



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

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 add^O x y z:

<u>Forward</u>	add^O I I O	addition
<u>Backward</u>	add^O O O I	decomposition
<u>Predicate</u>	add^O I I I	
<u>Generator</u>	add^O O O O	
	add^O I O I	subtraction
	add^O O I I	subtraction
	add^O O I O	
	add^O I O O	

Each Direction is a Function

Each Direction is a Function (Kind of)

	Functions				
<u>Forward</u>	add ^o	I	I	0	addition
	add ^o	I	0	I	subtraction
	add ^o	0	I	I	subtraction
Predicate	add ^o	I	I	I	

	Relations				
<u>Backward</u>	add^0	0	0	I	decomposition
<u>Generator</u>	add^0	0	0	0	
	add^0	0	I	0	
	add^0	I	0	0	

Relations are functions which return multiple answers

MINIKANREN Comes with an Overhead

Unifications

Occurs-check

Scheduling complexity

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 ≡ 0 ∧ y ≡ z);  
  ( fresh (x1 z1)  
    (x ≡ S x1 ∧  
      addo x1 y z1 ∧  
      z ≡ S z1) ) ]
```

```
addII0 :: Nat → Nat → Nat  
addII0 x y =  
  case x of  
    0 → y  
    S x1 → S (addII0 x1 y)
```

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)) ]
```

```
add00I :: Nat → Stream (Nat, Nat)
add00I z =
  return (0, z) 'mplus'
  case z of
    0 → Empty
    S z1 → do
      (x1, y) ← add00I 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)) ]
```

```
addI00 :: Nat → Stream (Nat, Nat)
addI00 x =
  case x of
    0 → do
      z ← genNat
      return (z, z)
    S x1 → do
      (y, z1) ← addI00 x1
      return (y, S z1)
```

```
genNat :: Stream Nat
genNat = Mature 0 (S <$> genNat)
```

Normalization

Mode analysis

Functional conversion

Normalization: Flat Term

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

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

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

Constructors inside constructors
Repeating variables

Normalization: Goal

\mathcal{K}_V^N	=	$c_1 \vee \dots \vee c_n$	$c_i \in \text{Conj}_V$	normal form
Conj_V	=	$g_1 \wedge \dots \wedge g_n$	$g_i \in \text{Base}_V$	normal conjunction
Base_V	=	$V \equiv \mathcal{FT}_V$		flat unification
		$R\ x_1 \dots x_k$	$x_j \in V, x_j - \text{distinct}$	flat call

~~Disjunctions within conjunctions~~
~~Empty disjunctions and conjunctions~~
~~Constructors as arguments of relation calls~~

Mode of a Variable

<u>Ground</u> term	no fresh variables	Cons 0 (Cons (S 0) Nil)
<u>Free</u> 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: $\text{ground} \rightarrow \text{ground}$

Mode 0: $\text{free} \rightarrow \text{ground}$

Mercury uses more complicated modes

Assign mode to every variable, make sure they are consistent

Modded Unification Types

assignment	$x^0 \equiv \mathcal{T}^I$
	$x^I \equiv y^0$
guard	$x^I \equiv \mathcal{T}^I$
match	$x^I \equiv \mathcal{T}$
generator	$x^0 \equiv \mathcal{T}$

\mathcal{T} contains both g and f variables

Mode Inference: Initialization

Input variables: $I \quad g \rightarrow g$
Output variables: $O \quad f \rightarrow g$
Other variables: $? \quad f \rightarrow ?$

```
let rec addo xI yI z0 = conde [  
  (xI  $\equiv$  0  $\wedge$  yI  $\equiv$  z0);  
  (xI  $\equiv$  S x1?  $\wedge$   
    addo x1? yI z1?  $\wedge$   
    z0  $\equiv$  S z1?)]
```

Mode Inference: Disjunction

Run inference on each disjunct independently

$$\frac{}{x^I \equiv 0 \wedge y^I \equiv z^0}$$

$$\frac{x^I \equiv S \ x_1^? \wedge \text{add}^o \ x_1^? \ y^I \ z_1^? \wedge \ z^0 \equiv S \ z_1^?}{}$$

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

$$\frac{}{x^I \equiv S \ x_1^?}$$

$$\frac{}{x^I \equiv S \ x_1^0}$$

$$\frac{}{z^0 \equiv S \ z_1^?}$$

$$\frac{}{z^0 \equiv S \ z_1^0}$$

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

$$\frac{\text{add}^o \ x_1^? \ y^I \ z_1^? \wedge \\ x^I \equiv S \ x_1^? \wedge \\ z^0 \equiv S \ z_1^?}{}$$

Mode Inference: Conjunction

$$\frac{\text{add}^o \ x_1^? \ y^I \ z_1^? \wedge}{\begin{array}{l} x^I \equiv S \ x_1^? \wedge \\ z^0 \equiv S \ z_1^? \end{array}}$$

$$\frac{\begin{array}{l} x^I \equiv S \ x_1^0 \wedge \\ \text{add}^o \ x_1^I \ y^I \ z_1^? \wedge \\ z^0 \equiv S \ z_1^? \end{array}}{}$$

Mode Inference: Conjunction

$$\frac{\text{add}^o \ x_1^? \ y^I \ z_1^? \wedge}{\begin{array}{l} x^I \equiv S \ x_1^? \wedge \\ z^0 \equiv S \ z_1^? \end{array}}$$

$$\frac{\begin{array}{l} x^I \equiv S \ x_1^0 \wedge \\ \text{add}^o \ x_1^I \ y^I \ z_1^? \wedge \\ z^0 \equiv S \ z_1^? \end{array}}{}$$

$$\frac{\begin{array}{l} x^I \equiv S \ x_1^0 \wedge \\ \text{add}^o \ x_1^I \ y^I \ z_1^0 \wedge \\ z^0 \equiv S \ z_1^I \end{array}}{}$$

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

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

Functional Conversion into Intermediate Language

Disjunction \rightarrow $\text{Sum}[\mathcal{F}_V]$

Conjunction \rightarrow $\text{Bind}[[V], \mathcal{F}_V]$

Relation call \rightarrow $R_i([V], [G])$

Unification \rightarrow $\text{Return}[\mathcal{T}_V]$
| $\text{Match}_V(\mathcal{T}_V, \mathcal{F}_V)$
| $\text{Guard}(V, V)$
| Gen_G

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 genz =
  case x of
    0 → do
      z ← genz
      return (z, z)
    S x1 → do
      (y, z1) ← addIOO x1 genz
      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_addz =
  return (0, 0) 'mplus'
do
  (z1, z) ← addIOO y gen_addz
  x ← multOOI y z1
  return (S x, z)
```

Functional Conversion into the Target Languages

HASKELL

TemplateHaskell to generate code

Stream monad

do-notation

OCAML

Hand-crafted (not so) pretty-printer

Stream monad

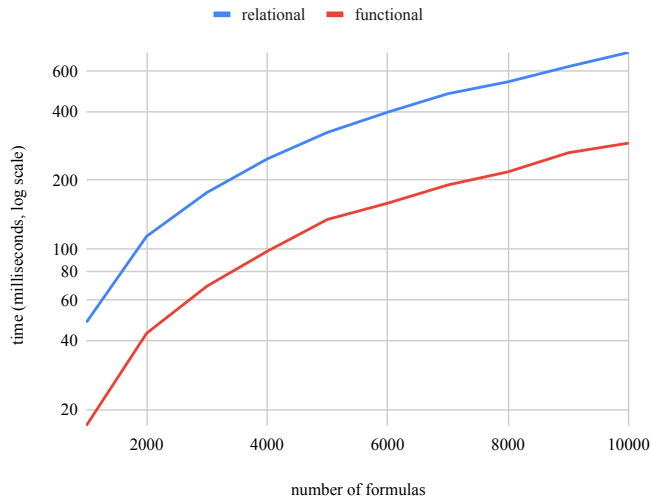
let*

Taking extra care to ensure laziness

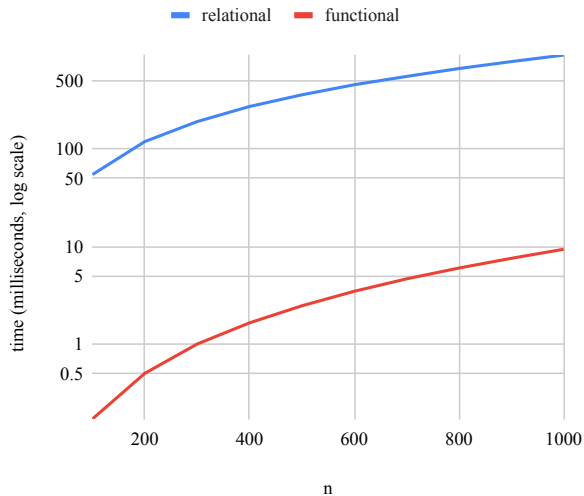
We converted relational interpreters and measured execution time

- 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

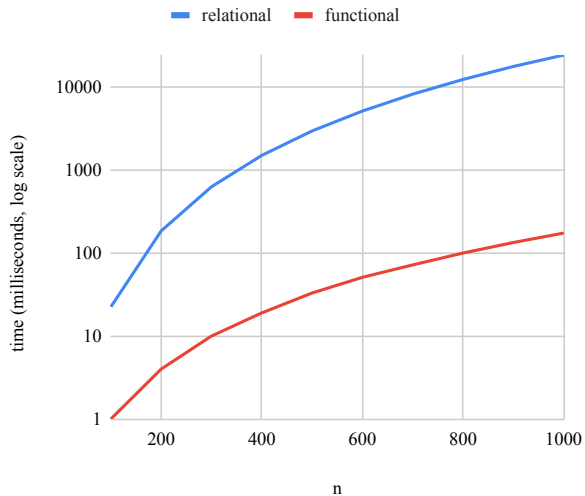
Generation of Logic Formulas: eval0 [true ; false ; true] q true



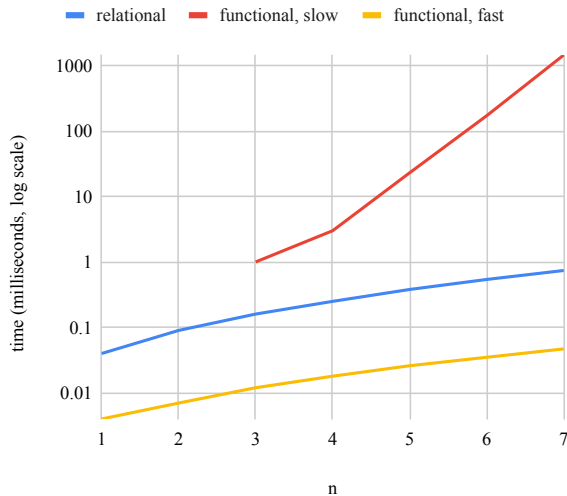
Multiplication: mulo n 10 q



Division: $\text{mul}_o (n/10) \ q \ n$



Multiplication Generation: take n (mulo 10 q r)



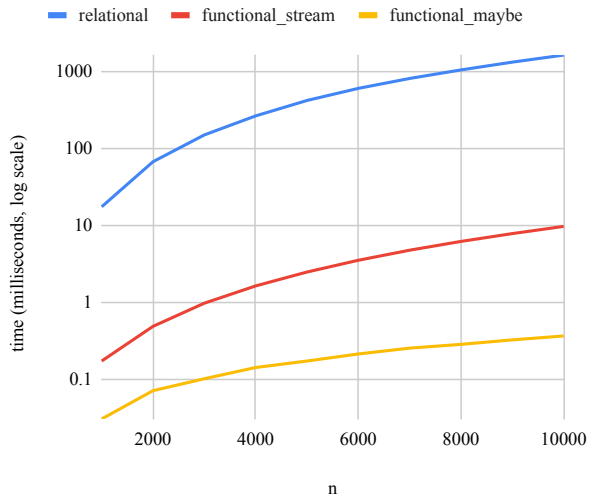
Maybe for Semi-Determinism

```
mul00II :: Nat → Nat → Stream Nat
mul00II x1 x2 =
  zero 'mplus' positive
where
  zero = do
    guard (x2 == 0)
    return 0
  positive = do
    x4 ← add0IOI x1 x2
    S <$> mul00II x1 x4
```

Maybe for Semi-Determinism

```
mul00II :: Nat → Nat → Maybe Nat
mul00II :: Nat → Nat → Stream Nat
mul00II x1 x2 =
  zero 'mplus' positive
where
  zero = do
    guard (x2 == 0)
    return 0
  positive = do
    x4 ← addoIOI x1 x2
    S <$> mul00II 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

- 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)]
```

Only good for sorting:

```
run q (sorto xs q)
```

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

Only good permutation generation:

```
run q (sorto 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