拡張された拡張可能作用

オレッグ・キセリョーヴ
東北大学大学院情報科学研究科
oleg@okmij.org

こんなコードは
どんな言語で
型安全に
書ける？

```
let x = ref ()
x := 1
let y = !x + 1
x := "文字"
-- let z = !x + 2 -- 型エラー
let z = !x ++ "列"
```

```
-- 実用例： 当初化
let x = ref Rec{lab1 = 1, lab2 = () {-未定-}}
let y = ...
x := !x{lab2 = f y}
```

型状態 (即ち, パラメーターされた)「モナド」も拡張・合成できるように
パラメーターされたモナドの型は、セッション型の一種

1 背景

1.1 普通の状態計算 (固定型の状態)

```
get :: State s s
put :: s → State s ()
```

```
t1 = do
  put 1
  x ← fmap (+1) get
  put 2
  -- put 'a'      ✗
  -- fmap not get ✗
```

1.2 拡張可能作用：より自由モナド

Open Union とは、直和 Either f gを任意個に一般化

```
type Union (r :: [* → *]) a              class (t :: k) ∈ (r :: [k])
inj    :: t ∈ r ⇒ t v → Union r v
prj    :: t ∈ r ⇒ Union r v → Maybe (t v)
decomp :: Union (t:r) v → Either (Union r v) (t v)
```

より自由モナド

```
data Eff r a where
  Pure  :: a → Eff r a
  Impure :: Union r x → (x → Eff r a) → Eff r a
instance Monad (Eff r) -- Functor無し
```

演算

```
data State s x where   -- 作用の記述子
  Get :: State s s
  Put :: s → State s ()

get :: State s ∈ r ⇒ Eff r s
get = Impure (inj Get) Pure
put :: State s ∈ r ⇒ s → Eff r ()
put s = Impure (inj (Put s)) Pure
```

解釈

```
runState :: Eff (State s : r) w → s → Eff r (w,s)
runState (Pure x) s = return (x, s)
runState (Impure u q) s = case decomp u of
  Right Get      → runState (q s) s
  Right (Put s)  → runState (q ()) s
  Left  u → Impure u (\x → runState (q x) s)
```

2 目標と問題点

2.1 型状態計算 (可変型の状態)

希望する例

```
test1 = do
  put True
  x ← get
  put 'a'
  y ← get
  return (x, y)
```

状態の型は変る可能。セッション型にも便利である

型付けるよ

```
test1 :: (State Bool ∈ r, State Char ∈ r) ⇒
Eff r (Bool, Char)
```

問題点 適切な型なものか! (なんで？)

2.2 パラメーターされたモナド

普通モナドではない

```
class Monadish (m :: k → k → * → *) where
  return :: a → m s s a
  (>>=) :: m s t a → (a → m t u b) → m s u b
```

2.3 より自由モナドとして実装

```
data FState s1 s2 a where
  Pure :: a → FState s s a
  Get  :: (s1 → FState s1 s2 b)        → FState s1 s2 b
  Put  :: sx → (() → FState sx s2 b) → FState s1 s2 b

instance Monadish FState where
  return = SPure
  Pure x  >>= k = k x
  Get k1  >>= k = Get (\x → k1 x >>= k)
  Put s k1 >>= k = Put s (\x → k1 x >>= k)
```

単純ね。

型状態計算のための拡張可能作用みたいが欲しい

3 導出

3.1 FState を一般化

```
data XEff (s1 :: k) (s2 :: k) a where
  Pure   :: a → XEff s s a
  Impure :: ?? s1 sx x → (x → XEff sx s2 b) → XEff s1 s2 b

instance Monadish XEff where
  return = Pure
  Pure x       >>= k = k x
  Impure fx k1 >>= k = Impure fx (\x → k1 x >>= k)
```

変数とその名前

```
data Var1 = Var1
data Var2 = Var2

with_var :: var → t var a1 a2 a3 → t var a1 a2 a3
with_var _ x = x
```

FState に対して特殊

```
data State var s1 s2 x where
  Get :: State s s s
  Put :: s2 → State s s2 ()

get :: var → XEff var s s s
get var = Impure (with_var var Get) Pure

runXEffState :: var → s1 → XEff var s1 s2 a → (a, s2)
```

```
test1 = do
  x0 ← get Var1
  put Var1 True
  x ← get Var1
  put Var1 'a'
  y ← get Var1
  return (x0, x, y)
```

```
runXEffState Var1 1 test1
-- ((1,True,'a'),'a')
```

複数の変数？
ここまで単純ですね。ところが...

要点

```
put Var1 True :: XEff [State Var1 a, State Var2 b] [State Var1 Bool, State Var2 b] ()
put Var2 True :: XEff [State Var1 a, State Var2 b] [State Var1 a, State Var2 Bool] ()
```

3.2 どのような Union？

```
data Cap (l :: * → * → *) -- 次のカパビリティ
class Member tag targ tnext ri ro | tag ri → targ, tag tnext ri → ro where
  inj :: tag targ (Cap tnext) v → Union ri ro v
  prj :: Union ri ro v → Maybe (tag targ (Cap tnext) v)

data DecompRes ri ro v where
  ReqOther :: Union ri ro v → DecompRes (t': ri) (t': ro) v
  ReqThis  :: t (Cap tnext) v → DecompRes (t': ri) (tnext': ri) v

decomp :: Union ri ro v → DecompRes ri ro v
decomp (UNow x)  = ReqThis x
decomp (UNext v) = ReqOther v
```

3.3 完成

```
data State var s tnext x where
  Get :: State var s (Cap (State var s)) s
  Put :: s2 → State var s (Cap (State var s2)) ()

get var    = Impure (inj (with_var var Get)) Pure
put var s2 = Impure (inj (with_var var (Put s2))) Pure

runState :: var → s1 → (XEff (State var s1 ': ri) (State var s2 ': ro) a) → XEff ri ro (a, s2)
runState _ s (Pure x) = return (x, s)
runState v s1 (Impure u q) = case decomp u of
   ReqThis GetS     → runState v s1 (q s1)
   ReqThis (PutS s2) → runState v s2 (q ())
   ReqOther u       → Impure u (\x → runState v s1 (q x))
```

test1 の推論された型: 限られた変化

```
test1 :: (Member (State Var1) t1 (State Var1 t1)    s  s1,
          Member (State Var1) t0 (State Var1 Bool) s1 s2,
          Member (State Var1) t2 (State Var1 t2)   s2 s3,
          Member (State Var1) t0 (State Var1 Char) s3 s4,
          Member (State Var1) t3 (State Var1 t3)   s4 s5) ⇒ XEff s s5 (t1, t2, t3)
```

(複数パーティ) セッション型ですね。

```
runState Var1 1 test1
-- runState Var1 1 test1 :: Num a ⇒ XEff ri ri ((a, Bool, Char), Char)
run (runState Var1 1 test1)          -- ((1,True,'a'),'a')
```

拡張可能: 複数パーティーセッション型

```
test2 = do
  x ← get Var2
  put Var2 "str "
  r ← test1
  y ← get Var2          run (runState Var2 () ∘ runState Var1 1 test2)
  return (r, x, y)      -- ((((1,True,'a'),(),"  str "),'a'),"  str ")
```