Genifer in Haskell

YKY (general.intelligence@Gmail.com)

Contents

1 Preface **2**

1.1 Literate programming 2

2 Message passing **3**

2.1 Processing goals 3

2.2 Processing answers 3

2.3 CloudHaskell 3

3 TV evaluation **4**

3.1 Combining multiple rules 4

3.2 Evaluating a single rule 4

4 First-order logic **5**

4.1 Unification 5

4.2 Substitution management 5

1 Preface

1.1 Literate programming

This file is Genifer.pdf, generated from Genifer.lhs by:

```
lhs2tex -o Genifer.tex Genifer.lhs
pdflatex Genifer.tex
```

You can run the source code contained in Genifer.lhs by:

```
ghci Genifer.lhs
```

2 Message passing

The agent waits for messages. An incoming message can be:

1. a goal
2. an answer

Perhaps the format can be:

1. goal: "G formula"
2. answer: "A goal-ID value"

2.1 Processing goals

Try to match the goal against rules in the (local) KB.

If there is a match, apply the rule, ie, instantiate it with appropriate substitutions, get the subgoals.

Check if the subgoals can be answered by the local KB. If yes, recurse.
If no, send the subgoal to the **Priority Queue**, ie, certain agent(s) in the network responsible for routing new goals.

2.2 Processing answers

Locate the instantiated rule in the "Instance Store".

Attach the answer to the instantiated rule.

If there are answers for all the subgoals in that rule, evaluate it (see §3 TV evaluation).

2.3 CloudHaskell

```haskell
import Remote

import Data.Typeable (Typeable)
import Data.Data (Data)
import Data.Binary (Binary, get, put)
import System.Random (randomR, getStdRandom)
import Control.Concurrent (threadDelay)
import Control.Monad (when)
import Control.Monad.Trans (liftIO)

main = putStrLn "Hello, World!"
```

3 TV evaluation

3.1 Combining multiple rules

If all rules are aligned in the "forward" direction, as in:

$$\begin{array}{rcl} H_1 & \leftarrow & A_1, A_2, A_3, ... \\ H_2 & \leftarrow & B_1, B_2, B_3, ... \\ & ... & \end{array}$$

apply Abram's technique:

$$\begin{aligned} P(H|A_i, B_i, ...) &= \frac{P(A_i, B_i, ...|X)P(X)}{P(A_i, B_i, ...)} && \text{– Bayes law} \\ &= \frac{P(A_i|X)P(B_i|X)...P(X)}{P(A_i, B_i, ...)} && \text{– Naive Bayes assumption} \\ &= \frac{P(H_1|A_i)P(A_i)P(H_2|B_i)P(B_i)...}{P(A_i, B_i, ...)P(X)^{n-1}} && \text{– Bayes law again} \end{aligned}$$

If any of the rules is malaligned, we need to reverse that rule first, by Bayes law:

$$P(B^*|H, B_{i\neq *}) = ?$$

3.2 Evaluating a single rule

FUNCTION: Given a rule, its arguments and parameters, and messages attached to each argument, calculate the resulting message.

INPUT:

op	=	operator of rule
head	=	list of head literals
body	=	list of body literals
subgoal	=	which argument is the subgoal
params	=	list of parameters, one element for each body literal
msgsH	=	list of messages for head literals
msgsB	=	list of messages for body literals

At this stage we assume the operator is always (probabilistic) "AND".

Algorithm:
Case 1: If subgoal = one of the heads, apply rule directly.
Case 2: If subgoal = one of the body literals, apply rule in reverse (via Bayes law).

```
evalRule op head body subgoal params msgsH msgsB
    | subgoal ≥ 0 = sum (zipWith (*) params msgsB)
    | otherwise = 0   -- TODO
```

4 First-order logic

4.1 Unification

The following code is borrowed from Takashi Yamamiya's web site:
http://propella.blogspot.com/2009/04/prolog-in-haskell.html

The type:

```
type Substitution = [(Term, Term)]
true = []
```

This is the code for applying a substitution:

```
  -- apply [(w"X", w"Y"), (w"Y", w"Z")] [(w"X"), (w"Y")] == [(w"Z"), (w"Z")]
apply :: Substitution → [Term] → [Term]
apply s ts = [applyTerm s t | t ← ts]
applyTerm [] (Var y n)     = Var y n
applyTerm ((Var x i, t) : s) (Var y j) | x ≡ y ∧ i ≡ j = applyTerm s t
   | otherwise               = applyTerm s (Var y j)
applyTerm s (Struct n ts) = Struct n (apply s ts)
```

And here is unify:

```
  -- unify (w"X") (w"apple") == Just [(w"X", w"apple")]
unify :: Term → Term → Maybe Substitution
unify (Var x n) (Var y m) = Just [(Var x n, Var y m)]
unify (Var x n) y = Just [(Var x n,  y)]
unify x (Var y m) = Just [(Var y m, x)]
unify (Struct a xs) (Struct b ys)
   | a ≡ b = unifyList xs ys
   | otherwise = Nothing
unifyList :: [Term] → [Term] → Maybe Substitution
unifyList [] [] = Just true
unifyList [] _ = Nothing
unifyList _ [] = Nothing
unifyList (x : xs) (y : ys) = do s ← unify x y
  s′ ← unifyList (apply s xs) (apply s ys)
  return (s ++ s′)
```

4.2 Substitution management