

Functional Programming

IO

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

WS 2022/23

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

```
1  -- sequence of function calls does not matter
2  f () + g () == g () + f ()
3  -- number of function calls does not matter
4  f () + f ( ) == 2 * f ()
```

How does IO fit in?

Bad example

Suppose we had

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

How does IO fit in?

Bad example

Suppose we had

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

- Consider

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

How does IO fit in?

Bad example

Suppose we had

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

- Consider

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

- Expect to read one input and use it twice

How does IO fit in?

Bad example

Suppose we had

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

- Consider

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

- Expect to read one input and use it twice
- By substitutivity, this expression must behave like

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

which reads two inputs!

How does IO fit in?

Bad example

Suppose we had

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

- Consider

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

- Expect to read one input and use it twice
- By substitutivity, this expression must behave like

```
1 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”.

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

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

Kinds of instructions

Primitive instructions

```
1  -- defined in the Prelude
2  putChar :: Char -> IO ()
3  getChar :: IO Char
4  writeFile :: FileName -> String -> IO ()
5  readFile :: FileName -> IO String
```

and many more

Kinds of instructions

Primitive instructions

```
1  -- defined in the Prelude
2  putChar :: Char -> IO ()
3  getChar :: IO Char
4  writeFile :: FileName -> String -> IO ()
5  readFile :: FileName -> IO String
```

and many more

No op instruction

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

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

Combining two instructions

The bind operator $>>=$

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

1 $(>>=) :: \mathbf{IO} \ a \rightarrow (a \rightarrow \mathbf{IO} \ b) \rightarrow \mathbf{IO} \ b$

The instruction $m \gg= f$

- executes $m :: \mathbf{IO} \ a$ first
- gets its result $x :: a$
- applies $f :: a \rightarrow \mathbf{IO} \ b$ to the result
- to obtain an instruction $f \ x :: \mathbf{IO} \ b$ 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

1 $(>>=) :: \mathbf{IO} \ a \rightarrow (a \rightarrow \mathbf{IO} \ b) \rightarrow \mathbf{IO} \ b$

The instruction $m \gg= f$

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

Example

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

More convenient: do notation

```
1 copyFile source target =  
2   undefined  
3  
4 doTwice io =  
5   undefined  
6  
7 doNot io =  
8   undefined
```

Translating do notation into >>= operations

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

Instructions vs functions

Functions

produce the same result each time they are called

Instructions vs functions

Functions

produce the same result each time they are called

Instructions

- are interpreted each time they are executed,
- the result depends on the context
- may be different each time

Underlying concept: **Monad**

What's a monad? (first approximation)

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

Underlying concept: **Monad**

What's a monad? (first approximation)

- 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

```
1 sortFile :: FilePath -> FilePath -> IO ()
2
3 -- sortFile inFile outFile
4 -- reads inFile, sorts its lines, and writes the result to outFile
5
6 -- recall
7 -- sort :: Ord a => [a] -> [a]
8 -- lines :: String -> [String]
9 -- unlines :: [String] -> String
```

Utilities

```
1 sequence :: [IO a] -> IO [a]
2 sequence_ :: [IO a] -> IO ()
```


Another hands-on task

Define a function

```
1 printTable :: [String] -> IO ()
2
3 {-
4  printTable ["New York", "Rio", "Tokio"]
5  outputs
6  1: New York
7  2: Rio
8  3: Tokio
9  -}
```

First encounter with monads

- A monad is an abstract data type of instructions returning results.
- The next instruction can depend on previous results.
- Instructions are just values.
- Haskell's IO operations are instructions of the IO monad.