Certified Software Development with Dependent Types in Idris

Lecture 4. Input/Output

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

One point from the last lecture

Defining new datatype

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

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

- `f1 : Vect 5 Nat`
- `f2 : Vect n Nat`
- `f3 : Vect n Nat -> Vect n Nat`
- `f4 : Vect n Nat -> Vect m Nat`

- `f5 : Vect n Nat -> (m : Nat ** Vect m Nat)`

Dependent pair

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

```
(m : Nat ** Vect m Nat) = Sigma Nat (\m => Vect m Nat)

(2 ** [0,0]) = MkSigma (2 : Nat) ([0,0] : Vect 2 Nat)
```

- `f4` — $\forall n\ \forall m$
- `f5` — $\forall n\ \exists m$

Functions repl and replWith

```
repl : String -> (String -> String) -> IO ()

replWith : a                  -- original state
        -> String             -- prompt
        -> (a ->                   -- current state
             String ->             -- user input
             Maybe (String,a)) -- Maybe (output, new state)
        -> IO ()
```

Example of replWith

```
summate : Int -> String -> Maybe (String, Int)
summate s y = case the Int (cast y) of
                0 => Nothing
                n => let s' = s + n in
                     Just ("Sum=" ++ cast s' ++ "\n", s')

main : IO ()
main = replWith 0 "> " summate
```

```
> 10
Sum=10
> 5
Sum=15
> 5
Sum=20
> hello
```

Input/Output with IO

```
module Main

main : IO ()
main = do
  putStr "Enter your name: "
  x <- getLine
  putStrLn ("Hello " ++ x ++ "!")
```

- Evaluating vs Executing
- IO-actions as descriptions of what should be done at runtime
- Sequencing

Reading and Validating Numbers

```
readNumber : IO (Maybe Nat)
readNumbers : IO (Maybe (Nat, Nat))
```

readNumbers.idr

```
readNumber : IO (Maybe Nat)
readNumber = do
   s <- getLine
   if all isDigit (unpack s)
     then pure (Just (cast s))
     else pure Nothing

readNumbers : IO (Maybe (Nat, Nat))
readNumbers = do
  Just n1 <- readNumber | Nothing => pure Nothing
  Just n2 <- readNumber | Nothing => pure Nothing
  pure (Just (n1, n2))
```

Writing Loops

```
countdown : Nat -> IO ()
```

countdown.idr

```
import System

countdown : Nat -> IO ()
countdown Z = putStrLn "Done"
countdown (S k) = do
  printLn (S k)
  usleep 1000000
  countdown k
```

Reading a Vector

readVect.idr

```
readVectLen : (len : Nat) -> IO (Vect len String)
readVectLen Z = pure []
readVectLen (S k) = do
  s <- getLine
  xs <- readVectLen k
  pure (s :: xs)
```

Given lehgth

```
readVect : IO (len ** Vect len String)
readVect = do
  s <- getLine
  if s == ""
    then pure (_ ** [])
    else do
           (_ ** xs) <- readVect
           pure (_ ** s :: xs)
```

Unknown lehgth

Reading and Zipping Two Vectors

```
zip : Vect n a -> Vect n b -> Vect n (a, b) -- Data.Vect.zip

readAndZip : IO (len ** Vect len (String, String))
```

```
readAndZip : IO (len ** Vect len (String, String))
readAndZip = do
  (len1 ** v1) <- readVect
  (len2 ** v2) <- readVect
  case exactLength len1 v2 of
    Nothing => pure (_ ** [])
    (Just v) => pure (_ ** zip v1 v)
```

```
exactLength : (len : Nat) -> Vect m a -> Maybe (Vect len a)
```

Record syntax (1)

Defining record

```
record Person where
    constructor MkPerson
    firstName, middleName, lastName : String
    age : Int

fred : Person
fred = MkPerson "Fred" "Joe" "Bloggs" 30
```

Accessing record

```
> firstName fred
"Fred" : String
> age fred
30 : Int
> :t firstName
firstName : Person -> String
```

NB!

- All accessor functions were generated automatically

Record syntax (2)

Updating record

```
> fred
MkPerson "Fred" "Joe" "Bloggs" 30
> record { firstName = "Jim" } fred
MkPerson "Jim" "Joe" "Bloggs" 30 : Person
> record { firstName = "Jim", age = 20 } fred
MkPerson "Jim" "Joe" "Bloggs" 20 : Person
```

Parameterized record

```
record Prod a b where
    constructor Times
    fst : a
    snd : b
```

Dependent record

```
record SizedClass (size : Nat) where
    constructor SizedClassInfo
    students : Vect size Person
    className : String
```

```
addStudent : Person -> SizedClass n -> SizedClass (S n)
addStudent p c = record { students = p :: students c } c
```

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/`