Predicate logic cont.

Syntax and semantics

Patrick D. Elliott

June 6, 2023

Homework

- Chapter 12, *Signaling Adversity* of ”Haskell from first principles”.

Predicate logic

Recap: Implementation in haskell

```haskell
type Name = String
newtype Var = Var Name deriving Eq

data Formula = Atomic String [Var]
    | Neg Formula
    | Formula `Impl` Formula
    | Formula `Conj` Formula
    | Formula `Disj` Formula
    | Forall Var Formula
    | Exists Var Formula
    deriving Eq
```

Writing helper functions

```
at :: String -> String -> Formula
at p v = Atomic p [Var "v"]

forall :: String -> Formula -> Formula
forall v = Forall (Var "v")
```

Example formula

```
_allDogsBark = forall "x" $ at "dog" "x" `Impl` at "bark" "x"
```

$$\forall x[dog(x) \rightarrow bark(x)]$$

- A semantics for predicate logic is stated relative to a *model* and an *assignment*.
  - A model consists of a *domain of individuals* D, and an interpretation function I mapping predicate symbols to boolean-valued functions.
    - I maps a predicate symbol of arity 0 to a boolean value.
    - I maps a predicate symbol of arity 1 to a function $f : D \to \{\textbf{True}, \textbf{False}\}$.
    - I maps a predicate symbol of arity 2 to a function $f : D \times D \to \{\textbf{True}, \textbf{False}\}$
    - ...and so on.
- You're probably more familiar with a presentation where I maps predicate symbols to *sets* rather than functions, but this is equivalent.

An *assignment function* g is a total function from the set of variables Var the domain of individuals D.

$$g_1 := \begin{bmatrix} x \rightarrow & \textbf{Bart} \\ y \rightarrow & \textbf{Milhouse} \\ z \rightarrow & \textbf{Bart} \\ \ldots & \end{bmatrix}$$

We can now define $[\![.]\!]^{M,g}$ for atomic sentences, where $[\![.]\!]^{M,g}$ is a total function from wffs of predicate logic to boolean values.

- $[\![P(x_1, \ldots, x_n)]\!]^{M,g} = I(P)(g(x_1), \ldots, g(x_n)))$

The semantics of the logical connectives is the same as in propositional logic.

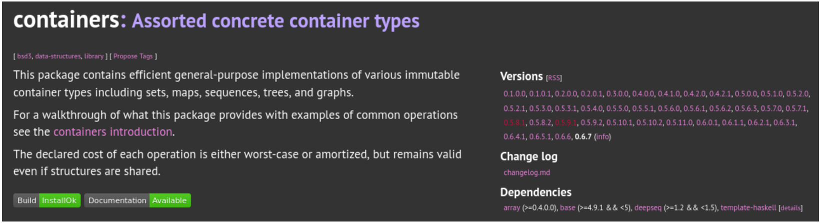
- $[\![\phi \wedge \psi]\!]^{M,g} = \mathbf{True}$ iff $[\![\phi]\!]^{M,g} = \mathbf{True}$ and $[\![\psi]\!]^{M,g} = \mathbf{True}$
- $[\![\phi \vee \psi]\!]^{M,g} = \mathbf{True}$ iff $[\![\phi]\!]^{M,g} = \mathbf{True}$ or $[\![\psi]\!]^{M,g} = \mathbf{True}$
- $[\![\phi \rightarrow \psi]\!]^{M,g} = \mathbf{True}$ iff $[\![\phi]\!]^{M,g} = \mathbf{False}$ or $[\![\psi]\!]^{M,g} = \mathbf{True}$
- $[\![\neg\phi]\!]^{M,g} = \mathbf{True}$ iff $[\![\phi]\!]^{M,g} = \mathbf{False}$

- The semantics for quantifiers is a little more involved.:
  - $[\![\exists x \phi]\!]^{M,g} =$
    True iff there is some assignment g' s.t. $g[x]g'$ and $[\![\phi]\!]^{M,g'} = \textbf{True}$
  - $[\![\forall x \phi]\!]^{M,g} =$
    True iff there is no assignment g' s.t. $g[x]g'$ and $[\![\phi]\!]^{M,g'} = \textbf{False}$

- $\mathrm{g}[x]\mathrm{g}'$ means that assignments g and g' differ *only* in the value they assign to x.

Interlude: hackage

- Hackage is the haskell package repository.
    - Chances are, if we can’t find the datatype/function we need in the the haskell prelude, we can find it in a package on hackage.
    - Today I’ll make use of one of the most ubiquitous haskell packages - *containers* - which provides an implementation of sets in the module `Data.Set`.



- To add packages from hackage to your haskell project there are basically two options.
    - Cabal, the official haskell project/package manager.
        - If you've installed ghc, you probably already have this installed.
    - Stack, an unofficial, but simple and widely-used project/package manager.

Qualified import

- It's good practice to use a *qualified* import for `Data.Set`, since some of the exported functions overlap with prelude (e.g., `delete`).
- We can use the *overloaded lists* language extension to simply express sets using list syntax.
  - Alternatively, you can build a set explicitly using `S.fromList`.

```haskell
{-# LANGUAGE OverloadedLists #-}

import qualified Data.Set as S

aSet :: S.Set Int
aSet = [1,2,1,3]
```

```
ghci> aSet
fromList [1,2,3]
```

Union, intersection, and deleting

- `Data.Set` means that we no longer have to worry about accidentally duplicating elements of a list.
- The order of elements in a set doesn't matter.

```
ghci> [1,2,3] `S.union` [1,2,4]
fromList [1,2,3,4]
ghci> [1,2,3] `S.intersection` [1,2,4]
fromList [1,2]
ghci> S.delete 1 [1,2,1,3]
fromList [2,3]
ghci> fromList [1,2] == fromList [2,1]
True
```

Model theoretic semantics

Implementing a model

- In order to implement a semantics for predicate logic in Haskell, we first need to implement a *model*.
- The most convenient choice for an entity type is `Int`, since we can define some predicates in terms of built in functions in Haskell.

```
newtype Entity = E Int deriving (Eq,Show,Ord)

domE :: S.Set Entity
domE = S.fromList $ E <$> [1..10]
```

- Note: since we want computation for quantificational statements such as "everyone left" to *terminate*, it's particularly important that we define a finite domain as a subset of the set of integers (i.e., `domE`).

Adding predicates

```
oddP :: [Entity] -> Bool
oddP [E n] = odd n
oddP _ = undefined

evenP :: [Entity] -> Bool
evenP [E n] = even n
evenP _ = undefined

isEqualR :: [Entity] -> Bool
isEqualR [E n, E n'] = n == n'
isEqualR _ = undefined

evenlyDivisibleR :: [Entity] -> Bool
evenlyDivisibleR [E n,E n'] = (n `rem` n') == 0
evenlyDivisibleR _ = undefined
```

- Note: functions are simply left undefined if the wrong number of arguments are supplied.

Interpretation function

```haskell
type I = String -> [Entity] -> Bool

lexicon :: String -> [Entity] -> Bool
lexicon "odd" = oddP
lexicon "even" = evenP
lexicon "evenlyDivisible" = evenlyDivisibleR
lexicon "isEqual" = isEqualR
lexicon _ = const True
```

Gathering variables using `Data.Set`

```haskell
allVars :: Formula -> S.Set Var
allVars s = case s of
  (Atomic p vs) -> S.fromList vs
  (Neg p) -> allVars p
  (p `Impl` q) -> allVars p `S.union` allVars q
  (p `Conj` q) -> allVars p `S.union` allVars q
  (p `Disj` q) -> allVars p `S.union` allVars q
  (Forall v p) -> v `S.insert` allVars p
  (Exists v p) -> v `S.insert` allVars p
```

Making assignments

```
import Control.Monad (replicateM)
import qualified Data.Map as M

type Assignment = M.Map Var Int

mkAssignments :: [Var] -> [Entity] -> S.Set Assignment
mkAssignments vs d = S.fromList [M.fromList $ zip vs es | es <-
 ↪  replicateM (length vs) d]
```

```haskell
import qualified Data.Map as M

type Dom = S.Set Entity

eval :: I -> Dom -> Assignment -> Formula -> Bool
eval i d g (Atomic p vs) = i p [ g M.! v | v <- vs]
eval i d g (Neg p) = not $ eval i d g p
eval i d g (p `Conj` q) = eval i d g p && eval i d g q
eval i d g (p `Disj` q) = eval i d g p || eval i d g q
eval i d g (p `Impl` q) = not (eval i d g p) || eval i d g q
eval i d g (Exists v p) = undefined
eval i d g (Forall v p) = undefined
```

Assignment modification

```haskell
modify :: Assignment -> Var -> Entity -> Assignment
modify g v x = M.insert v x g
```

Quantification via generalized conjunction/disjunction

```
eval i d g (Exists v p) = disjoin [ eval i d (modify g v x) p | x
 ↪  <- S.toList d]
eval i d g (Forall v p) = conjoin [eval i d (modify g v x) p | x
 ↪  <- S.toList d]
```

Maybe

The `Maybe` datatype

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

`Maybe` is used to explictly reason about undefinedness.

Writing a function with `Maybe`

- We use `Maybe` to write partial functions.
- For example, here is a safe version of `head` using `Maybe`:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

Kinds

- Kinds are types *one level up*, used to describe the types of *type constructors* such as `Maybe`.

```
ghci> :kind Int
Int :: *
ghci> :k Bool
Bool :: *
ghci> :k Char
Char :: *
```

Here is a datatype isomorphic to `Maybe`:

```
data Example a = Blah | Woot a
```

- Question: what is the *kind* of `Example`

- Question: What is the kind of `Maybe`?
- Question: What is kind of the tuple type constructor `(,)`?
- Question: What is kind of the list type constructor `[]`?
- Question: What is kind of the function type constructor `(->)`?

Applying type constructors

Which of the following are concrete types?

```
ghci> :k Maybe Maybe
ghci> :k Maybe Bool
ghci> :k Example (Maybe (Maybe Bool))
ghci> :k Maybe Example
ghci> :l Maybe (Example Int)
```

- A functor is a way to apply a function over or around some immutable structure.
- Functors are a notion from category theory (a mapping from categories to categories), implemented in Haskell as a *type-class*.
- Remember, type-classes categorize types based on certain well-defined behaviours.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Notice that f is a *higher-kinded type*.

Examples of `fmap`

```
ghci> fmap (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
ghci> fmap not (Just True)
Just False
ghci> fmap not Nothing
Nothing
```

This means that `[]` and `Maybe` are both higher-kinded types which implement the typeclass `Functor`.

`fmap` as function application

```
(<$>) :: Functor f =>
  (a -> b) -> f a -> f b
($) ::
  (a -> b) ->   a ->   b
```

```
ghci> (\n -> n+1) <$> Just 3
Just 4
```

Functor laws

- Instances of the `Functor` type class should abide by two basic laws.
  - Alert: ghci won't always warn you if you write a functor instance that doesn't obey these laws!
- The laws are:
  - Identity.
  - Composition.

The identity law

```
fmap id == id
```

- Question: what are the types of `fmap` and `id` in this expression?
- Question: what does this law guarantee?

```
fmap (f . g) == fmap f . fmap g
```

- If we compose two functions and **`fmap`** over some structure, we should get the same result as if we mapped and then composed them.

```
fmap ((+1) . (*2)) [1..5]
fmap (+1) . fmap (*2) $ [1..5]
```

By composing `fmap` with itself we can tunnel into complex expressions.

```
(fmap . fmap) (++ "lol") (Just ["Hi,","Hello"])
```

- Exercise: normalize this expression by hand.

$\mathcal{F}in$

References