Уменьшение цены абстракции при типобезопасном встраивании реляционнного языка программирования в OCaml

Дмитрий Косарев

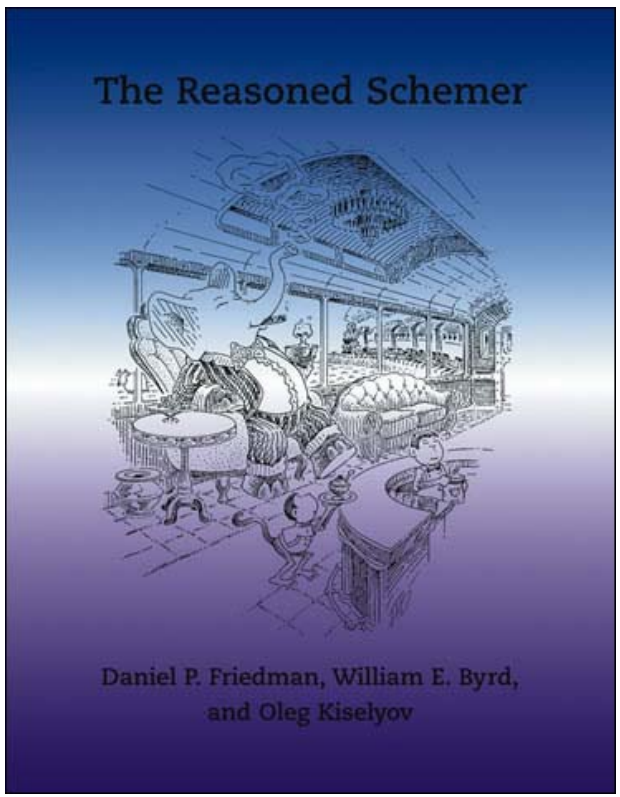
Санкт-Петербургский Государтсвенный Университет
JetBrains Research

Языки программирования и компиляторы
4 апреля, 2016
Ростов-на-Дону

Реляционное программирование на miniKanren

От программ-функций к программам-отношениям:

$$f : X \to Y \rightsquigarrow f^o \subseteq X \times Y$$



- Изначально DSL для Scheme/Racket с довольно минималистичной реализацией
- Семейство языков (μKanren, α-Kanren, cKanren, и т.д.)
- Встраивается как DSL в широкий набор языков (включая OCaml, Haskell, Scala, и т.д.)
- Daniel P. Friedman, William Byrd and Oleg Kiselyov. The Reasoned Schemer, The MIT Press, Cambridge, MA, 2005

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

$\text{append} : \alpha\ \text{list} \to \alpha\ \text{list} \to \alpha\ \text{list}$

$\text{append}^o \subseteq \alpha\ \text{list} \times \alpha\ \text{list} \times \alpha\ \text{list}$

```
let rec append xs ys =
  match xs with
  | []      -> ys
  | h :: tl ->
      h :: (append tl ys)
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

$\text{append} : \alpha\ \text{list} \to \alpha\ \text{list} \to \alpha\ \text{list}$

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

$\text{append}^o \subseteq \alpha\ \text{list} \times \alpha\ \text{list} \times \alpha\ \text{list}$

```
let rec append^o xs ys xys =
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

$\text{append} : \alpha\ \text{list} \to \alpha\ \text{list} \to \alpha\ \text{list}$

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

$\text{append}^o \subseteq \alpha\ \text{list} \times \alpha\ \text{list} \times \alpha\ \text{list}$

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

$\text{append} : \alpha\ \text{list} \to \alpha\ \text{list} \to \alpha\ \text{list}$

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

$\text{append}^o \subseteq \alpha\ \text{list} \times \alpha\ \text{list} \times \alpha\ \text{list}$

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

append : α list $\rightarrow \alpha$ list $\rightarrow \alpha$ list

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

append$^o \subseteq \alpha$ list $\times \alpha$ list $\times \alpha$ list

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

$\text{append} : \alpha\ \text{list} \to \alpha\ \text{list} \to \alpha\ \text{list}$

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

$\text{append}^o \subseteq \alpha\ \text{list} \times \alpha\ \text{list} \times \alpha\ \text{list}$

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

append : α list $\to \alpha$ list $\to \alpha$ list

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

append$^o \subseteq \alpha$ list $\times \alpha$ list $\times \alpha$ list

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
     (append^o t ys tys) )
```

Пример: реляционное слияние списков (OCaml/OCanren/miniKanren)

append: α list $\to \alpha$ list $\to \alpha$ list

```
let rec append xs ys =
  match xs with
  | []      → ys
  | h :: tl →
      h :: (append tl ys)
```

append$^o \subseteq \alpha$ list $\times \alpha$ list $\times \alpha$ list

```
let rec append^o xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
     (append^o t ys tys) )
```

В оригинальной реализации:

```
(define (append^o xs ys xys)
   (conde
      [(≡ '() xs) (≡ ys xys)]
      [(fresh (h t tys)
         (≡ `(,h . ,t) xs)
         (≡ `(,h . ,tys) xys)
         (append^o t ys tys))]))
```

Набросок минимальной реализации

Jason Hemann, Daniel P. Friedman. μKanren: A Minimal Functional Core for Relational Programming // Scheme'13:

Набросок минимальной реализации

Jason Hemann, Daniel P. Friedman. μKanren: A Minimal Functional Core for Relational Programming // Scheme'13:

Логические переменные	$X = \{x_1, x_2, \dots\}$
Символы (конструкторы)	$S = \{s_1, s_2, \dots\}$
Термы	$T = X \cup \{s\,(t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Подстановки	$\Sigma = T^X$

Набросок минимальной реализации

Jason Hemann, Daniel P. Friedman. μKanren: A Minimal Functional Core for Relational Programming // Scheme'13:

Логические переменные	$X = \{x_1, x_2, \dots\}$
Символы (конструкторы)	$S = \{s_1, s_2, \dots\}$
Термы	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Подстановки	$\Sigma = T^X$
Унификация	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$

Набросок минимальной реализации

Jason Hemann, Daniel P. Friedman. μKanren: A Minimal Functional Core for Relational Programming // Scheme'13:

Логические переменные	$X = \{x_1, x_2, \dots\}$
Символы (конструкторы)	$S = \{s_1, s_2, \dots\}$
Термы	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Подстановки	$\Sigma = T^X$
Унификация	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (подстановка + как создавать новые логические переменные)	σ
Goal (функция из состояния в ленивый список состояний)	$g : \sigma \to \sigma\ \text{stream}$
Конъюнкция $g \wedge g$	"bind"
Дизъюнкция $g \vee g$	"mplus"
Refinement: извлечение посчитанных ответов	$\text{refine} : \sigma \to X \to T$

Полиморфная унификация 1

Работает для всех логических типов $\alpha\, logic$ (он же α^o):

$$\equiv : \Sigma \to \alpha^o \to \alpha^o \to \Sigma_\perp$$

Полиморфная унификация 1

Работает для всех логических типов $\alpha\, logic$ (он же α^o):

$$\equiv : \Sigma \to \alpha^o \to \alpha^o \to \Sigma_\perp$$

Реализована как сравнение представлений значений в памяти.

Подход для описания типов (на примере логического списка)

```
type 'a logic = Var of int | Value of 'a
...
```

Подход для описания типов (на примере логического списка)

```
type 'a logic = Var of int | Value of 'a
...
type ('a, 'b) glist = Nil | Cons of 'a * 'b
```

Подход для описания типов (на примере логического списка)

```
type 'a logic = Var of int | Value of 'a
...
type ('a, 'b) glist = Nil | Cons of 'a * 'b

type 'a list = ('a, 'a list) glist (* Nil | Cons of 'a *)
```

Подход для описания типов (на примере логического списка)

```
type 'a logic = Var of int | Value of 'a
...
type ('a, 'b) glist = Nil | Cons of 'a * 'b

type 'a list = ('a, 'a list) glist (* Nil | Cons of 'a *)

type 'a llist = ('a , 'a llist) glist logic
```

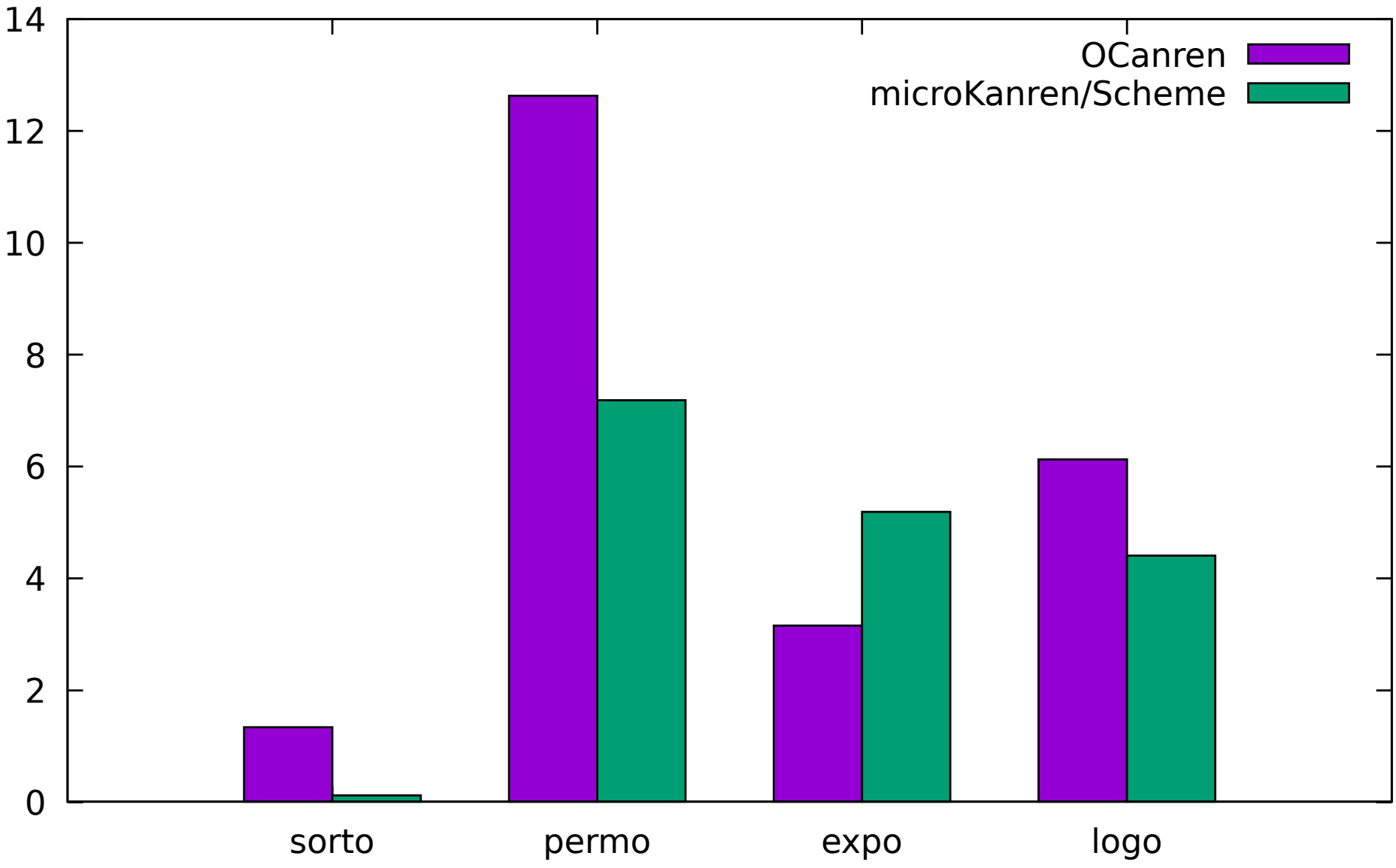
Подход для описания типов (на примере логического списка)

```
type 'a logic = Var of int | Value of 'a
...
type ('a, 'b) glist = Nil | Cons of 'a * 'b

type 'a list = ('a, 'a list) glist (* Nil | Cons of 'a *)

type 'a llist = ('a , 'a llist) glist logic

...
# Value Nil
-: 'a llist
# Value (Cons (Value 1), Value Nil)
-: int logic llist
# Value (Cons (Var 101), Value Nil)
-: int logic llist
```

Промежуточные результаты

Были представлены на ML Workshop 2016 (совмещённым с ICFP 2016)

- Типобезопасное встраивание miniKanren в OCaml
- Полиморфная унификация
- Регулярный подход для описания типов

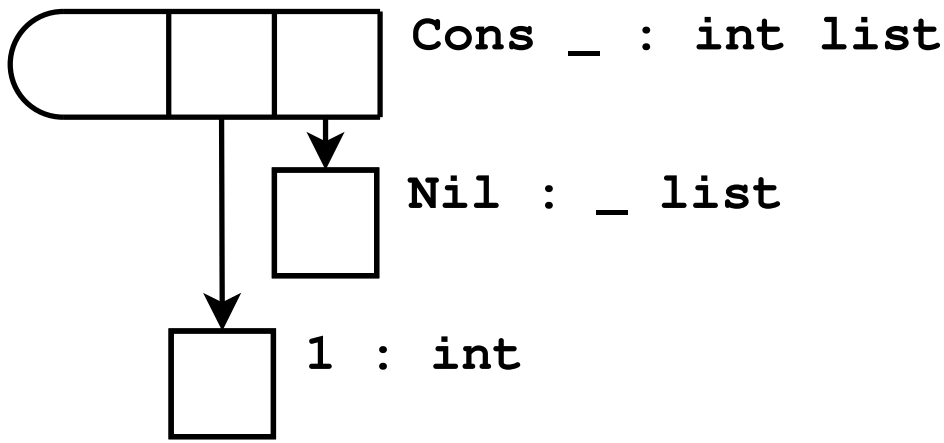

14
12
10
8
6
4
2
0
OCanren
microKanren/Scheme
sorto
permo
expo
logo

Дальнейшие задачи

- Найти причину замедления
- Ускорить
- Подход должен остаться типобезопасным

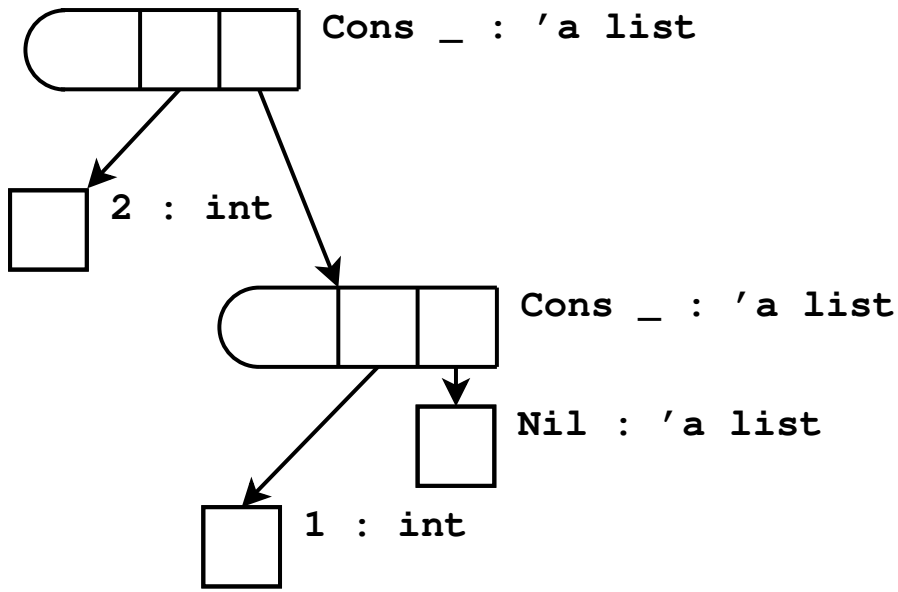
Полиморфная унификация 2

`Cons (1, Nil) : int list`



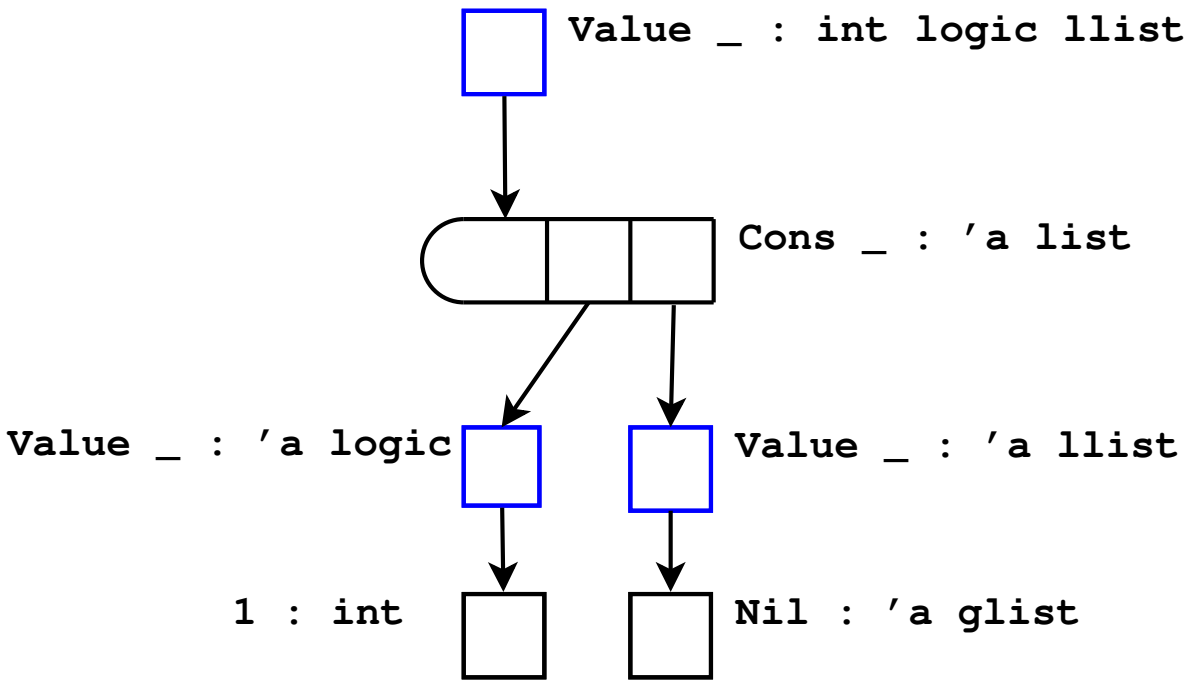
Полиморфная унификация 3

```
Cons (2, Cons (1, Nil)) : int list
```



Полиморфная унификация 4

```
Value (Cons (Value 2, Value Nil))
 : int llist
```



last

```
type 'a list = Nil | Cons of 'a * 'a list

let (_: int list) = Cons (1, Nil)
```

Cons (2, Cons (1, Nil)) : int list