Typed Embedding of a Relational Language in OCaml

Dmitrii Kosarev, Dmitrii Boulytchev

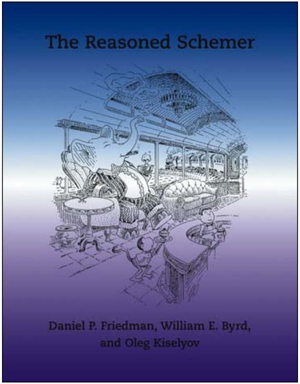
Saint-Petersburg State University
JetBrains Research

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

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

An Example: Relational List Append

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

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

```
let rec append xs ys
```

An Example: Relational List Append

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

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

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

An Example: Relational List Append

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

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

```
let rec append xs ys =
  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

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

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

```
let rec appendᵒ xs ys xys
```

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

An Example: Relational List Append

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

```
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))
```

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)
```

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

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

```
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)) |||
  (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

`append: α list → α list → α list`

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

`appendᵒ ⊆ α list × α list × α 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)
  )
```

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

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

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)	$g : \sigma \to \sigma\ \mathtt{stream}$
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)	$g : \sigma \to \sigma\ \texttt{stream}$
Conjunction $g \wedge g$	"bind"
Disjunction $g \vee g$	"mplus"
Refinement of answers	$\texttt{refine} : \sigma \to X \to T$

Unification and refinement are virtually the main things to implement

Dealing with Typed Terms

Dealing with Typed Terms

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

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically
- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad-hoc$ polymorphism in OCaml yet).

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad - hoc$ polymorphism in OCaml yet).

  Boilerplatish

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad-hoc$ polymorphism in OCaml yet).

  Boilerplatish

  Users would need to adjust their types to represent logical variables

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad-hoc$ polymorphism in OCaml yet).

  Boilerplatish

  Users would need to adjust their types to represent logical variables

- Polymorphic unification:

$$\equiv : \Sigma \to \alpha \to \alpha \to \Sigma_{\perp}$$

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad-hoc$ polymorphism in OCaml yet).

  Boilerplatish

  Users would need to adjust their types to represent logical variables

- Polymorphic unification:

$$\equiv : \Sigma \to \alpha \to \alpha \to \Sigma_{\perp}$$

  Has to be implemented in an untyped manner

Dealing with Typed Terms

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

  Doest not help in detecting the mistakes statically

- Bad solution: generate boilerplate unification code for each user type (there is no direct support for $ad-hoc$ polymorphism in OCaml yet).

  Boilerplatish

  Users would need to adjust their types to represent logical variables

- Polymorphic unification:

$$\equiv : \Sigma \to \alpha \to \alpha \to \Sigma_{\perp}$$

  Has to be implemented in an untyped manner

  Might be a good solution (lightweight, efficient), if type safety is justified

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 $\leadsto$ 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 $\leadsto$ 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).

Properties of Polymorphic Unification

It can be shown, that for our concrete implementation:

Properties of Polymorphic Unification

It can be shown, that for our concrete implementation:

- variables in a substitution are always associated with the terms of the same type;

Properties of Polymorphic Unification

It can be shown, that for our concrete implementation:

- variables in a substitution are always associated with the terms of the same type;
- all variables preserves their types, assigned by the compiler;

Properties of Polymorphic Unification

It can be shown, that for our concrete implementation:

- variables in a substitution are always associated with the terms of the same type;
- all variables preserves their types, assigned by the compiler;
- all variables occur in terms only in a “type-safe” positions:

$$t[x] \iff \text{the type of } x \text{ corresponds to the type of the hole of } t$$

Properties of Polymorphic Unification

It can be shown, that for our concrete implementation:

- variables in a substitution are always associated with the terms of the same type;
- all variables preserves their types, assigned by the compiler;
- all variables occur in terms only in a “type-safe” positions:

$$t[x] \iff \text{the type of } x \text{ corresponds to the type of the hole of } t$$

$$\leadsto$$

the refinement is type-safe, if a variable is refined in a state, which is an inheritor of the state that variable was created in.

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:

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

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:

$$\texttt{run}\ \bar{n}\ (\textbf{fun}\ q_1\ q_2\ \ldots\ q_n \to \texttt{g})\ (\textbf{fun}\ a_1\ a_2\ \ldots\ a_n \to \texttt{h})$$

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;
- $q_1, q_2 \ldots q_n$ — these fresh variables;
- $a_1, a_2 \ldots a_n$ — the streams of *refined* answers for the variables $q_1, q_2 \ldots q_n$ respectively;
- `h` — a *handler*, which can make use of refined answers.

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

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:

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

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
~>
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*.

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

$\leadsto$

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

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

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

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*.

$\mathbf{type}\ \mathtt{tree} = \mathtt{Leaf}\ \mathbf{of}\ \mathtt{int} \mid \mathtt{Node}\ \mathbf{of}\ \mathtt{tree} * \mathtt{tree}$

$\rightsquigarrow$

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

$\mathbf{type}\ \mathtt{tree} = (\mathtt{int},\ \mathtt{tree})\ \mathtt{tree}_f$

$\mathbf{type}\ \mathtt{tree}^o = ((\mathtt{int}^o,\ \mathtt{tree}^o)\ \mathtt{tree}_f)^o$

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

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*.

```
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

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

Example

.

Current Implementation

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