River: a functional reversible language

Jen-Shin Lin

December 12, 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*.

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. Here we develop our syntax with an explicit structures for *case*s where all the pattern matching will happen. Besides we also seperate *terms* from *expr*. This is helpful to expose *pattern* and making the implementation of pattern matching simpler.

Matchable The simple pattern matching can get most jobs done. But, since we are expecting all behaviors of variable will be captured in our system, we introduced *matchable*s for denoting variables in matching pattern. In general speaking, occurrences of variable indicate consuming exist information from environment and occurrences of matchable indicate introducing new information into environment. Matchables also provides the basical supporting for ensuring and overwatching the linearity of environment.

Linear context To overwatch the information-flow while computing a function, the linear typed programming show us a very good way - 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.

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.

Syntactic foundation for dup-eq .

Restricted function declaration ...

Global function register ...

Operational semantics ...

Combinator ...

Conclusion Finally we have our language with a operational semantics and reversed reduction. The source code can be found in here [https://dl.dropboxusercontent.com/u/11966021/src/River.zip]. Yet there are still lots of improvement can be done. The following are several possible directions. [1] *view-pattern*?: [2] *de bruijn index*: [3] *locally nameless*: [4] *hybrid context*: [5] *explicit variables management*:

```
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
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 -- add ‘Fin i‘ to make Lit impossible
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)
--
{-
TODO: add index onto ‘Term a‘ for limiting the number of atoms.
TODO: add de brjin index or even locally nameless
-}
```

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.