M

CFG

Syntax and semantics

Patrick D. Elliott

June 4, 2023

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
```

Helper functions

```haskell
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 $f : D \to \{\mathbf{True}, \mathbf{False}\}$.
        - I maps a predicate symbol of arity 2 to a function $f : D \times D \to \{\mathbf{True}, \mathbf{False}\}$
        - ...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* $\mathbf{g}$ 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.

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

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

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

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

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

Interlude: hackage

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`.

containers: Assorted concrete container types

[bsd3, data-structures, library] [Propose Tags]

This package contains efficient general-purpose implementations of various immutable container types including sets, maps, sequences, trees, and graphs.

For a walkthrough of what this package provides with examples of common operations see the containers introduction.

The declared cost of each operation is either worst-case or amortized, but remains valid even if structures are shared.

Build InstallOk Documentation Available

Versions [RSS]

0.1.0.0, 0.1.0.1, 0.2.0.0, 0.2.0.1, 0.3.0.0, 0.4.0.0, 0.4.1.0, 0.4.2.0, 0.4.2.1, 0.5.0.0, 0.5.1.0, 0.5.2.0, 0.5.2.1, 0.5.3.0, 0.5.3.1, 0.5.4.0, 0.5.5.0, 0.5.5.1, 0.5.6.0, 0.5.6.1, 0.5.6.2, 0.5.6.3, 0.5.7.0, 0.5.7.1, 0.5.8.1, 0.5.8.2, 0.5.9.1, 0.5.9.2, 0.5.10.1, 0.5.10.2, 0.5.11.0, 0.6.0.1, 0.6.1.1, 0.6.2.1, 0.6.3.1, 0.6.4.1, 0.6.5.1, 0.6.6, **0.6.7** (info)

Change log

changelog.md

Dependencies

array (>=0.4.0.0), base (>=4.9.1 && <5), deepseq (>=1.2 && <1.5), template-haskell [details]

- 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.

- 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`.

```
{-# LANGUAGE OverloadedLists #-}

import qualified Data.Set as S

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

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

- `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.

```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

```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
```

```haskell
import Control.Monad (replicateM)

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

Eval

```haskell
type Dom = [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
```

$\mathcal{F}in$

References