Semi-Automatic Functional Conversion of microKanren

Igor Engel, **Kate Verbitskaia**

JetBrains Research, Programming Languages and Tools Lab

30.05.2023

Relational Programming

One relation to solve many problems

Nondeterminism

Completeness of search

Relational Conversion: Easy

Given a function

```
let rec add x y =
  match x with
  | O → y
  | S x' → S (add x' y)
```

generate miniKanren relation

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

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` — addition
- *Backward* direction: $\texttt{add}^o$ `out out in` — decomposition
- *Predicate*: $\texttt{add}^o$ `in in in`
- *Generator*: $\texttt{add}^o$ `out out out`
- $\texttt{add}^o$ `in out in` — subtraction
- $\texttt{add}^o$ `out in in` — subtraction
- $\texttt{add}^o$ `out in out`
- $\texttt{add}^o$ `in out out`

Each Direction is a Function

Each Direction is a Function (kinda)

Straightforward functions:

- *Forward* direction: `add`^{o} `in in out` — addition
- `add`^{o} `in out in` — subtraction
- `add`^{o} `out in in` — subtraction
- *Predicate*: `add`^{o} `in in in`

Relations:

- *Backward* direction: `add`^{o} `out out in` — decomposition
- *Generator*: `add`^{o} `out out out`
- `add`^{o} `out in out`
- `add`^{o} `in out out`

These relations are functions which return multiple answers (list monad)

miniKanren Comes with an Overhead

Unifications

Occurs-check

Scheduling complexity

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' ∧
     addᵒ x' y z' ∧
     z ≡ S z')) ]
```

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

Addition in Backwards Direction: Nondeterminism

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

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

```
addIOO :: Nat → Stream (Nat, Nat)
addIOO x = case x of
           O → do
             z ← genNat
             return (z, z)
           S x' → do
             (y, z') ← addIOO 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' ∧
     addᵒ x' y z' ∧
     z ≡ S z')) ]
```

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

Order in Conjunctions

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

Order in Conjunctions: Slow Version

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

multIII :: Nat → Nat → Nat → Stream ()
...
multIII (S x') y z = do
  z' ← multIIO' x' y
  addIII y z' z
multIII _ _ _ = Empty
```

Order in Conjunctions: Faster Version

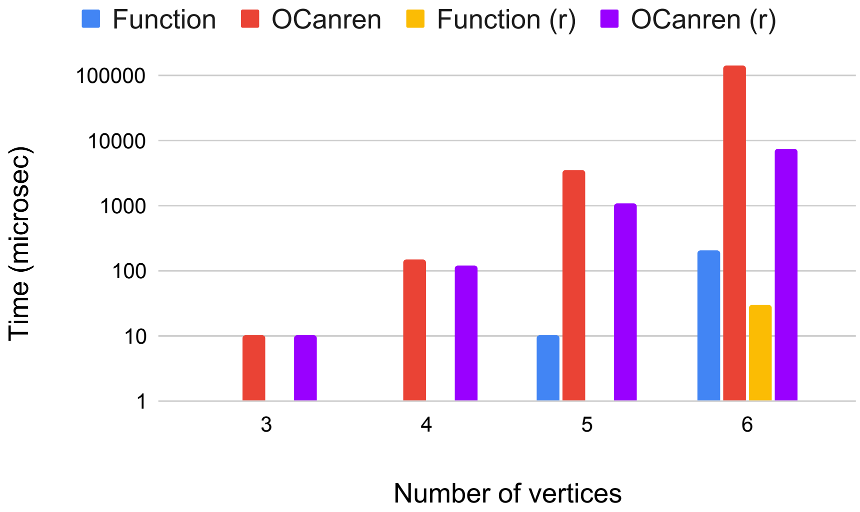
```
multIIO :: Nat → Nat → Stream Nat
...
multIIO (S x') y = do
  r' ← multIIO x' y
  addXY y r'
```

Evaluation

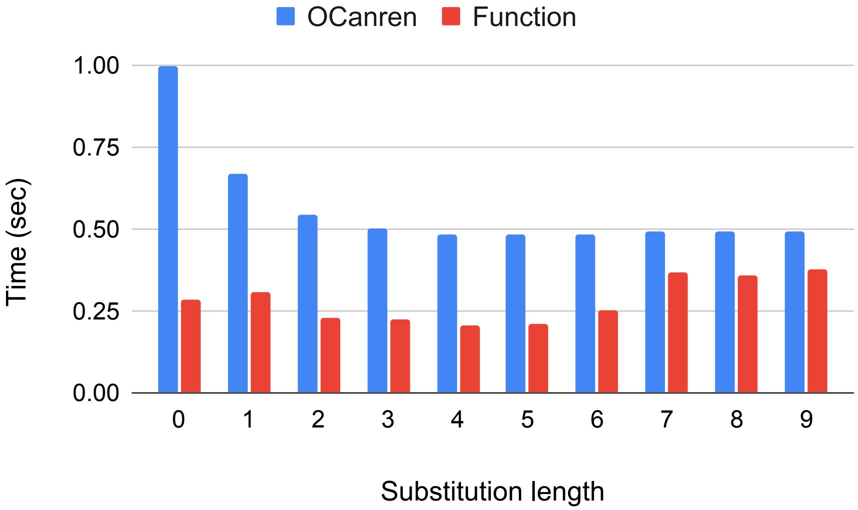
We manually converted relational interpreters and measured execution time

- 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