



# 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

# Specialization: a Method to Improve Programs

input program

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

# Specialization: a Method to Improve Programs

input program

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

known argument

```
evalo fm s true ←
```

# Specialization: a Method to Improve Programs

input program

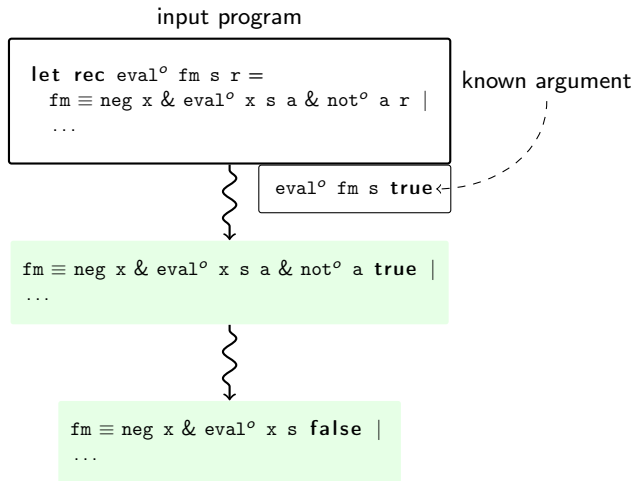
```
let rec evalo fm s r =  
  fm ≡ neg x & evalo x s a & noto a r |  
  ...
```

known argument

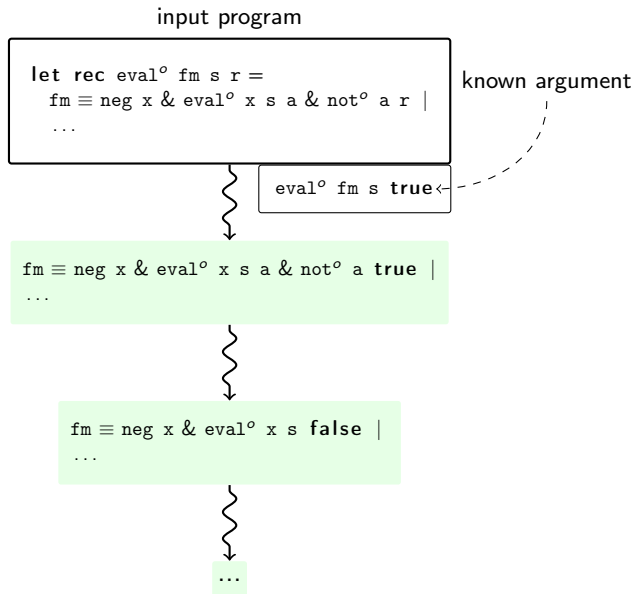
```
evalo fm s true ←
```

```
fm ≡ neg x & evalo x s a & noto a true |  
...
```

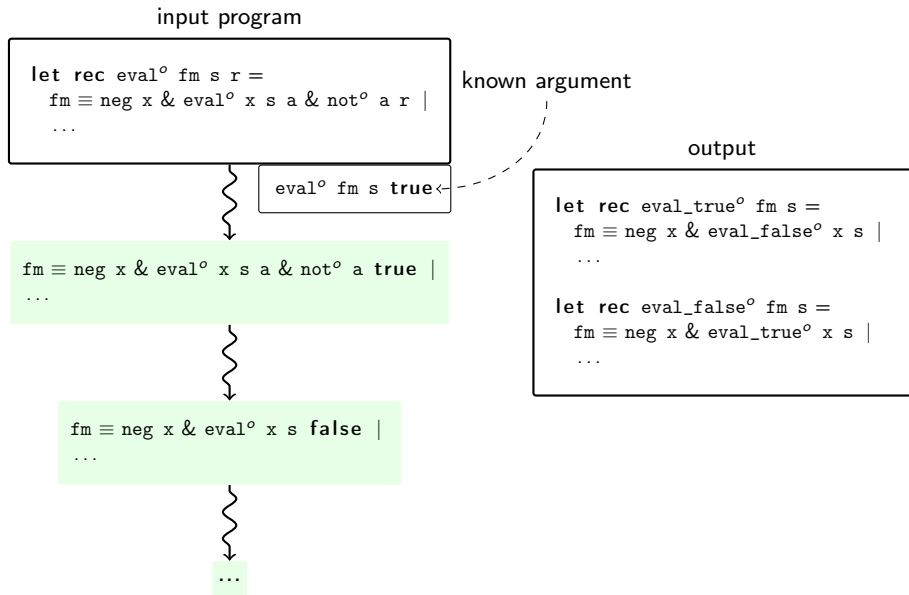
# Specialization: a Method to Improve Programs



# Specialization: a Method to Improve Programs



# Specialization: a Method to Improve Programs



# Partial Deduction: Specialization for Logic Programming

input

```
let double_appendo x y z r =  
  ocanren {  
    fresh t in  
      appendo x y t &  
      appendo t z r}  
  
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'))}
```

double\_append<sup>o</sup> x y z r

output

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

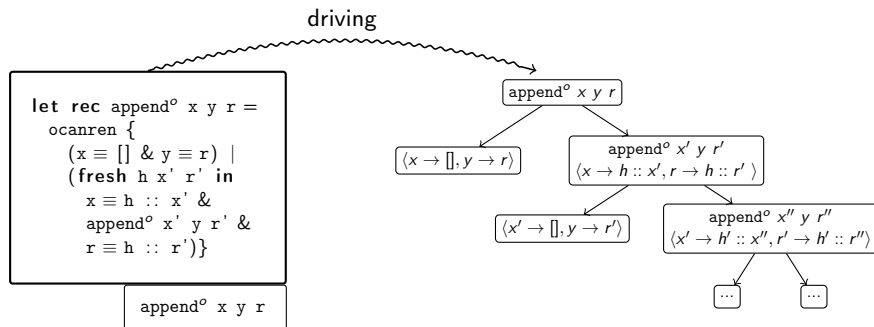


# Partial Deduction for MINIKANREN: Bird's-eye View

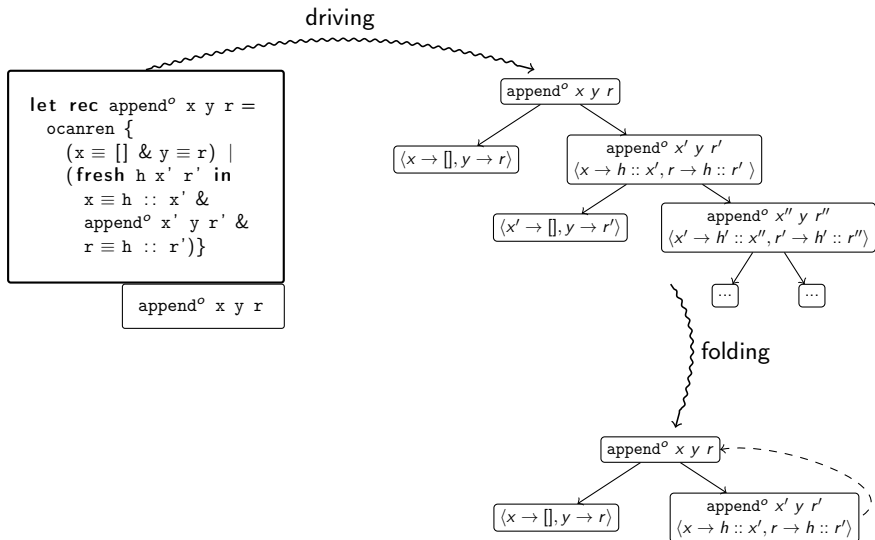
```
let rec appendo x y r =  
  ocanren {  
    (x ≡ [] & y ≡ r) |  
    (fresh h x' r' in  
      x ≡ h :: x' &  
      appendo x' y r' &  
      r ≡ h :: r'))}
```

```
appendo x y r
```

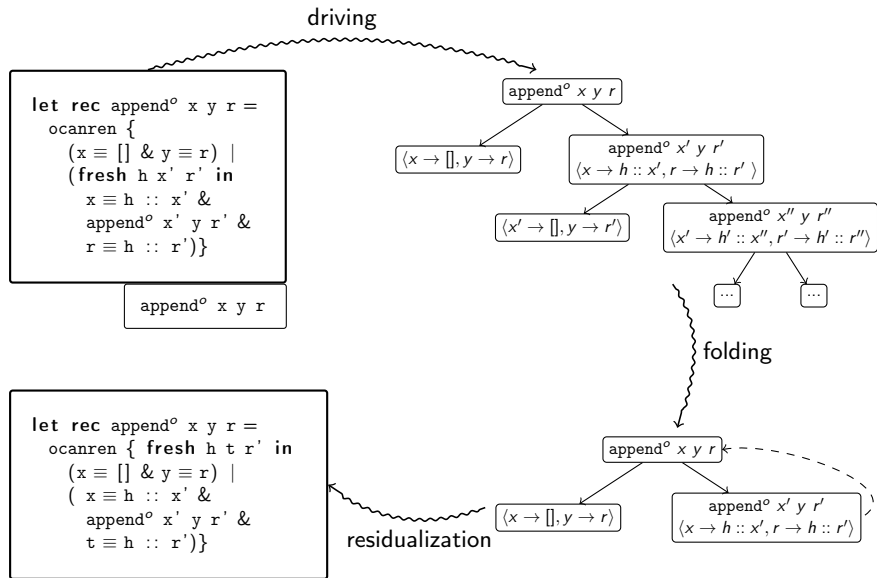
# Partial Deduction for MINIKANREN: Bird's-eye View



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



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



# Driving: Unfolding

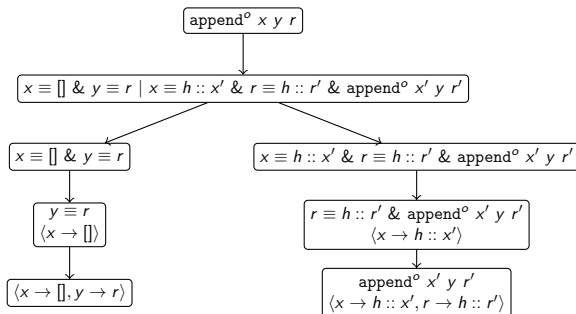
```
let rec appendo x y r =  
  ocanren {  
    (x  $\equiv$  [] & y  $\equiv$  r) |  
    (fresh h x' r' in  
      x  $\equiv$  h :: x' &  
      appendo x' y r' &  
      r  $\equiv$  h :: r')}
```

```
appendo x y r
```

# Driving: Unfolding

```
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') }  
  }
```

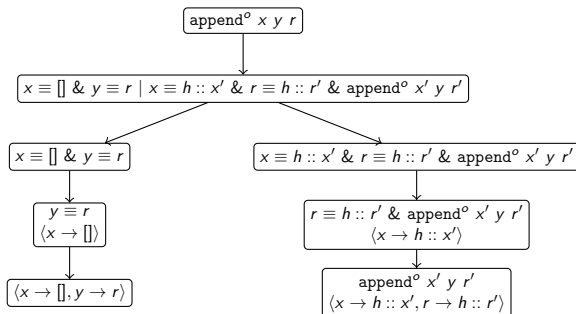
append<sup>o</sup> x y r



# Driving: Unfolding

```
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') }  
  }
```

append<sup>o</sup> x y r



substitution

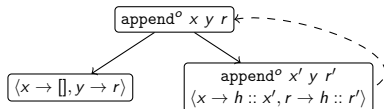
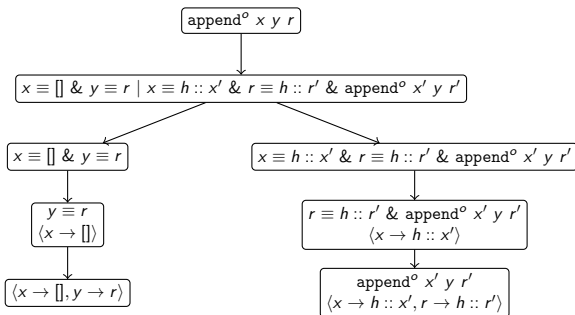
goal

append<sup>o</sup> x' y r'  
⟨x → h :: x', r → h :: r'⟩

# Driving: Unfolding

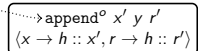
```
let rec appendo x y r =
  ocanren {
    (x ≡ [] & y ≡ r) |
    (fresh h x' r' in
      x ≡ h :: x' &
      appendo x' y r' &
      r ≡ h :: r')}
```

append<sup>o</sup> x y r



substitution

goal

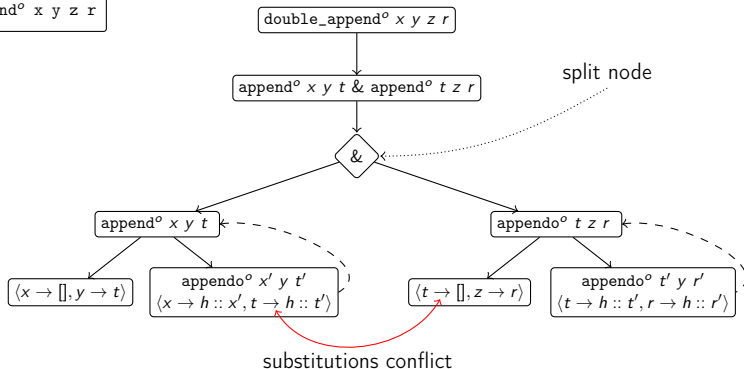




# Partial Deduction

```
let double_appendo x y z r =  
  ocanren {  
    fresh t in  
      appendo x y t &  
      appendo t z r}
```

```
double_appendo x y z r
```

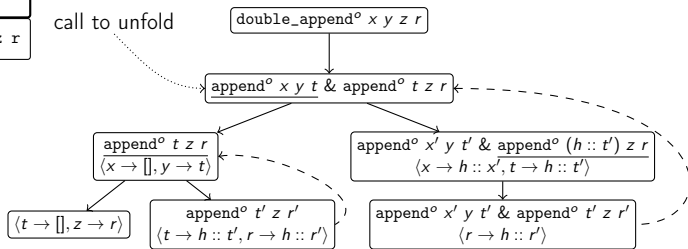


# Conjunctive Partial Deduction

```
let double_appendo x y z r =  
  ocanren {  
    fresh t in  
      appendo x y t &  
      appendo t z r}
```

double\_append<sup>o</sup> x y z r

call to unfold



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

# CPD: Split is Necessary

```
let rec reverseo xs sx =  
  ocanren {  
    (xs ≡ [] & sx ≡ []) |  
    (fresh h t t' in  
     xs ≡ h :: t &  
     reverseo t t' &  
     appendo t' [h] sx)
```

reverse<sup>o</sup> xs sx

reverse<sup>o</sup> xs sx

reverse<sup>o</sup> t t' & append<sup>o</sup> t' [h] sx  
⟨xs → h :: t⟩

reverse<sup>o</sup> s s' & append<sup>o</sup> s' [h'] t' & append<sup>o</sup> t' [h] sx  
⟨t → h' :: s⟩

...

reverse<sup>o</sup> xs sx

&

reverse<sup>o</sup> t t'  
⟨xs → h :: t⟩

append<sup>o</sup> t' [h] xs  
⟨xs → h :: t⟩

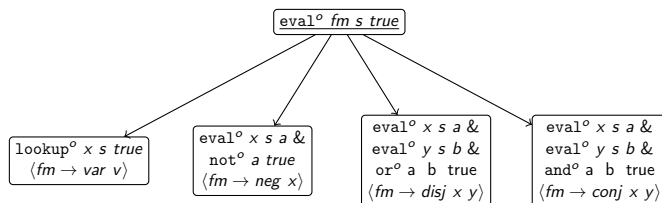
# Decisions in Partial Deduction

- 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

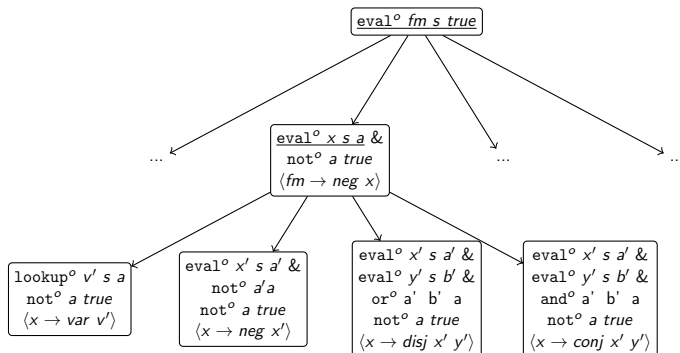
# Evaluator of Logic Formulas: Unfolding Step 1

```
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)
```

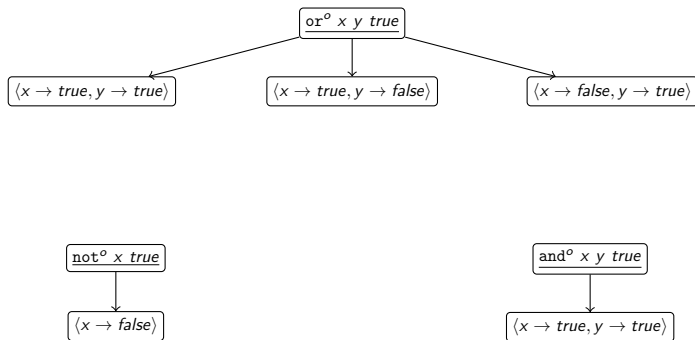
eval<sup>o</sup> fm s true



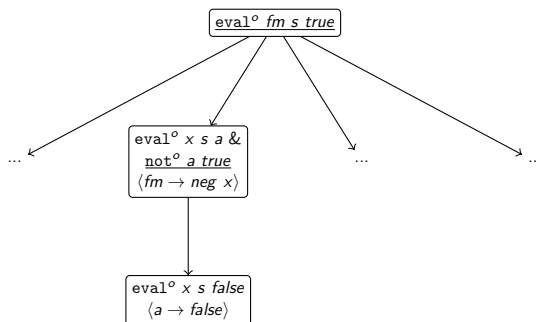
# Evaluator of Logic Formulas: Unfolding Step 2



# Unfolding of Boolean Connectives

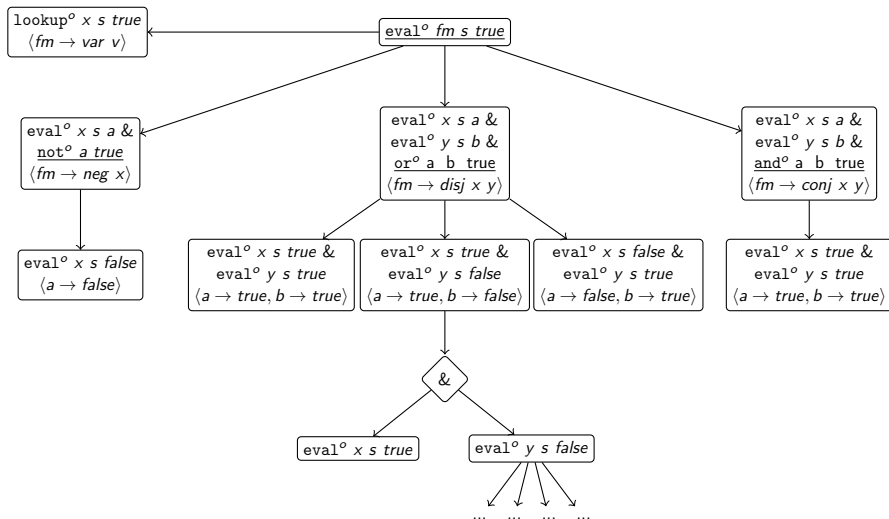


# Unfolding Boolean Connectives First

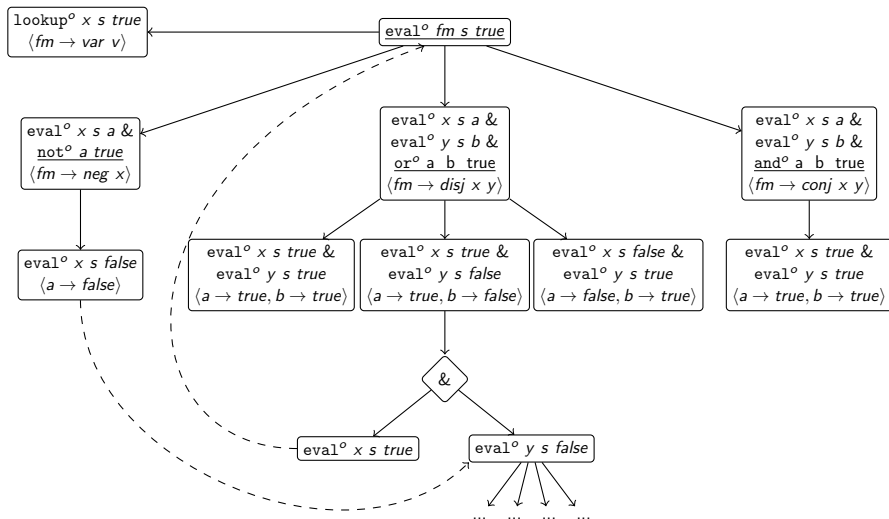




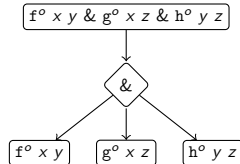
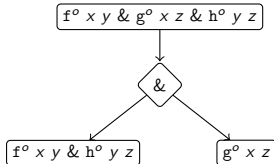
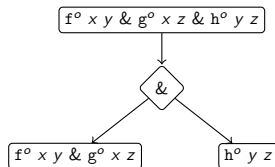
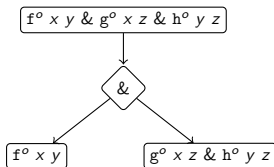
# Evaluator of Logic Formulas: Conservative PD



# Evaluator of Logic Formulas: Conservative PD



# Split: Which Way is the Right Way?



# 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 state (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 the call has less branches in the process tree than the relation can possible have, unfold the call



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

```
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

```
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)
```

```
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: Complexity of Relations

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

```
let oro x y r =  
  ocanren {  
    fresh a b in  
      (nando x x a & nando y y b & nando a b 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 ≡ false) }
```

# Evaluator of Logic Formulas: Evaluation

## Implementations:

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

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

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
ConsPD	0.93s	0.99s	0.96s	2.51s
Branching	3.11s	7.53s	3.53s	0.54s

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