Higher-Order Data

FUNCTION

ALL THE THINGS
Functions are data

Already seen:

• can pass functions as input
• can return functions as output
• can write function literals

Can also:

• store functions in data types!

```haskell
data Expr = Lit Int
          | Ref Var
          | BinOp (Int -> Int -> Int) Expr Expr
```

(map, filter, foldr, (.), partial application, \( f \rightarrow f \cdot f \))
Dictionary pattern

Basic idea:
• data type describes an interface
• values represent instances

Do “pure object-oriented programming” in Haskell!

(MonoidDict.hs, SetOO.hs)
Combinator pattern

Basic idea:

• **type** defines the **semantic domain** of a language
• **values** represent **semantic objects**
• **functions** define the **syntax**
  
  "shallow embedding"

A very successful pattern in Haskell!

(Env.hs, Parser.hs)
Higher-order abstract syntax pattern

A technique for implementing name bindings in ASTs

Basic idea:
• use host language (Haskell) bindings to encode object language bindings

A “shallow” feature in an otherwise “deep embedding”
Higher-order data tradeoffs

Advantages:
• easy to extend with new cases (e.g. operations, instances)
• reuse features of host language (e.g. bindings, type system)
• tends to lead to general, powerful designs (compositionality!)

Disadvantages:
• difficult to extend with new interpretations (e.g. semantics + pretty printing)
• cannot analyze internal structure of functions (e.g. get all variable names in HOAS)
• difficult to “override” features of the host language