Propositional Logic in Haskell

Executable Language Specification

Professor William L. Harrison

September 30, 2016

Today

- last time: "pencil and paper" language design (Propositional Logic)
- today: representing the design in Haskell
- HW3 out two days ago

Review: The Language Syntax as Context Free Grammar

Before giving a precise definition, let's consider an example. Let *Var* be an infinite set of symbols. We will refer to typical elements of *Var* with lower case roman letters (e.g., p, q, r, etc.). Assume $\{(,),\neg,\wedge\} \cap \mathit{Var} = \emptyset$, then let alphabet A be the set $\{(,),\neg,\wedge\} \cup \mathit{Var}$. Here is a CFG:

$$
\begin{aligned}
\mathit{Prop} &\rightarrow p \qquad \text{for any } p \in \mathit{Var} && (1)\\
\mathit{Prop} &\rightarrow (\neg \mathit{Prop}) && (2)\\
\mathit{Prop} &\rightarrow (\mathit{Prop} \supset \mathit{Prop}) && (3)
\end{aligned}
$$

This CFG defines a language, denoted $\mathcal{L}(\mathit{Prop})$.

Review: Deriving members of $\mathcal{L}(Prop)$

- How do we determine if a particular sequence[1] of symbols from A is in $\mathcal{L}(Prop)$?
- We perform a *derivation* of the string.
- For instance, is the string $(\neg p) \in \mathcal{L}(Prop)$? Yes, and here's the derivation:

$$\begin{aligned} Prop &\rightarrow (\neg Prop) \\ &\rightarrow (\neg p) \end{aligned}$$

[1] I use "string" and "sequence of symbols" interchangably.

Language Syntax as a Haskell `data` declaration

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop
```

Language Syntax as a Haskell `data` declaration

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop
```

Compare with CFG version

$$
\begin{array}{lcll}
Prop & \rightarrow & p & \text{for any } p \in Var \\
Prop & \rightarrow & (\neg Prop) & \\
Prop & \rightarrow & (Prop \supset Prop) &
\end{array}
$$

Example

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop

negp = Not (Atom "p")
```

Example

```
type Var  = String
data Prop = Atom Var
          | Not Prop
          | Imply Prop Prop

negp = Not (Atom "p")
```

Compare with:

$$\begin{aligned} Prop &\rightarrow (\ \neg\ Prop\ ) \\ &\rightarrow (\ \neg\ p\ ) \end{aligned}$$

Testing It Out, Part I

```
*PropLogic> negp
<interactive>:1:1:
    No instance for (Show Prop)
      arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show Prop)
    In a stmt of an interactive GHCi command: print it
*PropLogic>
```

"No instance for (Show Prop)"

- This means that we have to write a function, $show\ ::\ Prop \rightarrow String$.

“No instance for (Show Prop)”

- This means that we have to write a function, $\mathit{show} :: \mathit{Prop} \rightarrow \mathit{String}$.
- To write a function of a particular type, we **always** start off from its **type template**.
  - It is usually the right way to go.

"No instance for (Show Prop)"

- This means that we have to write a function, $show :: Prop \rightarrow String$.
- To write a function of a particular type, we **always** start off from its **type template**.
  - It is usually the right way to go.
- The type template for show is determined by the definition of Prop data type:

```
show (Atom p)              = undefined
show (Not prop)            = undefined
show (Imply prop1 prop2) = undefined
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)            =  p
show (Not prop)          =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =  p
show (Not prop)            =  "(-" ++ show prop ++ ")"
show (Imply prop1 prop2) =
```

Filling in type template of show

Recall String concatenation:

```
*PropLogic> "hey" ++ "pal"
"heypal"
```

Replacing `undefined`:

```
show (Atom p)              =  p
show (Not prop)            =  "(-" ++ show prop ++ ")"
show (Imply prop1 prop2) =
     "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

Show Prop

Make this into an instance declaration:

```
instance Show Prop where
  show (Atom p)             = p
  show (Not prop)           = "(-" ++ show prop ++ ")"
  show (Imply prop1 prop2) =
       "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

Show Prop

Make this into an instance declaration:

```
instance Show Prop where
  show (Atom p)             = p
  show (Not prop)           = "(-" ++ show prop ++ ")"
  show (Imply prop1 prop2) =
       "(" ++ show prop1 ++ " => " ++ show prop2 ++ ")"
```

```
*PropLogic> negp
(-p)
```

Another Instance Example

```
instance Eq Prop where
  (Atom p) == (Atom q)          = p == q
  (Not x) == (Not y)            = x == y
  (Imply x y) == (Imply u v) = (x == u) && (y == v)
  _ == _                         = False
```

Recall: Definitional Extensions

Definition (Disjunction, Conjunction and Equivalence)

Familiar connectives are defined by:

$$(\varphi \vee \gamma) \text{ is } \neg\varphi \supset \gamma \qquad \text{(disjunction)}$$

$$(\varphi \wedge \gamma) \text{ is } \neg(\neg\varphi \vee \neg\gamma) \qquad \text{(conjunction)}$$

$$(\varphi \leftrightarrow \gamma) \text{ is } (\varphi \supset \gamma) \wedge (\gamma \supset \varphi) \qquad \text{(equivalence)}$$

How do we represent these definitional extensions?

Defined Connectives as Functions

```
orPL ::
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::
```

Defined Connectives as Functions

```
orPL ::  Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL ::  Prop -> Prop -> Prop
andPL phi gamma =
```

Defined Connectives as Functions

```
orPL :: Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL :: Prop -> Prop -> Prop
andPL phi gamma =  Not (orPL (Not phi) (Not gamma))

iffPL ::
```

Defined Connectives as Functions

```
orPL :: Prop -> Prop -> Prop
orPL phi gamma  =  Imply (Not phi) gamma

andPL :: Prop -> Prop -> Prop
andPL phi gamma =  Not (orPL (Not phi) (Not gamma))

iffPL :: Prop -> Prop -> Prop
iffPL phi gamma = andPL (Imply phi gamma) (Imply gamma phi)
```

Recall: Axiom System for Propositional Logic

$$\varphi \supset (\gamma \supset \varphi) \tag{Ax.1}$$

$$(\varphi \supset (\gamma \supset \psi)) \supset ((\varphi \supset \gamma) \supset (\varphi \supset \psi)) \tag{Ax.2}$$

$$((\neg\gamma \supset \neg\varphi) \supset ((\neg\gamma \supset \varphi) \supset \gamma)) \tag{Ax.3}$$

There is only one inference rule in propositional logic, namely *Modus Ponens*.

$$\frac{\varphi \quad \varphi \supset \gamma}{\gamma} \text{ (MP)}$$

We want to represent axioms somehow.

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =  Imply pre post
   where pre  = Imply phi (Imply gamma psi)
         post = Imply
                    (Imply phi gamma)
                    (Imply phi psi)

axiom3 phi gamma =
```

Representing the Axioms as Functions

With this approach, we define functions that, given φ, γ, and ψ, provide us with the axiom instance.

```
axiom1 ::  Prop -> Prop -> Prop
axiom1 phi gamma     =  Imply phi (Imply gamma phi)

axiom2 ::  Prop -> Prop -> Prop -> Prop
axiom2 phi gamma psi =  Imply pre post
   where pre  = Imply phi (Imply gamma psi)
         post = Imply
                  (Imply phi gamma)
                  (Imply phi psi)

axiom3 phi gamma =  Imply pre post
   where pre  = Imply (Not gamma) (Not phi)
         post = Imply hyp gamma
            where hyp  = Imply (Not gamma) phi
```

Review: Axiom Instances

An instance of an axiom is a substitution of a wff for φ, γ, ψ
Instances of Axiom 1 $(\varphi \supset (\gamma \supset \varphi))$ include

Instance	Substitution
$A \supset (B \supset A)$	$[\varphi \mapsto A, \gamma \mapsto B]$
$A \supset ((A \supset A) \supset A)$	$[\varphi \mapsto A, \gamma \mapsto (A \supset A)]$
$\vdots$	$\vdots$

Axioms as a Data Type

```
data Axiom  = Ax1 Prop Prop
            | Ax2 Prop Prop Prop
            | Ax3 Prop Prop
              deriving Eq
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

```
*PropLogic> Ax1 negp (Atom "q")
((-p) => (q => (-p)))
*PropLogic>
```

Axioms as a Data Type, cont'd

```
instance Show Axioms where
  show (Ax1 phi gamma)     = show (axiom1 phi gamma)
  show (Ax2 phi gamma psi) = show (axiom2 phi gamma psi)
  show (Ax3 phi gamma)     = show (axiom3 phi gamma)
```

```
*PropLogic> Ax1 negp (Atom "q")
((-p) => (q => (-p)))
*PropLogic>
```

What about the inference rule in propositional logic, namely *Modus Ponens*.

$$\frac{\varphi \quad \varphi \supset \gamma}{\gamma} \text{ (MP)}$$

Review: Formal Proofs

Definition (Proof)

Let Φ be the sequence $\varphi_1, \ldots, \varphi_n$ of propositional wffs. Then, Φ is a *proof* of φ_n if, and only if, for each φ_i in Φ, φ_i is either:

- an instance of Ax.1, Ax.2, or Ax.3, or
- there are φ_j and φ_k such that $j < i$ and $k < i$ and φ_i follows from φ_j and φ_k by *MP*.

φ_n is said to be a **theorem.** Note that the definition of "theorem" is recursive.

Example Proof

Say I want to prove that $A \supset A$.

1. $(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))$ Ax.2
2. $A \supset ((A \supset A) \supset A)$ Ax.1
3. $(A \supset (A \supset A)) \supset (A \supset A)$ MP2,1
4. $A \supset (A \supset A)$ Ax.1
5. $A \supset A$ MP3,4

Review: Proof as Tree, $A \supset A$

$$
\dfrac{\dfrac{}{A \supset (A \supset A)}\ \text{(Ax.1)} \qquad \dfrac{\dfrac{}{A \supset ((A \supset A) \supset A)}\ \text{(Ax.1)} \qquad \dfrac{}{\begin{array}{l}(A \supset ((A \supset A) \supset A)) \supset \\ \qquad ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\ \text{(Ax.2)}}{(A \supset (A \supset A)) \supset (A \supset A)}\ \text{(MP)}}{A \supset A}\ \text{(MP)}
$$

Theorems

Hint: Theorem will be recursive.

```
data Theorem =
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
             | ModusPonens Theorem Theorem Prop
```

Theorems

Hint: Theorem will be recursive.

```
data Theorem = AxiomInst Axiom
             | ModusPonens Theorem Theorem Prop

instance Show Theorem where
  show (AxiomInst ax)             = show ax
  show (ModusPonens x y z)
     =         show x ++ "   " ++ show y
       ++ "\n--------------------------------\n    " ++
                       show z
```

Example

$$
\dfrac{
\dfrac{}{A \supset (A \supset A)}\,(\text{Ax.1})
\qquad
\dfrac{
\dfrac{}{A \supset ((A \supset A) \supset A)}\,(\text{Ax.1})
\qquad
\dfrac{}{\begin{array}{c}(A \supset ((A \supset A) \supset A)) \supset \\ ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\,(\text{Ax.2})
}{(A \supset (A \supset A)) \supset (A \supset A)}\,(\text{MP})
}{A \supset A}\,(\text{MP})
$$

Example

$$
\dfrac{
\dfrac{}{A \supset (A \supset A)}\,\text{(Ax.1)}
\qquad
\dfrac{
\dfrac{}{A \supset ((A \supset A) \supset A)}\,\text{(Ax.1)}
\quad
\dfrac{}{\begin{array}{c}(A \supset ((A \supset A) \supset A)) \supset \\ ((A \supset (A \supset A)) \supset (A \supset A))\end{array}}\,\text{(Ax.2)}
}{(A \supset (A \supset A)) \supset (A \supset A)}\,\text{(MP)}
}{A \supset A}\,\text{(MP)}
$$

```
a = Atom "A"

subproof = ModusPonens hyp1 hyp2 conc
  where hyp1 = AxiomInst (Ax1 a (Imply a a))
        hyp2 = AxiomInst (Ax2 a (Imply a a) a)
        conc = Imply (Imply a (Imply a a))
                     (Imply a a)
```

Example

$$\dfrac{\dfrac{}{A \supset (A \supset A)}\,(\text{Ax.1}) \qquad \dfrac{\dfrac{}{A \supset ((A \supset A) \supset A)}\,(\text{Ax.1}) \qquad \dfrac{}{(A \supset ((A \supset A) \supset A)) \supset ((A \supset (A \supset A)) \supset (A \supset A))}\,(\text{Ax.2})}{(A \supset (A \supset A)) \supset (A \supset A)}\,(\text{MP})}{A \supset A}\,(\text{MP})$$

```haskell
a = Atom "A"

subproof = ModusPonens hyp1 hyp2 conc
  where hyp1 = AxiomInst (Ax1 a (Imply a a))
        hyp2 = AxiomInst (Ax2 a (Imply a a) a)
        conc = Imply (Imply a (Imply a a))
                     (Imply a a)
```

```
*PropLogic> subproof
(A => ((A => A) => A))   ((A => ((A => A) => A)) => ((A => (A => A)) => (A => A)))
--------------------------------------------------------------------------------
                  ((A => (A => A)) => (A => A))
```

Food for Thought

Is it possible to define a `Theorem` value in Haskell that is not a theorem?

Next Time

Finish up the specification of the Propositional Logic in Haskell