On a Direction-Driven Functional Conversion

Kate Verbitskaia, Daniil Berezun, Dmitry Boulytchev

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

15.09.2022

Solvers from Verifiers

An inverse of a verifier is a solver

Verifier is much easier to implement than a solver

Inverse Computations

Given a program p:

$$[\![p]\!]x = y$$

Its inversion is:

$$[\![p^{-1}]\!]y = x$$

Program inverter:

$$[\![invtrans]\!]p = p^{-1}$$

Inverse interpreter:

$$[\![invint]\!][p, y] = x$$

miniKanren Works as an Inverse Interpreter

miniKanren can run a verifier backwards

run q (eval^o q **true**)

Principal Directions of miniKanren Relations

Every argument of a relation can be either `in` or `out`

For addition relation $\texttt{add}^o$ `x y z` there are 8 directions:

- *Forward* direction: $\texttt{add}^o$ `in in out`
- *Backwards* direction: $\texttt{add}^o$ `out out in`
- *Predicate*: $\texttt{add}^o$ `in in in`
- *Generator*: $\texttt{add}^o$ `out out out`
- $\texttt{add}^o$ `in out in`
- $\texttt{add}^o$ `out in in`
- $\texttt{add}^o$ `out in out`
- $\texttt{add}^o$ `in out out`

miniKanren Comes with an Overhead

Unifications

Scheduling complexity

Occurs-check

Functional Conversion

Given a relation and a principal direction, construct a functional program which generates the same answers as miniKanren would

Preserve completeness of the search

Both inputs and outputs are expected to be ground

Example: Addition in Forward Direction

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧
     z ≡ S z' ∧
     addᵒ x' y z') ) ]
```

```
addXY :: Nat → Nat → Nat
addXY x y =
  case x of
    O → y
    S x' → S (addXY x' y)
```

Addition in Backwards Direction: Nondeterminism

```
let rec add° x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧
     z ≡ S z' ∧
     add° x' y z') ) ]
```

```
addZ :: Nat → Stream (Nat, Nat)
addZ z =
  return (O, z) `mplus`
  case z of
    O → Empty
    S z' → do
      (x', y) ← addZ z'
      return (S x', y)
```

Free Variables in Answers: Generators

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧ z ≡ S z' ∧ addᵒ x' y z') ) ]
```

```
addX :: Nat → Stream (Nat, Nat)
addX x = case x of
           O → do
             z ← genNat
             return (z, z)
           S x' → do
             (y, z') ← addX x'
             return (y, S z')

genNat :: Stream Nat
genNat = Mature O (S <$> genNat)
```

Predicates

```
let rec addᵒ x y z = conde [
  (x ≡ O  ∧  y ≡ z);
  (fresh (x' z')
    (x ≡ S x'  ∧
     z ≡ S z'  ∧
     addᵒ x' y z') ) ]
```

```
addXYZ :: Nat → Nat → Nat → Stream ()
addXYZ x y z =
  case x of
    O | y == z → return ()
      | otherwise → Empty
    S x' →
      case z of
        O → Empty
        S z' → addXYZ x' y z'
```

Order in Conjunctions

```
let rec multᵒ x y z = conde [
  ...
  (fresh (x' r')
    (x ≡ S x') ∧
    (add y r' z) ∧
    (mult x' y r')
  )]
```

Order in Conjunctions: Slow Version

```
multXY' :: Nat → Nat → Stream Nat
...
multXY' (S x') y    = do
  (r', r) ← addX y
  multXYZ x' y r'
  return r

multXYZ :: Nat → Nat → Nat → Stream ()
...
multXYZ (S x') y     z = do
  z' ← multXY' x' y
  addXYZ y z' z
multXYZ _ _ _ = Empty
```

Order in Conjunctions: Faster Version

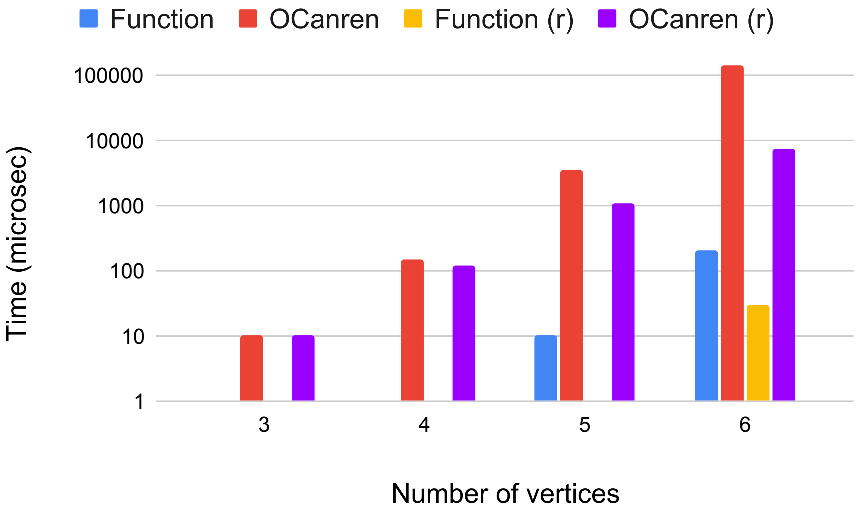
```
multXY :: Nat → Nat → Stream Nat
...
multXY (S x') y     = do
  r' ← multXY x' y
  addXY y r'
```

Evaluation

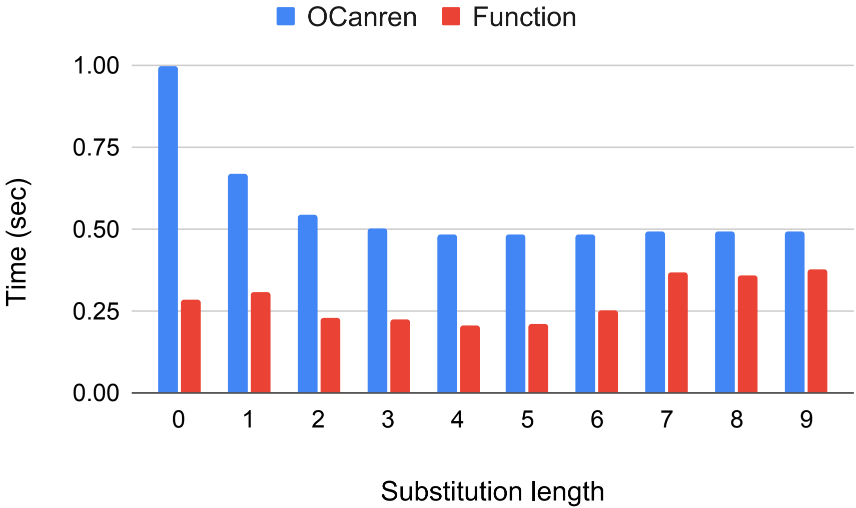
We manually converted relational interpreters and measured execution time

- Topologic sort
  - A verifier verifies that a vertex mapping sorts vertices topologically
  - Sort a DAG with an edge in between every pair of vertices
  - Two different representations: vertices sorted by their number, and with a reverse order
  - Sorting a graph with up to 6 vertices
- Logic formulas generation
  - Inverse computation of a logic formulas interpreter
  - Generate 10000 formulas which evaluate to `true`
  - Different substitution lengths

Evaluation: Topologic Sort



Evaluation: Logic Formulas Generation



Conclusion

Conclusion

- We presented a functional conversion scheme as a series of examples
- The conversion speeds up implementations considerably

Future work

- Implementation and formalization of the conversion scheme
- Finding a better way to order conjuncts
- Integration into a relational interpreters for solving framework