Obiektowy Caml

Paweł Boguszewski

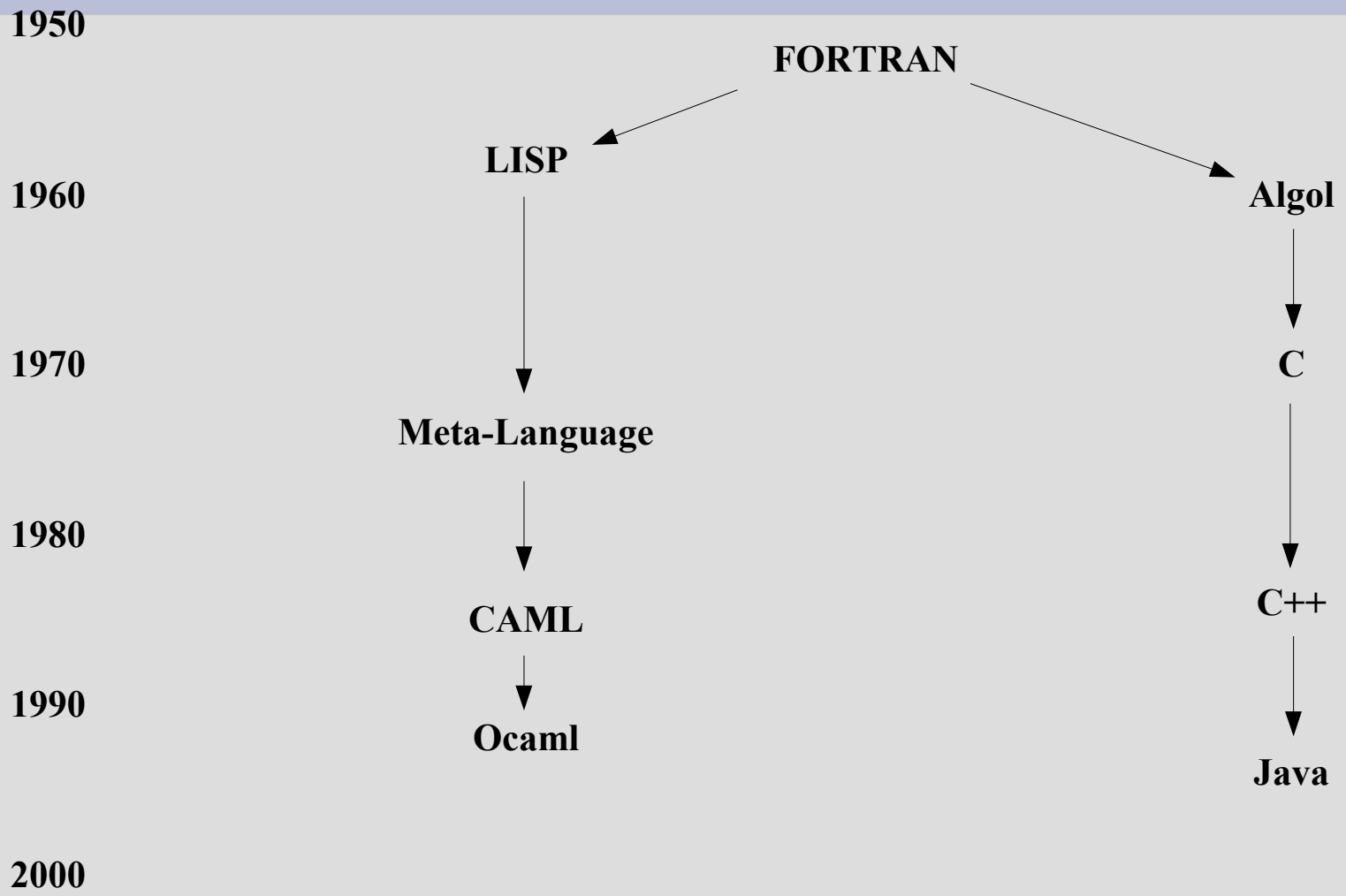
Plan prezentacji

- Charakterystyka języka
- Składnia
- Obiektowość w OCamlu
- Wyjątki
- Standardowe Moduły
- Narzędzia wspomagające
- Bibliografia

OCaml

- OCaml jest obiektowym, funkcyjnym językiem programowania stworzonym z myślą o szybkości i prostocie wyrażania programów.
- Wspiera zarówno funkcyjny, imperatywny jak i obiektowy styl programowania.
- Istnieje zarówno w wersji interpretowanej jak i kompilowanej.
- Jest statycznie typowany z inferencją typów.
- Posiada możliwość definiowania własnych typów.
- Automatyczne zarządzanie pamięcią.
- Pattern Matching.
- Możliwość rozłącznej kompilacji.

Pochodzenie



Historia

- 1987 – pierwsza implementacja stworzona głównie przez Ascander'a Suarez'a
- 1991 – powstaje Caml Light
- 1995 – Xavier Leroy tworzy Caml Special Light
- 1996 – powstaje pierwsza implementacja Ocaml'a
- 2000 – rozszerzenie języka do obecnego stanu, dokonane przez Jacques'a Garrigue

Składnia

- Podstawy
- Typy danych
- Funkcje
- Rekordy
- Programowanie imperatywne

Podstawy

```
#1+2*3;;
- : int = 7

#let pi = 4.0 *. atan 1.0;;
val pi : float = 3.1415926535897312

#let square x = x *. x;;
val square : float -> float = <fun>

#square(sin pi) +. square(cos pi);;
- : float = 1.
```

Typy danych

```
#(1 < 2) = false;;
- : bool = false

#'a';;
- : char = 'a'

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

# ["Hello"; "World"];;
- : string list = ["Hello"; "World"]
```

Funkcje

```
#let rec fib n =
   if n < 2 then 1 else fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>

#let rec sort lst =
   match lst with
     [] -> []
   | head :: tail -> insert head (sort tail)
 and insert elt lst =
   match lst with
     [] -> [elt]
   | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail
 ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
```

Funkcje jako wartości

```
#let compose f g = function x -> f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

#let cos2 = compose square cos;;
val cos2 : float -> float = <fun>

#List.map (function n -> n * 2 + 1) [0;1;2;3;4];;
- : int list = [1; 3; 5; 7; 9]
```

Rekordy

```
#type ratio = {num: int; denum: int};;
type ratio = { num : int; denum : int; }

#type number = Int of int | Float of float | Error;;
type number = Int of int | Float of float | Error

#type sign = Positive | Negative;;
type sign = Positive | Negative

#type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Programowanie imperatywne

- Pętle for i while

```
for i = 0 to len - 1 do
        (...)
done;
```

- Referencje

```
let i = ref j ...
```

- Tablice

```
[| 1, 2, 3, 4|]
```

System modułów

- Został stworzony, aby pogrupować ze sobą powiązane definicje i typy danych.
- Każdy moduł ma swoją przestrzeń nazw.
- Dla modułów istnieją sygnatury – interfejsy definiujące składniki modułu dostępne z zewnątrz.
- Moduły można parametryzować innymi modułami – funktory.
- Autonomiczność modułów pozwala użyć rozłącznej kompilacji do ich budowania.

Obiektowość w OCamlu

- OCaml dostarcza następujące mechanizmy do programowania obiektowego:
  - Klasy, obiekty
  - Parametryzowanie klas
  - Klasy abstrakcyjne
  - Wielodziedziczenie

Klasy i obiekty

Klasę definiujemy słowami kluczowymi:

class *nazwa parametry....* =
  object
    (...)
  end;;

W klasie mogą znajdować się definicje zmiennych lub metod.

val {mutable}? *nazwa* = *wyrażenie*

method *nazwa parametry...* = *(...)*

Klasy i obiekty

Obiekt danej klasy tworzymy:

```
#class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;

#let p = new point 7;;
```

Istnieje także możliwość definicji klasy w czasie tworzenia obiektu:

```
#let p =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
  end;;
```

Klasy i obiekty

- Metodę obiektu wywołujemy za pomocą znaku #

```
# p#print;;
```

- Jeżeli chcemy w danej klasie odwoływać się do własnych metod musimy dodać po słowie kluczowym „object” nazwę, po której będziemy odwoływali się do bieżącej klasy

```
class point =
  object (self)
      (...)
end;;
```

Metody wirtualne

- Podobnie jak w C++ w OCamlu możemy definiować wirtualne metody.
- Każda klasa posiadająca wirtualną metodę, musi być wirtualna.
- Nie można tworzyć instancji wirtualnych klas.

```
#class virtual abstract_point x_init =
  object (self)
    val mutable x = x_init
    method virtual get_x : int
    method get_offset = self#get_x - x_init
    method virtual move : int -> unit
  end;;
```

Dziedziczenie

- Klasy w OCamlu mogą dziedziczyć po sobie.
- Dziedziczenie zapisujemy poprzez słowo kluczowe **inherit**

```
#class colored_point x (c : string) =
  object
    inherit point x
    val c = c
    method color = c
  end;;
```

Dziedziczenie

- W przypadku gdy klasa ma metody prywatne, przy dziedziczeniu można je upublicznić w podklasach.
- Jeżeli chcemy zachować prywatność odziedziczonych metod należy ukryć je w sygnaturze.

Wielodziedziczenie

- Każda klasa może dziedziczyć z kilku innych.
- W przypadku gdy pojawia się konflikt nazw metod, obowiązuje zasada przysłaniania.
- Możemy tworzyć aliasy klas, po których dziedziczymy.

```
# class colored_point =
      object
            inherit point as p
            inherit color as c
end;;
```

Rzutowanie

- Podobnie jak w innych obiektowych językach w OCamlu każdy obiekt może zostać rzutowany na inny.
- Rzutować można jedynie na klasę po której dziedziczymy.
- Rzutowanie wykonujemy za pomocą słowa kluczowego „:>”

```
# let p = (cp : colored_point :> point ) ;;
```

Klasy parametryzowane

- Klasę możemy sparametryzować.
- Parametrem klasy może być zarówno typ danych, jak i obiekt.

```
#class ['a] circle (c : 'a) =
  object
    constraint 'a = #point
    val mutable center = c
    method center = center
    method set_center c = center <- c
    method move = center#move
  end;;
```

Klasy wzajemnie rekurencyjne

- Gdy klasy są wzajemnie powiązane, można zaimplementować je jako wzajemnie rekurencyjne (podobnie jak funkcje)

```
class window =
  object
    val mutable top_widget = (None : widget option)
    method top_widget = top_widget
  end
 and widget (w : window) =
  object
    val window = w
    method window = window
  end;;
```

Metody binarne

- Metodą binarną, nazywamy metodę przyjmującą jako parametr obiekt macierzystej klasy.

```
#class virtual comparable =
  object (_ : 'a)
    method virtual leq : 'a -> bool
  end;;

#class money (x : float) =
  object
    inherit comparable
    val repr = x
    method value = repr
    method leq p = repr <= p#value
  end;;
```

Mechanizm wyjątków

- Do sygnalizacji ewentualnych błędów ocaml dostarcza mechanizm wyjątków.
- Gdy zostanie podniesiony wyjątek, wykonanie programu jest przenoszone do najbliższego bloku obsługującego dany wyjątek.
- Wyjątki są stosowane zarówno przy programowaniu obiektowym jak i czysto funkcyjnym.
- Można używać dopasowywania wzorców przy obsłudze wyjątków.

Standardowe moduły

- OCaml posiada wbudowany rozbudowany zestaw modułów.
  - Tablice
  - Kolekcje
  - GC
  - Big_int
  - Moduły graficzne
  - Oo
  - Dynlink
  - Thread

Tablice

- Istnieje kilka modułów pozwalających tworzyć i operować na tablicach.
- Programista ma do dyspozycji tablice wielowymiarowe.
- Dla obliczeń numerycznych istnieją „duże” tablice przechowujące liczby.
- Zaimplementowano kilka algorytmów sortowania.

Kolekcje

- OCaml posiada moduły implementujące kolekcje znane min. z Javy lub C++ (STL).
  - HashTbl
  - Set
  - Map
  - Queue
  - Stack

GC

- Jest to moduł pozwalający kontrolować pracę oraz zbierać statystyki odśmiecacza.
- Istnieje możliwość zmiany rozmiaru stosu.
- Moduł daje kontrolę nad częstotliwością pracy odśmiecacza – można zmniejszyć narzut czasowy kosztem większej zajętości pamięci.

Oo

- Moduł umożliwia dokonywanie operacji na obiektach.
- Można zrobić kopię danego obiektu, oraz pobrać identyfikator obiektu (różny dla każdego wywołania programu)

Weak

- Jest to reprezentacja „słabych wskaźników” w OCamlu.
- Moduł daje nam do dyspozycji tablicę z kolekcją wskaźników, które możemy dowolnie ustawiać.
- Element na który pokazuje wskaźnik może być w każdym momencie usunięty przez GC, dlatego wskaźniki nazywane są „słabe”.

Narzędzia wspomagające

- OCaml zawiera szereg narzędzi wspomagających programowanie, min.:
  - ocamllex
  - ocamlprof
  - ocamlopt
  - ocamldoc
  - ocamldebug
  - Wtyczki do edytorów tekstowych

Bibliografia

- http://www.ocaml.org
- http://merjis.com/developers/ocaml_tutorial/
- http://www.bogonomicon.org/bblog/ocaml.sxi

Dziękuję

Pytania?