Interpreting logical expressions

Patrick D. Elliott

June 4, 2024

Propositional logic: semantics

- Expressions of propositional logic are interpreted relative to an *assignment function* (let's call it g).
  - This is a total function from the propositional variables in the language to truth-values.

$$g_1 := \begin{bmatrix} p \to \textbf{True} \\ q \to \textbf{True} \\ r \to \textbf{False} \\ \dots \end{bmatrix}$$

- Specifying a semantics for propositional logic involves recursively defining a *denotation function* $[\![.]\!]^g$, which maps expressions to truth-values.
    - If p is atomic, then $[\![p]\!]^g = g(p)$.
    - $[\![\neg\phi]\!]^g = \textbf{True}$ iff $[\![\phi]\!]^g = \textbf{False}$.
    - $[\![\phi \wedge \psi]\!]^g = \textbf{True}$ iff $[\![\phi]\!]^g = \textbf{True}$ & $[\![\psi]\!]^g = \textbf{True}$
    - $[\![\phi \vee \psi]\!]^g = \textbf{True}$ iff $[\![\phi]\!]^g = \textbf{True}$ or $[\![\psi]\!]^g = \textbf{True}$

- $[\![\neg(p \wedge q) \vee r]\!]^g = \mathbf{True}$ iff …
    - $[\![\neg(p \wedge q)]\!]^g = \mathbf{True}$ or $[\![r]\!]^g = \mathbf{True}$
    - $[\![(p \wedge q)]\!]^g = \mathbf{False}$ or $g(r) = \mathbf{True}$
    - $[\![p]\!]^g = \mathbf{False}$ or $[\![q]\!]^g = \mathbf{False}$ or $g(r) = \mathbf{True}$
    - $g(p) = \mathbf{False}$ or $g(q) = \mathbf{False}$ or $g(r) = \mathbf{True}$
- The denotation specifies the conditions a variable assignment g must meet in order for the sentence to be judged *true* relative to it.
- This information can be encoded as a truth-table.
- Given a fixed expression ϕ, you can model $[\![\phi]\!]$ as a function from assignments to truth-values; a truth-table gives a condensed extension of this function.

- $[\![\neg(p \wedge q) \vee r]\!]^g$ = **True** iff $g(p)$ = **False** or $g(q)$ = **False** or $g(r)$ = **True**

p	q	r	$\neg(p \wedge q) \vee r$
1	1	1	1
1	1	0	0
1	0	1	1
0	1	1	1
0	0	1	1
0	1	0	1
1	0	0	1
0	0	0	1

Truth-table: implementation in Haskell

- As an informal assignment, I asked you to try to implement a program that maps expressions of `PropL` (our recursive datatype for expressions of propositional logic) to a *truth-table*.
- In order to implement truth-tables in Haskell, we first need to decide on a couple of datatypes:
    - The type for a *variable assignment*.
    - The type for a *truth-table*.

- A straightforward option for a variable assignment is a *list of tuples*, where variables are paired with Truth-Values.
- Following our implementation of `PropL`, we'll assume that variables are just strings.
- We can declare a *type synonym* using the `type` keyword to declare a convenient abbreviation for this complex type:

```
type VarAssignment = [(String,Bool)]
```

```
ghci> [("p",True),("q",False),("r",True)]
```

- We need a way of ”looking up” what value an assignment gives to a variable.
- In order to do this, we’ll define a function `lookup` of type `VarAssignment -> String -> Bool`.

```
lookup :: VarAssignment -> String -> Bool
lookup [] _ = undefined
lookup ((key,val):ps) x = if key == x then val else lookup ps x
```

- `lookup` recurses through the list until it finds a tuple whose first element (the ”key”) matches the argument `x`; it returns the second element of that tuple (the ”value”).

- Exercise: what's the most general (i.e., maximally polymorphic) type that the definition of `lookup` is compatible with?

Aside: improving lookup

- Our definition of `lookup` will do for the purposes of this exercise, but it has some clear deficiencies:
    - Variables can be paired with multiple values (bad); `lookup` ignore all but the first paired value in the list.
    - `lookup` is unsafe (i.e., it's a partial function). Specifically, if the provided key isn't part of the variable assignment.
- The solution is to swap out the list of tuples with a data structure tailored for key value pairs - `Map` from the `Data.Map` module, which comes with a built-in *safe* lookup function.
    - We'll learn more about how to make functions like `Map` safe when we learn about the `Maybe` constructor, later in the semester.

- Now that we have some concept of an assignment, we can recursively define a denotation function `interpretAtA`, which takes a variable assignment, an expression of `PropL` and returns a boolean.
    - this is the core implementation of the semantics of propositional logic.
    - As a reminder, here's the `PropL` datatype.

```
data PropL = PVar String | PNot PropL | PropL `PAnd` PropL | PropL
↪  `POr` PropL deriving Eq
```

Base of the recursion

```
interpretAtA :: VarAssignment -> PropL -> Bool
interpretAtA v (PVar p) = lookup v p
```

- Note that since `lookup` is unsafe, `interpretAtA` is also unsafe.
- This means we need to be careful to provide `interpretAtA` with an assignment which provides a value for every variable in the expression (in fact, we'll automate this).

- We can use Haskell's built-in boolean operators to provide a complete denotational semantics for **`Propl`**.
- Note that the semantics here completely parallels the recursive definition of the denotation function.

```
interpretAtA :: VarAssignment -> PropL -> Bool
interpretAtA a (PVar p) = lookup a p
interpretAtA a (PNot p) = not (interpretAtA a p)
interpretAtA a (p `PAnd` q) = interpretAtA a p && interpretAtA a q
interpretAtA a (p `POr` q) = interpretAtA a p || interpretAtA a q
```

When we have a function definition which does different things depending on the form of the argument it receives (via pattern matching), we can often make the definition more terse by using a *case expression*.

```
interpretAtA a exp = case exp of
  (PVar p) -> lookup a p
  (PNot p) -> not (interpretAtA a p)
  (p `PAnd` q) -> interpretAtA a p && interpretAtA a q
  (p `POr` q) -> interpretAtA a p || interpretAtA a q
```

Example

```
ghci> _g1 = [("p",True),("q",False),("r",True)]
ghci> _form1 = PNot (PVar "p" `PAnd` PVar "q") `POr` PVar "r"
ghci> interpretAtA _g1 _form1
True
```

- We've defined `interpretA` which maps an assignment and a formula to a truth-value (our denotation function).
- Our next task will be to define a function that generates all "relevant" assignments, given a formula. We'll call this `mkAssignments`.
- First we need a list of all the variables which occur in a formula - we'll make use of our existing `gatherNames` function.

Here, we import a built-in function from the `Data.List` module for removing duplicate entries, rather than implementing it ourselves.

```
import Data.List (nub)

gatherNames' :: PropL -> [String]
gatherNames' (PVar s) = [s]
gatherNames' (PNot p) = gatherNames p
gatherNames' (PAnd p q) = gatherNames p ++ gatherNames q
gatherNames' (POr p q) = gatherNames p ++ gatherNames q

gatherNames = nub . gatherNames'
```

Given a formula `gatherNames` will give you a list of variables in that formula.

```
ghci> gatherNames (PNot (PVar "p" `PAnd` PVar "q") `POr` PVar "r")
["p","q","r"]
```

Generating assignments

- Now that we have a list of variables, we need to generate all of the possible assignments of those variables to truth values, i.e., we need a function `mkAssignments` of the following type:

```
mkAssignments :: [String] -> [VarAssignment]
```

- This is probably the hardest part of the task, and will involve some advanced list manipulation.

The secret sauce: `replicateM` from `Control.Monad`

```
ghci> import Control.Monad (replicateM)
ghci> replicateM 3 "01"
["000","001","010","011","100","101","110","111"]
ghci> replicateM 3 [True,False]
[[True,True,True],[True,True,False],[True,False,True],
 [True,False,False],[False,True,True],[False,True,False],
 [False,False,True],[False,False,False]]
```

We use this to create all sequences of boolean values of length n.

In order to make assignments, we *zip* a list of variables with a list of boolean values:

```
ghci> import Data.List (zip)

ghci> zip ["p","q","r"] [True,True,True]
[("p",True),("q",True),("r",True)]
```

We need to do this for every list of boolean values of length n, where n is the number of variables we have.

Putting it all together

```
mkAssignments :: [String] -> [VarAssignment]
mkAssignment vs = [zip vs ts | ts <- replicateM (length vs)
 ↪  [True,False]]
```

```
ghci> mkAssignments ["p","q"]
[[("p",True),("q",True)],
 [("p",True),("q",False)],
 [("p",False),("q",True)],
 [("p",False),("q",False)]]
```

To get all the ”relevant” assignments for a formula p, we first gather all the variables in p, and then apply `mkAssignments` to the resulting list.

```
pAssignments :: PropL -> [VarAssignment]
pAssignments = mkAssignments . gatherNames
```

N.b., it’s crucial that the definition of `gatherNames` removes duplicates (`nub`) in order for this to work properly.

In order to generate a truth-table, we simply pair each assignment in the output of `pAssignments`, with the denotation of the formula at that assignment:

```
mkTruthTable :: PropL -> [(VarAssignment,Bool)]
mkTruthTable p = [(a,interpretAtA a p) | a <- pAssignments p]
```

Example output

```
ghci> mkTruthTable (PNot (PVar "p" `PAnd` PVar "q") `POr` PVar
 ↪  "r")
[([("p",True),("q",True),("r",True)],True),
 ([("p",True),("q",True),("r",False)],False),
 ([("p",True),("q",False),("r",True)],True),
 ([("p",True),("q",False),("r",False)],True),
 ([("p",False),("q",True),("r",True)],True),
 ([("p",False),("q",True),("r",False)],True),
 ([("p",False),("q",False),("r",True)],True),
 ([("p",False),("q",False),("r",False)],True)]
```

$\mathcal{F}in$

References