Programovací jazyky F# a OCaml

Chapter 2.

Refactoring code using functions

What is "Refactoring"?

Refactoring is the process of changing a program's internal structure without modifying its existing functionality, in order to improve internal quality attributes of the software.

- Possible goals of refactoring:
  - Improve readability, simplify code structure
  - Improve maintainability, performance
  - Improve extensibility, reusability
- Today's topic
  - Creating reusable code using functions

Example from mathematics

- Geometric series

  Pattern that is repeated in many calculations

  Definition of a series: $S_n = \sum_{k=0}^{n} r^n$

- Common calculations with the series

  n-th element of the series: r^n

  Sum of first *n* elements: $\frac{1 - r^{n+1}}{1 - r}$

  number of elements

  ratio

  Both calculations are parameterized!

Refactoring in mathematics

- Reusing expressions in different context
  - We need to assign values to parameters
  - Wrap expressions into functions:

$$e(r,n) = r^n \qquad s(r,n) = \frac{1 - r^{n+1}}{1-r}$$

- Calling a function:
  - To get the actual value: $s\left(\frac{1}{2},3\right) = \frac{15}{8} \qquad e\left(\frac{1}{2},3\right) = \frac{1}{4}$
  - From other calculations: $S(r) = \lim_{n\to\infty} \sum_{k=0}^{n} e(r,k)$

Sum of the series

Refactoring using functions in F#

Simple function declaration

» Like value declarations with parameters:

Actually: Value *is* a function without parameters

let binding just like for values

Value or function name

Parameters (None for values)

```
> let nthTerm q n =
    pown q n;;

val nthTerm : int -> int -> int

> nthTerm 2 10;;
val it : int = 1024
```

Body: expression that uses parameters

Inferred type signature

Calling a function

Specifying types of parameters

» Functions are statically typed

```
> let nthTerm q n = pown q n;;
val nthTerm : int -> int -> int
```

Works only with integers!

» Specifying type using *type annotations*:

```
> let nthTerm (q:float) n = pown q n;;
val nthTerm : float -> int -> float
```

Annotation in function declaration

Annotation anywhere inside expression

```
> let nthTerm q n = ((pown q n):float);;
val nthTerm : float -> int -> float
```

Working with "any type" – possible but difficult

Creating parameterized functions

» What functions can we create from:

```
let percent = 3.0  // Interest rate
// Value of $100000 after 10 years
100000.0 * pown (1.0 + percent / 100.0) 10
```

Which part of the expression to parameterize?

```
let interestTenYears amount =
  amount * pown (1.0 + perc / 100.0) 10
```

Amount

```
let interestOneHundered years =
  100000.0 * pown (1.0 + perc / 100.0) years
```

Years

```
let interest amount years =
  amount * pown (1.0 + perc / 100.0) years
```

Both amount & years!

Refactoring using functions

» Reusing common parts of similar expressions

  We can “refactor” expressions as we need

» Turning sub-expressions into parameters?

  Which parts should be parameterized?

  Difficult decision – finding the balance!

» Using functions doesn’t change meaning

  Just like with mathematical expressions

Structuring code using modules

Organizing code

» Grouping related functionality together
  - For example objects in C#, modules in Pascal, …
  - How to do this with functions?

» In F#, we can use modules…
  - Groups related functions into a single "unit"
  - Modules do not have any private state
  - (… but F# supports object-oriented style too)

Declaring modules

» Can contain functions with the same name

Similar modules for different calculations

```
module Geometric =
  let nthTerm (q:float) n =
    pown q n
  let sumTerms (q:float) n =
    (1.0 - (pown q n)) / (1.0 - q)

module Arithmetic =
  let nthTerm d n =
    (float n) * d
  let sumTerms d n =
    0.5 * (float (n + 1)) * (nthTerm d n)
```

Module is not an expression

Contains function and value declarations

In OCaml, we need ";;" here!

Indentation in F#

Using modules

```
> Geometric.sumTerms 0.8 10;;
val it : float = 4.463129088
> Arithmetic.sumTerms 0.8 10;;
val it : float = 44.0
```

Directly using the dot-notation

```
open Arithmetic

nthTerm 2.0 10
sumTerms 2.0 10
```

Using the “open” directive to bring functions to the scope

- We cannot “open” module at runtime
  - Not needed frequently in functional programming
  - Other techniques (in F#, e.g. records or objects)

Understanding functions

Functions as values

Functional languages have the ability to use functions as first-class values. Functions can be assigned to symbols, passed as an argument, returned as the result, etc…

» We can write more expressible code

Essential for writing declarative programs

For example, assigning function value to a symbol:

Declares a new value "f"

```
> let f = Arithmetic.nthTerm;;
val f : (float -> int -> float)
> f 10.0 4;;
val it : float = 40.0
```

As the type shows, it is a function

Call it!

What is this good for?

» Choosing between functions at runtime:

```
let series = "g"

let sumFunc =
  match series with
  | "a" -> Arithmetic.sumTerms
  | "g" -> Geometric.sumTerms
  | _ -> failwith "unknown"

let res = sumFunc 2.0 10
```

Can be specified by the user: System.Console.ReadLine()

Dynamically choose which function to use

Run the function

Modules are quite useful here – similar structure!

Understanding function type

- We can return functions as the result too

  What is the type of this expression?

```
let add a =
  let addSecond b = a + b
  addSecond
```

addSecond : int -> int

a:int

b:int

add : int -> (int -> int)

- In F#, this means the same thing as:

```
> let add a b = a + b
val add : int -> int -> int
```

Parenthesis missing, but still same thing, just like: 1 + 2 + 3 = (1 + 2) + 3

Understanding function type

» Function with **N** parameters actually means

- **N = 1**: Function that returns the result as a value
- **N > 1**: Function that returns function of **N–1** parameters

» We work only with single-parameter functions

For example:

```
(float -> int -> int -> int -> float) =
(float -> (int -> (int -> (int -> float))))
```

This treatment of parameters is called *Currying*

Practical benefits of currying

» No need to provide all arguments at once

```
let r = Geometric.sumTerms 0.5 10
let r = (Geometric.sumTerms 0.5) 10
```

Same meaning!

» *Partial function application*:

```
let sumHalfs = Geometric.sumTerms 0.5
let r5 = sumHalfs 5
let r10 = sumHalfs 10
```

Create a function with *q=0.5*

Run the function with different *n*

What we've learned so far?

- Functions are values
  - Makes code more readable (sometimes!)
  - More ways to express abstraction we need
- We work with single-parameter functions
  - **The idea:** use smaller number of concepts
  - Functions of multiple parameters using *currying*
- Technically, F# compiler behaves more like C#

Functions as parameters

Aside: Printing in F#

» **printf** – "special" function for printing

```
> let name = "world"
  let num = 25
  let half = 0.5;;
(...)
> printf "Hello world!";;
Hello world!
> printf "Hello %s!" name;;
Hello world!
> printf "N = %d, F = %f" num half;;
N = 25, F = 0.500000
```

Prints a string

Format string – understood by the compiler

Number of parameters depends on format string

%s – string
%d – integer
%f – floating point

printfn – similar, adds new-line at the end

Functions as parameters

» Declaring function that takes a function:

```
> let printResults f =
    printfn "%f" (f 5)
    printfn "%f" (f 10)
  ;;
val printResults : (int -> float) -> unit
```

No syntactic difference!

Parameter type inferred by the compiler

Function as an argument

» **Note:** function types are not associative

Parenthesis sometimes matter!

`(int -> float) -> unit` ≠ `int -> (float -> unit)`

Using higher-order functions

- Higher-order functions (e.g. `printResults`)
- Providing compatible function as argument:

```
> let f n = 2.0 * float n;;
val f : int -> float
> printResults f;;
10.000000, 20.000000

> let f = Arithmetic.sumTerms 0.5;;
val f : (int -> float)
> printResults f;;
7.500000, 27.500000
> printResults (Arithmetic.sumTerms 0.5);;
```

Compatible type

Using partial function application

Compatible type

We can write this directly!

Lambda functions

» Creating functions without name

```
> (fun n -> float (n * n));;
val it : int -> float = <fun:clo@3>
```

The constructed value is a function

```
> let f = (fun n -> float (n * n));;
val f : int -> float
```

We can still create named function…

```
> printResults (fun n -> float (n * n));;
25.000000
100.000000
```

Using anonymous function as argument

» Useful especially with higher-order functions

Question

» (Using what we've seen,) can we write a program that will continue looping forever?
  - When writing down the evaluation of the program, can we get an infinite evaluation tree?

Example: Drawing function graphs

Homework #1

» Write a function **drawFunc** that takes a function as an argument and draws the graph of the given function (using WinForms).

The simplest possible signature is:

```
val drawFunc : (float32 -> float32) -> unit
```

Optionally, it can take two additional parameters to specify the X scale and Y scale.

Working with functions

» Mathematical operations with functions

Can be expressed using higher-order functions

```
let mirrorY (f:float32 -> float32) =
    (fun x -> f (-x))

let mirrorX (f:float32 -> float32) =
    (fun x -> -(f x))

let translate by (f:float32 -> float32) =
    (fun x -> (f x) + by)
```

Returns function g(x) such that g(x) = f(-x)

Takes any floating-point function as an argument

Builds the resulting function using lambda syntax

Working with functions

» Manipulating with functions:

```
> let f = translate 1.5f (mirrorX (fun x -> cos x));;
val f : (float32 -> float32)

> f 3.141592f;;
val it : float32 = 2.5f
```

» **Note**: Returning function could be simpler

```
let translate by (f:float32 -> float32) x =
    (f x) + by
translate 1.5f sin
```

Using partial function application

Arguably, this is less readable…

Homework #2

» Write a function differentiate that performs numerical differentiation of a function.

The signature should be:

```
val diff : (float32 -> float32) -> (float32 -> float32)
```

You can use the following (for some small "d"):

$$\lim_{d \to 0} \frac{f(x+d) - f(x)}{d}$$