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

input program

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

known argument

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

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

`fm` $\equiv$ `neg x` & $\texttt{eval}^o$ `x s` **false** |
...

Specialization: a Method to Improve Programs

input program

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

known argument

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

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

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

...

Specialization: a Method to Improve Programs

input program

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

known argument

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

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

```
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: 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_appendᵒ 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')}
```

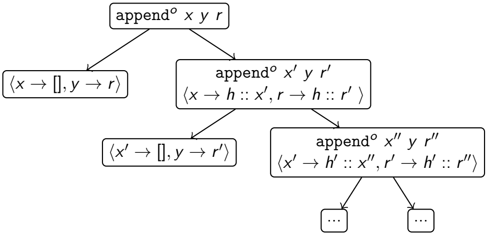
```
appendᵒ x y r
```

Partial Deduction for miniKanren: Bird's-eye View

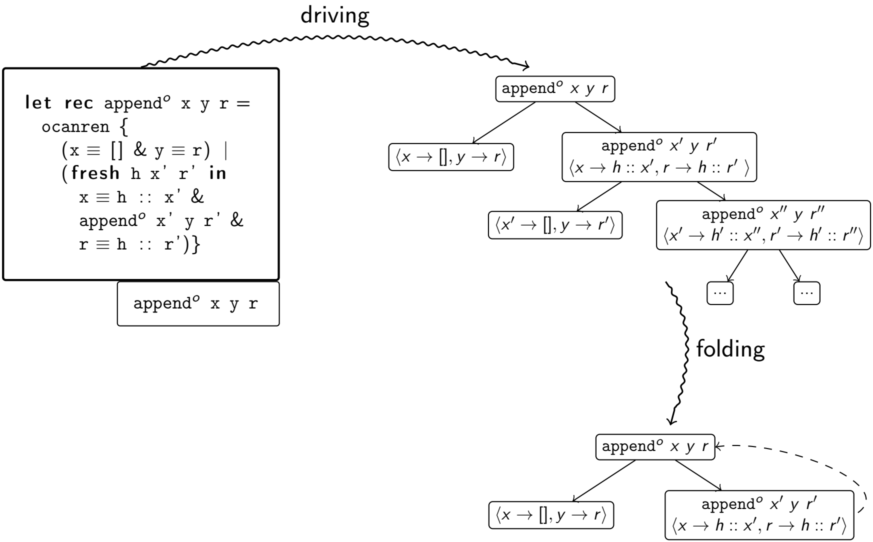
driving

```
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')}
```

```
append^o x y r
```



Partial Deduction for miniKanren: Bird's-eye View

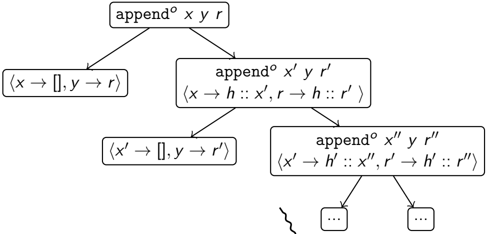


Partial Deduction for miniKanren: Bird's-eye View

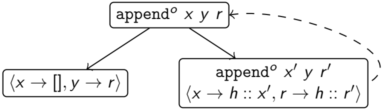
driving

```
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
```



folding



residualization

```
let rec appendᵒ x y r =
  ocanren { fresh h t r' in
    (x ≡ [] & y ≡ r) |
    (x ≡ h :: x' &
     t ≡ h :: r' &
     appendᵒ x' y r')}
```

Driving: Unfolding

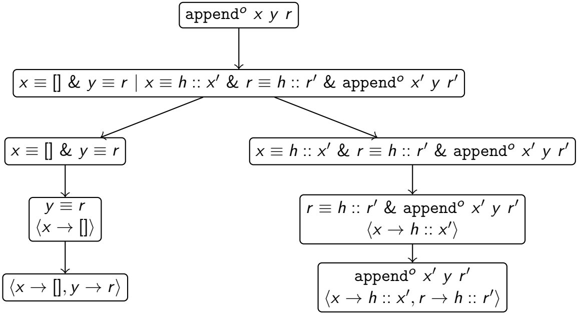
```
let rec appendᵒ x y r =
  ocanren {
    (x ≡ [] & y ≡ r) |
    (fresh h x' r' in
      x ≡ h :: x' &
      appendᵒ x' y r' &
      r ≡ h :: r')}
```

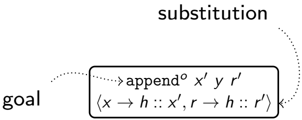
```
appendᵒ x y r
```

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')}
```

appendᵒ x y r

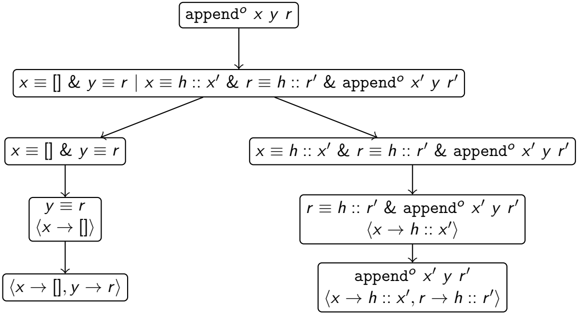


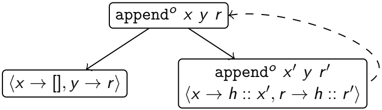


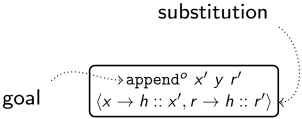
Driving: Unfolding

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

append° x y r



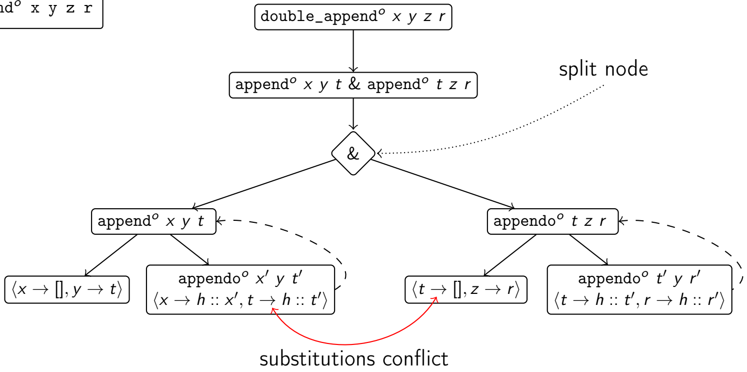




Partial Deduction

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

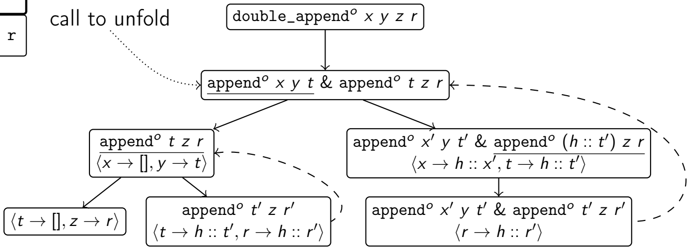
double_appendᵒ x y z r



Conjunctive Partial Deduction

```
let double_append^o x y z r =
  ocanren {
    fresh t in
      append^o x y t &
      append^o t z r}
```

```
double_append^o x y z r
```

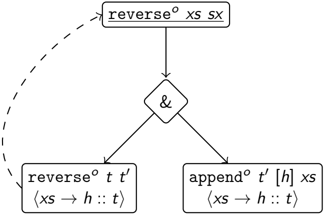


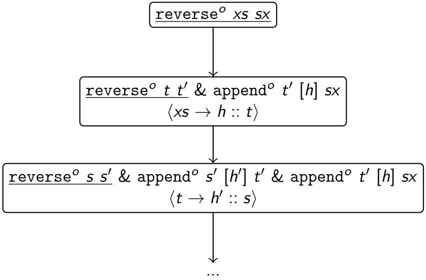

```
let double_append^o x y z r =
  ocanren {
    (x ≡ [] & append^o y z r) |
    (fresh h x' r' in
      x ≡ h :: x' &
      double_append^o x' y z r' &
      r ≡ h :: r')}
```

CPD: Split is Necessary

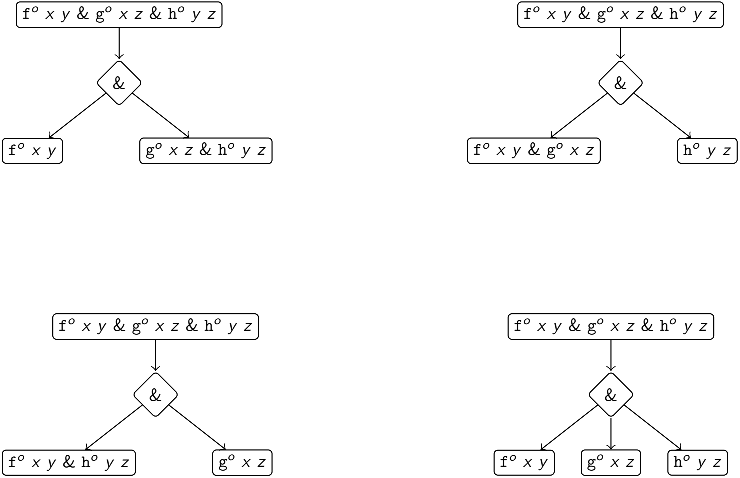
```
let rec reverseᵒ xs sx =
  ocanren {
    (xs ≡ [] & sx ≡ []) |
    (fresh h t t' in
      xs ≡ h :: t &
      reverseᵒ t t' &
      appendᵒ t' [h] sx}
```

reverse^o xs sx





Split: Which Way is the Right Way?



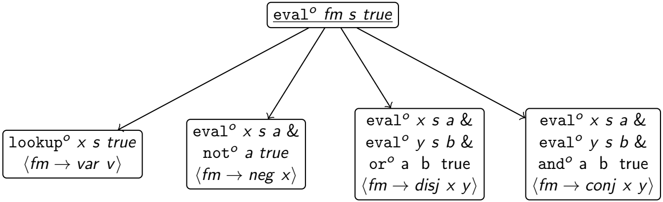
Decisions in Partial Deduction

- 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

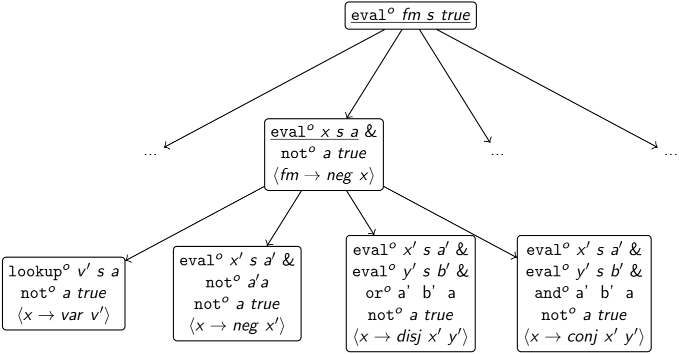
Evaluator of Logic Formulas: Unfolding Step 1

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

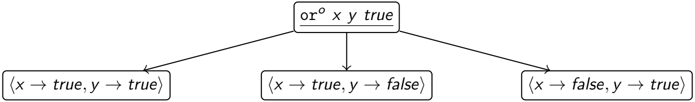
`eval^o fm s true`

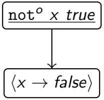


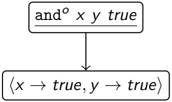
Evaluator of Logic Formulas: Unfolding Step 2



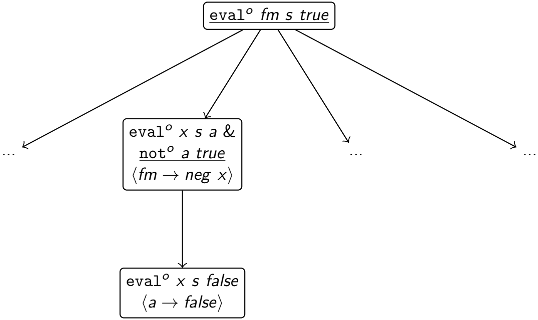
Unfolding of Boolean Connectives



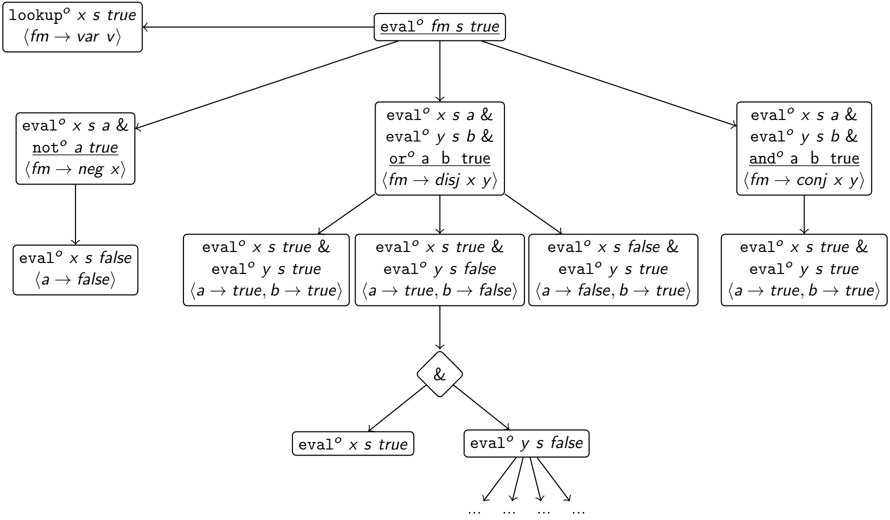




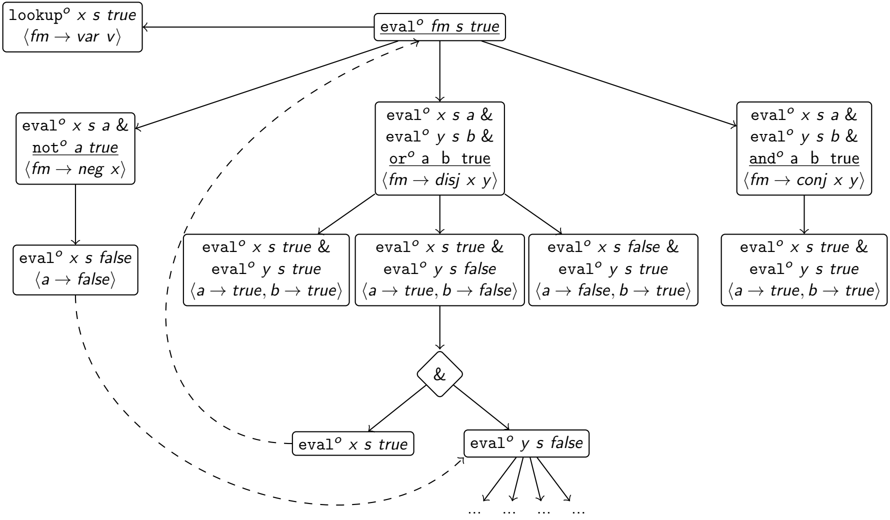
Unfolding Boolean Connectives First



Evaluator of Logic Formulas: Conservative PD



Evaluator of Logic Formulas: Conservative PD



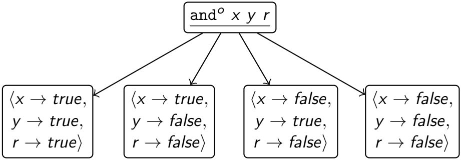
Conservative Partial Deduction

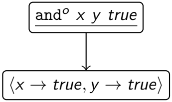
- Split conjunction into individual calls
- Unfold each call in isolation
- Unfold until embedding is encountered
- Find a call which narrows the search space (less-branching heuristics)
- Join the result of unfolding the selected call with the other calls not unfolded
- Continue driving the constucted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

Evaluator of Logic Formulas: Compexity of Relations

table-based implementation

```
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) }
```

implementation via nand^o

```
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 $\texttt{nand}^o$
- *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

Unification

Relation to find a unifier of two terms

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

Path Search

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