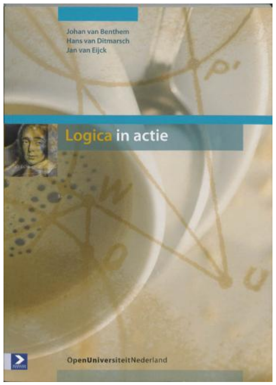
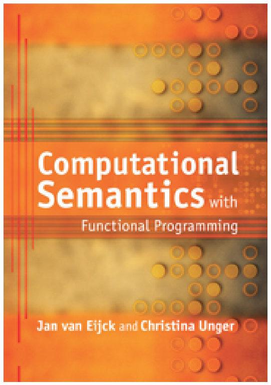
Logic in Action

Jan van Eijck

CKI, 8 November 2010




Computational Semantics with Functional Programming
Jan van Eijck and Christina Unger

Abstract

This talk will provide a tour of lightning visits of applications of logic:

- Logic Programming and Functional Programming
- Logic in Linguistics
- Knowledge Bases
- Propositional Logic Theorem Proving
- Formal Specification, Software Testing
- Puzzle Solving (logical puzzles, sudokus)

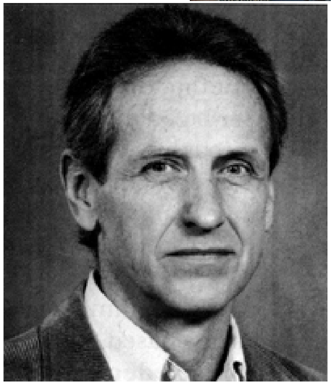
`http://www.logicinaction.org`

`http://www.cambridge.org/vaneijck-unger`

But First . . .

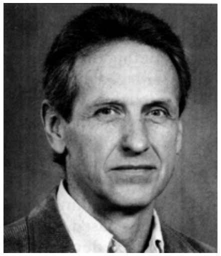
http://www.youtube.com/watch?v=OksHblHDij0

Heroes of Logic Programming



Heroes of Functional Programming

A Brief History of Logic Programming



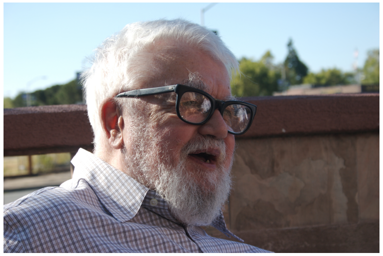
1930 Jacques Herbrand provides the proof-theoretical basis for automated theorem proving

1965 John Alan Robinson proposes resolution with unification [11]

± **1970** Robert Kowalski proposes a procedural interpretation of Horn clauses.

1972 Alain Colmerauer creates Prolog (with Philippe Roussel)

A Brief History of Functional Programming



1932 Alonzo Church presents the lambda calculus

1937 Alan Turing proves that lambda calculus and Turing machines have the same computational power.

1958 John McCarthy starts to implement LISP.

1978-9 Robin Milner cs develop ML.

1987 Agreement on a common standard for lazy purely functional programming: Haskell.

Haskell

http://www.haskell.org

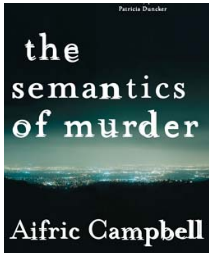
A Brief History of Formal Linguistics

1916 Ferdinand de Saussure, Cours de linguistique générale published posthumously. Natural language may be analyzed as a formal system.

1957 Noam Chomsky, Syntactic Structures, proposes to define natural languages as sets of grammatical sentences, and to study their structure with formal (mathematical) means. Presents a formal grammar for a fragment of English.

1970 Richard Montague, English as a Formal Language, proposes to extend the Chomskyan program to semantics and pragmatics. Presents a formal grammar for a fragment of English, including semantics (rules for computing meanings). Links the study of natural language to the study of formal languages (languages from logic and computer science).

Richard Montague (1930-1971)



Developed higher-order typed intensional logic with a possible-worlds semantics and a formal pragmatics incorporating indexical pronouns and tenses.

Program in semantics (around 1970): universal grammar.

Towards a philosophically satisfactory and logically precise account of syntax, semantics, and pragmatics, covering both formal and natural languages.

"The Proper Treatment of Quantification was as profound for semantics as Chomsky's Syntactic Structures was for syntax." (Barbara Partee on Montague, in the Encyclopedia of Language and Linguistics.)

- Chomsky: English can be described as a formal system.
- Montague: English can be described as a formal system with a formal semantics, and with a formal pragmatics.

Montague's program can be viewed as an extension of Chomsky's program.

The Program of Montague Grammar

- Montague's thesis: there is no essential difference between the semantics of natural languages and that of formal languages (such as that of predicate logic, or programming languages).
- The method of fragments: UG [9], EFL [8], PTQ [7]
- The misleading form thesis (Russell, Quine)
- Proposed solution to the misleading form thesis
- Key challenges: quantification, anaphoric linking, tense, intensionality.

Misleading Form

Aristotle's theory of quantification has two logical defects:

1. Quantifier combinations are not treated; only one quantifier per sentence is allowed.
2. 'Non-standard quantifiers' such as most, half of, at least five, . . . are not covered.

Frege's theory of quantification removed the first defect.

The Fregean view of quantifiers in natural language: quantified Noun Phrases are systematically misleading expressions.

Their natural language syntax does not correspond to their logic:

"Nobody is on the road" $\rightsquigarrow$ $\neg\exists x(\text{Person}(x) \wedge \text{OnTheRoad}(x))$

Solution to the Misleading Form Thesis

expression	translation	type
every	**every**	$(e \to t) \to ((e \to t) \to t)$
princess	P	$(e \to t)$
every princess	**every** P	$(e \to t) \to t$
laughed	S	$(e \to t)$
every princess laughed	(**every** P) S	t

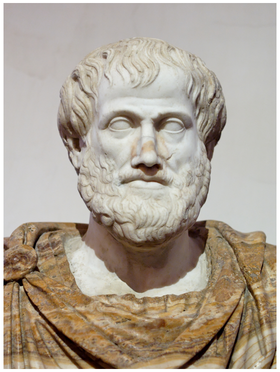
where **every** is a name for the constant $\lambda P \lambda Q. \forall x(Px \to Qx)$.

NL analysis, logic programming, functional programming

- Usefulness of typed lambda calculus for NL analysis.
- Linguist Barbara Partee: “Lambda’s have changed my life.”
- Computational linguistics: From Prolog to Haskell?
- Appeal of Prolog: Prolog-style unification [12], ‘Parsing as Deduction’ [10], useful didactic tool [1].
- But a new trend is emerging [3, 4]
- NLP Resources in Haskell: see

  `http://www.haskell.org/haskellwiki/Applications_and_libraries/Linguistics`

The Simplest Natural Language Engine You Can Get



All A are B No A are B

Some A are B Not all A are B

Aristotle interprets his quantifiers with existential import: All A are B and No A are B are taken to imply that there are A.

What can we ask or state with the Aristetelian quantifiers?

Questions and Statements (PN for plural nouns):

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

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

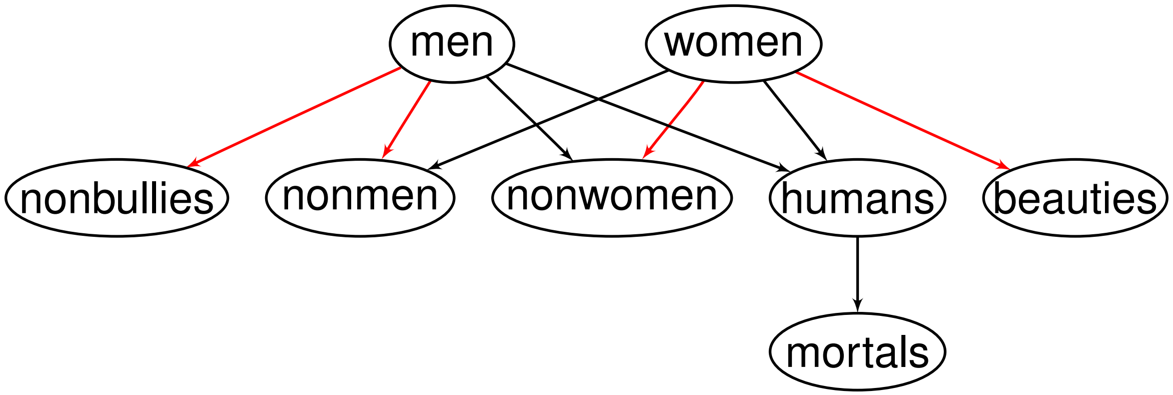
Example Interaction

```
jve@vuur:~/courses/lot2009$ ./Main
Welcome to the Knowledge Base.
Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All mammals are animals.
I knew that already.

Update or query the KB:
No mammals are birds.
```

```
OK.

Update or query the KB:
How about women?
All women are humans.
No women are men.

Update or query the KB:
All humans are mammals.
OK.

Update or query the KB:
How about women?
All women are animals.
All women are humans.
All women are mammals.
```

```
No women are birds.
No women are men.
No women are owls.

Update or query the KB:
```

Example Knowledge Base



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 $\widetilde{A} = \overline{A}$.

Computing the subset relation from the knowledge base:

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

Computing the non-subset relation from the knowledge base:

$$\frac{A \not\subseteq B}{\widetilde{B} \not\subseteq \widetilde{A}} \qquad \frac{A \supseteq B \quad B \not\subseteq C}{A \not\subseteq C} \qquad \frac{B \not\subseteq C \quad C \supseteq D}{B \not\subseteq D}$$

Reflexivity and existential import:

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

Consistency of a Knowledge Base

A Knowledge Base **K** is inconsistent if for some $A \subseteq B$:

$$\frac{\mathbf{K}}{A \subseteq B} \qquad \frac{\mathbf{K}}{A \not\subseteq B}$$

Otherwise **K** is consistent.

Soundness and Completeness of Inference System

Exercise 1 *An inference system is called sound if all conclusions that can be derived are valid, i.e. if all axioms are true and all inference rules preserve truth. Show that the inference system for Aristotelian syllogistics is sound.*

Exercise 2 *An inference system is called complete if it can derive all valid conclusions from a set of premisses. In other words: if* $A \subseteq B$ *does not follow from a knowledge base, then there is a class model for the knowledge base where* $A \not\subseteq B$*, and if* $A \not\subseteq B$ *does not follow from a knowledge base, then there is a class model for the knowledge base where* $A \subseteq B$*. Show that the inference system for Aristotelian syllogistics is complete.*

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`.

```haskell
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 transitive closure using a function for least fixpoint.
TC(R) = lfp $(\lambda S.S \cup S \cdot S)\ R$.

```
tc :: Ord a => Rel a -> Rel a
tc = lfp (\ s -> (sort.nub) (s ++ (s@@s)))

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 =
  tc ([(x,y)             | (x,y,True) <- kb ]
  ++  [(opp y,opp x) | (x,y,True) <- kb ]
  ++  [(x,x)             | (x,_,_)    <- kb ]
  ++  [(opp x,opp x) | (x,_,_)    <- kb ]
  ++  [(y,y)             | (_,y,_)    <- kb ]
  ++  [(opp y,opp y) | (_,y,_)    <- 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` holds. So the statement is derivable, hence true.
- `deriv kb (neg stmt)` holds. So the negation of `stmt` is derivable, hence true. So `stmt` is false.
- neither `deriv kb stmt` nor `deriv kb (neg stmt)` holds. So 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)
```

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 $A \nsubseteq B$ 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
```

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

Use of This

- Cognitive research focusses on this kind of quantifier reasoning. Links with cognition by refinement of this calculus . . . The “natural logic for natural language” enterprise: special workshop during Amsterdam Colloquium 2009 (see http://www.illc.uva.nl/AC2009/)

- “Our ultimate goal is to form an adequate model of parts of our language competence. Adequate means that the model has to be realistic in terms of complexity and learnability. We will not be so ambitious as to claim that our account mirrors real cognitive processes, but what we do claim is that our account imposes constraints on what the real cognitive processes can look like.” [2]

Alternative Set-up: Clausal Form for Propositional Logic

literals a literal is a proposition letter or its negation.

clause a clause is a set of literals.

clause sets a clause set is a set of clauses.

example The clause form of

$$(p \to q) \land (q \to r)$$

is

$$\{\{\neg p, q\}, \{\neg q, r\}\}.$$

Unit Propagation

If one member of a clause set is a singleton $\{l\}$ (a ‘unit’), then:

1. remove every other clause containing l from the clause set;
2. remove $\bar{l}$ from every clause in which it occurs.

The result of applying this rule is an equivalent clause set.

Example:

$$\{\{p\}, \{\neg p, q\}, \{\neg q, r\}, \{p, s\}\}.$$

yields:

$$\{\{p\}, \{q\}, \{\neg q, r\}\},$$

which in turn yields

$$\{\{p\}, \{q\}, \{r\}\}.$$

Unit Propagation is Complete for Horn Clauses

The Horn fragment of propositional logic consists of all clause sets where every clause has at most one positive literal.

HORNSAT is the problem of checking Horn clause sets for satisfiability. This check can be performed in polynomial time (linear in the size of the formula, in fact).

If unit propagation yields a clause set in which units $\{l\}, \{\bar{l}\}$ occur, the original clause set is unsatisfiable, otherwise the units in the result determine a satisfying valuation.

Recipe: if $\{l\}$ occurs is the final clause set, then map its proposition letter to the truth value that makes l true; map all other proposition letters to false.

Syllogistic Logic is in the Horn Fragment of Propositional Logic

Translation:

All A are B $\mapsto$ $\{\{\neg a, b\}\}$.

No A are B $\mapsto$ $\{\{\neg a, \neg b\}\}$.

Some A are B $\mapsto$ $\{\{a\}, \{b\}\}$.

Not all A are B $\mapsto$ $\{\{a\}, \{\neg b\}\}$.

Unit Propagation Implemented

```
data Lit = Pos String | Neg String deriving Eq

instance Show Lit where
  show (Pos x) = x
  show (Neg x) = '-': x

ng :: Lit -> Lit
ng (Pos x) = Neg x
ng (Neg x) = Pos x

type Clause = [Lit]
```

```
unitProp :: Lit -> [Clause] -> [Clause]
unitProp x cs = concat (map (unitP x) cs) where
  unitP :: Lit -> Clause -> [Clause]
  unitP x ys = if elem x ys
               then []
               else
                if elem (ng x) ys
                  then [delete (ng x) ys]
                  else [ys]

unit :: Clause -> Bool
unit [x] = True
unit  _  = False
```

```
propagate :: [Clause] -> Maybe ([Lit],[Clause])
propagate clauses =
  prop [] (concat $ filter unit clauses)
                   (filter (not.unit) clauses)
  where
    prop :: [Lit] -> [Lit] -> [Clause]
                 -> Maybe ([Lit],[Clause])
    prop xs [] clauses = Just (xs,clauses)
    prop xs (y:ys) clauses =
      if elem (ng y) xs
       then Nothing
       else prop (y:xs)(ys++newlits) clauses' where
        newclauses = unitProp y clauses
        zs         = filter unit newclauses
        clauses'   = newclauses \\ zs
        newlits    = concat zs
```

Example

```
a,b,c,d :: Lit
a = Pos "a"; b = Pos "b" ; c = Pos "c"; d = Pos "d"

example :: [Clause]
example = [[ng a,b],[ng b,c],[ng c,d],[a],[ng d]]

example1 :: [Clause]
example1 = [[ng a,b],[ng b,c],[ng c,d],[a],[d]]
```

Exercise 3 *Give an implementation of a syllogistic knowledge base using unit propagation for propositional logic as your inference engine.*

Automating First Order Logic: Alloy

Alloy (`http://alloy.mit.edu`) is a software specification tool based on first order logic plus some relational operators. Alloy automates predicate logic by using bounded exhaustive search for counterexamples in small domains [5].

Alloy does allow for automated checking of specifications, but only for small domains. The assumption that most software design errors show up in small domains is known as the small domain hypothesis [6].

Example Question

Here is a question about operations on relations.

Given a relation R, do the following two procedures boil down to the same thing?

First take the symmetric closure, next the transitive closure

First take the transitive closure, next the symmetric closure

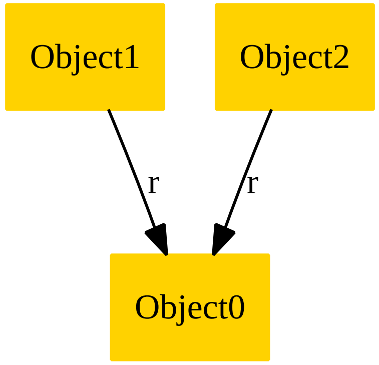
If we use R^+ for the transitive closure of R and $R \cup R^{\smallsmile}$ for the symmetric closure, then the question becomes:

$$(R \cup R^{\smallsmile})^+ \quad \overset{?}{=} \quad R^+ \cup R^{+\smallsmile}$$

Here is an Alloy version of this question:

```
sig Object { r : set Object }
assert claim { *(r + ~r) = *r + ~*r }
check claim
```

If you run this in Alloy, the system will try to find counterexamples. Here is a counterexample that it finds:



How to draw logical conclusions from a list of givens

Here is a story. Someone invites six people A, B, C, D, E, F to attend a conference. The email exchanges that follow yield the following information:

1. At least one of A, B will attend.
2. From the set $\{A, E, F\}$ exactly two will attend.
3. Either both B and C will attend or neither of them will.
4. One of A and D will attend, the other will not.
5. Same for C and D.
6. If D does not attend, then neither will E.

Use an Alloy specification to figure out who will attend the conference.

Solution

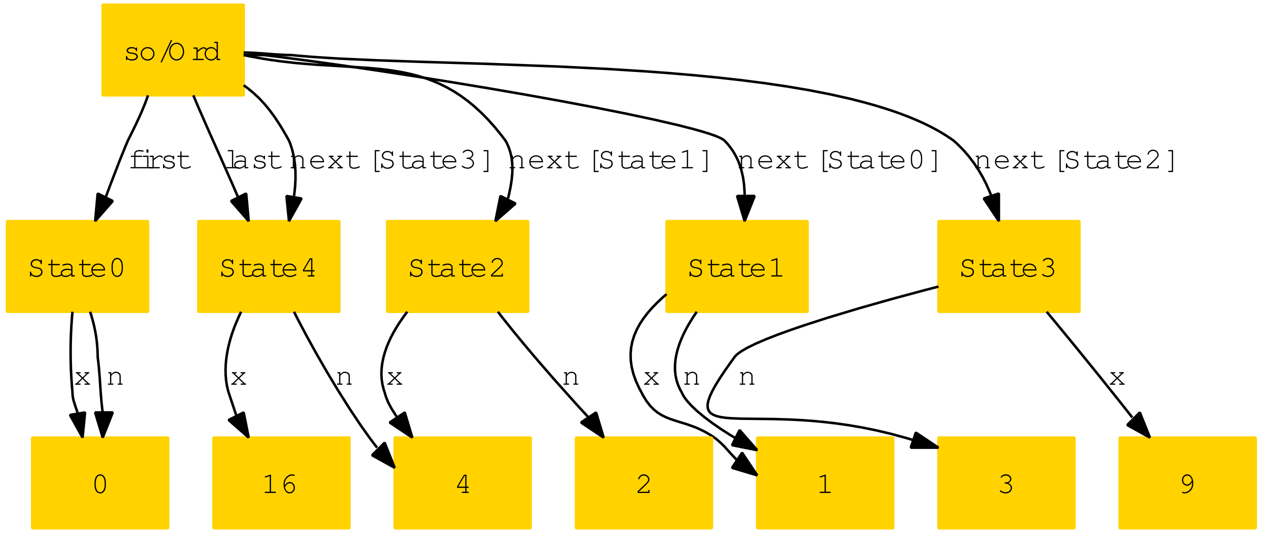
```
abstract sig Person {}
one sig A,B,C,D,E,F extends Person {}
sig Congress in Person {}

fact{
  some (A + B) & Congress
  #((A+E+F) & Congress) = 2
  B in Congress iff C in Congress
  A in Congress iff not D in Congress
  C in Congress iff not D in Congress
  not D in Congress => not E in Congress
}

run {}
```

Result

A (Congress)	B (Congress)	C (Congress)	D	E	F (Congress)

Software Specification With Alloy

```
module myexamples/invar
open util/ordering[State] as so
open util/integer as integer

sig State {
  x: Int,
  n: Int
}

fun inc [n : Int]: Int { add [n,Int[1]] }

pred init {
  let fs = so/first |
   { fs.x = Int[0] and fs.n = Int[0] }
}

pred extend [pre, post: State] {
  some X,N: Int | pre.x = X
                  and pre.n = N
                  and post.x = inc[add[X,add[N,N]]]
                  and post.n = inc[N]
}

fact createStates {
   init
   all s: State - so/last |
          let s' = so/next[s] | extend[s,s']
}

run {} for exactly 5 State, 6 int
```


so/Ord
first
last
next [State3]
next [State1]
next [State0]
next [State2]
State0
State4
State2
State1
State3
x
n
x
n
x
n
x
n
n
x
0
16
4
2
1
3
9

If there is time . . .

Sudoku solving with Alloy and Haskell . . .

Prolog

No time left. See the Computation Chapter of the ‘Logic in Action’ textbook.

Any Questions?

References

[1] P. Blackburn and J. Bos. Representation and Inference for Natural Language; A First Course in Computational Semantics. CSLI Lecture Notes, 2005.

[2] Jan van Eijck and Christina Unger. Computational Semantics with Functional Programming. Cambridge University Press, 2010.

[3] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. The Computer Journal, 32(2):108–121, 1989.

[4] Richard A. Frost. Realization of natural language interfaces using lazy functional programming. ACM Comput. Surv., 38(4), 2006.

[5] Daniel Jackson. Automating first-order relational logic. ACM SIGSOFT Software Engineering Notes, 25(6):130–139, 2000.

[6] Daniel Jackson. Software Abstractions; Logic, Language and Analysis. MIT Press, 2006.

[7] R. Montague. The proper treatment of quantification in ordinary English. In J. Hintikka, editor, Approaches to Natural Language, pages 221–242. Reidel, 1973.

[8] R. Montague. English as a formal language. In R.H. Thomason, editor, Formal Philosophy; Selected Papers of Richard Montague, pages 188–221. Yale University Press, New Haven and London, 1974.

[9] R. Montague. Universal grammar. In R.H. Thomason, editor, Formal Philosophy; Selected Papers of Richard Montague,

pages 222–246. Yale University Press, New Haven and London, 1974.

[10] F.C.N. Pereira and H.D. Warren. Parsing as deduction. In Proceedings of the 21st Annual Meeting of the ACL, pages 137–111. MIT, Cambridge, Mass., 1983.

[11] J.A. Robinson. A machine-oriented logic based on the resolution principle. Journal of the ACM, 12(1):23–41, 1965.

[12] S.M. Shieber. An Introduction to Unification Based Approaches to Grammar, volume 4 of CSLI Lecture Notes. CSLI, Stanford, 1986. Distributed by University of Chicago Press.