eq a` indicates that a is in the equality class.

The composition of two relations R and S on A.

```
(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s = nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Computation of reflexive transitive closure using a function for least fixpoint. $\mathsf{RTC}(R) = \mathsf{lfp}(\lambda S.S \cup R \cdot S)I$.

```
rtc :: Ord a => [a] -> Rel a -> Rel a
rtc xs r = lfp (\ s -> (sort.nub) (s ++ (r@@s))) i
  where i = [(x,x) | x <- xs ]

lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Assume that each class has an opposite class. The opposite of an opposite class is the class itself.

```
data Class = Class String | OppClass String
             deriving (Eq,Ord)

instance Show Class where
  show (Class xs)    = xs
  show (OppClass xs) = "non-" ++ xs

opp :: Class -> Class
opp (Class name)    = OppClass name
opp (OppClass name) = Class name
```

Declaration of the knowledge base:

```
type KB = [(Class, Class, Bool)]
```

A data types for statements and queries:

```
data Statement =
     All  Class   Class | No      Class Class
   | Some Class   Class | SomeNot Class Class
   | AreAll Class Class | AreNo   Class Class
   | AreAny Class Class | AnyNot  Class Class
   | What   Class
  deriving Eq
```

Show function for statements:

```
instance Show Statement where
  show (All as bs) =
    "All " ++ show as ++ " are " ++ show bs ++ "."
  show (No as bs)  =
    "No " ++ show as ++ " are " ++ show bs ++ "."
  show (Some as bs)   =
    "Some " ++ show as ++ " are " ++ show bs ++ "."
  show (SomeNot as bs) =
    "Some " ++ show as ++ " are not " ++ show bs ++ "."
```

and for queries:

```
show (AreAll as bs) =
  "Are all " ++ show as ++ show bs ++ "?"
show (AreNo as bs) =
  "Are no " ++ show as ++ show bs ++ "?"
show (AreAny as bs) =
  "Are any " ++ show as ++ show bs ++ "?"
show (AnyNot as bs) =
  "Are any " ++ show as ++ " not " ++ show bs ++ "?"
show (What as) =
  "What about " ++ show as ++ "?"
```

Classification of statements:

```
isQuery :: Statement -> Bool
isQuery (AreAll _ _)  = True
isQuery (AreNo _ _)   = True
isQuery (AreAny _ _)  = True
isQuery (AnyNot _ _)  = True
isQuery (What _)      = True
isQuery  _            = False
```

Negations of queries:

```
neg :: Statement -> Statement
neg (AreAll as bs) = AnyNot as bs
neg (AreNo as bs)  = AreAny as bs
neg (AreAny as bs) = AreNo as bs
neg (AnyNot as bs) = AreAll as bs
```

Use the transitive closure operation to compute the subset relation from the knowledge base.

```
subsetRel :: KB -> [(Class,Class)]
subsetRel kb = rtc
  (domain kb) ([(x,y)          | (x,y,True) <- kb ]
           ++  [(opp y,opp x) | (x,y,True) <- kb ])
```

The supersets of a particular class are given by a right section of the subset relation. I.e. the supersets of a class are all classes of which it is a subset.

```
supersets :: Class -> KB -> [Class]
supersets cl kb = rSection cl (subsetRel kb)
```

Computing the non-subset relation from the knowledge base:

```
nsubsetRel :: KB -> [(Class,Class)]
nsubsetRel kb =
  let
   r = nub ([(x,y) | (x,y,False) <- kb ]
         ++ [(opp y,opp x) | (x,y,False) <- kb ]
         ++ [(Class xs,OppClass xs) |
                   (Class xs,_,_)      <- kb ]
         ++ [(Class ys,OppClass ys) |
                   (_,Class ys,_)      <- kb ]
         ++ [(Class ys,OppClass ys) |
                   (_,OppClass ys,_)   <- kb ])
   s = [(y,x) | (x,y) <- subsetRel kb ]
  in s @@ r @@ s
```

The non-supersets of a class:

```
nsupersets :: Class -> KB -> [Class]
nsupersets cl kb = rSection cl (nsubsetRel kb)
```

Query of a knowledge base by means of yes/no questions is simple:

```
deriv :: KB -> Statement -> Bool
deriv kb (AreAll as bs) = elem bs (supersets as kb)
deriv kb (AreNo  as bs) = elem (opp bs) (supersets as kb)
deriv kb (AreAny as bs) = elem (opp bs) (nsupersets as kb)
deriv kb (AnyNot as bs) = elem bs (nsupersets as kb)
```

Caution: there are three possibilities:

- `deriv kb stmt` is true. This means that the statement is derivable, hence true.
- `deriv kb (neg stmt)` is true. This means that the negation of `stmt` is derivable, hence true. So `stmt` is false.
- neither `deriv kb stmt` nor `deriv kb (neg stmt)` is true. This means that the knowledge base has no information about `stmt`.

Open queries ("How about A?") are slightly more complicated.

We should take care to select the most natural statements to report on a class:

$A \subseteq B$ is expressed with 'all',

$A \subseteq \overline{B}$ is expressed with 'no',

$A \not\subseteq B$ is expressed with 'some not',

$A \not\subseteq \overline{B}$ is expressed with 'some'.

```
f2s :: (Class, Class, Bool) -> Statement
f2s (as, Class bs, True)      = All as (Class bs)
f2s (as, OppClass bs, True)   = No as (Class bs)
f2s (as, OppClass bs, False) = Some as (Class bs)
f2s (as, Class bs, False)     = SomeNot as (Class bs)
```

Get the domain of a knowledge base:

```
domain :: [(Class,Class,Bool)] -> [Class]
domain = nub . dom where
  dom [] = []
  dom ((xs, ys, _):facts) =
      xs : opp xs : ys : opp ys : dom facts
```

Check whether a class A is mentioned in the knowledge base:

```
mention :: Class -> (Class, Class, Bool) -> Bool
mention xs (ys, zs, _) =
   elem xs [ys,zs] || elem (opp xs) [ys,zs]
```

Filter the facts from the knowledge base that mention a class A:

```
filterKB :: Class -> KB -> KB
filterKB xs = filter (mention xs)
```

Tell about a class A by listing what the knowledge base says about A:

```
tellAbout1 :: KB -> Class -> [Statement]

tellAbout1 kb as = map f2s (filterKB as kb)
```

Giving an explicit account of a class:

```
tellAbout :: KB -> Class -> [Statement]
tellAbout kb as =
  [All as (Class bs) |
      (Class bs) <- supersets as kb,
      as /= (Class bs) ]
  ++
  [No as (Class bs) |
      (OppClass bs) <- supersets as kb,
      as /= (OppClass bs) ]
```

A bit of pragmatics: do not tell ‘Some A are B’ if ‘All A are B’ also holds.

```
++
[Some as (Class bs) |
    (OppClass bs) <- nsupersets as kb,
    as /= (OppClass bs),
    notElem (as,Class bs) (subsetRel kb) ]
```

Do not tell ‘Some A are not B’ if ‘No A are B’ also holds.

```
++
[SomeNot as (Class bs) |
    (Class bs) <- nsupersets as kb,
    as /= (Class bs),
    notElem (as,OppClass bs) (subsetRel kb) ]
```

To build a knowledge base we need a function for updating an existing knowledge base with a statement.

If the update is successful, we want an updated knowledge base. If it is not, we want to get an indication of failure. The Haskell `Maybe` data type gives us just this.

```
data Maybe a = Nothing | Just a
```

The update function checks for possible inconsistencies. E.g., a request to add an $A \subseteq B$ fact to the knowledge base leads to an inconsistency if $A \not\subseteq B$ is already derivable.

```
update  :: Statement -> KB -> Maybe (KB,Bool)

update (All as bs) kb
  | elem bs (nsupersets as kb)  = Nothing
  | elem bs (supersets as kb)   = Just (kb,False)
  | otherwise                   =
                   Just (((as,bs,True): kb),True)
```

A request to add $A \subseteq \overline{B}$ leads to an inconsistency if $A \not\subseteq \overline{B}$ is already derivable.

```
update (No as bs) kb
  | elem bs' (nsupersets as kb) = Nothing
  | elem bs' (supersets as kb)  = Just (kb,False)
  | otherwise                   =
                  Just (((as,bs',True):kb),True)
 where bs' = opp bs
```

Similarly for the requests to update with $A \not\subseteq \overline{B}$ and with $A \not\subseteq B$:

```
update (Some as bs) kb
  | elem bs' (supersets as kb)  = Nothing
  | elem bs' (nsupersets as kb) = Just (kb,False)
  | otherwise                   =
              Just (((as,bs',False):kb),True)
 where bs' = opp bs

update (SomeNot as bs) kb
  | elem bs (supersets as kb)   = Nothing
  | elem bs (nsupersets as kb)  = Just (kb,False)
  | otherwise                   =
               Just (((as,bs,False):kb),True)
```

Use this to build a knowledge base from a list of statements. Again, this process can fail, so we use the `Maybe` datatype.

```
makeKB :: [Statement] -> Maybe KB
makeKB = makeKB' []
     where
         makeKB' kb [] = Just kb
         makeKB' kb (s:ss) =
             case update s kb of
               Just (kb',_) -> makeKB' kb' ss
               Nothing      -> Nothing
```

Preprocessing of strings, to prepare them for parsing:

```
preprocess :: String -> [String]
preprocess = words . (map toLower) .
      (takeWhile (\ x -> isAlpha x || isSpace x))
```

This will map a string to a list of words:

```
Main> preprocess "Are any women sailors?"
["are","any","women","sailors"]
```

A simple parser for statements:

```
parse :: String -> Maybe Statement
parse = parse' . preprocess
  where
    parse' ["all",as,"are",bs] =
      Just (All (Class as) (Class bs))
    parse' ["no",as,"are",bs] =
      Just (No (Class as) (Class bs))
    parse' ["some",as,"are",bs] =
      Just (Some (Class as) (Class bs))
    parse' ["some",as,"are","not",bs] =
      Just (SomeNot (Class as) (Class bs))
```

and for queries:

```
parse' ["are","all",as,bs] =
  Just (AreAll (Class as) (Class bs))
parse' ["are","no",as,bs] =
  Just (AreNo (Class as) (Class bs))
parse' ["are","any",as,bs] =
  Just (AreAny (Class as) (Class bs))
parse' ["are","any",as,"not",bs] =
  Just (AnyNot (Class as) (Class bs))
parse' ["what", "about", as] = Just (What (Class as))
parse' ["how", "about", as]  = Just (What (Class as))
parse' _ = Nothing
```

Parsing a text to construct a knowledge base:

```
process :: String -> KB
process txt = maybe [] id
                (mapM parse (lines txt) >>= makeKB)
```

This uses the `maybe` function, for getting out of the `Maybe` type. Instead of returning `Nothing`, this returns an empty knowledge base.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe _ f (Just x) = f x
maybe z _ Nothing  = z
```

```
mytxt = "all bears are mammals\n"
     ++ "no owls are mammals\n"
     ++ "some bears are stupids\n"
     ++ "all men are humans\n"
     ++ "no men are women\n"
     ++ "all women are humans\n"
     ++ "all humans are mammals\n"
     ++ "some men are stupids\n"
     ++ "some men are not stupids"
```

```
Main> process mytxt
[(men,stupids,False),(men,non-stupids,False),
 (humans,mammals,True),(women,humans,True),
 (men,non-women,True),(men,humans,True),
 (bears,non-stupids,False),(owls,non-mammals,True),
 (bears,mammals,True)]
```

Now suppose we have a text file of declarative natural language sentences about classes. Here is how to turn that into a knowledge base.

```
getKB :: FilePath -> IO KB
getKB p = do
    txt <- readFile p
    return (process txt)
```

And here is how to write a knowledge base to file:

```
writeKB :: FilePath -> KB -> IO ()
writeKB p kb = writeFile p
                (unlines (map (show.f2s) kb))
```

The inference engine in action:

```
chat :: IO ()
chat = do
 kb <- getKB "kb.txt"
 putStrLn "Update or query the KB:"
 str <- getLine
 if str == "" then return ()
  else do
   case parse str of
     Just (What as) -> let info = tellAbout kb as in
       if info == [] then putStrLn "No info.\n"
       else putStrLn (unlines (map show info))
     Just stmt      ->
      if isQuery stmt then
        if deriv kb stmt then putStrLn "Yes.\n"
```

```
          else if deriv kb (neg stmt)
                 then putStrLn "No.\n"
                 else putStrLn "I don't know.\n"
        else case update stmt kb of
          Just (kb',True) -> do
                      writeKB "kb.txt" kb'
                      putStrLn "OK.\n"
          Just (_,False)  -> putStrLn
                      "I knew that already.\n"
          Nothing         -> putStrLn
                         "Inconsistent with my info.\n"
    Nothing        -> putStrLn "Wrong input.\n"
  chat
```

```
main = do
        putStrLn "Welcome to the Knowledge Base."
        chat
```

Demo

...

Conclusions

- What is the use of this?
- Cognitive research focusses on this kind of quantifier reasoning . . .
- Can this be used to meet cognitive realities?
- Links with cognition by refinement of this calculus: the 'Battaglini' calculus*
- Towards Rational Reconstruction of Cognitive Processing

* Fabian Battaglini, Inference and Interpretation, PhD Thesis under construction.