Relational Programming, Interpreters, and Program Synthesis

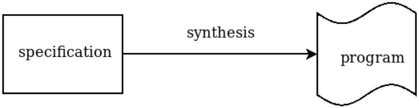
Dmitry Boulytchev

Saint Petersburg State University, Huawei

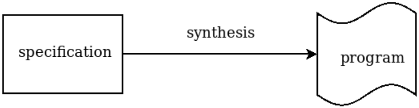
Software Engineering, Theory and Experimental Programming

February 22, 2023
Online

Program Synthesis

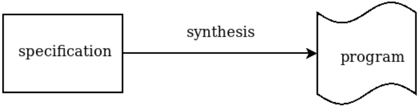


Program Synthesis

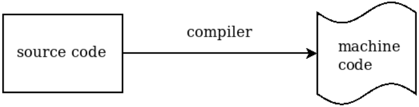


Applications of recursive arithmetic to the problem of circuit synthesis [Church, 1957]

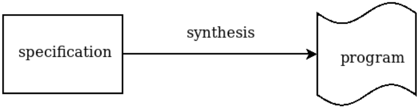
Program Synthesis



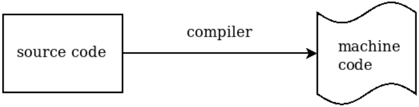
Applications of recursive arithmetic to the problem of circuit synthesis [Church, 1957]



Program Synthesis



Applications of recursive arithmetic to the problem of circuit synthesis [Church, 1957]



Solved?

“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.

“The Monkey’s Paw” Curse

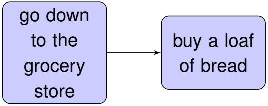
“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.

“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.

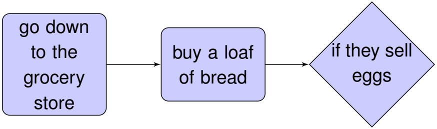
“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.



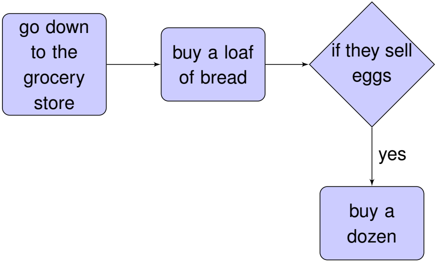
“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.



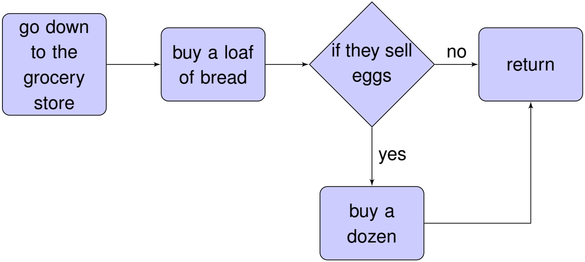
“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.



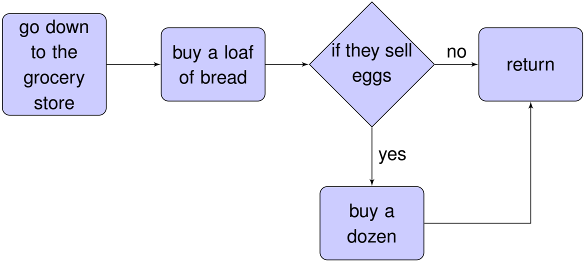
"The Monkey's Paw" Curse

"The Monkey's Paw" (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one's wishes.



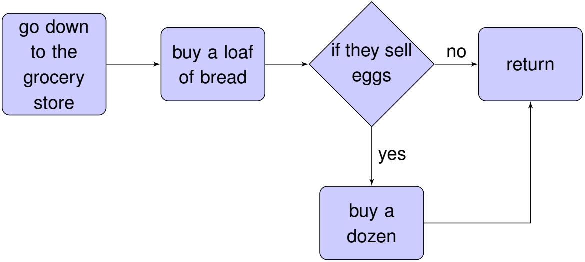
“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.



“The Monkey’s Paw” Curse

“The Monkey’s Paw” (*W. W. Jacobs*, 1902): a story of a magic talisman which *literally* fulfills one’s wishes.



Implementation vs. Specification

Program synthesis from polymorphic refinement types
[Polikarpova, Kuraj, Solar-Lezama, PLDI-2016]

Implementation vs. Specification

Program synthesis from polymorphic refinement types
[Polikarpova, Kuraj, Solar-Lezama, PLDI-2016]

specification	code

Implementation vs. Specification

Program synthesis from polymorphic refinement types
[Polikarpova, Kuraj, Solar-Lezama, PLDI-2016]

specification	code
insert :: x: $\alpha \rightarrow$ t: BST $\alpha \rightarrow$ {BST α | keys $\nu =$ keys t + [x]}	

Implementation vs. Specification

Program synthesis from polymorphic refinement types
[Polikarpova, Kuraj, Solar-Lezama, PLDI-2016]

specification	code
insert::x:$\alpha \to$ t: BST $\alpha \to$ {BST α | keys $\nu =$ keys t + [x]}	see below

```
insert = λx . λt . match t with
  | Empty → Node x Empty Empty
  | Node y l r → if x ≤ y ∧ y ≤ x
     then t
     else if y ≤ x
       then Node y l (insert x r)
       else Node y (insert x l) r
```

Implementation vs. Specification

Program synthesis from polymorphic refinement types
[Polikarpova, Kuraj, Solar-Lezama, PLDI-2016]

specification	**code**

insert :: x:α $\to$ t: BST α $\to$ {BST α | keys ν = keys t + [x]}
 	
termination measure size :: BST α $\to$ Int
measure keys :: BST α $\to$ Set α
data BST α **where**
  Empty :: {BST α | keys ν = []}
  Node :: x:α $\to$ l: BST{α | ν < x} $\to$ r: BST{α | x < ν}
    $\to$ {BST α | keys ν = keys l + keys r + [x]}
	insert = λx . λt . **match** t **with**
	  | Empty $\to$ Node x Empty Empty
	  | Node y l r $\to$ **if** x $\leq$ y $\wedge$ y $\leq$ x
	    then t
	    else if y $\leq$ x
	      then Node y l (insert x r)
	      else Node y (insert x l) r

The Program Synthesis Challenge

- Specifications tend to contain errors.
- Sometimes *imperative* description is more robust then *declarative*.
- Sometimes specification is more verbose and harder to write than implementation.

Program Transformations

Idea: make a program from another simpler one

Advantage: use conventional SE techniques to debug & test the simpler one

Program Inversion:

- sorting $\rightarrow$ permutations
- type checking $\rightarrow$ type inference
- verifier $\rightarrow$ solver

From Standard to Non-Standard Semantics by Semantics Modifiers [Abramov, Glück, 2001]

Program Synthesis by Inversion

$$p\ x \mapsto y$$

Program Synthesis by Inversion

$$p\,x \mapsto y$$

`sort` [5; 3; 2; 4; 1] = [1; 2; 3; 4; 5]

Program Synthesis by Inversion

$$p\ x \mapsto y$$

`sort` [5; 3; 2; 4; 1] = [1; 2; 3; 4; 5]

$$p^{-1}\ y \mapsto \{x \mid p\ x \mapsto y\}$$

Program Synthesis by Inversion

$$p\,x \mapsto y$$

`sort [5; 3; 2; 4; 1] = [1; 2; 3; 4; 5]`

$$p^{-1}\,y \mapsto \{x \mid p\,x \mapsto y\}$$

$\texttt{sort}^{-1}$ `[1; 2; 3; 4; 5] = { l | sort l = [1; 2; 3; 4; 5] }`

Program Synthesis by Inversion

$$p\,x \mapsto y$$

`sort [5; 3; 2; 4; 1] = [1; 2; 3; 4; 5]`

$$p^{-1}\,y \mapsto \{x \mid p\,x \mapsto y\}$$

$\texttt{sort}^{-1}$ `[1; 2; 3; 4; 5] = { l | sort l = [1; 2; 3; 4; 5] }`

$$\mathit{invert}\ p = p^{-1}$$

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

functional	**relational**
$p\ x \mapsto y$	

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

functional	**relational**
$p\ x \mapsto y$	$p^o\ x\ y \mapsto \{\textbf{success}, \textbf{failure}\}$

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

functional	**relational**
$p\ x \mapsto y$	$p^o\ x\ y \mapsto \{\textbf{success}, \textbf{failure}\}$
	↓ adding free variables

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

functional	**relational**
$p\ x \mapsto y$	$p^o\ x\ y \mapsto \{\textbf{success}, \textbf{failure}\}$
	↓ adding free variables
	$p^o\ x\ y \mapsto \{\text{substitutions for free variables in } x \text{ and } y\}$

Inversion by Relational Programming

Programs as *functions* vs. programs as *relations*

functional	**relational**
$p\ x \mapsto y$	$p^o\ x\ y \mapsto \{\textbf{success}, \textbf{failure}\}$
	↓ adding free variables
	$p^o\ x\ y \mapsto \{$substitutions for free variables in x and $y\}$
	$\texttt{sort}^o\ \alpha\ [1;2;3;4;5]$

MINIKANREN

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

MINIKANREN

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

PROLOG = HC + DFS + EXTRA-LOGIC FEATURES

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

PROLOG = HC + DFS + EXTRA-LOGIC FEATURES

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

Prolog = HC + DFS + Extra-Logic Features

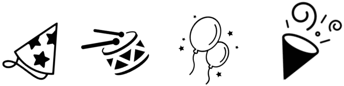
A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

Prolog = HC + DFS + Extra-Logic Features

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

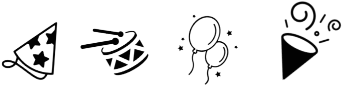


PROLOG = HC + DFS + EXTRA-LOGIC FEATURES

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]

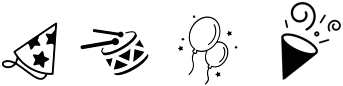


Prolog = HC + DFS + Extra-Logic Features

miniKanren = HC + Complete Interleaving Search

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]



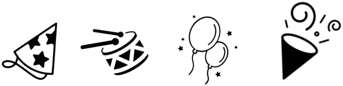
Prolog = HC + DFS + Extra-Logic Features

miniKanren = HC + Complete Interleaving Search

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]



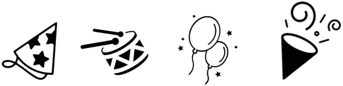
Prolog = HC + DFS + Extra-Logic Features

miniKanren = HC + Complete Interleaving Search

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]



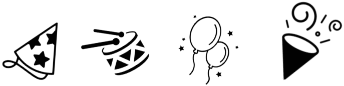
PROLOG = HC + DFS + EXTRA-LOGIC FEATURES

miniKanren = HC + COMPLETE INTERLEAVING SEARCH

miniKanren

A minimalistic relational language in the form of a DSL

The Reasoned Schemer [Byrd, Friedman, Kiselyov, 2005]



Prolog = HC + DFS + Extra-Logic Features

miniKanren = HC + Complete Interleaving Search

miniKanren: Syntax

Terms:

$$
\begin{aligned}
\mathcal{X} &= \{x_1, x_2, \ldots\} \\
\mathcal{F} &= \{f_1^{k_1}, f_2^{k_2}, \ldots\} \\
\mathcal{T}_X &= X \cup \{f^n(t_1, \ldots, t_n) \mid f^n \in \mathcal{F}, t_i \in \mathcal{T}_X\}
\end{aligned}
$$

MINIKANREN: Syntax

Terms:

$$
\begin{array}{rcl}
\mathcal{X} & = & \{x_1, x_2, \ldots\} \\
\mathcal{F} & = & \{f_1^{k_1}, f_2^{k_2}, \ldots\} \\
\mathcal{T}_X & = & X \cup \{f^n(t_1, \ldots, t_n) \mid f^n \in \mathcal{F}, t_i \in \mathcal{T}_X\}
\end{array}
$$

Goals (all terms are in $\mathcal{T}_\mathcal{X}$):

$$
\begin{array}{rcl}
\mathcal{G} & = & t_1 \equiv t_2 \\
 & & g_1 \wedge g_2 \\
 & & g_1 \vee g_2 \\
 & & \exists\, x\, .\, g \\
 & & R\, t_1 \ldots t_k
\end{array}
$$

miniKanren: Syntax

Terms:

$$\begin{array}{rcl} \mathcal{X} & = & \{x_1, x_2, \dots\} \\ \mathcal{F} & = & \{f_1^{k_1}, f_2^{k_2}, \dots\} \\ \mathcal{T}_X & = & X \cup \{f^n(t_1, \dots, t_n) \mid f^n \in \mathcal{F}, t_i \in \mathcal{T}_X\} \end{array}$$

Goals (all terms are in $\mathcal{T}_{\mathcal{X}}$):

$$\begin{array}{rcl} \mathcal{G} & = & t_1 \equiv t_2 \\ & & g_1 \wedge g_2 \\ & & g_1 \vee g_2 \\ & & \exists\, x \,.\, g \\ & & R\, t_1 \dots t_k \end{array}$$

Definitions:

$$R = \lambda\, x_1 \dots x_k \,.\, g$$

miniKanren → Prolog

$$example \;=\; \lambda x\, y\,.\, y \equiv A \wedge (\exists z\,.\, x \equiv B(z) \vee foo(y, z))$$

miniKanren → Prolog

$$example = \lambda x\, y\, .\, y \equiv A \wedge (\exists z\, .\, x \equiv B(z) \vee foo(y, z))$$

1. remove existential quantifiers:

$$example = \lambda x\, y\, .\, y \equiv A \wedge (x \equiv B(z) \vee foo(y, z))$$

miniKanren → Prolog

$$example = \lambda x\, y\,.\, y \equiv A \wedge (\exists z\,.\, x \equiv B(z) \vee foo(y, z))$$

1. remove existential quantifiers:

$$example = \lambda x\, y\,.\, y \equiv A \wedge (x \equiv B(z) \vee foo(y, z))$$

2. transform to DNF (by folding suitable subgoals into auxilliary definitions);

$$\begin{array}{rcl} example' & = & \lambda x\, y\, z\,.\, x \equiv B(z) \vee foo(y, z) \\ example & = & \lambda x\, y\,.\, y \equiv A \wedge example'(x, y, z) \end{array}$$

miniKanren → Prolog

$$example \;=\; \lambda x\, y\,.\, y \equiv A \wedge (\exists z\,.\, x \equiv B(z) \vee foo(y, z))$$

1. remove existential quantifiers:

$$example \;=\; \lambda x\, y\,.\, y \equiv A \wedge (x \equiv B(z) \vee foo(y, z))$$

2. transform to DNF (by folding suitable subgoals into auxilliary definitions);

$$\begin{array}{rcl} example' & = & \lambda x\, y\, z\,.\, x \equiv B(z) \vee foo(y, z) \\ example & = & \lambda x\, y\,.\, y \equiv A \wedge example'(x, y, z) \end{array}$$

3. introduce a separate Horn clause for each of the disjuncts, pulling the top-level unifications to the head's arguments:

$$\begin{array}{lll} example' & (b(Z), \_, Z). & \\ example' & (\_, Y, Z) & \vdash \; foo(Y, Z). \\ example & (X, a) & \vdash \; example'(X, a, Z). \end{array}$$

PROLOG → MINIKANREN

$$
\begin{array}{l}
append([], Y, Y). \\
append([H|T], Y, [H|TY]) \quad \vdash \quad append(T, Y, TY).
\end{array}
$$

PROLOG → MINIKANREN

$$
\begin{aligned}
&append([], Y, Y).\\
&append([H|T], Y, [H|TY]) \quad \vdash \quad append(T, Y, TY).
\end{aligned}
$$

1. rename the arguments in Horn clauses heads coherently adding explicit unifications where needed;

$$
\begin{aligned}
append(X, Y, Z) &\quad \vdash \quad X \equiv [] \wedge Y \equiv Z.\\
append(X, Y, Z) &\quad \vdash \quad X \equiv [H|T] \wedge Z \equiv [H|TY] \wedge append(T, Y, TY).
\end{aligned}
$$

PROLOG → MINIKANREN

$$append([], Y, Y).$$
$$append([H|T], Y, [H|TY]) \quad \vdash \quad append(T, Y, TY).$$

1. rename the arguments in Horn clauses heads coherently adding explicit unifications where needed;

$$append(X, Y, Z) \quad \vdash \quad X \equiv [] \wedge Y \equiv Z.$$
$$append(X, Y, Z) \quad \vdash \quad X \equiv [H|T] \wedge Z \equiv [H|TY] \wedge append(T, Y, TY).$$

2. introduce explicit existential quantifiers:

$$append(X, Y, Z) \quad \vdash \quad X \equiv [] \wedge Y \equiv Z.$$
$$append(X, Y, Z) \quad \vdash \quad \exists H\, T\, TY\, .\, X \equiv [H|T] \wedge Z \equiv [H|TY] \wedge append(T, Y, TY).$$

PROLOG $\rightarrow$ MINIKANREN

$$\begin{array}{l} append\,([], Y, Y). \\ append\,([H|T], Y, [H|TY]) \quad \vdash \quad append\,(T,\, Y,\, TY). \end{array}$$

1. rename the arguments in Horn clauses heads coherently adding explicit unifications where needed;

$$\begin{array}{lll} append\,(X, Y, Z) & \vdash & X \equiv [] \wedge Y \equiv Z. \\ append\,(X, Y, Z) & \vdash & X \equiv [H|T] \wedge Z \equiv [H|TY] \wedge append\,(T,\, Y,\, TY). \end{array}$$

2. introduce explicit existential quantifiers:

$$\begin{array}{lll} append\,(X, Y, Z) & \vdash & X \equiv [] \wedge Y \equiv Z. \\ append\,(X, Y, Z) & \vdash & \exists H\, T\, TY\,.\, X \equiv [H|T] \wedge Z \equiv [H|TY] \wedge append\,(T,\, Y,\, TY). \end{array}$$

3. join the clauses with the same heads into the one relational definition using disjunction:

$$append \;=\; \lambda x, y, z\,.\,(x \equiv [] \wedge y \equiv z) \vee \exists h\, t\, ty\,.\, x \equiv h : t \wedge append\; t\; y\; ty)$$

miniKanren: Semantics

- Denotational: least Herbrand model.

miniKanren: Semantics

- Denotational: least Herbrand model.
- Operational: *occurs check* + *interleaving* search:
  - sound & complete w.r.t. LHM;
  - refutationally *incomplete*.

miniKanren: Semantics

- Denotational: least Herbrand model.
- Operational: *occurs check* + *interleaving* search:
  - sound & complete w.r.t. LHM;
  - refutationally *incomplete*.

Typed Relational Conversion [Lozov, Vyatkin, Boulytchev, TFP-2017]

Certified Semantics for Relational Programming [Rozplokhas, Boulytchev, APLAS-2020]

Certified Semantics with Disequality [Rozplikhas, Boulytchev, miniKanren-2021]

Interleaving Search: Idea

Idea: build a semantics of a goal as a *state*-transforming function:

$$[\![\bullet]\!] : \mathcal{G} \to \mathit{St} \to \mathit{St}^*$$

Interleaving Search: Idea

Idea: build a semantics of a goal as a *state*-transforming function:

$$[\![\bullet]\!] : \mathcal{G} \to \mathit{St} \to \mathit{St}^*$$

- St — states (contain everything needed to make a step);
- St^* — a *lazy* stream of states.

Interleaving Search: Idea

Idea: build a semantics of a goal as a *state*-transforming function:

$$[\![\bullet]\!] : \mathcal{G} \to St \to St^*$$

- St — states (contain everything needed to make a step);
- St^* — a *lazy* stream of states.

Laziness is important for completeness:

$$foo = \lambda\, x\,.\, foo\, x \vee x \equiv 5$$

Interleaving Search: Blueprint

$$
\begin{array}{rcl}
[\![t_1 \equiv t_2]\!]\,\sigma & = & \left\{\begin{array}{ll} \varepsilon & , \quad t_1 \text{ and } t_2 \text{ do not have a unifier} \\ MGU(\sigma, t_1, t_2) & , \quad \text{otherwise} \end{array}\right. \\
[\![g_1 \wedge g_2]\!]\,\sigma & = & concat\ (map\ [\![g_2]\!]\ ([\![g_1]\!]\,\sigma)) \\
[\![g_1 \vee g_2]\!]\,\sigma & = & [\![g_1]\!]\,\sigma \oplus [\![g_2]\!]\,\sigma \\
[\![\exists\, x\,.\,g]\!]\,\sigma & = & [\![g\,[x \leftarrow \alpha]]\!]\,\sigma', \langle \alpha, \sigma' \rangle = \mathbf{fresh}\,\sigma \\
[\![R\,t_1 \dots t_k]\!]\,\sigma & = & [\![g\,[x_1 \leftarrow t_1]]\!]\,\sigma, R = \lambda\,x_1 \dots x_k\,.\,g
\end{array}
$$

Interleaving Search: Details

$$
\begin{array}{rcl}
\mathcal{A} & = & \{\alpha_1, \alpha_2, \ldots\} \text{ (all terms now in } \mathcal{T}_{\mathcal{X}\cup\mathcal{A}}) \\
\Sigma & = & \mathcal{A} \to \mathcal{T}_{\mathcal{A}} \\
St & = & \Sigma \times \mathfrak{P}(\mathcal{A}) \\
St_0 & = & \langle \Lambda, \mathcal{A} \rangle \\
MGU(\langle \sigma, P \rangle, t_1, t_2) & = & \langle mgu(t_1\sigma, t_2\sigma), P \rangle \\
\mathbf{fresh}\ \langle \sigma, P \rangle & = & \langle \alpha, \langle \sigma, P \setminus \{\alpha\} \rangle \rangle, \alpha \in P \\
f \oplus g & = & \left\{ \begin{array}{rcl} g & , & f = \varepsilon \\ \sigma(g \oplus f') & , & f = \sigma f' \end{array} \right.
\end{array}
$$

Backtracking, Interleaving, and Terminating Monad Transformers

[Kiselyov, Chung-chieh Shan, Friedman, Sabry, ICFP-2005]

OCanren: a Strongly Typed miniKanren for OCaml

Typed Embedding of a Relational Language in OCaml [Kosarev, Boulytchev, ML-2016]

OCanren: a Strongly Typed miniKanren for OCaml

Typed Embedding of a Relational Language in OCaml [Kosarev, Boulytchev, ML-2016]

```
let rec appendᵒ x y xy = conde [
  ((x ≡ nil ()) &&& (y ≡xy));
  call_fresh (fun h →
    call_fresh (fun t →
      call_fresh (fun ty →
        (x ≡ h % t)  &&&
        (h % ty ≡ xy) &&&
        (appendᵒ t y ty'))))
]
```

OCANREN: a Strongly Typed MINIKANREN for OCAML

Typed Embedding of a Relational Language in OCaml [Kosarev, Boulytchev, ML-2016]

```
let rec append^o x y xy = conde [
  ((x ≡ nil ()) &&& (y ≡ xy));
  call_fresh (fun h →
    call_fresh (fun t →
      call_fresh (fun ty →
        (x ≡ h % t)  &&&
        (h % ty ≡ xy) &&&
        (append^o t y ty’))))
]
```

```
let rec append^o x y xy = ocanren {
  x == [] & y == xy |
  fresh h, t, ty in
    x == h :: t &
    z == h :: ty &
    append^o t y ty
}
```

OCanren: a Strongly Typed miniKanren for OCaml

Typed Embedding of a Relational Language in OCaml [Kosarev, Boulytchev, ML-2016]

```
let rec append^o x y xy = conde [
  ((x ≡ nil ()) &&& (y ≡xy));
  call_fresh (fun h →
    call_fresh (fun t →
      call_fresh (fun ty →
        (x ≡ h % t)  &&&
        (h % ty ≡ xy) &&&
        (append^o t y ty'))))
]
```

```
let rec append^o x y xy = ocanren {
  x == [] & y == xy |
  fresh h, t, ty in
    x == h :: t &
    z == h :: ty &
    append^o t y ty
}
```

- Disequality constraints;
- Tabling;
- Wildcard variables;
- ...

OCanren: Typing

Types in the functional world:

```
type α list = [] | α :: α list
```

OCanren: Typing

Types in the functional world:

```
type α list = [] | α :: α list
```

The relational world case:

```
fresh h, t, ty in
  x == h :: t &
  z == h :: ty
```

OCanren: Typing

Types in the functional world:

```
type α list = [] | α :: α list
```

The relational world case:

```
fresh h, t, ty in
  x == h :: t &
  z == h :: ty
```

Types in the relational world:

- a *logic* list can be either a free variable
- ... or `[]`
- ... or `h :: t`, where
  - `h` is a logic element of list
  - `t` is a *logic* list

Relational types cannot be acquired by some parameterization of functional ones.

OCanren: Abstraction, Lifting, Injection

Type abstraction:

```
type (α, β) alist = [] | α :: β
type α list = (α, α list) alist
```

OCanren: Abstraction, Lifting, Injection

Type abstraction:

```
type (α, β) alist = [] | α :: β
type α list = (α, α list) alist
```

Logic type:

```
type α logic = Var of var | Val of α
```

OCanren: Abstraction, Lifting, Injection

Type abstraction:

```
type (α, β) alist = [] | α :: β
type α list = (α, α list) alist
```

Logic type:

```
type α logic = Var of var | Val of α
```

Logic lists:

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

OCanren: Abstraction, Lifting, Injection

Type abstraction:

```
type (α, β) alist = [] | α :: β
type α list = (α, α list) alist
```

Logic type:

```
type α logic = Var of var | Val of α
```

Logic lists:

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

Transformation α `list` → α `llist`:

- on *type* level: *lifting*
- on *value* level: *injection*

OCanren: Reification, Example

```
open OCanren
open GT

ocanren type nat = O | S of nat

let rec addo x y z = ocanren {
  x == O & y == z |
  fresh x', z' in
    x == S x' &
    z == S z' &
    addo x' y z'
}

let _ =
  Stream.iter (fun q → Printf.printf "q=%s\n" @@ GT.show(nat) q) @@
  run q
    (fun q → ocanren {addo (S O) (S (S O)) q})
    (fun q → q#reify nat_prj_exn)
```

Generic Programming with Combinators and Objects [Kosarev, Boulytchev, ML-2021]

Typed Relational Conversion: Motivation

Relational programming is hard and error-prone

Typed Relational Conversion: Motivation

Relational programming is hard and error-prone

```
let rec append x y =
  match x of
    []     → y
  | h :: t → h :: (append t y)
```

Typed Relational Conversion: Motivation

Relational programming is hard and error-prone

```
let rec append x y =
  match x of
    []      → y
  | h :: t  → h :: (append t y)
```

```
let rec append^o x y xy = ocanren {
  x == [] & y == xy |
  fresh h, t, ty in
    x == h :: t &
    z == h :: ty &
    append^o t y ty
}
```

Typed Relational Conversion: Motivation

Relational programming is hard and error-prone

```
let rec append x y =
  match x of
    []      → y
  | h :: t  → h :: (append t y)
```

```
let rec appendᵒ x y xy = ocanren {
  x == [] & y == xy |
  fresh h, t, ty in
    x == h :: t &
    z == h :: ty &
    appendᵒ t y ty
}
```

Unnesting:

```
let rec append x y =
  match x of
    []      → y
  | h :: t  →
      let ty = append t y in
      h :: ty
```

Unnesting: Higher-Order Case

```
let bar x =
  let f x = x in
  let g a = f in
  g A x
```

Unnesting: Higher-Order Case

```
let bar x =
  let f x = x in
  let g a = f in
  g A x
```

```
let barᵒ x r = ocanren {
  let f x r = x == r in
  let g a r = f == r in
  g A x r
}
```

Typed Relational Conversion: Idea

On the type level:

$\mathfrak{G}$ — the type of goals

$$\begin{array}{rcl} [\![a]\!] & \mapsto & a \to \mathfrak{G} \\ [\![t_1 \to t_2]\!] & \mapsto & [\![t_1]\!] \to [\![t_2]\!] \end{array}$$

Typed Relational Conversion: Idea

On the type level:

$$\mathfrak{G} \text{ — the type of goals}$$

$$\begin{array}{rcl} [\![a]\!] & \mapsto & a \to \mathfrak{G} \\ [\![t_1 \to t_2]\!] & \mapsto & [\![t_1]\!] \to [\![t_2]\!] \end{array}$$

On the term level:

- pure λ-calculus is left intact!
- pattern-matching goes to disjunction (too long to present);
- constructor:

$$[\![C\,(e_1, \ldots, e_k)]\!] \mapsto \begin{array}{l} \mathbf{fun}\, q \to \\ \quad \mathbf{fresh}\; q_1, \ldots, q_k\; \mathbf{in} \\ \quad\quad [\![e_1]\!]\, q_1 \wedge \\ \quad\quad \ldots \\ \quad\quad [\![e_k]\!]\, q_k \wedge \\ \quad\quad q \equiv \uparrow C\,(q_1, \ldots, q_k) \end{array}$$

Typed Relational Conversion: noCanren

```
let rec addo x y =
  match x with
    O    → y
  | S x' → S (addo x' y)
```

Typed Relational Conversion: noCanren

```
let rec addo x y =
  match x with
    O    → y
  | S x' → S (addo x' y)
```

```
let rec addo x y q5 = ocanren {
  fresh q1 in
    x q1 &
    (q1 == O & y q5 |
     fresh x', q2 in
       q1 == S x' &
       q5 == S q2 &
       addo ((==) x') y q2
    )
}
```

Typed Relational Conversion: NOCANREN

```
let rec addo x y =
  match x with
    O     → y
  | S x'  → S (addo x' y)
```

```
let rec addo x y q5 = ocanren {
  fresh q1 in
    x q1 &
    (q1 == O & y q5 |
     fresh x', q2 in
       q1 == S x' &
       q5 == S q2 &
       addo ((==) x') y q2
    )
}
```

- + administrative reductions;
- + “best practices” for relational programming.

Typed Relational Conversion [Lozov, Vyatkin, Boulytchev, 2017]

Interpreters and Relational Interpreters

Conventional interpreter:

$$eval\ p\ x \mapsto y \;\Leftrightarrow\; p\ x \mapsto y$$

Interpreters and Relational Interpreters

Conventional interpreter:

$$\mathit{eval}\ p\ x \mapsto y \ \Leftrightarrow\ p\ x \mapsto y$$

Relational interpreter:

$$\mathit{eval}^o\ p\ x\ y \mapsto \{\theta_i\} \text{ such that } \forall i\ p\theta_i\ x\theta_i \mapsto y\theta_i$$

Interpreters and Relational Interpreters

Conventional interpreter:

$$eval\ p\ x \mapsto y \ \Leftrightarrow\ p\ x \mapsto y$$

Relational interpreter:

$$eval^{o}\ p\ x\ y \mapsto \{\theta_i\} \text{ such that } \forall i\ p\theta_i\ x\theta_i\ \mapsto y\theta_i$$

Benefits: with a relational interpreter for a certain language all programs in this language can be executed relationally

A Unified Approach to Solving Seven Programming Problems (Functional Pearl) [Byrd, Ballantyne, Rosenblatt, Might, ICFP-2017]

Search Problems, Verifiers, Solvers

Search problem: given a combinatorial object Ω find some object s satisfying property $\mathcal{P}$.

Search Problems, Verifiers, Solvers

Search problem: given a combinatorial object Ω find some object s satisfying property $\mathcal{P}$.

Examples: Hamiltonian path, SAT, etc.

Search Problems, Verifiers, Solvers

Search problem: given a combinatorial object Ω find some object s satisfying property $\mathcal{P}$.

Examples: Hamiltonian path, SAT, etc.

Verifier:

$$\mathit{verify}\ \Omega\ s = \begin{cases} \textbf{true} & , \quad s \text{ is a solution} \\ \textbf{false} & , \quad \text{otherwise} \end{cases}$$

Search Problems, Verifiers, Solvers

Search problem: given a combinatorial object Ω find some object s satisfying property $\mathcal{P}$.

Examples: Hamiltonian path, SAT, etc.

Verifier:

$$\mathit{verify}\ \Omega\ s = \begin{cases} \mathbf{true} & , \quad s \text{ is a solution} \\ \mathbf{false} & , \quad \text{otherwise} \end{cases}$$

Solver:

$$\mathit{verify}^o\ \Omega\ \sigma\ \mathbf{true}$$

Search Problems, Verifiers, Solvers

Search problem: given a combinatorial object Ω find some object s satisfying property $\mathcal{P}$.

Examples: Hamiltonian path, SAT, etc.

Verifier:

$$\mathit{verify}\ \Omega\ s = \begin{cases} \textbf{true} & , \quad s \text{ is a solution} \\ \textbf{false} & , \quad \text{otherwise} \end{cases}$$

Solver:

$$\mathit{verify}^o\ \Omega\ \sigma\ \textbf{true}$$

Relational Interpreters for Search Problems [Verbitskaya, Berezun, Lozov, Boulytchev, MK-2019]

Water Pouring Puzzle: Description

Water pouring puzzle

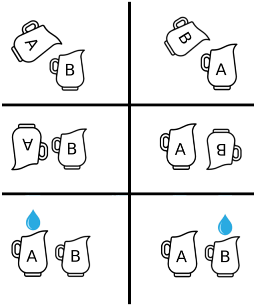
From Wikipedia, the free encyclopedia

> This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
> *Find sources:* "Water pouring puzzle" – news · newspapers · books · scholar · JSTOR *(July 2017) (Learn how and when to remove this template message)*

Water pouring puzzles (also called **water jug problems**, **decanting problems**,[1][2] **measuring puzzles**, or **Die Hard with a Vengeance puzzles**) are a class of puzzle involving a finite collection of water jugs of known integer capacities (in terms of a liquid measure such as liters or gallons). Initially each jug contains a known integer volume of liquid, not necessarily equal to its capacity.

Puzzles of this type ask how many steps of pouring water from one jug to another (until either one jug becomes empty or the other becomes full) are needed to reach a goal state, specified in terms of the volume of liquid that must be present in some jug or jugs.[3]

Starting state of the standard puzzle



Water Pouring Puzzle: Functional Verifier

Water Pouring Puzzle: Functional Verifier

$$\mathcal{M} = \{A \rightarrow B, B \rightarrow A, \uparrow A, \uparrow B, \downarrow A, \downarrow B\}$$

Water Pouring Puzzle: Functional Verifier

$$\mathcal{M} = \{A \rightarrow B, B \rightarrow A, \uparrow A, \uparrow B, \downarrow A, \downarrow B\}$$
$$\mathcal{C} = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

Water Pouring Puzzle: Functional Verifier

$$\mathcal{M} = \{A \to B, B \to A, \uparrow A, \uparrow B, \downarrow A, \downarrow B\}$$

$$\mathcal{C} = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

$$\mathcal{C} \xrightarrow{\mathcal{M}^*} \mathcal{C}$$

Water Pouring Puzzle: Functional Verifier

$$\mathcal{M} = \{A \to B, B \to A, \uparrow A, \uparrow B, \downarrow A, \downarrow B\}$$

$$\mathcal{C} = \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

$$\mathcal{C} \xrightarrow{\mathcal{M}^*} \mathcal{C}$$

```
type move = AB | BA | FillA | FillB | EmptyA | EmptyB

let step (capA, capB, a, b) = function
| FillA  → (capA, capB, capA, b)
| FillB  → (capA, capB, a, capB)
| EmptyA → (capA, capB, 0, b)
| EmptyB → (capA, capB, a, 0)
| AB     → let diff = capB − b in
           (capA, capB, a − diff, b + (min diff a))
| BA     → let diff = capA − a in
           (capA, capB, a + (min diff b), b − diff)

let eval = fold step
```

Water Pouring Puzzle: Relational Solver

$$
\begin{array}{lll}
\textbf{run}_\mu & \{ & \texttt{\textbf{fresh}}\ \texttt{a},\ \texttt{b}\ \texttt{\textbf{in}} \\
 & & \texttt{eval}^{o}(5,\ 3,\ 0,\ 0)\ \mu\ (\_,\ \_,\ \texttt{a},\ \texttt{b})\ \& \\
 & & (\texttt{a} == 1 \mid \texttt{b} == 1) \\
 & \} & \mapsto [\mu \mapsto (\text{FillB, BA, FillB, BA})], \ldots
\end{array}
$$

Water Pouring Puzzle: Relational Solver

$$
\mathbf{run}_\mu \quad \left\{ \begin{array}{l} \texttt{\textbf{fresh}}\ \texttt{a},\ \texttt{b}\ \texttt{\textbf{in}} \\ \texttt{eval}^{o}(5,\ 3,\ 0,\ 0)\ \mu\ (\_,\ \_,\ \texttt{a},\ \texttt{b})\ \& \\ (\texttt{a}\ ==\ 1 \mid \texttt{b} == 1) \end{array} \right\} \mapsto [\mu \mapsto (\text{FillB, BA, FillB, BA})], \ldots
$$

- Other puzzles: Einstein problem, Jeep problem, Bridge & Torch, Hanoi towers, etc.
- Unification (verifier $\rightarrow$ unifier).
- Type checker $\rightarrow$ type inferencer (STLC).

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

Constructors, values, and patterns:

$$
\begin{array}{rcl}
\mathcal{C} & = & \{C_1^{k_1}, \dots, C_n^{k_n}\} \\
\mathcal{V} & = & \mathcal{C}\,\mathcal{V}^* \\
\mathcal{P} & = & \_ \mid \mathcal{C}\,\mathcal{P}^*
\end{array}
$$

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

Constructors, values, and patterns:

$$\begin{array}{rcl} \mathcal{C} & = & \{C_1^{k_1}, \dots, C_n^{k_n}\} \\ \mathcal{V} & = & \mathcal{C}\,\mathcal{V}^* \\ \mathcal{P} & = & \_ \mid \mathcal{C}\,\mathcal{P}^* \end{array}$$

Declarative semantics of pattern-matching:

$$\langle v; p_1, \dots, p_k \rangle \longrightarrow i,\ 1 \le i \le k+1$$

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

"Switch" language:

$$
\begin{array}{rcl}
\mathcal{M} & = & \bullet \\
& | & \mathcal{M}\,[\mathbb{N}] \\
\mathcal{S} & = & \textbf{return}\,\mathbb{N} \\
& | & \textbf{switch}\ \mathcal{M}\ \textbf{with}\ [\mathcal{C} \rightarrow \mathcal{S}]^*\ \textbf{otherwise}\ \mathcal{S}
\end{array}
$$

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

"Switch" language:

$$
\begin{array}{rcl}
\mathcal{M} & = & \bullet \\
 & \mid & \mathcal{M}\,[\mathbb{N}] \\
\mathcal{S} & = & \textbf{return}\,\mathbb{N} \\
 & \mid & \textbf{switch}\ \mathcal{M}\ \textbf{with}\ [\mathcal{C} \rightarrow \mathcal{S}]^*\ \textbf{otherwise}\ \mathcal{S}
\end{array}
$$

The semantics of switch language:

$$v \vdash \pi \Longrightarrow_{\mathcal{S}} i$$

Application: Pattern-Matching Synthesis

Relational Synthesis for Pattern Matching [Kosarev, Boulytchev, APLAS-2020]

"Switch" language:

$$
\begin{array}{rcl}
\mathcal{M} & = & \bullet \\
 & | & \mathcal{M}\,[\mathbb{N}] \\
\mathcal{S} & = & \textbf{return}\,\mathbb{N} \\
 & | & \textbf{switch}\ \mathcal{M}\ \textbf{with}\ [\mathcal{C} \to \mathcal{S}]^*\ \textbf{otherwise}\ \mathcal{S}
\end{array}
$$

The semantics of switch language:

$$v \vdash \pi \Longrightarrow_{\mathcal{S}} i$$

Pattern-matching synthesis problem: for a given ordered sequence of patterns $p_1, \ldots, p_k$ find a switch program π, such that

$$\forall v \in \mathcal{V},\ \forall 1 \le i \le n+1 : \langle v; p_1, \ldots, p_n \rangle \longrightarrow i \Longleftrightarrow v \vdash \pi \Longrightarrow_{\mathcal{S}} i$$

Application: Declarative UI Synthesis

On a Declarative Guideline-Directed UI Layout Synthesis [Kosarev, Lozov, Fokin, Boulytchev, miniKanren-2022]

Application: Declarative UI Synthesis

On a Declarative Guideline-Directed UI Layout Synthesis [Kosarev, Lozov, Fokin, Boulytchev, miniKanren-2022]

- Structure: a set of UI controls and relations between them.

Application: Declarative UI Synthesis

On a Declarative Guideline-Directed UI Layout Synthesis [Kosarev, Lozov, Fokin, Boulytchev, miniKanren-2022]

- Structure: a set of UI controls and relations between them.
- Layout: a set of primitives describing the placements of UI controls.

Application: Declarative UI Synthesis

On a Declarative Guideline-Directed UI Layout Synthesis [Kosarev, Lozov, Fokin, Boulytchev, miniKanren-2022]

- Structure: a set of UI controls and relations between them.
- Layout: a set of primitives describing the placements of UI controls.
- Guideline: a set of rules mapping structure elements to layout primitives.

Application: Declarative UI Synthesis

On a Declarative Guideline-Directed UI Layout Synthesis [Kosarev, Lozov, Fokin, Boulytchev, miniKanren-2022]

- Structure: a set of UI controls and relations between them.
- Layout: a set of primitives describing the placements of UI controls.
- Guideline: a set of rules mapping structure elements to layout primitives.
- Layout synthesis problem: for a given structure find all maximal sets of non-contradictory layout primitive instances prescribed by a guideline.

Thank you!

Links:

- `http://minikanren.org/` — the main MINIKANREN site;
- `https://github.com/PLTools/OCanren` — OCANREN implementation;
- `https://github.com/PLTools/noCanren` — NOCANREN implementation.

Future research:

- Performance improvements.
- Heuristic search.
- Extensions.
- More applications.

dboulytchev@gmail.com