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

Nondeterminism

Completeness of search

One relation to solve many problems

Relational Interpreters for Search Problems

Given a function

```
eval fm =
  match fm with
  | Conj (x, y) → and (eval x) (eval y)
  | ...
```

Generate a miniKanren relation

```
evalᵒ fm u = fresh (x y v w)
  (conde [
    (fm ≡ Conj x y ∧ andᵒ v w u ∧ evalᵒ x v ∧ evalᵒ y w);
    ...])
```

Run it to solve a search problem: `run q (evalᵒ q true)`

Principal Directions of miniKanren Relations

Every argument of a relation can be either input (`I`) or output (`O`)

~~Partially known terms such as (`Cons _.0 Nil`)~~

The 8 directions of the addition relation $\texttt{add}^o$ `x y z`:

Forward	$\texttt{add}^o$ `I I O`	addition
Backward	$\texttt{add}^o$ `O O I`	decomposition
Predicate	$\texttt{add}^o$ `I I I`	
Generator	$\texttt{add}^o$ `O O O`	
	$\texttt{add}^o$ `I O I`	subtraction
	$\texttt{add}^o$ `O I I`	subtraction
	$\texttt{add}^o$ `O I O`	
	$\texttt{add}^o$ `I O O`	

Each Direction is a Function

Each Direction is a Function (Kind of)

Functions

Forward	add^o `I I O`	addition
	add^o `I O I`	subtraction
	add^o `O I I`	subtraction
Predicate	add^o `I I I`	

Relations

Backward	add^o `O O I`	decomposition
Generator	add^o `O O O`	
	add^o `O I O`	
	add^o `I O O`	

Relations are functions which return multiple answers

miniKanren Comes with an Overhead

Unifications

Occurs-check

Scheduling complexity

Functional Conversion

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

Preserve the completeness of the search

Both inputs and outputs are expected to be ground

Example: Addition in the 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)
```

Example: Addition in the Forward Direction

```
addoIIO x0 x1 =
  msum [do {guard (x0 == 0);
            let {x2 = x1};
            return x2},
        do {x3 ← case x0 of
                {S y3 → return y3;
                 _ → mzero};
            x4 ← addoIIO x3 x1;
            let {x2 = S x4};
            return x2}]
```

Addition in the Backward Direction: Nondeterminism

```
let rec addᵒ x y z = conde [
  (x ≡ 0 ∧ y ≡ z);
  ( fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁) ) ]
```

```
addOOI :: Nat → Stream (Nat, Nat)
addOOI z =
  return (0, z) `mplus`
  case z of
    0 → 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 ≡ 0 ∧ y ≡ z);
  ( fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S 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)
```

Conversion Scheme

Normalization

Mode analysis

Functional conversion

Normalization: Flat Term

Flat terms: a variable or a constructor with distinct variables for arguments

$$\mathcal{FT}_V = V \cup \{\mathcal{C}\ x_0 \dots x_k \mid x_j \in V, x_j - \mathit{distinct}\}$$

$$\begin{aligned}
C(x_1, x_2) \equiv C(C(y_1, y_2), y_3) &\iff x_1 \equiv C(y_1, y_2) \wedge x_2 \equiv y_3 \\
C(C(x_1, x_2), x_3) \equiv C(C(y_1, y_2), y_3) &\iff x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3 \\
x \equiv C(y, y) &\iff x \equiv C(y_1, y_2) \wedge y_1 \equiv y_2
\end{aligned}$$

~~Constructors inside constructors~~
~~Repeating variables~~

Normalization: Goal

$$
\begin{array}{rclll}
\mathcal{K}_V^N & = & c_1 \vee \ldots \vee c_n & c_i \in \mathrm{Conj}_V & \text{normal form} \\
\mathrm{Conj}_V & = & g_1 \wedge \ldots \wedge g_n & g_i \in \mathrm{Base}_V & \text{normal conjunction} \\
\mathrm{Base}_V & = & V \equiv \mathcal{FT}_V & & \text{flat unification} \\
 & \mid & R\, x_1 \ldots x_k & x_j \in V, x_j - \mathit{distinct} & \text{flat call}
\end{array}
$$

~~Disjunctions within conjunctions~~

~~Empty disjunctions and conjunctions~~

~~Constructors as arguments of relation calls~~

Mode of a Variable

Ground term	no fresh variables	`Cons O (Cons (S O) Nil)`
Free variable	a fresh variable	`_.0`

Once we know that a variable is ground, it stays ground in later conjuncts

Mode of a variable: mapping between its instantiations

Mode `I`: `ground` $\rightarrow$ `ground`
Mode `O`: `free` $\rightarrow$ `ground`

Mercury uses more complicated modes

Modded Goal

Assign mode to every variable, make sure they are consistent

Modded Unification Types

$$\begin{array}{rl} \text{assignment} & x^{\mathtt{O}} \equiv \mathcal{T}^{\mathtt{I}} \\ & x^{\mathtt{I}} \equiv y^{\mathtt{O}} \\ \text{guard} & x^{\mathtt{I}} \equiv \mathcal{T}^{\mathtt{I}} \\ \text{match} & x^{\mathtt{I}} \equiv \mathcal{T} \\ \text{generator} & x^{\mathtt{O}} \equiv \mathcal{T} \end{array}$$

$\mathcal{T}$ contains both g and f variables

Mode Inference: Initialization

Input variables:	$\mathtt{I}$	$g \to g$
Output variables:	$\mathtt{O}$	$f \to g$
Other variables:	$?$	$f \to ?$

$$
\begin{array}{l}
\textcolor{blue}{\text{let}}\ \ \textcolor{blue}{\text{rec}}\ \ \mathtt{add}^o\ \mathtt{x}^\mathtt{I}\ \mathtt{y}^\mathtt{I}\ \mathtt{z}^\mathtt{O} = \textcolor{blue}{\text{conde}}\ [\\
\quad (\mathtt{x}^\mathtt{I} \equiv \mathtt{O}\ \wedge\ \mathtt{y}^\mathtt{I} \equiv \mathtt{z}^\mathtt{O});\\
\quad (\mathtt{x}^\mathtt{I} \equiv \mathtt{S}\ \mathtt{x}_1^?\ \wedge\\
\quad\ \ \mathtt{add}^o\ \mathtt{x}_1^?\ \mathtt{y}^\mathtt{I}\ \mathtt{z}_1^?\ \wedge\\
\quad\ \ \mathtt{z}^\mathtt{O} \equiv \mathtt{S}\ \mathtt{z}_1^?)]
\end{array}
$$

Mode Inference: Disjunction

Run inference on each disjunct independently

$$\mathtt{x}^{\mathtt{I}} \equiv \mathtt{0} \;\wedge\; \mathtt{y}^{\mathtt{I}} \equiv \mathtt{z}^{\mathtt{O}}$$

$$\begin{array}{l} \mathtt{x}^{\mathtt{I}} \equiv \mathtt{S}\ \mathtt{x}_1^{?} \;\wedge \\ \mathtt{add}^{o}\ \mathtt{x}_1^{?}\ \mathtt{y}^{\mathtt{I}}\ \mathtt{z}_1^{?} \;\wedge \\ \mathtt{z}^{\mathtt{O}} \equiv \mathtt{S}\ \mathtt{z}_1^{?} \end{array}$$

Mode Inference: Unification

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

$$\frac{}{\mathtt{x}^{\mathtt{I}} \equiv \mathtt{S}\ \mathtt{x}_1^{?}} \qquad \frac{}{\mathtt{x}^{\mathtt{I}} \equiv \mathtt{S}\ \mathtt{x}_1^{0}}$$

$$\frac{}{\mathtt{z}^{0} \equiv \mathtt{S}\ \mathtt{z}_1^{?}} \qquad \frac{}{\mathtt{z}^{0} \equiv \mathtt{S}\ \mathtt{z}_1^{0}}$$

Mode Inference: Conjunction

Pick a conjunct according to the priority, propagate groundness

1. Guard
2. Assignment
3. Match
4. Call with some ground arguments
5. Unification-generator
6. Call with all free arguments

Mode Inference: Conjunction

$$
\begin{array}{l}
\hline
\mathtt{add}^o\ \mathtt{x}_1^?\ \mathtt{y}^{\mathtt{I}}\ \mathtt{z}_1^?\ \wedge \\
\mathtt{x}^{\mathtt{I}} \equiv \mathtt{S}\ \mathtt{x}_1^?\ \wedge \\
\mathtt{z}^{\mathtt{O}} \equiv \mathtt{S}\ \mathtt{z}_1^? \\
\hline
\end{array}
$$

Mode Inference: Conjunction

$$\begin{array}{l} \texttt{add}^o\ \texttt{x}_1^{?}\ \texttt{y}^{\texttt{I}}\ \texttt{z}_1^{?} \wedge \\ \texttt{x}^{\texttt{I}} \equiv \texttt{S}\ \texttt{x}_1^{?} \wedge \\ \texttt{z}^{\texttt{O}} \equiv \texttt{S}\ \texttt{z}_1^{?} \end{array}$$

$$\begin{array}{l} \texttt{x}^{\texttt{I}} \equiv \texttt{S}\ \texttt{x}_1^{\texttt{O}} \wedge \\ \texttt{add}^o\ \texttt{x}_1^{\texttt{I}}\ \texttt{y}^{\texttt{I}}\ \texttt{z}_1^{?} \wedge \\ \texttt{z}^{\texttt{O}} \equiv \texttt{S}\ \texttt{z}_1^{?} \end{array}$$

Mode Inference: Conjunction

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

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

$$\begin{array}{l}\mathtt{x}^{\mathtt{I}} \equiv \mathtt{S}\ \mathtt{x}_1^{\mathtt{O}}\ \wedge\\ \mathtt{add}^o\ \mathtt{x}_1^{\mathtt{I}}\ \mathtt{y}^{\mathtt{I}}\ \mathtt{z}_1^{\mathtt{O}}\ \wedge\\ \mathtt{z}^{\mathtt{O}} \equiv \mathtt{S}\ \mathtt{z}_1^{\mathtt{I}}\end{array}$$

Order in Conjunctions: Slow Version

```
let rec multᵒ x y z = conde [
  ( fresh (x₁ r₁)
    (x ≡ S x₁) ∧
    (addᵒ y r₁ z) ∧
    (multᵒ x₁ y r₁));
  ...]
```

```
multIIO₁ :: Nat → Nat → Stream Nat
multIIO₁ (S x₁) y    = do
  (r₁, r) ← addIOO 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 the generate-and-test behavior

Order in Conjunctions: Faster Version

```
let rec multᵒ x y z = conde [
  ( fresh (x₁ r₁)
    (x ≡ S x₁) ∧
    (addᵒ y r₁ z) ∧
    (multᵒ x₁ y r₁));
  ...]
```

```
multIIO :: Nat → Nat → Stream Nat
multIIO (S x₁) y = do
  r₁ ← multIIO x₁ y
  addIIO y r₁
...
```

Functional Conversion: Intermediate Language

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

Functional Conversion into Intermediate Language

$$
\begin{array}{rcl}
\text{Disjunction} & \rightarrow & \text{Sum}\left[\mathcal{F}_V\right] \\
\text{Conjunction} & \rightarrow & \text{Bind}\left[\left(\left[V\right], \mathcal{F}_V\right)\right] \\
\text{Relation call} & \rightarrow & R_i(\left[V\right], \left[G\right]) \\
\text{Unification} & \rightarrow & \text{Return}\left[\mathcal{T}_V\right] \\
 & | & \text{Match}_V\left(\mathcal{T}_V, \mathcal{F}_V\right) \\
 & | & \text{Guard}\left(V, V\right) \\
 & | & \text{Gen}_G
\end{array}
$$

Functional Conversion: Generators

In the untyped miniKanren it is only possible to generate all terms

Instead pass generators to functions as additional arguments
It is up to the user what generator to pass

```
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)
```

Functional Conversion: Generators

We pass a generator for every variable in rhs of a unification-generator

Generators used in calls should be passed to the parent function

In a typed version, it should be possible to automatically derive generators from types

```
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 the Target Languages

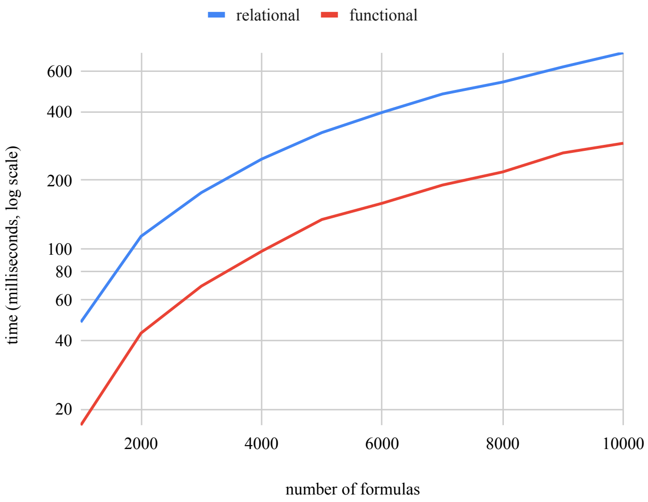
Haskell	OCaml
TemplateHaskell to generate code	Hand-crafted (not so) pretty-printer
Stream monad	Stream monad
do-notation	let*
	Taking extra care to ensure laziness

Evaluation

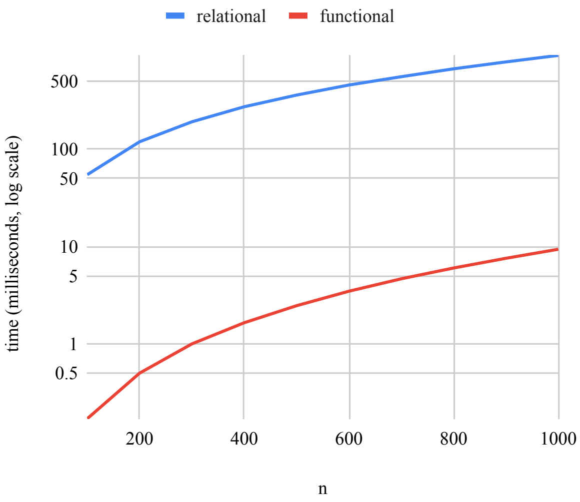
We converted relational interpreters and measured execution time

- 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

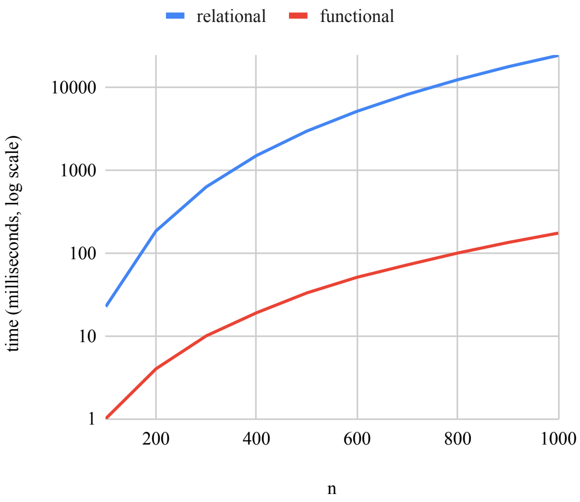
Generation of Logic Formulas: `evalo [true; false; true] q true`



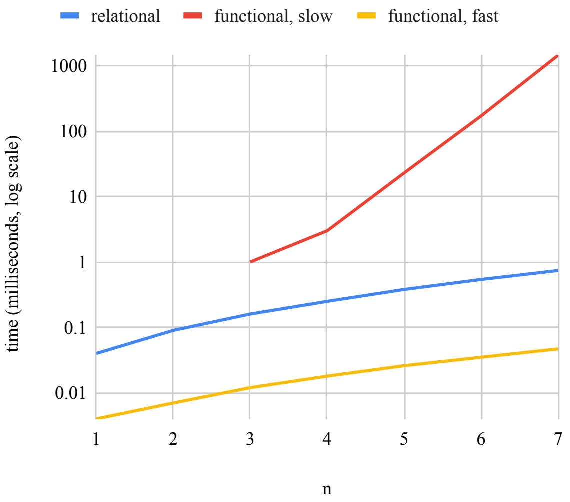
Multiplication: `mulo n 10 q`



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



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



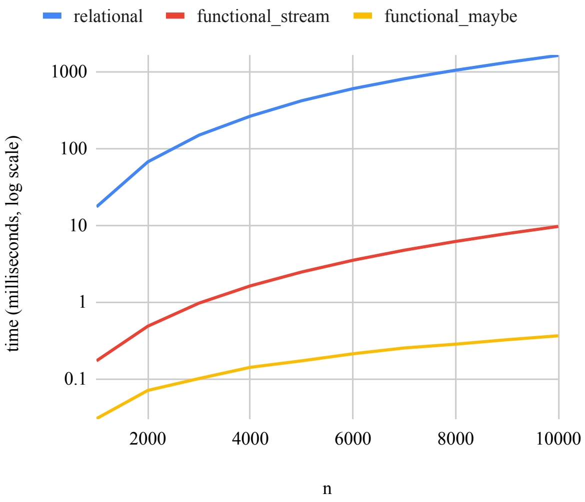
Maybe for Semi-Determinism

```
muloOII :: Nat → Nat → Stream Nat
muloOII x1 x2 =
    zero `mplus` positive
  where
    zero = do
      guard (x2 == 0)
      return 0
    positive = do
      x4 ← addoIOI x1 x2
      S <$> muloOII x1 x4
```

Maybe for Semi-Determinism

```
muloOII :: Nat → Nat → Maybe Nat
~~muloOII :: Nat → Nat → Stream Nat~~
muloOII x1 x2 =
    zero `mplus` positive
  where
    zero = do
      guard (x2 == 0)
      return 0
    positive = do
      x4 ← addoIOI x1 x2
      S <$> muloOII x1 x4
```

Maybe for Semi-Determinism: `mulo q 10 1000`



Need for Determinism Check

Simply replacing the type of monad from `Stream` to `Maybe` improves performance 10 times for relations on natural numbers

Pure (no monad) version is even faster

Use determinism check to figure out when replacing `Stream` is feasible

Need for Partial Deduction

Running a relational interpreter backwards fixes some arguments

```
run q (evalᵒ q true)
```

Augmenting functional conversion with partial deduction must be beneficial

Conclusion

Conclusion

- We presented a functional conversion scheme
- The conversion speeds up implementations considerably
- We implemented the conversion scheme in Haskell
- We found some way to order conjuncts

We are currently working on

- Determinism check
- Integration with partial deduction
- Integration into the framework of using relational interpreters for solving

Relational Sort

```
let rec sortᵒ x y = conde [
  (x ≡ [] ∧ y ≡ []);
  ( fresh (s xs xs₁)
    y ≡ s :: xs₁ ∧
    smallesto x s xs ∧
    sorto xs xs₁)]
```

```
let rec sortᵒ x y = conde [
  (x ≡ [] ∧ y ≡ []);
  ( fresh (s xs xs₁)
    y ≡ s :: xs₁ ∧
    sorto xs xs₁ ∧
    smallesto x s xs )]
```

Only good for sorting:

run q (sort^o xs q)

Only good permutation generation:

run q (sort^o q xs)

Relational Sort: Sorting

	Relation		Function
	`sorto` `smallesto`	`smallesto` `sorto`	
`[3;2;1;0]`	0.077s	0.004s	0.000s
`[4;3;2;1;0]`	timeout	0.005s	0.000s
`[31;...;0]`	timeout	1.058s	0.006s
`[262;...;0]`	timeout	timeout	1.045s

Relational Sort: Generating Permutations

	Relation		Function
	`smallesto sorto`	`sorto smallesto`	
`[0;1;2]`	0.013s	0.004s	0.004s
`[0;1;2;3]`	timeout	0.005s	0.005s
`[0;...;6]`	timeout	0.999s	0.021s
`[0;...;8]`	timeout	timeout	1.543s