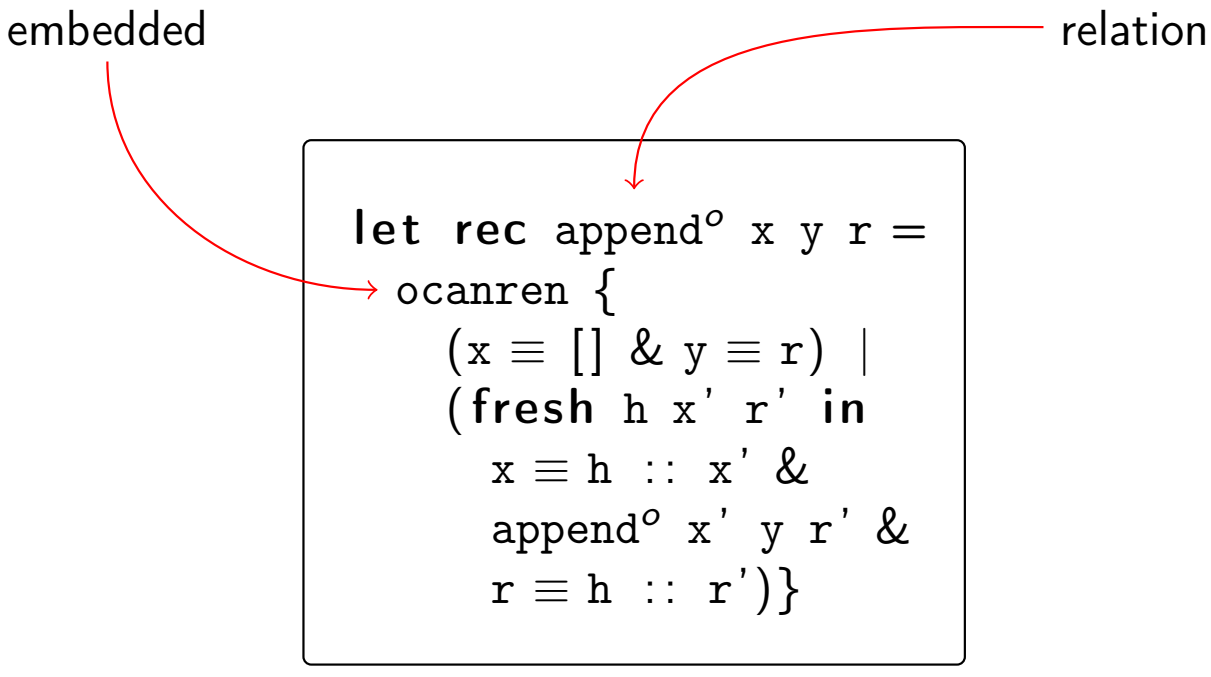
An Empirical Study of Partial Deduction for miniKanren

Kate Verbitskaia, Daniil Berezun, Dmitry Boulytchev

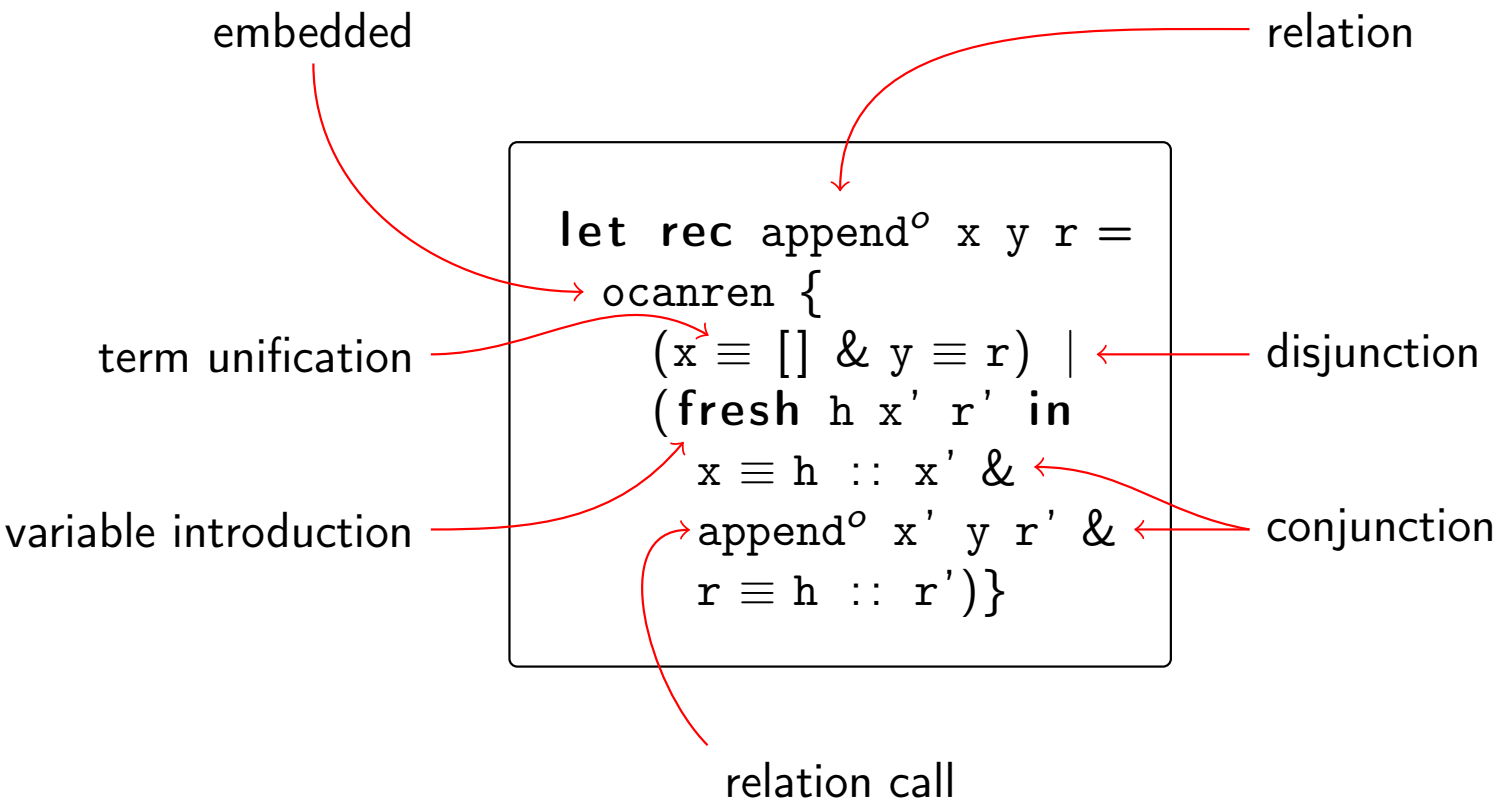
JetBrains Research, Programming Languages and Tools Lab
Saint Petersburg State University

28.03.2021

miniKanren: Relational Programming Language (Family)



miniKanren: Relational Programming Language (Family)



miniKanren: Querying

```
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** appendᵒ [1] [2] q
  - ⟨ q → [1,2] ⟩

miniKanren: Querying

```
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** appendᵒ [1] [2] q
  - ⟨ q → [1,2] ⟩
- **fresh** x, y **in** appendᵒ x y [1,2]
  - ⟨ x → [], y → [1,2] ⟩
  - ⟨ x → [1], y → [2] ⟩
  - ⟨ x → [1,2], y → [] ⟩

miniKanren: Querying

```
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** appendᵒ [1] [2] q
  - ⟨ q → [1,2] ⟩
- **fresh** x, y **in** appendᵒ x y [1,2]
  - ⟨ x → [], y → [1,2] ⟩
  - ⟨ x → [1], y → [2] ⟩
  - ⟨ x → [1,2], y → [] ⟩
- **fresh** x, y, z **in** appendᵒ x y z
  - ⟨ x → [], y → $_0$, z → $_0$ ⟩
  - ⟨ x → [$_0$], y → $_1$, z → ($_0$: $_1$) ⟩
  - ...

miniKanren: Semantics

- Interleaving search
- Complete search
  - All existing answers will be eventually found
  - It is safe to reorder within conjunctions and disjunctions
  - Reordering affects the efficiency of programs

Relational Interpreters for Search Problems

Recognizer run backwards = solver

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

Core issue: running relational interpreter backwards is slow

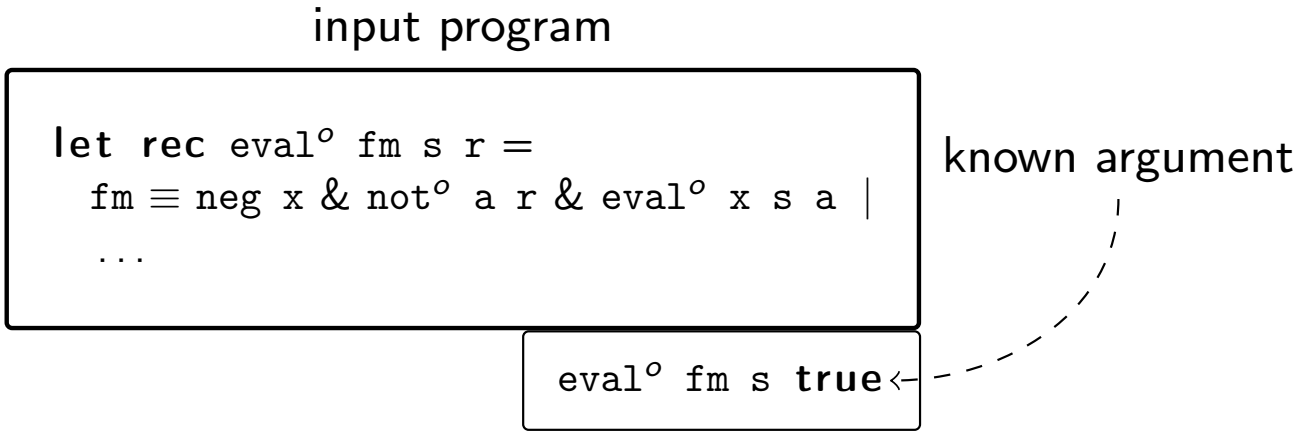
Possible solution: partial deduction

Partial Deduction: a Method to Improve Logic Programs

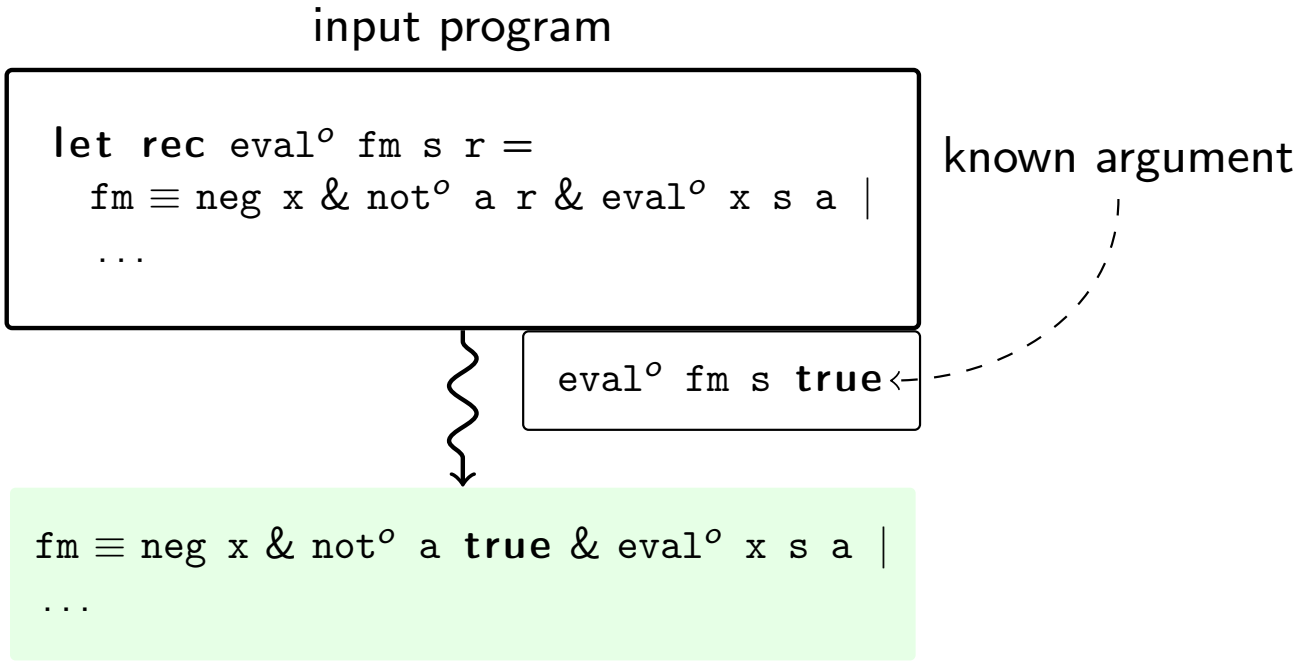
input program

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

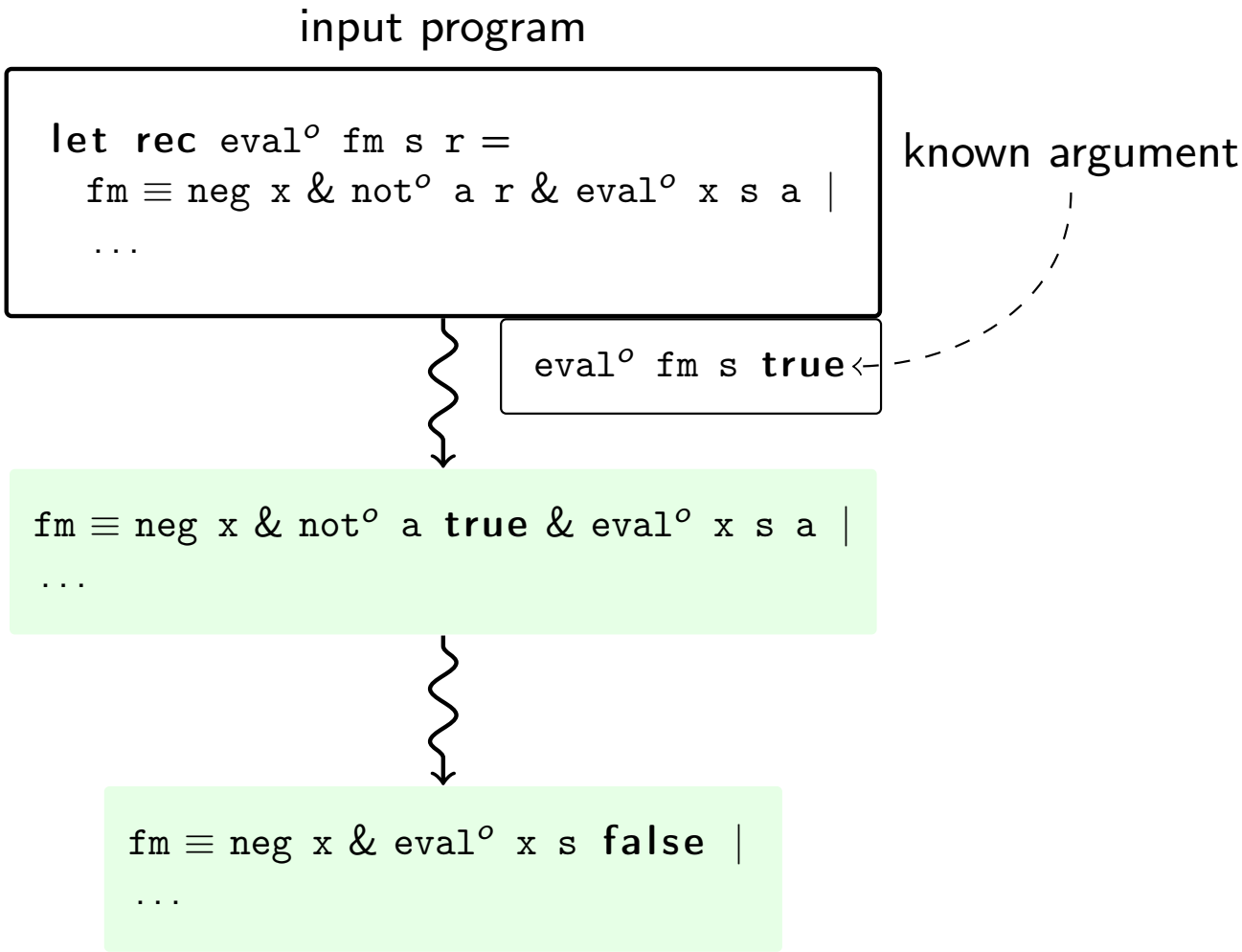
Partial Deduction: a Method to Improve Logic Programs



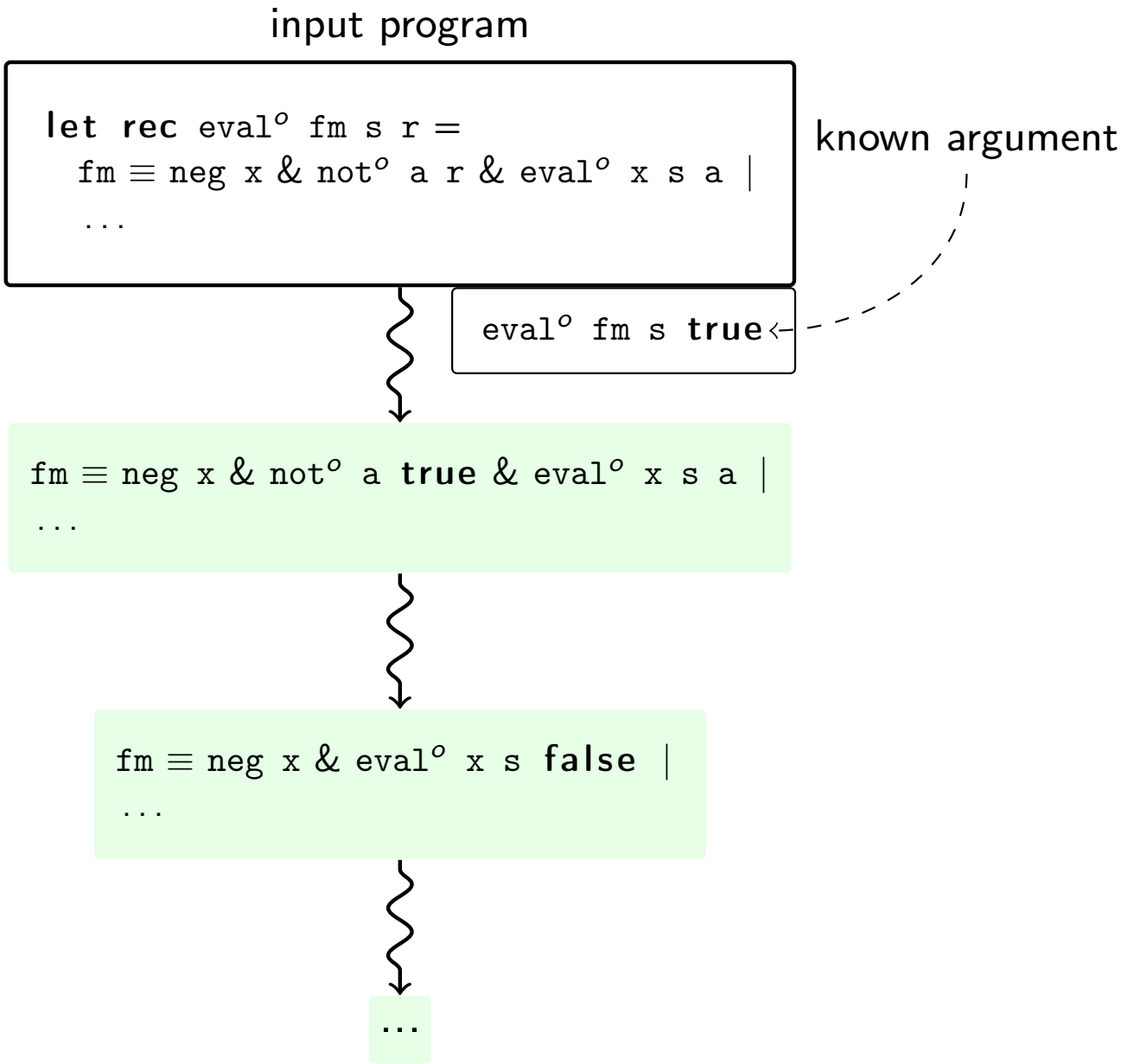
Partial Deduction: a Method to Improve Logic Programs



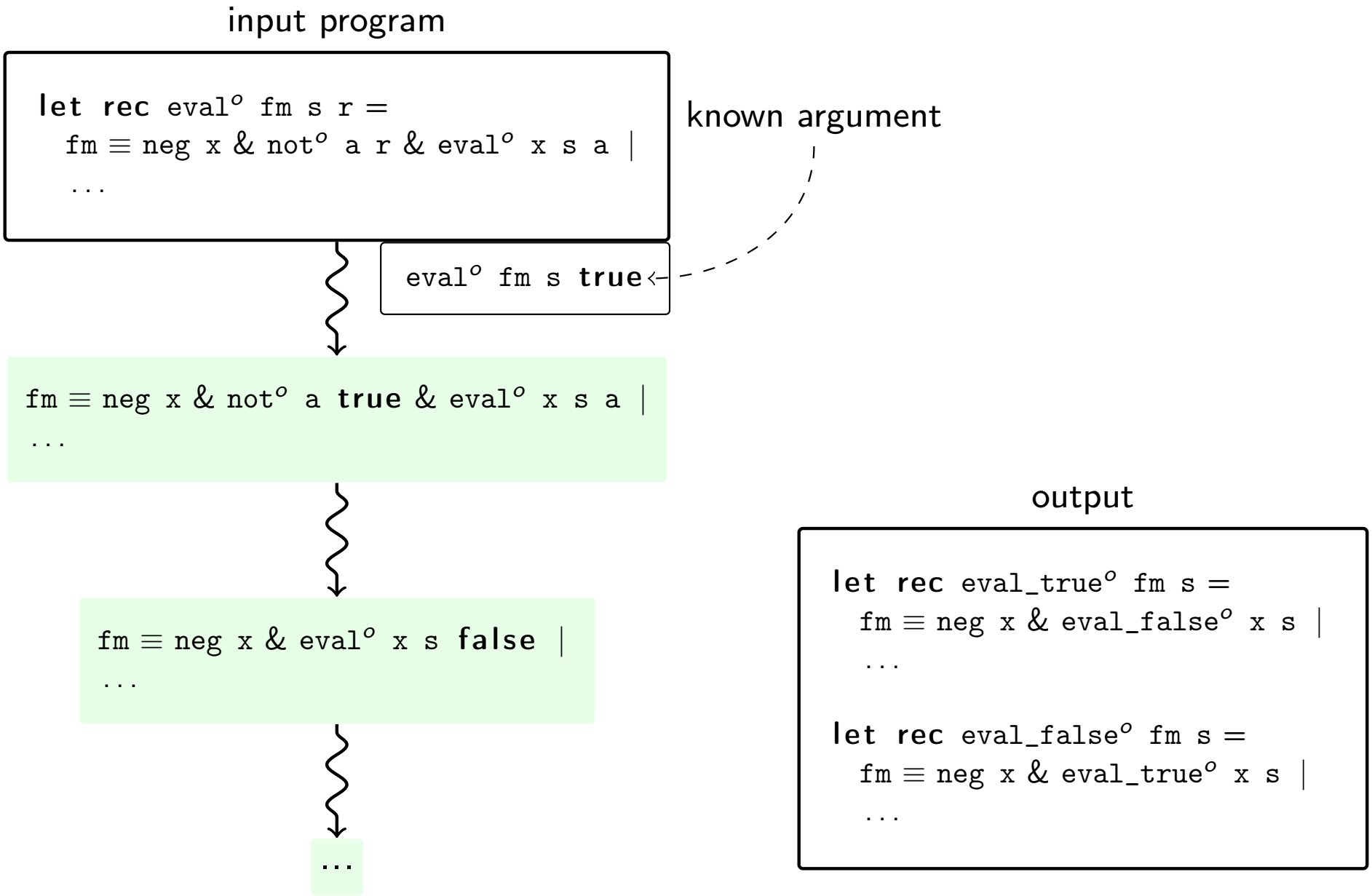
Partial Deduction: a Method to Improve Logic Programs



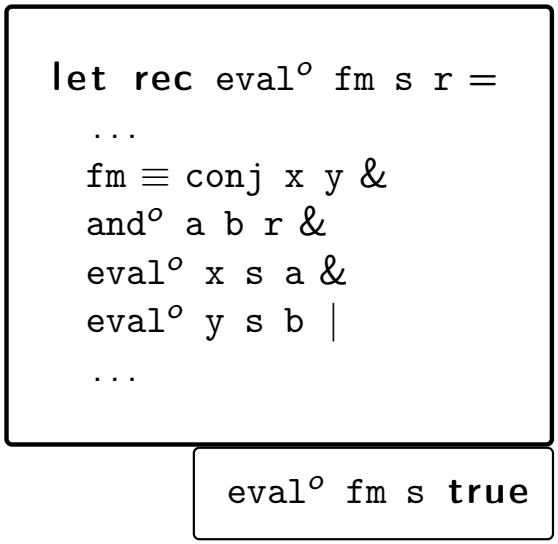
Partial Deduction: a Method to Improve Logic Programs



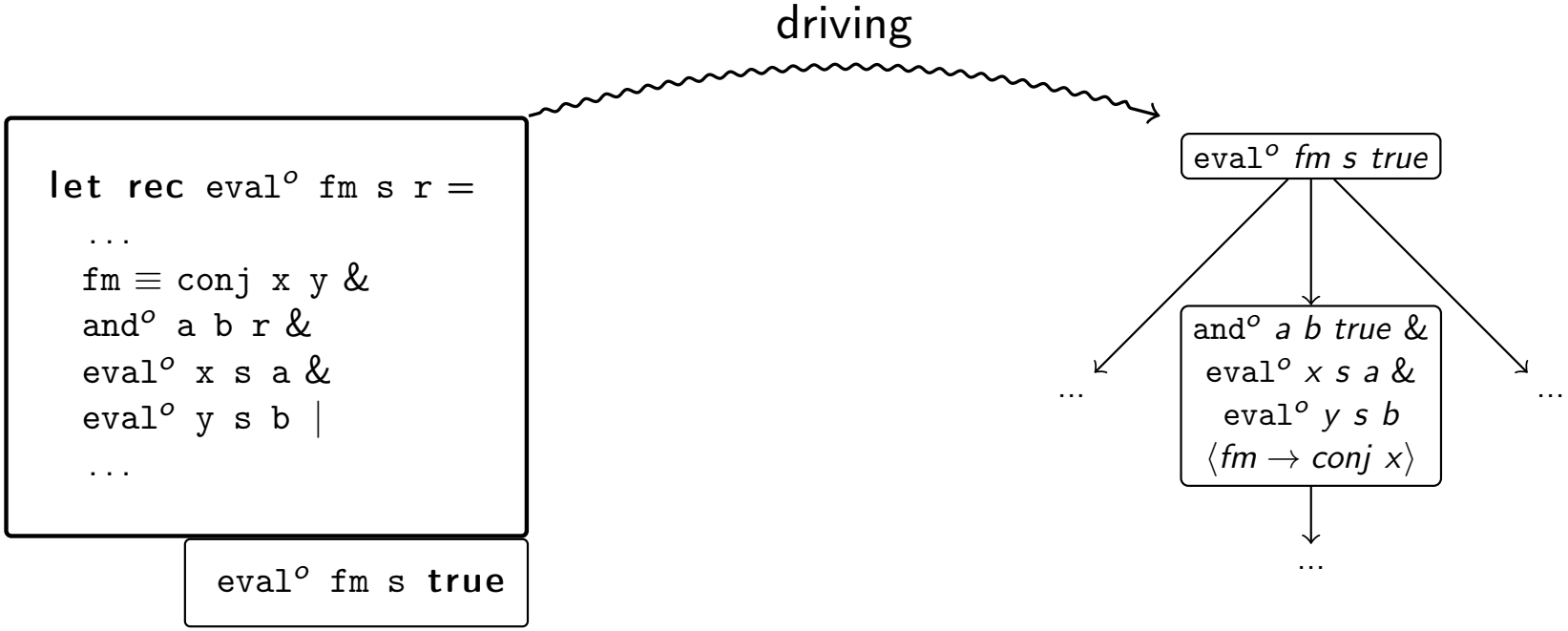
Partial Deduction: a Method to Improve Logic Programs



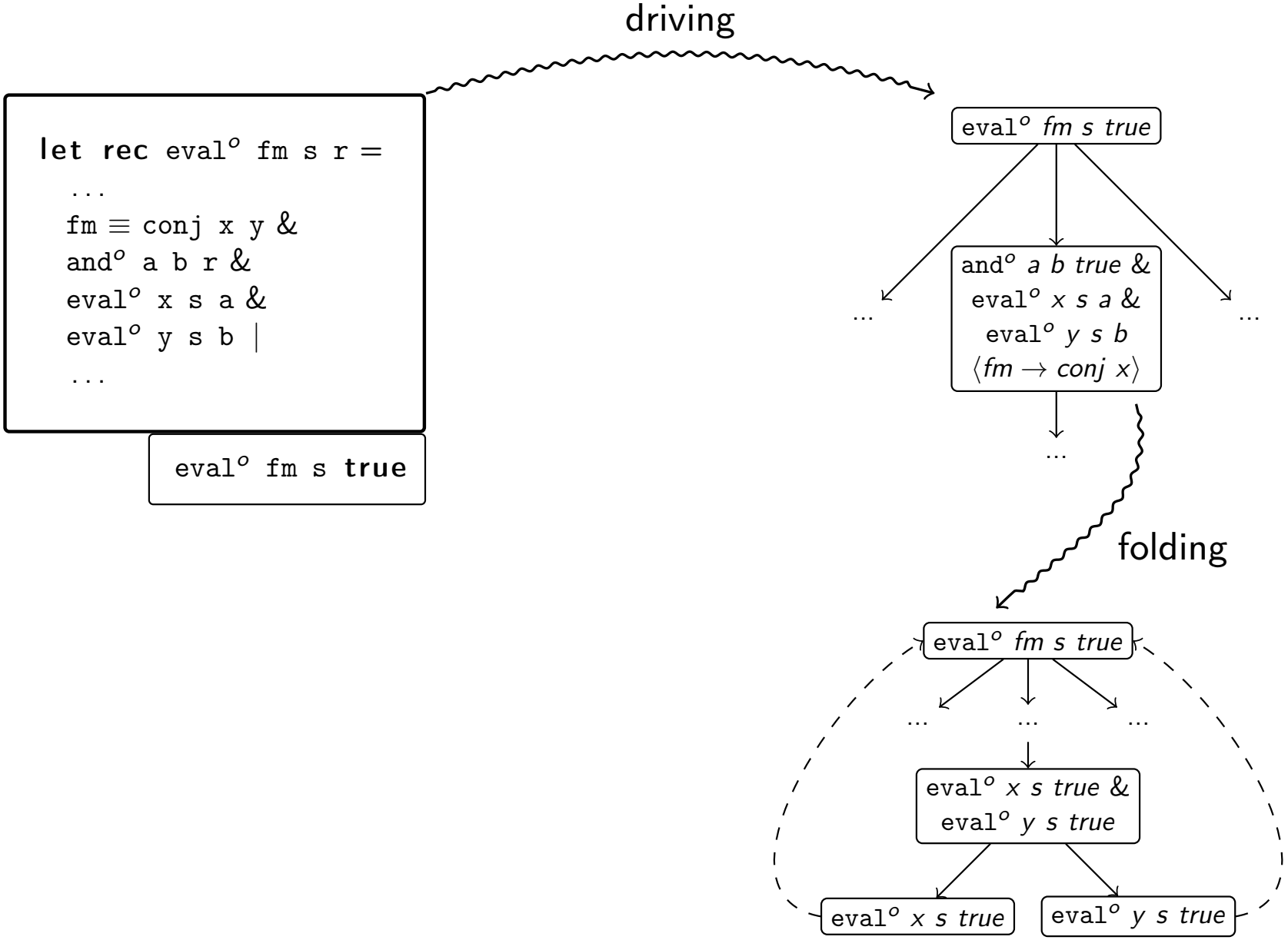
Partial Deduction for MINIKANREN: Bird's-eye View



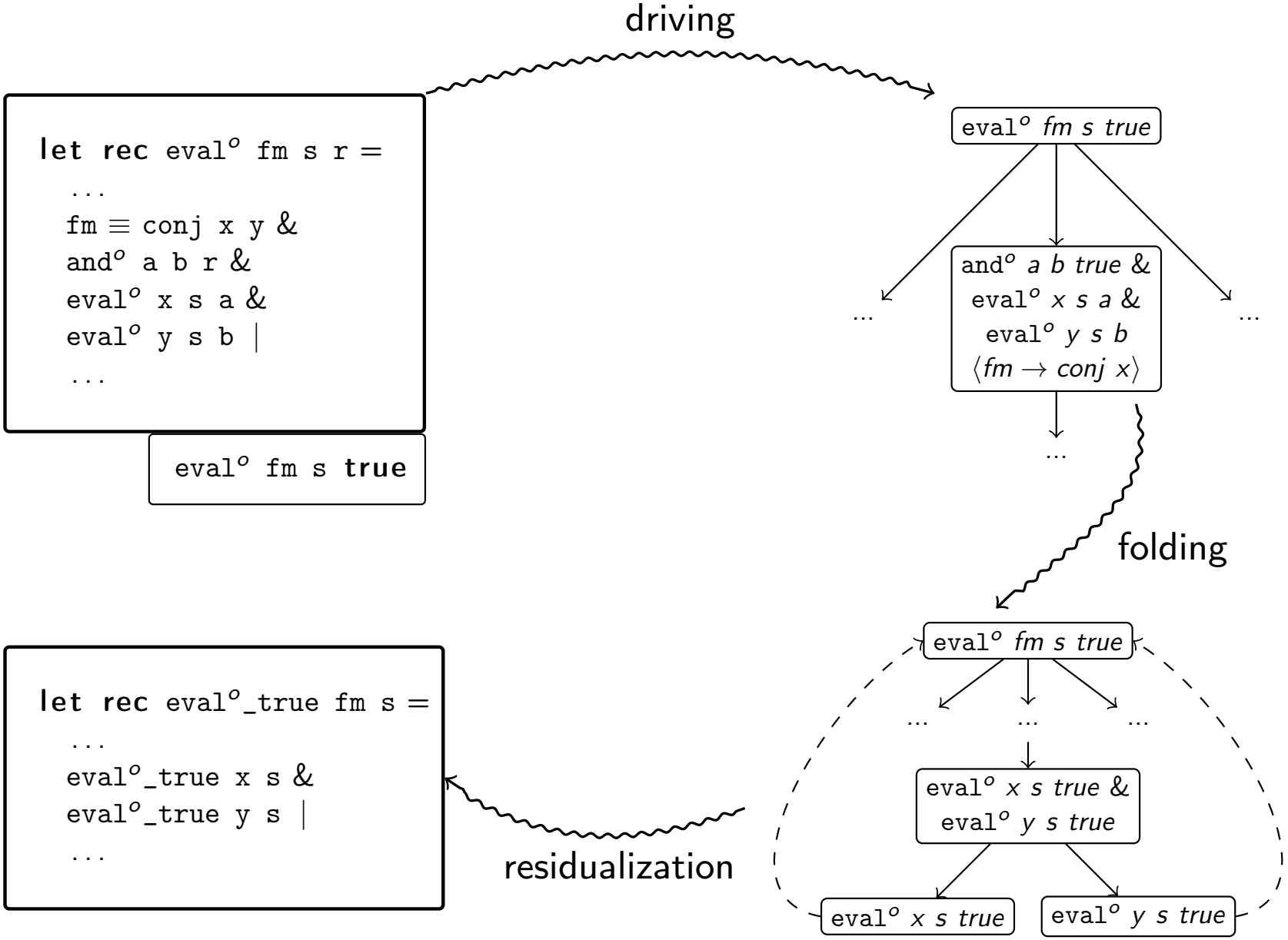
Partial Deduction for MINIKANREN: Bird's-eye View



Partial Deduction for miniKanren: Bird's-eye View



Partial Deduction for miniKanren: Bird's-eye View



Driving: Unfolding

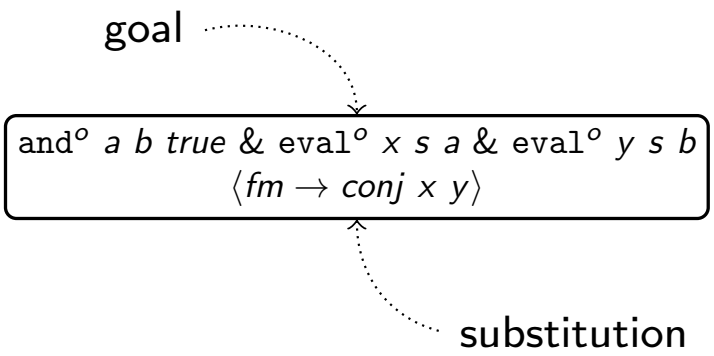
```
let rec eval^o fm s r =
  ...
  fm ≡ conj x y & and^o a b r &
  eval^o x s a & eval^o y s b |
  ...

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) }
```

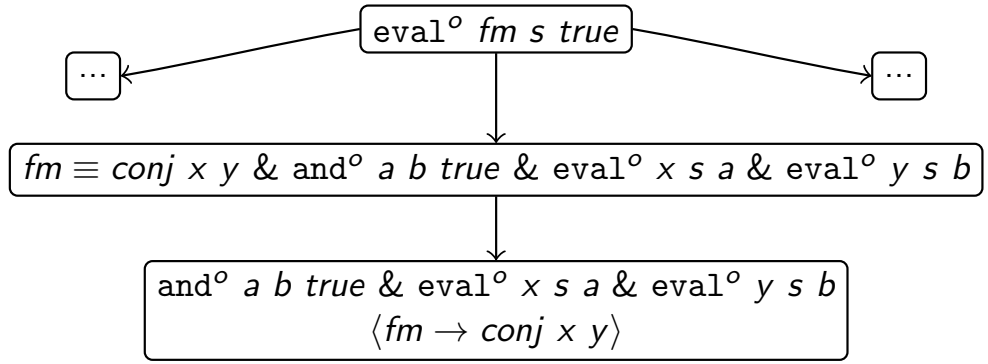
```
eval^o fm s true
```

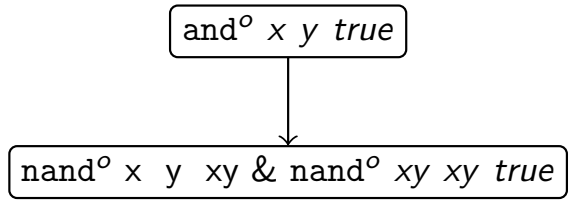
Driving: Unfolding

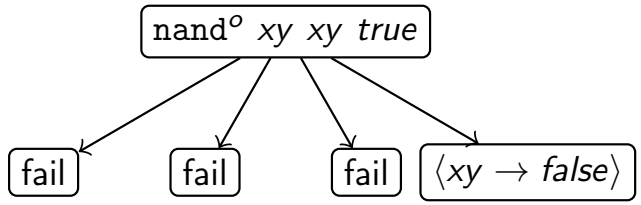
```
let rec eval^o fm s r =
  ...
  fm ≡ conj x y & and^o a b r &
  eval^o x s a & eval^o y s b |
  ...

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

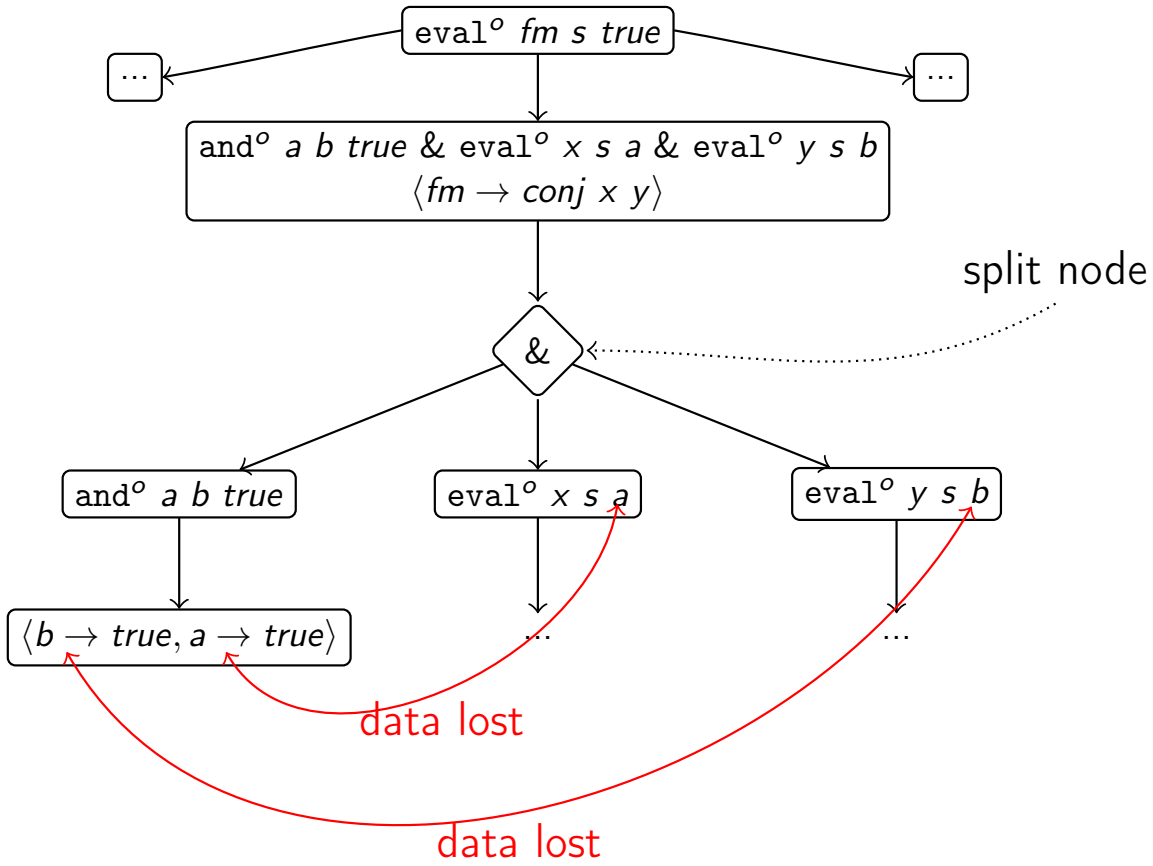
```
let rec eval^o fm s r =
  ...
  fm ≡ conj x y & and^o a b r &
  eval^o x s a & eval^o y s b |
  ...
```

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

Partial Deduction

```
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 |
  ...
```

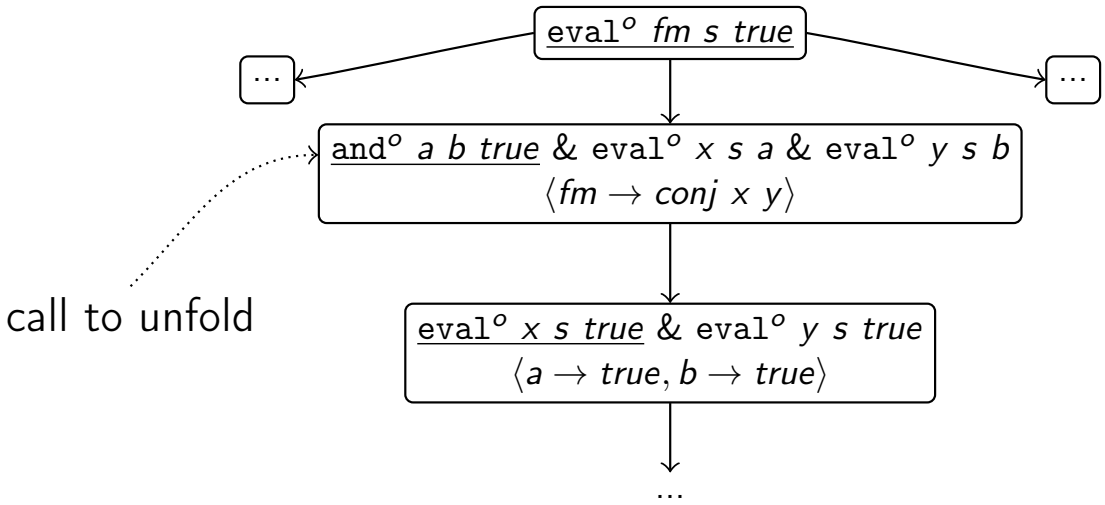
`eval^o fm s true`



Conjunctive Partial Deduction: Left-to-right 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 |
  ...
```

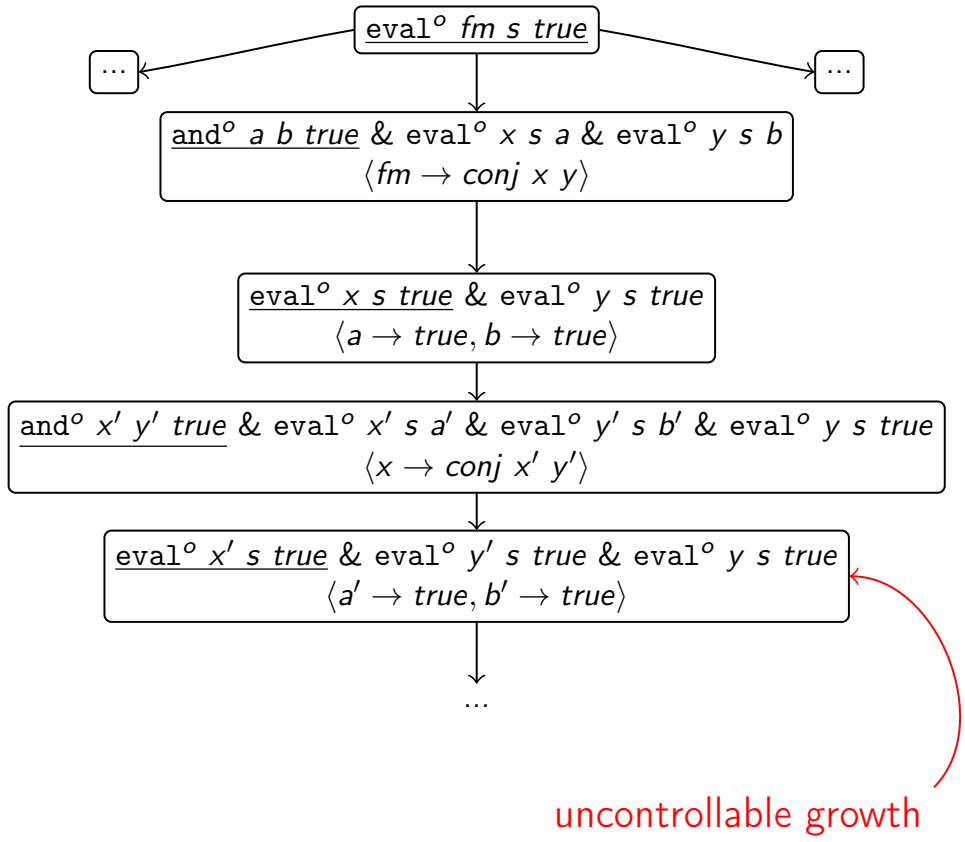
`eval^o fm s true`



CPD: Split is Necessary

```
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 |
  ...
```

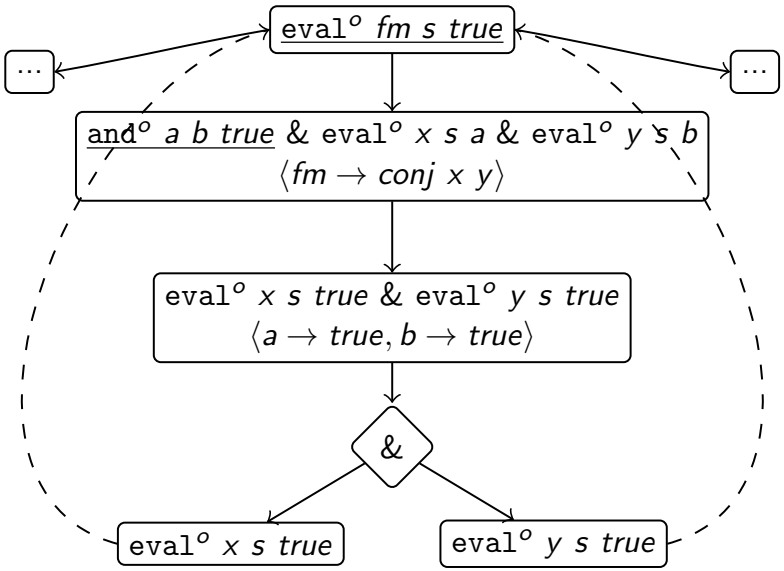
`eval^o fm s true`

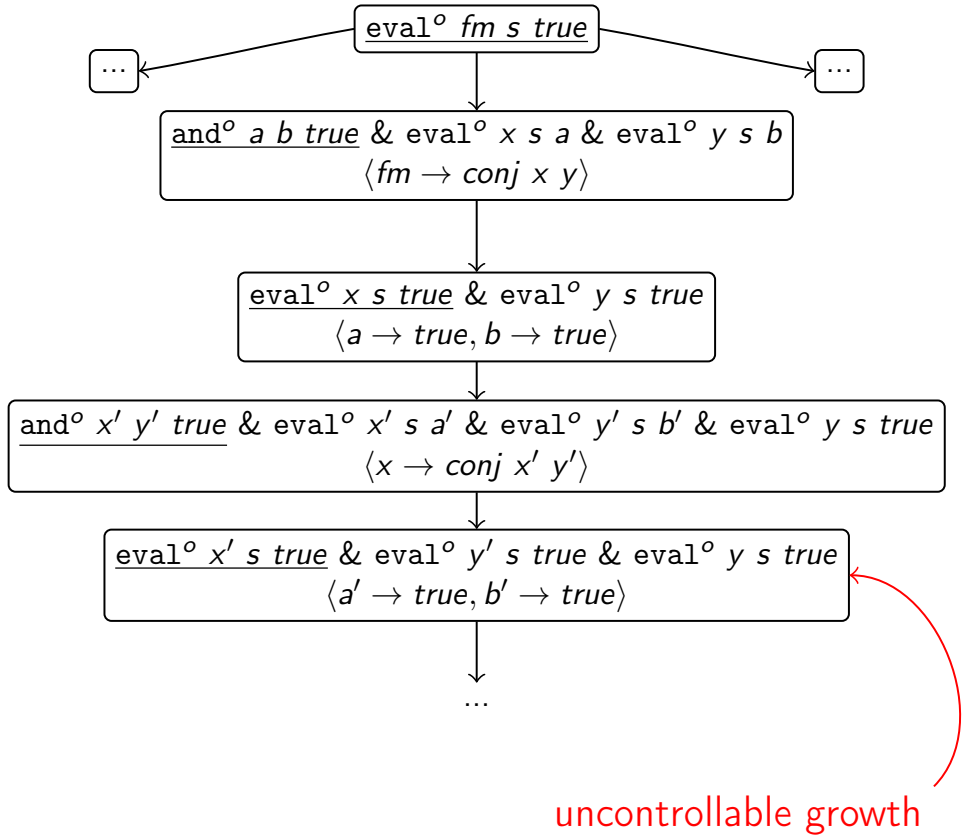


CPD: Split is Necessary

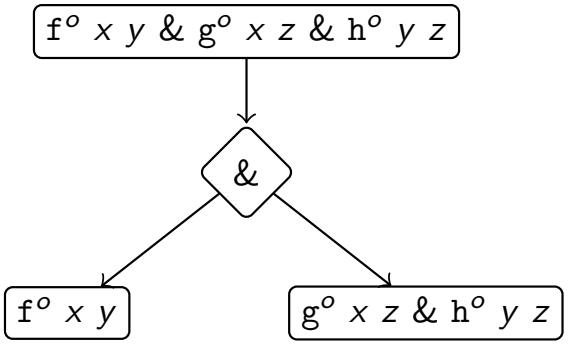
```
let rec eval^o fm s r =
  ...
  fm ≡ conj x y & and^o a b r &
  eval^o x s a & eval^o y s b |
  ...
```

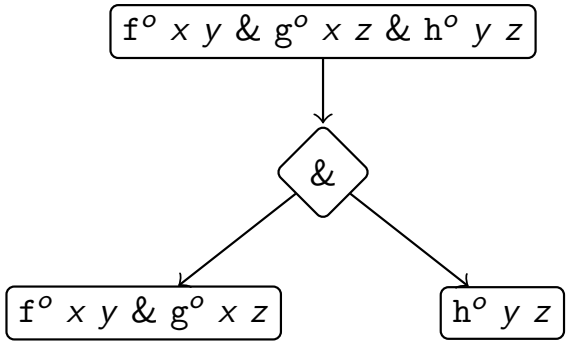
`eval^o fm s true`

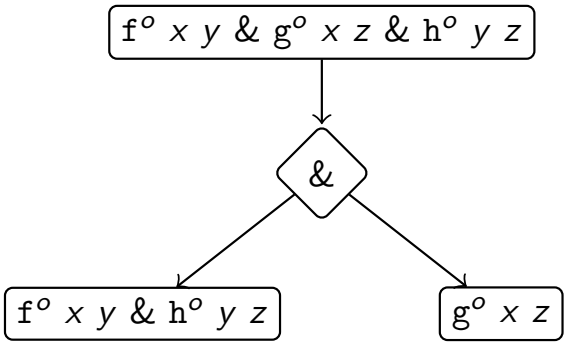


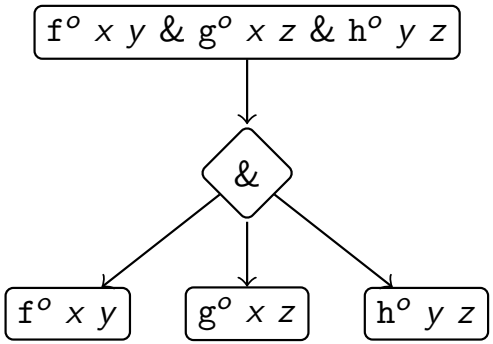


Split: Which Way is the Right Way?









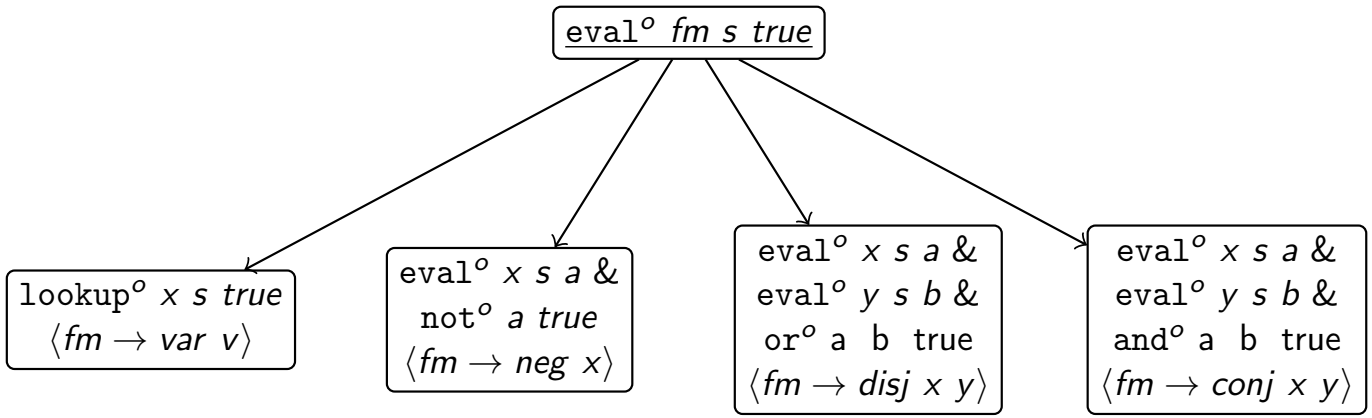
Decisions in Partial Deduction

- 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

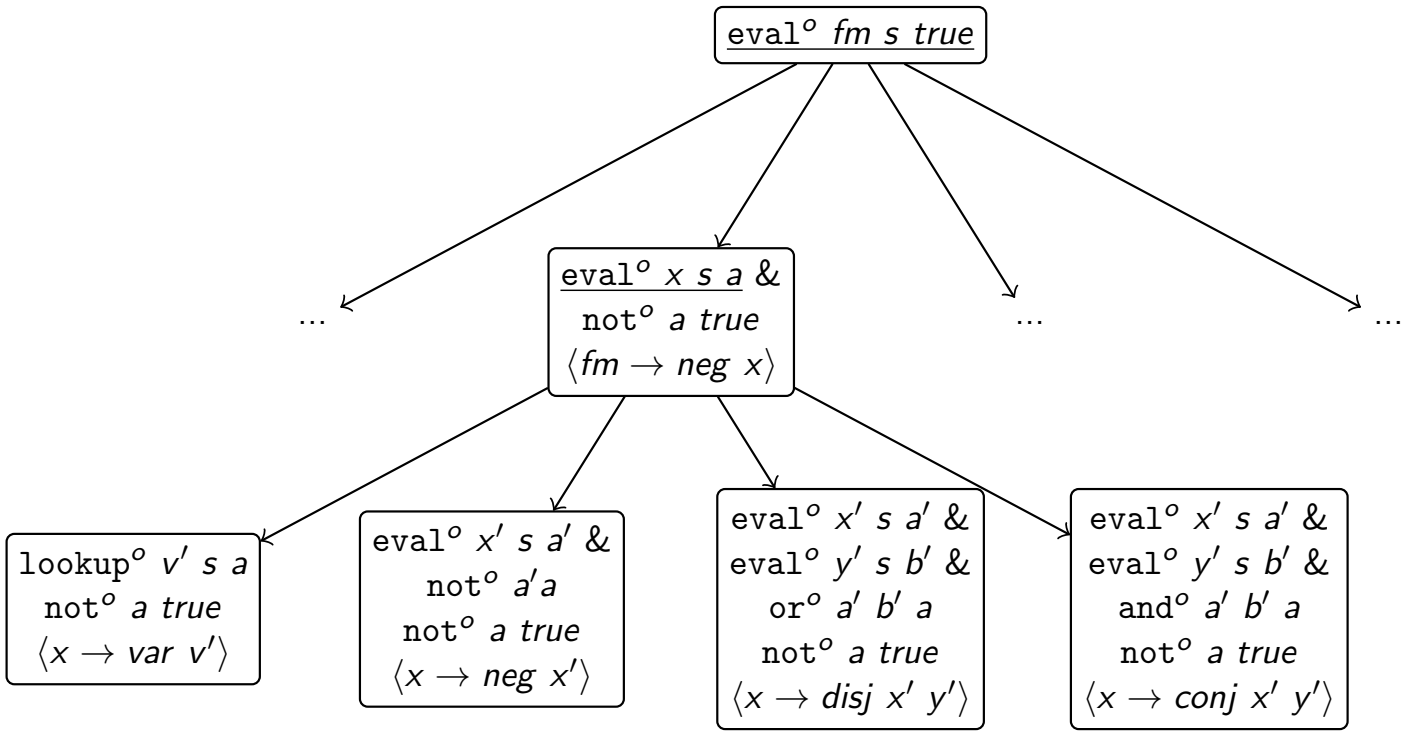
Evaluator of Logic Formulas: Unfolding Step 1

```
let rec eval^o fm s r =
  ocanren { fresh v x y a b in
    (fm ≡ var v & lookup^o v s r) |
    (fm ≡ neg x & eval^o x s a & not^o a r) |
    (fm ≡ conj x y & eval^o x s a & eval^o y s b & and^o a b r) |
    (fm ≡ disj x y & eval^o x s a & eval^o y s b & or^o a b r) }
```

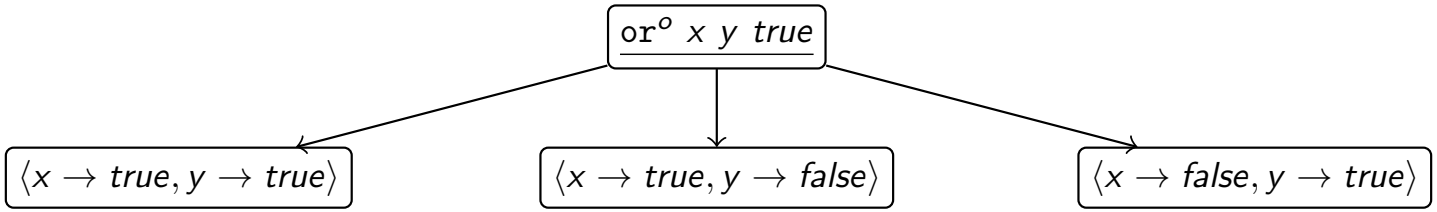
`eval^o fm s true`

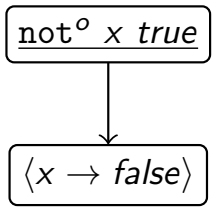


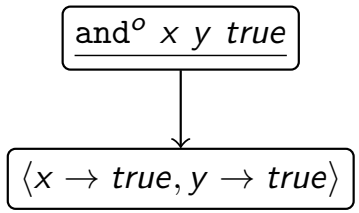
Evaluator of Logic Formulas: Unfolding Step 2



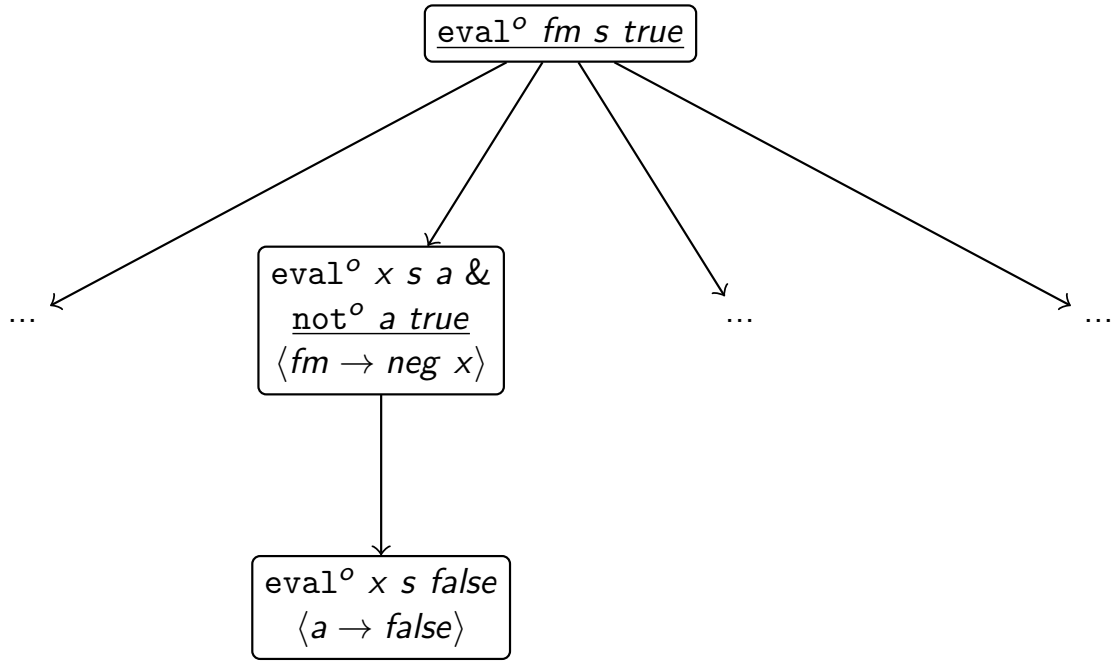
Unfolding of Boolean Connectives



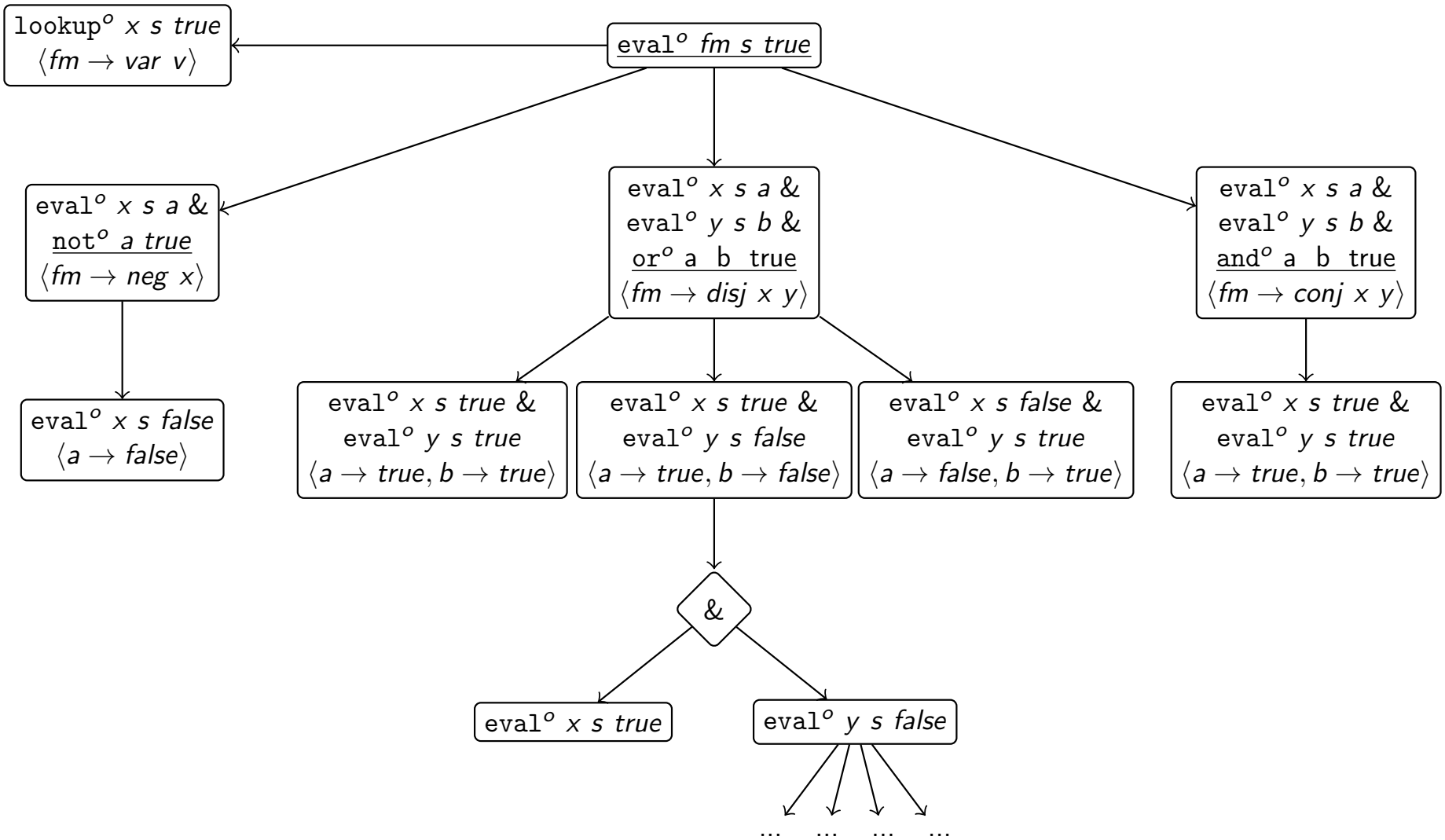




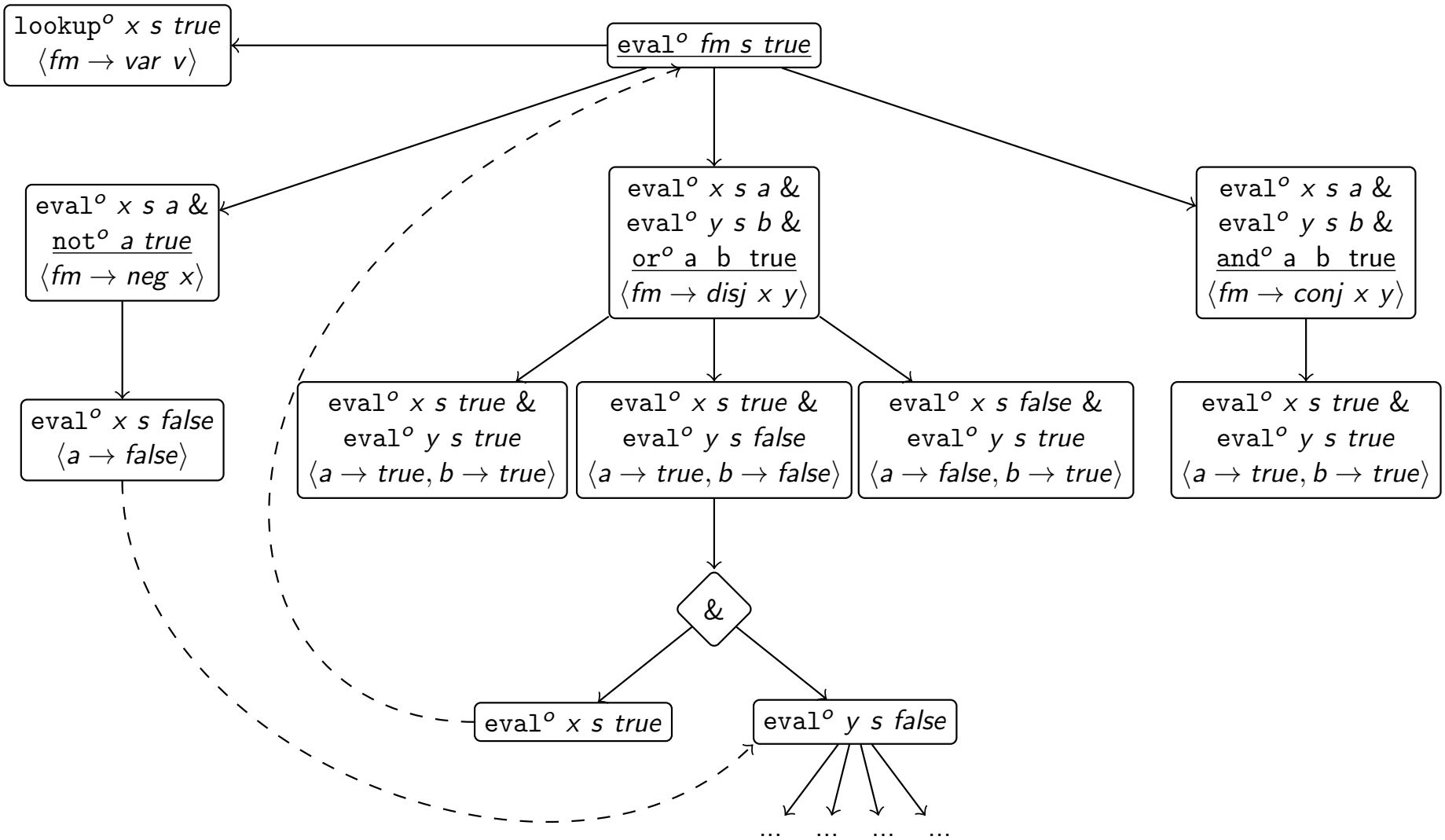
Unfolding Boolean Connectives First



Evaluator of Logic Formulas: Conservative PD



Evaluator of Logic Formulas: Conservative PD



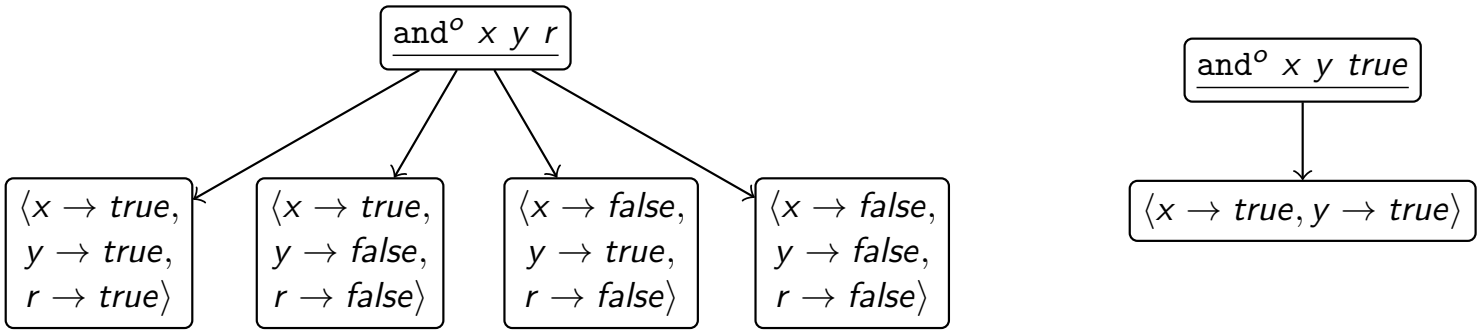
Conservative Partial Deduction

- 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

Evaluation: Comparison with ECCE

ECCE is a partial deduction system for Prolog

To compare to ECCE we did the following steps:

- Convert miniKanren program into Prolog
- Run the default transformation with ECCE
- Convert generated Prolog back into miniKanren

Evaluator of Logic Formulas: Order of Calls

boolean connective last

```
let rec eval^o fm s r =
  ocanren { fresh v x y a b in
    (fm ≡ var v & lookup^o v s r) |
    (fm ≡ neg x & eval^o x s a & not^o a r) |
    (fm ≡ conj x y & eval^o x s a & eval^o y s b & and^o a b r) |
    (fm ≡ disj x y & eval^o x s a & eval^o y s b & or^o a b r) }
```

Evaluator of Logic Formulas: Order of Calls

boolean connective last

```
let rec eval^o fm s r =
  ocanren { fresh v x y a b in
    (fm ≡ var v & lookup^o v s r) |
    (fm ≡ neg x & eval^o x s a & not^o a r) |
    (fm ≡ conj x y & eval^o x s a & eval^o y s b & and^o a b r) |
    (fm ≡ disj x y & eval^o x s a & eval^o y s b & or^o a b r) }
```

boolean connective first

```
let rec eval^o fm s r =
  ocanren { fresh v x y a b in
    (fm ≡ var v & lookup^o v s r) |
    (fm ≡ neg x & not^o a r & eval^o x s a) |
    (fm ≡ conj x y & and^o a b r & eval^o x s a & eval^o y s b) |
    (fm ≡ disj x y & or^o a b r & eval^o x s a & eval^o y s b) }
```

Evaluator of Logic Formulas: Compexity of Relations

table-based implementation

```
let rec and^o 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 and^o 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) }
```

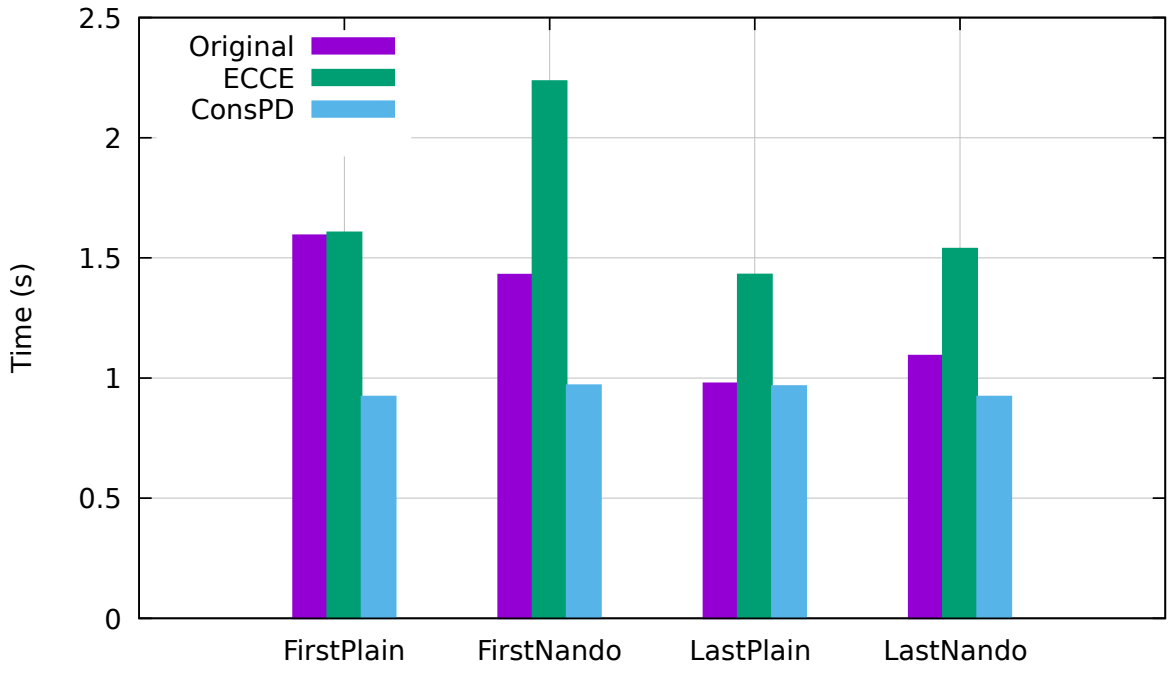
implementation via nand^o

```
let 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) }
```

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



Query: find 1000 formulas which evaluate to true

Typechecker-Term Generator: Language

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

Figure: Language syntax

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

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

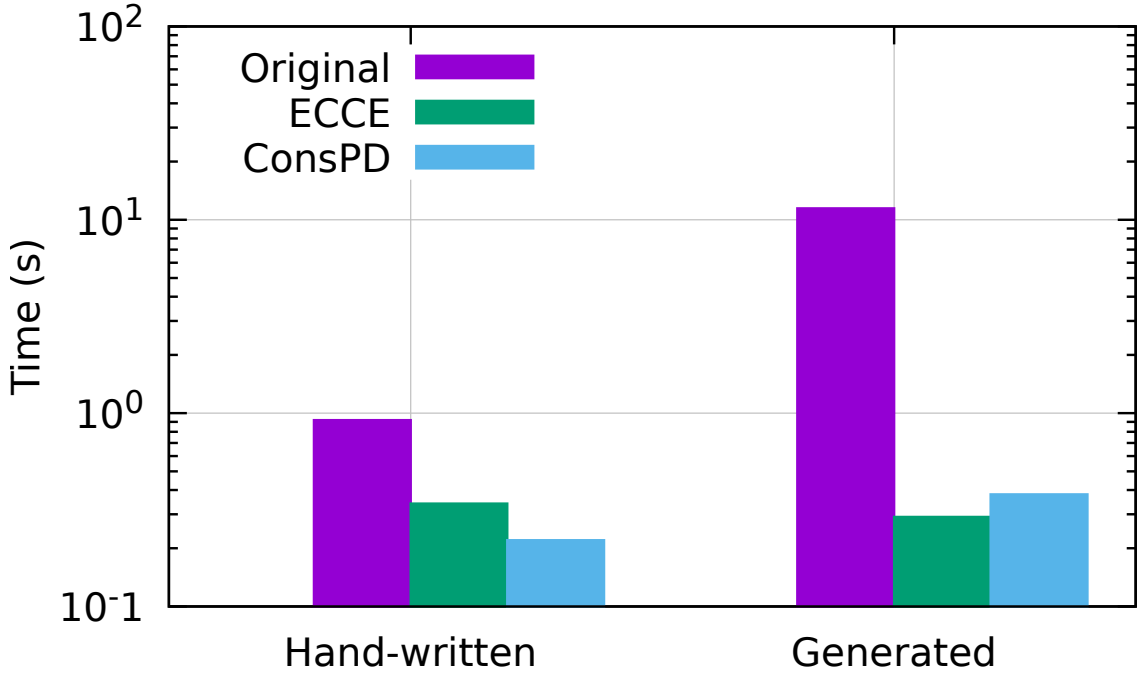
$$
\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t * s : Int} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t < s : Bool} \qquad \frac{\Gamma \vdash c : Bool, \Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \underline{if}\ c\ \underline{then}\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

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

Partial deduction changes the order of answers

Measuring time when order is different does not make much sense

Partial deduction reduces the number of unifications needed to compute an answer

Discussion: Deterministic Unfolding and Tupling

ConsPD often splits too much failing to do tupling

Because of the deterministic unfolding, ECCE fails to tuple `maxmin`

```
max([],M,M).
max([H|T],N,M) :- H =< N, max(T,N,M).
max([H|T],N,M) :- H > N, max(T,H,M).

min([],M,M).
min([H|T],N,M) :- H > N, min(T,N,M).
min([H|T],N,M) :- H =< N, min(T,H,M).

maxmin([H],H,H).
maxmin([H|T],Max,Min) :- max(T,H,Max),min(T,H,Min).
```

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