Уменьшение цены абстракции при типобезопасном встраивании реляционнного языка программирования в OCaml

Дмитрий Косарев

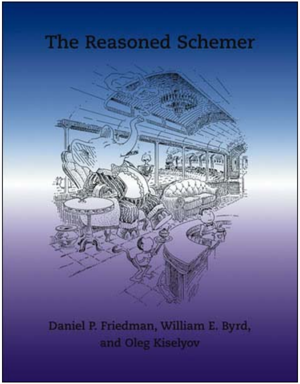
Санкт-Петербургский Государтсвенный Университет
JetBrains Research

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 appendᵒ xs ys xys =
```

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

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

```
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 appendᵒ 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 appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
```

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 appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
```

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 appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
```

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 appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
     (appendᵒ t ys tys) )
```

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 appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
  |||
  (fresh (h t tys)
     (xs  ≡ h % t)
     (xys ≡ h % tys)
     (appendᵒ t ys tys) )
```

В оригинальной реализации:

```
(define (appendᵒ xs ys xys)
   (conde
      [(≡ '() xs) (≡ ys xys)]
      [(fresh (h t tys)
         (≡ `(,h . ,t) xs)
         (≡ `(,h . ,tys) xys)
         (appendᵒ 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)	$g : \sigma \to \sigma$ stream
Конъюнкция $g \wedge g$	"bind"
Дизъюнкция $g \vee g$	"mplus"
Refinement: извлечение посчитанных ответов	refine $: \sigma \to X \to T$

Полиморфная унификация 1 из 7

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 из 7

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.

Систематическое конструирование логических типов

```
type α logic = Var of int | Value of α
...
```

Систематическое конструирование логических типов

```
type α logic = Var of int | Value of α
...
type (α, β) glist = Nil | Cons of α * β
```

Систематическое конструирование логических типов

```
type α logic = Var of int | Value of α
...
type (α, β) glist = Nil | Cons of α * β

type α list = (α, α list) glist
```

Систематическое конструирование логических типов

```
type α logic = Var of int | Value of α
...
type (α, β) glist = Nil | Cons of α * β

type α list = (α, α list) glist

type α llist = (α, α llist) glist logic
```

Систематическое конструирование логических типов

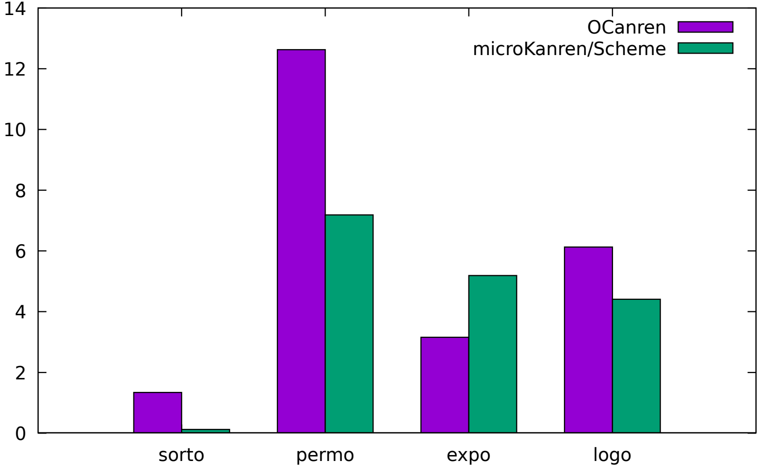
```
type α logic = Var of int | Value of α
...
type (α, β) glist = Nil | Cons of α * β

type α list = (α, α list) glist

type α llist = (α, α llist) glist logic

...
# Value Nil
-: α 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

Результаты сравнения производительности

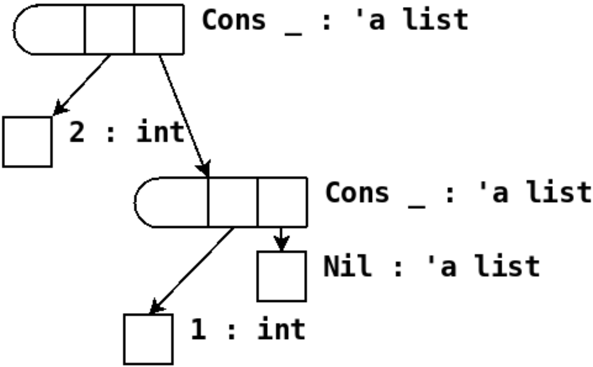


Дальнейшие задачи

- Найти причину замедления
- Ускорить
- Подход должен остаться типобезопасным

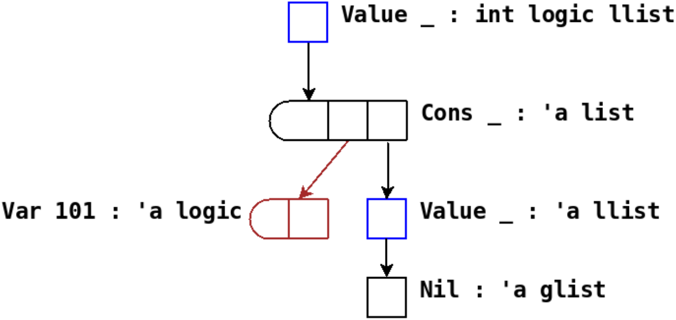
Представление термов

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



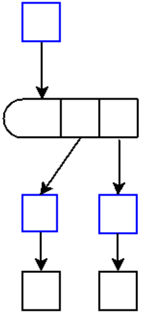
Тегированное представление логических значений

```
Value (Cons (Var 101, Value Nil))
 : int llist
```

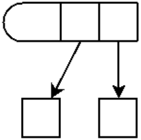


Рост размера термов из-за тегирования

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



`Cons (2, Nil) : int list`



План улучшения реализации

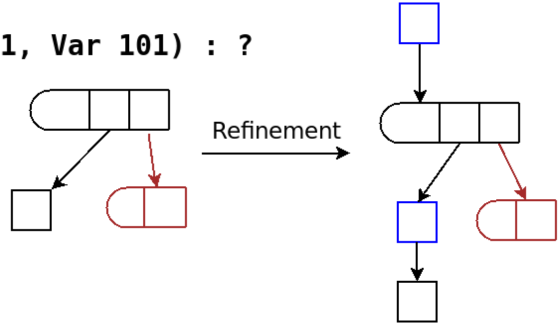
- Новое представление деревьев
  - Значению нельзя присвоить конкретный тип, нужен абстрактный тип значений.
  - Предоставить интерфейс для конструирования логических значений
  - Дополнительные действия по преобразованию абстрактного логического значения в типизируемое
- Модернизировать подход по описанию типов логических значений
- Не потерять типовую безопасность

Основная идея

- Унифицировать нетипизированнные термы
- Преобразовывать к типизируемому представлению при refine
- Запоминать формальные типы значений при каждом преобразовании к логическому значению

```
Value (Cons (Value 1, Var 101)) : int llist
```

```
Cons (1, Var 101) : ?
```



Тип injected

```
type ('a, 'b) injected
```

Тип 'a – это исходный тип, а тип 'b – его логическое представление
Представление ground-типов совпадает с представлением 'a.

Конструирование логических значений для простых типов

```
type ('a, 'b) injected
```

Конструирование логических значений для простых типов

```
type ('a, 'b) injected

val lift: 'a → ('a,'a) injected
val inj : ('a,'b) injected → ('a,'b logic) injected
```

Конструирование логических значений для простых типов

```
type ('a, 'b) injected

val lift: 'a → ('a,'a) injected
val inj : ('a,'b) injected → ('a,'b logic) injected
```

Например для чисел

```
# inj (lift 5)
-: (int, int logic) injected
```

Конструирование логических значений для простых типов

```
type ('a, 'b) injected

val lift: 'a → ('a,'a) injected
val inj : ('a,'b) injected → ('a,'b logic) injected
```

Например для чисел

```
# inj (lift 5)
-: (int, int logic) injected
```

Оба введенных примитива оставляют переданное значение как есть (identity)

Конструирование логических значений для сложных типов

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end
```

Конструирование логических значений для сложных типов

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end

# Make1(Option).distrib
...
```

Конструирование логических значений для сложных типов

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end

# Make1(Option).distrib
...
# let some x  = inj @@ distrib (Some x)
-: (α, β) injected →    (α option, β option logic) injected
```

Конструирование логических значений для сложных типов

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end

# Make1(Option).distrib
...
# let some x  = inj @@ distrib (Some x)
-: (α, β) injected →    (α option, β option logic) injected
```

Здесь fmap нужен для доказательства того, что тип является функтором, т.е. чтобы можно было описать примимитив distrib, который позволяет "снять" тип со значения, ничего не делая со значением (он тоже identity).

Восстановление посчитанных значений

Необходимо, так как значения в типе (_,_) injected хранятся в нетипизированном виде.

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end
```

Восстановление посчитанных значений

Необходимо, так как значения в типе (_,_) injected хранятся в нетипизированном виде.

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end

  # Make1(Option).reify
  -: ( (α, β) injected →    β) →
```

Восстановление посчитанных значений

Необходимо, так как значения в типе (_,_) injected хранятся в нетипизированном виде.

```
module Option = struct
  type α option = None | Some of α
  let fmap = ....
end

  # Make1(Option).reify
   -: ( (α, β) injected →    β) →
       (α option, β option logic) injected →    β option logic
```

При построении reify функция fmap используется по существу.

Текущая реализация

- Репозиторий: https://github.com/dboulytchev/OCanren
- Реализация μKanren с неравенствами (disequality constraints)
- Работает на большинстве оригинальных примеров
- Быстрее μKanren (https://github.com/Kakadu/ocanren-perf)