Standard ML

Podivný funkcionální jazyk

Augustin Žídek

augustin<at>zidek<dot>eu

22. dubna 2013

Úvod

Úvod

- Robin Milner 1970, Edinburgh

Úvod

- Robin Milner 1970, Edinburgh
- Metajazyk pro strojové dokazování vět

Úvod

- Robin Milner 1970, Edinburgh
- Metajazyk pro strojové dokazování vět
- Funkcionální jazyk, lambda calculus (anonymní funkce), silné typy, algebraické datatypy, pattern matching

Úvod

- Robin Milner 1970, Edinburgh
- Metajazyk pro strojové dokazování vět
- Funkcionální jazyk, lambda calculus (anonymní funkce), silné typy, algebraické datatypy, pattern matching
- Není čistě funkcionální – imperativní prvky

Úvod

- Robin Milner 1970, Edinburgh
- Metajazyk pro strojové dokazování vět
- Funkcionální jazyk, lambda calculus (anonymní funkce), silné typy, algebraické datatypy, pattern matching
- Není čistě funkcionální – imperativní prvky
- Ovlivnil: Miranda, Haskell, Clojure, Erlang, Java 8

Úvod

- Robin Milner 1970, Edinburgh
- Metajazyk pro strojové dokazování vět
- Funkcionální jazyk, lambda calculus (anonymní funkce), silné typy, algebraické datatypy, pattern matching
- Není čistě funkcionální – imperativní prvky
- Ovlivnil: Miranda, Haskell, Clojure, Erlang, Java 8
- Standard ML `www.smlnj.org/`
- Moscow ML `www.itu.dk/~sestoft/mosml.html`

Rychlokurz syntaxe

```
1 val a = ~5;
```

Rychlokurz syntaxe

```
1 val a = ~5;
```

```
1 val b = ((a * 2 - 3) div 7) mod 3;
```

Rychlokurz syntaxe

```
1 val a = ~5;
```

```
1 val b = ((a * 2 - 3) div 7) mod 3;
```

```
1 val c = ~0.42;
2 val d = c / 3.0;
```

Rychlokurz syntaxe

```
1 val a = ~5;
```

```
1 val b = ((a * 2 - 3) div 7) mod 3;
```

```
1 val c = ~0.42;
2 val d = c / 3.0;
```

Rychlokurz syntaxe

```
1 val a = ~5;
```

```
1 val b = ((a * 2 - 3) div 7) mod 3;
```

```
1 val c = ~0.42;
2 val d = c / 3.0;
```

Typy

```
1 val e = a + c; (* Zle *)
2 val e = real(a) + c; (* Dobre *)
```

Co ML odpovídá

```
1  > val a = ~5 : int
2  > val b = 1 : int
3  > val c = ~0.42 : real
4  > val d = ~0.14 : real
5  File "Test.sml", line 7, characters 12-13:
6  ! val e = a + c; (* Zle *)
7  !                ^
8  ! Type clash: expression of type
9  !   real
10 ! cannot have type
11 !   int
```

Pole

```
1 val empty = []; (* Nebo nil *)
2 val arr1 = [1, 2, ~3];
3 val arr2 = [4, 5];
```

Pole

```
1 val empty = []; (* Nebo nil *)
2 val arr1 = [1, 2, ~3];
3 val arr2 = [4, 5];
```

```
1 val arr3 = 8 :: arr1;
2 val concat = arr1 @ arr3;
```

Pole

```
1 val empty = []; (* Nebo nil *)
2 val arr1 = [1, 2, ~3];
3 val arr2 = [4, 5];
```

```
1 val arr3 = 8 :: arr1;
2 val concat = arr1 @ arr3;
```

```
1 val arr4 = [7] @ arr1;
2 val arr5 = 7 :: arr1;
```

Podmínky

```
1 if <expr1> then
2   <expr2>
3 else
4   <expr3>;
```

První program

Rekurzivní faktoriál

```
1 fun fac(0) = 1
2   | fac(n) = n * fac(n - 1);
3
4 val fac5 = fac(5);
```

První program

Rekurzivní faktoriál

```
1 fun fac(0) = 1
2   | fac(n) = n * fac(n - 1);
3
4 val fac5 = fac(5);
```

```
1 fun fac(0 : int) : int = 1
2   | fac(n : int) : int = n * fac(n - 1);
3
4 val fac5 = fac(5);
```

První program

Rekurzivní faktoriál

```
1 fun fac(0) = 1
2   | fac(n) = n * fac(n - 1);
3
4 val fac5 = fac(5);
```

```
1 fun fac(0 : int) : int = 1
2   | fac(n : int) : int = n * fac(n - 1);
3
4 val fac5 = fac(5);
```

ML odpovídá

```
1 > val fac = fn : int -> int
2 > val fac5 = 120 : int
```

Tail recursion

Rekurzivní faktoriál

```
1 fun fac(0) = 1
2   | fac(n) = n * fac(n - 1);
```

Tail recursion

Rekurzivní faktoriál

```
1  fun fac(0) = 1
2    | fac(n) = n * fac(n - 1);
```

```
1  fac(5)
2  5 * fac(4)
3  5 * (4 * fac(3))
4  5 * (4 * (3 * fac(2)))
5  5 * (4 * (3 * (2 * fac(1))))
6  5 * (4 * (3 * (2 * 1)))
7  5 * (4 * (3 * 2))
8  5 * (4 * 6)
9  5 * 24
10 120
```

Tail recursion

Tail-rekurzivní faktoriál

```
1 fun fac(0, prod) = prod
2   | fac(n, prod) = fac(n - 1, n * prod);
```

Tail recursion

Tail-rekurzivní faktoriál

```
1 fun fac(0, prod) = prod
2   | fac(n, prod) = fac(n - 1, n * prod);
```

```
1 > val fac = fn : int * int -> int
```

Tail recursion

Tail-rekurzivní faktoriál

```
1 fun fac(0, prod) = prod
2   | fac(n, prod) = fac(n - 1, n * prod);
```

```
1 > val fac = fn : int * int -> int
```

```
1 fac(5, 1)
2 fac(4, 5)
3 fac(3, 20)
4 fac(2, 60)
5 fac(1, 120)
6 120
```

Java vs ML

ML kompilátor umí odhalit tail-recursion a optimalizovat ji jako iteraci

Java vs ML

ML kompilátor umí odhalit tail-recursion a optimalizovat ji jako iteraci

```
1 fun fac(0, prod) = prod
2   | fac(n, prod) = fac(n - 1, n * prod);
```

```
1 int fac(int n, int prod) {
2     if (n == 1) {
3         return prod;
4     }
5     else {
6         return fac(n - 1, n * prod);
7     }
8 }
```

Java vs ML

ML

```
1 fac(3, 1)
2 fac(2, 3)
3 fac(1, 6)
4 6
```

Java

```
fac(3, 1)
return (fac(2, 3))
return (return (fac(1, 6))
return (return 6)
return 6
6
```

Cvičení

```
1 fun tst [] = true
2   | tst (x::l) = false;
3
4 val isTst = tst([1, 2, 3, 4]);
```

Cvičení

```
1 fun tst [] = true
2   | tst (x::l) = false;
3
4 val isTst = tst([1, 2, 3, 4]);
```

```
1 > val tst = fn : a list -> bool
```

Funkcionální nádhera

V ML lze funkci:

Funkcionální nádhera

V ML lze funkci:

1. Předat jako argument

Funkcionální nádhera

V ML lze funkci:

1. Předat jako argument
2. Vrátit z funkce

Funkcionální nádhera

V ML lze funkci:

1. Předat jako argument
2. Vrátit z funkce
3. Dát do pole

Funkcionální nádhera

V ML lze funkci:

1. Předat jako argument
2. Vrátit z funkce
3. Dát do pole
4. **Nelze** testovat shodnost

Bezejmenné/anonymní/lambda funkce

Bezejmenné/anonymní/lambda funkce

```
1 (fn n => n * 2);
```

Bezejmenné/anonymní/lambda funkce

```
1 (fn n => n * 2);
```

```
1 (fn n => n * 2) 17;
2 > val it = 34 : int
```

Curried functions

Curried functions

Obecná násobící funkce

```
1 val multiply = (fn a => (fn b => a * b));
2 > val multiply = fn: int -> (int -> int)
```

Curried functions

Obecná násobící funkce

```
1 val multiply = (fn a => (fn b => a * b));
2 > val multiply = fn: int -> (int -> int)
```

```
5 val multiply2 = multiply 2;
6 val test = multiply2 21;
```

Curried functions

Obecná násobící funkce

```
1 val multiply = (fn a => (fn b => a * b));
2 > val multiply = fn: int -> (int -> int)
```

```
7 val multiply2 = multiply 2;
8 val test = multiply2 21;
```

```
1 > val multiply2 = fn : int -> int
2 > val test = 42 : int
```

Curried functions – zkratkovitý zápis

Curried functions – zkratkovitý zápis

```
1 fun funName arg1 arg2 = ...;
2
3 fun funName =
4   (fn arg1 => (fn arg2 => ...));
```

Curried functions – zkratkovitý zápis

```
1 fun funName arg1 arg2 = ...;
2
3 fun funName =
4   (fn arg1 => (fn arg2 => ...));
```

```
1 val multc = (fn a => (fn b => a * b));
2 fun mults a b = a*b;
```

```
1 > val multc = fn : int -> int -> int
2 > val mults = fn : int -> int -> int
```

Curried functions – zkratkovitý zápis

```
1 fun funName arg1 arg2 = ...;
2
3 fun funName =
4   (fn arg1 => (fn arg2 => ...));
```

```
1 val multc = (fn a => (fn b => a * b));
2 fun mults a b = a*b;
```

```
1 > val multc = fn : int -> int -> int
2 > val mults = fn : int -> int -> int
```

```
1 val multiply3 = mults 3;
2 > val multiply3 = fn : int -> int
```

Funkce `map`

Funkce `map`

```
1 fun map f [] = []
2   | map f (x::xs) = f(x) :: map f xs;
```

Funkce `map`

```
1 fun map f [] = []
2   | map f (x::xs) = f(x) :: map f xs;
```

```
1 > val (a, b) map =
2     fn : (a -> b) -> a list -> b list
```

Funkce `map`

```
1 val lst = [1,2,3,4,5,6,7,8,9,10];
```

Funkce `map`

```
1 val lst = [1,2,3,4,5,6,7,8,9,10];
```

```
1 val powList =  map (fn n => n * n) lst;
```

Funkce `map`

```
1 val lst = [1,2,3,4,5,6,7,8,9,10];
```

```
1 val powList =  map (fn n => n * n) lst;
```

```
1 val powry =  map (fn n => n * n);
2 > val powry = fn : int list -> int list
```

Funkce `exists`

```
1 fun exists p [] = false
2   | exists p (x::xs) =
3     (p x) orelse exists p xs;
```

Funkce `exists`

```
1 fun exists p [] = false
2   | exists p (x::xs) =
3     (p x) orelse exists p xs;
```

```
1 > val a exists =
2     fn : (a -> bool) -> a list -> bool
```

Ukázka `exists`

```
1 fun odd(n) = (n mod 2 = 1);
2
3 val lst2 = [2,4,6,8,9]; (* Milion prvku *)
```

Ukázka `exists`

```
1 fun odd(n) = (n mod 2 = 1);
2
3 val lst2 = [2,4,6,8,9]; (* Milion prvku *)
```

```
1 exists odd lst2;
2 > val it = true : bool
```

Funkce `filter`

```
1 fun filter p [] = []
2   | filter p (x::xs) =
3     if (p x) then
4       x :: filter p xs
5     else
6       filter p xs;
```

Funkce `filter`

```
1 fun filter p [] = []
2   | filter p (x::xs) =
3     if (p x) then
4       x :: filter p xs
5     else
6       filter p xs;
```

```
1 > val a filter =
2     fn : (a -> bool) -> a list -> a list
```

Ukázka `filter`

```
1 fun odd(n) = (n mod 2 = 1);
2
3 val lst3 = [1,2,3,4,5,6,7,8,9,10];
```

Ukázka `filter`

```
1 fun odd(n) = (n mod 2 = 1);
2
3 val lst3 = [1,2,3,4,5,6,7,8,9,10];
```

```
1 filter odd lst3;
2 > val it = [1, 3, 5, 7, 9] : int list
```

Lazy lists

Lazy lists

- Seznamy nekonečné délky

Lazy lists

- Seznamy nekonečné délky
- Prvky se počítají jen *na požádání*

Lazy lists

- Seznamy nekonečné délky
- Prvky se počítají jen *na požádání*

```
1 datatype 'a sequence = Nil
2 | Stream of 'a * (unit -> 'a sequence);
3
4 Stream(x, f);
```

Lazy lists

- Seznamy nekonečné délky
- Prvky se počítají jen *na požádání*

```
1 datatype 'a sequence = Nil
2 | Stream of 'a * (unit -> 'a sequence);
3
4 Stream(x, f);
```

```
1 fun head(Nil) = Nil
2   | head(Stream(x, _)) = x;
3
4 fun tail(Nil) = Nil
5 |   tail(Stream(_, f)) = f();
```

Ukázka lazy listu

```
1 fun from k = Stream(k, fn() => from(k+1));
```

Ukázka lazy listu

```
1 fun from k = Stream(k, fn() => from(k+1));
```

```
1 from 1;
2 > val it = Stream(1, fn) : int seq
3 tail(it);
4 > val it = Stream(2, fn) : int seq
5 tail(it);
6 > val it = Stream(3, fn) : int seq
7 tail(it);
8 > val it = Stream(4, fn) : int seq
```

Více informací

www.cl.cam.ac.uk/teaching/1112/FoundsCS/materials.html

Dotazy

augustin<at>zidek<dot>eu

Děkuji za pozornost