Certified Software Development with Dependent Types in Idris

Lecture 10. Effectful Computations

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

Содержание

Content

Pure and effectful computations

- Pure computations
- Effectful computations:
  - input/output (console, files)
  - managing state
  - random numbers
  - non-determinism
  - raising/handling exceptions
- Combining effects
- Changing effects (due to the results of previous effectful computations)

Effectful "Hello world"

"Hello world" example (hello.idr)

```
module Main

import Effects
import Effect.StdIO

hello : Eff () [STDIO]
hello = putStrLn "Hello world"

main : IO ()
main = run hello
```

- Effects — library for effectful computations
- `Eff result_type effects`
- `STDIO` — name of effect for console I/O (defined in `Effect.StdIO`)
- `putStrLn` — function defined for STDIO effect
- `run` — "runner" of an effectful computation

```
$ idris hello.idr -o hello -p effects
$ ./hello
Hello world
```

Type function Eff

```
SimpleEff.Eff : (t : Type) ->
                (input_effs : List EFFECT) -> Type
TransEff.Eff  : (t : Type) ->
                (input_effs : List EFFECT) ->
                (output_effs : List EFFECT) -> Type
DepEff.Eff    : (t : Type) ->
                (input_effs : List EFFECT) ->
                (output_effs_fn : t -> List EFFECT) -> Type
```

```
EffM : (m : Type -> Type) -> (t : Type)
       -> (List EFFECT)
       -> (t -> List EFFECT) -> Type
```

SimpleEff.Eff implementation

```
Eff : (x : Type) -> (es : List EFFECT) -> Type
Eff x es = {m : Type -> Type} -> EffM m x es (\v => es)
```

Effect and EFFECT

```
Effect : Type
Effect = (x : Type) -> Type -> (x -> Type) -> Type
```

- The return type of the computation.
- The input resource.
- The computation to run on the resource given the return value.

```
data EFFECT : Type where
  MkEff : Type -> Effect -> EFFECT
```

- The input resource.
- The effect.

Running Effectful Computation

Automatically inferred environment

```
runPure : {env : Env id xs} -> Eff a xs -> a
run : Applicative m => {env : Env m xs} -> Eff a xs -> m a
```

- Environment should contain resources for all effects.

Default values for environment

```
interface Default a where
    default : a
```

Setting explicit environment

```
runPureInit : Env id xs -> Eff a xs -> a
runInit : Applicative m => Env m xs -> Eff a xs -> m a
```

Content

Content

STDIO Effect

```
module Effect.StdIO

STDIO : EFFECT

putChar   : Char ->   Eff () [STDIO]
putStr    : String -> Eff () [STDIO]
putStrLn  : String -> Eff () [STDIO]

getStr    : Eff String [STDIO]
getChar   : Eff Char [STDIO]

print     : Show a => a -> Eff () [STDIO]
printLn   : Show a => a -> Eff () [STDIO]

Handler StdIO IO where
```

Simple Example (name.idr)

```
hello : Eff () [STDIO]
hello = do
  putStr "Name? "
  x <- getStr
  putStrLn ("Hello " ++ trim x ++ "!")

main : IO ()
main = run hello
```

Content

Stateful Computations

- Computation is called stateful if there is an ability to keep and update some state (variable).
- This state variable normally has initial value which can be changed any number of times in the course of a computation.
- This ability is an effect implemented by several functions (get/put and others).

Stateful Computation in General

```
f : (x1 : a1) -> (x2 : a2) -> ... -> Eff t [...,STATE state,...]
```

- `t` — type for the result of computation
- `STATE` — name of an effect in Idris (`STATE : Type -> EFFECT`)
- `state` — type of state variable (resource)

State Effect

```
module Effect.State

STATE : Type -> EFFECT

get     :               Eff x  [STATE x]
put     : x ->          Eff () [STATE x]
putM    : y ->          Eff () [STATE x] [STATE y]
update  : (x -> x) -> Eff () [STATE x]
updateM : (x -> y) -> Eff () [STATE x] [STATE y]

Handler State m where
```

Stack of Integers as State

```
Stack : Type
Stack = List Int

push : Int -> Eff () [STATE Stack]
pop : Eff Int [STATE Stack]
```

```
push : Int ->
       Eff () [STATE Stack]
push a = do
  st <- get
  put (a :: st)
```

```
pop : Eff Int [STATE Stack]
pop = do
  (x :: xs) <- get
  put xs
  pure x
```

- do-blocks are used for sequencing effectful computations
- `x <- e` binds the result of an effectful operation `e` to a variable `x`.
- `pure e` turns a pure value `e` into the result of an effectful operation.

We can implement push in different ways

Original Version

```
push : Int -> Eff () [STATE Stack]
push a = do
  st <- get
  put (a :: st)
```

get and put

```
get :      Eff t [STATE t]
put : t -> Eff () [STATE t]
```

!-notation

```
push a = put $ a :: !get
```

Via update : (x -> x) -> Eff () [STATE x]

```
push = update . (::)
```

Example: Expression in Reverse Polish Notation

```
4 19 2 * + === 4 + (19 * 2)
```

Evaluation algorithm using stack

1. Process string word by word from left to right
    1. Every number goes to the stack
    2. For any operation
        1. pop two numbers off the stack
        2. perform operation over them
        3. push result back to the stack
2. When the string is over result is found at the top of the stack (assuming input string is correct)

Implementing Task Components

Operation Processing

```
process_tops : (Int -> Int -> Int) -> Eff () [STATE Stack]
process_tops op = do
  x <- pop
  y <- pop
  push (x `op` y)
```

Or

```
process_tops op = update (\(x::y::xs) => (x `op` y) :: xs)
```

Or even

```
process_tops op = push (!pop `op` !pop)
```

Implementing Task Components (2)

Processing one word

```
step : String -> Eff () [STATE Stack]
step "+" = process_tops (+)
step "*" = process_tops (*)
step n   = push (cast n)
```

Splitting string and running computation

```
evalRPN : String -> Int
evalRPN s = runPure $ do
    mapE (\s => step s) (words s)
    pop -- result is found at the top of the stack
```

```
main : IO ()
main = putStrLn $ cast $ evalRPN "4 19 2 * +"
```

Content

SYSTEM Effect

```
module Effect.System

import Effects
import System

SYSTEM : EFFECT

getArgs : Eff (List String) [SYSTEM]
time    : Eff Int [SYSTEM]
getEnv  : String -> Eff (Maybe String) [SYSTEM]
system  : String -> Eff Int [SYSTEM]

Handler System IO where
```

Content

RND Effect

```
module Effect.Random

RND : EFFECT

srand       : Integer ->                     Eff () [RND]
rndInt      : Integer -> Integer ->          Eff Integer [RND]
rndFin      : (k : Nat) ->                   Eff (Fin (S k)) [RND]
rndSelect   : List a ->                      Eff (Maybe a) [RND]
rndSelect'  : Vect (S k) a ->                Eff a [RND]

Handler Random m where
```

Content

EXCEPTION Effect

```
module Effect.Exception

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]

          Handler (Exception a) Maybe where
          Handler (Exception a) List where
          Handler (Exception a) (Either a) where
          Handler (Exception a) (IOExcept a) where
Show a => Handler (Exception a) IO where
```

Example: Parsing Number (exc.idr)

```
data EErr = NotANumber | OutOfRange

parseNumber : Int -> String -> Eff Int [EXCEPTION EErr]
parseNumber lim str = do
  when (not (all isDigit (unpack str))) (raise NotANumber)
  let x = cast str
  if (0 <= x && x <= lim)
      then pure x
      else raise OutOfRange
```

```
Idris> the (Either EErr Int) (run (parseNumber 42 "20"))
Right 20 : Either EErr Int
Idris> the (Either EErr Int) (run (parseNumber 42 "50"))
Left OutOfRange : Either EErr Int
Idris> the (Either EErr Int) (run (parseNumber 42 "xxx"))
Left NotANumber : Either EErr Int
Idris> the (Maybe Int) (run (parseNumber 42 "xxx"))
Nothing : Maybe Int
```

Using parseNumber

```
work : Int -> String -> Eff Int [EXCEPTION EErr]
work up s = do
  n <- parseNumber up s
  pure (n + 1)

io : Eff () [STDIO]
io = do
  putStr "Number (0-10)? "
  s <- getStr
  case run (work 10 s) of
    Right n => putStrLn $ "OK: " ++ show n
    Left _ => putStrLn "Error"

main : IO ()
main = run io
```

Executing Program

```
$ ./exc
Number (0-10)? 5
OK: 6
$ ./exc
Number (0-10)? xxx
Error
$ ./exc
Number (0-10)? 42
Error
```

Content

SELECT Effect

```
import Effects
import Effect.Select

SELECT : EFFECT

select : List a -> Eff a [SELECT]

Handler Selection Maybe where
Handler Selection List where
```

Content

Content

STATE and STDIO (stateful-hello.idr)

```
hello : Eff () [STATE Int, STDIO]
hello = do
  putStr "Name? "
  putStrLn ("Hello " ++ trim !getStr ++ "!")
  update (+1)
  putStrLn ("I've said hello to: " ++ show !get ++ " people")
  hello

main : IO ()
main = run hello
```

STDIO and SYSTEM (sys.idr)

```
module Main

import Effects
import Effect.StdIO
import Effect.System

printArgs : Eff () [STDIO, SYSTEM]
printArgs = do
  args <- getArgs
  printLn args

main : IO ()
main = run printArgs
```

```
$ idris sys.idr -o sys -p effects
$ ./sys arg1 "arg 2" arg3
["./sys", "arg1", "arg 2", "arg3"]
```

RND, STDIO, and SYSTEM (rnd.idr)

```
dice3 : Eff (Integer, Integer, Integer) [RND]
dice3 = do
  a <- rndInt 1 6
  b <- rndInt 1 6
  c <- rndInt 1 6
  pure (a,b,c)

cast_dice : Eff () [RND,STDIO,SYSTEM]
cast_dice = do
  t <- time
  srand t
  (a, b, c) <- dice3
  printLn (a,b,c)

main : IO ()
main = run cast_dice
```

SELECT and EXCEPTION (select.idr)

```
triple : Int ->
         Eff (Int, Int, Int) [SELECT, EXCEPTION String]
triple max = do
  z <- select [1..max]
  y <- select [1..z]
  x <- select [1..y]
  if (x * x + y * y == z * z)
     then pure (x, y, z)
     else raise "No triple"
```

Executing in Different Contexts

```
main : IO ()
main = do
  print $ the (Maybe _) $ run (triple 10)
  print $ the (List _) $ run (triple 10)
```

```
$ ./select
Just (3, (4, 5))
[(3, (4, 5)), (6, (8, 10))]
```

Content

Labelled Effects (calc.idr)

```
calc_step : Int -> Eff () ['Sum ::: STATE Int,
                           'Prod ::: STATE Int]
calc_step a = do
  'Sum :- update (+a)
  'Prod :- update (*a)
```

- Symbols 'Sum and 'Prod
- Labelling effect and operations over it

```
main : IO ()
main = printLn  $ runPureInit ['Sum := 0, 'Prod := 1]
                              (calc_step 5)
```

- Labelling initial state

Creating Labelled Effects

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> Eff a [x] -> Eff a [l ::: x]
(:=)  : (l : lbl) -> res -> LRes l res
```

Using Labelled Effects

```
calc : Eff (Int, Int) ['Sum ::: STATE Int,
                        'Prod ::: STATE Int]
calc = do
  calc_step 5
  calc_step 10
  calc_step 20
  s <- 'Sum :- get
  p <- 'Prod :- get
  pure (s, p)

main : IO ()
main = printLn $ runPureInit ['Sum := 0, 'Prod := 1] calc
```

Labelled Effects and STDIO

```
calc_IO : Eff () ['Sum ::: STATE Int,
                  'Prod ::: STATE Int,
                  STDIO]
calc_IO = do
  let x = trim !getStr
  case all isDigit (unpack x) of
     False => printLn (!('Sum :- get), !('Prod :- get))
     True => do
              calc_step (cast x)
              calc_IO

main : IO ()
main = runInit ['Sum := 0, 'Prod := 1, ()] calc_IO
```

Content

Simple Arithmetic Expressions

```
data Expr = Val Integer
          | Add Expr Expr

eval : Expr -> Integer
eval (Val x) = x
eval (Add l r) = eval l + eval r
```

```
Idris> eval (Add (Val 10) (Val 50))
60 : Integer
```

Expressions with Variables

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr

Env : Type
Env = List (String, Integer)

eval : Expr -> Eff Integer [EXCEPTION String, STATE Env]
eval (Val x) = pure x
eval (Add l r) = pure $ !(eval l) + !(eval r)
eval (Var x) = case lookup x !get of
                    Nothing => raise $
                                 "No such variable " ++ x
                    Just val => pure val
```

Running Expressions with Variables

```
runEval : List (String, Integer) -> Expr -> Maybe Integer
runEval args expr = run (eval' expr)
  where
    eval' : Expr -> Eff Integer [EXCEPTION String, STATE Env]
    eval' e = do
        put args
        eval e
```

Expressions with Random Numbers

```
data Expr = Val Integer
          | Var String
          | Add Expr Expr
          | Random Integer
```

```
eval : Expr -> Eff Integer [EXCEPTION String, RND, STATE Env]
...
eval (Random upper) = rndInt 0 upper
```

Expressions with Random Numbers and Printing

```
eval (Random upper) = do val <- rndInt 0 upper
                         putStrLn $ "Random: " ++ (show val)
                         pure val
```

```
Can't solve goal
        SubList [STDIO]
                [EXCEPTION String,
                 RND,
                 STATE (List (String, Integer))]
```

Giving alias to effects collection

```
EvalEff : Type -> Type
EvalEff t = Eff t [STDIO, EXCEPTION String, RND, STATE Env]

eval : Expr -> EvalEff Integer
```

Running eval

```
runEval : List (String, Integer) -> Expr -> IO Integer
runEval args expr = run (eval' expr)
  where eval' : Expr -> EvalEff Integer
        eval' e = do put args
                     eval e

main : IO ()
main = do
  r <- runEval [("a", 5)] (Add (Var "a") (Random 10))
  printLn r
```

Content

Content

Type function Eff

```
SimpleEff.Eff : (t : Type) ->
                (input_effs : List EFFECT) -> Type
TransEff.Eff  : (t : Type) ->
                (input_effs : List EFFECT) ->
                (output_effs : List EFFECT) -> Type
DepEff.Eff    : (t : Type) ->
                (input_effs : List EFFECT) ->
                (output_effs_fn : x -> List EFFECT) -> Type
```

```
EffM : (m : Type -> Type) -> (t : Type)
       -> (List EFFECT)
       -> (t -> List EFFECT) -> Type
```

SimpleEff.Eff

```
Eff : (x : Type) -> (es : List EFFECT) -> Type
Eff x es = {m : Type -> Type} -> EffM m x es (\v => es)
```

List as a State

```
readInt : Eff () [STATE (List Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

Vect as a State

```
readInt : Eff () [STATE (Vect n Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

Wrong!

- We change not only state but also its type!

```
readInt : Eff ()[STATE (Vect n Int), STDIO]
                [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             putM (cast x :: !get)
```

```
putM : y -> Eff () [STATE x] [STATE y]
```

Problem

- If we have read not a number can we extend vector?

```
readInt : DepEff.Eff Bool [STATE (Vect n Int), STDIO]
          (\ok => if ok then [STATE (Vect (S n) Int), STDIO]
                        else [STATE (Vect n Int), STDIO])
```

```
readInt = do let x = trim !getStr
             case all isDigit (unpack x) of
                  False => pureM False
                  True => do putM (cast x :: !get)
                             pureM True
```

```
pureM : (val : a) -> EffM m a (f val) f
```

Using readInt

```
readN : (n : Nat) ->
        Eff () [STATE (Vect m Int), STDIO]
               [STATE (Vect (n + m) Int), STDIO]
readN Z = pure ()
readN {m} (S k) = case !readInt of
                      True => readN k
                      False => readN (S k)
```

Error!

```
...
Specifically:
        Type mismatch between
                plus k (S m)
        and
                S (plus k m)
```

Using readInt: correct implementation (read.idr)

```
readN : (n : Nat) ->
        Eff () [STATE (Vect m Int), STDIO]
               [STATE (Vect (n + m) Int), STDIO]
readN Z = pure ()
readN {m} (S k) =
    case !readInt of
      True => rewrite plusSuccRightSucc k m in readN k
      False => readN (S k)
```

Content

File Management Protocol

- It is necessary to open a file for reading before reading it
- Opening may fail, so the programmer should check whether opening was successful
- A file which is opened for reading must not be written to, and vice versa
- When finished, an open file handle should be closed
- When a file is closed, its handle should no longer be used

FILE_IO Effect

```
module Effect.File

import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open      : ???
close     : ???
readLine  : ???
writeLine : ???
eof       : ???

Handler FileIO IO where
```

- Modes: Read | Write
- Effect is parameterized over open file
- OpenFile incapsulates file handle and is parameterized over Mode
- open should result in OpenFile with specified Mode
- readLine and writeLine should check Mode of OpenFile

File Opening

```
open : (fname : String)
       -> (m : Mode)
       -> Eff Bool [FILE_IO ()]
                   (\res => [FILE_IO (case res of
                                          True => OpenFile m
                                          False => ())])
```

Other Functions

```
close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine  : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]
```

Example

Reading file to a list of strings

```
readFile : Eff (List String) [FILE_IO (OpenFile Read)]
readFile = readAcc [] where
    readAcc : List String ->
              Eff (List String) [FILE_IO (OpenFile Read)]
    readAcc acc = if (not !eof)
                     then readAcc (!readLine :: acc)
                     else pure (reverse acc)
```

Dumping file

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
                  True => do putStrLn (show !readFile)
                             close
                  False => putStrLn ("Error!")
```

No direct work with file handle!

Let's try to make a mistake!

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
                  True => putStrLn (show !readFile)
                  False => putStrLn ("Error!")
```

Type Checking Error!

```
Type checking ./files.idr
files.idr:16:56:
When checking right hand side of Main.case block in dumpFile
        ...
        Specifically:
                Type mismatch between
                        ()
                and
                        OpenFile Read
```

Let's try to make another mistake. . .

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
                  False => do putStrLn (show !readFile)
                              close
                  True => putStrLn ("Error!")
```

```
Type checking ./files.idr
files.idr:17:33:
When checking right hand side of Main.case block in dumpFile
        ...
        Specifically:
                Type mismatch between
                        OpenFile m
                and
                        ()
```

More mistakes are on the way!

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Write) of
                  True => do putStrLn (show !readFile)
                             close
                  False => putStrLn ("Error!")
```

```
Type checking ./files.idr
files.idr:17:32:
When checking right hand side of Main.case block in dumpFile
        ...
        Specifically:
                Type mismatch between
                        OpenFile m
                and
                        ()
```

Pattern-matching bind

Another implementation for dumpFile

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name  = do
   True <- open name Read | False => putStrLn "Error"
   putStrLn (show !readFile)
   close
```

Pattern Matching

```
do
  pat <- val | <alternatives>
  p
```

Desugared Variant

```
do
  x <- val
  case x of
     pat => p
     <alternatives>
```

Example: checking command line arguments

```
emain : Eff () [FILE_IO (), SYSTEM, STDIO]
emain = do
  [prog, name] <- getArgs | [] => putStrLn "Can't happen!"
                          | [prog] =>
                              putStrLn "No arguments!"
                          | _ =>
                              putStrLn "Too many arguments!"
  dumpFile name

main : IO ()
main = run emain
```

Bibliography

1. The Effects Tutorial
   http://docs.idris-lang.org/en/latest/effects/index.html