



Relational Interpreters for Search Problems

Petr Lozov, **Kate Verbitskaia**, Dmitry Boulytchev

JetBrains Research, Programming Languages and Tools Lab
Saint Petersburg State University

22.08.2019

Recognition vs. Search

X — alphabet

$$L \subseteq X^*$$

if $\omega \in L$, denote the *witness* of this fact p_ω

Recognition vs. Search

X — alphabet

$$L \subseteq X^*$$

if $\omega \in L$, denote the *witness* of this fact p_ω

$$\text{Recognition: } V(\omega, p_\omega) = \begin{cases} 1, & \omega \in L \\ 0, & \omega \notin L \end{cases}$$

Recognition vs. Search

X — alphabet

$$L \subseteq X^*$$

if $\omega \in L$, denote the *witness* of this fact p_ω

$$\text{Recognition: } V(\omega, p_\omega) = \begin{cases} 1, & \omega \in L \\ 0, & \omega \notin L \end{cases}$$

$$\text{Search: } S(\omega) = p_\omega$$

Propositional Formulas: Recognition

```
let rec eval st = function
| Conj (l, r) → eval st l && eval st r
| Disj (l, r) → eval st l || eval st r
| Neg  e      → not (eval st e)
| Var  x      → List.assoc x st
```

Propositional Formulas: Recognition

```
let rec eval st = function
| Conj (l, r) → eval st l && eval st r
| Disj (l, r) → eval st l || eval st r
| Neg  e      → not (eval st e)
| Var  x      → List.assoc x st
```

```
# eval [( 'x,true);( 'y,false)] (Conj (Var 'x) (Neg (Var 'y))));;
```

```
— : bool = true
```

Propositional Formulas: Search

```
let rec solve st b = function
| Var n → ( match assoc_opt n st with
            | None → [extend st n b]
            | Some b' when b == b' → [st]
            | _ → [])
| Conj (l, r) when b →
    concat @@
    map (λ st → solve st b r) @@
    solve st b l
| Conj (l, r) → solve st b l @ solve st b r
| Neg e → solve st (not b) e
| Disj (l, r) → solve st b (Neg (Conj (Neg l, Neg r)))
```

Search is Hard¹

Is it possible to generate a search procedure from a recognizer?

¹compared to recognition

$$V^R(\omega, p_\omega, q)$$

$$V^R(\omega, p_\omega, 1), \quad \text{if } \omega \in L, p_\omega \text{ — a witness}$$

$$V^R(\omega, p_\omega, 0), \quad \text{otherwise}$$

Relational Interpretation for Recognition and Search

$$V^R(\omega, p_\omega, ?) \rightsquigarrow V(\omega, p_\omega)$$

$$V^R(\omega, ?, 1) \rightsquigarrow S(\omega)$$

Only one program to implement!

Propositional Formulas: Relational Interpreter

```
let rec evalo st fm u = ocanren (  
  fresh x, y, z, v, w in  
    fm == conj x y & evalo st x v & evalo st y w & ando v w u |  
    fm == disj x y & evalo st x v & evalo st y w & oro v w u |  
    fm == neg x & evalo st x v & noto v u |  
    fm == var z & assoco z st u  
)
```

Relational Programming is Hard²

```
let rec hanoio a b c moves a' b' c' = ocanren (
  moves == [] & a == a' & b == b' & c == c' |
  fresh f, t, moves', pin_f, pin_t, pin_f_res, pin_t_res, a'', b'', c'' in
    moves == (f, t) :: moves' &
    (f == A & t == B & pin_f == a & pin_f_res == a'' & pin_t == b & pin_t_res == b'' & c'' == c |
     f == A & t == C & pin_f == a & pin_f_res == a'' & pin_t == c & pin_t_res == c'' & b'' == b |
     f == B & t == A & pin_f == b & pin_f_res == b'' & pin_t == a & pin_t_res == a'' & c'' == c |
     f == B & t == C & pin_f == b & pin_f_res == b'' & pin_t == c & pin_t_res == c'' & a'' == a |
     f == C & t == A & pin_f == c & pin_f_res == c'' & pin_t == a & pin_t_res == a'' & b'' == b |
     f == C & t == B & pin_f == c & pin_f_res == c'' & pin_t == b & pin_t_res == b'' & a'' == a) &
  fresh top_f, rest_f in
    pin_f == top_f :: rest_f &
    (pin_t == [] |
     fresh top_t, rest_t in
       pin_t == top_t :: rest_t & lto top_f top_t true
    ) &
    pin_f_res == rest_f &
    pin_t_res == top_f :: pin_t &
    hanoio a'' b'' c'' moves' a' b' c'
)
```

²compared to functional programming

Relational Programming is Hard²

```
let rec hanoio a b c moves a' b' c' = ocanren (
  moves == [] & a == a' & b == b' & c == c' |
  fresh f, t, moves', pin_f, pin_t, pin_f_res, pin_t_res, a'', b'', c'' in
    moves == (f, t) :: moves' &
    (f == A & t == B & pin_f == a & pin_f_res == a'' & pin_t == b & pin_t_res == b'' & c'' == c |
     f == A & t == C & pin_f == a & pin_f_res == a'' & pin_t == c & pin_t_res == c'' & b'' == b |
     f == B & t == A & pin_f == b & pin_f_res == b'' & pin_t == a & pin_t_res == a'' & c'' == c |
     f == B & t == C & pin_f == b & pin_f_res == b'' & pin_t == c & pin_t_res == c'' & a'' == a |
     f == C & t == A & pin_f == c & pin_f_res == c'' & pin_t == a & pin_t_res == a'' & b'' == b |
     f == C & t == B & pin_f == c & pin_f_res == c'' & pin_t == b & pin_t_res == b'' & a'' == a) &
  fresh top_f, rest_f in
    pin_f == top_f :: rest_f &
    (pin_t == [] |
     fresh top_t, rest_t in
       pin_t == top_t :: rest_t & lto top_f top_t true
    ) &
    pin_f_res == rest_f &
    pin_t_res == top_f :: pin_t &
    hanoio a'' b'' c'' moves' a' b' c'
)
```

It took 3 people 6 hours to implement

²compared to functional programming

Ways to Create Relational Interpreters

- Manual implementation
- Interpretation of functional programs with a relational interpreter
- Relational conversion

Ways to Create Relational Interpreters

- Manual implementation
- **Interpretation of functional programs with a relational interpreter**
- Relational conversion

Relational Interpretation of Functional Programs

- Implement good relational interpreter of a Turing-complete language
- Implement functional recognizer
- Run functional recognizer with a relational interpreter

Running relational interpreter comes with a price

Are there ways to get rid of it?

Interpreter:

```
eval prog input == output
```

Interpreter:

```
eval prog input == output
```

Consider that a part of the input is known: `input == (static, dynamic)`

Interpreter:

$$\text{eval prog input} == \text{output}$$

Consider that a part of the input is known: $\text{input} == (\text{static}, \text{dynamic})$

Specializer:

$$\begin{aligned} & \text{spec prog static} \Rightarrow \text{prog}_{\text{spec}} \\ \text{eval prog (static, dynamic)} & == \text{eval prog}_{\text{spec}} \text{ dynamic} \end{aligned}$$

- Specializers can fail to remove all interpretation overhead
- Jones-optimal specializer: the specialized interpreter is as efficient as the program it was specialized for
- There exists a Jones-optimal specializer for a logical language [Leuschel, 2004]
- Not for miniKanren
- Jones-optimality is hard to achieve

Ways to Create Relational Interpreters

- Manual implementation
- Interpretation of functional programs with a relational interpreter
- **Relational conversion**

Relational Conversion for Relational Interpreter

- Implement a functional recognizer (verifier): $V(\omega, p_\omega)$
- Transform it into a relation: $V^R(\omega, p_\omega, q)$
- Specialize
 - Redundancy introduced with the relational conversion
 - Direction ($q == 1$)
 - Known data (ω)
- The result is a search routine

Relational Conversion (Unnesting) [Byrd 2009]

- Introduce a new variable for each subexpression
- For every n -ary function create an $(n+1)$ -ary relation, where the last argument is unified with the result
- Transform **if**-expressions and pattern matchings into disjunctions with unifications for patterns
- Introduce into scope free variables (with **fresh**)
- Pop unifications to the top

Relational Conversion: Step 1

Introduce a new variable for each subexpression

```
let rec append a b =  
  match a with  
  | []          → b  
  | x :: xs →  
    x :: append xs b
```

```
let rec append a b =  
  match a with  
  | []          → b  
  | x :: xs →  
    let q = append xs b in  
    x :: q
```

Relational Conversion: Step 2

Introduce a new argument for result

`let rec append a b = ...` `let rec appendo a b c = ...`

Relational Conversion: Step 3

Transform **if**-expressions and pattern matchings into disjunctions with unifications for patterns

```
let rec append a b =  
  match a with  
  | []          → b  
  | x :: xs     →  
    let q = append xs b in  
    x :: q
```

```
let rec appendo a b c =  
  ocanren (  
    (a == [] & b == c) |  
    ( a == x :: xs &  
      appendo xs b q &  
      c == x :: q))
```

Relational Conversion: Step 4

Introduce free variables into scope (with **fresh**)

```
let rec appendo a b c =  
  ocanren (  
    (a == [] & b == c) |  
    ( (a == x :: xs) &  
      (appendo xs b q) &  
      (c == x :: q)))
```

```
let rec appendo a b c =  
  ocanren (  
    (a == [] & b == c) |  
    (fresh x, xs, q in  
      a == x :: xs &  
      appendo xs b q &  
      c == x :: q))
```

Relational Conversion: Step 5

Pop unifications to the top

```
let rec appendo a b c =  
  ocanren (  
    (a == [] & b == c) |  
    (fresh x, xs, q in  
      a == x :: xs &  
      appendo xs b q &  
      c == x :: q))
```

```
let rec appendo a b c =  
  ocanren (  
    (a == [] & b == c) |  
    (fresh x, xs, q in  
      a == x :: xs &  
      c == x :: q &  
      appendo xs b q))
```

Forward Execution is Efficient, Backward Execution is not

Forward execution is efficient, since it mimics the execution of a function

Relational conversion for $f_1\ x_1 \ \&\& \ f_2\ x_2$:

```
 $\lambda\ res \rightarrow \text{ocanren } ($   
  fresh  $p\ \text{in}$   
     $f_1\ x_1\ p \ \&$   
     $(\ p = \text{false} \ \& \ res = \text{false} \mid$   
       $p = \text{true} \ \& \ f_2\ x_2\ res))$ 
```

Computes $f_2\ x_2\ res$ only if $f_1\ x_1\ p$ fails

It is not the best strategy, if res is known

Relational Conversion Aimed at Backward Execution

This conversion of $f_1\ x_1 \ \&\& \ f_2\ x_2$ is better for the backward execution, but not for forward

```
λ res → ocanren (  
  res == false & f1 x1 false |  
  f1 x1 true & f2 x2 res)
```

There is no single strategy suitable for all cases

There is no Single Good Strategy

Is there a way to automatically generate relations efficient in the specified directions?

Specialization: Not Only for Direction

When solving a search problem, we know its search space

$$V^R(\omega, ?, 1) \rightsquigarrow S(\omega)$$

Partial Deduction: Specialization for Logic Language

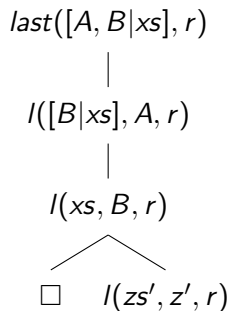
- Given:
 - Logic program
 - Goal
- Result: specialized program
- How:
 - Construct a *partial* SLD-tree
 - Generate a program from the tree
- Hopefully, all excessive computations are done statically and do not come to the specialized program

Partial Deduction: Example

```
last([x|xs], r) ← l(xs, x, r).  
l([], x, x).  
l([z|zs], x, r) ← l(zs, z, r).  
  
← last([A,B|xs], r).
```

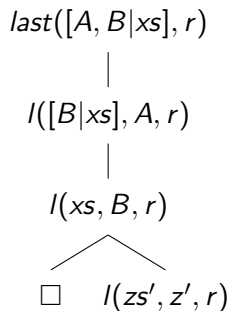
Partial Deduction: Example

```
last([x|xs], r) ← l(xs, x, r).  
l([], x, x).  
l([z|zs], x, r) ← l(zs, z, r).  
  
← last([A,B|xs], r).
```



Partial Deduction: Example

$\text{last}([x|xs], r) \leftarrow \text{l}(xs, x, r).$
 $\text{l}([], x, x).$
 $\text{l}([z|zs], x, r) \leftarrow \text{l}(zs, z, r).$
 $\leftarrow \text{last}([A,B|xs], r).$



$\text{last}([A,B], B).$
 $\text{last}([A,B,z'|zs'], r) \leftarrow \text{l}(zs', z', r).$
 $\text{l}([], x, x).$
 $\text{l}([z|zs], x, r) \leftarrow \text{l}(zs, z, r).$

Partial Deduction: Conjunctions

$$\begin{array}{c} f(x, y) \vee f(y, z) \\ \swarrow \quad \searrow \\ f(x, y) \quad f(y, z) \end{array}$$



Partial Deduction: Conjunctions

$$\begin{array}{c} f(x, y) \vee f(y, z) \\ \swarrow \quad \searrow \\ f(x, y) \quad f(y, z) \end{array}$$



$$\begin{array}{c} f(x, y) \wedge f(y, z) \\ \swarrow \quad \searrow \\ f(x, y) \quad f(y, z) \end{array}$$



Conjunctive Partial Deduction

- Fully automatic program transformation
- For pure logic language
- Features:
 - Specialization
 - Deforestation
 - Tupling

Deforestation

Deforestation — program transformation which eliminates intermediate data structures

```
let double_appendo x y z xyz =  
  ocanren (  
    fresh t in  
      appendo x y t &  
      appendo t z xyz )
```

```
let rec appendo x y xy =  
  ocanren (  
    x == [] & xy == y |  
    fresh h, t, ty in  
      x == h :: t &  
      xy == h :: t' &  
      appendo t y t')
```

```
let rec double_appendo x y z xyz =  
  ocanren (  
    x == [] & appendo y z xyz |  
    (fresh h, t, t' in  
      x == h :: t &  
      xyz == h :: t' &  
      double_appendo t y z t')
```

Tupling

Tupling — program transformation which eliminates multiple traversals of the same data structure

```
let max_lengtho xs m l = ocanren (maxo xs m & lengtho xs l)
```

```
let rec lengtho xs l = ocanren (  
  xs == [] & l == 0 |  
  (fresh h, t, m in  
    xs == h :: t & l == succ m & lengtho t m))
```

```
let maxo xs m = max1o xs 0 m
```

```
let rec max1o xs n m = ocanren (  
  xs == [] & m == n |  
  (fresh h, t in  
    xs == h :: t &  
    (leo h n true & max1o t n m |  
     gto h n true & max1o t h m))))
```

Tupling

Tupling — program transformation which eliminates multiple traversals of the same data structure

```
let max_lengtho xs m l = ocanren (max_length1o xs m 0 l)
```

```
let rec max_length1o xs m n l = ocanren (  
  xs == [] & m == n & l == 0 |  
  (fresh h, t, l1 in  
    xs == h :: t &  
    l == succ l1 &  
    ( leo h n & max_length1o t m n l |  
      gto h n & max_length1o t m h l)))
```

- Local control: compute a partial SLD-tree per a relation of interest
 - Having a conjunction of atoms, which atom should be selected?
 - When to stop building a tree?
- Global control: determine which relations are of interest
 - Do not process the same conjunction twice
 - If a conjunction *embeds* something processed before, *generalize* it
 - How to define *embedding*?
 - How to *generalize*?

- Local control
 - Deterministic unfold (only one nondeterministic unfold per tree)
 - Selectable conjunct: leftmost call which does not have any predecessor embedded into it
 - Variant check
 - Stop when there are no selectable conjuncts
- Global control
 - Variant check
 - Generalization: split conjunction in maximally connected subconjunctions + most specific generalization
 - Homeomorphic embedding extended for conjunctions
- Residualization
 - A definition per a local tree
 - Redundant Argument Filtering

Compare

- Unnesting
- Unnesting strategy aimed at backward execution
- Unnesting + CPD
- Interpretation of functional verifier with relational interpreter

Tasks

- Path search
- Search for a unifier of two terms

Directed graph is a tuple $(N, E, start, end)$, where:

- N — set of nodes
- E — set of edges
- Functions $start, end : E \rightarrow N$ return a start (end) node of an edge

Path Search

Directed graph is a tuple $(N, E, start, end)$, where:

- N — set of nodes
- E — set of edges
- Functions $start, end : E \rightarrow N$ return a start (end) node of an edge

Path is a sequence $\langle n_0, e_0, n_1, e_1, \dots, n_k, e_k, n_{k+1} \rangle$, such that

$$\forall i \in \{0 \dots k\} : n_i = start(e_i) \text{ and } n_{i+1} = end(e_i)$$

Path Search

Directed graph is a tuple $(N, E, start, end)$, where:

- N — set of nodes
- E — set of edges
- Functions $start, end : E \rightarrow N$ return a start (end) node of an edge

Path is a sequence $\langle n_0, e_0, n_1, e_1, \dots, n_k, e_k, n_{k+1} \rangle$, such that

$$\forall i \in \{0 \dots k\} : n_i = start(e_i) \text{ and } n_{i+1} = end(e_i)$$

Path search problem is to find the set of paths in a given graph

Path Search: Relational Conversion

```
let rec is_path ns g =  
  match ns with  
  | x1 :: x2 :: xs → elem (x1, x2) g && is_path (x2 :: xs) g  
  | [-]           → true
```

Path Search: Relational Conversion

```
let rec is_path ns g =  
  match ns with  
  | x1 :: x2 :: xs → elem (x1, x2) g && is_path (x2 :: xs) g  
  | [-]           → true  
  
let rec is_path° ns g res = ocanren (  
  fresh el in (ns == [el] & res == true) |  
  (fresh x1, x2, xs, res_elem, res_is_path in  
    ns == x1 :: (x2 :: xs) &  
    elem° (x1, x2) g res_elem &  
    is_path° (x2 :: xs) g res_is_path &  
    ( res_elem == false & res == false |  
      res_elem == true  & res == res_is_path )))
```

This relation is inefficient for “is_path° q <graph> true”

Path Search: Specialized Relation

```
let rec is_patho ns g res = ocanren (  
  fresh e1 in (ns == [e1] & res == true) |  
  (fresh x1, x2, xs, res_elem, res_is_path in  
    res_elem == true &  
    res_is_path == true &  
    ns == x1 :: (x2 :: xs) &  
    elemo (x1, x2) g res_elem &  
    is_patho (x2 :: xs) g res_is_path)))
```

Better performance for “is_path^o q <graph> true”

Path Search: Specialized Relation

```
let rec is_patho ns g res = ocanren (  
  fresh e1 in (ns == [e1] & res == true) |  
  (fresh x1, x2, xs, res_elem, res_is_path in  
    res_elem == true &  
    res_is_path == true &  
    ns == x1 :: (x2 :: xs) &  
    elemo (x1, x2) g res_elem &  
    is_patho (x2 :: xs) g res_is_path)))
```

Better performance for “is_path^o q <graph> true”

This can be achieved automatically with CPD

Evaluation: Path Search

Path length	5	7	9	11	13	15
Only conversion	0.01	1.39	82.13	>300	—	—
Backward oriented conversion	0.01	0.37	2.68	2.91	4.88	10.63
Conversion and CPD	0.01	0.06	0.34	2.66	3.65	6.22
Scheme interpreter	0.80	8.22	88.14	191.44	>300	—

Table: Searching for paths in the graph (seconds)

Term:

- Variable (X, Y, \dots)
- Some constructor applied to terms ($nil, cons(H, T), \dots$)

Term:

- Variable (X, Y, \dots)
- Some constructor applied to terms ($nil, cons(H, T), \dots$)

Substitution maps variables to terms

Substitution can be *applied* to a term by simultaneously substituting variables for their images

Term:

- Variable (X, Y, \dots)
- Some constructor applied to terms ($nil, cons(H, T), \dots$)

Substitution maps variables to terms

Substitution can be *applied* to a term by simultaneously substituting variables for their images

Unifier is a substitution σ which equalizes terms: $t\sigma = s\sigma$

Term:

- Variable (X, Y, \dots)
- Some constructor applied to terms ($nil, cons(H, T), \dots$)

Substitution maps variables to terms

Substitution can be *applied* to a term by simultaneously substituting variables for their images

Unifier is a substitution σ which equalizes terms: $t\sigma = s\sigma$

Problem: given two terms with free variables, find their unifier

Unification: Functional Verifier

```
let rec check_uni subst t1 t2 =  
  match t1, t2 with  
  | Constr (n1, a1), Constr (n2, a2) →  
    eq_nat n1 n2 && forall2 subst a1 a2  
  | Var_ v      , Constr (n, a)  →  
    begin match get_term v subst with  
    | None   → false  
    | Some t → check_uni subst t t2  
    end  
  | Constr (n, a) , Var_ v      →  
    begin match get_term v subst with  
    | None   → false  
    | Some t → check_uni subst t1 t  
    end  
  | Var_ v1      , Var_ v2      →  
    match get_term v1 subst with  
    | Some t1' → check_uni subst t1' t2  
    | None     → match get_term v2 subst with  
                  | Some _ → false  
                  | None   → eq_nat v1 v2
```

Unification: Relational Conversion

Does not fit the slide.

Evaluation: Unification

Terms	$f(X, a)$	$f(a \% b \% nil, c \% d \% nil, L)$	$f(X, X, g(Z, t))$
	$f(a, X)$	$f(X \% XS, YS, X \% ZS)$	$f(g(p, L), Y, Y)$
Only conversion	0.01	>300	>300
Backward oriented conversion	0.01	0.11	2.26
Conversion and CPD	0.01	0.07	0.90
Scheme interpreter	0.04	5.15	>300

Table: Searching for a unifier of two terms (seconds)

Functional recognizer + unnesting + specialization = search

Future work

- Generate functional program from relational to reduce interpretation overhead
- Some other specialization technique, less ad-hoc than CPD