Semi-Automated Direction-Driven Functional Conversion

Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research, Programming Languages and Tools Lab

miniKanren workshop @ ICFP 2023

08.09.2023

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 x₁ → S (add x₁ y)
```

generate miniKanren relation

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁)) ]
```

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 (kind of)

Straightforward functions:

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

Relations:

- *Backward* direction: $\texttt{add}^o$ `out out in` — decomposition
- *Generator*: $\texttt{add}^o$ `out out out`
- $\texttt{add}^o$ `out in out`
- $\texttt{add}^o$ `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 (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁)) ]
```

```
addIIO :: Nat → Nat → Nat
addIIO x y =
  case x of
    O → y
    S x₁ → S (addIIO x₁ y)
```

Addition in the Backward Direction: Nondeterminism

```
let rec addᵒ x y z = conde [
  (x ≡ O  ∧  y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁  ∧
     addᵒ x₁ y z₁  ∧
     z ≡ S z₁)) ]
```

```
addOOI :: Nat → Stream (Nat, Nat)
addOOI z =
  return (O, z) `mplus`
  case z of
    O  → Empty
    S z₁  → do
      (x₁, y) ← addOOI z₁
      return (S x₁, y)
```

Free Variables in Answers: Generators

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧ z ≡ S z₁ ∧ addᵒ x₁ y z₁) ) ]
```

```
addIOO :: Nat → Stream (Nat, Nat)
addIOO x = case x of
           O → do
             z ← genNat
             return (z, z)
           S x₁ → do
             (y, z₁) ← addIOO x₁
             return (y, S z₁)

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

Predicates

```
let rec addᵒ x y z = conde [
  (x ≡ O ∧ y ≡ z);
  (fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁)) ]
```

```
addIII :: Nat → Nat → Nat → Stream ()
addIII x y z =
  case x of
    O | y == z → return ()
      | otherwise → Empty
    S x₁ →
      case z of
        O → Empty
        S z₁ → addIII x₁ y z₁
```

Conversion Scheme

- Normalization
- Mode analysis
- Functional conversion

Normalization: Flat Term

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

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

Examples:

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

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

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

Normalization: Goal

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

Mode of a Variable

Mode of a variable: mapping between its instantiations

Ground term contains 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* $\rightarrow$?

$$
\begin{aligned}
&\textbf{let rec}\ \mathtt{add}^o\ \mathtt{x}^{g\rightarrow g}\ \mathtt{y}^{g\rightarrow g}\ \mathtt{z}^{f\rightarrow g} = \textbf{conde}\\
&\quad(\mathtt{x}^{g\rightarrow g} \equiv \mathtt{0} \wedge \mathtt{y}^{g\rightarrow g} \equiv \mathtt{z}^{f\rightarrow g});\\
&\quad(\mathtt{x}^{g\rightarrow g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\rightarrow ?} \wedge\\
&\quad\ \mathtt{add}^o\ \mathtt{x}_1^{f\rightarrow ?}\ \mathtt{y}^{g\rightarrow g}\ \mathtt{z}_1^{f\rightarrow ?} \wedge\\
&\quad\ \mathtt{z}^{f\rightarrow g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\rightarrow ?})
\end{aligned}
$$

Mode Inference: Disjunction

Run inference on each disjunct independently

$$\mathtt{x}^{g \to g} \equiv \mathtt{0} \wedge \mathtt{y}^{g \to g} \equiv \mathtt{z}^{f \to g}$$

$$\begin{aligned}
&\mathtt{x}^{g \to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f \to ?} \wedge \\
&\mathtt{add}^o\ \mathtt{x}_1^{f \to ?}\ \mathtt{y}^{g \to g}\ \mathtt{z}_1^{f \to ?} \wedge \\
&\mathtt{z}^{f \to g} \equiv \mathtt{S}\ \mathtt{z}_1^{f \to ?}
\end{aligned}$$

Mode Inference: Unification

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

$$\mathtt{x}^{g\rightarrow g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\rightarrow ?} \Rightarrow \mathtt{x}^{g\rightarrow g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\rightarrow g}$$

$$\mathtt{z}^{f\rightarrow g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\rightarrow ?} \Rightarrow \mathtt{z}^{f\rightarrow g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\rightarrow g}$$

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

$$\begin{array}{l} \mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to ?} \ \wedge \\ \mathtt{add}^o\ \mathtt{x}_1^{f\to ?}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{f\to ?} \ \wedge \\ \mathtt{z}^{g\to g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\to ?} \end{array}$$

$$\begin{array}{l} \mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g} \ \wedge \\ \mathtt{add}^o\ \mathtt{x}_1^{f\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{f\to ?} \ \wedge \\ \mathtt{z}^{g\to g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\to ?} \end{array}$$

$$\begin{array}{l} \mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g} \ \wedge \\ \mathtt{add}^o\ \mathtt{x}_1^{f\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{f\to g} \ \wedge \\ \mathtt{z}^{g\to g} \equiv \mathtt{S}\ \mathtt{z}_1^{f\to g} \end{array}$$

Order in Conjunctions

```
let rec multᵒ x y z = conde [
  ...
  (fresh (x₁ r')
    (x ≡ S x₁) ∧
    (addᵒ y r' z) ∧
    (multᵒ x₁ y r')
  )]
```

Order in Conjunctions: Slow Version

```
multIIO₁ :: Nat → Nat → Stream Nat
...
multIIO₁ (S x₁) y  = do
  (r₁, r) ← addX y
  multIII x₁ y r₁
  return r

multIII :: Nat → Nat → Nat → Stream ()
...
multIII (S x₁) y z = do
  z₁ ← multIIO₁ x₁ y
  addIII y z₁ z
multIII _ _ _ = Empty
```

Premature grounding of z_1 leads to generate-and-test behavior

Order in Conjunctions: Faster Version

```
multIIO :: Nat → Nat → Stream Nat
...
multIIO (S x₁) y = do
  r' ← multIIO x₁ y
  addXY y r'
```

Functional Conversion: Intermediate Language

$\mathcal{F}_V$	$=$	$\textbf{Return}\,[\mathcal{T}_V]$	return a tuple of terms
	$\mid$	$\textbf{Match}_V\,(\mathcal{T}_V, \mathcal{F}_V)$	match a variable against a pattern
	$\mid$	$\textbf{Bind}\,[([V]\,, \mathcal{F}_V)]$	monadic bind on streams
	$\mid$	$\textbf{Sum}\,[\mathcal{F}_V]$	concatenation of streams
	$\mid$	$\textbf{Guard}\,(V, V)$	equality check
	$\mid$	$\textbf{Gen}_G$	generator
	$\mid$	$R_i([V]\,, [G])$	function call

Functional Conversion into Intermediate Language

- Disjunction $\to$ **Sum** $[\mathcal{F}_V]$
- Conjunction $\to$ **Bind** $[([V], \mathcal{F}_V)]$
- Relation call $\to$ $R_i([V], [G])$
- Unification $\to$ **Return** $[\mathcal{T}_V]$ or $\mathbf{Match}_V\,(\mathcal{T}_V, \mathcal{F}_V)$ or **Guard** (V, V) or $\mathbf{Gen}_G$

Functional Conversion: Generators

- Untyped version of miniKanren — the only natural generator is «all terms»
- Solution — pass generators like additional arguments of a function

```
addIOO :: Nat → Stream Nat → Stream (Nat, Nat)
addIOO x gen_z = case x of
          O → do
            z ← gen_z
            return (z, z)
          S x1 → do
            (y, z1) ← addIOO x1 gen_z
            return (y, S z1)
```

- A generator is required for every unification containing *free* terms on both sides, as well as transitive requirements, for passing to other calls
- In a typed version, it should be possible to automatically derive generators producing all elements of a type

```
multOIO :: Nat → Stream Nat → Stream Nat
multOIO y gen_add_z =
    return (O, O) `mplus`
    do
        (z1, z) ← addIOO y gen_add_z
        x ← muloOII y z1
        return (S x, z)
```

Functional Conversion into Haskell

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

Functional Conversion into OCaml

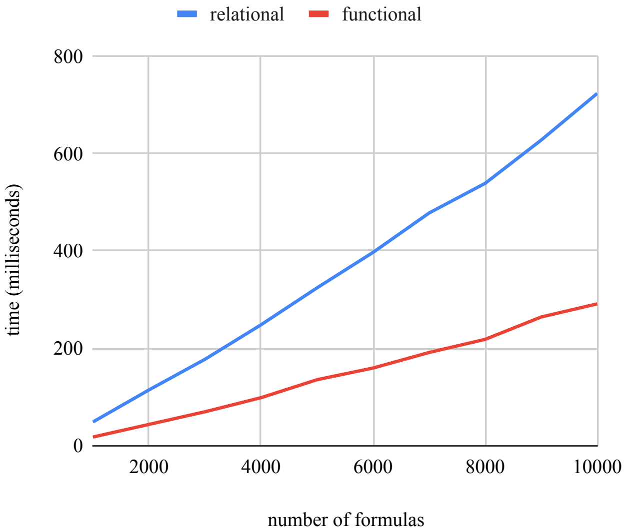
- Hand-crafted (not so) pretty-printer
- Stream monad
- let*
- Taking extra care to employ laziness

Evaluation

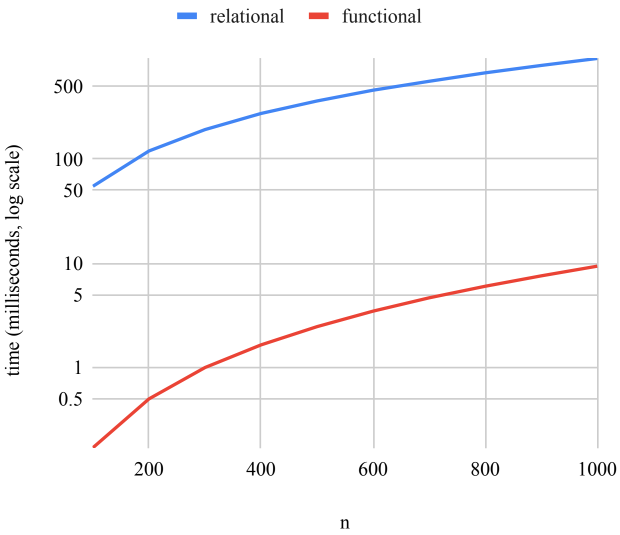
We converted relational interpreters and measured execution time

- Logic formulas generation
  - Inverse computation of an evaluator of logic formulas
  - Generating formulas which evaluate to **true**
- Multiplication relation
  - Forward direction: multiplication
  - Backward direction: division
  - Generation

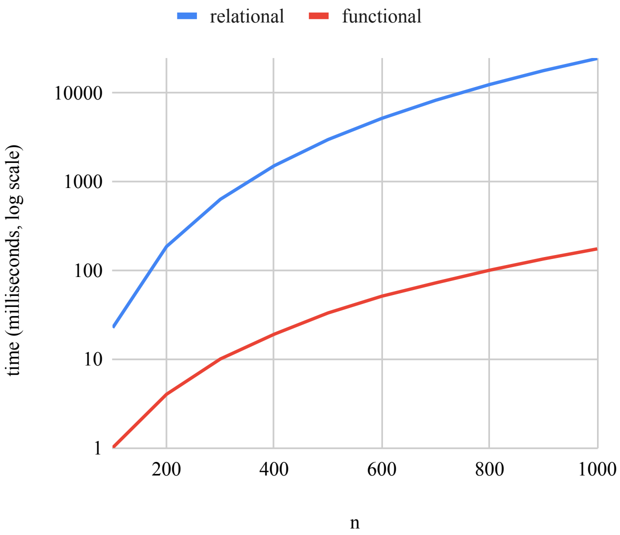
Generation of Logic Formulas: `evalo` [**true** ; **false** ; **true**] `q` **true**



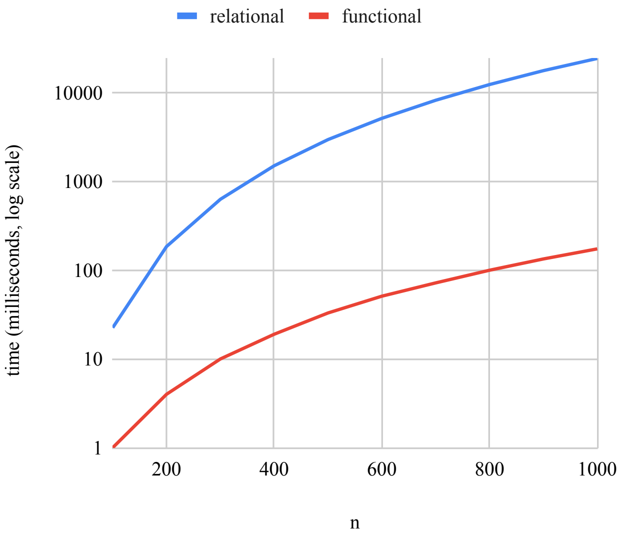
Multiplication: `mulo n 10 q`



Division: `mulo (n/10) q n`



Multiplication Generation: `take n (mulo 10 q r)`



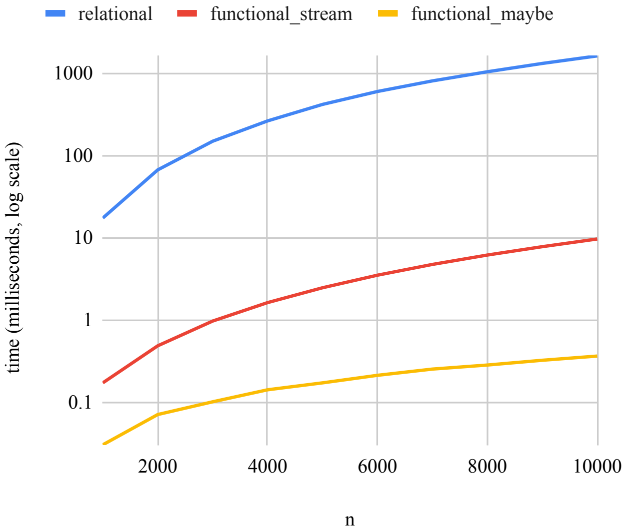
Data Types

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 i₁) conde [
    (i ≡ Zero ∧ st ≡ Cons (v, t));
    (i ≡ Succ i₁ ∧ st ≡ Cons (h, t) ∧ elemᵒ i₁ t v)]
```

Need for Determinism Check: `mulo q 10 1000`



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