

Copattern matching and first-class observations in OCaml, with a macro

Дмитрий Косарев

19 февраля, 2018

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

Мотивация

- Конечные
 - Например: список, дерево, ...
 - Индуктивные типы и pattern matching
- Бесконечные
 - Например: stream, бесконечное дерево, ...
 - Коиндуктивные типы и copattern matching

Мотивация

- Конечные
 - Например: список, дерево, ...
 - Индуктивные типы и pattern matching
- Бесконечные
 - Например: stream, бесконечное дерево, ...
 - Коиндуктивные типы и copattern matching

Copatterns: Programming Infinite Structures by Observations Abel, Pientka, Thibodeau и Setzer (POPL, 2013)
и другие статьи от тех же авторов

Объявление бесконечных значений

```
let rec fib () = 0 :: 1 :: map2 (+) (fib ()) (tl (fib ()))
```

В call-by-value языках присутствуют проблемы.

Объявление бесконечных значений

```
let rec fib () = 0 :: 1 :: map2 (+) (fib ()) (tl (fib ()))
```

В call-by-value языках присутствуют проблемы.

```
type 'a lazy_list = C of 'a * 'a lazy_list Lazy.t
```

```
let rec fib = C (0, lazy (C (1, lazy (map2' (+) fib (tl' fib)))))
```

Идея

- Copattern matching вместо pattern matching
- Строим бесконечное вычисление из результатов “наблюдений” (observations)
- Copattern matching предлагает потенциальные варианты того, что можно построить

Copattern-matching. Пример. Фибоначчи

Объявление корекурсивного типа

```
type 'a stream = {  
  Head: 'a           $\leftarrow$  'a stream  
  Tail: 'a stream  $\leftarrow$  'a stream  
}
```

Копаттерн-матчинг. Пример использования в расширенном синтаксисе OCaml

```
let corec fib : int stream with  
| .. #Head  $\rightarrow$  0  
| .. #Tail : int stream with  
| .. #Tail#Head  $\rightarrow$  1  
| .. #Tail#Tail  $\rightarrow$  map2stream (+) fib (fib#Tail)
```


Что хочется получить

- Копаттерны для OCaml
- Используя дуальность, реинтерпретировать это в терминах pattern matching
- Типобезопасно
- Эффективно

Что получилось


- OCaml-4.04+copatterns .
- Типобезопасно, но с GADT и 2nd-order polymorphic types
- Эффективно (**lazy cofunction**)

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

Синтаксис GADT

```
type ( $\alpha, \beta, \dots$ ) typ =  
  |  $C_1$  of  $\tau_{11}$  *  $\dots$  *  $\tau_{1m_1}$   
  |  $\dots$   
  |  $C_n$  of  $\tau_{n1}$  *  $\dots$  *  $\tau_{nm_n}$ 
```

Синтаксис GADT

```
type ( $\alpha, \beta, \dots$ ) typ =  
  |  $C_1$  of  $\tau_{11}$  *  $\dots$  *  $\tau_{1m_1}$   
  |  $\dots$   
  |  $C_n$  of  $\tau_{n1}$  *  $\dots$  *  $\tau_{nm_n}$ 
```

```
type ( $\alpha, \beta, \dots$ ) typ =  
  |  $C_1$  :  $\tau_{11}$  *  $\dots$  *  $\tau_{1m_1}$   $\rightarrow (t_{11}, t_{12}, \dots)$  typ  
  |  $\dots$   
  |  $C_n$  :  $\tau_{n1}$  *  $\dots$  *  $\tau_{nm_n}$   $\rightarrow (t_{n1}, t_{n2}, \dots)$  typ
```

GADT

Они же

- Generalized Algebraic Data Type
- First-class phantom type (James&Hinze, 2003)
- Guarded recursive datatype (Xi&Chen, 2003)
- Equality-qualified type (Sheard&Pasalic, tile here, 2004)

Каноничный пример

```
type 'a expr =  
| Int      : int           → int expr  
| Bool     : bool         → bool expr  
| EqualInt : int expr * int expr → bool expr
```

EqualInt (Int 5, Int 6)

- : bool expr = EqualInt (Int 5, Int 6)

Каноничный пример

```
type 'a expr =  
| Int      : int           → int expr  
| Bool     : bool         → bool expr  
| EqualInt : int expr * int expr → bool expr
```

```
EqualInt (Int 5, Int 6)
```

```
- : bool expr = EqualInt (Int 5, Int 6)
```

```
EqualInt (Bool true, Int 6)
```

```
Error: This expression has type bool expr  
      but an expression was expected of type int expr  
Type bool is not compatible with type int
```


Равенство типов 1/2 (слайд, которого нет)

```
module Num = struct  
  type t  
end
```

```
(* type num *)
```

```
type _ dataFormat =  
  | Date  : Num.t dataFormat  
  | Number: Num.t dataFormat  
  | String: string dataFormat  
  | Bool  : bool dataFormat
```

```
let test: Num.t dataFormat → unit = function  
  | Number → ()  
  | Date   → ()  
  | _      → _ (* required *)
```

Равенство типов 2/2 (слайд, которого нет)

```
type (_,_) eq = Eq: ('a,'a) eq
module Num: sig
  type t
  val eq: (t,string) eq
end = struct .. end
```

```
type _ dateFormat =
  | Date   : Num.t  dateFormat
  | Number: Num.t  dateFormat
  | String: string dateFormat
```

```
let test: Num.t dateFormat → unit = function
  | Number → ()
  | Date   → ()
  | _      → _ (* required *)
```

```
let () = match Num.eq with Eq → test String
```

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

GADT и наблюдение за ленивыми списками

```
type ('o, 'a) stream_query =  
| Head : ('a, 'a) stream_query  
| Tail : ('a stream, 'a) stream_query
```

Результат преобразования объявления типа коданных

```
and 'a stream = Stream of {  
  dispatch : 'o . ('o, 'a) stream query  $\rightarrow$  'o  
}
```

А в System F тип конструктора `Stream` записывается так:

$$\forall 'a . (\forall 'o . ('o, 'a) \text{ stream_query} \rightarrow 'o) \rightarrow 'a \text{ stream}$$

Jacques Garrigue and Didier Rémy. 1999.

Фибоначчи после трансформации

```
1 let rec fib : int stream =  
2   let dispatch : type o. (o, int) stream query → o = function  
3   | Head → 0  
4   | Tail →
```

Фибоначчи после трансформации

```
1 let rec fib : int stream =  
2   let dispatch : type o. (o, int) stream query → o = function  
3   | Head → 0  
4   | Tail →  
5     let dispatch : type o. (o, int) stream query → o = function  
6     | Head → 1  
7     | Tail → map2 (+) fib (tail fib)  
8   in Stream { dispatch }  
9   in Stream { dispatch }
```

```
let tail (Stream { dispatch }) = dispatch Tail
```

Исходные фибоначчи с копаттернами

```
let corec fib : int stream with  
| ..#Head → 0  
| ..#Tail : int stream with  
| ..#Tail#Head → 1  
| ..#Tail#Tail → map2 (+) fib (fib#Tail)
```

При преобразовании произошел unnesting

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

Преобразование

$$\lambda^C \rightarrow \lambda^G$$

Термы

t, u	$::=$	
x		Переменная
D		Observation request
$K t$		Конструктор с 1 аргументом
$t t$		Применение
$t \cdot t$		Observation
$\mu^+ f : \sigma := \lambda \bar{x} \{ \bar{b} \}$		Функция
$\mu^- f : \sigma := \lambda \bar{x} \{ \bar{b} \}$		Коданные

Ветки мэтчинга и значения

branch ::=

$\bullet \Rightarrow t$	Suspension
$K x$	Деконструирование
$\cdot D \Rightarrow t$	Observation

value ::=

$\lambda^- \bar{x} \{ \bar{b} \} \Rightarrow t$	Коданные
$\lambda^+ \bar{x} \{ \bar{b} \} \Rightarrow t$	Функция
$K v$	Данные
D	Request

Семантика малого шага для λ^C

$$\begin{array}{lcl}
 E[t] & \xrightarrow{\text{SCxt}} & E[t'] \quad t \rightarrow t' \\
 \mu^\diamond f : \sigma := \lambda \bar{x} \{\bar{b}\} & \xrightarrow{\text{SUnr}} & \lambda^\diamond \bar{x} \{(\bar{b}[f \mapsto \mu^\diamond f : \sigma := \lambda \bar{x} \{\bar{b}\}])\} \\
 (\lambda^\diamond x \bar{x} \{\bar{b}\})v & \xrightarrow{\text{SPush}} & \lambda^\diamond \bar{x} \{\bar{b}[x \mapsto v]\} \text{ если SEval не применим} \\
 (\lambda^\diamond x \{\bullet \Rightarrow t \mid \bar{b}\})v & \xrightarrow{\text{SEval}} & t[x \mapsto v] \\
 (\lambda^+ \bullet \{K x \Rightarrow t \mid \bar{b}\})(K v) & \xrightarrow{\text{SDes}} & t[x \mapsto v] \\
 (\lambda^+ \bullet \{K x \Rightarrow t \mid \bar{b}\})(K' v) & \xrightarrow{\text{SDesF}} & (\lambda^+ \bullet \{\bar{b}\})(K' v), \text{ если } K \neq K' \\
 (\lambda^- \bullet \{\cdot D \Rightarrow t \mid \bar{b}\}) D & \xrightarrow{\text{SObs}} & t \\
 (\lambda^- \bullet \{\cdot D \Rightarrow t \mid \bar{b}\}) D' & \xrightarrow{\text{SObsF}} & (\lambda^- \bullet \{\bar{b}\})D', \text{ если } D \neq D'
 \end{array}$$

где $\diamond \in \{+, -\}$

Типы

τ	$::=$	
α		Типовая переменная
$\varepsilon^+(\bar{\tau})$		Данные
$\varepsilon^-(\bar{\tau})$		Коданные
$\tau \rightarrow \tau$		Стрелка
$\tau \leftarrow \varepsilon^-(\bar{\tau})$		Observation request

$$\varepsilon^+(\bar{\alpha}) := \Sigma_i K_i : \forall \bar{\alpha}. \tau_i \rightarrow \varepsilon^+(\bar{\tau}_i)$$

$$\varepsilon^-(\bar{\alpha}) := \times_i D_i : \forall \bar{\alpha}. \tau_i \leftarrow \varepsilon^-(\bar{\tau}_i)$$

Одного аргумента у K – достаточно

$$\begin{aligned}\varepsilon^+(\bar{\alpha}) &:= \Sigma_i K_i : \forall \bar{\alpha}. \tau_i \rightarrow \varepsilon^+(\bar{\tau}_i) \\ \varepsilon^+(\bar{\alpha}) &:= \times_i D_i : \forall \bar{\alpha}. \tau_i \leftarrow \varepsilon^-(\bar{\tau}_i)\end{aligned}$$

$$unit = DUnit : unit \leftarrow unit$$

$$pair(\alpha, \beta) = \begin{cases} DFst : \forall \alpha \beta. \alpha \leftarrow pair(\alpha, \beta) \\ DSnd : \forall \alpha \beta. \beta \leftarrow pair(\alpha, \beta) \end{cases}$$

Окружения с именами типов и констрейнты

$\Gamma ::=$

\bullet	пусто
$\Gamma\alpha$	Связанная переменная типа
$\Gamma(x : \alpha)$	Связанная переменная

$C ::=$

true	Тривиальный
false	Пустой
$\tau = \tau$	Атомарное равенство
$C \wedge C$	Конъюнкция

Правила для типов

Тут показывать скриншот :)

Теоремы λ^C

Теорема (про редукцию):

C выполнимо. Если $\Gamma, C \vdash t : \tau$ и $t \rightarrow t'$, тогда $\Gamma, C \vdash t' : \tau$.

Теорема (Про прогресс).

Констрейнты C выполнимы. Если $\Gamma, C \vdash t : \tau$, то либо t это конечное значение или существует t' такое что $t \rightarrow t'$.

Синтаксис и семантика λ^G

- Нет observations ни в синтаксисе, ни в правилах для семантики (просто выкинули)
- Конструкторы могут иметь 0 или много аргументов

Синтаксис и семантика λ^G

$$\lambda^C: \quad \varepsilon^+(\bar{\alpha}) := \Sigma_i K_i : \forall \bar{\alpha}. \tau_i \rightarrow \varepsilon^+(\bar{\tau}_i)$$

$$\lambda^G: \quad \varepsilon^+(\bar{\alpha}) := \Sigma_i K_i : \forall \bar{\alpha}. \bar{\sigma} \rightarrow \varepsilon^+(\bar{\tau}_i)$$

В λ^G аргументы конструктора могут иметь полиморфный тип, а к самому конструктору приписывается схемат типов второго порядка.

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

Схема трансляции

Типы

$$\begin{aligned}
 \llbracket \alpha \rrbracket &= \alpha \\
 \llbracket \tau \rightarrow \rho \rrbracket &= \llbracket \tau \rrbracket \rightarrow \llbracket \rho \rrbracket \\
 \llbracket \varepsilon^+(\bar{\tau}) \rrbracket &= \varepsilon^+(\llbracket \bar{\tau} \rrbracket) \\
 \llbracket \tau \leftarrow \varepsilon^-(\bar{\tau}) \rrbracket &= \varepsilon_r^-(\llbracket \tau \rrbracket \llbracket \bar{\tau} \rrbracket) \\
 \llbracket \varepsilon^-(\bar{\tau}) \rrbracket &= \varepsilon_d^-(\llbracket \bar{\tau} \rrbracket)
 \end{aligned}$$

Термы

$$\begin{aligned}
 \llbracket x \rrbracket &= x \\
 \llbracket tu \rrbracket &= \llbracket t \rrbracket \llbracket u \rrbracket \\
 \llbracket t \cdot u \rrbracket &= \llbracket u \rrbracket \llbracket t \rrbracket \\
 \llbracket Kt \rrbracket &= K \llbracket t \rrbracket \\
 \llbracket D \rrbracket &= d \\
 \llbracket \mu^- f : \forall \bar{\alpha}. \tau := \lambda \bar{x}. \{\bar{b}\} \rrbracket &= \mu^+ f : \forall \bar{\alpha}. \llbracket \tau \rrbracket := \lambda \bar{x}. K^{\varepsilon^-(\bar{\tau})} \llbracket \bar{b} \rrbracket_{\bar{\alpha}, O(\tau)}^\perp \\
 \llbracket \mu^+ f : \forall \bar{\alpha}. \tau := \lambda \bar{x}. \{\bar{b}\} \rrbracket &= \mu^+ f : \forall \bar{\alpha}. \llbracket \tau \rrbracket := \lambda \bar{x}. \llbracket \bar{b} \rrbracket \\
 \llbracket Kx \Rightarrow t \rrbracket &= Kx \Rightarrow \llbracket t \rrbracket \\
 \llbracket \vee \cdot D_i \Rightarrow t \rrbracket_{\bar{\alpha}, \varepsilon^-(\bar{\tau})}^\perp &= \mu^+ _ : \forall \beta. \varepsilon_r^-(\beta, \bar{\tau}) \rightarrow \beta := \lambda \bullet. \vee_i D_i \Rightarrow \llbracket t \rrbracket
 \end{aligned}$$

Преобразование объявлений типов

Индуктивные

$$\llbracket \varepsilon^+(\bar{\alpha}) := \Sigma_i K_i : \forall \bar{\alpha}. \tau_i \rightarrow \varepsilon^+(\bar{\tau}_i) \rrbracket = \varepsilon^+(\bar{\alpha}) := \Sigma_i K_i : \forall \bar{\alpha}. \llbracket \tau_i \rrbracket \rightarrow \varepsilon^+(\llbracket \bar{\tau} \rrbracket_i)$$

Коиндуктивные

$$\begin{aligned} \llbracket \varepsilon^+(\bar{\alpha}) := \times_i D_i : \forall \bar{\alpha}. \tau_i \leftarrow \varepsilon^-(\bar{\tau}_i) \rrbracket = \\ \left\{ \begin{array}{l} \varepsilon_r^-(\bar{\alpha}) := \Sigma_i D_i : \forall \bar{\alpha}. \varepsilon_r^-(\llbracket \tau_i \rrbracket, \llbracket \bar{\tau}_i \rrbracket) \\ \varepsilon_d^-(\bar{\alpha}) := K^{\varepsilon^-(\bar{\alpha})} : \forall \bar{\alpha}. (\forall \beta. \varepsilon_r^-(\beta, \bar{\alpha}) \rightarrow \beta) \rightarrow \varepsilon_d^-(\bar{\alpha}) \\ \forall i, \quad d_i := \mu^+ _ : \forall \bar{\alpha}. \varepsilon_d^-(\llbracket \bar{\tau}_i \rrbracket) \rightarrow \llbracket \tau_i \rrbracket := \lambda \bullet . K^{\varepsilon^-(\bar{\alpha})} c \Rightarrow c D_i \end{array} \right. \end{aligned}$$

Table of Contents

Мотивация

GADT

GADT и ленивые списки

Формализация

Трансляция

Unnesting

Unnesting паттернов и копаттернов. Пример.

```
type _ repr =  
| Int  : int → int repr  
| Bool : bool → bool repr  
  
type _ grepr = {  
  QInt  : int  ← int grepr;  
  QBool : bool ← bool grepr }
```

```
let corec f : type a . a repr → a grepr with  
| (..(Int n))#QInt  → n  
| (..(Bool b))#QBool → b
```

⇓

```
let f : type a . a repr → a grepr = fun x →  
  let dispatch : type o . (o, a) grepr_query → o = function  
  | QInt  → (match x with Int n → n)  
  | QBool → (match x with Bool b → b)  
in  
  Qrepr { dispatch }
```

“Лишние” паттерны. Два примера

```
let corec zeros : int stream with  
| ..#Head → 0  
| ..#Head → 1      (* <= to remove *)  
| ..#Tail → zeros
```

“Лишние” паттерны. Два примера

```
let corec zeros : int stream with  
| ..#Head → 0  
| ..#Head → 1      (* <= to remove *)  
| ..#Tail → zeros
```

```
let corec f : int → int stream with  
| (.. n)#Head → 0  
| (.. n)#Tail → f (n - 1)  (* <= to remove *)  
| (.. n)#Tail : int stream with  
| (.. n)#Tail#Head → n  
| (.. n)#Tail#Tail → f (n + 1)
```

Более глубокие считаются более сильными

Ленивые вычисления

```
let rec fib : int stream =  
  let dispatch : type o. (o, int) stream query → o = function  
    | Head → 0  
    | Tail →  
      let dispatch : type o. (o, int) stream query → o = function  
        | Head → 1  
        | Tail → map2 (+) fib (tail fib)  
      in Stream { dispatch }  
  in Stream { dispatch }
```

Можно заменить каждый `dispatch` на `dispatch` с мемоизацией (на слайдах нет)

First-order и high-order observations

- По описанию копаттерна сгенерировались функции...
- Их можно передавать в другие функции и получать high-order

Поиск в глубину (dfs)

1