A Case Study in Functional Conversion and Mode Inference in miniKanren

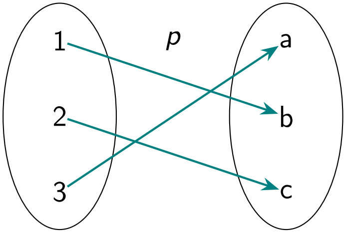
Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research

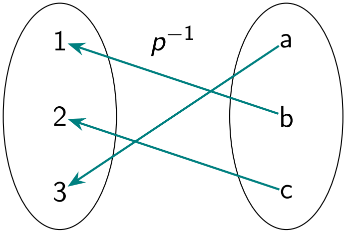
PEPM @ POPL 2024

January 16, 2024

Program Inversion



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



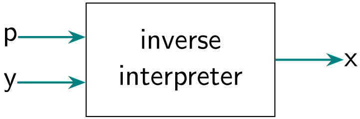
$$[\![p^{-1}]\!](y) = x$$

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

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

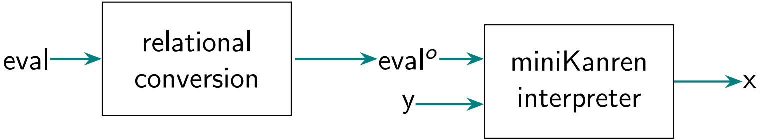
$$\textbf{run } z\ (\mathrm{add}^o\ 0\ 1\ z) = \{z \mapsto 1\}$$

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

Relational Interpreters for Search

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

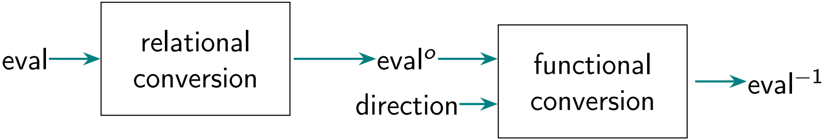
```
eval^o fm u = fresh (x y v w)
    (fm ≡ Conj x y ∧
     and^o v w u ∧
     eval^o x v ∧
     eval^o y w);
     ...
```



Relational Interpreters for Search: the Issue

It is slow

Functional Conversion to the Rescue



- Generate the same answers as miniKanren would
- Hopefully faster

miniKanren Syntax

relation

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

MINIKANREN Syntax

relation

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

relation call

miniKanren Syntax

relation

term unification

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

relation call

miniKanren Syntax

relation

term unification

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

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

relation call

relation

disjunction

term unification

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

conjunction

variable introduction

relation call

Example: Addition in the Forward Direction

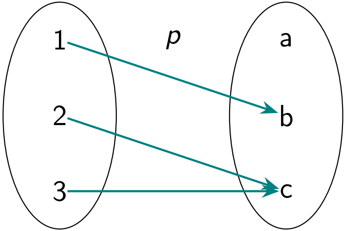
```
let rec addᵒ x y z =
  (x ≡ 0 ∧ y ≡ z) ∨
  ( fresh (x₁ z₁)
    (x ≡ S x₁ ∧
     addᵒ x₁ y z₁ ∧
     z ≡ S z₁) )
```

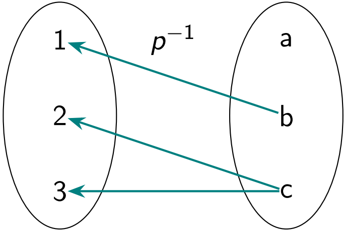
$$\text{add}^\circ\ 0\ 1\ z = \{z \mapsto 1\}$$

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

$$\text{addIIO}\ 0\ 1 = 1$$

Noninjective $\leftrightarrow$ Nondeterministic





$$\text{add}^{\circ}\ x\ y\ 1 = [\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}]$$

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)
```

$\text{add}^{\circ}\ x\ y\ 1 = [\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}]$

addOOI 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) )
```

$\text{add}^\circ\ 1\ y\ z = \{z \mapsto S\ y\}$

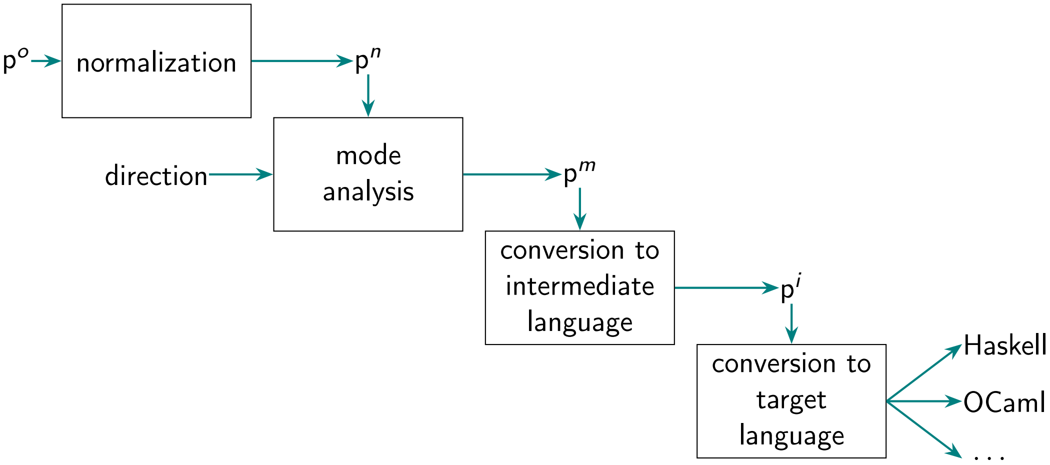
genNat = $[0,\ 1,\ 2, \ldots]$

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

```
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)
```

Conversion Scheme



Normalization: Flat Term

Eliminate nested constructors and repeated variables

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

$$\begin{aligned} C\left(x, y\right) \equiv C\left(C\left(v, u\right), w\right) &\iff x \equiv C\left(v, u\right) \land y \equiv w \\ add^{\circ}(x, x, z) &\iff add^{\circ}\left(x, y, z\right) \land x \equiv y \end{aligned}$$

Normalization: Goal

Eliminate disjunctions within conjunctions

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

Modes — Data Flow

Ground term	`S (S O)`
Free variable	`x`

Once a variable is ground, it stays ground

$$\text{Mode} : \text{Inst} \mapsto \text{Inst}$$

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

Modded Unification Types

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

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

Ordering Heuristic: Example

```
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

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

Functional Conversion into Intermediate Language

$$
\begin{array}{rcl}
\text{Disjunction} & \rightarrow & \mathcal{F} < | > \ldots < | > \mathcal{F} \\
\\
\text{Conjunction} & \rightarrow & \left(\overline{V} \leftarrow \mathcal{F}\right)^{*} \\
\\
\text{Relation call} & \rightarrow & R \ \overline{V} \ \overline{Gen_G} \\
\\
\text{Unification} & \rightarrow & \text{return } \mathcal{T}^{*} \\
& | & case \ V \ of \ \ \mathcal{T} \rightarrow \mathcal{F} \\
& | & V \equiv \mathcal{T} \\
& | & \text{Gen}_G
\end{array}
$$

Functional Conversion: Generators

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

$$add^{\circ}\ 1\ y\ z = [\{z \rightarrow S\ y\}]$$

Functional Conversion: Generators

```
multOIO :: Nat → Stream Nat → Stream Nat
multOIO y gen_add_y =
  return (0, 0) <|>
  do
    (z_1, z) ← addIOO y gen_add_y
    x ← muloOII y z_1
    return (S x, z)
```

Relational Sort

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

- ✓ sorting
- 🕒 permutations

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

- 🕒 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

Possible speedups

- Determinism
- Partial deduction

Maybe for Semi-Determinism

```
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

```haskell
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
```

10x faster

Need for Partial Deduction

```
run q (evalᵒ q true)

run q (eval_trueᵒ q)
```

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

Functional Conversion: Example

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

```
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
    }
]
```