



# 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

# Solvers from Verifiers

Relational interpreter = verifier

Relational interpreter being run backward = solver

```
evalo prog ?? res
```

```
isPatho path graph res
```

```
unifyo term term' subst res
```

```
run q (isPatho q graph True) — searches for all paths in the graph
```

- Implement a functional program which verifies the solution for a program
- Transform it into a relation
- Specialize for the backward direction
- The result can search for solutions

## 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 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 =  
  (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  $f_1\ x_1 \ \&\& \ f_2\ x_2$ :

```
 $\lambda\ res \rightarrow$   
  fresh (p) (  
    ( $f_1\ x_1\ p$ )  $\wedge$   
    (conde [  
      ( $p \equiv \uparrow\mathbf{false} \wedge res \equiv \uparrow\mathbf{false}$ );  
      ( $p \equiv \uparrow\mathbf{true} \wedge f_2\ x_2\ res$ )]))
```

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 backward execution, but not forward

```
 $\lambda \text{ res} \rightarrow$   
  conde [  
    ( $\text{res} \equiv \uparrow \text{false} \wedge f_1 \ x_1 \uparrow \text{false}$ );  
    ( $f_1 \ x_1 \uparrow \text{true} \wedge f_2 \ x_2 \text{ res}$ )]
```

There is no one strategy suitable for all cases

Better is to use an automatic specializer

# Specialization

Interpreter: given a program and input computes an output

`eval prog input == output`

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

Specializer: given a program and static input, generates a new program, which evaluates to the same output as the original

`spec prog static  $\Rightarrow$  progspec`

`eval prog (static, dynamic) == eval progspec dynamic`

# 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 maxLengtho xs m l = maxo xs m ∧ lengtho xs l
```

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

```
let maxo xs m = max1o xs (zero ()) m
```

```
let rec max1o xs n m = conde [  
  (xs ≡ nil () ∧ m ≡ n);  
  (fresh (h t) (  
    (xs ≡ h % t) ∧  
    (conde [  
      (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 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)])))]
```

- 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*?



- 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

## 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

# Evaluation: Path Search

<Task formulation>

<The size of the program before and after unnesting and specialization>

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)

# Evaluation: Unification

<Task formulation>

<The size of the program before and after unnesting and specialization>

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)

# Conclusion & Future Work

Functional verifier + unnesting + specialization = solver

Future

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