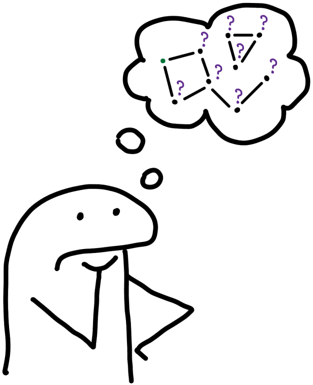
Enabling Relational Programming through Specialization

Ekaterina Verbitskaia

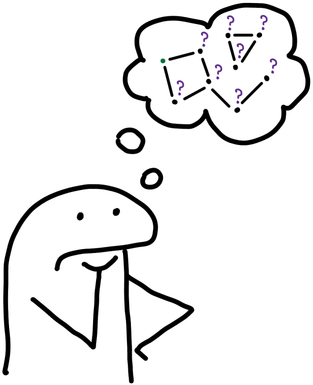
JetBrains Research, Programming Lanuages and Program Analysis Lab
Constructor University, Bremen

June 5, 2024

Find Reachable Vertices

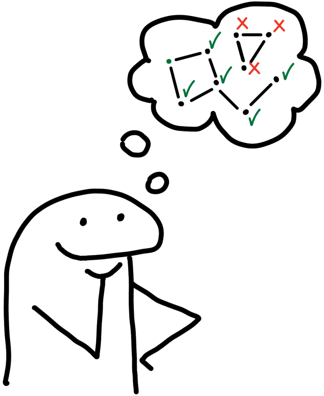


Find Reachable Vertices



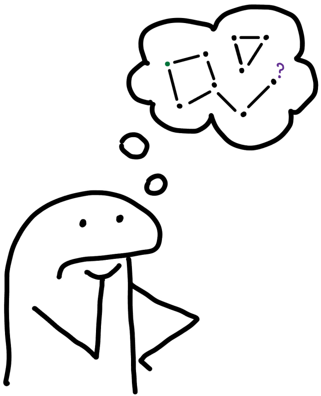
```
procedure DFS(G, v) =
  S.push(v)
  while !S.empty do
    v = S.pop()
    if (!seen[v]) then
      seen.add(v)
      for (_, w) in G.edges(v) do
        S.push(w)
```

Find Reachable Vertices

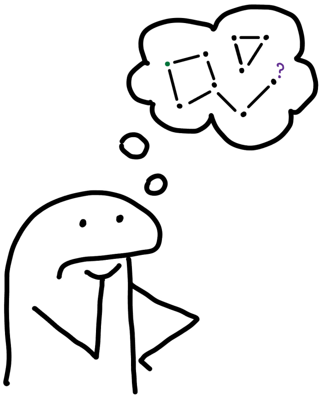


```
procedure DFS(G, v) =
  S.push(v)
  while !S.empty do
    v = S.pop()
    if (!seen[v]) then
      seen.add(v)
      for (_, w) in G.edges(v) do
        S.push(w)
```

Find a Path

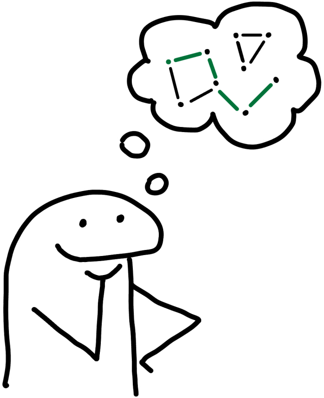


Find a Path



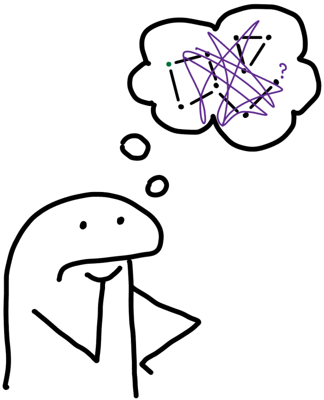

```
procedure DFS(G, v, u) =
  S.push(v, [])
  while !S.empty do
    if (v == u) then return path
    (v, path) = S.pop()
    if (!seen[v]) then
      seen.add(v)
      for (_, w) in G.edges(v) do
        S.push(w, path.add(w))
    return none
```

Find a Path

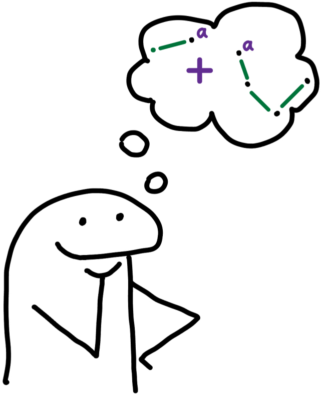


```
procedure DFS(G, v, u) =
  S.push(v)
  path.push(v)
  while !S.empty do
    if (v == u) then return path
    v = S.pop()
    if (!seen[v]) then
      seen.add(v)
      for (_, w) in G.edges(v) do
        S.push(w)
    path.pop()
    return none
```

Find All Paths

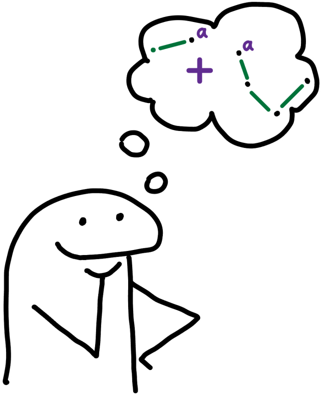


What is a Path?



$$
\begin{array}{rcl}
path(v, u) & = & [v], \mathit{if}\ v = u \\
 & \mid & \exists w : edge(v, w) \wedge path(w, u)
\end{array}
$$

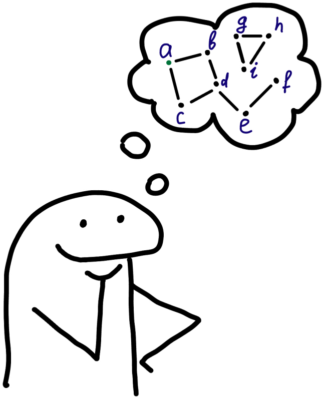
What is a Path?



$$\begin{array}{rcl} path(v, u) & = & [v], \textit{if } v = u \\ & \mid & \exists w : edge(v, w) \wedge path(w, u) \end{array}$$

```
path(V, V, [V]).
path(V, U, [V|P]) :- edge(V, W), path(W, U, P).
       |   |     |
 start   end   path                          % Prolog
```

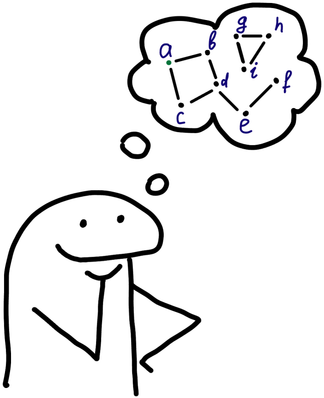
Logic Programming: Querying Paths




```
edge(a, b).
edge(a, c).
...
edge(i, g).

path(V, V, [V]).
path(V, U, [V|P]) :- edge(V, W), path(W, U, P).
```

```
? path(a, f, _).                      ? path(a, h, _).
true       <- predicate ->            false

? path(a, f, P).

P = [a, b, d, e, f]
P = [a, c, d, e, f]          <- nondeterminism
false
```

Logic Programming: Querying Reachable Vertices




```
edge(a, b).
edge(a, c).
...
edge(i, g).

path(V, V, [V]).
path(V, U, [V|P]) :- edge(V, W), path(W, U, P).
```

```
? path(a, X, _).
             |
X = a        ignore
X = b
X = d
X = e
X = f
false
```

Relational Programming in miniKanren


```
(define (patho v u p)                          ; Racket
  (fresh (w p1)
    (conde
      [(== v u) (== p `(,v))]
      [(edge v w)
       (patho w u p1)
       (== p (cons v p1))])))

(run* (q) (patho 'a 'e q))
```

```
let rec path^o v u p =                         (* OCaml *)
  (v ≡ u ∧ p ≡ [v]) ∨
  (fresh (w p_1)
    (edge^o v w ∧
      path^o w_1 u p_1 ∧
      p ≡ v : p_1) )
```

miniKanren

The Anatomy of miniKanren

relation

$$
\begin{array}{l}
\texttt{let rec path}^o\ \texttt{v u p =}\\
\quad (\texttt{v} \equiv \texttt{u} \ \wedge\ \texttt{p} \equiv \texttt{[]})\ \vee\\
\quad (\texttt{fresh (w p}_1\texttt{)}\\
\qquad (\texttt{edge}^o\ \texttt{v w} \ \wedge\\
\qquad\ \texttt{path}^o\ \texttt{w}_1\ \texttt{u p}_1 \ \wedge\\
\qquad\ \texttt{p} \equiv \texttt{v : p}_1\texttt{))}
\end{array}
$$

The Anatomy of MINIKANREN

relation

$$
\begin{array}{l}
\texttt{let rec path}^{o}\ \texttt{v u p =} \\
\quad \texttt{(v} \equiv \texttt{u} \ \wedge \ \texttt{p} \equiv \texttt{[])} \ \vee \\
\quad \texttt{(fresh (w p}_1\texttt{)} \\
\quad\quad \texttt{(edge}^{o}\ \texttt{v w} \ \wedge \\
\quad\quad \texttt{path}^{o}\ \texttt{w}_1\ \texttt{u p}_1 \ \wedge \\
\quad\quad \texttt{p} \equiv \texttt{v : p}_1\texttt{))}
\end{array}
$$

relation call

The Anatomy of miniKanren

term unification

relation

```
let rec pathᵒ v u p =
  (v ≡ u  ∧  p ≡ [])  ∨
  (fresh (w p₁)
    (edgeᵒ v w ∧
     pathᵒ w₁ u p₁ ∧
     p ≡ v : p₁) )
```

relation call

The Anatomy of miniKanren

term unification

relation

$$
\begin{array}{l}
\texttt{let rec path}^o\ \texttt{v u p =} \\
\quad (\texttt{v} \equiv \texttt{u} \ \wedge\ \texttt{p} \equiv \texttt{[]}) \ \vee \\
\quad (\texttt{fresh (w p}_1\texttt{)} \\
\quad\quad (\texttt{edge}^o\ \texttt{v w} \ \wedge \\
\quad\quad\ \ \texttt{path}^o\ \texttt{w}_1\ \texttt{u p}_1 \ \wedge \\
\quad\quad\ \ \texttt{p} \equiv \texttt{v : p}_1\texttt{))}
\end{array}
$$

relation call

conjunction

The Anatomy of miniKanren

term unification

relation

```
let rec path^o v u p =
  (v ≡ u  ∧  p ≡ [])  ∨
  (fresh (w p_1)
    (edge^o v w ∧
     path^o w_1 u p_1 ∧
     p ≡ v : p_1) )
```

disjunction

relation call

conjunction

term unification

relation

```
let rec path^o v u p =
  (v ≡ u  ∧  p ≡ [])  ∨
  (fresh (w p_1)
    (edge^o v w ∧
     path^o w_1 u p_1 ∧
     p ≡ v : p_1) )
```

variable introduction

disjunction

relation call

conjunction

The Semantics of miniKanren: Unification

$$f(x, A, g(z), y) \equiv f(h(A, y), A, g(B), y)$$

$$\{x \mapsto h(A, y), z \mapsto B\}$$

$(\equiv)$ :: Term $\rightarrow$ Term $\rightarrow$ Subst $\rightarrow$ Subst

$$f(x, x) \equiv f(h(A, y), g(B))$$

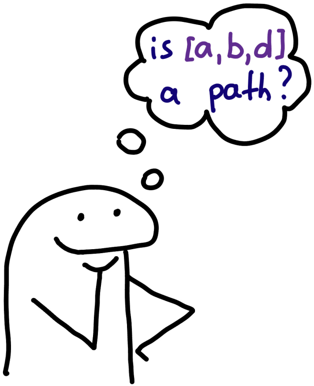
$$\varnothing$$

The Semantics of miniKanren: Stream

```
data Stream a = Empty | Mature a (Stream a)

instance Alternative Stream where
  empty = Empty

  (Mature h tl) <|> y = Mature h (y <|> tl)
  Empty         <|> y = y

instance Monad Stream where
  Empty >>= _ = mzero
  Mature x xs >>= g = mplus (g x) (xs >>= g)

( ∧ ) = (>>=)
( ∨ ) = (<|>)
```

Solvers from Verifiers

Verifiers




```
let rec is_path path =
  match path with
  | [], [_] → true
  | u :: v :: t →
      if edge u v
      then is_path (v :: t)
      else false
```


```
let rec dfs ... =
  push stack ...
  while ...
    ... pop stack
    if seen ...
      ... dfs ...
    ... return ...
```

Solvers from Verifiers


```
let rec is_path path =
  match path with
  | [], [_] → true
  | u :: v :: t →
      if edge u v
      then is_path (v :: t)
      else false
```

relational conversion

```
let rec path^o v u p =
  (v ≡ u ∧ p ≡ [v]) ∨
  (fresh (w p1)
    (edge^o v w ∧
     path^o w1 u p1 ∧
     p ≡ v : p1) )
```

verifier

solver

Solvers from Verifiers: Examples


```
let rec eval^o st fm u =
  fresh (x y v w z)
    (fm ≡ Conj x y  ∧
     eval^o st x v  ∧
     eval^o st y w  ∧
     and^o v w u)  ∨
     ...
```

evaluation

program generation

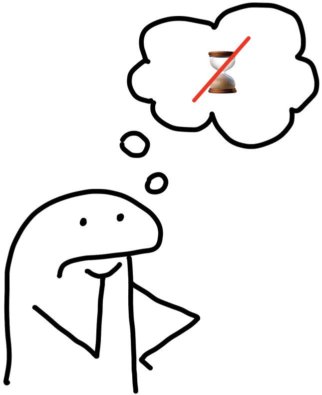
```
let rec type^o e t =
  (fresh (x y)
    (e ≡ Int x  ∧  t = TInt)  ∨
    (e ≡ Plus x y  ∧
     type^o x Int  ∧
     type^o y Int  ∧
     t ≡ Int))  ∨
     ...
```

type checking

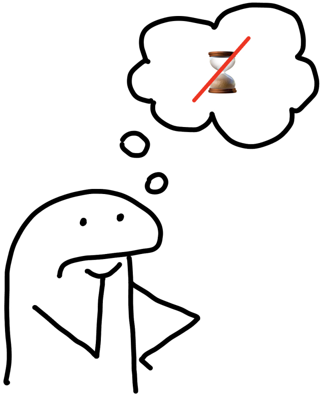
type inference

type inhabitance

The Issue



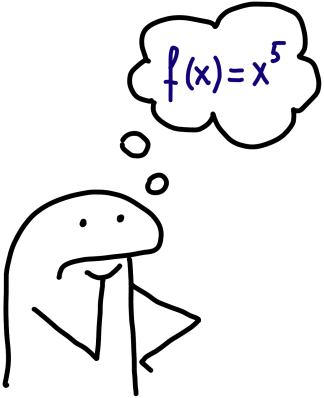
The Issue



- Unifications are expensive
- The order of conjunctions is finicky
- But! We know something that can help

Specialization

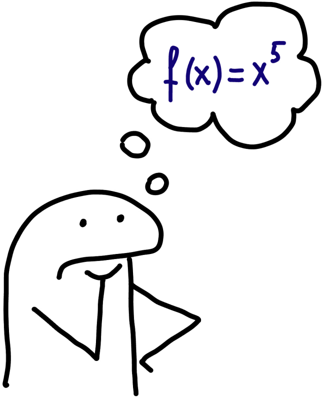
Specialization




```
let rec power base exp =
  if exp == 0
  then 1
  else base * power base (exp - 1)
```

```
let rec power_5 base =                        (* power base 5 *)
  base * base * base * base * base
```

Specialization



$$program : I_{static} \times I_{dynamic} \rightarrow O$$

$$program_{I_{static}} : I_{dynamic} \rightarrow O$$

Same outputs for the same inputs

Specialization for miniKanren

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x  ∧  notᵒ a r ∧ evalᵒ x s a ∨
  ...
```

Specialization for miniKanren

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x  ∧  notᵒ a r ∧ evalᵒ x s a ∨
  ...
```

```
evalᵒ fm s true
```

known argument

Specialization for miniKanren

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x  ∧  notᵒ a r ∧ evalᵒ x s a ∨
  ...
```

known argument

```
evalᵒ fm s true
```

```
fm ≡ neg x  ∧  notᵒ a true ∧ evalᵒ x s a ∨
...
```

Specialization for miniKanren

input program

$\texttt{let rec eval}^o\ \texttt{fm s r} =$

$\quad \texttt{fm} \equiv \texttt{neg x} \ \wedge\ \texttt{not}^o\ \texttt{a r} \wedge \texttt{eval}^o\ \texttt{x s a} \vee$

$\quad \ldots$

known argument

$\texttt{eval}^o\ \texttt{fm s true}$

$\texttt{fm} \equiv \texttt{neg x} \ \wedge\ \texttt{not}^o\ \texttt{a true} \wedge \texttt{eval}^o\ \texttt{x s a} \vee$

$\ldots$

$\texttt{fm} \equiv \texttt{neg x} \ \wedge\ \texttt{eval}^o\ \texttt{x s false} \vee$

$\ldots$

Specialization for miniKanren

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x ∧ notᵒ a r ∧ evalᵒ x s a ∨
  ...
```

known argument

```
evalᵒ fm s true
```

```
fm ≡ neg x ∧ notᵒ a true ∧ evalᵒ x s a ∨
...
```

```
fm ≡ neg x ∧ evalᵒ x s false ∨
...
```

...

Specialization for miniKanren

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x ∧ notᵒ a r ∧ evalᵒ x s a ∨
  ...
```

```
evalᵒ fm s true
```

known argument

```
fm ≡ neg x ∧ notᵒ a true ∧ evalᵒ x s a ∨
...
```

```
fm ≡ neg x ∧ evalᵒ x s false ∨
...
```

...

output

```
let rec eval_trueᵒ fm s =
  fm ≡ neg x ∧ eval_falseᵒ x s ∨
  ...

let rec eval_falseᵒ fm s =
  fm ≡ neg x ∧ eval_trueᵒ x s ∨
  ...
```

Specialization for miniKanren: Bird's Eye View

```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y  ∧
  andᵒ a b r ∧
  evalᵒ x s a ∧
  evalᵒ y s b ∨
  ...
```

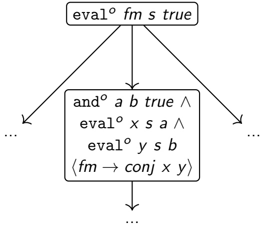
```
evalᵒ fm s true
```

Specialization for miniKanren: Bird's Eye View

```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y  ∧
  andᵒ a b r ∧
  evalᵒ x s a ∧
  evalᵒ y s b ∨
  ...
```

```
evalᵒ fm s true
```

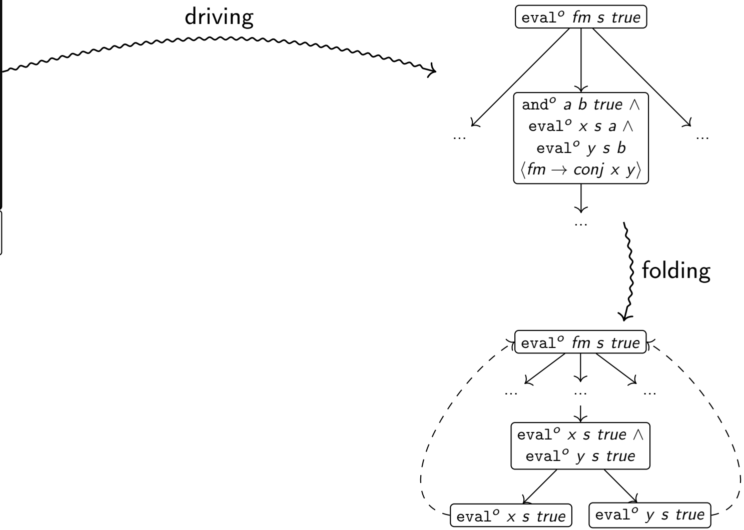
driving



Specialization for miniKanren: Bird's Eye View

```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y  ∧
  andᵒ a b r ∧
  evalᵒ x s a ∧
  evalᵒ y s b ∨
  ...
```

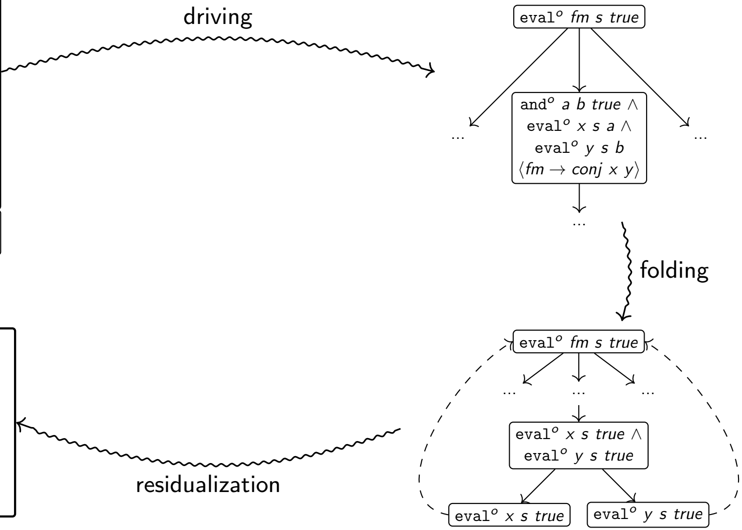
```
evalᵒ fm s true
```



Specialization for MINIKANREN: Bird's Eye View

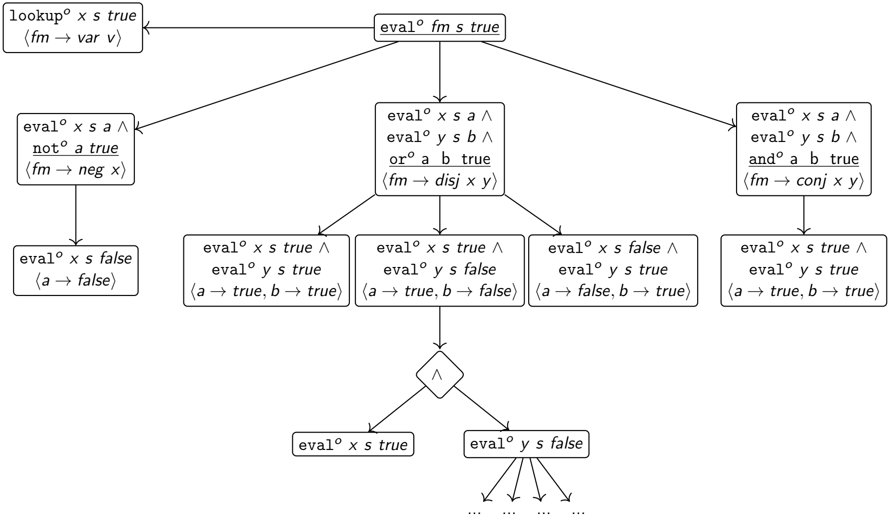
```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y  ∧
  andᵒ  a b r ∧
  evalᵒ  x s a ∧
  evalᵒ  y s b ∨
  ...
```

```
evalᵒ fm s true
```

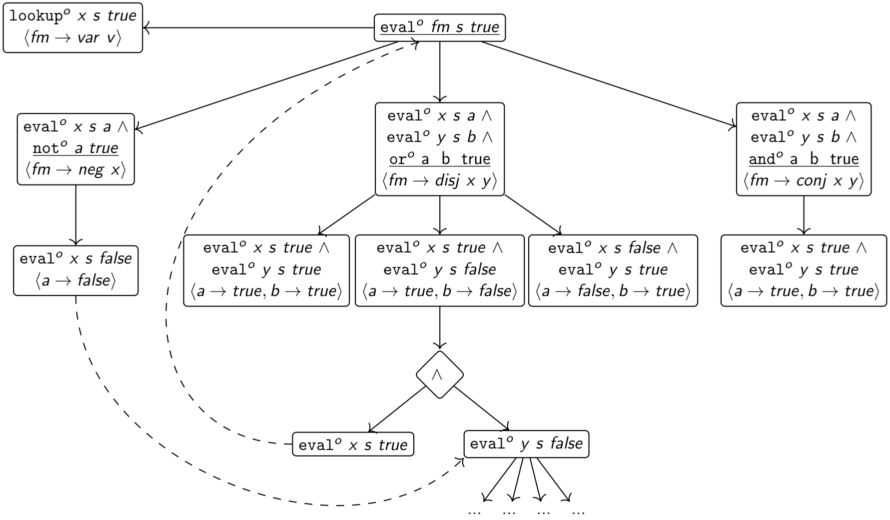



```
let rec evalᵒ_true fm s =
  ...
  fm ≡ conj x y  ∧
  evalᵒ_true x s ∧
  evalᵒ_true y s ∨
  ...
```

Specialization for miniKanren: ConsPD



Specialization for miniKanren: ConsPD



Evaluator of Logic Formulas: Order of Calls

boolean connective last

```
let rec evalᵒ fm s r =
  fresh (v x y a b) (
    (fm ≡ var v  ∧  lookupᵒ v s r) ∨
    (fm ≡ neg x  ∧  evalᵒ x s a ∧ notᵒ a r) ∨
    (fm ≡ conj x y  ∧  evalᵒ x s a ∧ evalᵒ y s b ∧ andᵒ a b r) ∨
    (fm ≡ disj x y  ∧  evalᵒ x s a ∧ evalᵒ y s b ∧ orᵒ a b r) )
```

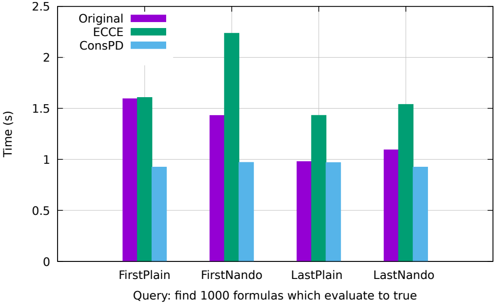
boolean connective first

```
let rec evalᵒ fm s r =
  fresh (v x y a b)
    (fm ≡ var v  ∧  lookupᵒ v s r) ∨
    (fm ≡ neg x  ∧  notᵒ a r ∧ evalᵒ x s a) ∨
    (fm ≡ conj x y  ∧  andᵒ a b r ∧ evalᵒ x s a ∧ evalᵒ y s b) ∨
    (fm ≡ disj x y  ∧  orᵒ a b r ∧ evalᵒ x s a ∧ evalᵒ y s b) }
```

Evaluator of Logic Formulas: Evaluation

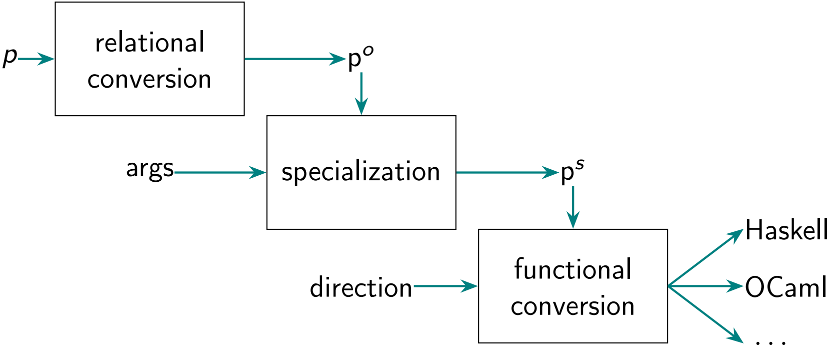
	Implementation	Placement
FirstPlain	table-based	before
LastPlain	table-based	after
FirstNando	via nand^o	before
LastNando	via nand^o	after

Table: Different implementations of eval^o



Functional Conversion

Functional Conversion



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) )
```

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

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

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

Addition in the Backward Direction: Nondeterminism

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

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

genNat = [0, 1, 2, ...]

addIOO 1 = [(0,1), (1,2), (2,3), ...]

```
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)
```

Modes: Data Flow

Ground term	S (S O)
Free variable	x

Once a variable is ground, it stays ground

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

$$\begin{array}{rl} \text{Mode I:} & \text{ground} \rightarrow \text{ground} \\ \text{Mode O:} & \text{free} \rightarrow \text{ground} \end{array}$$

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 mult° x y z = conde [
  (fresh (x₁ r₁)
    (x ≡ S x₁) ∧
    (add° y r₁ z) ∧
    (mult° 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

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
...
```

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

Maybe for Semi-Determinism

```
muloOII :: Nat → Nat → Stream Nat
muloOII x1 x2 =
    zero <|> positive
  where
    zero = do
      guard (x2 == O)
      return O
    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
```

10x faster

Conclusion

- We created a specialization method for miniKanren
- We implemented a functional conversion scheme
- They both speed up implementations considerably