How FP Deals With Effects

PoolC 양제성

Source: 2024/02/07 (Wed)

© 2024. Je Sung Yang all rights reserved.

목차

1st Session

1. 함수형 프로그래밍 Intro
   - Overall Structure
   - Historical Review (CS + Math)
2. 함수형 패러다임
   - Core of Functional Thinking
   - FP Fact-Checking
3. FP는 정말 순수한가?
   - Optimizing with Purity
   - Effect Handling Basics

2nd Session

1. Lazy Evaluation
   - How Lazy Evaluation Works
   - Infinite Data Structure
   - Laziness & Purity
2. From Functor to Monad
   - Functor in PL
   - Monad in PL
3. Impurity in Pure World?
   - Side Effect in Pure World
   - Uniqueness Typing
   - IO Monad

© 2024. Je Sung Yang all rights reserved.

How Lazy Evaluation Works

“Evaluation on demand”

© 2024. Je Sung Yang all rights reserved.

How Lazy Evaluation Works

"Evaluation on demand"

<Let's code!>

© 2024. Je Sung Yang all rights reserved.

How Lazy Evaluation Works

Thunk: "A delayed computation"

© 2024. Je Sung Yang all rights reserved.

How Lazy Evaluation Works

Haskell

```haskell
xs = [1 .. 10] ++ undefined -- Thunk
ys = take 3 xs -- Thunk
main = print ys -- Force evaluation lazily
{-
  print (take 3 xs)
  print (take 3 ([1 .. 10] ++ undefined))
  print (1 : take 2 ([2 .. 10] ++ undefined))
  print (1 : 2 : take 1 ([3 .. 10] ++ undefined))
  print (1 : 2 : 3 : take 0 ([4 .. 10] ++ undefined))
  print (1 : 2 : 3 : [])
  print [1, 2, 3]
-}
```

© 2024. Je Sung Yang all rights reserved.

Infinite Data Structure

```haskell
ones :: [Int]
ones = 1 : ones
```

© 2024. Je Sung Yang all rights reserved.

Infinite Data Structure


```haskell
ones :: [Int]
ones = 1 : ones
```

```haskell
ones = 1 : ones
ones = 1 : 1 : ones
ones = 1 : 1 : 1 : ones
```

© 2024. Je Sung Yang all rights reserved.

Infinite Data Structure


```haskell
ones = 1 : ones -- Infinite list
main = print (take 3 ones)
{-
  print (take 3 ones)
  print (1 : take 2 ones)
  print (1 : 1 : take 1 ones)
  print (1 : 1 : 1 : take 0 ones)
  print (1 : 1 : 1 : [])
  print [1, 1, 1]
-}
```

Haskell

© 2024. Je Sung Yang all rights reserved.

Infinite Data Structure

<Let's code!>

Let's see more interesting examples...

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

"Laziness (generally) needs purity"

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Scenario: file pointer-based sequential read

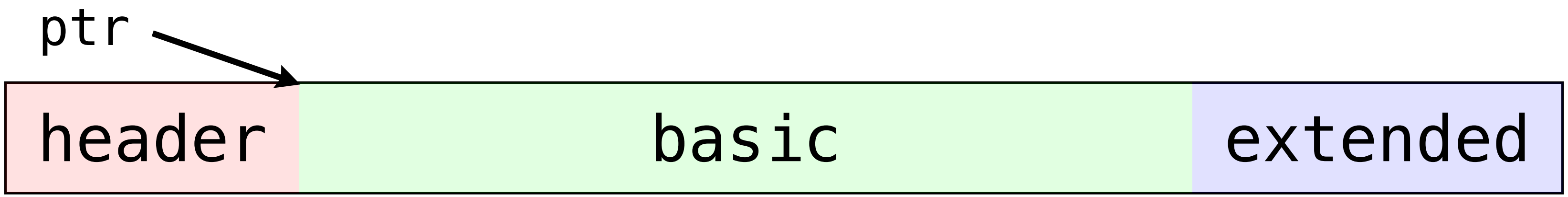
Assume that we are reading a config file structured like below.

`header`	`basic`	`extended`

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Expected: `readHeader` → `readBasicConfig` → `readExtendedConfig`

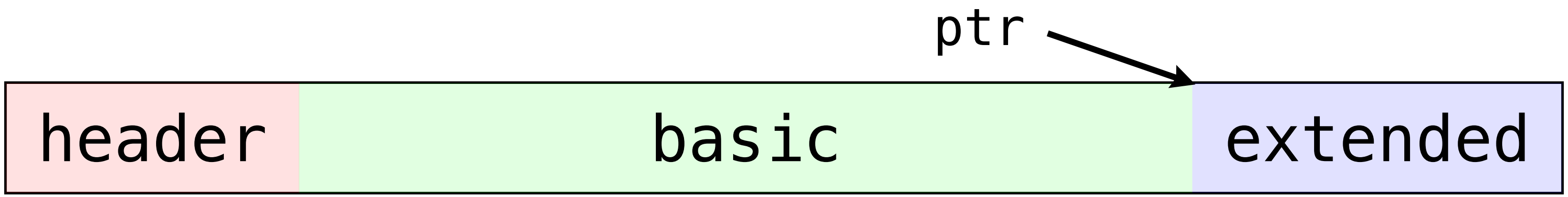



```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Expected: `readHeader` → `readBasicConfig` → `readExtendedConfig`

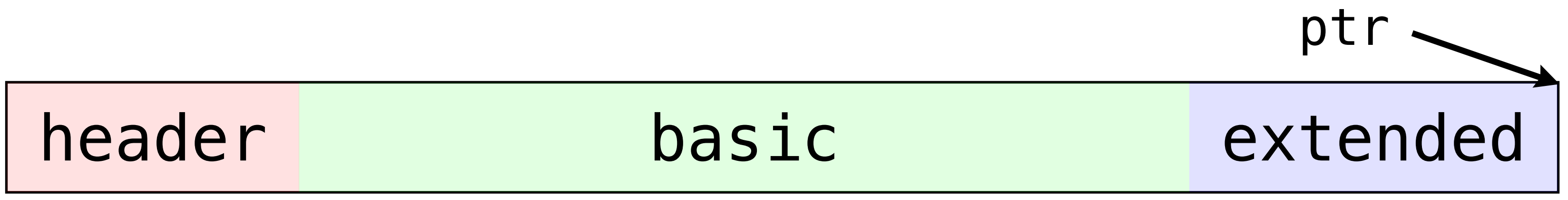



```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Expected: `readHeader` → `readBasicConfig` → `readExtendedConfig`

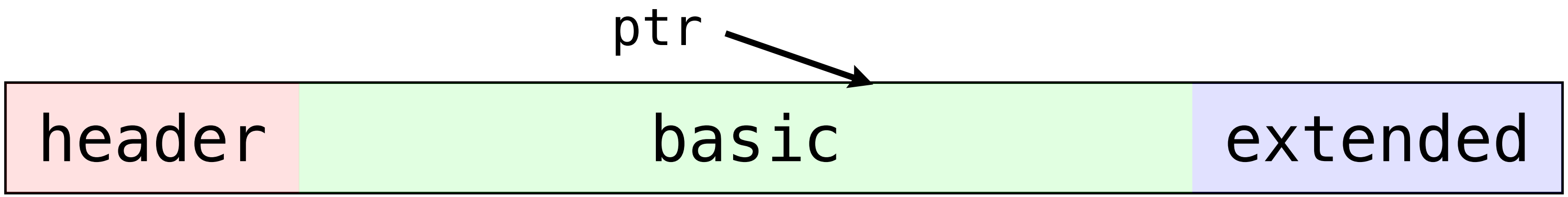



```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Lazy case: `readBasicConfig → readHeader → readExtendedConfig`

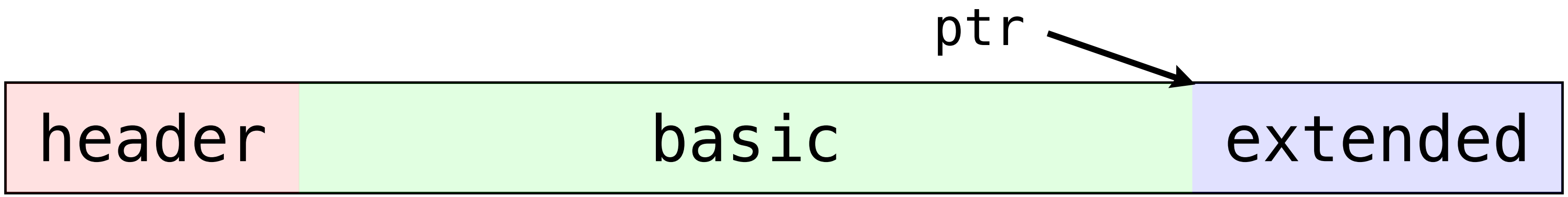



```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Lazy case: `readBasicConfig` → `readHeader` → `readExtendedConfig`

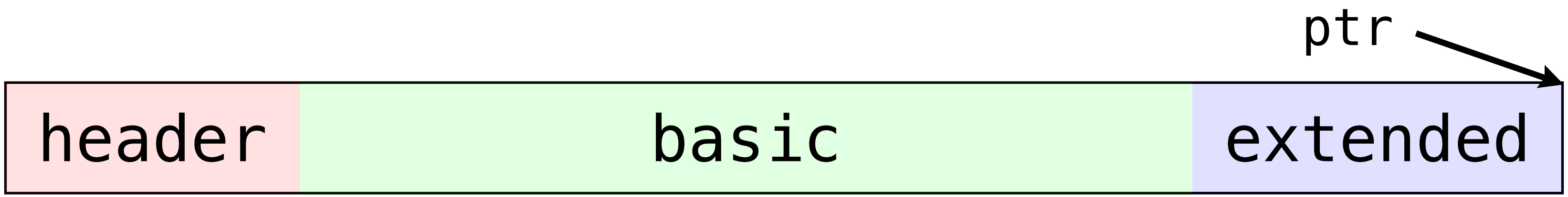



```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Laziness & Purity

Lazy case: `readBasicConfig` → `readHeader` → `readExtendedConfig`




```haskell
getConfig :: File -> Config
getConfig f =
  let
    header = readHeader f
    basic = readBasicConfig f
    extended = readExtendedConfig (headerVersion header) f
  in Config basic extended
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL

`i.e. List, Maybe(Optional, Option)`

A type constructor F is a Functor if

```
fmap :: (T -> U) -> (F<T> -> F<U>) (= lift)
```

is given. (for arbitrary types T, U)

© 2024. Je Sung Yang all rights reserved.

Functor in PL

Functor laws

1. `fmap (id) ≡ id (where id x = x)`
2. `fmap (f ∘ g) ≡ (fmap f) ∘ (fmap g)`

   Haskell ver. `fmap (f . g) ≡ (fmap f) . (fmap g)`

© 2024. Je Sung Yang all rights reserved.

Functor in PL

How can we implement `fmap` for `Maybe`?

© 2024. Je Sung Yang all rights reserved.

Functor in PL


```
                 f                       fmap f
fmap :: (T -> U) -> (Maybe<T> -> Maybe<U>)
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL

```
            f                   fmap f
fmap :: (T -> U) -> (Maybe<T> -> Maybe<U>)
```

```
fn fmap_f<T, U>(opt_t: Maybe<T>) -> Maybe<U> {
  if (opt_t.is_nothing())
    return Maybe<U>();
  else
    return Maybe<U>(f(opt_t.value()));
}
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL

$$\texttt{fmap} :: \overbrace{(T \rightarrow U)}^{\texttt{f}} \rightarrow \overbrace{(\texttt{Maybe}\langle T\rangle \rightarrow \texttt{Maybe}\langle U\rangle)}^{\texttt{fmap f}}$$

```
fn fmap<T, U>(f: T -> U) -> (Maybe<T> -> Maybe<U>) {
  fn fmap_f<T, U>(opt_t: Maybe<T>) -> Maybe<U> {
    if (opt_t.is_nothing())
      return Maybe<U>();
    else
      return Maybe<U>(f(opt_t.value()));
  }
  return fmap_f;
}
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL

```
               f                         fmap f
fmap :: (T -> U) -> (Maybe<T> -> Maybe<U>)

fmap :: (a -> b) -> Maybe a -> Maybe b
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL


```haskell
data Maybe a = Nothing | Just a
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL


```haskell
data Maybe a = Nothing | Just a

fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f (Just x) = Just (f x)
fmap _ Nothing  = Nothing
```

© 2024. Je Sung Yang all rights reserved.

Functor in PL

```haskell
type Functor :: (* -> *) -> Constraint
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
  {-# MINIMAL fmap #-}
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL

A Functor M is a Monad if

```
fmap :: (T -> U) -> M<T> -> M<U> (= lift)
return :: T -> M<T>               (= unit)
join :: M<M<T>> -> M<T>           (= flat)
```

is given. (for arbitrary types T, U)

© 2024. Je Sung Yang all rights reserved.

Monad in PL

A Functor M is a Monad if

```
fmap :: (T -> U) -> M<T> -> M<U>      (= lift)
return :: T -> M<T>                   (= unit)
bind :: M<T> -> ((T -> M<U>) -> M<U>) (≡ flatMap)
```

Haskell ver. `(>>=) :: M a -> (a -> M b) -> M b`

is given. (for arbitrary types T, U)

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Monad laws

1. `bind m return ≡ m`

   Haskell ver. `m >>= return ≡ m`

2. `bind (return x) f ≡ f x`

   Haskell ver. `return x >>= f ≡ f x`

3. `bind (bind m f) g ≡ bind m (\x -> f x >>= g)`

   Haskell ver. `(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)`

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Semantic of Monad

“A monoid in the category of endofunctors"

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Semantic of Monad

~~“A monoid in the category of endofunctors"~~

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Semantic of Monad

If M is a Monad, M<T> is an **extension of T**, where

- operations on T can also be extended
- the same extension on itself(i.e. M<M<T>>) is
    - meaningless, or
    - logically equial to the original, or
    - can be seen as the original in some aspect

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Semantic of Monad

- Maybe <T>: T or Nothing
- Maybe <Maybe <T>>: T or Nothing or Nothing
  = Maybe <T>

Meaning is preserved! (but type changed)

- return :: T → Maybe <T>
- join :: Maybe <Maybe <T>> → Maybe <T>

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Semantic of Monad

- List <T>: bunch of Ts
- List <List <T>>: bunch of bunches of Ts

  ≃ List <T>

Meaning is preserved! (but type changed)

- return :: T → List <T>
- join :: List <List <T>> → List <T>

© 2024. Je Sung Yang all rights reserved.

Monad in PL

How can we implement `return` for `Maybe`?

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
return :: T -> Maybe<T>
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL

```
return :: T -> Maybe<T>
```

```
fn returnM<T>(t: T) -> Maybe<T> {
  return Maybe<T>(t);
}
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL

How can we implement `bind` for `Maybe`?

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
bind :: Maybe<T> -> ((T -> Maybe<U>) -> Maybe<U>)
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
bind :: (Maybe<T>, (T -> Maybe<U>)) -> Maybe<U>
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
bind :: (Maybe<T>, (T -> Maybe<U>)) -> Maybe<U>
```

```
fn bind<T, U>(m: Maybe<T>, f: T -> Maybe<U>) -> Maybe<U> {
  if (m.is_nothing())
    return Maybe<U>();
  else
    return f(m.value());
}
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
bind :: Maybe<T> -> ((T -> Maybe<U>) -> Maybe<U>)
```

```
fn bind<T, U>(m: Maybe<T>) -> ((T -> Maybe<U>) -> Maybe<U>) {
  fn bind_f(f: T -> Maybe<U>) {
    if (m.is_nothing())
      return Maybe<U>();
    else
      return f(m.value());
  }
  return bind_f;
}
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
return :: T -> Maybe<T>
return :: a -> Maybe a
```

```
bind :: Maybe<T> -> ((T -> Maybe<U>) -> Maybe<U>)
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```haskell
data Maybe a = Nothing | Just a

return :: a -> Maybe a
return x = Just x
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```haskell
data Maybe a = Nothing | Just a

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
(Just x) >>= f = f x
Nothing  >>= _ = Nothing
```

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Any use cases of **Monad** and **`bind`**?

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Any use cases of **Monad** and **`bind`**?

Scenario: Multiple HTTP requests dependent on each other

© 2024. Je Sung Yang all rights reserved.

Monad in PL

Any use cases of **Monad** and `bind`?

Scenario: Multiple HTTP requests dependent on each other

`<Let’s code!>`

© 2024. Je Sung Yang all rights reserved.

Monad in PL


```
f :: a -> Maybe b
g :: b -> Maybe c
```

```
(>>=) :: Maybe b -> (b -> Maybe c) -> Maybe c
f x >>= g :: Maybe c
```

© 2024. Je Sung Yang all rights reserved.

Side Effect in Pure World

Haskell

```haskell
whatIsYourName :: IO ()
whatIsYourName = do
  putStr "What is your name? "
  firstName <- getLine  -- same input!
  lastName <- getLine   -- same input!
  putStrLn ("Hello, " ++ firstName ++ " " ++ lastName)
```

© 2024. Je Sung Yang all rights reserved.

Side Effect in Pure World

1. What is IO in the type signature?
2. How can we perform side effects in a language where side effects are not allowed?

© 2024. Je Sung Yang all rights reserved.

Uniqueness Typing

"A value with a **unique type** is **guaranteed to have at most one reference to it at run-time**, which means that it can safely be updated in-place, reducing the need for memory allocation and garbage collection."

The Idris Tutorial

© 2024. Je Sung Yang all rights reserved.

Uniqueness Typing

Clean

```
module hello
import StdEnv

Start :: *World -> *World
Start world
  # (console, world) = stdio world
  # console = fwrites "Hello, World!\n" console
  # (ok, world) = fclose console world
  | not ok = abort "ERROR: cannot close console\n"
  | otherwise = world
```

© 2024. Je Sung Yang all rights reserved.

IO Monad

Let's **hide "World"** from users!

© 2024. Je Sung Yang all rights reserved.

IO Monad

Haskell

```haskell
type WorldT a = World -> (a, World)

readStrT :: WorldT String
readStrT = readStr

printStrT :: String -> WorldT ()
printStrT str world = ((), printStr str world)

(>>>=) :: WorldT a                -- World -> (a, World)
        -> (a -> WorldT b)        -- a -> World -> (b, World)
        -> WorldT b               -- World -> (b, World)
m >>>= f = uncurry f . m
```

© 2024. Je Sung Yang all rights reserved.

```haskell
whatIsYourPureNameT :: WorldT ()
whatIsYourPureNameT =
  printStrT "What is your name?" >>>= \_ ->
  readStrT                       >>>= \firstName ->
  readStrT                       >>>= \lastName ->
  printStrT ("Hello, " ++ firstName ++ " " ++ lastName)
```

© 2024. Je Sung Yang all rights reserved.