Everything Looks Like a Function

Jack Kelly
http://jackkelly.name/talks

February 21, 2023

Tonight's Goals

- Notice that data often directly represents functions
- Practice shifting perspective between data $\leftrightarrow$ functions
- Explore *representable functors* —
  a specific way of converting between functions and data.

What do we do with data?

- Why do we have data at all?
- What does this data declaration represent?

Match data type

```
data Match a = Anything | This a

match :: Eq a => Match a -> a -> Bool
match Anything _ = True
match (This a) a' = a == a'
```

What do we do with data?

- Why do we have data at all?
- What does this data declaration represent?

`Match` data type

```
data Match a = Anything | This a

match :: Eq a => Match a -> a -> Bool
match Anything _ = True
match (This a) a' = a == a'
```

- Type `Match a` represents a subset of functions `a -> Bool`
- `match :: Eq a => Match a -> (a -> Bool)` could be seen as an interpreter for `Match a` — it turns the data structure into the predicate it represents.

Functions as Data

- Why store the data? Why not store the function itself?
- The type `Region` is good for exactly one thing

Region Handling

```
type Region = Point -> Bool

circle :: Radius -> Region
outside :: Region -> Region
(/\) :: Region -> Region -> Region

annulus :: Radius -> Radius -> Region
annulus r1 r2 =
  outside (circle r1) /\ circle r2
```

- From: Haskell vs. Ada vs. C++ vs. Awk vs. ...
  An Experiment in Software Prototyping Productivity
- See also: John Hughes revisiting "why functional programming matters"
  Lambda Jam 2017: `https://www.youtube.com/watch?v=vGVJYoKIzjU`

Syntax Trees

Language

```
data Expr
  = Lit Int
  | Var Text
  | Add Expr Expr

-- (x + 2) + y
sample :: Expr
sample =
  Add
    (Add (Var "x") (Lit 2))
    (Var "y")
```

Operations

```
eval :: Map Text Int ->
  Expr -> Maybe Int

prettyPrint :: Expr -> Text

variables :: Expr -> Set Text
```

- We want to do more than just `evaluate` syntax trees!
- Parsing straight to functions subverts that goal

Functions vs. Data

Functions

- Composable
- Opaque
- Flexible
- Supports one operation: apply

Data

- Serialisable
- Inspectable
- Rigid
- Supports many operations

- Looks like the expression problem!
- See also: "initial" vs. "final" encodings of data
  - `https://peddie.github.io/encodings/encodings-text.html`

Example: Amazonka Path Prefixes

- AWS API Gateway is a "serverless" product from AWS, which routes requests from one endpoint to different backend services.
- To manage WebSocket connections on an AWS API Gateway, you need to make AWS API calls to its deployed domain name and path prefix.
- Amazonka can override services to support custom domain names, but not path prefixes. I was asked to add support for this.
- The only thing we ever do with a `pathPrefix` is prepend it to the request `path`. Should we instead store `pathHook :: Path -> Path`, for more flexibility?

Example: Functions and Maps

- The primary operation on a `Map k v` is
  `lookup :: Ord k => Map k v -> (k -> Maybe v)`
- We could say that representing this function `k -> Maybe v` is the whole reason we have a `Map k v`
- But `Map k v` is much more easily serialisable and incrementally editable
- If you have a function `f :: x -> y` and `x` is finitely enumerable, you can serialise the functions by writing every `(x, f x)` pair into a map or association list.
- Interestingly, functions are *contravariant* in `x` but the map is *covariant* in `x`.

Converting Between Functions and Data

Representable Functors

```
-- From package ‘adjunctions‘, simplified
class Functor f => Representable f where
  type Rep f :: Type
  index :: f a -> (Rep f -> a)
  tabulate :: (Rep f -> a) -> f a
```

- Isomorphism between `f` and `(->) (Rep f)`
- Methods convert `f a` $\Leftrightarrow$ `Rep f -> a`
- `f` can implement instances from the reader functor for free

Representable Functors — Example (1)

- How can we fetch every parameter we care about on startup?
- How do we know we have them all?

Parameter Service

```
fetchParameters ::
  [Text] -> IO (Map Text Text)
```

Representable Functors — Example (1)

- How can we fetch every parameter we care about on startup?
- How do we know we have them all?

Parameter Service

```
fetchParameters ::
  [Text] -> IO (Map Text Text)
```

Parameter Names

```
data Parameter =
  Foo | Bar | Baz
```

Parameter Records

```
data Parameters a = Parameters
  { foo :: a
  , bar :: a
  , baz :: a
  }
  deriving (Functor,
     Foldable, Traversable)

names :: Parameters Text
names =
  Parameters "a" "b" "c"
```

Representable Functors — Example (2)

Representable instance

```
instance Representable Parameters where
  type Rep Parameters = Parameter

  index :: Parameters a -> Parameter -> a
  index Parameters{..} p = case p of
    Foo -> foo
    Bar -> bar
    Baz -> baz

  tabulate :: (Parameter -> a) -> Parameters a
  tabulate f = Parameters
    { foo = f Foo, bar = f Bar, baz = f Baz }
```

Representable Functors — Example (3)

Fetching all Parameters

```
fetchParametersByName ::
  Parameters Text -> IO (Maybe (Parameters Text))
fetchParametersByName names = do
  values <- fetchParameters (toList names)

  let structure :: Params (Maybe Text)
      structure = tabulate $ \p ->
        Map.lookup (index names p) values

  -- sequence :: Parameters (Maybe a) -> Maybe (Parameters a)
  pure (sequence structure)
```

Rank-2 Representables

Higher-Kinded Data

```
data Parameters f = Parameters
  { foo :: f Int
  , bar :: f Text
  , baz :: f Bool
  }
```

- Representable functors generalise to heterogeneous data
- Could fetch all parameters *and parse* them to correct types

```
fetchParametersByName ::
  Parameters (Const Text) ->
  IO (Maybe (Parameter Identity))
```

What's Next

- What have we learned?
  - It's worth thinking about the functions your data could represent
  - It's worth thinking about the data your functions could represent
  - Representable functors seem doable in many languages
- Where else can we go?
  - Functions aren't as opaque as you might think (Compiling to Categories)
  - Functions can be a good model of how things should behave (Denotational Design)
  - If you can convert from a function $\Rightarrow$ data $\Rightarrow$ function, you can send entire functions to other languages/machines/... in a disciplined way