Programvací jazyky F# a OCaml

Preface

Introduction to F# and functional programming

O této přednášce...

- Vznikla minulý rok (Milan Straka)
  - Rozsáhlý přehled jazyků OCaml a F#
  - ... včetně poměrně pokročilých vlastností
- Tento rok trochu jiný obsah...
  - Spíše úvod do funkcionálního programování a F#
  - Jak FP souvisí s ostatními předměty na MFF?
  - ... pokud bude čas tak i více ☺
- Webové stránky: http://tomasp.net/mff

Za co bude zápočet?

- Alternativní metody
  - Zajímavější cesta pro ty, které to zajímá :-)
  - Nějaká esej, článek nebo referát...
  - Nějaký projekt (něco zajímavého vymyslíme!)
- Za ***x*** bodů z domácích úkolů...
  - Domácí úkoly budou na webových stránkách
  - Budou *strašně těžké* (viz. alternativní metody)

Functional Programming

Functional programming

» 1930s – Lambda calculus
  - Theoretical foundation of functional languages
  - Attempt to formalize all mathematics

» 1958 – LISP
  - First functional (computer) programming language

» 1978 – ML (meta-language)
  - Originally used in theorem proving systems
  - Useful as a general purpose language too!

OCaml and F#

- 1990 – Haskell
  - Strict and lazy language, many advanced features
- 1996 – OCaml (based on ML)
  - Combines functional and object-oriented features
- 2002 – F# (based on OCaml)
  - Microsoft Research functional language for .NET
  - Now official part of Visual Studio 2010

Why functional programming?

- Functional abstractions hide *how* the code executes and specify *what* result we're trying to get
- Functional code is easier to understand, modify and reason about (assuming mathematical thinking)
- Code is more easily composable and testable
- Makes it easier to avoid repetitive patterns
- **Big topic today:** Parallel and asynchronous code

Programvací jazyky F# a OCaml

Chapter 1.

Expression as a basic building block

Functional programming

Functional programming is a style of programming that emphasizes the ***evaluation of expressions****, rather than* ***execution of commands****. The expressions in these languages are formed by using functions to combine basic values.*

[Hutton ed. 2002]

» Evaluation of expressions

This is exactly how mathematics works!

For example:

Roots of a quadratic equation: $\dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Program as an expression

» Writing equation as a single expression
  - Equations can get long and hard to read
  - How to break long equations into pieces?

» Another idea from mathematics:

$$Let\ discriminant\ \boldsymbol{D}\ be: b^2 - 4ac$$

$$Roots\ of\ quadratic\ equation: \frac{-b \pm \sqrt{D}}{2a}$$

…called **let binding** in functional languages

Expressions at a small scale…

» Calculating: $2x^2 - 5x + 3$ for $x = 2$:

```
> 2*(2*2) - 5*2 + 3;;
val it : int = 1
```

"‚;;" specifies the end of the input

F# interactive prints the result

» Using let binding to declare value *x*:

```
> let x = 2;;
val x : int = 2
```

Declared symbol with its value

```
> 2*(x*x) - 5*x + 3;;
val it : int = 1
```

We can re-use the symbol

Expressions at a larger scale…

» Writing real programs as expressions

Example: describing drawings and animations

```
let greenCircle = circle Brushes.OliveDrab 100.0f
let blueCircle = circle Brushes.SteelBlue 100.0f

let drawing =
 compose (translate -35.0f 35.0f greenCircle)
         (translate 35.0f -35.0f blueCircle)

af.Animation <- forever drawing;;
```

Declares two circle drawings

Composes complex drawing from circles

Animation that doesn't move

Exception: Modifies state of some object

Expressions at a larger scale…

» Example continued: composing animations

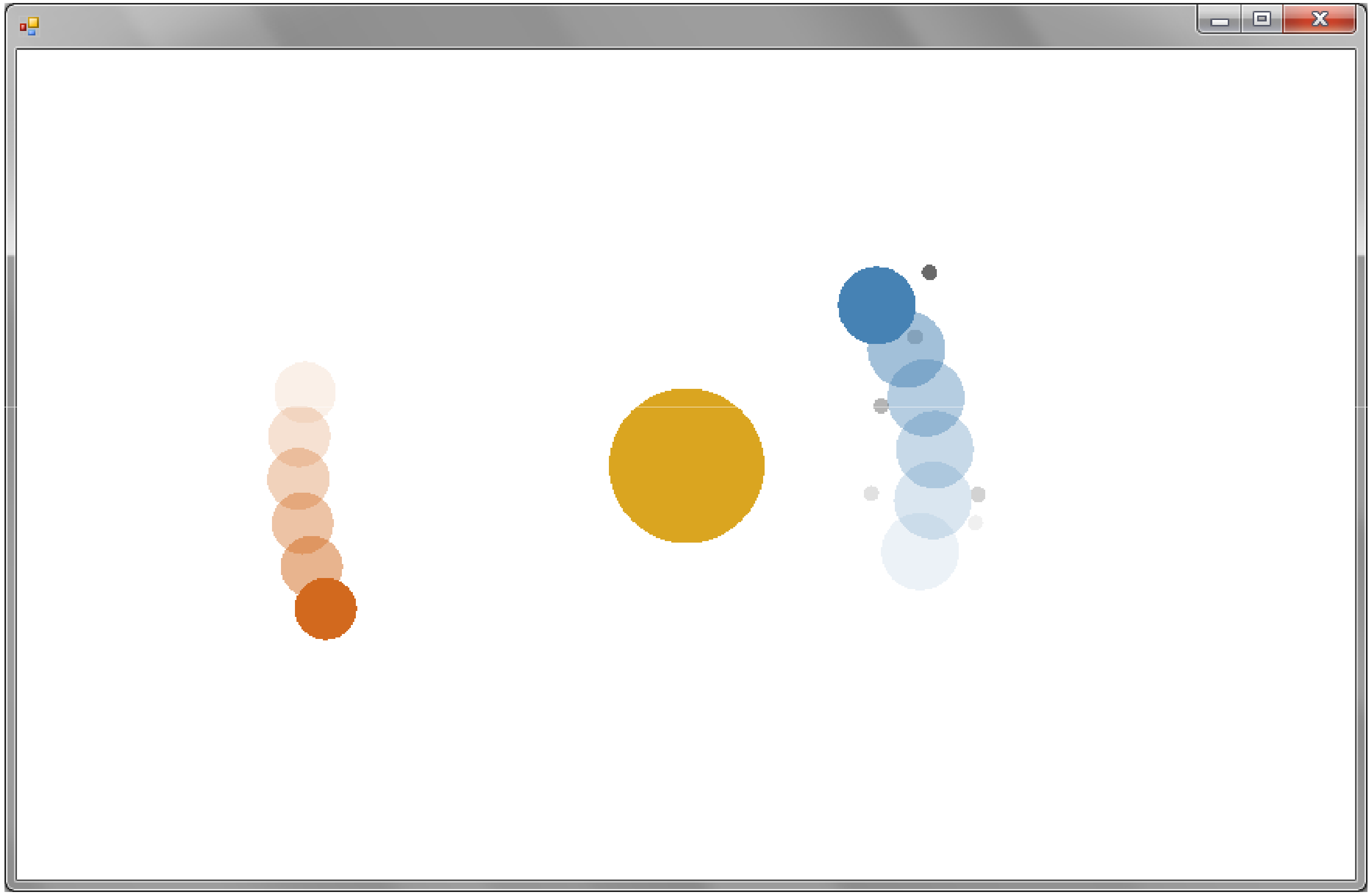
Declare “animated” solar system objects

```
let sun = circle (forever Brushes.Goldenrod) 100.0f.forever
let earth = circle (forever Brushes.SteelBlue) 50.0f.forever
let mars  = circle (forever Brushes.Chocolate) 40.0f.forever
let moon = circle (forever Brushes.DimGray) 10.0f.forever

let planets =
  sun -- (rotate 160.0f 1.0f
            (earth -- (rotate 40.0f 12.0f moon)))
      -- (rotate 250.0f 0.7f mars)
```

Compose solar system from objects

Solar system animation



Program as an expression

» The examples so far were single expressions!

» Declarative programming

  The program describes the results we want to get

  …not steps that should be performed to get it

» Composability

  Build complex programs from simpler pieces

  …without unexpected interactions

» Easy to reason about

  Calculating the result of an expression is easy

Calculating with expressions

Calculating with numbers

» Unary and binary operators:

```
> -5 + 10;;
val it : int = 5
> -5 + 10 * 2;;
val it : int = 15
> (-5 + 10) * 2;;
val it : int = 10
```

– unary "–", binary "+"

– standard precedence

– specifying precedence

» Calculating with floating point numbers:

```
> 1.5 * (12.2 + 7.8);;
val it : float = 30.0
> 1.5 * (12 + 8);;
error FS0001: The type 'int'
does not match the type 'float'
```

– works as expected

– oops! what happened?

Expressions have a type…

» Each expression has a type

Compiler forbids expressions with wrong types

```
// The most important types
> 42;;
val it : int = 42
> 42.0;;
val it : float = 42.0
> 42.0f;;
val it : float32 = 42.0f
```

```
// Other interesting types
> 42uy;;
val it : byte = 42uy
> 42.0UL;;
val it : uint64 = 42UL
> 42I;;
val it : BigInteger = 42I
```

» Type of binary numeric operators

```
// Overloaded
val (+) : int -> int -> int
val (+) : float -> float -> float
```

Converting numeric values

» No conversions happen automatically

```
> int 10.0;;
val it : int = 10
> int 10.0f;;
val it : int = 10
> float 10;;
val it : float = 10.0
```

Calling a function

Different type of the parameter

» Conversion functions are overloaded too:

```
val int : float -> int
val int : float32 -> int
```

Actually, it is generic – works with any type

Any type that satisfies some conditions…

```
val int : 'T -> int   // Where 'T supports conversion to int
```

Calling operators and functions

» Reading the type of a function

```
val pown : float -> int -> float
```

result

first parameter

second parameter

Why we always use the “->” symbol?

» Calculating square roots of: $2x^2 - 5x + 3$

```
> (-(-5.0) + sqrt ((pown -5.0 2) - 4.0*2.0*3.0)) / 2.0*2.0;;
val it : float = 6.0
> (-(-5.0) - sqrt ((pown -5.0 2) - 4.0*2.0*3.0)) / 2.0*2.0;;
val it : float = 4.0
```

Value bindings

» Calculating square roots – again!

```
let a, b, c = 2.0, -5.0, 3.0

(-b + sqrt ((pown b 2) - 4.0*a*c)) / 2.0*a
(-b - sqrt ((pown b 2) - 4.0*a*c)) / 2.0*a
```

» Even better – using discriminant:

```
let a, b, c = 2.0, -5.0, 3.0
let d = (pown b 2) - 4.0*a*c

(-b + sqrt d) / 2.0*a
(-b - sqrt d) / 2.0*a
```

Value binding as an expression

» Value binding is also an expression

```
> let n = 2 * 3 in 100*n + n;;
val it : int = 606

> 10 + (let n = 2 * 3 in 100*n + n) + 20;;
val it : int = 636
```

» If we use line-break in F#, we don't need "in"

White-space sensitive

```
>
10 + (let n = 2 * 3
      100 * n + n) + 20;;
val it : int = 636
```

We can still do the same thing!

Printing to console & unit type

Printing to the console

» This doesn't look like an expression:

```
printfn "Hello world!"
```

Side-effect: Evaluation of an expression modifies the state of the world (e.g. global variables or console)

Avoided where possible, but sometimes needed…

» …but, how can this be an expression?

Introducing the "unit" type…

```
> printfn "Hello world!";;
val it : unit = ()
```

Introducing the “unit” type

» Unit type represents no information

   It has exactly one value written as “( )”

» Functions without result return unit value

```
> let unitValue = ();;
val unitValue : unit = ()

> let unitValue = printfn "Hello world!";;
Hello world!
val unitValue : unit = ()
```

Calculating with unit values

» Sequencing expressions using semicolon:

```
> let n = (printfn "calculating"; 10 + 4);;
calculating
val n : int = 14
```

» In F# we can use new-line instead of “;”

```
let n = (printfn "calculating"
         10 + 4);;
```

» Ignored value should be unit

```
> let n = (10 + 2; 10 + 4);;
warning FS0020: This expression should
have type 'unit', but has type 'int'.
val n : int = 14
```

Ignoring values

» Generic function ignore:

```
val ignore : 'T -> unit
```

Ignores any value and returns unit instead

» The example from the previous slide:

```
> let n = (ignore (10 + 2); 10 + 4);;
val n : int = 14
```

Useful especially when working with .NET

Conditions and Booleans

Calculating with Booleans

- Another useful type – values **true** and **false**

```
> let b1, b2 = true, false;;
> b1 && b2;;
val it : bool = false
> b1 || b2;;
val it : bool = true
```

- Operators have short-circuiting behavior

```
> true || (printfn "test"; true);;
val it : bool = true
```

Second argument not evaluated

```
> true && (printfn "test"; true);;
test
val it : bool = true
```

Needs second argument too!

Writing conditions using “if”

```
if d = 0.0 then printfn "one solution"
elif d < 0.0 then printfn "no solutions"
else printfn "two solutions"
```

» This is also an expression!

```
> let n = (if d=0.0 then 1 elif d<0.0 then 0 else 2)
val n : int = 2
```

The “else” branch may be missing

Returns “unit” in any case

```
> if d < 0.0 then printfn "yay!";;
val it : unit = ()
> let n = (if (d = 0.0) then 1)
error FS0001: This expression was expected
to have type ‘unit’ but here has type ‘int’
```

What can this return?

Conditions using "match"

» Test value against multiple cases (patterns)

```
match d with
| 0.0 ->
    printfn "one solution"
| discr when discr < 0.0 ->
    printfn "no solutions"
| _ ->
    printfn "two solutions"
```

Constant pattern

Binding with condition

"match-all" pattern

» We'll talk about patterns in detail later

Patterns decompose complex data types

Evaluating expressions

Evaluation is a reduction

» Example:
```
let d = (pown -5.0 2) - 4.0*2.0*3.0 in
(5.0 + sqrt d) / 2.0*2.0
```

» We can follow one of the two rules:

Rule 1: Evaluate value of symbols first:

```
⇝ let d = 1.0 in (5.0 + sqrt d) / 2.0*2.0
⇝ (5.0 + sqrt 1.0) / 2.0*2.0
⇝ ... ⇝ 6.0
```

Rule 2: Replace symbols with expression

```
⇝ (5.0 + sqrt ((pown -5.0 2) - 4.0*2.0*3.0)) / 2.0*2.0
⇝ ... ⇝ 6.0
```

Example

» Analyze the evaluation tree of an expression:

Step-by-step application of the two rules

```
let num = (let ten = 5 + 1 in ten + ten + 1) in
num * 2
```

» What is the shortest path?

Which path will the F# compiler follow?

Homework #1

» Find an expression where ***evaluating the value of symbols first*** is better and another expression where ***replacing symbols with expressions*** is better .

Better means smaller number of reduction steps (no unnecessary calculations).

Homework #2

» Write expression that prints "yes" if the value of **n** is less than 10 and "no" otherwise. Without using **if** and **match** construct.