Functional programming

Common FP techniques for software engineers

Some slides are based on Graham Hutton's public slides

The agenda

- Why is FP interesting and important?
- A short introduction to Haskell
- Functional programming techniques:
  - List comprehensions
  - Recursion
  - Immutability
  - Pure functions
  - Pattern matching
  - Higher-order functions
  - Lazy evaluation
  - (Function composition)
  - (type checking, QuickCheck)

Not a lecture on Haskell!

Police helps dog bite victim

GÖTEBORGS UNIVERSITET

CHALMERS

Programming languages

- Programs are written in *programming languages*
  - Not natural language, must be unambiguous
  - Syntax and semantics
- There are hundreds of different programming languages, each with their strengths and weaknesses
- A large system will often contain components in many different languages
- Important that a language is *expressive*

Two major paradigms

Imperative programming:

- Instructions are used to change the computer's state:
  - `x := x+1`
  - `deleteFile(”slides.pdf”)`
- Run the program by following the *instructions* top-down
- Describing *how* to solve

Declarative programming:

- Functions are used to *declare* dependencies between data values:
  - `y = f(x)`
  - `x = 32`
- Dependencies drive evaluation
- Describing *what* to solve

Functional programming

- *Functions* are:
  - used to declare dependencies between data values: `y = f(x)`
  - the basic building blocks of (functional) programs
  - used to *compose* functions into other functions
  - *only* dependent on the argument (in so-called pure functions)
- *Functional programming* is a *style* of programming in which the basic method of computation is the *application of functions to arguments*
- A *functional programming language* is one that *supports* and *encourages* the functional style
- FP has a strong mathematical foundation: the *lambda (λ) calculus*
- There are many FP languages: Haskell, Erlang, ML, Scala, …

Haskell

Haskell is:

- a very high-level language
  - allows you to focus on the import aspects of programming
- expressive and concise
  - can achieve a lot with a little effort
- strongly, statically typed
- lazily evaluated
- pure and has monads for `IO`
- *not* a particularly high-performance language
  - prioritizes programmer-time over computer-time

Variables and arguments

- Functions are *abstractions* of calculations, which we want to perform with varying values
- To capture the varying parts of a calculation we can introduce *variables*, which abstract away from particular values and thus *vary*
- When we apply a function on a value, we *substitute* the variable with the given value
- The given value in a function application is also called an *argument*
  - An argument or the given value can be regarded as the input to a function
- A different name for a variable in a function is a *parameter*
  - A function is parametrized over a variable

```
f x = x * 3 + 1

ghci> f 3
10
```

`f` is applied to an argument in this case the value 3. In the calculation (definition) of `f` we can substitute `x` with the value 3.

LIST COMPREHENSIONS

List comprehensions

- In mathematics, the *comprehension* notation can be used to construct new *sets* from old sets:

$$\{x^2 | x \in \{1 \ldots 5\}\}$$

The set {1,4,9,16,25} of all numbers x^2 such that x is an element of the set {1...5}

- In Haskell, a similar comprehension notation can be used to construct new *lists* from old lists:

```
[x^2 | x <- [1..5]]
```

The list [1,4,9,16,25] of all numbers `x^2` such that `x` is an element of the list [1..5]

- In Python:

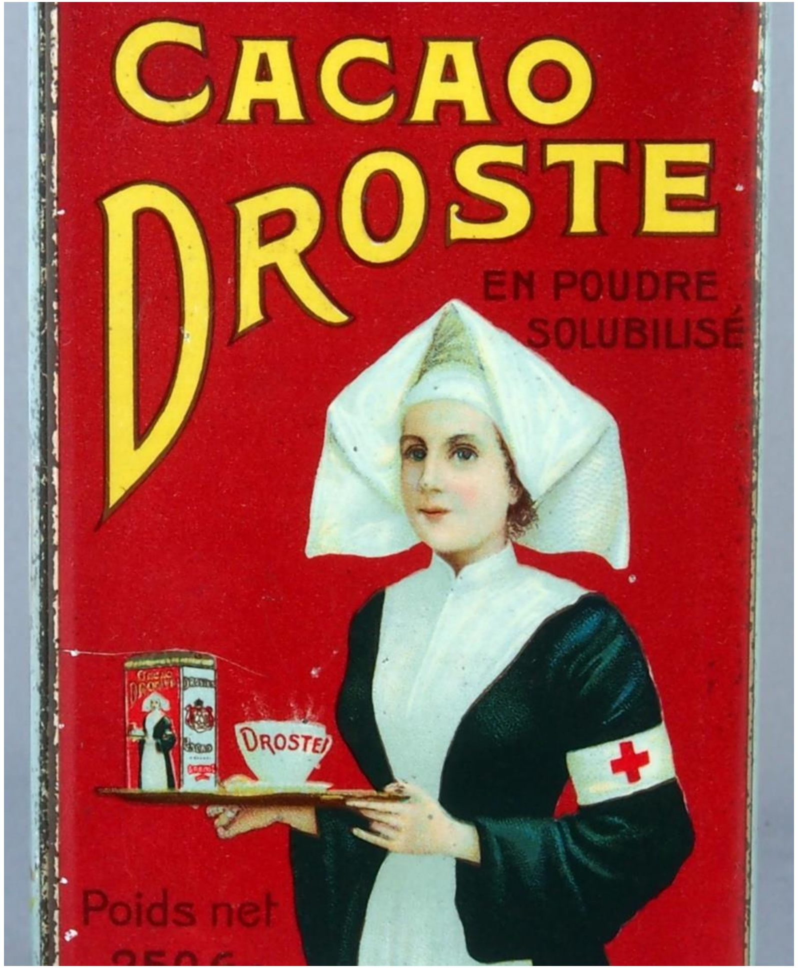
```
[x**2 for x in range(1,6)]
```

IMMUTABILITY AND PURITY

Immutability and purity

- Use mutable data structures with care
  - Try to treat mutable data structures as immutable
  - Use programming principles: defensive copying and mutate-by-copy
- Separate the side-effects
  - Create so-called *pure* functions
  - This makes testing a lot easier
  - You can use equational reasoning
    - `f(1) - f(1) == 0`

RECURSION



GÖTEBORGS UNIVERSITET
CHALMERS

Why is recursion useful?

- Some functions, such as factorial, are *simpler* to define in terms of other functions.
- As we shall see, however, many functions can *naturally* be defined in terms of themselves.
- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of *induction*.

Recursive functions

- In Haskell, functions can also be defined in terms of themselves. Such functions are called *recursive*.
- `fac 0 = 1` is appropriate because 1 is the identity for multiplication: `1*x = x = x*1`.
- The recursive definition *diverges* on integers < 0 because the base case is never reached:

```
ghci> fac (-1)
*** Exception: stack overflow
```

```
fac 0 = 1
fac n = n * fac (n-1)


fac 2
  =>
2 * fac 1
  =>
2 * (1 * fac 0)
  =>
2 * (1 * 1)
  =>
2 * 1
  =>
2
```

`fac` maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

PATTERN MATCHING

Pattern matching

- Many functions have a particularly clear definition using *pattern matching* on their arguments
- A variable or wildcard matches *everything*
- Patterns are matched *in order*, that is top-down

`not` maps `False` to `True`, and `True` to `False`

```
def not_(x: bool) -> bool:
    match x:
        case False: return True
        case _:     return False
```

HIGHER-ORDER FUNCTIONS

How to be productive as a programmer

- Use general-purpose functions from the standard libraries when possible
- Create general, re-usable functions and use them many times
  - Identify common patterns
  - Avoid copying and modifying code
- Use libraries from other sources
  - But… finding and learning how to use libraries can also takes time
  - But… dependencies can be a nightmare
- Less code
  => fewer bugs
  => lower maintenance burden

How to make functions more general

- Add extra *parameters* to make functions more versatile
- Create functions that work for many types instead of just one
  - *Polymorphic functions,* also known as *generic functions*
  - *Overloaded functions,* which work for a range of types
- Create *higher-order functions*
  - A key ingredient in functional programming

What is a higher-order function?

- It's a function that takes another function as an argument
- `even` is a first-order function
- `map` and `filter` are higher-order functions

```
> even 1
False

> even 2
True

> map even [1,2,3,4,5]
[False,True,False,True,False]

> filter even [1,2,3,4,5]
[2,4]
```

Higher-order functions

- A function is called *higher-order* if it takes a function as an argument
- *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.

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

`twice` is higher-order because it takes a function as its first argument

The `map` function

- The higher-order library function called `map` applies a function to every element of a list.
- The map function can be defined in a particularly simple manner using a list comprehension.
- Alternatively, for the purposes of proofs, the map function can also be defined using recursion.

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

map _ []     = []
map f (x:xs) = f x : map f xs
```

```
ghci> map (+1) [1,3,5,7]
[2,4,6,8]
```

The `filter` function

- The higher-order library function `filter` selects every element from a list that satisfies a predicate.
- `filter` can be defined using a list comprehension.
- Alternatively, it can be defined using recursion.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]

filter _ []       = []
filter p (x:xs)
  | p x         = x : filter p xs
  | otherwise = filter p xs
```

```
ghci> filter even [1..10]
[2,4,6,8,10]
```

Case study: sums, products, and conjunction of lists

- Common pattern:
  - combining the elements of a list with an operator
- Differences:
  - the *operator* and the *base case*

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

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

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

Factoring out the differences

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

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

`f` maps the empty list to some value `v`, and any non-empty list to some function ⊕ applied to its head and `f` of its tail.

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

v = 0
⊕ = +

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

v = 1
⊕ = *

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

v = `True`
⊕ = &&

GÖTEBORGS UNIVERSITET
CHALMERS

Factoring out the differences

- The higher-order library function `foldr` (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value `v` as arguments.
- `foldr` itself can be defined using recursion.
- However, it is best to think of `foldr` non-recursively, as simultaneously replacing each `(:)` in a list by a given function, and `[]` by a given value.

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

```
  sum [1,2,3]
=
  foldr (+) 0 [1,2,3]
=
  foldr (+) 0 (1:(2:(3:[])))
=
  1+(2+(3+0))
=
  6
```

replace every `(:)` with + and `[]` with 0

LAZY EVALUATION

GÖTEBORGS UNIVERSITET
CHALMERS

Lazy evaluation

- Expressions in Haskell are evaluated using lazy evaluation, which:
  - Only evaluates *when necessary*
  - *Terminates* whenever possible
  - Supports programming with *infinite* data structures, such as lists
  - Enables to write *modular* programs

Evaluating expressions

- Evaluate: `square n = n * n`

```
  square (1 + 2)
=
  square 3
=
  3 * 3
=
  9
```

Apply `(+)` first

```
  square (1 + 2)
=
  (1 + 2) * (1 + 2)
=
  3 * (1 + 2)
=
  3 * 3
=
  9
```

Apply `square` first

GÖTEBORGS UNIVERSITET
CHALMERS

Evaluation strategies and termination

- There are two main strategies for deciding which *reducible expression* (redex) to consider next:
    - *Innermost* redex: does not contain another reducible expression
    - *Outermost* redex: expression not contained by another
- Any way of evaluating the *same* expression will give the *same* result, provided it *terminates*
- Outermost evaluation may give a result when innermost fails to terminate
- If any evaluation sequence terminates, so will outermost

FUNCTION COMPOSITION

Function composition

- The library function `(.)` returns the *composition* of two functions as a single function.
- The composition function works just like in math.
- (the function `isSpace` is defined in `Data.Char`)

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

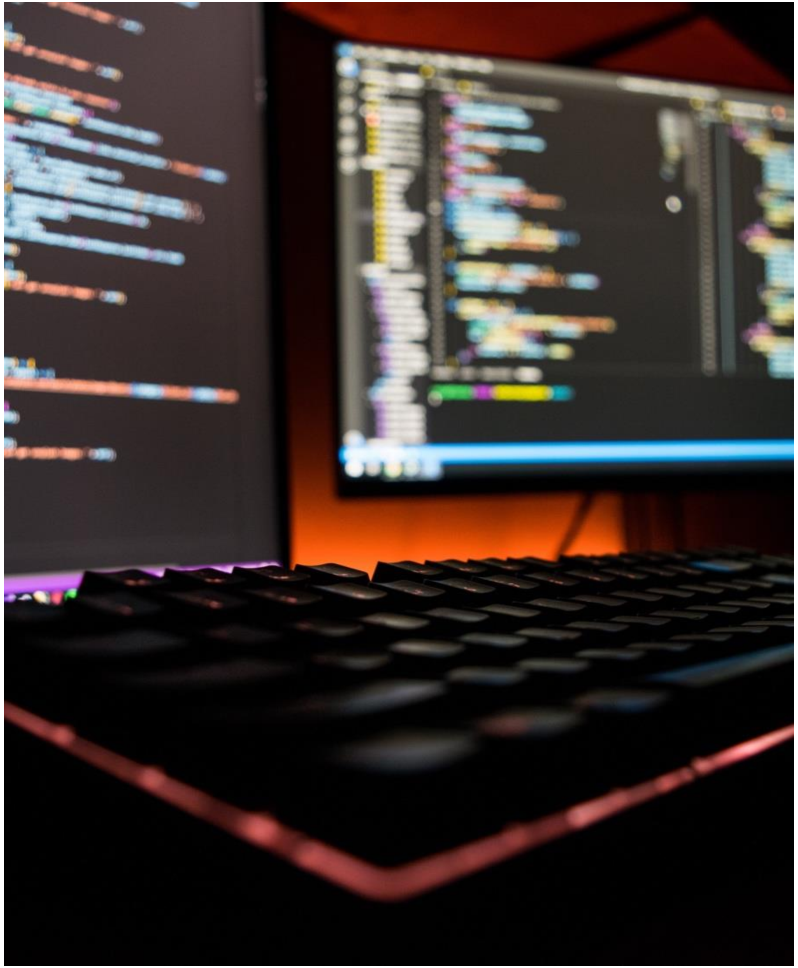
```
removeSpaces s = filter (not . isSpace) s

ghci> removeSpaces "abc def \n ghi"
"abcdefghi"
```

```
odd = not . even
```

TYPE CHECKING

GÖTEBORGS UNIVERSITET
CHALMERS

SUMMARY



Pointers

- Articles:
  - Why Functional Programming Matters, John Hughes
  - QuickCheck: a lightweight tool for random testing of Haskell programs, Koen Claessen and John Hughes
  - Composing contracts: an adventure in financial engineering, Simon Peyton Jones
- Books:
  - The Craft of Functional Programming, Simon Thompson
  - Programming in Haskell, Graham Hutton

FP programming courses

- (Introduction to) Functional Programming
- Data Structures (with FP)
- Advanced Functional Programming
- Parallel Functional Programming
- Formal Methods in Software Development
- …

GÖTEBORGS UNIVERSITET

CHALMERS

CONTACT ME

Alex Gerdes

alexg@chalmers.se

EDIT 6114

GÖTEBORGS UNIVERSITET

CHALMERS