CS4450 Lecture 9 Notes

Friday, September 7, 2018 3:04 PM

Today: Finish up Propositional Logic language specification and truth tables.

The TAs (Trevor and Matt) will have additional office hours in 319 Engineering Building. We will not have a lecture on Monday.

```
-- List data type
data [a] = [] | a : [a]
-- Two data constructors: empty list, [], and "cons", ":"
-- E.g., "[1,2,3]" is really "1 : (2 : (3 : []))". This is a recursive
data definition. The bracket method is "syntactic sugar", which means
it's easier for humans to read.
-- It takes argument a and list a (which is represented as [a]) and
returns [a].
(:) :: -> a -> [a] -> [a]
```

The data constructors C1 and C2 are also used in patterns:

```
data T = C1 A | C2 B
```

Patterns are data destructors, so when you have a data declaration with constructors C1 and C2, you can use them as patterns to destruct the data.

The pattern "peels off the constructor".

```
-- Ex. length:
length [] = 0
length (x:xs) = 1 + length xs
```

What the above does is that it does a recursive call on xs. You do a pattern match that defines it as xs.

Case Expressions

A Case Expression is something where you can say "case on some value, then do pattern matching on that value". It's *kind of* like a switch statement in C.

Definitional Extension: The previous definition is just *syntactic sugar* for the case expression definition.

Please note the Syntactic "Banana Peel" (Syntactic Peculiarity) of clauses: Clauses "True -> False" and "False -> True" need to line up column wise. Dr. Harrison calls it a "Banana Peel" because it's a small thing that will trip you up.

Graceless Error Reporting (Two Ways):

1. You can use the "error" function to throw an error, or "undefined". Error is type Char and undefined can be any type.

2. You can say 1 + undefined or "hey" ++ undefined. It's useful if there's a case that you don't know what you need to fill it with yet, but want it to compile.

Local definitions with where: You can locally define variables using where.

Comments: Can use -- for single line comments or {- and -} for multi line comments.

Each row in a truth table is kind of "variable binding". You replace A and B with T or F in a boolean algebra function to evaluate it. Variable binding is assigning variables to values.

Haskell Representation :[("A",True),("B",False)]. You can think of this as a Row.

You can even define a type Row as follows:

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

Safe lookup:

Lookup is kind of a key value system with an association list. You can take the key and get the value.

Maybe: You can use maybe and instead of the program crashing (if you used error), it would return nothing.

What is "Eq"? It's a comparison. You're going through the list and seeing if the key is associated with a given value. Please check the Safe lookup slide for the code example.

```
-- Truth tables
type Row = [(String,Bool)]
row1, row2, row3, row4 :: Row
row1 = [("A",True),("B",True)]
row1 = [("A",True),("B",False)]
row1 = [("A",False),("B",True)]
row1 = [("A",False),("B",False)]
eval :: Prop -> Row -> Bool
eval (Atom a) r = case lookup a r of
Just b -> b
Nothing -> error (a ++ " is unbound")
eval (Not p) r = not (eval p r)
-- This is one method of doing it.
eval (Imply p q) r = imply (eval p r) (eval q r)
where
imply True False = False
imply _ _ = True
-- An alternative is this, it's a case expression:
eval (Imply p q) r = case (eval p r, eval q r) of
(True,False) -> False
(_,_) -> True
-- In a case expression, you'll try and match the first case first,
then the second, and then the third in order. The _ is a wildcard, so
they'll match anything.
-- Could have used just one wildcard, but Professor Harrison does this
way because it looks better to him.
-- We can either type "eval ex row3" and then "eval ex row4" on
separate lines to evaluate the rows, or we can also use map.
map (eval ex) [row1, row2, row3, row4] -- Will print out
[True,True,True,True].  It takes the function and applies it to each row
so that you can get the values out.
```

One more comment: How do we write out the rows? Two variables is no big deal, but what about 10?

What we really want is something that takes a Prop, goes into it, grabs a list of variables out of it, generates all of the rows of the variables, and then does the map and outputs the result. That will be homework 2.