Programming Languages

Foundations of functional programming

Foundations of functional programming

Exercises

Functional programming

A central problem in computing is processing information:

Functional programming consists of defining functions and applying them to process information.

"Functions" are truly (partial) functions:

- Applying a function has no side effects.
- The same input always corresponds to the same output.
- Data structures are immutable.

Functions are data like any other:

- They can be passed as parameters.
- They can be returned as results.
- They can be part of data structures.
  (e.g., a binary tree with functions in its nodes).

Functional programming

A functional program is given by a set of equations:

Example

```
length []       = 0
length (x : xs) = 1 + length xs
```

```
    length [10, 20, 30]
≡   length (10 : (20 : (30 : [])))
=   1 + length (20 : (30 : []))
=   1 + (1 + (length (30 : [])))
=   1 + (1 + (1 + length []))
=   1 + (1 + (1 + 0))
=   1 + (1 + 1)
=   1 + 2
=   3
```

Expressions

Expressions are sequences of symbols used to represent data, functions, and functions applied to data.
(Recall: functions are also data).

An expression can be:

1. A **constructor**:

   ```
   True  False  []  (:)  0  1  2  ...
   ```

2. A **variable**:

   ```
   length  sort  x  xs  (+)  (*)  ...
   ```

3. The **application** of one expression to another:

   ```
   sort list
   not True
   not (not True)
   (+) 1
   ((+) 1) (square 5)
   ```

4. There are also expressions of other forms, as we will see.
   The three above are the fundamental ones.

Expressions

We agree that application is **left-associative**:

```
f x y       ≡  (f x) y          ≢  f (x y)
f a b c d   ≡  (((f a) b) c) d
```

Example

```
    [1, 2]
≡   1 : [2]
≡   (:) 1 [2]
≡   ((:) 1) [2]
≡   ((:) 1) (2 : [])
≡   ((:) 1) ((:) 2 [])
≡   ((:) 1) (((:) 2) [])
```

Expressions

Example

```
addOne = (+) 1
```

```
    addOne (addOne 5)
=   ((+) 1) (addOne 5)
≡   1 + addOne 5
=   1 + ((+) 1) 5
≡   1 + (1 + 5)
=   1 + 6
=   7
```

Types

There are sequences of symbols that are not well-formed expressions.

Example

```
1,,2      )f x(
```

There are expressions that are well-formed but meaningless.

Example

```
True + 1
0 1
[[], (+)]
```

Types

A **type** is a specification of the invariant of a piece of data or a function.

Example

```
99          :: Int
not         :: Bool -> Bool
not True    :: Bool
(+)         :: Int -> (Int -> Int)
(+) 1       :: Int -> Int
((+) 1) 2   :: Int
```

The type of a function expresses a **contract**.

Typing conditions

For a program to be **well-typed**:

1. All expressions must have a type.
2. Each variable must always be used with the same type.
3. Both sides of an equation must have the same type.
4. The argument of a function must have the domain type.
5. The result of a function must have the codomain type.

$$\frac{\texttt{f :: a -> b} \qquad \texttt{x :: a}}{\texttt{f x :: b}}$$

Only well-typed programs make sense.

It is not necessary to explicitly write types. (Inference).

Types

We agree that "->" is **right-associative**:

```
a -> b -> c       ≡  a -> (b -> c)    ≢  (a -> b) -> c
a -> b -> c -> d  ≡  a -> (b -> (c -> d))
```

Example

```
add4 :: Int -> Int -> Int -> Int -> Int
add4 a b c d = a + b + c + d
```

It can be thought of as:

```
add4 :: Int -> (Int -> (Int -> (Int -> Int)))
(((add4 a) b) c) d = a + b + c + d
```

Polymorphism

Some expressions have more than one type.

We use *type variables* `a`, `b`, `c` to denote unknown types:

```
id   ::  a -> a
[]   ::  [a]
(:)  ::  a -> [a] -> [a]
fst  ::  (a, b) -> a
snd  ::  (a, b) -> b
```

Example

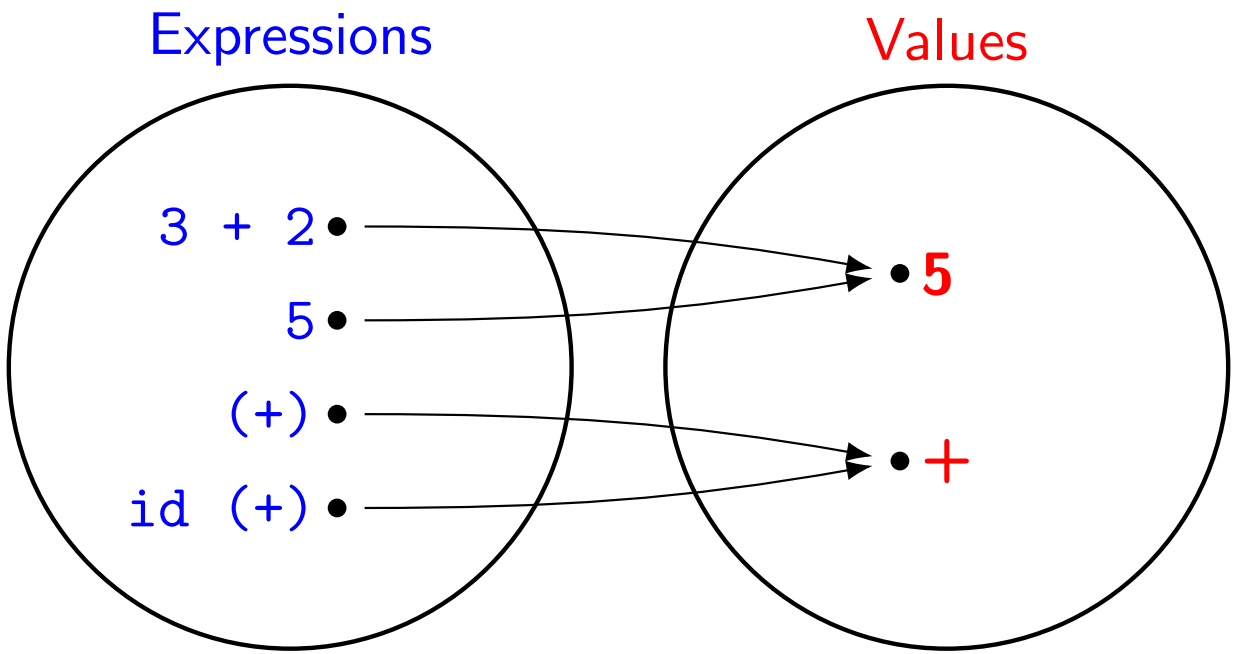
```
flip f x y = f y x
```

What type does `flip` have?

```
    flip (:) [2, 3] 1
=   (:) 1 [2, 3]
≡   1 : [2, 3]
=   [1, 2, 3]
```

Computation model

Given an expression, its *value* is computed using the equations:



Some well-typed expressions have no value. E.g.: `1 / 0`.
We say such expressions are undefined or have value $\perp$.

Computation model

A functional program is given by a set of equations.
More precisely, by a set of **oriented equations**.

An equation `e1 = e2` is interpreted from two points of view:

1. **Denotational point of view.**
   It declares that `e1` and `e2` have the same meaning.
2. **Operational point of view.**
   Computing the value of `e1` reduces to computing the value of `e2`.

Computation model

The *left-hand* side of an equation is not an arbitrary expression.
It must be a function applied to **patterns**.

A pattern can be:

1. A variable.
2. A wildcard _.
3. A constructor applied to patterns.

The left-hand side must not contain repeated variables.

Example

Which equations are syntactically well-formed?

```
sumFirst (x : y : z : _) = x + y + z
predecessor (n + 1) = n
equals x x = True
```

Computation model

Evaluating an expression consists of:

1. Find the outermost subexpression that matches the left-hand side of an equation.
2. Replace the subexpression matching the left-hand side of the equation with the expression corresponding to the right-hand side.
3. Continue evaluating the resulting expression.

Evaluation stops when one of the following occurs:

1. The expression is a constructor or an applied constructor.

   ```
   True      (:) 1      [1, 2, 3]
   ```

2. The expression is a *partially* applied function.

   ```
   (+)      (+) 5
   ```

3. An *error state* is reached.
   An error state is an expression that does not match the equations defining the applied function.

Computation model

Example: result — constructor

```
tail :: [a] -> [a]
tail (_ : xs) = xs
```

```
tail (tail [1, 2, 3]) ⇝ tail [2, 3] ⇝ [3]
```

Example: result — partially applied function

```
const :: a -> b -> a
const x y = x
```

```
const (const 1) 2  ⇝  const 1
```

Computation model

Example: undefined — error

```
head :: [a] -> a
head (x : _) = x
```

```
head (head [[], [1], [1, 1]]) ⇝ head []
                              ⇝ ⊥
```

Example: undefined — non-termination

```
loop :: Int -> a
loop n = loop (n + 1)
```

```
loop 0 ⇝ loop (1 + 0)
       ⇝ loop (1 + (1 + 0))
       ⇝ loop (1 + (1 + (1 + 0)))
         ...
```

Computation model

Example: non-strict evaluation

```
undefined :: Int
undefined = undefined
```

```
          head (tail [undefined, 1, undefined])
    ⇝     head [1, undefined]
    ⇝     1
```

Computation model

Example: infinite lists

```
from :: Int -> [Int]
from n = n : from (n + 1)
```

```
    from 0
⇝   0 : from 1
⇝   0 : (1 : from 2)
⇝   0 : (1 : (2 : from 3))   ⇝   ...
```

```
    head (tail (from 0))
⇝   head (tail (0 : from 1))
⇝   head (from 1)
⇝   head (1 : from 2)
⇝   1
```

Computation model

Note. In Haskell, the order of equations matters.
If multiple equations match, the first one is always used.

Example

```
isShort (_ : _ : _) = False
isShort _           = True
```

```
isShort []      ⇝  True
isShort [1]     ⇝  True
isShort [1, 2]  ⇝  False
```

Higher-order functions

Let's define function composition ("`g . f`").

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
```

Another way to define it (using "lambda" notation):

```
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f = \ x -> g (f x)
```

Higher-order functions

What do the following functions have in common?

```
doubleL :: [Float] -> [Float]
doubleL []       = []
doubleL (x : xs) = x * 2 : doubleL xs

isEvenL :: [Int] -> [Bool]
isEvenL []       = []
isEvenL (x : xs) = x `mod` 2 == 0 : isEvenL xs

lengthL :: [[a]] -> [Int]
lengthL []       = []
lengthL (x : xs) = length x : lengthL xs
```

They all follow the scheme:

```
g []       = []
g (x : xs) = f x : g xs
```

Higher-order functions

How can we abstract the scheme?

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs

doubleL xs = map (\ x -> x * 2) xs
isEvenL xs = map (\ x -> x `mod` 2 == 0) xs
lengthL xs = map length xs
```

Another way:

```
doubleL    = map (* 2)
isEvenL    = map ((== 0) . (`mod` 2))
lengthL    = map length
```

Higher-order functions

What relationship exists between the following functions?

```
negatives :: [Int] -> [Int]
negatives []       = []
negatives (x : xs) = if x < 0
                     then x : negatives xs
                     else negatives xs

nonEmpty :: [[a]] -> [[a]]
nonEmpty []       = []
nonEmpty (x : xs) = if not (null x)
                    then x : nonEmpty xs
                    else nonEmpty xs
```

Both follow the scheme:

```
g []       = []
g (x : xs) = if p x
             then x : g xs
             else g xs
```

Higher-order functions

How can we abstract the scheme?

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs

negatives = filter (< 0)
nonEmpty  = filter (not . null)
```

Exercise (homework)

```
merge     :: (a -> a -> Bool) -> [a] -> [a] -> [a]
mergesort :: (a -> a -> Bool) -> [a] -> [a]
```

The first parameter is a function that determines a total order relation between elements of type a.

Exercises

Some recursion schemes

a) Define a function:

```
operation :: (a -> a -> a) -> [a] -> a
```

that given a binary operation (assumed associative) and a non-empty list returns the result of performing the operation among all the elements.

Informally:

$$\texttt{operation}\ (\bigstar)\ [x_1, x_2, \ldots, x_n] = x_1 \bigstar x_2 \bigstar \ldots \bigstar x_n.$$

b) Define summation and product as particular cases.

Some recursion schemes

a) Define a function:

```
while :: (a -> Bool) -> (a -> a) -> a -> a
```

that given a condition, a state transformation function, and an initial state, returns the final result reached by repeatedly applying the transformation function as long as the condition holds.

b) Using the function `while`, define the function that computes the n-th element of the Fibonacci sequence iteratively.

Recall that:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_n + F_{n+1}$$

Infinite binary trees

A **binary tree** (BT) is represented as a value of the type:

```
data BT a = Nil | Bin (BT a) a (BT a)
```

An **infinite BT** (IBT) is represented as a value of the type:

```
data IBT a = IBin (IBT a) a (IBT a)
```

Define a function:

```
prunedFromLevel :: Int -> IBT a -> BT a
```

such that `prunedFromLevel` n t denotes the BT resulting from pruning the tree t starting from level n.

Infinite binary trees

Consider the data types for **directions**, and **paths** (finite lists of directions):

```
data Direction = Left | Right
type Path = [Direction]
```

An infinite binary tree whose nodes have data of type a can also be represented as a **function of paths** that, given a path, indicates the value of the node at that position:

```
type PathFunction a = Path -> a
```

Exercise

a) Define a function:

```
pathFunctionOf :: IBT a -> PathFunction a
```

that given an IBT returns its path function.

b) Define the inverse function:

```
ibtOf :: PathFunction a -> IBT a
```

that given a path function returns its IBT.

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Recommended reading

Chapter 4 of Bird's book.

Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2015.

Comments: types

Note. We said:

> "Each variable must always be used with the same type."

Is the following program well-typed?

```
successor :: Int -> Int
successor x = x + 1

opposite :: Bool -> Bool
opposite x = not x
```

Yes. There are two "x" with different types but they are different variables.
The program could be rewritten as:

```
successor x = x + 1
opposite y = not y
```