nedap
technology that matters

Functional programming

Part four!

Evaluation order(1)

- From the first lecture, we recall that functions can be evaluated in different orders.
- Consider the following function:

```
inc :: Int -> Int
inc n = n + 1
```

Evaluation order(2)

```
inc :: Int -> Int
inc n = n + 1

  inc (2*3)
=       {applying *}
  inc 6
=       {applying inc}
  6 + 1
=       {applying +}
  7
```

Evaluation order(3)

```
inc :: Int -> Int
inc n = n + 1

  inc (2*3)
=       {applying inc}
  (2*3) + 1
=       {applying *}
  6 + 1
=       {applying +}
  7
```

Evaluation order(4)

- In Haskell, any two different ways of evaluating the same expression will always produce the same final value, provided that both terminate.
- Note that in imperative languages this is not necessarily the case

Evaluation order(5)

- Given the imperative expression n + (n = 1), with n = 0 at the start

```
   n + (n = 1)
=      {applying n}
   0 + (n = 1)
=      {applying =}
   0 + 1
=      {applying +}
   1
```

Evaluation order(6)

- Given the imperative expression n + (n = 1), with n = 0 at the start

```
    n + (n = 1)
=       {applying =}
    n + 1
=       {applying n}
    1 + 1
=       {applying +}
    2
```

nedap
technology that matters

Evaluation strategies(1)

- Consider the expression `f x y`
- Applying a function is also called ‘reducing’
- An expression that can still be reduced is called ‘reducible expression’ or ‘redex’ for short.
- Expressions can comprise sub-expressions

Evaluation strategies(2)

- Example:

```
mult :: (Int, Int) -> Int
mult (x,y) = x * y

mult (1+2, 2+3)
```

Evaluation strategies(3)

- Applying the inner redexes first is called ‘innermost evaluation’
- With innermost evaluation, arguments are fully evaluated before a function is applied
- This evaluation strategy is also called pass-by-value

Evaluation strategies(4)

- Applying the outermost redex first is called ‘outermost evaluation’
- This allows functions to be applied before their arguments are evaluated.
- This evaluation strategy is also called pass-by-name.
- Note that for some functions require their arguments to be fully evaluated (e.g. +, *), even under outermost evaluation. These functions are called strict.

Lambda expressions

- Outermost/innermost evaluation also works with lambda expressions
- Evaluation in Haskell does not ‘look inside’ lambda’s!
- Example:

```
  (\x -> 1 + 2) 0
=   { applying the lambda }
  1 + 2
=   { applying + }
  3
```

- This is the only valid evaluation order for this expression

Termination

- Consider the following function:

```
inf :: Int
inf = 1 + inf
```

- Regardless of evaluation strategy, this function does not terminate
- Consider the function fst:

```
fst (x,_) = x
```

- Now, consider the expression

```
fst (0, inf)
```

- How does this evaluate?

Number of reductions

- Different evaluation orders require a different number of reductions
- Outermost evaluation often requires more steps, in particular when arguments are used multiple times
- Haskell solves this by a mechanism called ‘sharing’

Infinite structures

- Consider the following function:
```
ones :: [Int]
ones = 1 : ones
```
- Now consider the expression
```
head ones
```
under different evaluation strategies

Modular programming

- Lazy evaluation allows separating ‘control’ and ‘data’
- For example, take 3 ones separates control (take) from the data (ones).
- Note that you still need to take care to avoid non-termination

Strict application

- Haskell is lazy by default
- Strictness can be forced using $!
- f $! x will force evaluation of x before f
- This is sometimes useful for performance purposes