Homework Assignment 5 – Solutions

October 16, 2013

1 Implementing the syntax of L_0

This is the definition of L_0 syntax we need to implement:

(1) L_0 syntax:
   a. Basic expressions:
      i. Names: *Dick*, *Noam*, *John*, *Muhammad*
      ii. One-place predicates: *HasMustache*, *IsBald*
      iii. Two-place predicates: *Knows*, *Loves*
   b. Formulas:
      i. If δ is a one-place predicate and α is a name, then $\delta\ \alpha$ is a formula. Make sure that you display such a formula as $\delta(\alpha)$ by appropriately defining *show*.
      ii. If γ is a two-place predicate and α and β are names, then $\gamma\ \beta\ \alpha$ is a formula. Make sure that you display such a formula as $\gamma(\alpha, \beta)$ by appropriately defining *show*.
      iii. If φ is a formula, then $\sim\varphi$ (the negation of φ) is a formula.
      iv. If φ and ψ are formulas, then $\varphi \wedge \psi$ (the conjunction of φ and ψ) is a formula.
      v. If φ and ψ are formulas, then $\varphi \vee \psi$ (the disjunction of φ and ψ) is a formula.

We define the syntax of L_0 in a separate module:

```
ghci 1> :l L0syn
```

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

(2) **import** *Data.List*

The *L0syn* module defines the recursive type of sentential formulas *Form*. The main difference from the syntax of propositional logic introduced in the lecture notes is the definition of atomic formulas, which are now defined in terms of names, one-place predicates and two-place predicates:

(3) Implementing the syntax of L_0:

```
data Name = Dick | Noam | John | Muhammad deriving (Eq, Show)
data Pred1 = HasMustache | IsBald deriving (Eq, Show)
data Pred2 = Knows | Loves deriving (Eq, Show)
data Form = P1 Pred1 Name | P2 Pred2 Name Name |
            Ng Form | Cnj [Form] | Dsj [Form] deriving Eq
```

These are the types of the five formula constructors:

```
ghci 2> :t P1
P1 :: Pred1 → Name → Form
```

```
ghci 3> :t P2
P2 :: Pred2 → Name → Name → Form
```

```
ghci 4> :t Ng
Ng :: Form → Form
```

```
ghci 5> :t Cnj
Cnj :: [Form] → Form
```

```
ghci 6> :t Dsj
Dsj :: [Form] → Form
```

The definition of the *show* function is modified as the hw assignment requires:

(4)
```
instance Show Form where
  show (P1 pred1 name) = show pred1 ++ "(" ++ show name ++ ")"
  show (P2 pred2 name1 name2) = show pred2 ++ "(" ++ show name2 ++ "," ++ show name1 ++ ")"
  show (Ng f) = "~ " ++ show f
  show (Cnj fs) = "(" ++ intercalate " /\\ " (map show fs) ++ ")"
  show (Dsj fs) = "(" ++ intercalate " \\/ " (map show fs) ++ ")"
```

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

```
ghci 7> let form1 = P1 HasMustache Noam
```

```
ghci 8> form1
HasMustache (Noam)
```

```
ghci 9> let form2 = P2 Loves John Dick
```

```
ghci 10> form2
Loves (Dick, John)
```

ghci 11> **let** *form3* = *Ng form1*

ghci 12> *form3*
$\sim$ *HasMustache* (*Noam*)

ghci 13> **let** *form4* = *Ng* \$ *Dsj* [*form1*, *form2*]

ghci 14> *form4*
$\sim$ (*HasMustache* (*Noam*) $\vee$ *Loves* (*Dick*, *John*))

ghci 15> **let** *form5* = *Ng* \$ *Cnj* [*form3*, *form4*]

ghci 16> *form5*
$\sim$ ($\sim$ *HasMustache* (*Noam*) $\wedge$ $\sim$ (*HasMustache* (*Noam*) $\vee$ *Loves* (*Dick*, *John*)))

2 The semantics of L_0

We will implement a more general version of the semantics of L_0 as follows:

- assume that all models have 4 entities *Nixon*, *Chomsky*, *Mitchell* and *Ali*, i.e., the domain of entities is [*Nixon*, *Chomsky*, *Mitchell*, *Ali*]
- assume that the semantic values of the four names *Dick*, *Noam*, *John*, *Muhammad* are fixed in the obvious way: *Dick* denotes (Richard) *Nixon*, *Noam* denotes *Chomsky*, *John* denotes *Mitchell* and *Muhammad* denotes *Ali*
- generate all possible models that satisfy the above two constraints, i.e., generate all possible (combinations of) appropriate denotations for the one-place and two-place predicates listed above;
- in particular, one-place predicates should denote subsets of the domain of entities [*Nixon*, *Chomsky*, *Mitchell*, *Ali*] and two-place predicates should denote sets of pairs of entities, i.e., subsets of [(x, y) | $x \leftarrow$ [*Nixon*, *Chomsky*, *Mitchell*, *Ali*], $y \leftarrow$ [*Nixon*, *Chomsky*, *Mitchell*, *Ali*]]
- given all these models, define tautologies, satisfiability, contradictions, entailment and Context Set updates just as we did in the lecture notes on propositional logic

We load the module that defines the semantics for L_0:

ghci 17> :*l L0sem*

Note that this module imports the syntax module *L0syn* right at the very beginning (in addition to the *Data.List* module):

(5) **import** *Data.List*
import *L0syn*

We begin by defining the domain of entities underlying all our models:

(6) **data** $Entity = Nixon \mid Chomsky \mid Mitchell \mid Ali$ **deriving** $(Eq, Show, Enum, Bounded)$

This is type e (for entity) in Montagovian semantics.

We derive *Eq* b/c we want to be able to check whether two entities are identical or not. We derive *Show* so that we can display this type of semantic value in `ghci`. We derive *Enum* because we can (since all constructors have no fields / are of 0-arity).

Finally, we derive *Bounded* so that we can retrieve the full domain of entities in a convenient manner. In particular, we define the list of all entities, and also the list of all entity pairs (needed for two-place predicates) as follows:

(7) $entities :: [Entity]$
$entities = [minBound \,..\, maxBound]$
$entityPairs :: [(Entity, Entity)]$
$entityPairs = [(e1, e2) \mid e1 \leftarrow entities, e2 \leftarrow entities]$

Here's the result of these two definitions:

ghci 18> *entities*
$[Nixon, Chomsky, Mitchell, Ali]$

ghci 19> *entityPairs*
$[(Nixon, Nixon), (Nixon, Chomsky), (Nixon, Mitchell), (Nixon, Ali), (Chomsky, Nixon), (Chomsky, Chomsky), (Chomsky, Mitchell), (Chomsky, Ali), (Mitchell, Nixon), (Mitchell, Chomsky), (Mitchell, Mitchell), (Mitchell, Ali), (Ali, Nixon), (Ali, Chomsky), (Ali, Mitchell), (Ali, Ali)]$

Now let's think about the type of the interpretation function *eval* for L_0. The *eval* function should take a basic interpretation function as its first argument (i.e., the model), a formula as its second argument, and return a truth value, namely the semantic value of the formula under consideration relative to the model under consideration.

Since a basic interpretation function is just a function from basic expressions *BasicExp* (names, one-place predicates and two-plae predicates) to corresponding semantic values – let's call them associated basic interpretations *BasicInt* – the type of the interpretation function *eval* is:

(8) $eval :: (BasicExp \rightarrow BasicInt) \rightarrow Form \rightarrow Bool$

The type *Form* of formulas is already defined in the *L0syn* module, and the type *Bool* is made available by default in Haskell. So we only need to define the type of basic expressions *BasicExp* and the type of associated basic interpretations *BasicInt*.

Both these types are sum types:

- for the *BasicExp* type, we need to sum together names, one-place pred.s and two-place pred.s
- for the *BasicInt* type, we need to sum together *Entity* values (semantic values for names, i.e., type e), $Entity \rightarrow Bool$ functions (semantic values for one-place pred.s, i.e., type $\langle e, t \rangle$) and $Entity \rightarrow Entity \rightarrow Bool$ functions (semantic values for two-place pred.s, i.e., type $\langle e, \langle e, t \rangle \rangle$)

We therefore define the *BasicExp* and *BasicInt* types as shown below. We use record syntax for both of them so that we can easily extract the individual expressions and the individual denotations wrapped together in these sum types:

(9) a. $\textbf{data}\ BasicExp = NameExp\ \{nameExp :: Name\}\ |$
$\quad Pred1Exp\ \{pred1Exp :: Pred1\}\ |$
$\quad Pred2Exp\ \{pred2Exp :: Pred2\}$

b. $\textbf{data}\ BasicInt = NameInt\ \{nameInt :: Entity\}\ |$
$\quad Pred1Int\ \{pred1Int :: (Entity \rightarrow Bool)\}\ |$
$\quad Pred2Int\ \{pred2Int :: (Entity \rightarrow Entity \rightarrow Bool)\}$

Record syntax automatically makes available functions that extract particular types of basic expressions or interpretations for basic expressions:

```
ghci 20> :t nameExp
nameExp :: BasicExp → Name
```

```
ghci 21> :t pred1Exp
pred1Exp :: BasicExp → Pred1
```

```
ghci 22> :t pred2Exp
pred2Exp :: BasicExp → Pred2
```

```
ghci 23> :t nameInt
nameInt :: BasicInt → Entity
```

```
ghci 24> :t pred1Int
pred1Int :: BasicInt → Entity → Bool
```

```
ghci 25> :t pred2Int
pred2Int :: BasicInt → Entity → Entity → Bool
```

Our goal now is to define models / valuations for L_0, which are functions from basic expressions to their corresponding interpretations, i.e., functions of type $BasicExp \rightarrow BasicInt$.

To this end, we first define three helper functions. The first two helper functions take lists of entities and lists of entity pairs and convert them to appropriate semantic values for one-place pred.s and two-place pred.s, respectively. We define them as shown below:

(10) a. $list2Pred1Value :: [Entity] \rightarrow (Entity \rightarrow Bool)$
$list2Pred1Value\ xs = \lambda x \rightarrow elem\ x\ xs$

b. $list2Pred2Value :: [(Entity, Entity)] \rightarrow (Entity \rightarrow Entity \rightarrow Bool)$
$list2Pred2Value\ xs = \lambda y\ x \rightarrow elem\ (x, y)\ xs$

The third helper function, named *basicIntFun*, takes two lists of entities and two lists of pairs of entities as arguments – these are the denotations for our one-place pred.s and our two-place pred.s,

respectively – and returns a model, i.e., a function of type $BasicExp \to BasicInt$, constructed based on those 4 lists. As required in the hw assignment, we always assign the same entities to the four names. The definition of this function is provided below:

(11)
$$\begin{aligned}
&basicIntFun :: [Entity] \to [Entity] \to [(Entity, Entity)] \to [(Entity, Entity)] \to BasicExp \to BasicInt\\
&basicIntFun _\ _\ _\ _\ (NameExp\ Dick) = NameInt\ Nixon\\
&basicIntFun _\ _\ _\ _\ (NameExp\ Noam) = NameInt\ Chomsky\\
&basicIntFun _\ _\ _\ _\ (NameExp\ John) = NameInt\ Mitchell\\
&basicIntFun _\ _\ _\ _\ (NameExp\ Muhammad) = NameInt\ Ali\\
&basicIntFun\ xs\ _\ _\ _\ (Pred1Exp\ HasMustache) = Pred1Int\ (list2Pred1Value\ xs)\\
&basicIntFun\ _\ xs\ _\ _\ (Pred1Exp\ IsBald) = Pred1Int\ (list2Pred1Value\ xs)\\
&basicIntFun\ _\ _\ xs\ _\ (Pred2Exp\ Knows) = Pred2Int\ (list2Pred2Value\ xs)\\
&basicIntFun\ _\ _\ _\ xs\ (Pred2Exp\ Loves) = Pred2Int\ (list2Pred2Value\ xs)
\end{aligned}$$

We are now almost ready to generate all the models for L_0. We only need to define a *powerset* function that will generate all possible lists of entities and all possible lists of pairs of entities that we will assign as denotations for our one-place pred.s and two-place pred.s, respectively. This function is defined below:

(12)
$$\begin{aligned}
&powerset :: [a] \to [[a]]\\
&powerset\ [\,] = [[\,]]\\
&powerset\ (x : xs) = map\ (x:)\ p +\!\!+ p\ \textbf{where}\ p = powerset\ xs
\end{aligned}$$

Make sure you understand this recursive definition. For example, make sure you understand why $powerset\ [1\,..\,3]$ is evaluated as shown below. In particular: why are the elements of the powerset listed in that particular order?

```
ghci 26> powerset [1..3]
    [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

We are finally ready to provide the recursive definition of the *eval* function, which is actually the definition of the semantics of L_0. As we already indicated, the *eval* function is of type $(BasicExp \to BasicInt) \to Form \to Bool$:

- it takes a model / valuation as its first argument, i.e., an assignment of semantic values to basic expressions, which is just a function of type $BasicExp \to BasicInt$;
- it takes a formula of type *Form* as its second argument;
- it returns a truth value of type *Bool* as its value; this truth value is the semantic value of the formula under consideration relative to the model under considertation.

(13) The semantics of L_0:

$$\begin{aligned}
&eval :: (BasicExp \to BasicInt) \to Form \to Bool\\
&eval\ m\ (P1\ pred1\ name) = (pred1Int\ \$\ m\ (Pred1Exp\ pred1))\ (nameInt\ \$\ m\ (NameExp\ name))\\
&eval\ m\ (P2\ pred2\ name1\ name2) =\\
&\quad (pred2Int\ \$\ m\ (Pred2Exp\ pred2))\ (nameInt\ \$\ m\ (NameExp\ name1))\\
&\qquad\qquad\qquad\qquad (nameInt\ \$\ m\ (NameExp\ name2))\\
&eval\ m\ (Ng\ f) = \neg\ (eval\ m\ f)\\
&eval\ m\ (Cnj\ fs) = all\ (eval\ m)\ fs\\
&eval\ m\ (Dsj\ fs) = any\ (eval\ m)\ fs
\end{aligned}$$

The final three clauses (for *Ng*, *Cnj* and *Dsj*) are the same as for the propositional logic semantics we discussed in the lecture notes. Only the first two clauses, which deal with the interpretation of atomic formulas, are different. In both cases, we make extensive use of the record-syntax functions automatically made available by our definition of the *BasicInt* type.

An example will make these first two clauses much clearer. Let's take the first model in our *allVals* list and name it *exampleInt1* for convenience.

```
ghci 27> let exampleInt1 = allVals !! 0
```

The semantic values we assign to names, one-place pred.s and two-place pred.s have the same sum type *BasicInt*:

```
ghci 28> :t exampleInt1 (NameExp Dick)
exampleInt1 (NameExp Dick) :: BasicInt
```

```
ghci 29> :t exampleInt1 (Pred1Exp HasMustache)
exampleInt1 (Pred1Exp HasMustache) :: BasicInt
```

```
ghci 30> :t exampleInt1 (Pred2Exp Knows)
exampleInt1 (Pred2Exp Knows) :: BasicInt
```

But the functions *nameInt*, *pred1Int* and *pred2Int* we automatically made available when we used record syntax to define the sum type *BasicInt* assign names, one-place pred.s and two-place pred.s denotations of the appropriate types:

```
ghci 31> :t nameInt (exampleInt1 (NameExp Dick))
nameInt (exampleInt1 (NameExp Dick)) :: Entity
```

```
ghci 32> :t pred1Int (exampleInt1 (Pred1Exp HasMustache))
pred1Int (exampleInt1 (Pred1Exp HasMustache)) :: Entity → Bool
```

```
ghci 33> :t pred2Int (exampleInt1 (Pred2Exp Knows))
pred2Int (exampleInt1 (Pred2Exp Knows)) :: Entity → Entity → Bool
```

It is precisely these denotations that we use when we want to determine whether an atomic formula is true or not:

- when we want to evaluate an atomic formula of the form *P1 pred1 name*, i.e., in which a one-place predicate is predicated of a name, we need to extract the function of type *Entity* → *Bool* assigned to *pred1* by model *m*, which we do by means of *pred1Int* $ *m* (*Pred1Exp pred1*), and we need to apply this function to the entity denoted by *name* in model *m*, which we extract by means of *nameInt* $ *m* (*NameExp name*);

- correspondingly, when we want to evaluate an atomic formula of the form *P2 pred2 name1 name2*, i.e., in which a two-place predicate relates two names, we need to extract the function of type $Entity \to Entity \to Bool$ assigned to *pred2* by model *m*, which we do by means of $pred2Int\ \$\ m\ (Pred2Exp\ pred2)$, and we need to apply this functions to the entities denoted by *name1* and *name2*, which we extract by means of $nameInt\ \$\ m\ (NameExp\ name1)$ and $nameInt\ \$\ m\ (NameExp\ name1)$.

Now that we understand the general structure of all the clauses of the *eval* function, we can check whether our models satisfy the requirements specified in the hw assignment. In particular, we want to check that all the names receive their appropriate denotations – and they do:

ghci 34> $nameInt\ (exampleInt1\ (NameExp\ Dick))$
$Nixon$

ghci 35> $nameInt\ (exampleInt1\ (NameExp\ Noam))$
$Chomsky$

ghci 36> $nameInt\ (exampleInt1\ (NameExp\ John))$
$Mitchell$

ghci 37> $nameInt\ (exampleInt1\ (NameExp\ Muhammad))$
Ali

To finish examining the model *exampleInt1*, we take a look at the denotations of one-place and two-place pred.s:

ghci 38> $[x \mid x \leftarrow entities, pred1Int\ (exampleInt1\ (Pred1Exp\ HasMustache))\ x]$
$[Nixon, Chomsky, Mitchell, Ali]$

ghci 39> $[x \mid x \leftarrow entities, pred1Int\ (exampleInt1\ (Pred1Exp\ IsBald))\ x]$
$[Nixon, Chomsky, Mitchell, Ali]$

ghci 40> $[(x,y) \mid x \leftarrow entities, y \leftarrow entities, pred2Int\ (exampleInt1\ (Pred2Exp\ Knows))\ y\ x]$
$[(Nixon,Nixon),(Nixon,Chomsky),(Nixon,Mitchell),(Nixon,Ali),(Chomsky,Nixon),(Chomsky,Chomsky),(Chomsky,Mitchell),(Chomsky,Ali),(Mitchell,Nixon),(Mitchell,Chomsky),(Mitchell,Mitchell),(Mitchell,Ali),(Ali,Nixon),(Ali,Chomsky),(Ali,Mitchell),(Ali,Ali)]$

ghci 41> $[(x,y) \mid x \leftarrow entities, y \leftarrow entities, pred2Int\ (exampleInt1\ (Pred2Exp\ Loves))\ y\ x]$
$[(Nixon,Nixon),(Nixon,Chomsky),(Nixon,Mitchell),(Nixon,Ali),(Chomsky,Nixon),(Chomsky,Chomsky),(Chomsky,Mitchell),(Chomsky,Ali),(Mitchell,Nixon),(Mitchell,Chomsky),(Mitchell,Mitchell),(Mitchell,Ali),(Ali,Nixon),(Ali,Chomsky),(Ali,Mitchell),(Ali,Ali)]$

It is precisely the denotations of these predicates that vary from model to model. For example, if we look at the 2000010th model, some of the denotations of these predicates are different:

```
ghci 42> let exampleInt2 = allVals !! 2000009
```

```
ghci 43> [x | x ← entities, pred1Int (exampleInt2 (Pred1Exp HasMustache)) x]
    [Nixon, Chomsky, Mitchell, Ali]
```

```
ghci 44> [x | x ← entities, pred1Int (exampleInt2 (Pred1Exp IsBald)) x]
    [Nixon, Chomsky, Mitchell, Ali]
```

```
ghci 45> [(x, y) | x ← entities, y ← entities, pred2Int (exampleInt2 (Pred2Exp Knows)) y x]
    [(Nixon, Nixon),(Nixon, Chomsky),(Nixon, Mitchell),(Nixon, Ali),(Chomsky, Nixon),(Chomsky,
    Chomsky),(Chomsky, Mitchell),(Chomsky, Ali),(Mitchell, Nixon),(Mitchell, Chomsky),(Mitchell,
    Mitchell),(Ali, Ali)]
```

```
ghci 46> [(x, y) | x ← entities, y ← entities, pred2Int (exampleInt2 (Pred2Exp Loves)) y x]
    [(Nixon, Chomsky),(Nixon, Mitchell),(Nixon, Ali),(Chomsky, Nixon),(Chomsky, Mitchell),
    (Chomsky, Ali),(Mitchell, Chomsky),(Mitchell, Mitchell),(Mitchell, Ali),(Ali, Chomsky),(Ali,
    Mitchell)]
```

We evaluate formulas relative to these models in the expected way:

```
ghci 47> eval exampleInt1 (P2 Loves Noam Noam)
    True
```

```
ghci 48> eval exampleInt2 (P2 Loves Noam Noam)
    False
```

```
ghci 49> eval exampleInt1 $ Ng (P2 Loves Noam Noam)
    False
```

```
ghci 50> eval exampleInt2 $ Ng (P2 Loves Noam Noam)
    True
```

3 Tautologies, satisfiability and contradictions

Because we can generate all models for a formula, we can in principle 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), just like we did for propositional logic.

The definition of the relevant predicates are provided below:

(14) $tautology :: Form \to Bool$
$tautology\ f = all\ (\lambda m \to eval\ m\ f)\ allVals$

(15) $satisfiable :: Form \to Bool$
$satisfiable\ f = any\ (\lambda m \to eval\ m\ f)\ allVals$

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

But actually evaluating whether these predicates are true or false when applied to particular formulas is not really feasible.

This is because the number of models in our *allVals* list is extremely big:

```
ghci 51> 2↑4∗2↑4∗2↑16∗2↑16
    1099511627776
```

While Haskell's laziness enables us to work with a very large list like *allVals* as long as we don't try to actually evaluate to much of it, evaluating the above predicates might require evaluating the entire *allVals* list, which is not really feasible.

If a formula is satisfiable, we might find a satisfying model fairly quickly, e.g.:

```
ghci 52> satisfiable (P1 HasMustache John)
    True
```

```
ghci 53> satisfiable $ Dsj [P1 HasMustache John, Ng (P1 HasMustache John)]
    True
```

```
ghci 54> satisfiable $ P2 Loves Noam Noam
    True
```

```
ghci 55> satisfiable $ Ng (P2 Loves Noam Noam)
    True
```

But this is not guaranteed to happen. And checking for tautologies or contradictions definitely requires evaluating the entire *allVals* list.

Compiling the Haskell program will help a little bit. But what we really need is to think about better ways to identify tautologies and contradictions that do not involve a naive brute-force search through the space of all possible models.

4 Checking for entailment between two formulas

The same point applies to determining entailment between two formulas. It is easy to define entailment (it's the same definition as the one for propositional logic):

(17) $implies :: Form \to Form \to Bool$
$implies\ f1\ f2 = contradiction\ (Cnj\ [f1, Ng\ f2])$

But actually checking whether a formula *f1* entails another formula *f2* requires evaluating all the models in *allVals*, so it is pretty hopeless.

5 Context Set update

We define the update function as shown below:

(18) $update :: [(BasicExp \to BasicInt)] \to Form \to [(BasicExp \to BasicInt)]$
$update\ vals\ f = [m \mid m \leftarrow vals, eval\ m\ f]$

Calling this function is easy to evaluate as long as the input Context Set is not too large, for example:

```
ghci 56> let cs1 = take 2000010 allVals
```

```
ghci 57> length cs1
2000010
```

```
ghci 58> let cs2 = update cs1 (P2 Loves Noam Noam)
```

```
ghci 59> length cs2
1000448
```