Semi-Automatic Functional Conversion for 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 add^o x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x' z')
    (x ≡ S x' ∧
     add^o 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'
```

Conversion Scheme

- Normalization
- Mode analysis
- Functional conversion

Normalization: Flat Term

Flat terms: a var or a constructor which takes distinct vars as arguments:

$$\mathcal{FT}_V = V \cup \{\mathcal{C}_i\,(x_1, \ldots, x_{k_i}) \mid x_j \in V\}$$

Examples:

$$C\,(x_1, x_2) \equiv C\,(C\,(y_1, y_2)\,, y_3) \iff x_1 \equiv C\,(y_1, y_2) \land x_2 \equiv y_3$$

$$C\,(C\,(x_1, x_2)\,, x_3) \equiv C\,(C\,(y_1, y_2)\,, y_3) \iff x_1 \equiv y_1 \land x_2 \equiv y_2 \land x_3 \equiv y_3$$

$$x \equiv C\,(y, y) \iff x \equiv C\,(y_1, y_2) \land y_1 \equiv y_2$$

Normalization: Goal

$$
\begin{array}{llll}
\mathcal{K}_V^N & = & \bigvee(c_1, \dots, c_n), c_i \in \mathsf{Conj}_V & \text{normal form} \\
\mathsf{Conj}_V & = & \bigwedge(g_1, \dots, g_n), g_i \in \mathsf{Base}_V & \text{normal conjunction} \\
\mathsf{Base}_V & = & V \equiv \mathcal{FT}_V & \text{flat unification} \\
 & \mid & R_i^d(x_1, \dots, x_{k_i}), d \in \mathsf{Delay}, x_j \in V & \text{flat call} \\
\mathsf{Delay} & = & \{\mathsf{Delay}, \mathsf{NoDelay}\} &
\end{array}
$$

Mode of a Variable

Mode of a variable: mapping between its instantiations

Ground term has no variables
Free variable: fresh variable, no info about its instantiation
Once we know that a variable is *ground*, it stays *ground* in subsequent conjuncts

Mode *in*: *ground* $\rightarrow$ *ground*
Mode *out*: *free* $\rightarrow$ *ground*

Mercury uses more complicated modes

Modded Goal

Assign mode to every variable, make sure they are consistent

Modded Unification

- Assignments: $x^{\text{out}} \equiv \mathcal{T}^{\text{in}}$ and $x^{\text{in}} \equiv y^{\text{out}}$
- Guards: $x^{\text{in}} \equiv \mathcal{T}^{\text{in}}$
- Match: $x^{\text{in}} \equiv \mathcal{T}$ ($\mathcal{T}$ contains both *in* and *out* variables)
- Generators: $x^{\text{out}} \equiv \mathcal{T}$

Mode Inference: Initialization

- For all input variables: *ground* $\rightarrow$?
- For all other variables: *free* $\rightarrow$?

$$
\begin{array}{l}
\textbf{let rec } \texttt{add}^o\ (\texttt{x},\ g \rightarrow g)\ (\texttt{y},\ g \rightarrow g)\ (\texttt{z},\ f \rightarrow g) = \textbf{conde}\ [\\
\quad ((\texttt{x},\ g \rightarrow g)\ \equiv \texttt{O}\ \wedge\ (\texttt{y},\ g \rightarrow g) \equiv (\texttt{z},\ f \rightarrow g)); \\
\quad (((\texttt{x},\ g \rightarrow g) \equiv \texttt{S}\ (\texttt{x'},\ f \rightarrow ?)\ \wedge \\
\qquad \texttt{add}^o\ (\texttt{x'},\ f \rightarrow ?)\ (\texttt{y},\ g \rightarrow g)\ (\texttt{z'},\ f \rightarrow ?)\ \wedge \\
\qquad (\texttt{z}, f \rightarrow g) \equiv \texttt{S}\ (\texttt{z'},\ f \rightarrow ?)))\ ]
\end{array}
$$

Mode Inference: Disjunction

Run inference on each disjunct independently

$$((\texttt{x},\ \texttt{g} \to \texttt{g})\ \equiv \texttt{O}\ \wedge\ (\texttt{y},\ \texttt{g} \to \texttt{g}) \equiv (\texttt{z},\ \texttt{f} \to \texttt{g}))$$

$$
\begin{aligned}
&(((\texttt{x},\ \texttt{g} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{x'},\ \texttt{f} \to ?)\ \wedge \\
&\quad \texttt{add}^o\ (\texttt{x'},\ \texttt{f} \to ?)\ (\texttt{y},\ \texttt{g} \to \texttt{g})\ (\texttt{z'},\ \texttt{f} \to ?)\ \wedge \\
&\quad (\texttt{z},\texttt{f} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{z'},\ \texttt{f} \to ?)))
\end{aligned}
$$

Mode Inference: Unification

Propagate the groundness information according to the 4 types of modded unifications

$$(((\texttt{x},\ \texttt{g} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{x'},\ \texttt{f} \to ?)) \Rightarrow ((\texttt{x},\ \texttt{g} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{x'},\ \texttt{f} \to \texttt{g})))$$

$$(((\texttt{z},\texttt{f} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{z'},\ \texttt{f} \to ?)) \Rightarrow ((\texttt{z},\texttt{f} \to \texttt{g}) \equiv \texttt{S}\ (\texttt{z'},\ \texttt{f} \to \texttt{g})))$$

Mode Inference: Conjunction

Pick a conjunct according to the priority, propagate groundness

- Guards
- Assignments
- Matches
- Calls with at least one ground argument
- Generators

Mode Inference: Conjunction

```
(((x, g → g) ≡ S (x', f → ?)  ∧
  add^o (x', f → ?) (y, g → g) (z', f → ?)  ∧
  (z,f → g) ≡ S (z', f → ?)))
```

```
(((x, g → g) ≡ S (x', f → g)  ∧
  add^o (x', f → g) (y, g → g) (z', f → ?)  ∧
  (z,f → g) ≡ S (z', f → ?)))
```

```
(((x, g → g) ≡ S (x', f → g)  ∧
  add^o (x', f → g) (y, g → g) (z', f → g)  ∧
  (z,f → g) ≡ S (z', f → g)))
```

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'
```

Functional Conversion: Intermediate Language

$$
\begin{array}{lcll}
\mathcal{F}_V & = & \text{Return}\,[\mathcal{T}_V] & \text{return a tuple of terms} \\
& | & \text{Match}_V\,(\mathcal{T}_V, \mathcal{F}_V) & \text{match a variable against a pattern} \\
& | & \text{Bind}\,[([V]\,, \mathcal{F}_V)] & \text{monadic bind on streams} \\
& | & \text{Sum}\,[\mathcal{F}_V] & \text{concatenation of streams} \\
& | & \text{Guard}\,(V, V) & \text{equality check} \\
& | & \text{Gen}_G & \text{generator} \\
& | & R_i^d([V]\,, [G]), d \in \text{Delay} & \text{function call}
\end{array}
$$

Functional Conversion into Intermediate Language

- Disjunction $\rightarrow$ Sum $[\mathcal{F}_V]$
- Conjunction $\rightarrow$ Bind $[([V], \mathcal{F}_V)]$
- Relation call $\rightarrow R_i^d([V], [G]), d \in$ Delay
- Unification $\rightarrow$ Return $[\mathcal{T}_V]$ or Match$_V(\mathcal{T}_V, \mathcal{F}_V)$ or Guard (V, V) or Gen$_G$

Functional Conversion into Haskell

- TemplateHaskell to generate code
- Stream monad
- do-notation

Functional Conversion into OCaml

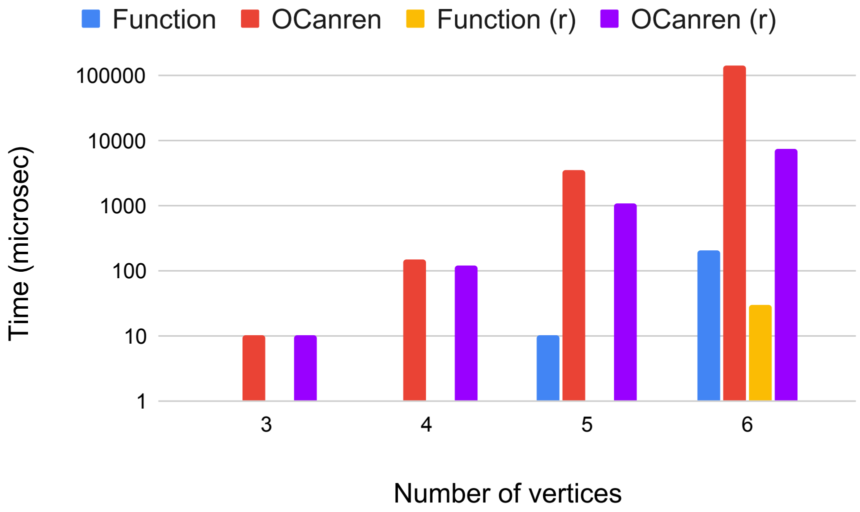
- Hand-crafted (not so) pretty-printer
- Stream monad
- let*
- Taking extra care to employ laziness

Evaluation (last year)

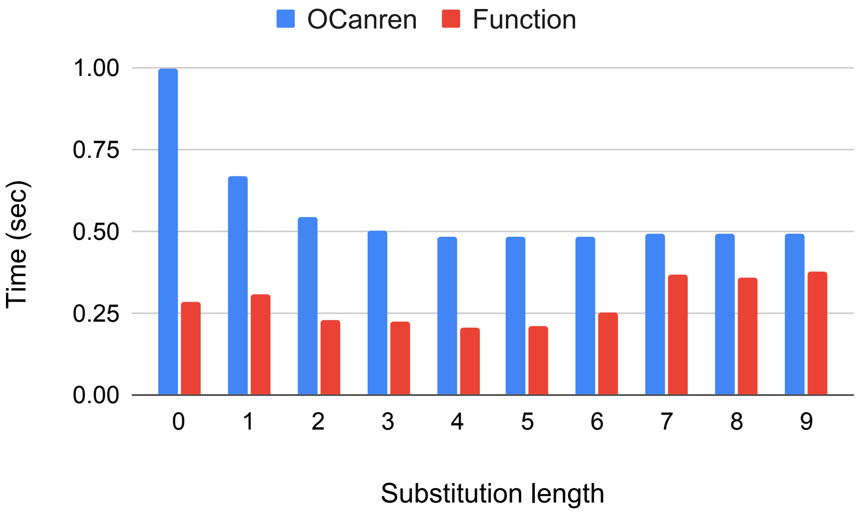
We manually converted relational interpreters and measured execution time

- 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 (last year)



Evaluation: Logic Formulas Generation (last year)



Evaluation: Addition Relation (Time in Seconds)

- addIIO x = 10000, y = 0
  - Fun: 0.007
  - Rel: 1.533
- addOII y = 0, z = 10000
  - Fun: 0.009
  - Rel: 1.547
- addIOI x = 10000, z = 10000
  - 0.009
  - 3.029
- addIII x = 10000, y = 0, z = 10000
  - 0.008
  - 3.041
- addIOO x = 0, n = 1000
  - Fun: 0.143
  - Rel: 0.000
- addOOO n = 1000
  - Fun: 0.074
  - Rel: 0.585

Evaluation: Proposition Evaluator (Time in Seconds)

- evalo [true; false; true] fm true
  - Fun: 0.759
  - Rel: 0.308

Data Types

We generate weird data type declarations:

```
type term =
  | Conj of (term* term)
  | Cons of (term* term)
  | Disj of (term* term)
  | Falso
  | Neg of term
  | Succ of term
  | Trueo
  | Var of term
  | Zero
```

```
elemᵒ i st v =
  fresh (h t i') conde [
    (i ≡ Zero ∧ st ≡ Cons (v, t));
    (i ≡ Succ i' ∧ st ≡ Cons (h, t) ∧ elemᵒ i' t v)]
```

Need for Determinism Check

- Replacing Stream with Maybe improves performance about 10 times for relations on natural numbers
- Functional (no monad) version is still faster
- Use determinism check to figure out when replacing Stream is feasible
- How to combine different monads naturally?

Need for Partial deduction

miniKanren can run a verifier backwards to get solver

run `q` (`eval`^{o} `q` **true**)

Augmenting functional conversion with partial deduction must be beneficial

Conclusion

Conclusion

- We presented a functional conversion scheme as a series of examples
- The conversion speeds up implementations considerably
- We implemented the conversion scheme in Haskell
- Found some way to order conjuncts

Future work

- Integration with partial deduction
- Integration into a relational interpreters for solving framework