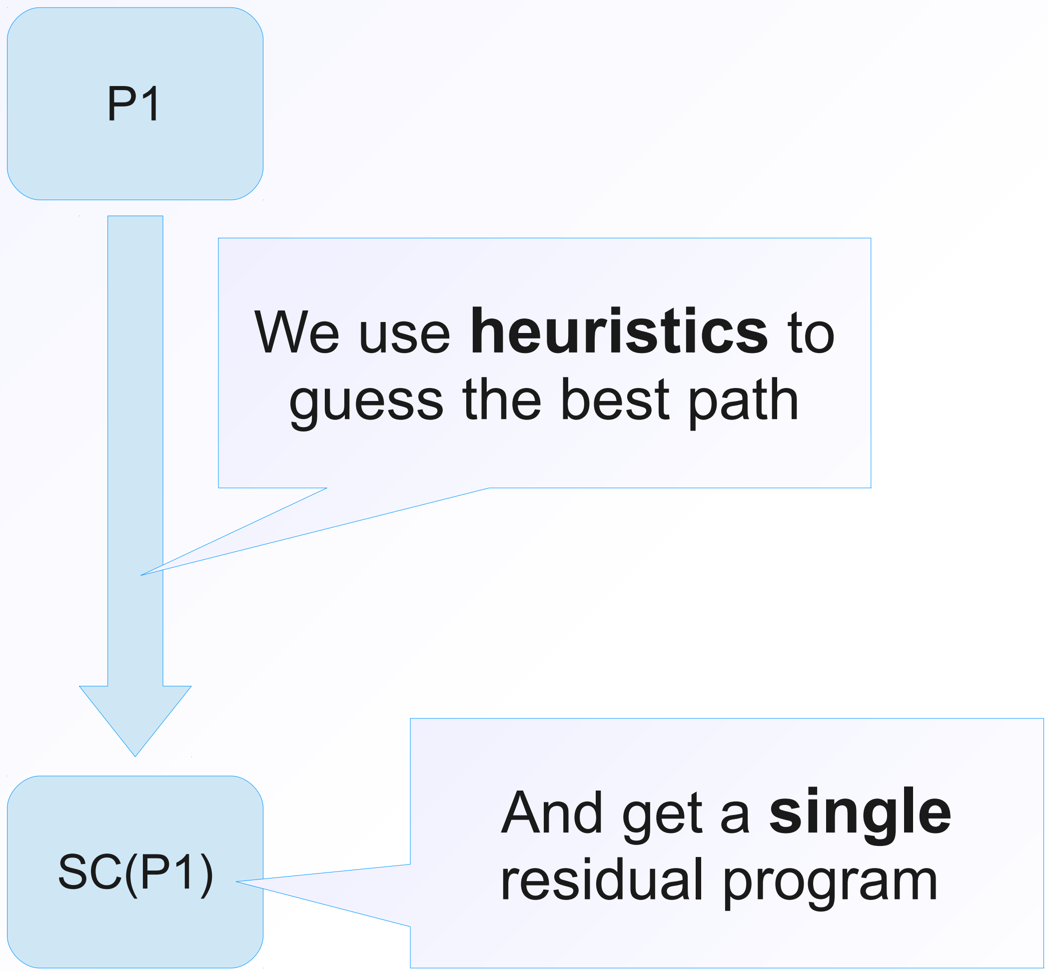
Overgraph Representation for Multi-Result Supercompilation

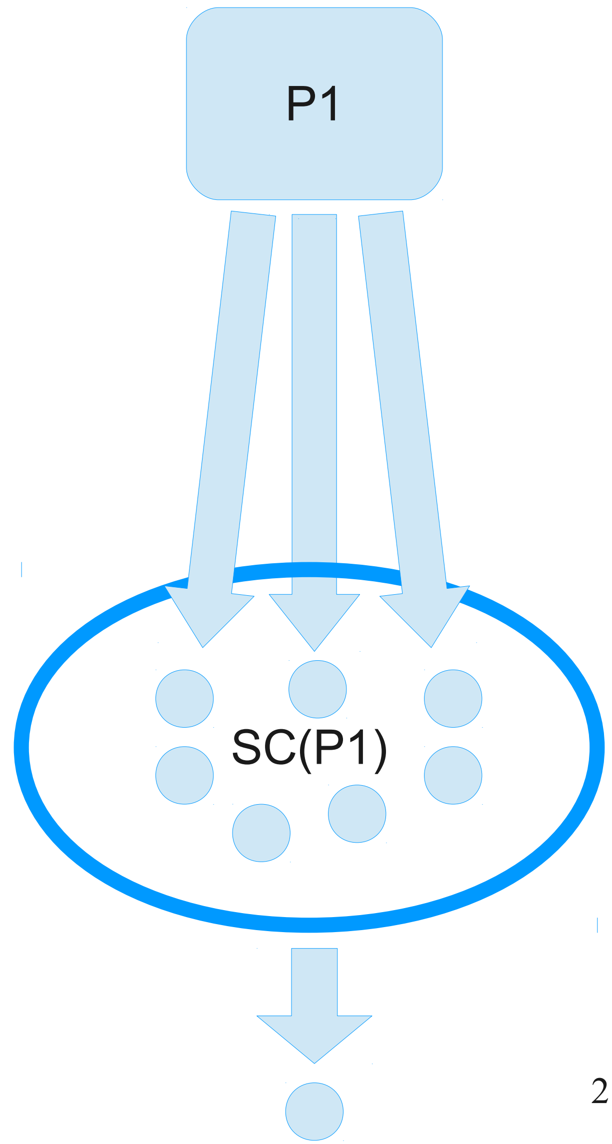
Sergei Grechanik

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Meta 2012

General Idea of Multi-Resultness





General Idea of Multi-Resultness

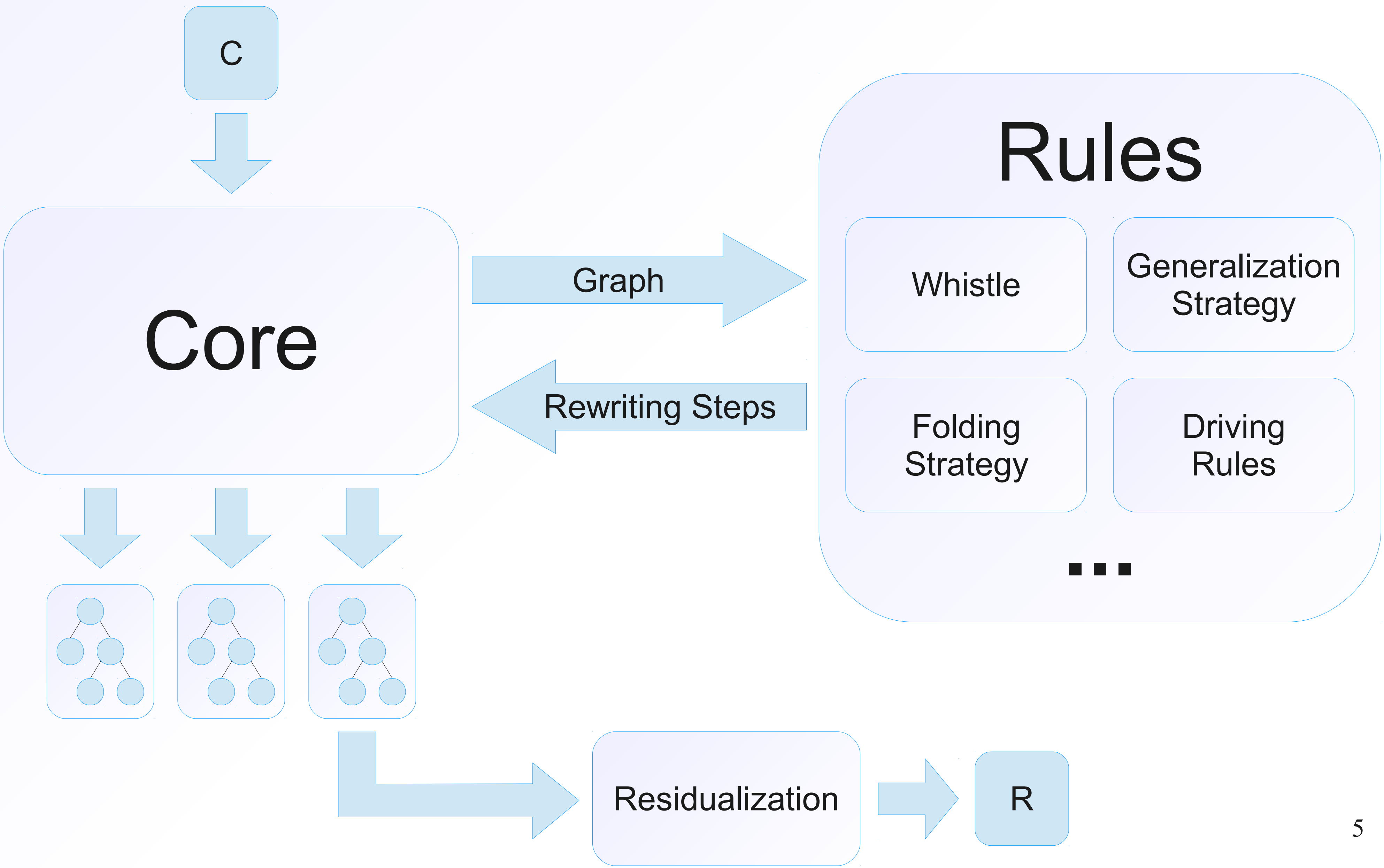
A problem

Millions of residual programs

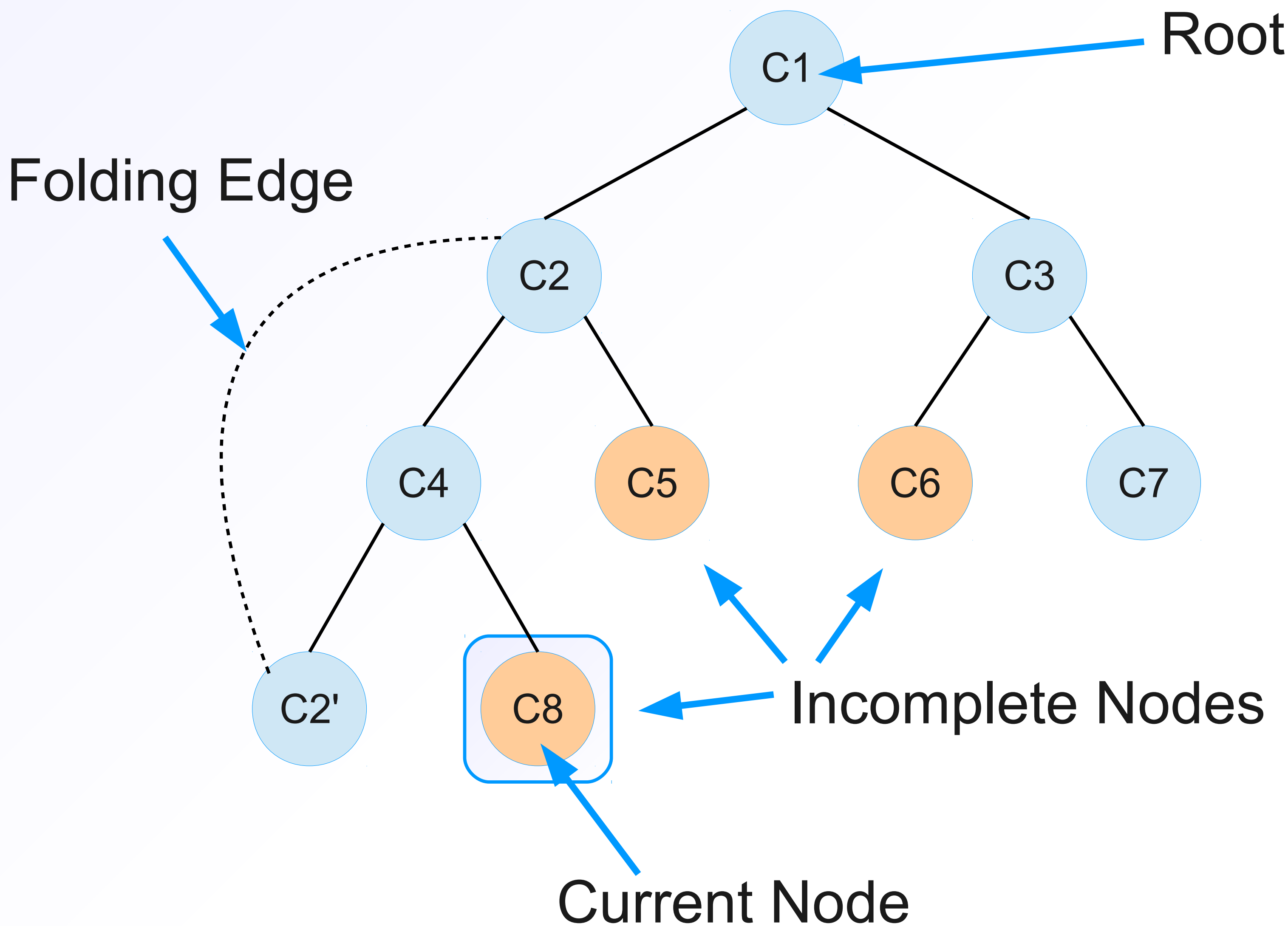
A solution

Overgraph – a compact representation for sets of graphs

MRSC Toolkit Architecture

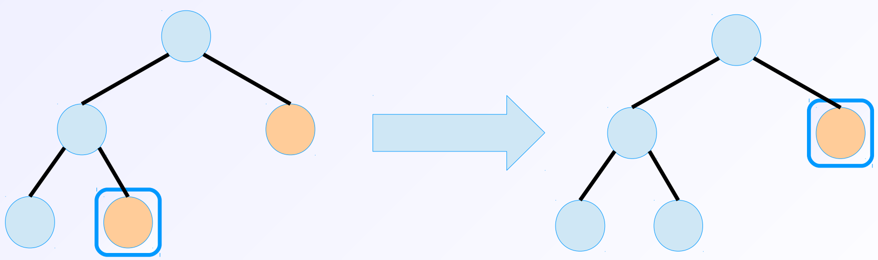


MRSC: Graphs of Configurations

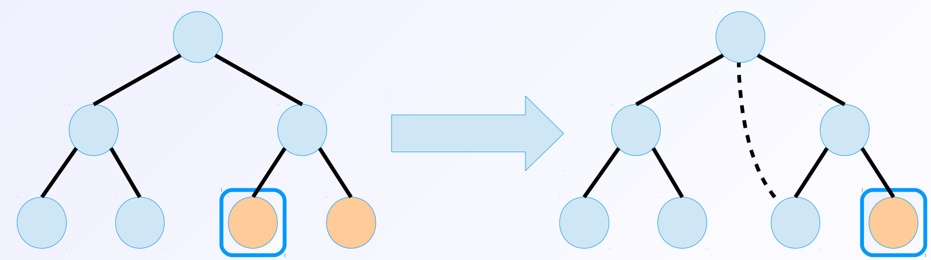


MRSC: Graph Rewriting Steps

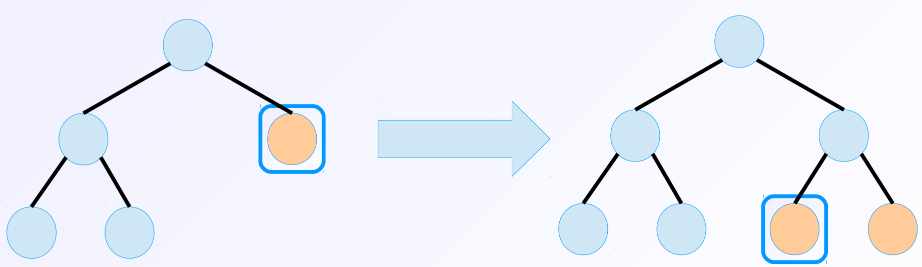
Complete



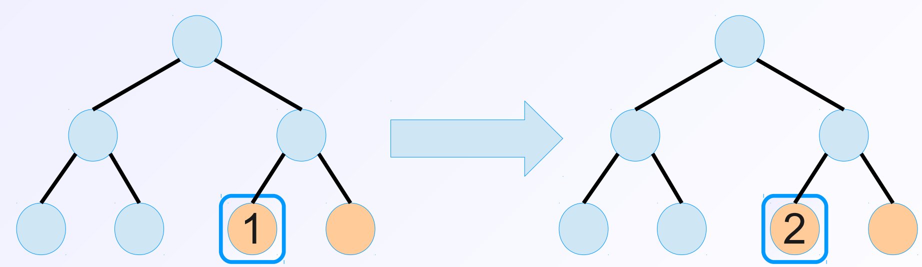
Fold



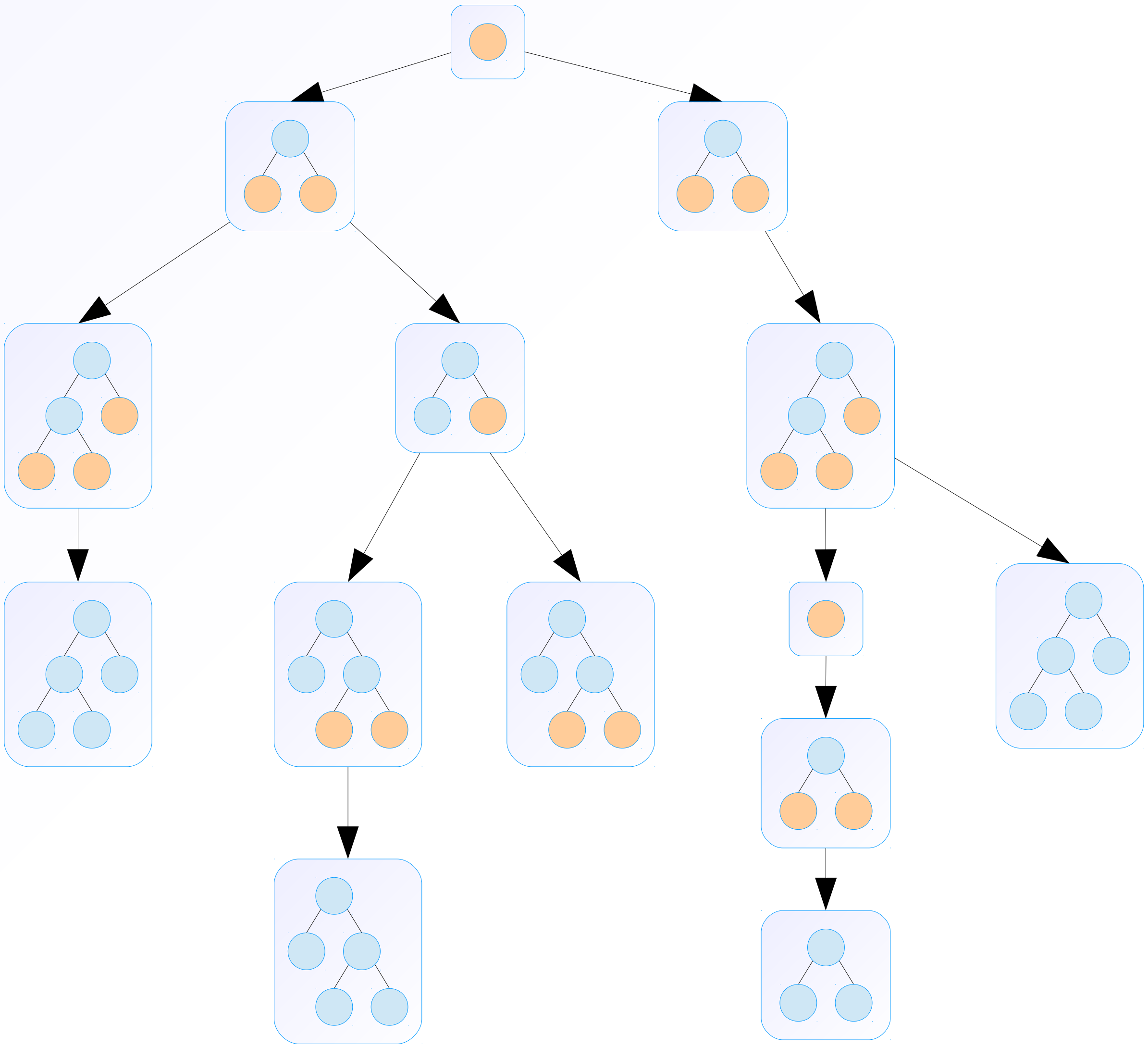
AddChildNodes



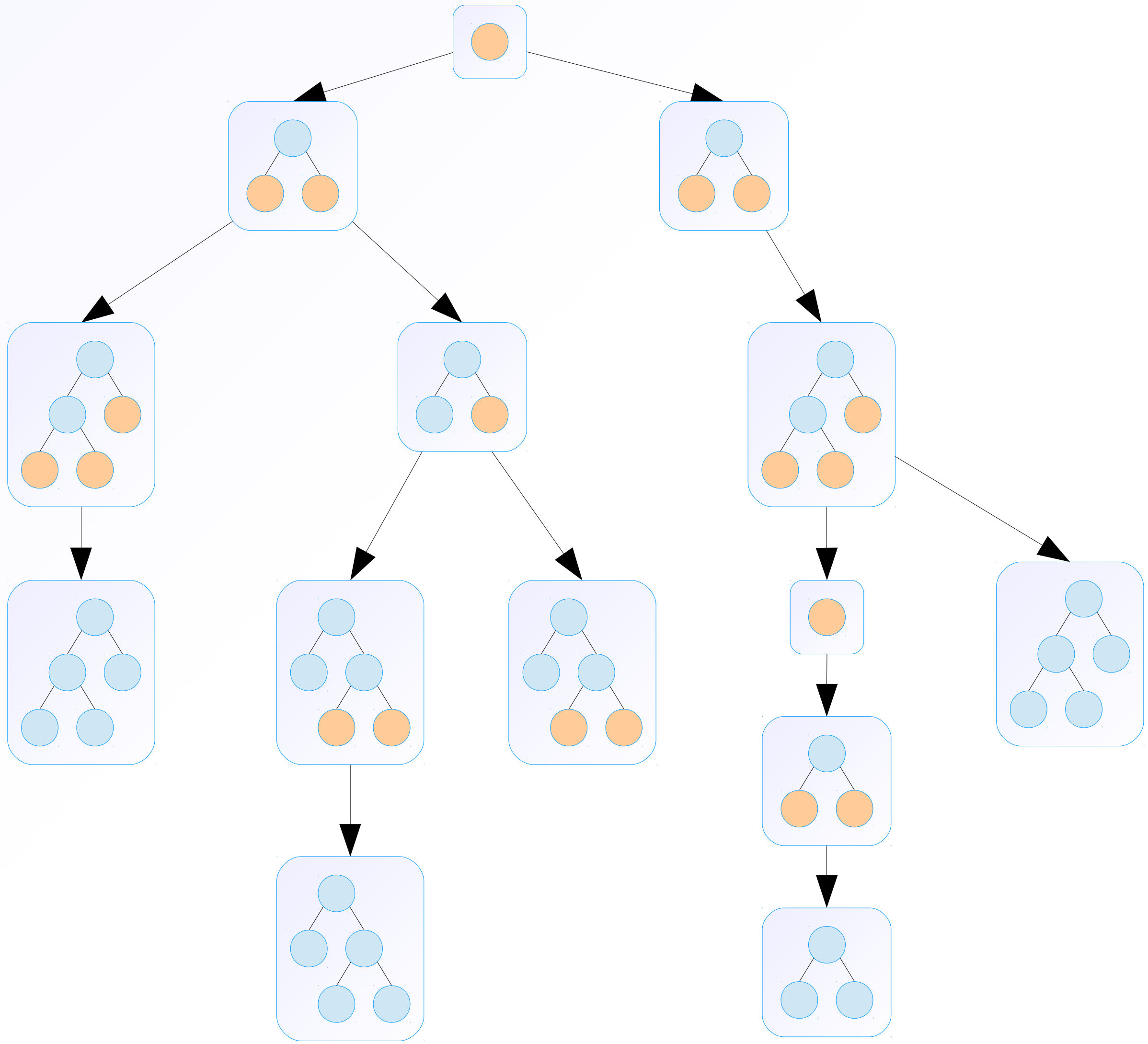
Rebuild



MRSC: Tree of Graphs

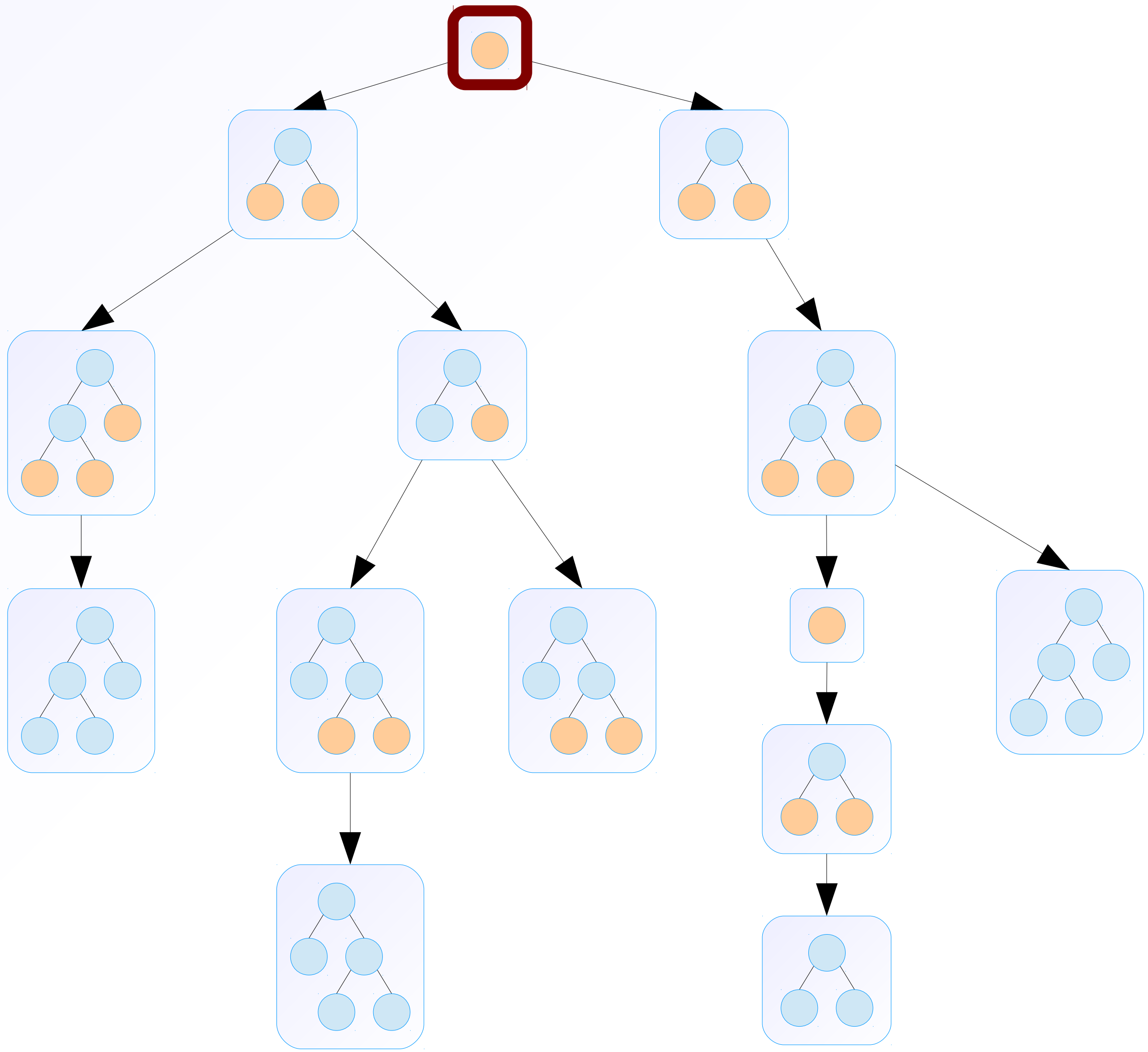


MRSC: Tree of Graphs



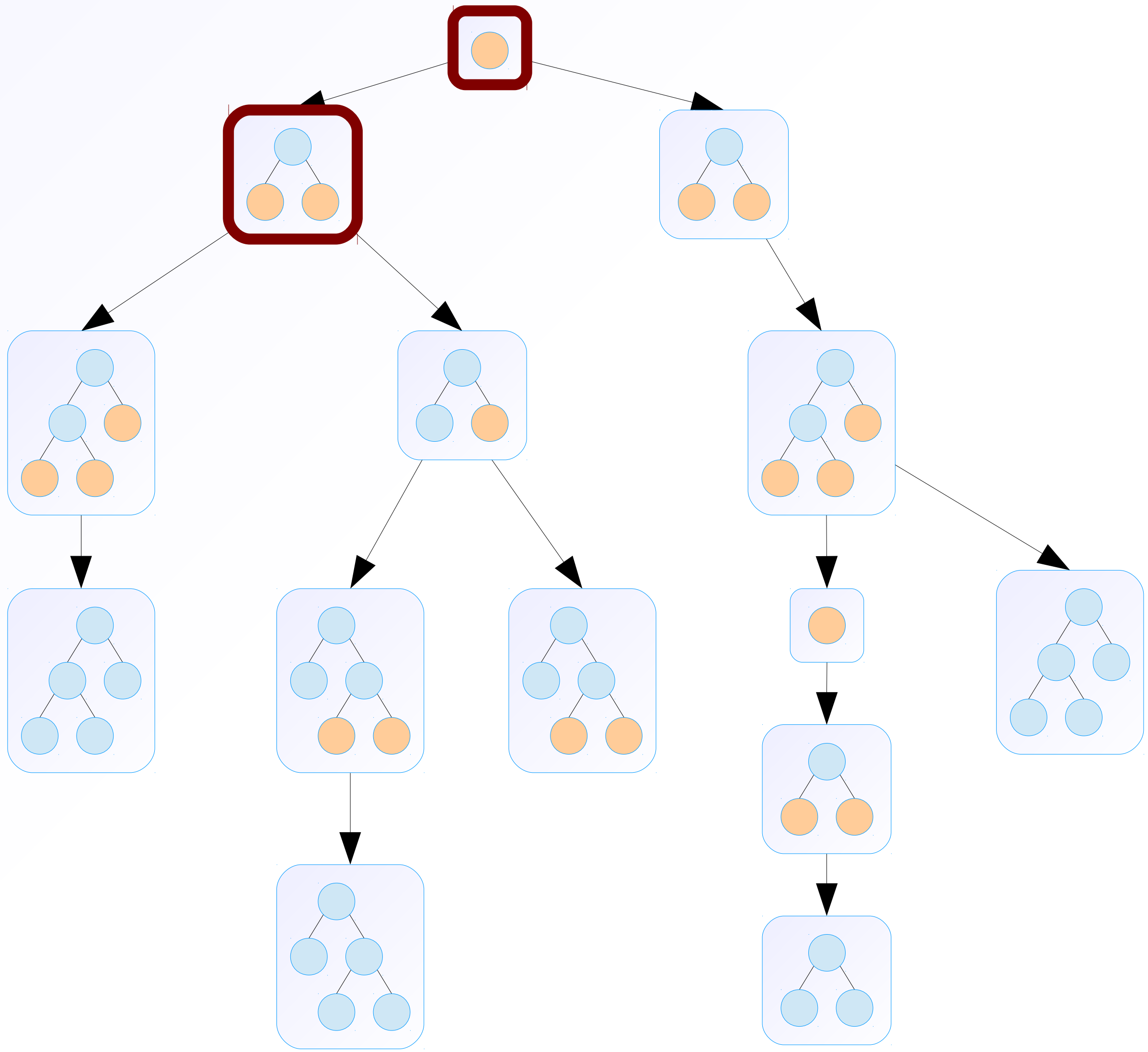
Depth-First Traversal of the Tree of Graphs

MRSC: Tree of Graphs



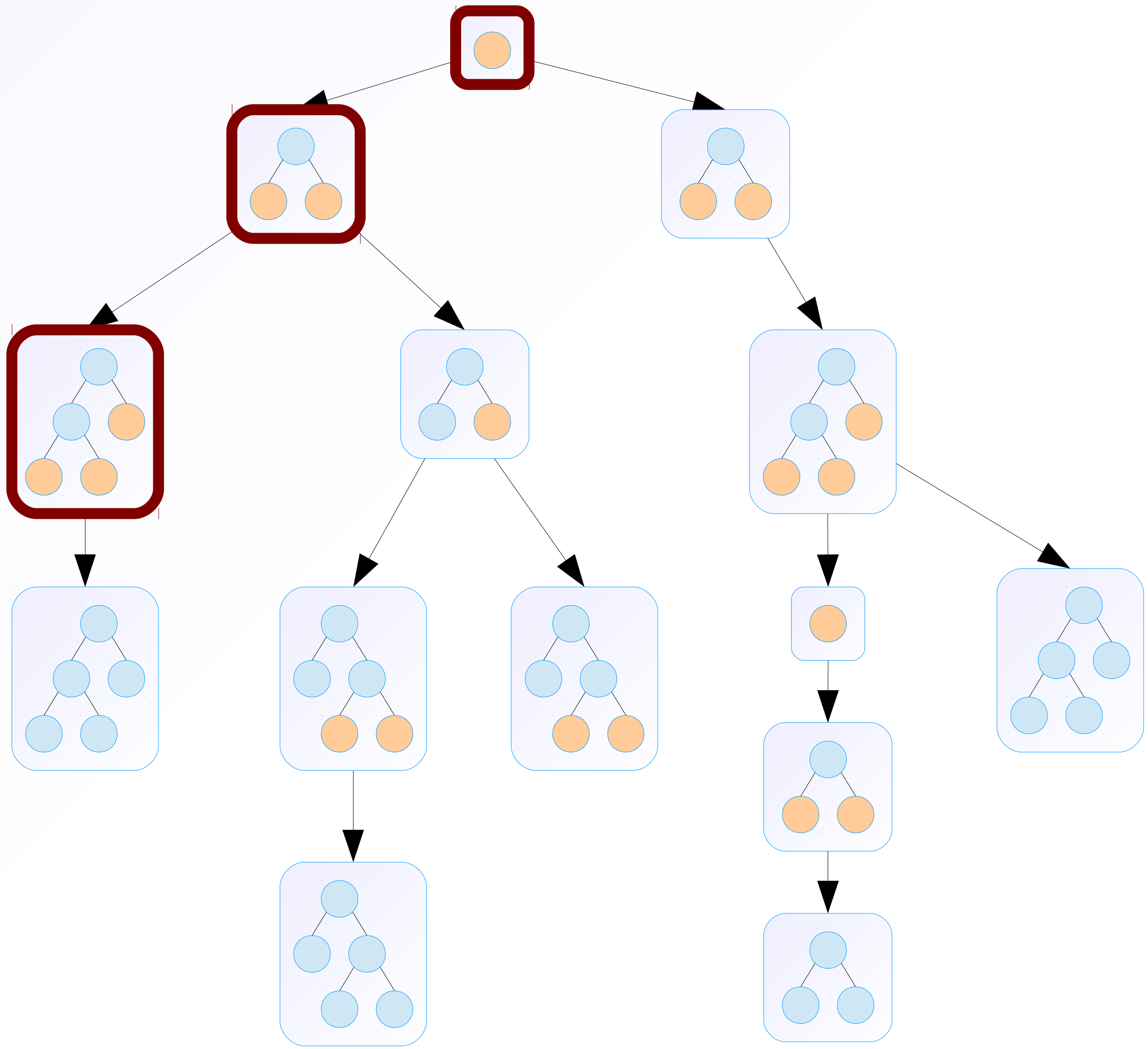
Depth-First Traversal of the Tree of Graphs

MRSC: Tree of Graphs



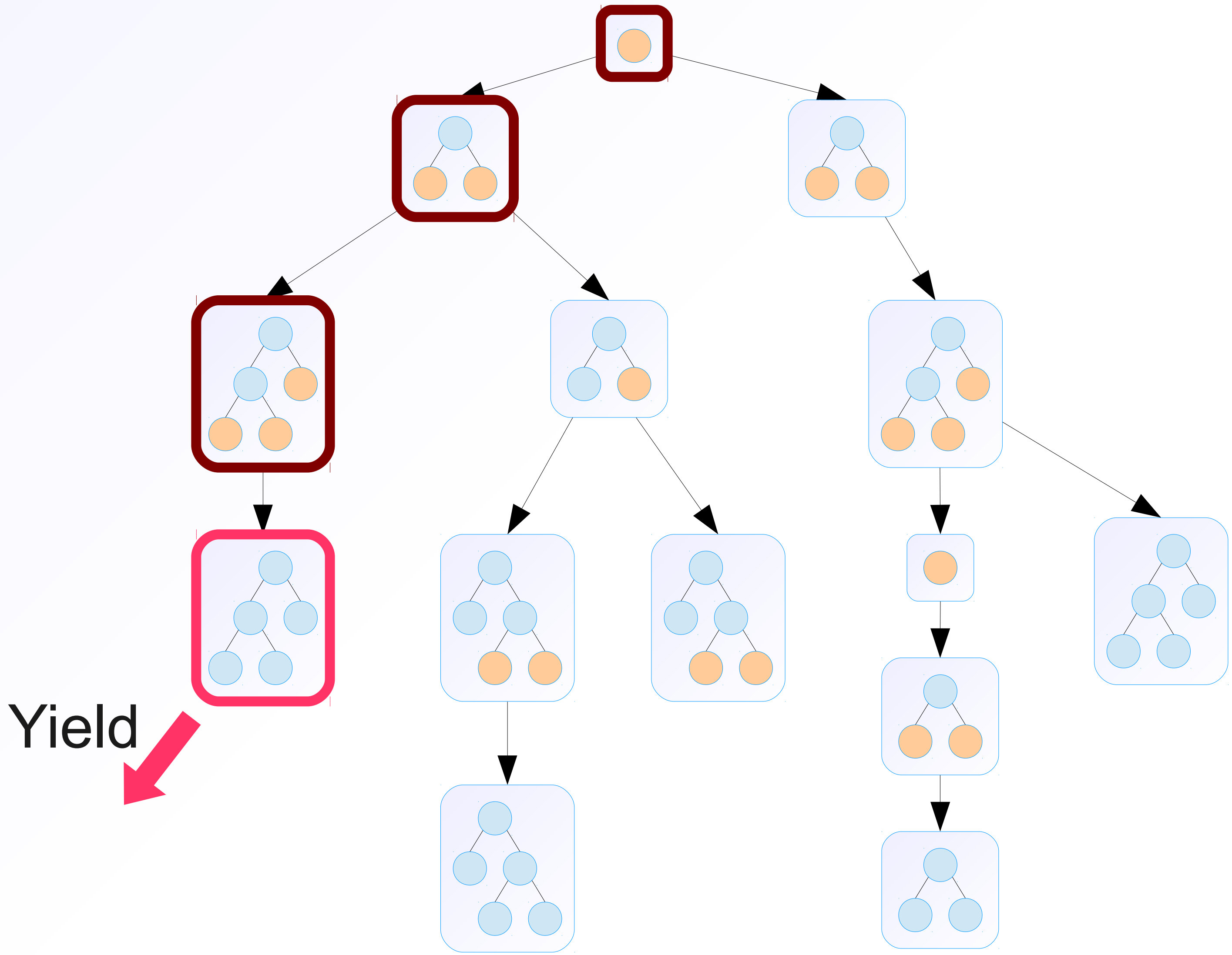
Depth-First Traversal of the Tree of Graphs

MRSC: Tree of Graphs



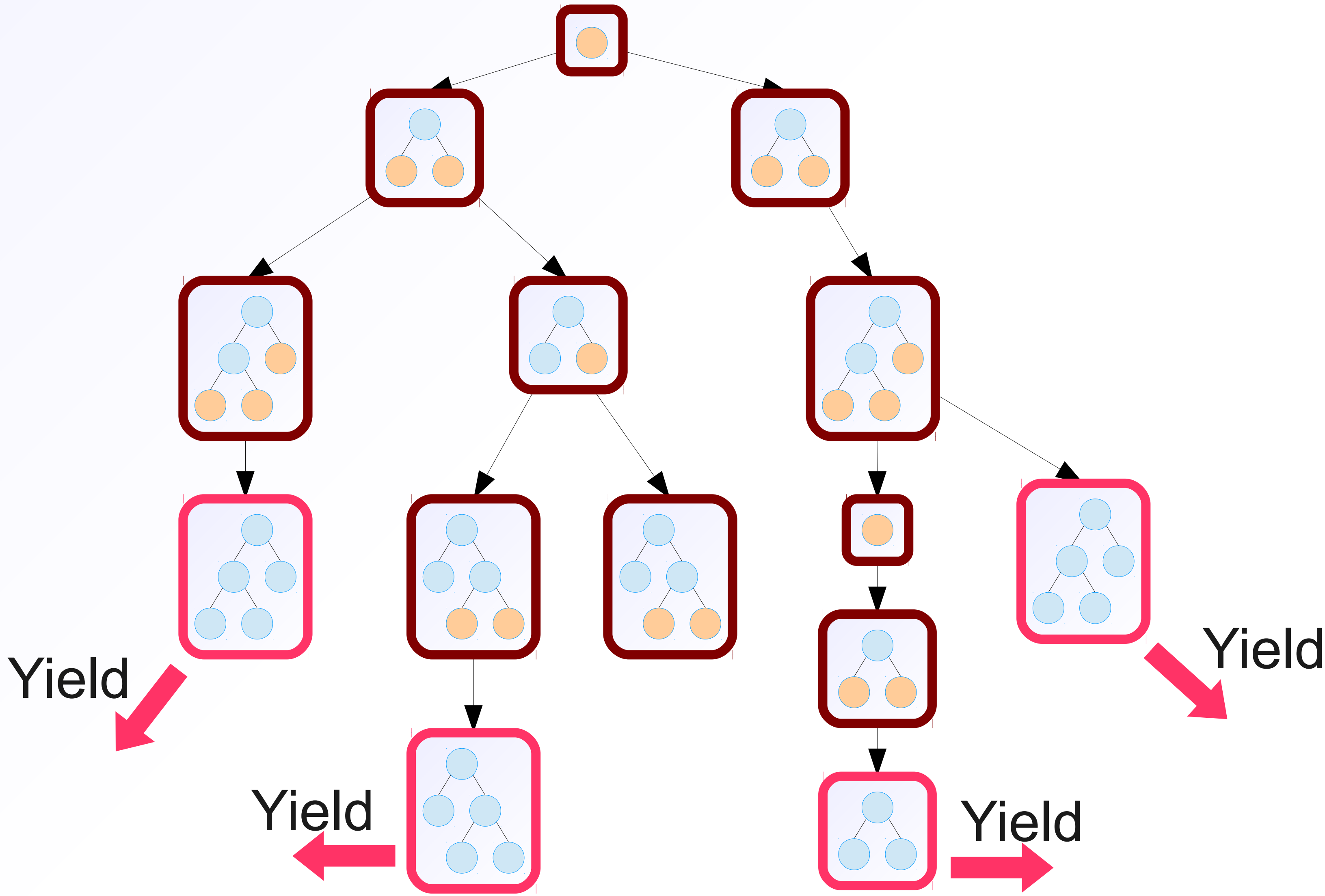
Depth-First Traversal of the Tree of Graphs

MRSC: Tree of Graphs



Depth-First Traversal of the Tree of Graphs

MRSC: Tree of Graphs

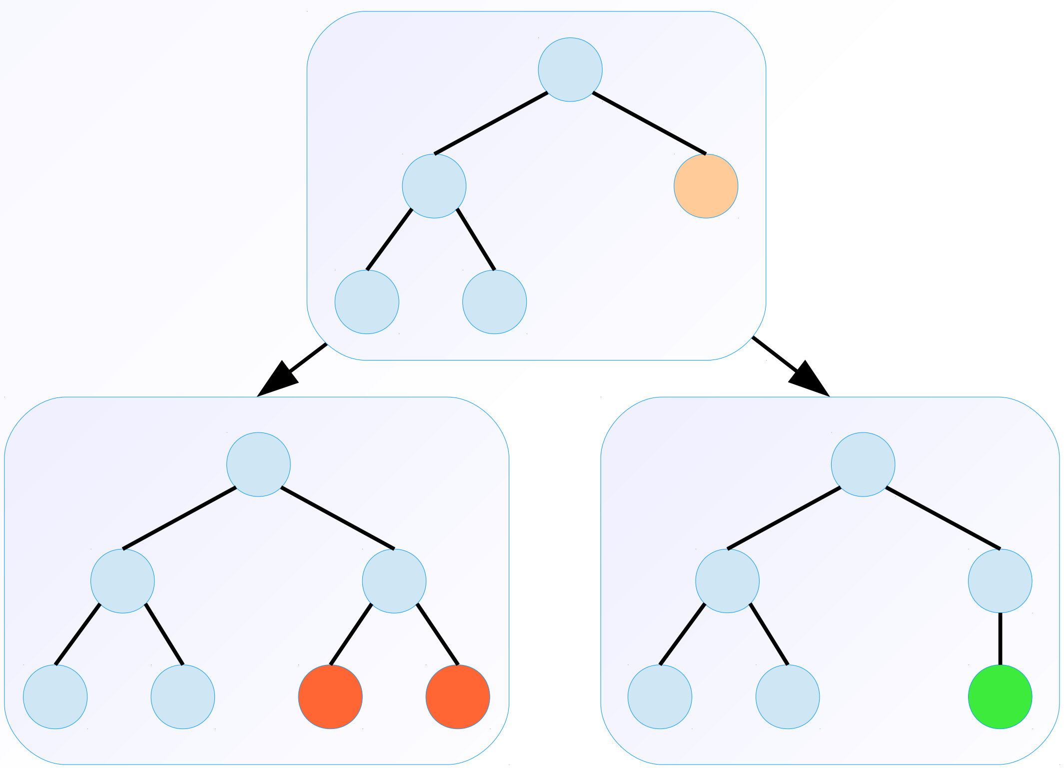


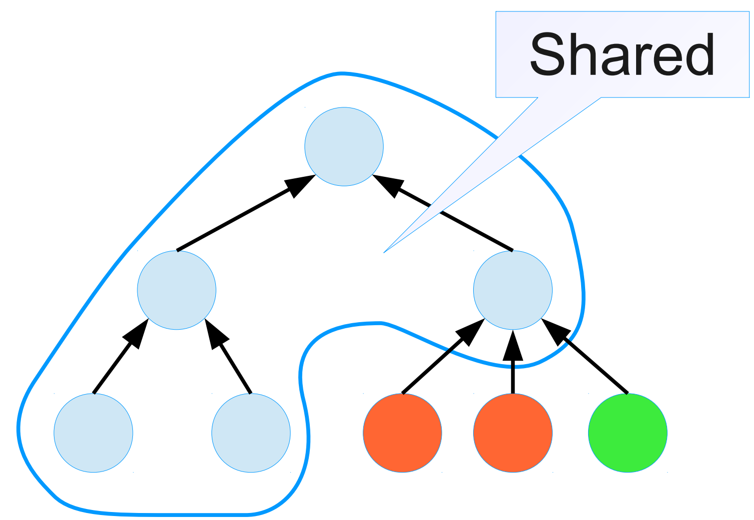
Depth-First Traversal of the Tree of Graphs

Combinatorial Explosion

Too many graphs

- Use some **heuristics**
- **Share** some parts of graphs

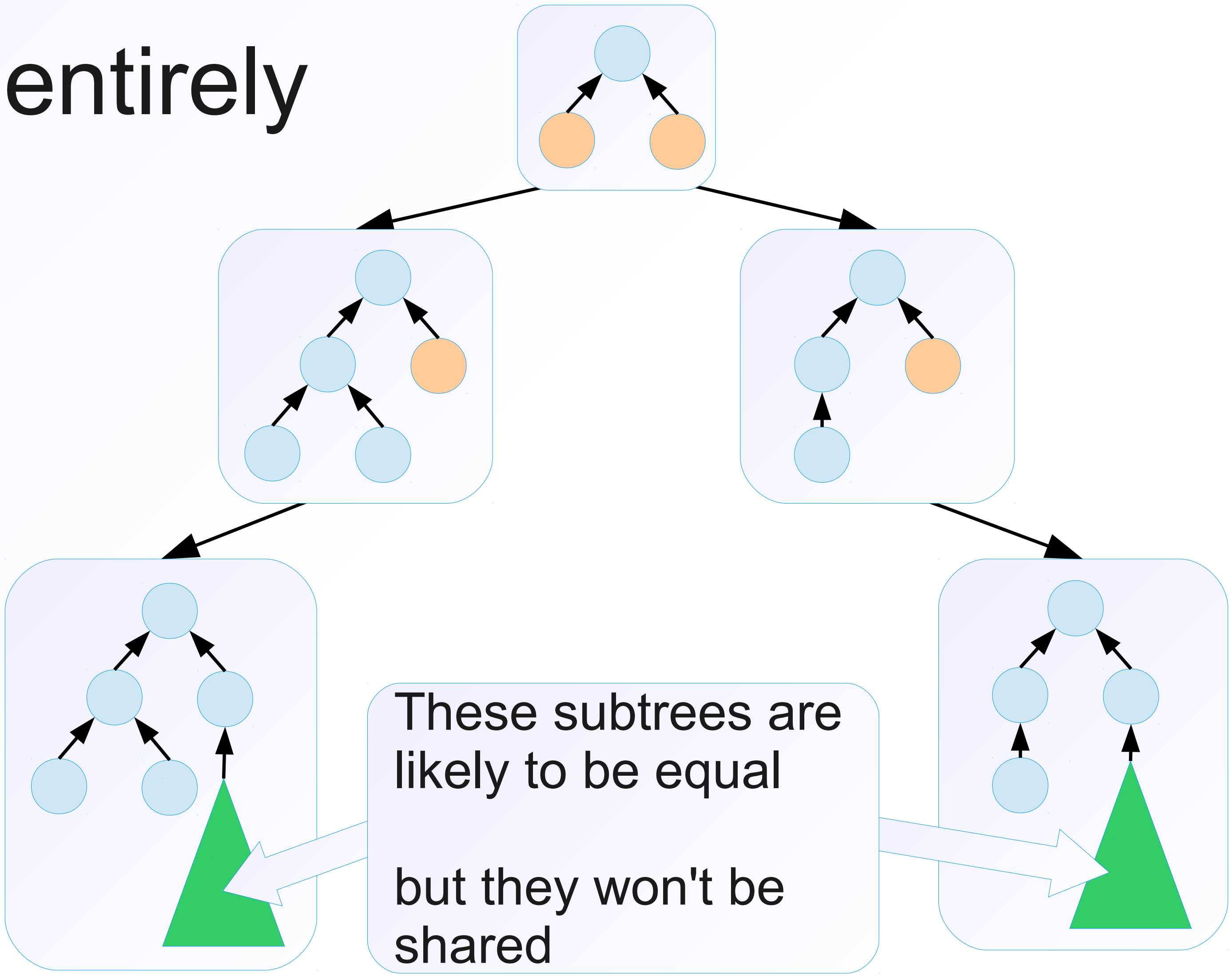




Spaghetti Stack (MRSC)

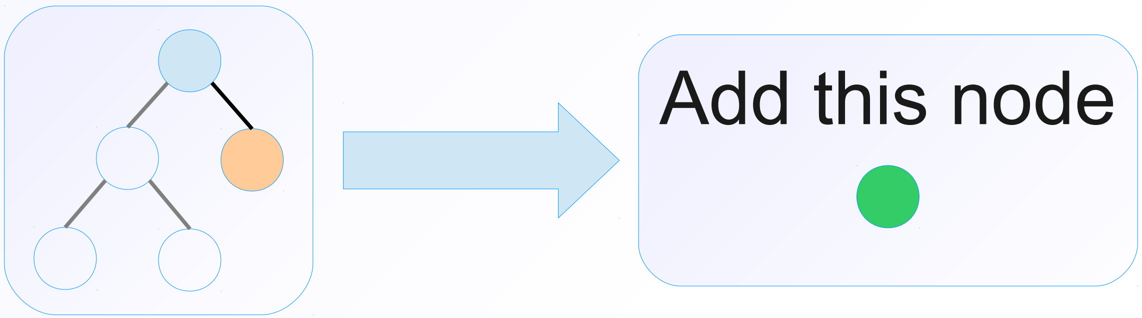
Do Spaghetti Stacks Solve the Problem?

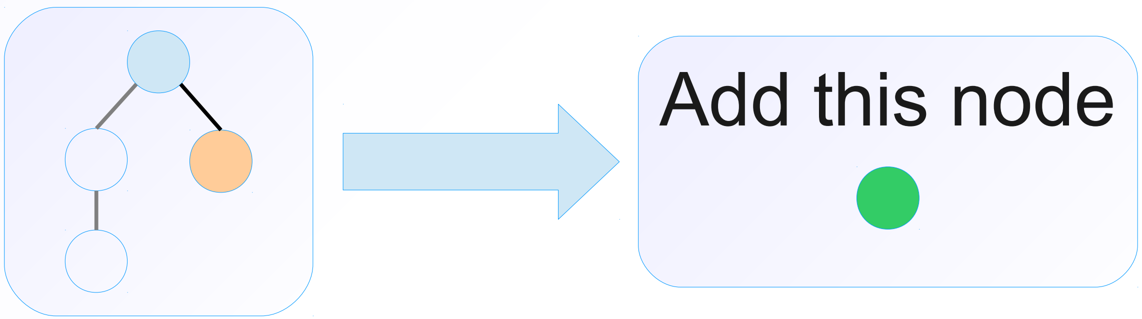
Not entirely



Rules : Graph → [Step]

Rules transform graphs into rewriting steps

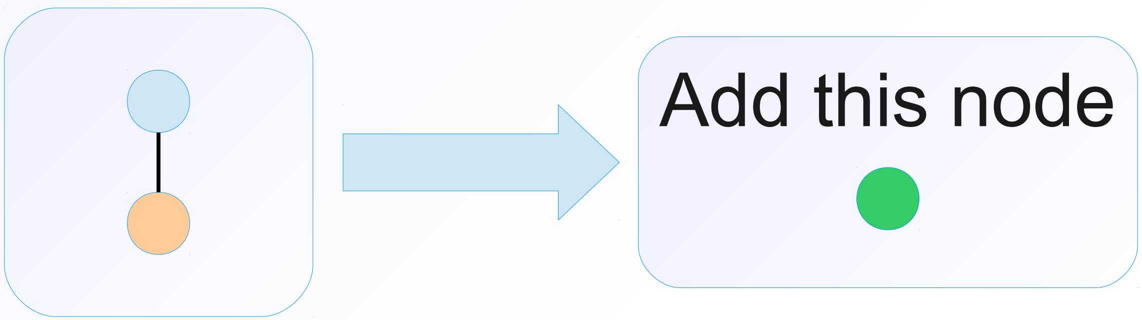




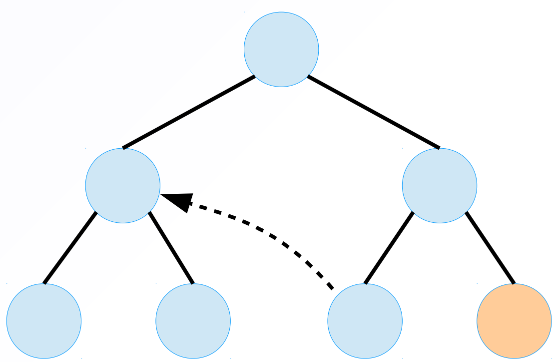
But usually they don't need the whole graph, just a path from the root to the current node

Rules : Path → [Step]

- Let's try to restrict rules to work on paths



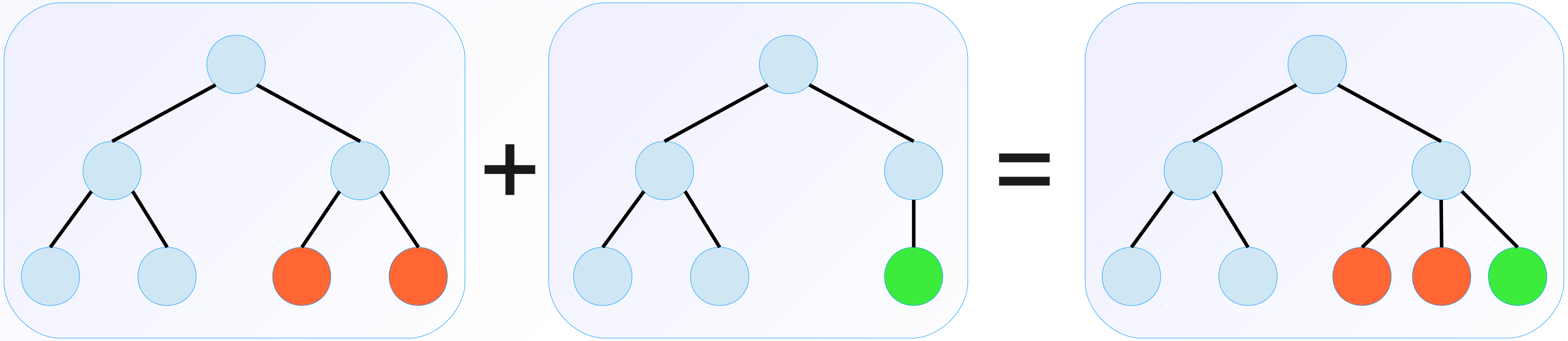
- We would lose an interesting ability to fold with cross edges



- We would need some new **representation** to make use of this new property

Overtree Representation

Let's combine all configuration trees into one big overtree



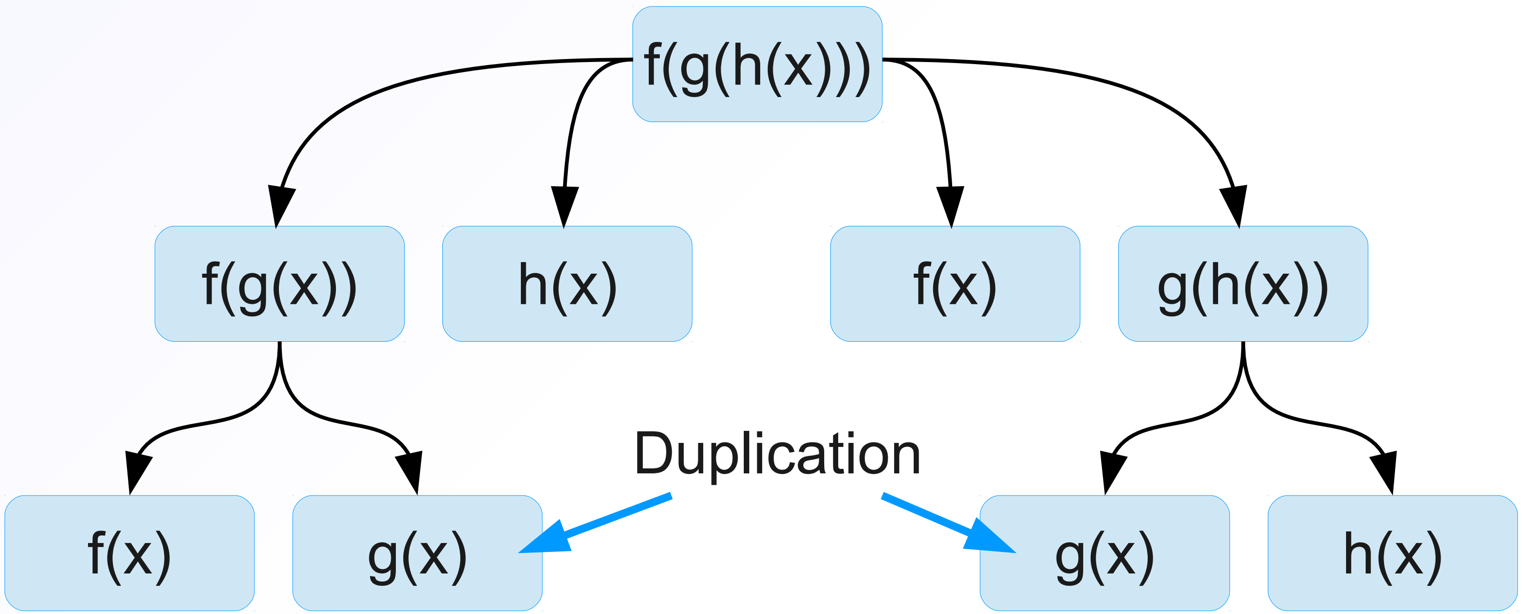
An overtree represents a **set** of trees

```
data Tree = Tree (F Tree)

data OTree = OTree [F OTree]
```

Do Overtrees Solve the Problem?

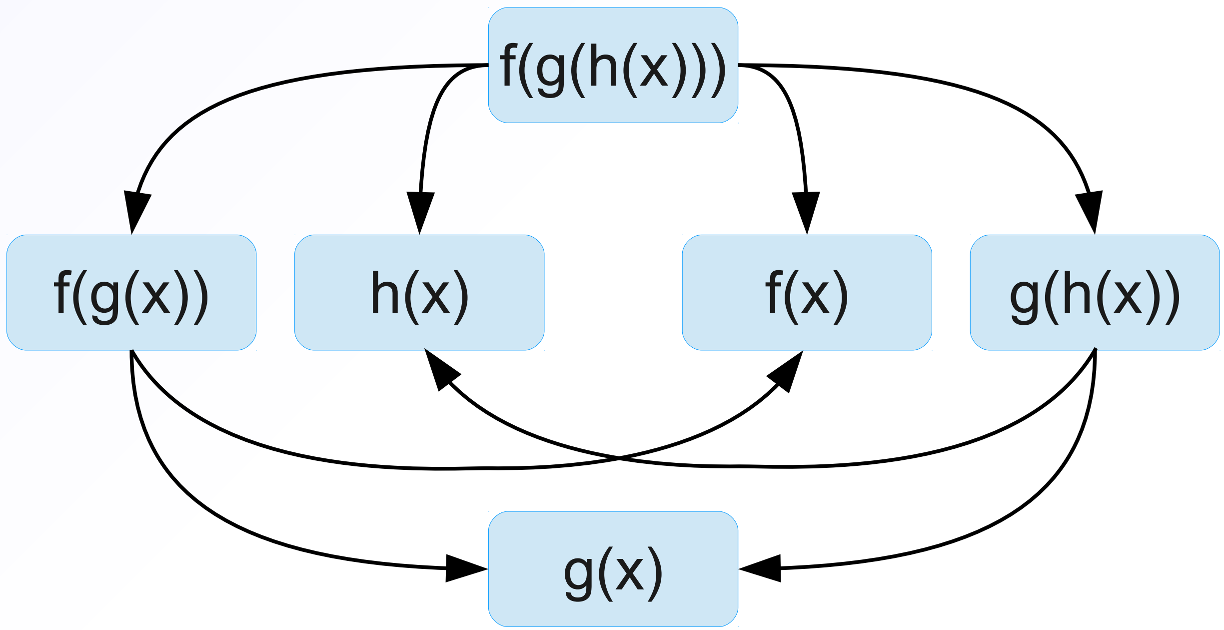
- They are a bit better, but still...



- We've already lost cross edges
- Are we going to lose folding edges completely?

Overgraph

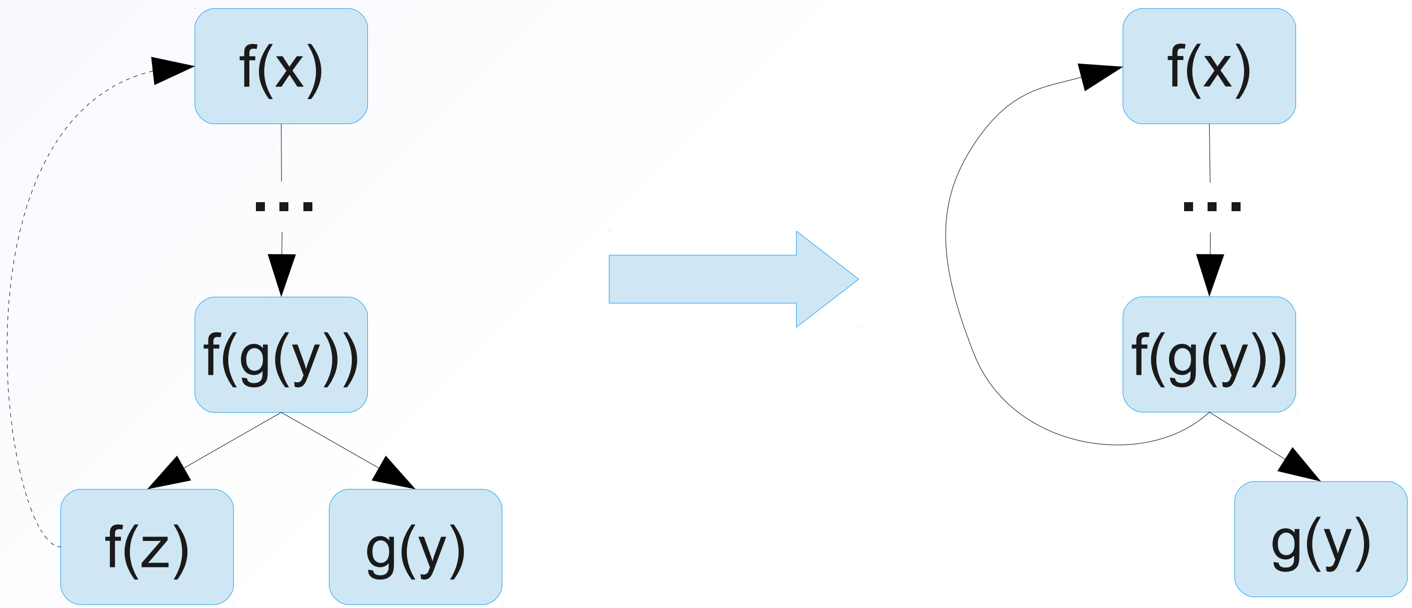
- Let's just glue together nodes equivalent **up to renaming**



- Each configuration corresponds to no more than one node

Folding

We don't need special folding edges



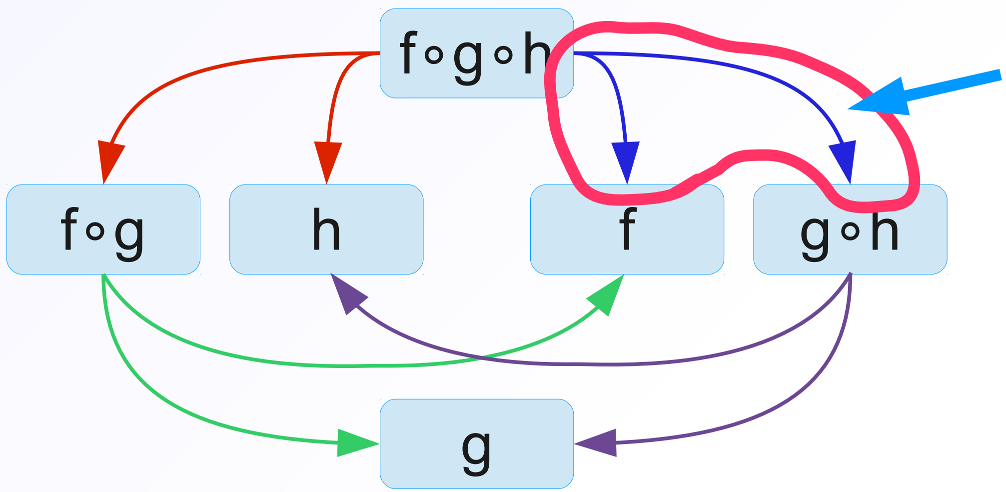
Advantages and Problems

- Overgraphs are more compact
- Overgraphs are cleaner
  - One configuration — one node
  - No special folding edges
- Overgraphs contain more information

- Each node can have multiple parents
  - Can we use binary whistles?
  - How can we control generalization?
- How to apply rules?
- How to extract residual programs?

Hyperedges

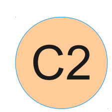
- We will call **bundles of edges** hyperedges



Hyperedge
$f \circ g \circ h \rightarrow (f\ ,\ g \circ h)$

- Hyperedges represent steps like driving and generalization
- Completion step can be represented as a hyperedge with **zero destination nodes**

$C1 \rightarrow ()$



incomplete nodes have no outgoing hyperedges

Supercompilation with Overgraphs

1) Overgraph **Construction**

Add nodes and edges while possible

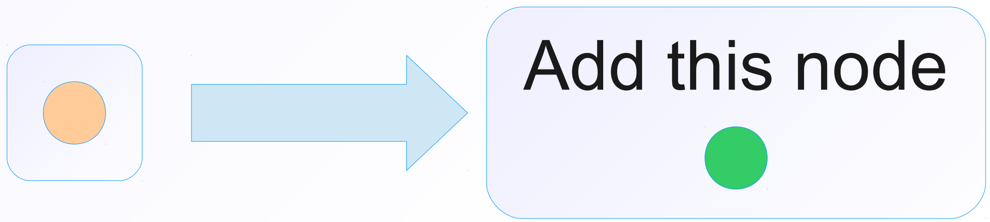
2) Overgraph **Truncation**

Remove useless nodes and edges

3) **Residualization**

Overgraph Construction

- `Rule : Configuration → [Step]`



- `Rule : Overgraph → [Hyperedge]`

In what order should we apply the rules?

`r` is monotone if for all graphs `G` and `H`:

$$G \subseteq H \Rightarrow r(G) \subseteq r(H)$$

If all rules are monotone we can apply them in any order

Rules

- We can also write rules in this form:

$$\frac{\text{precondition}}{\text{hyperedges to add}}$$

- Examples:

$$\frac{\neg\ \text{UnaryWhistle(c)}}{c \longrightarrow \text{drive(c)}} \qquad \frac{\text{always}}{c \longrightarrow \text{generalize(c)}}$$

$$\frac{\text{min\_depth(c)} < 42}{c \longrightarrow \text{drive(c)}}$$

This precondition is monotone

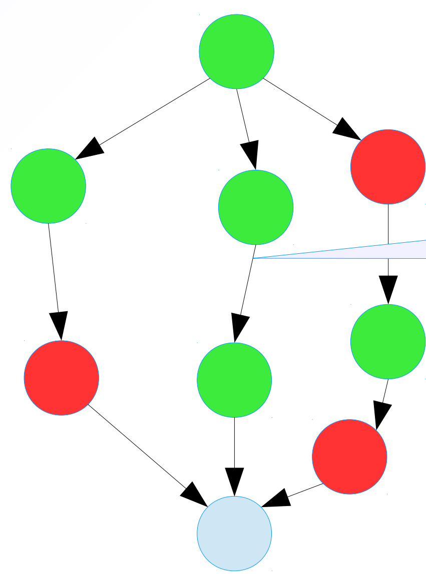
Binary Whistles

$$\frac{\neg \exists\, d \in G : \text{BinaryWhistle}(c,d)}{c \longrightarrow \text{drive}(c)}$$

NOT monotone

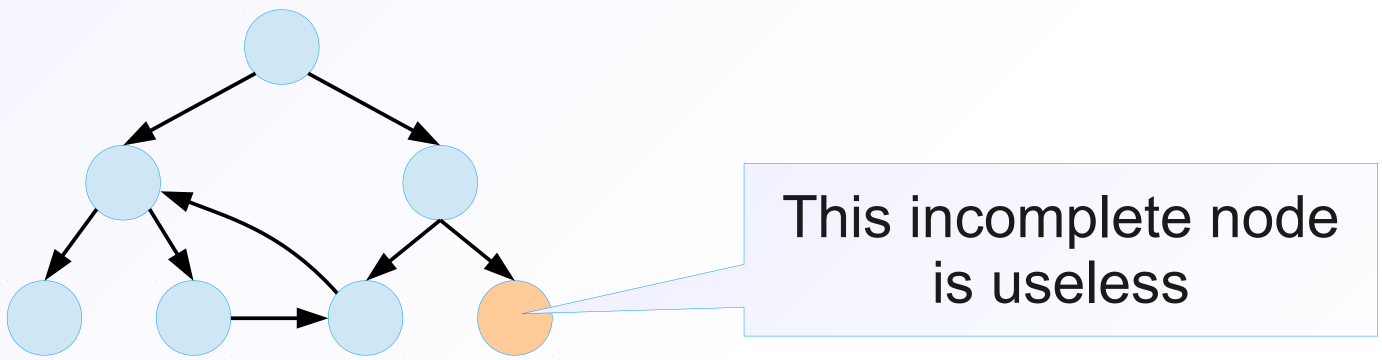
$$\frac{\exists \text{ path } p \text{ from root to } c : \quad \forall\, d \in p : \neg\text{BinaryWhistle}(c,d)}{c \longrightarrow \text{drive}(c)}$$

OK

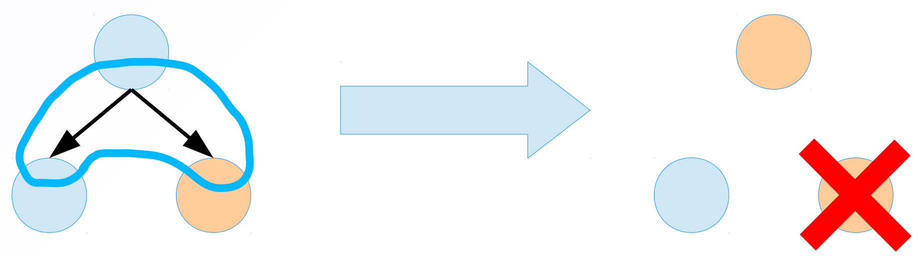


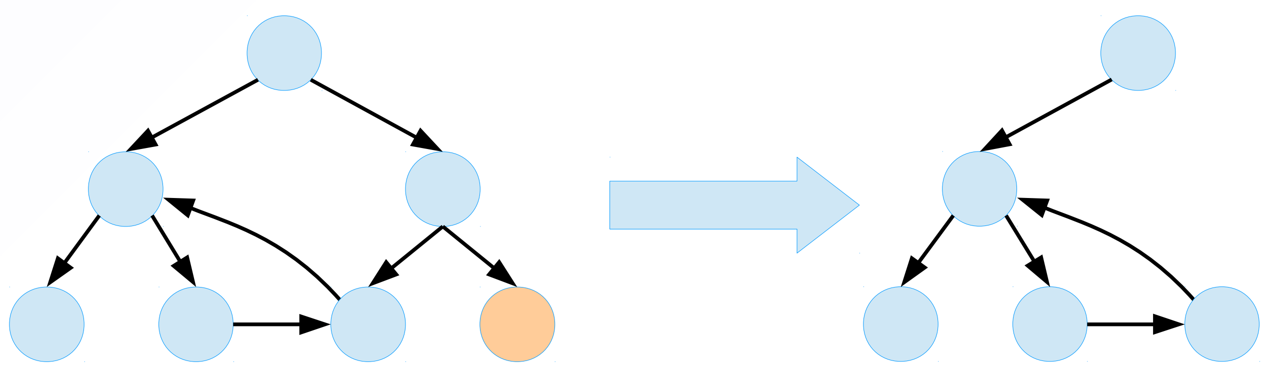
This green path won't disappear

Overgraph Truncation

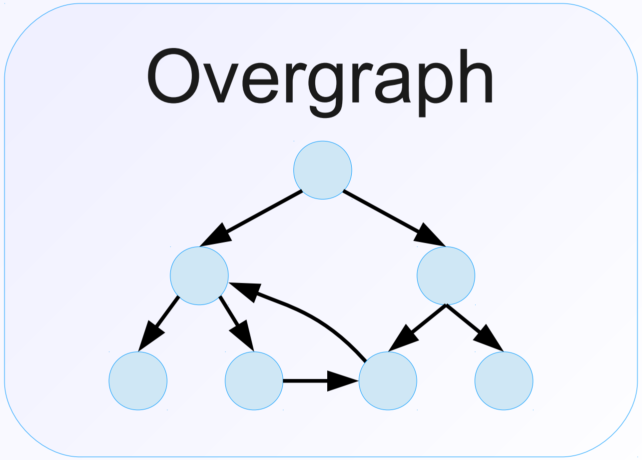


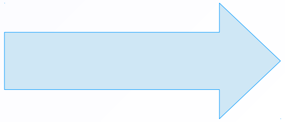
We should remove all incident hyperedges

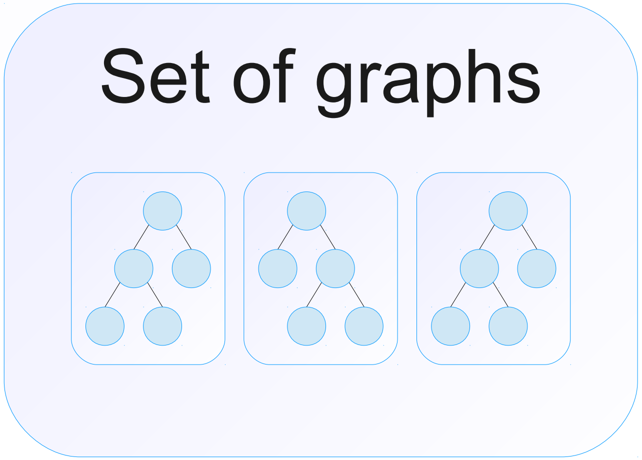




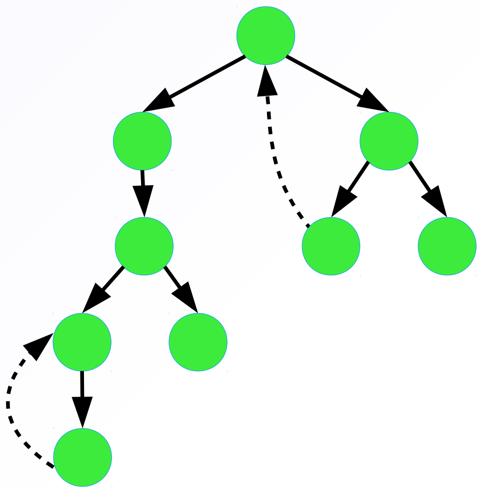
Residualization





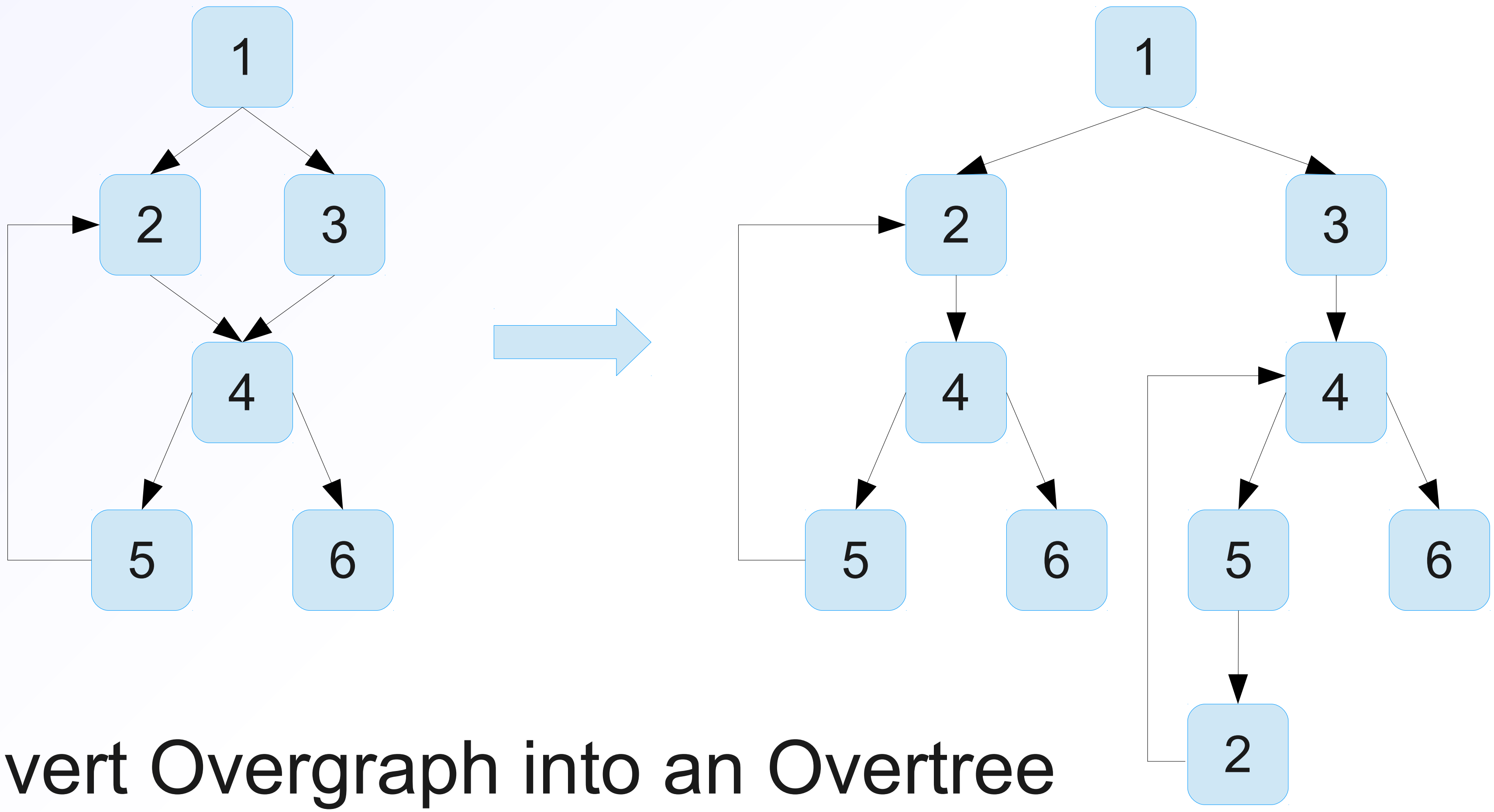


Building a full set of graphs should be avoided!



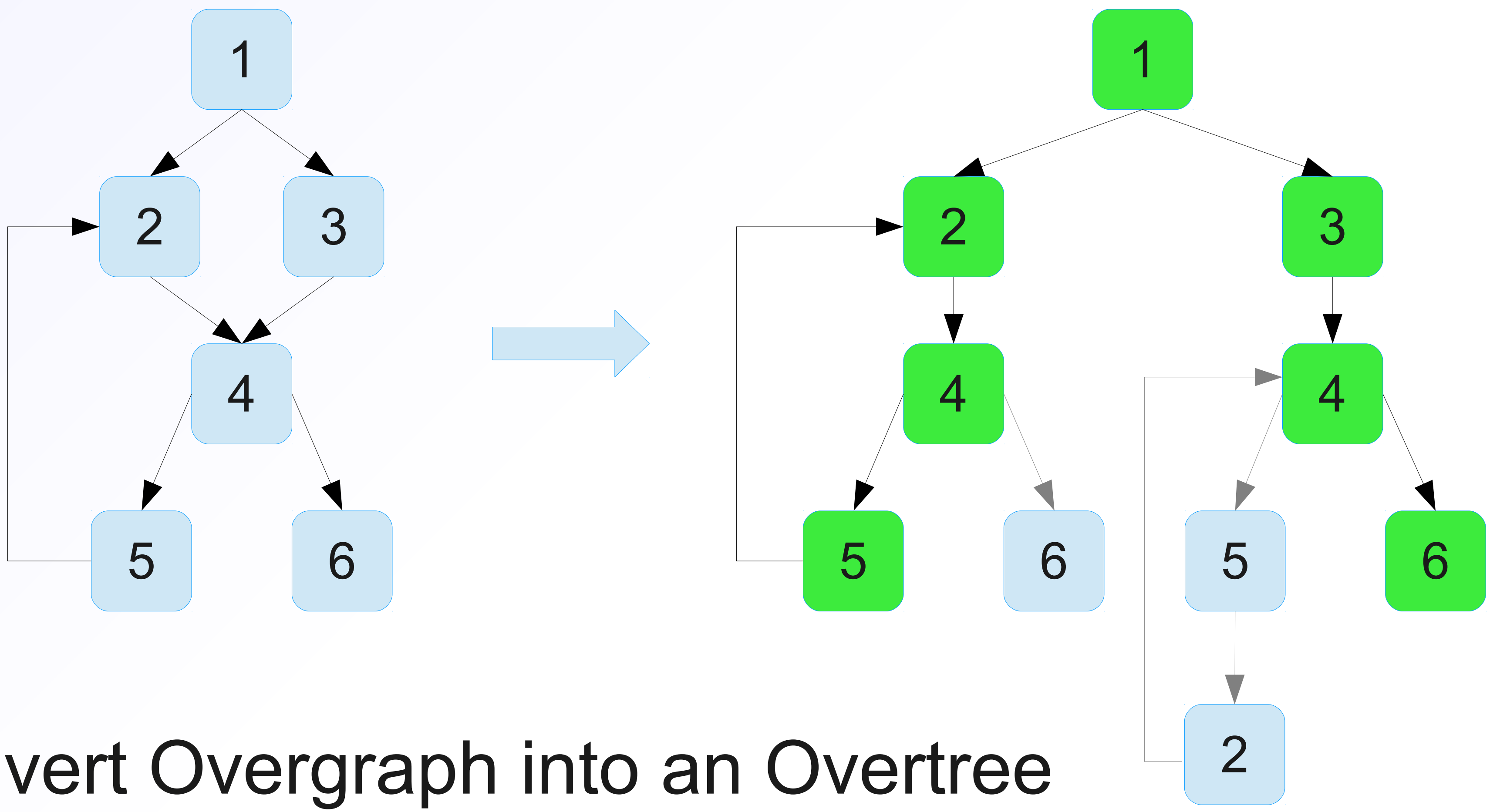
We will represent residual programs as **trees with back edges**
(i.e. no subprogram sharing)

Naive Residualization Algorithm



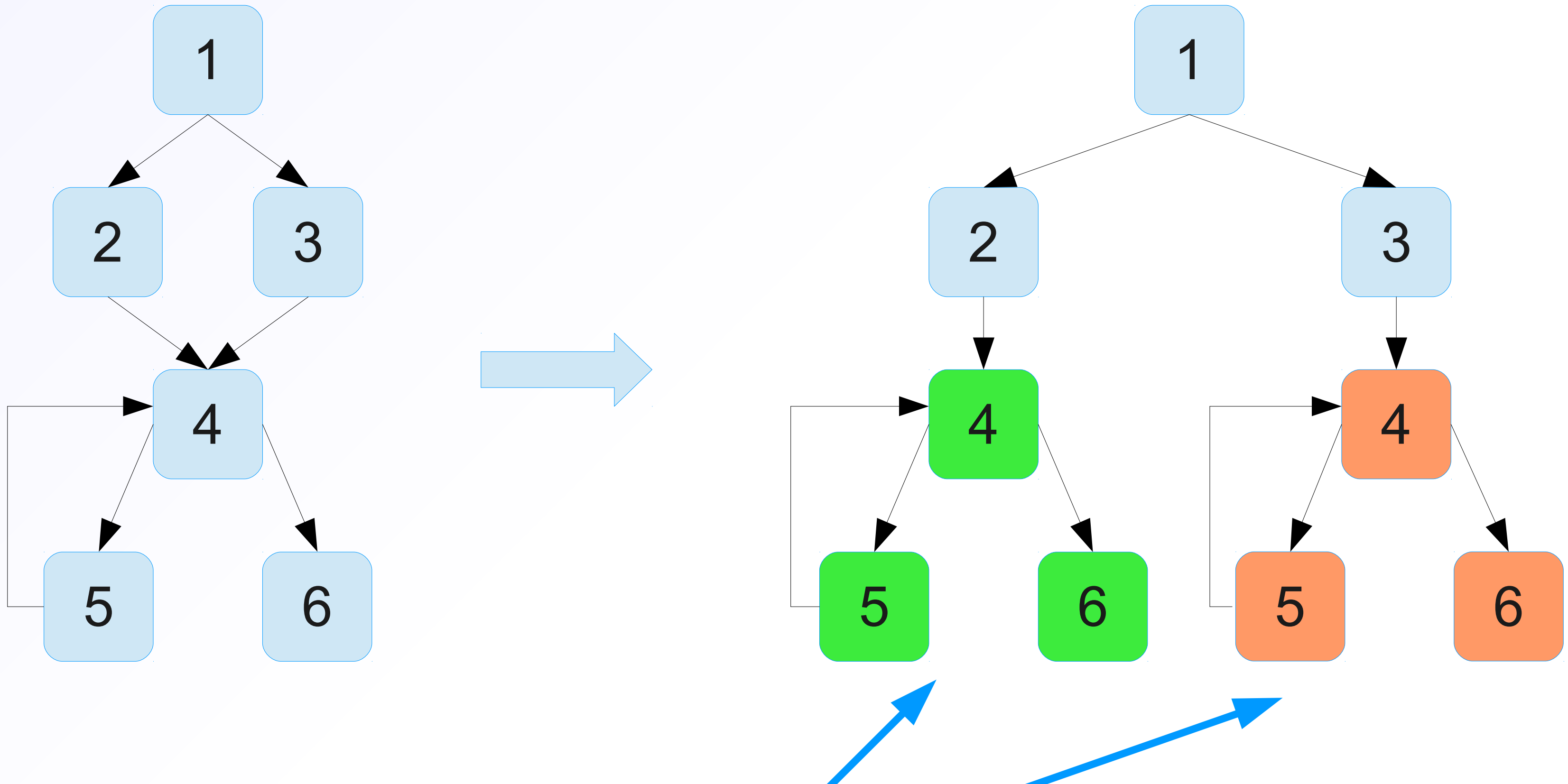
Convert Overgraph into an Overtree
and then convert it into a set of trees

Naive Residualization Algorithm



Convert Overgraph into an Overtree and then convert it into a set of trees

Suboptimality



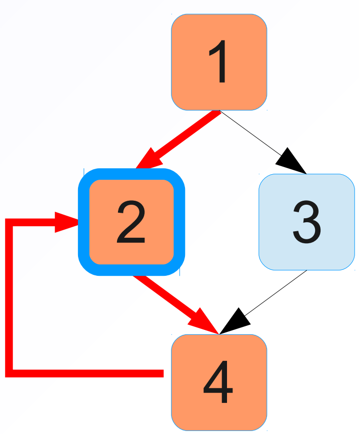
Absolutely identical subtrees

Idea: Cache intermediate results

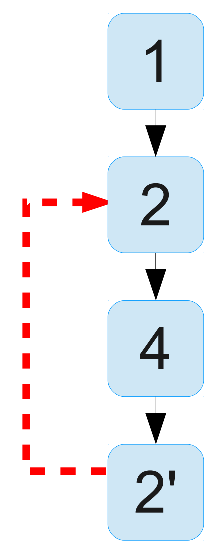
More Formal Definition

```
R : Node → [Node] → [Tree]
```

```
R n h | n ∈ h = [Fold(n)]
R n h | otherwise =
   [n → (r1 ... rk) |
               n → (d1 ... dk)  ∈  G,
               ri ∈ R di (n:h)]
```



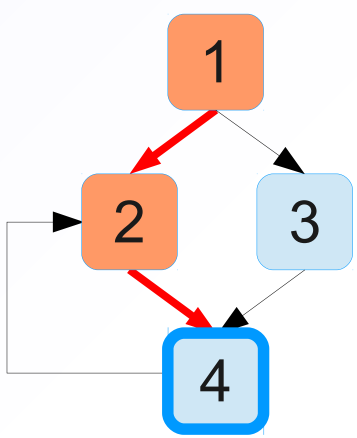

```
h = [4, 2, 1]
n = 2
```



More Formal Definition

```
R : Node → [Node] → [Tree]
```

```
R n h | n ∈ h = [Fold(n)]
R n h | otherwise =
   [n → (r1 ... rk) |
              n → (d1 ... dk)  ∈  G,
              ri ∈ R di (n:h)]
```

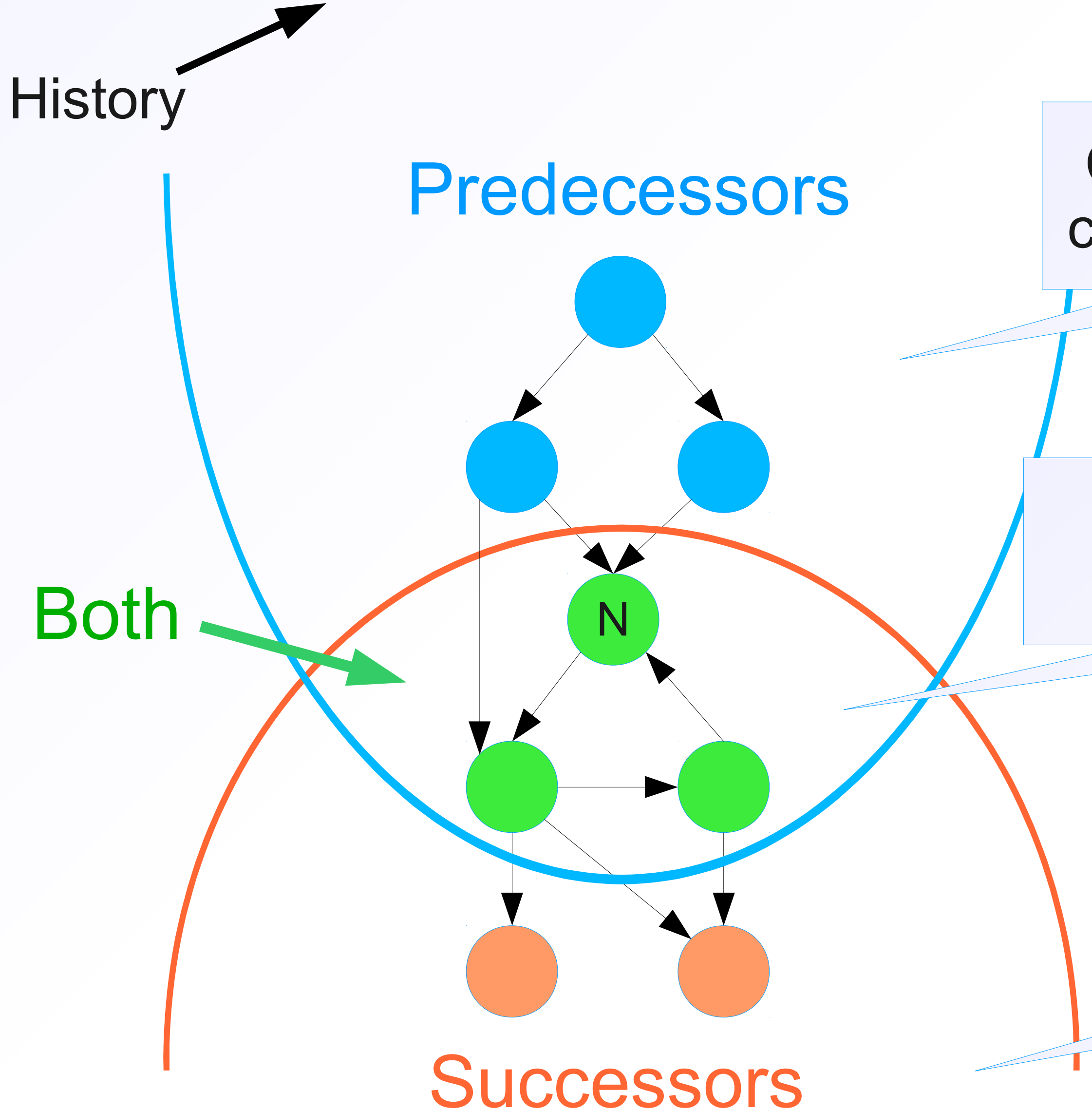



```
h = [2, 1]
n = 4

R 2 [4, 2, 1]
```

History Structure

`R : Node → [Node] → [Tree]`



Can be in a history but cannot be folded against

These can influence folding

Won't be in a history

Enhanced Residualization

- Removing pure predecessors from history won't change the result

```
R n h = R n (h ∩ succs(n))
```

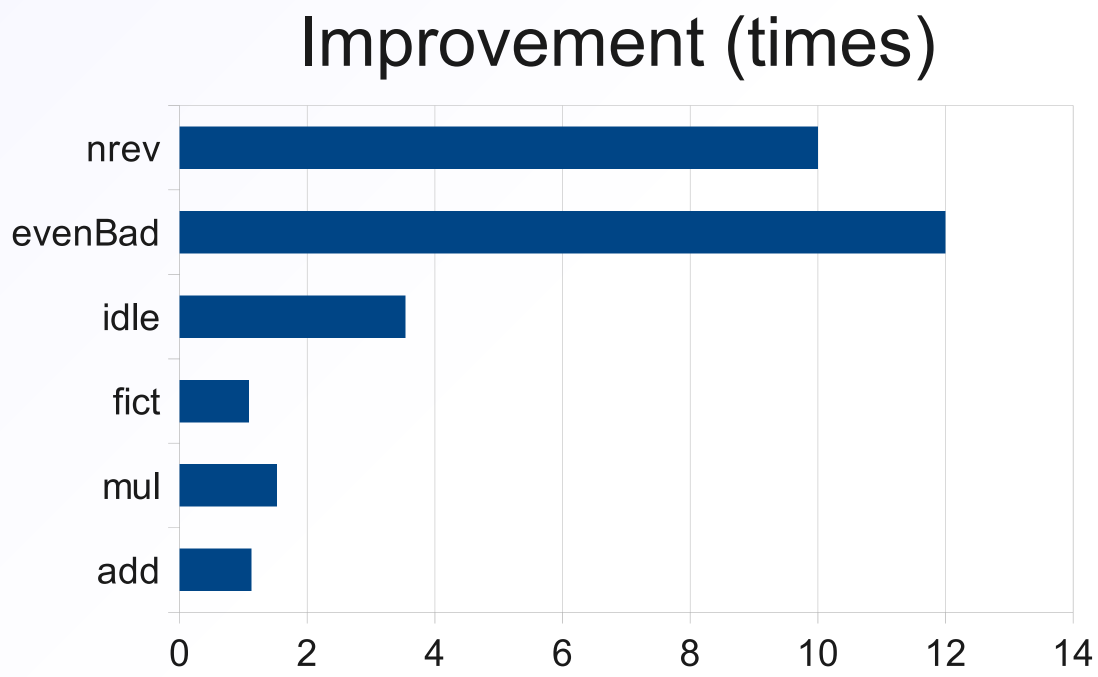
- Let's rewrite residualization algorithm this way:

```
R n h | n ∈ h = [Fold(n)]
R n h | otherwise =
  [n → (r1 ... rk) |
             n → (d1 ... dk)  ∈  G,
             ri ∈ R di (n:h ∩ succs(di))]
```

- Now we can just apply memoization

Evaluation of Residualization Algorithms

- Caching improves performance



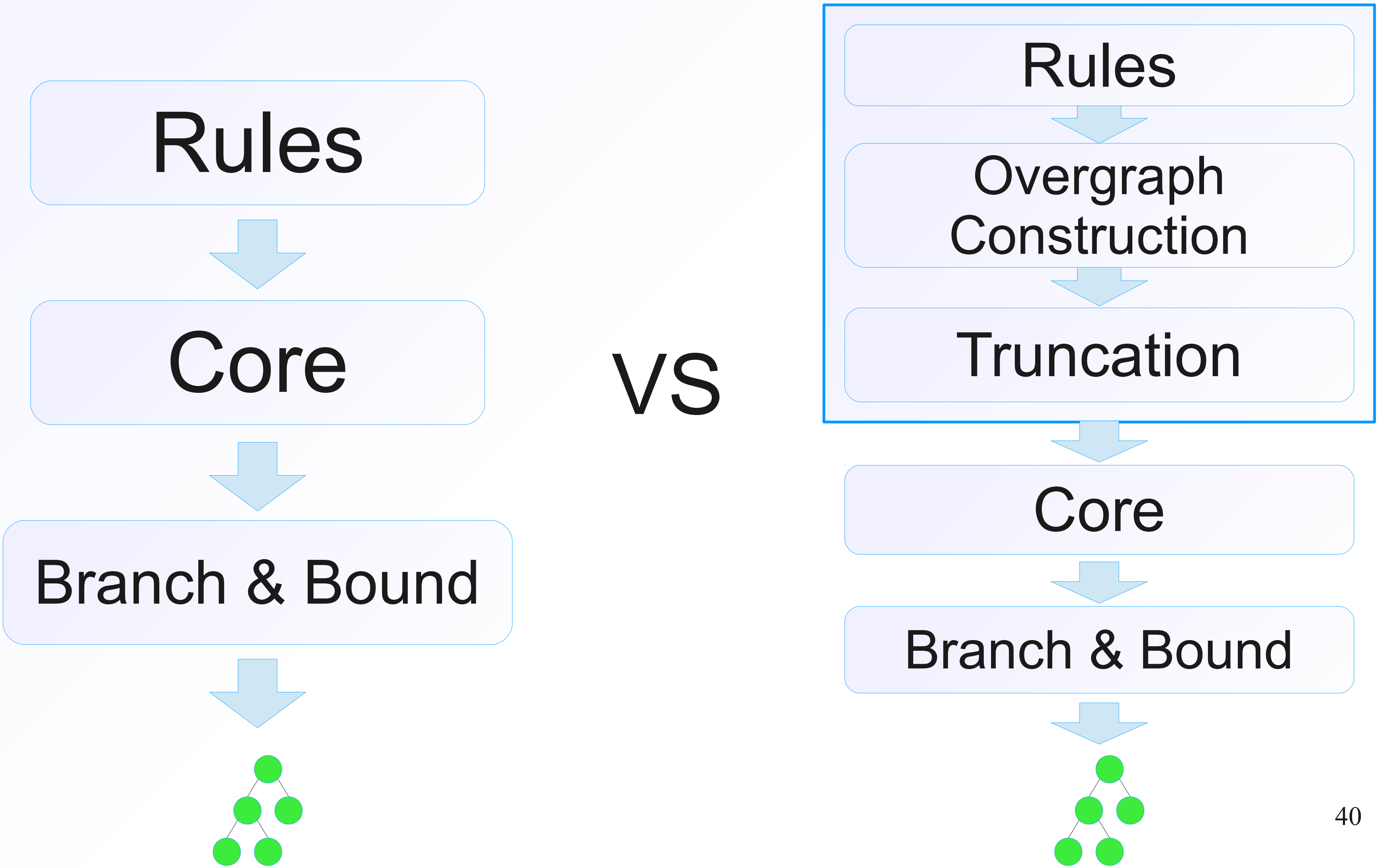
- But the algorithms produce trees with back edges

Turned out it is not very useful for most tasks

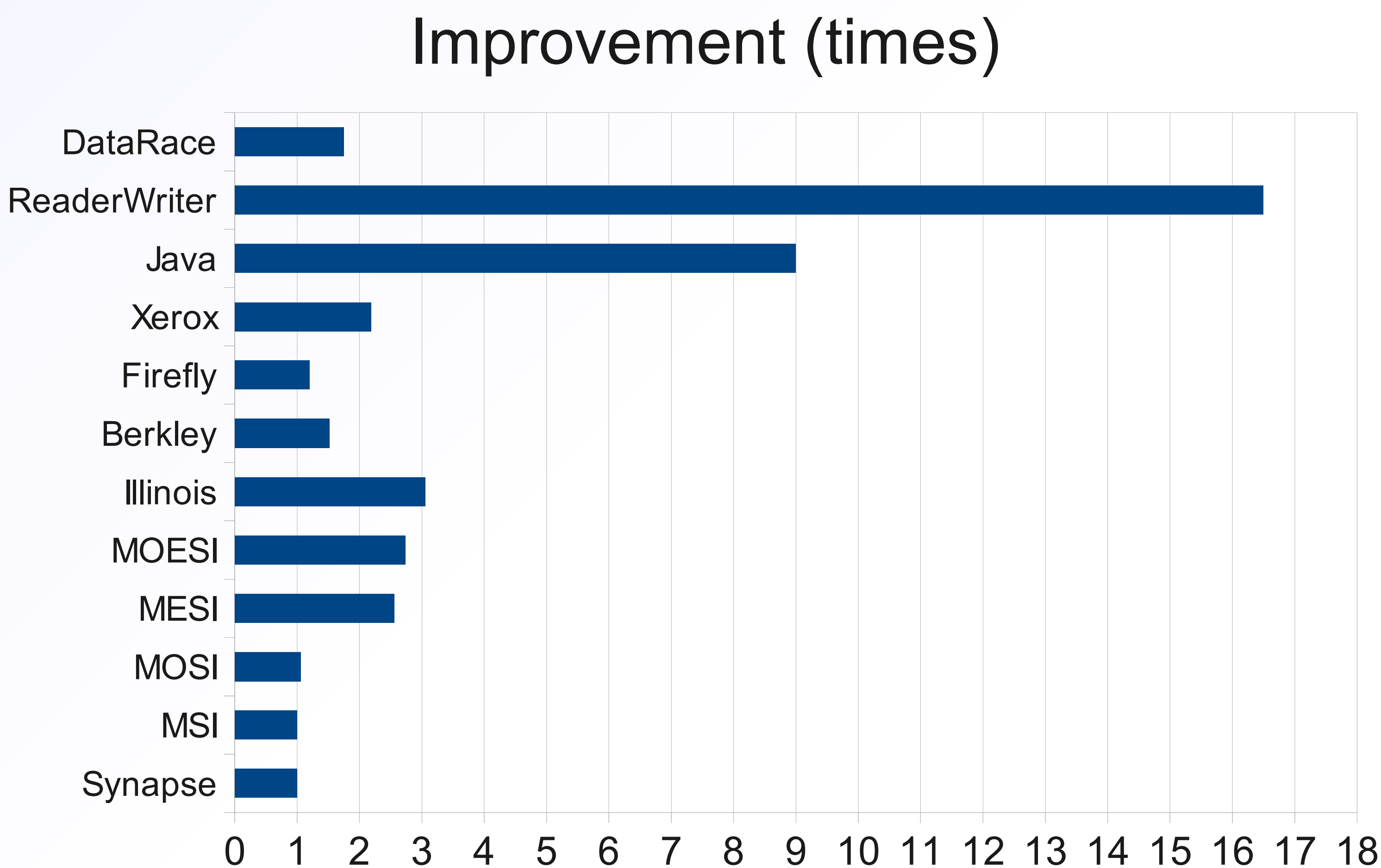
Example: Counter Systems

- The task is to find the minimal proof of a counter system's safety
- A proof is a **graph**, not a tree with back edges
- MRSC uses **cross edges** to simulate graphs
- But overgraphs may be still useful because they enable **truncation**

Experiment with Counter Systems



Experimental Results



(in terms of the number of visited nodes)

Why overgraphs were useful?

- We could compute **sets of successors**
- We could **truncate** an overgraph

An overgraph contains a lot of **information** about relations between configurations

This is even more important than its compactness

Further Work

- Experiments with subgraph-producing residualization algorithms
  - need graph-based language
  - tree-producing algorithm seems unsuitable for real-world tasks
- Searching for heuristics (whistles etc) useful for overgraph representation
- Applying overgraphs to higher-level supercompilation

Conclusions

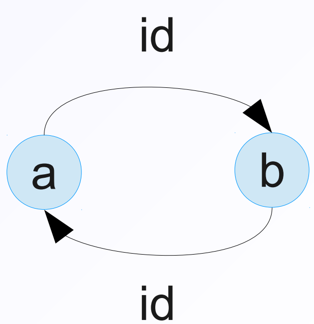
We suggested the Overgraph representation

- An Overgraph is a very compact representation
- Rules, Whistles and Residualization were generalized to Overgraphs
- The implementation has shown its usefulness
  - Caching residualization algorithm
  - Truncation for counter systems
- Overgraph contains a lot of information, so it is possible to analyze multiple graphs at once

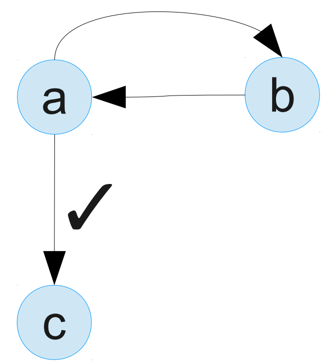
Please return to the previous slide

Correctness

- It is possible that not all of the trees extracted from an overgraph represent correct programs



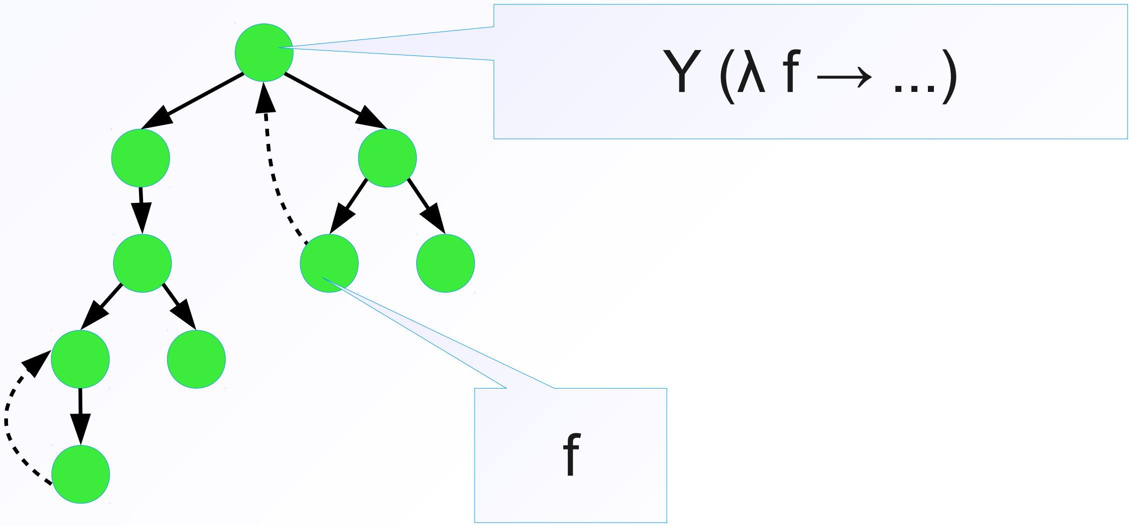
$$\frac{\begin{array}{c} a = b \\ b = a \end{array}}{a = a}$$



- Usually it is not a problem for single-level supercompilation

Language used in experiments

- The language is essentially based on trees with back edges



- Higher order
- Explicit fixed point combinator
- No let-expressions

Overgraph vs E-PEG

- Essentially the same idea applied to different domains
- We work with functional languages, so we have a clear recursion rather than incomprehensible cycles
- We don't have symmetric equalities
- We decided to residualize to trees, they naturally “residualize” to graphs
  - Should we do the same?

There are no more slides