Introduction

Formal Methods

- Application of theoretical computer science fundamentals
- Logic calculi
- Formal languages
- Automata theory
- Program semantics
- Type systems
- Algebraic data types

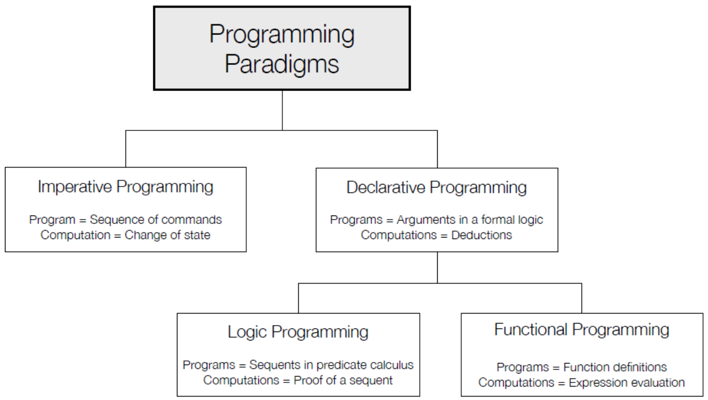
Formal Language

- Set of strings of symbols
- Constrained by specific rules
- Programming languages
- Usage: Specify, invent, transform, analyse, verify, reason about programming languages
- Informal: living natural languages

Execution-based vs. Rule-based thinking

	Execution-based thinking	Rule-based thinking
Basis	Implementation	Rule-based calculus
Focus	Execution runs	Properties
Example 0	Counting with one's fingers	Algebra
Example 1	Truth tables	Propositional calculus
Example 2	Debugging / Test cases	Reasoning about program correctness
First steps	Initially easy to visualize	Needs mental training
Abstraction	Low degree of abstraction possible	High degree of abstraction possible
Scalability	Limits the complexity of problems that can be solved (e.g. truth table with 20 variables)	Allows larger and more complex problems to be solved

Programming Paradigms



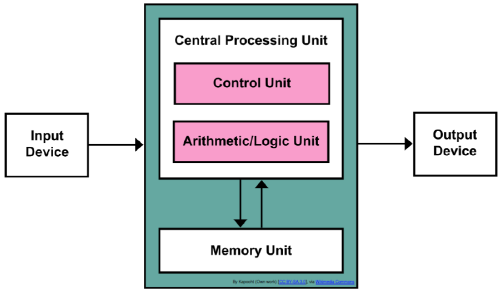
Imperative

- Befehlend
- Focuses on **how** a program operates
- Commands change a program's state

Common building blocks:

- Assignment: $x := x + 1$
- Sequential composition: $(\ldots; \ldots)$
- Conditional execution: $(if \ldots then \ldots else)$
- Repetition: $(while \ldots do \ldots)$ / $(goto \ldots)$

Von Neumann architecture



Declarative

- Expresses the logic of a computation without describing its control flow
- Describes **what** the program should accomplish
- **how** - left to the language's implementation
- eliminates / minimises side effects

Examples:

- Spreadsheets
- Regular expressions
- Query languages
- Functional programming languages
- Logic programming

Functional Programming Introduction

Basic Features

- Referential transparency
- Functions as first-class citizens
- Higher-order functions
- Algebraic data types
- Pattern matching
- Recursion
- Types & type inference
- Haskell specific: Type classes, Functors, Applicatives, Monads

What is FP?

- Declarative programming paradigm
- Foundation: Chruch's Lambda calculus
- Pure: No (or controlled) mutable state
- Pure: Expressions are by default side effect free

Why functions?

- Simple concept and properties
- High level of abstraction possible
- Powerful reasoning more easily possible

Why use FP?

- Easier to reason about
- Easier to write
- Easier to get right

No Mutable State

- Referential Transparency: WYSIWYG for programmers
- $f(x)$ only depends on the def. of f and the value of x
- **No** mutable variables
- **No** assignments
- **No** imperative control structures
- All data structures are immutable

Problem with mutable state

- Every statement can potentially change the underlying state of the program
- Executions of a statement can depend on previously executed statements

Functions are first-class Citizens

- just like any other values: $1, true$
- Can be anonymous: $(\lambda x.x + 1)$
- Can be input/output to other functions
- Can be composed in powerful ways $f o g$

More FP in the future

- Increased expectations on reliability of software
- Increased demands on scalability, complexity, performance
- FP can surpass the limitations of the mainstream
- FP is an active area of applied research
- Increased adoption of FP features in mainstream languages (generics)

Haskell Introduction

Standard Prelude

Select the first element of a list

```
head [1,2,3,4,5]
1
```

Remove the first element of a list

```
tail [1,2,3,4,5]
[2,3,4,5]
```

Select the nth element of a list

```
[1,2,3,4,5] !! 2
3
```

Select the first n elements of a list

```
take 3 [1,2,3,4,5]
[1,2,3]
```

Remove the first n elements from a list

```
drop 3 [1,2,3,4,5]
[4,5]
```

Calculate the length of a list

```
length [1,2,3,4,5]
5
```

Calculate the sum of a list of numbers

```
sum [1,2,3,4,5]
15
```

Calculate the product of a list of numbers

```
product [1,2,3,4,5]
120
```

Append two lists

```
[1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Reverse a list

```
reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Function Application Syntax

```
f a b + c * d
```

- Function application is denoted using space
- Multiplication is denoted using *
- Function application has higher prio than other operators

Mathematics	Haskell
$f(x)$	`f x`
$f(x,y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x,g(y))$	`f x (g y)`
$f(x)\ g(y)$	`f x * g y`

Useful GHCi Commands

Command	Meaning
`:load` *name*	load script name
`:reload`	reload current script
`:set editor` *name*	set editor to *name*
`:edit` *name*	edit script *name*
`:edit`	edit current script
`:type` *expr*	show type of *expr*
`:?`	show all commands
`:quit`	quit GHCi

Naming Requirements

- Function and argument names: begin with lowercase letter
- List arguments: s-suffix, by convention: xs, ns, nss

The Layout Rule

- In a sequence of definitions, definitions must begin in the same column
- Avoids the need for braces and semicolons

Types and Classes

Type

- Name for a collection of related values
- e.g. $Bool = False|True$
- Every well-formed expression has a type
- Type can be automatically calculated at compile time: *type inference*
- Removing the need for type checks at run time =¿ safer /faster

Type Error

- Applying a function to one or more arguments of the wrong type

Basic Types

- **Bool**: logical values
- **Char**: single values
- **String**: strings of chars
- **Int**: fixed-precision int
- **Integer**: arbitrary-precision integers
- **Float**: floating-point numbers

List Types

- Sequence of values of the same type
- Type of a list says nothing about the length
- Lists of lists possible

```
[['a'], ['b','c']] :: [[Char]]
```

Tuple Types

- Sequence of values of different types
- Type of a tuple encodes its size
- Type of components is unrestricted

```
(False, 'a', True) :: (Bool, Char, Bool)
```

Function Types

- Mapping from values of one type to values of another
- Argument and result types are unrestricted
- Functions with multiple arguments / results possible using lists / tuples

```
add :: (Int, Int) -> Int
add (x,y) = x + y
```

Curried Functions

- Multiple input arguments typically implemented by returning functions as results
- Functions that take their arguments one at a time
- Functions with more that two arguments can be curried by returning nested functions

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

Currying Conventions

- $->$ operator is right associative
- Rightmost type is the result
- Precending types are the input
- Consequence: Function application is left associative
- All functions in Haskell are normally defined in curried form

Polymorphic Functions

- Function type contains one or more type variables
- Type variables must begin with lowercase letters

```
length :: [a] -> Int
```

Overloaded Functions

- Function type contains one or more class constraints
- Class constraints are expressed as type classes
- Can be instantiated to any types that satisfy the constraints

```
(+) :: Num a => a -> a -> a
```

Type classes in Haskell:

- Num: Numeric types
- Eq: Equality types
- Ord: Ordered types

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
```

Defining Functions

Conditional Expressions

- Can be nested
- Must always have an else branch

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

Guarded Equations

- Make definitions using multiple conditions
- Easier to read

```
abs n
  | n >= 0 = n
  | otherwise = -n
```

Pattern Matching

- Can be defined many different ways
- Some ways may be more efficient
- _: wildcard pattern, matches any argument
- Patterns are matched in order
- Patterns may not repeat variables

```
not :: Bool -> Bool
not False = True
not True = False
```

List Patterns

- Non-empty lists are constructed by repeated use of *cons* operator: (:)
- Functions on lists can be defined using $x : xs$ patterns

```
[1,2,3,4] = 1 : (2 : (3 : (4 : [])))
```

Lambda Expressions

- Used to define anonymous functions
- e.g. $\backslash x- > x + x$
- Can give a formal meaning to functions defined using currying

```
add x y = x + y
add = \x -> (\y -> x + y)
```

Avoid naming functions that are only referenced once:

```
odds n = map f [0 .. n - 1]
        where
          f x = x * 2 + 1
odss n = map (\x -> x * 2 + 1) [0 .. n - 1]
```

Operators

- **Prefix** notation is used for function application
- Is **Infix** notation desired, one may use operators (e.g. +)
- Functions can be converted into operators using backticks: '*div*'
- Operators can be converted to functions using brackets: (+)

List Comprehension

- Define functions in very compact manner
- Elegant way to perform iteration in declarative style

```
factors n = [x | x <- [1 .. n], n `mod` x == 0]
prime n = factors n == [1, n]
primes n = [x | x <- [2 .. n], prime x]
```

Generators

- States how to generate values for a variable

```
x <- [1 .. 5]
```

Multiple Generators

- Order of generators changes order of elements
- Multiple generators act like nested loops

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Dependant Generators

- Later generators can depend on variables, introduced by earlier generators

```
[(x,y) | x <- [1 .. 3], y <- [x .. 3]]
```

Guards

- Restrict values produced by earlier generators

```
[x | x <- [1 .. 10], even x]
```

zip - Function

- Maps two lists to a list of pairs

```
zip :: [a] -> [b] -> [(a, b)]
```

String Comprehensions

- String: sequence of chars enclosed in double quotes
- Internally strings are represented as lists of chars
- Polymorphic list-functions can be applied to strings

```
"abc" :: String
['a', 'b', 'c'] :: [Char]
```

Recursive Functions

Advantages

- Some functions are simpler to define using recursion
- Properties can be proven using induction

Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result.

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

Why are they useful?

- **Common programming idioms** can be encoded as functions within the language itself
- **Domain specific languages** can be defined as collections of higher-order functions
- **Algebraic properties** of higher-order functions can be used to reason about programs

Examples

map

Applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
-- defined using list comprehension --
map f xs = [f x | x <- xs]
-- defined using recursion --
map f [] = []
map f (x : xs) = f x : map f xs
-- for example: --
map (+1) [1,3,5,7]
[2,4,6,8]
```

filter

Selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
-- defined using list comprehension --
filter p xs = [x | x <- xs, p x]
-- defined using recursion --
filter p [] = []
filter p (x : xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
-- example --
filter even [1 .. 10]
[2,4,6,8,10]
```

foldr

Why is foldr Useful?

- Some recursive functions on lists are easier to define
- Properties of functions can be proved using albegraic properties of foldr
- Advanced program optimisations can be simpler if foldr is used in place of explicit recursion

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f [] = v
f (x : xs) = x ⊕ f xs
```

Some function $\oplus$ is applied to the head of non-empty lists, and f to its tail. The value **v** is typically the identity element of $\oplus$.

For example:

```
sum [] = 0
sum (x : xs) = x + sum xs

product [] = 1
product (x : xs) = x * product xs

and [] = True
and (x : xs) = x && and xs
```

The higher-order library function **foldr** (fold right) uses this pattern of recursion with the function $\oplus$ and the value **v** as arguments:

```
-- defined using recursion --
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x : xs) = f x (foldr f v xs)
-- example --
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
-- other examples --
length = foldr (\ _ n -> 1 + n) 0
reverse = foldr (\ x xs -> xs ++ [x]) []
-- append function (++) --
(++ ys) = foldr (:) ys
```

Other Library Functions

(.): returns the composition of two functions as a single function.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \ x -> f (g x)
-- example --
odd :: Int -> Bool
odd = not . even
```

all: decides if every element of a list satisfies a given predicate.

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]
-- example --
all even [2,4,6,8,10]
True
adslnasdlasd
```

any: decides if at least one element of a list satisfies a predicate.

```
any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]
-- example --
any (== ' ') "abc def"
True
```

takeWhile: selects elements from a list while a predicate holds.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x : xs)
  | p x = x : takeWhile p xs
  | otherwise = []
-- example --
takeWhile (/= ' ') "abc def"
"abc"
```

dropWhile: drops elements from a list while a predicate holds.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x : xs)
  | p x = dropWhile p xs
  | otherwise = x : xs
-- example --
dropWhile (== ' ') " abc"
"abc"
```

Declaring Types and Classes

Type Declarations

- New name for an existing type
- Can make other types easier to read
- Can have type parameters
- Can be nested
- Cannot be recursive

```
type String = [Char]
-------------------------
type Pos = (Int, Int)
origin :: Pos
origin = (0, 0)

left :: Pos -> Pos
left (x,y) = (x-1, y)

-- Type parameter --
type Pair a = (a,a)
mult :: Pair Int -> Int
mult (m,n) ) m*n

-- Nested --
type Trans = Pos -> Pos
```

Data Declarations

- Completely new type by specifying its values
- Values of new types can be used the same ways as built in types
- Constructors may also have parameters
- May also have type parameters
- Can be recursive

```
data Bool = False | True
data Answer = Yes | No | Unknown

-- Parameter --
data Shape = Circle Float | Rect Float Float
square :: Float -> Shape
square n = Rect n n

-- Type parameters --
data Maybe a = Nothing | Just a

-- Recursive --
data Nat = Zero | Succ Nat
Succ (Succ (Succ Zero)) = 3
```

Newtype Declarations

If a new type has a **single constructor** with a **single element**, it can be declared using the *newtype* mechanism. For example, a type of natural numbers:

```
newtype Nat = N Int
```

Polymorphism

1. Ad-hoc Polymorphism: function with the same name denotes different implementations (function overloading / interfaces)
2. Parametric Polymorphism: Code written to work with many possible types
3. Subtype Polymorphism: one type can be substituted for another (subtype / supertype)

Type Classes

Class Declarations

Name of the declared type class

Type parameter

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

Functions required for a type to be a member of this type class.

Default implementation (optional).

Extending Classes

Name of the type class to be extended

Name of the resulting extended type class

Additional functions required

```
class Eq a => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
```

- All members of class *Ord* are also members of type class *Eq*
- Members of *Ord* therefore also require *(==)* to be defined

Instance Declaration

A Type can be declared to be an instance of a type class using an instance declaration.

Constraint on type m

In order for (Maybe m) to be a member of the type class Eq , it is required that the type m belongs to the type class Eq.

```
instance Eq m => Eq (Maybe m) where
  Just x == Just y     = x == y
  Nothing == Nothing   = True
  _ == _               = False
```

Derived Instances

Default implementations for the built-in type classes *Eq, Ord, Show, Read* can be generated automatically for **data** declarations using the **deriving keyword**:

```
data Bool = False | True
  deriving (Eq, Ord, Show, Read)
-- examples --
> False == False
True
> False < True
True
> show False
"False"
>read "False" :: Bool
False
```

Type Constructures & Kinds

Type Constructors construct one type from another. This behaviour is expressed using kinds. A kind is the type of a 'type'.

```
K ::= * | K -> K
```

Pronounced "type" or "star"

Kind of ordinary/proper/concrete/basic types

Kind of "type constructors" or "type operators"

Examples:

```
data Maybe a = Nothing | Just a
type Pair a = (a, a)
```

Values, Types & Kinds

Only types of kind * can have a value.

Kind	*	*	*	*	*	* -> *	* -> * -> *
Type	Bool	Char	[a] -> [a]	Maybe Char	a -> Maybe a	Maybe	(,)
Value	True	'a'	reverse	Just 'a'	Just		

Why are Kinds useful?
Just like how

- Types are used to prevent the user from making errors at the value level
- Kinds are used to prevent the user from making errors at the type level

Records

Convenient syntax when data constructors have multiple parameters:

```
data Person = Person
  { name :: String
  , age :: Int
  , children :: [Person]
  } deriving (Show)
```

```
data Person = Person String Int [Person]
  deriving (Show)

name :: Person -> String
name (Person n _ _) = n

age :: Person -> Int
age (Person _ a _) = a

children :: Person -> [Person]
children (Person _ _ c) = c
```

Modules

- Collection of related values, types, classes, etc.
- Name: Identifier that starts with a capital letter
- Naming convention for hierarchy: seperated using '.'
- Each module **Module.Name** must be contained in a file called **Module.Name.hs**

Main Aims:

- Organising
- Defining visibility

Lambda Calculus

- Indroduced by Alonzo Church in 1930s
- Part of an investigation into the foundations of mathematics
- Expresses computation based on function abstraction
- Expresses application using variable binding and substitution
- Most compact and elegant programming language
- Basics for functional programming
- **Pure:** no pre-defined constants
  - Only reserved words: 'λ', '.', '(' and ')',
- Proofs that function evaluation is enough for functional programming

Sequents in LC

Sequents in **LC** have the identical form as in
PC (Propositional Calculus) and
FoPCe (First-Order Predicate Calculus with Equality).

$$H \vdash G$$

Hypotheses: Finite set of predicates

Turnstile/tee: "therefore", "entails"...

Goal: Single predicate

Under the hypotheses H, prove the goal G

Syntax in LC

Formulae in LC can eiter be:

- predicates (P)
- λ-Terms (M)

Predicate

Equality relation

Note: Predicates consist solely of equalities between λ-terms.

$$P \quad ::= \quad M = M$$

$$M \quad ::= \quad x \;\mid\; \lambda x.\ M \;\mid\; M\ M$$

λ-terms

Variable (placeholder)

λ-abstraction (anonymous function definition)

Application of one λ-term to another

Conventions

Application binds tighter than abstraction
$\lambda x. M_1 M_2$ represents $\lambda x.(M_1 M_2)$ and not $(\lambda x. M_1) M_2$
Application is left associative
$M_1 M_2 M_3$ represents $(M_1 M_2) M_3$ and not $M_1 (M_2 M_3)$

Free and Bound Variables

Bound variables are basically placeholders. Their names have no significance and can be renamed.

bound free

$$(\lambda y.\ y)\ a\ b$$

binding occurrence

α-equivalence:

$$\lambda x.\ M \mathrel{\widehat{=}} \lambda y.\ [x := y]M \quad \text{if} \quad (y\ \underline{\text{nfin}}\ M) \qquad (\mathrel{\widehat{=}}_{\lambda\alpha})$$

substitution not free in aka α-equivalence

Proof rules of LC

Lambda Calculus contains only one proof rule schema of major significance: β-**reduction**

$$\frac{}{\mathsf{H} \vdash (\lambda x.\ M)\ N = [x := N]M}\ \beta$$

... used to generate equalities such as:

$$(\lambda x.\ x)\ a = a$$
$$(\lambda x.\ x\ b\ c)\ a = a\ b\ c.$$

β-Reduction defines what function application means in the context of lambda calculus. It states that applying a lambda abstraction $(\lambda x.M)$ to another lambda term (N) results in all occurences of the formal parameter (x) within the body of the abstraction (M) being replaced with the actual parameter (N) supplied.

$$\frac{(\lambda x.\ square\ x)\ 5}{= square\ 5}$$

$$\frac{(\lambda x.\ square\ x)\ (\lambda y.\ square\ y)\ 5}{= (square\ (\lambda y.\ square\ y))\ 5}$$

Computation with LC

Computation in the lambda calculus is done by repeatedly applying the rule schema β to achieve β-reduction.

Evaluation in LC = Reduction

Normal Form

A λ-term is said to be in **normal form** if no further reductions can be applied to it. It is possible for a λ-term to offer several opportunities for reduction simultaneously:

$$\begin{aligned} &(\lambda x.\ square\ x)\ (\underline{(\lambda y.\ square\ y)\ 5}) \\ &= \underline{(\lambda x.\ square\ x)\ (square\ 5)} && \because \beta \\ &= square\ (square\ 5) && \because \beta \end{aligned}$$

$$\begin{aligned} &\underline{(\lambda x.\ square\ x)\ ((\lambda y.\ square\ y)\ 5)} \\ &= square\ (\underline{(\lambda y.\ square\ y)\ 5)} && \because \beta \\ &= square\ (square\ 5) && \because \beta \end{aligned}$$

Confluence: Every λ-term has at most one normal form

Function Application Syntax

Functions are **first-class citizens**. Functions and their arguments belong to the same syntactic category.

- Prefix notation is used instead of infix notation
- Parameters do not need to be enclosed in parantheses

Currying

Motivation: LC only allows unary (1 argument) function application.
Currying: A function with several arguments can be thought of as a series of higher order functions, each with being unary.

- Not just a smart syntactic trick
- Makes functional definitions more concise, modular and reusable

Definitions

- Not strictly necessary
- Sometimes convenient
- Adding definitions as equalities within the hypotheses of LC sequents

$$\boxed{square\ =\ \lambda x.\ *\ x\ x} \ \vdash\ (\lambda x.\ square\ x)\ ((\lambda y.\ square\ y)\ 5) = (*\ (*\ 5\ 5)\ (*\ 5\ 5))$$

Delta-Reduction

δ-Reduction: substitution of a defined symbol with its definition

$$square\ =\ \lambda x.\ *\ x\ x\ \ \vdash\ (\lambda x.\ square\ x)\ ((\lambda y.\ square\ y)\ 5) = (*\ (*\ 5\ 5)\ (*\ 5\ 5))$$

$$\begin{aligned} &(\lambda x.\ square\ x)\ (\underline{(\lambda y.\ square\ y)\ 5}) \\ &= \underline{(\lambda x.\ square\ x)\ (square\ 5)} && \because \beta \\ &= square\ (\underline{square}\ 5) && \because \beta \\ &= square\ (\underline{(\lambda x.\ *\ x\ x)\ 5}) && \because \delta \\ &= \underline{square}\ (*\ 5\ 5) && \because \beta \\ &= \underline{(\lambda x.\ *\ x\ x)\ (*\ 5\ 5)} && \because \delta \\ &= (*\ (*\ 5\ 5)\ (*\ 5\ 5)) && \because \beta \end{aligned}$$

Evaluation Strategies

redex: reducible expression (any $\beta\delta$-reducible sub-term)
evaluation strategy: order in which redexes are reduced

- λ-terms may have more than one redex at any stage of the evaluation
- LC does not place any constraints on the order in which redexes are reduced
- The order plays an important role in the **length of derivations** and their **termination**

Leftmost Innermost (aka. applicative order / innermost first)

1. The innermost redex is reduced First
2. In case there is more than one innermost redex, the leftmost innermost redex is reduced First

Note: A redex is innermost, if there is no other redex inside it.

$$\begin{aligned} &\underline{square}\ (square\ 5) \\ &= (\lambda x.\ *\ x\ x)\ (\underline{square}\ 5) && \because \delta \\ &= (\lambda x.\ *\ x\ x)\ (\underline{(\lambda x.\ *\ x\ x)\ 5}) && \because \delta \\ &= \underline{(\lambda x.\ *\ x\ x)\ (*\ 5\ 5)} && \because \beta \\ &= (*\ (*\ 5\ 5)\ (*\ 5\ 5)) && \because \beta \end{aligned}$$

- A functions arguments are substituted into the body of a function after they are reduced
- A functions arguments are reduced exactly once
- Parameter passing: **Call by value**

Leftmost Outermost (aka. normal order / outermost first)

1. The outermost redex is reduced First
2. In case there is more than one outermost redex, the leftmost-outermost redex is reduced First

Note: A redex is outermost, if there is no other redex outside it.

$$\begin{aligned} &\underline{square}\ (square\ 5) \\ &= \underline{(\lambda x.\ *\ x\ x)\ (square\ 5)} && \because \delta \\ &= (*\ (\underline{square}\ 5)\ (square\ 5)) && \because \beta \\ &= (*\ (\underline{(\lambda x.\ *\ x\ x)\ 5})\ (square\ 5)) && \because \delta \\ &= (*\ (*\ 5\ 5)\ (\underline{square}\ 5)) && \because \beta \\ &= (*\ (*\ 5\ 5)\ (\underline{(\lambda x.\ *\ x\ x)\ 5})) && \because \delta \\ &= (*\ (*\ 5\ 5)\ (*\ 5\ 5)) && \because \beta \end{aligned}$$

- A functions arguments are substituted into the body of a function before they are reduced
- A functions arguments are reduced as often as they are needed
- **If a normal form exists, leftmost outermost will find it**
- Parameter passing: **Call by name**

Lazy Evaluation

Implementation technique to make call by name more efficient. Uses memorizing (caching) to avoid computing the same expression more than once. **Default for Haskell**

Encoding Data and Operations

The pure lambda calculus **does not have any primitive data types** such as Booleans, numbers, tuples, etc.

Boolean Algebra

Data:

$$\top = \lambda x.\lambda y.\ x$$
$$\bot = \lambda x.\lambda y.\ y$$

Operations:

$$\wedge = \lambda p.\lambda q.\ p\ q\ p$$
$$\vee = \lambda p.\lambda q.\ p\ p\ q$$
$$\neg = \lambda p.\ p\ \bot\ \top$$

Arithmetic

Data:

$$\begin{aligned} 0 &= \lambda f.\lambda x.\ x \\ 1 &= \lambda f.\lambda x.\ f\ x \\ 2 &= \lambda f.\lambda x.\ f\ (f\ x) \\ 3 &= \lambda f.\lambda x.\ f\ (f\ (f\ x)) \\ &\vdots \end{aligned}$$

Operations:

$$\begin{aligned} succ &= \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x) \\ + &= \lambda m.\lambda n.\ m\ succ\ n \\ * &= \lambda m.\lambda n.\ m\ (+\ n)\ 0 \\ power &= \lambda b.\lambda e.\ e\ b \\ pred &= \lambda n.\lambda f.\lambda x.\ n\ (\lambda g.\lambda h.\ h\ (g\ f))\ (\lambda u.\ x)\ (\lambda u.\ u) \\ - &= \lambda m.\lambda n.\ n\ pred\ m \\ isZero &= \lambda n.\ n\ (\lambda x.\ \bot)\ \top \\ \leq &= \lambda m.\lambda n.\ isZero\ (-\ m\ n) \\ areEqual &= \lambda m.\lambda n.\ (\wedge\ (\leq\ m\ n)\ (\leq\ n\ m)) \end{aligned}$$

Pairs

Data:

$$pair = \lambda x.\lambda y.\lambda f.\ f\ x\ y$$

Operations:

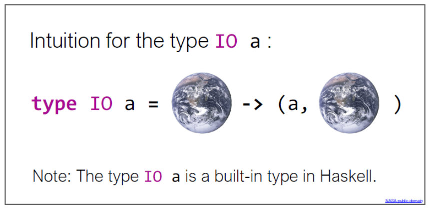
$$fst = \lambda p.\ p\ \top$$
$$snd = \lambda p.\ p\ \bot$$

Interactive Programming

The Problem:

- Haskell programs are pure mathematical functions:
  - **Haskell programs have no side effects**
- Reading from the keyboard and writing to the screen are side effects
  - **Interactive programs have side effects**

The Solution:
Interactive programs can be written in Haskell by using *types* to distinguish *pure expressions* from *impure actions* (aka. commands) that may involve side effects.



- **IO Char:** The type of actions that return a character
- **IO ():** The type of purely side effecting actions that return no result value
  - (): Type of tuples with no value (unit type), like *void*

Basic Actions

```
getChar :: IO Char
```

Reads a character from the keyboard, echoes it to the screen and returns the character as its result value.

```
putChar :: Char -> IO ()
```

Writes the character **c** to the screen and returns no result value.

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

Returns the value **v** without performing any interaction.

Sequencing

A sequence of actions can be combined as a single composite action using the keyword **do**:

```
act :: IO (Char, Char)
act = do {
   x <- getChar;
   getChar;
   y <- getChar;
   return (x, y)
}
```

Reading a string from the keyboard:

```
getLine:: IO String
getLine = do {
   x <- getChar
   if x == '\n'
      then
         return []
   else
      do {
         xs <- getLine
         return (x : xs)
      }
}
```

Writing a string to the screen:

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x : xs) = do putChar x
                     putStr xs
```

Writing a string and moving to a new line

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Programming in Haskell

Functors

The Idea

- Both *Lists* and *Maybe* wrap values of vertain types
- We need a generic way to apply functionsto the wrapped values

Definition

Functors are types that wrap values of other types and allow us to map functions over the wrapped value.

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

Restriction: Functors can only map unary functions over them.

Type Class Laws:

- Concerning the compiler: the only requirement to be a Functor is an implementation of **fmap** with the proper type
- Nevertheless, The **fmap** function should not change the structure of the Functor, only its elements
- Such requirements are expressed as *type class laws*
- The compiler will not notice if you violate a type class law, but users of your interface will

1. fmap has to preserve identity:

```
fmap id = id
```

2. fmap has to preserve function composition:

```
fmap (g . h) = fmap g . fmap h
```

Instances

List and **Maybe** are instances of Functor:

```
instance Functor [] where
   fmap = map

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

Other instances of **Functor** in the standard Prelude:

- []
- Maybe
- IO
- Option
- ((->) r)

Usage

Generalizing **inc** and **sqr**:

```
inc :: Functor f => f Int -> f Int
inc = fmap (+1)

sqr :: Functor f => f Int -> f Int
sqr = fmap (^2)

-- Usage --
> inc [1,2,3]
[2,3,4]
> inc (Just 1)
Just 2
> inc Nothing
Nothing
```

Applicative Functors

Idea

Provide a generic function to wrap values:

```
pure :: a -> f a
```

Provide generalized function application:

```
(<*>) :: f (a -> b) -> f a -> f b
```

Definition

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

Instances

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

instance Applicative [] where
   pure x = [x]
   fs <*> xs = [f x | f <- fs, x <- xs]
```

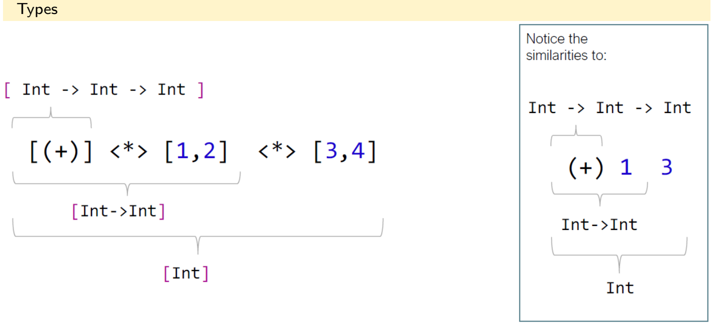
Usage

```
> pure (+) <*> (Just 11) <*> (Just 31)
(Just 42)
> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
> pure (+) <*> [1,2] <*> [3,4]
[4,5,5,6]
> [(*), (+)] <*> [1,2] <*> [3,4]
[3,4,6,8,4,5,5,6]
```

Evaluation

```
pure (+) <*> [1,2] <*> [3,4]
= [(+)] <*> [1,2] <*> [3,4]
= [(+) 1,(+) 2] <*> [3,4]
= [(+) 1 3,(+) 1 4,(+) 2 3,(+) 2 4]
= [4,5,5,6]
```

Types



Type Class Laws

1. Identity: pure has to preserve identity:

```
pure id <*> x = x
```

2. Homomorphism: pure has to preserve function application:

```
pure (g x) = pure g <*> pure x
```

3. Interchange: Order of evaluation must not matter when applying an effectful function to a pure argument:

```
x <*> pure y = pure (\f -> f y) <*> x
```

4. Composition: $<*>$ must be associative

```
x <*> (y <*> z) = (pure (.) <*> x <*> y) <*> z
```

The IO Applicative Functor

```
instance Applicative IO where
   pure = return
   mf <*> mx = do
      f <- mf
      x <- mx
      return (f x)
-- allows for simple composition of IO actions --
getChars :: Int -> IO String
getChars 0 = return []
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

Effectful Programming

Effectful means that arguments and return values are no longer just plain (pure) values, but many also have so-called *effects*:

- The possibility of failure: e.g. option type **Maybe**
- Aggregating multiple results: e.g. using the list type []
- Performing IO: e.g. using the action type **IO**

Monads

Motivation

```
-- ADT for expressions --
data Expr = Val Int
   | Div Expr Expr

-- problematic eval fn --
eval (Val n) = n
eval (Div l r = eval l `div` eval r)

-- employ Maybe to fail gracefully --
safediv _ 0 = Nothing
safediv l r = Just (l `div` r)

-- eval using safediv (ugly AF) --
eval (Val n) = Just n
eval (Div l r) = case eval l of
   Nothing -> Nothing
   (Just x) -> case eval r of
      Nothing -> Nothing
      (Just y) -> safediv x y

-- eval using Maybe Monad --
eval (Val n) = Just n
eval (Div l r) = do
   x <- eval l
   y <- eval r
   safediv x y
```

The Idea

- We cannot use **fmap** to simplify **eval** due to mismatching types
- We can exploit the repeating pattern we observed

Provide a function called: (>>=) (aka. bind)
Binds an effectful value into an effectful function:

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

Definition

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

Instances

```
instance Monad Maybe where
   Nothing >>= _ = Nothing
   (Just x) >>= f = f x

instance Monad [] where
   xs >>= f = [y | x <- xs, y <- f x]
```

Evaluation

```
do {n <- pure 10; m <- pure 2; safediv n m}
= pure 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (do syntax with explicit λ parentheses)
= Just 10 >>= (\n -> (pure 2 >>= (\m -> safediv n m)))     (definition of pure)
= (\n -> (pure 2 >>= (\m -> safediv n m))) 10              (definition of >>=)
= pure 2 >>= (\m -> safediv 10 m)                          (function application)
= Just 2 >>= (\m -> safediv 10 m)       (definition of pure)
= (\m -> safediv 10 m) 2                (definition of >>=)
= safediv 10 2                          (function application)
= Just (10 `div` 2)           (definition of safediv)
= Just 5                      (definition of div)
```

Types

$$\underbrace{\underbrace{\texttt{pure 10}}_{\texttt{Maybe Int}}\ \texttt{>>=}\ \underbrace{\texttt{(\textbackslash n -> }\underbrace{\texttt{(}\underbrace{\texttt{pure 2}}_{\texttt{Maybe Int}}\ \texttt{>>=}\ \underbrace{\texttt{(\textbackslash m -> safediv n m)}}_{\texttt{Int -> Maybe Int}}\texttt{)}}_{\texttt{Maybe Int}}\texttt{)}}_{\texttt{Int -> Maybe Int}}}_{\texttt{Maybe Int}}$$

Type Class Laws

1. return is the identity for bind (>>=):

```
return x >>= f = f x
mx >>= return = mx
```

2. bind (>>=) must be associative:

```
(mx >>= f) >>= g = mx >>= (\x -> (f x >>= g))
```

Overview

	Application operator	Type of application operator	Function	Argument	Result
Pure function application	juxtapose , (`$`)	`(a -> b) -> a -> b`	pure	pure	pure
Functor	`fmap`, (`<$>`)	`(a -> b) -> f a -> f b`	pure	effectual	effectual
Applicative functor	(`<*>`)	`f (a -> b) -> f a -> f b`	pure (generalisation of (`<$>`) to support multiple arguments)	effectual	effectual
Monad	(`>>=`)	`m a -> (a -> m b) -> m b`	effectual	effectual	effectual