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 interpeter = verifier

Relational intepreter 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

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
```

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{append}^o\ \mathtt{a\ b\ c} = \\
\quad (\mathtt{a} \equiv [\,]\ \wedge\ \mathtt{b} \equiv \mathtt{c})\ \vee \\
\quad (\quad (\mathtt{a} \equiv \mathtt{x}\ ::\ \mathtt{xs})\ \wedge \\
\qquad (\mathtt{append}^o\ \mathtt{xs\ b\ q})\ \wedge \\
\qquad (\mathtt{c} \equiv \mathtt{x}\ ::\ \mathtt{q}))
\end{array}
$$

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 $\texttt{f}_1\ \texttt{x}_1$ && $\texttt{f}_2\ \texttt{x}_2$:

```
λ res →
  fresh (p) (
    (f₁ x₁ p) ∧
    (conde [
      (p ≡ ↑false  ∧  res ≡ ↑false);
      (p ≡ ↑true   ∧  f₂ x₂ res)]))
```

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

It is not the best strategy, if we know what res is

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

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

There is no one strategy suitable for all cases

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 evaluated to the same output as the original

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

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

Conjunctive Partial Deduction

CPD — specialization for prolog
Features: specialization, deforestation, tupling

Deforestation

<example doubleAppendo>

Tupling

<example maxLengtho>

CPD: Intuition

Symbolic execution + ensuring termination
Treating conjunctions as a whole
Embedding + Generalization?

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

Evaluation: Path Search

<Task formulation>
<The size of the program before and after unnesting and specialization>
<Table with time mesurements>

Evaluation: Unification

<Task formulation>
<The size of the program before and after unnesting and specialization>
<Table with time mesurements>

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