An Empirical Study of Partial Deduction for miniKanren

Kate Verbitskaia, Daniil Berezun, Dmitry Boulytchev

JetBrains Research, Programming Languages and Tools Lab
Saint Petersburg State University

28.03.2021

miniKanren: Relational Programming Language (Family)

embedded

relation

```
let rec append^o x y r =
  ocanren {
    (x ≡ [] & y ≡ r) |
    (fresh h x' r' in
      x ≡ h :: x' &
      append^o x' y r' &
      r ≡ h :: r')}
```

term unification

disjunction

variable introduction

conjunction

relation call

```
let rec appendᵒ x y r =
  ocanren {
    (x ≡ [] & y ≡ r) |
    (fresh h x' r' in
      x ≡ h :: x' &
      appendᵒ x' y r' &
      r ≡ h :: r')}
```

- **fresh** q **in** $append^o$ [1] [2] q
  - $\langle$ q $\to$ [1,2] $\rangle$
- **fresh** x, y **in** $append^o$ x y [1,2]
  - $\langle$ x $\to$ [], y $\to$ [1,2] $\rangle$
  - $\langle$ x $\to$ [1], y $\to$ [2] $\rangle$
  - $\langle$ x $\to$ [1,2], y $\to$ [] $\rangle$
- **fresh** x, y, z **in** $append^o$ x y z
  - $\langle$ x $\to$ [], y $\to$ $\_{}_0$, z $\to$ $\_{}_0$ $\rangle$
  - $\langle$ x $\to$ [$\_{}_0$], y $\to$ $\_{}_1$, z $\to$ ($\_{}_0$: $\_{}_1$) $\rangle$
  - ...

Relational Interpreters for Search Problems

Recognizer backwards = solver

- Write recognizer in functional language
- Run relational conversion to get relational interpreter from the recognizer
- Run relational interpreter backwards

Core issue: running relational interpreter backwards is slow

Possible solution: partial deduction

Partial Deduction: a Method to Improve Logic Programs

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x & notᵒ a r & evalᵒ x s a |
  ...
```

Partial Deduction: a Method to Improve Logic Programs

input program

```
let rec evalᵒ fm s r =
  fm ≡ neg x & notᵒ a r & evalᵒ x s a |
  ...
```

known argument

```
evalᵒ fm s true
```

Partial Deduction: a Method to Improve Logic Programs

input program

let rec eval^o fm s r =
  fm ≡ neg x & not^o a r & eval^o x s a |
  ...

known argument

eval^o fm s **true**

fm ≡ neg x & not^o a **true** & eval^o x s a |
...

Partial Deduction: a Method to Improve Logic Programs

input program

let rec eval^o fm s r =
fm $\equiv$ neg x & not^o a r & eval^o x s a |
...

known argument

eval^o fm s **true**

fm $\equiv$ neg x & not^o a **true** & eval^o x s a |
...

fm $\equiv$ neg x & eval^o x s **false** |
...

Partial Deduction: a Method to Improve Logic Programs

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 |
...
```

...

Partial Deduction: a Method to Improve Logic Programs

input program

```
let rec eval^o fm s r =
  fm ≡ neg x & not^o a r & eval^o x s a |
  ...
```

known argument

```
eval^o fm s true
```

```
fm ≡ neg x & not^o a true & eval^o x s a |
...
```

```
fm ≡ neg x & eval^o x s false |
...
```

...

output

```
let rec eval_true^o fm s =
  fm ≡ neg x & eval_false^o x s |
  ...

let rec eval_false^o fm s =
  fm ≡ neg x & eval_true^o x s |
  ...
```

Partial Deduction for miniKanren: Bird's-eye View

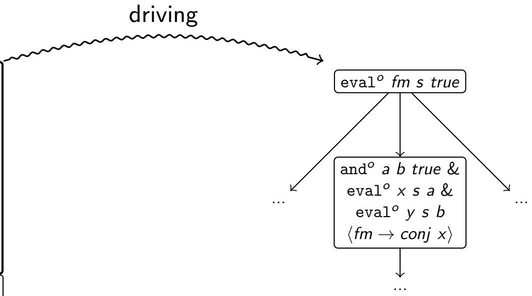
```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y &
  andᵒ a b r &
  evalᵒ x s a &
  evalᵒ y s b |
  ...
```

```
evalᵒ fm s true
```

Partial Deduction 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 |
  ...
```

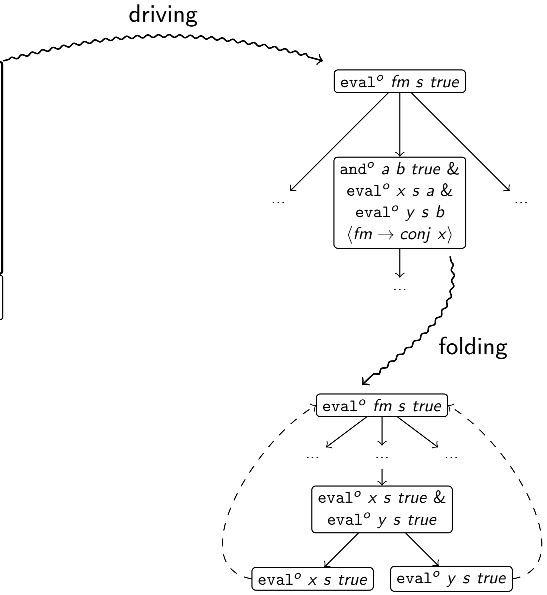
```
evalᵒ fm s true
```



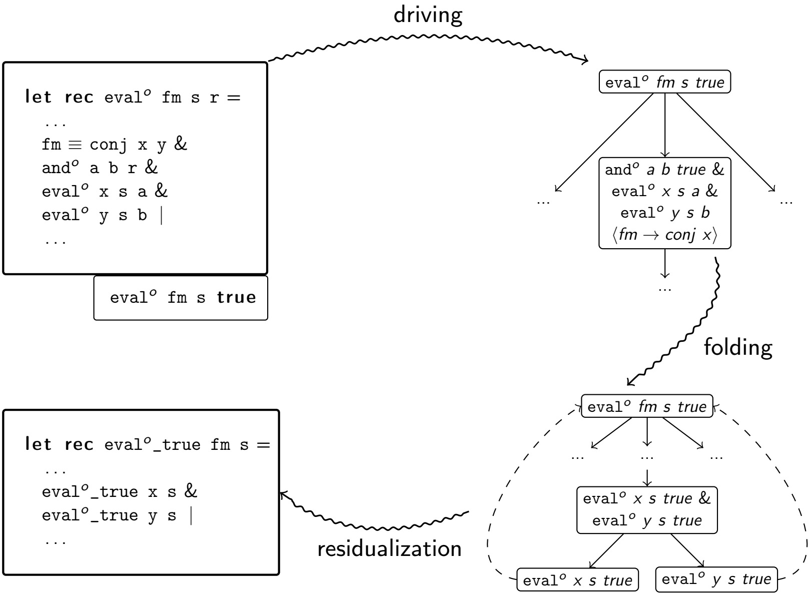
Partial Deduction for miniKanren: Bird's-eye View

```
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
```



Partial Deduction for miniKanren: Bird's-eye View



Driving: Unfolding

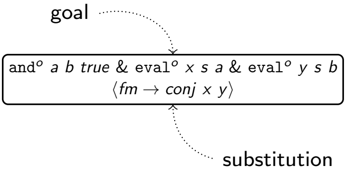
```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y & andᵒ a b r &
  evalᵒ x s a & evalᵒ y s b |
  ...

let andᵒ x y r =
  ocanren {
    fresh xy in
      (nandᵒ x y xy & nandᵒ xy xy r) }

let rec nandᵒ x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ false) |
    (x ≡ true & y ≡ false & r ≡ true) |
    (x ≡ false & y ≡ true & r ≡ true) |
    (x ≡ false & y ≡ false & r ≡ true) }
```

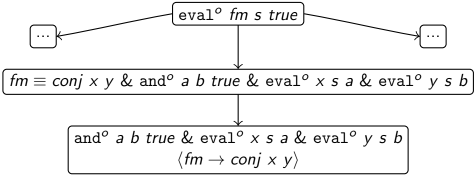
```
evalᵒ fm s true
```

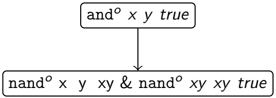
Driving: Unfolding

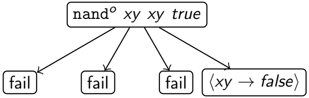
```
let rec eval^o fm s r =
  ...
  fm ≡ conj x y & and^o a b r &
  eval^o x s a & eval^o y s b |
  ...

let and^o x y r =
  ocanren {
    fresh xy in
      (nand^o x y xy & nand^o xy xy r) }

let rec nand^o x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ false) |
    (x ≡ true & y ≡ false & r ≡ true) |
    (x ≡ false & y ≡ true & r ≡ true) |
    (x ≡ false & y ≡ false & r ≡ true) }
```

```
eval^o fm s true
```









Partial Deduction

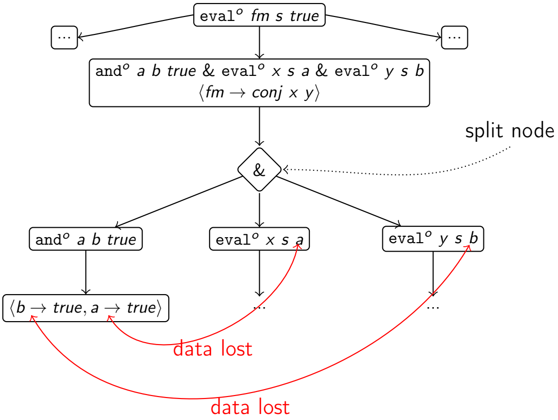
```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y & andᵒ a b r &
  evalᵒ x s a & evalᵒ y s b |
  ...
```

```
evalᵒ fm s true
```

Partial Deduction

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

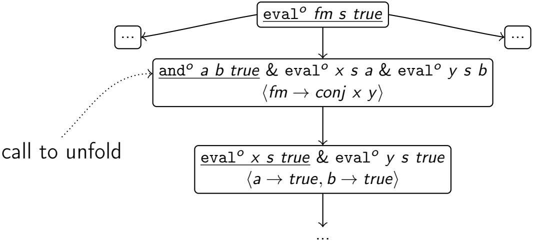
evalᵒ fm s **true**



Conjunctive Partial Deduction: Left-to-right Unfolding

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

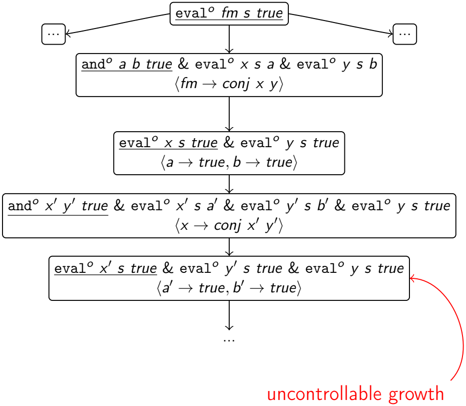
```
evalᵒ fm s true
```



CPD: Split is Necessary

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

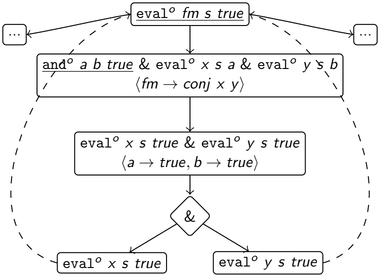
```
evalᵒ fm s true
```

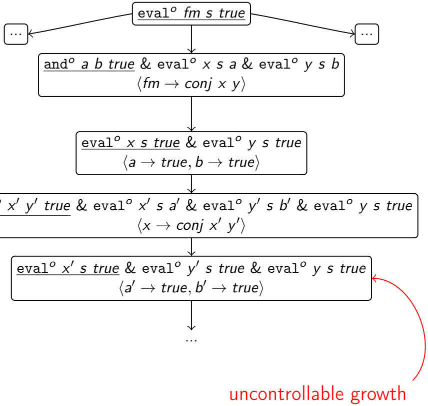


CPD: Split is Necessary

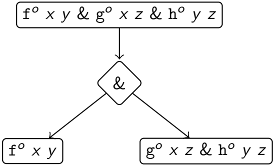
```
let rec evalᵒ fm s r =
  ...
  fm ≡ conj x y & andᵒ a b r &
  evalᵒ x s a & evalᵒ y s b |
  ...
```

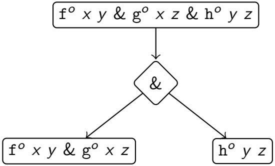
evalᵒ fm s **true**

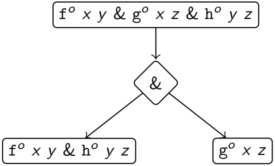


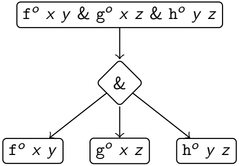


Split: Which Way is the Right Way?









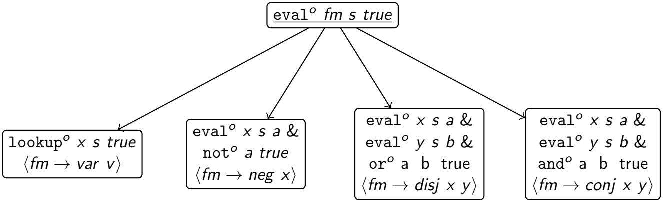
Decisions in Partial Deduction

- What to unfold: which calls, how many calls?
  - CPD: the leftmost call, which does not have a predecessor *embedded* into it
- How to unfold: to what depth a call should be unfolded?
  - CPD: unfold once
- When to stop driving?
  - When a goal is an instance of some goal in the process tree
- When to split?
  - When there is a predecessor embedded into the goal

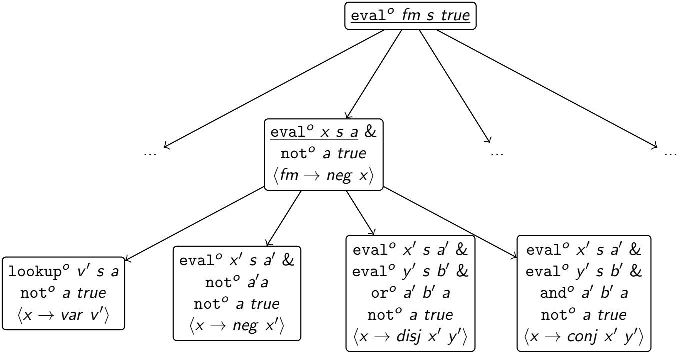
Evaluator of Logic Formulas: Unfolding Step 1

```
let rec evalᵒ fm s r =
  ocanren { fresh v x y a b in
    (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 & oroᵒ a b r) }
```

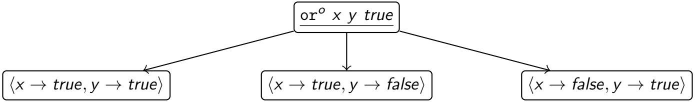
evalᵒ fm s **true**

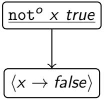


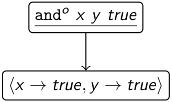
Evaluator of Logic Formulas: Unfolding Step 2



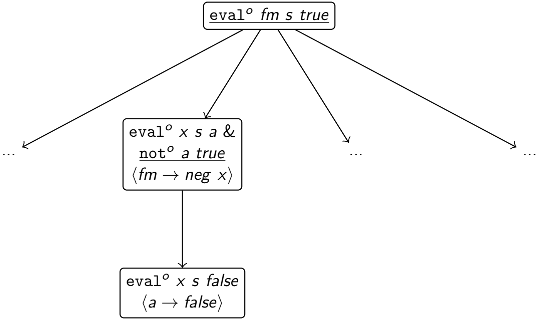
Unfolding of Boolean Connectives



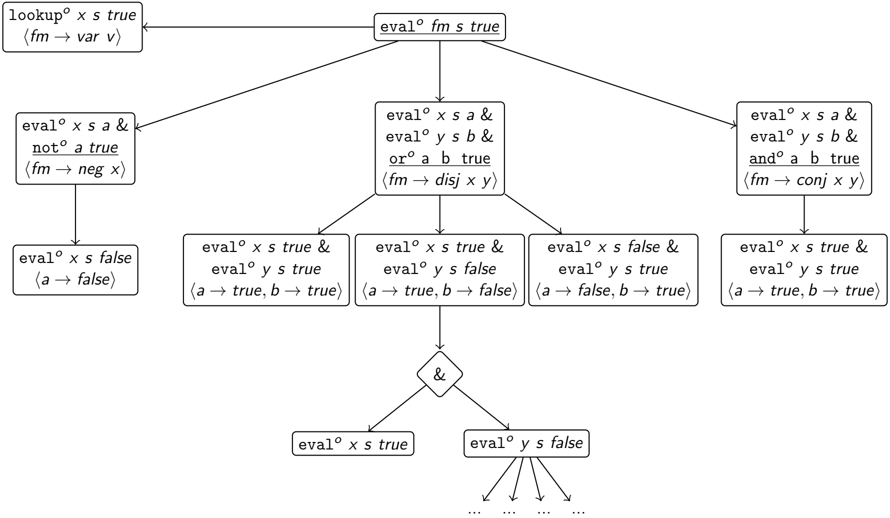




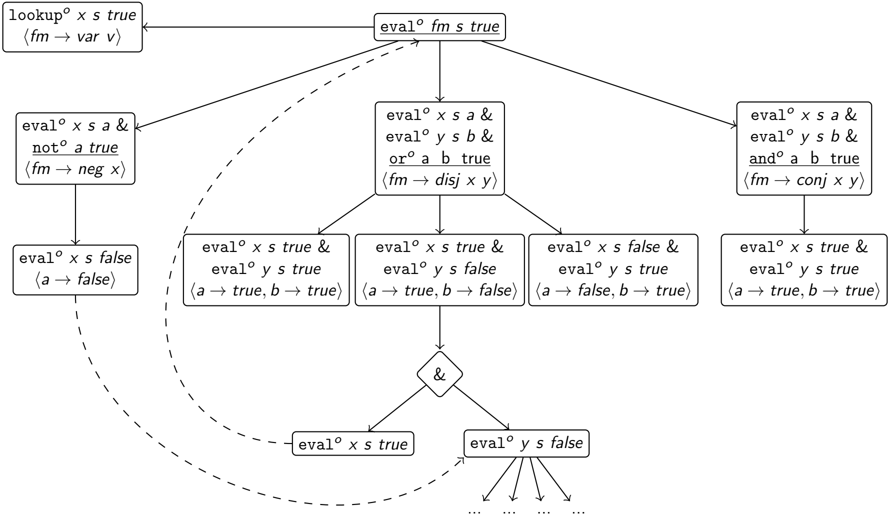
Unfolding Boolean Connectives First



Evaluator of Logic Formulas: Conservative PD



Evaluator of Logic Formulas: Conservative PD



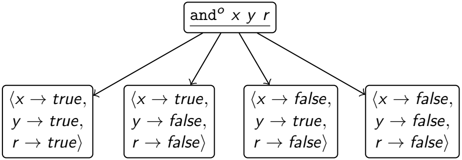
Conservative Partial Deduction

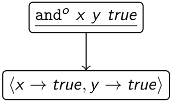
- Split conjunction into individual calls
- Unfold each call in isolation
- Unfold until embedding is encountered
- Find a call which narrows the search space (less-branching heuristics)
- Join the result of unfolding the selected call with the other calls not unfolded
- Continue driving the constucted conjunction

Less-branching Heuristics

Less-branching heuristics is used to select a call to unfold

If a call in the context unfolds into less branches than it does in isolation, select it





Evaluation

We implemented the Conservative Partial Deduction and compared it with CPD[1] on the following relations

- Four implementations of an evaluator of logic formulas
- Two implementations of a typechecker for a simple language

[1] ECCE partial deduction system

Evaluator of Logic Formulas: Order of Calls

boolean connective last

```
let rec evalᵒ fm s r =
  ocanren { fresh v x y a b in
    (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 & oroᵒ a b r) }
```

Evaluator of Logic Formulas: Order of Calls

boolean connective last

```
let rec evalᵒ fm s r =
  ocanren { fresh v x y a b in
    (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 & oroᵒ a b r) }
```

boolean connective first

```
let rec evalᵒ fm s r =
  ocanren { fresh v x y a b in
    (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 & oroᵒ a b r & evalᵒ x s a & evalᵒ y s b) }
```

Evaluator of Logic Formulas: Compexity of Relations

table-based implementation

```
let rec andᵒ x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ true) |
    (x ≡ true & y ≡ false & r ≡ false) |
    (x ≡ false & y ≡ true & r ≡ false) |
    (x ≡ false & y ≡ false & r ≡ false) }
```

Evaluator of Logic Formulas: Compexity of Relations

table-based implementation

```
let rec andᵒ x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ true) |
    (x ≡ true & y ≡ false & r ≡ false) |
    (x ≡ false & y ≡ true & r ≡ false) |
    (x ≡ false & y ≡ false & r ≡ false) }
```

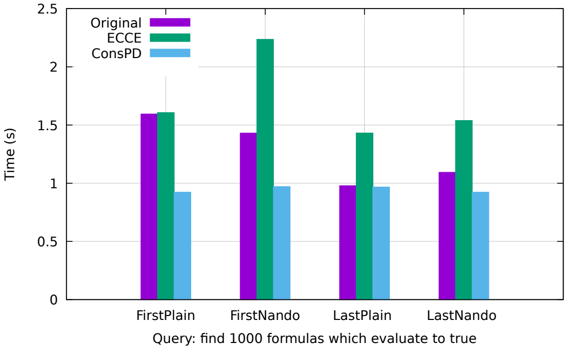
implementation via nand^o

```
let andᵒ x y r =
  ocanren {
    fresh xy in
      (nandᵒ x y xy & nandᵒ xy xy r) }

let rec nandᵒ x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ false) |
    (x ≡ true & y ≡ false & r ≡ true) |
    (x ≡ false & y ≡ true & r ≡ true) |
    (x ≡ false & y ≡ false & r ≡ true) }
```

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



Typechecker-Term Generator: Language

$$
\begin{array}{rllll}
term = & BConst\ of\ Bool & \mid IConst\ of\ Int & \mid Var\ of\ Int \\
& \mid term + term & \mid term * term & \\
& \mid term = term & \mid term < term & \\
& \mid \underline{let}\ term\ \underline{in}\ term & \mid \underline{if}\ term\ \underline{then}\ term\ \underline{else}\ term &
\end{array}
$$

Figure: Language syntax

$$\frac{}{\Gamma \vdash IConst\ i : Int} \qquad \frac{}{\Gamma \vdash BConst\ b : Bool} \qquad \frac{}{\Gamma \vdash Var\ v : \tau}\ \Gamma[v] \equiv \tau$$

$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t + s : Int} \qquad \frac{\Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash t = s : Bool} \qquad \frac{\Gamma \vdash v : \tau_v,\ (\tau_v :: \Gamma) \vdash b : \tau}{\Gamma \vdash \underline{let}\ v\ b : \tau}$$

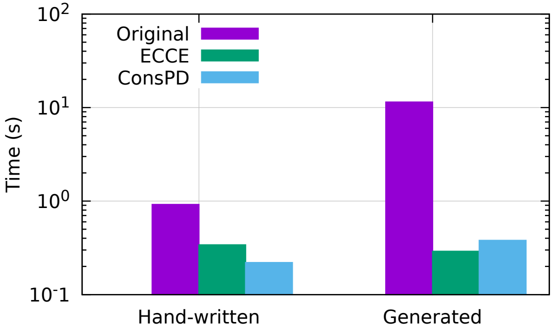
$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t * s : Int} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t < s : Bool} \qquad \frac{\Gamma \vdash c : Bool, \Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \underline{if}\ c\ \underline{then}\ t\ \underline{else}\ s : \tau}$$

Figure: Typing rules implemented in typecheck^o relation

Typechecker-Term Generator: Evaluation

Implementations:

- Hand-coded typing rules in miniKanren
- Generated from functional typechecker by relational conversion



Discussion: Order of Answers

Example from evalo.
Uselessness of measuring time.
Let's measure unifications

Discussion: Deterministic Unfolding and Tupling

maxlen works maxmin does not work

Conclusion

- We developed and implemented Conservative Partial Deduction
  - Less-branching heuristics
- Evaluation shows some improvement, but not for every query
- Future work:
  - Develop models to predict execution time
  - Develop specialization which is more predictable, stable and well-behaved