Уменьшение цены абстракции при типобезопасном встраивании реляционнного языка программирования в OCaml

Дмитрий Косарев

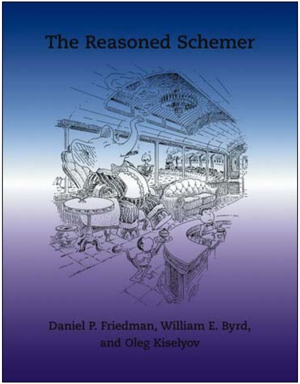
Санкт-Петербургский Государтсвенный Университет
JetBrains Research

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)

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

```
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)

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

```
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))
```

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)

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

```
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)
```

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)
     (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))]))
```

A Sketch of Minimalistic Implementation

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

A Sketch of Minimalistic Implementation

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

Logic variables $\qquad X = \{x_1, x_2, \dots\}$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\,(t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\,(t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\,(t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \ldots\}$
Symbols (constructors)	$S = \{s_1, s_2, \ldots\}$
Terms	$T = X \cup \{s\,(t_1, \ldots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ
Goal (a function from a state to a stream of states)	$g : \sigma \to \sigma$ stream

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ
Goal (a function from a state to a stream of states)	$g : \sigma \to \sigma$ stream
Conjunction $g \wedge g$	"bind"

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ
Goal (a function from a state to a stream of states)	$g : \sigma \to \sigma$ stream
Conjunction $g \wedge g$	"bind"
Disjunction $g \vee g$	"mplus"

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ
Goal (a function from a state to a stream of states)	$g : \sigma \to \sigma$ stream
Conjunction $g \wedge g$	"bind"
Disjunction $g \vee g$	"mplus"
Refinement of answers	refine $: \sigma \to X \to T$

A Sketch of Minimalistic Implementation

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

Logic variables	$X = \{x_1, x_2, \dots\}$
Symbols (constructors)	$S = \{s_1, s_2, \dots\}$
Terms	$T = X \cup \{s\ (t_1, \dots, t_k) \mid s \in S,\ t_i \in T\}$
Substitutions	$\Sigma = T^X$
Unification	$(\equiv) : \Sigma \to T \to T \to \Sigma_\perp$
State (a substitution + some info to create fresh variables)	σ
Goal (a function from a state to a stream of states)	$g : \sigma \to \sigma$ stream
Conjunction $g \wedge g$	"bind"
Disjunction $g \vee g$	"mplus"
Refinement of answers	refine $: \sigma \to X \to T$

Unification and refinement are virtually the main things to implement

Polymorphic Unification

Works for all logic types α^o:

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

Polymorphic Unification

Works for all logic types α^o:

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

Is implemented as the standard algorithm with triangular substitution and occurs check by traversing runtime representation, using unsafe interface `Obj`.

Polymorphic Unification

Works for all logic types α^o:

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

Is implemented as the standard algorithm with triangular substitution and occurs check by traversing runtime representation, using unsafe interface Obj.

Pitfalls:

- compiler loses the track of types after the results of unification are stored in a substitution $\rightsquigarrow$ refinement has to be implemented untyped as well;

Polymorphic Unification

Works for all logic types α^o:

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

Is implemented as the standard algorithm with triangular substitution and occurs check by traversing runtime representation, using unsafe interface Obj.

Pitfalls:

- compiler loses the track of types after the results of unification are stored in a substitution $\rightsquigarrow$ refinement has to be implemented untyped as well;
- the safety of unification/refinement implementation has to be justified separately;

Polymorphic Unification

Works for all logic types α^o:

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

Is implemented as the standard algorithm with triangular substitution and occurs check by traversing runtime representation, using unsafe interface Obj.

Pitfalls:

- compiler loses the track of types after the results of unification are stored in a substitution $\rightsquigarrow$ refinement has to be implemented untyped as well;
- the safety of unification/refinement implementation has to be justified separately;
- states must not escape their scope (otherwise the coherence between variable types and terms, stored in states, can be lost).

Capturing the States

States and refinement function are hidden and can not be accessed directly.

Capturing the States

States and refinement function are hidden and can not be accessed directly.

The refinement is performed transparently as the top-level running primitive is invoked:

Capturing the States

States and refinement function are hidden and can not be accessed directly.

The refinement is performed transparently as the top-level running primitive is invoked:

$$\text{run}\ \bar{n}\ \ (\text{fun}\ \ q_1\, q_2\, \ldots\, q_n\ \rightarrow \text{g})\ \ (\text{fun}\ \ a_1\, a_2\, \ldots\, a_n\ \rightarrow \text{h})$$

Capturing the States

States and refinement function are hidden and can not be accessed directly.

The refinement is performed transparently as the top-level running primitive is invoked:

$$\text{run}\ \bar{n}\ \ (\text{fun}\ \ q_1\, q_2\, \dots\, q_n\ \to \text{g})\ \ (\text{fun}\ \ a_1\, a_2\, \dots\, a_n\ \to \text{h})$$

Here:

- run — the only way to run goals;
- $\bar{n}$ — a numeral, describing the number of fresh variables, available for running the goal g; numerals can be manufactured quantum satis using the successor function, which is provided as well;
- $q_1, q_2 \dots q_n$ — these fresh variables;
- $a_1, a_2 \dots a_n$ — the streams of refined answers for the variables $q_1, q_2 \dots q_n$ respectively;
- h — a handler, which can make use of refined answers.

The framework guarantees, that variables are refined only in correct states.

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{aligned} \uparrow_t &: t \to t^o \\ \downarrow_t &: t^o \to t \end{aligned}$$

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{rcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{rcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
```

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{lcl} \uparrow_t & : & t \rightarrow t^o \\ \downarrow_t & : & t^o \rightarrow t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
```

$\rightsquigarrow$

```
type ('int, 'tree) tree_f = Leaf of 'int | Node of 'tree * 'tree
```

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{lcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
⇝
type ('int, 'tree) tree_f = Leaf of 'int | Node of 'tree * 'tree
type tree  = (int, tree) tree_f
```

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{lcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
⇝
type ('int, 'tree) tree_f = Leaf of 'int | Node of 'tree * 'tree
type tree  = (int, tree) tree_f
type tree^o = ((int^o, tree^o) tree_f)^o
```

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{rcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
```

$\leadsto$

```
type ('int, 'tree) tree_f = Leaf of 'int | Node of 'tree * 'tree
type tree  = (int, tree) tree_f
type tree^o = ((int^o, tree^o) tree_f)^o
```

```
let rec (↑tree) t = ↑∀ ( fmap_tree_f (↑∀) (↑tree) t)
```

Converting from- and to User Types

Injecting a user-type into logic domain and projecting the logical results back:

$$\begin{array}{lcl} \uparrow_t & : & t \to t^o \\ \downarrow_t & : & t^o \to t \end{array}$$

Can be done systematically using generic programming:

- "$\uparrow_\forall$", "$\downarrow_\forall$" are polymorphic shallow injection/projection;
- for the deep case, make the type a functor and use *fmap*.

```
type tree = Leaf of int | Node of tree * tree
```

$\leadsto$

$\texttt{type ('int, 'tree) tree}_f \texttt{ = Leaf of 'int | Node of 'tree * 'tree}$
$\texttt{type tree = (int, tree) tree}_f$
$\texttt{type tree}^o \texttt{ = ((int}^o\texttt{, tree}^o\texttt{) tree}_f\texttt{)}^o$

$\texttt{let rec } (\uparrow_{\mathrm{tree}}) \texttt{ t = } \uparrow_\forall \texttt{ (} \mathit{fmap}_{\mathrm{tree}_f} \texttt{ (}\uparrow_\forall\texttt{) (}\uparrow_{\mathrm{tree}}\texttt{) t)}$

$\texttt{let rec } (\downarrow_{\mathrm{tree}}) \texttt{ l = } \mathit{fmap}_{\mathrm{tree}_f} \texttt{ (}\downarrow_\forall\texttt{) (}\downarrow_{\mathrm{tree}}\texttt{) (}\downarrow_\forall \texttt{ l)}$

Example

Current Implementation

- Repository: https://github.com/dboulytchev/OCanren
- Implements μKanren + disequality constraints
- Passes most of the original tests
- Outperforms μKanren on long queries

last

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

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