



# A Case Study in Functional Conversion and Mode Inference in miniKanren

Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research

PEPM @ POPL 2024

16.01.2024

# Program inversion

Many complicated programs are inverse of simple ones

Type inference or habitation is inverse of type checking

Program inversion: Given a program  $f$ , produce inverse program  $f^{-1}$

Given  $\text{typecheck}(\text{program}, \text{types}) = \text{true}$ , produce  $\text{typecheck}^{-1}(\text{program}, \text{true}) = \text{types}$ .

A program is a relation between its inputs and outputs

miniKanren can evaluate relations in both directions

Write a simple verifier, convert to miniKanren, get a solver

Problem: miniKanren may be slow. So convert it back!

Subset of logic programming, focus on pure relations


Extra-logical features (cuts, side-effects, search order manipulation) discouraged

Interleaving search guarantees that all answers are found

# miniKanren

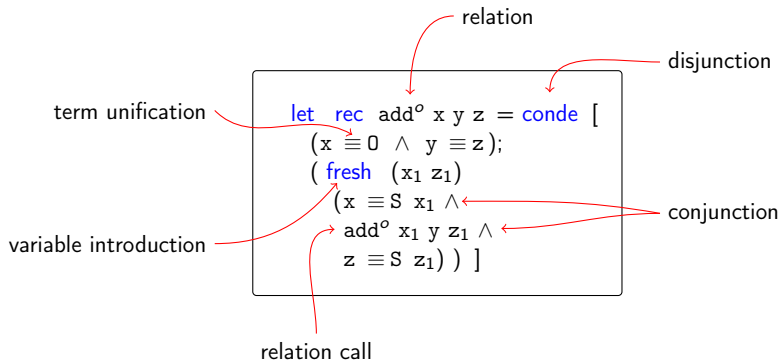
miniKanren is a simple relational language designed to be implemented as shallow embedding.

relation



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

miniKanren is a simple relational language designed to be implemented as shallow embedding.



Stream is a list-like structure, representing nondeterministic computation with interleaving search

$$[1, 2, 3] >>= f = f(1) \langle | \rangle f(2) \langle | \rangle f(3)$$

$$[1, 2, 3] \langle | \rangle [a, b, c] = [1, a, 2, b, 3, c]$$

miniKanren implementation - Stream of substitutions  
Conversion to functional language - Stream of values

## 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) ) ]
```

---

$\text{add}^o\ 1\ 2\ z = \{z \rightarrow 3\}$

---

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

---

$\text{addII0}\ 1\ 2 = 3$



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

Speed improvement: up to 3 orders of magnitude on benchmark of multiplication

## 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) ) ]
```

---

$$\langle \text{fresh } x, y \text{ in add}^o x y 2 \rangle = [$$
$$\{x \rightarrow 0, y \rightarrow 2\},$$
$$\{x \rightarrow 1, y \rightarrow 1\},$$
$$\{x \rightarrow 2, y \rightarrow 0\}$$
$$]$$

---

```
addOOI :: Nat → Stream (Nat, Nat)
addOOI z =
  return (0, z) <|>
  case z of
    0 → Empty
    S z1 → do
      (x1, y) ← addOOI z1
      return (S x1, y)
```

---

$$\text{addOOI } 2 = [(0, 2), (1, 1), (2, 0)]$$

## 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) ) ]
```

---

$\langle \text{fresh } y, z \text{ in add}^o 1 y z \rangle = [\{z \rightarrow S y\}]$

$\text{genNat} = [0, 1, 2, 3, \dots]$

$\text{addIOO } 1 = [(0, 1), (1, 2), (2, 3), \dots]$

---

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

$\text{genNat} :: \text{Stream Nat}$

$\text{genNat} =$

$(\text{return } 0) <|> (S <\$> \text{genNat})$

---

- ① Normalization
- ② Mode analysis
- ③ Functional conversion

Eliminate nested constructors and repeated variables

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

$$\begin{aligned} C(x, y) \equiv C(C(a, b), c) &\iff x \equiv C(a, b) \wedge y \equiv c \\ add^\circ(x, x, z) &\iff add^\circ(x_1, x_2, z) \wedge x_1 \equiv x_2 \end{aligned}$$

# Normalization: Goal

$\mathcal{K}_V^N$	$=$	$c_1 \vee \dots \vee c_n$	$c_i \in \mathbf{Conj}_V$	normal form
$\mathbf{Conj}_V$	$=$	$g_1 \wedge \dots \wedge g_n$	$g_i \in \mathbf{Base}_V$	normal conjunction
$\mathbf{Base}_V$	$=$	$V \equiv \mathcal{FT}_V$		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

Instantiation describes whether at a given point a variable has a known value:

<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 is a transition between instantiations, associated with each use of a variable

Mode I:  $\text{ground} \rightarrow \text{ground}$

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

Taken together, modes represent data flow.

Mercury uses more complicated modes

# Modded Unification Types

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

$\mathcal{T}$  contains at least one  $f$  variable



## 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));  
  ...]  

```

---

---

```
multIIIO1 :: Nat → Nat → Stream Nat  
multIIIO1 (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 ← multIIIO1 x1 y  
  addIII y z1 z  
multIII _ _ _ = Empty  
...  

```

---

Premature grounding of  $z_1$  leads to the generate-and-test behavior  
 $\Omega(x!)$  complexity.

## 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  
...
```

---

$O(xy)$  complexity, 10x faster than relational version

Priority:

- ① Guard
- ② Assignment
- ③ Match
- ④ Recursion, same direction
- ⑤ Call, some ground
- ⑥ Unification-generator
- ⑦ Call, all free

# Functional Conversion: Intermediate Language

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

# Functional Conversion into Intermediate Language

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

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

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

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



# 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

## Conclusion

- We presented a functional conversion scheme
- The conversion speeds up implementations considerably
- We implemented the conversion scheme in Haskell

We are currently working on

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

## Maybe for Semi-Determinism

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

## Maybe for Semi-Determinism

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

## 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



## Functional Conversion: Example

---

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

---

---

```
data Term = 0 | S Term  
addoII0 :: Term → Term → Stream Term  
addoII0 x y = msum [  
  do {  
    guard (x == 0);  
    z ← return y;  
    return z  
  },  
  do {  
    S x1 ← return x;  
    z1 ← addoII0 x1 y;  
    z ← return (S z1);  
    return z  
  }  
]
```

---