Functional Programming

Part I

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

Uder, 30.05.2019

Plan

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and addresses

What is Functional Programming?

A different approach to programming

Functions and values

rather than

Assignments and addresses

It will make you a better programmer

Functional vs Imperative Programming: Variables

Functional (Haskell)

```haskell
x :: Int
x = 5
```

- Variable `x` has value 5 forever
- It's ok to replace `x` by 5 whenever needed

Functional vs Imperative Programming: Variables

Functional (Haskell)

```haskell
x :: Int
x = 5
```

- Variable x has value 5 forever
- It's ok to replace x by 5 whenever needed

Imperative (Java / C)

```java
int x = 5;
...
x = x+1;
```

- Variable x can change its content over time
- Current value of x needs to be looked up in the store

Functional vs Imperative Programming: Functions

Functional (Haskell)

```
f :: Int -> Int -> Int
f x y = 2*x + y

f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

Functional vs Imperative Programming: Functions

Functional (Haskell)

```
f :: Int -> Int -> Int
f x y = 2*x + y

f 42 16 // always 100
```

Return value of a function **only** depends on its inputs

Imperative (Java)

```
boolean flag;
static int f (int x, int y) {
  return flag ? 2*x + y , 2*x - y;
}

int z = f (42, 16); // who knows?
```

Return value depends on non-local variable `flag`

Functional vs Imperative Programming: Laziness

Haskell

```
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

Functional vs Imperative Programming: Laziness

Haskell

```
x = expensiveComputation
g anotherExpensiveComputation
```

- The expensive computation will only happen if `x` is ever used.
- Another expensive computation will only happen if `g` uses its argument.

Java

```
int x = expensiveComputation;
g (anotherExpensiveComputation)
```

- Both expensive computations will happen anyway.
- Laziness can be simulated, but it's complex!

Many features that make programs more concise

- Pattern Matching
- Higher-order functions
- Algebraic datatypes
- Polymorphic types
- Parametric overloading
- Type inference
- Monads & friends (for IO, concurrency, . . .)
- Comprehensions
- Metaprogramming
- Domain specific languages
- . . .

Plan

Predefined Types

Every Haskell value has a type

`Bool`	— **True** :: **Bool**, **False** :: **Bool**
`Char`	— 'x' :: **Char**, '?' :: **Char**, ...
`Double`, `Float`	— 3.14 :: **Double**
`Integer`	— 4711 :: **Integer**
`Int`	— machine integers ($\geq$ 30 bits signed integer)
`()`	— the unit type, single value () :: ()
a −> b	— function types
(a, b)	— tuple types
[a]	— list types
String	— "xyz":: **String**, ...

Functions

Examples.hs

```haskell
dollarRate = 1.3671

-- |convert EUR to USD
usd euros = euros * dollarRate
```

- dollarRate defines a constant
- usd is a function
- Its type **Double** -> **Double** is inferred by the Haskell compiler
- To compute, a function call usd arg is replaced by the right hand side of its definition

  usd arg $\longrightarrow$ arg $*$ dollarRate $\longrightarrow$ arg $*$ 1.3671 $\longrightarrow$ ...

Recursive functions

Compute `x^n` without using the built-in operator

```
-- compute x to n-th power
power x n | n == 0 = 1
power x n | n > 0 = x * power x (n - 1)
```

- defined using guarded equations
- computation chooses the first equation such that the guard is true
  - power 5 0 $\rightarrow$ 1
  - power 5 2 $\rightarrow$ 5 $*$ power 5 1 $\rightarrow$ 5 $*$ (5 $*$ power 5 0) $\rightarrow$ 5 $*$ (5 $*$ 1)

Tuples

```
-- example tuples
examplePair :: (Double, Bool) -- Double x Bool
examplePair = (3.14, False)

exampleTriple :: (Bool, Int, String) -- Bool x Int x String
exampleTriple = (False, 42, "Answer")

exampleFunction :: (Bool, Int, String) -> Bool
exampleFunction (b, i, s) = not b && length s < i
```

Summary

- Syntax for tuple type like syntax for tuple values
- Tuples are **immutable**: in fact, **all values are**!
  Once a value is defined it cannot change!

Typing for Tuples

Typing Rule

$$\frac{e_1 :: t_1 \qquad e_2 :: t_2 \qquad \dots \qquad e_n :: t_n}{(e_1, \dots, e_n) :: (t_1, \dots, t_n)}\ \textsc{Tuple}$$

If

- $e_1, \dots, e_n$ are Haskell expressions
- $t_1, \dots, t_n$ are their respective types
- Then the tuple expression $(e_1, \dots, e_n)$ has the tuple type $(t_1, \dots, t_n)$.

Lists

- The "duct tape" of functional programming
- Collections of things of the same type
- For any type a, [a] is the type of lists with elements of type a
  e.g. [**Bool**] is the type of lists of **Bool**
- Syntax for list type like syntax for list values
- Lists are **immutable**: once a list value is defined it cannot change!

Constructing lists

The values of type [a] are ...

- either [], the empty list
- or x:xs where x has type a and xs has type [a]
  ":" is pronounced "cons"
- [] and (:) are the **list constructors**

Constructing lists

The values of type [a] are ...

- either [], the empty list
- or x:xs where x has type a and xs has type [a]
  ":" is pronounced "cons"
- [] and (:) are the **list constructors**

Typing Rules for Lists

$$\textsc{Nil}\quad [] :: [t] \qquad\qquad \textsc{Cons}\quad \frac{e_1 :: t \qquad e_2 :: [t]}{(e_1 : e_2) :: [t]}$$

- The empty list can serve as a list of any type t
- If there is some t such that e_1 has type t and e_2 has type $[t]$, then $(e_1 : e_2)$ has type $[t]$.

List shorthands

Equivalent ways of writing a list

1:(2:(3:[ ]))	—	standard, fully parenthesized
1:2:3:[ ]	—	(:) associates to the right
[1,2,3]	—	bracketed notation

Typing Lists

Quiz Time

Which of the following expressions have type [**Bool**]?

```
[ ]
True : [ ]
True : False
False : (False : [ ])
(False : False) : [ ]
(False : []) : [ ]
(True : (False : (True : []))) : (False:[]):[ ]
```

Plan

Functions on lists

Definition by **pattern matching**

```
-- double every element of a list of integers: double [3,6,12] == [6,12,24]
doubles :: [Integer] -> [Integer]
doubles [] = []
doubles (x:xs) = (2 * x) : doubles xs
```

Functions on lists

Definition by **pattern matching**

```
-- double every element of a list of integers: double [3,6,12] == [6,12,24]
doubles :: [Integer] -> [Integer]
doubles [] = []
doubles (x:xs) = (2 * x) : doubles xs
```

Argument value is checked against patterns

- patterns contain constructors ([] and :) and variables
- patterns are checked in sequence; matching equation is chosen
- constructors are checked against argument value
- variables are bound to the values in corresponding position in the argument

Functions on lists

Definition by **pattern matching**

```haskell
-- double every element of a list of integers: double [3,6,12] == [6,12,24]
doubles :: [Integer] -> [Integer]
doubles [] = []
doubles (x:xs) = (2 * x) : doubles xs
```

Argument value is checked against patterns

- patterns contain constructors ([] and :) and variables
- patterns are checked in sequence; matching equation is chosen
- constructors are checked against argument value
- variables are bound to the values in corresponding position in the argument

Example evaluation

doubles (3 : 6 : 12 : []) → (2 ∗ 3) : doubles (6 : 12 : []) →
6 : (2 ∗ 6) : doubles (12 : []) → . . .

Higher-order functions

Definition

A higher-order function takes a function argument or returns it as a result.

Higher-order functions

Definition

A higher-order function takes a function argument or returns it as a result.

Example

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

- twice takes a function f and an argument and applies f two times to the argument.
- (+1) is the function that adds one to its argument
- What is twice (+1)?

Higher-order functions

Definition

A higher-order function takes a function argument or returns it as a result.

Example

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

- twice takes a function f and an argument and applies f two times to the argument.
- (+1) is the function that adds one to its argument
- What is twice (+1)?
- That's the function that adds two to its argument!

Higher-order functions

Definition

A higher-order function takes a function argument or returns it as a result.

Example

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

- twice takes a function f and an argument and applies f two times to the argument.
- (+1) is the function that adds one to its argument
- What is twice (+1)?
- That's the function that adds two to its argument!
- But what about twice twice (+1)?

Higher-order functions

Definition

A higher-order function takes a function argument or returns it as a result.

Example

```
twice :: (a -> a) -> (a -> a)
twice f x = f (f x)
```

- twice takes a function f and an argument and applies f two times to the argument.
- (+1) is the function that adds one to its argument
- What is twice (+1)?
- That's the function that adds two to its argument!
- But what about twice twice (+1)?
- Adds four!

map: Apply Function to Every Element of a List

Definition

```haskell
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

(map is in the standard Prelude - no need to define it)

map: Apply Function to Every Element of a List

Definition

```
-- map f [x1, x2, ..., xn] = [f x1, f x2, ..., fn]
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

(map is in the standard Prelude - no need to define it)

Defining doubles using map

```
doubles xs = map (*2) xs
```

foldr: Reduce a List

Abstracting over value and combining function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op e [] = e
foldr op e (x:xs) = x ‘op‘ foldr’ op e xs
```

where

- `e :: b` is a value replacing the empty list
- `op :: a -> b -> b` is a combining function for list element and recursive call on the rest

foldr: Reduce a List

Abstracting over value and combining function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op e [] = e
foldr op e (x:xs) = x ‘op‘ foldr’ op e xs
```

where

- `e :: b` is a value replacing the empty list
- `op :: a -> b -> b` is a combining function for list element and recursive call on the rest

Example: many functions become one-liners with foldr

```
sum xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
```

foldr: Reduce a List

Abstracting over value and combining function

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op e [] = e
foldr op e (x:xs) = x `op` foldr op e xs
```

where

- `e :: b` is a value replacing the empty list
- `op :: a -> b -> b` is a combining function for list element and recursive call on the rest

Example: many functions become one-liners with foldr

```
sum xs = foldr (+) 0 xs
product xs = foldr (*) 1 xs
```

Also known as reduce

map + reduce = MapReduce

Plan

Referential transparency and substitutivity

Recall the beginning

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

Referential transparency and substitutivity

Recall the beginning

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

Referential transparency 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 an operation that reads a number from the terminal

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

How does IO fit in?

Bad example

Suppose we had an operation that reads a number from the terminal

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

- Consider

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

How does IO fit in?

Bad example

Suppose we had an operation that reads a number from the terminal

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

- Consider

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

- Expectation: read one input and use it twice

How does IO fit in?

Bad example

Suppose we had an operation that reads a number from the terminal

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

- Consider

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

- Expectation: 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 an operation that reads a number from the terminal

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

- Consider

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

- Expectation: 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 I/O is a side effect

The dilemma

Haskell is a pure language, but I/O is a side effect

A contradiction?

The dilemma

Haskell is a pure language, but I/O 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 I/O

The main function

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

```
main :: IO ()
main = putStrLn "Hello World!"
```

- an instruction describes the **effect** of the program
- effect = IO action, imperative state change, ...
- Here: print a string on the terminal

Kinds of instructions

Primitive instructions

```
-- predefined
putChar :: Char -> IO ()
getChar :: IO Char
writeFile :: FileName -> String -> IO ()
readFile :: FileName -> IO String
```

and many more

Kinds of instructions

Primitive instructions

```
-- predefined
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 >>=

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

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

The instruction `m >>= f`

- first executes `m :: IO a`
- gets its result `x :: a`
- applies `f :: a -> 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 -> (a -> IO b) -> IO b
```

The instruction `m >>= f`

- first executes `m :: IO a`
- gets its result `x :: a`
- applies `f :: a -> 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
```

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 type for instructions that produce values
- built-in operators combination $>>=$ and no-op **return**
- abstracts over different interpretations (computations)

Underlying concept: **Monad**

What's a monad?

- abstract type for instructions that produce values
- built-in operators combination $>>=$ and no-op **return**
- abstracts over different interpretations (computations)

IO is a special case of a monad

- one very useful application for monads
- built into Haskell
- but there's more to the concept!

Intermezzo: Quicksort!

Quicksort implementation

```
qsort [] = []
qsort (piv:xs) = qsort (filter (<= piv) xs) ++ piv : qsort (filter (> piv) xs)
```

- (**filter** (<= piv) xs) — elements of xs less than or equal to piv
- predefined, but . . .

Intermezzo: Quicksort!

Quicksort implementation

```
qsort [] = []
qsort (piv:xs) = qsort (filter (<= piv) xs) ++ piv : qsort (filter (> piv) xs)
```

- (**filter** (<= piv) xs) — elements of xs less than or equal to piv
- predefined, but ...

filter implemented with foldr

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr op []
  where op x | p x = (x :)
             | otherwise = id
```

Intermezzo: Quicksort!

Quicksort implementation

```
qsort [] = []
qsort (piv:xs) = qsort (filter (<= piv) xs) ++ piv : qsort (filter (> piv) xs)
```

- (**filter** (<= piv) xs) — elements of xs less than or equal to piv
- predefined, but . . .

filter implemented with foldr

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr op []
  where op x | p x = (x :)
             | otherwise = id
```

filter implemented with foldr

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = foldr op [] xs
  where op x xs | p x = (x :) xs
                | otherwise = id xs
```

Plan

Algebraic Datatypes

- Signature facility for defining datatypes in functional languages
- Originally introduced in the language Hope in the 1970s
- Describe a new datatype by declaring its constructor functions and giving the type of their arguments
- Constructor functions do not evaluate their arguments (like the list constructors [] and :)
- Constructors can be used for pattern matching on the left side of function definitions

Example scenario

Model a card game

- represent the game items!
- define game logic on the representations!

Data model for card games

Description

- A card has a **Suit** and a **Rank**
- A card beats another card if it has the same suit, but higher rank

Data model for card games

Description

- A card has a **Suit** and a **Rank**
- A card beats another card if it has the same suit, but higher rank

A card has a Suit

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

Data model for card games

Description

- A card has a **Suit** and a **Rank**
- A card beats another card if it has the same suit, but higher rank

A card has a Suit

```
data Suit = Spades | Hearts | Diamonds | Clubs
```

Explanation

- new type consisting of four values
- `Suit`: the name of the new type
- `Spades`, `Hearts`, `...`: the names of its **constructors**.
- Type and constructor names must be capitalized

More data

A card has a suit and a **rank**:

```
data Rank = Numeric Integer | Jack | Queen | King | Ace
```

The constructor `Numeric` is different: it takes an argument.

```
Main> :t Numeric
Numeric :: Integer -> Rank
```

Defining a function on Rank

Ordering ranks by pattern matching

```
-- rankBeats r1 r2 returns True, if r1 beats r2
rankBeats :: Rank -> Rank -> Bool
rankBeats _ Ace = False
rankBeats Ace _ = True
rankBeats _ King = False
rankBeats King _ = True
rankBeats _ Queen = False
rankBeats Queen _ = True
rankBeats _ Jack = False
rankBeats Jack _ = True
rankBeats (Numeric n1) (Numeric n2) = n1 > n2
-- ^^ pattern match on constructor
-- yields its argument
```

Example

```
rankBeats Queen Jack == True
rankBeats Queen King == False
```

Datatypes can be recursive

Binary Trees

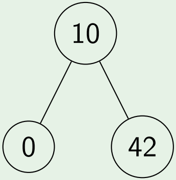
A binary tree is either empty or a node with a data element and two subtrees.

```
data BTree a = Leaf | Node (BTree a) a (BTree a)
```

- Parameterised over the type a of elements
- Recursive datatype definition

Example

```
bt :: BTree Int
bt = Node (Node Leaf 0 Leaf)
          10
          (Node Leaf 42 Leaf)
```



Plan

Parametric polymorphism

Most higher-order functions are polymorphic

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

- a and b are type variables
- the functions can be used for **any** types instantiated for a and b
- they work uniformly for all these instances

Haskell integrates overloading with polymorphism

Restricted polymorphism

- Some functions work on parametric types, but are restricted to specific instances
- Types contain type variables and **constraints** like **Eq** a, **Ord** a etc

Haskell integrates overloading with polymorphism

Restricted polymorphism

- Some functions work on parametric types, but are restricted to specific instances
- Types contain type variables and **constraints** like **Eq** a, **Ord** a etc

Examples

```
-- elem x xs : is x an element of list xs?
-- type a must support equality
elem :: Eq a => a -> [a] -> Bool
-- insert x xs : insert x into sorted list xs
-- type a must support comparison
insert :: Ord a => a -> [a] -> [a]
-- square x : compute the square of x
-- type a supports numeric operations
square :: Num a => a -> a
```

Type classes

- Each constraint mentions a **type class**
  like `Eq`, `Ord`, `Num`, ...
- A type class is a set of types that support the same operations
  e.g. members of `Eq` must support == and /=
- Type classes form a hierarchy
  e.g. `Eq a => Ord a`
  "must belong to **Eq** before you belong to **Ord**"
- Many classes are predefined, but you can roll your own

Classes and Instances

- A class declaration **only** specifies a signature (i.e., the class members and their types)

```
class Num a where
  (+), (*), (-) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a
```

- A separate instance declaration specifies that a type belongs to a class by giving definitions for all class members

```
instance Num Int where ...
instance Num Integer where ...
instance Num Double where ...
instance Num Float where ...
```

Example: Equality

The type class Eq

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y) -- default definition
```

An instance must only provide (==).

Example: Equality

The type class Eq

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y) -- default definition
```

An instance must only provide (==).

Question

Does equality make sense at every type?

Defining Eq for pairs

When are two pairs equal?

Defining Eq for pairs

When are two pairs equal?

Solution

```
instance (Eq a, Eq b) => Eq (a, b) where
   (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Defining Eq for pairs

When are two pairs equal?

Solution

```
instance (Eq a, Eq b) => Eq (a, b) where
  (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Is this definition recursive?

Defining Eq for pairs

When are two pairs equal?

Solution

```
instance (Eq a, Eq b) => Eq (a, b) where
  (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2
```

Is this definition recursive?

NO!

Defining Eq for Suit

Manual definition

```haskell
data Suit = Spades | Hearts | Diamonds | Clubs

instance Eq Suit where
  Spades == Spades = True
  Hearts == Hearts = True
  Diamonds == Diamonds = True
  Clubs == Clubs = True
  _ == _ = False // any other combination
```

Defining Eq for Suit

Manual definition

```
data Suit = Spades | Hearts | Diamonds | Clubs

instance Eq Suit where
  Spades == Spades = True
  Hearts == Hearts = True
  Diamonds == Diamonds = True
  Clubs == Clubs = True
  _ == _ = False // any other combination
```

Boring to write boilerplate code ...

Defining Eq for Suit

Manual definition

```
data Suit = Spades | Hearts | Diamonds | Clubs

instance Eq Suit where
  Spades == Spades = True
  Hearts == Hearts = True
  Diamonds == Diamonds = True
  Clubs == Clubs = True
  _ == _ = False // any other combination
```

Boring to write boilerplate code ...

Automatic derivation of Eq instance

```
data Suit = Spades | Hearts | Diamonds | Clubs
    deriving (Eq)
```

Abstract data types using type classes

- Abstract data types separate the specification of the interface from the implementation

Interface

```
class Stack s where
  push :: s a -> a -> s a
  pop :: s a -> s a
  top :: s a -> a
  init :: s a
```

Implementation

```
instance Stack [] where
  push = flip (:)
  pop = tail
  top = head
  init = []
```

Plan

Search trees

Definition

A search tree is a binary tree such that at each node Node l x r the element x is greater than every element in the left subtree l and x is less than every element in the right subtree r.

Insert value into a search tree

```
insert :: Ord a => a -> BTree a -> BTree a
insert x Leaf = Node Leaf x Leaf
insert x node@(Node l y r)
      | x < y = Node (insert x l) y r
      | x > y = Node l y (insert x r)
      | otherwise = node
```

- **Ord** a =>... means that the type a must admit comparison

Expression trees

Arithmetic expressions comprising constants and binary operators

Definition

```
data Term
  = Con Integer
  | Bin Term Op Term

data Op
  = Add
  | Sub
  | Mul
  | Div
```

Expression trees

Arithmetic expressions comprising constants and binary operators

Definition

```haskell
data Term
  = Con Integer
  | Bin Term Op Term

data Op
  = Add
  | Sub
  | Mul
  | Div
```

Interpretation

```haskell
eval :: Term -> Integer
eval (Con n) = n
eval (Bin t op u) = sys op (eval t) (eval u)

sys :: Op -> (Integer -> Integer -> Integer)
sys Add = (+)
sys Sub = (-)
sys Mul = (*)
sys Div = div
```

Expression trees

Arithmetic expressions comprising constants and binary operators

Definition

```haskell
data Term
  = Con Integer
  | Bin Term Op Term

data Op
  = Add
  | Sub
  | Mul
  | Div
```

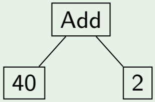
Interpretation

```haskell
eval :: Term -> Integer
eval (Con n) = n
eval (Bin t op u) = sys op (eval t) (eval u)

sys :: Op -> (Integer -> Integer -> Integer)
sys Add = (+)
sys Sub = (-)
sys Mul = (*)
sys Div = div
```

Example

```haskell
term = Bin (Con 40) Add (Con 2)
```



Plan

Extending the interpreter

Possible extensions

- Error handling
- Counting evaluation steps
- Variables, state
- Output

... but without changing the structure of the interpreter!

Extending the interpreter

Possible extensions

- Error handling
- Counting evaluation steps
- Variables, state
- Output

Effects!!!

. . . but without changing the structure of the interpreter!

Extending the interpreter

Possible extensions

- Error handling
- Counting evaluation steps
- Variables, state
- Output

Effects!!!

... but without changing the structure of the interpreter!

Monads to the rescue

- In each case, we can phrase the extension as a type of commands, as we've seen in the case of I/O.
- Classical FP approach to incorporate effects

Commands for error handling

Interface to error handling: three operations

- raise an error message
- combine computations with error handling (standard bind operation)
- return a value without an error (standard return operation)

Commands for error handling

Interface to error handling: three operations

- raise an error message
- combine computations with error handling (standard bind operation)
- return a value without an error (standard return operation)

Datatype for error signaling

```
data Exception a = Raise String
                 | Return a
```

- error messages are represented by strings
- a is the type of normally returned values

The Monad Interface

The type class Monad

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

- **NEW:** `m` is a type variable that can stand for `IO`, `[]`, and other **type constructors**. Like Exception.
- Types of bind and return abstracted over **IO**
- **do** notation: instead of `m1 >>= \x -> m2` write

```haskell
do x <- m1
   m2
```

Monadic interpretation

Error signaling as a monad

```
instance Monad Exception where
  return a = Return a
  m >>= f = case m of
              Raise s -> Raise s
              Return v -> f v
  fail s = Raise s
```

Monadic interpretation

Error signaling as a monad

```
instance Monad Exception where
  return a = Return a
  m >>= f = case m of
              Raise s -> Raise s
              Return v -> f v
  fail s = Raise s
```

Monadic interpretation

```
eval :: Term -> Exception Integer
eval (Con n) = return n
eval (Bin t op u) = do
  v <- eval t
  w <- eval u
  if (op == Div && w == 0)
    then fail "div by zero"
    else return (sys op v w)
```

Why would you write an interpreter in this style?

- Easy modification to support other effects / monads.
- It's possible to combine monads modularly to support combinations of effects.

A monad for counting

The state monad

```
data ST s a = ST (s -> (a, s))
exST (ST sas) = sas

instance Monad (ST s) where
  return a = ST (\s -> (a, s))
  m >>= f = ST (\s -> let (a, s') = exST m s in exST (f a) s')

-- primitive for reading and writing the state
get :: ST s s
get = ST (\s -> (s, s))

put :: s -> ST s ()
put s = ST (const ((), s))

type Count a = ST Integer a

incr :: Count ()
incr = get >>= \i -> put (i+1)
```

Monadic interpreter with reduction count

Implementation

Evaluation

```
eval :: Term -> Count Integer
eval (Con n) = return n
eval (Bin t op u) = do
  v <- eval t
  w <- eval u
  incr
  return (sys op v w)
```

Typical monads

Already used

- Identity monad
- Exception monad
- State monad
- I/O monad

Others

- Writer monad: supports an output operation
  Monoid w =>**Monad** (Writer w) where **data** Writer w a = Writer a w
- Maybe monad: computation that may or may not return a result
- List monad: Multiple results / nondeterminism, backtracking

Modular computations

- Types given to eval restrict to a single monad
- Better to just state requirements on the monad

Modular evaluation

```
incr :: MonadState Integer m => m ()
incr = get >>= \i -> put (i+1)

eval :: (MonadState Integer m, MonadError String m)
      => Term -> m Integer
eval (Con n) = return n
eval (Bin t op u) = do
  v <- eval t
  w <- eval u
  incr
  if (op == Div && w == 0)
    then throwError "div by zero"
    else return (sys op v w)
```

Conclusion

Still to come

- Yet more typing features
  - polymorphic functions as parameters
  - type-level computation (associated classes and type families)
  - dependent types
- Concurrency, parallelism, GPU programming
- Metaprogramming
- Domain specific languages
- ...

Industrial users of Haskell

https://wiki.haskell.org/Haskell_in_industry

References

- Paper by the original developers of Haskell in the conference on History of Programming Languages (HOPL III): http://dl.acm.org/citation.cfm?id=1238856
- The Haskell home page: `http://www.haskell.org`
- Haskell libraries repository: `https://hackage.haskell.org/`
- Haskell Tool Stack: `https://docs.haskellstack.org/en/stable/README/`
- The complete lecture materials `https://github.com/proglang/FunctionalProgramming`