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

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
```

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

- : bool = true
```

Propositional Formulas: Search

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

Search is Hard[1]

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

[1] compared to recognition

Relational Interpreter

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

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

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

Relational Interpretation for Recognition and Search

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

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

Only one program to implement!

Propositional Formulas: Relational Interpreter

```
let rec evalᵒ st f u =
  fresh (x y z v w) (
    conde [
      ?& [f ≡ conj x y; evalᵒ st x v; evalᵒ st y w; andᵒ v w u];
      ?& [f ≡ disj x y; evalᵒ st x v; evalᵒ st y w; orᵒ v w u];
      ?& [f ≡ neg  x  ; evalᵒ st x v; notᵒ v u];
      ?& [f ≡ var  z  ; assocᵒ z st u];
    ])
```

Relational Programming is Hard[2]

```
let eval_hanoi a b c moves a' b' c' =
  conde [
    ?& [moves ≡ nil (); a ≡ a'; b ≡ b'; c ≡ c';];
    fresh (f t moves' pin_f pin_t pin_f_res pin_t_res a'' b'' c'') (
      ?& [ moves ≡ (pair f t) % moves';
           conde [
             ?& [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) (
             ?& [
                  pin_f ≡ top_f % rest_f;
                  conde [ pin_t ≡ nil ();
                          fresh (top_t rest_t) (
                            ?& [pin_t ≡ top_t % rest_t;
                                lt^o top_f top_t tru^o;])];
                  pin_f_res ≡ rest_f;
                  pin_t_res ≡ top_f % pin_t;
                  eval_hanoi a'' b'' c'' moves' a' b' c';])])])
```

This took 3 people 6 hours to implement it

[2] compared to functional programming

Ways to Create Relational Interpreters

- Manual implementation
- Relational interpretation of functional programs
- Using relational conversion

Ways to Create Relational Interpreters

- Manual implementation
- **Relational interpretation of functional programs**
- Using 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

Interpretation Overhead

Running relational interpreter comes with a price

Are there ways to get rid of it?

Specialization

Interpreter:

`eval prog input == output`

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

Specializer:

$\texttt{spec prog static} \Rightarrow \texttt{prog}_{spec}$

$\texttt{eval prog (static, dynamic)} == \texttt{eval prog}_{spec}\ \texttt{dynamic}$

Jones-Optimality

- Specializers also introduce interpretation overhead
- Jones-optimal specializer: the specialized program is not slower than the interpretation
- 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
- Relational interpretation of functional programs
- **Using relational conversion**

Relational Conversion for Relational Interpreter

- Implement a functional recognizer (verifier)
- Transform it into a relation
- Specialize for the backward direction
- The result is a search routine

Relational Conversion [Byrd 2009]

Relational programming is complicated, why not let users write a verifier as a function and then translate it into miniKanren?

- 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 variable for each subexpression

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

```
let rec append^o 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 =
  (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 =
  (a ≡ [] ∧ b ≡ c) ∨
  (   (a ≡ x :: xs) ∧
      (appendᵒ xs b q) ∧
      (c ≡ x :: q))
```

```
let rec appendᵒ a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
      (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 =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (a ≡ x :: xs) ∧
     (appendᵒ xs b q) ∧
     (c ≡ x :: q)))
```

```
let rec appendᵒ a b c =
  (a ≡ [] ∧ b ≡ c) ∨
  (fresh (x xs q) (
     (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 $\texttt{f}_1\ \texttt{x}_1$ && $\texttt{f}_2\ \texttt{x}_2$:

$$
\begin{array}{l}
\lambda\ \texttt{res}\ \rightarrow \\
\quad \textbf{fresh}\ (\texttt{p})\ (\\
\qquad (\texttt{f}_1\ \texttt{x}_1\ \texttt{p})\ \wedge \\
\qquad (\textbf{conde}\ [\\
\qquad\quad (\texttt{p} \equiv \uparrow\textbf{false}\ \wedge\ \texttt{res} \equiv \uparrow\textbf{false}); \\
\qquad\quad (\texttt{p} \equiv \uparrow\textbf{true}\ \ \wedge\ \texttt{f}_2\ \texttt{x}_2\ \texttt{res})]))
\end{array}
$$

Computes $\texttt{f}_2\ \texttt{x}_2\ \texttt{res}$ only if $\texttt{f}_1\ \texttt{x}_1\ \texttt{p}$ fails

It is not the best strategy, if $\texttt{res}$ is known

Relational Conversion Aimed at Backward Execution

This coversion of $\texttt{f}_1\ \texttt{x}_1$ && $\texttt{f}_2\ \texttt{x}_2$ is better for backward execution, but not forward

$$
\begin{array}{l}
\lambda\ \texttt{res}\ \rightarrow \\
\quad \textbf{conde}\ [\\
\quad\quad (\texttt{res} \equiv \uparrow\textbf{false}\ \wedge\ \texttt{f}_1\ \texttt{x}_1\ \uparrow\textbf{false}); \\
\quad\quad (\texttt{f}_1\ \texttt{x}_1\ \uparrow\textbf{true}\ \ \wedge\ \texttt{f}_2\ \texttt{x}_2\ \texttt{res})]
\end{array}
$$

There is no one strategy suitable for all cases

There is no one 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

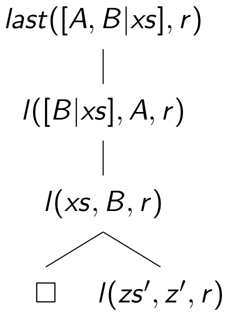
- Given:
  - Logic program
  - Goal
- Result: specialized program
- How:
  - Construct a *partial* SLD-tree
  - Generate a program from the tree
- Hopefully, all excessive computations are done statically and will not be in the generated program

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).
```

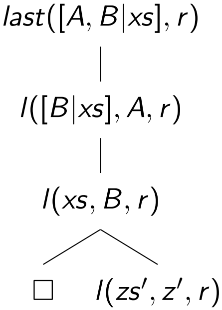
Partial Deduction: Example

```
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

```
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).
```




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

Partial Deduction: Conjunctions

$$f(x, y) \vee f(y, z)$$

$$f(x, y) \qquad f(y, z)$$

$\checkmark$

Partial Deduction: Conjunctions

$$f(x, y) \vee f(y, z)$$

$$f(x, y) \qquad f(y, z)$$

$\checkmark$

$$f(x, y) \wedge f(y, z)$$

$$f(x, y) \qquad f(y, z)$$

X

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 doubleAppendᵒ x y z xyz =
  (fresh (t) (
     (appendᵒ x y t) ∧
     (appendᵒ t z xyz)))

let rec appendᵒ x y xy = conde [
  (x ≡ nil () ∧ xy ≡ y);
  (fresh (h t ty) (
     (x  ≡ h % t)  ∧
     (xy ≡ h % t') ∧
     (appendᵒ t y t')))]
```

```
let rec doubleAppendᵒ x y z xyz = conde [
   (x ≡ nil () ∧ appendᵒ y z xyz);
   (fresh (h t t') (
      (x ≡ h % t)   ∧
      (xyz ≡ h % t') ∧
      (doubleAppendᵒ t y z t')))]
```

Tupling

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

```
let maxLength° xs m l = max° xs m ∧ length° xs l

let rec length° xs l = conde [
  (xs ≡ nil () ∧ l ≡ zero ());
  (fresh (h t m) (
    xs ≡ h % t ∧ l ≡ succ m ∧ length° t m))]

let max° xs m = max°₁ xs (zero ()) m

let rec max°₁ xs n m = conde [
  (xs ≡ nil () ∧ m ≡ n);
  (fresh (h t) (
    (xs ≡ h % t) ∧
    (conde [
      (le° h n ↑true ∧ max°₁ t n m);
      (gt° h n ↑true ∧ max°₁ t h m)]))))]
```

Tupling

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

```
let maxLengthᵒ xs m l = maxLength₁ᵒ xs m (zero ()) l

let rec maxLength₁ᵒ xs m n l = conde [
  (xs ≡ nil () ∧ m ≡ n ∧ l ≡ zero ());
  (fresh (h t l₁)
     (xs ≡ h % t) ∧
     (l ≡ succ l₁) ∧
     (conde [
       (leᵒ h n ∧ maxLength₁ᵒ t m n l);
       (gtᵒ h n ∧ maxLength₁ᵒ t m h l)]))]
```

CPD: Intuition

- Local control: compute a partial SLDNF-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*?

CPD: Implementation

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

Evaluation

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

Path Search

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

- N — set of nodes
- E — set of edges
- Functions $start, end : E \to N$ 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 isPath ns g =
  match ns with
  | x₁ :: x₂ :: xs → elem (x₁, x₂) g && isPath (x₂ :: xs) g
  | [_]            → true
```

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

```
let rec isPathᵒ ns g res = conde [
  (fresh (el) ((ns ≡ el % nil ()) ∧ (res ≡ ↑true));
  (fresh (x₁ x₂ xs resElem resIsPath) (
    (ns ≡ x₁ % (x₂ % xs)) ∧
    (elemᵒ (pair x₁ x₂) g resElem) ∧
    (isPathᵒ (x₂ % xs) g resIsPath) ∧
    (conde [
      (resElem ≡ ↑false  ∧  res ≡ ↑false);
      (resElem ≡ ↑true   ∧  res ≡ resIsPath)]))))]
```

This relation is inefficient for "isPath^o q <graph> **true**"

Path Search: Specialized Relation

$$
\begin{array}{l}
\textbf{let rec } \texttt{isPath}^o \ \texttt{ns g res} = \textbf{conde } [\\
\quad (\textbf{fresh } (\texttt{el}) \ ((\texttt{ns} \equiv \texttt{el} \,\%\, \texttt{nil} \ ()) \ \wedge \ (\texttt{res} \equiv \uparrow\textbf{true}))); \\
\quad (\textbf{fresh } (\texttt{x}_1 \ \texttt{x}_2 \ \texttt{xs resElem resIsPath}) \ (\\
\qquad (\texttt{resElem} \equiv \uparrow\textbf{true}) \ \wedge \\
\qquad (\texttt{resIsPath} \equiv \uparrow\textbf{true}) \ \wedge \\
\qquad (\texttt{ns} \equiv \texttt{x}_1 \,\%\, (\texttt{x}_2 \,\%\, \texttt{xs})) \ \wedge \\
\qquad (\texttt{elem}^o \ (\texttt{pair } \texttt{x}_1 \ \texttt{x}_2) \ \texttt{g resElem}) \ \wedge \\
\qquad (\texttt{isPath}^o \ (\texttt{x}_2 \,\%\, \texttt{xs}) \ \texttt{g resIsPath})))]
\end{array}
$$

Better performance for "$\texttt{isPath}^o \ \texttt{q} <\texttt{graph}> \textbf{true}$"

Path Search: Specialized Relation

```
let rec isPathᵒ ns g res = conde [
  (fresh (el) ((ns ≡ el % nil ()) ∧ (res ≡ ↑true)));
  (fresh (x₁ x₂ xs resElem resIsPath) (
    (resElem ≡ ↑true) ∧
    (resIsPath ≡ ↑true) ∧
    (ns ≡ x₁ % (x₂ % xs)) ∧
    (elemᵒ (pair x₁ x₂) g resElem) ∧
    (isPathᵒ (x₂ % xs) g resIsPath)))]
```

Better performance for "$\texttt{isPath}^o$ 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)

Unification

Term:

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

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, X)	f(a % b % nil, c % d % nil, L) f(X % XS, YS, X % ZS)	f(X, X, g(Z, t)) 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)

Conclusion & Future Work

Funcional verifier + unnesting + specialization = solver

Future

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