

# Уменьшение цены абстракции при встраивании реляционного DSL в OCaml

Дмитрий Косарев

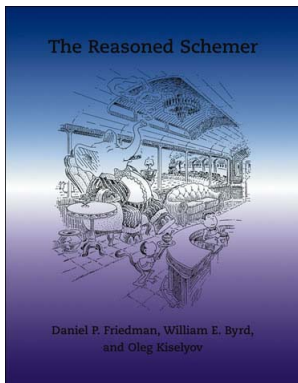
Лаборатория языковых инструментов

16 декабря, 2017

# Реляционное программирование на miniKanren

От программ-функций к программам-отношениям:

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



- Изначально DSL для Scheme/Racket с довольно минималистичной реализацией
- Семейство языков ( $\mu$ Kanren,  $\alpha$  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)

`append:  $\alpha$ list  $\rightarrow$   $\alpha$ list  $\rightarrow$   $\alpha$ list`

`appendo  $\subseteq$   $\alpha$ list  $\times$   $\alpha$ list  $\times$   $\alpha$ list`

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

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

`append:  $\alpha$ list  $\rightarrow$   $\alpha$ list  $\rightarrow$   $\alpha$ list`

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

`appendo  $\subseteq$   $\alpha$ list  $\times$   $\alpha$ list  $\times$   $\alpha$ list`

```
let rec appendo xs ys xys =
```

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

`append:  $\alpha$ list  $\rightarrow$   $\alpha$ list  $\rightarrow$   $\alpha$ list`

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

`appendo  $\subseteq$   $\alpha$ list  $\times$   $\alpha$ list  $\times$   $\alpha$ list`

```
let rec appendo xs ys xys =  
  ((xs  $\equiv$  nil)  $\&\&$  (xys  $\equiv$  ys))
```

## 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  $\equiv$  nil) &&& (xys  $\equiv$  ys))  
  |||  
  (fresh (h t tys)
```

## 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  $\equiv$  nil) &&& (xys  $\equiv$  ys))  
  |||  
  (fresh (h t tys)  
    (xs  $\equiv$  h % t))
```

## 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  $\equiv$  nil) &&& (xys  $\equiv$  ys))  
  |||  
  (fresh (h t tys)  
    (xs  $\equiv$  h % t)  
    (xys  $\equiv$  h % tys))
```



## 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  $\equiv$  nil) &&& (xys  $\equiv$  ys))  
  |||  
  (fresh (h t tys)  
    (xs  $\equiv$  h % t)  
    (xys  $\equiv$  h % tys)  
    (appendo t ys tys) )
```

## 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  $\equiv$  nil) &&& (xys  $\equiv$  ys))  
  |||  
  (fresh (h t tys)  
    (xs  $\equiv$  h % t)  
    (xys  $\equiv$  h % tys)  
    (appendo t ys tys) )
```

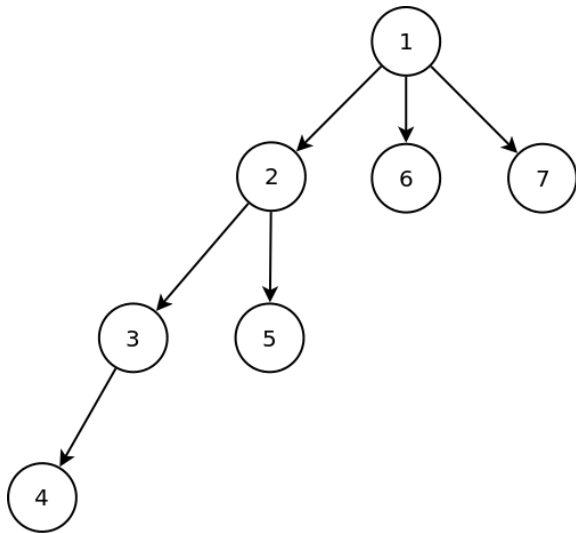
В оригинальной реализации:

```
(define (appendo xs ys xys)  
  (conde  
    [( $\equiv$  '() xs) ( $\equiv$  ys xys)]  
    [(fresh (h t tys)  
      ( $\equiv$  '(,h . ,t) xs)  
      ( $\equiv$  '(,h . ,tys) xys)  
      (appendo t ys tys))]))
```

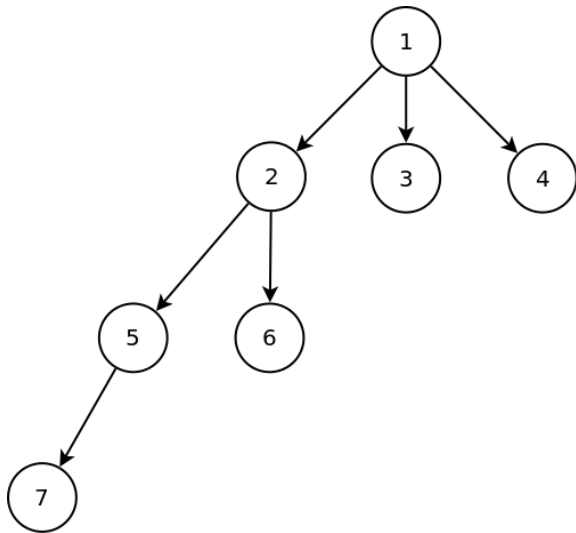
# miniKanren vs. Prologs (1)

- Стратегия поиска: interleaving.

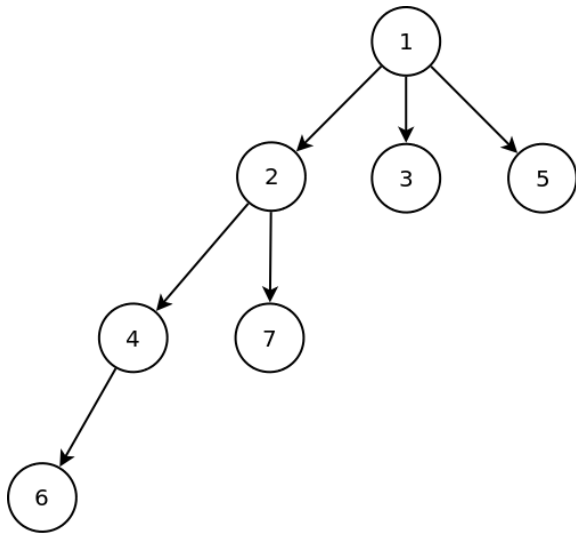
## Поиск в глубину (dfs)



## Поиск в ширину (bfs)



## Interleaving поиск



## miniKanren vs. Prologs (2)

- Стратегия поиска: interleaving search.
  - Поиск в глубину (DFS) быстрее при конечном переборе.
  - Чтобы DFS завершился в Prolog введены дополнительные синтаксические конструкции (cuts).
- miniKanren реализован как DSL – встраивается в другие языки.

# Цель работы

OCanren – DSL для типобезопасного встраивания miniKanren в OCaml.



# Изначальные ожидания

- Меньше ошибок при разработке.
- Сходная производительность, или даже лучше.

## Предыдущие типобезопасные встраивания

- HaskellKanren 
- ukanren 
- miniKanren-ocaml 
- Molog 
- MiniKanrenT 

# Предыдущие типобезопасные встраивания

- HaskellKanren 
- ukanren 
- miniKanren-ocaml 
- Molog 
- MiniKanrenT 

Различия могут быть в следующем:

- Как именно используются типы?
- Как производится унификация?

# Как используются типы?

- Один заранее созданный тип, на котором унификация разрешена.
  - ✗ Нет поддержки пользовательских типов.
  - ✓ “Универсальная” функция унификации.

ukanren, miniKanren-ocaml, HaskellKanren

# Как используются типы?

- Один заранее созданный тип, на котором унификация разрешена.
  - ✗ Нет поддержки пользовательских типов.
  - ✓ “Универсальная” функция унификации.

ukanren, miniKanren-ocaml, HaskellKanren

- О размещении логических переменных беспокоится пользователь.
  - ✗ Неудобно.
  - ✗ Функция унификации для каждого типа своя.
  - ✓ Поддержка произвольных типов данных.

miniKanrenT, Molog

## Как сделано в OCanren?

- Один тип для разделения логических переменных от конкретных термов.

**type**  $\alpha$  logic = Var **of** int | Value **of**  $\alpha$

- Который используется в пользовательских типах.

# Как сделано в OCanren?

- Один тип для разделения логических переменных от конкретных термов.

**type**  $\alpha$  logic = Var **of** int | Value **of**  $\alpha$

- Который используется в пользовательских типах.
- Полиморфная унификация – одна для всех типов.
  - ✗ Подход специфичен для OCaml.
  - ✗ Написано в типонебезопасном стиле...
  - ✓ ... но это скрыто от пользователя.
  - ✓ Идеологически как в первоисточнике.

# Как сделано в OCanren?

- Один тип для разделения логических переменных от конкретных термов.

**type**  $\alpha$  logic = Var **of** int | Value **of**  $\alpha$

- Который используется в пользовательских типах.
- Полиморфная унификация – одна для всех типов.
  - ✗ Подход специфичен для OCaml.
  - ✗ Написано в типонебезопасном стиле...
  - ✓ ... но это скрыто от пользователя.
  - ✓ Идеалогически как в первоисточнике.
- Пользователь должен следовать рекомендациям по описанию типов.



# Систематическое конструирование логических типов (на примере списка)

```
type  $\alpha$  logic = Var of int | Value of  $\alpha$   
...
```

## Систематическое конструирование логических типов (на примере списка)

```
type  $\alpha$  logic = Var of int | Value of  $\alpha$   
...  
type ( $\alpha$ ,  $\beta$ ) glist = Nil | Cons of  $\alpha$  *  $\beta$ 
```

## Систематическое конструирование логических типов (на примере списка)

```
type  $\alpha$  logic = Var of int | Value of  $\alpha$   
...  
type ( $\alpha$ ,  $\beta$ ) glist = Nil | Cons of  $\alpha$  *  $\beta$   
  
type  $\alpha$  list = ( $\alpha$ ,  $\alpha$  list) glist
```

## Систематическое конструирование логических типов (на примере списка)

```
type  $\alpha$  logic = Var of int | Value of  $\alpha$   
...  
type ( $\alpha$ ,  $\beta$ ) glist = Nil | Cons of  $\alpha$  *  $\beta$   
  
type  $\alpha$  list = ( $\alpha$ ,  $\alpha$  list) glist  
  
type  $\alpha$  llist = ( $\alpha$ ,  $\alpha$  llist) glist logic
```

# Систематическое конструирование логических типов (на примере списка)

```
type  $\alpha$  logic = Var of int | Value of  $\alpha$ 
...
type ( $\alpha$ ,  $\beta$ ) glist = Nil | Cons of  $\alpha$  *  $\beta$ 

type  $\alpha$  list = ( $\alpha$ ,  $\alpha$  list) glist

type  $\alpha$  llist = ( $\alpha$ ,  $\alpha$  llist) glist logic

...
# Value Nil
-:  $\alpha$  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.
- Подход для описания типов.

# Промежуточные результаты

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

- Типобезопасное встраивание miniKanren в OCaml.
  - Полиморфная унификация.
  - Подход для описания типов.
- ✓ Типы помогают выявить некоторые ошибки.

# Промежуточные результаты

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

- Типобезопасное встраивание miniKanren в OCaml.
  - Полиморфная унификация.
  - Подход для описания типов.
- 
- ✓ Типы помогают выявить некоторые ошибки.
  - ✗ Выигрыша в скорости нет.
  - ✗ Даже наоборот.



## Дальнейшие задачи

- Найти причину замедления.
- Ускорить.
- Подход должен остаться типобезопасным.

# Полиморфная унификация в OScanren

Работает для всех логических типов  $\alpha^o$ :

$$(\equiv) : \Sigma \rightarrow \alpha^o \rightarrow \alpha^o \rightarrow \Sigma_{\perp}$$

# Полиморфная унификация в OScanren

Работает для всех логических типов  $\alpha^o$ :

$$(\equiv) : \Sigma \rightarrow \alpha^o \rightarrow \alpha^o \rightarrow \Sigma_{\perp}$$

Стандартный алгоритм с треугольной подстановкой и occurs check, использующий типонебезопасный интерфейс `Obj`.

# Полиморфная унификация в OSanren

Работает для всех логических типов  $\alpha^o$ :

$$(\equiv) : \Sigma \rightarrow \alpha^o \rightarrow \alpha^o \rightarrow \Sigma_{\perp}$$

Стандартный алгоритм с треугольной подстановкой и occurs check, использующий типонебезопасный интерфейс `Obj`.

Тонкости:

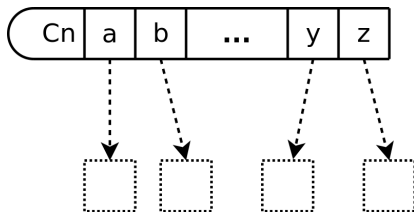
- Типонебезопасный стиль – компилятор теряет информацию о типах.
- Дополнительная стадия восстановления типов (refinement).

# Алгебраические типы данных в памяти

```
type ( $\alpha, \beta, \dots$ ) typ =  
|  $C_1$  of  $t_{11}$  *  $\dots$  *  $t_{1m}$   
|  $\dots$   
|  $C_n$  of  $t_{n1}$  *  $\dots$  *  $t_{nm}$ 
```

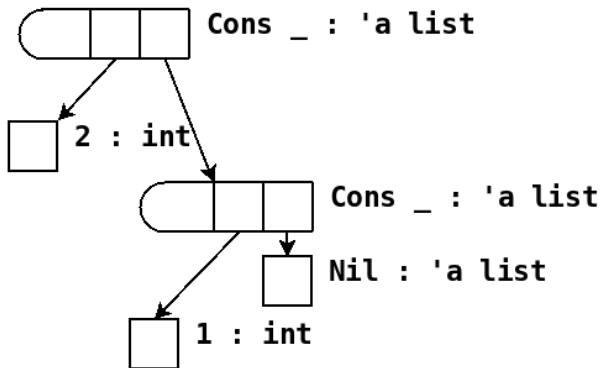
# Алгебраические типы данных в памяти

**type** ( $\alpha, \beta, \dots$ ) typ =  
|  $C_1$  **of**  $t_{11}$  \* ... \*  $t_{1m}$   
| ...  
|  $C_n$  **of**  $t_{n1}$  \* ... \*  $t_{nm}$



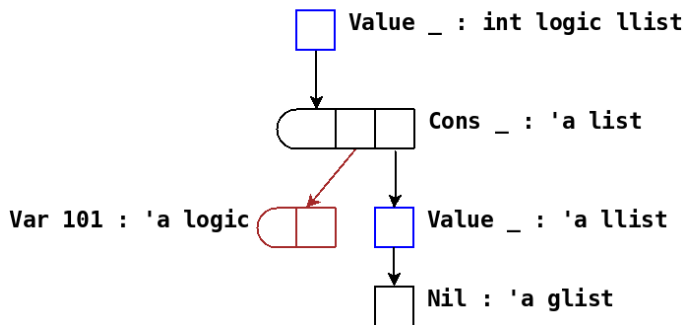
## Представление термов в памяти (пример для списка)

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



# Тегированное представление логических значений

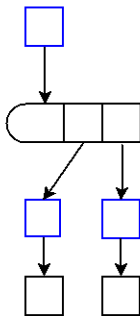
**Value (Cons (Var 101, Value Nil))  
: int llist**



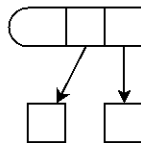


## Рост размера термов из-за тегирования

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



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

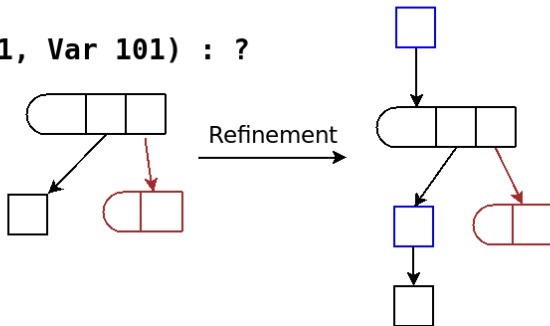


## Основная идея

- Унифицировать нетипизируемые термы.
- Преобразовывать к типизируемому представлению в самом конце (стадия refinement).
- Запоминать формальные типы значений при каждом преобразовании к логическому значению.

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

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



# План улучшения реализации

- Новое представление деревьев
  - Значению нельзя присвоить конкретный тип.
  - Нужен абстрактный тип значений.
  - Предоставить интерфейс для конструирования логических значений.
  - Преобразование абстрактного логического значения в типизируемое (во время `refinement`).
- Модернизировать подход по описанию типов логических значений.
- Не потерять типовую безопасность.

## Тип injected

**type** ( $\alpha$ ,  $\beta$ ) injected

Тип  $\alpha$  — это исходный тип, а тип  $\beta$  — его логическое представление

# Конструирование логических значений для простых типов

**type**  $(\alpha, \beta)$  injected

**val** lift:  $\alpha \rightarrow (\alpha, \alpha)$  injected

**val** inj :  $(\alpha, \beta)$  injected  $\rightarrow (\alpha, \beta \text{ logic})$  injected

# Конструирование логических значений для простых типов

```
type ( $\alpha$ ,  $\beta$ ) injected
```

```
val lift:  $\alpha \rightarrow (\alpha, \alpha)$  injected
```

```
val inj : ( $\alpha$ ,  $\beta$ ) injected  $\rightarrow (\alpha$ ,  $\beta$  logic) injected
```

Например, для чисел:

```
# inj (lift 5)
```

```
-: (int, int logic) injected
```

# Конструирование логических значений для простых типов

```
type ( $\alpha$ ,  $\beta$ ) injected
```

```
val lift:  $\alpha \rightarrow (\alpha, \alpha)$  injected
```

```
val inj : ( $\alpha$ ,  $\beta$ ) injected  $\rightarrow (\alpha$ ,  $\beta$  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  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end
```



# Конструирование логических значений для сложных типов

```
module Option = struct  
  type  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end  
  
# Make1(Option).distrib  
...
```

# Конструирование логических значений для сложных типов

```
module Option = struct  
  type  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end  
  
# Make1(Option).distrib  
...  
# let some x = inj (distrib (Some x))  
-: ( $\alpha$ ,  $\beta$ ) injected  $\rightarrow$  ( $\alpha$  option,  $\beta$  option logic) injected
```

# Конструирование логических значений для сложных типов

```
module Option = struct
  type  $\alpha$  option = None | Some of  $\alpha$ 
  let fmap = ...
end

# Make1(Option).distrib
...
# let some x = inj (distrib (Some x))
-: ( $\alpha$ ,  $\beta$ ) injected  $\rightarrow$  ( $\alpha$  option,  $\beta$  option logic) injected
```

Примитив **distrib**, который позволяет конструировать из значений типа **(\_, \_) injected** другие значения типа **(\_, \_) injected**.

Он ничего не делает со значением (identity).

## Восстановление посчитанных значений (refinement)

Необходимо, так как значения в типе  $(\alpha, \beta)$  `injected` хранятся в нетипизированном виде.

```
module Option = struct  
  type  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end
```

## Восстановление посчитанных значений (refinement)

Необходимо, так как значения в типе  $(\alpha, \beta)$  `injected` хранятся в нетипизированном виде.

```
module Option = struct  
  type  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end  
  
# Make1(Option).reify  
: (( $\alpha$ ,  $\beta$ ) injected  $\rightarrow$   $\beta$ )  $\rightarrow$ 
```


## Восстановление посчитанных значений (refinement)

Необходимо, так как значения в типе  $(\alpha, \beta)$  `injected` хранятся в нетипизированном виде.

```
module Option = struct  
  type  $\alpha$  option = None | Some of  $\alpha$   
  let fmap = ...  
end  
  
# Make1(Option).reify  
: (( $\alpha$ ,  $\beta$ ) injected  $\rightarrow$   $\beta$ )  $\rightarrow$   
  ( $\alpha$  option,  $\beta$  option logic) injected  $\rightarrow$   
   $\beta$  option logic
```

При построении `reify` функция `fmap` используется по существу.

# Итоговые результаты

- Типобезопасная реализация.
- С поддержкой неравенств термов (disequality constraints) и occurs check.
- Сравнима по скорости с faster-miniKanren .
- Типы выявляют простые ошибки.
- Типобезопасность могут заменить некоторые проверки во время выполнения (abstento, numero, simbolo)...
- ... сокращая количество примитивов с 8 до 5 ( $\equiv$ ,  $\neq$ , конъюнкция, дизъюнкция, fresh).

Репозиторий 