Certified Software Development with Dependent Types in Idris

Lecture 3. Defining Data Types

Vitaly Bragilevsky (bravit@sfedu.ru)

I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science
Southern Federal University, Rostov-on-Don, Russia
part of Erasmus+ teaching mobility agreement with University of Twente

Administrivia

- Course materials (public repo):
  `https://github.com/bravit/csd-utwente` (`git.io/vw72I`)
- Assignments (private repo):
  `https://github.com/mmcs-sfedu-courses/csd-utwente-assignments`
- Assignments will be published every two or three classes

How to build complex data?

- Basic datatypes: Nat, Int, Double, Bool, Char, String,...
- Product: combine several components (conjunction)
- Sum: choose one over several alternatives (disjunction)
- Sums of Products

Flavours of datatypes in Idris

- Enumerations
- Union types
- Recursive types
- Generic types
- Dependent types

Unified mechanism to define data types in Idris

Defining new datatype

```
data typename : type where
   alt_1 : arg1 -> arg2 -> ... -> typename args
   ...
   alt_n : arg1 -> arg2 -> ... -> typename args
```

- `type` can be either Type or type of Type-valued function
- `typename` is a type constructor
- `alt_i` are data constructors
- when `type` is a Type-valued function typename can take arguments
- we can provide useful information either in type constructor or in data constructor

Unified mechanism to define data types in Idris

Example 1: type for booleans

```
data Bool : Type where
  False : Bool
  True : Bool
```

Example 2: type for pair of natural numbers

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Boolean value

```
b : Bool
b = True
```

Pair of naturals

```
p : NPair
p = MkNPair 2 3
```

Simplified syntax for bools and pairs

Example 1: type for booleans

```
data Bool : Type where
  False : Bool
  True : Bool
```

Example 2: type for pair of natural numbers

```
data NPair : Type where
  MkNPair : Nat -> Nat -> NPair
```

Simplified syntax

```
data Bool = False | True

data NPair = MkNPair Nat Nat
```

Boolean value

```
b : Bool
b = True
```

Pair of naturals

```
p : NPair
p = MkNPair 2 3
```

Sums: there can be more than two alternatives

Ordering (Prelude)

```
data Ordering = LT | EQ | GT

compare : Ord a => a -> a -> Ordering
```

Suits

```
data Suit = Spades | Clubs | Diamonds | Hearts

isRed : Suit -> Bool
isRed Diamonds = True
isRed Hearts = True
isRed _ = False
```

Products: types can be generic

Generic pair

```
data Pair : Type -> Type -> Type where
  MkPair : a -> b -> Pair a b
```

Generic pair (simplified syntax)

```
data Pair a b = MkPair a b
```

```
p : Pair Nat Char
p = MkPair 3 'x'
```

Syntactic sugar

```
p : (Nat, Char)
p = (3, 'x')
```

Natural numbers as sum of products

Type for naturals

```
data Nat : Type where
  Z : Nat
  S : Nat -> Nat
```

Simplified syntax

```
data Nat = Z | S Nat
```

Natural values

```
n0 : Nat
n0 = Z

n2 : Nat
n2 = S (S Z)

n5 : Nat
n5 = S(S(S(S(S Z))))
```

Functions over naturals

```
odd : Nat -> Bool
odd Z     = False
odd (S k) = not (odd k)

plus : Nat -> Nat -> Nat
plus Z     y = y
plus (S k) y = S (plus k y)
```

Mutually recursive functions

- Every function should be defined before use, but we can use 'mutual' declaration.

```
mutual
  even : Nat -> Bool
  even Z     = True
  even (S k) = oneven k

  oneven : Nat -> Bool
  oneven Z     = False
  oneven (S k) = even k
```

Maybe a: computation with a possibility of failure

Maybe a

```
data Maybe a = Nothing | Just a
```

Example: previous natural number

```
prev : Nat -> Maybe Nat
prev Z = Nothing
prev (S k) = Just k
```

Analysing Maybe a

```
prev_str : Nat -> String
prev_str n =
  case prev n of
    Nothing => "there is no predecessor of " ++ show n
    Just m => "predecessor to " ++ show n
              ++ " is " ++ show m
```

Useful function for Maybe a

```
maybe : Lazy b -> (a -> b) -> Maybe a -> b
```

- First argument Lazy b — default value (used in case of Nothing), not evaluated until actually used (lazily evaluated)

Analysing Maybe a

```
prev_str : Nat -> String
prev_str n = maybe ("there is no predecessor of " ++ show n)
                   (\ m => "predecessor to " ++ show n
                            ++ " is " ++ show m)
                   (prev n)
```

Either a b

Either a b

```
data Either a b = Left a | Right b
```

- Idea 1: two possible results
- Idea 2: actual result (Right) or failure with explanation (Left)

```
either : Lazy (a -> c) -> Lazy (b -> c) -> Either a b -> c
```

Lists are sums of products too!

```
data List a = Nil | (::) a (List a)
```

```
xs : List Integer
xs = 1 :: 2 :: 3 :: 4 :: 5 :: Nil

ys : List Integer
ys = [1,2,3,4,5] -- or even [1..5]
```

```
map : (a -> b) -> List a -> List b
map f []        = []
map f (x :: xs) = f x :: map f xs
```

Binary Search Trees

bstree.idr

```
data BSTree : Type -> Type where
     Empty : Ord a => BSTree a
     Node : Ord a => (left : BSTree a) ->
                       (val : a) ->
                       (right : BSTree a) -> BSTree a

insert : a -> BSTree a -> BSTree a
insert x Empty = Node Empty x Empty
insert x orig@(Node left val right)
      = case compare x val of
             LT => Node (insert x left) val right
             EQ => orig
             GT => Node left val (insert x right)
```

Vectors (Data.Vect)

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

Example: rotating vector

rotate.idr

Broken version

```
rotate : Vect n a -> Vect n a
rotate [] = []
rotate (x :: xs) = xs ++ [x]
```

Error message

```
When checking right hand side of rotate':
Type mismatch between
        Vect (k + 1) a (Type of xs ++ [x])
and
        Vect (S k) a (Expected type)
Specifically:
        Type mismatch between
                plus k 1
        and
                S k
```

Example: rotating vector

Broken version

```
rotate : Vect n a -> Vect n a
rotate [] = []
rotate (x :: xs) = xs ++ [x]
```

Error message

```
When checking right hand side of rotate':
Type mismatch between
        Vect (k + 1) a (Type of xs ++ [x])
and
        Vect (S k) a (Expected type)

Specifically:
        Type mismatch between
                plus k 1
        and
                S k
```

WTF?

```
plus k 1 /= S k ?
```

What's the problem?

Definition of plus (revisited)

```
plus : Nat -> Nat -> Nat
plus Z     y = y
plus (S k) y = S (plus k y)
```

- So, `plus` is defined via recursion over the first argument.

```
plus 1 k = plus (S Z) k = S (plus Z k) = S k
```

- But

```
plus k 1 = ???
```

- Typechecker has no information about `k` so it stucks.

How to deal with this?

- Method 1: persuade typechecker (write proof) — later
- Method 2: rewrite function — now

rotate.idr

Revised version

```
rotate' : Vect n a -> Vect n a
rotate' [] = []
rotate' (x :: xs) = ins_last x xs
  where
    ins_last : (x : a) -> (xs : Vect k a) -> Vect (S k) a
    ins_last x [] = [x]
    ins_last x (y :: xs) = y :: ins_last x xs
```

Finite sets

```
data Fin : Nat -> Type where
   FZ : Fin (S k)
   FS : Fin k -> Fin (S k)
```

Example: list of values of `Fin 3`

- `FZ`
- `FS FZ` — `FZ` here has type `Fin 2`
- `FS (FS FZ)` — `FZ` here has type `Fin 1`

- Type `Fin 0` is uninhabited
- `Fin n` is often used for indices (over vectors, for example):

Using Fin for indexing vector

```
index : Fin n -> Vect n a -> a
```

indexing.idr

```
v : Vect 5 Nat
v = [1,2,3,4,5]

v3 : Nat
v3 = index 3 v

v6 : Nat
v6 = index 6 v -- does not typecheck
```

Combining Fin and Maybe while indexing

```
tryIndex : Integer -> Vect n a -> Maybe a
tryIndex = ???
```

```
integerToFin : Integer -> (n : Nat) -> Maybe (Fin n)
```

tryIndex.idr

```
tryIndex : Integer -> Vect n a -> Maybe a
tryIndex {n} i xs = case integerToFin i n of
                         Nothing => Nothing
                         (Just i') => Just (index i' xs)
```

Dependent pair

```
data Sigma : (a : Type) -> (P : a -> Type) -> Type where
   MkSigma : {P : a -> Type} -> (x : a) -> P x -> Sigma a P
```

```
vec : Sigma Nat (\n => Vect n Int)
vec = MkSigma 2 [3, 4]
```

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

Using dependent pair: filter

```
filter : (a -> Bool) -> Vect n a -> ???
```

```
filter : (a -> Bool) -> Vect n a -> (p : Nat ** Vect p a)
```

filtering.idr

```
evens : Vect m Nat -> (n ** Vect n Nat)
evens v = filter (\a : Nat => mod a 2 == 0) v
```

Lectures Plan

1. Introduction
2. Defining Functions
3. Defining Data Types

---

4. 09/05, 10:45 — Input/Output
5. 10/05, 10:45 — First Class Types
6. 11/05, 15:45 — Interfaces
7. 12/05, 15:45 — Equality and Decidability
8. 13/05, 13:45 — Theorem Proving

---

9. 17/05, 10:45 — Relations aka Predicates over Types
10. 18/05, 15:45 — Controlling Effects
11. 19/05, 14:30 — Implementing EDSLs

Bibliography

- Idris Tutorial: Types and Functions
  http://docs.idris-lang.org/en/latest/tutorial/typesfuns.html
- Idris Libraries Source Code
  https://github.com/idris-lang/Idris-dev/tree/master/libs/