# Notes for Lecture 4 (Fall 2022 week 2 part 2): Type declarations and Boolean functions

Jana Dunfield

September 11, 2022

The code for this lecture is in `lec4.hs`.

## 1 Type declarations

Haskell allows (but almost never requires) *type declarations*. A type declaration can be written before[1] the relevant definition, like this:

```
triple :: Integer -> Integer
triple z = 3 * z
```

The symbol `::` is read "has type".



```
 triple    ::      Integer   ->        Integer

"triple has type   Intege    ->        Integer"
                             ^^        ^^^^^^^^
                argument type   function  result type
Synonyms:
                                       ret
                   domain              codomain
                                       range
```





We can also read `::` as saying "*should* have the type": if you write a type declaration that disagrees with the definition, Haskell will not let you load the file. For example, if we change the definition of `triple` so that it returns the Boolean value `True`, we get an error:

```
triple :: Integer -> Integer
triple z = True

*Main> :load lec4
[1 of 1] Compiling Main             ( lec4.hs, interpreted )

lec4.hs::12:
    Couldn't match expected type 'Integer' with actual type 'Bool'
    In the expression: True
    In an equation for 'triple': triple z = True
Failed, modules loaded: none.
```

[1] You're allowed to write a type declaration *after* the relevant definition, which I find confusing.

This is a situation where writing a type declaration is helpful, because it gives GHC more information about what we're trying to do. If we didn't write a type declaration, GHC would accept a definition like

```
triple z = True
```

which is a sensible function, but it doesn't have type I expect for a function named `triple`.

## 2 Functions and arguments

The logical operator NAND (not-AND) returns the negation of what AND would do, as specified by the following truth table (cf. CISC 204). I'm writing `nand` as a prefix operator because that's the simplest thing to do in Haskell.

| `p` | `q` | `nand p q` |
|---|---|---|
| `True` | `True` | `False` |
| `True` | `False` | `True` |
| `False` | `True` | `True` |
| `False` | `False` | `True` |



The shortest way to write this operator in Haskell is using the `not` and `&&` (AND) operators:

```
nand :: Bool -> Bool -> Bool
nand p q = not (p && q)
```



Load the file `lec4.hs` and check that `nand` corres[...]or example, `nand False True` should print `True`.



Another way to write `nand` is not quite as short:

```
lambdanand = (\p -> \q -> not (p && q))
```

As far as Haskell is concerned, this is the same as `nand` (in fact, GHC turns the definition of `nand` into the definition of `lambdanand`, because the lambda \ is a "more fundamental" feature in the language). If you experiment with calling `nand` and `lambdanand`, they will behave identically. The comments in `lec4.hs` go into this in more detail.

### 2.1 How many arguments?

There are two ways to read the Haskell type

```
Bool -> Bool -> Bool
```

- **First way:** Given two things of type `Bool`, return something of type `Bool`.
- **Second way:** Given one thing of type `Bool`, returns something of type `Bool -> Bool`.

The first way seems consistent with applying `nand` (or `lambdanand`) to two arguments, like this:

```
nand True False
```

Or like this (where the second argument is itself a function application):

```
nand True (not True)
```

But Haskell actually works the second way. Haskell sees the type

```
Bool -> Bool -> Bool
```

as being

```
Bool -> (Bool -> Bool)
```

If you recall syntax trees (or parse trees) for logical formulas from CISC 204, both of these—with and without parentheses—have the same syntax tree:

```
        ->
       /  \
      /    \
    Bool   ->
          /  \
         /    \
       Bool   Bool
```



Starting at the top, from the root, the general picture is

```
        ->
       /  \
      /    \
argument   result
 type       type
```



**Exercise 1.** Draw the syntax trees for the two types `Integer -> Bool) -> Int`.

For the type of `nand`, the argument type is `Bool` and the result type is `Bool -> Bool`.

If a function's argument type is `Bool` we should be able to apply the function to something (like `True` or `False`) of type `Bool`. And if the result type is `Bool -> Bool`, the result should be something of type `Bool -> Bool`. Haskell does allow this: we can define a function `lambdanandTrue` that is the result of applying `lambdanand` to `True`:

```
lambdanandTrue = lambdanand True
```

Haskell can tell us the type of this new function:

```
*Main> :type lambdanandTrue
lambdanandTrue :: Bool -> Bool
```

If we apply `lambdanandTrue` to something of type `Bool`, we get a `Bool` back:

```
*Main> lambdanandTrue False
True
```

When we apply `lambdanand` (or `nand`) to two arguments, we are really applying it—a function whose argument type is `Bool`—to one argument, and *then* applying the resulting *function* to something else of type `Bool`.

# 3 Guards

We could write `nand` by mechanically translating the truth table into `if-then-else` expressions. I will name this function `sheffer`, because the NAND operation is also called the "Sheffer stroke". The three functions `nand`, `lambdanand` and `sheffer` all "do the same thing"—they behave the same—but `sheffer` works a little differently.

```
sheffer p q =
  if (p == True) && (q == True) then False
  else if (p == True) && (q == False) then True
       else if (p == False) && (q == True) then True
            else True
```

**Exercise 2.** Without using `:type`, what is the type of `sheffer`? (Use `:type` to check your answer.)

**Exercise 3.** If we change the first line from "`sheffer p q =`" to "`sheffer = \p -> \q -> `", does the type of `sheffer` change?



The definition of `sheffer` is verbose. Without switching to a radically different way of writing the function (the definition of `nand` is much shorter but doesn't resemble the truth table at all), can we make it a little shorter?



We can, using *gua* the bar as "such that" (as in the mathe al numbers greater than pi). We can equally well read it as "where".



So the first equation in `guardsheffer` can be read " here `p` is `True` and `q` is `True` is defined to be `False`". The second is read " re `p` is `True` and `q` is `False` is defined to be `True`", and so on.

Haskell tries each of the guards in order. As soon as a guard evaluates to `True`, Haskell switches to evaluating the right-hand side (to the right of the = sign).

```
guardsheffer p q
  | (p == True) && (q == True)   = False
  | (p == True) && (q == False)  = True
  | (p == False) && (q == True)  = True
  | (p == False) && (q == False) = True
```

Since the last three lines all return `True`, we can replace their guards with `otherwise`. As soon as Haskell sees `otherwise`, it uses that right-hand side (below, `True`).

```
-- Alternate alternate alternate way of writing nand,
-- using _guards_ with a "default" or "fall-through".
guardsheffer2 p q
  | (p == True) && (q == True)   = False
  | otherwise                    = True
```

**Exercise 4.** Without trying it, what should happen if we replace `otherwise` in `guardsheffer2` with `True`?

Check your answer by editing the file, then applying `guardsheffer2` to various arguments.