Program Synthesis-based Program Optimization

Woosuk Lee

Hanyang University ERICA Campus

2024. 11. 13 @ UNIST

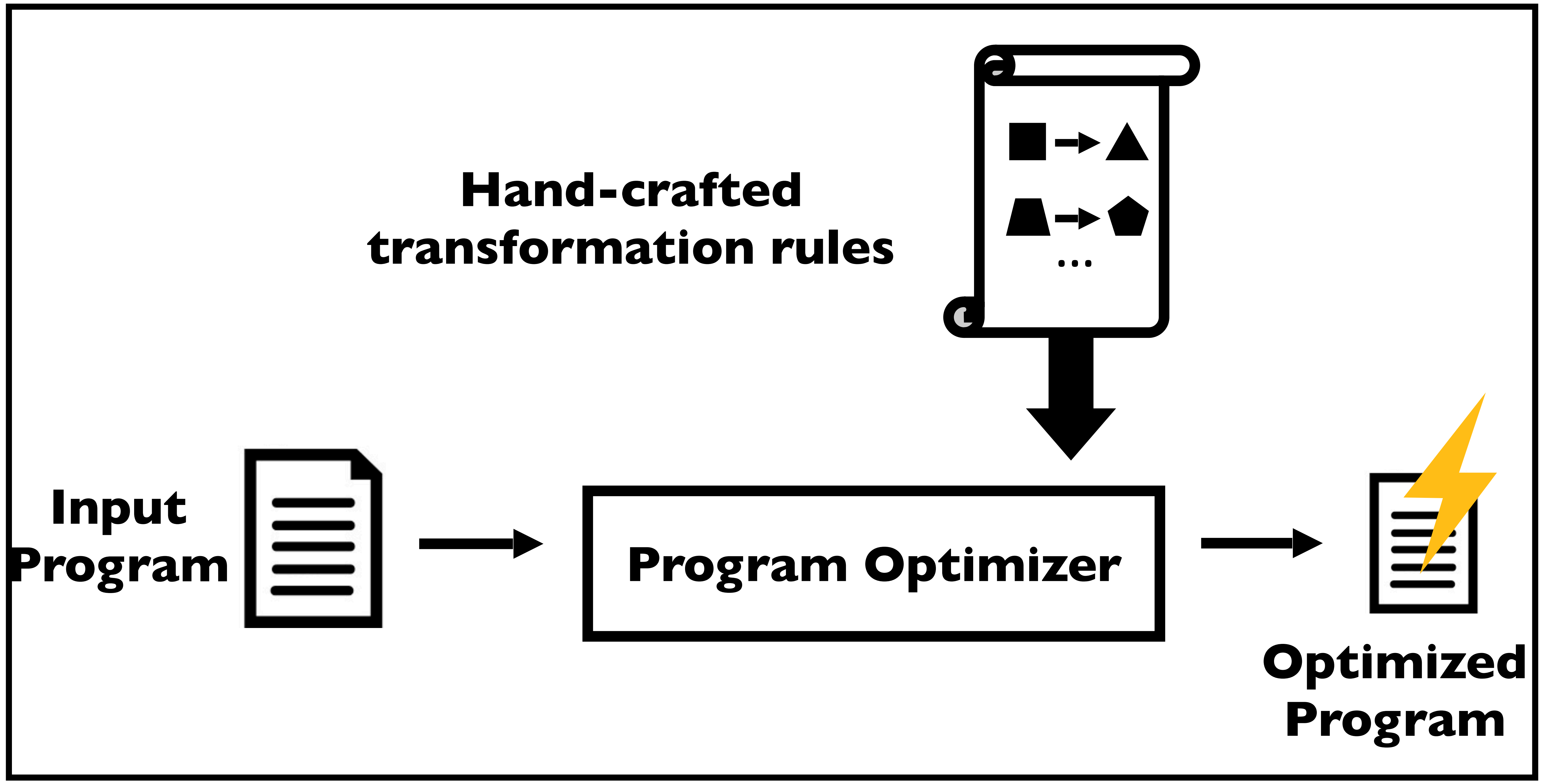
HANYANG UNIVERSITY

Who am I

- Associate Professor at Hanyang Univ. ERICA (2024.9 ~)
- Assistant Professor at Hanyang Univ. ERICA (2018.9 ~ 2024.8)
- Research Interests: program analysis, program synthesis
- Vita
  - 2012 UC Berkeley visiting researcher
  - 2016 Ph.D, Seoul National University (Advisor: Kwangkeun Yi)
  - 2016-2017 Postdoctoral fellow, Georgia Tech (Mayur Naik)
  - 2017-2018 Postdoctoral fellow, University of Pennsylvania (Mayur Naik)

Motivation

- Program optimization
  - Transforming into a better (e.g., cost) program
  - Applying transformation rules (e.g., $x + 0 \rightarrow x$)



- Rules in prior methods
  - Hand-crafted by domain experts
  - Limited search space
- Application order in prior methods
  - Heuristics by domain experts
  - May miss the optimal solutions

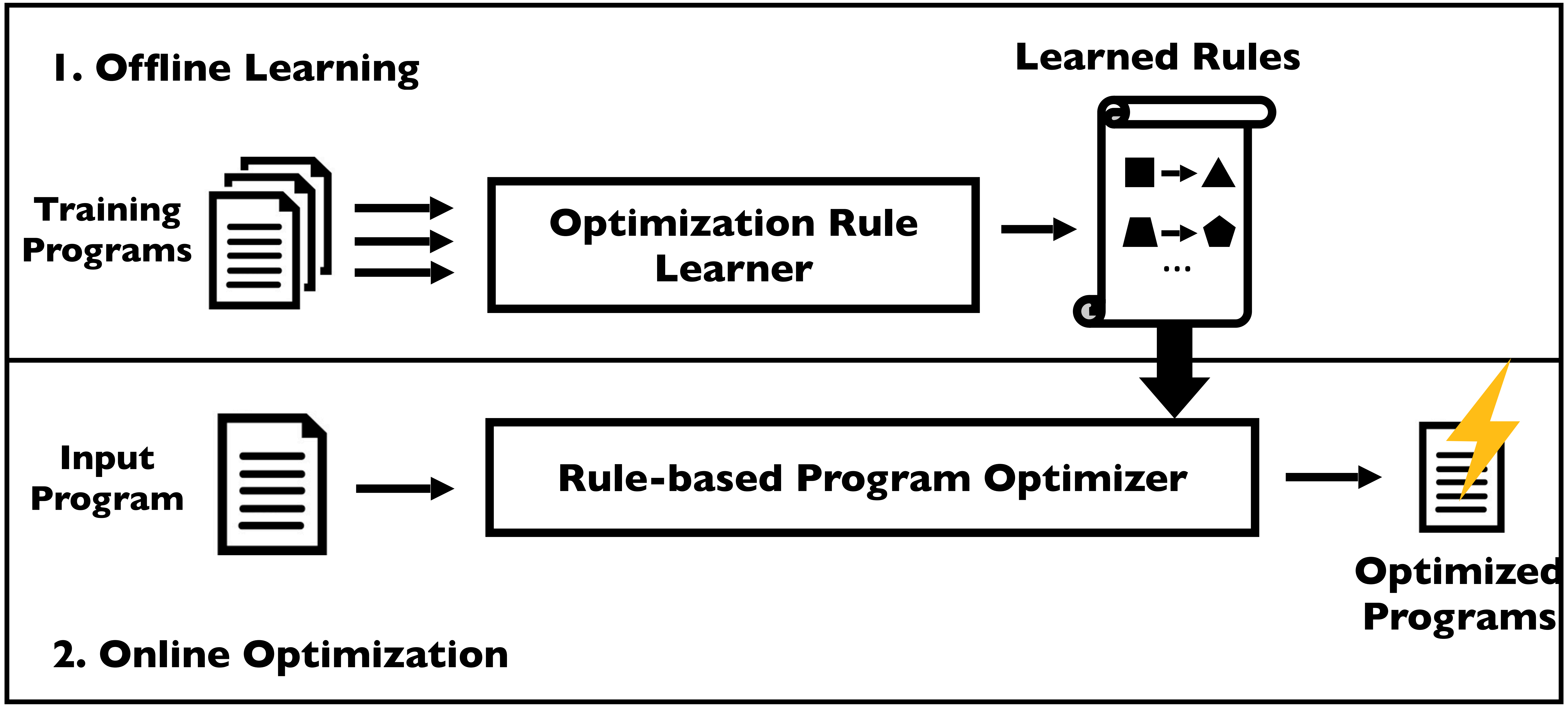
Our Solution

Discovering new rules (by *Program synthesis*) + Systematically applying the rules (by *Term rewriting*) + Finding optimal solutions by exhaustive search (by *Equality saturation*)

When rule discovery is time consuming

: Offline learning +

Online optimization

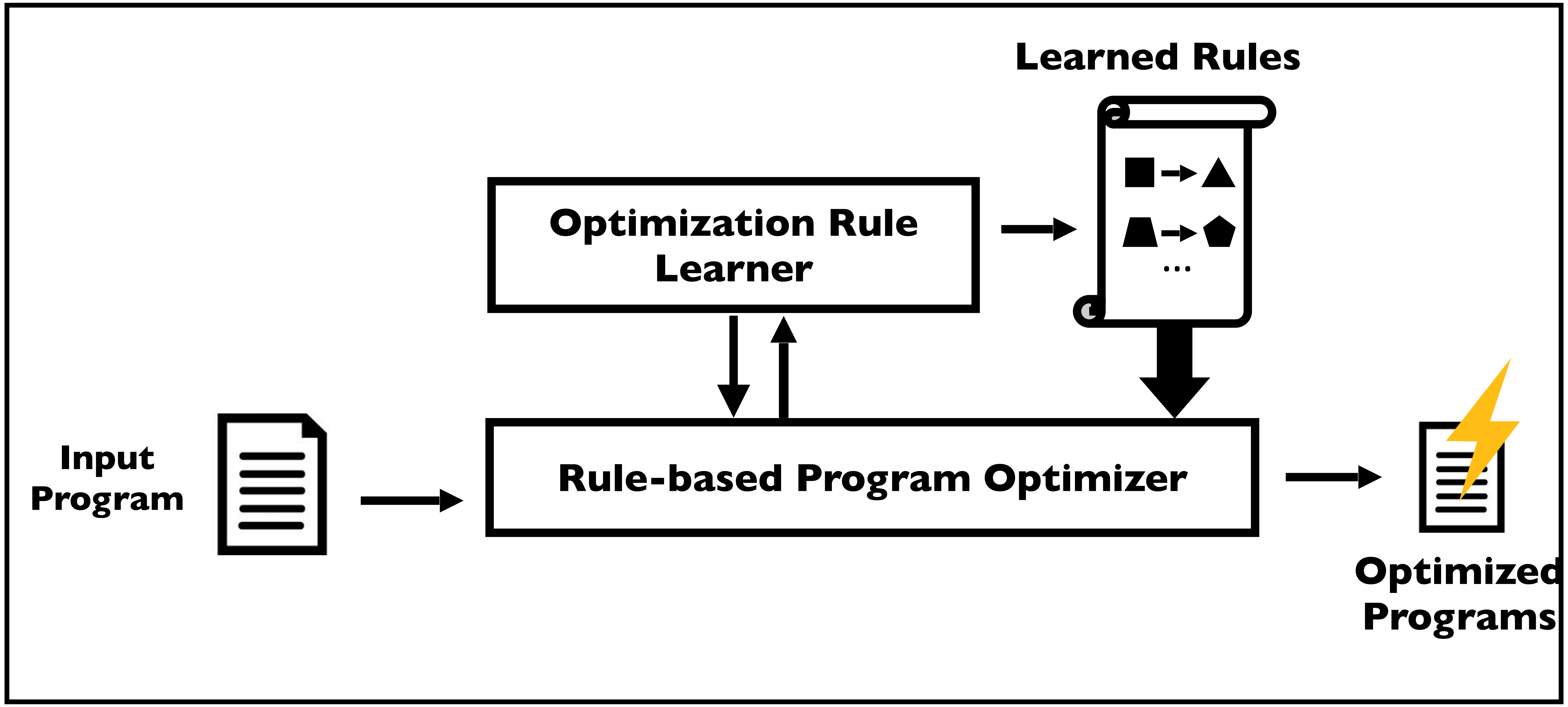


Our Solution

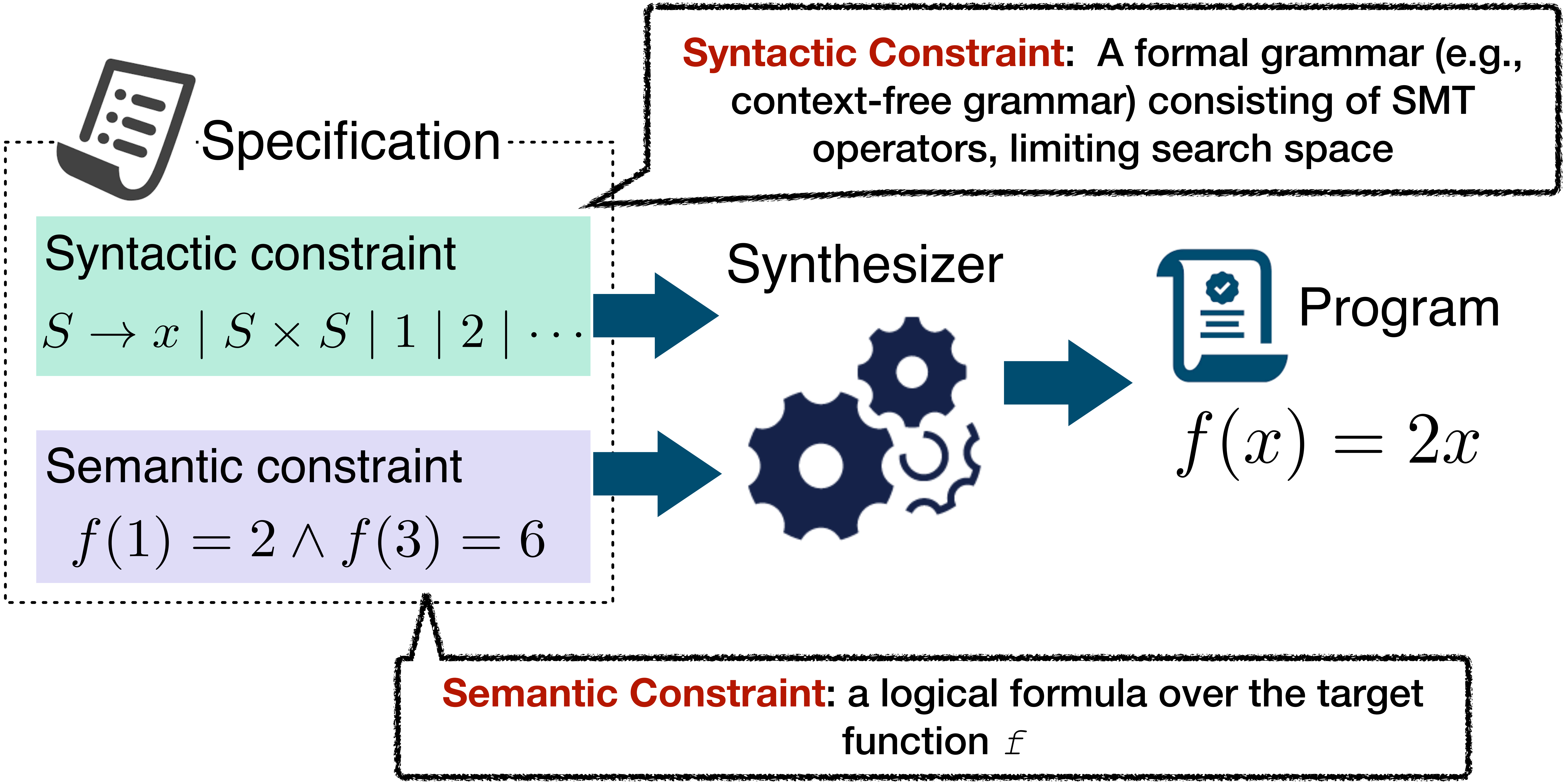
Discovering new rules (by *Program synthesis*) + Systematically applying the rules (by *Term rewriting*) + Finding optimal solutions by exhaustive search (by *Equality saturation*)

When rule discovery is cheap

: Online learning + optimization

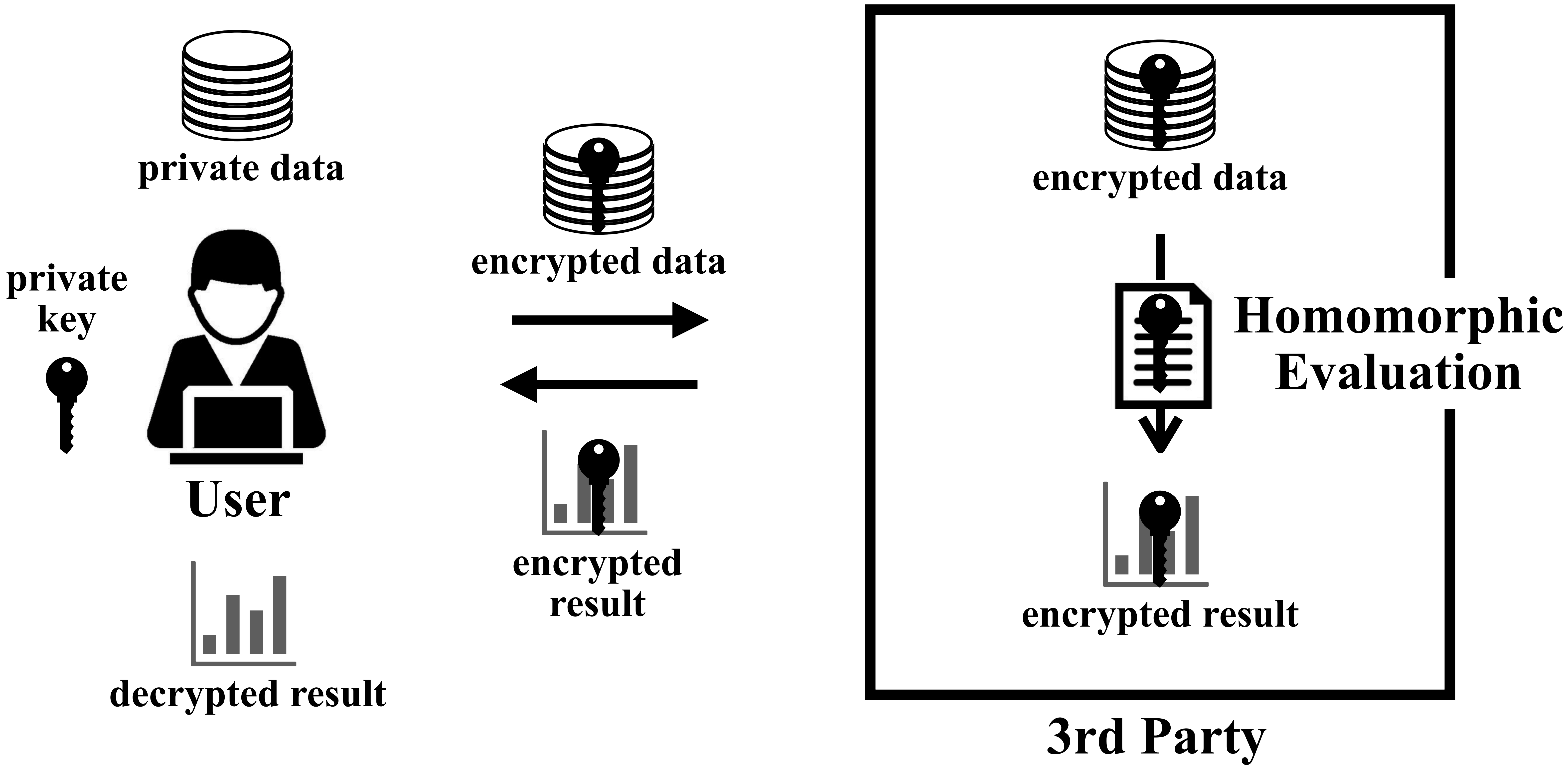


Enabled by *Program Synthesis*



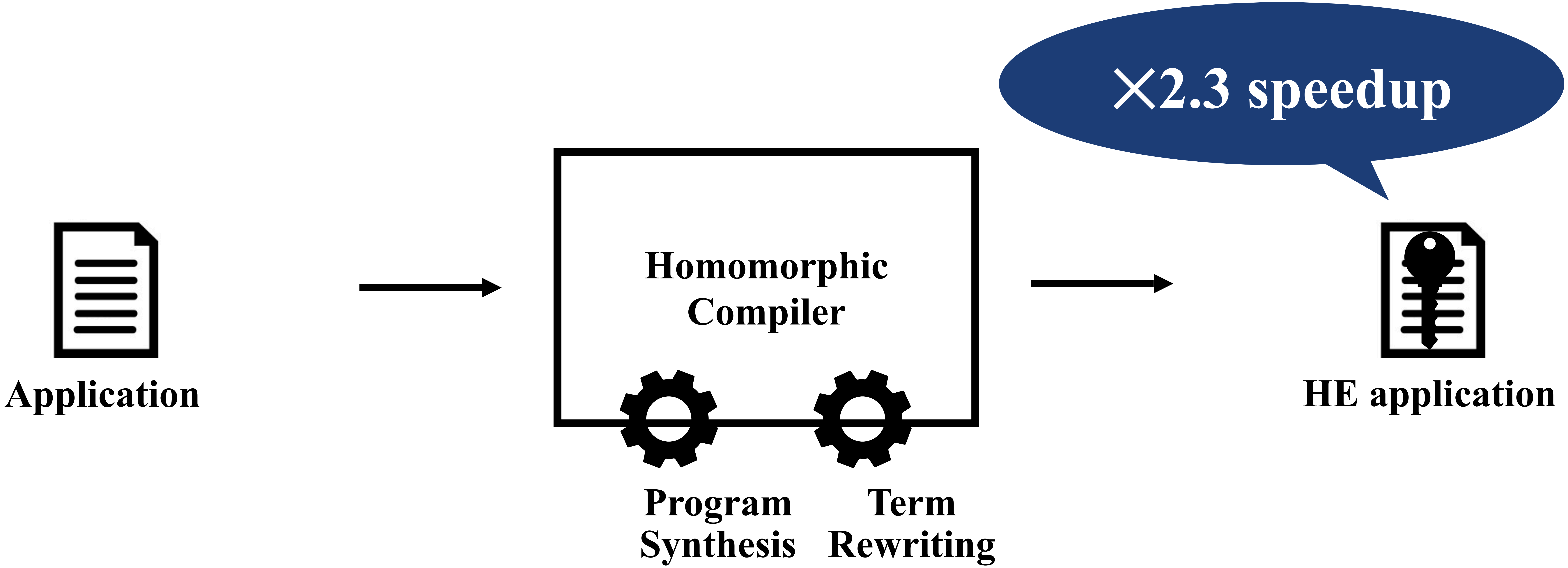
Case 1: Homomorphic Encryption (HE) (1/2)

- Allows computation on encrypted data
- Enables the outsourcing of private data storage/processing



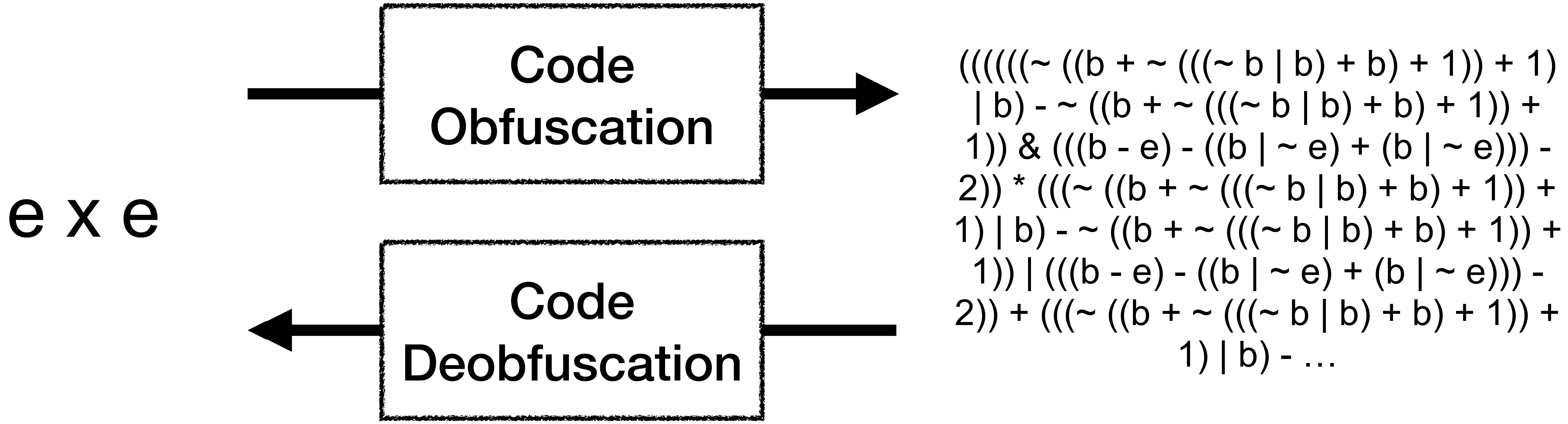
Case 1: Homomorphic Encryption (HE) (1/2)

- HE Compilers generate HE applications automatically
- Better optimization effect than the SOTA with hand-crafted rules



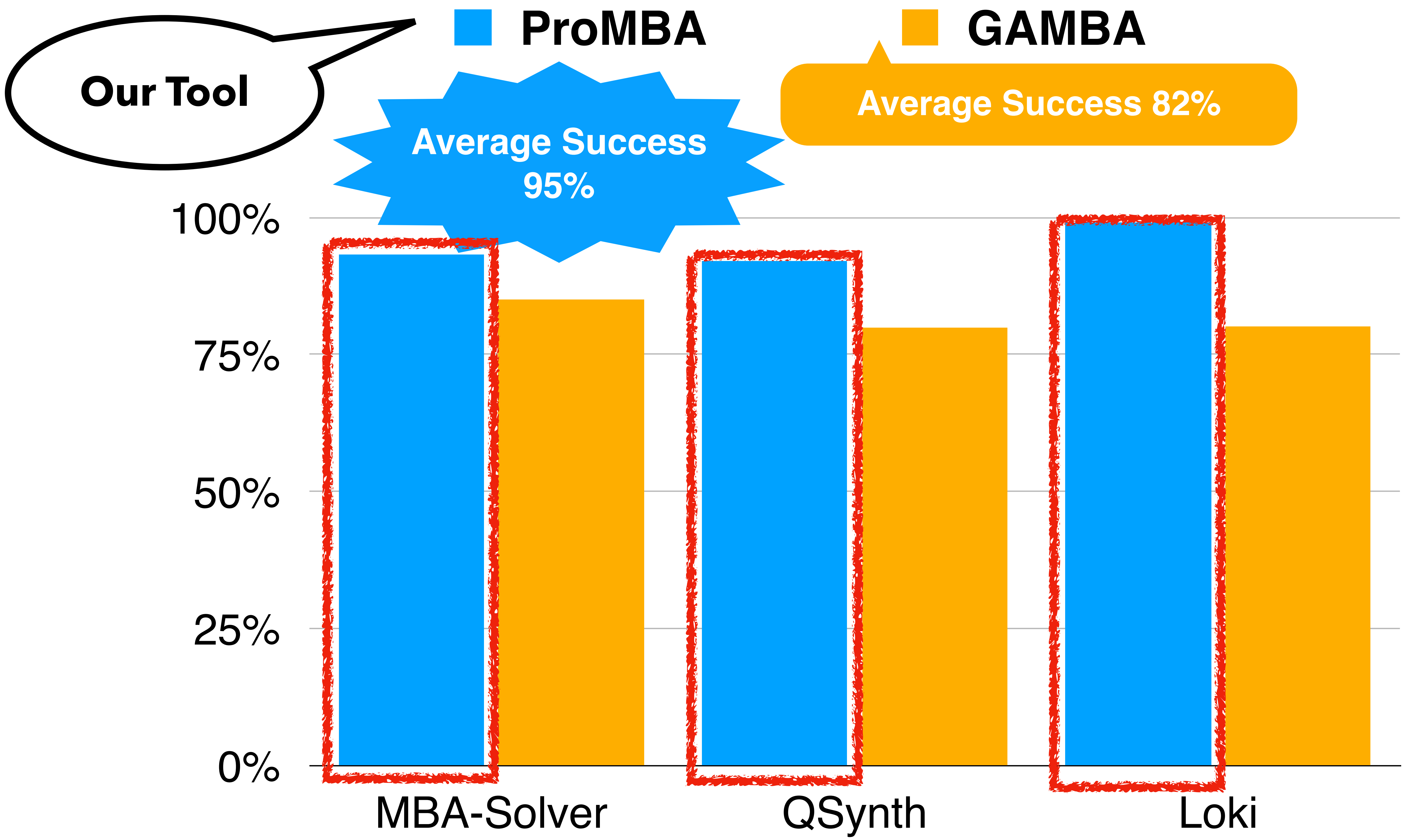
Case 2: Simplifying Obfuscated Code (1/2)

- Obfuscation: transforming programs into complex ones
  - Evasion of malware detection 👿 Copyright protection 😇
- De-obfuscation: simplifying obfuscated programs



Case 2: Simplifying Obfuscated Code (2/2)

- Success : generating simpler or as simple as original code
- Higher success rate than the SOTA based on handcrafted rules



Papers

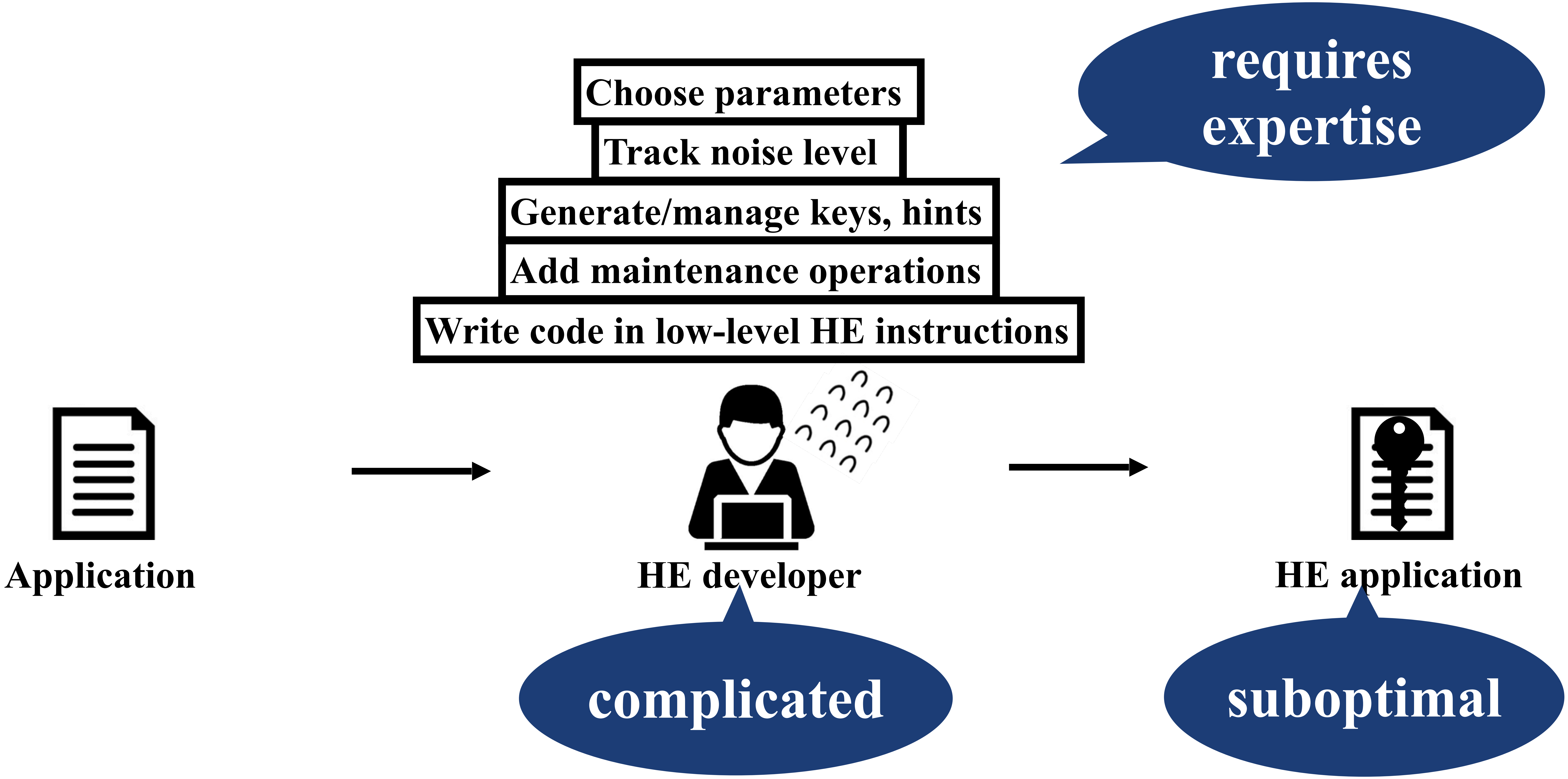
- Case 1 : Optimizing compiler for homomorphic encryption
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search,* **ACM TOPLAS 2023**
  - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting,* **ACM PLDI 2020**
- Case 2: Deobfuscation of bit-manipulating code
  - Jaehyung Lee and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting,* **ACM CCS 2023**
  - Jaehyung Lee, Seoksu Lee, Eunsun Cho and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Equality Saturation,* **IEEE TDSC (Submitted)**
- Core technology: high-performance program synthesis
  - Yongho Yoon, Woosuk Lee, and Kwangkeun Yi, Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. **ACM PLDI 2023**

Contents

- **Case 1 : Optimizing compiler for homomorphic encryption**
    - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search*, **ACM TOPLAS 2023**
    - Dongkwon Lee, Woosuk Lee, Hakjoo Oh and Kwangkeun Yi, *Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting*, **ACM PLDI 2020**
- Case 2: Deobfuscation of bit-manipulating code
- Lessons from the two cases
- Core technology: high-performance program synthesis

Homomorphic Encrytion (1/2)

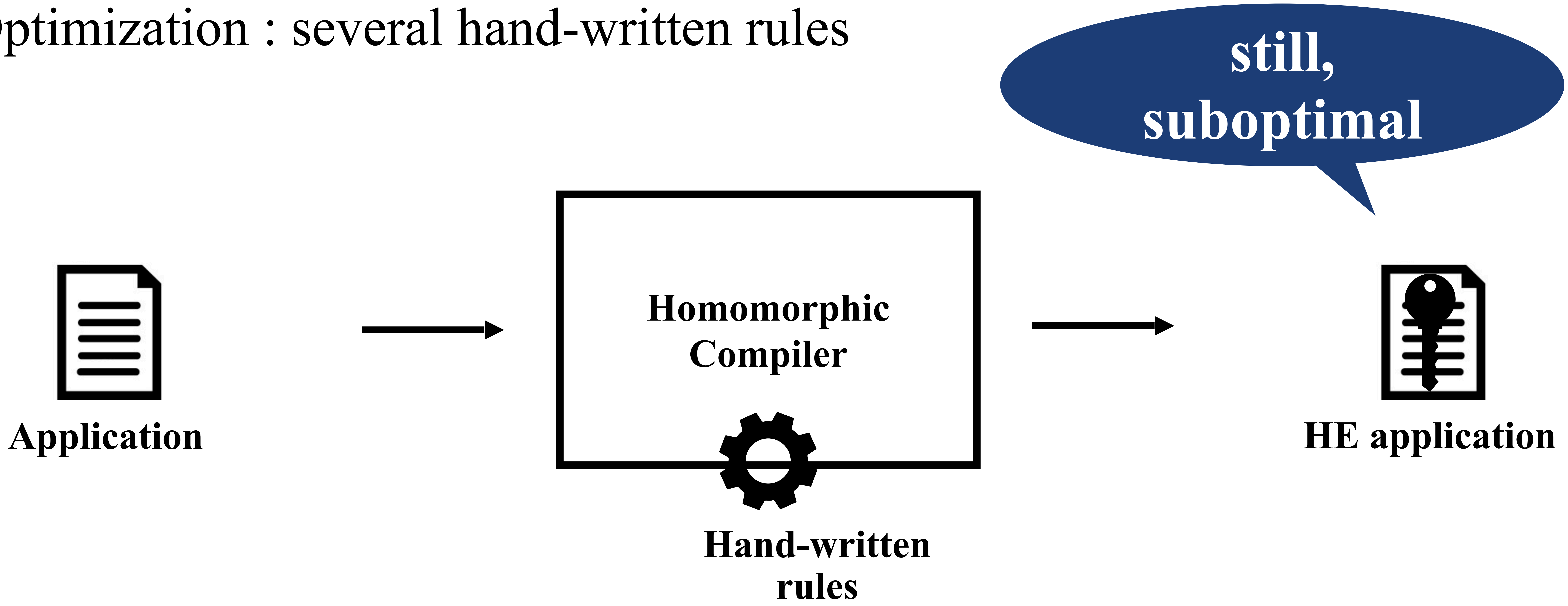
Building HE applications



Homomorphic Encrytion (2/2)

Existing Homomorphic Compiler

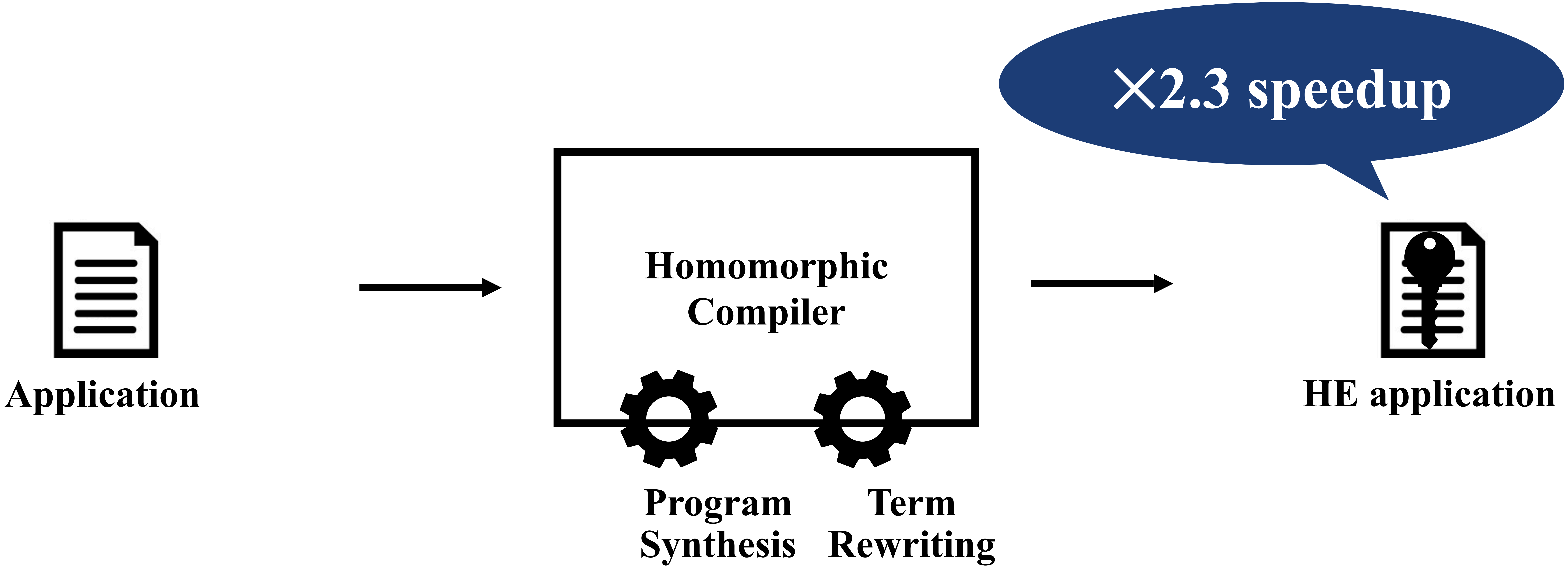
- Generates HE applications automatically
- Optimization : several hand-written rules



Our Contribution

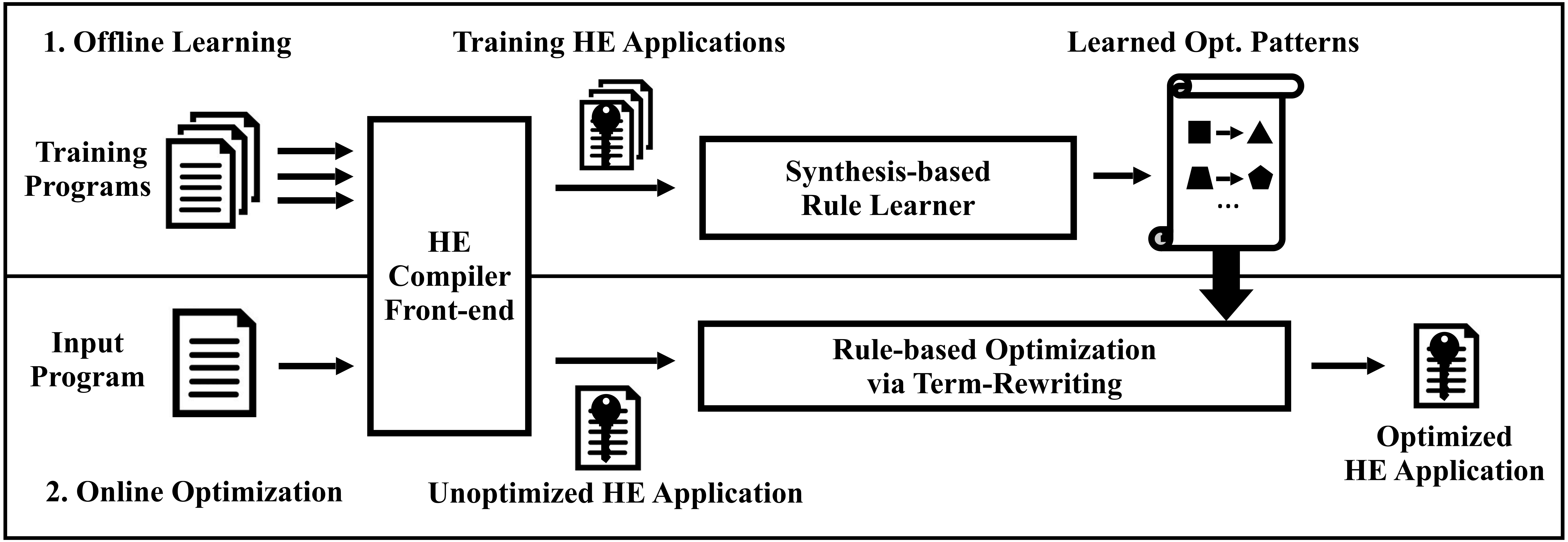
Automatic, Aggressive HE optimization Framework

- Generates HE applications automatically
- Optimization : machine-found rules by program synthesis + applying by term rewriting



Lobster

- Offline Learning via Program Synthesis + Online Optimization via Term Rewriting



Simple HE Scheme

- Based on approximate common divisor problem
- p : integer as a secret key
- q : random integer
- $r(\ll |p|)$: random noise for security

$$Enc_p(\mu \in \{0,1\}) = pq + 2r + \mu$$
$$Dec_p(c) = (c \bmod p) \bmod 2$$
$$Dec_p(Enc_p(\mu)) = Dec_p(pq + 2r + \mu) = \mu$$

- For ciphertexts $\underline{\mu_i} \leftarrow Enc_p(\mu_i)$, the following holds

$$Dec_p(\underline{\mu_1} + \underline{\mu_2}) = \mu_1 + \mu_2$$
$$Dec_p(\underline{\mu_1} \times \underline{\mu_2}) = \mu_1 \times \mu_2$$

- The scheme can evaluate all boolean circuits as $+$ and $\times$ in $\mathbb{Z}_2 = \{0,1\}$ are equal to XOR and AND

Performance Hurdle : Growing Noise

- Noise increases during homomorphic operations.
- For $\underline{\mu_i} = pq_i + 2r_i + \mu_i$

$$\underline{\mu_1} + \underline{\mu_2} = p(q_1 + q_2) + \boxed{2(r_1 + r_2) + (\mu_1 + \mu_2)} \text{ double increase}$$

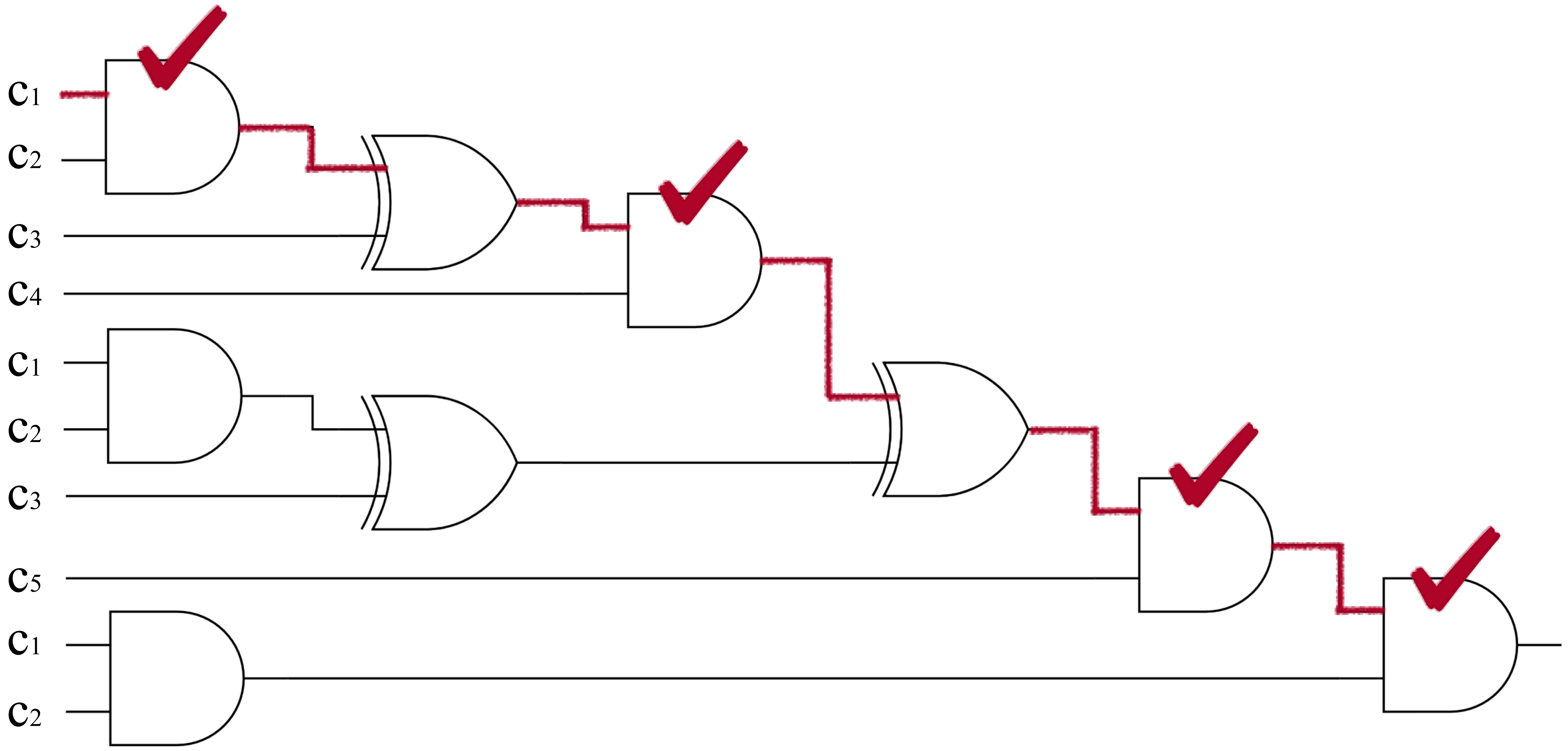
$$\underline{\mu_1} \times \underline{\mu_2} = p(pq_1q_2 + \cdots) + \boxed{2(2r_1r_2 + r_1\mu_2 + r_2\mu_1) + (\mu_1 \times \mu_2)} \text{ quadratic increase}$$

noise

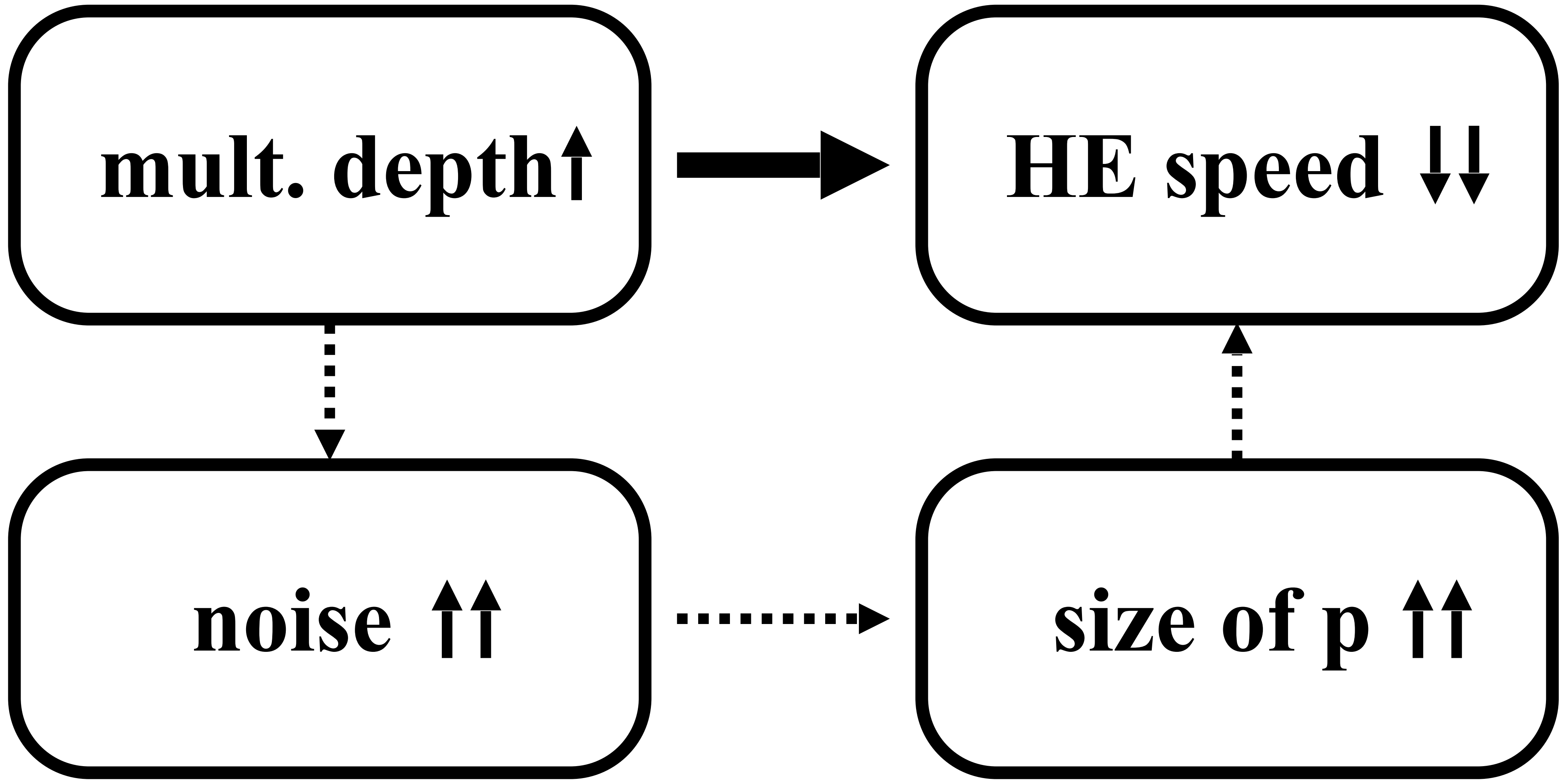
- **if (noise $> p$) then incorrect results**

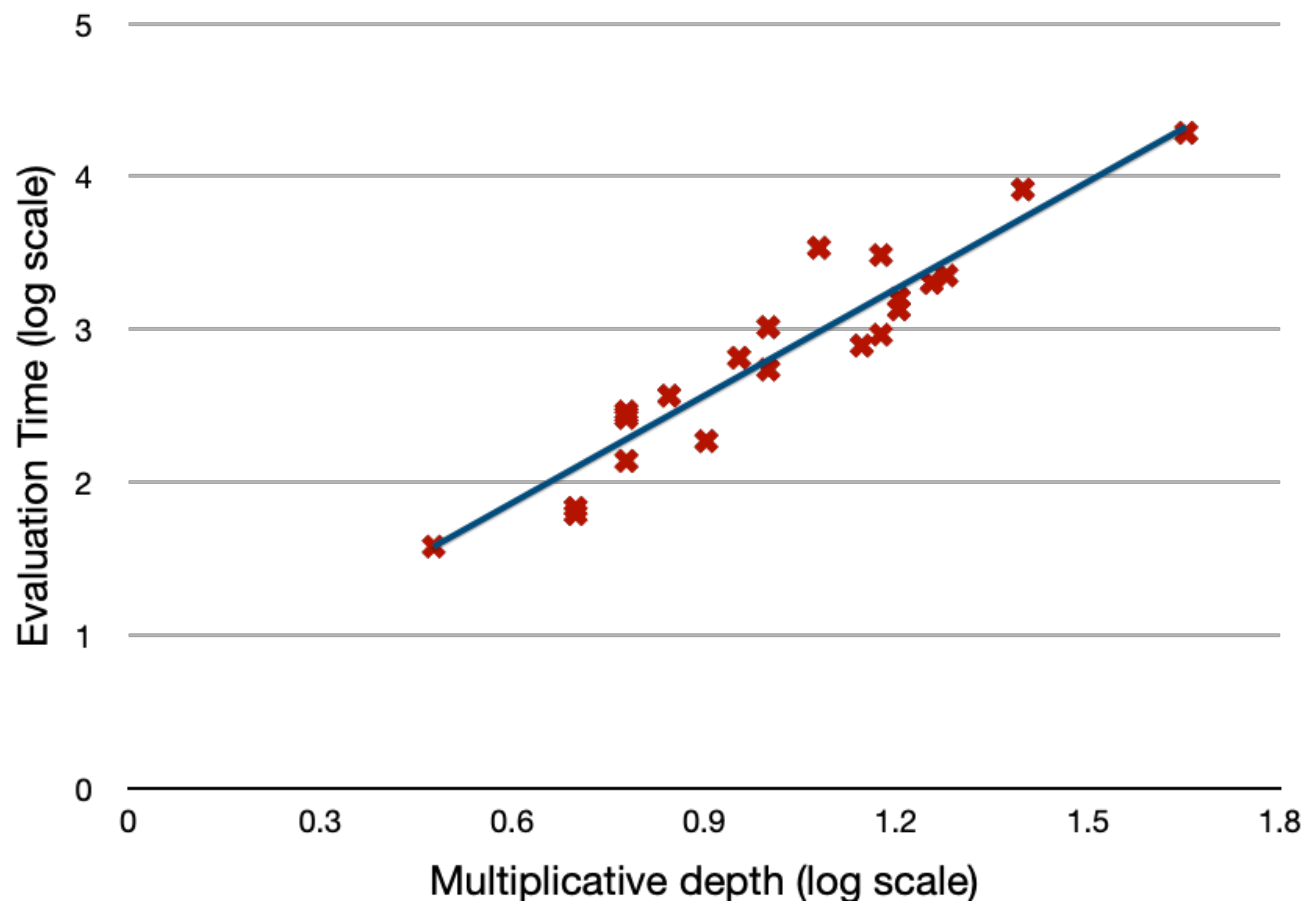
Multiplicative Depth : a Decisive Performance Factor

The maximum number of sequential multiplications from input to output



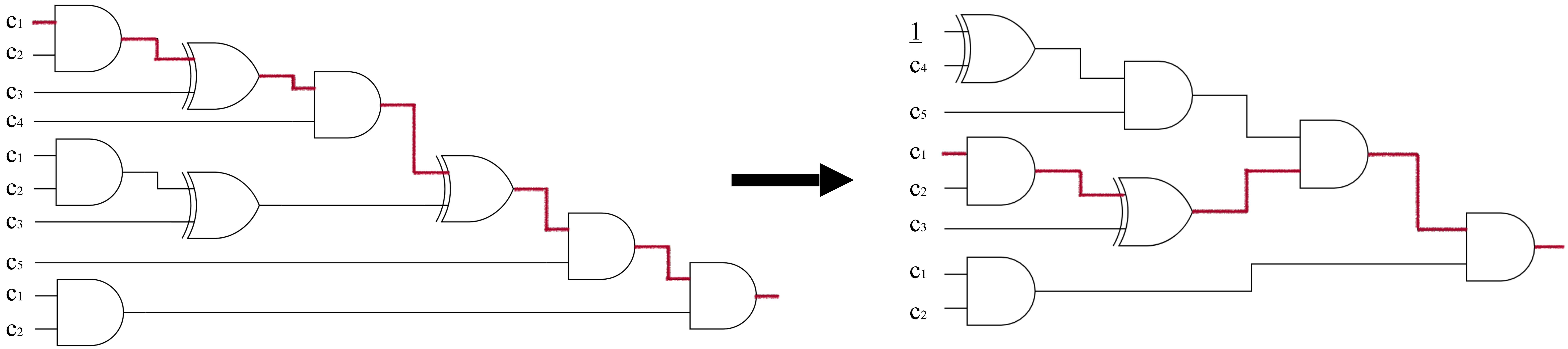
depth 4





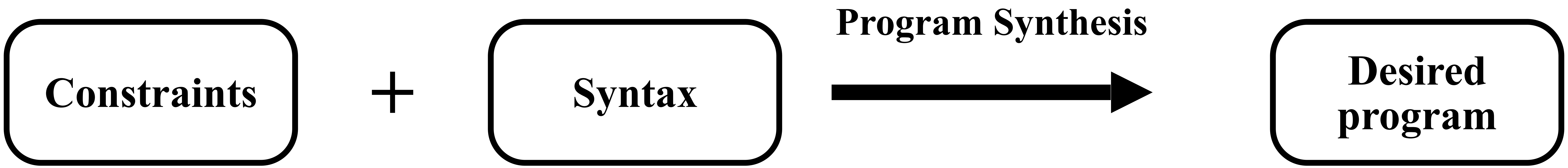
What is HE optimization?

- Finding a new circuit that has smaller mult. depth



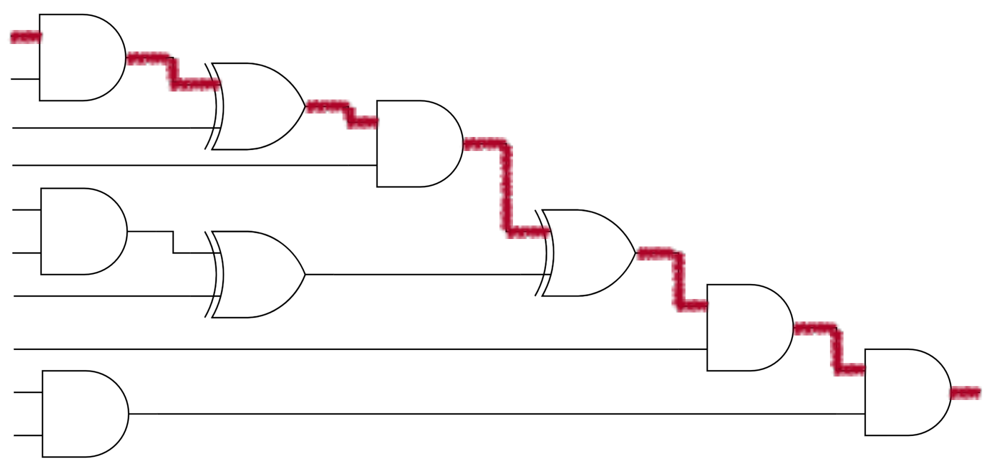
depth 4 depth 3

HE optimization via Synthesis



HE optimization via Synthesis

Constraints + Syntax —Program Synthesis→ Desired program

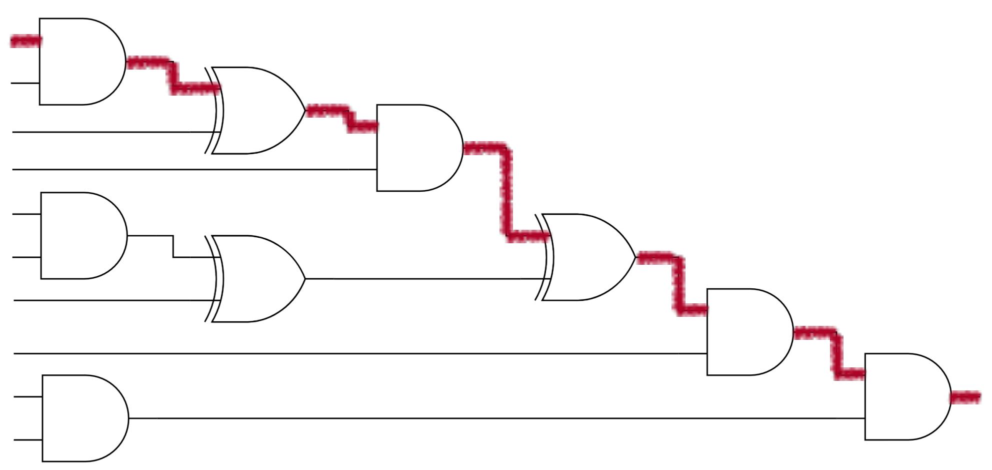


depth 4

same semantics

HE optimization via Synthesis

Constraints + **Syntax** —**Program Synthesis**→ **Desired program**



depth 4

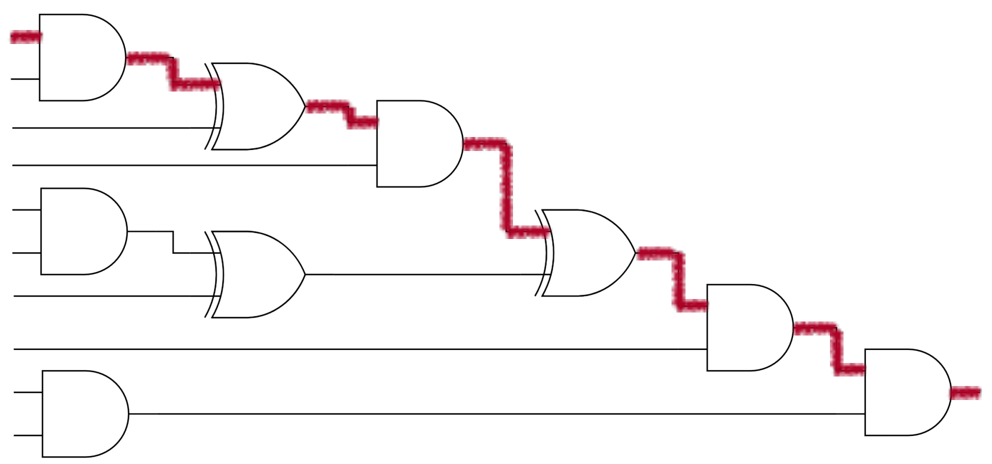
same semantics

$$
\begin{aligned}
S &\rightarrow d_3 \\
d_3 &\rightarrow d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 &\rightarrow d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 &\rightarrow d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 &\rightarrow 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{aligned}
$$

depth-restricting syntax

HE optimization via Synthesis

Constraints + **Syntax** —Program Synthesis→ **Desired program**

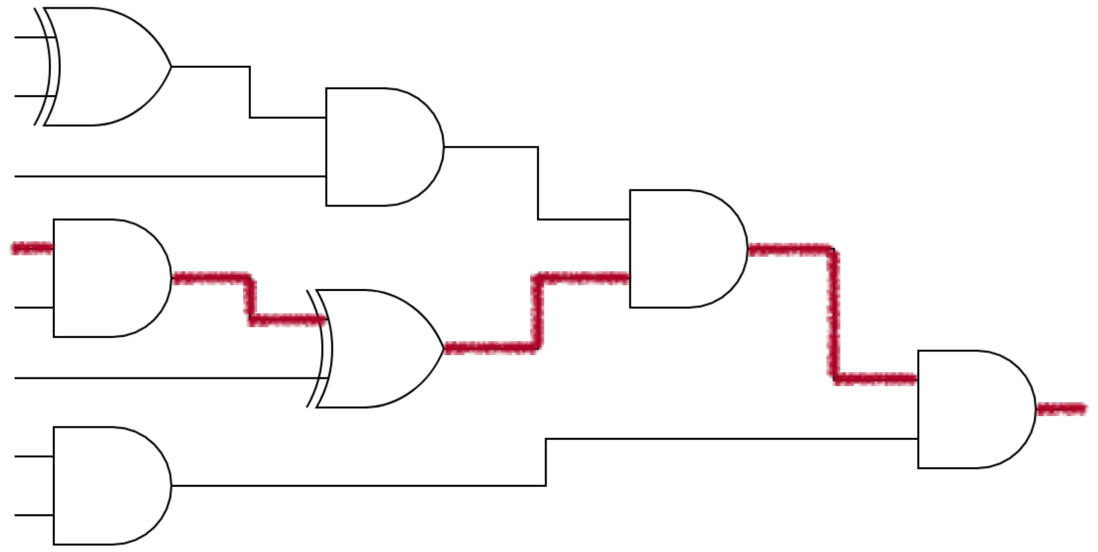


depth 4

same semantics

$$
\begin{array}{rcl}
S & \longrightarrow & d_3 \\
d_3 & \longrightarrow & d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 & \longrightarrow & d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 & \longrightarrow & d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 & \longrightarrow & 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{array}
$$

depth-restricting syntax

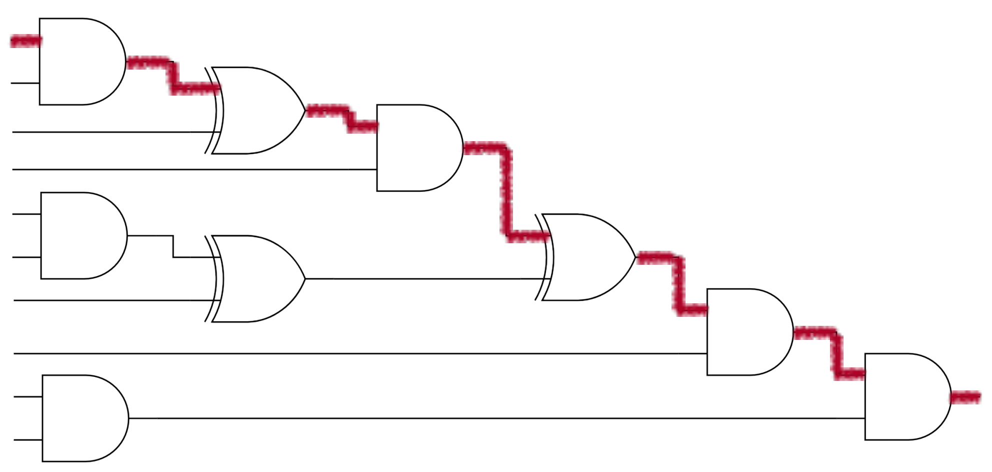


depth 3

optimized HE circuit

HE optimization via Synthesis

Constraints + **Syntax** → (Optimizing Synthesis) → **Desired program**

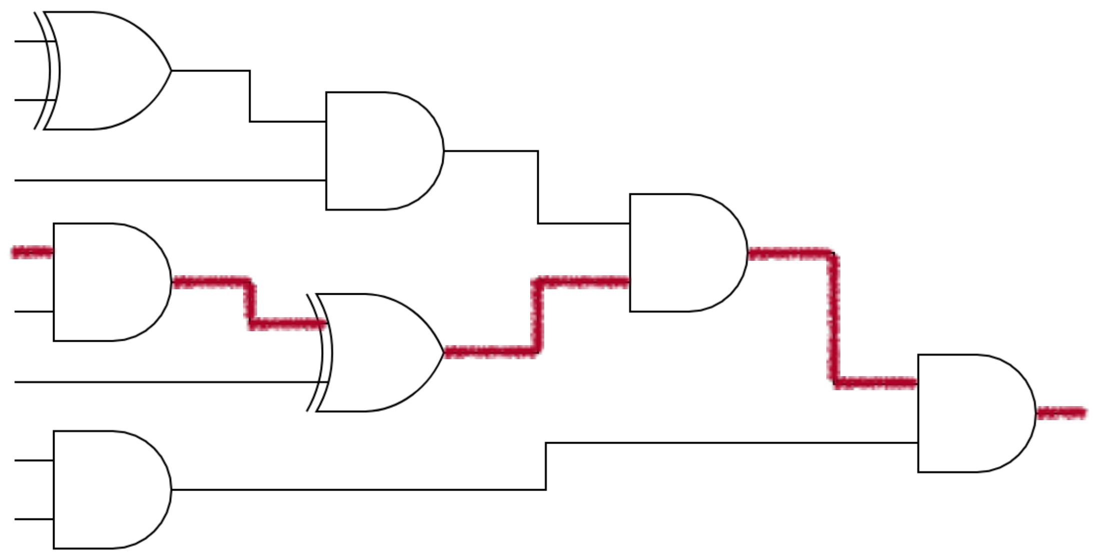


depth 4

same semantics

$$
\begin{aligned}
S &\rightarrow d_3 \\
d_3 &\rightarrow d_2 \wedge d_2 \mid d_3 \oplus d_3 \mid d_2 \\
d_2 &\rightarrow d_1 \wedge d_1 \mid d_2 \oplus d_2 \mid d_1 \\
d_1 &\rightarrow d_0 \wedge d_0 \mid d_1 \oplus d_1 \mid d_0 \\
d_0 &\rightarrow 0 \mid 1 \mid c_1 \mid c_2 \mid c_3 \mid c_4 \mid c_5
\end{aligned}
$$

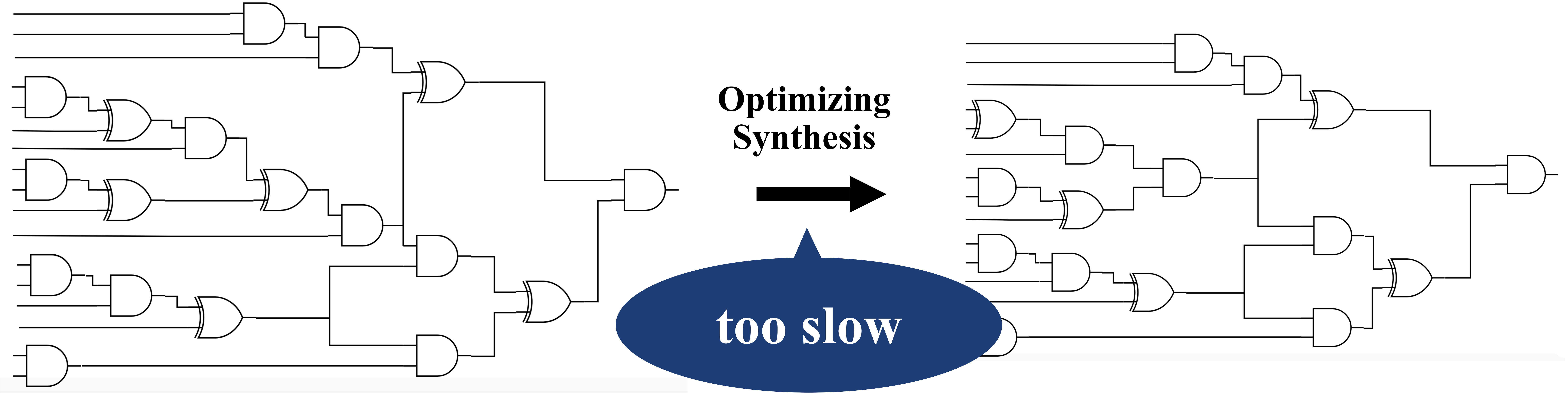
depth-restricting syntax



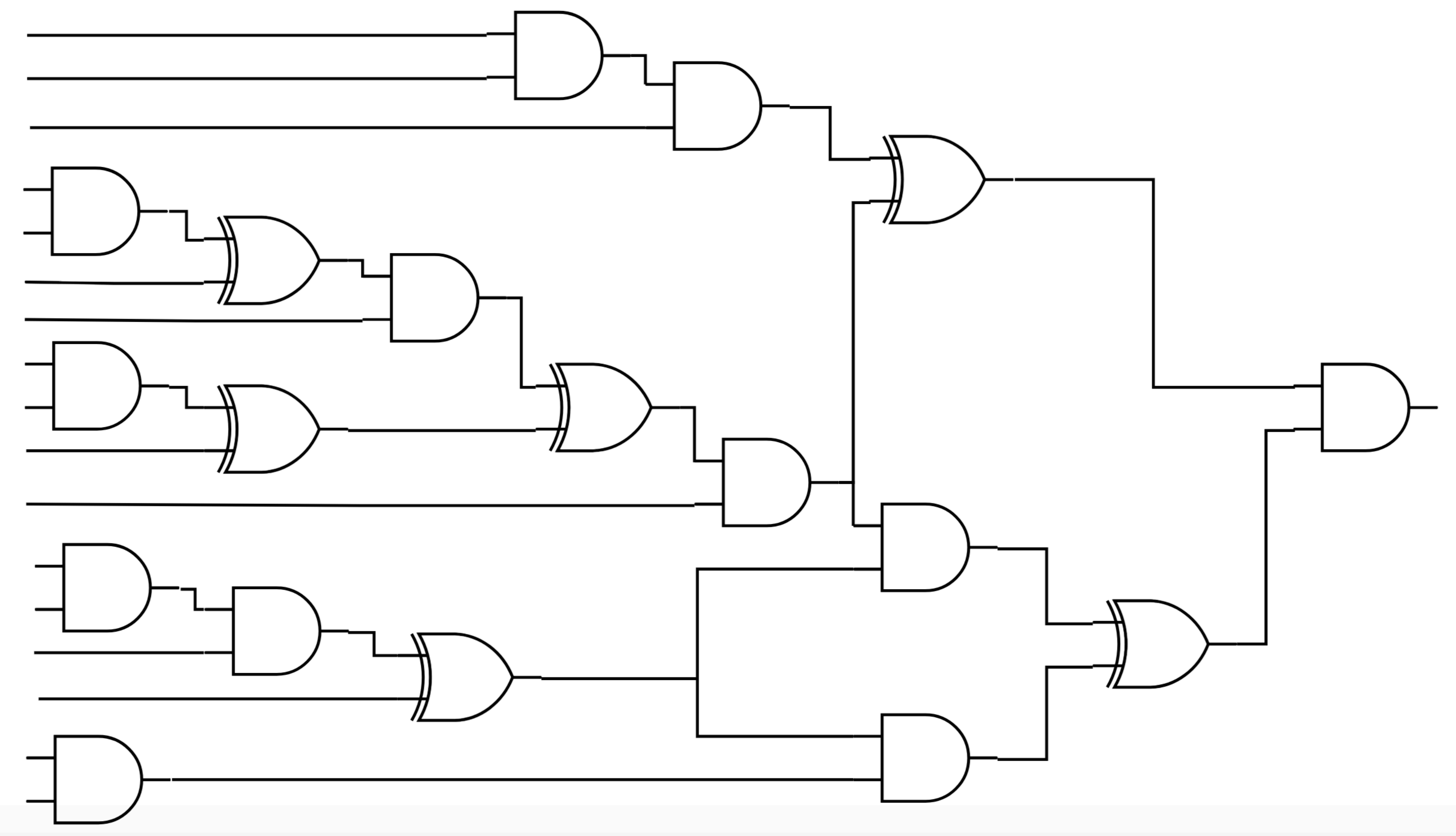
depth 3

optimized HE circuit

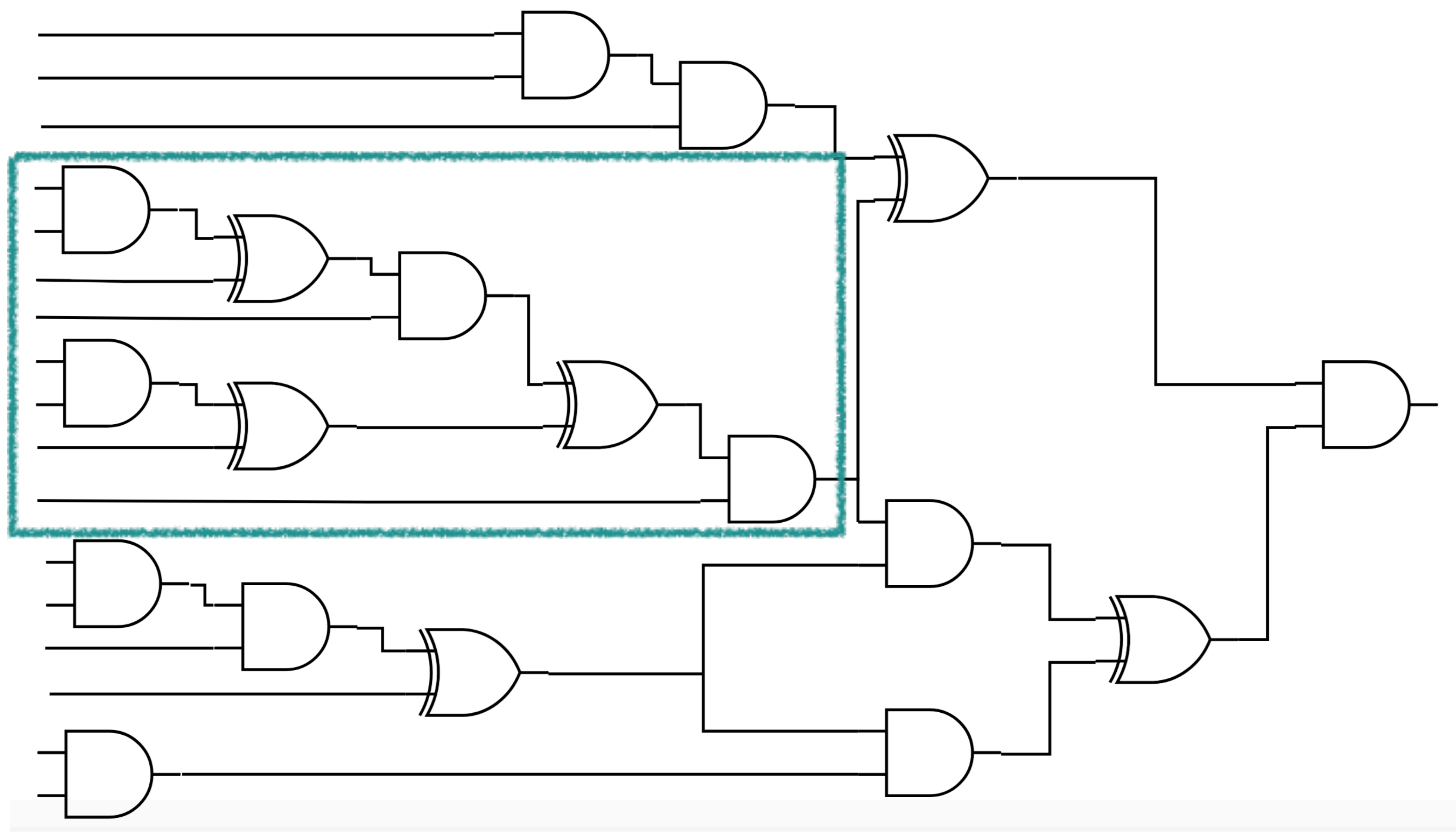
Hurdle : Synthesis Scalability



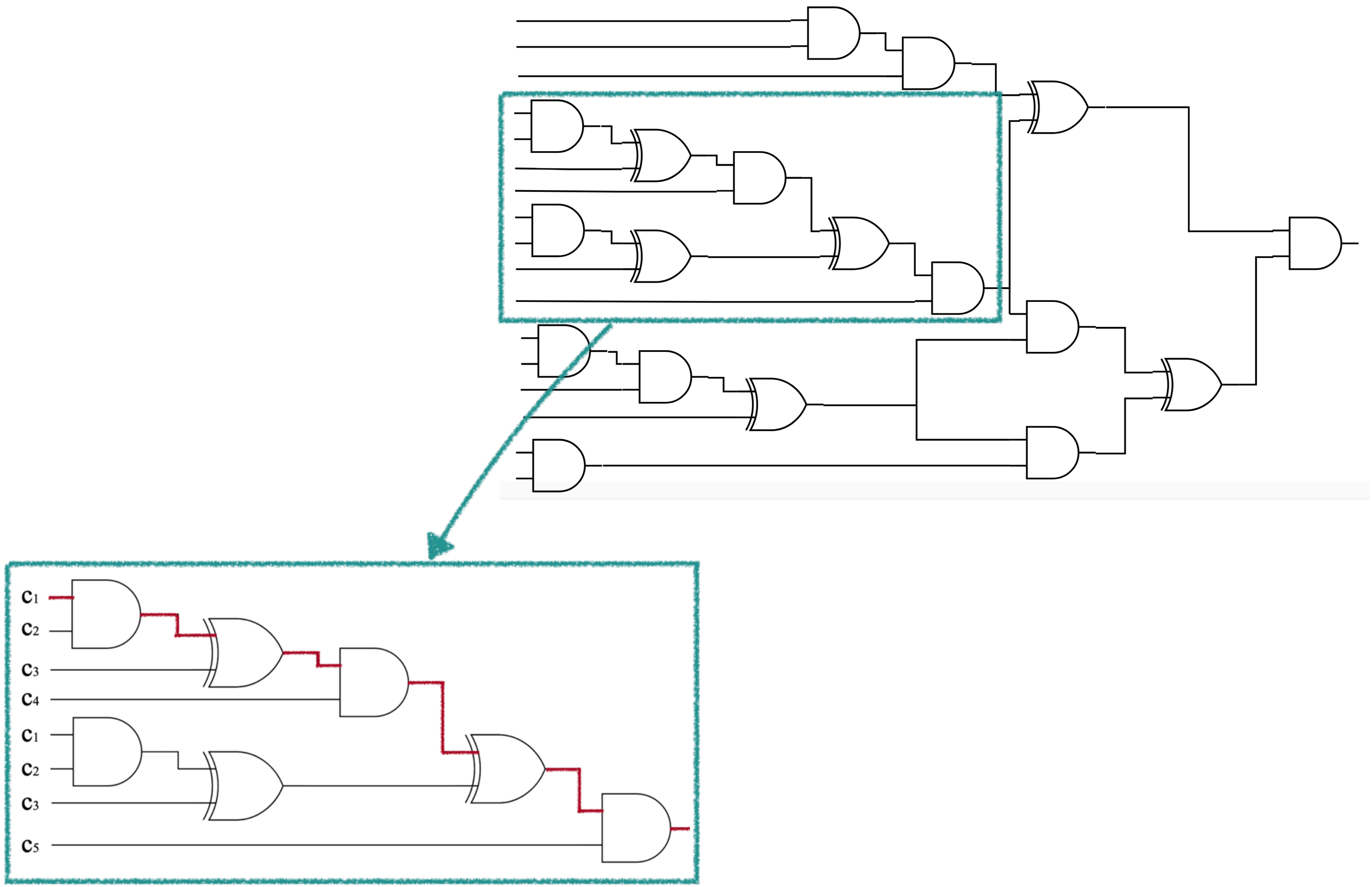
Solution1 : Synthesis via Localization



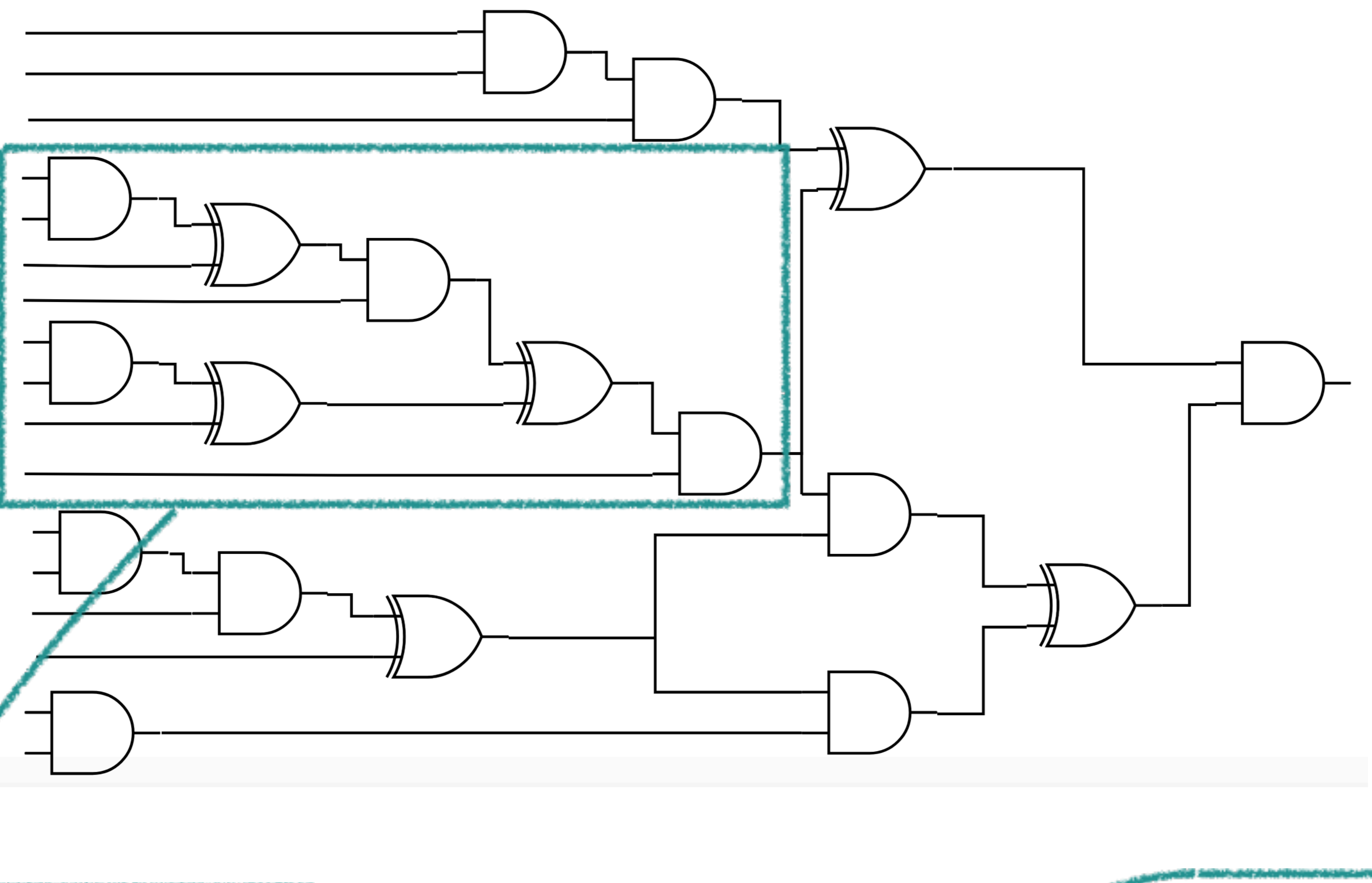
Solution1 : Synthesis via Localization

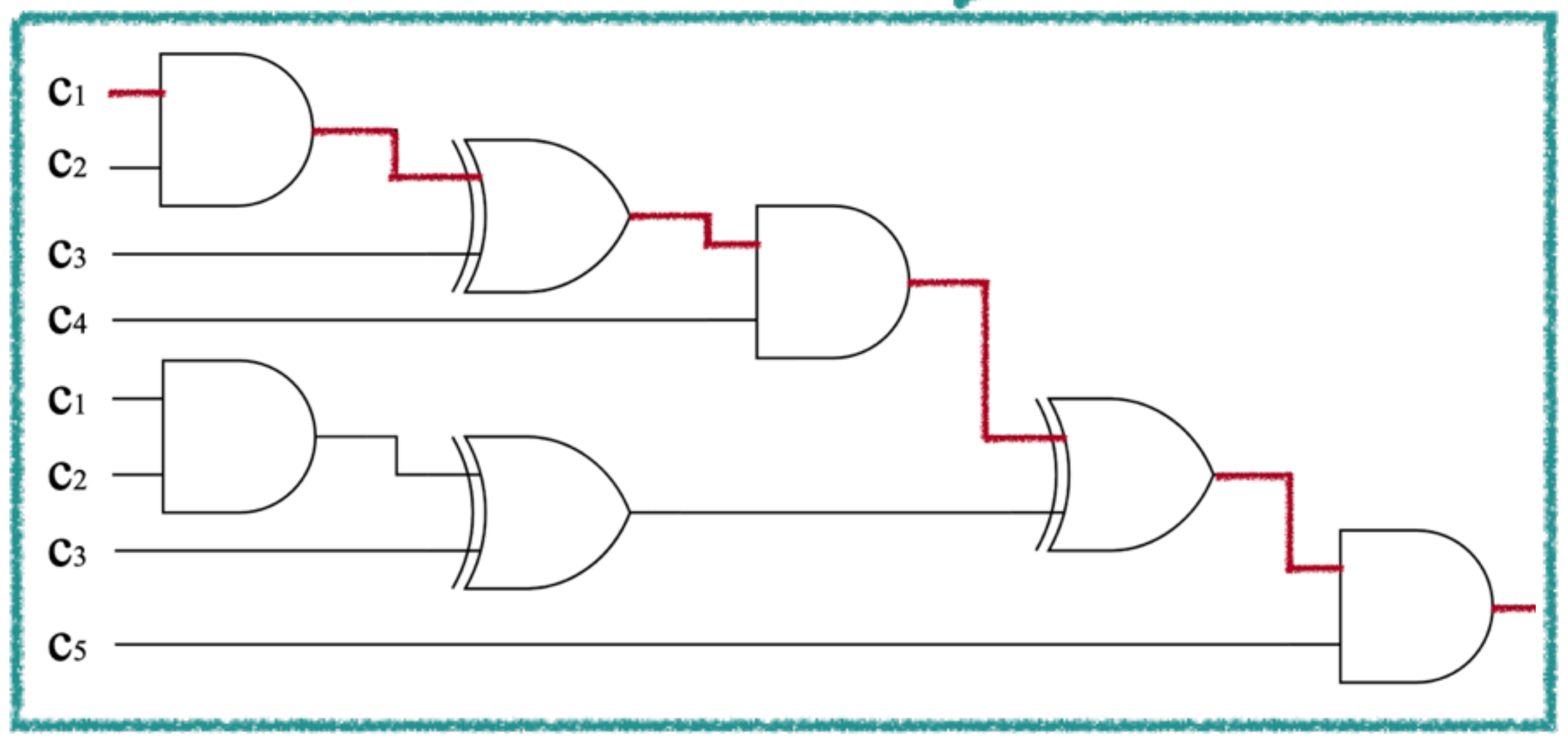


Solution1 : Synthesis via Localization

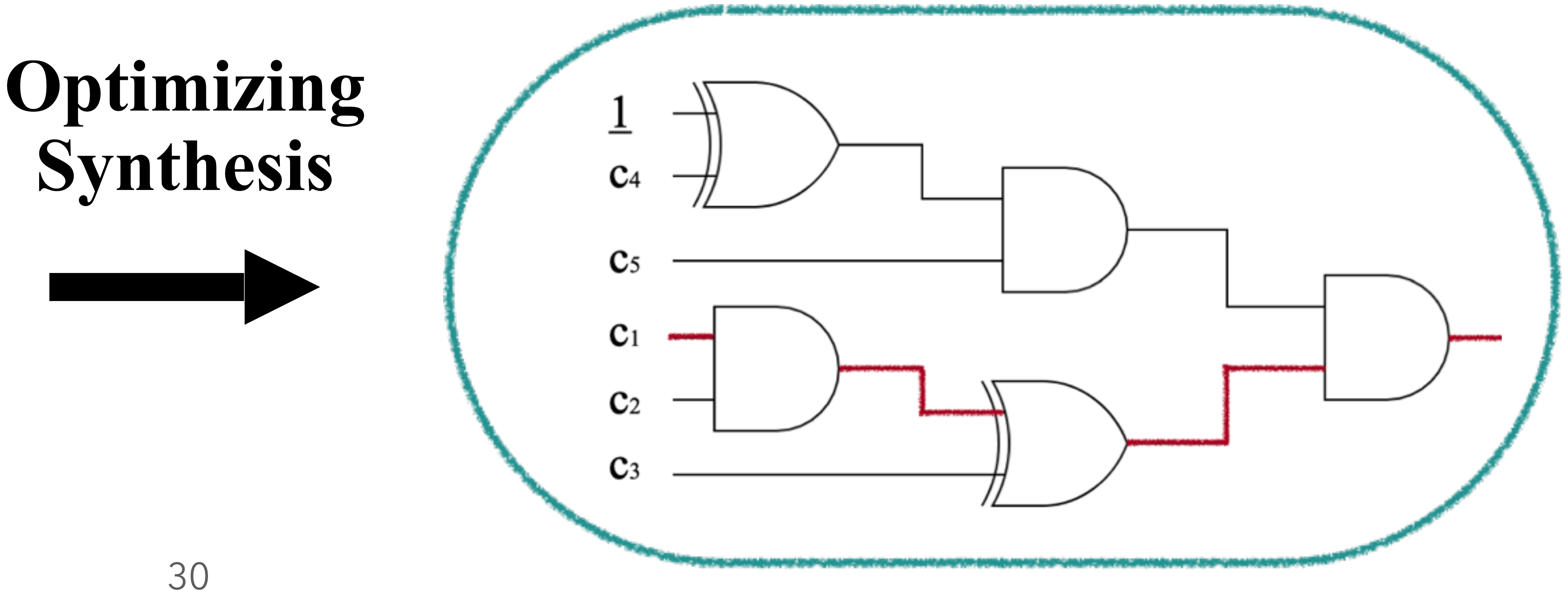


Solution1 : Synthesis via Localization

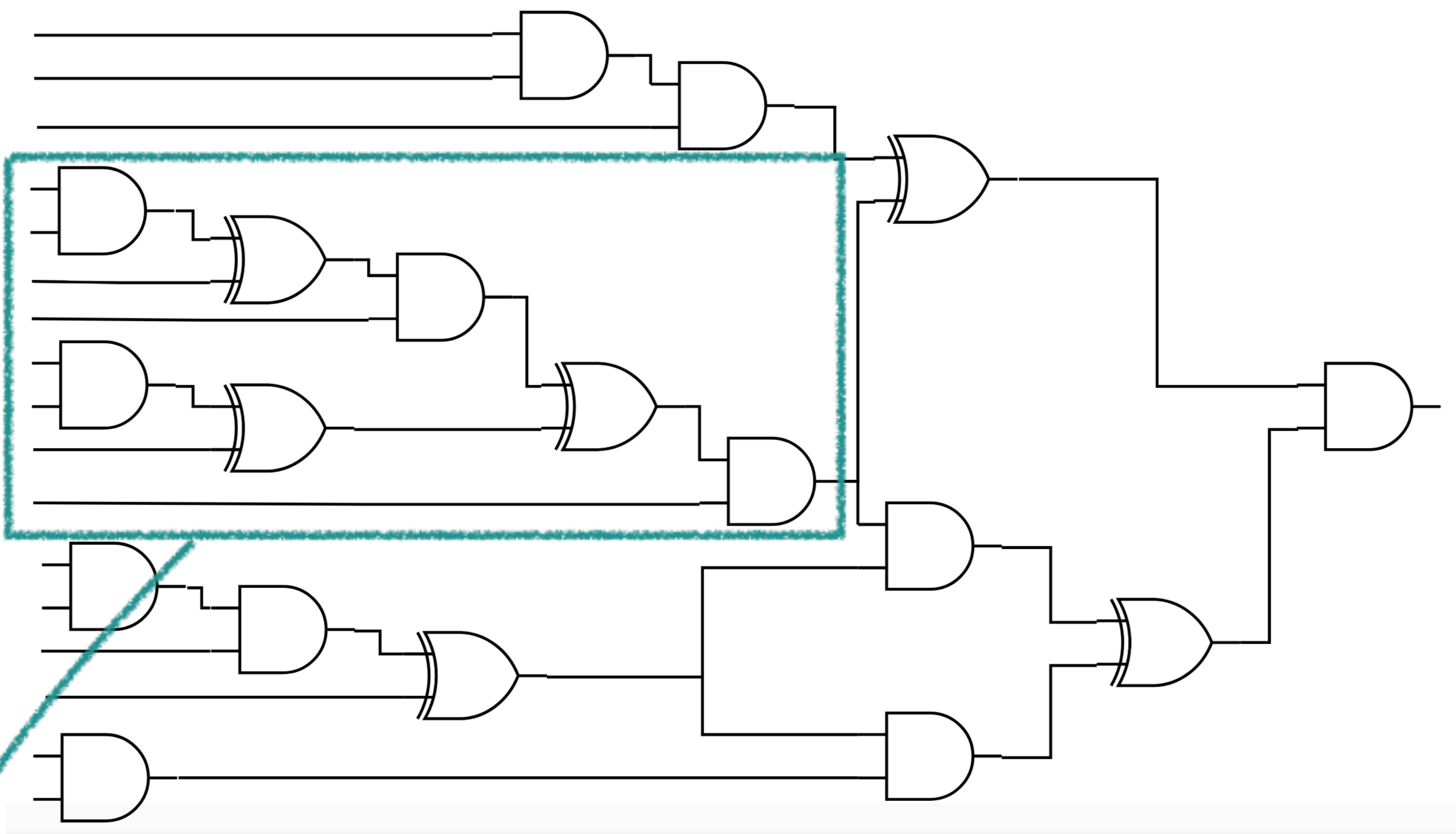


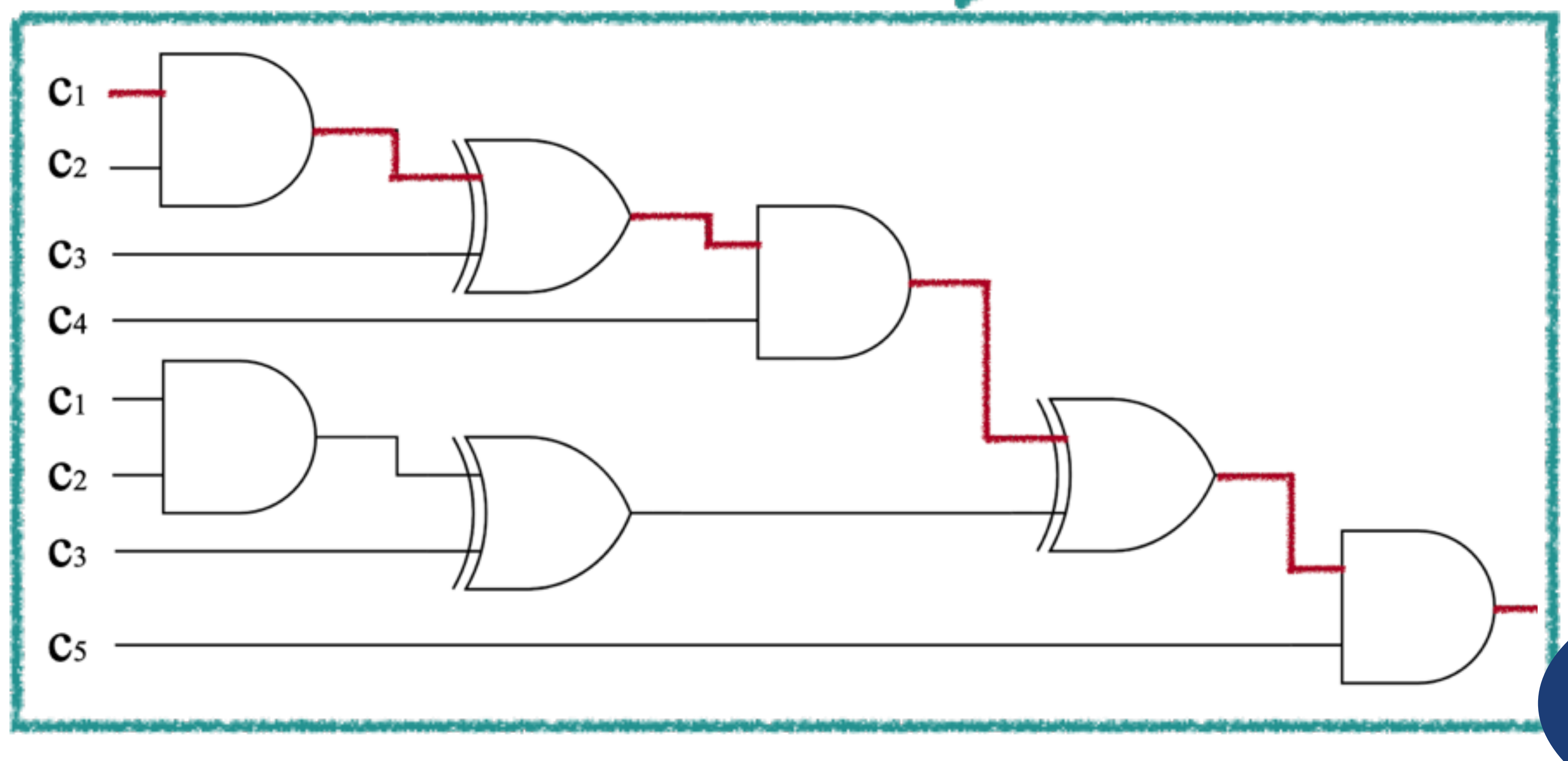


Optimizing Synthesis

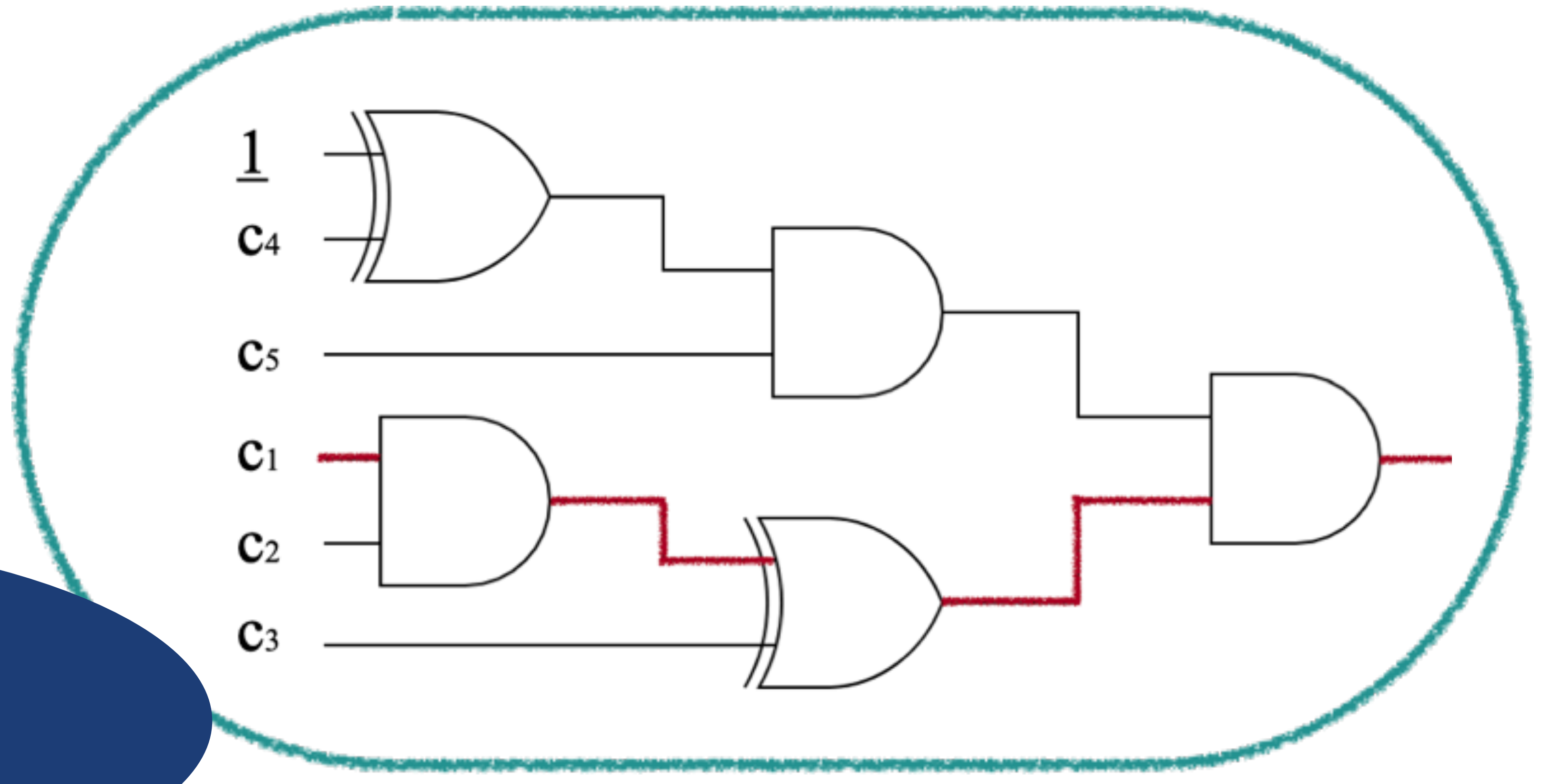


Solution1 : Synthesis via Localization



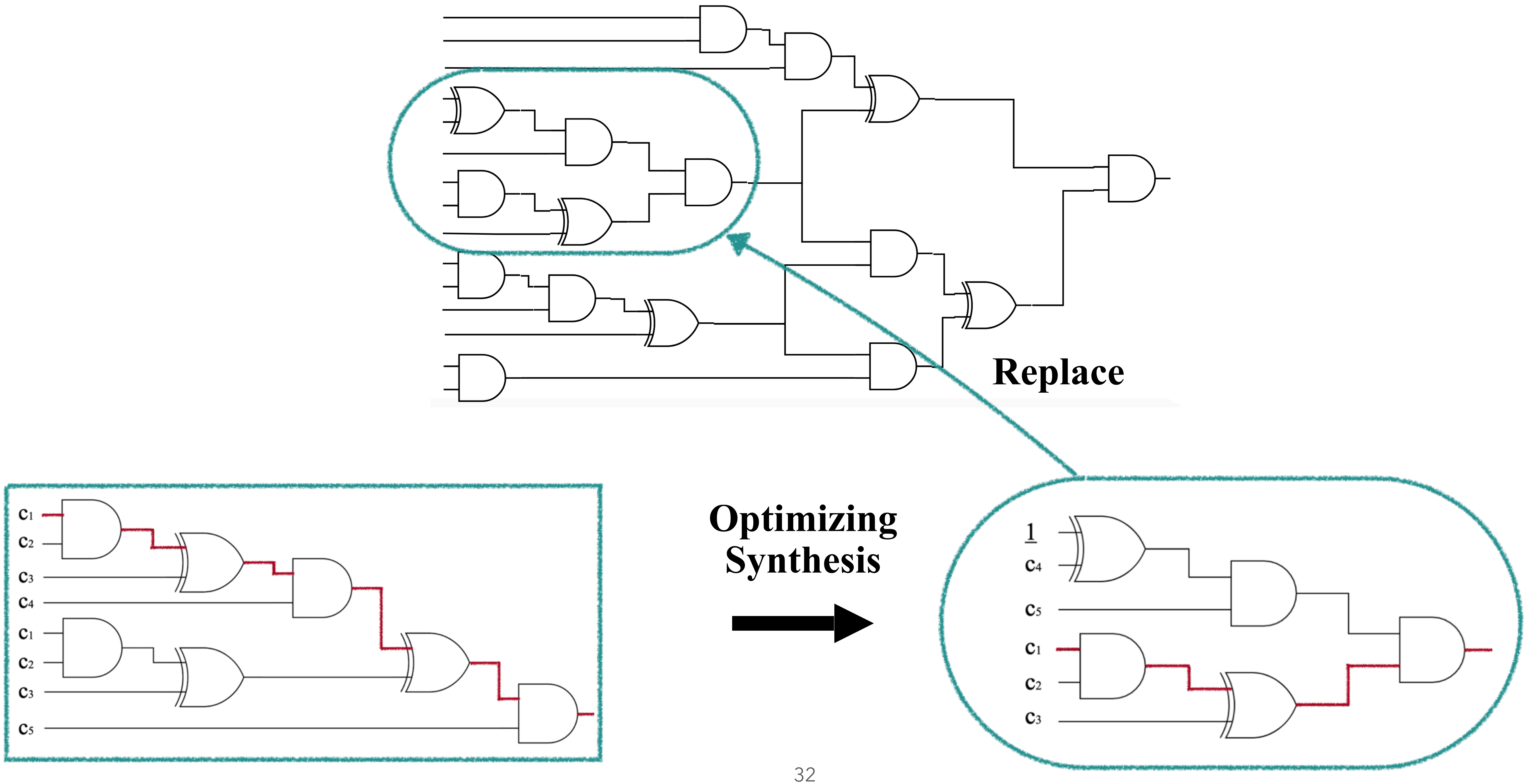


Optimizing Synthesis



scalable

Solution1 : Synthesis via Localization



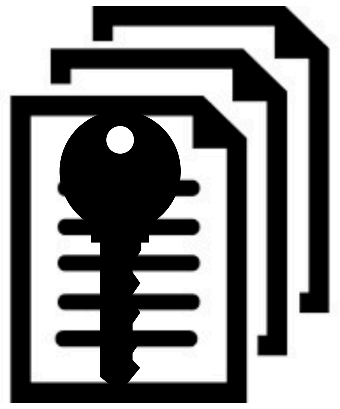
Solution 2: Learning Successful Synthesis Patterns

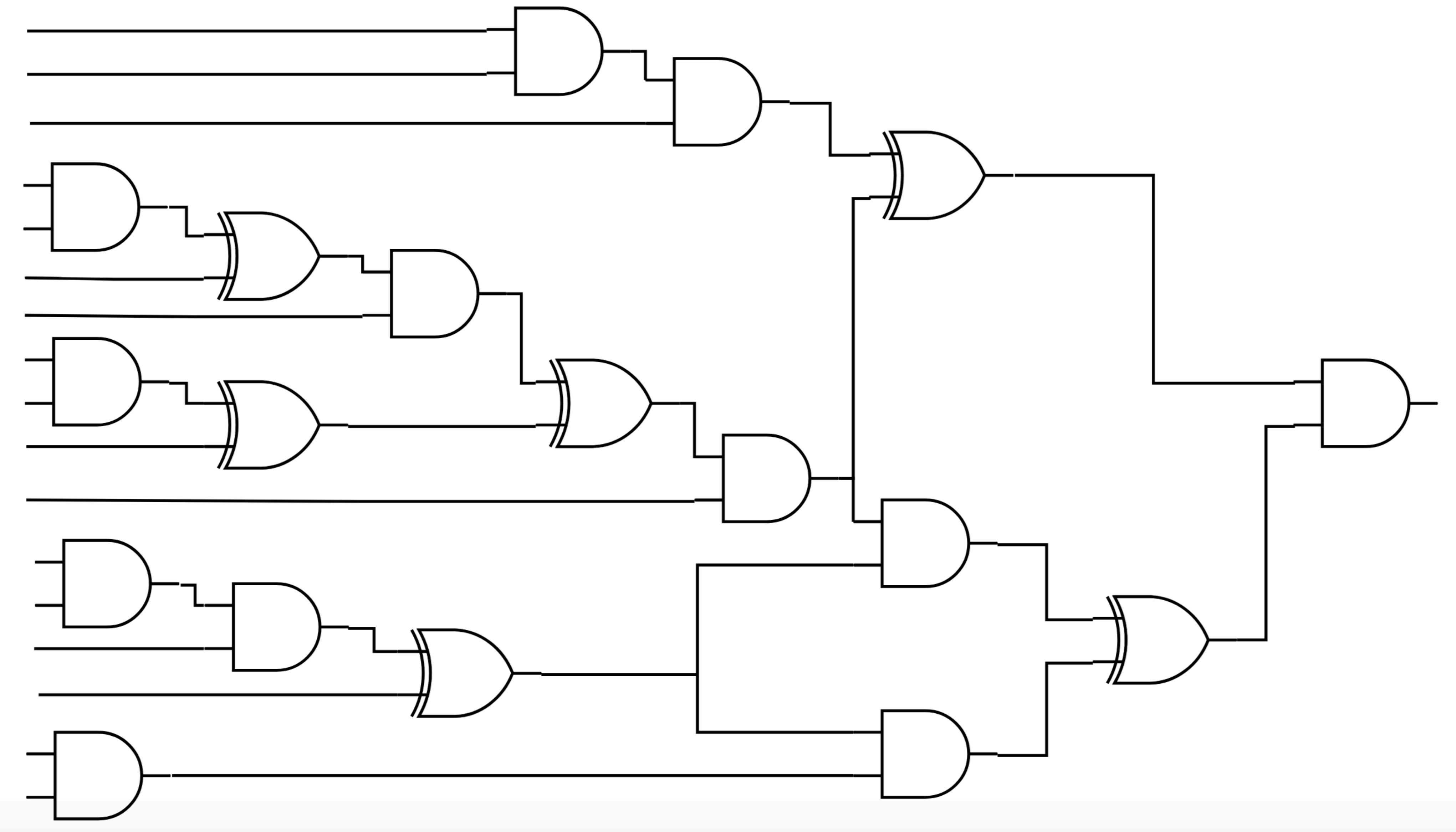
- Offline Learning
  - Collect successful synthesis patterns

- Online Optimization
  - Applying the patterns by term rewriting

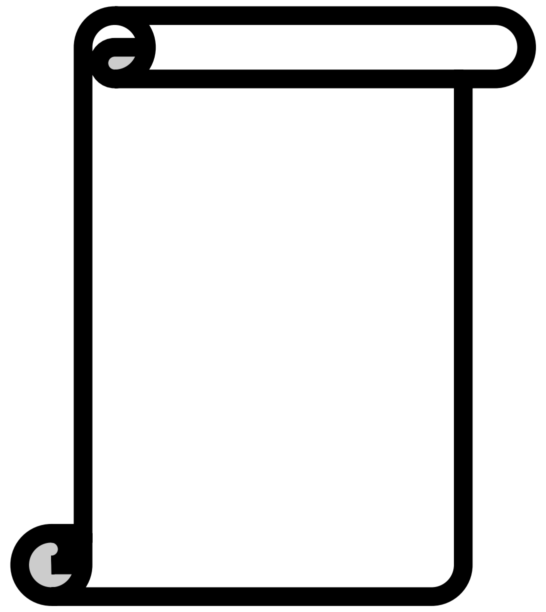
Offline Learning to Collect Opt. Patterns

Training
HE Applications



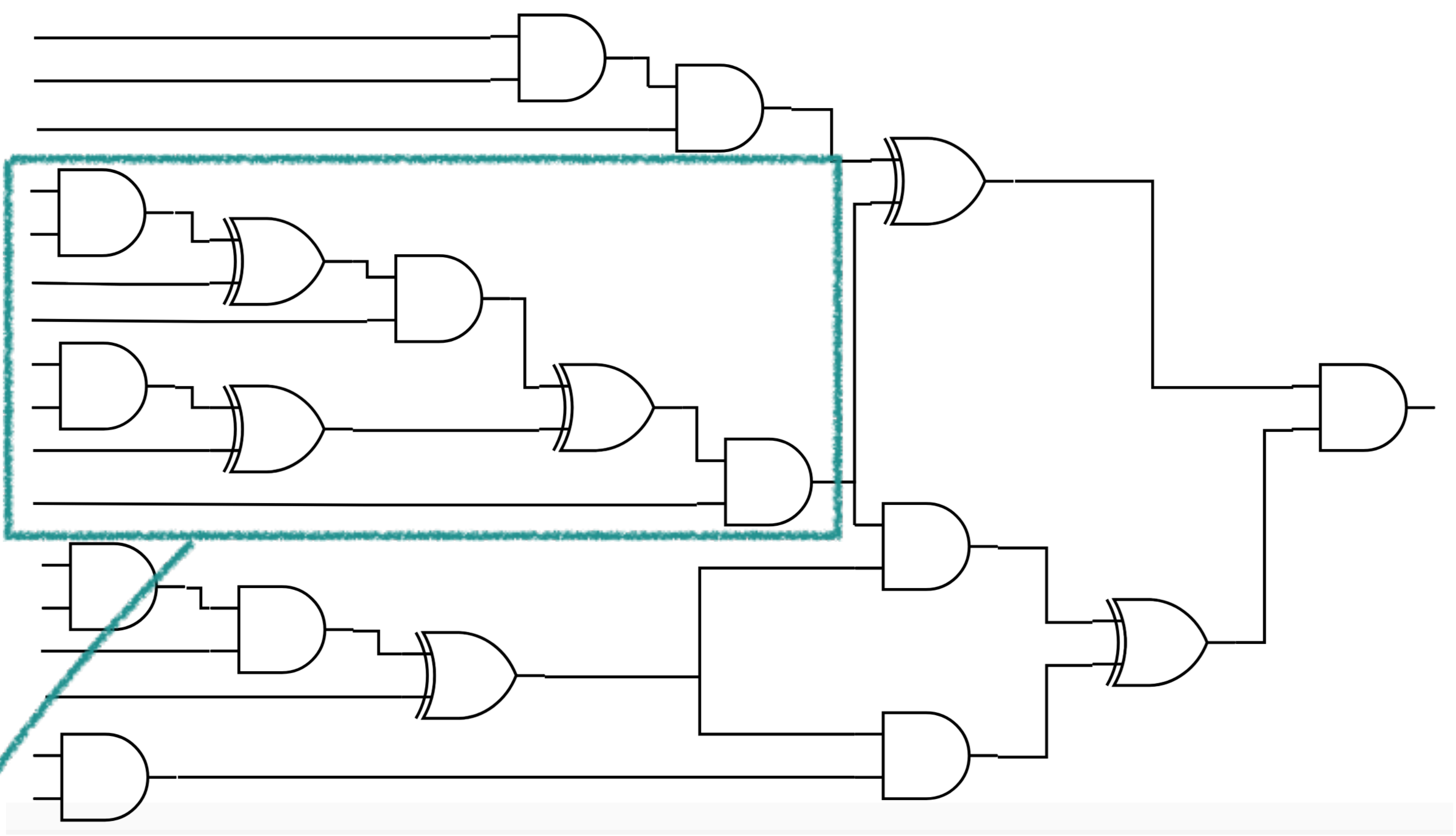


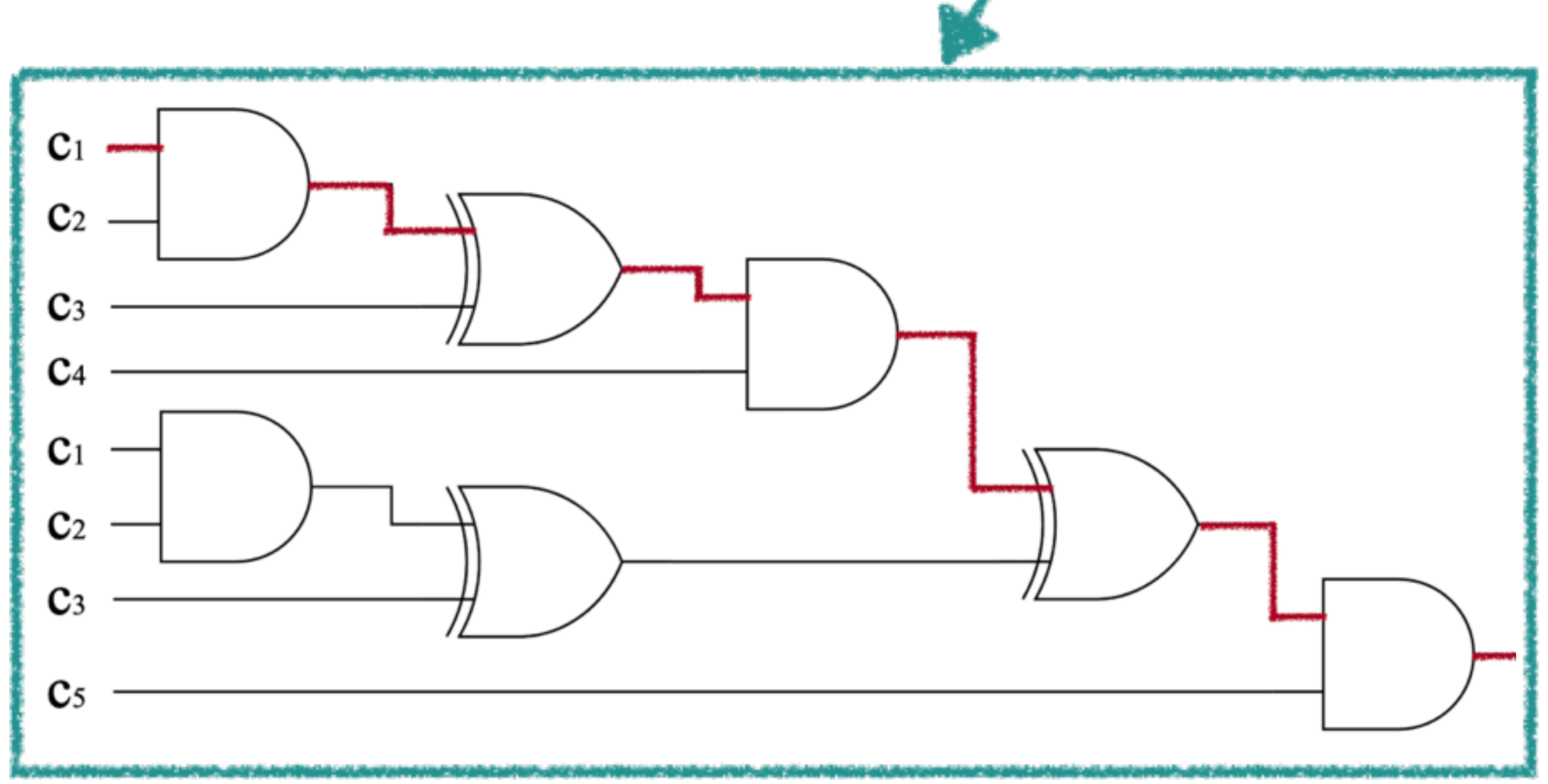
Collected
Opt. Patterns



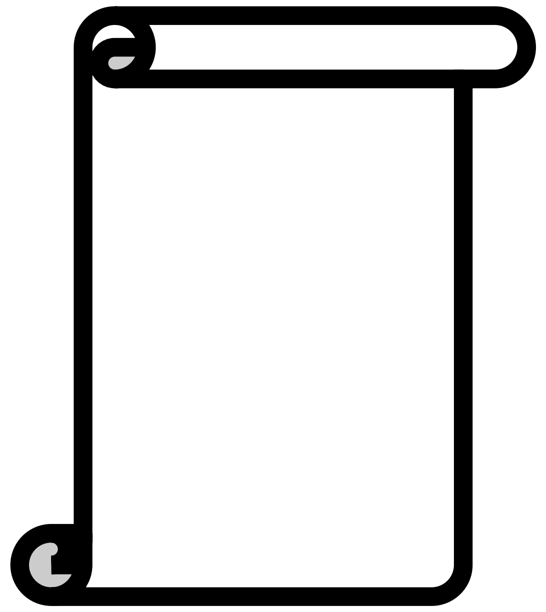
Offline Learning to Collect Opt. Patterns

Training
HE Applications

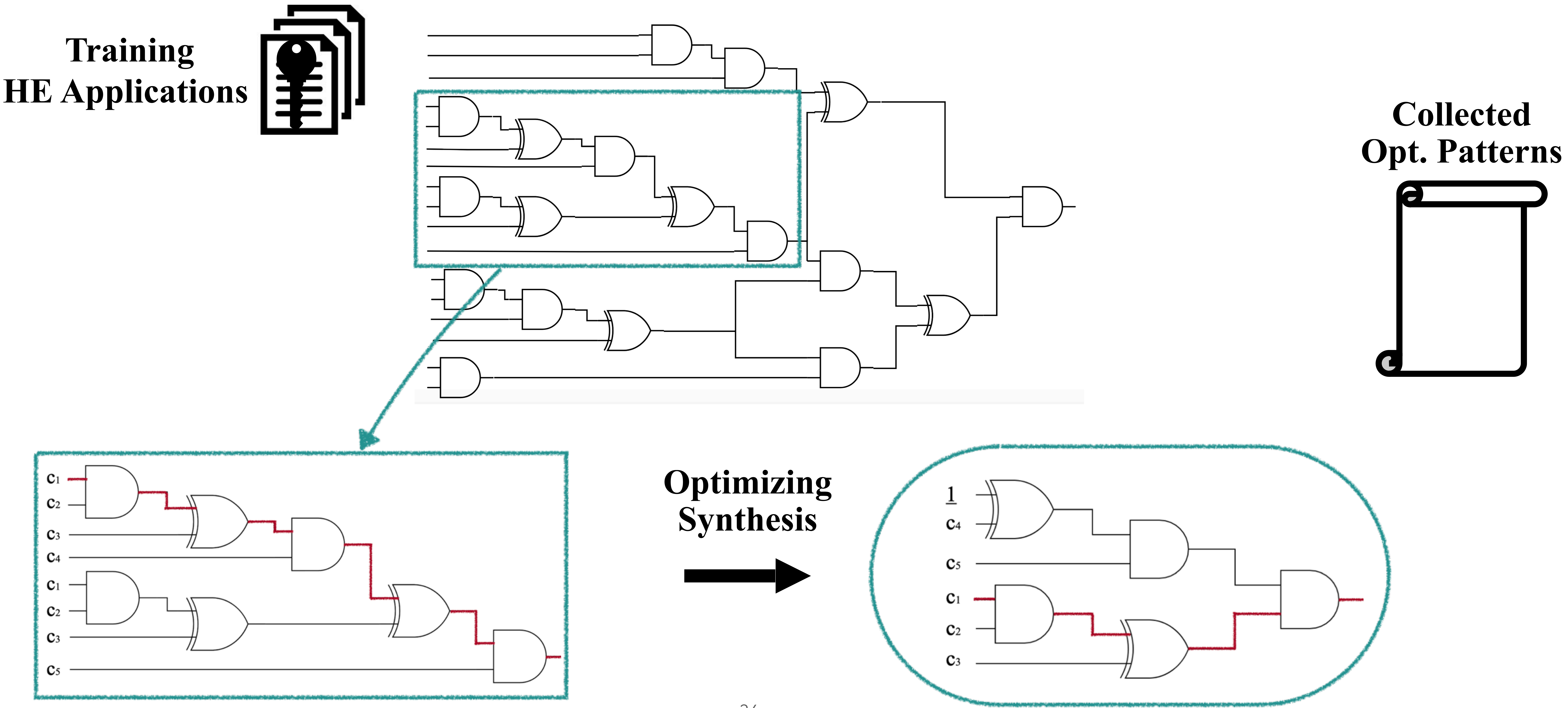




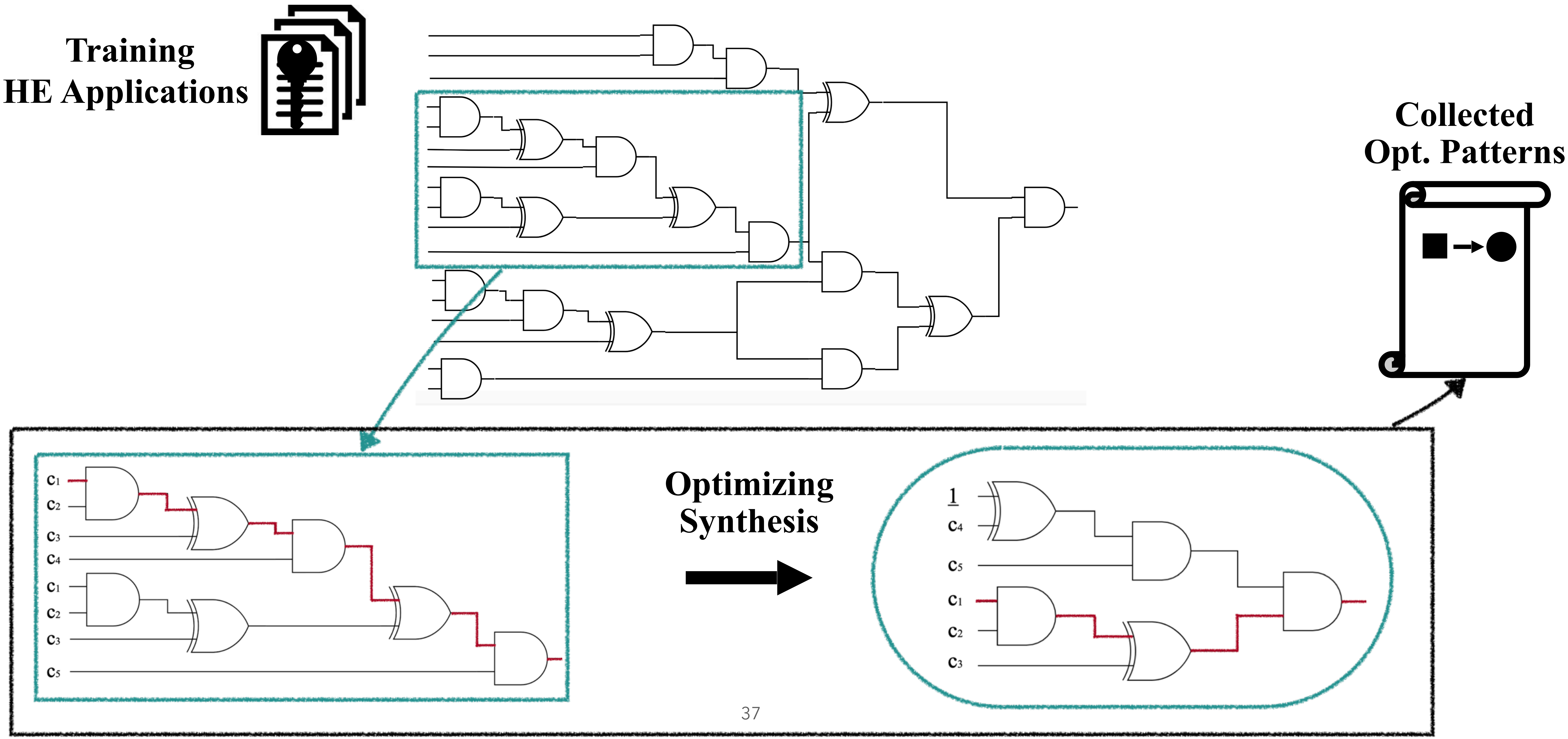
Collected
Opt. Patterns



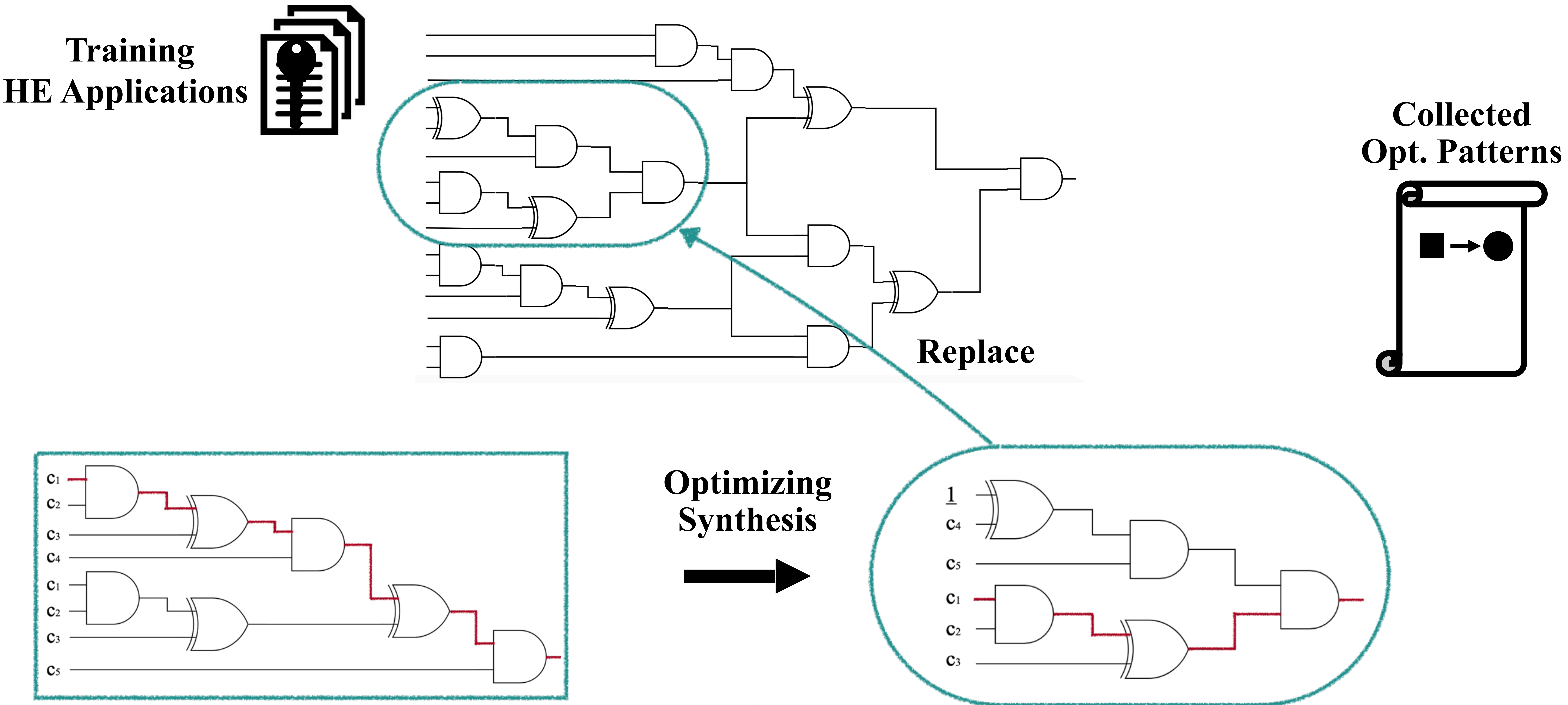
Offline Learning to Collect Opt. Patterns



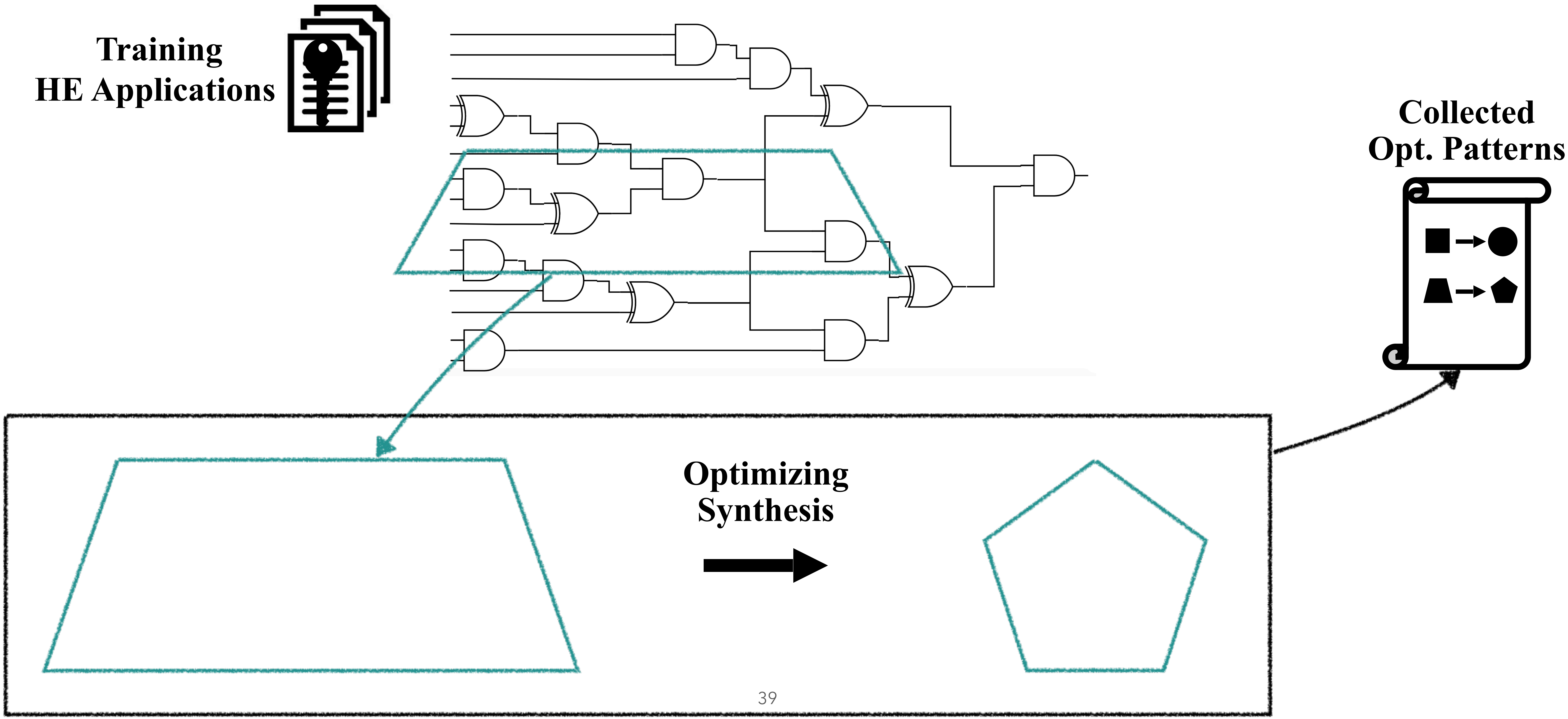
Offline Learning to Collect Opt. Patterns



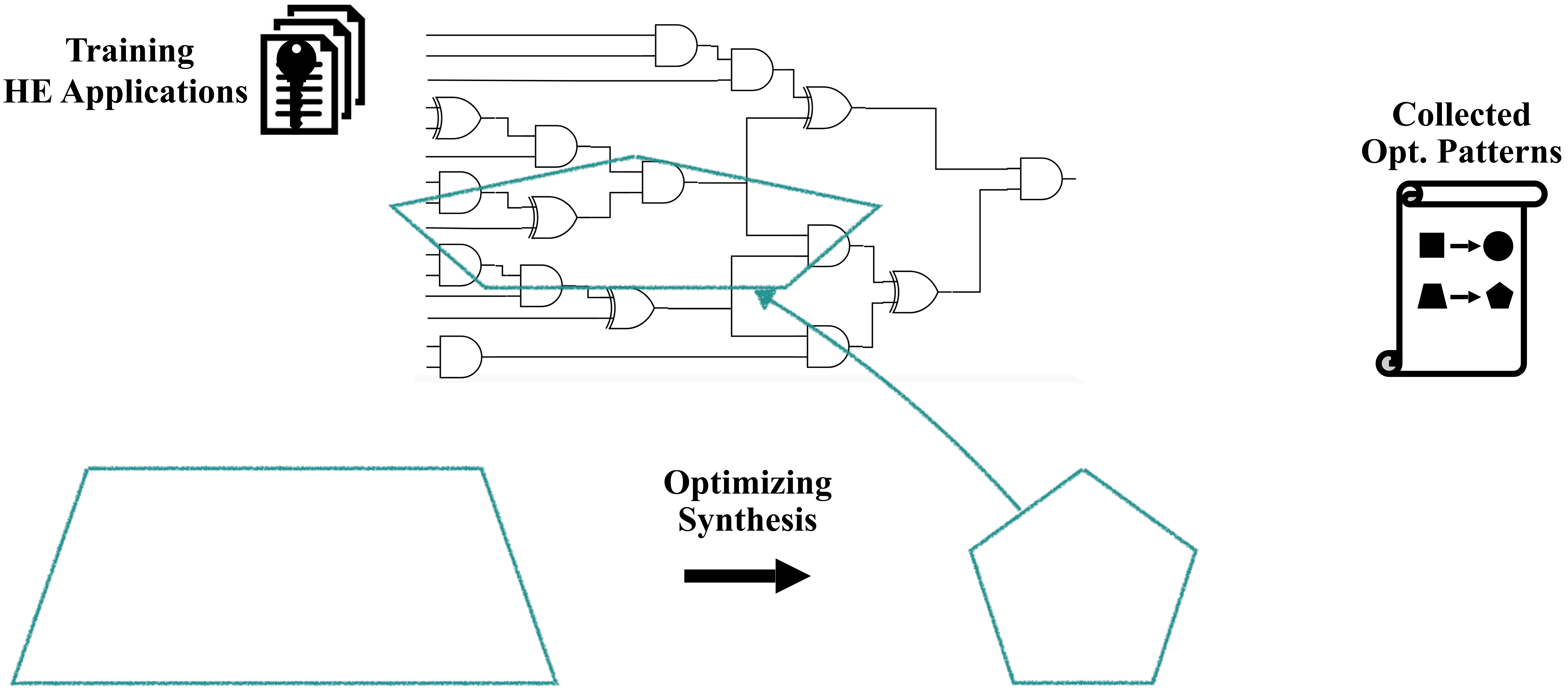
Offline Learning to Collect Opt. Patterns



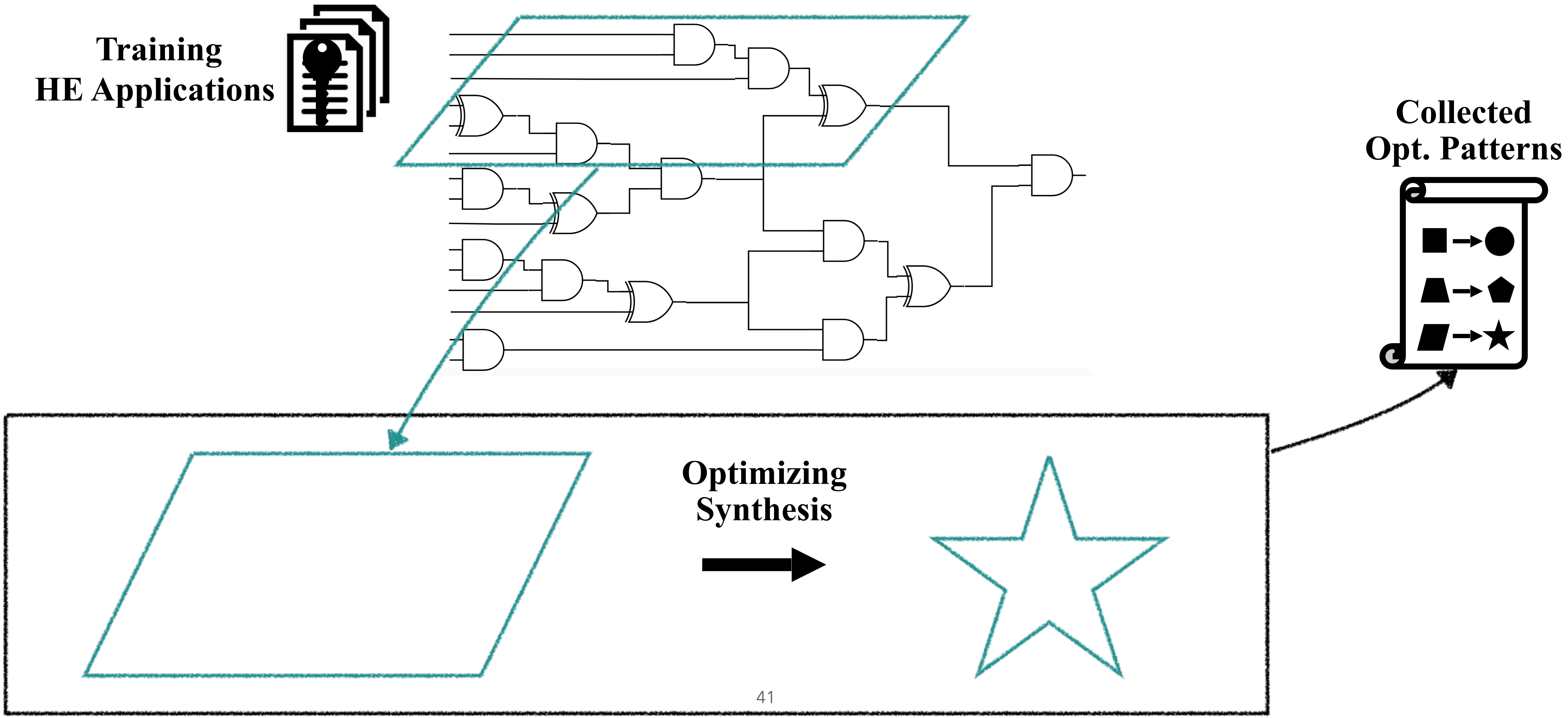
Offline Learning to Collect Opt. Patterns



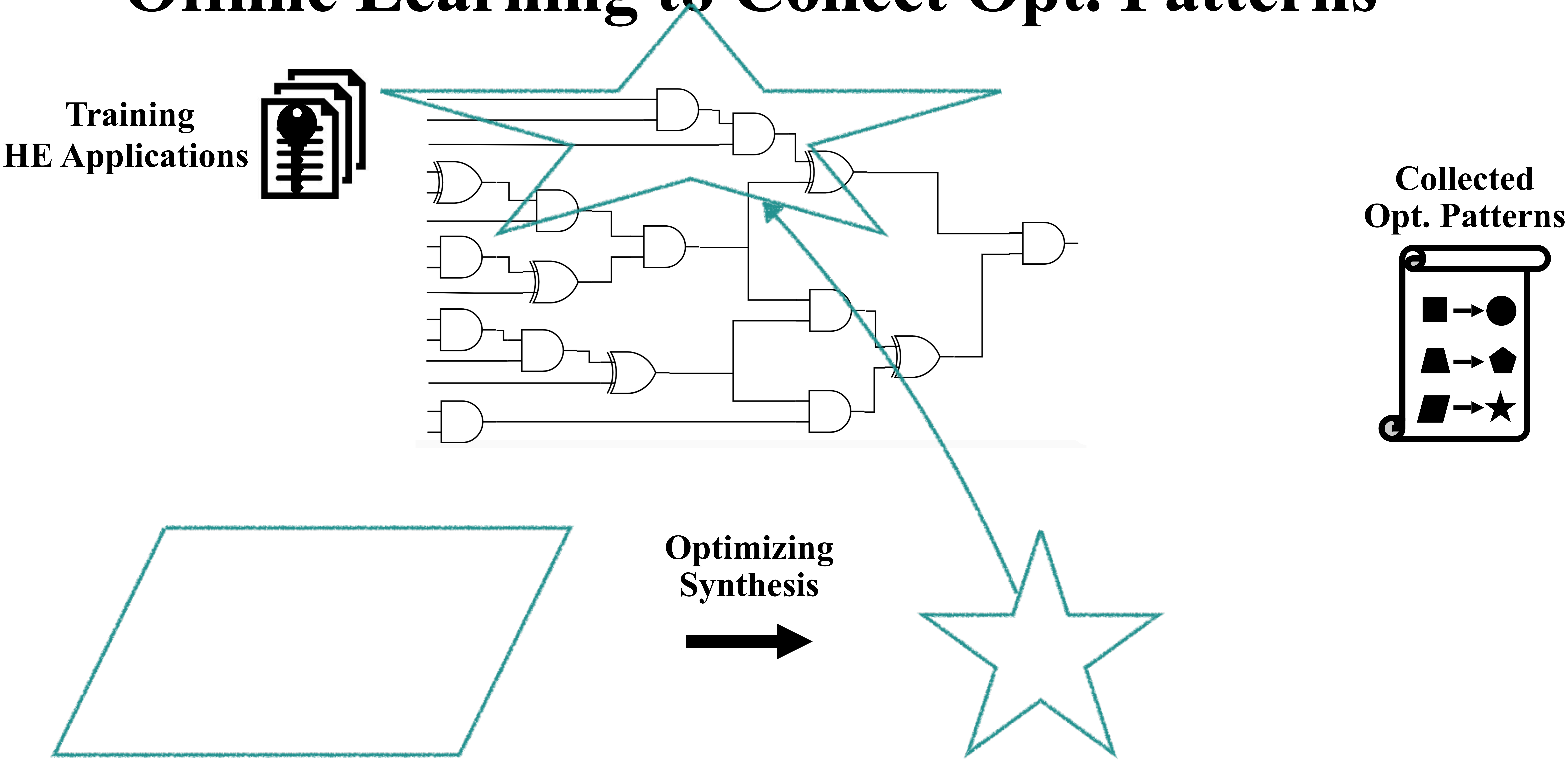
Offline Learning to Collect Opt. Patterns



Offline Learning to Collect Opt. Patterns

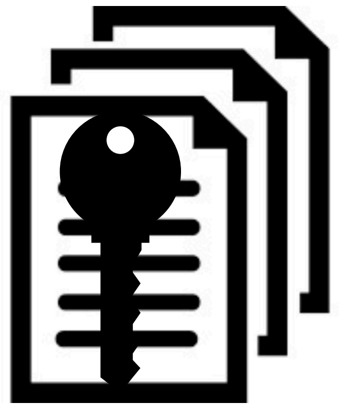


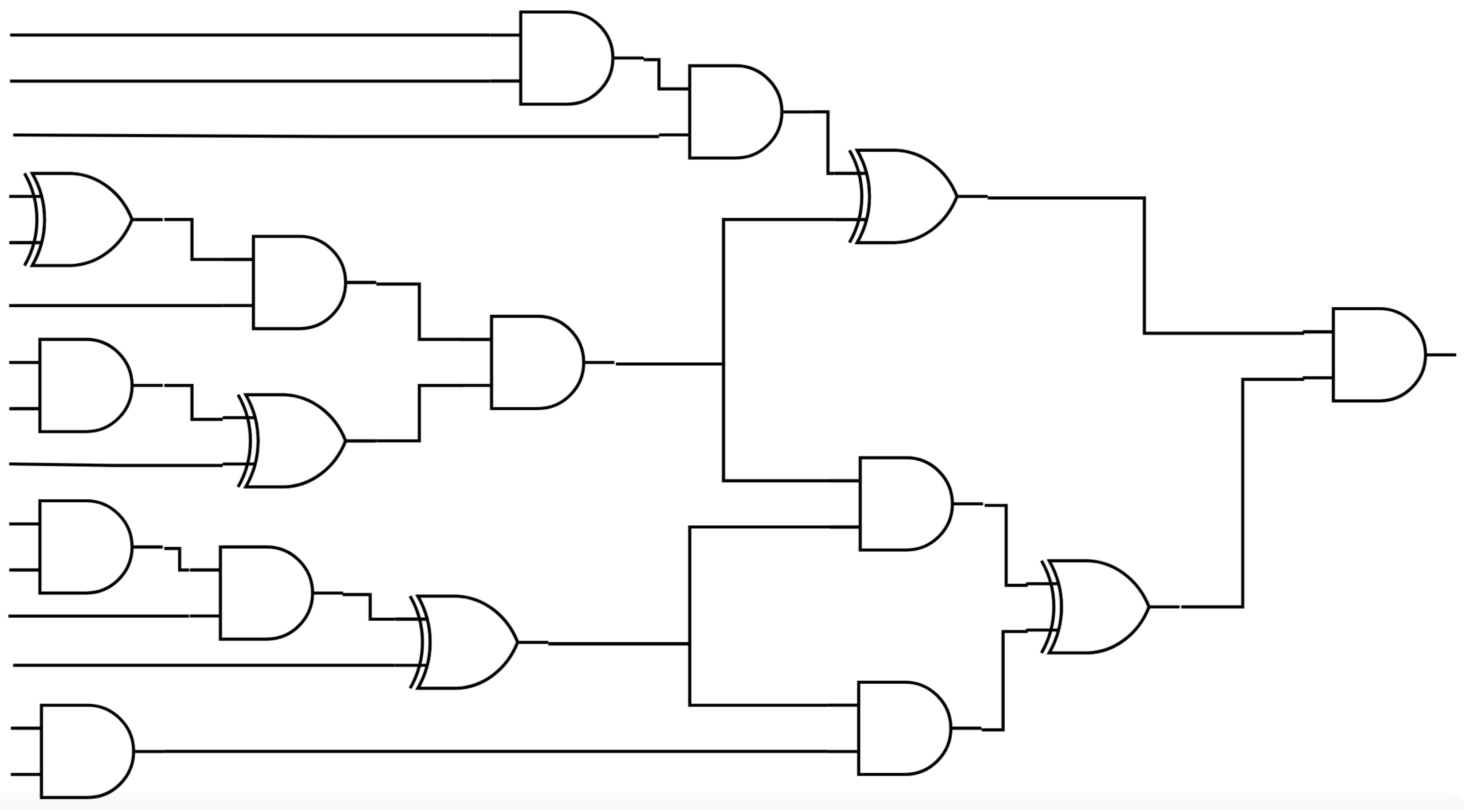
Offline Learning to Collect Opt. Patterns



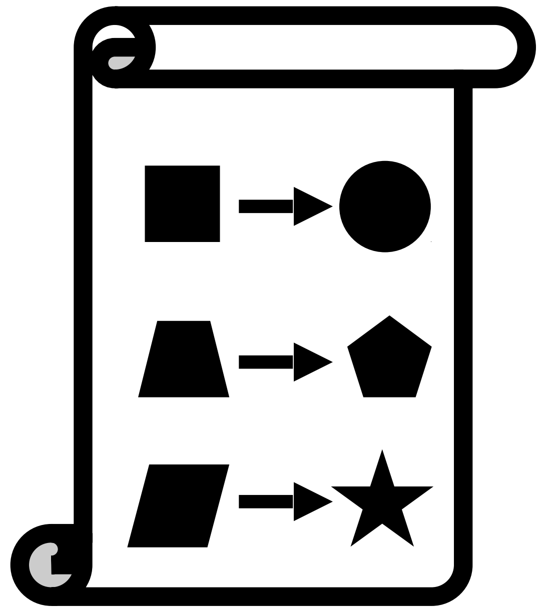
Offline Learning to Collect Opt. Patterns

Training
HE Applications



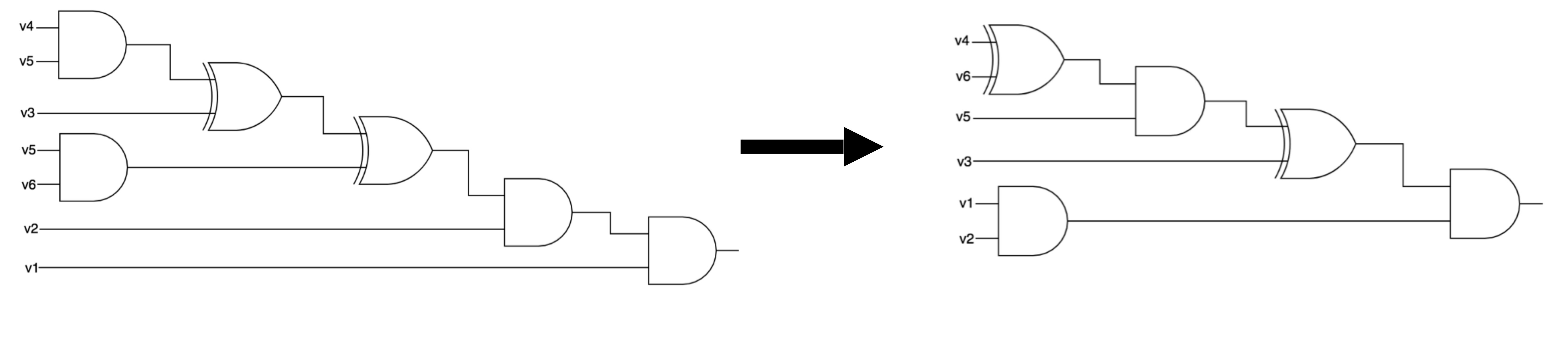


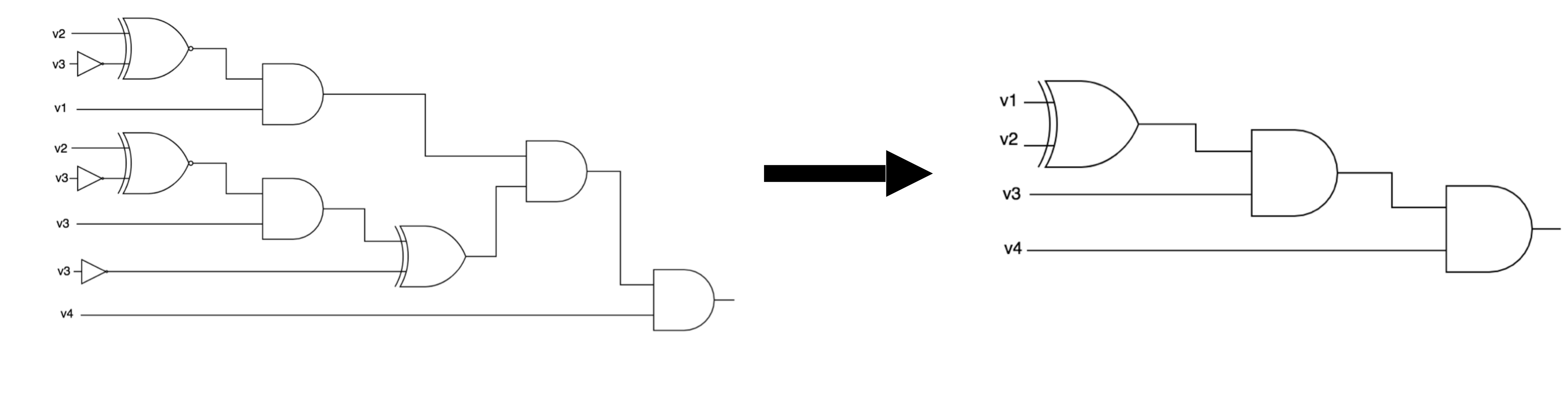
Collected
Opt. Patterns

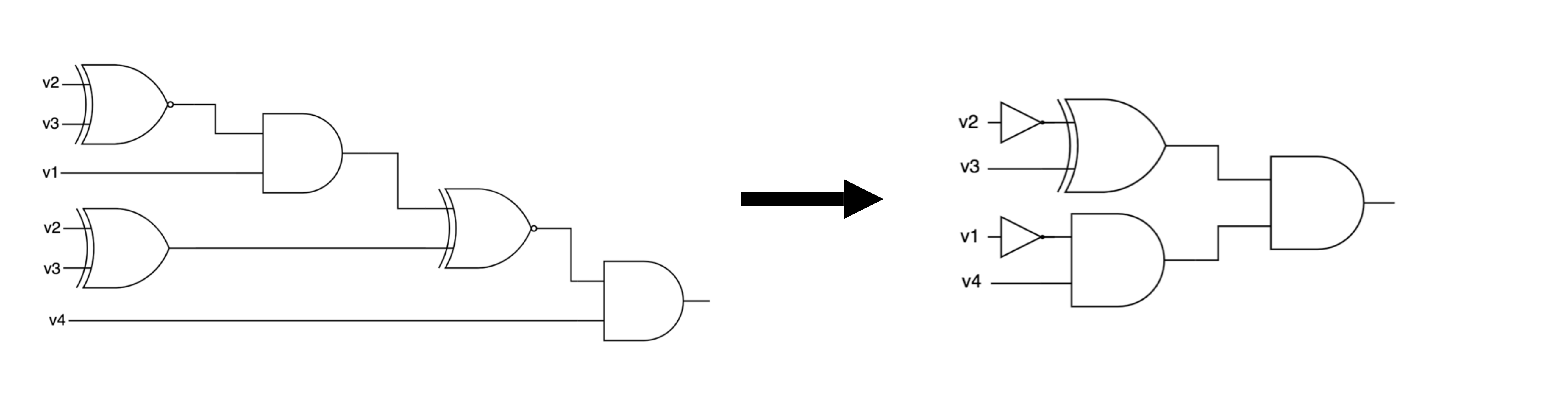


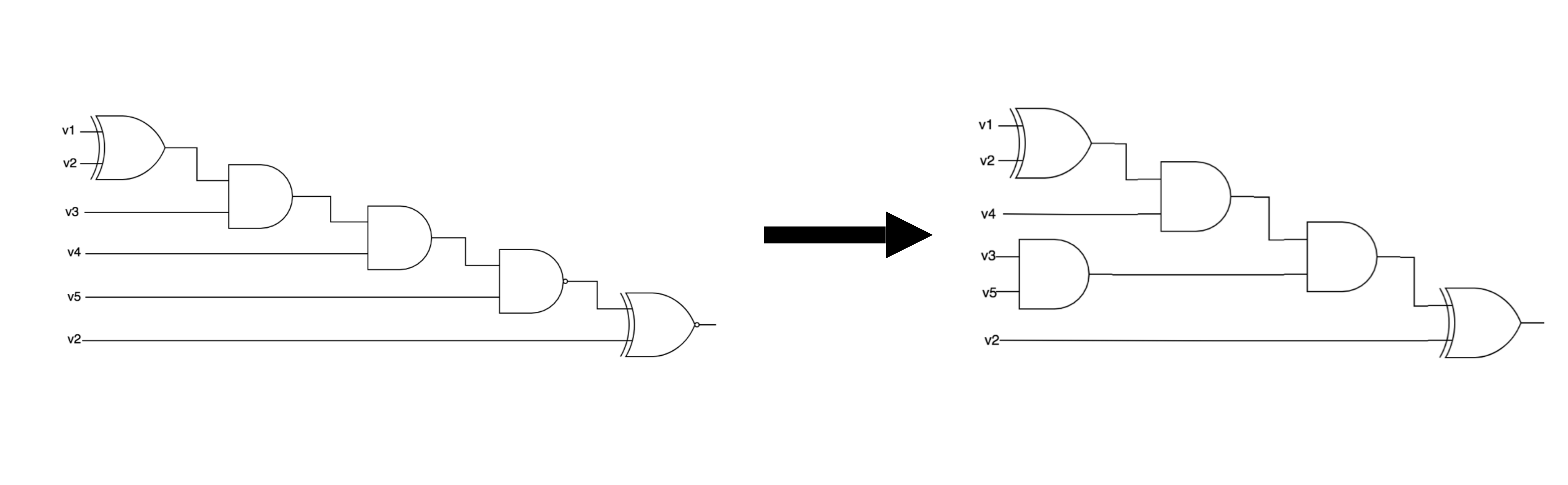
400 Opt.
patterns

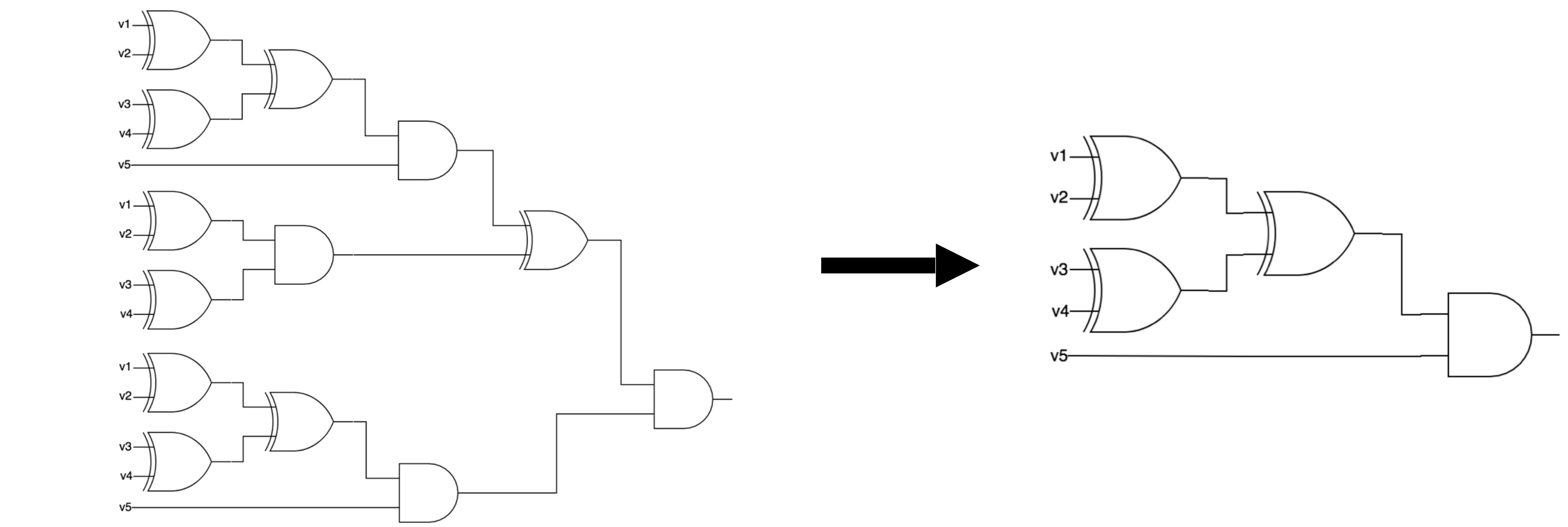
Learned Optimization Patterns : examples

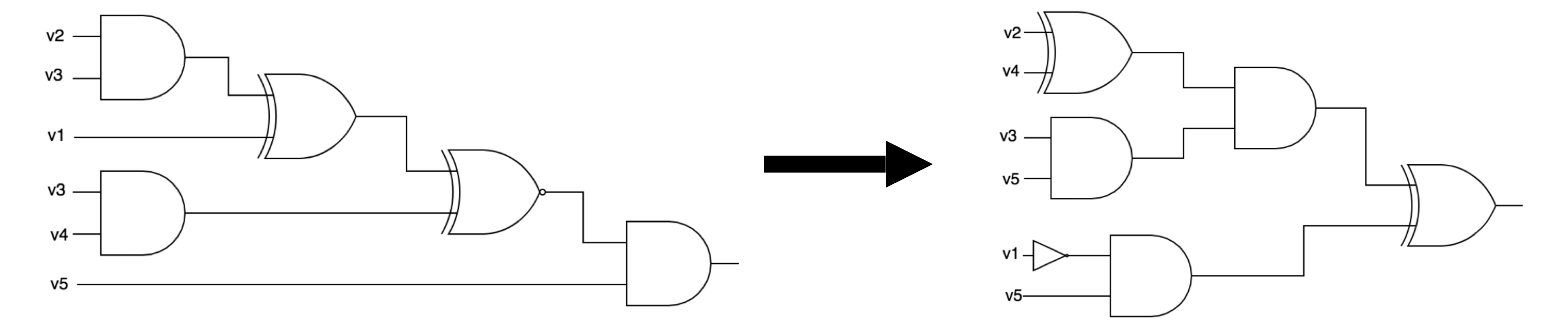






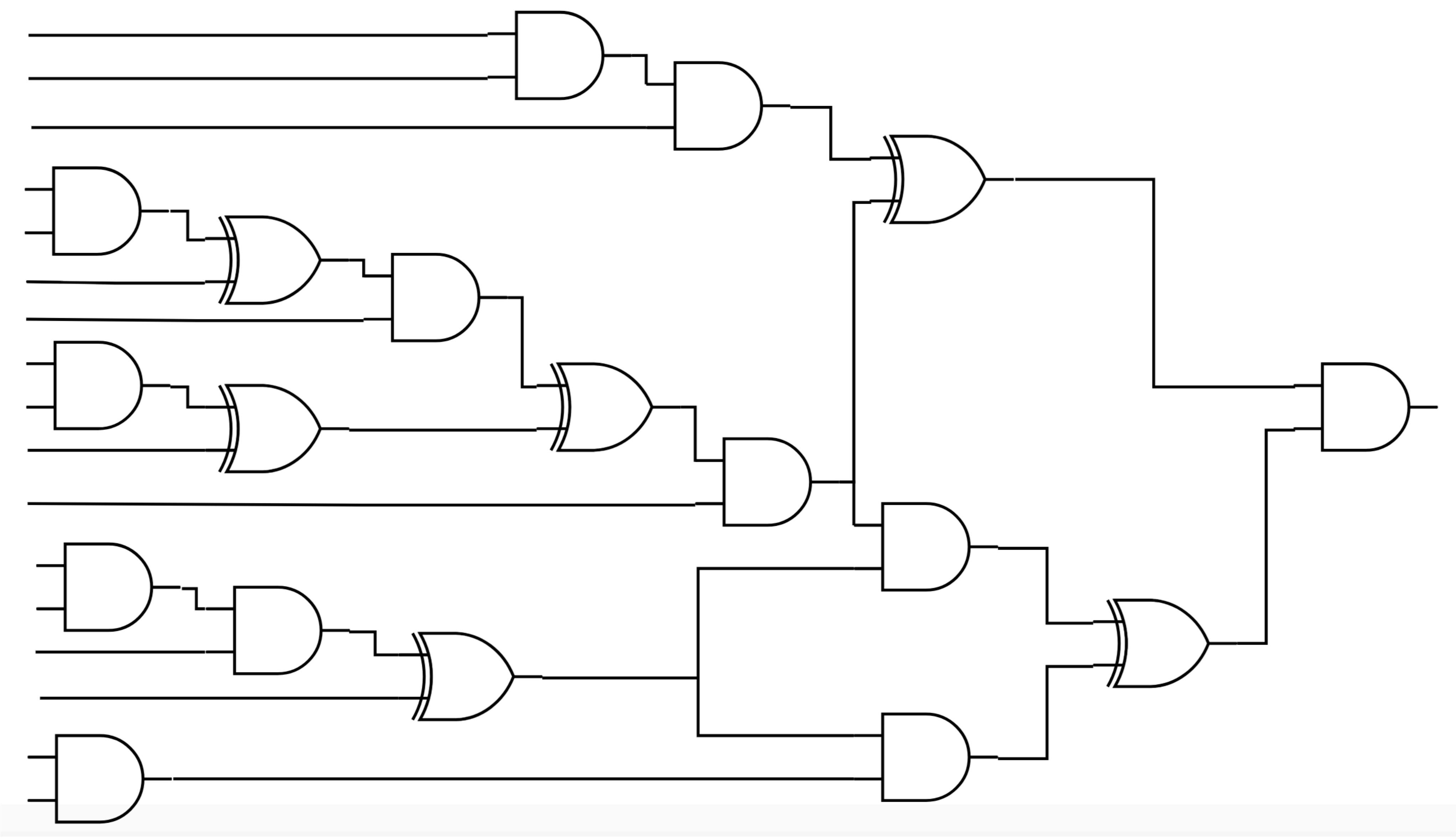




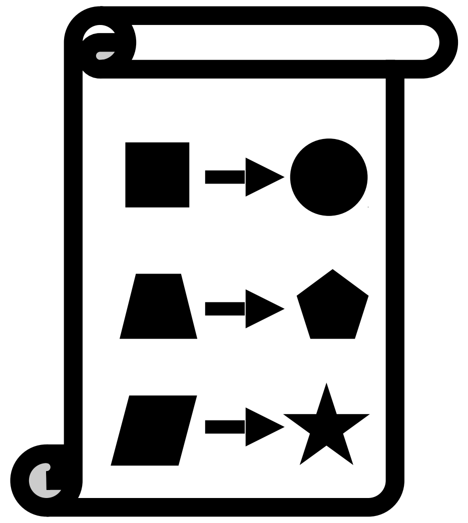


Online Rule-based Optimization

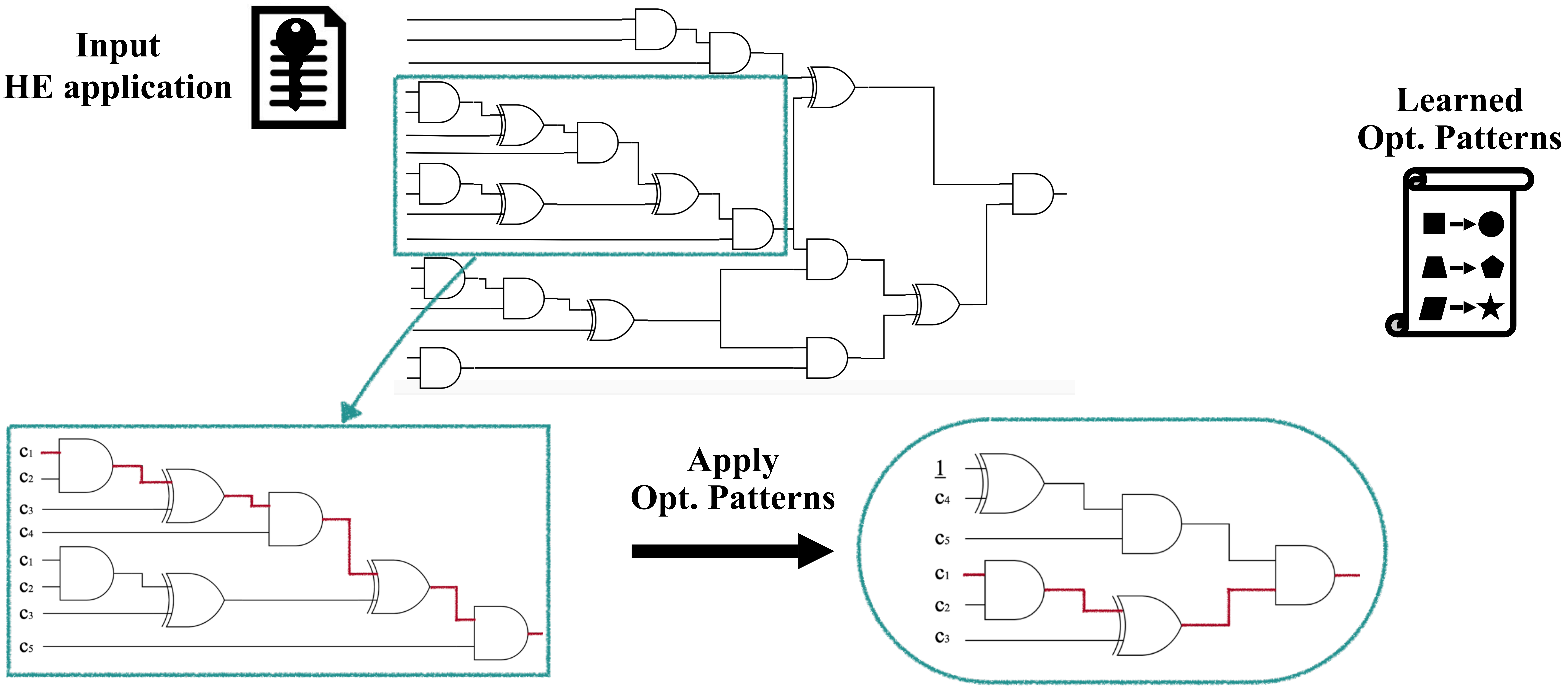
Input
HE application



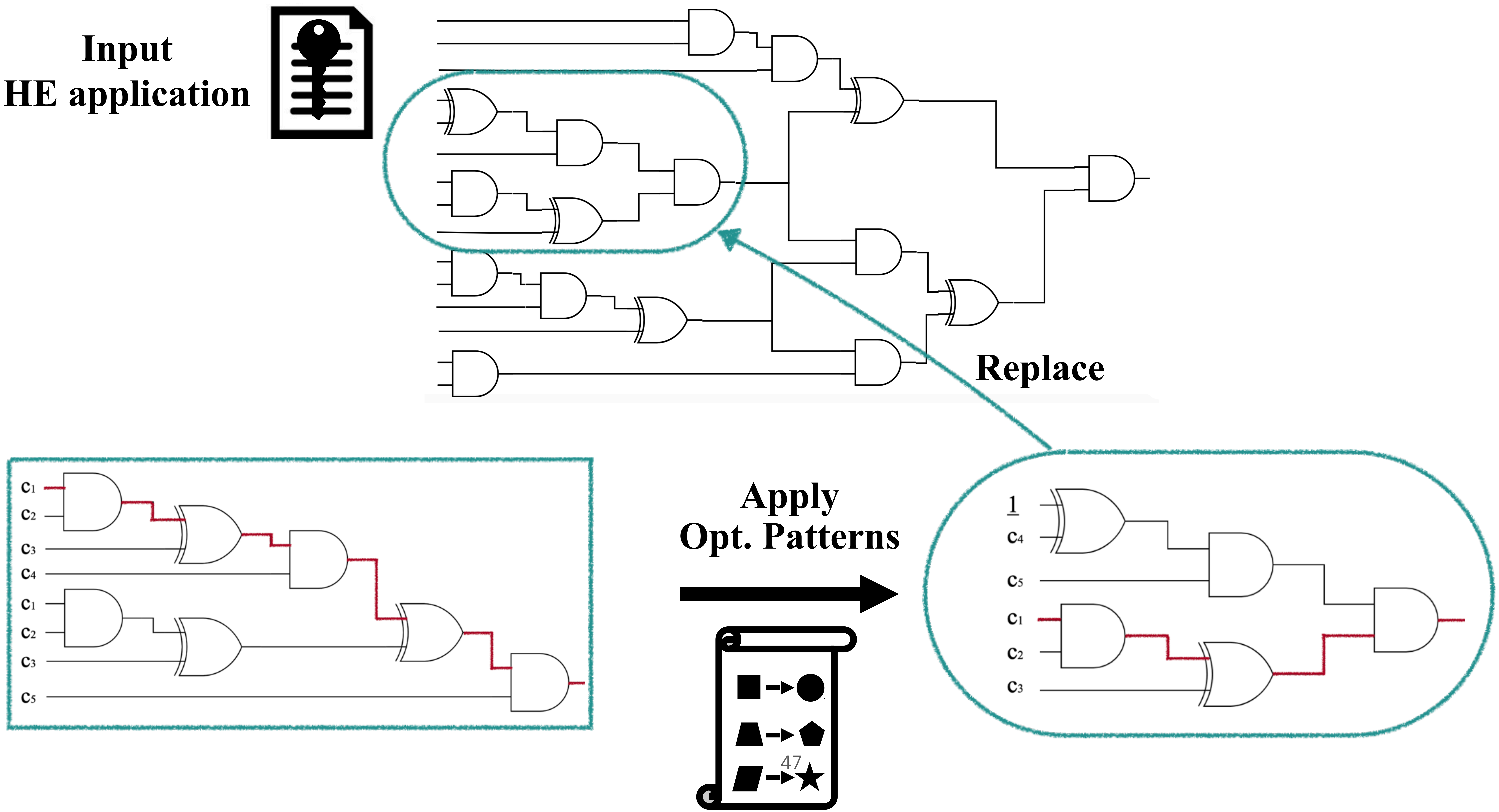
Learned
Opt. Patterns



Online Rule-based Optimization

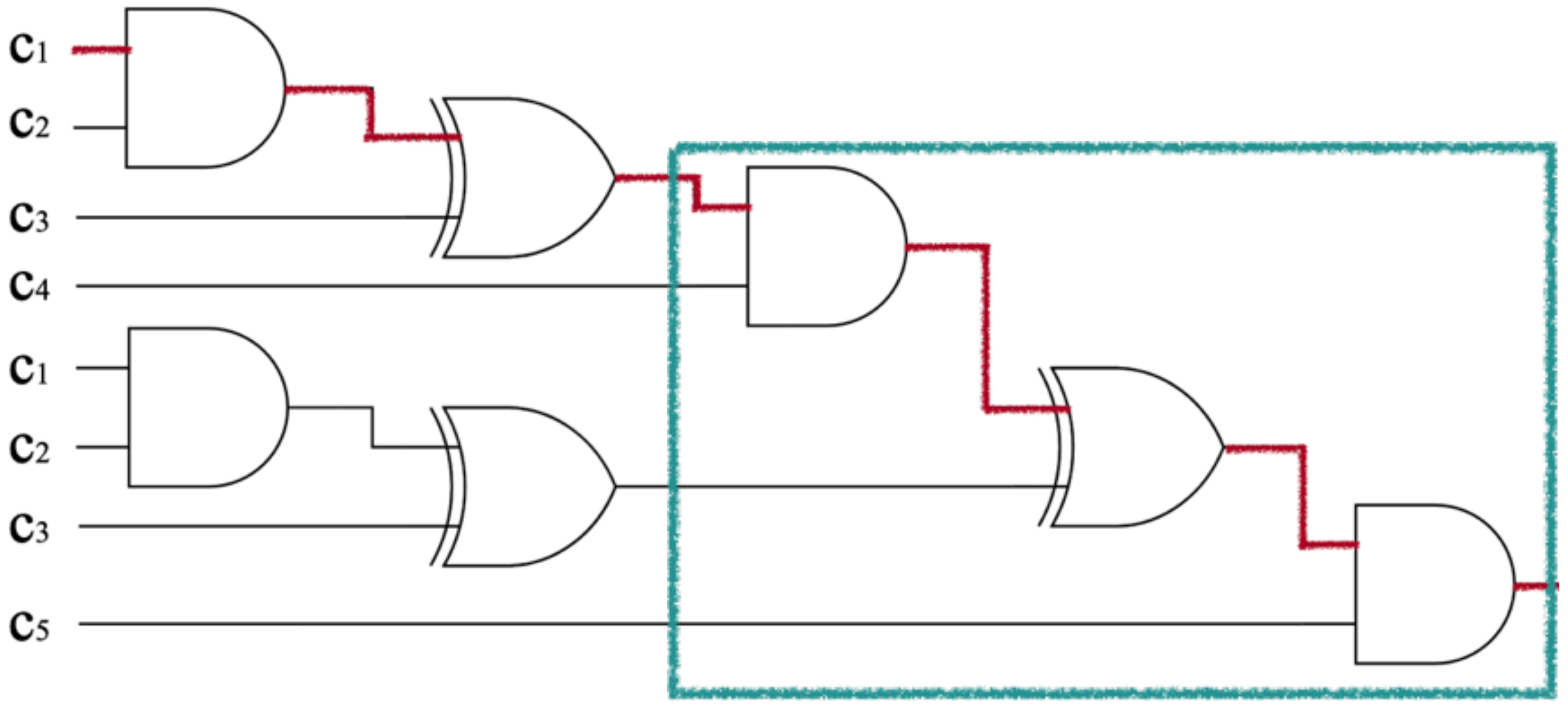


Online Rule-based Optimization

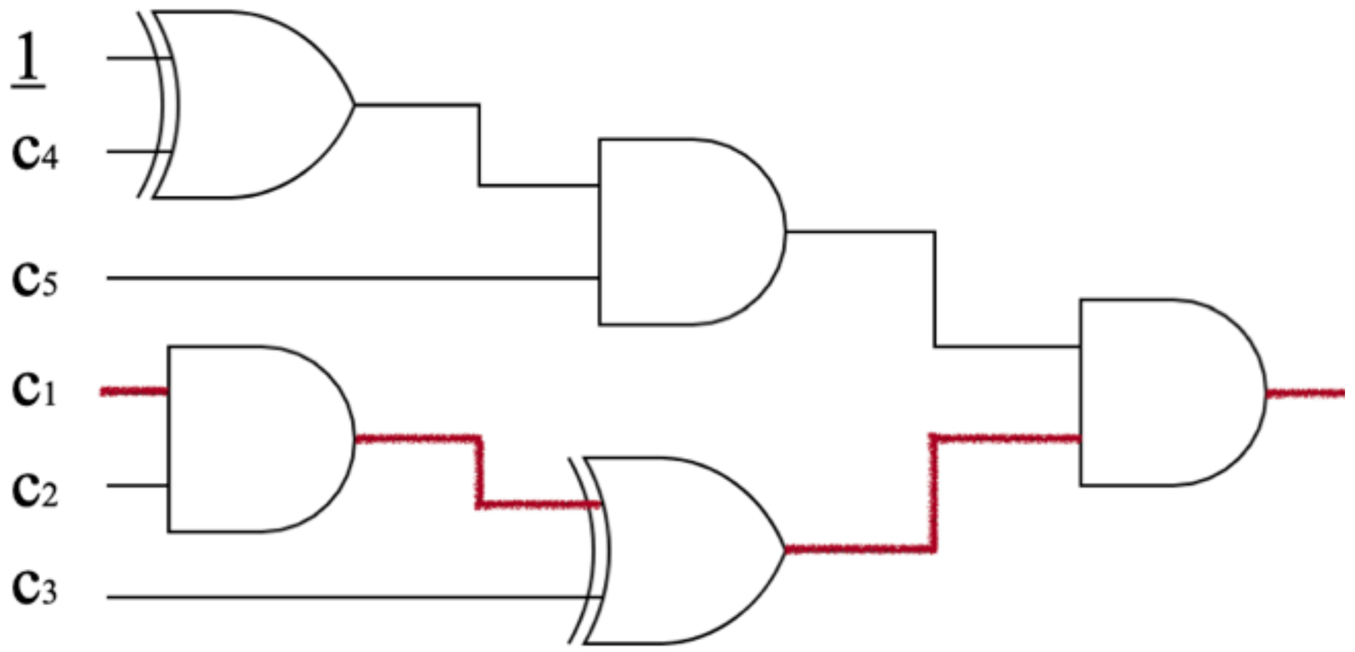


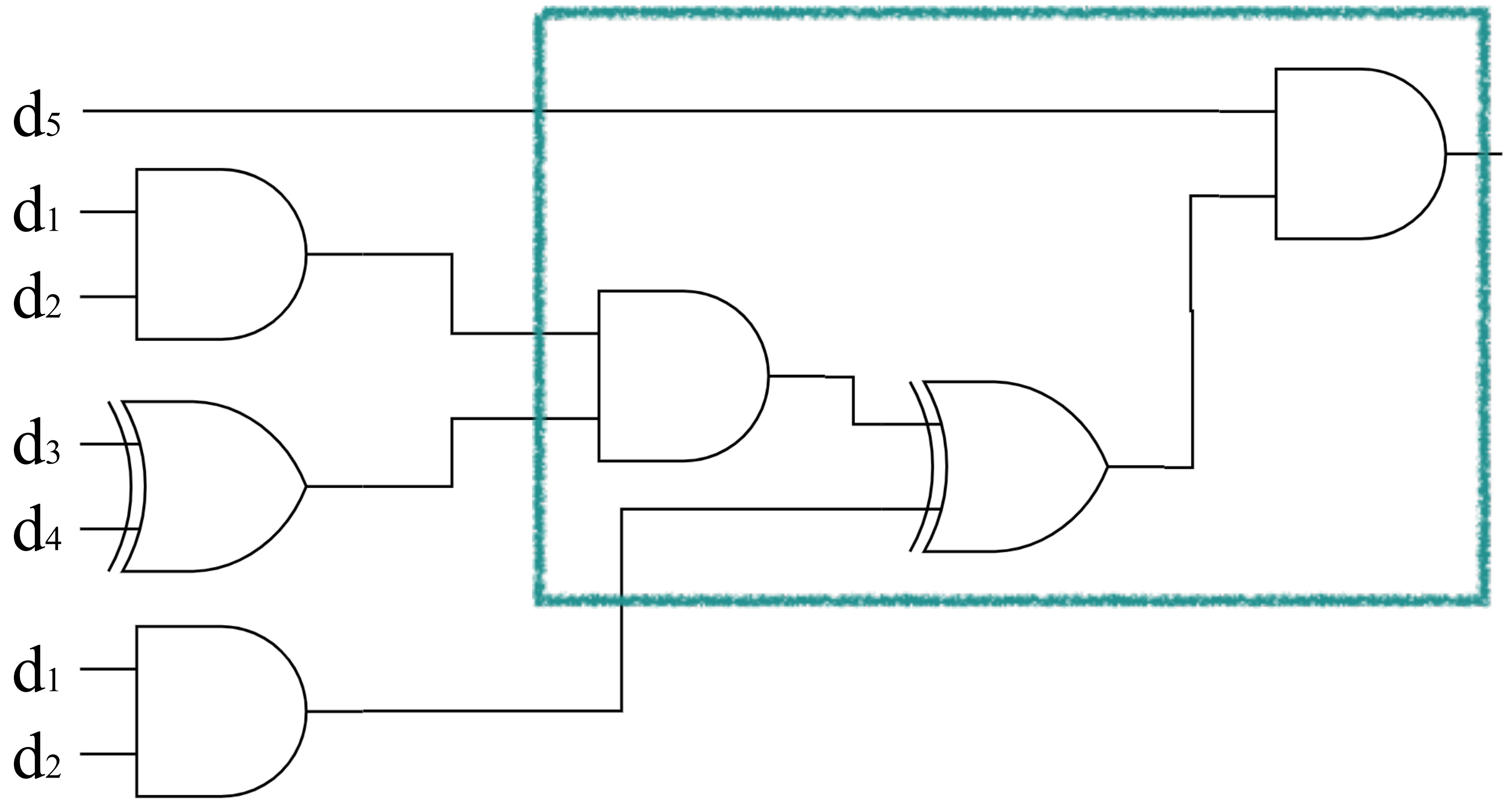
Applying Learned Optimization Patterns (1/2)

Syntactic Matching is Not Effective



Learned Opt. Patterns →



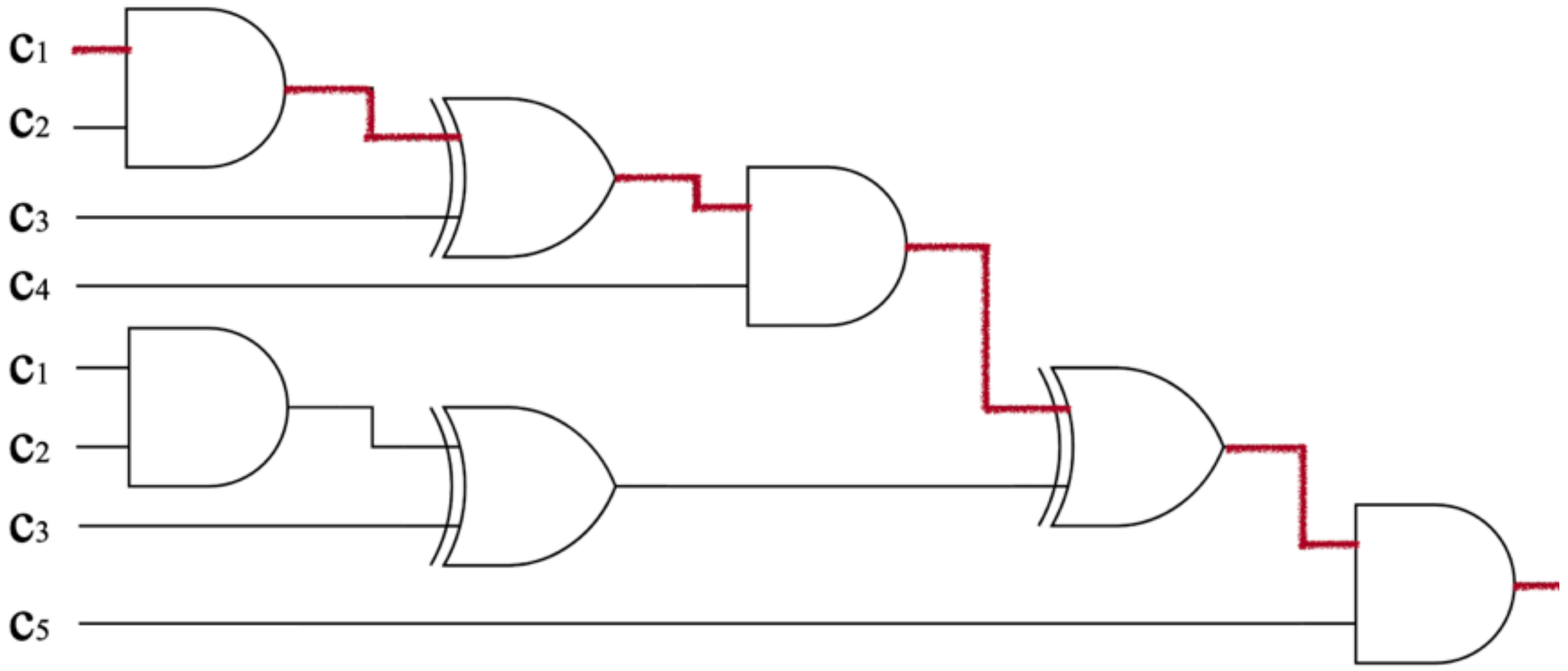


New Input Circuit Optimization →

?

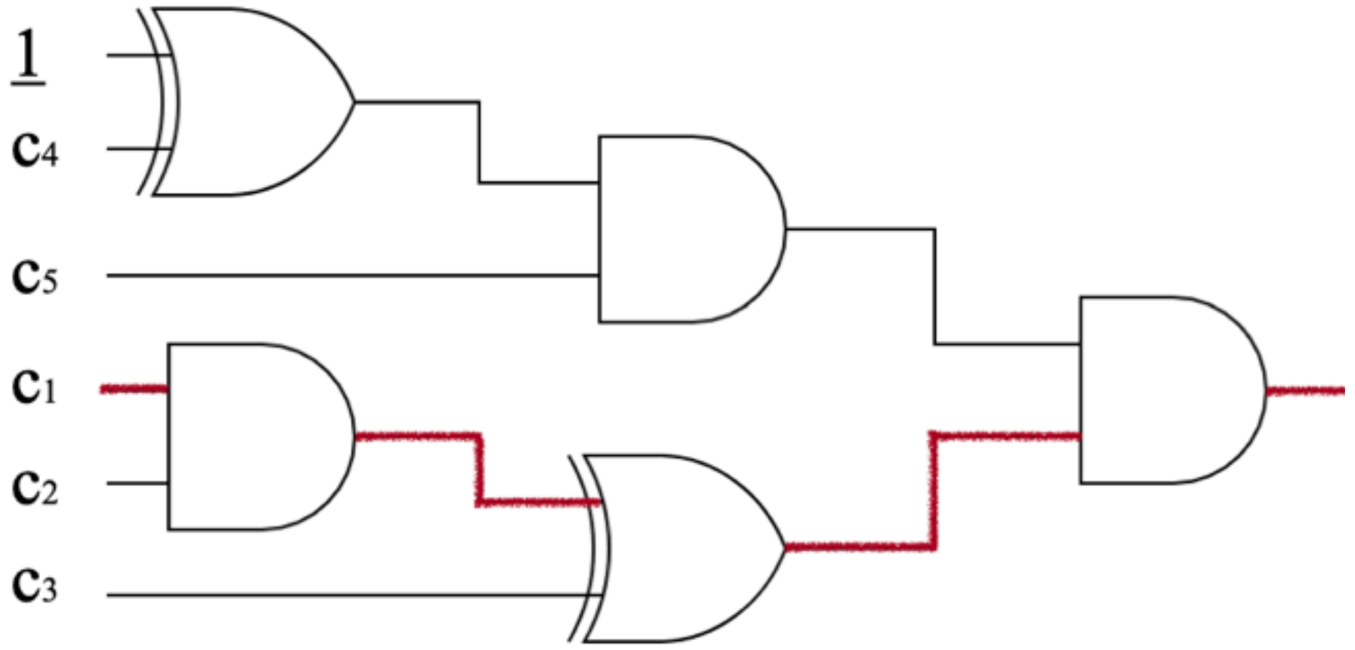
Applying Learned Optimization Patterns (1/2)

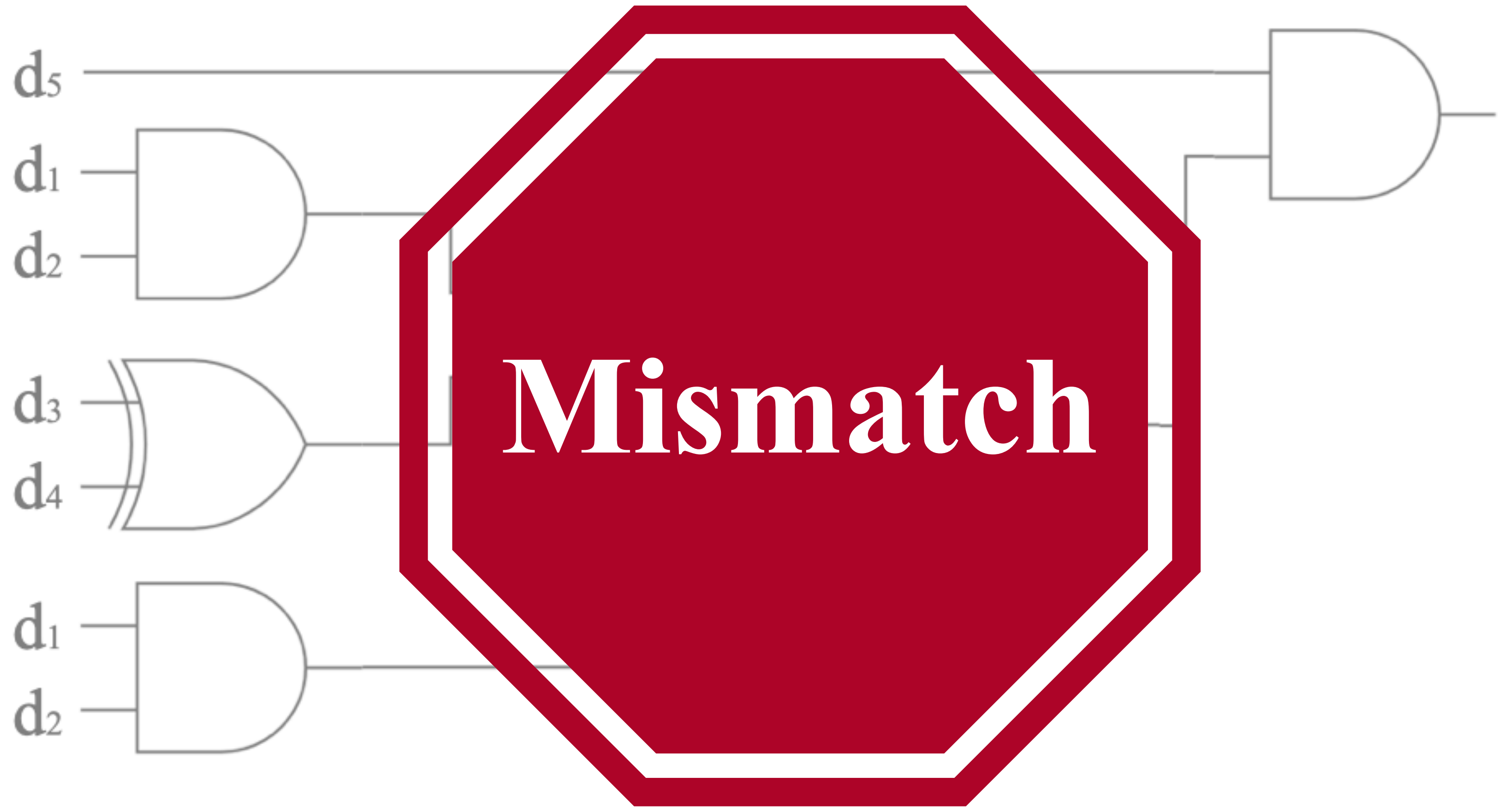
Syntactic Matching is Not Effective



Learned Opt. Patterns

→





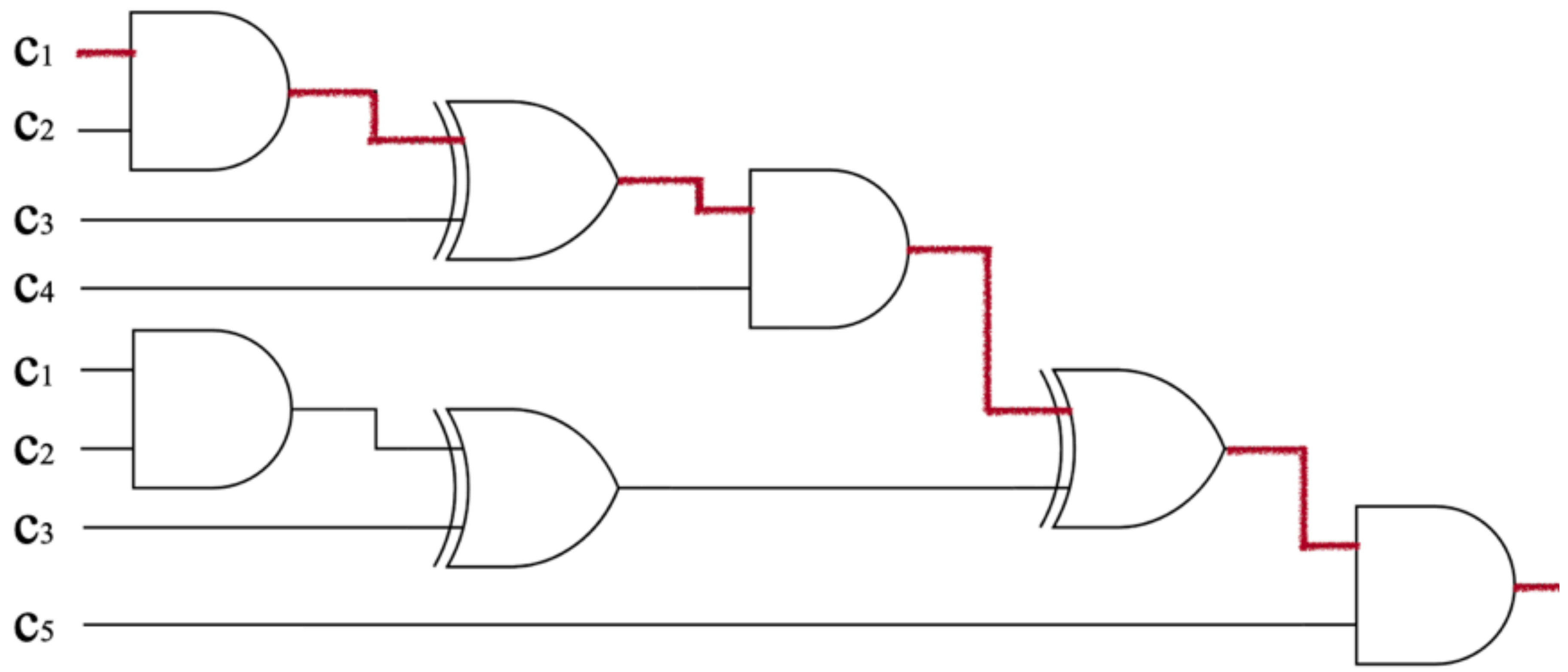
New Input Circuit Optimization

→

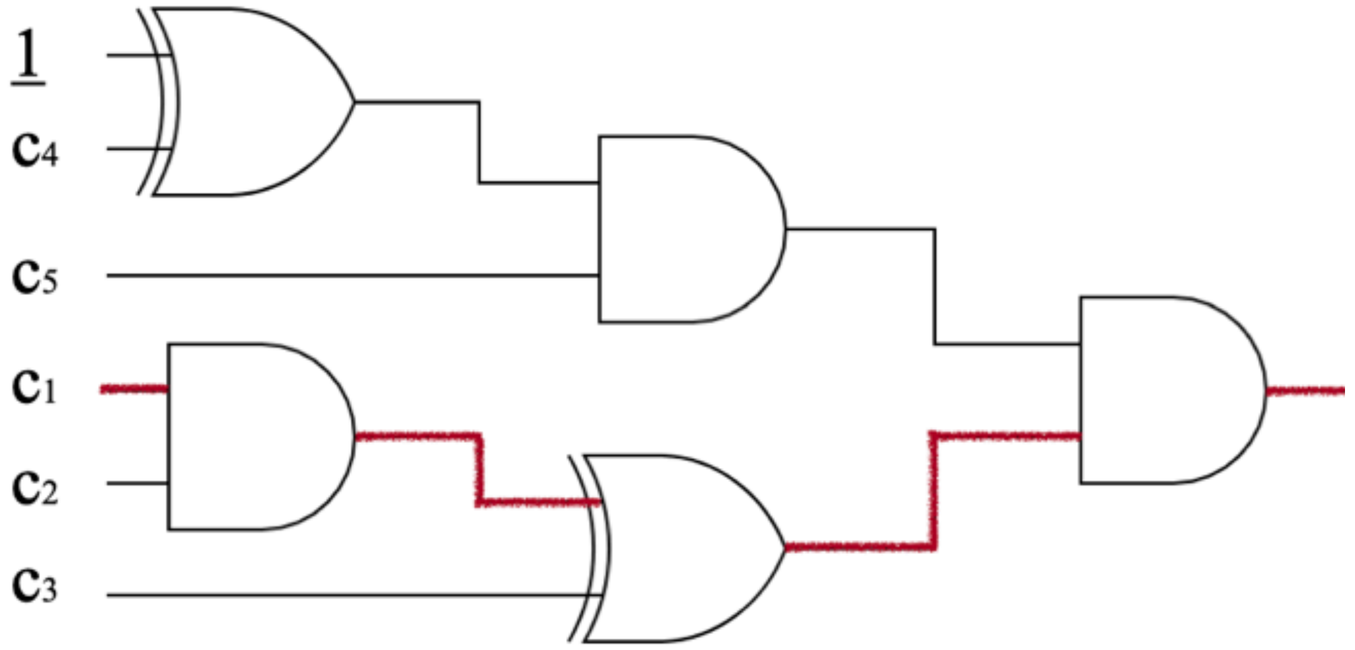
?

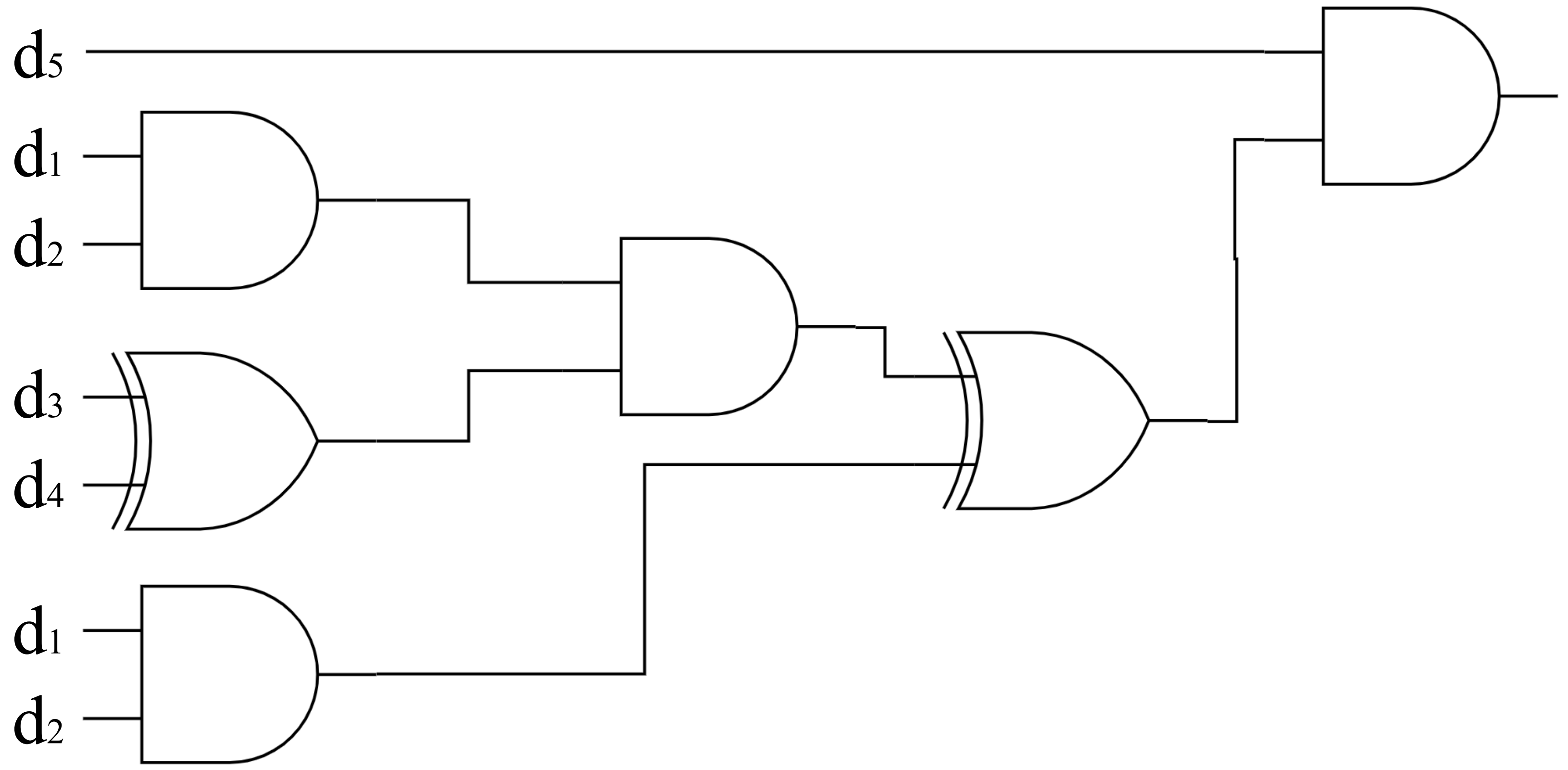
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



Learned Opt. Patterns



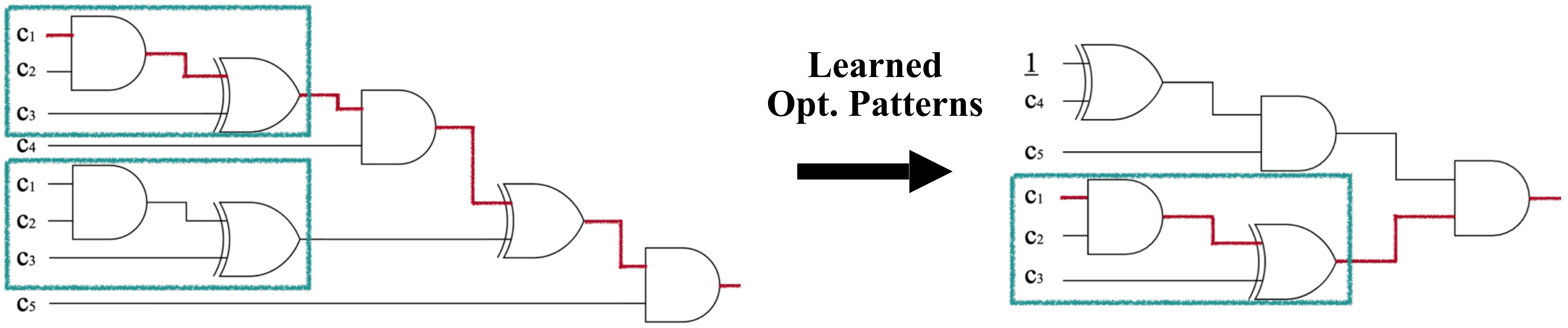


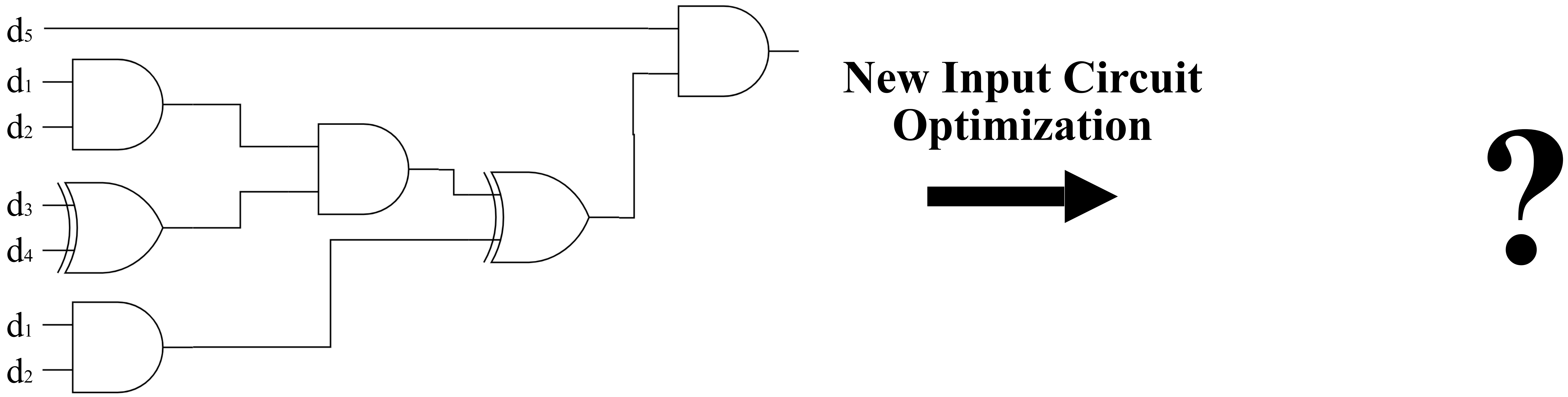
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

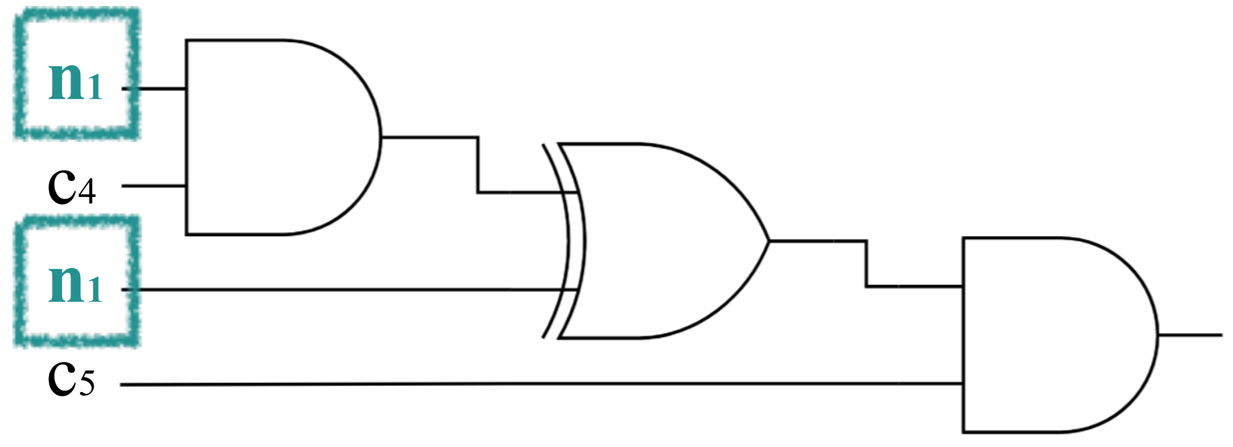
Normalization + Equational Matching



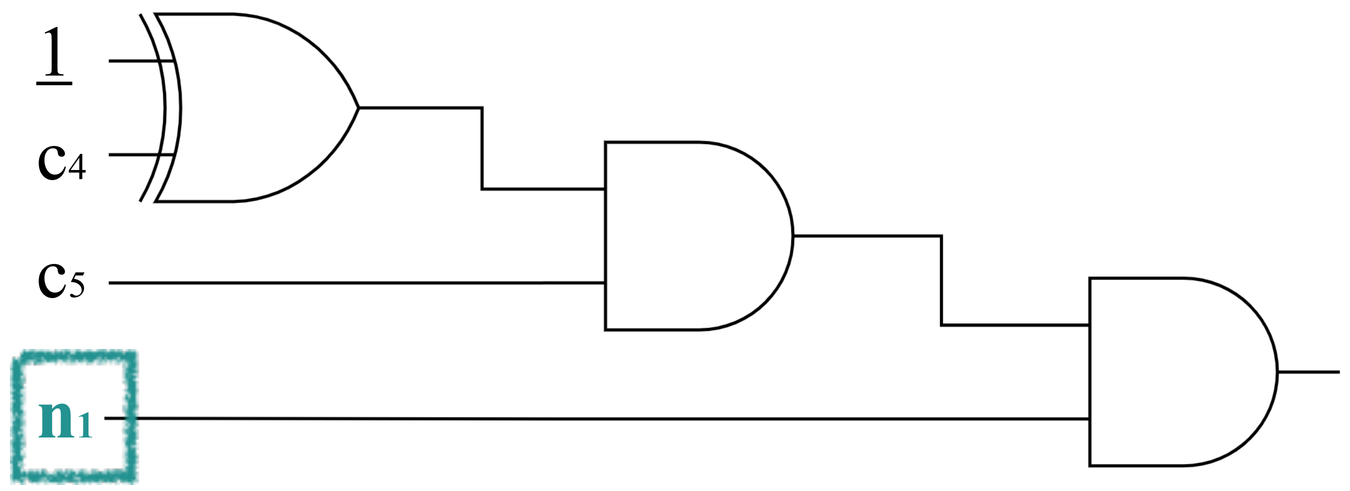


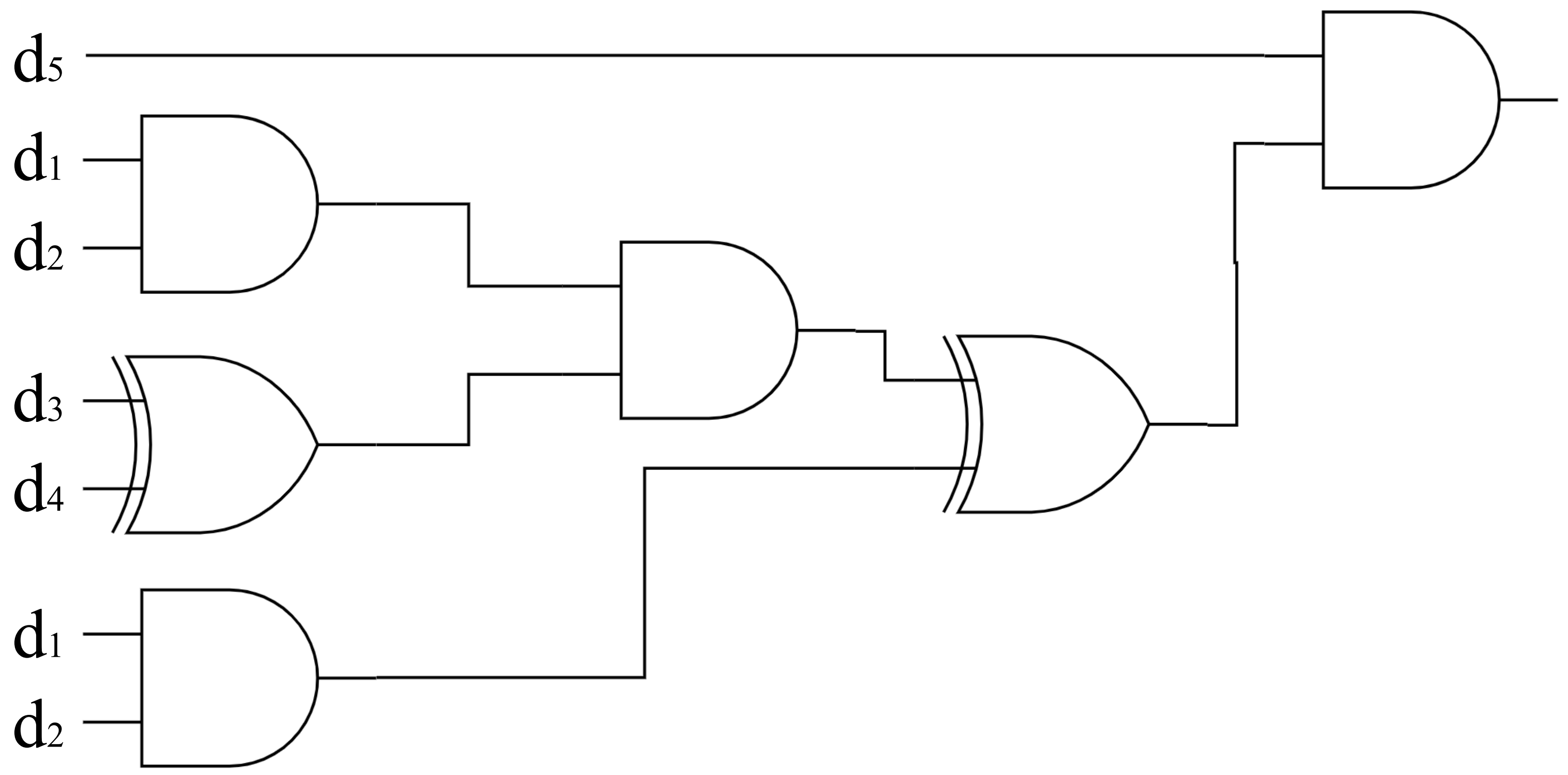
Applying Learned Optimization Patterns (2/2)

Normalization + Equational Matching



Normalized Opt. Patterns





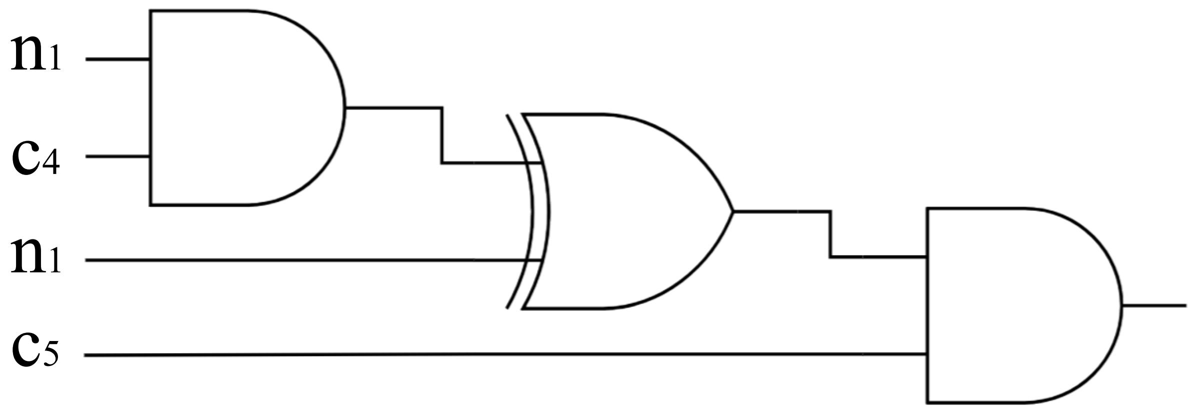
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

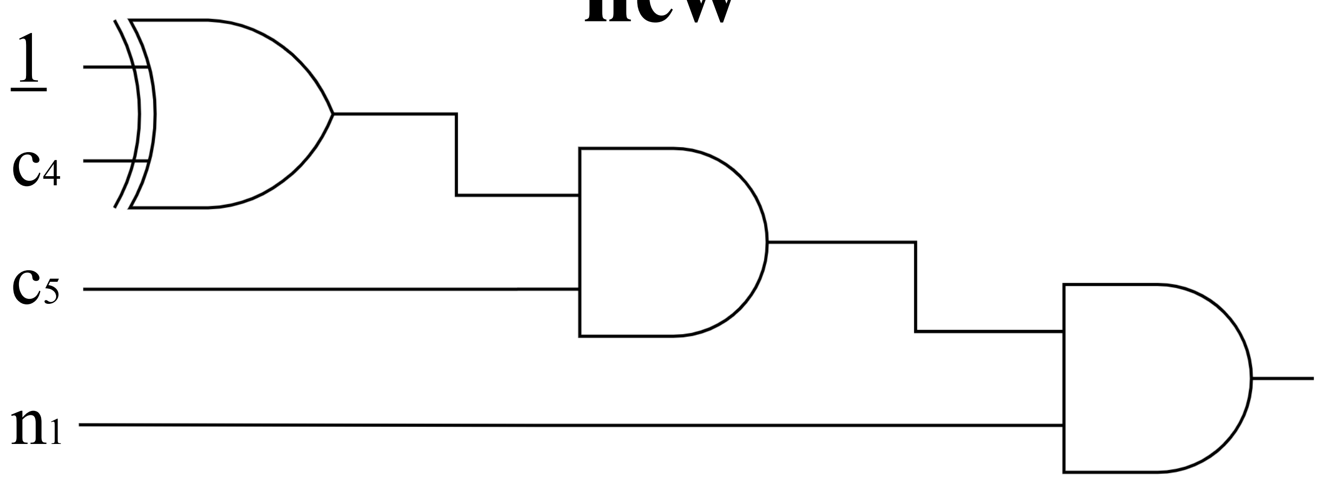
Normalization + Equational Matching

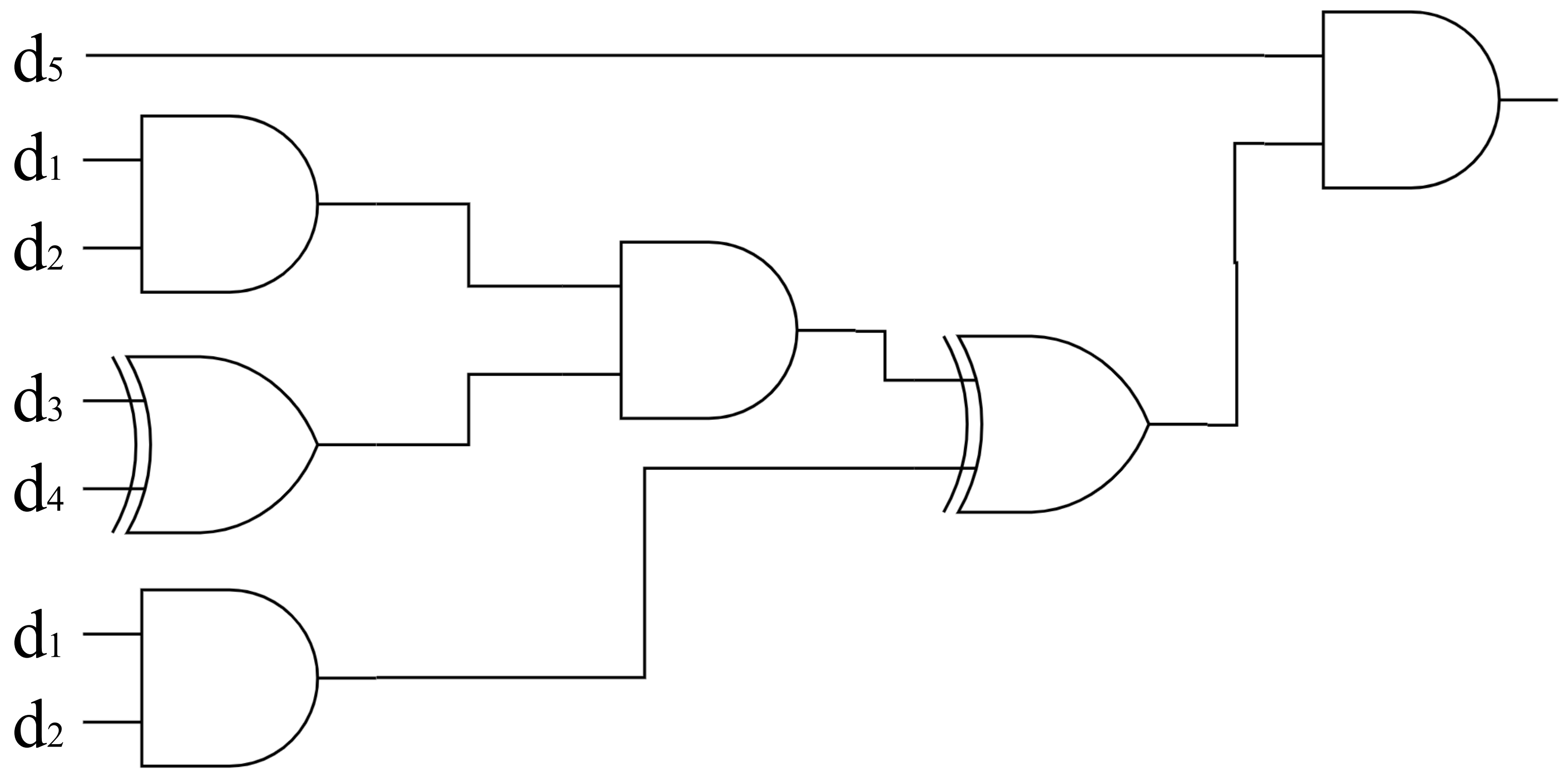
old



Normalized Opt. Patterns

new





target

New Input Circuit Optimization

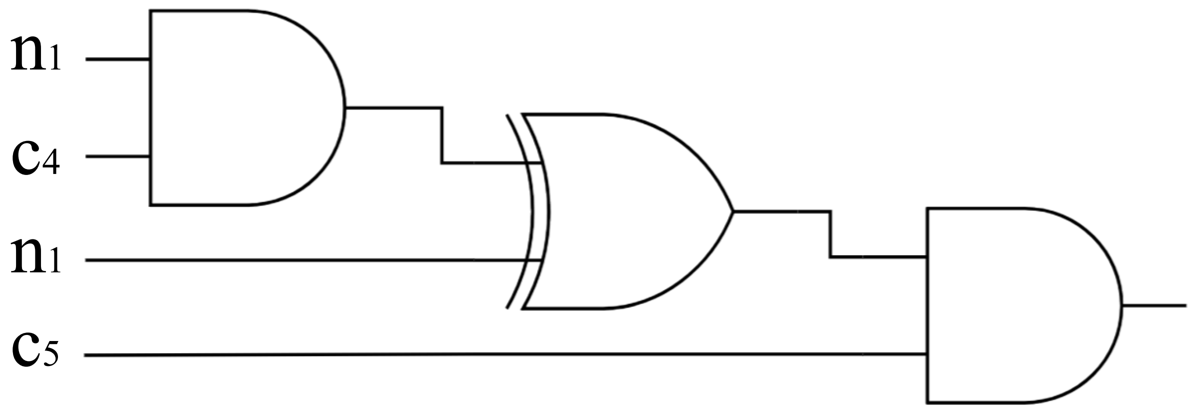
?

target'

Applying Learned Optimization Patterns (2/2)

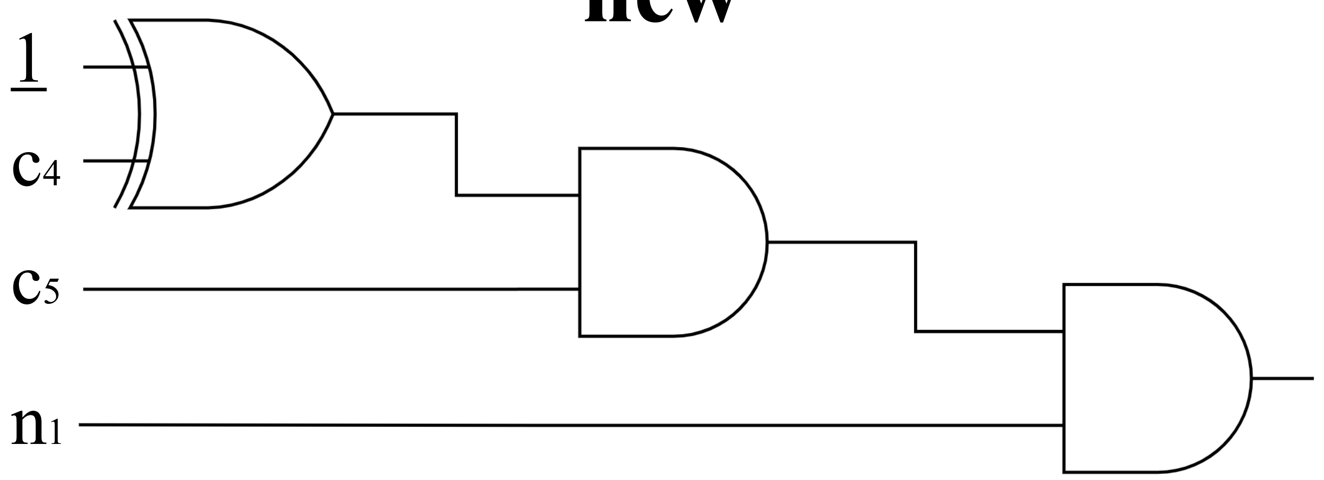
Normalization + Equational Matching

old



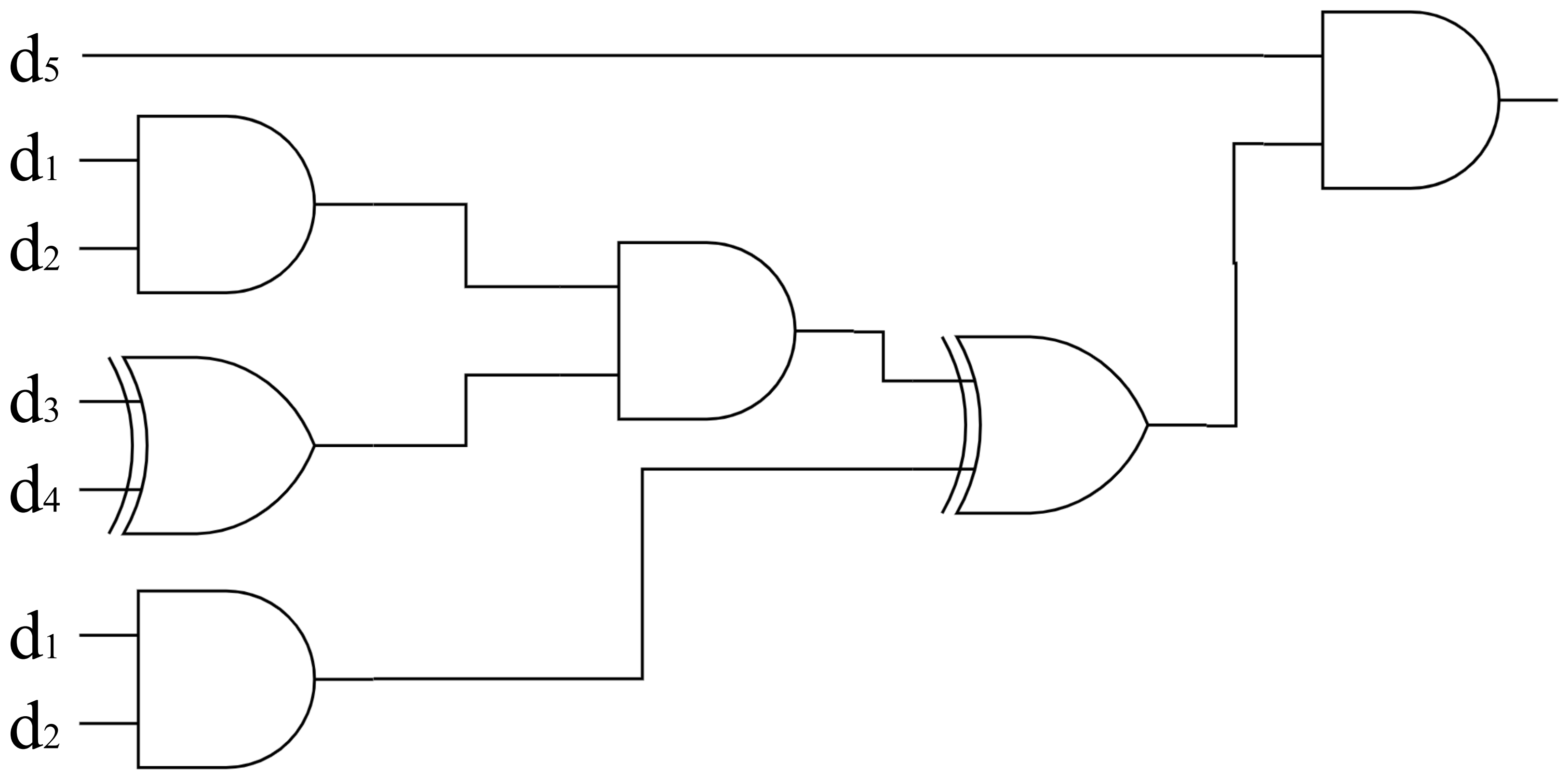
Normalized Opt. Patterns

new



Find substitution σ

(considering commutativity)



target

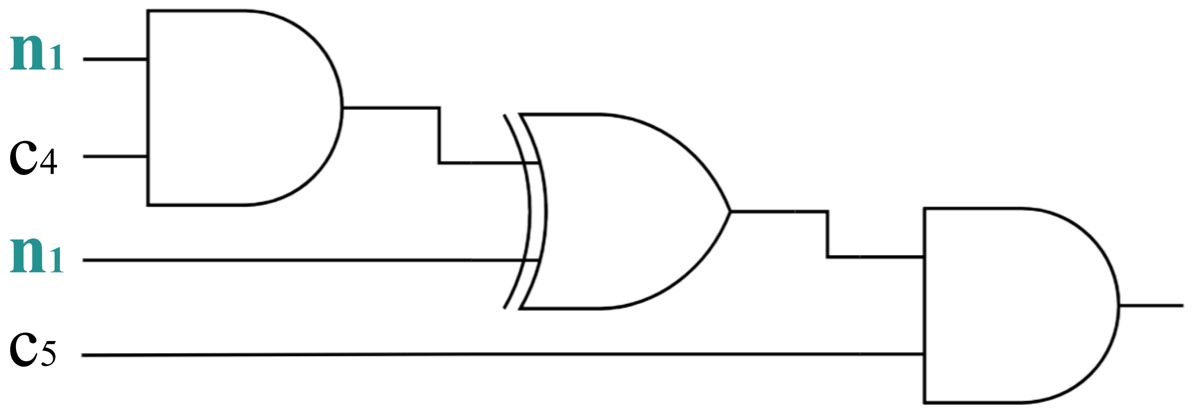
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

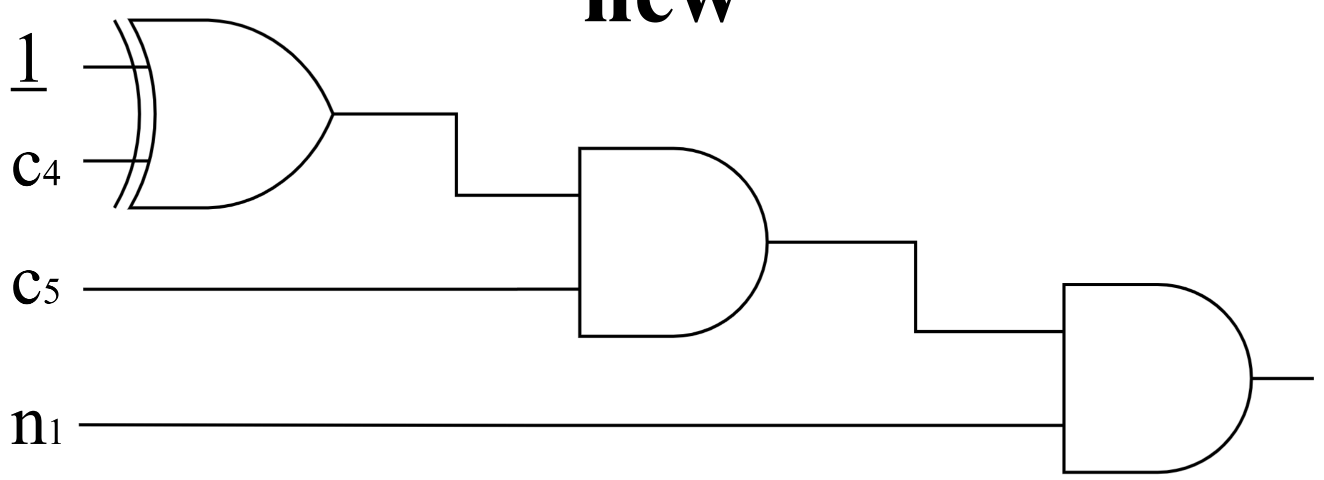
Normalization + Equational Matching

old



Normalized Opt. Patterns

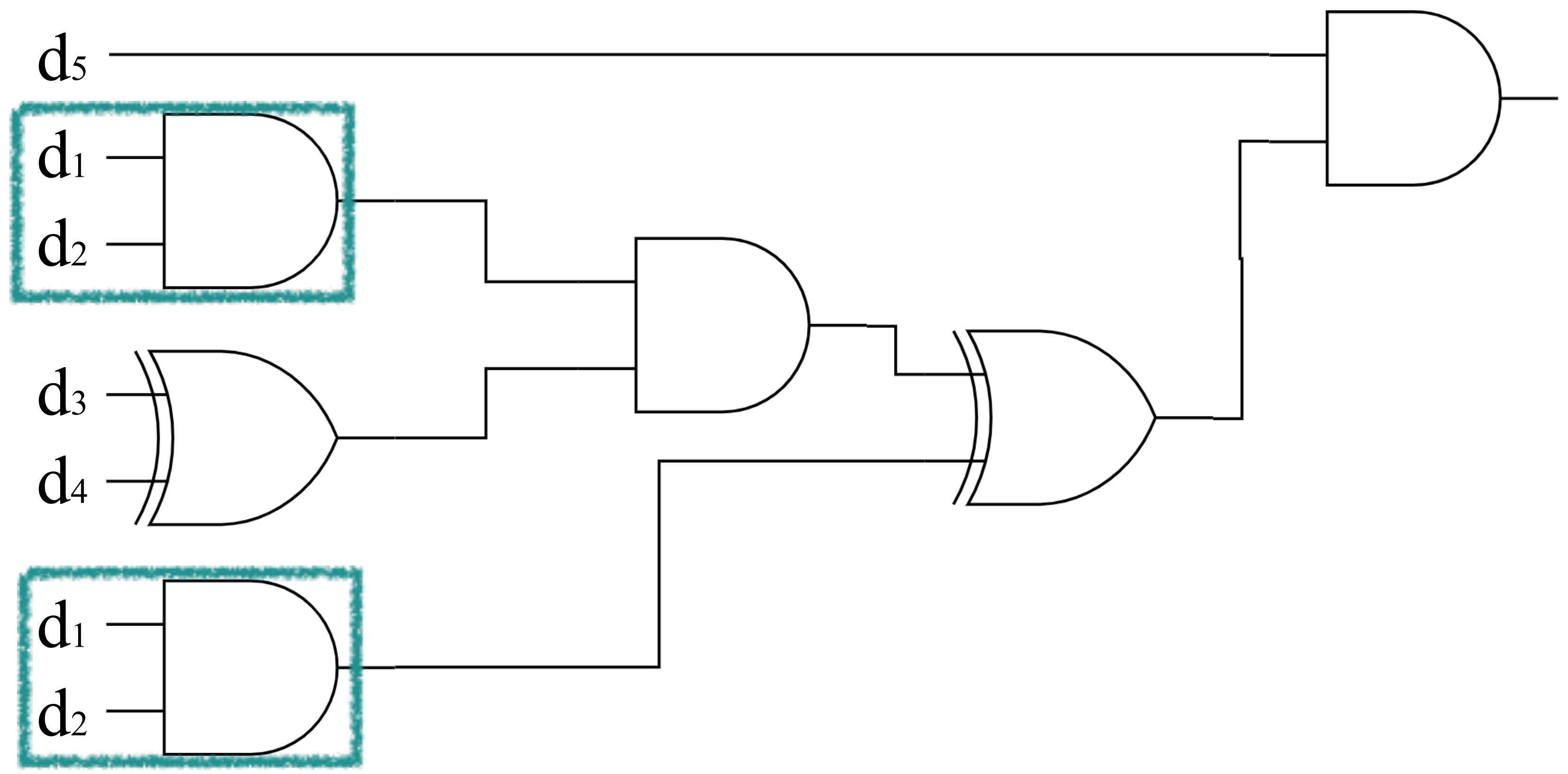
new



Find substitution σ

(considering commutativity)

$\sigma = \{n1 \mapsto d1 \text{ and } d2,$



target

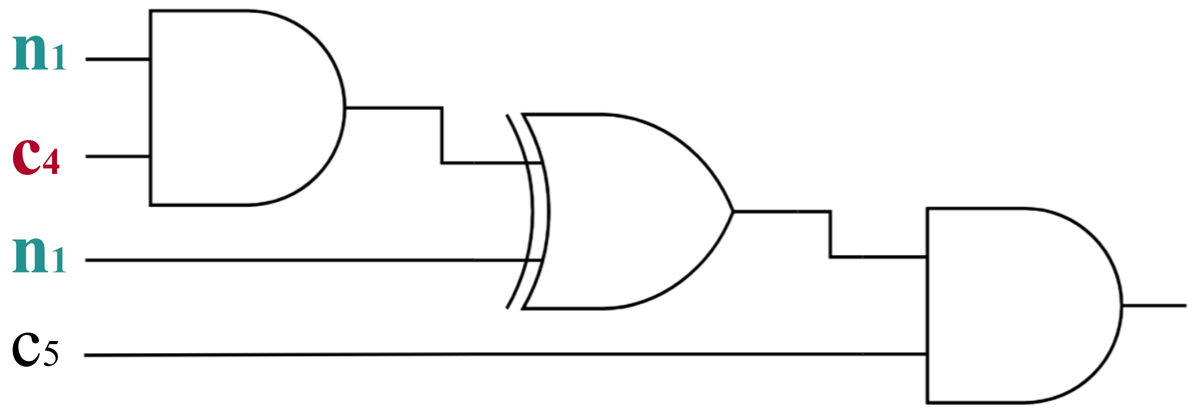
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

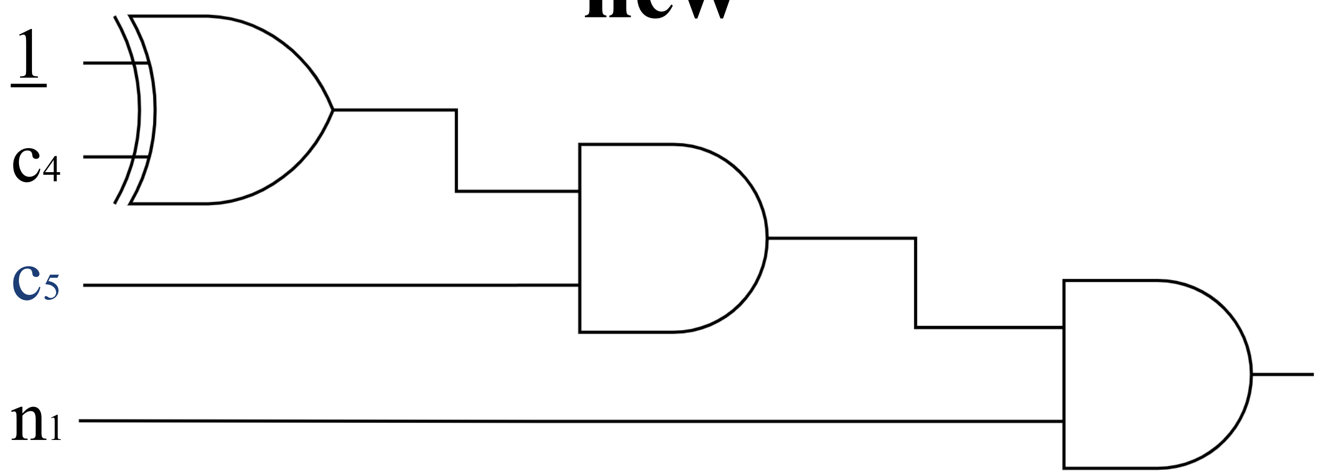
Normalization + Equational Matching

old



Normalized Opt. Patterns

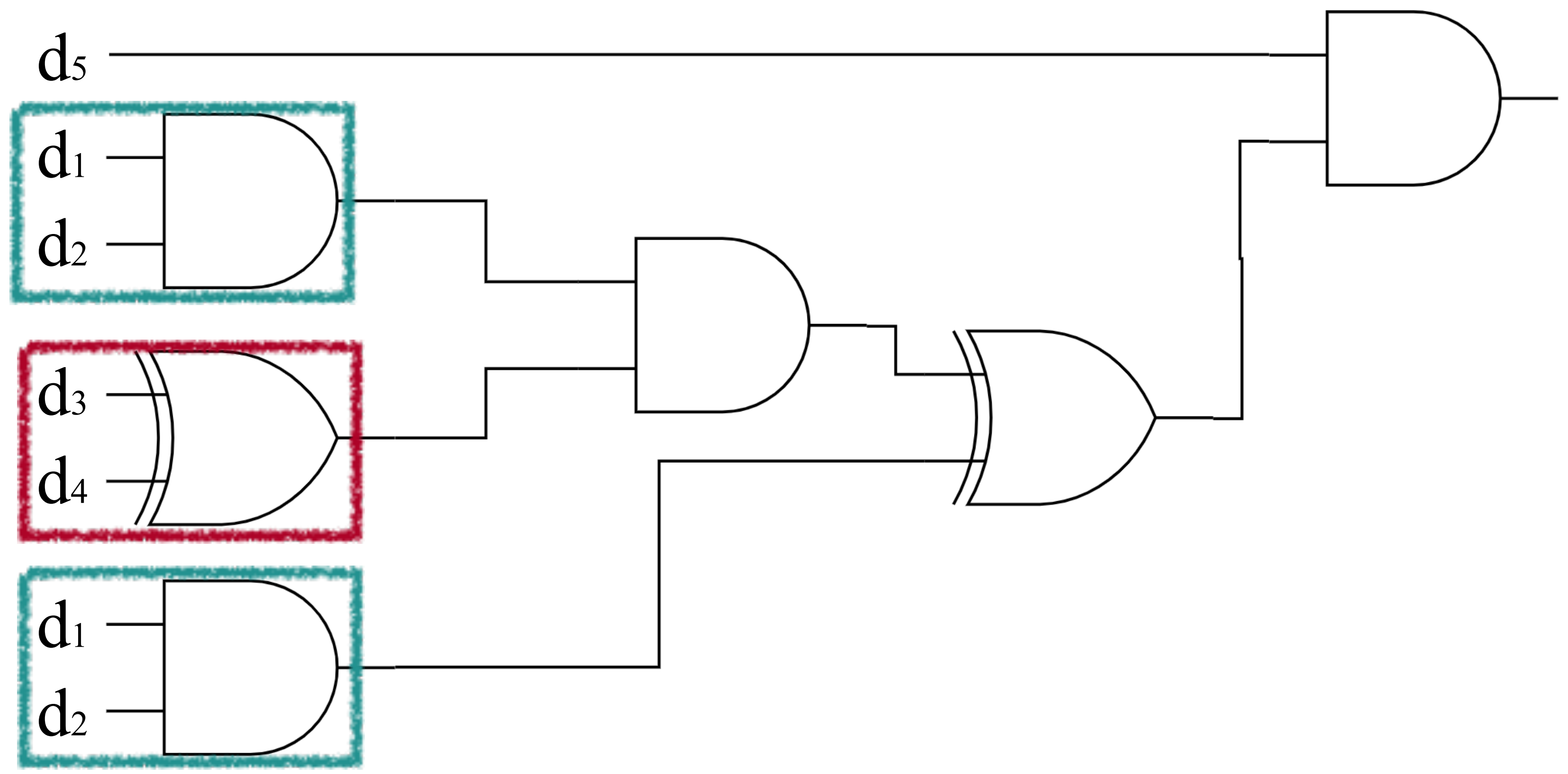
new



Find substitution σ

(considering commutativity)

σ = {n1 ↦ d1 and d2,
c4 ↦ d3 xor d4,



target

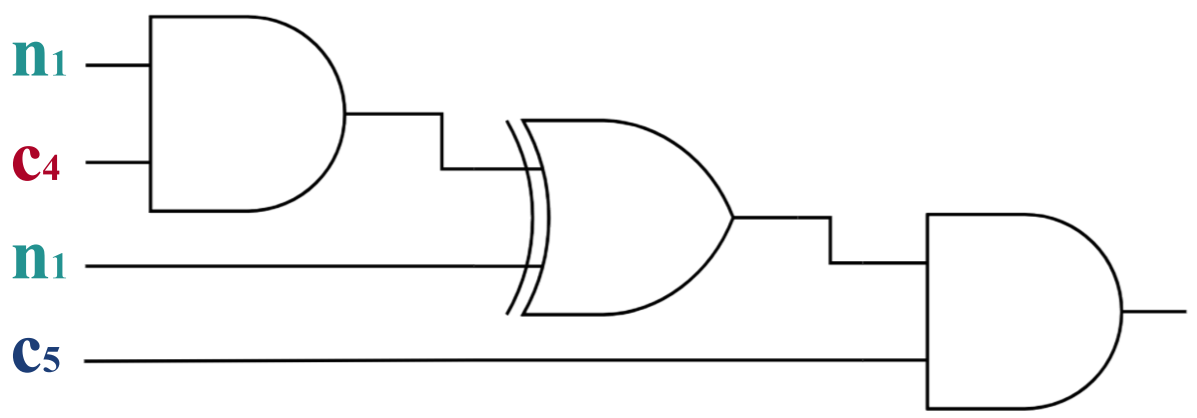
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

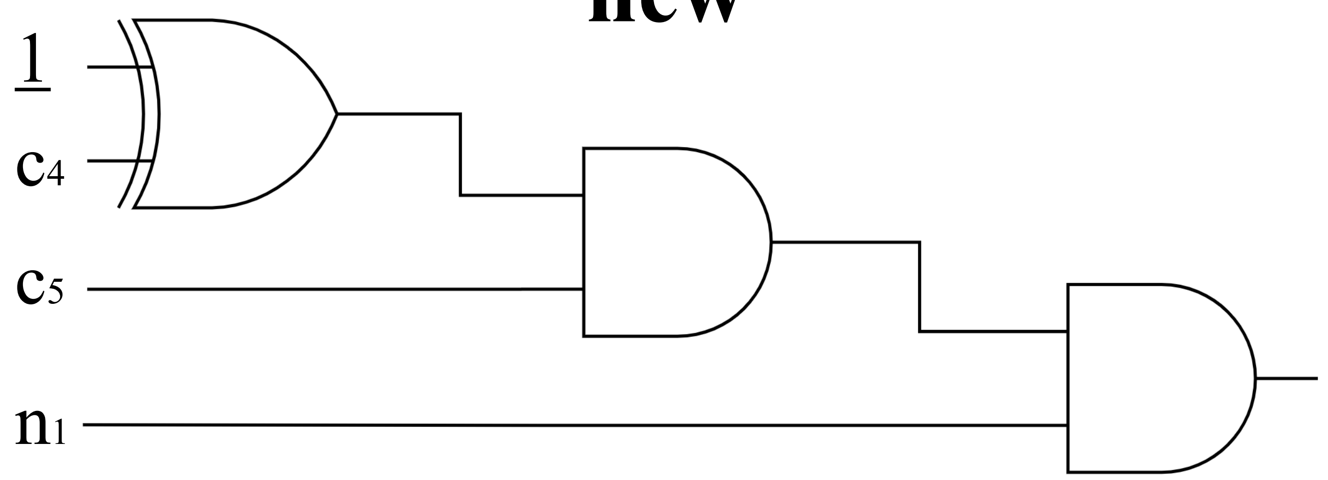
Normalization + Equational Matching

old



Normalized
Opt. Patterns

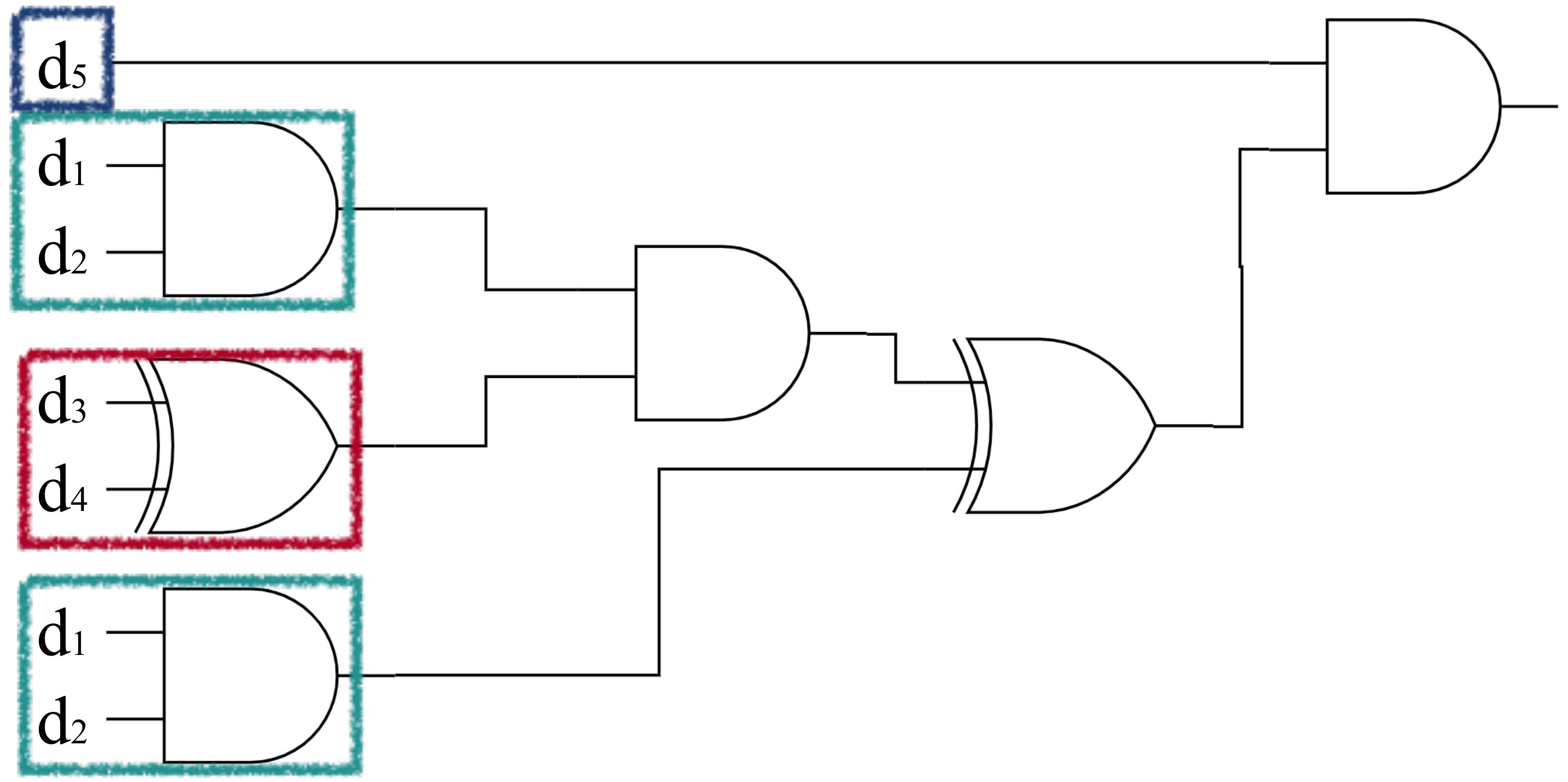
new



Find substitution σ

(considering commutativity)

σ = {n1 ↦ d1 and d2,
c4 ↦ d3 xor d4,
c5 ↦ d5}



target

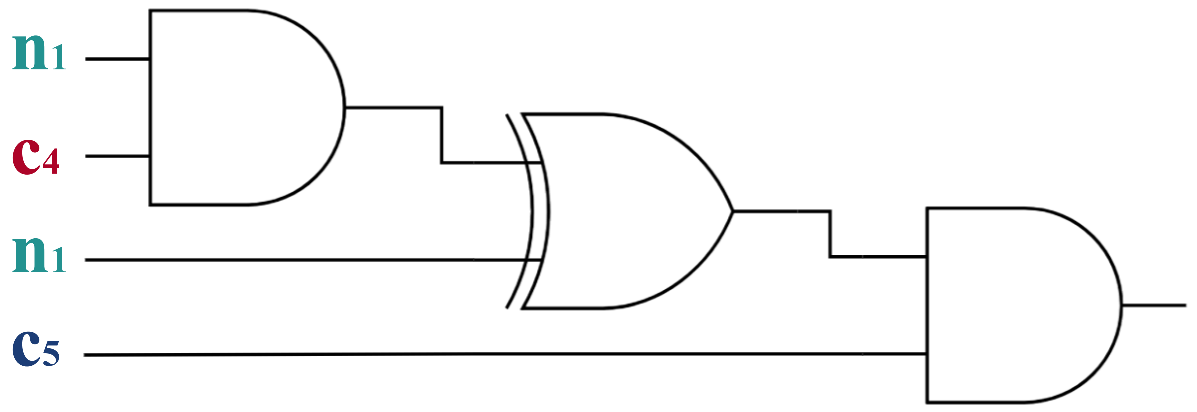
New Input Circuit
Optimization

?

Applying Learned Optimization Patterns (2/2)

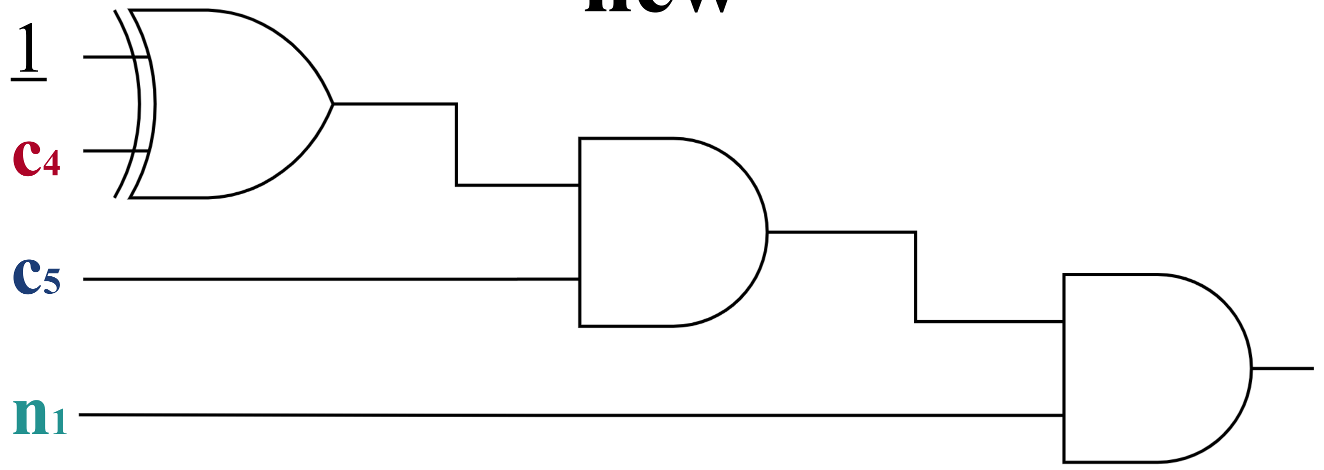
Normalization + Equational Matching

old



Normalized Opt. Patterns

new

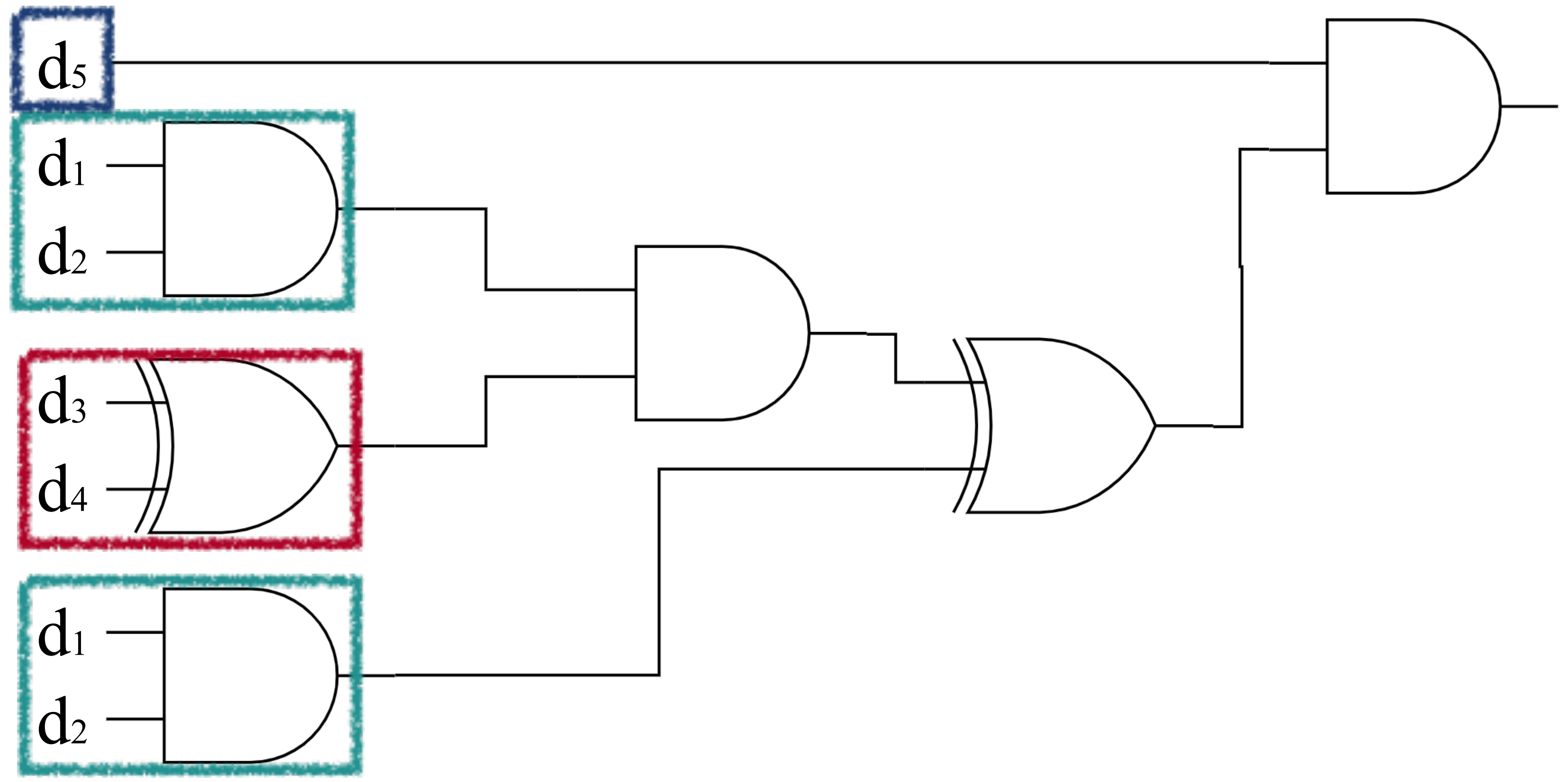


Find substitution σ

(considering commutativity)

Apply substitution σ

σ = {n1 ↦ d1 and d2,
c4 ↦ d3 xor d4,
c5 ↦ d5}



target

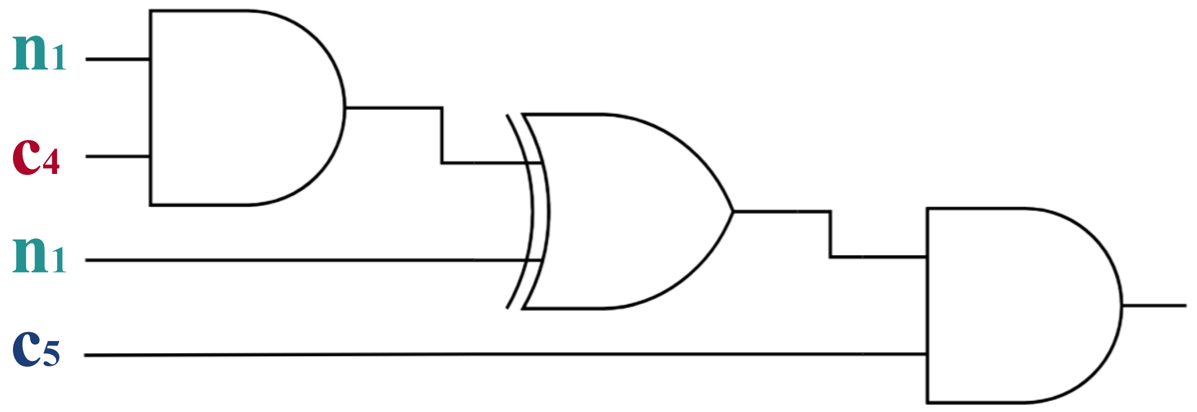
New Input Circuit Optimization

?

Applying Learned Optimization Patterns (2/2)

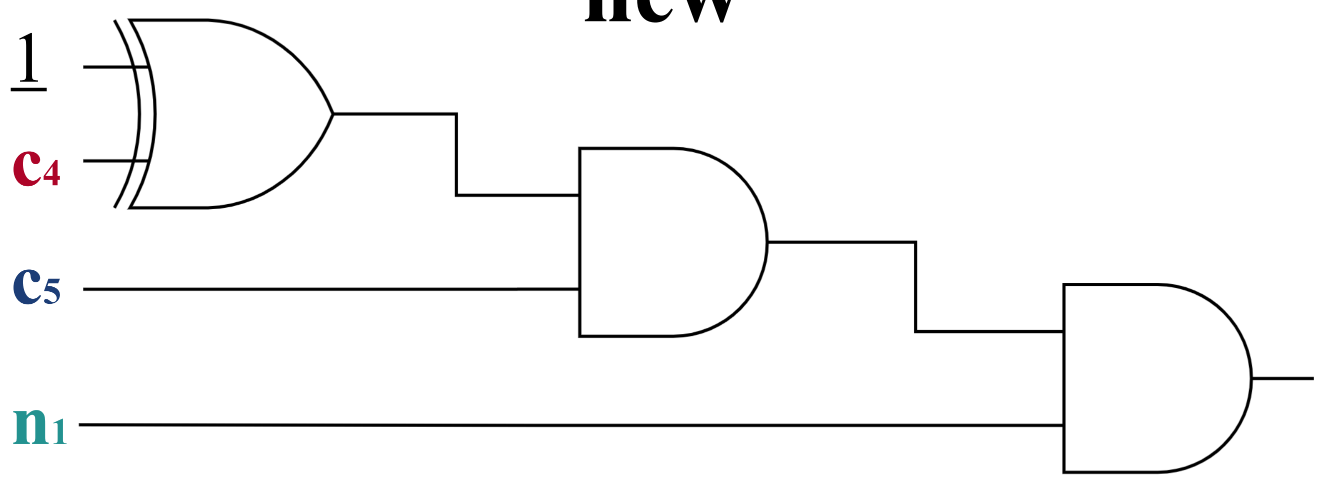
Normalization + Equational Matching

old



Normalized Opt. Patterns →

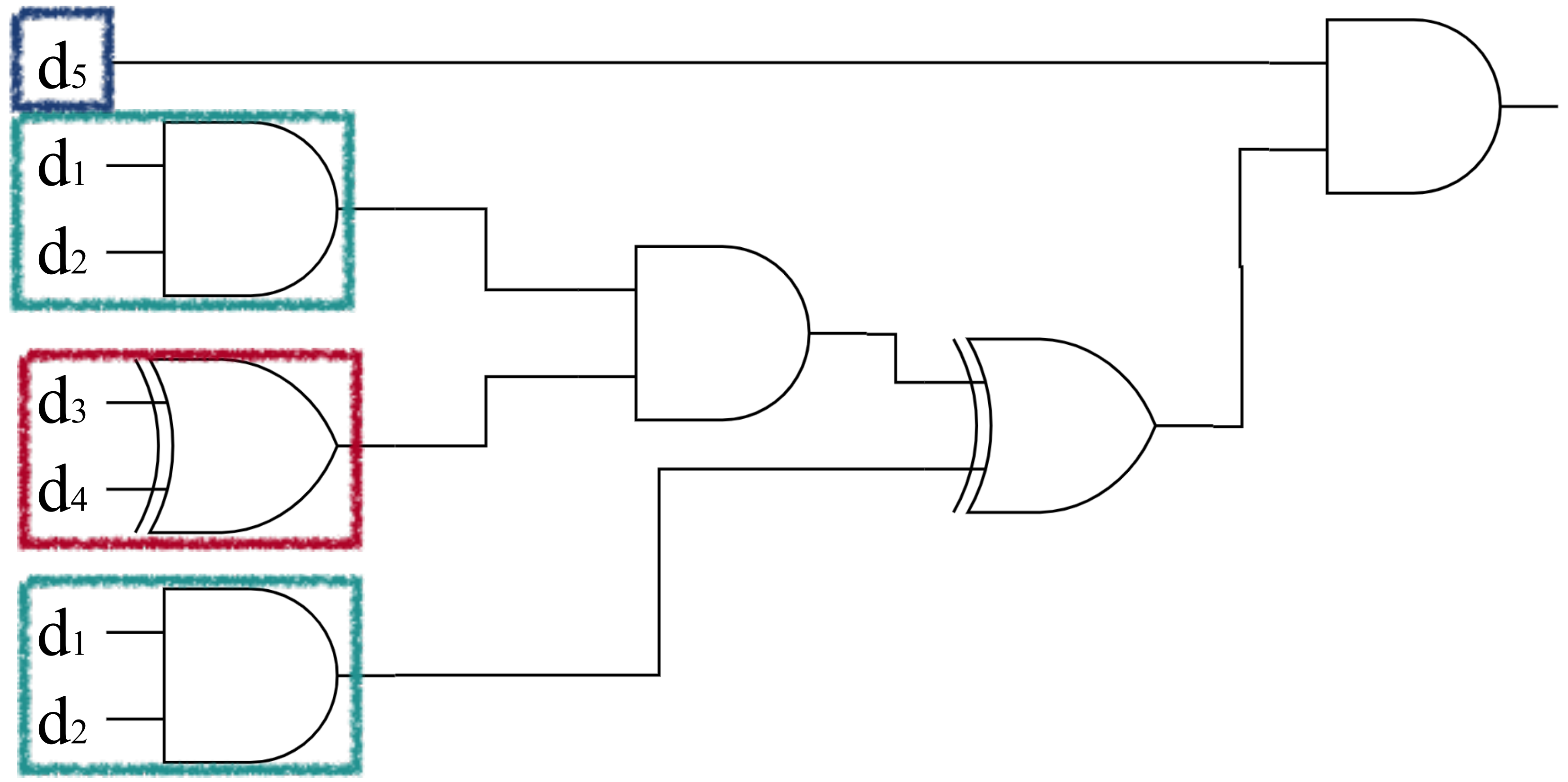
new



Find substitution σ (considering commutativity)

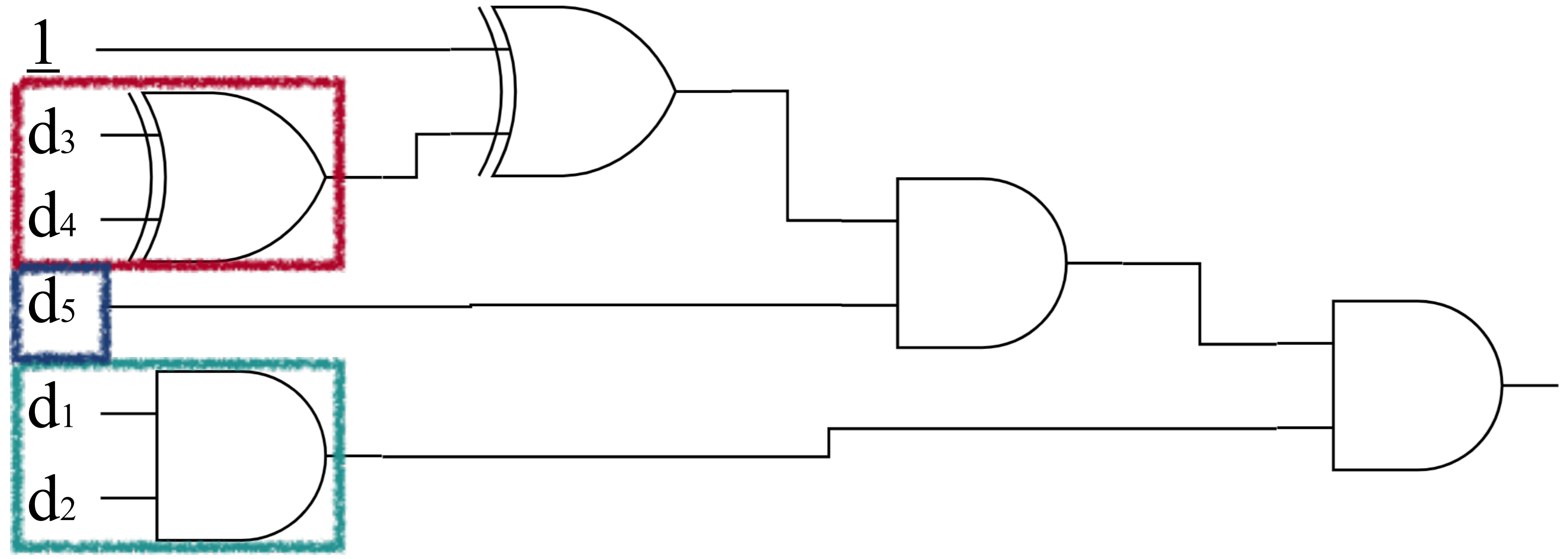
Apply substitution σ

σ = {n1 ↦ d1 and d2,
c4 ↦ d3 xor d4,
c5 ↦ d5}



target

New Input Circuit Optimization →

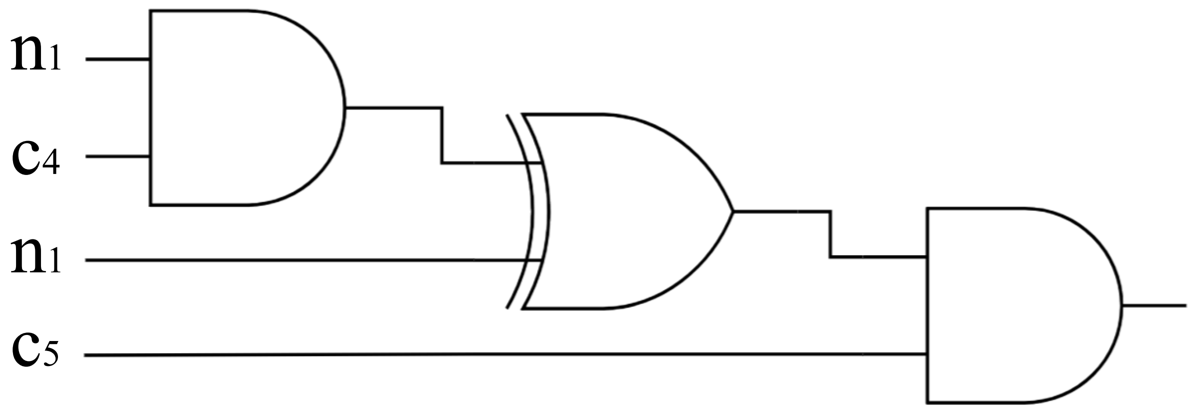


optimized target

Applying Learned Optimization Patterns (2/2)

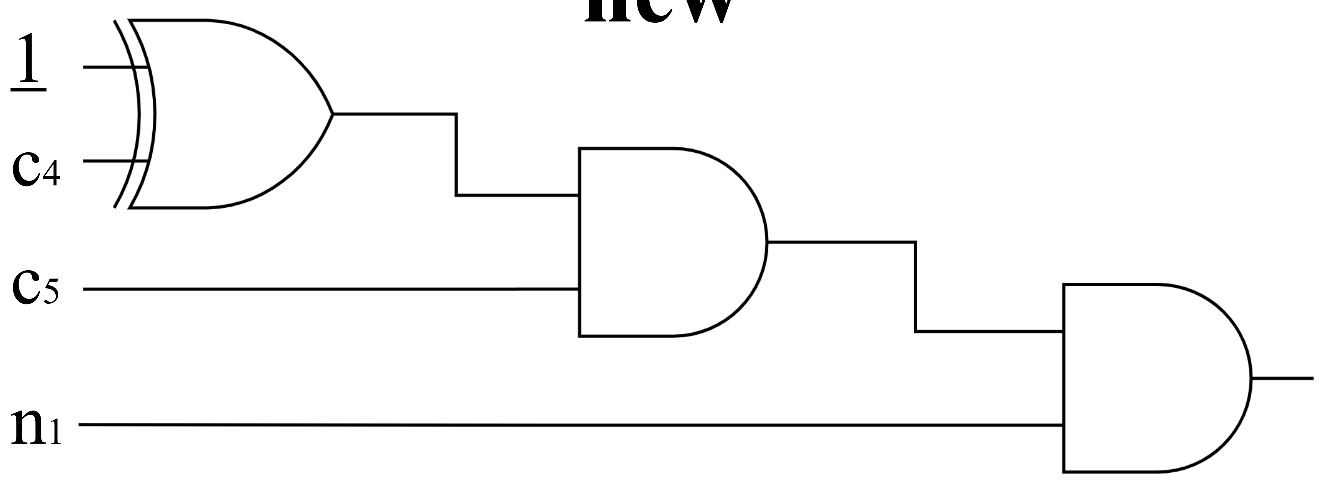
Normalization + Equational Matching

old



Normalized Opt. Patterns

new

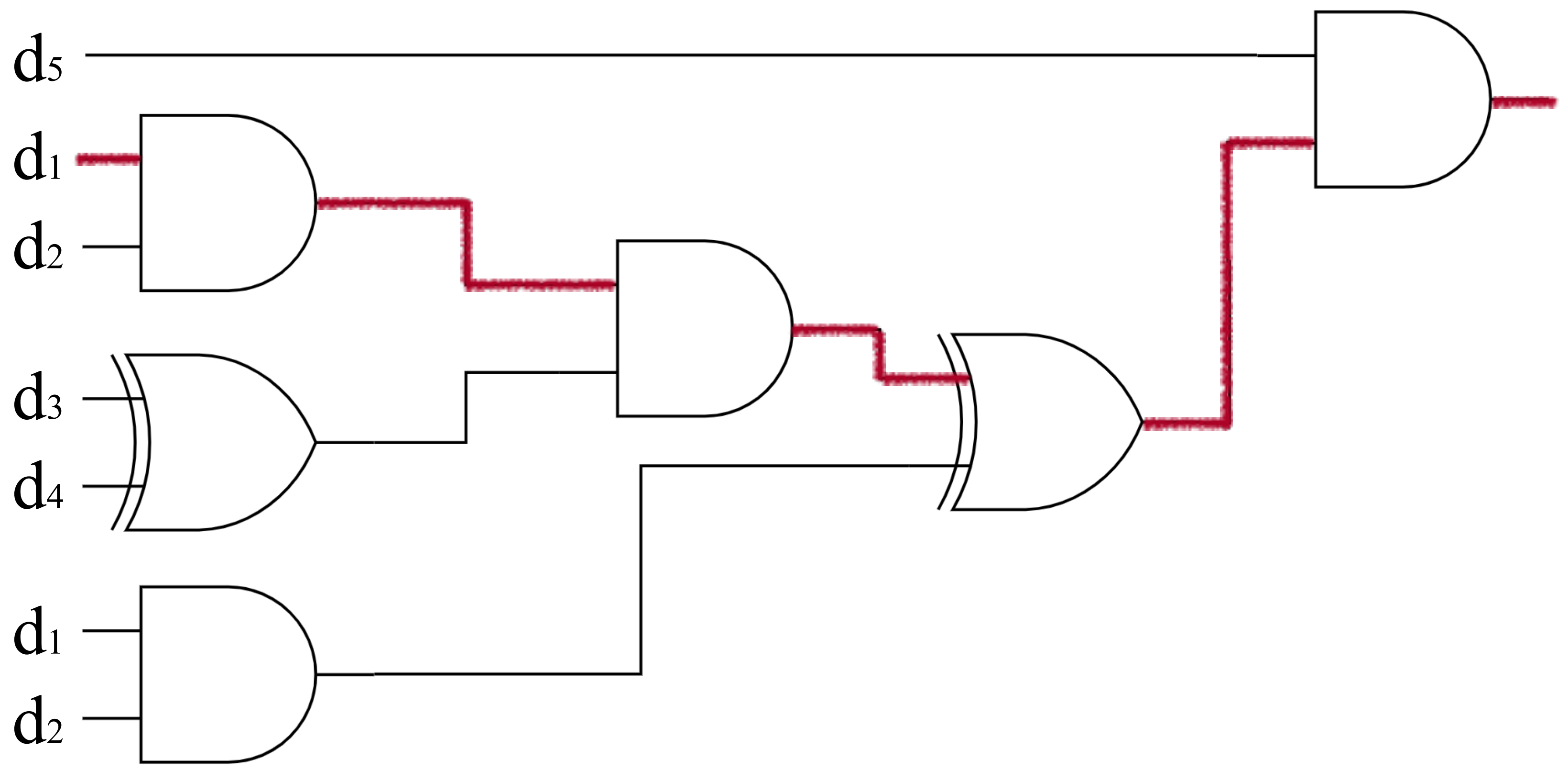


Find substitution σ

(considering commutativity)

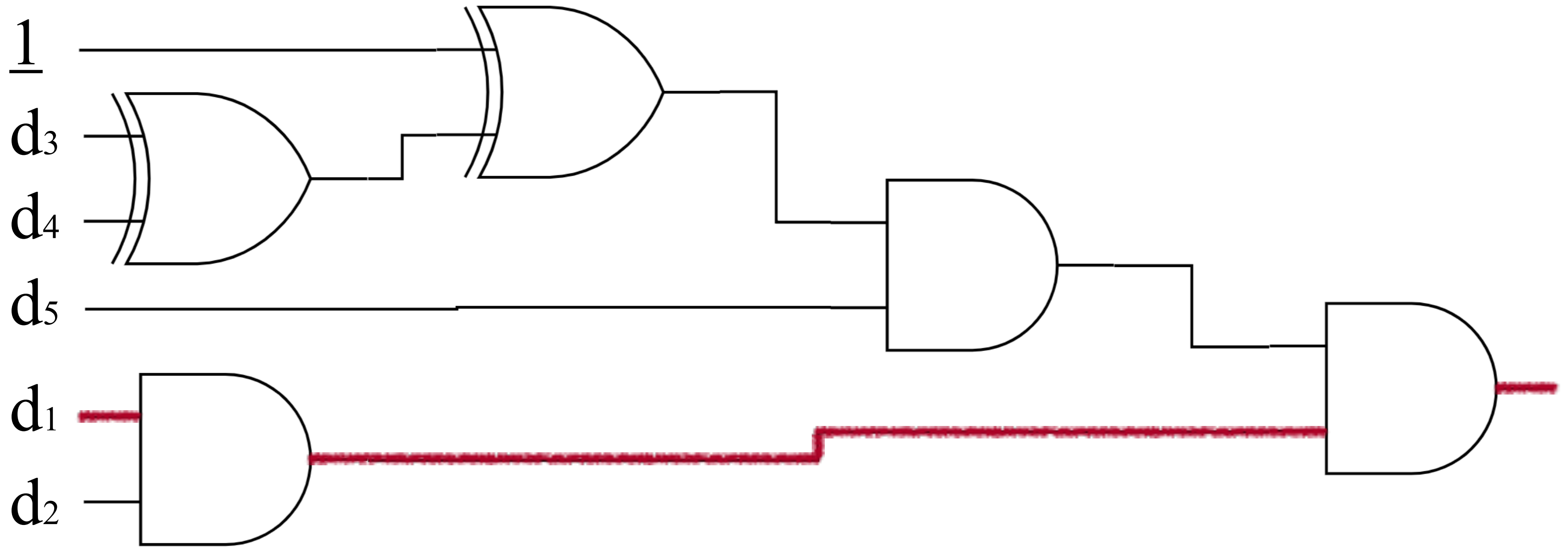
Apply substitution σ

$\sigma = \{n1 \mapsto d1 \text{ and } d2,$
$c4 \mapsto d3 \text{ xor } d4,$
$c5 \mapsto d5\}$



target
depth 3

New Input Circuit Optimization

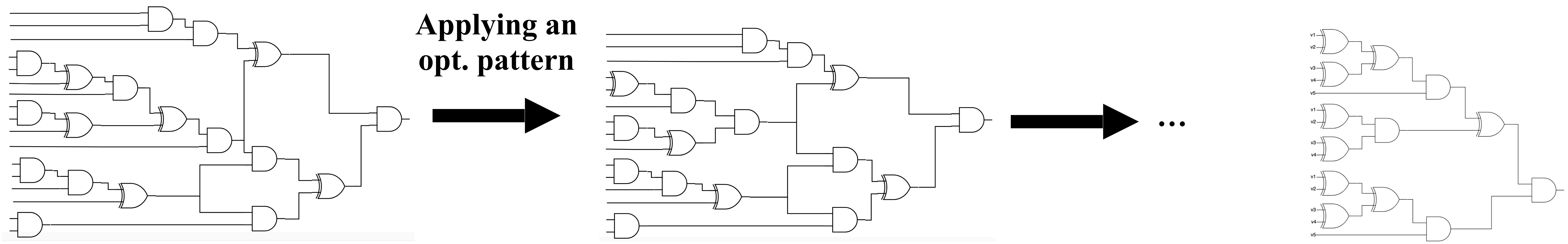


optimized target
depth 2

Applying Learned Optimization Patterns

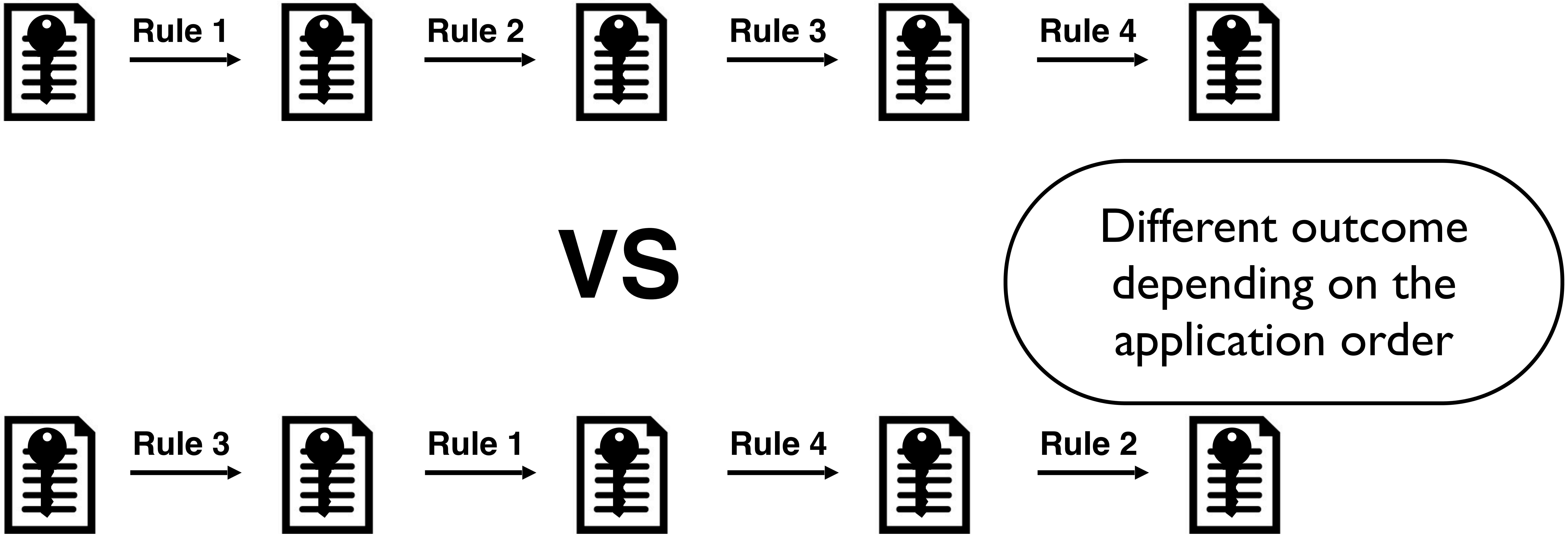
Formal properties

(Soundness) semantics unchanged



(Termination) finitely many rule applications

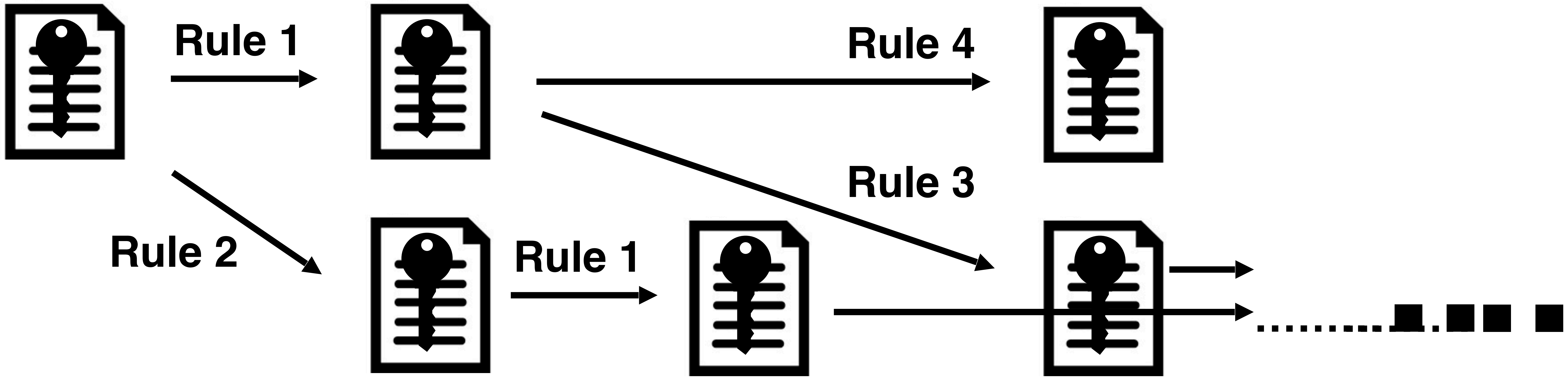
Phase-Ordering Problem



Existing Solutions

- Using a pre-defined application order (e.g., LLVM optimization passes)

- Backtracking (i.e., maintaining top-k candidates)



Equality Saturation

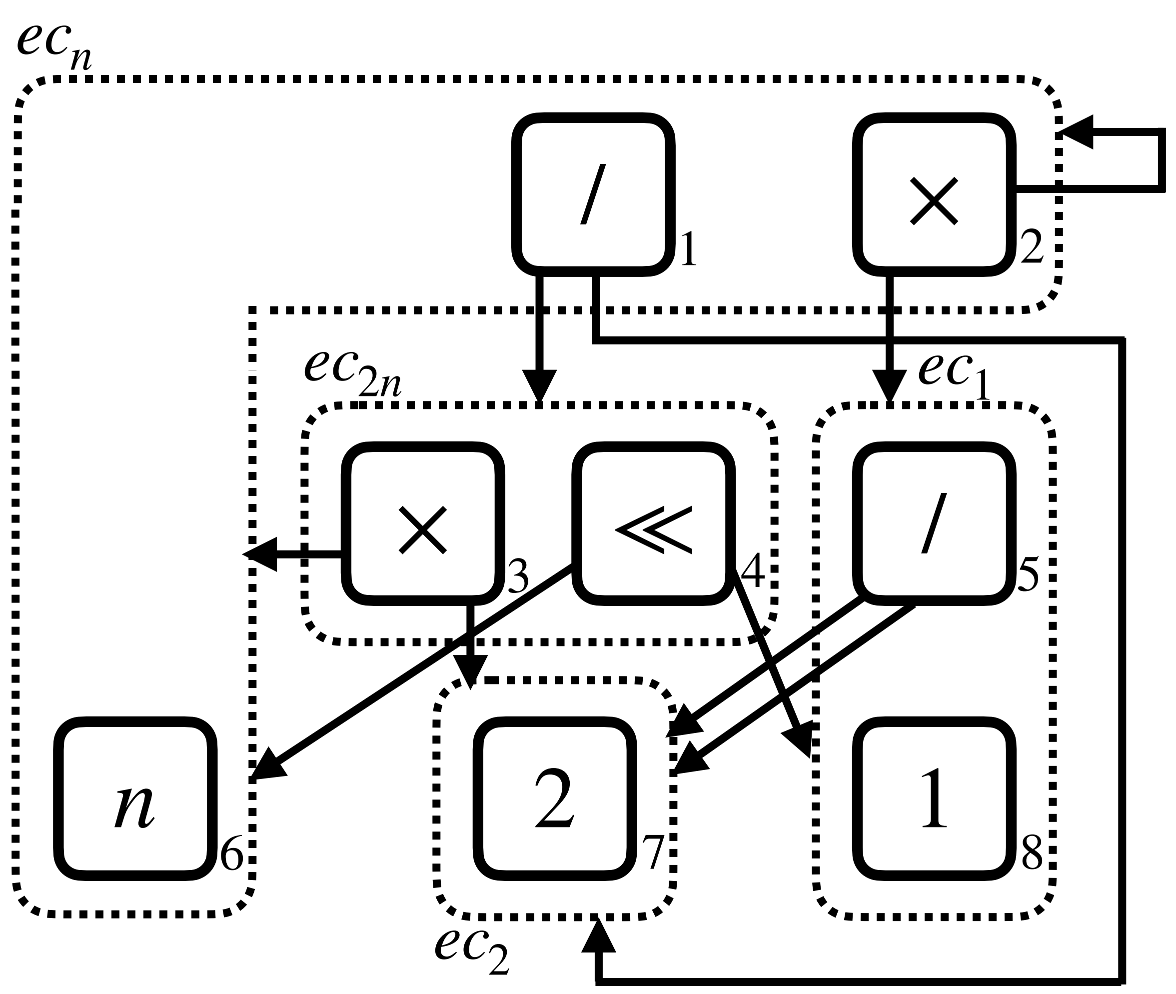
- A solution to the phase ordering problem

- Obtains results of all possible orderings and extract the best one among them

- Enabled by ***E-graph***, a very efficient data structure

E-Graph

- E-graph = e-nodes + e-classes
  - E-classes = set of e-nodes
  - E-node = a node whose children are e-classes



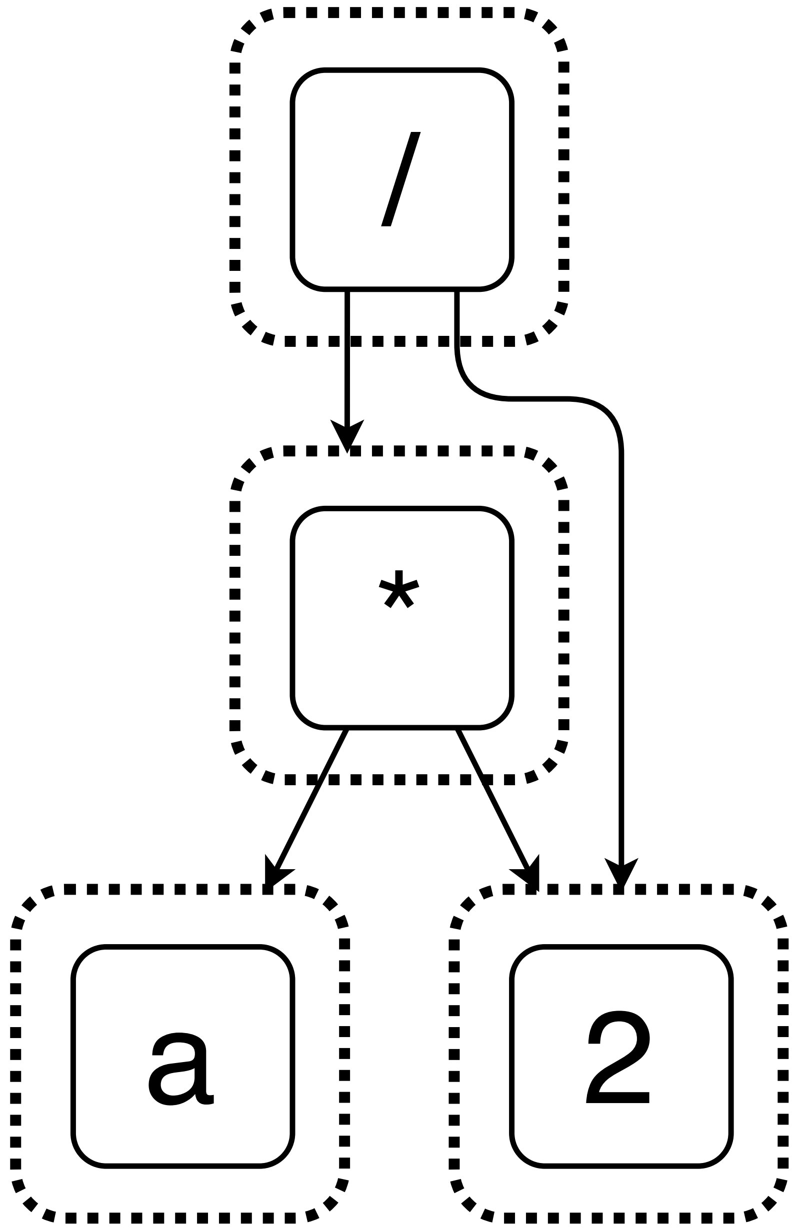
- Meaning
  - E-node (bold): expressions with sub-expressions represented by children e-classes
  - E-class (doted): semantically equivalent e-nodes

Example

- Optimizing $(a \times 2)/2$ using the following rules:

(1) $x \times 2 \rightarrow x \ll 1$ (2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$ (4) $1 \times x \rightarrow x$

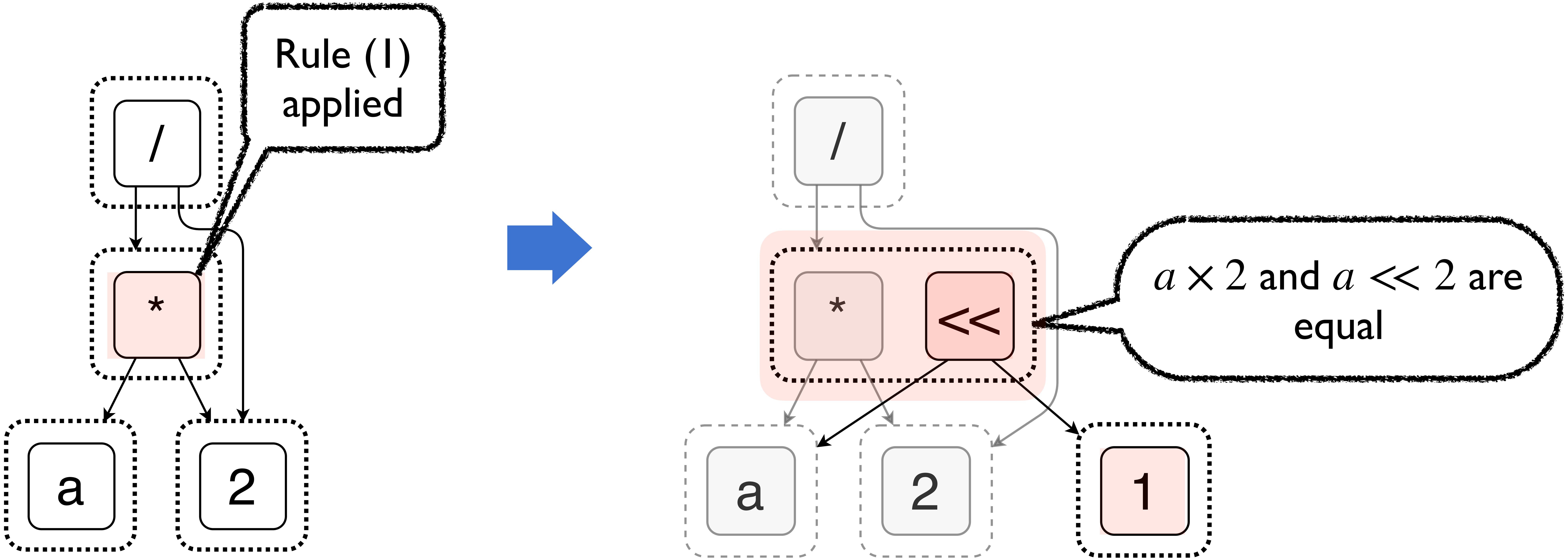


Example

- Optimizing $(a \times 2)/2$ using the following rules:

(1) $x \times 2 \rightarrow x \ll 1$ (2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$ (4) $1 \times x \rightarrow x$



Example

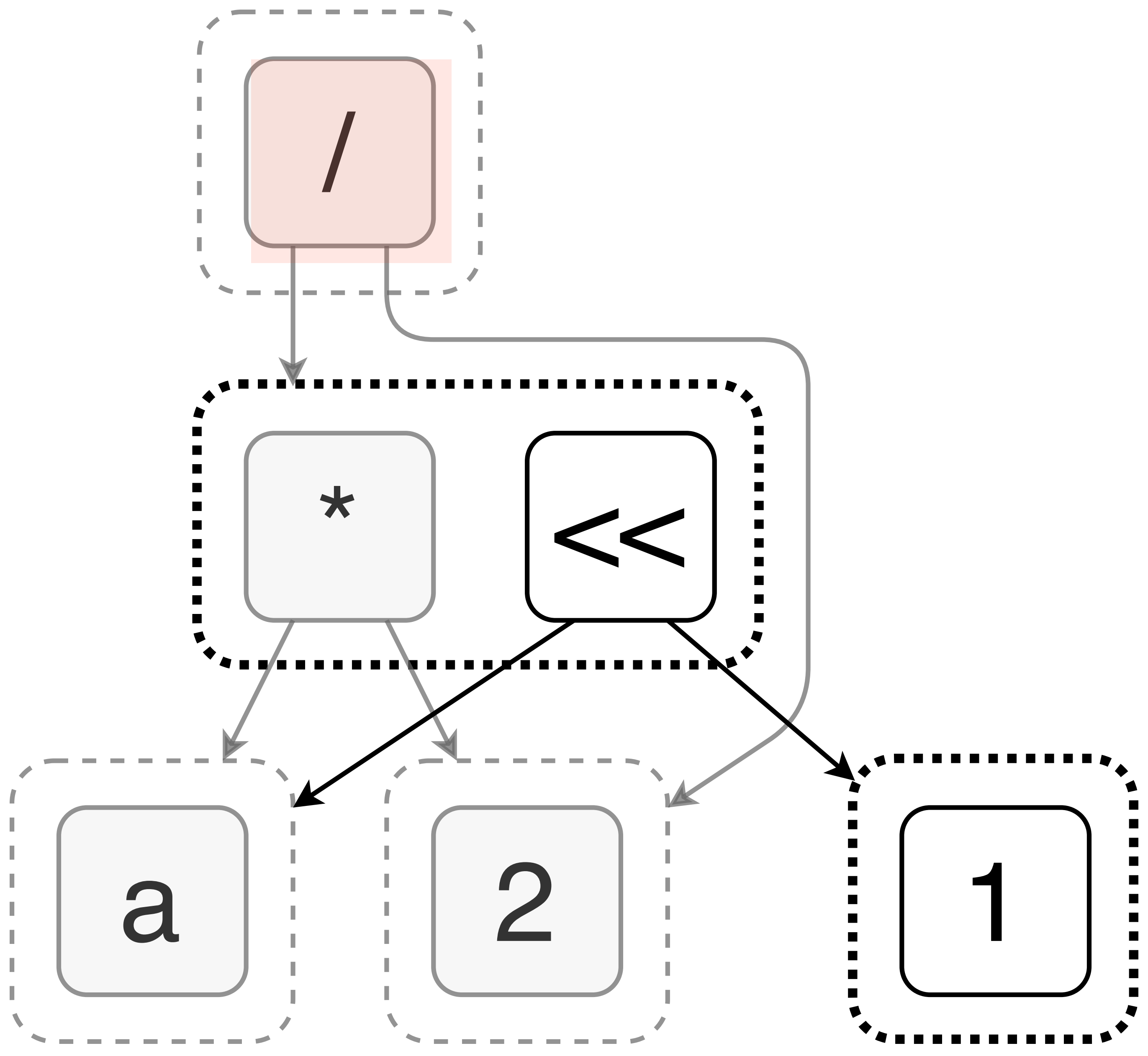
- Optimizing $(a \times 2)/2$ using the following rules:

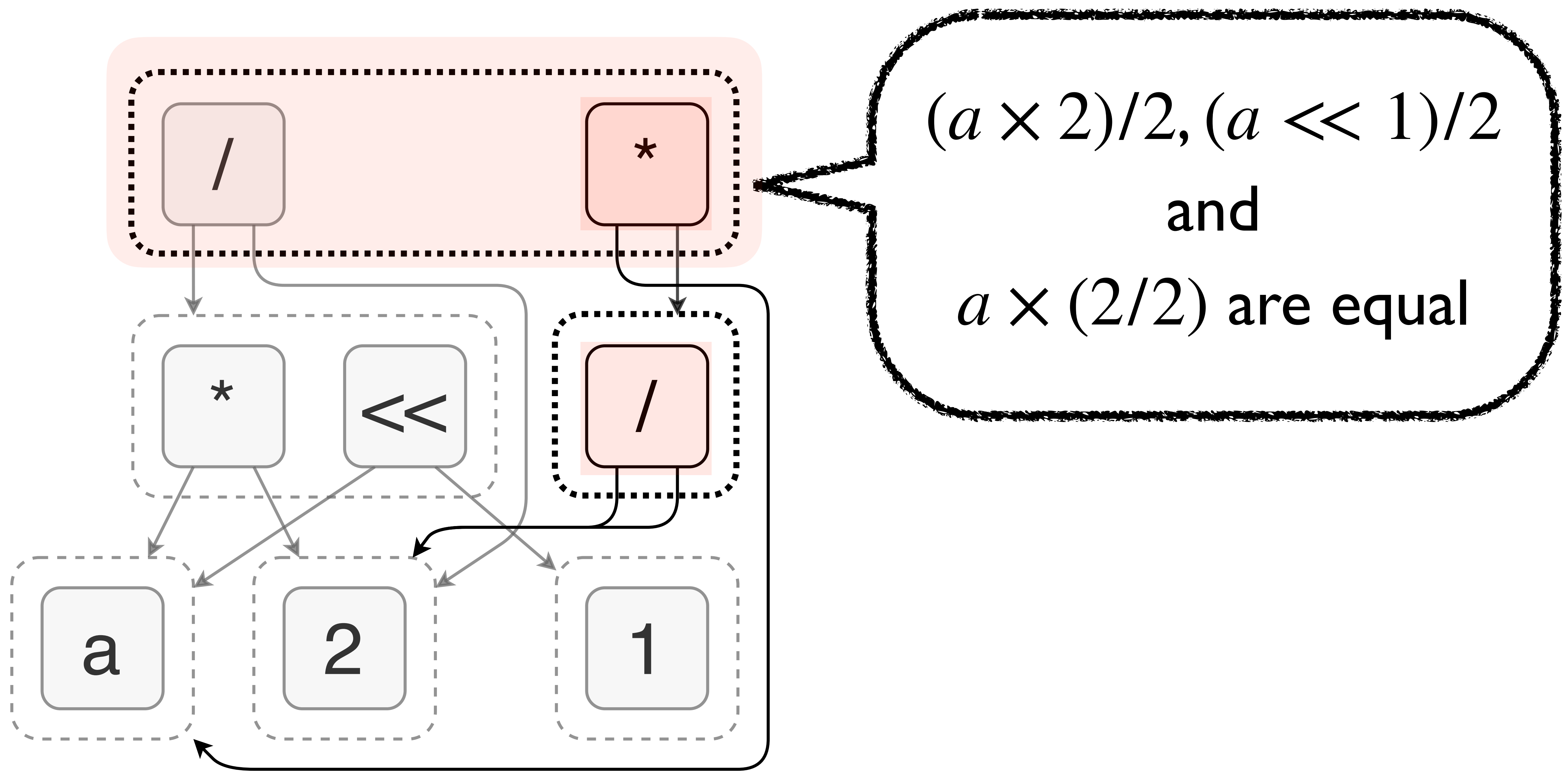
(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$





Example

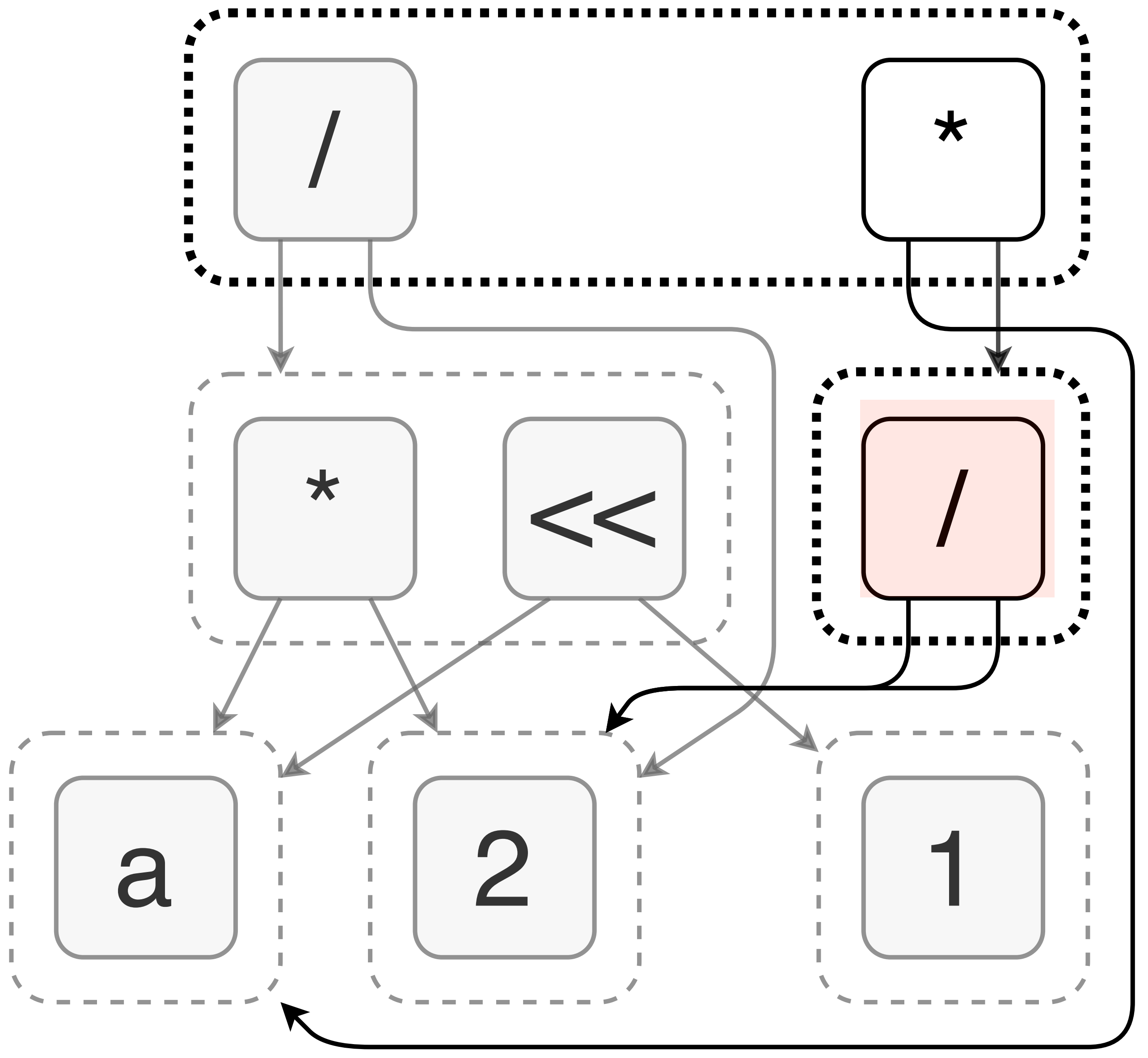
- Optimizing $(a \times 2)/2$ using the following rules:

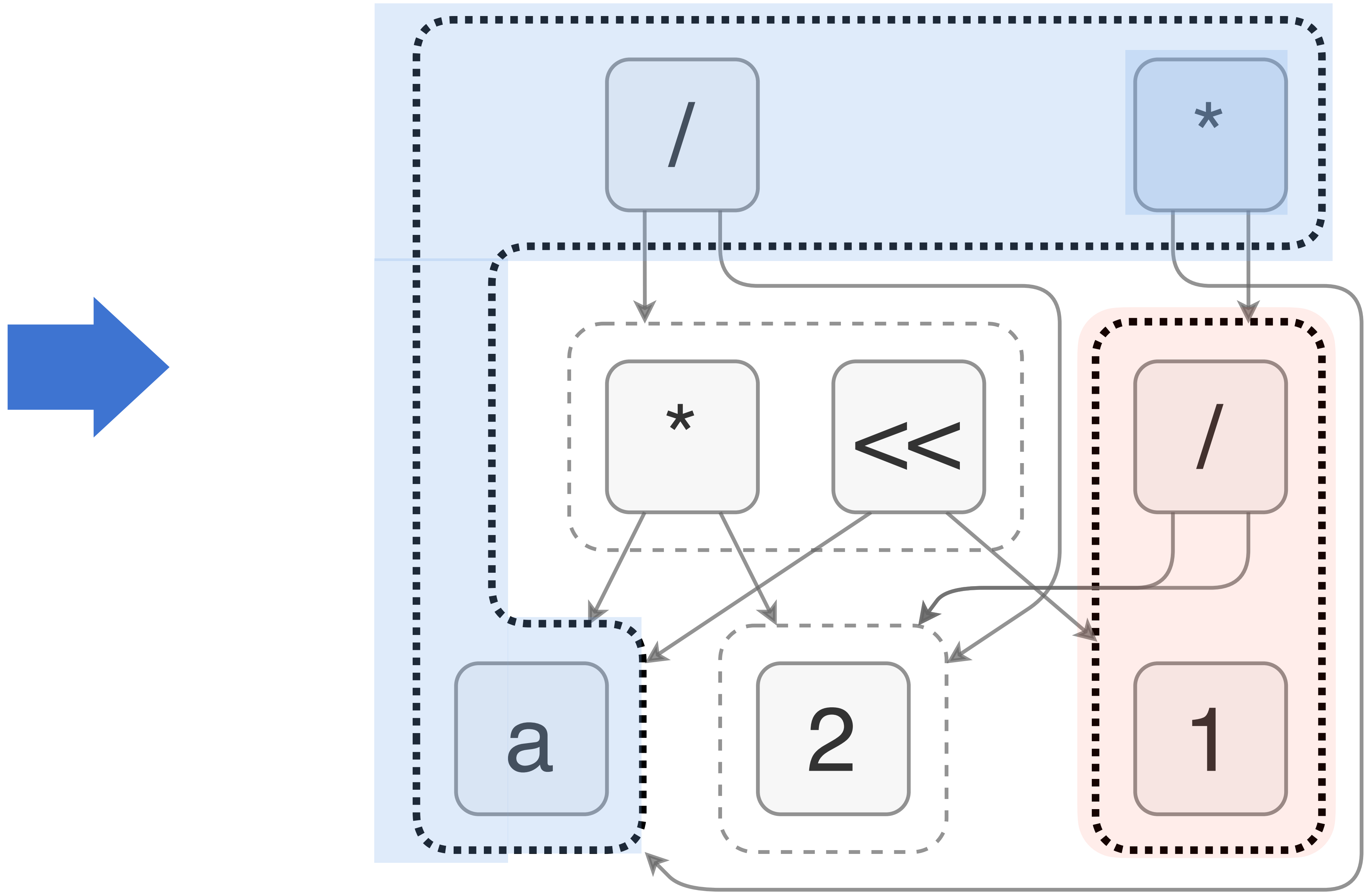
(1) $x \times 2 \rightarrow x \ll 1$

(2) $(x \times y)/z \rightarrow x \times (y/z)$

(3) $x/x \rightarrow 1$

(4) $1 \times x \rightarrow x$





Example

- Optimizing $(a \times 2)/2$ using the following rules:

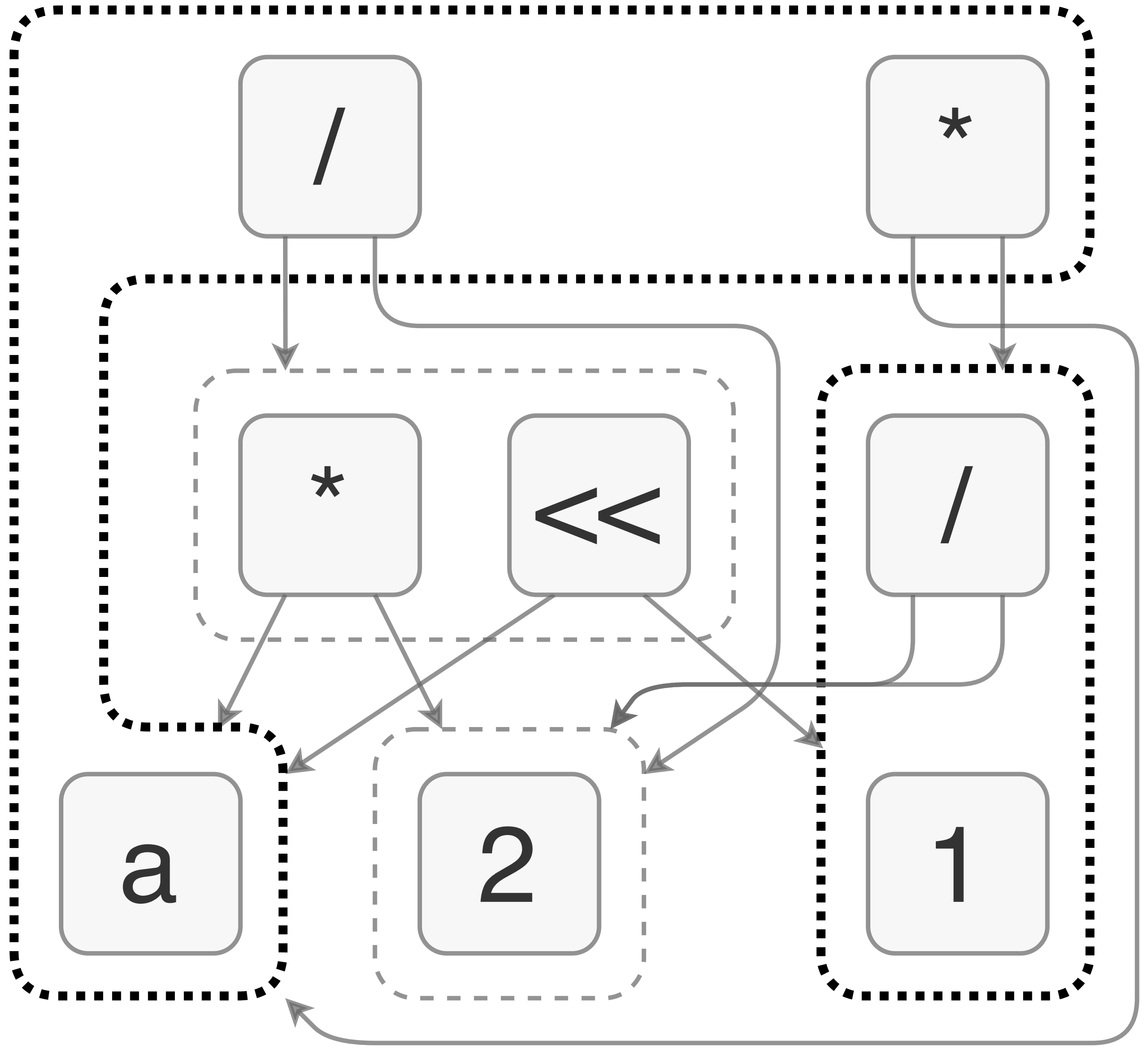
  (1) $x \times 2 \rightarrow x \ll 1$ (2) $(x \times y)/z \rightarrow x \times (y/z)$

  (3) $x/x \rightarrow 1$ (4) $1 \times x \rightarrow x$

- More rule application can't change the graph

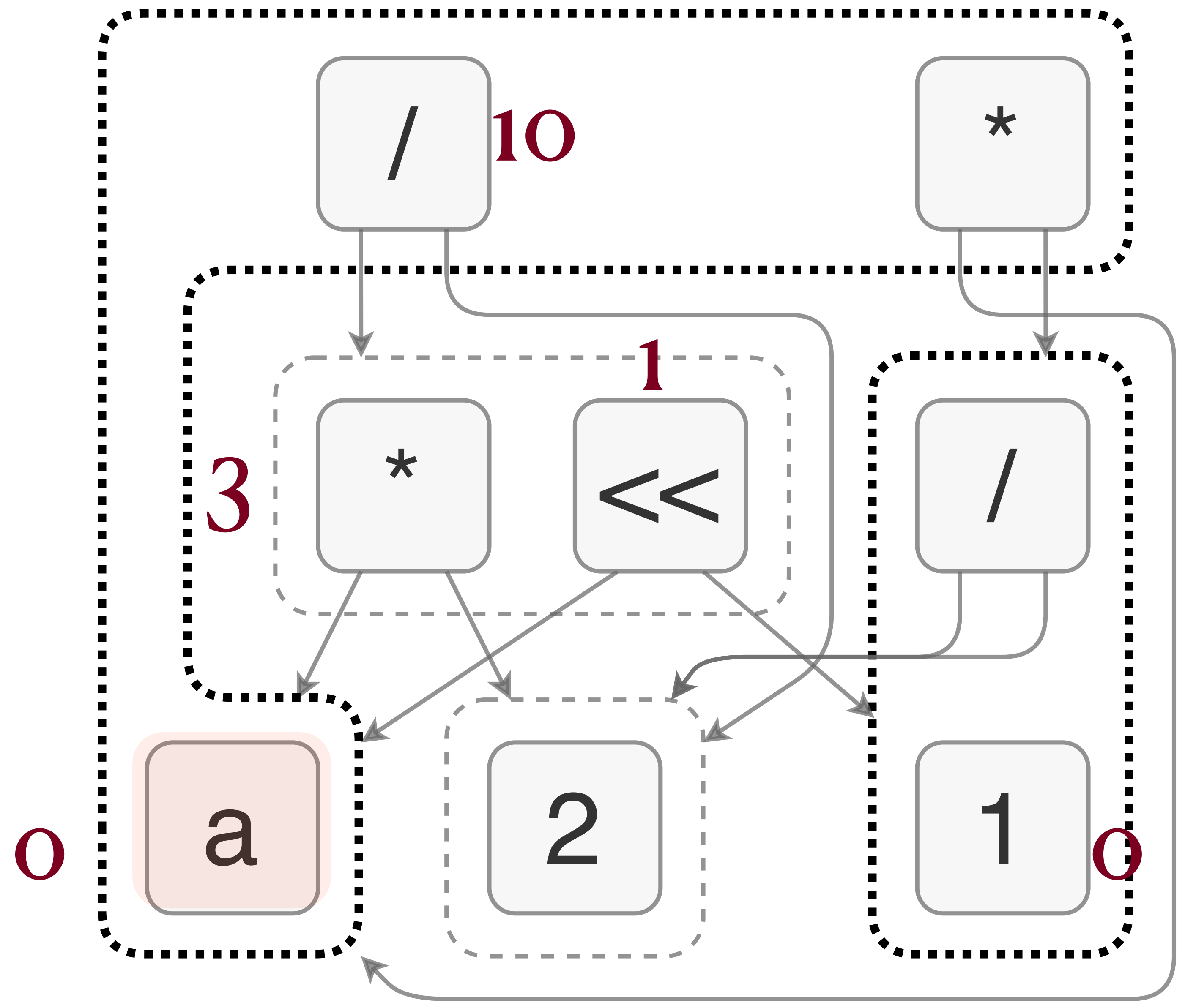
  — *Saturation!*

  - Exprs represented by the root node's e-class is *all* exprs obtainable by applying the rules in *all* possible orders



Extracting an Optimal Solution

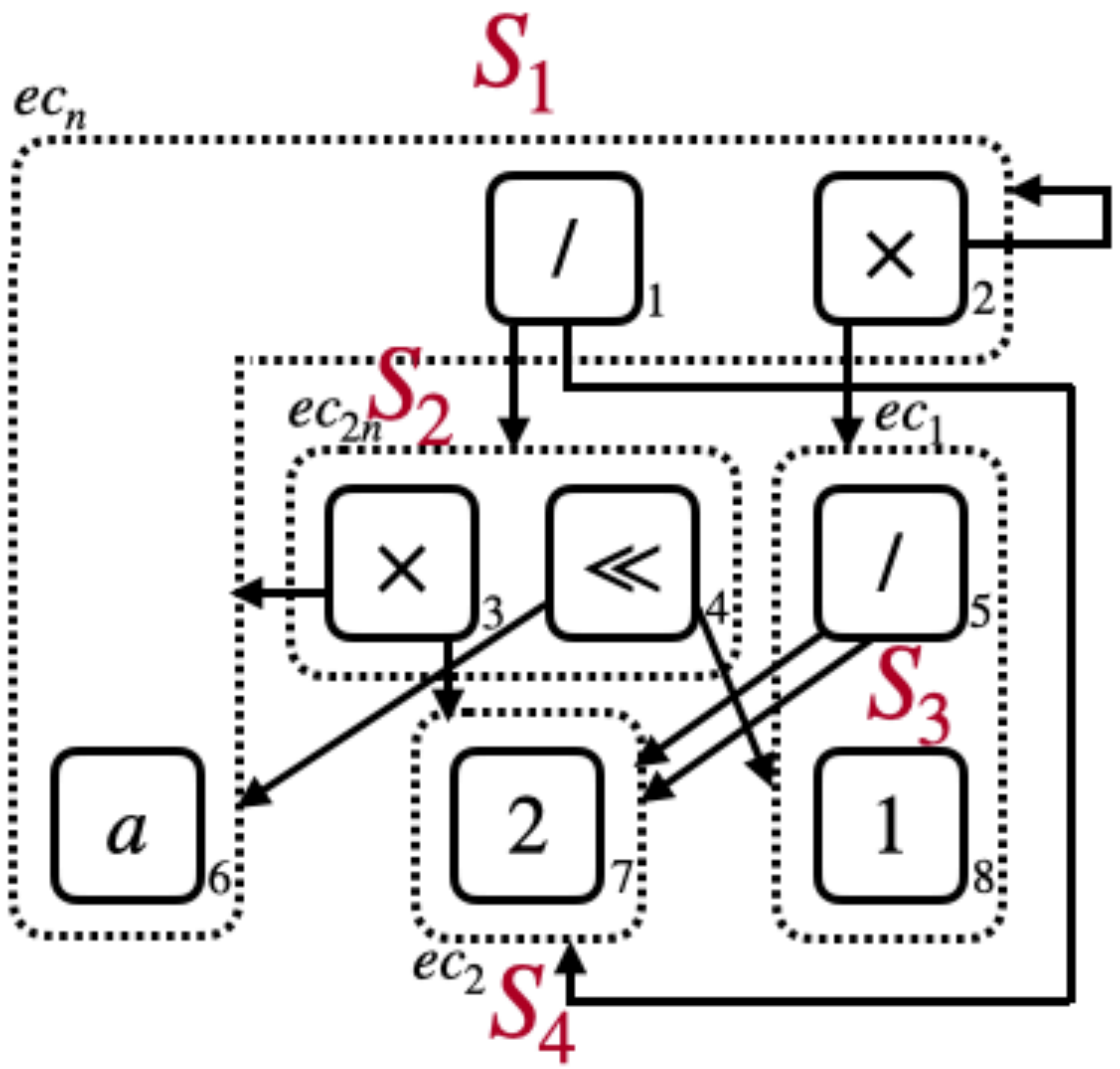
- Extract an expression of the best score after saturation†
  - e.g., greedy method using scores assigned for each kind of e-node
  - By integer linear programing in more complicated cases



† Termination is not always guaranteed

Fundamental Meaning of Equality Saturation

- E-graph ≅ Grammar representing semantically equivalent exprs
  - (E-class ≅ non-terminal, E-node ≅ production rule)
- Equality saturation = **grammar induction**



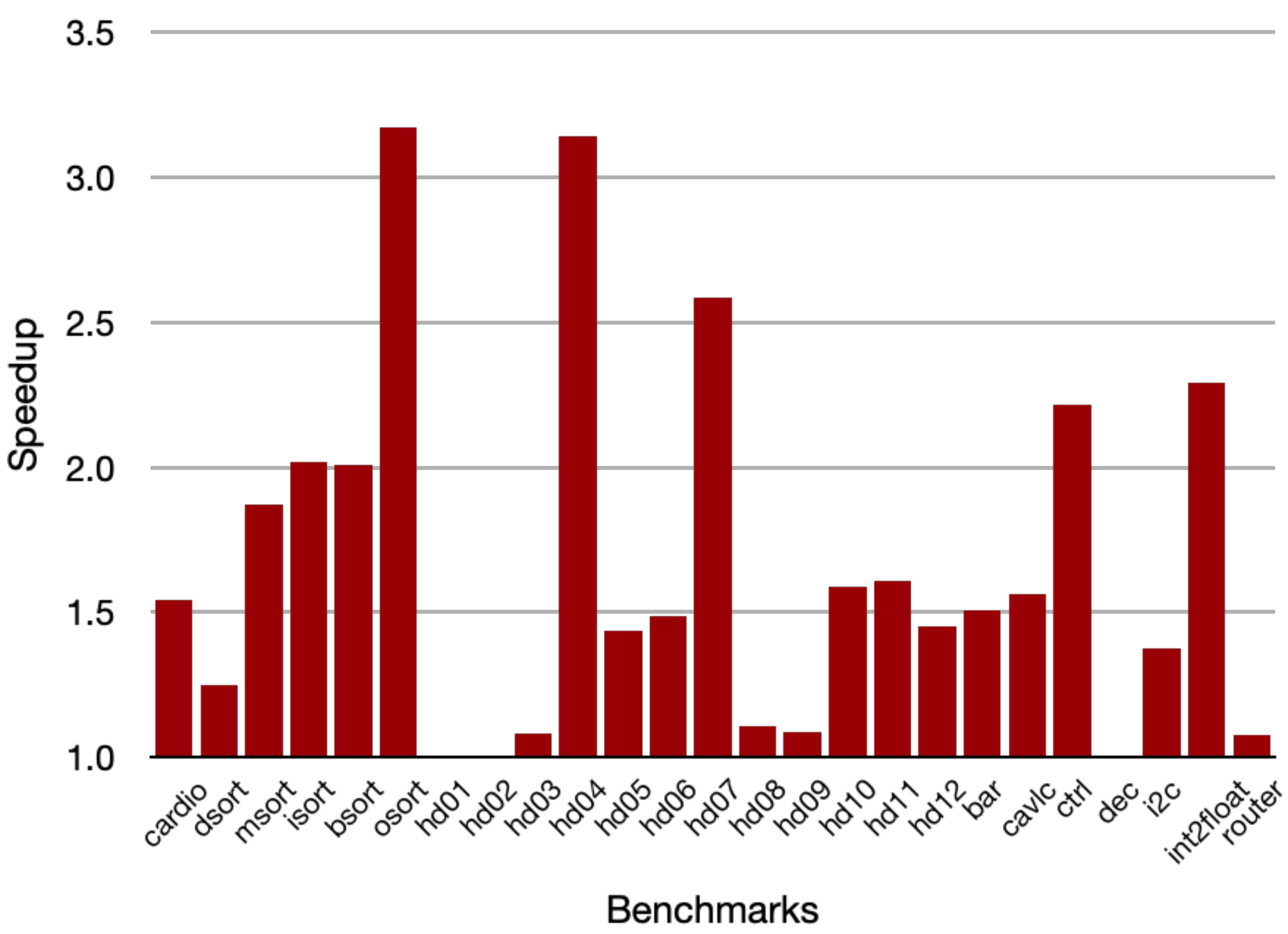
$$
\begin{aligned}
S_1 &\rightarrow S_2 \,/\, S_4 \\
&\;\;|\;\; S_1 * S_3 \\
&\;\;|\;\; a \\
S_2 &\rightarrow S_1 \ll S_3 \\
&\;\;|\;\; S_1 * S_4 \\
S_3 &\rightarrow S_4 \,/\, S_4 \\
&\;\;|\;\; 1 \\
S_4 &\rightarrow 2
\end{aligned}
$$

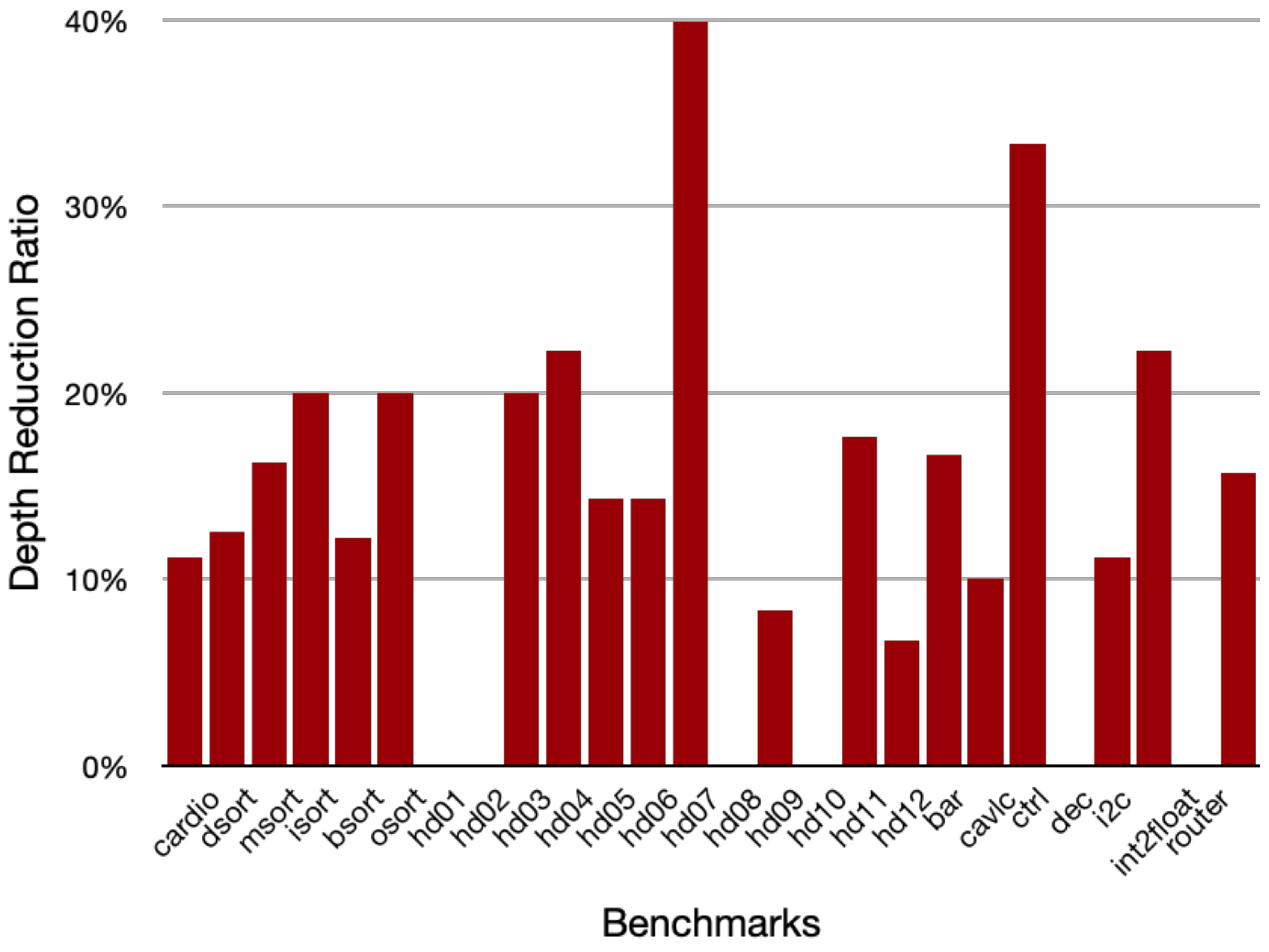
Evaluation

- 25 HE algorithms from 4 sources
  - Cingulata benchmarks
  - Sorting benchmarks
  - Hackers Delight benchmarks
  - EPFL benchmarks
- Baseline tool: Cingulata
  - A HE compiler using optimization rules written by domain experts

Lobster Performance

- Achieved an average of 2x, up to 3.1x faster performance compared to Cingulata (with up to a 40% reduction in multiplication depth)

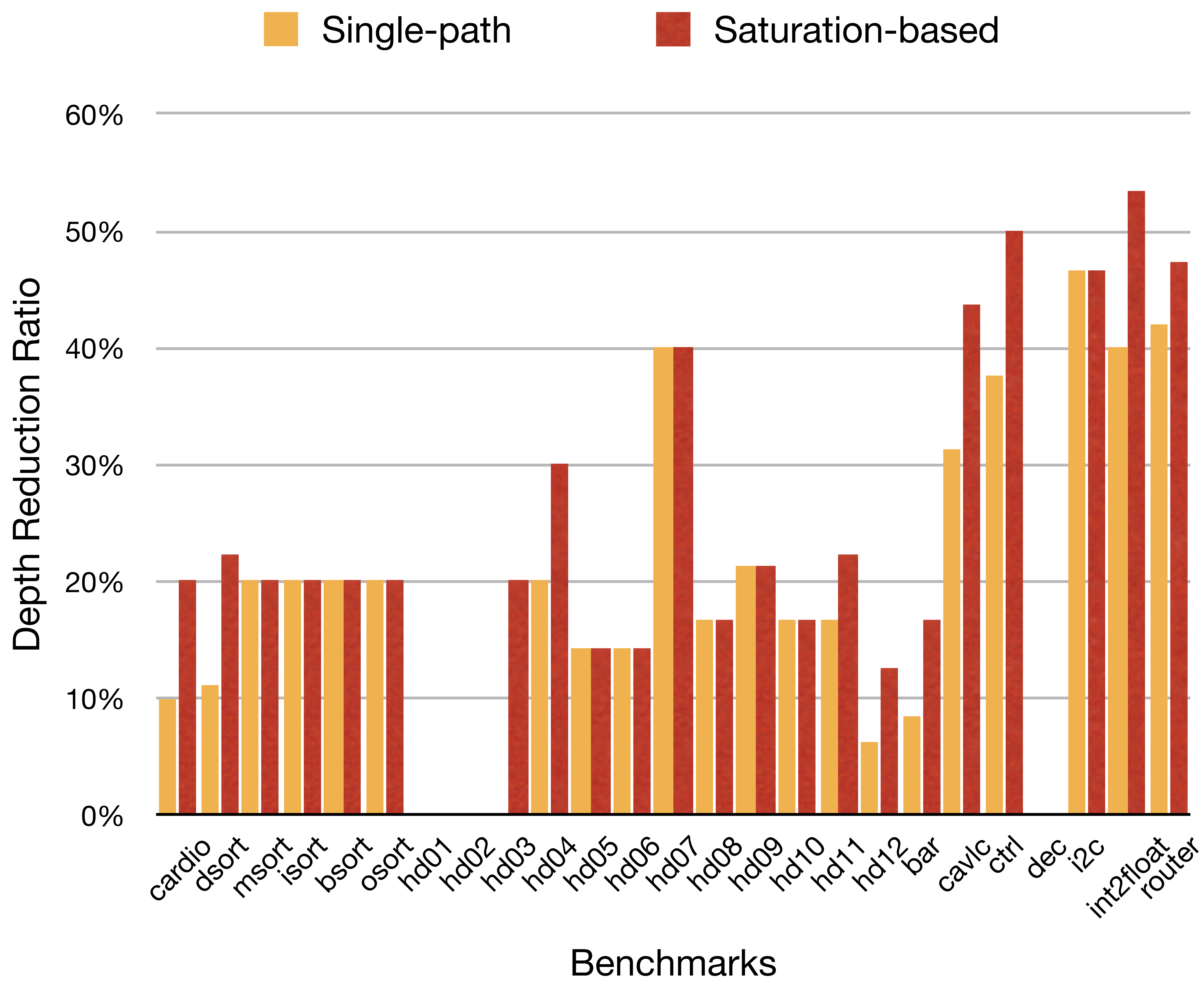




Efficacy of Equality Saturation

Success rate↑ : 19 → 22 in the number of successfully optimized programs

- Execution time: x2.03 → x2.26
- Reduction in multiplicative depth: 21.9% → 25.1%



Contents

- Case 1 : Optimizing compiler for homomorphic encryption

- **Case 2: Deobfuscation of bit-manipulating code**
  - Jaehyung Lee and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting*, **ACM CCS 2023**
  - Jaehyung Lee, Seoksu Lee, Eunsun Cho and Woosuk Lee, *Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Equality Saturation*, **IEEE TDSC (Submitted)**

- Lessons from the two cases

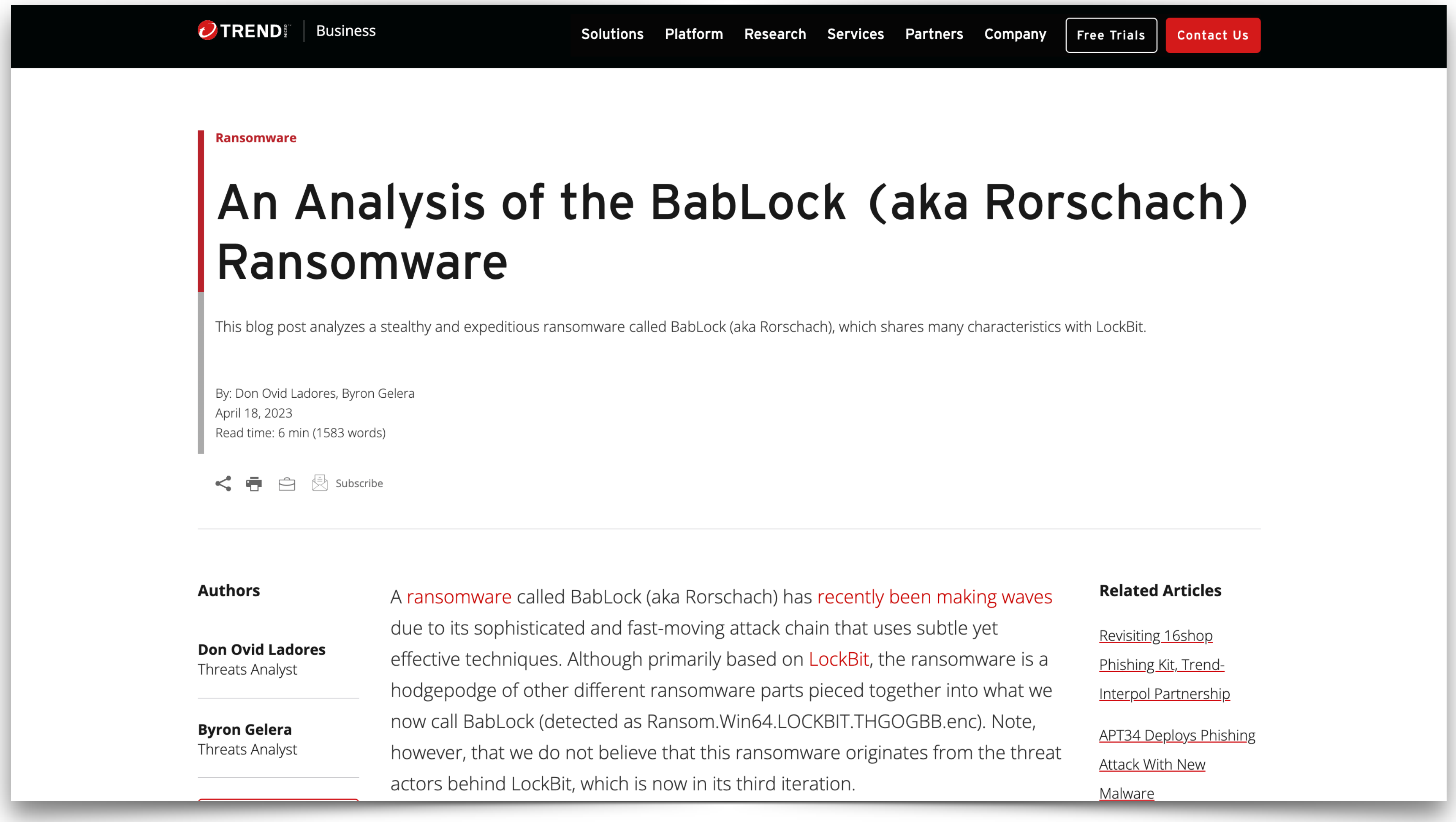
- Core technology: high-performance program synthesis

Mixed Boolean Arithmetic (MBA)

- Program expressions with logical operators (AND, OR, XOR…) and bitwise arithmetic operators (+,-,*,/,%,…)
  - e.g., $8458(x \vee y \wedge z)^3\,((xy) \wedge x \vee t) + x + 9(x \vee y)yz^3$
- MBA obfuscation: transforming arbitrary bitwise expressions into highly complex MBA expressions while maintaining their meaning

Popular MBA Obfuscation

- The cost of obfuscation and executing obfuscated code is low.
  - Only basic operations are added, and the execution flow remains unchanged (no additional calls to user/system functions, etc.)
- Theoretical foundation: any bitwise expressions can be obfuscated in infinitely many ways. Deobfuscation is NP-hard
- Widely adopted by various tools
  - Code obfuscation(Tigress, VMProtect)
  - DRM(Irdeto)
  - Being used for malware



Previous Approaches for MBA Deobfuscation

- **Term Rewriting** : SSPAM [Eyrolles et al. 2016]

Cannot handle a wide range of MBA obfuscation rules

- **Program Synthesis** : Syntia [Blazytko et al. 2017], QSynth [David et al. 2020], Xyntia [Menguy et al. 2021]

No Guarantee of Correctness

- **Neural Network Inference** : NeuReduce [Feng et al. 2020]

Unscalable

- **Algebraic Methods** : MBA-Solver [Xu et al. 2021], SiMBA [Reichenwallner et al. 2022]

Limited to a specific class of MBA expressions

Our Goal

- Term Rewriting : “Rewrite with fi…

To overcome the limitations, we should achieve :

- **Soundness** : Guarantee of correctness
- **Generality** : Covers arbitrary MBA expression
- **Flexibility** : Regardless of obfuscation rules
- **Scalability** : Covers huge MBA expression

  - **MBA-Solver** [Xu et al. 2021], **SiMBA** [Reichenwallner et al. 2022]

Limited to specific set of transformations (linear)

Program Synthesis-Based MBA Deobfuscation

Constraints + **Grammar** —**Synthesize**→ **Result**

(((((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) + (((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a)))))) - (((~ e) + 1) ^ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) …

$$
\begin{aligned}
S &\rightarrow\ !S \mid S \oplus S \mid S \wedge S \\
&\mid\ S \vee S \mid -S \mid S + S \\
&\mid\ S \times S \mid S - S \mid S >> S \\
&\mid\ S << S \mid V \mid C \\
V &\rightarrow \texttt{b} \mid \texttt{e} \mid \cdots \\
C &\rightarrow \texttt{0x00} \mid \texttt{0x01} \mid \cdots
\end{aligned}
$$

1 + a

Same semantics **Syntax** **Deobfuscated expression**

Can cover arbitrary MBA Exprs

Challenge : Scalability

(((((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) + (((~ e) + 1) & (~ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a)))))) - (((~ e) + 1) ^ ((((- e) - 1) | (- a)) + (((- e) - 1) & (- a))))) …

Synthesize → 1 + a

Too Slow! 😡

size of obfuscated expression ⬆ -> deobfuscation performance ⬇

Solution 1: Synthesis via Localization

- Sub-expressions are chosen for replacement

((((- e) & (~ (((~ e) | (- a)) + ((~ e) & (- a))))) +
((- e) & (~ (((~ e) | (- a)) + ((~ e) & (- a)))))) …
(v1 & ((v2 * v3) & (v2 - v1)))
…

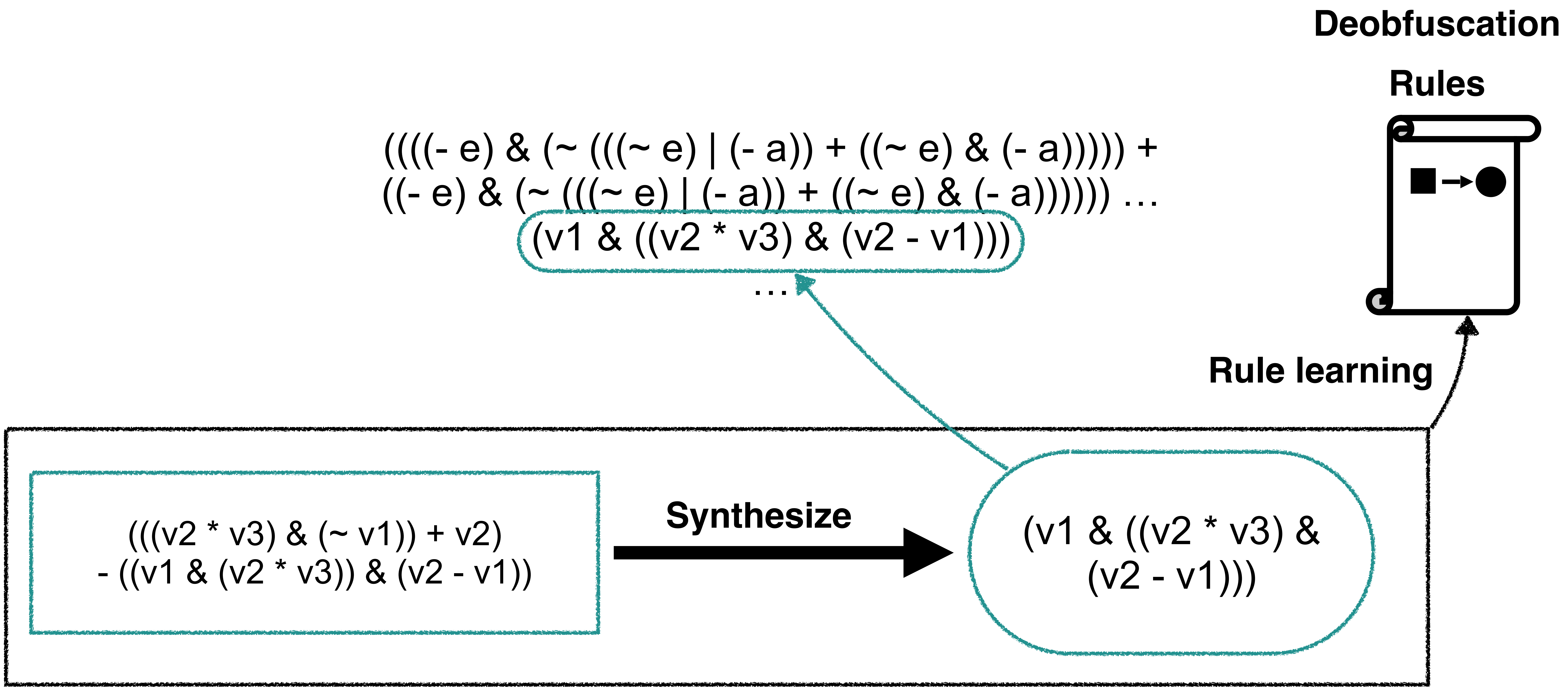
(((v2 * v3) & (~ v1)) + v2)
- ((v1 & (v2 * v3)) & (v2 - v1))

Synthesize

(v1 & ((v2 * v3) &
(v2 - v1)))

Scalable 🙂

Solution 2: Learning Successful Synthesis Patterns



Comparison to the HE Optimization

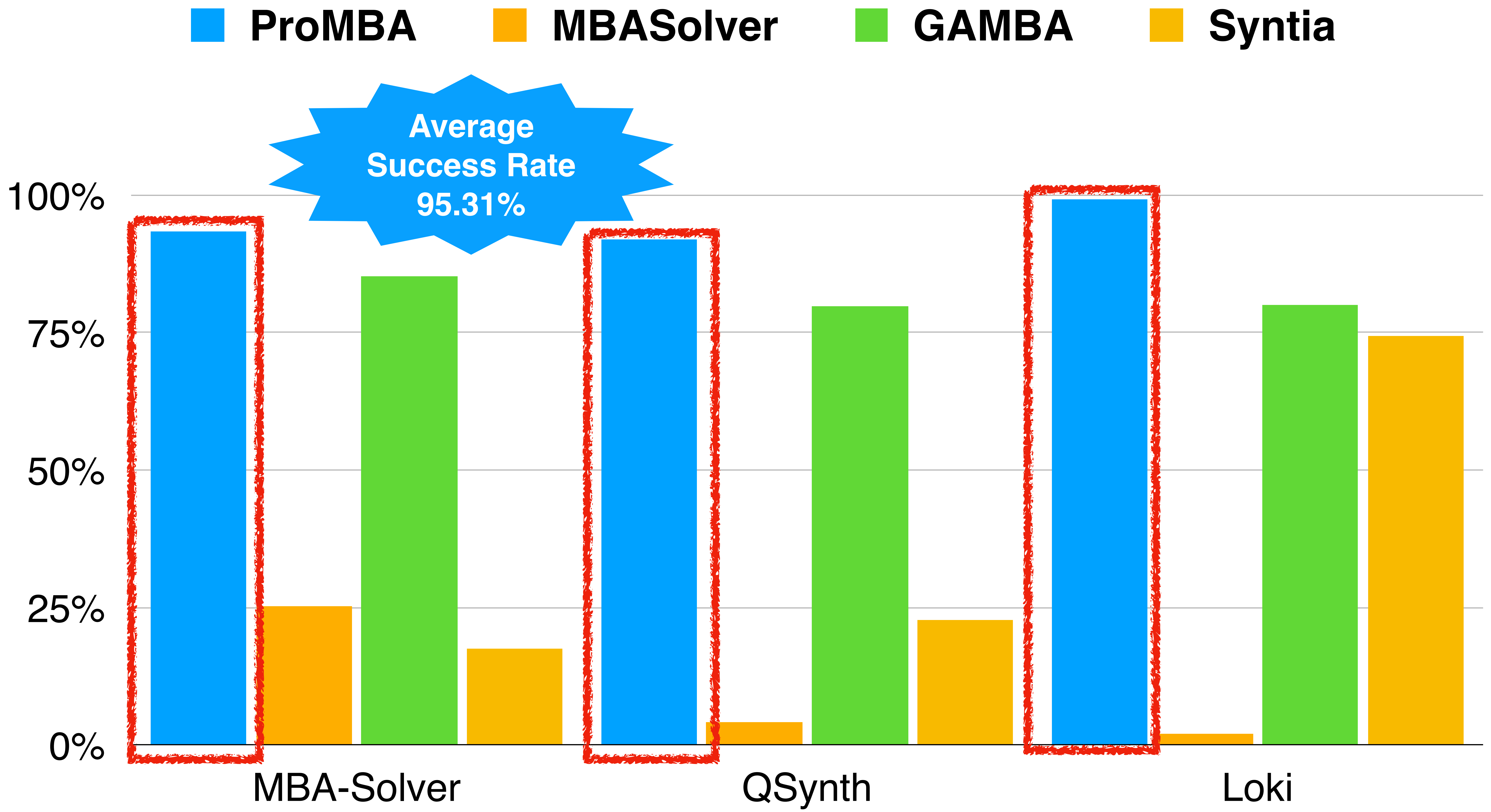
- **Commons** : Synthesis via localization → Learning rules → Term rewriting + Equality saturation
- **Major Diffs** : Learning and applying rules directly online (without offline learning)
  - Rules used in MBA obfuscation are highly diverse, making offline-learned rules ineffective for deobfuscating new MBA expressions.
  - Advances in program synthesis have enabled faster rule synthesis.
- **Others**: using algebraic methods for certain types of MBA expressions (linear MBA), selecting target subexpressions for replacement, etc.

Evaluation

- ProMBA: performs term rewriting first and then equality saturation
- 4000 MBA obfuscated expressions from prior work
  - From three categories with different sizes, from small to large
- Baseline tools
  - MBASolver [PLDI '22] : Algebraic method. Correctness guarantee
  - Syntia [USENIX '17] : Heuristics. No correctness guarantee
  - GAMBA [WORMA'23] : Algebraic + heuristics. Correctness guarantee
- Success: (1) size of deobfuscated result $\leq$ size of original expression, (2) deobfuscated result has the same meaning as the original one

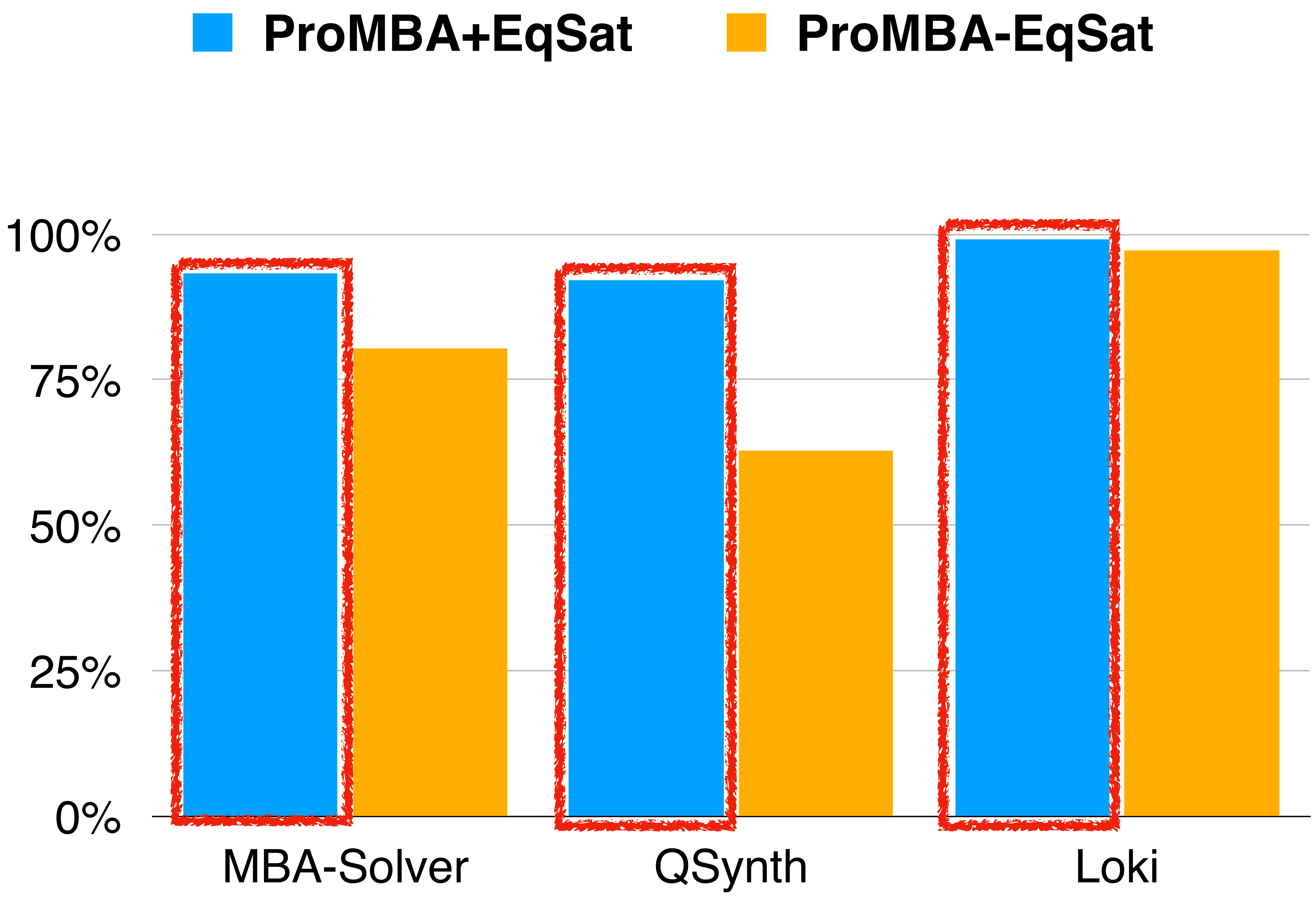
Results

- An average deobfuscation success rate of 95.3%, significantly outperforming other tools (13%, 82.5%, 39.4%)



Efficacy of Equality Saturation

- Increased success rate: 84% → 95%
- Reduced average size of deobfuscated results : 9.4 → 7.9 (in AST nodes)



Contents

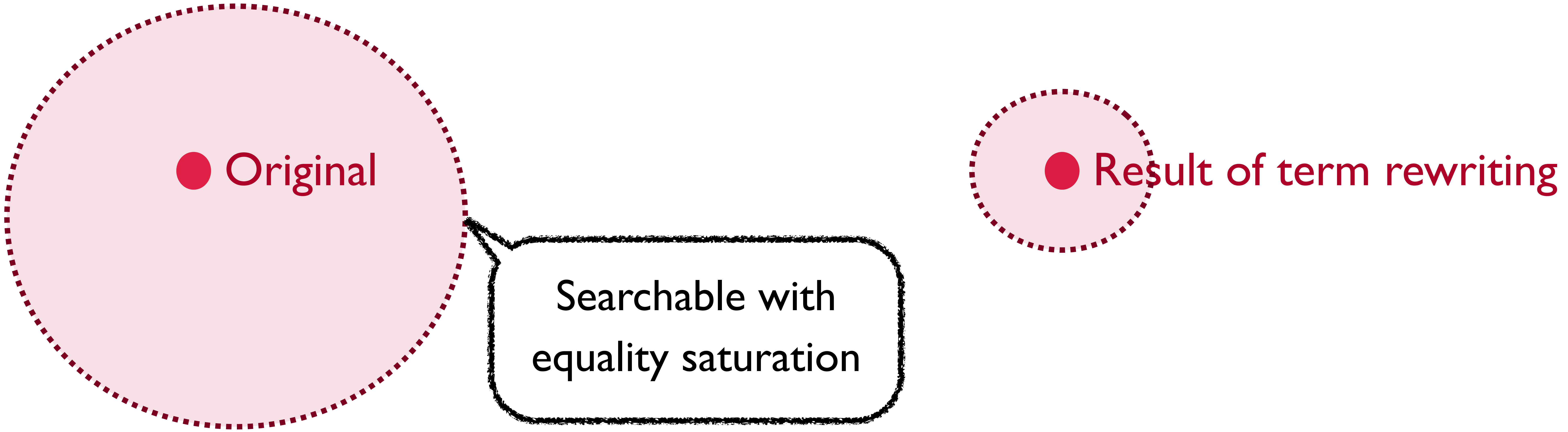
- Case 1 : Optimizing compiler for homomorphic encryption
- Case 2: Deobfuscation of bit-manipulating code
- **Lessons from the two cases**
- Core technology: high-performance program synthesis

Lesson 1 : Both Term Rewriting and Equality Saturation are Necessary (1/2)

- Equality saturation is for overcoming the phase-ordering problem, the limitation of term rewriting. But, equality saturation alone is insufficient.
- The main issue is its high computational cost.
- In the case of homomorphic encryption optimization
    - EqSat alone causes OOM (256GB) for large circuits (depths > 25)
    - Even smaller circuits may not reach saturation within 12 hours
- For MBA deobfuscation — lower success rate (92% → 61.8%) when with equality saturation alone (with early termination to avoid high cost)

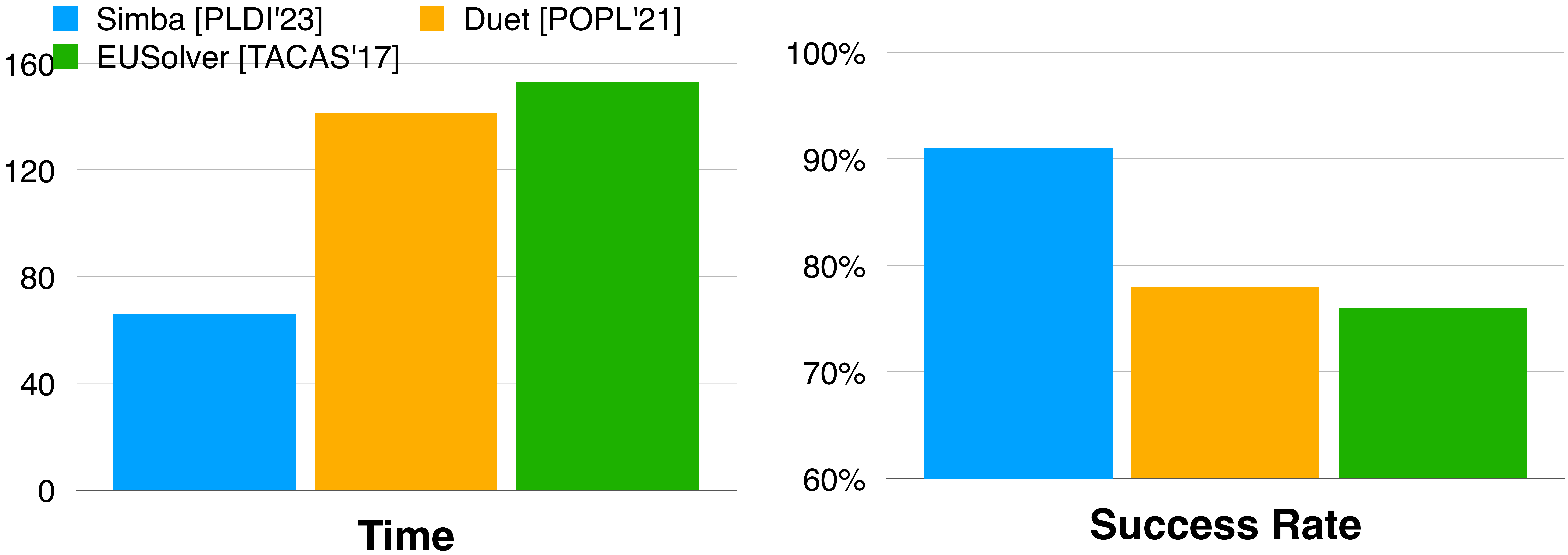
Lesson 1 : Both Term Rewriting and Equality Saturation are Necessary (2/2)

- The larger the original expression and the more rules there are, the greater the search space and computational cost
- It is beneficial to do term rewriting first and then equality saturation.
  - Reducing the size of the original expression decreases the search space and provides direction to the exploration process.



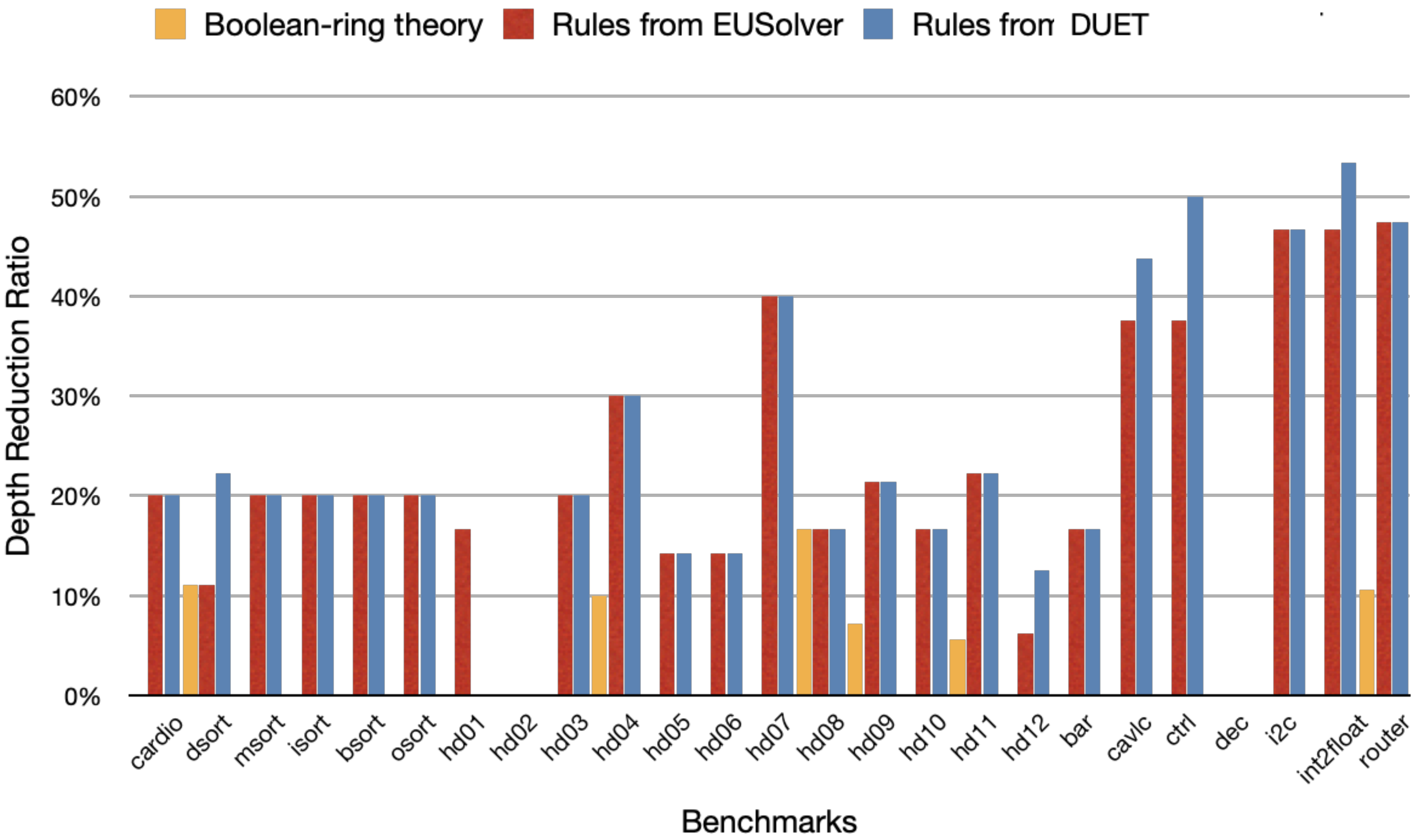
Lesson 2 : Performance of Synthesis is key (1/2)

- Rules discovered by a better synthesizer lead to better optimization
  - Performance : Simba > Duet > EUSolver
- Case of MBA



Lesson 2 : Performance of Synthesis is key (2/2)

- Case of HE optimization



Contents

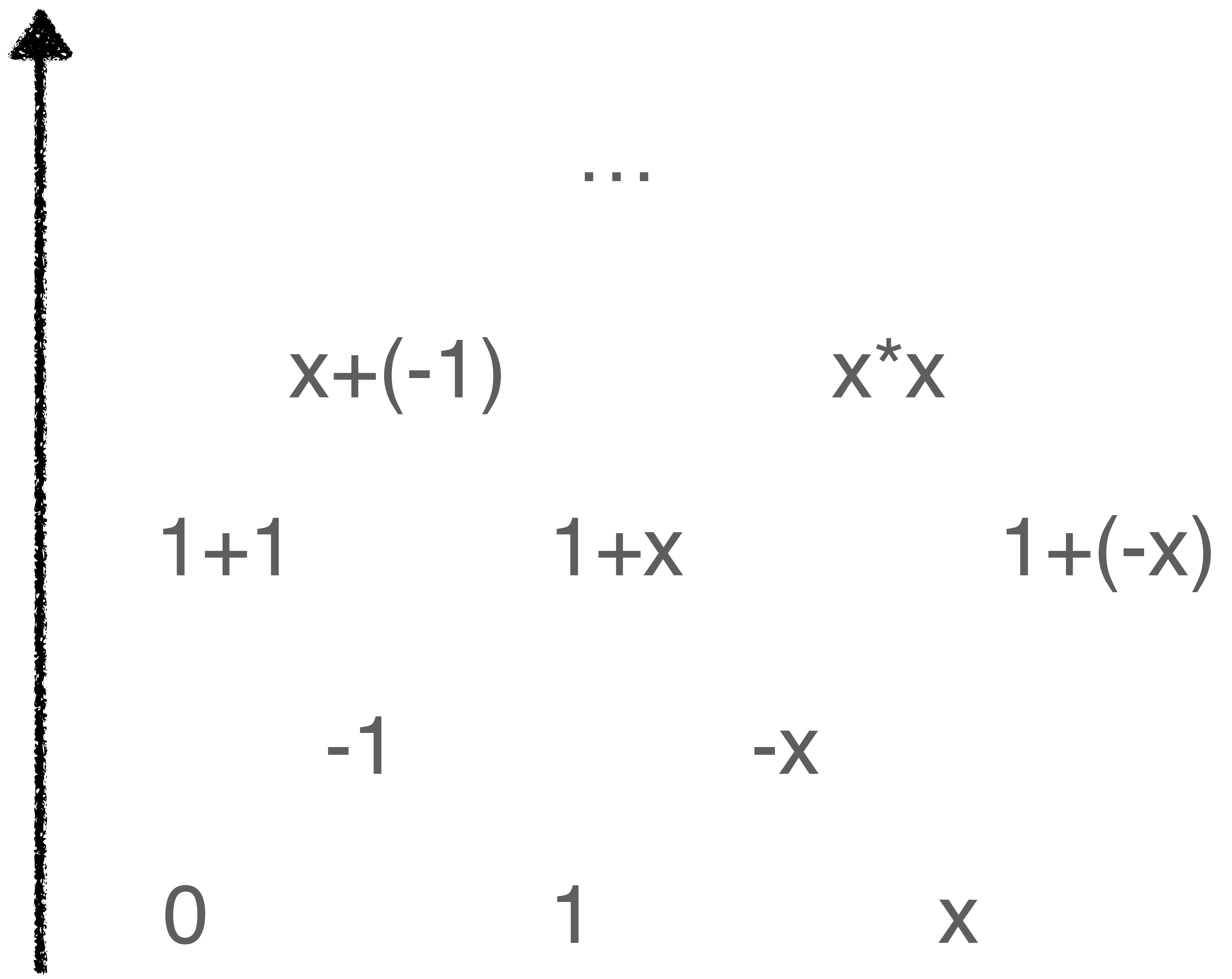
- Case 1 : Optimizing compiler for homomorphic encryption

- Case 2: Deobfuscation of bit-manipulating code

- Lessons from the two cases

- **Core technology: high-performance program synthesis**
    - Yongho Yoon, Woosuk Lee, and Kwangkeun Yi, Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. **ACM PLDI 2023**

Two Synthesis Strategies — Bottom-Up



$$
\begin{aligned}
E \quad \rightarrow \quad & 0 \\
| \quad & 1 \\
| \quad & x \\
| \quad & E + E \\
| \quad & E * E \\
| \quad & -E
\end{aligned}
$$

Two Synthesis Strategies — Top-Down

0 1 x E+E E*E -E

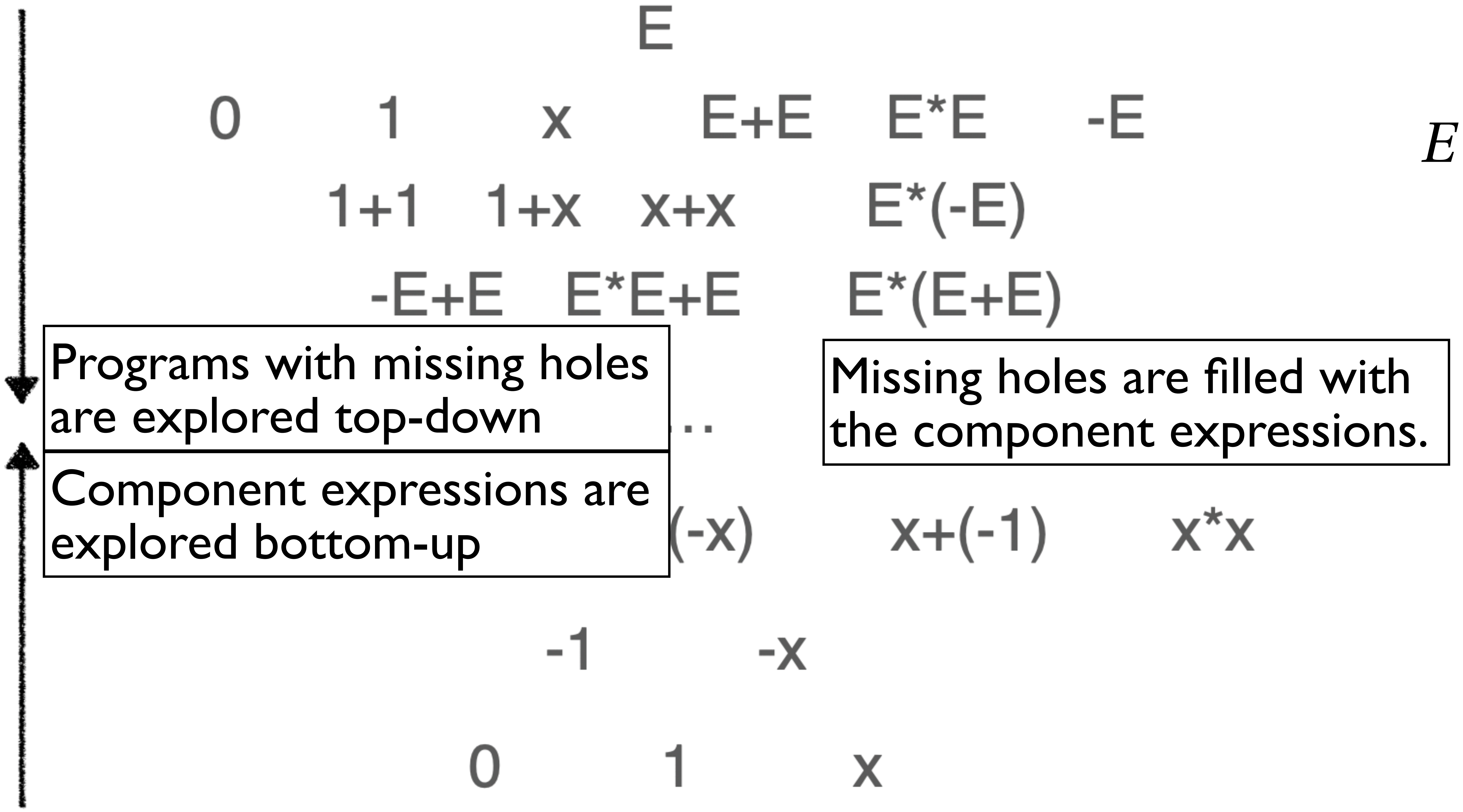
1+1 1+x x+x E*(-E)

-E+E E*E+E E*(E+E)

…

$$
\begin{aligned}
E \quad \rightarrow \quad & 0 \\
| \quad & 1 \\
| \quad & x \\
| \quad & E + E \\
| \quad & E * E \\
| \quad & -E
\end{aligned}
$$

Bidirectional Synthesis



$$
\begin{array}{rcl}
E & \rightarrow & 0 \\
 & | & 1 \\
 & | & x \\
 & | & E + E \\
 & | & E * E \\
 & | & -E
\end{array}
$$

*(POPL'21) Woosuk Lee, "Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis"

Synthesis + Static Analysis

- Can prune infeasible program candidates
  - "Infeasible" = partial program that can never satisfy the given spec no matter how we fill in the holes
  - The more component expressions, the higher impact of the pruning
- "**Fairly precise"** static analysis for pruning infeasible candidates
  - Input: spec(input-output examples) and an incomplete partial program
  - Output: "May be feasible" or "Infeasible"

Example

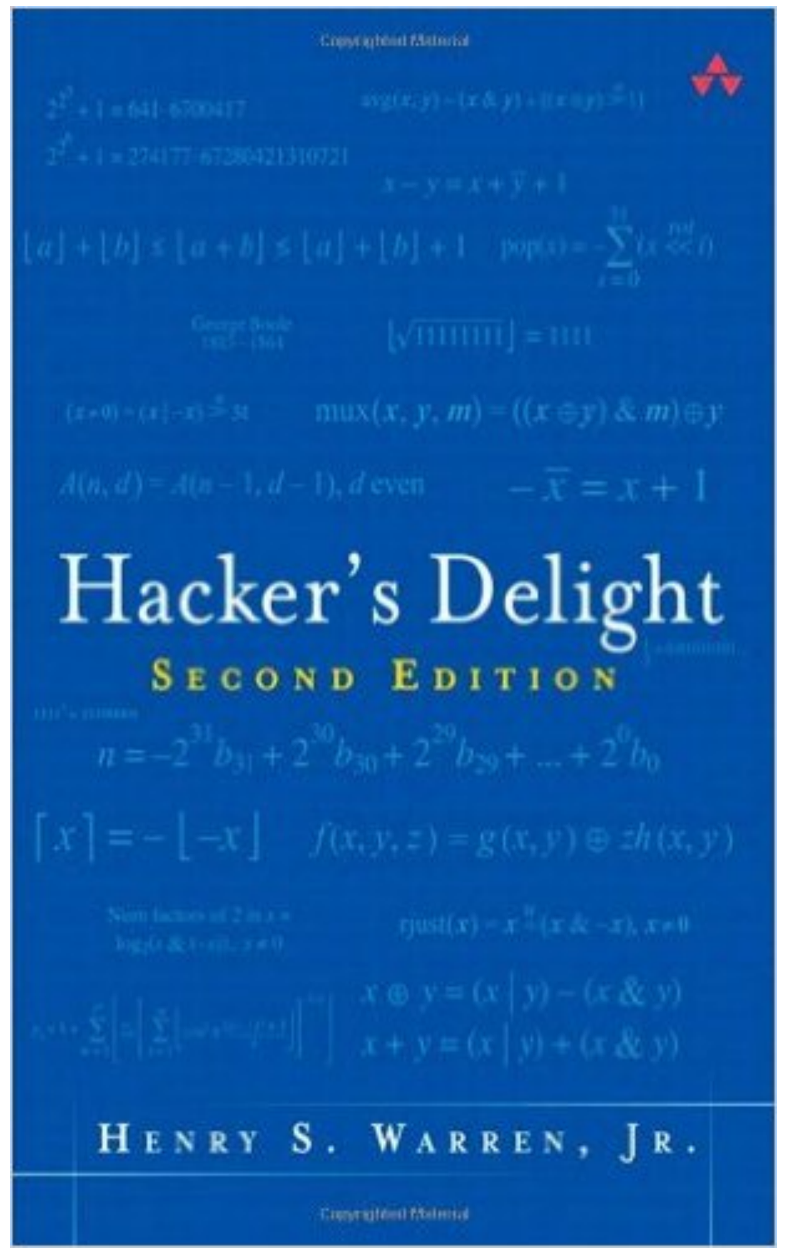
- Goal: turn off all bits from the first bit to the rightmost 0 from a given bitvector
- Target function f(x: BitVec) : BitVec.
- Syntactic constraint:

$$
\begin{array}{lll}
S & \rightarrow & x \mid 0001_2 \\
 & \mid & S \wedge S \mid S \vee S \mid S \oplus S \\
 & \mid & S + S \mid S \times S \mid S/S \mid S >> S \\
 & & \ldots
\end{array}
$$

input bit-vector and bit-vector literals

bitwise logical binary operators

bitwise arithmetic binary operators



- Semantic constraint : $f(1011_2) = 0011_2$
- Solution: $f(x) = ((x + 0001_2) \oplus x) >> 0001_2$

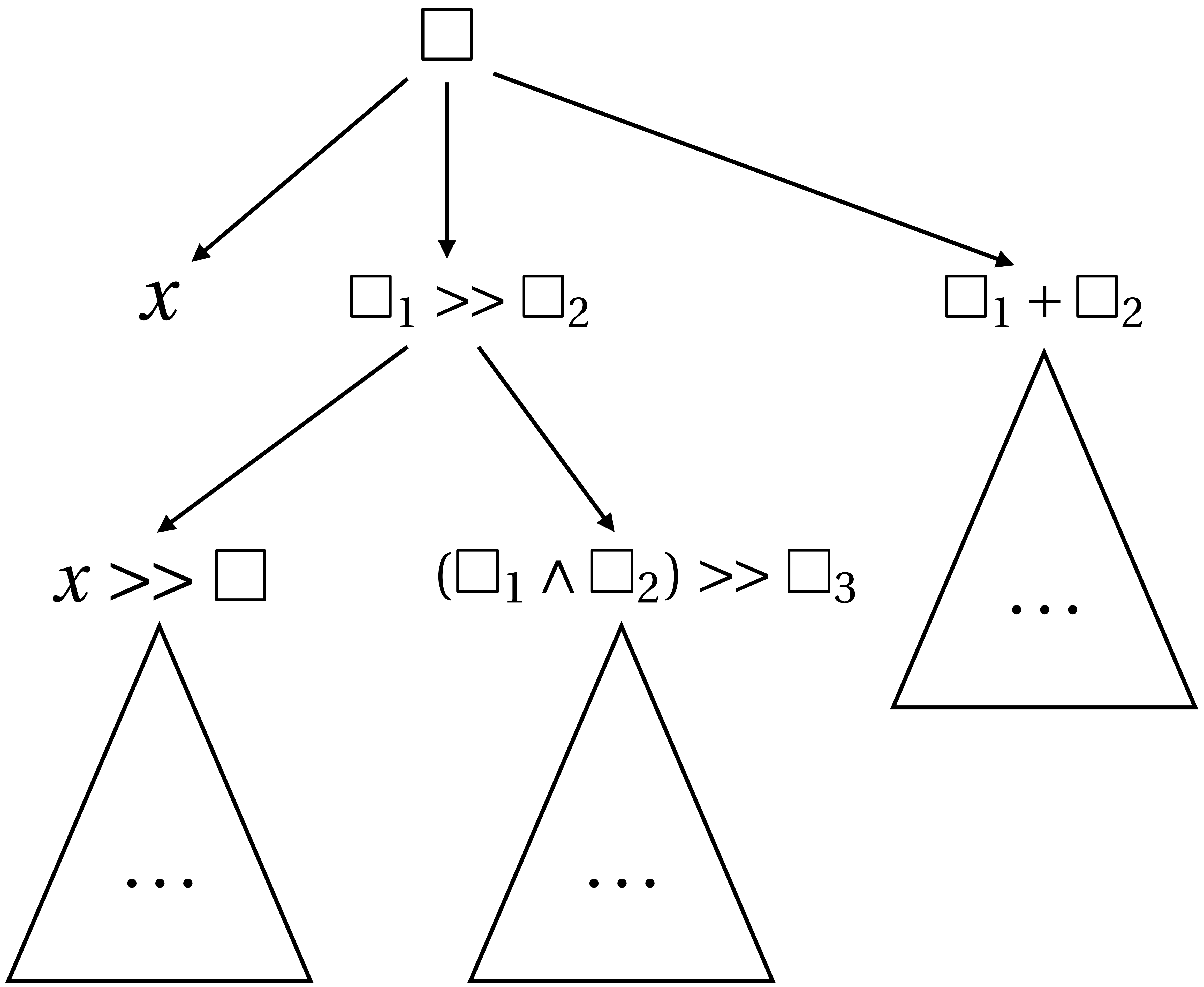
Example

Syntactic Spec

$$
\begin{array}{lll}
S & \rightarrow & x \mid 0001_2 \\
 & \mid & S \wedge S \mid S \vee S \mid S \oplus S \\
 & \mid & S + S \mid S \times S \mid S / S \mid S >> S
\end{array}
$$

Semantic Spec

$$f(1011_2) = 0011_2$$



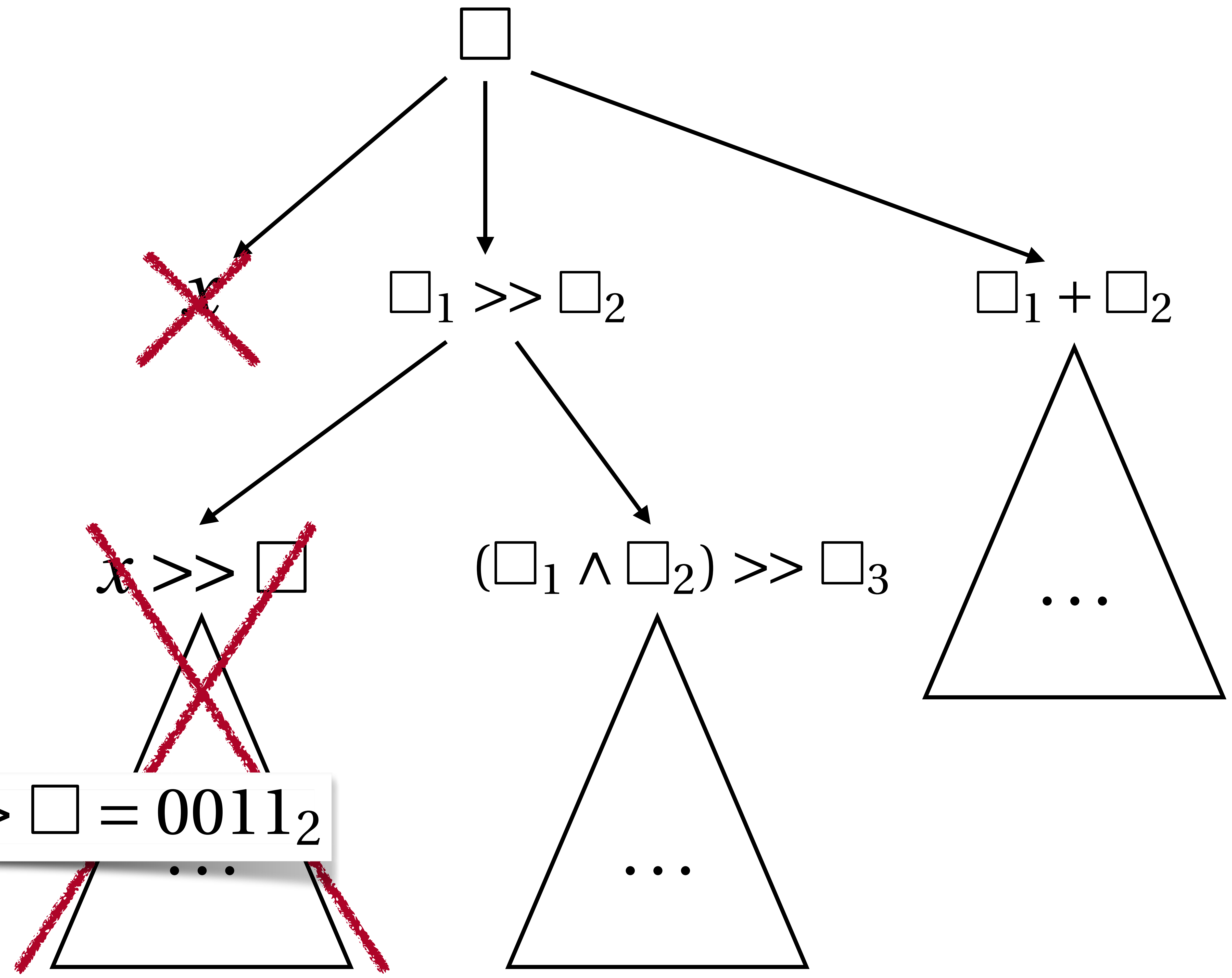
Example

Syntactic Spec

$$
\begin{aligned}
S \quad \rightarrow \quad & x \mid 0001_2 \\
\mid \quad & S \wedge S \mid S \vee S \mid S \oplus S \\
\mid \quad & S + S \mid S \times S \mid S / S \mid S >> S
\end{aligned}
$$

Semantic Spec

$f(1011_2) = 0011_2$



$$\nexists \square . f(1011_2) = 1011_2 >> \square = 0011_2$$

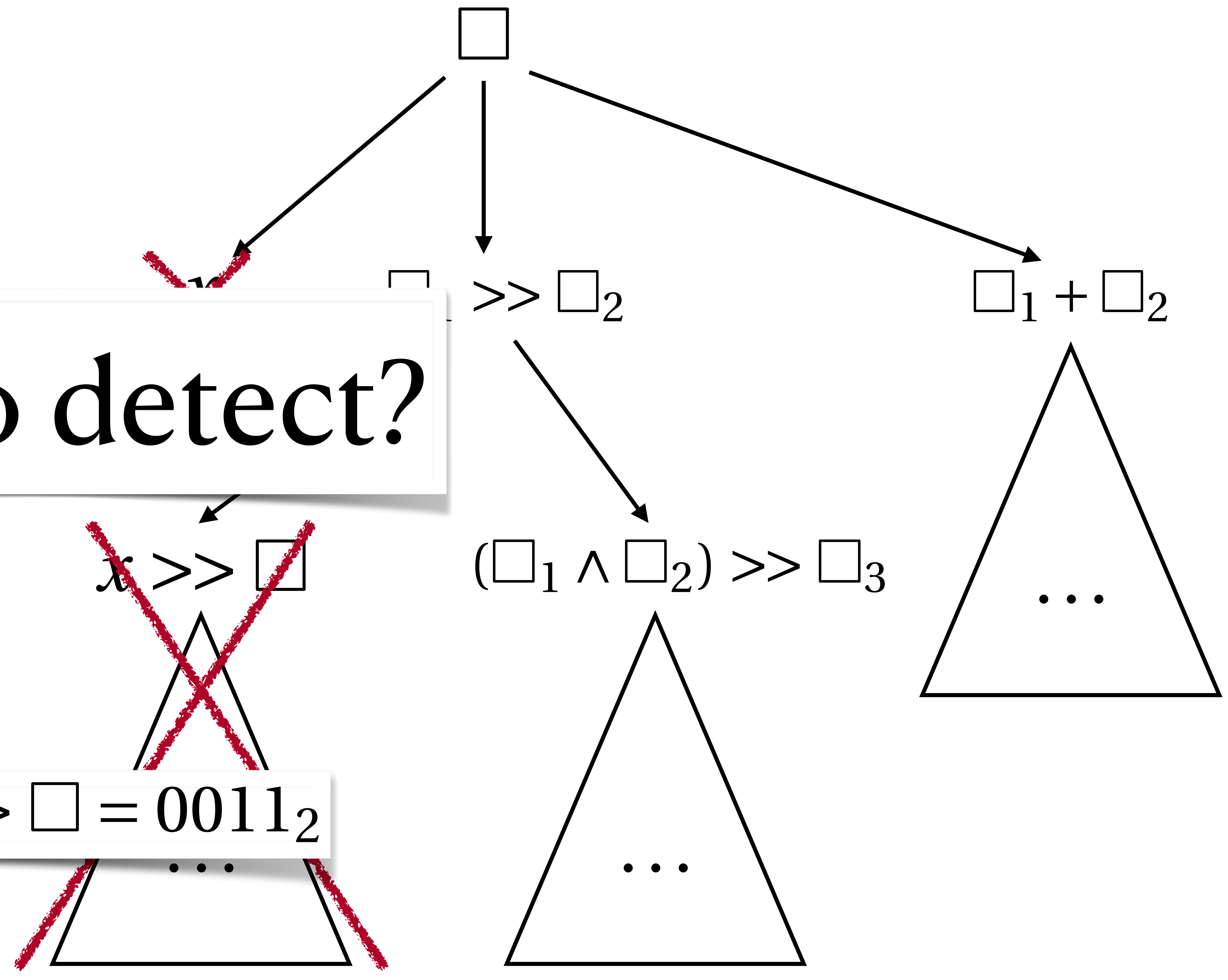
Example

Syntactic Spec

$$
\begin{array}{rcl}
S & \rightarrow & x \mid 0001_2 \\
 & \mid & S \land S \mid S \lor S \mid S \oplus S \\
 & \mid & S + S \mid S \times S \mid S / S \mid S >> S
\end{array}
$$

Semantic Spec

$$f(1011_2) = 0011_2$$

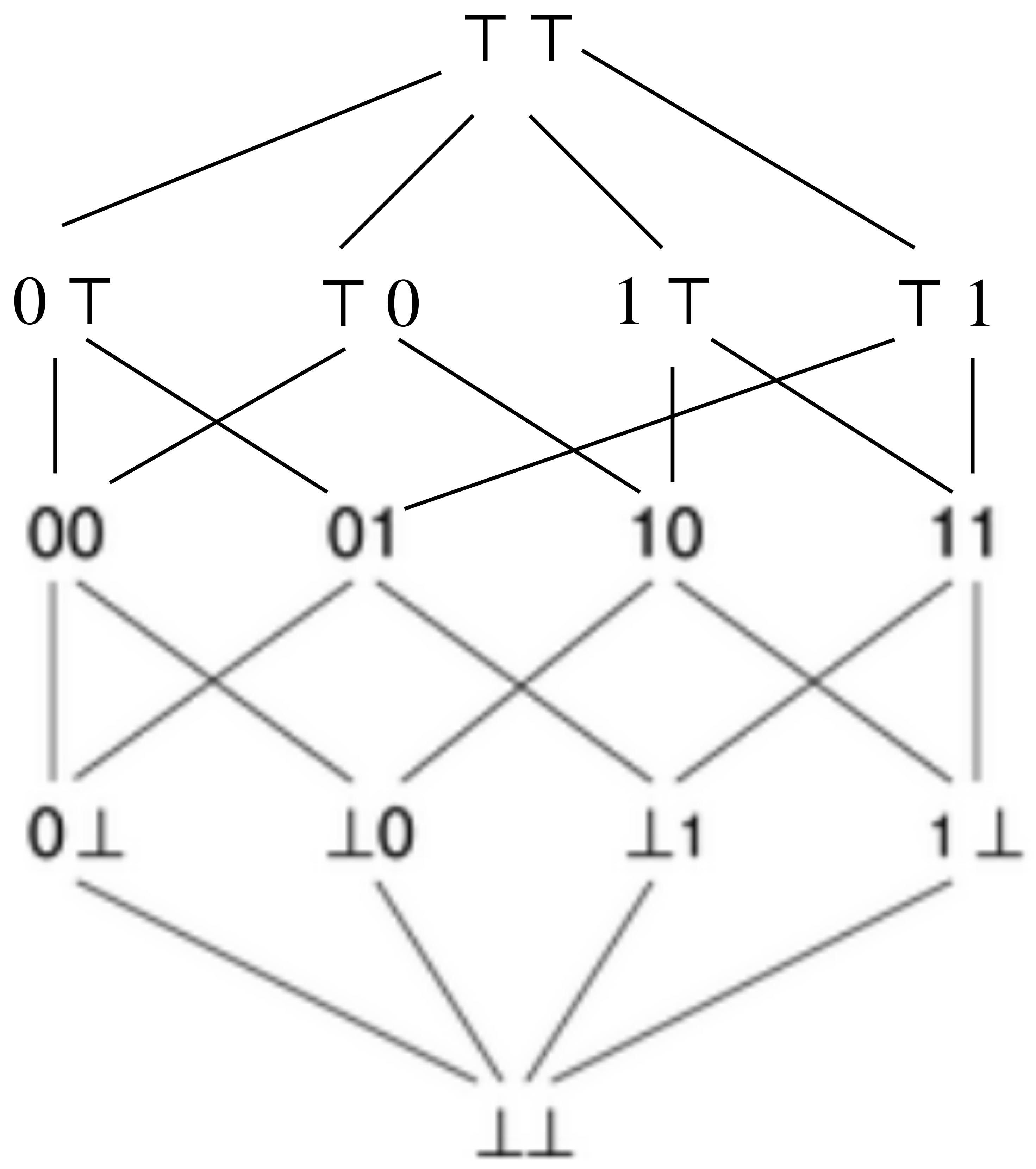


How to detect?

$$\nexists \Box . f(1011_2) = 1011_2 >> \Box = 0011_2$$

Static Analysis

- Simulation of program execution with “abstract" values instead of concrete ones
  - Abstraction = over-approximation (e.g., concrete : {0, 2, 6} → abstract : even)
- Bitfield abstract domain
  - Each bit is represented by $\{0, 1, \bot, \top\}$
  - $\top$: unknown, $\bot$: no value
- e.g., $\top 01 \top$ represents a set $\{0010_2, 0011_2, 1010_2, 1011_2\}$
- Abstract operators (denoted with #)
  - e.g., $1\top 10 \wedge^{\#} 00\top\top = 00\top 0$



Using Forward Analysis

Checking only output feasibility

Semantic Spec

$f(1011_2) = 0011_2$

Candidate Partial Program

$f(x) = x \lor \square$

Forward

$$
\begin{aligned}
x &\mapsto 1011 \\
\square &\mapsto \top\top\top\top
\end{aligned}
$$

Using Forward Analysis

Checking only output feasibility

Semantic Spec

$f(1011_2) = 0011_2$

Candidate Partial Program

$f(x) = x \vee \square$

Forward

$$\begin{aligned} x &\mapsto 1011 \\ \square &\mapsto \top\top\top\top \\ x \vee \square &\mapsto 1\top 11 \not\sqsupseteq 0011 \end{aligned}$$

Infeasible output

Limitation of Forward Analysis

Checking only output feasibility

Semantic Spec

$f(1011_2) = 0011_2$

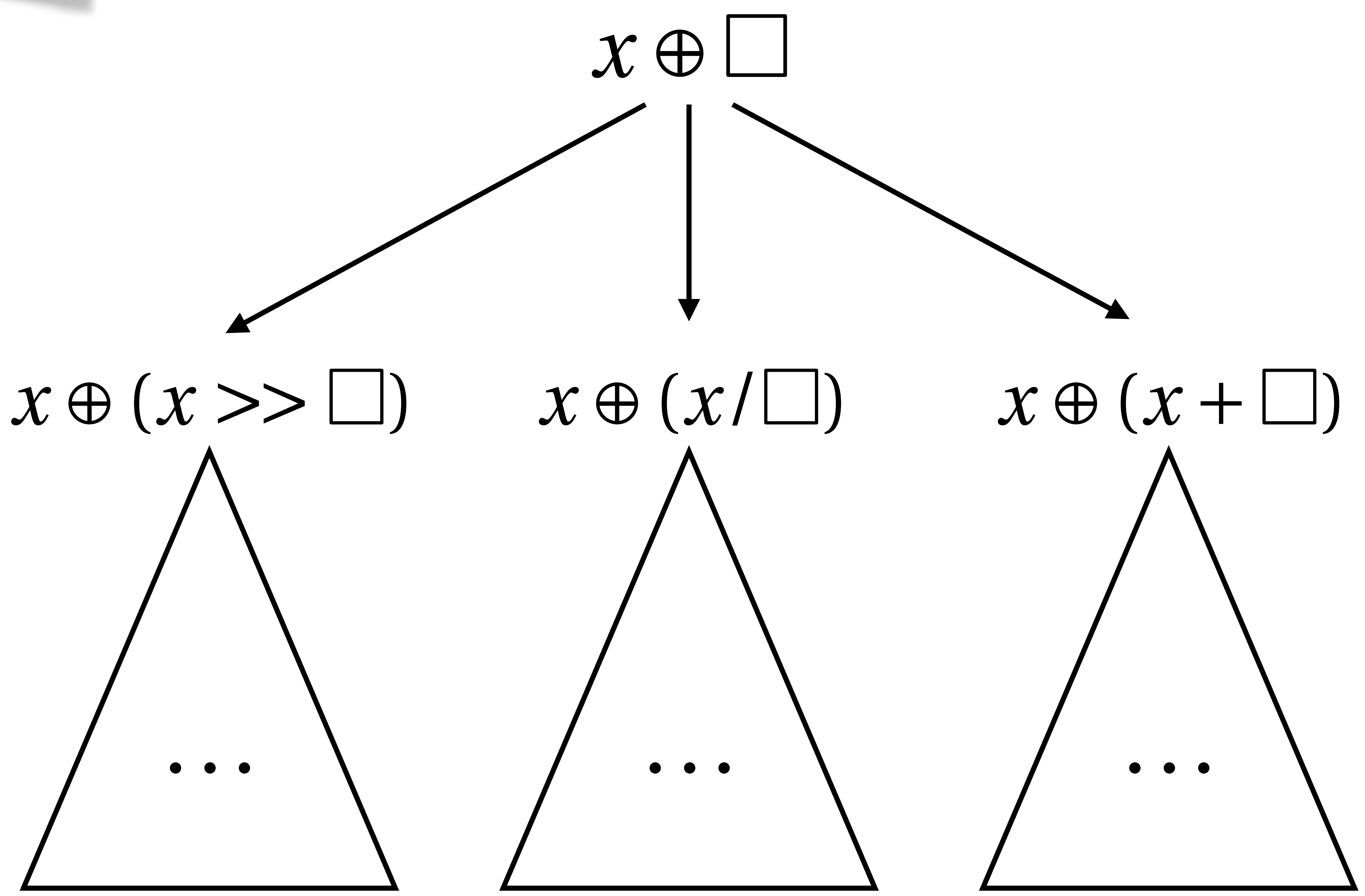
Candidate Partial Program

$f(x) = x \oplus \square$

Forward

$$
\begin{aligned}
x &\mapsto 1011 \\
\square &\mapsto \top\top\top\top \\
x \oplus \square &\mapsto \top\top\top\top \;\sqsupseteq 0011
\end{aligned}
$$

Nothing to do



Need: Backward Analysis Too

Output feasibility + Hole Precondition

Semantic Spec

$f(1011_2) = 0011_2$

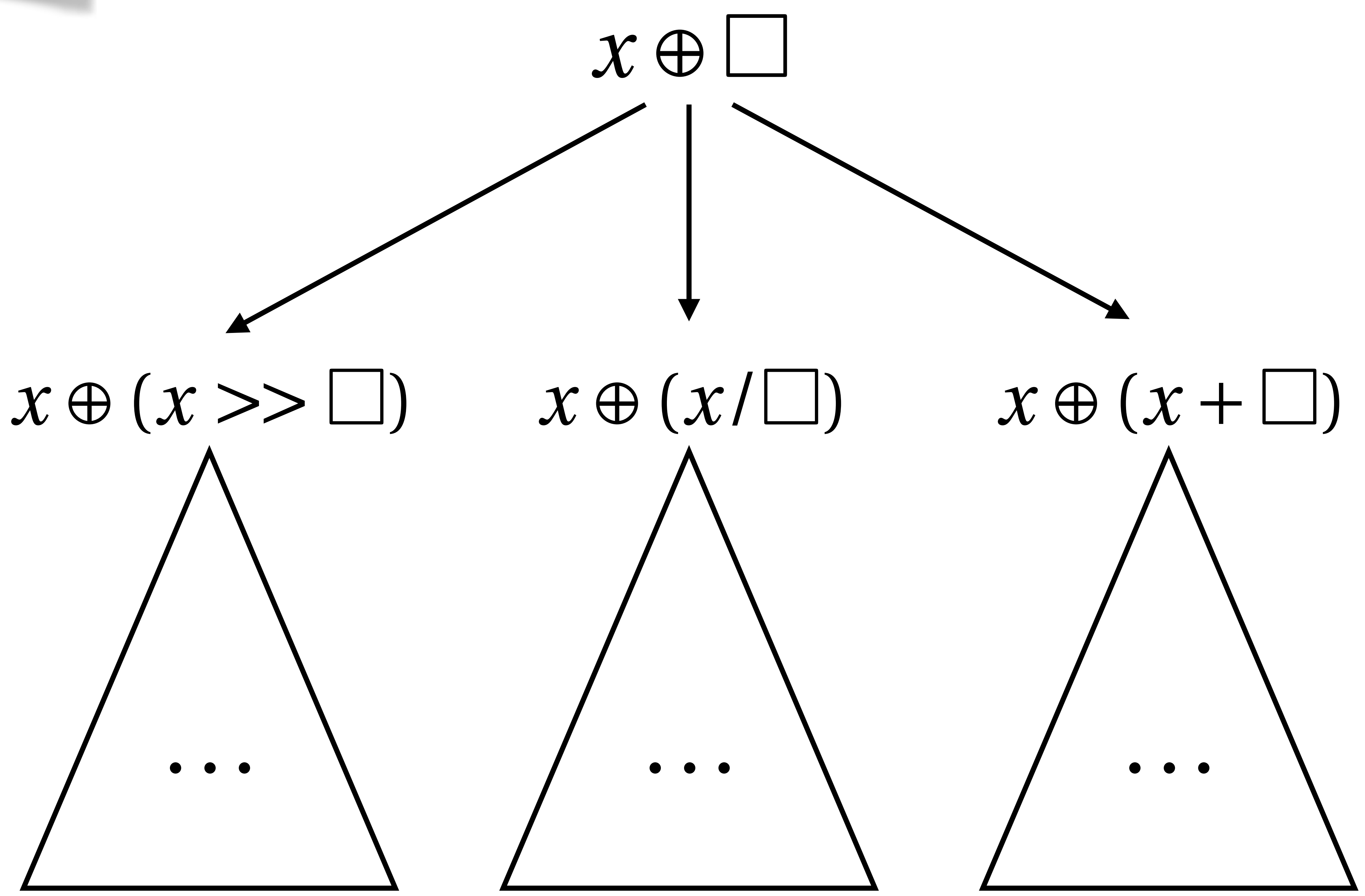
Candidate Partial Program

$f(x) = x \oplus \square$

Forward

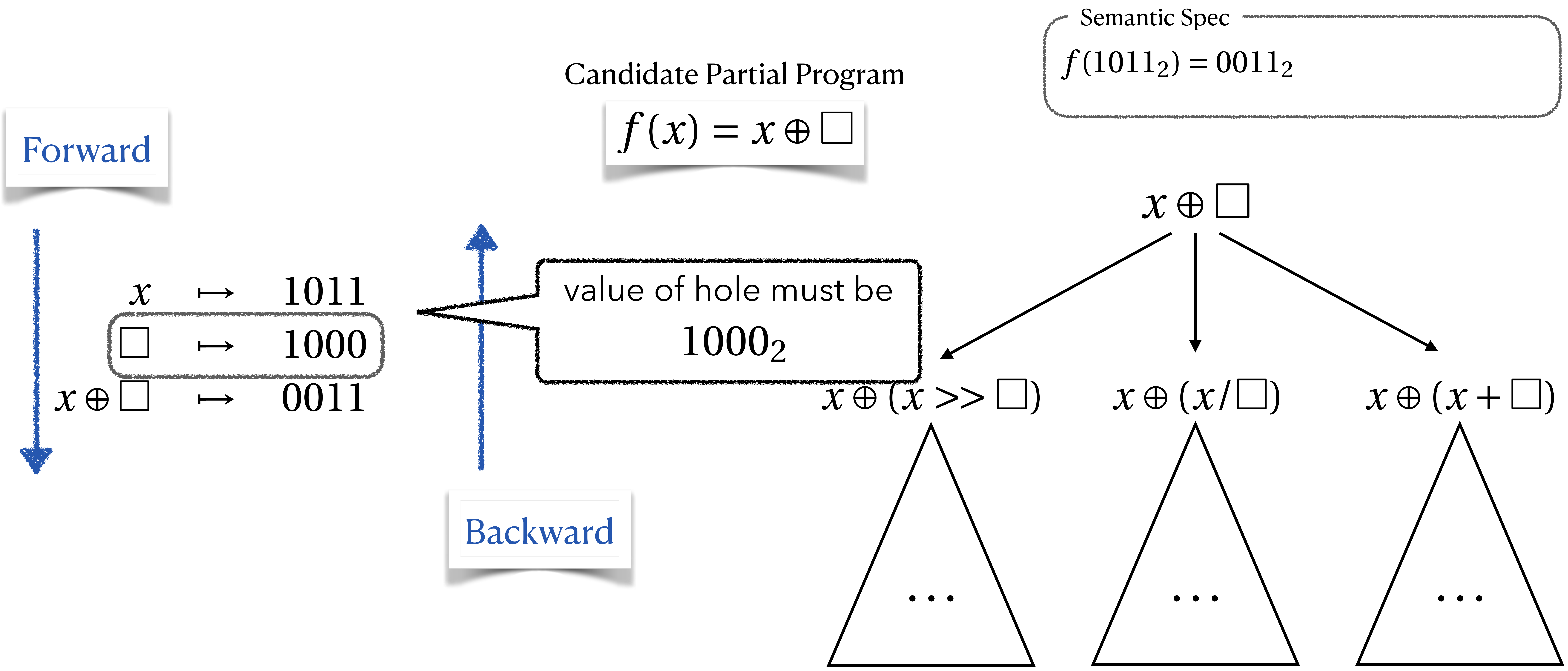
$$x \mapsto 1011$$
$$\square \mapsto \top\top\top\top$$
$$x \oplus \square \mapsto 0011$$

Backward



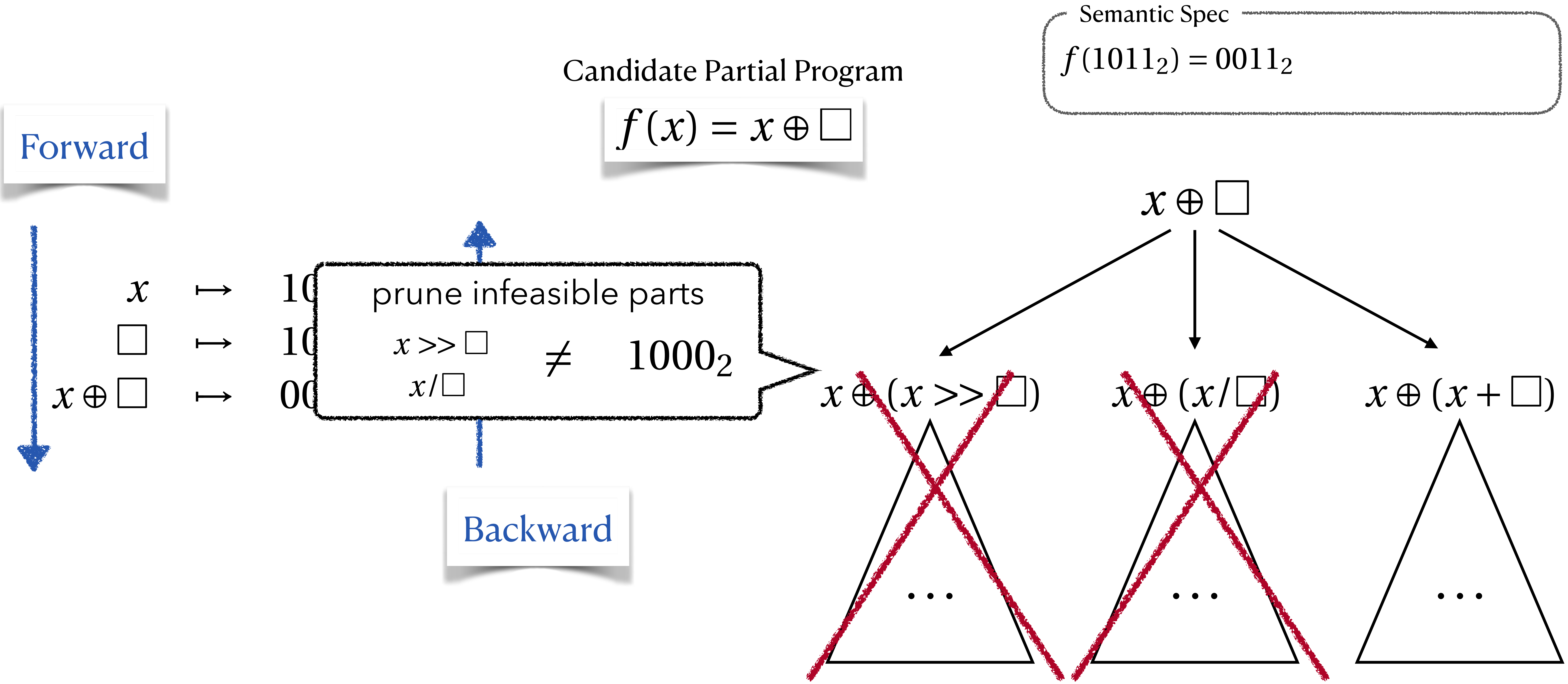
Need: Backward Analysis Too

Output feasibility + Hole Precondition



Need: Backward Analysis Too

Output feasibility + Hole Precondition

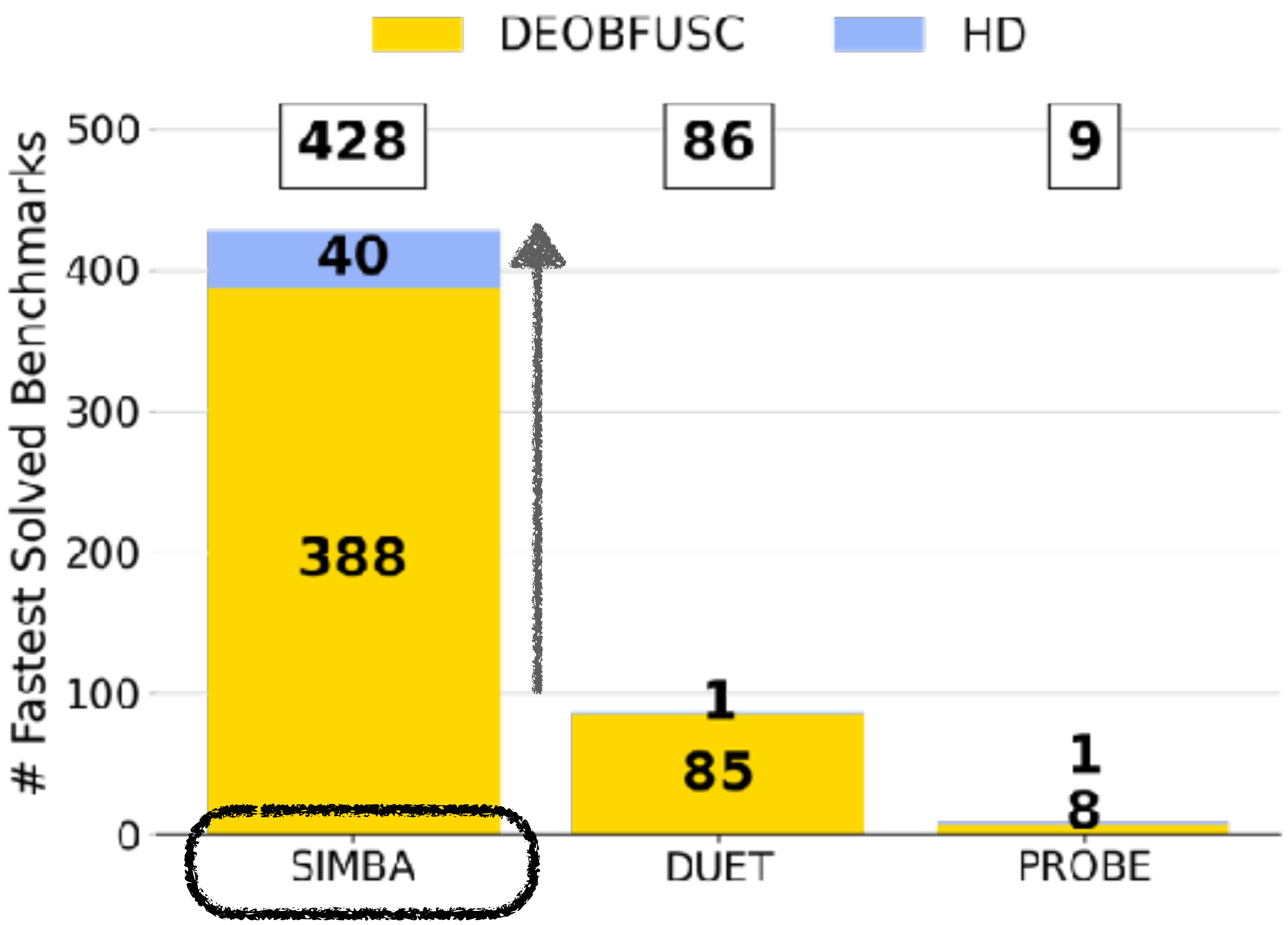


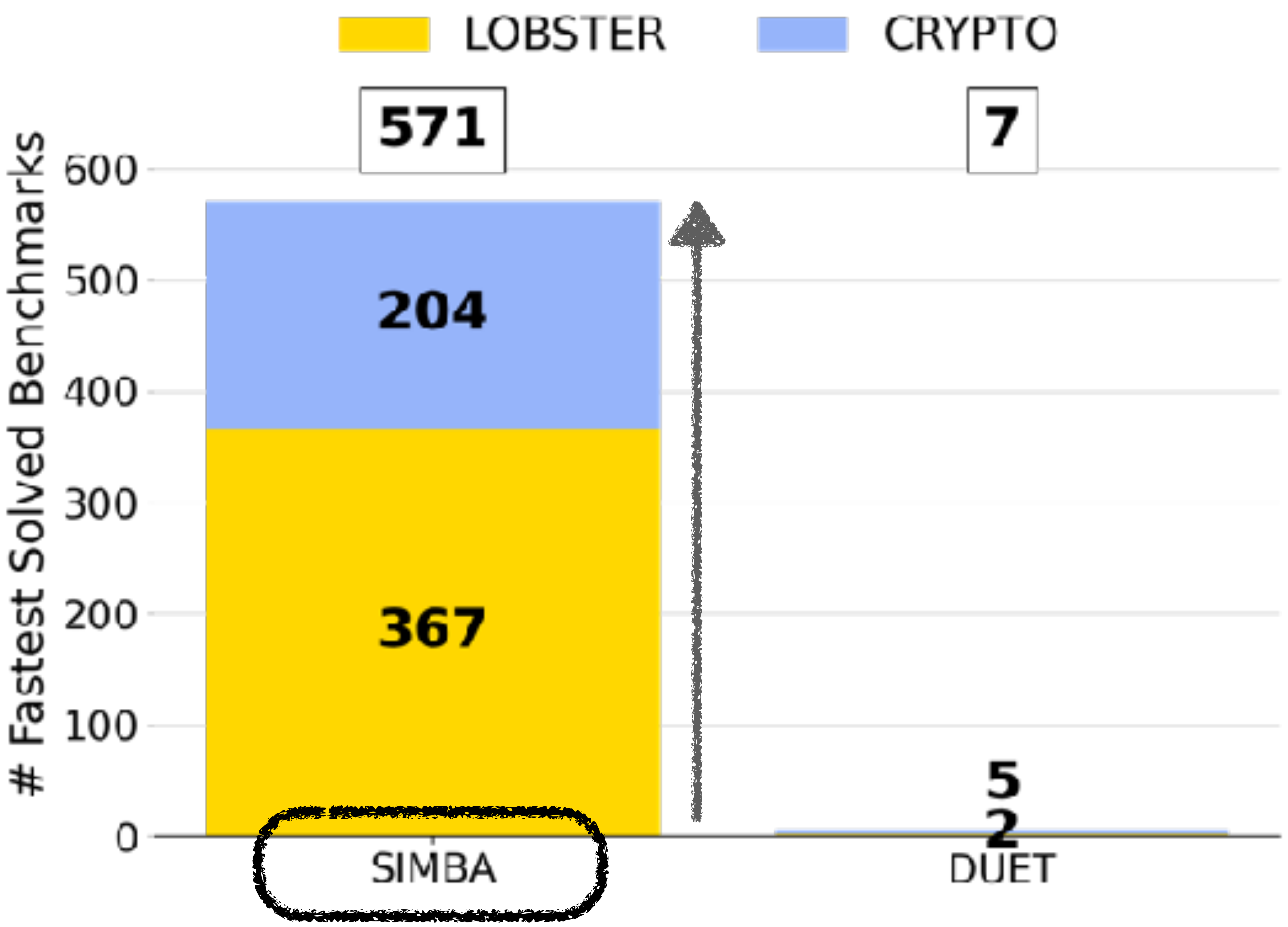
Evaluation

- Our tool: **Simba**
- Benchmark: 1,125 synthesis tasks from 4 sources
  - HD: 44 from hacker's delight
  - Deobfsc: 500 from the program deobfuscation tasks in prior work
  - Lobster: 369 from optimizing homomorphic evaluation circuits
  - Crypto: 212 from generating circuits resilient to side-channel attacks
- Baseline tools
  - **duet**: Woosuk Lee, "Combining the Top-Down Propagation and Bottom-Up Enumeration for Inductive Program Synthesis", POPL'21
  - **probe**: Barke et al., Just-in-Time Learning for Bottom-Up Enumerative Synthesis, OOPSLA'20

Results

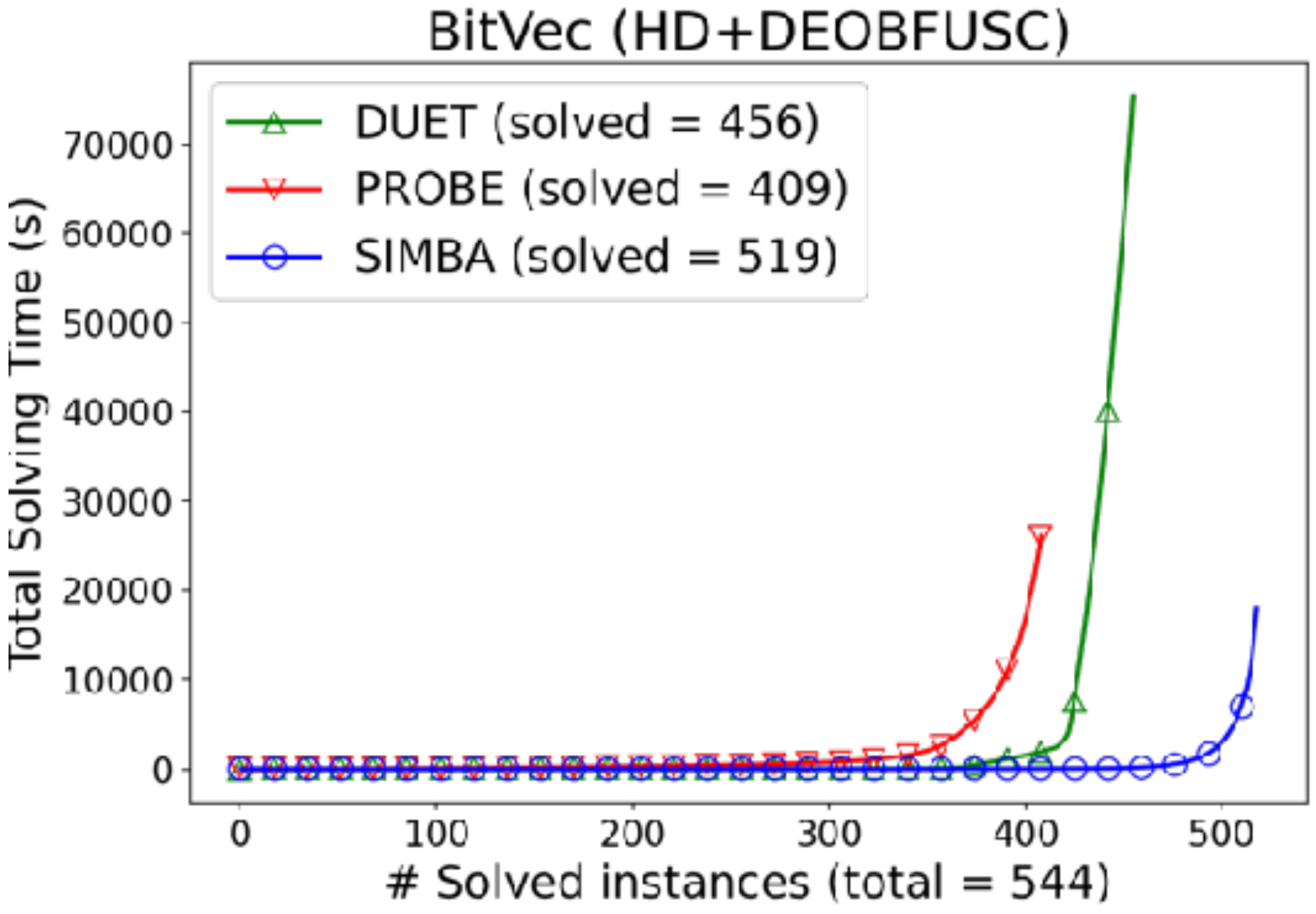
- Siginificantly outperforms the baseline tools

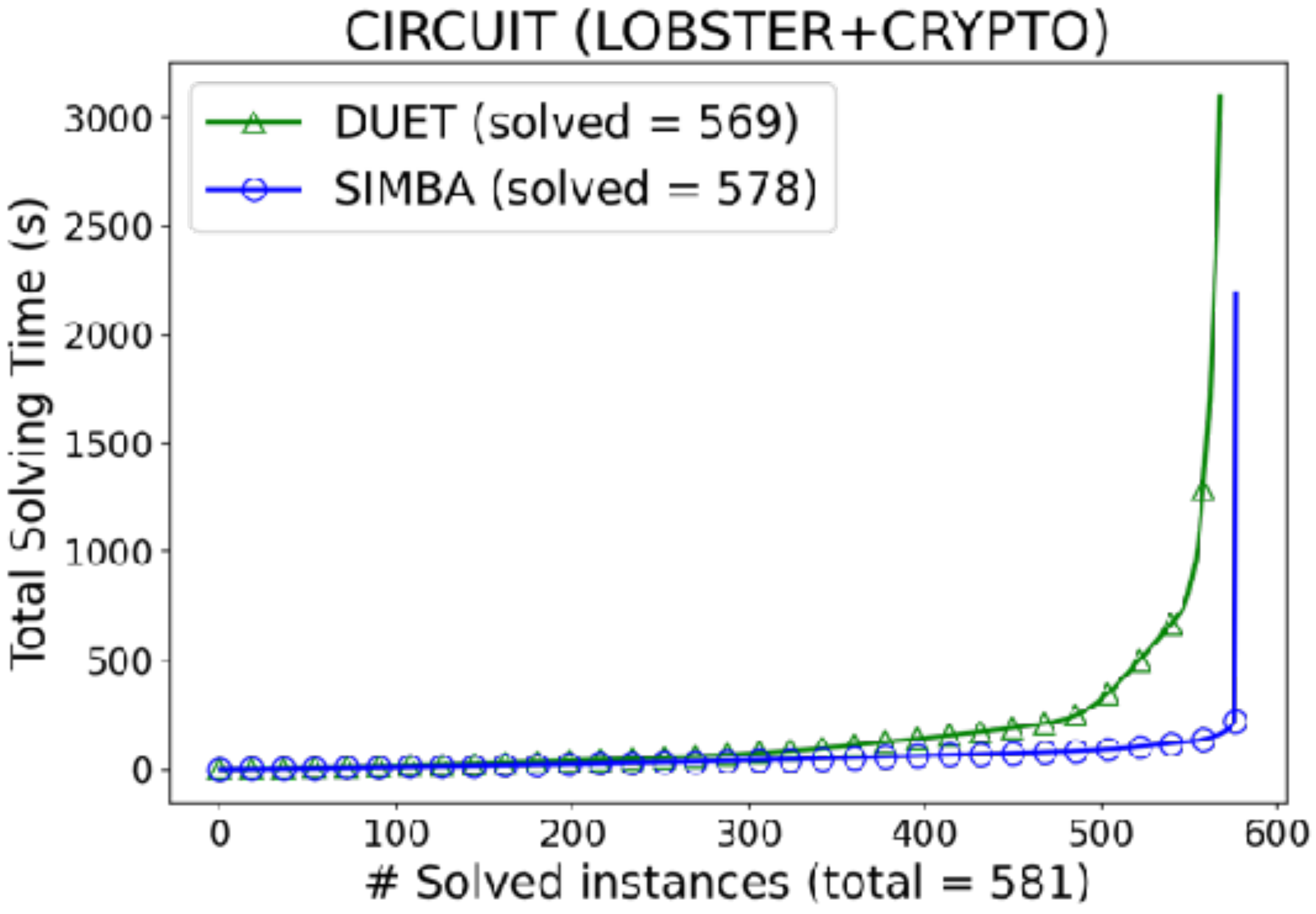




Results

- Siginificantly outperforms the baseline tools





Conclusion

- By using advanced search algorithms
  - program synthesis, term rewriting, and equality saturation
  - and by leveraging the high performance of modern computers
- In certain cases
  - Low-level languages
  - e.g., Boolean circuits and bitwise integers
- We can achieve better optimization than domain experts
  - By discovering new optimization rules
  - and sophisticated rule application orders that yield (nearly)optimal results