A Case Study in Functional Conversion and Mode Inference in miniKanren

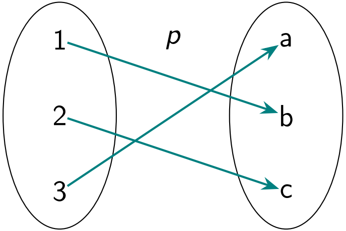
Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research

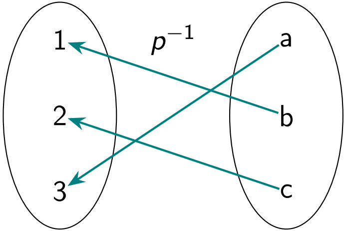
PEPM @ POPL 2024

January 16, 2024

Program Inversion



$$[\![p]\!](x) = y$$



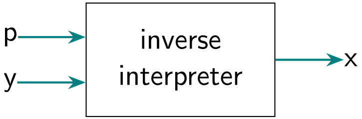
$$[\![p^{-1}]\!](y) = x$$

program synthesis: program evaluation^{-1}

type inference: type checking^{-1}

Inverse Interpreter

$$
\begin{aligned}
[\![p]\!](x) &= y \\
[\![p^{-1}]\!](y) &= x \\
[\![invInt]\!](p, y) &= x
\end{aligned}
$$



miniKanren as an Inverse Interpreter

$$
\begin{array}{l}
\textbf{let rec } \mathrm{add}^o\ x\ y\ z\ = \\
\quad (x \equiv 0 \ \wedge\ y \equiv z) \ \vee \\
\quad (\ \textbf{fresh}\ (x_1\ z_1) \\
\qquad (x \equiv S\ x_1 \ \wedge \\
\qquad\ \mathrm{add}^o\ x_1\ y\ z_1 \ \wedge \\
\qquad\ z \equiv S\ z_1)\ )
\end{array}
$$

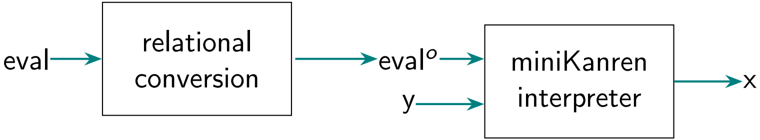
$\textbf{run}\ z\ (\mathrm{add}^o\ 0\ 1\ z)\ = \{z \mapsto 1\}$

$\textbf{run}\ x,\ y\ (\mathrm{add}^o\ x\ y\ 1)\ =$
$[\{x \mapsto 0,\ y \mapsto 1\},\ \{x \mapsto 1,\ y \mapsto 0\}]$

Relational Interpreters for Search

```
eval (Conj x y) =
  eval x && eval y
 ...
```

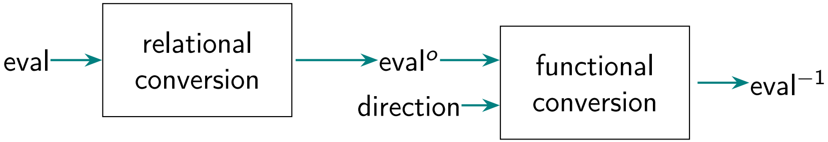
```
eval^o fm u = fresh (x y v w)
    (fm ≡ Conj x y ∧
     and^o v w u ∧
     eval^o x v ∧
     eval^o y w);
     ...
```



Relational Interpreters for Search: the Issue

It is slow

Functional Conversion to the Rescue



- Generate the same answers as MINIKANREN would
- Inputs: ground
- Outputs: ground
- Hopefully faster

MINIKANREN Syntax

relation

$$
\begin{array}{l}
\textsf{let}\ \ \textsf{rec}\ \ \mathtt{add}^o\ \mathtt{x}\ \mathtt{y}\ \mathtt{z}\ = \\
\quad (\mathtt{x} \equiv \mathtt{0}\ \wedge\ \mathtt{y} \equiv \mathtt{z})\ \vee \\
\quad (\ \textsf{fresh}\ \ (\mathtt{x}_1\ \mathtt{z}_1) \\
\qquad (\mathtt{x} \equiv \mathtt{S}\ \mathtt{x}_1\ \wedge \\
\qquad\ \mathtt{add}^o\ \mathtt{x}_1\ \mathtt{y}\ \mathtt{z}_1\ \wedge \\
\qquad\ \mathtt{z} \equiv \mathtt{S}\ \mathtt{z}_1)\ )
\end{array}
$$

miniKanren Syntax

relation

$$
\begin{array}{l}
\textsf{let rec } \mathtt{add}^o\ \mathtt{x}\ \mathtt{y}\ \mathtt{z}\ = \\
\quad (\mathtt{x} \equiv \mathtt{0} \ \wedge\ \mathtt{y} \equiv \mathtt{z}) \ \vee \\
\quad (\ \textsf{fresh}\ \ (\mathtt{x}_1\ \mathtt{z}_1) \\
\qquad (\mathtt{x} \equiv \mathtt{S}\ \mathtt{x}_1 \ \wedge \\
\qquad \mathtt{add}^o\ \mathtt{x}_1\ \mathtt{y}\ \mathtt{z}_1 \ \wedge \\
\qquad \mathtt{z} \equiv \mathtt{S}\ \mathtt{z}_1)\ )
\end{array}
$$

relation call

miniKanren Syntax

relation

term unification

```
let rec addᵒ x y z =
  (x ≡ 0 ∧ y ≡ z) ∨
  ( fresh (x₁ z₁)
     (x ≡ S x₁ ∧
      addᵒ x₁ y z₁ ∧
      z ≡ S z₁) )
```

relation call

miniKanren Syntax

relation

term unification

$$
\begin{array}{l}
\textsf{let}\ \ \textsf{rec}\ \ \mathtt{add}^o\ \mathtt{x}\ \mathtt{y}\ \mathtt{z}\ = \\
\quad (\mathtt{x}\ \equiv \mathtt{0}\ \wedge\ \mathtt{y}\ \equiv \mathtt{z})\ \vee \\
\quad (\ \textsf{fresh}\ \ (\mathtt{x}_1\ \mathtt{z}_1) \\
\quad\quad (\mathtt{x}\ \equiv \mathtt{S}\ \mathtt{x}_1\ \wedge \\
\quad\quad\ \ \mathtt{add}^o\ \mathtt{x}_1\ \mathtt{y}\ \mathtt{z}_1\ \wedge \\
\quad\quad\ \ \mathtt{z}\ \equiv \mathtt{S}\ \mathtt{z}_1)\ )
\end{array}
$$

conjunction

relation call

miniKanren Syntax

relation

disjunction

term unification

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{add}^o\ \mathtt{x}\ \mathtt{y}\ \mathtt{z} = \\
\quad (\mathtt{x} \equiv \mathtt{0} \ \wedge\ \mathtt{y} \equiv \mathtt{z}) \ \vee \\
\quad (\ \textbf{fresh}\ (\mathtt{x}_1\ \mathtt{z}_1) \\
\qquad (\mathtt{x} \equiv \mathtt{S}\ \mathtt{x}_1 \ \wedge \\
\qquad \mathtt{add}^o\ \mathtt{x}_1\ \mathtt{y}\ \mathtt{z}_1 \ \wedge \\
\qquad \mathtt{z} \equiv \mathtt{S}\ \mathtt{z}_1)\ )
\end{array}
$$

conjunction

relation call

miniKanren Syntax

relation

disjunction

term unification

```
let rec addᵒ x y z =
  (x ≡ 0 ∧ y ≡ z) ∨
  ( fresh (x₁ z₁)
     (x ≡ S x₁ ∧
      addᵒ x₁ y z₁ ∧
      z ≡ S z₁) )
```

conjunction

variable introduction

relation call

Example: Addition in the Forward Direction

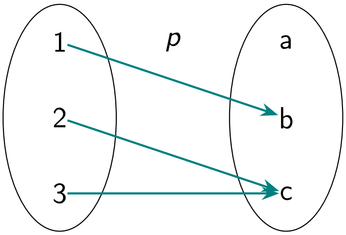
```
let rec addᵒ x y z =
  (x ≡ O ∧ y ≡ z) ∨
  ( fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁) )
```

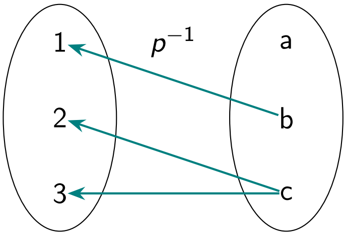
$\text{add}^{\circ}\ 1\ 2\ z = \{z \mapsto 3\}$

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

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

Nondeterministic Inversion





$$\texttt{run x, y (add}^o\texttt{ x y 1)} = [\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}]$$

Stream

- Represents nondeterminism
- List-like
- Interleaving search

$$[1,2,3] >>= f = f(1) <|> f(2) <|> f(3)$$

$$[1,2,3] <|> [a,b,c] = [1,a,2,b,3,c]$$

- miniKanren: `Stream` of substitutions
- Functional conversion: `Stream` of values

Addition in the Backward Direction: Nondeterminism

```
let rec addᵒ x y z =
  (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) <|>
  case z of
    0 → Empty
    S z₁ → do
      (x₁, y) ← addOOI z₁
      return (S x₁, y)
```

$\texttt{run}\ x,\ y\ (\text{add}^o\ x\ y\ 1) = [\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}]$

$\texttt{addOOI}\ x\ y\ 1 = [(0,\ 1),\ (1,\ 0)]$

Free Variables in Answers: Generators

```
let rec addᵒ x y z =
  (x ≡ O ∧ 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 =
   (return O) <|> (S <$> genNat)
```

$\text{run } y,\ z\ (\text{add}^o\ 1\ y\ z) = \{z \mapsto S\ y\}$

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

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

Conversion Scheme

1. Normalization
2. Mode analysis
3. Functional conversion

Normalization: Flat Term

Eliminate nested constructors and repeated variables

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

$$\begin{aligned} C\,(x, y) \equiv C\,(C\,(v, u)\,, w) \quad &\Longleftrightarrow \quad x \equiv C\,(v, u) \wedge y \equiv w \\ \mathit{add}^{\circ}(x, x, z) \quad &\Longleftrightarrow \quad \mathit{add}^{\circ}\,(x, y, z) \wedge x \equiv y \end{aligned}$$

Normalization: Goal

Eliminate disjunctions within conjunctions

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

Mode of a Variable

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

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

Mode `I`: `ground` $\to$ `ground`

Mode `O`: `free` $\to$ `ground`

Taken together, modes represent data flow.

Mercury uses more complicated modes

Modded Unification Types

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

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

Order in Conjunctions

```
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
 ...
```

$O(xy)$

10x faster

```
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
 ...
```

generate-and-test

$\Omega(x!)$

Mode Inference: Ordering Heuristic

Priority:

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

Functional Conversion: Intermediate Language

$$
\begin{array}{lcll}
\mathcal{F}_V & = & \mathcal{F}_V <|> \cdots <|> \mathcal{F}_V & \text{interleaving} \\
 & | & (\overline{V} \leftarrow \mathcal{F}_V)^* & \text{monadic bind on streams} \\
 & | & \text{return}\, \mathcal{T}_V^* & \text{return a tuple of terms} \\
 & | & V == \mathcal{T}_V & \text{equality check} \\
 & | & case\ V\ of\ \mathcal{T}_V \rightarrow \mathcal{F}_V & \text{match a variable against a pattern} \\
 & | & R_i\ \overline{V}\ \overline{Gen_G} & \text{function call} \\
 & | & \text{Gen}_G & \text{generator}
\end{array}
$$

Functional Conversion into Intermediate Language

$$
\begin{array}{rcl}
\text{Disjunction} & \rightarrow & <\,|\,> \mathcal{F}_V^* \\[2ex]
\text{Conjunction} & \rightarrow & \text{Bind}\,(V^*, \mathcal{F}_V)^* \\[2ex]
\text{Relation call} & \rightarrow & R_i(V^*, G^*) \\[2ex]
\text{Unification} & \rightarrow & \text{return}\ \mathcal{T}_V^* \\
 & | & \text{Match}_V\,(\mathcal{T}_V, \mathcal{F}_V) \\
 & | & \text{Guard}\,(V, \mathcal{T}_V) \\
 & | & \text{Gen}_G
\end{array}
$$

Functional Conversion: Generators

```
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

```
multOIO :: Nat → Stream Nat → Stream Nat
multOIO y gen_add_z =
  return (O, O) <|>
  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

Relational Sort

$$
\begin{array}{l}
\textsf{let rec } \texttt{sort}^o\ \texttt{x}\ \texttt{y} = \\
\quad (\texttt{x} \equiv [] \ \wedge\ \texttt{y} \equiv []) \ \vee \\
\quad (\ \textsf{fresh}\ (\texttt{s}\ \texttt{xs}\ \texttt{xs}_1) \\
\qquad \texttt{y} \equiv \texttt{s} :: \texttt{xs}_1 \ \wedge \\
\qquad \textbf{smallest}^o\ \textbf{x}\ \textbf{s}\ \textbf{xs} \ \wedge \\
\qquad \textbf{sort}^o\ \textbf{xs}\ \textbf{xs}_1)
\end{array}
$$

- ✓ sorting
- ⏲ permutations

$$
\begin{array}{l}
\textsf{let rec } \texttt{sort}^o\ \texttt{x}\ \texttt{y} = \\
\quad (\texttt{x} \equiv [] \ \wedge\ \texttt{y} \equiv []) \ \vee \\
\quad (\ \textsf{fresh}\ (\texttt{s}\ \texttt{xs}\ \texttt{xs}_1) \\
\qquad \texttt{y} \equiv \texttt{s} :: \texttt{xs}_1 \ \wedge \\
\qquad \textbf{sort}^o\ \textbf{xs}\ \textbf{xs}_1 \ \wedge \\
\qquad \textbf{smallest}^o\ \textbf{x}\ \textbf{s}\ \textbf{xs})
\end{array}
$$

- ⏲ sorting
- ✓ permutations

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

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

```haskell
muloOII :: Nat → Nat → Stream Nat
muloOII x1 x2 =
    zero <|> 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 <|> positive
  where
    zero = do
      guard (x2 == 0)
      return 0
    positive = do
      x4 ← addoIOI x1 x2
      S <$> muloOII 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 =
  (x ≡ 0 ∧ y ≡ z) ∨
  ( fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁) )
```

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