

# Functional Programming

## IO

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2019

# Referential transparency and substitutivity

## Remember the first class?

- Every variable and expression has just one value  
**referential transparency**
- Every variable can be replaced by its definition  
**substitutivity**

# Referential transparency and substitutivity

## Remember the first class?

- Every variable and expression has just one value  
**referential transparency**
- Every variable can be replaced by its definition  
**substitutivity**

## Enables reasoning

```
-- sequence of function calls does not matter
f () + g () == g () + f ()

-- number of function calls does not matter
f () + f ( ) == 2 * f ()
```

# How does IO fit in?

## Bad example

Suppose we had

```
input :: () -> Integer
```

# How does IO fit in?

## Bad example

Suppose we had

```
input :: () -> Integer
```

- Consider

```
let x = input () in  
x + x
```

# How does IO fit in?

## Bad example

Suppose we had

```
input :: () -> Integer
```

- Consider

```
let x = input () in
x + x
```
- Expect to read one input and use it twice

# How does IO fit in?

## Bad example

Suppose we had

```
input :: () -> Integer
```

- Consider

```
let x = input () in
x + x
```
- Expect to read one input and use it twice
- By substitutivity, this expression must behave like

```
input () + input ()
```

which reads two inputs!

# How does IO fit in?

## Bad example

Suppose we had

```
input :: () -> Integer
```

- Consider

```
let x = input () in
x + x
```
- Expect to read one input and use it twice
- By substitutivity, this expression must behave like

```
input () + input ()
```

which reads two inputs!
- VERY WRONG!!!



# The dilemma

Haskell is a pure language, but IO is a side effect

# The dilemma

Haskell is a pure language, but IO is a side effect

A contradiction?

# The dilemma

Haskell is a pure language, but IO is a side effect

A contradiction?

No!

- Instead of performing IO operations directly, there is an abstract type of **IO instructions**, which get executed lazily by the operating system
- Some instructions (e.g., read from a file) return values, so the abstract IO type is parameterized over their type
- Keep in mind: instructions are just values like any other

# Haskell IO

## The main function

Top-level result of a program is an IO “instruction”.

```
main :: IO ()  
main = undefined
```

- an instruction describes the **effect** of the program
- effect = IO action, imperative state change, ...

# Kinds of instructions

## Primitive instructions

```
-- defined in the Prelude  
putChar    :: Char -> IO ()  
getChar    :: IO Char  
writeFile  :: FileName -> String -> IO ()  
readFile   :: FileName -> IO String
```

and many more

# Kinds of instructions

## Primitive instructions

```
-- defined in the Prelude  
putChar    :: Char -> IO ()  
getChar    :: IO Char  
writeFile  :: FileName -> String -> IO ()  
readFile   :: FileName -> IO String
```

and many more

## No op instruction

```
return :: a -> IO a
```

The IO instruction `return 42` performs no IO, but yields the value 42.

# Combining two instructions

## The bind operator $\gg=$

Intuition: next instruction may depend on the output of the previous one

$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

The instruction  $m \gg= f$

- executes  $m :: IO\ a$  first
- gets its result  $x :: a$
- applies  $f :: a \rightarrow IO\ b$  to the result
- to obtain an instruction  $f\ x :: IO\ b$  that returns  $a\ b$
- and executes this instruction to return  $a\ b$

# Combining two instructions

## The bind operator $>>=$

Intuition: next instruction may depend on the output of the previous one

$(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

The instruction  $m >>= f$

- executes  $m :: IO\ a$  first
- gets its result  $x :: a$
- applies  $f :: a \rightarrow IO\ b$  to the result
- to obtain an instruction  $f\ x :: IO\ b$  that returns a  $b$
- and executes this instruction to return a  $b$

## Example

```
readFiles f1 f2 =  
  readFile f1 >>= \xs1 -> readFile f2
```



## More convenient: do notation

```
copyFile source target =  
  undefined
```

```
doTwice io =  
  undefined
```

```
doNot io =  
  undefined
```

# Translating do notation into >>= operations

- $\text{do } \textit{lastaction}$   
→  
 $\textit{lastaction}$
- $\text{do } \{ x \leftarrow \textit{action1}; \textit{instructions} \}$   
→  
 $\textit{action1} \gg= \backslash x \rightarrow \text{do } \{ \textit{instructions} \}$
- $\text{do } \{ \textit{action1}; \textit{instructions} \}$   
→  
 $\textit{action1} \gg \text{do } \{ \textit{instructions} \}$
- $\text{do } \{ \text{let } \textit{binding}; \textit{instructions} \}$   
→  
 $\text{let } \textit{binding} \text{ in } \text{do } \{ \textit{instructions} \}$

# Instructions vs functions

## Functions

behave the same each time they called

# Instructions vs functions

## Functions

behave the same each time they called

## Instructions

may be interpreted differently each time they are executed, depending on context

# Underlying concept: **Monad**

## What's a monad?

- abstract datatype for instructions that produce values
- built-in combination  $>>=$
- abstracts over different interpretations (computations)

# Underlying concept: **Monad**

## What's a monad?

- abstract datatype for instructions that produce values
- built-in combination  $>>=$
- abstracts over different interpretations (computations)

## IO is a special case of a monad

- one very useful application for monad
- built into Haskell
- but there's more to the concept
- many more instances to come!

# Hands-on task

Define a function

```
sortFile :: FilePath -> FilePath -> IO ()

-- sortFile inFile outFile
-- reads inFile, sorts its lines, and writes the result to outFile

-- recall
-- sort :: Ord a => [a] -> [a]
-- lines :: String -> [String]
-- unlines :: [String] -> String
```

# Utilities

```
sequence :: [IO a] -> IO [a]
```

```
sequence_ :: [IO a] -> IO ()
```



## Another hands-on task

Define a function

```
printTable :: [String] -> IO ()  
  
{-  
printTable ["New York", "Rio", "Tokio"]  
outputs  
1: New York  
2: Rio  
3: Tokyo  
-}
```

## First meeting with monads

- abstract data type of instructions returning results
- next instruction can depend on previous result
- instructions are just values
- basis for Haskell's standard IO