Title: Tutorial 1B: Crash course on Haskell programming

Speakers: Aike Liu

Collection: Mini-Course of Numerical Conformal Bootstrap

Date: April 24, 2023 - 3:30 PM

URL: https://pirsa.org/23040138

A Crash Course on Haskell

Bootstrap Mini Course: Tutorial 1b

Aike Liu
California Institute of Technology

Caltech PERIMETER INSTITUTE

SIMONS FOUNDATION

Outline

- Haskell is a **functional programming language** designed in the late 1980s.
- Compared with object-oriented programming, such as C++ or Python, functional programming is based on concepts of **mathematical functions** rather than objects that encapsulate data.
- Haskell is a **lazy**, **pure** and has a **strong type** system, known for its easiness to write and debug.

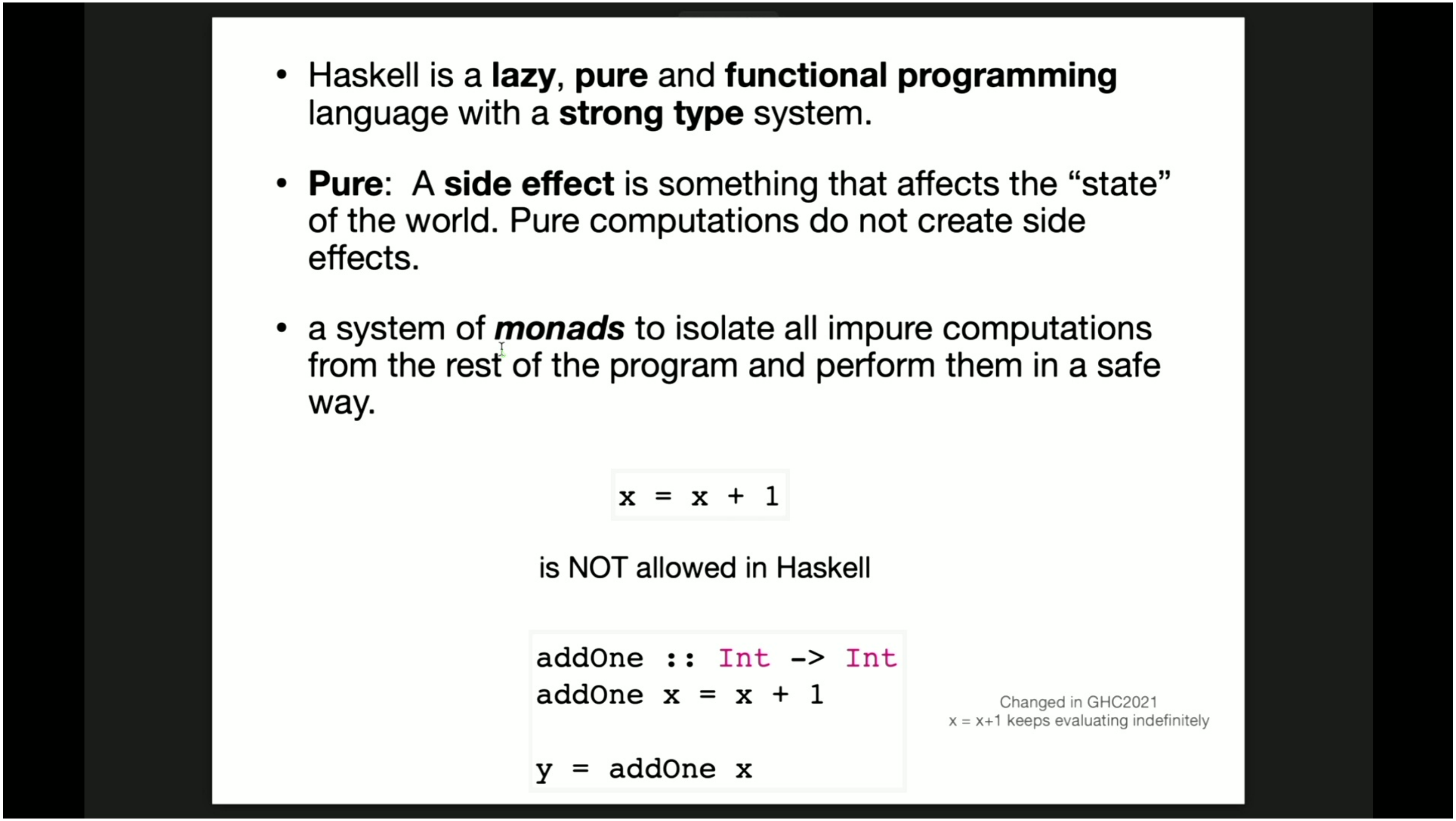
- Haskell is a **lazy**, **pure** and **functional programming** language with a **strong type** system.
- **Lazy**: a value is not evaluated until it is actually needed

```
int myfunc ()
{
  int x = 1 + 2;
  int y = x * 3;
  return y;
}
```

Evaluation happens line by line

```
let x = 1 + 2
    y = x * 3
in y

f x y = if x > 0
          then x
          else y

g = f 1 (1/0)
```

The addition of 1 and 2 is not actually evaluated until it is required to compute **y**


• Haskell is a lazy, pure and functional programming language with a strong type system.
• Pure: A side effect is something that affects the "state" of the world. Pure computations do not create side effects.
• a system of monads to isolate all impure computations from the rest of the program and perform them in a safe way.
x = x + 1
is NOT allowed in Haskell
addOne :: Int -> Int
addOne x = x + 1
y = addOne x
Changed in GHC2021
x = x+1 keeps evaluating indefinitely

- Haskell is a **lazy**, **pure** and **functional programming** language with a **strong type** system.
- **Functional**: composing functions to create programs

```
int myfunc ()
{
  int x = 1 + 2;
  int y = x * 3;
  int z = f(x, y);
  return z;
}
```

```
square :: Int -> Int
square x = x * x

numbers = [1, 2, 3, 4, 5]

map :: (a -> b) -> [a] -> [b]
map square :: [Int] -> [Int]

squares = map square numbers
```

- Haskell is a **lazy**, **pure** and **functional programming** language with a **strong type** system.

- A **strong type** system that catches errors at compile time.

```
square :: Int -> Int
square x = x * x

(5::Int) + 10
(5::Int) + (10::Double)

square 1.5
```

- The Glasgow Haskell Compiler (**GHC**) is a robust, fully-featured, optimizing compiler and interactive environment for Haskell 98.
- The Haskell Tool Stack (**Stack**) is a program for developing Haskell projects.

```
stack new
stack init

stack build
stack install

stack ghc -- Main.hs -o exec

stack ghci
```

- Haskell online tutorials
  - http://learnyouahaskell.com/chapters
  - Yet Another Haskell Tutorial
  - functors, applicative, monads explained

Basic Syntax `start ghci`

- Declaring a variable

```
x = 5
```

- Basic arithmetic operations

```
2 + 3, 4 * 5, 6 - 7, ==, /=
```

- Function declaration

```
addInt :: Int -> Int -> Int
addInt a b = a + b
```

- Naming is case-sensitive
  - values: names start with lower-case letters.
  - types of values: start with upper-case letters.
- Parentheses aren’t required around function arguments
- commenting: - - or {- -}

Conditional Expressions

```
ghci> :l ConditionalExp
```

if/then/else

```
absNum x =
     if x < 0
       then (- x)
       else x
```

cases

```
someMap x = case x of
       0 -> 1
       1 -> 2
       _ -> -1
```

let/in

```
squarePlusOne x =
  let x2 = x * x
  in x2 + 1
```

where

```
anotherSquare x = x2 + 1
  where x2 = x*x
```

Guards

```
comparison x y | x < y = ".."
               | x > y = ".."
               | otherwise = ".."
```

aliu7@login1:~/Boot-mini/examples

aliu7@login1:~/Boot-mini/examples (ssh)

```
   |
40 |              z = x ^ 2
   |                    ^
Ok, one module loaded.
ghci> sq
sqrt               squarePlusOne
ghci> squarePlusOne 5

                   warning:  -Wtype-defaults



26
ghci>
```

aliu7@login1:~/Boot-mini/examples/src (ssh)

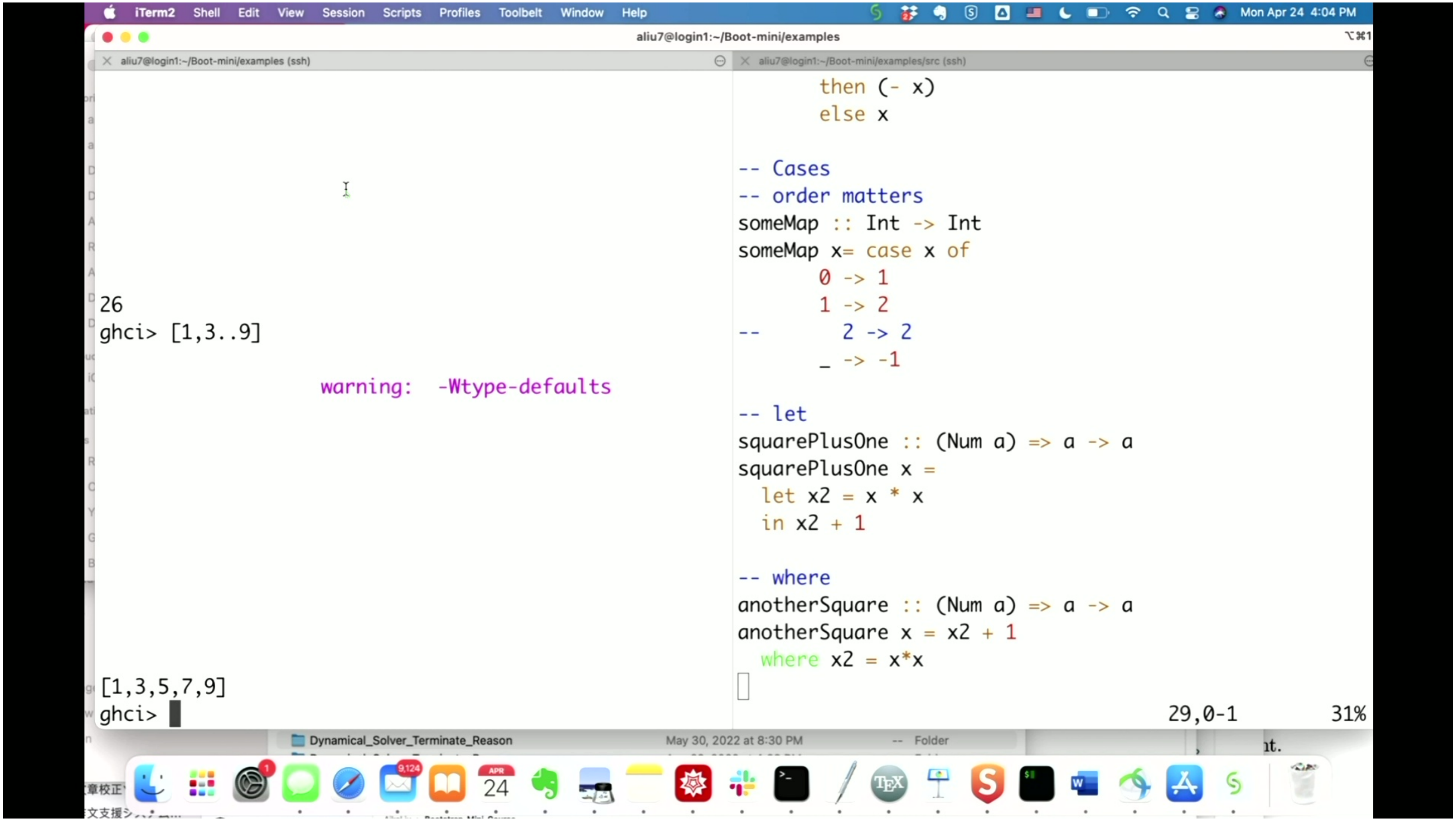
```haskell
       then (- x)
       else x

-- Cases
-- order matters
someMap :: Int -> Int
someMap x= case x of
       0 -> 1
       1 -> 2
--       2 -> 2
       _ -> -1

-- let
squarePlusOne :: (Num a) => a -> a
squarePlusOne x =
  let x2 = x * x
  in x2 + 1

-- where
anotherSquare :: (Num a) => a -> a
anotherSquare x = x2 + 1
  where x2 = x*x

                                       29,0-1        31%
```

Data structures

List: hold an arbitrary number of elements of the same type

:t check types

```
ghci> :t [1,2]
[1,2] :: Num a => [a]
ghci> 0:[1,2] :: [Int]
[0,1,2]
ghci> 1:2:3:[]
[1,2,3]
ghci> [1,2] ++ [3,4] :: [Int]
[1,2,3,4]
ghci> head [1,2,3,4] :: Int
1
ghci> tail [1,2,3,4] :: [Int]
[2,3,4]
ghci> take 2 [1,2,3,4] :: [Int]
[1,2]
ghci> integers = [1..]
```


iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
Mon Apr 24 4:04 PM
aliu7@login1:~/Boot-mini/examples
aliu7@login1:~/Boot-mini/examples (ssh)
aliu7@login1:~/Boot-mini/examples/src (ssh)
26
ghci> [1,3..9]
warning: -Wtype-defaults
[1,3,5,7,9]
ghci>
then (- x)
else x
-- Cases
-- order matters
someMap :: Int -> Int
someMap x= case x of
0 -> 1
1 -> 2
-- 2 -> 2
_ -> -1
-- let
squarePlusOne :: (Num a) => a -> a
squarePlusOne x =
let x2 = x * x
in x2 + 1
-- where
anotherSquare :: (Num a) => a -> a
anotherSquare x = x2 + 1
where x2 = x*x
29,0-1 31%
Dynamical_Solver_Terminate_Reason May 30, 2022 at 8:30 PM -- Folder

Data structures

Tuples: hold a fixed number of elements of possibly different types.

```
ghci> :t (1,2)
(1,2) :: (Num a, Num b) => (a, b)
ghci> :t (1,'a')
(1,'a') :: Num a => (a, Char)
ghci> fst (1::Int,'a')
1
ghci> snd (1::Int,'a')
'a'
```

Functions

(.) is function composition

($) is function application

Partial application: calling a function with fewer arguments than it expects.

```
f . g
f $ x
f . g x ~ f (g x) ~ f $ g x

ghci> :l BasicSyntax
```

Haskell Type System

- **statically-typed**: all variables must have a specific type determined at compile-time.
- **type inference**: compiler can deduce the type of an expression based on its usage.
- **strongly-typed**: once the type of a value is determined, the language will NOT implicitly convert the type.
- **polymorhpic**: functions and values can be defined with type variables.

```
ghci> :t 2::Int
2::Int :: Int
ghci> :t 2
2 :: Num a => a
ghci> x::Int; x=2
ghci> :t x
x :: Int
ghci> (5::Int) + 10
15
ghci> (5::Int) +
(10::Double)
<interactive>:8:13: error:
    • Couldn't match
expected type 'Int' with
actual type 'Double'
ghci> :t map
map :: (a -> b) -> [a] ->
[b]
```

Data Types

Maybe

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

the name of the datatype

constructors

```
firstElement :: [a] -> Maybe a
firstElement []     = Nothing
firstElement (x:xs) = Just x
```

User-defined Types

```
ghci> :l Colors
```

```haskell
data Color = Red
           | Orange
           | Custom Int Int Int
```

the name of the datatype

constructors

Record

```haskell
data AnyColor = AnyColor
              { rNumber :: Int
              , gNumber :: Int
              , bNumber :: Int
              }
```

fields

Type synonyms

```haskell
type List3D a = [(a,a,a)]
```

A **newtype** a datatype with only one constructor and this constructor can have only one argument.

```haskell
newtype MyInt = MyInt Int
```

Data Types

Maybe

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

the name of the datatype

constructors

```
firstElement :: [a] -> Maybe a
firstElement []     = Nothing
firstElement (x:xs) = Just x
```

User-defined Types

```
ghci> :l Colors
```

```haskell
data Color = Red
           | Orange
           | Custom Int Int Int
```

the name of the datatype

constructors

Record

```haskell
data AnyColor = AnyColor
                { rNumber :: Int
                , gNumber :: Int
                , bNumber :: Int
                }
```

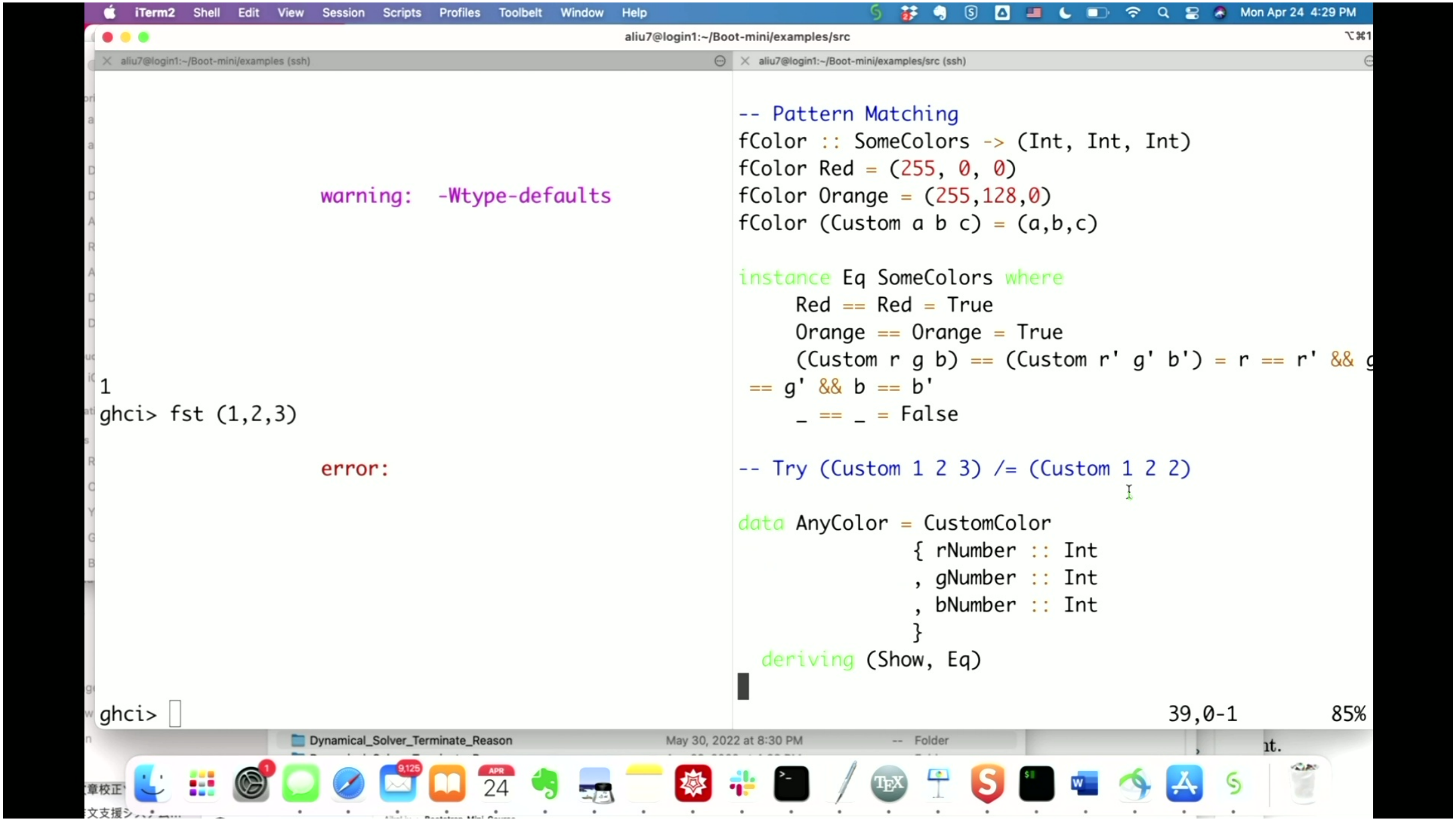
fields

Type synonyms

```haskell
type List3D a = [(a,a,a)]
```

A **newtype** a datatype with only one constructor and this constructor can have only one argument.

```haskell
newtype MyInt = MyInt Int
```


iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
Mon Apr 24 4:29 PM
aliu7@login1:~/Boot-mini/examples/src
aliu7@login1:~/Boot-mini/examples (ssh)
aliu7@login1:~/Boot-mini/examples/src (ssh)
warning: -Wtype-defaults
1
ghci> fst (1,2,3)
error:
ghci>
-- Pattern Matching
fColor :: SomeColors -> (Int, Int, Int)
fColor Red = (255, 0, 0)
fColor Orange = (255,128,0)
fColor (Custom a b c) = (a,b,c)
instance Eq SomeColors where
Red == Red = True
Orange == Orange = True
(Custom r g b) == (Custom r' g' b') = r == r' && g
== g' && b == b'
_ == _ = False
-- Try (Custom 1 2 3) /= (Custom 1 2 2)
data AnyColor = CustomColor
{ rNumber :: Int
, gNumber :: Int
, bNumber :: Int
}
deriving (Show, Eq)
39,0-1 85%
Dynamical_Solver_Terminate_Reason
May 30, 2022 at 8:30 PM
Folder

User-defined Types

```
ghci> :l Colors
```

```haskell
data Color = Red
           | Orange
           | Custom Int Int Int
```

the name of the datatype

constructors

Record

```haskell
data AnyColor = AnyColor
              { rNumber :: Int
              , gNumber :: Int
              , bNumber :: Int
              }
```

fields

Type synonyms

```haskell
type List3D a = [(a,a,a)]
```

A **newtype** a datatype with only one constructor and this constructor can have only one argument.

```haskell
newtype MyInt = MyInt Int
```

Type Class

type class: a set of types that share <u>a common set of methods</u>, implemented differently for different types.

```
ghci>:info Num
```

Eq Class
==, /=

Int, Bool, Char…

Num Class
+, -, *, abs…

Int, Double, Float…

Show Class
show, showPrec…

Int, Bool, Char…

Instance Declaration

To declare a type to be an instance of a class, we need to define the functions that are required by the class for that data type.

```
data MyDataType = ...
instance MyClass MyDataType where
    function1 = ...
    function2 = ...
    ...
```

Eq class has two operations == and /= .

Definition of either == or /= is required.

```
instance Eq Color where
    Red == Red = True
    …
```

```
data Color = Red
           | Orange
           | Custom Int Int Int
  deriving (Eq)
```

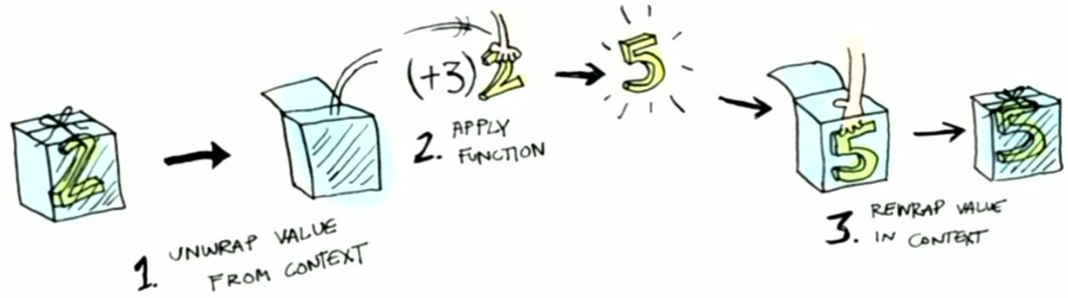
- Basic Syntax
- Conditional Expressions
- Data Structures
- Type and Type Class
- **Functor, Applicative and Monad**

Figure Ref: Functors, Applicatives, And Monads In Pictures

more classes: Functor

```
ghci> :l Functor_Applicative_Monad
```

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```



Here's what is happening behind the scenes when we write `fmap (+3) (Just 2)`:

```haskell
f <$> x = fmap f x
```

infix notation

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

lists are also instances of Functor

```haskell
map :: (a -> b) -> [a] -> [b]
```

Data Types

Maybe

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

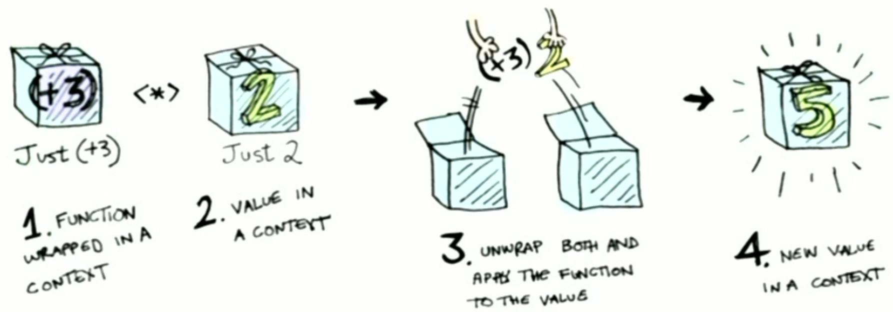
the name of the datatype

constructors

```
firstElement :: [a] -> Maybe a
firstElement []     = Nothing
firstElement (x:xs) = Just x
```

more classes: Applicative

```
class (Functor f) => Applicative f  where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```



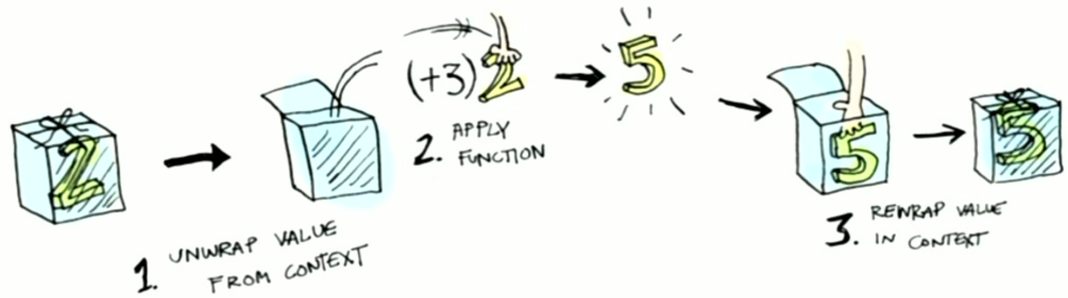

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

```
[(+1),(*100),(*5)] <*> [1,2,3]
```

more classes: Functor

ghci> :l Functor_Applicative_Monad

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```



Here's what is happening behind the scenes when we write `fmap (+3) (Just 2)` :

```
f <$> x = fmap f x
```

infix notation

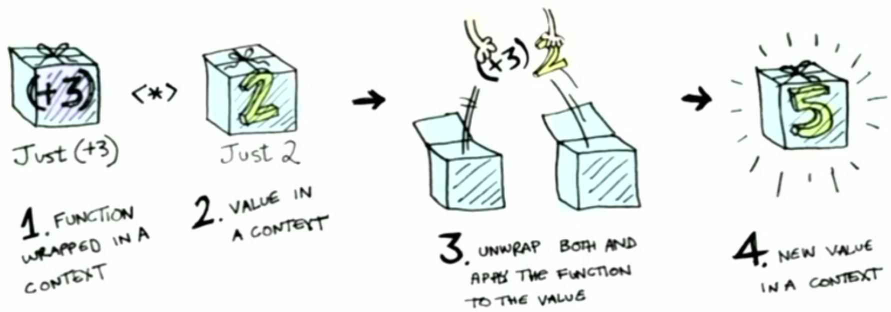
```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

lists are also instances of Functor

```
map :: (a -> b) -> [a] -> [b]
```

more classes: Applicative

```
class (Functor f) => Applicative f  where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```




```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

```
[(+1),(*100),(*5)] <*> [1,2,3]
```

iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help — Mon Apr 24 4:52 PM

aliu7@login1:~/Boot-mini/examples

aliu7@login1:~/Boot-mini/examples (ssh)

```
ghci> mycolor

                    error:

ghci> :l Colors
[1 of 1] Compiling Colors           ( /central/home/al
iu7/Boot-mini/examples/src/Colors.hs, interpreted )


  warning:  -Wmissing-export-lists


  |
1 | module Colors where
  | ^
Ok, one module loaded.
ghci> mycolor
CustomColor {rNumber = 0, gNumber = 0, bNumber = 255}
ghci> CustomColor 1 2 3
CustomColor {rNumber = 1, gNumber = 2, bNumber = 3}
ghci> rNumber mycolor
0
ghci>
```

aliu7@login1:~/Boot-mini/examples/src (ssh)

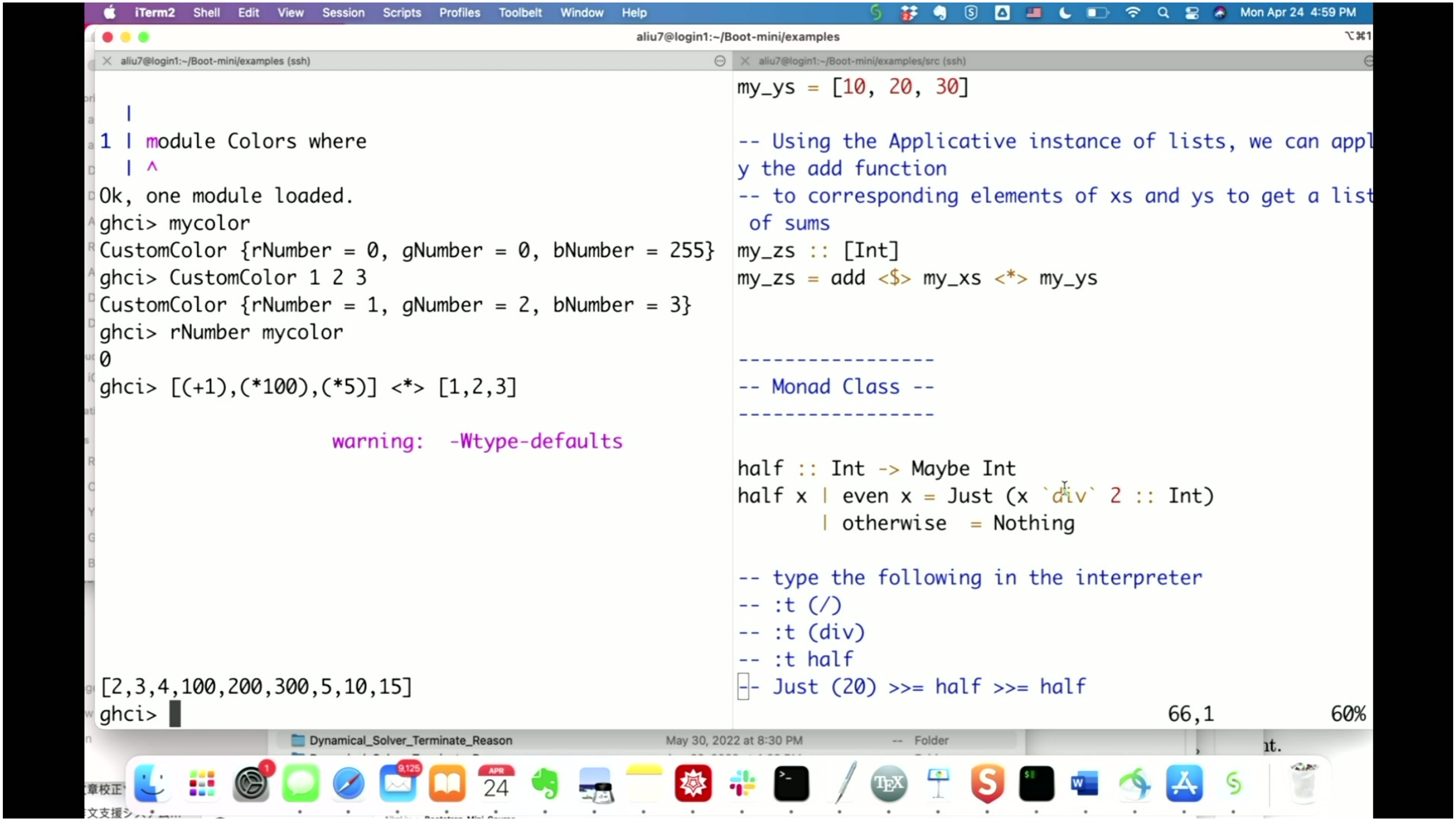
```
-- Type the following in the interpreter
-- pure 1 :: [Int]
-- [(+1),(*100),(*5)] <*> [1,2,3]
-- pure 1 :: Maybe Int
-- Just (+3) <*> Just 1
-- Nothing <*> Just 1
-- Just (+3) <*> Nothing

add :: Int -> Int -> Int
add x y = x + y

my_xs :: [Int]
my_xs = [1, 2, 3]

my_ys :: [Int]
my_ys = [10, 20, 30]

-- Using the Applicative instance of lists, we can appl
y the add function
-- to corresponding elements of xs and ys to get a list
 of sums
my_zs :: [Int]
                                    50,1          38%
```

Dynamical_Solver_Terminate_Reason   May 30, 2022 at 8:30 PM   --   Folder

more classes: Monad

```
class Monad m where
  return :: a -> m a
  fail   :: String -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b

   instance Monad Maybe where
     return a = Just a
     Nothing >>= f = Nothing
     Just x >>= f = f x
     fail _ = Nothing
```

```
Just (20) >>= half >>= half >>= half
```


iTerm2 Shell Edit View Session Scripts Profiles Toolbelt Window Help
Mon Apr 24 4:59 PM
aliu7@login1:~/Boot-mini/examples
aliu7@login1:~/Boot-mini/examples (ssh)
aliu7@login1:~/Boot-mini/examples/src (ssh)
|
1 | module Colors where
| ^
Ok, one module loaded.
ghci> mycolor
CustomColor {rNumber = 0, gNumber = 0, bNumber = 255}
ghci> CustomColor 1 2 3
CustomColor {rNumber = 1, gNumber = 2, bNumber = 3}
ghci> rNumber mycolor
0
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
warning: -Wtype-defaults
[2,3,4,100,200,300,5,10,15]
ghci>
my_ys = [10, 20, 30]
-- Using the Applicative instance of lists, we can apply the add function
-- to corresponding elements of xs and ys to get a list of sums
my_zs :: [Int]
my_zs = add <$> my_xs <*> my_ys

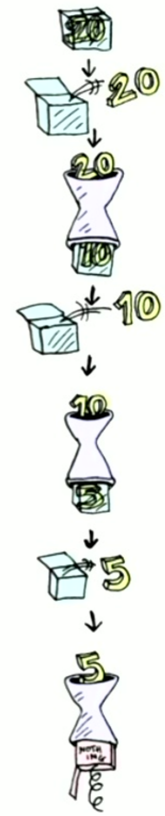
-- Monad Class --
----
half :: Int -> Maybe Int
half x | even x = Just (x `div` 2 :: Int)
| otherwise = Nothing
-- type the following in the interpreter
-- :t (/)
-- :t (div)
-- :t half
-- Just (20) >>= half >>= half
66,1 60%
Dynamical_Solver_Terminate_Reason May 30, 2022 at 8:30 PM -- Folder

more classes: Monad

```
class Monad m where
  return :: a -> m a
  fail   :: String -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b

instance Monad [] where
    return x = [x]
    l >>= f  = concatMap f l
    fail _   = []
```

```
[1,2,3] >>=
   (\x -> [4,5] >>=
      (\y -> return (x,y)))
```

```
[1,2,3] >>= (\x -> [(x,4),(x,5)])
```



Do Notation

```
Just 3 >>=
  (\x -> Just "!" >>=
    (\y -> Just (show x ++ y)))
```

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    return (show x ++ y)
```

```
[1,2,3] >>=
   (\x -> [4,5] >>=
      (\y -> return (x,y)))
```

```
cross l1 l2 = do
   x <- l1
   y <- l2
   return (x,y)
```

Practice

1. Finish coding practice in Exercise.hs
    1. type "stack ghci"
    2. A small questionnaire, call this function

       ghci > doYouEnjoyHaskell

    3. Edit the file Exercise.hs in another terminal window and type :r to reload your file.

       ghci > :r