



A Case Study in Functional Conversion and Mode Inference in miniKanren

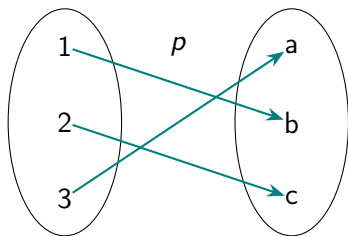
Kate Verbitskaia, Igor Engel, Daniil Berezun

JetBrains Research

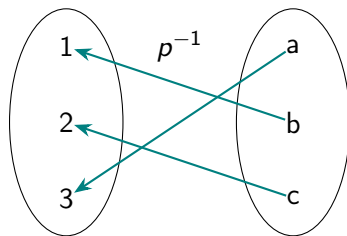
PEPM @ POPL 2024

January 16, 2024

Program Inversion



$$\llbracket p \rrbracket(x) = y$$



$$\llbracket p^{-1} \rrbracket(y) = x$$

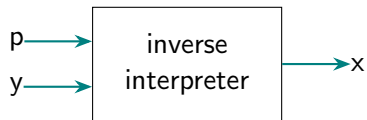
program synthesis: program evaluation⁻¹
type inference: type checking⁻¹

Inverse Interpreter

$$\llbracket p \rrbracket(x) = y$$

$$\llbracket p^{-1} \rrbracket(y) = x$$

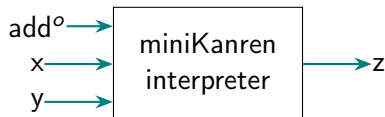
$$\llbracket invInt \rrbracket(p, y) = x$$



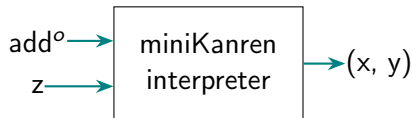
miniKanren as an Inverse Interpreter

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

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



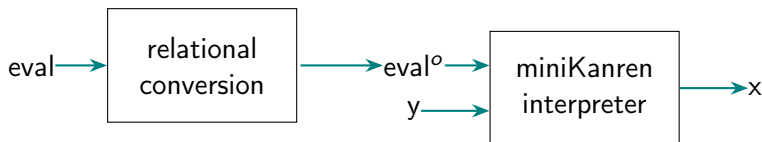
`run` x, y (add^o x y 1) =
[$\{x \mapsto 0, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 0\}$]



Relational Interpreters for Search

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

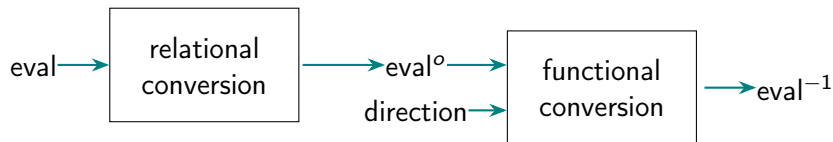
```
evalo fm u = fresh (x y v w)  
  (fm ≡ Conj x y ∧  
   ando v w u ∧  
   evalo x v ∧  
   evalo y w);  
...
```



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

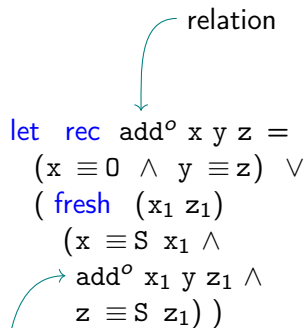
MINIKANREN Syntax

relation



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


MINIKANREN Syntax



```
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

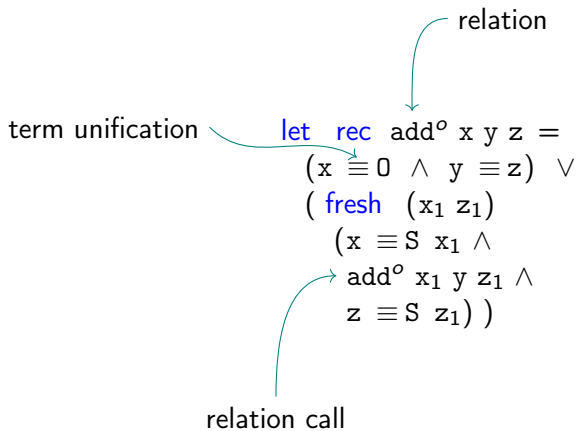
MINIKANREN Syntax

term unification

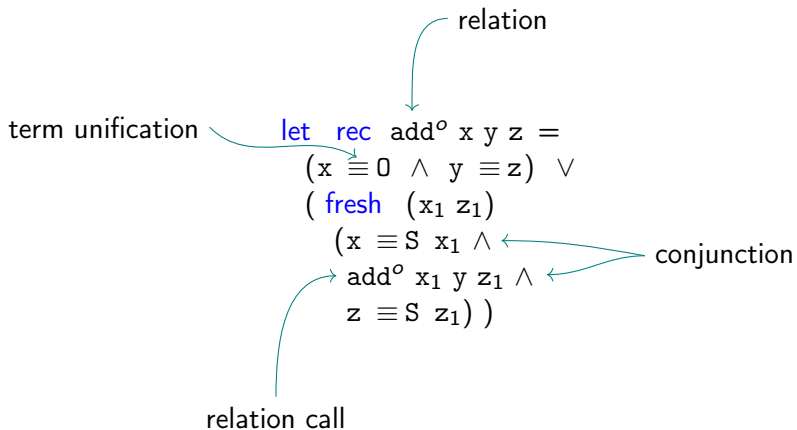
relation

let rec add^o x y z =
 (x \equiv 0 \wedge y \equiv z) \vee
 (fresh (x₁ z₁)
 (x \equiv S x₁ \wedge
 add^o x₁ y z₁ \wedge
 z \equiv S z₁))

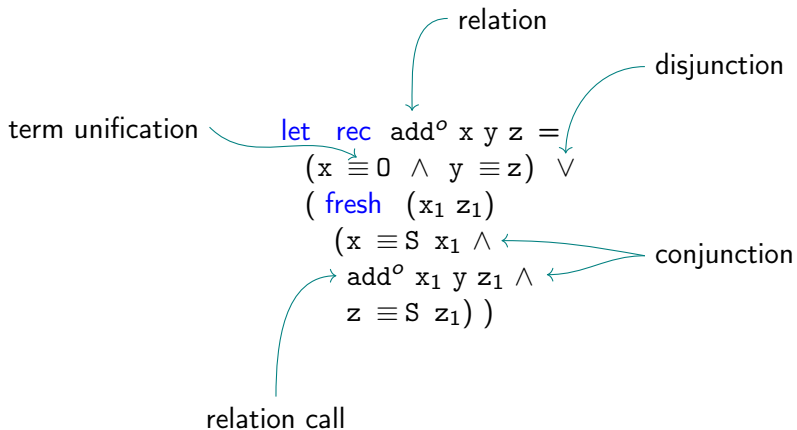
relation call

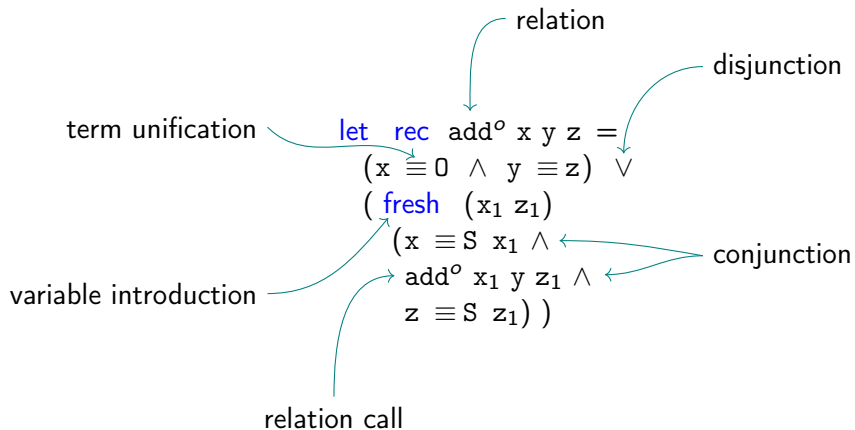


MINIKANREN Syntax



MINIKANREN Syntax





Example: Addition in the Forward Direction

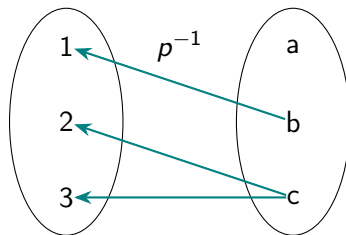
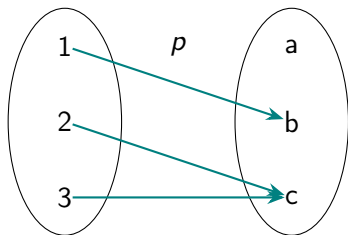
```
let rec addo x y z =  
  (x ≡ 0 ∧ y ≡ z) ∨  
  (fresh (x1 z1)  
   (x ≡ S x1 ∧  
    addo x1 y z1 ∧  
    z ≡ S z1))
```

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

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

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

Nondeterministic Inversion



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

- 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))
```

```
add00I :: Nat → Stream (Nat, Nat)  
add00I z =  
  return (0, z) <|>  
  case z of  
    0 → Empty  
    S z1 → do  
      (x1, y) ← add00I z1  
      return (S x1, y)
```

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

$\text{add00I}\ 1 = [(0,1), (1,0)]$

Free Variables in Answers: Generators

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

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

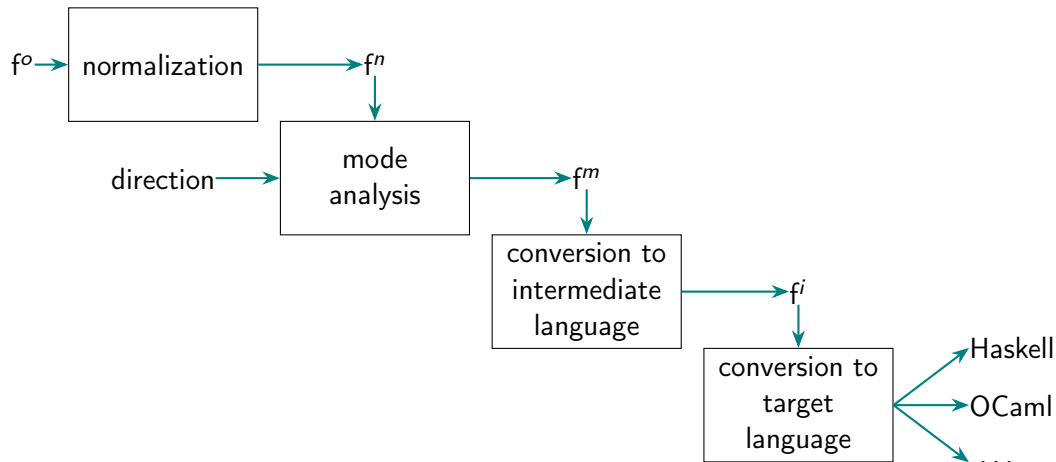
$\text{genNat} = [0, 1, 2, \dots]$

$\text{addI00 } 1 = [(0,1), (1,2), (2,3), \dots]$

```
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 =  
  (return 0) <|> (S <$> genNat)
```

Conversion Scheme



Normalization: Flat Term

Eliminate nested constructors and repeated variables

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

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

Normalization: Goal

Eliminate disjunctions within conjunctions

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

Ground term s (s 0)

Free variable x

Once a variable is ground, it stays ground

Mode : Inst \mapsto Inst

Mode I: ground \rightarrow ground

Mode 0: free \rightarrow ground

Modded Unification Types

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

Order in Conjunctions

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

```
multIIIO :: Nat → Nat → Stream Nat  
multIIIO (S x1) y = do  
  r1 ← multIIIO x1 y  
  addIIIO y r1  
  ...
```

$O(xy)$ 10x faster

```
multIIIO1 :: Nat → Nat → Stream Nat  
multIIIO1 (S x1) y = do  
  (r1, r) ← addIOO y  
  multIII x1 y r1      generate-and-test  
  return r  
  ...  
multIII :: Nat → Nat → Nat → Stream ()  
multIII (S x1) y z = do  
  z1 ← multIIIO1 x1 y  
  addIII y z1 z  
multIII _ _ _ = Empty  
  ...
```

$\Omega(x!)$

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));  
  ...]  
...
```

```
multII0 :: Nat → Nat → Stream Nat  
multII0 (S x1) y = do  
  r1 ← multII0 x1 y  
  addII0 y r1  
...  
...
```

Functional Conversion: Intermediate Language

\mathcal{F}	$=$	$\mathcal{F} < > \dots < > \mathcal{F}$	interleaving
		$(\overline{V} \leftarrow \mathcal{F})^*$	monadic bind on streams
		return \mathcal{T}^*	return a tuple of terms
		$V == \mathcal{T}$	equality check
		<i>case</i> V <i>of</i> $\mathcal{T} \rightarrow \mathcal{F}$	match a variable against a pattern
		$R \ \overline{V} \ \overline{Gen}_G$	function call
		Gen _G	generator

Functional Conversion into Intermediate Language

Disjunction $\rightarrow \mathcal{F} < | > \dots < | > \mathcal{F}$

Conjunction $\rightarrow (\overline{V} \leftarrow \mathcal{F})^*$

Relation call $\rightarrow R \ \overline{V} \ \overline{Gen_G}$

Unification \rightarrow **return** \mathcal{T}^*
| *case* V *of* $\mathcal{T} \rightarrow \mathcal{F}$
| $V \equiv \mathcal{T}$
| **Gen_G**

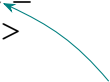
Functional Conversion: Generators

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

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

Functional Conversion: Generators

```
mult0IO :: Nat → Stream Nat → Stream Nat
mult0IO y gen_add_y =
  return (0, 0) <|>
  do
    (z1, z) ← addIO0 y gen_add_y
    x ← mulo0II y z1
    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

```
muloII0 :: Nat → Nat → Stream Nat
muloII0 x y =
  zero <|> positive
  where
    zero = do
      guard (x == 0)
      return 0
    positive = do
      (S x1) ← return x
      z1 ← muloII0 x1 y
      z ← addoII0 z1 y
      return z
```

Maybe for Semi-Determinism

```
muloII0 :: Nat → Nat → Maybe Nat
muloII0 :: Nat → Nat → Stream Nat
muloII0 x y =
  zero <|> positive
  where
    zero = do
      guard (x == 0)
      return 0
    positive = do
      (S x1) ← return x
      z1 ← muloII0 x1 y
      z ← addoII0 z1 y
      return z
```



10x
faster

Need for Partial Deduction

```
run q (evalo q true)
```

```
run q (eval_trueo q)
```

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

```
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  
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  
  }  
]
```
