Currying

Jack Kelly
http://jackkelly.name/talks

January 21, 2020

Back in 2006...

- I'd just started 2^{nd} year uni thinking I knew what programming was...
  - ...and slammed straight into Haskell

One-arg function

- Here's how you write a function:

```
f :: Int -> Int
f x = x * 2
```

- Okay, fair enough.

Two-arg function

- And here's how you write a function of two arguments:

```
f :: Int -> Int -> Int
f x y = x + y
```

- Wait, what?

What's up with the arrows?

- 2006!Jack: "This looks silly! Functions should have one argument and one arrow."
- Today!Jack: "Joke's on you, kiddo. Functions take only one argument. That's what the arrow *means*!"

In Haskell and Elm, functions are curried. It works like this:

- All functions take one argument.
  - Functions with "multiple" arguments actually return other functions
- `->` in a type associates to the *right*:
  - `a -> b -> c -> d` means `a -> (b -> (c -> d))`
- Function application associates to the *left*:
  - `f x y z` means `((f x) y) z`

Tonight

- Add/remove redundant parens to get new perspectives
- Practice shifting between these perspectives
- Some implications of currying in library design
- Examples in Elm where possible, Haskell where necessary

List.map

```
-- Given a function and a list, apply that
-- function to each element of the list
List.map : (a -> b) -> List a -> List b


-- Lift a function on elements to a function on
-- lists (Function transformer!)
List.map : (a -> b) -> (List a -> List b)
```

Dict.remove

```
-- Given a key and a Dict, return that Dict
-- minus the entry at key
Dict.remove
  : comparable -> Dict comparable v
  -> Dict comparable v


-- Given a key, return a function
-- which subtracts it from a Dict
Dict.remove
  : comparable
  -> (Dict comparable v -> Dict comparable v)
```

Dict.insert

```
-- Given a key, value, and Dict, return the Dict
-- plus an entry associating the key and value.
Dict.insert
  : comparable -> v -> Dict comparable v
  -> Dict comparable v
```

```
-- Given a key and a value, return a function
-- which adds that association to a Dict
Dict.insert
  : comparable -> v
  -> (Dict comparable v -> Dict comparable v)
```

flip

```
-- Swap the first two arguments of a function.
-- (Why "first two"? c could be a function!)
flip : (a -> b -> c) -> (b -> a -> c)
```

```
-- Supply the second argument to a function
flip : (a -> b -> c) -> b -> (a -> c)
```

(«)

```
-- Haskell calls this (.)
(<<) : (b -> c) -> (a -> b) -> (a -> c)


-- Apply a function "under"
-- the first argument of another
(<<) : (b -> c) -> (a -> b) -> a -> c
```

(«) — What if c was a function?

- Remember that type variables can stand for anything, including other functions:
- Borrowed notation: (~) is the operator for "type equality" in Haskell

```
-- c ~ (d -> e)
(<<) : (b -> c) -> (a -> b) -> (a -> c)


-- Stick a function "in front of"
-- the first argument
(<<) : (b -> d -> e) -> (a -> b)
     -> a -> d -> e
```

liftA2

```
-- Combine the "f of a" and "f of b",
-- according to the given function
liftA2
  :: Applicative f
  => (a -> b -> c) -> f a -> f b -> f c


-- Lift a binary function "over f"
-- (Function transformer!)
liftA2
  :: Applicative f
  => (a -> b -> c) -> (f a -> f b -> f c)
```

(<*>)

```
-- Apply the "f of a" to the "f of function"
(<*>)
  :: Applicative f => f (a -> b) -> f a -> f b


-- Distribute f over ->
(<*>)
  :: Applicative f => f (a -> b) -> (f a -> f b)
```

Lens — view

```
-- Given a lens and the structure being
-- zoomed into, return the thing the
-- lens "looks at"
view :: Lens' s a -> s -> a


-- Turn a lens into a getter function
view
  :: Lens' s a
  -> (s -> a)
```

Lens — set

```
-- Given a lens, a new valur for a part
-- and the structure being zoomed into,
-- update the thing the lens "looks at"
set :: Lens' s a -> a -> s -> s


-- Turn a lens into a setter function
set
  :: Lens' s a
  -> (a -> s -> s)

-- Turn a lens and a new value
-- into an update function
set
  :: Lens' s a -> a
  -> (s -> s)
```

Lens — over

```
-- Given a lens and "update function"
-- on the part, update the whole
over :: Lens' s a -> (a -> a) -> s -> s


-- Given a lens,
-- lift a function on the part
-- into a function on the whole
over
  :: Lens' s a
  -> (a -> a)
  -> (s -> s)
```

traverse

```
-- Map elements of a structure to actions,
-- evaluate them left to right,
-- and collect the results.
traverse
  :: (Applicative f, Traversable t)
  => (a -> f b) -> t a -> f (t b)


-- Lift a function on items that returns an
-- action, to a function over traversable
-- structures (Function transformer!)
traverse
  :: (Applicative f, Traversable t)
  => (a -> f b) -> (t a -> f (t b))
```

Using the "Function Transformer" perspective (1)

```
-- We want to transform `xss`,
-- so use a hole for the function to construct
doubleMap :: (a -> b) -> [[a]] -> [[b]]
doubleMap f xss = _ xss

-- `map` lifts a function
-- `[a] -> [b]` to `[[a]] -> [[b]]`
-- Now we want to create a function
-- `[a] -> [b]` in the gap.
-- Use `map` again.
doubleMap f xss = (map _) xss

-- Now `f` will fit the hole
doubleMap f xss = (map (map _)) xss
doubleMap f xss = (map (map f)) xss
```

Using the "Function Transformer" perspective (2)

```
-- Can we simplify this?
doubleMap f xss = (map (map f)) xss

-- Eta reduce
doubleMap f = map (map f)

-- Definition of (.)
doubleMap f = (map . map) f

-- Eta reduce
doubleMap = map . map
```

What just happened?

```haskell
-- Work through the type variables on paper
-- to understand why
map . map :: (s -> t) -> ([[s]] -> [[t]])

-- Hint: apply ‘map‘ twice to ‘(.)‘, which
-- has the following type:
(.)
  :: (b -> c)
  -> (a -> b)
  -> (a -> c)
```

Where else does this work?

A lot of these "function transformers" compose nicely:

- `fmap . fmap`
  `:: (Functor f1, Functor f2)`
  `=> (a -> b) -> f1 (f2 a) -> f1 (f2 b)`
- `liftA2 . liftA2`
  `:: (Applicative f1, Applicative f2)`
  `=> (a -> b -> c)`
  `-> f1 (f2 a) -> f1 (f2 b) -> f1 (f2 c)`
- `foldMap . foldMap`
  `:: (Foldable t1, Foldable t2, Monoid m)`
  `=> (a -> m) -> t1 (t2 a) -> m`
- `traverse . traverse`
  `:: (Traversable t1, Traversable t2, Applicative f)`
  `-> (a -> f b) -> t1 (t2 a) -> f (t1 (t2 b))`

Why does this work so well?

- Partial application makes argument order really important
- Good API design $\implies$ good argument order
- "The data structure is the final argument"
  - Folklore in Haskell, explicit design rule in Elm
  - `https://package.elm-lang.org/help/design-guidelines#the-data-structure-is-always-the-last-argument`

Takeaways

When you get home:

- Paste your favourite functions into a text editor
- Add and remove "redundant" parens from the type signatures
- See familiar functions in a new light

Some suggestions:

- `always : a -> b -> a` (Haskell calls this `const`)
- `curry :: ((a, b) -> c) -> a -> b -> c`
- `uncurry :: (a -> b -> c) -> (a, b) -> c`
  - Also check out `traverse . uncurry`