A Survey of Data Types

Neil McGhani

and Peter Hancock, Conor McBride, Thorsten Altenkirch ...
MSP group, University of Strathclyde, The Scottish Free State

Overview

- **Question:** What makes a good programming language
  - **Answer 1:** The ability to write space invaders
  - **Answer 2:** A clean interface between programmer/problem
- **Key Idea:** Abstraction!
  - Humans are good at abstraction - FP allows us to be abstract
  - FP (Haskell) is taking over the world!
- **My Research:** The design of high level programming languages
  - ... based upon mathematical structure.
  - Tools: logic, type theory, category theory

Inductive Data Types:

- **Question:** What would a data type consist of?
  - A mechanism for defining/constructing elements of the type
  - And a mechanism for using/consuming elements of the type
- **Answer:** Call the mechanism for defining natural numbers *constructors* and write them down

```
data Nat where
     Zero :: Nat
     Succ :: Nat -> Nat
```

- **Examples:** Thus we have `Zero`, `Succ Zero`, `Succ (Succ Zero)`
  - `Nat` is inductive, ie the *smallest* type which is closed under having a constant and a unary operation. `Nat = 1 + Nat`

- **Question:** How do we consume data?
  - Recursor replaces constructors with other functions

```
rec :: a -> (a -> a) -> Nat -> a
rec z s Zero = z
rec z s (Succ n) = s (rec z s n)
```

- **Examples:** `rec` encodes common prog. patterns/code reuse

```
plus :: Nat -> Nat -> Nat
plus n m = rec m (+n) n
```

- **Extensionality:** `rec` is the unique such function

```
h Zero = z and h (Succ n) = s (h n) => h = rec z s
```

A Slightly More Sophisticated Example

- **Question:** What about constructing lists
  - A list is either the empty list
  - Or, it is an element and a list
- **Answer:** Thus we have the Haskell definition

```
data List a  where
      Nil  :: List a
      Cons :: a -> List a -> List a
```

  - So `List a` is the smallest set closed under these operations, ie least solution of `Z = 1 + a x Z`
- **Polymorphism:** Notice that we make lists of any type - code reuse

Recursion Combinators for Lists?

- **Recursion:** Replacement functions for constructors
  - Our constructors had types

```
Nil  :: List a          Cons :: a -> List a -> List a
```

- **Answer:** Use a recursion combinator + extensionality rule

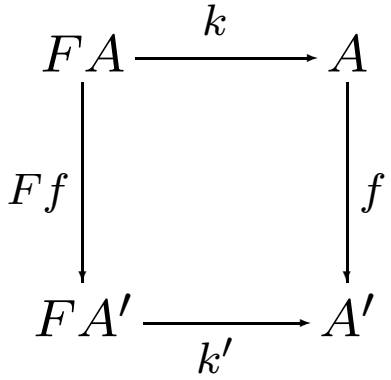
```
rec :: b -> (a -> b -> b) -> List a -> b
rec n c Nil          = n
rec n c (Cons a as) = c a (rec n c as)
```

- **Examples:** Can you express the following via `rec`?
  - `sum ::  List Int -> Int`

What's going on here?

- **Question:** How do we *predict* the rules for `Nat` and `List a`
  - Answer should *predict* the rules for advanced data types
- **Wrong:** Not as syntax trees ... too operational/syntax chasing!
  - Perefer mathematical/semantic structure
- **Answer:** Abstract constructors into an operation on types
  - For `Nat`, $FX = 1 + X$
  - For `List a`, $FX = 1 + a \times X$
  - F describes how new elements are built from old. F generates the data type which is the least X such that $FX \cong X$

- **Key Idea:** `Zero` and `Nat` form a function $F\ Nat \to Nat$
  - Define an F-algebra to be a type A and a function $k : FA \to A$
  - Define an F-algebra morphism from (A, k) and (A', k') to be a program $f : A \to A'$ such that

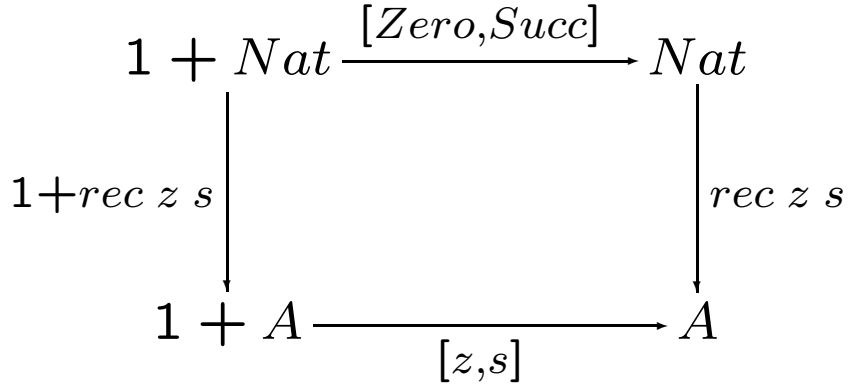


  F must be a Haskell functor, ie have an `fmap` operation

```
fmapF :: (a -> b) -> F a -> F b
```

- **Key Idea:** Everything you need to know to compute and reason about `Nat` can be derived from the following single fact:
  - `Nat` carries the structure of a F-algebra (when $FX = 1 + X$) and there is exactly one F-algebra morphism from this F-algebra to any other F-algebra.
  - Or, `Nat` is the initial F-algebra.

- **Amazing:** Everything captured in the following picture



- **Key Idea:** `List a` carries the structure of a F-algebra (when $FX = 1 + a \times X$) and there is exactly one F-algebra morphism from this F-algebra to any other F-algebra.

- **In a Nutshell:** `List a` is the initial F-algebra.

- **Amazing 2:** Everything captured in the following picture

$$\begin{array}{ccc} 1 + a \times List\ a & \xrightarrow{[Nil,Cons]} & List\ a \\ \Big\downarrow{\scriptstyle 1+rec\ n\ c} & & \Big\downarrow{\scriptstyle rec\ n\ c} \\ 1 + a \times A & \xrightarrow[[n,c]]{} & A \end{array}$$

- **Abstractly:** Every inductive data type arises in this way

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{in} & \mu F \\ {\scriptstyle F\ k}\downarrow & & \downarrow{\scriptstyle rec\ k} \\ FA & \xrightarrow[k]{} & A \end{array}$$

- **Reflection:** Mathematical structure gives generic programming

```
class Functor f where
  fmap : (a -> b) -> f a -> f b

data Mu f = In (f (Mu f))

rec :: (f a -> a) -> Mu f -> a
rec k (In t) = k (fmap (rec k) t)
```

Inductive Families

- **Examples:** Here are two data types
  - Balanced Binary Trees are a type with a constraint

```
data BTree a where
      BLeaf :: a -> BTree a
      BNode :: BTree (a,a) -> BTree a
```

  - Lambda Terms contain variable binding

```
data Lam :: Nat -> Set where
      Var :: a -> Lam a
      App :: Lam a -> Lam a -> Lam a
      Abs :: Lam (a + 1) -> Lam a
```

- **Examples:** As before, constructors make elements of the types

```
BNode (BLeaf (2,4)) :: BTree Int
```

- **Question:** Are these inductive types
  - Let's hope so since then we can have generic, structured programming

- **Answer:** These are NOT inductive types
  - Compare the equations

```
List a = 1 + a x (List a)                    Z = 1 + a x Z
BTree a = a + BTree (a,a)                    Z = a + ???
Lam a = a + (Lam a) x (Lam a) + Lam (a+1)    Z = a + Z x Z + ???
```

- **Key Idea:** These types are indexed and indexes change

- **Families of Inductive Types:** Two facts about `List`
  - `List :: * -> *` is a family of types, one for each input type
  - `List a` is inductive - `List a` does not depend upon `List b` for `a /= b`
- **Inductive Families:** The second property does not generalise
  - `BTree :: * -> *` is a family
  - `BTree a` depends upon `BTree (a,a)`
- **Solution:** Define the whole family `BTree` inductively

```
BTree  = Id + BTree . (-,-)
```

Just use IAS to predict behaviour of `BTree`

- **Key Idea:** Since `BTree :: * -> *` we must
  - replace type with type-indexed types
  - replace programs with polymorphic programs

- **Define:** `F :: (* -> *) -> * -> *` by

$$F\ B\ X = Id + B.(-,-)$$

and then predict `BTree` by the picture

$$\begin{array}{ccc} Id + BTree.(-,-) & \xrightarrow{[BLeaf,BNode]} & BTree \\ \Big\downarrow{\scriptstyle Id+\ rec\ bl\ bn.(-,-)} & & \Big\downarrow{\scriptstyle rec\ bl\ bn} \\ Id + X.(-,-) & \xrightarrow[[bl,bn]]{} & X \end{array}$$

- **Results:** We get ...

```
data BTree a where
   BLeaf :: a -> BTree a
   BNode :: BTree (a,a) -> BTree a

rec :: (All a . a -> X a) -> (All a . X (a, a) -> X a) ->
       (All a . BTree a -> X a)
rec bl bn (BLeaf x) = bl x
rec bl bn (BNode t) = bn (rec bl bn t)
```

- And we can derive other computational infrastructure for `BTree`
  - Church Encodings, Short Cut Fusion optimisations
  - Induction Rules, effectfull ariants. And GADTs

- **Question:** How do we understand `Lam : Nat -> Set` where

```
data Lam : Nat -> Set where
    Var : Fin n -> Lam n
    App : Lam n -> Lam n -> Lam n
    Abs : Lam (n+1) -> Lam n
```

- **Answer:** Use inital algebra semantics where
  - Types are replaced by Nat-indexed types
  - Programs are by Nat-indexed programs
  - See `Lam` as the initial algebra of the functor $F : (Nat \to \mathsf{Set}) \to Nat \to \mathsf{Set}$ defined by

$$F\ K\ n \ = \ Fin\ n \ + \ (K\ n) \times (K\ n) \ + \ K(n+1)$$

State of the Art: Induction Recursion

- **Definition:** A universe is a pair $(U : \mathsf{Set}, T : U \to \mathsf{Set})$
  - Think of U as codes for sets and T as a decoder
  - Fundamental structure in dependent type theory

- **Question:** Can we define a universe containing N and closed under Σ-types $\Sigma a : A.Ba$.

$$\begin{aligned} U &= 1 + \Sigma u{:}U.\, Tu \to U \\ T(in_1 *) &= N \\ T(in_2(u, f)) &= \Sigma a{:}Tu.\, T(fa) \end{aligned}$$

- **Key Observation:** We can't define U first and then T
  - Can't use IAS with U-indexed sets and functions

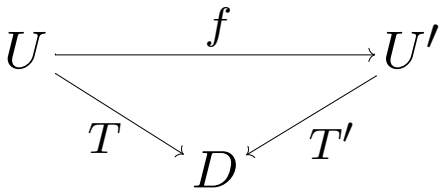
- **Key Idea:** If D is a type, so is IR D and there are three constructors
  - If $d : D$, then $\iota d : \mathsf{IR}\ D$
  - If $A : \mathsf{Set}$ and $f : A \to \mathsf{IR}\ D$, then $\sigma A f : \mathsf{IR}\ D$
  - If $A : \mathsf{Set}$ and $F : (A \to\ D) \to \mathsf{IR}\ D$, then

$$\delta A F : \mathsf{IR}\ D$$

- **Example:** Writing $+$ for $\sigma 2$, the code for a universe closed under Σ-types is. And for a universe c_Π closed under Π-types?

$$c_\Sigma = \quad \iota N + \delta(X : 1 \to \mathsf{Set}).\delta(Y : X \to \mathsf{Set}).\iota(\Sigma X Y) \quad : \quad \mathsf{IR}\ \mathsf{Set}$$

- **Key Idea:** Interpret as functors $F : \mathsf{Fam}\ D \to \mathsf{Fam}\ D$
  - Objects of $\mathsf{Fam}\ D$ are pairs $(U : \mathsf{Set}, T : U \to D)$. Morphisms are maps f such that



- **Semantics of IR codes:** $[\![-]\!] : \mathsf{IR}\ D \to \mathsf{Fam}\ D \to \mathsf{Fam}\ D$

$$[\![\iota\ d]\!](U,T) = (1, \mathsf{const}\ d)$$

$$[\![\sigma\ A\ f]\!](U,T) = \coprod_{a:A} [\![f a]\!](U,T)$$

$$[\![\delta\ A\ F]\!](U,T) = \coprod_{g:A\to U} [\![F(T \circ g)]\!](U,T)$$

- **Initial Algebra Semantics:** The Foundation of data types
  - All you need do is choose a *category* and a *functor*
  - IAS will generate for you a data type, recrusion operator, church encoding, shortcut fusion rule, induction rule etc.

- **Induction Recursion:** State of the art is data type theory
  - Allows inductively defined indexed data to be generated at the same time as the indexes themselves.
  - All one needs to do is instantiate the usual IAS in a category of families