Functional Programming Techniques for Philosophy and Linguistics

Chris Barker and Jim Pryor, NASSLLI 2016

Friday Handout

Reader monad for binding (CB)

Yesterday we saw (briefly) how a Reader monad could be used for intensionalizing an extensional grammar. We mentioned that the Reader monad could also be used to implement a binding mechanism. Here's how. The type of a boxed alpha in the Reader monad is Ent -> alpha. For our lifting operations, we have

```
up = \a \x.a
map2 = \f xx yy . \x (f (xx w) (yy w))
```

This is just like our intensionalization monad. So a sentence without any pronouns lifts cleanly into the monad:

```
Ann (saw Bill)
unlifted: saw (ann) (bill)
lifted: (map2 saw) (up ann) (up bill)
      = (\xx yy . \x (saw (xx x) (yy x))) (\x.ann) (\x.bill)
      = \x (saw ((\x.ann) x) ((\x.bill) x))
      = \x (saw (ann)(bill))
```

This is just a boxed truth value. If the sentence contains a pronoun, the pronoun's value depends on an externally-supplied entity.

```
him = \x.x
```

Now we have a treatment of how the pronoun depends on a value supplied at the top level for "Ann saw him":

```
(map2 saw) (up ann) (him)
   = (\xx yy . \x (saw (xx x) (yy x))) (\x.ann) (\x.x)
   = \x (saw ((\x.ann) x) ((\x.x) x))
   = \x (saw (ann) (x))
```

Furthermore, we can put in non-innocent operators that bind the pronoun, such as a quantifier like "someone":

```
someone = \PP \exists y . PP (up y) y
```

So "someone_i said Ann saw him_i" with binding is

```
someone ((map2 said) ((map2 saw) (up ann) (him)))
  = someone (map2 said) (\x (saw (ann) (x)))
  = someone ((\xx yy . \x (said (xx x) (yy x)))
          (\x (saw (ann) (x))))
  = someone (\yy . \x (said (saw (ann) (x)) (yy x)))
  = ((\PP \exists y . PP (up y) y)
     (\yy . \x (said (saw (ann) (x)) (yy x))))
  = \exists y . (\yy . \x (said (saw (ann) (x)) (yy x))) (up y) y
  = \exists y . (\x (said (saw (ann) (x)) ((up y) x))) y
  = \exists y . (\x (said (saw (ann) (x)) y)) y
  = \exists y . said (saw (ann) y) y
```

Paraphrase: there exists a y such that y said Ann saw y. This is the bound reading.

It's easy to generalize to allow multiple variables to be bound independently.

Running tally (JP, using State monad)

* Suppose you're trying to use the State monad to keep a running side-tally of how often certain arithmetic operations have been used in computing a complex expression. You've settled upon the design plan of using the State monad, and defining a function like this:

let counting_plus xx yy = tick >>= λ_. map2 (+) xx yy

How should you define the operation *tick* to make this work? The intended behavior is that after running:

let zz = counting_plus (up 1) (counting_plus (up 2) (up 3)) in runState zz 0

you should get a payload/at-issue result of 6 (that is, 1+(2+3)) and a final side-tally of 2 (because + was used twice).

```
> let -- xx >> yy = xx >>= \_ -> yy
      tick :: State Int ()      tick = modify succ
      counting_plus xx yy = tick >> map2 (+) xx yy
      zz :: State Int Float
      zz = counting_plus (up 1) (counting_plus (up 2) (up 3))
  in runState zz 0
(6.0, 2)
```

* Instead of the design in the previous problem, suppose you had instead chosen to do things this way:

let counting_plus' xx yy = map2 (+) xx yy >>= tock

How should you define the operation *tock* to make this work, with the same behavior as before?

```
> let tock :: Float -> State Int Float
      tock = \z -> modify succ >> up z
      counting_plus' xx yy = map2 (+) xx yy >>= tock
      zz' = counting_plus' (up 1) (counting_plus' (up 2) (up 3))
  in runState zz' 0
(6.0, 2)
```

Lower level, all binding is Static:

a1. extensional, no variables Maybe or Set $\boxed{\alpha}$ = Maybe α	**a2**. extensional, with let/λ-binding Maybe (Reader) $\boxed{\alpha}$ = G→Maybe α	**a3**. extensional, with quantifiers Set (Reader) $\boxed{\alpha}$ = G→Set α	**a4**. a3 + epistemic "might"
b1. intensional, no variables Maybe (Intensional) $\boxed{\alpha}$ = W→Maybe α	**b2**. intensional, with let/λ-binding Maybe (Intensional (Reader)) $\boxed{\alpha}$ = W→G→Maybe α	**b3**. intensional, with quantifiers Set (Intensional (Reader)) $\boxed{\alpha}$ = W→G→Set α	**b4**. b3 + epistemic "might"
For sentences, α = ( )	For sentences, α = ( )	For sentences, α = Sequence of Entities	

Upper level, all binding is Dynamic:

(no binding, so same as a1)	**c2**. extensional, with let/λ-binding Maybe (State) $\boxed{\alpha}$ = G→(Maybe α)×G	**c3**. extensional, with quantifiers Set (State) $\boxed{\alpha}$ = G→(Set α)×G	**c4**. c3 + epistemic "might"
(no binding, so same as b1)	**d2**. intensional, with let/λ-binding Maybe (Intensional (State)) $\boxed{\alpha}$ = W→G→(Maybe α)×G	**d3**. intensional, with quantifiers Set (Intensional (State)) $\boxed{\alpha}$ = W→G→(Set α)×G	**d4**. d3 + epistemic "might"

data **Entity** = *Ann* | *Bill* | *Carol* | *Dave* | *Ella* | *Frank*

data **Term** = *Name* Entity | *Var* Char
data **Atomic** = *Pred0* String | *Pred1* String Term | *Pred2* String Term Term
data **Sent** = *Atom* Atomic | *Conj* Sent Sent | *Cond* Sent Sent | *Neg* Sent | *Disj* Sent Sent |
Let Char Term Sent | *Exists* Char Sent | *Forall* Char Sent | *Possibly* Sent

Extension of binary predicate = Set (2-tuples)
Extension of unary predicate = Set (1-tuples or Entities themselves)
Extension of nullary predicates / sentence constants ($\top$, $\bot$) =
Set (0-tuples or Borings) = { } (= False) or { ( ) } (= True)
(Later, we'll replace the Set (Boring) with Set (Sequence of Entities), and there again { } will encode falsity, and { … } will encode truth. Note that {emptyseq} also encodes truth, and ≠ { }.)

GSV have: { (my G: var→peg) × (my α: peg→entity) × World } [[sentence]]

My sentence meanings for b3: $\alpha \rightarrow (\text{World} \rightarrow G \rightarrow \{ \alpha \}) = \alpha \rightarrow \boxed{\alpha}_{b3} = \text{Kleisli}_{b3}\ \alpha$

My sentence meanings for a1: $\text{Boring} \rightarrow (\text{World} \rightarrow \underline{\{ \text{Boring} \}})$

$$\frac{\simeq \text{Bool}}{\simeq \text{Set World}}$$

$$\simeq \text{Set World}$$

Auxiliary function

guard0 f = λa. if f then ⇧a else zero ; in Haskell: guard f >> pure a
guard1 f j = λa. j a >>= λx. if f x then ⇧a else zero
guard2 : (β → γ → Bool) → (α → $\boxed{\beta}$) → (α → $\boxed{\gamma}$) → (α → $\boxed{\alpha}$)
guard2 f j k = λa. j a >>= λx. k a >>= λy. if f x y then ⇧a else zero

---

Semantics for a1 (extensional, no variables)

type **X** α = MaybeT Identity α = Maybe α
mapresult2$_X$ f xx yy = f xx yy ; like map2$_X$, except that we don't unwrap Maybes and Sets

type **Alpha** = ( )
type **Kleisli** β = Alpha → $\boxed{\beta}_X$
run : Kleisli Alpha → Bool
run k = let a0 = ( ) : Alpha
in case k a0 of { *None* ⟶ *False* | *One* _ ⟶ *True* }

cond a1s a2s = case (a1s, a2s) of { (*One* _, *None*) ⟶ *False* | else *True* }

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of *Name* e ⟶ ⇧e
Var v ⟶ ???

eval_sent : Sent → Kleisli Alpha
eval_sent sent = case sent of

Atom (*Pred0* s) ⟶ λa. guard0 (extensionOf s) a
Atom (*Pred1* s term1) ⟶ λa. guard1 (extensionOf s)
(eval_term term1) a
Atom (*Pred2* s term1 term2) ⟶ λa. guard2 (extensionOf s)
(eval_term term1)
(eval_term term2) a
ϕ ∧ ψ ⟶ eval_sent ϕ >=> eval_sent ψ

ϕ ⊃ ψ ⟶ let phi = eval_sent ϕ
psi = eval_sent ψ
f a a1s a2s = guard0 (cond a1s a2s) a
in λa. mapresult2$_X$ (f a) (phi a) ((phi >=> psi) a)
For each live α: retain it iff each way it can make ϕ true (each of its descendents in a1s)
also makes ϕ ∧ ψ true (has some descendent in a2s)

¬ ϕ ⟶ eval_sent (ϕ ⊃ ⊥)

ϕ ∨ ψ ⟶ eval_sent ((¬ ϕ) ⊃ ψ)

Let var term ϕ ⟶ ???
Exists var ϕ ⟶ ???
All var ϕ ⟶ ???

Semantics for a2 (extensional, with static let/λ-binding)

```
type Assign d = [(Var, d)]
```

assignments are implemented as lists of Var × d, where d is either an Entity or a Peg
we start with an empty assignment
assign0 = [ ] : Assign Entity
later, *assign0* = [ ] : Assign Peg
lookup var g = the d that g (most recently) associates with var, else undefined
assoc (var, d) g = (var, d) ◁ g

```
type X α = MaybeT (Reader (Assign Entity)) α = Assign Entity → Maybe α
mapresult2X f xx yy = λr. f (xx r) (yy r)

type Alpha = ( )
type Kleisli β = Alpha → [β]X
run : Kleisli Alpha → Bool
run k = let a0 = ( ) : Alpha
            assign0 = [ ] : Assign Entity
        in case k a0 assign0 of { None ⟶ False | One _ ⟶ True }

query = liftT ask
shift f xx = localshift f xx
with_binding var j k = λa. j a >>= λx. shift (assoc (var, x)) (k a)
cond a1s a2s = case (a1s, a2s) of { (One _, None) ⟶ False | else True }

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of   Name e ⟶ ⇧e
                                    Var v ⟶ map (lookup v) query
eval_sent : Sent → Kleisli Alpha
eval_sent sent = case sent of

  Atom (Pred0 s)                ⟶ λa. guard0 (extensionOf s) a
  Atom (Pred1 s term1)          ⟶ λa. guard1 (extensionOf s)
                                       (eval_term term1) a
  Atom (Pred2 s term1 term2)    ⟶ λa. guard2 (extensionOf s)
                                       (eval_term term1)
                                       (eval_term term2) a
  ϕ ∧ ψ ⟶ eval_sent ϕ >=> eval_sent ψ

  ϕ ⊃ ψ ⟶ let phi = eval_sent ϕ
                psi = eval_sent ψ
                f a a1s a2s = guard0 (cond a1s a2s) a
            in λa. mapresult2X (f a) (phi a) ((phi >=> psi) a)
  For each live α: retain it iff each way it can make ϕ true (each of its descendents in a1s)
                                also makes ϕ ∧ ψ true (has some descendent in a2s)
  …
  Let var term ϕ ⟶ with_binding var (eval_term term) (eval_sent ϕ)
  Exists var ϕ ⟶ ???
  All var ϕ ⟶ ???
```

Semantics for a3 (extensional, with static quantifiers)

```
type Assign d = [(Var, d)]
   lookup var g = the d that g (most recently) associates with var, else undefined
   assoc (var, d) g = (var, d) ◁ g

type X α = SetT (Reader (Assign Peg)) α = Assign Peg → Set α
mapresult2_X f xx yy = λr. f (xx r) (yy r)

type Alpha = [ Entity ]
type Kleisli β = Alpha → β_X
run : Kleisli Alpha → Bool
run k = let a0 = [ ] : Alpha
            assign0 = [ ] : Assign Peg
        in case k a0 assign0 of { ∅ ⟶ False | else True }

query = liftT ask
shift f xx = localshift f xx
with_binding var j k = λa. j a >>= λx. shift (assoc (var, len a)) (k (a ▷ x))
cond a1s a2s = all (λa. any (a isPrefixOf) a2s) a1s

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of   Name e  ⟶  ⇧e
                                    Var v  ⟶ map (nth_from a ∘ lookup v) query
eval_sent : Sent → Kleisli Alpha
eval_sent sent = case sent of

   Atom (Pred0 s)                ⟶ λa. guard0 (extensionOf s) a
   Atom (Pred1 s term1)          ⟶ λa. guard1 (extensionOf s)
                                       (eval_term term1) a
   Atom (Pred2 s term1 term2)  ⟶ λa. guard2 (extensionOf s)
                                       (eval_term term1)
                                       (eval_term term2) a
   ϕ ∧ ψ  ⟶  eval_sent ϕ >=> eval_sent ψ

   ϕ ⊃ ψ  ⟶ let phi = eval_sent ϕ
                 psi = eval_sent ψ
                 f a a1s a2s = guard0 (cond a1s a2s) a
             in λa. mapresult2_X (f a) (phi a) ((phi >=> psi) a)
   For each live α: retain it iff each way it can make ϕ true (each of its descendents in a1s)
                                also makes ϕ ∧ ψ true (has some descendent in a2s)
    …
    Let var term ϕ  ⟶  with_binding var (eval_term term) (eval_sent ϕ)
    Exists var ϕ  ⟶  with_binding var (⇧_inner domain) (eval_sent ϕ)
    All var ϕ  ⟶ eval_sent (¬ Exists var (¬ ϕ))
```

Semantics for c3 (extensional, with dynamic quantifiers)

```
type X α = SetT (State (Assign Peg)) α = Assign Peg → (Set α) × Assign Peg
mapresult2_X f xx yy = λs. f (xx s) (yy s) s

type Alpha = [ Entity ]
type Kleisli β = Alpha → [β]_X
run : Kleisli Alpha → Bool
run k = let a0 = [ ] : Alpha
            assign0 = [ ] : Assign Peg
        in case k a0 assign0 of { (∅,_) → False | else True }

query = liftT get
shift f xx = liftT (modify f) >> xx
add_binding var j = λa. j a >>= λx. shift (assoc (var, len a)) (⇧ (a ▷ x))
cond a1s a2s = all (λa. any (a isPrefixOf) a2s) a1s

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of   Name e  →  ⇧ e
                                    Var v  →  map (nth_from a ∘ lookup v) query
eval_sent : Sent → Kleisli Alpha
eval_sent sent = case sent of

   Atom (Pred0 s)                →  λa. guard0 (extensionOf s) a
   Atom (Pred1 s term1)          →  λa. guard1 (extensionOf s)
                                        (eval_term term1) a
   Atom (Pred2 s term1 term2)    →  λa. guard2 (extensionOf s)
                                        (eval_term term1)
                                        (eval_term term2) a
   ϕ ∧ ψ  →  eval_sent ϕ >=> eval_sent ψ

   ϕ ⊃ ψ  → let phi = eval_sent ϕ
                psi = eval_sent ψ
                f a (a1s,_) (a2s,_) s = (guard0 (cond a1s a2s) a, s)
            in λa. mapresult2_X (f a) (phi a) ((phi >=> psi) a)
   For each live α: retain it iff each way it can make ϕ true (each of its descendents in a1s)
                                also makes ϕ ∧ ψ true (has some descendent in a2s)
    …
    Let var term ϕ  →  add_binding var (eval_term term) >=> eval_sent ϕ
    Exists var ϕ  →  add_binding var (⇧_inner domain) >=> eval_sent ϕ

      or: λa. concat [(add_binding var (λ_. ⇧e) >=> eval_sent ϕ) a | e ∈ domain]

      (equivalent here, but latter is needed to handle closet example when we have Might)
      this doesn't use the (serial) plus, union operations from yesterday's definitions for SetT
      but instead we impose a monoidal structure on assignments and SetT (Statelikes):
          g1 <> g2 = if one of g1, g2 extends the other, then the longer/equal one, else undefined
      and ListT (Statelike monad) gets a zero = the SetT's zero
      and xx <> yy = get >>= \g0 -> xx >>= getboth >>=
                        \(a1s, g1) -> put g0 >> yy >>= modifyboth (\(a2s, g2) -> (a1s <> a2s, g1 <> g2))
      and concat { } = zero; concat {xx, yy} = xx <> yy; etc.

    All var ϕ  →  eval_sent (¬ Exists var (¬ ϕ))
                  with dynamic binding, can also use: (Exists var ⊤) ⊃ ϕ
```

Semantics for d3 (intensional, with dynamic quantifiers)

```
type X α = SetT (ReaderT World (State (Assign Peg))) α
         = World → Assign Peg → (Set α) × Assign Peg
mapresult2_X f xx yy = λw. λs. f (xx w s) (yy w s) s

type Alpha = [ Entity ]
type Kleisli β = Alpha → ⌈β⌉_X
run : Kleisli Alpha → Bool
run k = let a0 = [ ] : Alpha
            assign0 = [ ] : Assign Peg
        in case k a0 world0 assign0 of { (∅,_) ⟶ False | else True }

getworld = liftT ask
query = (liftT ∘ liftT) get
shift f xx = (liftT ∘ liftT) (modify f) >> xx
add_binding var j = λa. j a >>= λx. shift (assoc (var, len a)) (⇧(a ▷ x))
cond a1s a2s = all (λa. any (a isPrefixOf) a2s) a1s

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of   Name e  ⟶  ⇧e
                                    Var v  ⟶  map (nth_from a ∘ lookup v) query
eval_sent : Sent → Kleisli Alpha
eval_sent sent = case sent of

  Atom (Pred0 s)                ⟶  λa. getworld >>= λw. guard0 (intensionOf s w) a
  Atom (Pred1 s term1)          ⟶  λa. getworld >>= λw. guard1 (intensionOf s w)
                                                          (eval_term term1) a
  Atom (Pred2 s term1 term2)    ⟶  λa. getworld >>= λw. guard2 (intensionOf s w)
                                                          (eval_term term1)
                                                          (eval_term term2) a
  ϕ ∧ ψ  ⟶  eval_sent ϕ >=> eval_sent ψ

  ϕ ⊃ ψ  ⟶ let phi = eval_sent ϕ
                 psi = eval_sent ψ
                 f a (a1s,_) (a2s,_) s = (guard0 (cond a1s a2s) a, s)
             in λa. mapresult2_X (f a) (phi a) ((phi >=> psi) a)
  For each live α: retain it iff each way it can make ϕ true (each of its descendents in a1s)
                                also makes ϕ ∧ ψ true (has some descendent in a2s)
  …
  Let var term ϕ  ⟶  add_binding var (eval_term term) >=> eval_sent ϕ
  Exists var ϕ  ⟶  add_binding var (⇧_inner domain) >=> eval_sent ϕ
    or: λa. concat [(add_binding var (λ_. ⇧e) >=> eval_sent ϕ) a | e ∈ domain]
    (equivalent here, but latter is needed to handle closet example when we have Might)
  All var ϕ  ⟶ eval_sent (¬ Exists var (¬ ϕ))
                with dynamic binding, can also use: (Exists var ⊤) ⊃ ϕ
  Possibly ϕ  ⟶ λa. let phi = eval_sent phi
                        check g = any (λw. nonempty ((phi a) w g)) worlds
                    in query >>= λg. guard0 (check g) a
```

Semantics for d4 (intensional, with dynamic quantifiers, continuation-style)

```
type X α = SetT (ReaderT World (State (Assign Peg))) α
         = World → Assign Peg → (Set α) × Assign Peg
mapresult2_X f xx yy = λw. λs. f (xx w s) (yy w s) s

type Alpha = [ Entity ]
type Kleisli β = Alpha → ⌈β⌉_X
run : Kleisli Alpha → Bool
run k = let a0 = [ ] : Alpha
            assign0 = [ ] : Assign Peg
        in case k (guard0 True) a0 world0 assign0 of { (∅,_) ⟶ False | else True }

getworld = liftT ask
query = (liftT ∘ liftT) get
shift f xx = (liftT ∘ liftT) (modify f) >> xx
add_binding var j = λa. j a >>= λx. shift (assoc (var, len a)) (⇧(a ▷ x))
cond a1s a2s = all (λa. any (a isPrefixOf) a2s) a1s

eval_term : Term → Kleisli Entity
eval_term term = λa. case term of   Name e  ⟶  ⇧e
                                    Var v  ⟶  map (nth_from a ∘ lookup v) query
eval_sent : Sent → Kleisli Alpha → Kleisli Alpha
eval_sent sent = case sent of

  Atom (Pred0 s)                ⟶ λprev. prev >=> λa. getworld >>= λw. guard0 (intensionOf s w) a
  Atom (Pred1 s term1)          ⟶ λprev. prev >=> λa. getworld >>= λw. guard1 (intensionOf s w)
                                                                       (eval_term term1) a
  Atom (Pred2 s term1 term2)    ⟶ λprev. prev >=> λa. getworld >>= λw. guard2 (intensionOf s w)
                                                                       (eval_term term1)
                                                                       (eval_term term2) a
  φ ∧ ψ  ⟶ eval_sent ψ (eval_sent φ)

  φ ⊃ ψ  ⟶ let phi = eval_sent φ
                psi = eval_sent ψ
                f a (a1s,_) (a2s,_) s = (guard0 (cond a1s a2s) a, s)
                old xx yy = λa. mapresult2_X (f a) (xx a) (yy a)
            in λprev. prev >=> old (phi (guard0 True)) (psi (phi (guard0 True)))
  For each live α: retain it iff each way it can make φ true (each of its descendents in a1s)
                                also makes φ ∧ ψ true (has some descendent in a2s)
  …
  Let var term φ  ⟶ λprev. eval_sent φ (prev >=> add_binding var (eval_term term))
  Exists var φ  ⟶ ~~add_binding var (⇧_inner domain) >=> eval_sent φ~~

      λprev. λa. concat [ eval_sent φ (prev >=> add_binding var (λ_. ⇧e)) a | e ∈ domain]
  All var φ  ⟶ eval_sent (¬ Exists var (¬ φ))
                with dynamic binding, can also use: (Exists var ⊤) ⊃ φ

  Possibly φ  ⟶ λprev. λa. let phi = eval_sent phi
                              check g = any (λw. nonempty ((phi prev a) w g)) worlds
                           ~~in query >>= λg. guard0 (check g) a~~
                           in λw g. let (a', g') = prev w g in if check g then (a',g') else ([ ], g')
```