Types and strings

Patrick D. Elliott

April 18, 2023

- Next week we'll finish our initial survey of the very basics of Haskell, by talking more about types and typeclasses.
- In the following couple of weeks (May 2; May 9) we'll start implementing some basic natural language fragments in haskell, following the textbook *Computational Semantics with Functional Programming* (van Eijck, Jan and Unger, Christina, 2010).
    - Note: you should be able to access this textbook electronically via the HHU library.

- Do the *building functions* exercises at the end of chapter 3 (p83-85)
- Do chapter exercises 1-10 at the end of chapter 4.
- Optionally read chapter 3 and 4 of *Haskell from first principles* (skip 4.4, on numeric types).

Types and strings

Types in Haskell are a way of *categorizing values*; they provide a syntactic restriction on how complex expressions are built.

You might be familiar with types if you've ever taken a semantics course before.

- *is happy*: $\langle e, t\rangle$
- *Henning*: e
- *and* : $\langle t, \langle t, t\rangle\rangle$

Note that formal semantics often uses a different convention for functional types, but it's easy to translate between the two:

$$\langle a, b\rangle := a \to b$$

- Haskell has a more complex and powerful type-system than the one you might be used to from formal semantics.
    - Formal semantics typically uses the *simply-typed lambda calculus* as a basis.
    - Haskell is based on System F, i.e., the *polymorphic lambda calculus*, which allows for universal quantification over types.
    - Various *language extensions* exist to make Haskell's type system even more powerful (dependent types, linear types, etc).
    - In this course, we won't go beyond anything expressible in the polymorphic lambda calculus.

You can find out the type of any haskell expression quite easily using the `:type` command in GHCi:

```
ghci> :t "hello haskell!"
"hello haskell!" :: String
ghci> :t 'a'
'a' :: Char
```

- Note that single characters are enclosed in single quotes.
- The double colon `::` is interpreted as *has the type*.

We explicitly annotate expressons with their type using `::`.

```
ghci> :t ("hello haskell!" :: String)
"hello haskell!" :: String
```

If we annotate an expression with the wrong type, we'll get an error:

```
ghci> :t ("hello haskell!" :: Char)
<interactive>:1:2: error:
    • Couldn't match type ‘[Char]’ with ‘Char’
      Expected: Char
        Actual: String
```

`String` is actually a name for a *complex type*, `[Char]`.

That is to say, strings in haskell are actually just *lists of characters*.

In general, for any type `a`, the type `[a]` is the type of a list of things of type `a`.

We can print strings to the standard output in GHCi using the `putStrLn` or `putStr` functions.

```
ghci> putStrLn "hello haskell!"
hello haskell!
```

Examine the type of `putStrLn`. You'll notice something quite interesting.

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
```

In Haskell, we use arrow notation for function types (we'll come back to this later). `IO ()` is a special type to indicate that the program has some effect beyond evaluation of functions and arguments.

Printing strings from a source file

```haskell
  -- print1.hs

module Print1 where

main :: IO ()
main = putStrLn "hello world!"
```

If we load `print1.hs` from GHCi and execute `main`, *hello world!* will be printed to the standard output.

The `main` function

In haskell `main` is the default action when building an executable, or running it in GHCi, and it must always be of type `IO ()`.

```
putStrLn :: String -> IO ()
```

Input/output is much more complicated in Haskell than in most other programming languages, since it involves exploiting Haskell's type system to reason about *side effects*. This will be a topic for later in the semester.

Concatenating strings

There are two functions for concatenating strings in the haskell prelude:

```
(++) :: [a] -> [a] -> [a]
concat :: [[a]] -> [a]
```

- `++` is an infix operator, whereas `concat` is just an ordinary function.
- Note that `a` in the type signature is a *type variable*. Free variables in type signatures are implicitly universally quantified in Haskell.
- This means that both `++` and `concat` are *polymorphic* functions; they can be used to combine lists more generally.

In formal semantics, functional types are often written using angled-brackets (e.g., $\langle e, t \rangle$), following the convention used by (Heim, Irene and Kratzer, Angelika, 1998).

Haskell uses arrow notation, which is more commonly found in the computer science/programming language literature, although some semantics texts use arrow notation (Carpenter, Bob, 1998).

Arrow notation in Haskell is *right associative*:

- `a -> b -> c` $\Longleftrightarrow$ `a -> (b -> c)`

Types primer ii

Let's look again at the type for list concatenation:

```
(++) :: [a] -> [a] -> [a]
```

- `(->)` is a type *constructor*. It takes two types `a`, `b` and returns the type of a function from as to bs.
- One important feature of haskell is the possibility of defining arbitrary constructors; `([.])` takes a type `a` and returns the type of a list of as.
- Remember, free type variables are implicitly universally quanitified, which means that list concatenation is defined for something of type `[a]`, where `a` can be *any type*.

Strings as lists of chars

```
"hello haskell!"
['h','e','l','l','o',' ','h','a','s','k','e','l','l','!']
```

- Strings surrounded by double quotes are really just *syntactic sugar* for lists of characters.
- Syntactic sugar is just a notational convention built into the language that makes our lives as programmers easier.
- Lists are actually also syntactic sugar! We'll learn what lists really are in a bit.

What do you think the following evaluates to?

```
[1,2,3] ++ [4,5,6]
```

What happens if we try to evaluate the following:

```
"hello" ++ [4,5,6]
```

```
ghci> head "Henning"
'H'
ghci> tail "Henning"
"enning"
ghci> take 0 "Henning"
""
ghci> take 3 "Henning"
"Hen"
ghci> drop 3 "Henning"
"ning"
ghci> "Henning" !! 2
'n'
```

What happens when you run the following in GHCi:

```
ghci> "yo" !! 2
```

Let's examine the type of `!!`; as expected, its a function from a list of as, to an integer, to an a.

```
(!!) :: [a] -> Int -> a
```

Note however, that this isn't a *total* function; there are some lists and integers for which this function will be undefined.

Partial functions in haskell are considered *unsafe*, because the type system doesn't prevent us from providing an illicit value as an argument to the function.

In Haskell, it's good practice to avoid unsafe functions wherever possible.

This is because the type-checker is an extremely powerful programming aid - if a program type-checks successfully, we can generally be reasonably sure that it will run without any errors and give back a sensible result.

This promise only holds just so long as we use *total* functions. There is some neat type-level machinery in haskell to rewrite functions like `(!!)` as total functions which we'll learn about later in the semester!

Building lists with `cons`

The final list manipulation function we'll look at is an important one: `cons`.

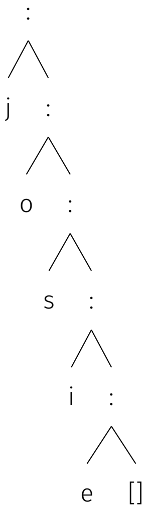
```
ghci> 'h' : []
[h]
ghci> 'h' : "enning"
"henning"
```

In haskell, lists are built up by successive application of `cons`:

```
'h' : ('e' : ('n' : ('n' : ('i' : ('n' : ('g' : []))))))
```

Since `:` is right associative we can drop the parentheses.

Lists in haskell are therefore *singly-linked lists of characters*.


:
j
:
o
:
s
:
i
:
e
[]

- For most industrial applications, singly-linked lists of chars would be a terrible choice.
- On the other hand, this means that strings ”come for free” on the basis of chars and extremely general list manipulation functions.
- For anything we do in this class, performance won’t be an issue. For serious work with strings, the standard is the Haskell `text` library.

Prolegomenon to types

- Types are syntactic categories used to restrict what counts as a valid expression.
- Basic ingredients:
    - A set of primitive types.
    - A recursive rule for constructing complex (i.e., functional) types.
    - Rules for computing the type of a complex expression from the types of its parts.

- Let's keep things simple, and start with just two primitive types:

$$\mathbf{Typ} := \{\mathtt{Int}, \mathtt{Bool}\}$$

- We'll assume that integers are possible values and have the type `Int`:

$$73 :: \mathtt{Int}$$

- We'll also assume two primitive values with the type `Bool`:

$$\mathbf{true} :: \mathtt{Bool}, \mathbf{false} :: \mathtt{Bool}$$

We'll now state a recursive rule for complex (functional) types, using the Haskell convention for types.

- If $\mathtt{a} \in \textbf{Typ}$, then $\mathtt{a}$ is a type.
- If $\mathtt{a}$ is a type, and $\mathtt{b}$ is a type, then $\mathtt{a} \to \mathtt{b}$ is a type.
- Nothing else is a type.

This means that we have many complex types like the following:

- $(\mathtt{Bool} \to \mathtt{Bool}) \to \mathtt{Int}$
- $\mathtt{Int} \to \mathtt{Int}$

- We can assign some useful operations their types:

$$(+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

$$(-) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

$$\textbf{factorial} :: \texttt{Int} \to \texttt{Int}$$

$$\textbf{odd} :: \texttt{Int} \to \texttt{Bool}$$

$$\textbf{even} :: \texttt{Int} \to \texttt{Bool}$$

$$\textbf{and} :: \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}$$

Functional applications: Let $\beta :: \mathsf{a} \to \mathsf{b}$, $\alpha :: \mathsf{a}$ be an expression of the SLTC. $\beta(\alpha)$ is an expression of type b.

Abstractions: Let $\beta :: \mathsf{b}$ be an expression of the SLTC, and υ a variable of type a. $\lambda\upsilon.\beta$ is an expression of type $\mathsf{a} \to \mathsf{b}$.

Can you infer the types of the following expressions? Go step by step.

$$\mathbf{and}(\mathbf{odd}(4))(t)$$

$$\lambda x.\mathbf{odd}(\mathbf{factorial}(x))$$

$$\lambda f.f(\lambda x.(+)(x)(2))$$

Often, you can *infer* the type of an expression without specifying the type of all of its sub-parts.

When you try to compile a haskell source file, or evaluate an expression in GHCi, the compiler will attempt to check that it is well-typed, by inferring the types of any expressions that don't have an explicit type provided.

Since haskell's type system is more expressive than we have here, the type-inference algorithm is quite complicated (the compiler is based on an algorithm called *Hindley-Milner*).

In a first order type-system, we can only state typed identity functions.

What is the type of *the* identity function?

$$\lambda x.x :: ?$$

Consider the following functions:

$$\mathbf{not} :: \texttt{Bool} \rightarrow \texttt{Bool}$$

$$\mathbf{not}' :: \lambda f.\lambda x.\mathbf{not}(f(x))$$

$$\mathbf{not}'' :: \lambda r.\lambda x.\lambda y.\mathbf{not}(r(x)(y))$$

- What are the types of not’ and not”?
- Is there a way of expressing all three functions as a single-operation? If not, why not?

Remember the expression ω:

$$(\lambda x.xx)(\lambda x.xx)$$

- Try to give it a concrete type.
- This problem is related to the lack of Turing completeness of the SLTC.
- On the other hand, because the SLTC is relatively constrained it has some extremely nice logical properties:
    - The SLTC is a sound and complete logic.
    - *Type-checking* (checking whether an expression is well-typed), and *type inference* are decidable.

Types in haskell

Some of the primitive types we've seen so far:

- `Int`
- `Char`
- `[Char]`
- `String`
- `Bool`

Data decalarations are declarations used for defining *types*.

We call the values that inhabit the type they are defined in data constructors.

The simplest kind of data declaration we see in Haskell is for a sum type. Consider the data declaration for `Bool`:

```
data Bool = False | True
```

The name immediately following the `data` keyword is the name of the type, which shows up in type signatures.

The *data constructors* follow the equals sign; sum types are declared by separating the constructors with `|`, which stands in for logical disjunction.

You can inspect the data declaration associated with a particular type by using the `:i` command in GHCi.

```
ghci> :i Bool
type Bool :: *
data Bool = False | True
-- ...
```

Depending on the version of ghc, this will also give you a bunch of extraneous information (the first line is the *kind signature*, and after the data decalaration we have information about *type classes* - we'll learn about these later).

It's easy to declare your own sum types in haskell. Consider the following:

```
data E = John | Mary | Bill | Sue
```

This declares a new type `E` whose inhabitants are all (and only) the values `John`, `Mary`, `Bill`, `Sue`.

We can define functions that take our new constructors as arguments by using *pattern matching*.

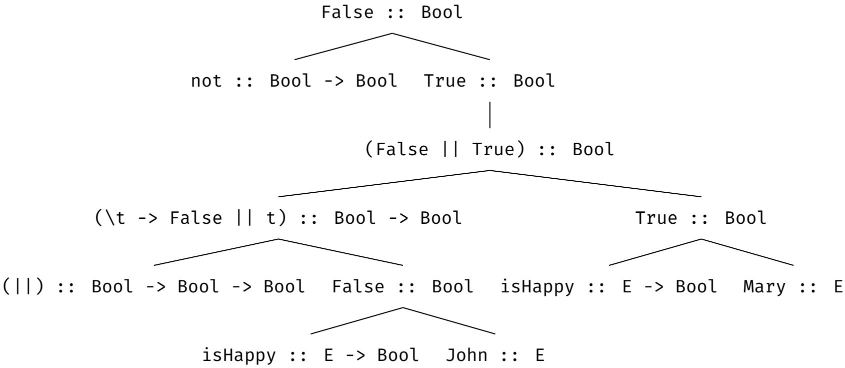
```
isHappy :: E -> Bool
isHappy Mary = True
isHappy _ = False
```

Note that the underscore is interpreted as an *elsewhere* condition.

What do you think the result of evaluating the following will be?

```
not (isHappy John || isHappy Mary)
```

Modelling composition: first steps



Recall our basic type for individuals.

```
data E = John | Mary | Bill | Sue
```

We haven't given ghc any further information about this type, so there's not much we can do with it. See what happens if you evaluate the following:

```
John == John
```

What about the following:

```
True == True
```

The reason for the contrast here is that `Bool` by default is an instance of the type class `Eq`, which is the class of types that contain things that can be compared and determined to be equal in value.

Since we didn't explicitly say that `E` is an instance of `Eq`, ghc doesn't assume that it is.

Likewise, try evaluating the following in ghci:

```
ghci> John
```

We'll learn later how to declare typeclass instances, but in the mean time ghc has convenient mechanisms for automatically generating sensible typeclass instances for simple types.

```
data E = John | Mary | Bill | Sue deriving (Eq,Show)
```

- Inspect the type of `id`.
- Now inspect the type of `(==)`, which is a function that tests for equality.
    - Polymorphism is used to constrain typeclasses.
    - The fewer typeclass constraints on a polymorphic type signature, the fewer assumptions the polymorphic function can make about its arguments.

What do you think will happen if you declare the in a source file?

```haskell
same :: Eq a => a -> b -> Bool
same a b = a == b
```

Remember that free type variables are *implicitly universally quantified*.

```haskell
id :: a -> a
```

Informally, this means that the type of `id` is `a -> a`, for all `a` in the set of types.

Type class constraints restrict the universal quantification to just types which belong to particular classes:

```haskell
(==) :: Eq => a -> a -> Bool
```

This means that the type of `(==)` is `a -> a -> Bool`, for all `a` that belong to the `Eq` class.

Combining typeclass restrictions

Typeclass restrictions can be combined. We've alluded to this before, but the typeclass `Show` is used to classify types whose inhabitants can be converted into strings (via the `show`) function.

What does the following function do?

```
func :: (Eq a, Show a) => a -> a -> String
func a b = if
  a == b
  then (show a) ++ " is equal to " ++ (show b)
  else "try again!"
```

Haskell has syntactic sugar for conditional states like *if A then B*, which are conventionally written as follows:

```
if _condition then _expressionA else _expressionB
```

You can use conditionals anywhere where you could use `_expressionA` or `_expressionB` (the expressions must be of the same type).

What does the following function do?

```
toyFunc n = if even n then n + 1 else n - 1
```

It's important to remember that anything that isn't function-argument application in haskell is *syntactic sugar*.

To illustrate, we could implement conditionals as a standard function:

```
cond :: Bool -> a -> a -> a
cond True a b = a
cond False a b = b

toyFunc2 n = cond (even n) (n + 1) (n - 1)
```

Tuples

Tuples are a ubiquitous syntactic construct, defined in haskell as a special kind of type known as a *product type*.

Let's look at the data declaration for tuples:

```
(,) a b = (,) a b
```

- This is quite different from what we've seen so far.
    - The datatype declaration involves a function (called a *type constructor*) that takes two type arguments `a`, `b`.
    - Type constructors create types from types.
    - For example, `(,) Int String` is a distinct type from `(,) String Int`.
    - `(a,b)` is *syntactic sugar* for `(,) a b`.

Working with tuples

Consider some tuples:

```
("haskell", "rocks")
("haskell", 1)
```

We can write functions `fst` and `snd` using pattern matching to extract the elements of a tuple (these are provided already in the prelude).

```
fst :: (a,b) -> a
fst (a,b) = a
snd :: (a,b) -> b
snd (a,b) = b
```

- Write a function `swap` that takes a tuple, and swaps the elements around.
- write a function `condTup` that takes a bool `t`, two tuples, `(a,b)`, `(c,d)`, and gives back a tuple of tuples `(a,c)` if `t` is true, and `(b,d)` otherwise (tip: think carefully about the type signature!).

Solution

```haskell
swap :: (a,b) -> (b,a)
swap (a,b) = (b,a)

condTup :: Bool -> (a,a) -> (b,b) -> (a,b)
condTup True (a,b) (c,d) = (a,c)
condTup False (a,b) (c,d) = (b,d)
```

$\mathcal{F}in$

References

Carpenter, Bob (1998). *Type-Logical Semantics*, MIT Press.

Heim, Irene and Kratzer, Angelika (1998). *Semantics in Generative Grammar*, Blackwell.

van Eijck, Jan and Unger, Christina (2010). *Computational Semantics with Functional Programming*, Cambridge University Press.