



# 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

27.08.2020

# 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 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 |  
...
```

# 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 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 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 |  
...
```

...

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 |  
  ...
```

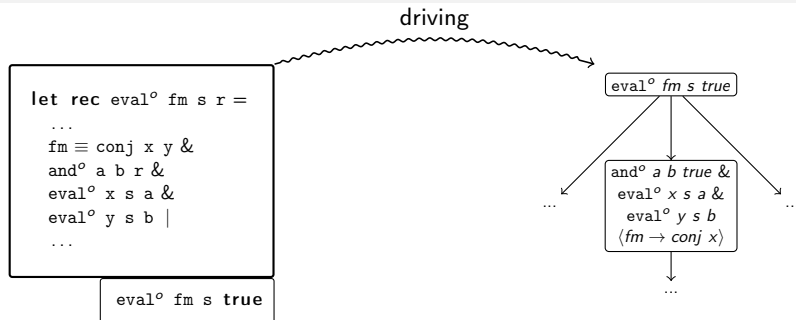
# Partial Deduction for MINIKANREN: Bird's-eye View

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

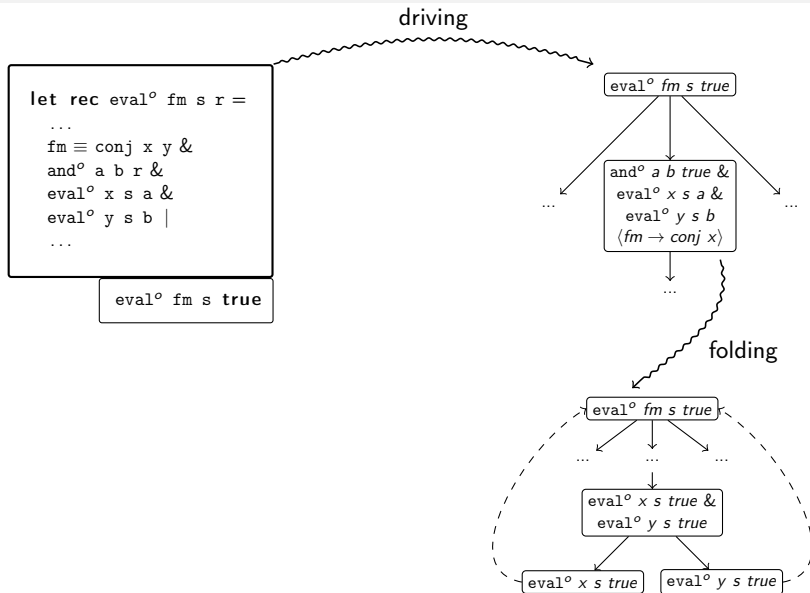
```
evalo fm s true
```



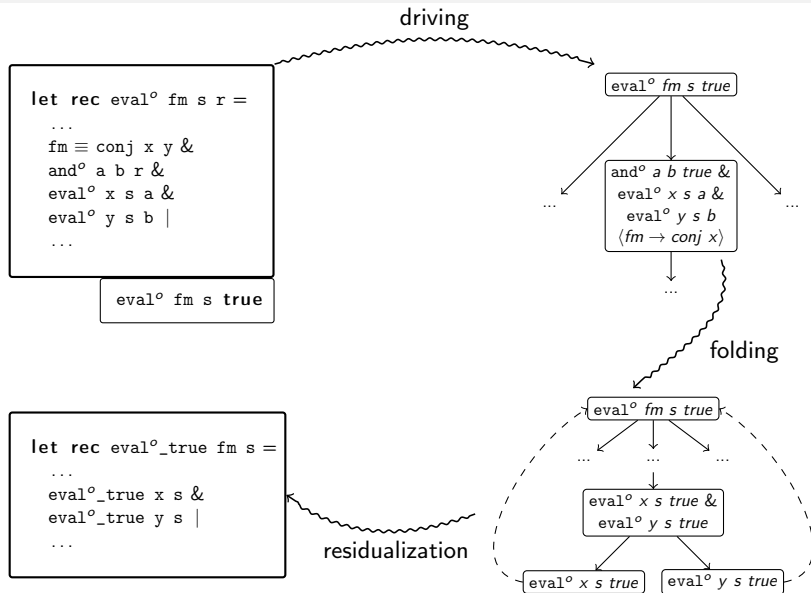
# Partial Deduction for MINIKANREN: Bird's-eye View



# Partial Deduction for MINIKANREN: Bird's-eye View



# 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 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) }

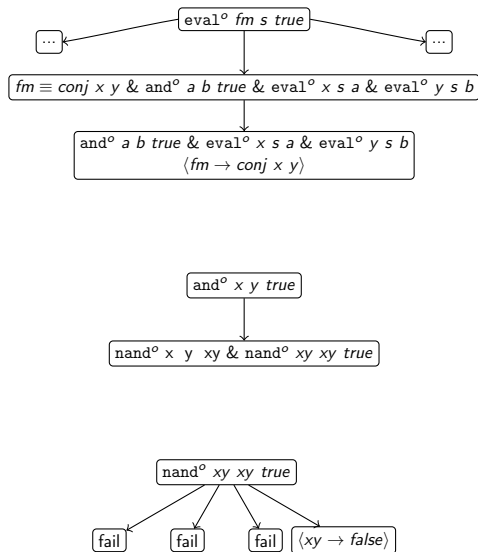
let rec nando x y r =
  ocanren {
    (x ≡ true & y ≡ true & r ≡ false) |
    (x ≡ true & y ≡ false & r ≡ true) |
    (x ≡ false & y ≡ true & r ≡ true) |
    (x ≡ false & y ≡ false & r ≡ true) }
  
```

eval<sup>o</sup> fm s true

goal

and<sup>o</sup> a b true & eval<sup>o</sup> x s a & eval<sup>o</sup> y s b  
 ⟨fm → conj x y⟩

substitution



# Partial Deduction

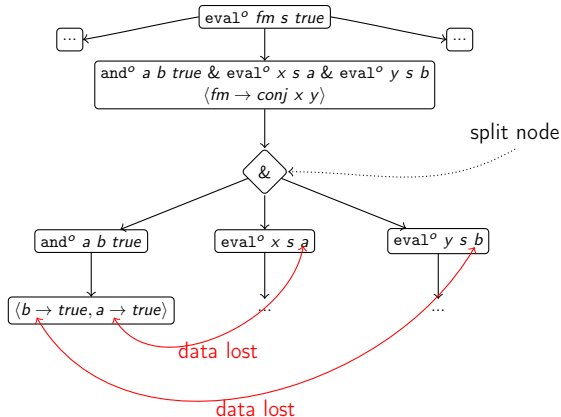
```
let rec evalo fm s r =  
  ...  
  fm ≡ conj x y & ando a b r &  
  evalo x s a & evalo y s b |  
  ...
```

```
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 |  
  ...
```

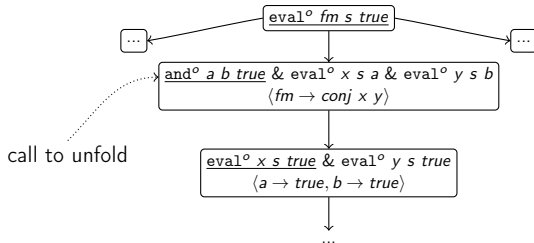
eval<sup>o</sup> fm s true



# 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 |  
  ...
```

eval<sup>o</sup> fm s true





# CPD: Split is Necessary

```
let rec evalo fm s r =
```

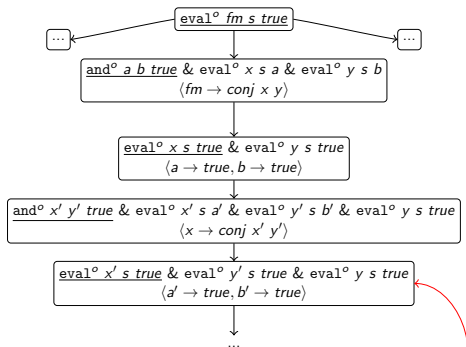
```
...
```

```
fm ≡ conj x y & ando a b r &
```

```
evalo x s a & evalo y s b |
```

```
...
```

```
evalo fm s true
```



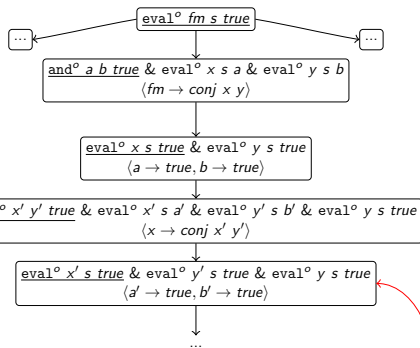
uncontrollable growth

# CPD: Split is Necessary

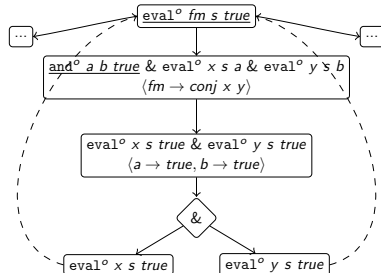
```
let rec evalo fm s r =
```

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

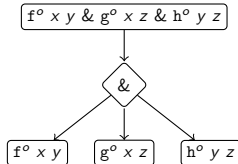
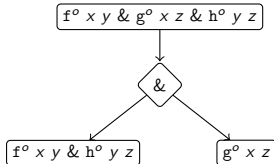
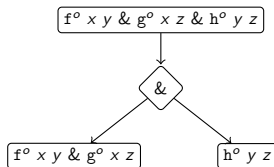
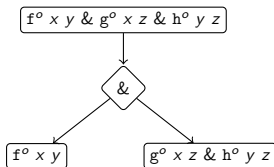
```
evalo fm s true
```



uncontrollable growth



# Split: Which Way is the Right Way?



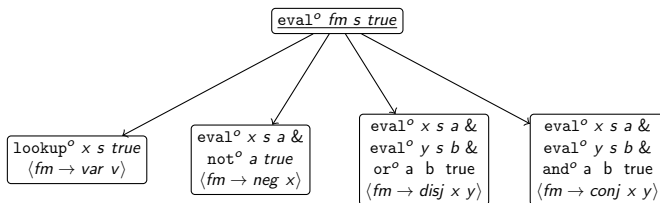
# Decisions in Partial Deduction

- 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

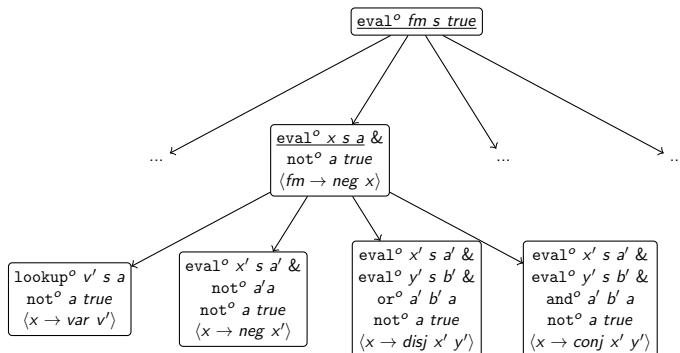
# Evaluator of Logic Formulas: Unfolding Step 1

```
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 & oro a b r) }
```

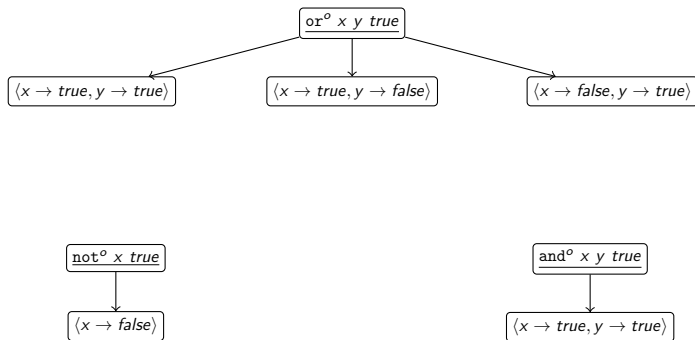
eval<sup>o</sup> fm s true



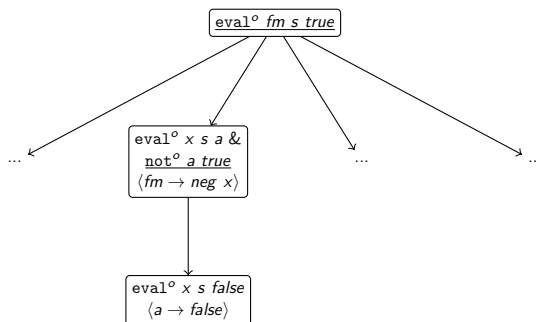
# Evaluator of Logic Formulas: Unfolding Step 2



# Unfolding of Boolean Connectives

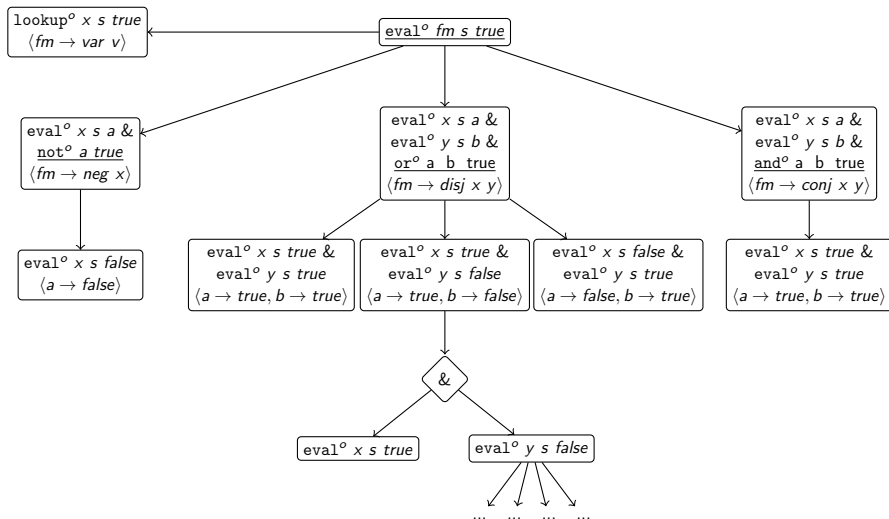


# Unfolding Boolean Connectives First

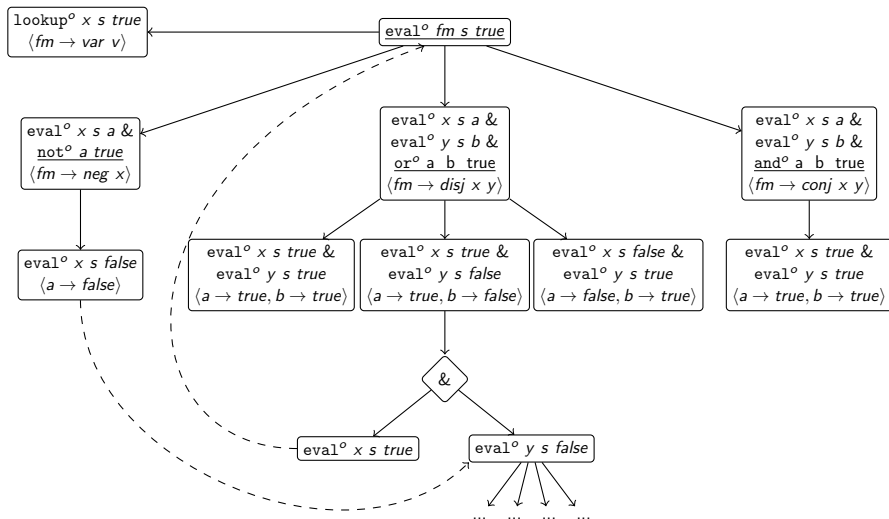




# Evaluator of Logic Formulas: Conservative PD



# Evaluator of Logic Formulas: Conservative PD



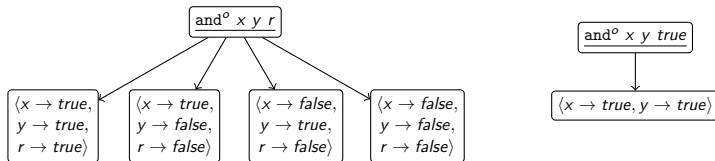
# Conservative Partial Deduction

- 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 constructed 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



We implemented the Conservative Partial Deduction and compared it with CPD for `MINIKANREN` and CPD with branching heuristics on the following relations

- Two implementations of an evaluator of logic formulas
- A program to compute a unifier of two terms
- A program to search for paths of a specific length in a graph

# 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 & oro 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: Complexity 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: Complexity 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<sup>o</sup>

```
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

Implementations:

- *last*: boolean connectives last, implemented via `nand`<sup>o</sup>
- *plain*: boolean connectives first, straightforward implementation

Query: find 1000 formulas which evaluate to true

	last	plain
Original	1.06s	1.84s
CPD	—	1.13s
Branching	3.11s	7.53s
ConsPD	0.93s	0.99s

Table: Evaluation results

Relation to find a unifier of two terms

Query: unification of terms  $f(X, X, g(Z, t))$  and  $f(g(p, L), Y, Y)$

Relation to search for paths in a graph

Query: find 5 paths in a graph with 20 vertices and 30 edges

# Evaluation Results

	last	plain	unify	isPath
Original	1.06s	1.84s	—	—
CPD	—	1.13s	14.12s	3.62s
Branching	3.11s	7.53s	3.53s	0.54s
ConsPD	0.93s	0.99s	0.96s	2.51s

Table: Evaluation results

# 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