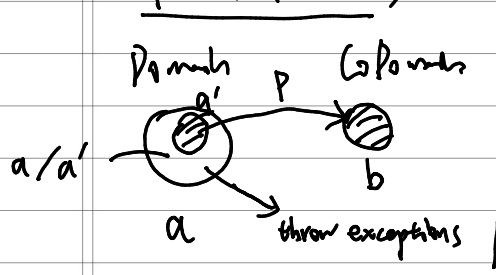
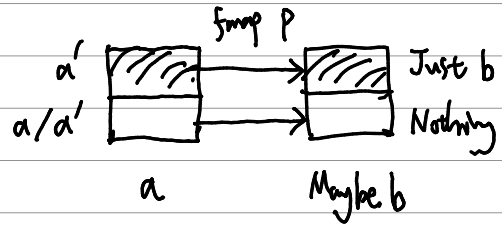
Partial Functions





Defined on a subset of Domain: $a' \subset a$

In regular programming language, primitive types (e.g. String, Integer) are not reshapable in the type system.

So function like $p :: a' \to b$

at best can have its type encoded as $a \to b$

where the program throws exceptions

whenever the input $x \notin a'$ (or $x \in a/a'$)

The programmer can handle this at logic level

by injecting conditionals / guard like

if $x \notin a'$ then return 'undefined' / new Error();

But 'undefined' / Error $\notin b$ either.

So b has to be extended. $\Rightarrow b' =$ | Success b | Error

Generally, it's Maybe type: | Just b | Nothing

1) Partial Function: $a \xrightarrow{P} b \Rightarrow a \to \boxed{b \mid \text{Error}} \triangleq \boxed{\text{Maybe}\ b}$

2) Function refers environment/config: $a \xrightarrow{e} b \Rightarrow (a, e) \to b \Rightarrow a \to e \to b$

make reference explicit in inputs

currying:

$\Rightarrow a \to (e \to b) \triangleq \boxed{\text{Reader}\ e}\ b$

3) Function mutates state / external data structure: $a \xrightarrow{s \to s\ (s+1)} b \Rightarrow (a, s) \to (b, s) \Rightarrow a \to (s \to (b, s)) \triangleq \boxed{\text{State}\ s}\ b$

current state ↓ next state ↓

currying:

4) Function has indeterministic returns: $a \xrightarrow{?} b \Rightarrow a \to [b]$ (or $\boxed{\text{List}}\ b$)

e.g. opponent's next move in chess

(a potentially infinite list of possible values)

5) IO

Generalize Maybe, Reader e, State s, List $\Rightarrow$ Effects

Call Functors, but not composable under Functor type class

able to compose them $\Leftarrow$ $a \to \boxed{m}\ b$

kleisli arrows

$(m :: * \to *)$ unary

Prove State is also a Functor

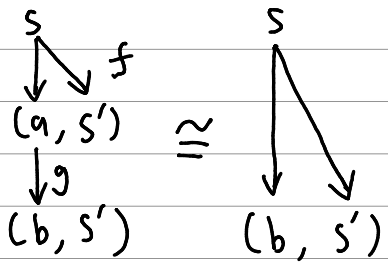
```
data State s a = State (s -> (a, s))

instance Functor (State s) where
  fmap :: (a->b) -> (State s) a -> (State s) b
                    = s->(a,s)    = s->(b,s)

  fmap g (State f) = State (\s ->
                     let (a, s') = f s
                         b = g a
                     in (b, s'))
```

λ-expression

anonymous function which invokes a hole/ place holder



Function Composition / Arrow : `(.) :: (b->c)->(a->b)->(a->c)`

Kleisli Arrow Composition : `(<=<) :: Monad m =>`
`(b->m c)->(a->m b)->(a->m c)`

> example: List

```
(<=<) :: (b -> [c]) -> (a -> [b]) -> (a -> [c])
g <=< f = \a -> let [b] bs = f a
                    [[c]] css = fmap g bs
                    cs = concat css
                in cs
```

(concat :: [[c]] -> [c]
concatenate all inner lists into one list)

point-free style:

```
g <=< f = concat . fmap g . f
```

```
class Applicative m => Monad m where
  (>>=) :: ma -> (a -> mb) -> mb   (also called Bind)
  Return :: a -> ma   (wrap a value into the Monadic type, Applicative's property)

instance Monad Maybe where
  Return :: a -> Maybe a
  Return x = Just x

instance Monad (Reader e) where
  Return :: a -> (Reader e) a   ignorely the input/environment
  Return x = Reader (\e -> x)
```

```
instance Monad (State s) where
  return :: a -> (State s) a
  return x = State (\s -> (x, s))
```

↑ the state transition doesn't modify the given state

```
instance Monad [] where
  return :: a -> [a]
  return x = [x]   < a list with a single value
                     (x : Nil)
```

Laws

1) Left Identity: return <=< f = f
2) Right Identity: f <=< return = f
3) Associativity: (h <=< g) <=< f = h <=< (g <=< f)