M

From types to typeclasses

Patrick D. Elliott

May 7, 2024

Homework

Next week we'll start implementing *linguistic* data structures using Haskell.

In preparation, you can read Chapter 4 of *Computational Semantics with Functional Programming* ”Formal syntax for fragments”.

Interlude: the indirect approach to semantics

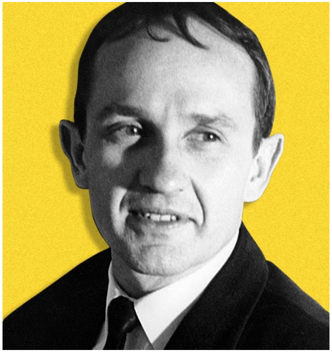
Rolling your own datatypes

In haskell, *sum types* can be used to model primitive types with fixed domains of entities:

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

In a sense, we're modelling a small fragment of English proper names "John", "Mary", "Bill", "Sue", interpreted as individual denoting constants.

Interlude: Montagovian fragments



”There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed I consider it possible to comprehend the syntax and semantics of both kinds of languages with a single natural and mathematically precise theory.” (Montague, 1970)

Compositional semantics: the indirect approach

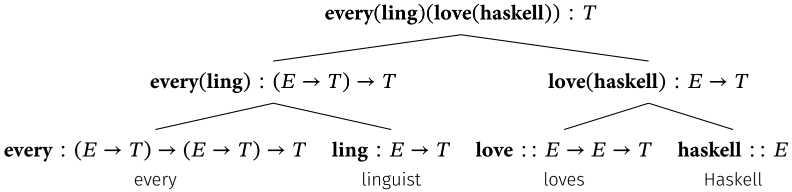
- Montague developed an influential technique for giving a denotational semantics for a *fragment* of a natural language, like English.
    - Montague's idea (inspired by the philosoher Gottlob Frege) was to first *translate* sentences of English into sentences of a logical language.
    - A denotational semantics could then be provided for the logical language quite straightforwardly.
    - This is called the *indirect approach* to compositional semantics.

The indirect approach cont.

Montague used *Intensional Logic* as the logical language, but more contemporary work in semantics typically uses the *Simply-Typed Lambda Calculus*, either implicitly or explicitly (Carpenter, 1998).

- every $\rightsquigarrow \lambda R.\lambda P.\mathbf{every}(R)(P) : (E \to T) \to T$
- and $\rightsquigarrow \lambda t.\lambda u.\mathbf{and}(t)(u) : T \to T \to T$
- Josie $\rightsquigarrow \mathbf{josie} : E$

The indirect approach cont.


every(ling)(love(haskell)) : T
every(ling) : $(E \to T) \to T$
love(haskell) : $E \to T$
every : $(E \to T) \to (E \to T) \to T$
ling : $E \to T$
love :: $E \to E \to T$
haskell :: E
every
linguist
loves
Haskell

- The truth-conditions of the sentence are provided by the denotational semantics of the logical language.

$$[\![\mathbf{ling}]\!] = \{\, (x, \mathbf{True}) \mid x \text{ is a linguist} \,\} \cup \{\, (x, \mathbf{False}) \mid x \text{ isn't a linguist} \,\}$$

The Haskell connection

There is no important theoretical difference between natural languages and the artificial languages of programmers.

- A modern (re-)rendering of Montague's conjecture:
  - English sentences can be systematically translated into *haskell expressions*.
  - Determining whether or not the sentence is true amounts to *normalization*.

```
and(even(4))(odd(2))
-- False
```

Typeclasses

Doing things with datatypes

Recall our basic type for individuals.

```haskell
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:

```haskell
John == John
```

What about the following:

```haskell
True == True
```

Basic typeclasses

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.

```
type Bool :: *
data Bool = False | True
          -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
```

Since we didn’t explicitly say that `E` is an instance of `Eq`, ghc doesn’t assume that it is (you can try typing in `:i E` into ghci to verify this).

Basic typeclasses cont.

Likewise, try evaluating the following in ghci. What do you think is responsible for the different results?

```
ghci> John
```

```
ghci> True
```

Deriving typeclasses

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

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

- By default, an `Eq` instance for a sum type considers data constructors to be identical only to themselves.
- The default `Show` instance for a sum type simply converts the name of the data constructor into a printable string.

Constrained polymorphism

- 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 more typeclass constraints we add to a polymorphic type signature, the more assumptions the polymorphic function can make about its arguments.

Typeclass syntax

- Typeclass constraints are applied to types using the `=>` syntax at the beginning of the type signature.
- Multiple typeclass constraints are separated by commas in parentheses.
- Typeclass constraints are interpreted *conjunctively*, e.g.,

```
f :: (Class1 a, Class2 a) => a -> a
```

This is interpeted as a *universal quantification* over types, where the *restriction* of the universal is provided by the typeclasses:

$$\forall t \in \mathbf{Typ}[(t \in \mathbf{Class1} \wedge t \in \mathbf{Class2}) \rightarrow f : (t \rightarrow t)]$$

Typeclass syntax cont.

- `=>` is a binding operator; it comes with a variable, and binds all matching variables in its scope.
- Typeclass constraints with different variables restrict different universal quantifiers.

```haskell
f2 :: (Class1 a, Class2 b) => a -> b
```

$$\forall t, t' \in \mathbf{Type}[(t \in \mathbf{Class1} \wedge t' \in \mathbf{Class2}) \to f_2 : (t \to t')]$$

Using typeclasses

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
```

What about the following?

```haskell
same2 :: (Eq a, Eq b) => a -> b -> Bool
same a b = a == b
```

Using typeclasses cont.

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

```
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:

```
(==) :: 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?

```haskell
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!"
```

Why do we need both typeclass constraints here?

Interlude: conditionals and tuples

Conditionals

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

```haskell
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?

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

Conditionals and syntactic sugar

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

As an exercise, implement conditional statements as a standard function:

```
cond :: Bool -> a -> a -> a
```

Test your answer by rewriting `toyFunc` using `cond`.

Lists and tuples

We learned earlier about *lists* in haskell, of type `[a]`, for any type `a`.

```haskell
myList1 :: [Int]
myList1 = [2,4,6,8]

myList2 :: [Char]
myList2 = "I'm a string"
```

The primary limitation of lists is that they can only contain *elements of the same type*.

A ubiquitous data structure in haskell used for elements of (potentially) distinct types is the *tuple*.

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
("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).

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

N-tuples

Unlike lists, tuples have a *fixed number* of elements.

```
("Haskell", 1, "Rocks") :: (String, Int, String)
('a', 'b', "Hello", 73) :: (Char, Char, String, Int)
```

The `fst` and `snd` functions won't work for *n-tuples*, where $n > 2$; why not?

Tuples under the hood

Unlike lists, tuples in haskell aren't singly-linked. To see this, try evaluating the following:

```
ghci> (1,2,3) == ((1,2),3)
ghci> (1,2,3) == (1,(2,3))
```

In fact, a 2-tuple involves a distinct constructor to a 3-tuple.

```
ghci> (,,) 1 2 3
(1,2,3)
ghci> (,,,) 1 2 3 4
(1,2,3,4)
ghci> (,,) 1 2 3 4
-- type mismatch error
```

This explains why `fst` and `snd` don't work!

Exercise

- 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!).
- Write functions `fst5` and `snd5` that apply to 5-tuples. Is it possible to write an *unsafe* index function for tuples?

Tuples and currying

- Functions in Haskell strictly take one argument and return one result; sometimes that result is itself a function.
- When a function appears to take multiple arguments, in fact those arguments are *curried*, i.e., addition has the following type signature:

```
(+) :: Num a => a -> a -> a
```

Currying means that we can pass around the result of *partially applying* a function that takes multiple arguments.

```
ghci> myPartial = (+) 4
ghci> myPartial 6
10
```

Exercise: uncurrying

Exercise: write a function `myAddition` that takes a *tuple* as its sole argument.

Exercise: generalized (un)currying

This exercise is a bit harder:

- Part 1: write a function `myUncurry` of type `(a -> b -> c) -> (a,b) -> c`.
- Part 2: write a function `myCurry` of type `((a,b) -> c) -> a -> b -> c`.
- Part 3: now do the same thing, but for functions which take 3 arguments.
    - Is it possible to write a generalized function `myCurryN` that curries a function that takes `n` arguments?

Polymorphism

Parameteric polymorphism

Type signatures in haskell can be (parametrically) polymorphic. Recall that typeclasses constrain what we can do with arguments to a polymorphic function.

- Try to write a function of type `a -> a` that does soemthing other than return the input value.
- There are two possible implementations of the function with type signature `a -> a -> a`. Write them both.
- How many implementations can can `a -> b -> b` have?

Function composition

Combinators (remember those from the lambda calculus?) in haskell are polymorphic functions.

Function composition is an infix operator `f . g`.

Here's one way of writing its definition:

```
f . g = \x -> f $ g x
```

This will be useful in the following exercises.

Type-kwon-do

- In all of the following cases, the goal is to make the program pass the type checker by modifying the `???` declaration, and it alone.

Type-kwon-do 1

```haskell
f :: Int -> String
f = undefined

g :: String -> Char
g = undefined

h :: Int -> Char
h = ???
```

Type-kwon-do 2

```haskell
data A
data B
data C

q :: A -> B
q = undefined

w :: B -> C
w = undefined

e :: A -> C
e = ???
```

Type-kwon-do 3

```haskell
data X
data Y
data Z

xz :: X -> Z
xz = undefined

yz :: Y -> Z
yz = undefined

xform :: (X, Y) -> (Z, Z)
xform = ???
```

Type-kwon-do 4

```
munge :: (x -> y)
  -> (y -> (w, z))
  -> x
  -> w
munge = ???
```

Fin

References