River: a functional reversible language

Jen-Shin Lin

December 13, 2016

From our previous researches and studies, we did get some idea of how can we start we developing on functional reversible language. On the one hand, we confirm that indeed linear logic and linear types cannot be directly applied here. However the way of how it explicitly presents and manages information, variables and their flow is a very helpful guide for us. On the other hand the syntax created from reverible logic lacks some essential mechanism or operator such that it can be use pratically and easily. Therefore we combine several techniques and notions so far we know from (i) linear logic and types and its term-assignment [1, 9, 2, 6, 4, 8, 5] (ii) previous works on reversibility [10, 7, 3, 11] , and come up with a new functional language with reversibility, named *river*. It is (i) designed based on pattern calculus rather than λ calculus; and (ii) is implemented with the idea of how can we expose and manipulate information in linear type and programming; and (iii) has a mechanism for reversible computation. The follows are the major features or techniques we used to develop.

Pattern calculus To reveal and monitor the information-flow within a given function, one way to go is to realise two techniques (i) precisely describe very variable in environment, which will be explain later; and (ii) apply delicate pattern matching. The first thing, we develop our syntax with an explicit structures for *case*s where all the pattern matching will happen. As one can see in source **Expr.hs**, *case*s can only appeared within the case of two kind of syntax: *MatEq* and *Match* where are the places pattern-matching can be happened. Besides we also seperate the definition of *Terms* out of *Expr*. This is helpful to expose *pattern* and making the implementation of pattern matching simpler. As matter of fact, by doing this, we can be able to isolate *MTerm* which is in fact representing what a *pattern* could be.

Matchable The simple pattern matching can get most jobs done. But, since we are expecting all behaviors of variable will be captured or supervisable in our system. We introduced *matchable*s for denoting variables in matching pattern. This leads us to the result that terms is our language can and will be refined as two subsets: *VTerm* and *MTerm* in **Expr.hs**. In general speaking, *occurrences of variable*, namely *VTerm*, indicate consuming exist information from environment and *occurrences of matchable*, *MTerm*, on the other hand, indicate introducing new information into environment. Matchables also provides the basical supporting for ensuring and overwatching the linearity of environment. This function is defined as *zipMatEnv* in **Ctx.hs**.

Linear context To overwatch information-flow while computing a function, namely an *Expr*, the linear typed programming show us a very good way of how – we restrict that every variable can be used *once and only once*. When a function is getting to be evaluated, input values will be put into environment (as variables) then every use of those value (information) will be precisely controlled by variables and matchables. One variable die then one matchable will be borned, vice versa. This is implemented as the variable-matchable thing mentioned above, but to make this correct, we must limit the happening of arbitrary function applications. In fact, we ask that applications can be happened only in the case of *LetIn* in **Expr.hs**.

So far we introduced the theniques we used for enhancing (or, in someway, limiting) the capability of our language. Next we will explain several minor efforts that simply improve the usability of it.

Syntax for branch control The most vital problem of reversible computation is how can we deal and trace branches and mergings, the backwards branches. These branches can be grouped as two kinds. The first one called computation branches which is presented in river as *case*s that have intact structures for both of forewards and backwards evaluations. The other branches are produced by using same value twice or more. This is an problem because, when one compute backwards, these two information may not be the same thing. The linearity property can actually prevent this, but, in practical, we still need to duplicate value or variable. Hence we develop a specific technique for this purpose that is in fact implemented as two parts of our syntax – *DupIn* and *MatEq* which is defined in **Expr.hs**. They are the foundation stone of our reversibility here since all possible information-creating/-removing will be described by them.

Restricted function declaration In λ calculus, we can define arbitrary function via λ abstraction. But not suitable for us since it lackes ability of pattern matching because every constructor are another functions there. This is the reason why we use pattern calculus rather λ calculus. A interesting feature in pattern calculus is there is no λ abstraction. The way to write a function in pattern calculus is to define a global declaration for it together with several *case*s where is the function body defined. This *case*-instead-of-λ thing make us deal no λ abstraction and make the whole evaluating simpler.

Since there is no way to define a lambda function in river, all primitive functions neede to be pre-defined. Therefore we define a table to globally describe what functions we have and how they will be computed. This is implemented in **Func.hs** where we have the table named *globalFuns* and a querying function *findFun*.

Combinator As one can easily find out in **Expr.hs**, the whole syntax are defined is several different layers each of which as defined in terms of haseklll-data, which introduces new constructors. This leads us to a situation of tons of constructors matching and restructing. To release ourself from this, we introduce several combinators: *int2nat, nat2int, redN, mat, var*, one can find their definitions in **Expr.hs** and their usages in all other files, mostly **Func.hs** and **RedB.hs**.

All the source codes can be found in the following appendix.

Appendix: Source code

Expr.hs

```
module Expr where
--
type FName   = String
type FunName = String
--
data Nat = Z | S Nat deriving (Show, Eq)

int2nat :: Int -> Nat
int2nat 0 = Z
int2nat n = S $ int2nat (n-1)
nat2int :: Nat -> Int
nat2int Z = 0
nat2int (S n) = 1 + (nat2int n)
--
data Val     = Pair Val Val
             | N Nat
             | B Bool
             deriving (Show, Eq)
--
redN :: Val -> Val
```

```haskell
redN (N (S n)) = N n
--
data Mat      = Mat FName
                deriving (Show, Eq)
data Var      = Var FName
                deriving (Show, Eq)
--
data Term a   = Lit Val
                | Atom a
                | Prod (Term a) (Term a)
                | Fst  (Term a)
                | Snd  (Term a)
                --
                | NatS (Term a)
                deriving (Show, Eq)
--
type MTerm    = Term Mat
type VTerm    = Term Var
--
mat :: FName -> MTerm
mat = Atom . Mat
var :: FName -> VTerm
var = Atom . Var
--
-- call-by-name function application
type FApp     = (FunName, [VTerm])
--
data Case     = (:->) {uncasePatt :: MTerm, uncaseExpr :: Expr}
                deriving (Show, Eq)
--
data Expr     = Term VTerm
                | LetIn MTerm (Either VTerm FApp) Expr
                | DupIn MTerm VTerm Expr
                | Match VTerm [Case]
                | MatEq (VTerm, VTerm) Case Case
                deriving (Show, Eq)
```

Func.hs

```haskell
module Func where
--
import Expr
--
type FSpace = [FSpec]
data FSpec = FSpec { fname :: FunName
                   , fargs :: [Mat]
                   , fbody :: Expr
                   }
           deriving (Show, Eq)
--
findFun :: FunName -> FSpace -> FSpec
findFun fn [] = error $ "there is no such function: " ++ fn
findFun fn (fs:fss)
    | fn == fname fs = fs
    | otherwise = findFun fn fss
--
globalFuns :: FSpace
globalFuns =
      [ FSpec "succ" [Mat "#0"] succExpr
      , FSpec "plus" [Mat "#0", Mat "#1"] plusExpr
      , FSpec "plusR" [Mat "#0"] plusRExpr
      , FSpec "neg" [Mat "#0"] negExpr
      , FSpec "and" [Mat "#0", Mat "#1"] andExpr
      ]
--
succExpr :: Expr
succExpr =
   Match (var "#0")
      [ (Lit $ N Z)  :->
          (Term $ Lit $ N (S Z))
      , (NatS $ mat "u") :->
          LetIn (mat "u2")
              (Right ("succ", [var "u"]))
              (Term $ NatS $ var "u2")
      ]
plusExpr :: Expr
plusExpr =
   Match (var "#1")
      [ (Lit (N Z))  :->
          DupIn (Prod (mat "a") (mat "b")) (var "#0")
              (Term $ Prod (var "a") (var "b"))
      , (NatS $ mat "u") :->
          LetIn (Prod (mat "x2") (mat "u2"))
              (Right ("plus", [var "#0", var "u"]))
              (Term $ Prod (var "x2") (NatS $ var "u2"))
      ]
plusRExpr :: Expr
plusRExpr = LetIn (Prod (mat "#0_a") (mat "#0_b"))
   (Left $ var "#0") $
      MatEq (var "#0_a", var "#0_b")
          ((mat "x")  :-> (Term $ Prod (var "x") (Lit $ N Z)))
          ((Prod (mat "x") (NatS (mat "u"))) :->
              (LetIn (Prod (mat "x2") (mat "u2"))
                  (Right ("plusR", [Prod (var "x") (var "u")]))
                  (Term $ Prod (var "x2") (NatS $ var "u2"))))
negExpr :: Expr
negExpr =
   Match (var "#0")
      [ (Lit (B True))  :->
          (Term $ Lit $ B False)
      , (Lit (B False)) :->
          (Term $ Lit $ B True)
      ]
andExpr :: Expr
andExpr = undefined
```

Ctx.hs

```
module Ctx where
--
import Expr
--
type Env = [(Var, Val)]
--
```

```haskell
find :: Var -> Env -> (Val, Env)
find va [] = error $ "there is no such variable" ++ (show va)
find v1 ((v2, val) : env)
    | v1 == v2 = (val, env)
    | otherwise = let (val2, env2) = find v1 env
        in (val2, (v2, val) : env2)
--
appSigma :: Env -> VTerm -> (Val, Env)
appSigma env (Lit val) = (val, env)
appSigma env (Atom va) = find va env
appSigma env (Prod vt1 vt2) =
    let (val1, env1) = appSigma env vt1
        (val2, env2) = appSigma env1 vt2
    in (Pair val1 val2, env2)
appSigma env (Fst  vt) = appSigma env vt
appSigma env (Snd  vt) = appSigma env vt
appSigma env (NatS vt) =
    let (N nat, env1) = appSigma env vt
    in (N $ S nat, env1)
--
appSigmaList :: Env -> [VTerm] -> [Val] -> ([Val], Env)
appSigmaList env [] vs = (reverse vs, env)
appSigmaList env (vt : vts) vs =
    let (val, env') = appSigma env vt
    in appSigmaList env' vts (val : vs)
--
zipMatEnv :: MTerm -> Val -> Env -> Env
zipMatEnv (NatS mt) (N (S nat)) env = zipMatEnv mt (N nat) env
zipMatEnv (Prod mt1 mt2) (Pair v1 v2) env =
    zipMatEnv mt2 v2 $ zipMatEnv mt1 v1 $ env
-- zipMatEnv (Atom (Mat name)) (N n) env = (Var name, N n) : env
-- zipMatEnv (Atom (Mat name)) (B b) env = (Var name, B b) : env
-- zipMatEnv (Atom (Mat name)) (Pair v1 v2) env =
    -- (Var name, Pair v1 v2) : env
-- these three are identical, merged as follow
zipMatEnv (Atom (Mat name)) v env = (Var name, v) : env
zipMatEnv mt v env = error $ "\n\tMT{"++(show mt)
    ++ "} \n\tVal{"++(show v)++"} \n\tEnv{"++(show env)++"}"
--
zipMatEnvC :: (MTerm, Val) -> Env -> Env
zipMatEnvC (m, v) e = zipMatEnv m v e
```

RedB.hs

```haskell
module RedB where
--
import Expr
import Func
import Subs
import Ctx
--
redBeta :: Env -> Expr -> (Val, Env)
----------------------------------
redBeta env (Term vt) =
    appSigma env vt
----------------------------------
redBeta env (LetIn mt (Left vt) e) =
    let (val, env') = appSigma env vt
        newEnv = zipMatEnv mt val env'
    in redBeta newEnv e
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
redBeta env (LetIn mt (Right (fun, vts)) e) =
    let fspec = findFun fun globalFuns
        (vals, env') = appSigmaList env vts []
        res = apply fspec vals
        newEnv = zipMatEnv mt res env'
    in redBeta newEnv e
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
-- MTerm in dup will be limited in terms of Prod
-- VTerm in dup will be limited in terms of Atom
redBeta env (DupIn (Prod (Atom (Mat ma1)) (Atom (Mat ma2))) (Atom va) e) =
        let (val, env') = find va env
            newEnv = (Var ma2, val) : (Var ma1, val) : env'
        in redBeta newEnv e
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
redBeta env (Match vt cases) =
    let (val, env1) = appSigma env vt
        (env2, e) = matching val cases
    in redBeta (env1 ++ env2) e
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
redBeta env (MatEq (Atom va1, Atom va2) case1 case2) =
    let (val1, env1) = find va1 env
        (val2, env2) = find va2 env1
    in if val1 == val2
        then case case1 of
            ((Atom (Mat ma)) :-> e1) -> redBeta ((Var ma, val2) : env2) e1
            ((NatS (Atom (Mat ma))) :-> e2) ->
                redBeta ((Var ma, redN val1) : env2) e2
        else case case2 of
            ((Prod mt1 mt2) :-> e1) ->
                let newEnv = zipMatEnv mt2 val2 $ zipMatEnv mt1 val1 $ env2
                in redBeta newEnv e1
-- -- -- -- -- -- -- -- -- -- -- -- -- -- --
apply :: FSpec -> [Val] -> Val
apply (FSpec _ args body) vals =
    fst $ redBeta (foldr zipMatEnvC [] $ zip (fmap Atom args) vals) body
    -- we cannot check the equality of length of args and vals here.. :(
-- -- -- --
matching :: Val -> [Case] -> (Env, Expr)
--
matching (Pair v1 v2) (((Prod mt1 mt2) :-> e) : cs) =
    let env = zipMatEnv mt2 v2 $ zipMatEnv mt1 v1 []
    in (env, e)
--
matching (N nat)      (((Lit (N n)) :-> e) : cs)
    | nat == n = ([], e)
    | otherwise = matching (N nat) cs
--
matching (N nat)      (((Atom ma) :-> e) : cs) =
    (zipMatEnv (Atom ma) (N nat) [], e)
--
matching (N (S nat))  (((NatS mt) :-> e) : cs) =
    maybe (matching (N (S nat)) cs) id $ localMatchingN (N nat) (mt :-> e)
--
matching (B b)        (((Lit (B b')) :-> e) : cs)
    | b == b' = ([], e)
    | otherwise = matching (B b) cs
```

```
--
matching (B b)          (((Atom ma) :-> e) : cs) =
    (zipMatEnv (Atom ma) (B b) [], e)
--
matching val (_ : cs) = matching val cs
matching val [] = error $ "Non-exhaustive patterns for: " ++ (show val)
-- -- -- --
localMatchingN :: Val -> Case -> Maybe (Env, Expr)
localMatchingN (N n) ((Atom (Mat ma)) :-> e) = Just ([(Var ma, N n)], e)
localMatchingN (N n) ((Lit (N m)) :-> e)
    | n == m = Just ([], e)
    | otherwise = Nothing
localMatchingN (N (S nat)) ((NatS mt) :-> e) =
    localMatchingN (N nat) (mt :-> e)
localMatchingN _ _ = Nothing
--
test_neg = redBeta [(Var "#0", B False)] negExpr
test_plus m n = redBeta
    [(Var "#0", N $ int2nat m), (Var "#1", N $ int2nat n)] plusExpr
test_succ m = redBeta
    [(Var "#0", N $ int2nat m)] succExpr
test_plusR (m,n) = redBeta
    [(Var "#0", Pair (N $ int2nat m) (N $ int2nat n))] plusRExpr
```

References

[1] Benton, N., Bierman, G., and Paiva, V. Term assignment for intuitionistic linear logic. Tech. rep., 1992.

[2] Benton, N., Bierman, G. M., Hyland, J. M. E., and de Paiva, V. A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications* (1993), M. Bezem and J. F. Groote, Eds., Springer-Verlag LNCS 664, pp. 75–90.

[3] Brown, G., and Sabry, A. Reversible communicating processes. In *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015.* (2015), pp. 45–59.

[4] Girard, J.-Y. Linear logic: Its syntax and semantics, 1995.

[5] Novitzká, V., and Mihályi, D. Resource-oriented programming based on linear logic. In *Acta Polytechnica Hungarica* (2007).

[6] Ronchi della Rocca, S., and Roversi, L. Lambda calculus and intuitionistic linear logic. In *Logic Colloquium '94* (Clermont Ferrand, France, 1994).

[7] Sparks, Z., and Sabry, A. Superstructural reversible logic. In *3rd International Workshop on Linearity* (2014).

[8] Turner, D. N., and Wadler, P. Operational interpretations of linear logic, 1998.

[9] Wadler, P. A taste of linear logic. In *Mathematical Foundations of Computer Science* (Gdańsk, Poland, 1993), A. M. Borzyszkowski and S. Sokolowski, Eds., Springer-Verlag LNCS 781, pp. 185–210.

[10] Yokoyama, T., Axelsen, H. B., and Glück, R. Towards a reversible functional language. In *Proceedings of the Third International Conference on Reversible Computation* (Berlin, Heidelberg, 2012), RC'11, Springer-Verlag, pp. 14–29.

[11] Yokoyama, T., Axelsen, H. B., and Glück, R. Fundamentals of reversible flowchart languages. *Theor. Comput. Sci. 611*, C (Jan. 2016), 87–115.