Typed Embedding of a Relational Language in OCaml

Dmitrii Kosarev, Dmitrii Boulytchev

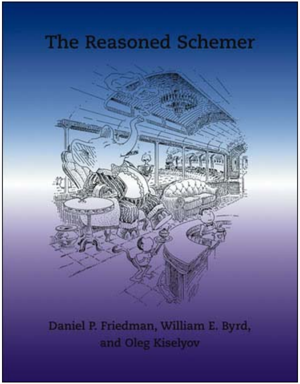
Saint-Petersburg State University
JetBrains Research

ML Family Workshop
September 22, 2016
Nara, Japan

Relational Programming in miniKanren

From programs as *functions* to programs as *relations*:

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



- Daniel P. Friedman, William Byrd and Oleg Kiselyov. *The Reasoned Schemer*, The MIT Press, Cambridge, MA, 2005
- A DSL for Scheme/Racket with rather simple minimal implementation
- A family of languages (μKanren, α-Kanren, cKanren etc.)
- Implemented as DSL for a wide range of host languages (including OCaml, Haskell, Scala etc.)

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

```
let rec append xs ys
```

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

```
let rec append xs ys =
  match xs with
```

An Example: Relational List Append

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

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

```
let rec append xs ys =
  match xs with
  | []    → ys
```

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

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

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

```
let rec appendᵒ xs ys xys
```

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

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list} \rightarrow \alpha\,\texttt{list}$

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

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

```
let rec appendᵒ xs ys xys =
  ((xs ≡ nil) &&& (xys ≡ ys))
```

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

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

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

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

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

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

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

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

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

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

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

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

An Example: Relational List Append

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

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

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

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

An Example: Relational List Append

$\texttt{append}: \alpha\,\texttt{list} \to \alpha\,\texttt{list} \to \alpha\,\texttt{list}$

$\texttt{append}^o \subseteq \alpha\,\texttt{list} \times \alpha\,\texttt{list} \times \alpha\,\texttt{list}$

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

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

```
(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 $X = \{x_1, x_2, \dots\}$

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)	$g : \sigma \to \sigma\ \mathtt{stream}$

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)	$g : \sigma \to \sigma\ \texttt{stream}$
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)	$g : \sigma \to \sigma\ \texttt{stream}$
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)	$g : \sigma \to \sigma\ \mathtt{stream}$
Conjunction $g \wedge g$	"bind"
Disjunction $g \vee g$	"mplus"

Unification is virtually the main thing to implement

Dealing with Typed Terms

- Non-solution:
  - implement unification for a fixed term representation;
  - convert user-type data to- and from that universal representation.

Polymorphic Unification

- Types are erased in substitution
- Have to be reconstructed properly during refinement
- States must not escape the scope

Datatype-Generic Pattern

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

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

Datatype-Generic Pattern

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.

Datatype-Generic Pattern

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
```

Datatype-Generic Pattern

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
```

Datatype-Generic Pattern

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
```

Datatype-Generic Pattern

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

$\rightsquigarrow$

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

Datatype-Generic Pattern

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*.

$\textbf{type}\ \texttt{tree} = \texttt{Leaf}\ \textbf{of}\ \texttt{int} \mid \texttt{Node}\ \textbf{of}\ \texttt{tree} * \texttt{tree}$

$\rightsquigarrow$

$\textbf{type}\ (\texttt{'int}, \texttt{'tree})\ \texttt{tree}_f = \texttt{Leaf}\ \textbf{of}\ \texttt{'int} \mid \texttt{Node}\ \textbf{of}\ \texttt{'tree} * \texttt{'tree}$

$\textbf{type}\ \texttt{tree} = (\texttt{int}, \texttt{tree})\ \texttt{tree}_f$

$\textbf{type}\ \texttt{tree}^o = ((\texttt{int}^o, \texttt{ltree})\ \texttt{tree}_f)^o$

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

Datatype-Generic Pattern

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*.

$\textbf{type}\ \texttt{tree = Leaf}\ \textbf{of}\ \texttt{int}\ |\ \texttt{Node}\ \textbf{of}\ \texttt{tree} * \texttt{tree}$

$\rightsquigarrow$

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

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

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

Capturing the States

.

Example

.

Current Implementation

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