Języki platformy .NET

Paweł Różański Wojciech Walewski

21 października 2005

Chcielibyśmy zdążyć dziś opowiedzieć o:

- SML / NJ, SML.NET,
- OCaml, OCamlIL, F#,
- Nemerle,

a następnym razem zdążyć o:

- SmallTalk, #SmallTalk, S#,
- Java vs. J#,
- Python, IronPython,
- Boo,
- Lua.

Język SML

Język SML

```
fun sublist (x::xs) =
  let
     fun insert d nil = nil
       | insert d (ys::yss) = [ys] @ [[d] @ ys] @ insert d yss
  in
     insert x (sublist xs)
  end
  | sublist nil = [[]];
```

```
fun quick (xs : int list) =
let
  fun quicker (xs, ys) = case xs of
      [] => ys
    | [x] => x::ys
    | a::bs =>
      let fun partition (left, right, []) =
                quicker (left, a::quicker (right, ys))
            | partition (left, right, x::xs) = if x < a then
                    partition (x::left, right, xs)
                else partition (left, x::right, xs)
      in
          partition([], [], bs)
      end
in
  quicker (xs, [])
end
```

Porównanie składni SMLa i OCamla

Porównanie składni SMLa i OCamla

Literały

SML

```
- 3;
> val it = 3 : int

- 3.141;
> val it = 3.141 : real

- "Hello world";
> val it = "Hello world" : string

- #"J";
> val it = #"J" : char

- true;
> val it = true : bool
```

OCaml

```
# 3;;
- : int = 3

# 3.141;;
- : float = 3.141

# "Hello world";;
- : string = "Hello world"

# 'J';;
- : char = 'J'

# true;;
- : bool = true
```

SML

```
- ();
> val it = () : unit

- (true,"hi");
> val it = (true, "hi") : bool * string

- [1, 2, 3];
> val it = [1, 2, 3] : int list

- #[1, 2, 3];
> val it = #[1, 2, 3] : int vector
Standard nie specyfikuje wektorów, ale
większość implementacji je wspiera - bibl.

Nie ma literałów tablicowych - bibl.
```

OCaml

```
# ();;
- : unit = ()

# (true,"hi");;
- : bool * string = true, "hi"

# [1; 2; 3];;
- : int list = [1; 2; 3]

Nie ma wektorów - używa się tablic

# [|1; 2; 3|];;
- : int array = [|1; 2; 3|]
```

Wyrażenia

SML

```
~3*(1+7) div 2 mod 3

~1.0/2.0 + 1.9*x

a orelse b andalso c



fn f => fn x => fn y => f(x, y)



fn 0 => 0
 | n => 1
```

OCaml

```
-3*(1+7)/2 mod 3

-1.0 /. 2.0 +. 1.9 *. x

a || b && c
lub (niezalecane)
a or b & c

fun f -> fun x -> fun y -> f(x, y)
lub
fun f x y -> f(x,y)

function 0 -> 0
         | n -> 1
```

Kontrola przepływu

SML

```
if 3 > 2 then "X" else "Y"

if 3 > 2 then print "hello" else ()


while true do
   print "X"


Nie ma pętli for
 - używamy rekursji lub while
```

OCaml

```
if 3 > 2 then "X" else "Y"

if 3 > 2 then print_string "hello"
Uwaga: wyrażenie musi mieć typ unit

while true do
   print_string "X"
done

for i = 1 to 10 do
   print_endline "Hello"
done
```

Deklaracje wartości

SML

```
val name = expr

fun f x y = expr

val rec fib = fn n =>
   if n < 2
   then n
   else fib(n-1) + fib(n-2)
lub
fun fib n =
   if n < 2
   then n
   else fib(n-1) + fib(n-2)
```

OCaml

```
let name = expr

let f x y = expr

let rec fib = fun n ->
   if n < 2
   then n
   else fib(n-1) + fib(n-2)
lub
let rec fib n =
   if n < 2
   then n
   else fib(n-1) + fib(n-2)
```

Deklaracje typów

SML

```
type t =  int -> bool

type ('a,'b) assoc_list = ('a * 'b) list

datatype 'a option = NONE | SOME of 'a

datatype t = A of int | B of u
withtype u = t * t

datatype v = datatype t
```

OCaml

```
type t = int -> bool

type ('a,'b) assoc_list = ('a * 'b) list

type 'a option = None | Some of 'a

type t = A of int | B of u
and  u = t * t

type v = t = A of int | B of u
```

Dopasowywania wzorców — matching

SML

```
fun getOpt(NONE, d) =  d
  | getOpt(SOME x, _) = x

fun getOpt(opt, d) =
   case opt of
        NONE => d
      | SOME x => x
fun take 0 xs = []
  | take n nil = raise Empty
  | take n (x::xs) = x :: take (n-1) xs


Nie ma strażników, ale są if-y :)
```

OCaml

```
let get_opt = function
     (None, d) -> d
   | (Some x, _) -> x
let get_opt(opt, d) =
   match opt with
        None -> d
      | Some x -> x
let rec take n xs =
   match n, xs with
        0, xs -> []
      | n, [] -> failwith "take"
      | n, x::xs -> x :: take (n-1) xs
let rec fac = function
     0 -> 1
   | n when n > 0 -> n*fac(n - 1)
   | _ -> raise Hell
```

Referencje

SML

```
val r = ref 0

!r



r := 1



fun f(ref x) = x

r1 = r2
r1 <> r2
```

OCaml

```
let r = ref 0

!r
lub
r.contents

r := 1
lub
r.contents <- 1

let f {contents=x} = x

r1 == r2
r1 != r2
```

Deklaracje lokalne

SML

```
fun p (x,y) =
   let
      val xx = x * x
      val yy = y * y
   in
      Math.sqrt(xx + yy)
   end
```

```
local
   fun sqr x = x * x
in
   fun p (x,y) = Math.sqrt(sqr x + sqr y)
end
```

OCaml

```
let pt x y =
   let xx = x *. x in
   let yy = y *. y in
   sqrt(xx +. yy)
```

```
Nie ma local - używa się let, deklaracji
globalnych lub modułów pomocniczych
```

Deklaracje lokalne cd.

SML

```
let
   structure X = F(A)
in
   X.value + 10
end
```

Nie ma deklaracji struktur let in,
ale niektóre implementacje je wspierają

```
let
   open M
   datatype t = A | B
   exception E
in
   expr
end
```

OCaml

```
let module X = F(A) in
   X.value + 10
```

Eksperymentalne rozszerzenie języka

Nie ma lokalnego open, type,
ani deklaracji wyjątków - używa się
deklaracji globalnych lub powyższego
let module

F# a OCaml - główne różnice

F# a OCaml - główne różnice

- Moduły, obiekty w stylu OCamla, etykiety i opcjonalne argumenty nie są wspierane, podobno z powodu żadkiego ich używania.
- Niektóre z identyfikatorów są teraz słowami kluczowymi, np. `upcast`, `downcast`, `null`, `inline`. Słowo `using` zarezerwowano na przyszłość.
- Nie można używać wyrażeń na najwyższym poziomie modułów. W OCamlu można napisać:

```
let x = 3;;
Printf.printf "x + x = %d" (x + x)
```

co wydaje się autorom języka niejasne. W F# można to zapisać na dwa sposoby:

```
let x = 3
do Printf.printf "x + x = %d" (x + x)
let x = 3
let () = Printf.printf "x + x = %d" (x + x)
```

- Pomniejsze różnice w parsowaniu wyrażeń. Np. `!x.y.z` jest teraz rozumiane jako `!(x.y.z)`, a nie jako `(!x).y.z`
- Stringi są teraz w Unicode i niezmienne
- Deklaracja "include" nie jest już wspierana
- Definicje z tą samą nazwą nie są dozwolone w module ani w typie modułu, np.

```
let x = 1
let x = 3
```

- Ograniczenie modułu sygnaturą nie może ograniczyć poliformiczności jego elementów, np. jeśli moduł deklaruje

```
let f x = x
```

a sygnatura deklaruje

```
val f : int -> int
```

to f musi być explicité mniej polimorficzne:

```
let f (x : int) = x
```

SML.NET

SML.NET

- Kompilator napisany w SML / NJ,
- Zgodny ze standardem SML'97,
- Kompilacja przyrostowa (incremental),
- Automatyczna analiza zależności,
- Produkuje weryfikowalny CLR IL,
- Rozszerza Standard ML. Potrafi zarówno korzystać, jak i produkować klasy, interfejsy, delegatów itd. dla .NET,
- Optymalizuje od razu cały program, bądź bibliotekę.
- Wstrzymywanie (odraczanie) większości kompilacji aż do momentu po linkowaniu (pierwsze uruchomienie jest wooolne)
- Brak środowiska interaktywnego
- Ujawnione na zewnątrz interfejsy aplikacji i bibliotek mogą odnosić się tylko do typów CLR. Odnosi się to nawet do interfejsów udostępnianych innym programom SML.NET, chyba że linkowanie odbywa się w czasie kompilacji

Rozszerzenia SML.NET dla .NET

Rozszerzenia SML.NET dla .NET

- Przestrzenie nazw, klasy i zagnieżdżanie
  - Przestrzenie nazw (np. `System`) i podprzestrzenie nazw (np. `System.Drawing`) to struktury i podstruktury
  - Klasy to identyfikatory typów, wartości typów funkcyjnych, oraz struktury zawierające wiązania wartości (value bindings) odpowiadające statycznym polom i metodom, i wiązania podstruktur dla klas zagnieżdżonych
  - Strukturami można manipulować: związać na nowo, przekazywać funktorom, ograniczać sygnaturą i otwierać (`open` odpowiada `using` w C#)

Przykład

- Typy
  - Typy wbudowane

Typ w .NET	Typ w C#	Typ w SML.NET
System.Boolean	bool	bool
System.Byte	byte	Word8.word
System.Char	char	char
System.Double	double	real
System.Single	float	Real32.real
System.Int32	int	int
System.Int64	long	Int64.int
System.Int16	short	Int16.int
System.SByte	sbyte	Int8.int
System.String	string	string
System.UInt16	ushort	Word16.word
System.UInt32	uint	word
System.UInt64	ulong	Word64.word
System.Exception	System.Exception	exn
System.Object	object	object

- Typy cd.
  - Nazwane typy .NET takie jak klasy, typy wartości, wyliczeniowe, interfejsy czy delegaci. Można się do nich odwoływać przy użyciu składni jak w C#.
  - Typy tablicowe. Zachowują się jak w .NET: rozmiar ustalany w czasie tworzenia, indeksowanie od zera, równość oparta na identyczności, a nie wartości, rzucany wyjątek podczas dostępu poza zakres. .NETowy `System.IndexOutOfRangeException` to w SML.NET `Subscript`.
  - Wartości `null`. W .NET operacje jak wywołanie metody, dostęp do pola itp. rzucają `NullReferenceException`, jeśli ich główny operand jest `null`. W SMLu nie ma takiego pojęcia, jak wartość `null`. Dlatego używa się

    ```
    datatype 'a option = NONE | SOME of 'a
    ```

- Typy cd.
  - Wartości `null` cd. Do wyłuskania wartości służy funkcja `valOf`, która ma typ `'a option -> 'a` i rzuca wyjątek `Option` dla `NONE`.
  - Wartości typu referencyjnego, które przekraczają granicę między SML.NET a .NET są interpretowane jako wartości typu `option`, np. metoda

    ```
    public static string Join(string separator, string[] value);
    ```

    klasy `System.String` jest mapowana na funkcję o sygnaturze

    ```
    val Join : string option*string option array option
                    -> string option
    ```

Przykład

- Typy cd.
  - Typy *interop* to wszystkie powyższe typy i nowe typy SML.NET. Typy *interop* mogą być używane w rozszerzeniach SMLa, takich jak rzutowania, niejawne koercje itd.
- Obiekty
  - Tworzenie:

```
val fonts = map System.Drawing.Font
                    [("Times", 10.0), ("Garamond", 12.0)]
```

  - Tworzenie i wywoływanie obiektów delegata. Typy delegatów są mapowane na dwa wiązania SMLa: sam typ, tak jak typy klas, oraz wiązanie funkcji konstruktora delegata, które bierze funkcję SMLową jako jedyny argument. Przykład:

- Obiekty cd.
  - Tworzenie i wywoływanie obiektów delegata. W C# mamy

    ```
    public delegate int BinaryOp(int x, int y);
    ```

    jest w SMLu typem `BinaryOp` i funkcją o sygnaturze

    ```
    val BinaryOp : ((int*int)->int) -> BinaryOp
    ```

    używaną do skonstruowania obiektu delegata z funkcji SMLa, np.

    ```
    val adder = BinaryOp(op+)
    ```

    Aby zastosować delegata, używamy metodę `Invoke` (C#-owy cukier)

    ```
    val 3 = adder.#Invoke(1, 2)
    ```

Przykład

- Obiekty cd.
  - Rzutowania i wzorce rzutowań. Wprowadzono nową składnię dla C#-owych rzutowań:

```
open System.Drawing System.Color
val c = SolidBrush( get_Red()) :> Brush
```

Rzutowań można używać jako dopasowania wzorca, np.

```
fun test x = (do_some_stuff x)
    handle y :> ArithmeticException => f y
         | _ :> DivideByZeroException => g x
```

- Pola, metody, propercje
  - Statyczne pola i metody klasy są mapowane na wiązania funkcji i wartości w strukturze odpowiadającej tej klasie (np. statyczne pole `PI` klasy `System.Math` odpowiada wiązaniu wartości `PI` w strukturze `System.Math`)
  - Dla nie-statycznych pól i metod wprowadzono nową składnię: `exp.#name`
  - Propercje to w C# cukier syntaktyczny: dla propercji *P* istnieją funkcje `get_`*P* i `set_`*P*
  - Dla pól niezmiennych (`readonly` i `const` w C#) wprowadzono słowo kluczowe `option`, dla oznaczenia możliwości wartości `null`, np.

    ```
    public static string language_name = "C#";
    ```

    ma typ `string option`
  - Pola zmienne traktuje się tak, jakby były typem referencyjnym (`!`, `:=`)

Przykład

- Typy wartości
  - Odpowiednikiem ich są C#-owe struktury. Ponieważ są one zamknięte (*sealed*), nie potrzebują być *boxed*, by zapewnić jednolitą reprezentację, ani nieść deskryptorów typu, służących do rzutowań w dół i wywoływania metod wirtualnych
  - W językach funkcjonalnych wartości powinny być efektywnie niezmienne: wywołanie czegokolwiek na typie wartości nie może go zmieniać. W SML.NET takie wywołanie najpierw kopiuje wartość, a potem przekazuje adres kopii metodzie jako adres `this`. Jeśli `p : Pair`, to wywołania:

    ```
    p.#invert()
    let val r = p.#x in r:= 1 end
    ```

    nie zmieniają `p`. Jeśli nam na tym zależy, to trzeba to zrobić tak:

    ```
    (ref p).#invert()
    let val r = ref (Pair(1, 2)) in r.#invert(); !r end
    ```

Przykład

- Typy wartości cd.
  - *Boxing*. W SML.NET opakowanie wartości sprowadza się do rzutowania w górę (działa również dla typów prymitywnych)

    ```
    p :> System.Object
    1 :> System.Object
    ```
  - *Unboxing*, w SML.NET jest realizowany poprzez rzutowanie w dół, może rzucać `System.InvalidCastException`. Dla `obj : System.Object`

    ```
    obj :> Pair
    obj :> int
    ```
- Definiowanie nowych typów
  - Klasy
  - Interfejsy

Przykłady