Implementacja języka funkcyjnego z rodziny ML z wykorzystaniem infrastruktury LLVM

Mateusz Lewko

6 września 2018

Spis treści

Obecnie

- Jest wiele języków z rodziny ML

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny
  - Częściową aplikację

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny
  - Częściową aplikację
  - Zagnieżdżone funkcje

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny
  - Częściową aplikację
  - Zagnieżdżone funkcje
  - Funkcje wyższych rzędów

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny
  - Częściową aplikację
  - Zagnieżdżone funkcje
  - Funkcje wyższych rzędów
  - System modułów (OCaml, SML) lub obiektowe klasy (F#)

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - Polimorfizm parametryczny
  - Częściową aplikację
  - Zagnieżdżone funkcje
  - Funkcje wyższych rzędów
  - System modułów (OCaml, SML) lub obiektowe klasy (F#)
  - Trwałe rekordy, funkcje wzajemnie rekurencyjne, inferencja typów, algebraiczne typy danych, itp.

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - **Polimorfizm parametryczny $\Rightarrow$ Opakowywanie argumentów we wskaźnik**
  - Częściową aplikację
  - Zagnieżdżone funkcje
  - Funkcje wyższych rzędów
  - System modułów (OCaml, SML) lub obiektowe klasy (F#)
  - Trwałe rekordy, funkcje wzajemnie rekurencyjne, inferencja typów, algebraiczne typy danych, itp.

Obecnie

- Jest wiele języków z rodziny ML
- Zawierają
  - **Polimorfizm parametryczny $\Rightarrow$ Opakowywanie argumentów we wskaźnik**
  - Częściową aplikację
  - Zagnieżdżone funkcje
  - Funkcje wyższych rzędów
  - **System modułów (OCaml, SML)** lub obiektowe klasy (F#)
  - Trwałe rekordy, funkcje wzajemnie rekurencyjne, inferencja typów, inferencja typów, algebraiczne typy danych, itp.

Motywacja

- Wady opakowywania we wskaźnik

Motywacja

- Wady opakowywania we wskaźnik
  - Narzut pamięciowy — nawet 3x w przypadku typu int

Motywacja

- Wady opakowywania we wskaźnik
  - Narzut pamięciowy — nawet 3x w przypadku typu int
  - Narzut czasowy
    - Automatyczne zarządzenie pamięcią
    - Konieczność odczytywania pamięci ze sterty
    - Gorsze wykorzystanie pamięci cache

Motywacja

- Wady opakowywania we wskaźnik
  - Narzut pamięciowy — nawet 3x w przypadku typu int
  - Narzut czasowy
    - Automatyczne zarządzenie pamięcią
    - Konieczność odczytywania pamięci ze sterty
    - Gorsze wykorzystanie pamięci cache
- Wady systemu modułów

Motywacja

- Wady opakowywania we wskaźnik
  - Narzut pamięciowy — nawet 3x w przypadku typu int
  - Narzut czasowy
    - Automatyczne zarządzenie pamięcią
    - Konieczność odczytywania pamięci ze sterty
    - Gorsze wykorzystanie pamięci cache
- Wady systemu modułów
  - Brak możliwości przeładowaniu operatorów i funkcji (np. dla różnych typów numerycznych)

Motywacja

- Wady opakowywania we wskaźnik
  - Narzut pamięciowy — nawet 3x w przypadku typu int
  - Narzut czasowy
    - Automatyczne zarządzenie pamięcią
    - Konieczność odczytywania pamięci ze sterty
    - Gorsze wykorzystanie pamięci cache
- Wady systemu modułów
  - Brak możliwości przeładowaniu operatorów i funkcji (np. dla różnych typów numerycznych)
  - Nietrywialne w implementacji i skomplikowane w użyciu

Język MonoML

Język MonoML

- **Polimorfizm parametryczny $\longrightarrow$ Monomorfizacja**

Język MonoML

- **Polimorfizm parametryczny $\longrightarrow$ Monomorfizacja**
- **Częściową aplikację $\longrightarrow$ Bazowana na modelu push/enter**

Język MonoML

- **Polimorfizm parametryczny $\longrightarrow$ Monomorfizacja**
- **Częściową aplikację $\longrightarrow$ Bazowana na modelu push/enter**
- **Klasy typów (ad–hoc polimorfizm)**

Język MonoML

- **Polimorfizm parametryczny $\longrightarrow$ Monomorfizacja**
- **Częściową aplikację $\longrightarrow$ Bazowana na modelu push/enter**
- **Klasy typów (ad–hoc polimorfizm)**
- Zagnieżdżone funkcje
- Funkcje wyższych rzędów
- Trwałe rekordy, funkcje wzajemnie rekurencyjne, inferencja typów

Opis problemu

Kod do skompilowania

```
let twice f x = f (f x)
let _ =
  print_int   (twice identity 42  );
  print_float (twice identity 42.0)
```

Opis problemu

Kod do skompilowania

```
let twice f x = f (f x)
let _ =
  print_int   (twice identity 42  );
  print_float (twice identity 42.0)
```

Wygenerowany LLVM IR #1

```
define i32 @twice(i32 (i32)*, i32) {
    ...
}
```

Opis problemu

Kod do skompilowania

```
let twice f x = f (f x)
let _ =
  print_int   (twice identity 42  );
  print_float (twice identity 42.0)
```

Wygenerowany LLVM IR #2

```
define float @twice(float (float)*, float) {
    ...
}
```

Opis problemu

Kod do skompilowania

```
let twice f x = f (f x)
let _ =
  print_int   (twice identity 42  );
  print_float (twice identity 42.0)
```

Argumenty opakowane we wskaźnik

```
define i8* @twice(i8* (i8*)*, i8*) {
    ...
}
```

Moje podejście — Monomorfizacja

Po monomorfizacji

```
let twice_int (f : int -> int) (x : int) : int =
  f (f x)
let twice_float (f : float -> float) (x : float)
  : float = f (f x)

let _ =
  print_int   (twice_int identity_int 42  );
  print_float (twice_float identity_float 42.0)
```

Testy wydajnościowe
Cele

- Porównanie czasów wykonania funkcji polimorficznej i monomorficznej

Testy wydajnościowe
Cele

- Porównanie czasów wykonania funkcji polimorficznej i monomorficznej
  - w MonoMLu

Testy wydajnościowe
Cele

- Porównanie czasów wykonania funkcji polimorficznej i monomorficznej
  - w MonoMLu
  - w Haskellu, Javie i Standard MLu

Testy wydajnościowe
Cele

- Porównanie czasów wykonania funkcji polimorficznej i monomorficznej
  - w MonoMLu
  - w Haskellu, Javie i Standard MLu
- Narzut czasowy wywoływania funkcji w MonoMLu na tle innych języków

Testy wydajnościowe
Przygotowanie

Funkcja polimorficzna

```
let rec sum n (curr : 'a) (x : 'a) : 'a =
    if n = 0 then curr
    else sum (n - 1) (add curr x) x
```

Testy wydajnościowe
Przygotowanie

Funkcja polimorficzna

```
let rec sum n (curr : 'a) (x : 'a) : 'a =
    if n = 0 then curr
    else sum (n - 1) (add curr x) x
```

Monomorficzna

```
let rec sum n (curr : int) (x : int) : int =
    if n = 0 then curr
    else sum (n - 1) (add curr x) x
```

Testy wydajnościowe
Przygotowanie

Funkcja polimorficzna

```
sumPoly :: Num a => Int -> a ->  a -> a
sumPoly 0 curr _ = curr
sumPoly n curr x = sumPoly (n - 1) (curr + x) $! x
```

Monomorficzna

```
sumMono :: Int# -> Int# -> Int# -> Int#
sumMono 0# curr _ = curr
sumMono n curr x = sumMono (n -# 1#) (curr +# x) x
```

Testy wydajnościowe
Wyniki

Język	Wersja	Czas (ms)	σ	x
Haskell (GHC)	Mono	**39.1**	8.2	0.10
Haskell (GHC)	Poli	**696.8**	63.2	1.86

Testy wydajnościowe
Wyniki

Język	Wersja	Czas (ms)	σ	x
Haskell (GHC)	Mono	39.1	8.2	0.10
Haskell (GHC)	Poli	696.8	63.2	1.86
Java	Mono	**140.1**	65.7	0.37
Java	Poli	**564.9**	24.8	1.50

Testy wydajnościowe
Wyniki

Język	Wersja	Czas (ms)	σ	x
Haskell (GHC)	Mono	39.1	8.2	0.10
Haskell (GHC)	Poli	696.8	63.2	1.86
Java	Mono	140.1	65.7	0.37
Java	Poli	564.9	24.8	1.50
SML (MLton)	Mono	**151.0**	13.7	0.40
SML (SML/NJ)	Poli	**357.6**	14.4	0.95

Testy wydajnościowe
Wyniki

Język	Wersja	Czas (ms)	σ	x
Haskell (GHC)	Mono	39.1	8.2	0.10
Haskell (GHC)	Poli	696.8	63.2	1.86
Java	Mono	140.1	65.7	0.37
Java	Poli	564.9	24.8	1.50
SML (MLton)	Mono	151.0	13.7	0.40
SML (SML/NJ)	Poli	357.6	14.4	0.95
Mono ML	Mono	**327.0**	52.3	0.88
Mono ML	Poli	**375.4**	46.9	1.00

Częściowa aplikacja i generowanie funkcji

Opis problemu

```
let diverge cond x y = if cond then x else y

val apply : (int -> int -> int) -> int -> int -> int
let apply f x y =
  let z = diverge true x y
  f y z

...
apply (diverge true) 2 3
```

Częściowa aplikacja i generowanie funkcji

Opis problemu

```
val f : int -> int -> int
f y z
------------------
(diverge true) y z
```

```
%result = call i32 %f(i32 %y, i32 %z)
;-----------------------------------
%f = @diverge(i1 %cond, i32 %x, i32 %y)
```

Częściowa aplikacja i generowanie funkcji
Implementacja

```
struct function {
  void (**fn)();
  unsigned char *args;
  unsigned char left_args;
  unsigned char arity;
  int used_bytes;
};
```

Częściowa aplikacja i generowanie funkcji
Implementacja

Oryginalna funkcja

```
val diverge : bool -> int -> int -> int
```

Wygenerowane funkcje

```
i32 @diverge_main(i1 %cond, i32 %x, i32 %y)

i32 @diverge_entry0(i8 %args_cnt, i8* %env, i1 %cond
                                , i32 %x  , i32 %y)

i32 @diverge_entry1(i8 %args_cnt, i8* %env, i32 %x
                                          , i32 %y)

i32 @diverge_entry2(i8 %args_cnt, i8* %env, i32 %y)
```

Częściowa aplikacja i generowanie funkcji
Implementacja

Wygenerowane funkcje

```
@diverge_entry_points = global [3 x void ()*] [
    @diverge_entry0,
    @diverge_entry1,
    @diverge_entry2
]
```

Częściowa aplikacja i generowanie funkcji
Implementacja

Funkcja częściowo zaaplikowana

```
val diverge : bool -> int -> int -> int
(diverge true)
```

```
env = malloc (1); env[0] = true
function papp = {
  fn          = diverge_entry_points[1],
  args        = env,
  left_args   = 2,
  arity       = 3,
  used_bytes = 1,
}
```

Częściowa aplikacja i generowanie funkcji
Implementacja

Wywołanie funkcji częściowo zaaplikowanej

```
val f : int -> int -> int
f y z
```

```
f : function
f.fn(2, f.args, y, z);
```

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową
  - Zapisanie argumentów do środowiska

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową
  - Zapisanie argumentów do środowiska
- Funkcja wołana

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową
  - Zapisanie argumentów do środowiska
- Funkcja wołana
  - Odzyskanie argumentów ze środowiska

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową
  - Zapisanie argumentów do środowiska
- Funkcja wołana
  - Odzyskanie argumentów ze środowiska
  - Wywołanie funkcji głównej

Częściowa aplikacja i generowanie funkcji
Implementacja

- Funkcja wołająca (call site):
  - Sprawdzenie czy należy wywołać funkcję
  - Zrzutowanie wskaźnika na funkcję wejściową
  - Zapisanie argumentów do środowiska
- Funkcja wołana
  - Odzyskanie argumentów ze środowiska
  - Wywołanie funkcji głównej
  - Ewentualne przekazanie nadmiarowych argumentów lub zapisanie ich do środowiska

Klasy typów
Standardowa implementacja

Klasy typów
Standardowa implementacja

Haskell

```
class Show a where
  show :: a -> String
```

Klasy typów
Standardowa implementacja

Haskell

```
class Show a where
  show :: a -> String
```

OCaml

```
type 'a show = {
  show : 'a -> string
}
```

Klasy typów
Standardowa implementacja

Haskell

```
instance Show String where
  show s = "'" ++ s ++ "'"

instance Show Bool where
  show True = "true"
  show False = "false"
```

Klasy typów
Standardowa implementacja

OCaml

```
let show_string = {
  show = fun s -> "'" ^ s ^ "'"
}

let show_bool = {
  show = function
         | false -> "false"
         | true  -> "true"
}
```

Klasy typów
Standardowa implementacja

OCaml

```
let printArg show_instance arg =
  show_instance.show arg
  |> printf "arg: %s"

let main1 = printArg show_bool true
let main2 = printArg show_string "Hello"
```

Klasy typów
Standardowa implementacja

Haskell

```
printArg :: Show a => a -> IO ()
printArg arg = putStrLn ("arg: " ++ show arg)

main1 = printArg True
main2 = printArg "Hello World"
```

Klasy typów
Implementacja w MonoMLu

MonoML

```
class Num 'a where
  add : 'a -> 'a -> 'a
type pair = { a : int; b : int }

instance Num int where
  let add x y = x + y
instance Num pair where
  let add (x : pair) (y : pair) =
    { x with a = x.a + y.a; b = x.b + y.b }

let _ =
  let sum  : int  = add 2 3
  let sum2 : pair = add {a = 2; b = 1} {a = 3; b = 4}
```

Klasy typów
Implementacja w MonoMLu

MonoML

```
type pair = { a : int; b : int }

let Num_add_int x y = x + y
let Num_add_pair (x : pair) (y : pair) =
  { x with a = x.a + y.a; b = x.b + y.b }

let _ =
  let sum  = Num_add_int 2 3
  let sum2 = Num_add_pair {a = 2; b = 1} {a = 3; b = 4}
```

Podsumowanie

- Polimorfizm parametryczny
  - Efektywna implementacja dzięki monomorfizacji
  - Brak narzutu wydajnościowego i pamięciowego
  - Ułatwia dalsze optymalizacje
  - Dłuższy czas kompilacji
  - Większy rozmiar wynikowego programu
- Klasy typów
  - Ułatwiona implementacja dzięki monomorfizacji
  - Wprowadziły ad–hoc polimorfizm (przeładowanie funkcji)
  - Brak dodatkowego narzutu
- Częściowa aplikacja
  - Bazowana na push/enter z modyfikacjami
  - Wspiera przekazywanie argumentów przez wartość