SWAPPER: A Framework for Automatic Generation of Formula Simplifiers based on Conditional Rewrite Rules

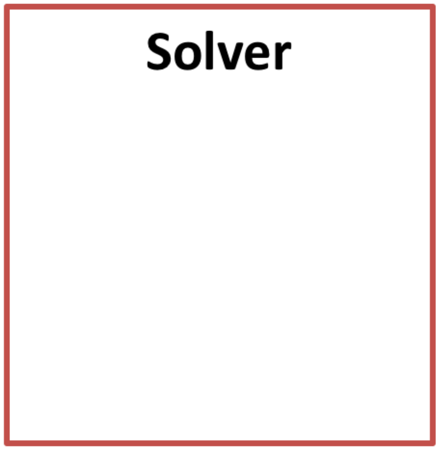
Rohit Singh Armando Solar-Lezama

Massachusetts Institute of Technology

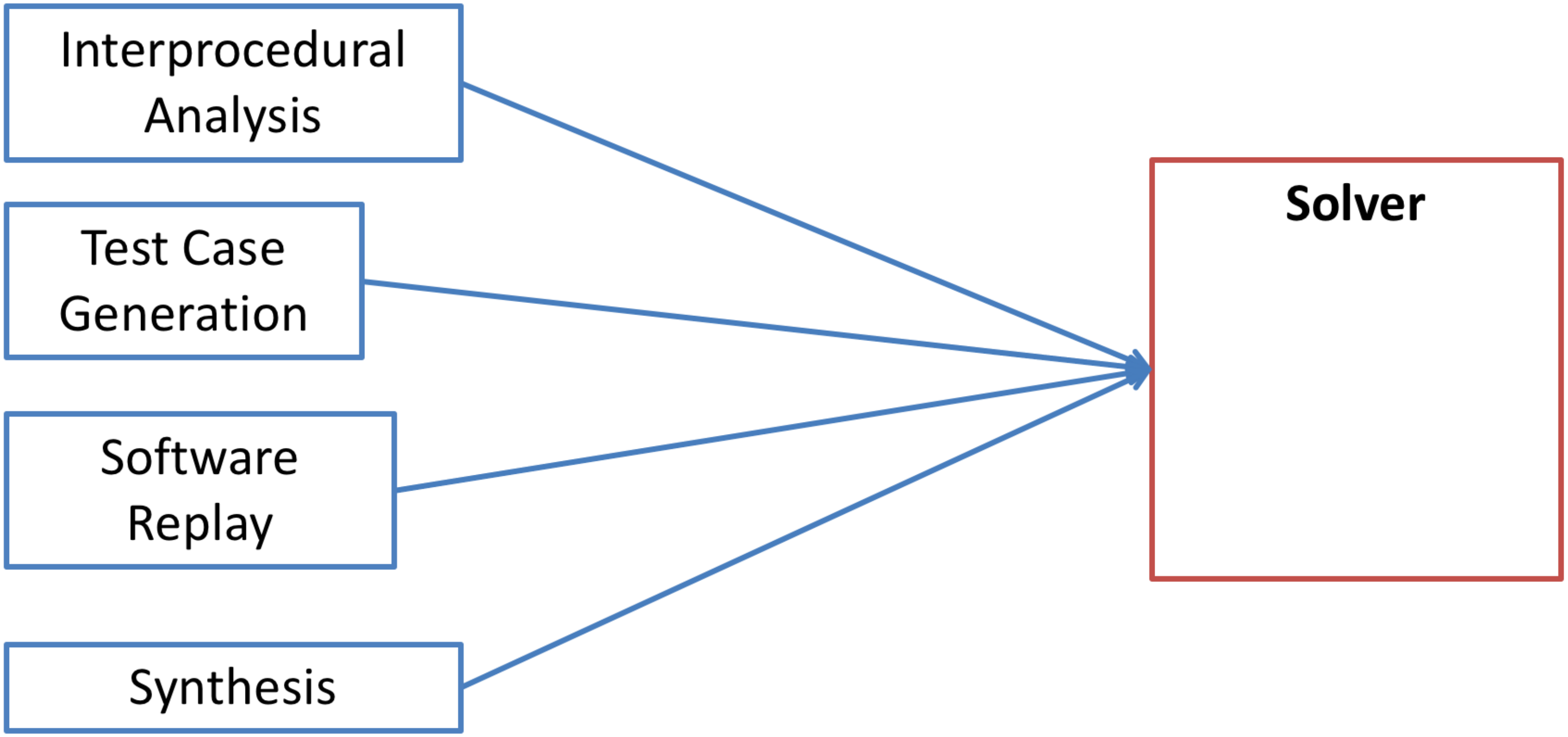
Formal Methods in Computer-Aided Design 2016
Mountain View, CA, USA

General Constraint Solvers

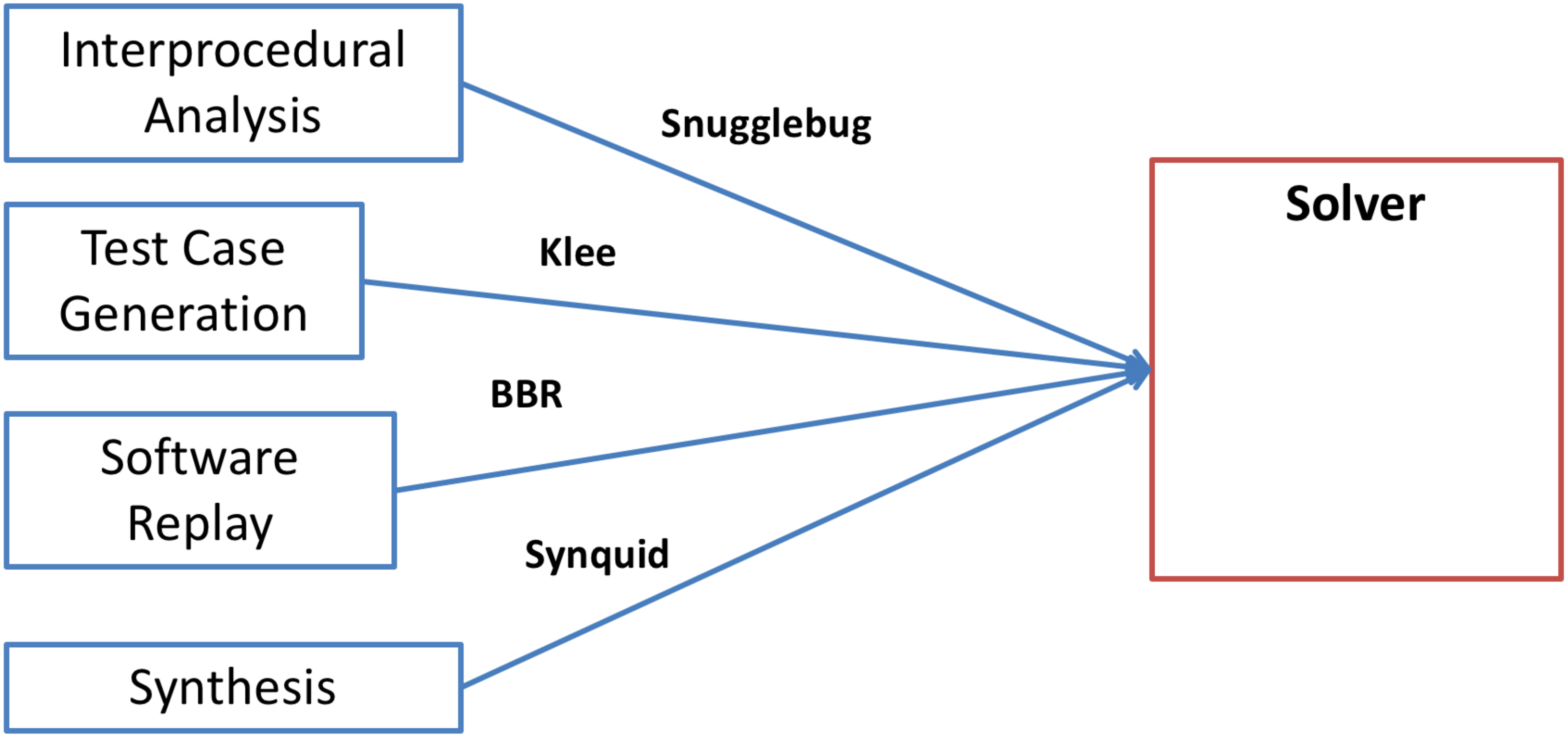
General Constraint Solvers



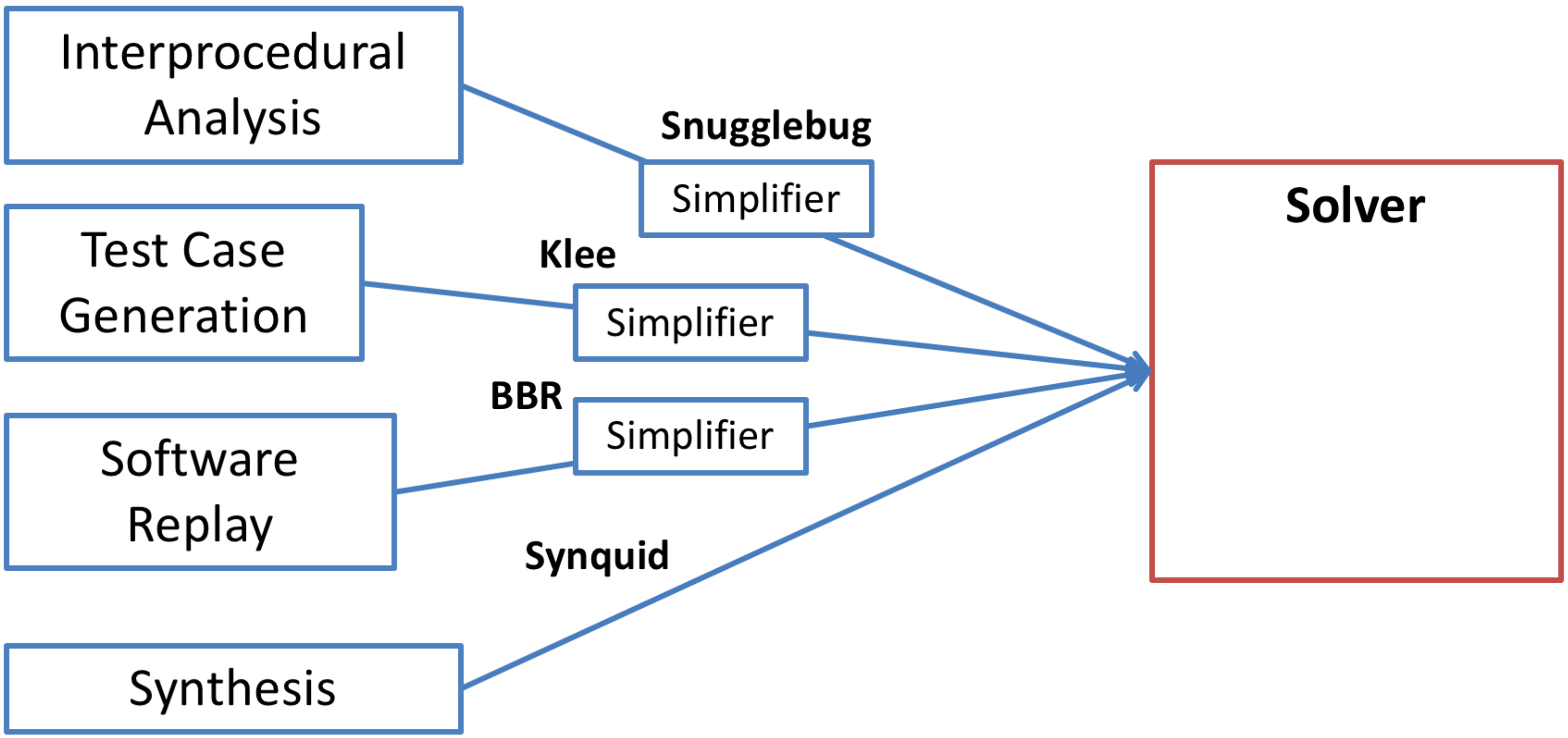
General Constraint Solvers



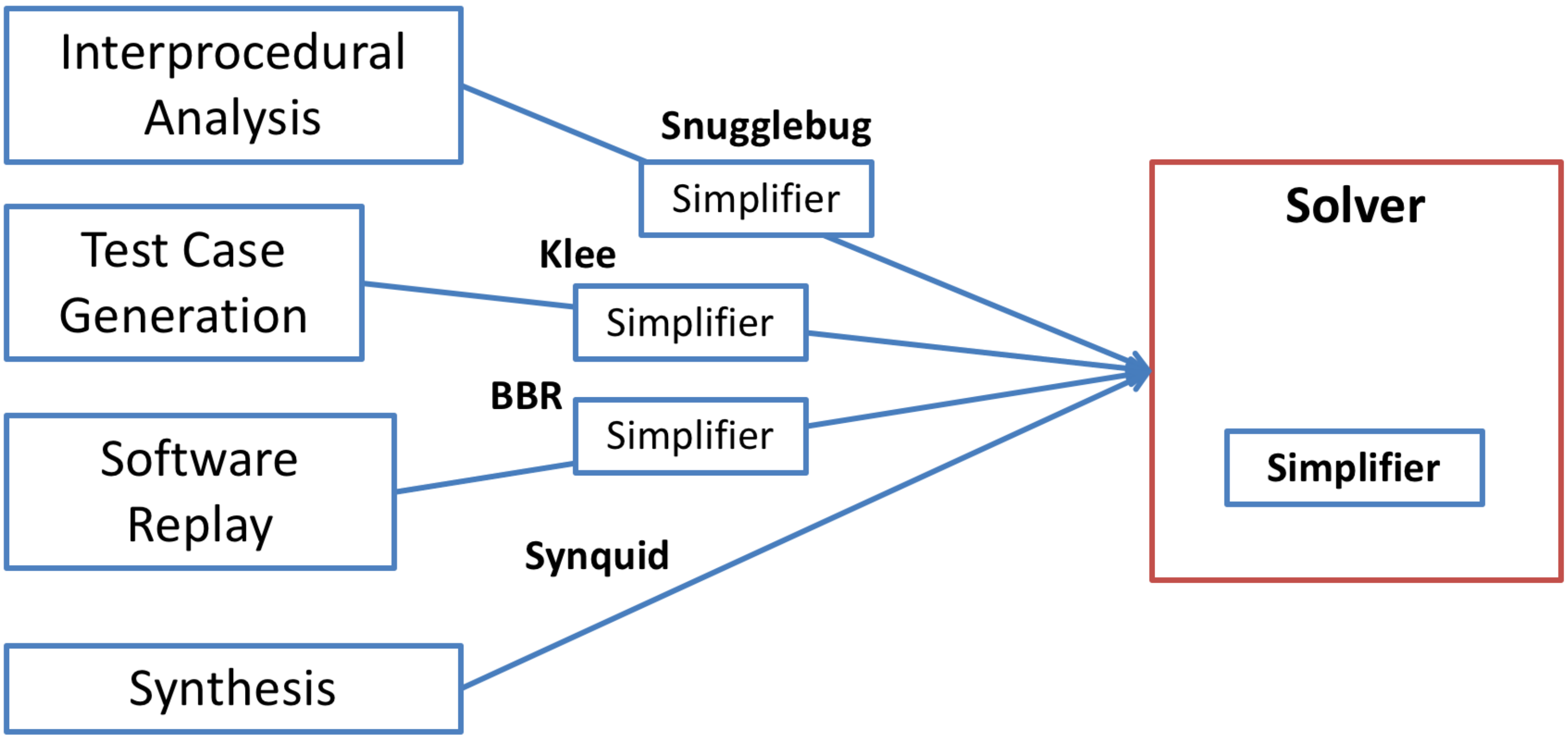
General Constraint Solvers



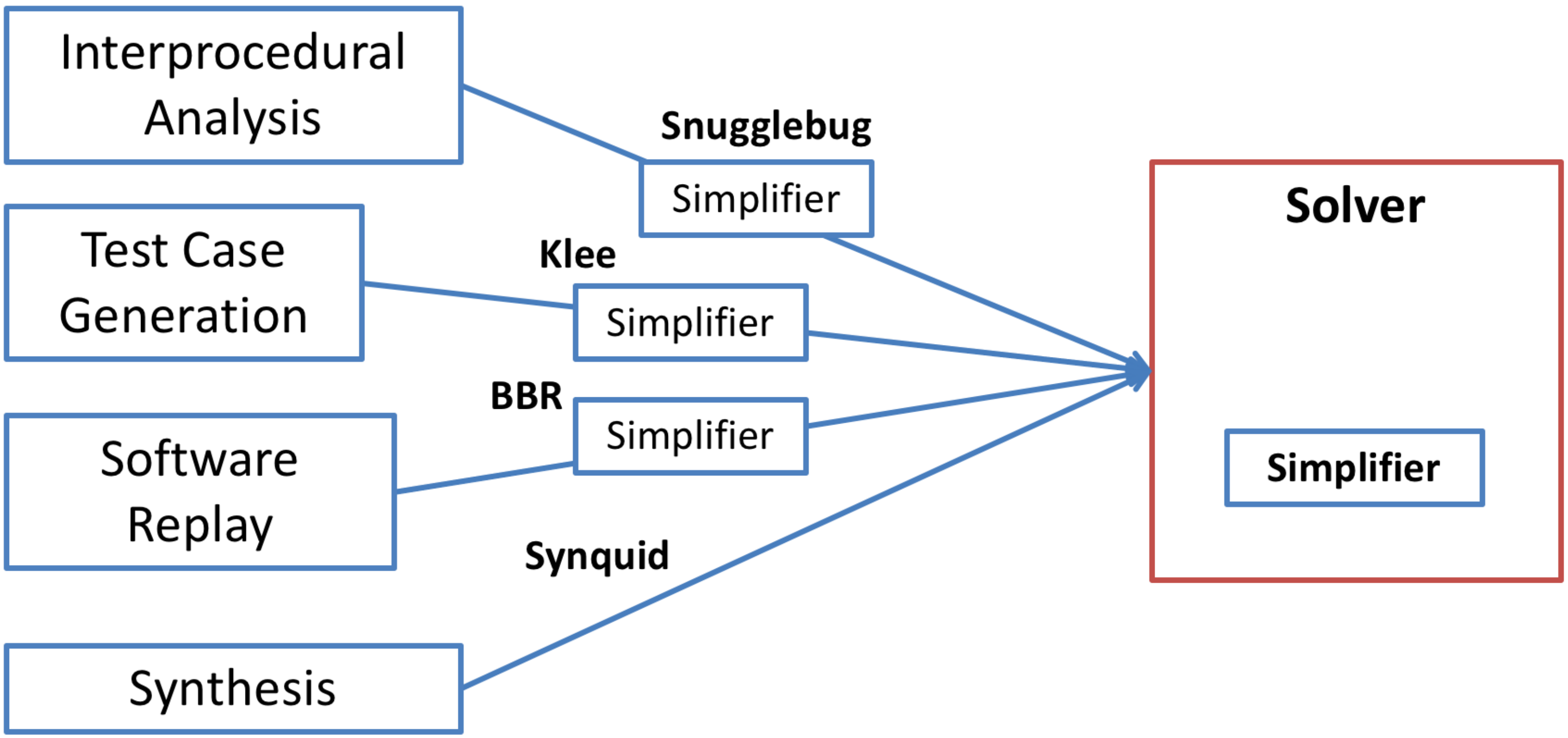
General Constraint Solvers



General Constraint Solvers

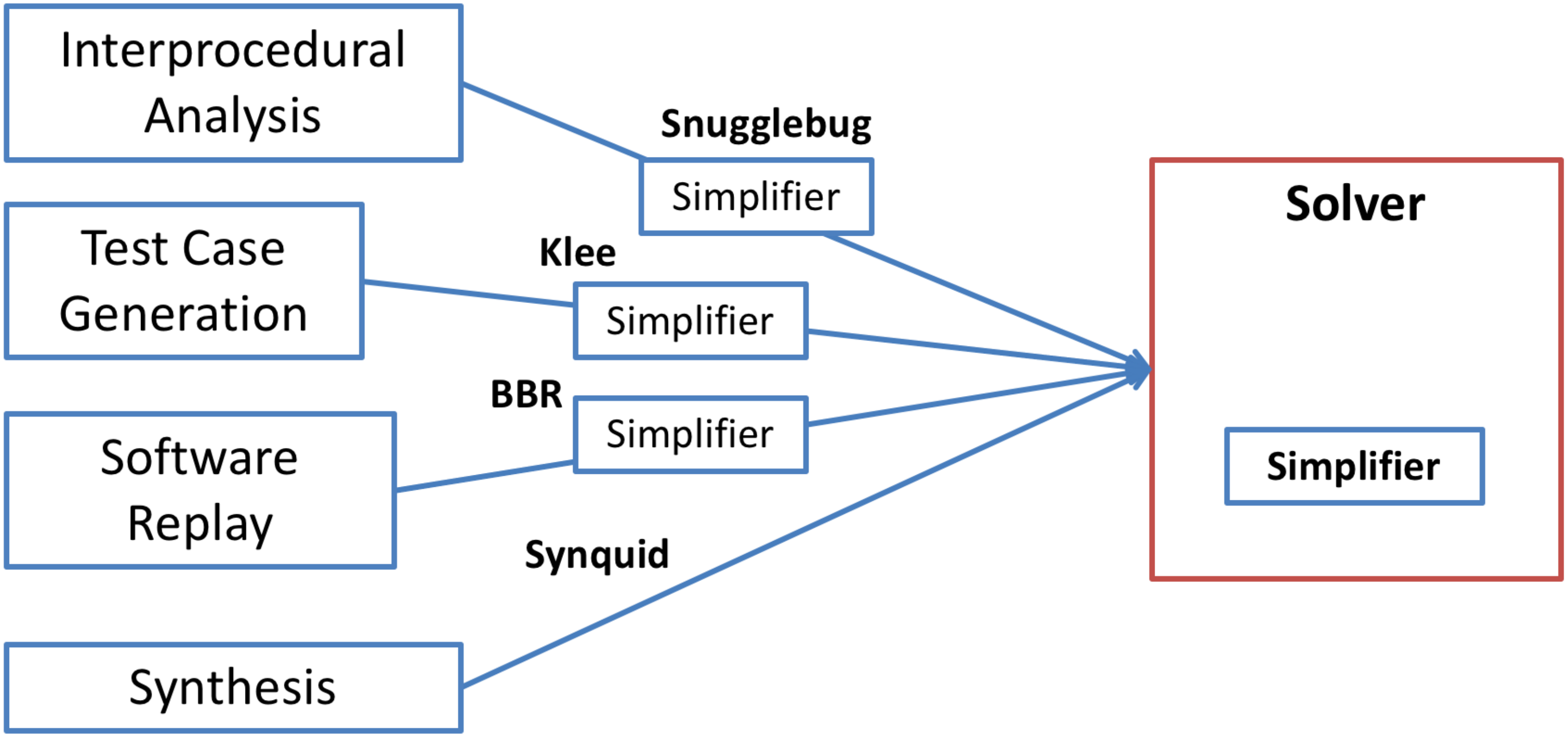


General Constraint Solvers



Simplifiers are very application specific

General Constraint Solvers



Simplifiers are very application specific

Not every tool can afford a custom simplifier

Custom simplifiers are expensive

Custom simplifiers are expensive

Trial and Error

Custom simplifiers are expensive

Trial and Error

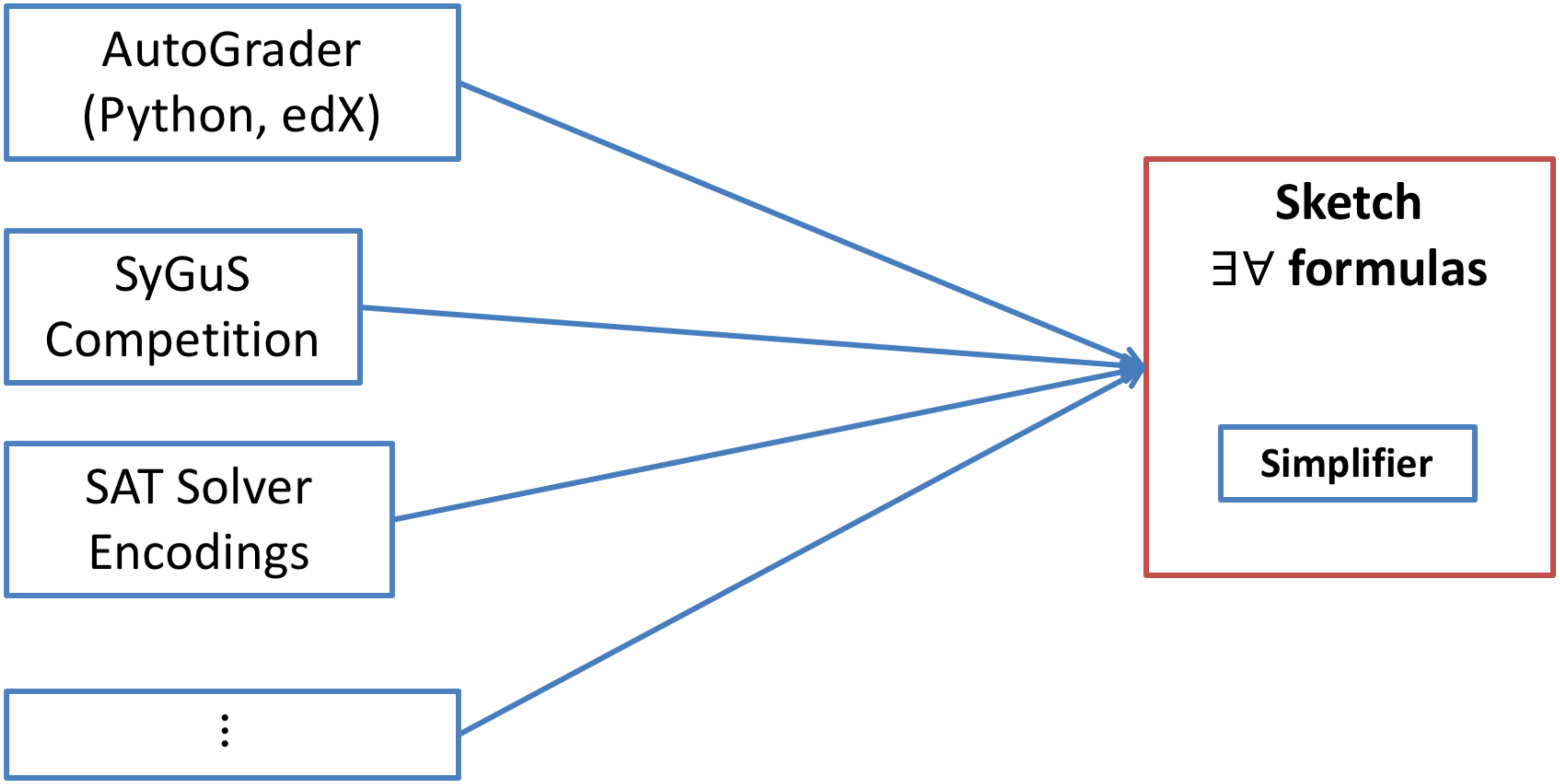
Each Try is hard

Target: Sketch Solver

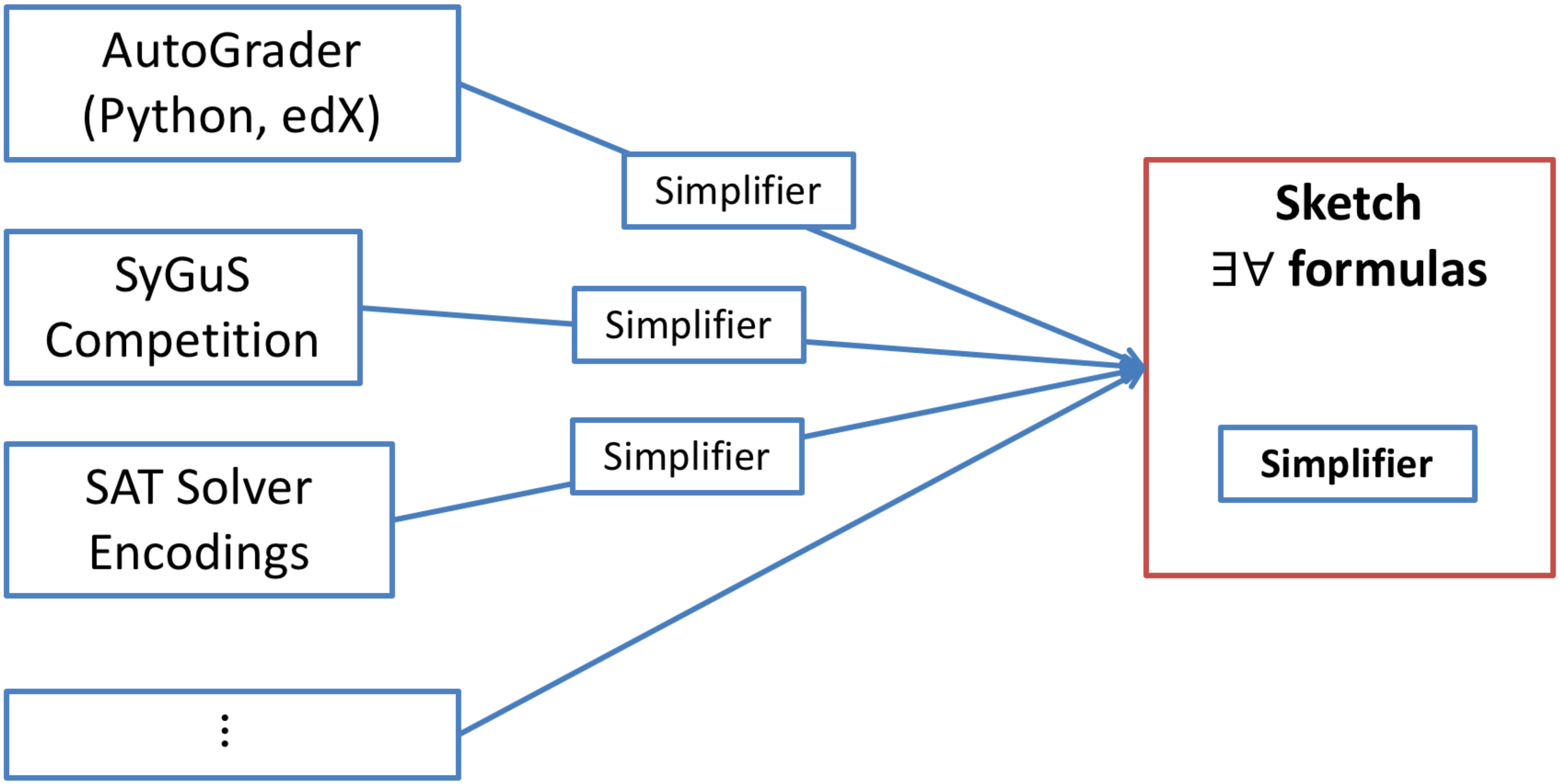
Sketch
∃∀ formulas

Target: Sketch Solver

Target: Sketch Solver



Target: Sketch Solver



Auto-generate efficient domain-specific simplifiers

Sketch Simplifier

Sketch Simplifier

Messy low-level C++ code

Sketch Simplifier

Messy low-level C++ code

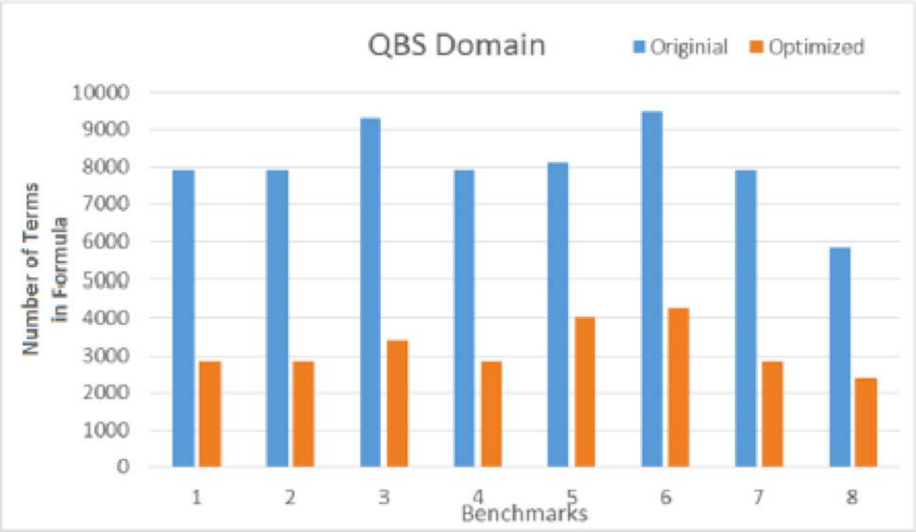
Employs simple declarative *Rewrite rules*

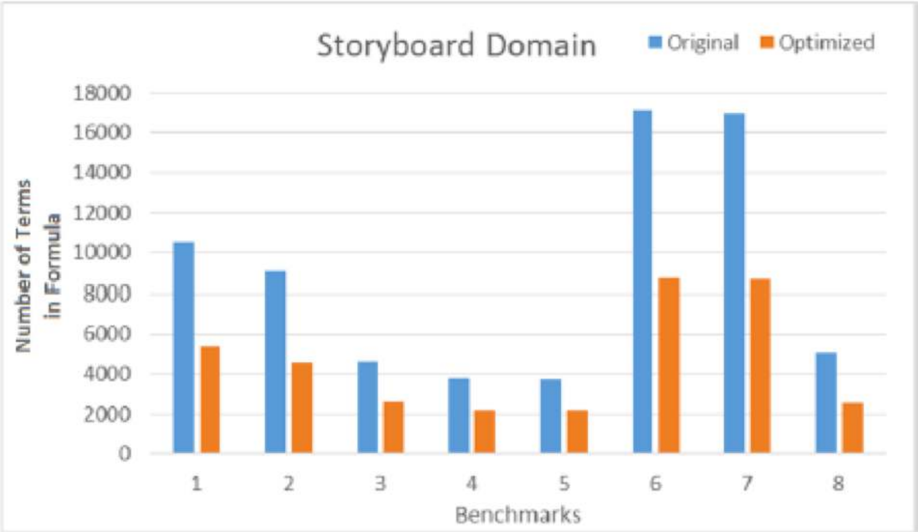
Sketch Simplifier

Messy low-level C++ code

Employs simple declarative *Rewrite rules*

Huge impact on performance



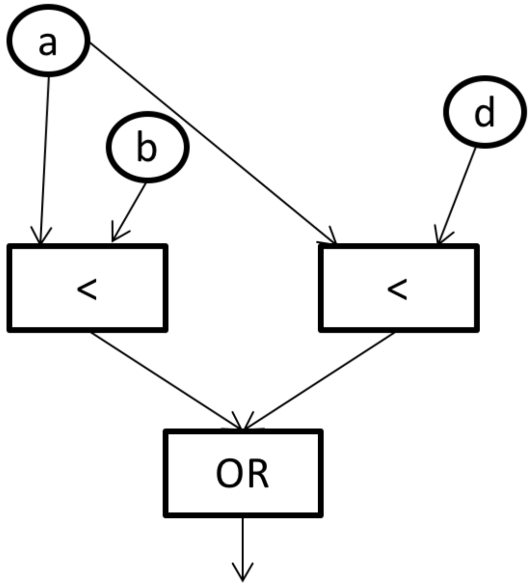


Internal Representation

- Internal language for constraints
- Theory of Arrays, booleans and integer arithmetic

Internal Representation

- Internal language for constraints
- Theory of Arrays, booleans and integer arithmetic

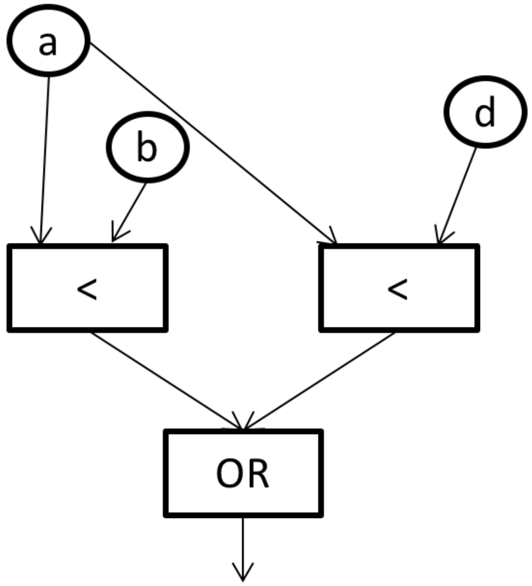


Directed Acyclic Graphs

Internal Representation

- Internal language for constraints
- Theory of Arrays, booleans and integer arithmetic

$$or(lt(a, b), lt(a, d))$$



Directed Acyclic Graphs

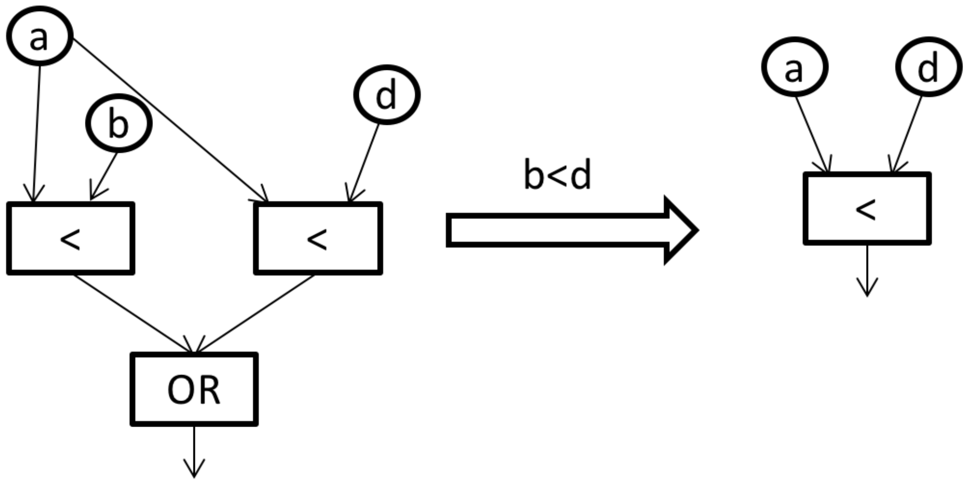
Conditional Rewrite Rules

Conditional Rewrite Rules

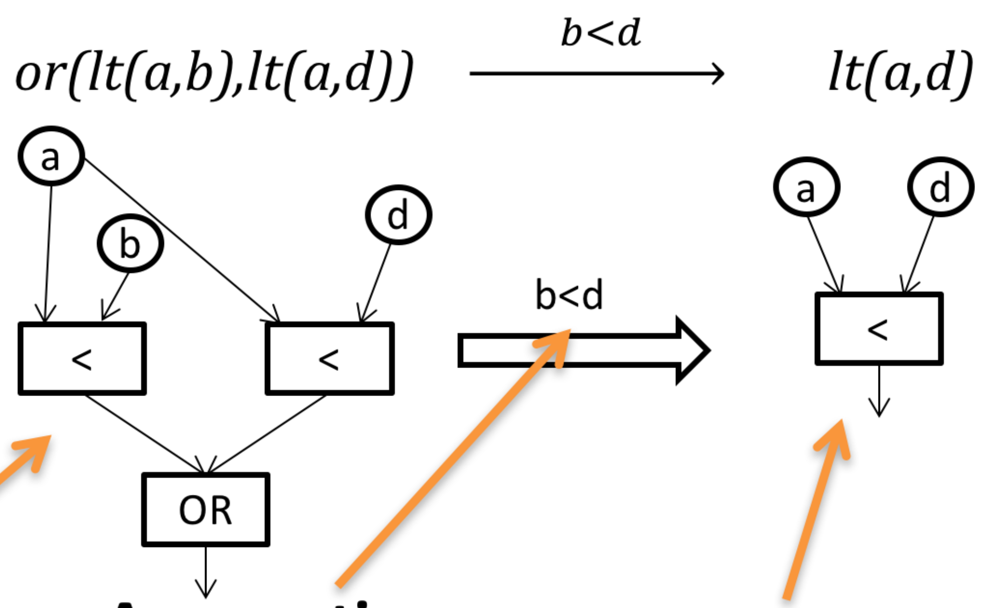
$$or(lt(a,b),lt(a,d)) \xrightarrow{b<d} lt(a,d)$$

Conditional Rewrite Rules

$$or(lt(a,b),lt(a,d)) \xrightarrow{b<d} lt(a,d)$$



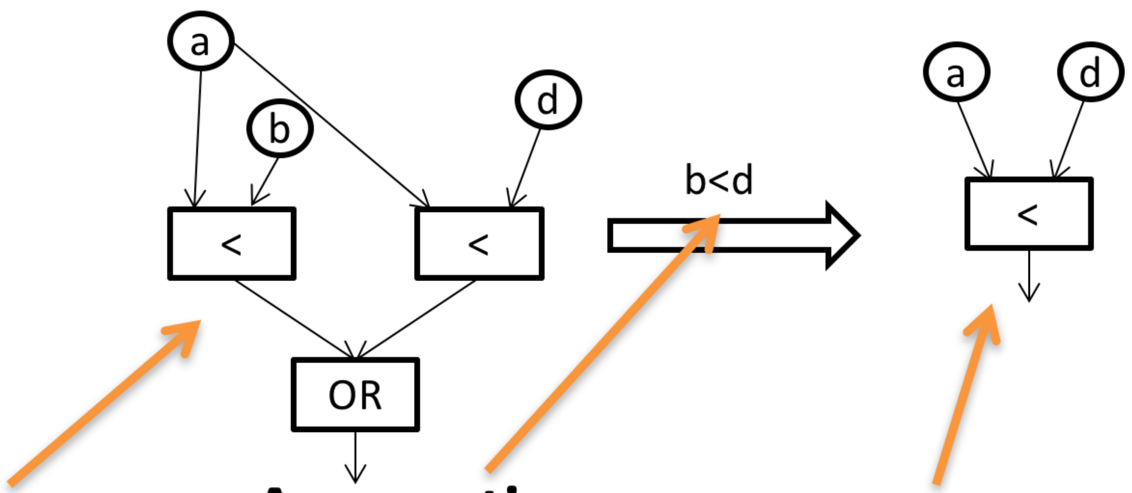
Conditional Rewrite Rules



- **Pattern** $\xrightarrow{\textbf{Assumptions}}$ **Pattern**

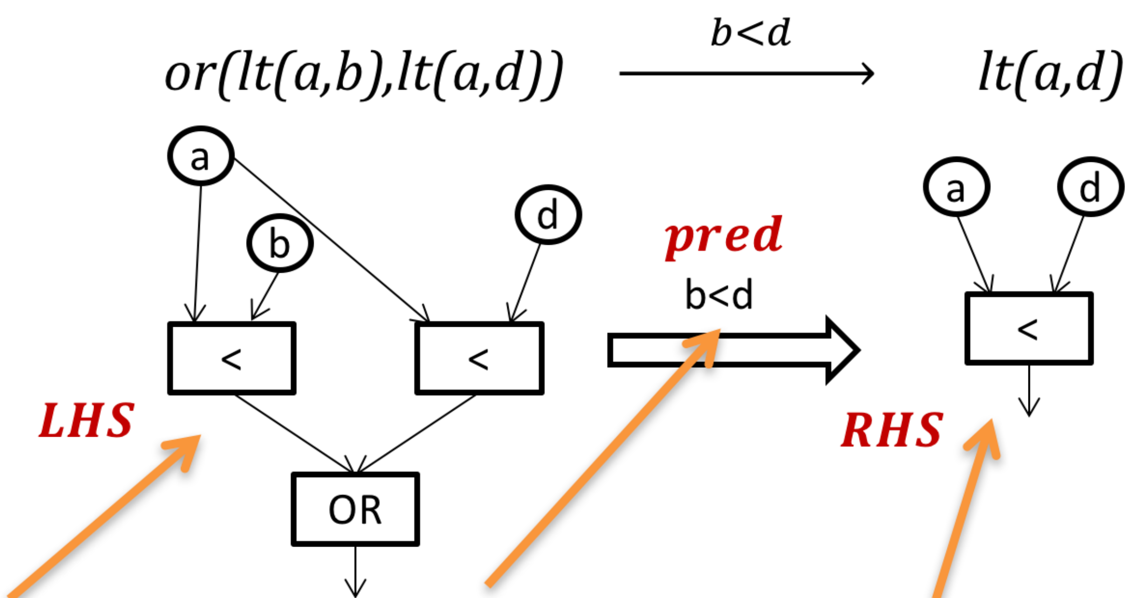
Conditional Rewrite Rules

$$or(lt(a,b),lt(a,d)) \xrightarrow{b<d} lt(a,d)$$



- **Pattern** $\xrightarrow{\textbf{Assumptions}}$ **Pattern**
- **Inputs :** a b d

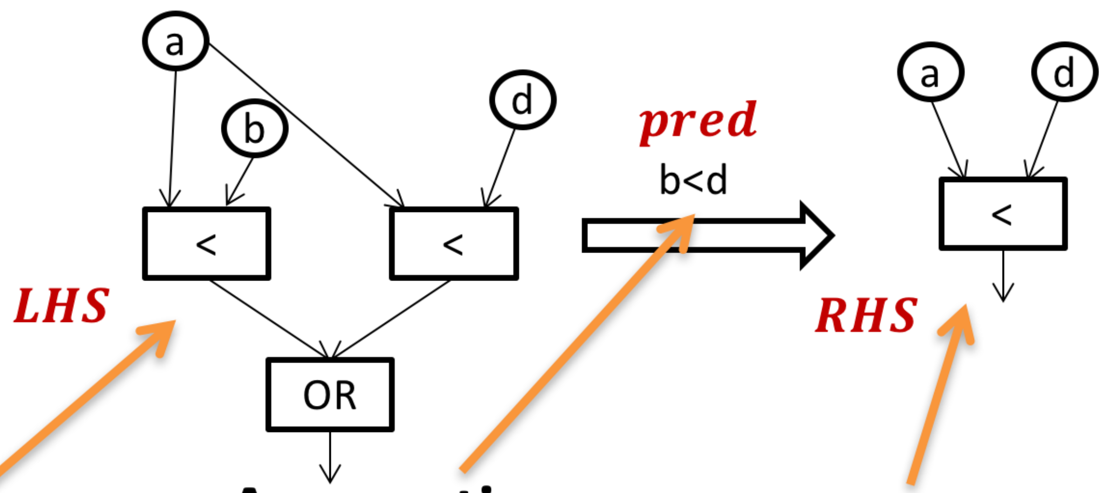
Conditional Rewrite Rules



- **Pattern** $\xrightarrow{\textbf{Assumptions}}$ **Pattern**
- **Inputs :** ⓐ ⓑ ⓓ
- $\forall x\ pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$

Conditional Rewrite Rules

$$or(lt(a,b),lt(a,d)) \xrightarrow{b<d} lt(a,d)$$



- **Pattern** $\xrightarrow{\textbf{Assumptions}}$ **Pattern**
- **Inputs :** 

- $\forall x \; pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$
- Sketch Simplifier: apply in order at each node

Rule application code

Code for implementing Rewrite Rules

$$and(lt(plus(a,e),x), \quad lt(plus(e,b),x)\ ) \xrightarrow{b<a} lt(plus(a,e),x)$$

Rule application code

Code for implementing Rewrite Rules

```
if(nfather->type == LT && nmother->type == LT){
  // (a+e<x) & (b+e<x) ---> a+e<x when b<a
  if(nfather->mother->type == PLUS && nmother->mother-
>type == PLUS){
    bool_node* nfm = nfather->mother;
    bool_node* nmm = nmother->mother;

    bool_node* nmmConst = nmm->mother;
    bool_node* nmmExp = nmm->father;
    if(isConst(nmmExp)){
      bool_node* tmp = nmmExp;
      nmmExp = nmmConst;
      nmmConst = tmp;
    }
    bool_node* nfmConst = nfm->mother;
    bool_node* nfmExp = nfm->father;
    if(isConst(nfmExp)){
      bool_node* tmp = nfmExp;
      nfmExp = nfmConst;
      nfmConst = tmp;
    }
if(isConst(nfmConst) && isConst(nmmConst) && nfmExp== nmmExp){
      if(val(nfmConst) < val(nmmConst)){
        return nmother;
      }else{
        return nfather;
      }
    }
  }
}
```

$$and(lt(plus(a,e),x),\ lt(plus(e,b),x))\ \xrightarrow{b<a}\ lt(plus(a,e),x)$$

Problem Statement

Given a corpus of benchmark problems (formulas) from a domain:

Problem Statement

Given a corpus of benchmark problems (formulas) from a domain:

- Learn *commonly occurring* patterns

Problem Statement

Given a corpus of benchmark problems (formulas) from a domain:

- Learn *commonly occurring* patterns
- Learn *impactful* conditional Rewrite Rules

Problem Statement

Given a corpus of benchmark problems (formulas) from a domain:

- Learn *commonly occurring* patterns
- Learn *impactful* conditional Rewrite Rules
- Generate an *efficient* simplifier from these rules

Problem Statement

Given a corpus of benchmark problems (formulas) from a domain:

- Learn *commonly occurring* patterns
- Learn *impactful* conditional Rewrite Rules
- Generate an *efficient* simplifier from these rules

Solution: **SWAPPER** framework

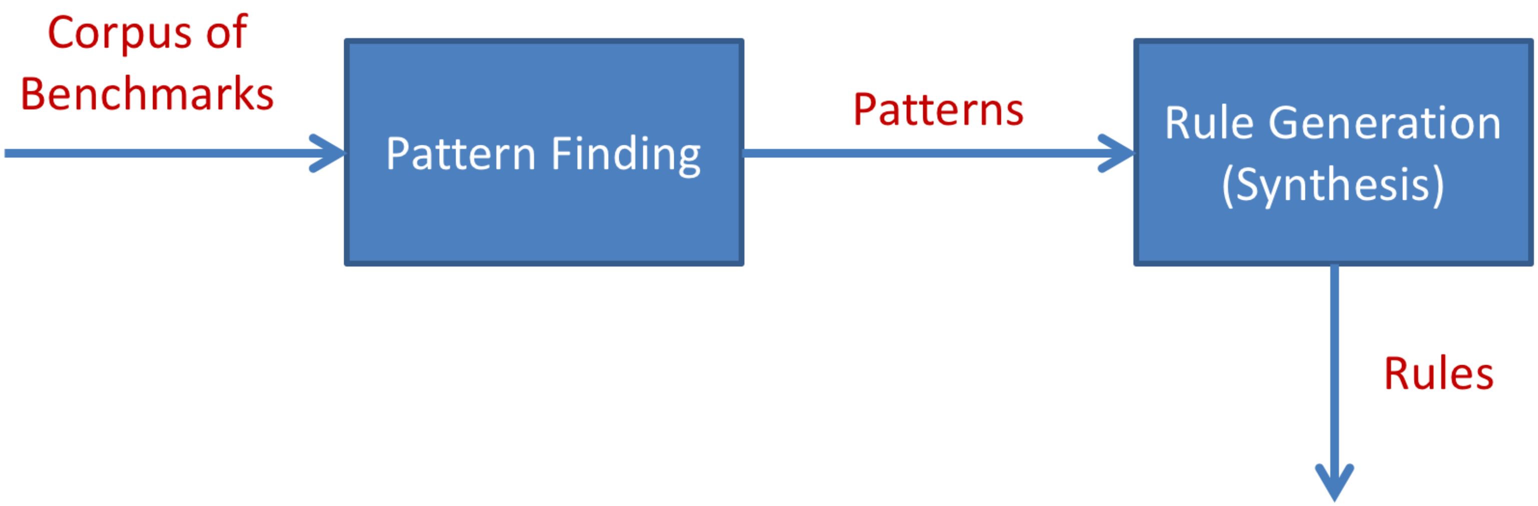
SWAPPER framework

SWAPPER framework

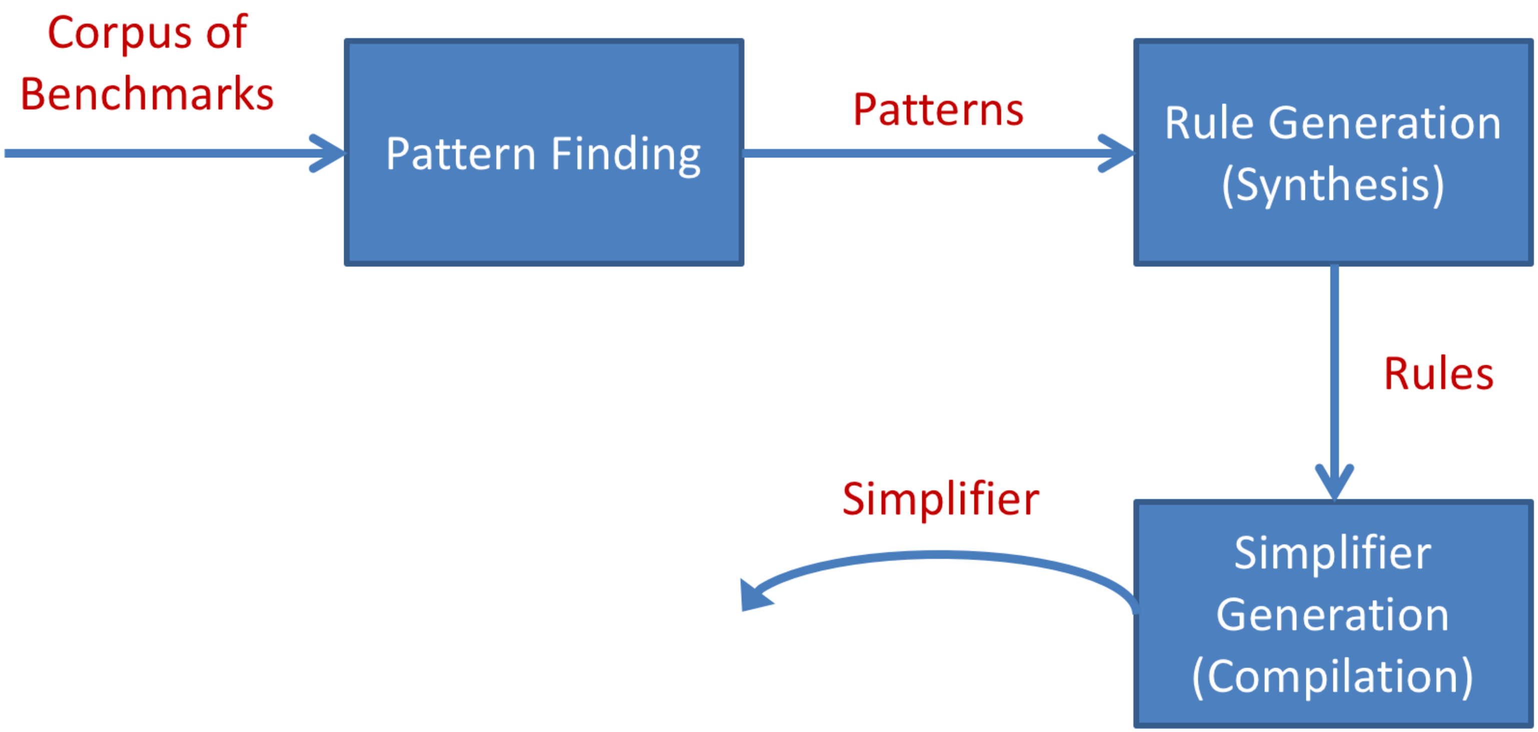
Corpus of Benchmarks

SWAPPER framework

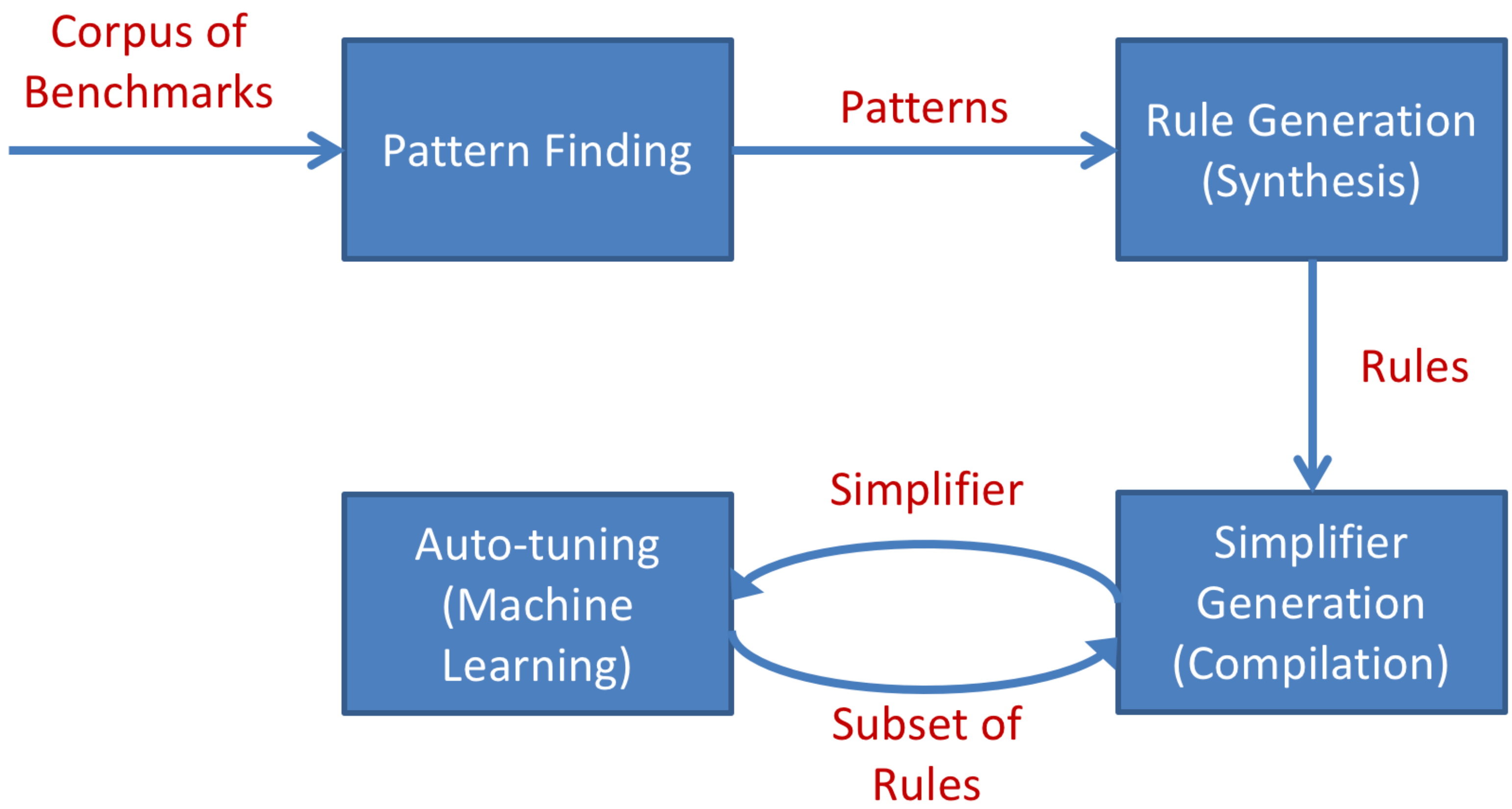
SWAPPER framework



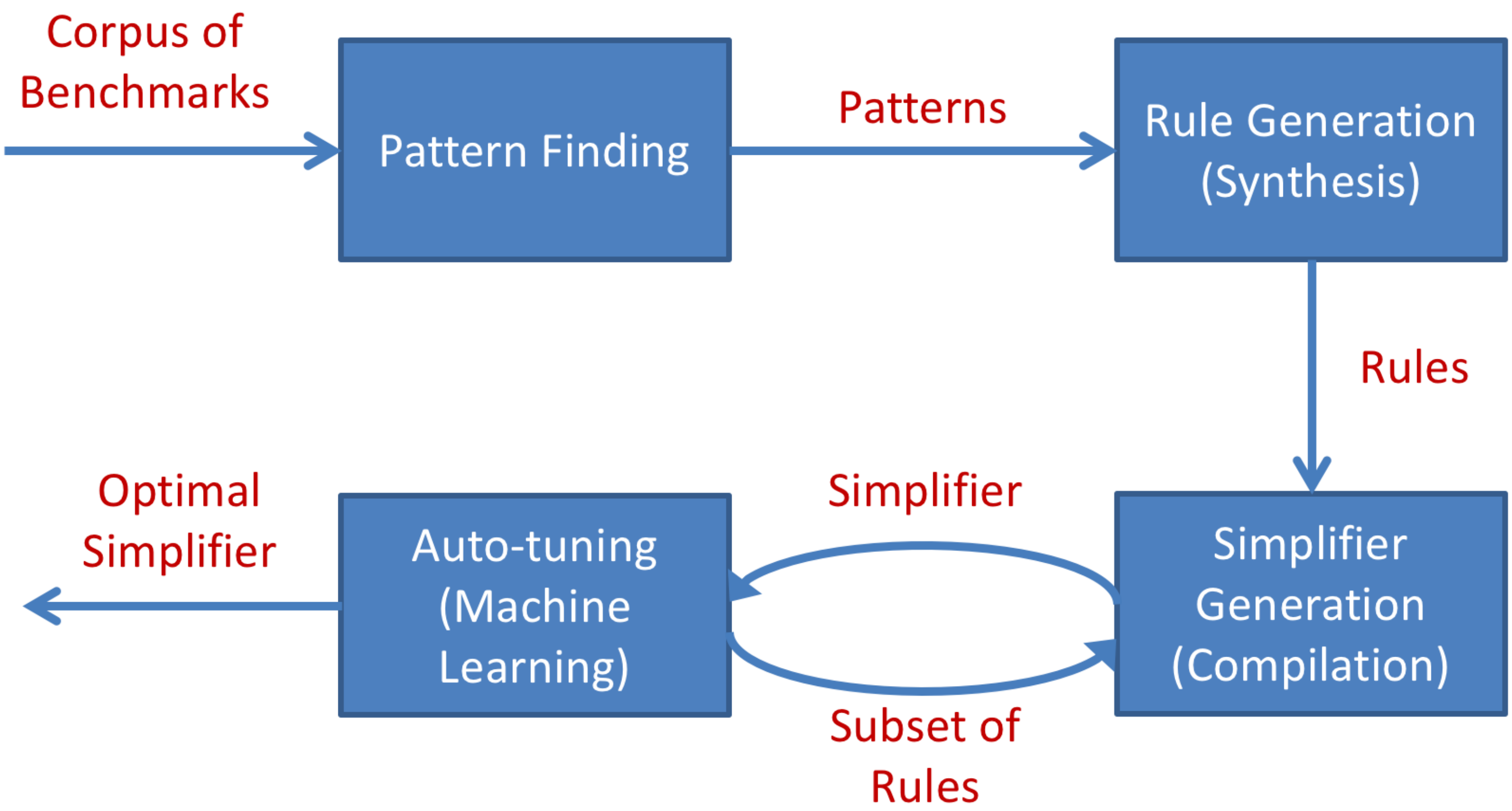
SWAPPER framework



SWAPPER framework



SWAPPER framework



Related Work

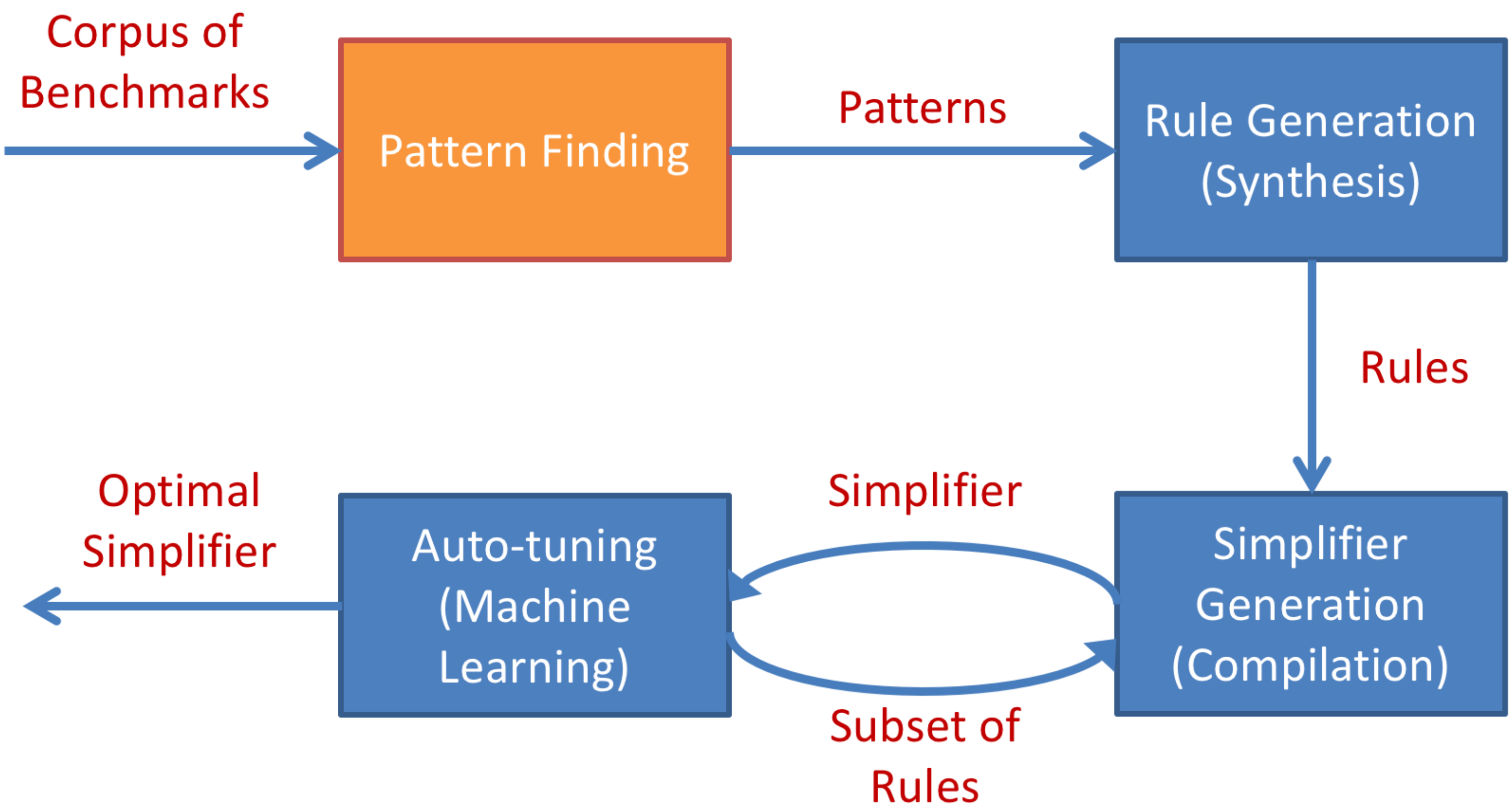
Related Work

- Peephole optimizations
  - Alive DSL [PLDI 15] : no synthesis of rules
  - Automatic generation by enumeration [ASPLOS 06]: no semantic guards

Related Work

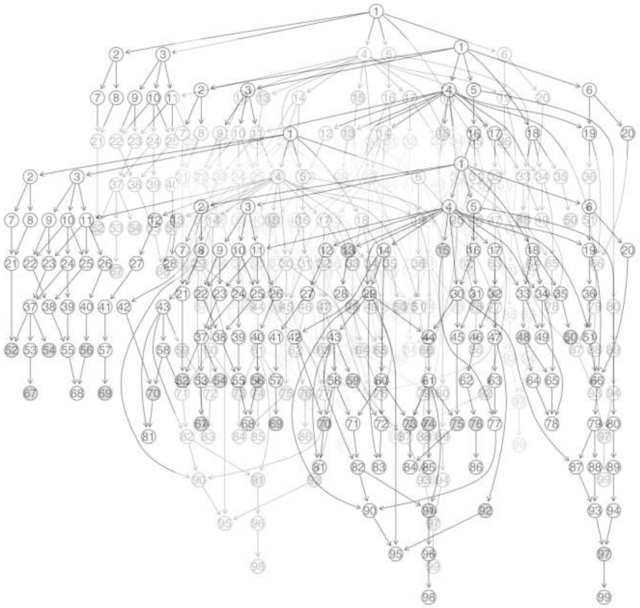
- Peephole optimizations
  - Alive DSL [PLDI 15] : no synthesis of rules
  - Automatic generation by enumeration [ASPLOS 06]: no semantic guards
- Term/Graph Rewriting:
  - Stratego/XT [ASF+SDF 97], GrGen

SWAPPER framework



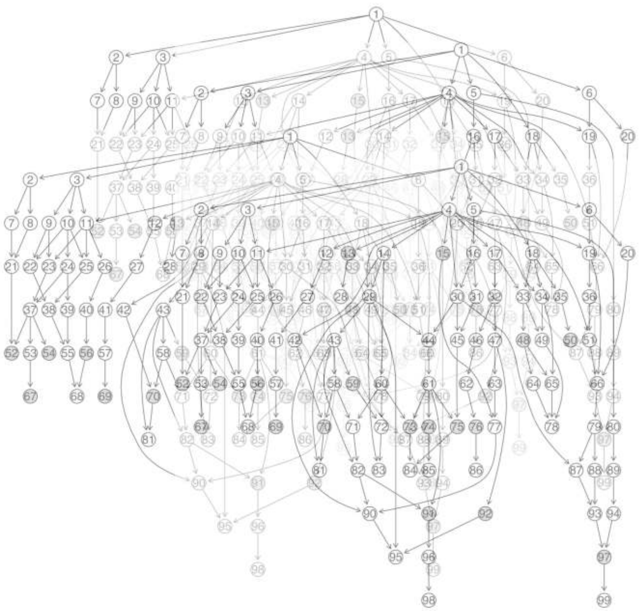
Pattern Finding

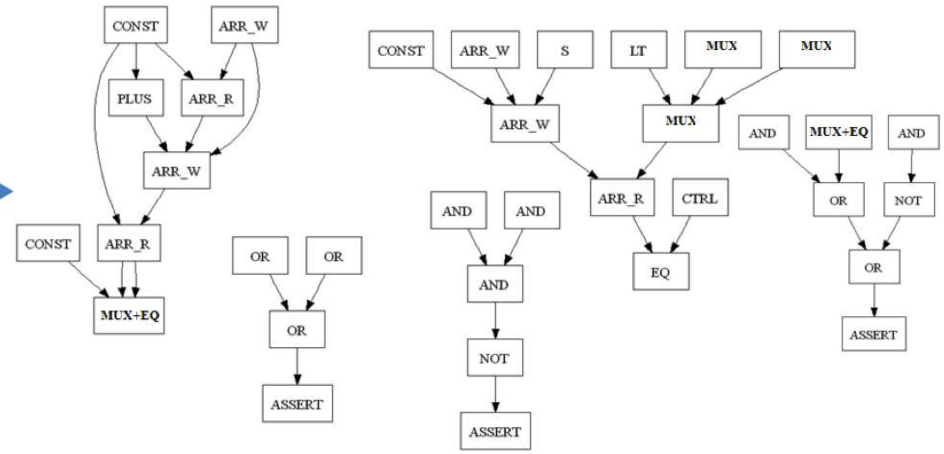
Pattern Finding



Corpus of Formulas (DAGs)

Pattern Finding

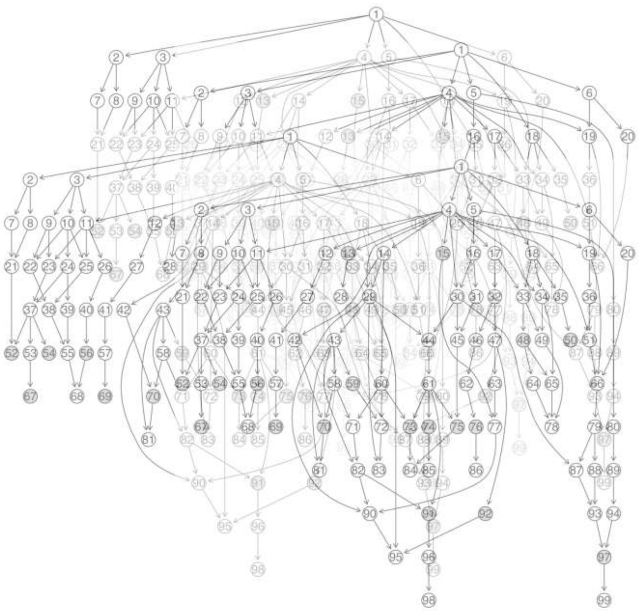


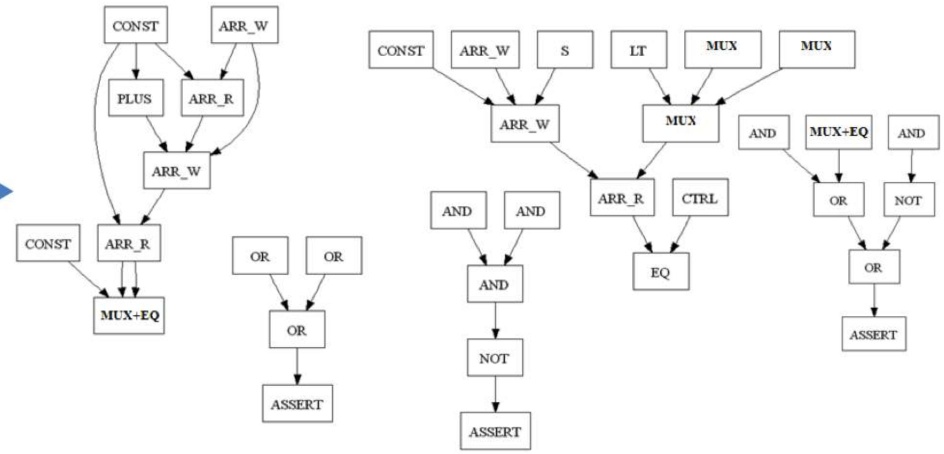


Corpus of Formulas (DAGs)

Patterns (Sub-formulas)

Pattern Finding



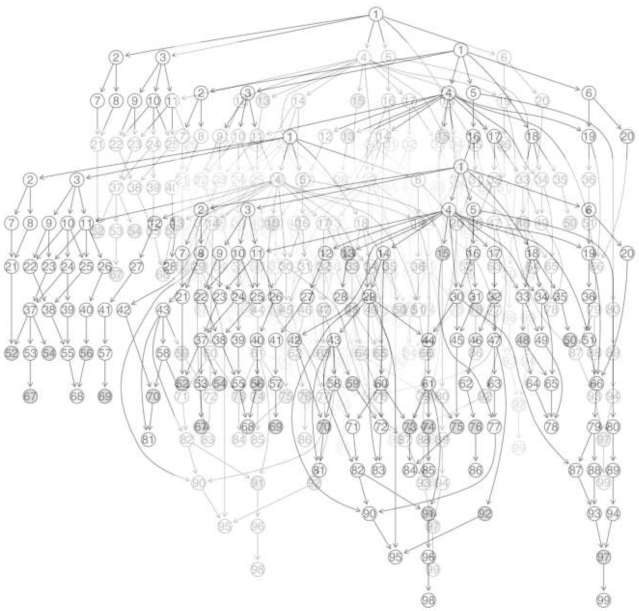


Corpus of Formulas (DAGs)

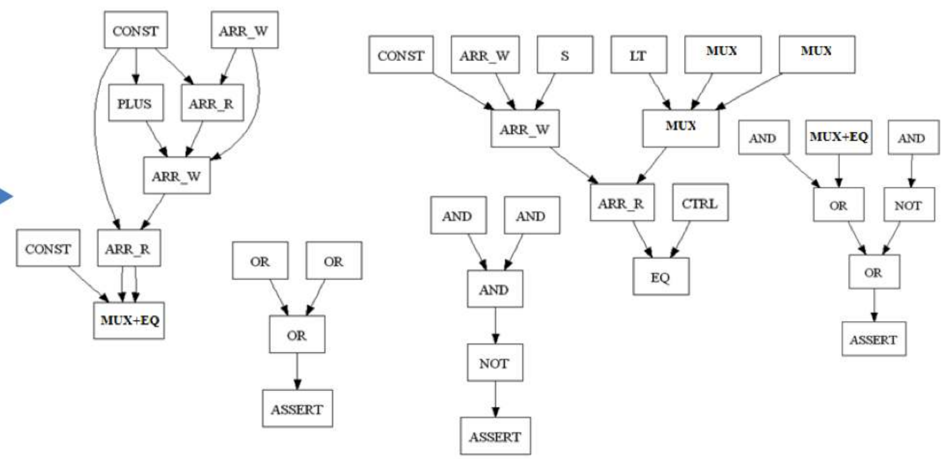
Patterns (Sub-formulas)

Commonly Occurring Patterns ⇒ More applicable rules

Pattern Finding



Corpus of Formulas
(DAGs)

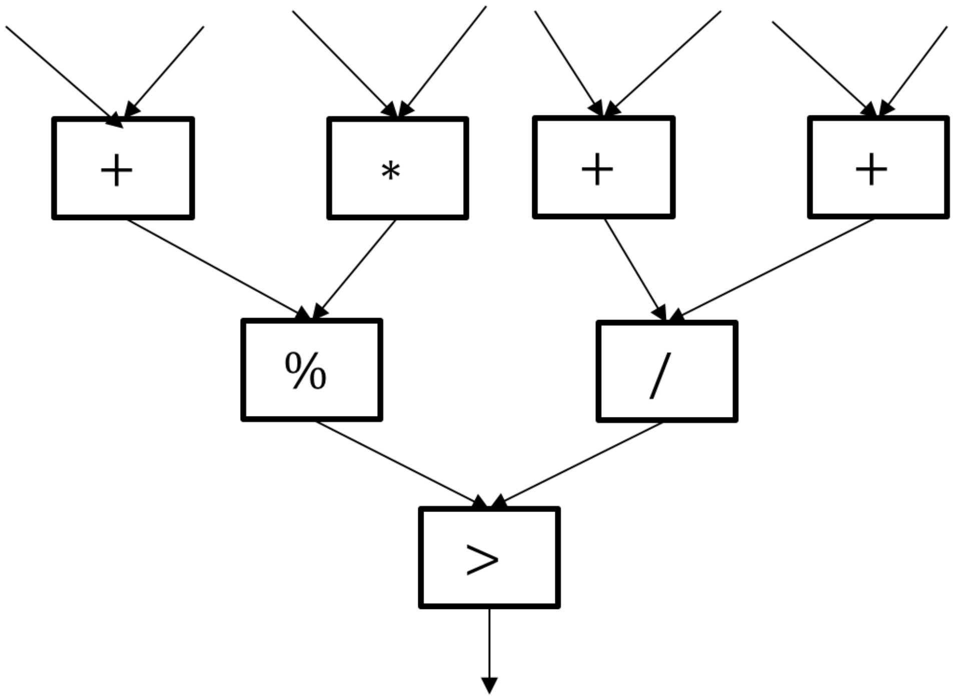


Patterns
(Sub-formulas)

Commonly Occurring Patterns ⇒ More applicable rules

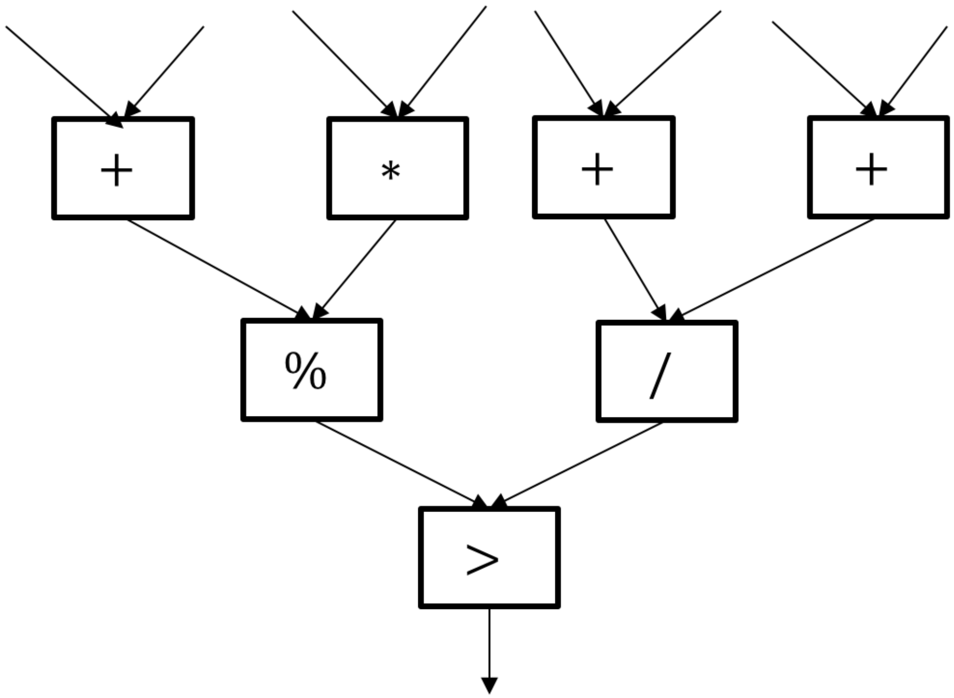
Different from Motif Discovery

Representative Sampling



Formula Trees

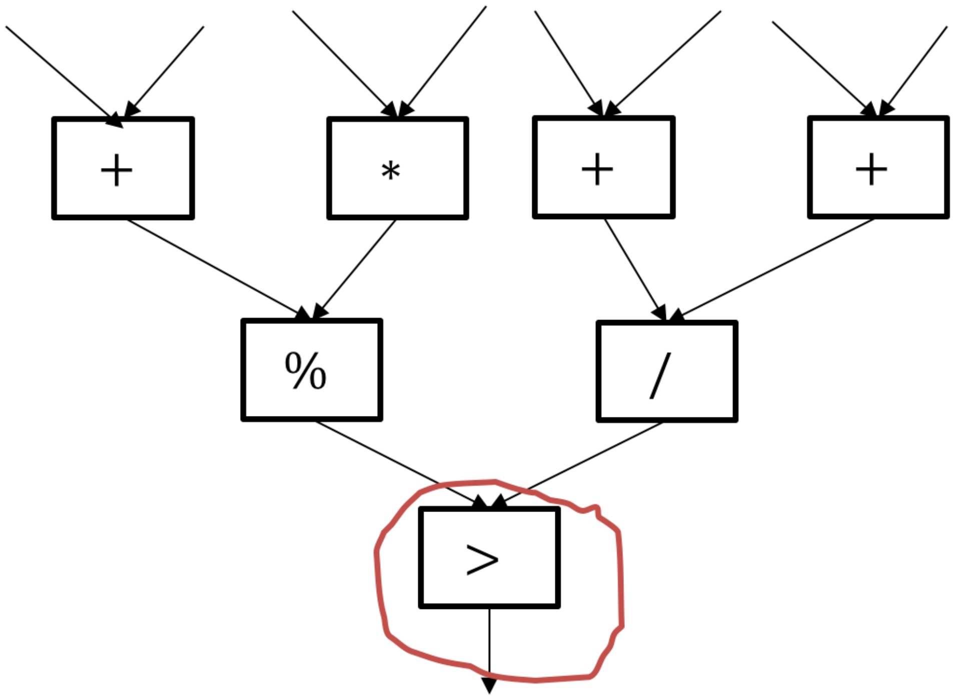
Representative Sampling



$$\frac{1}{|nodes|}$$

Formula Trees

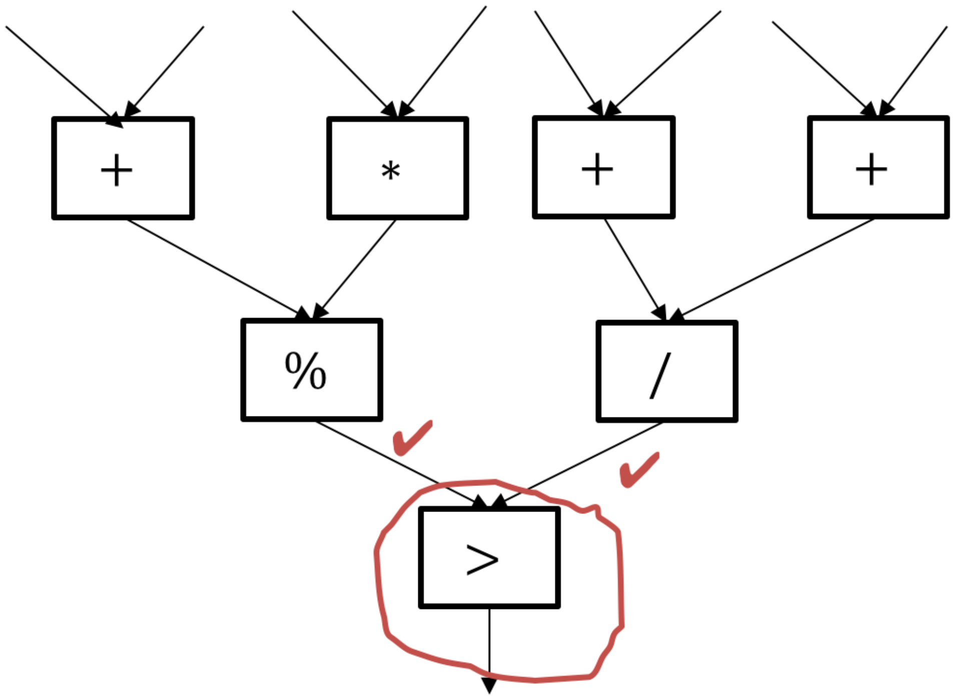
Representative Sampling



$$\frac{1}{|nodes|}$$

Formula Trees

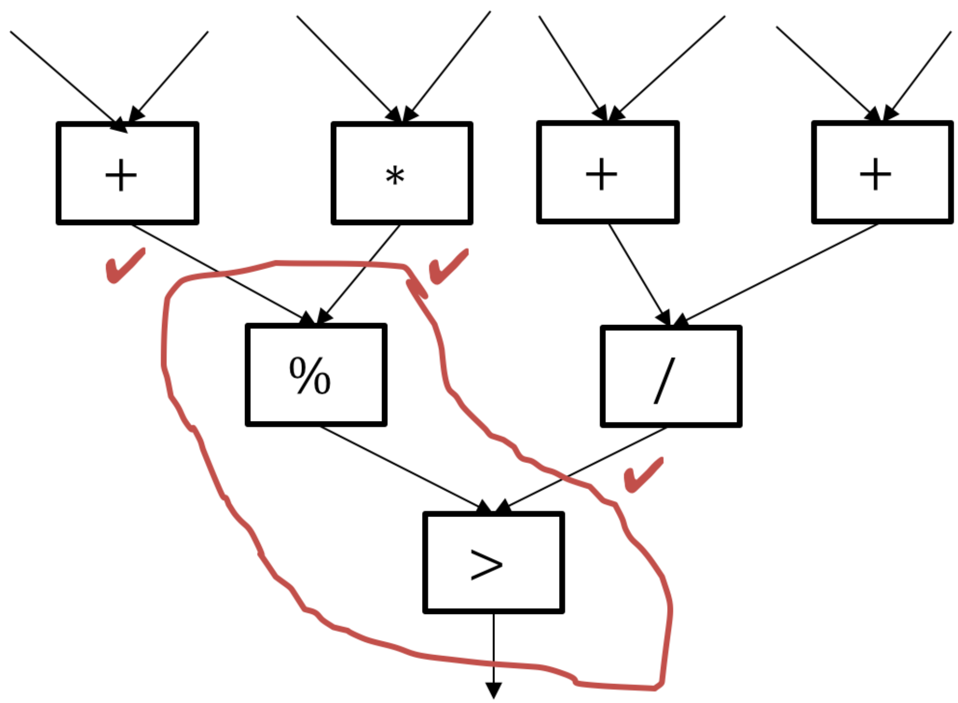
Representative Sampling



$$\frac{1}{|nodes|} \times \frac{1}{2}$$

Formula Trees

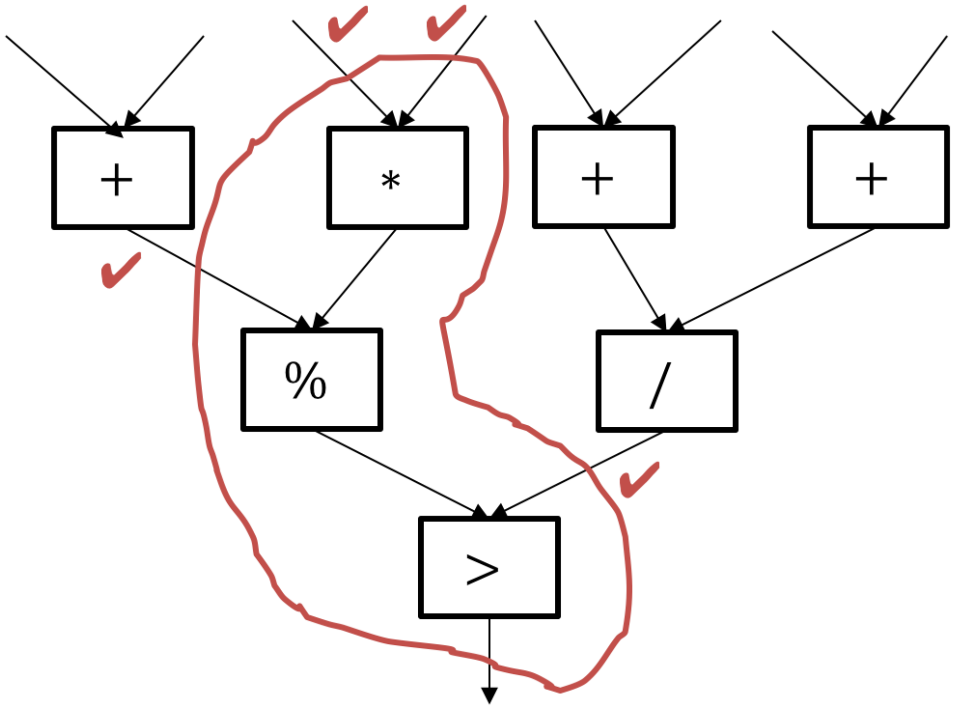
Representative Sampling



$$\frac{1}{|nodes|} \times \frac{1}{2} \times \frac{1}{3}$$

Formula Trees

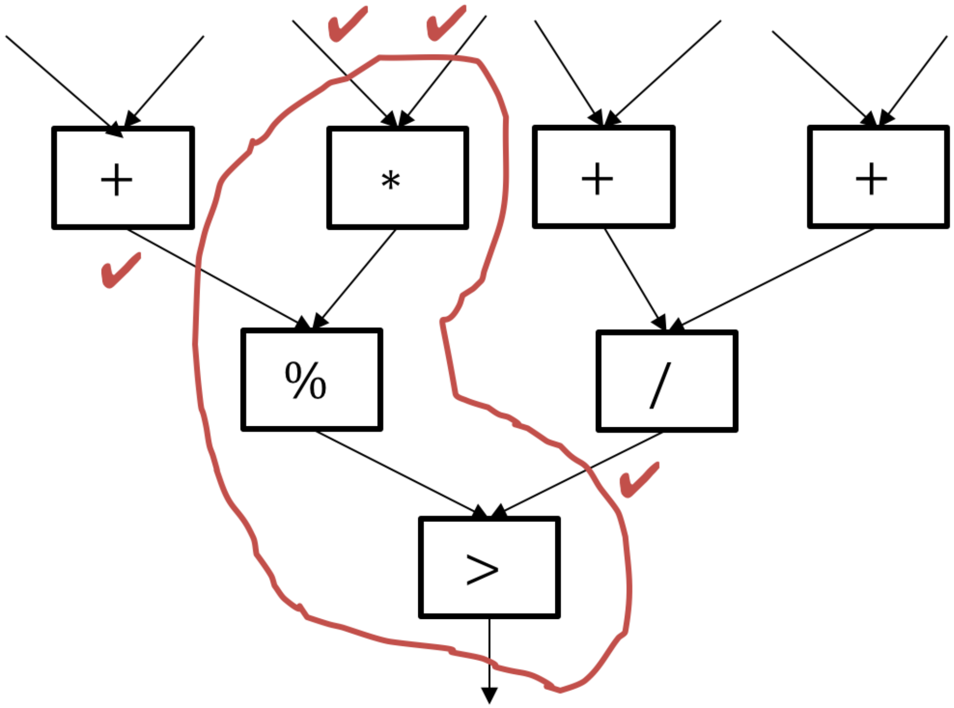
Representative Sampling



$$\frac{1}{|nodes|} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4} \cdots$$

Formula Trees

Representative Sampling



$$\frac{1}{|nodes|} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4} \cdots$$

Formula Trees

Probability independent of structure

Representative Sampling

- Naïve algorithm
  - Sample a node at random
  - Maintain a set of "boundary" edges
  - Sample from the boundary and repeat

Representative Sampling

- Naïve algorithm
  - Sample a node at random
  - Maintain a set of “boundary” edges
  - Sample from the boundary and repeat
- Works for K-ary trees

Pattern Finding: Sampling

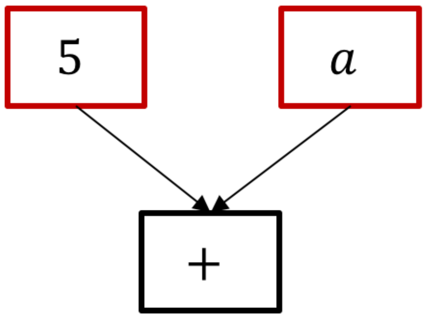
- Issues:

Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)

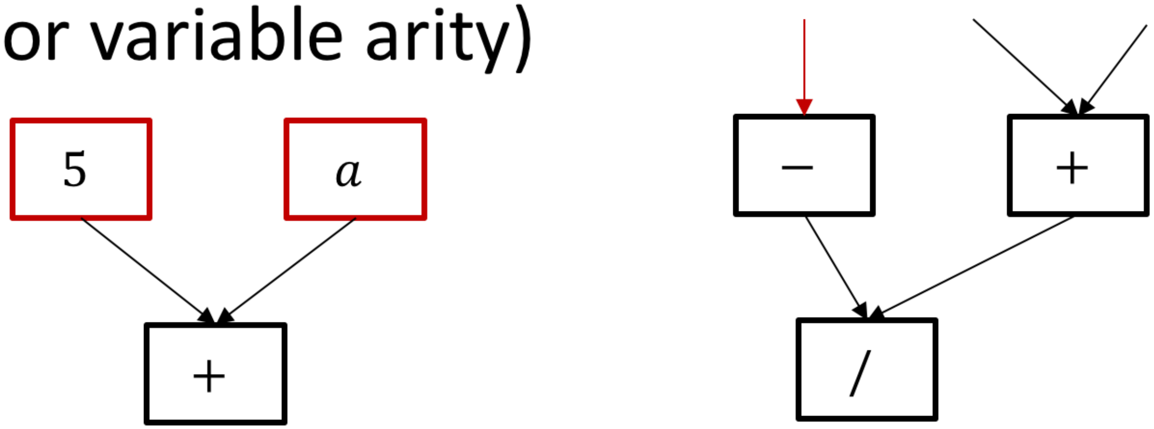
Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)



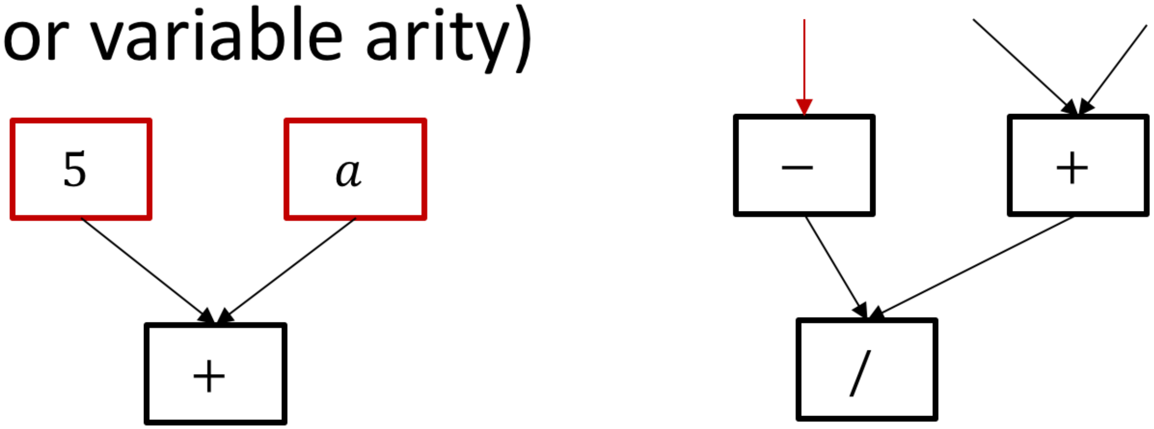
Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)



Pattern Finding: Sampling

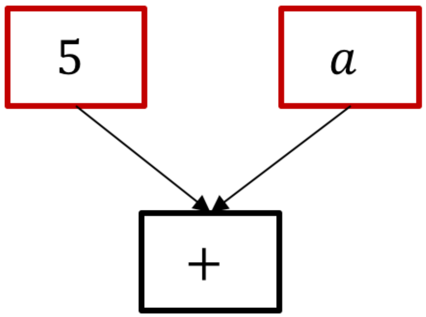
- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)

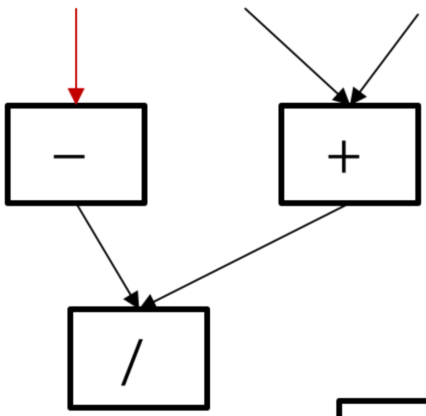


  - For DAGs, Finding same pattern in multiple ways

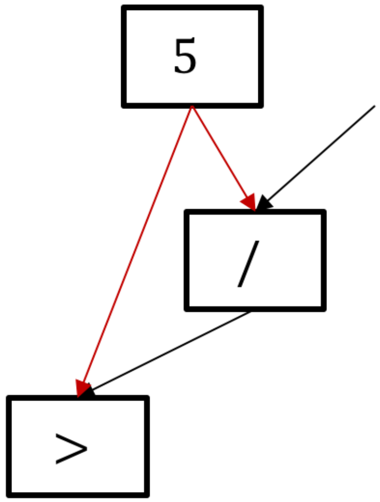
Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)



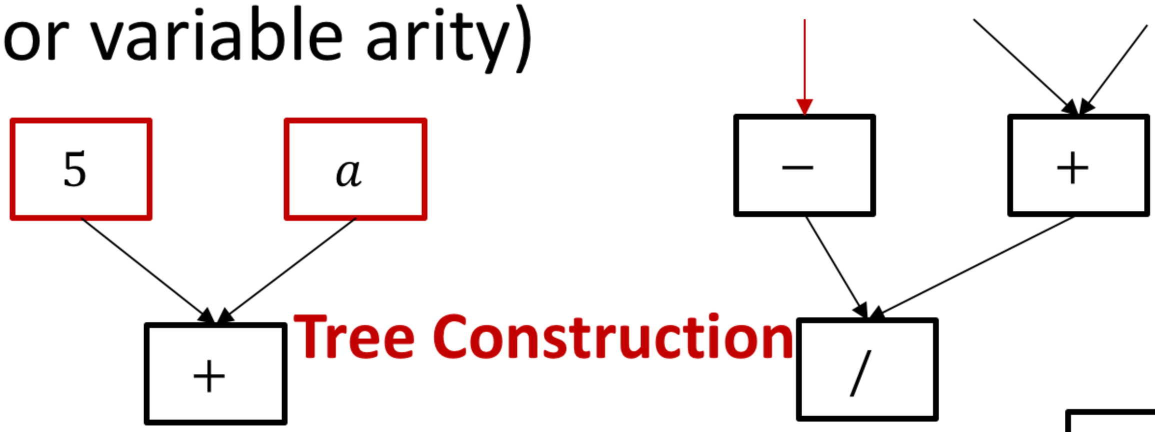


  - For DAGs, Finding same pattern in multiple ways

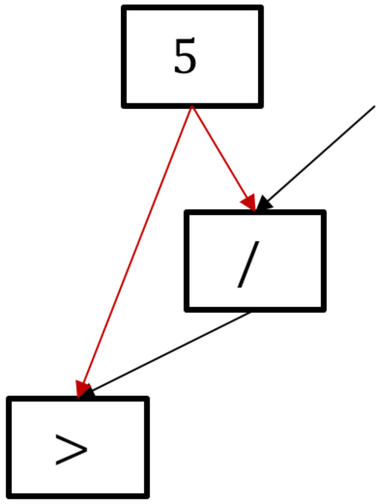


Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)

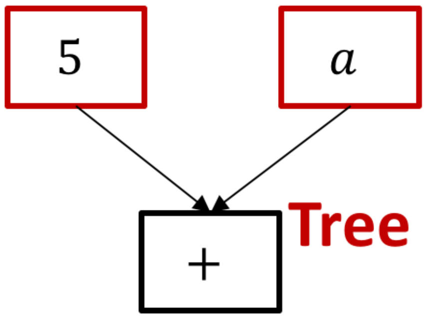


  - For DAGs, Finding same pattern in multiple ways

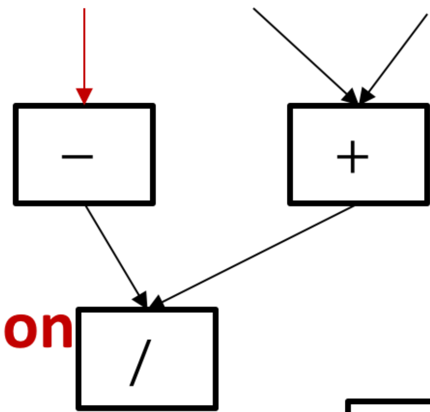


Pattern Finding: Sampling

- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)

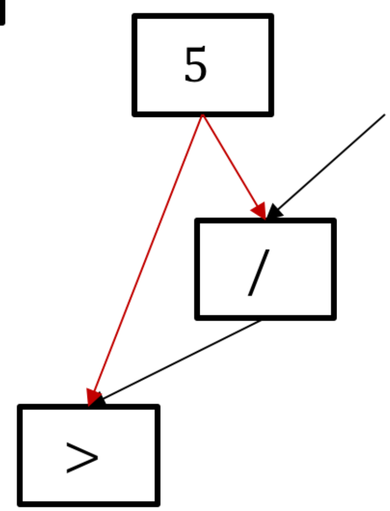


Tree Construction



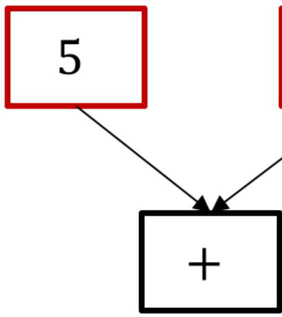
  - For DAGs, Finding same pattern in multiple ways

BFS Ordering

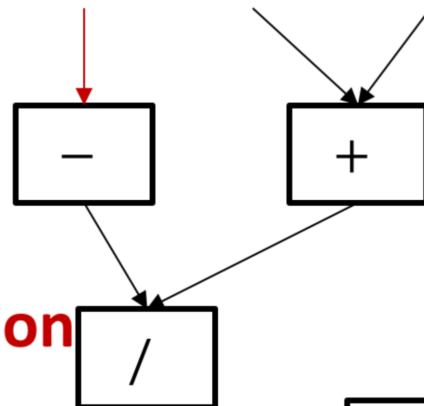


Pattern Finding: Sampling

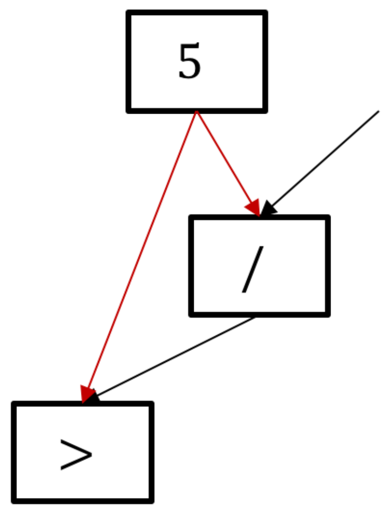
- Issues:
  - Dealing with missing edges (e.g. reaching top-most nodes or variable arity)



Tree Construction



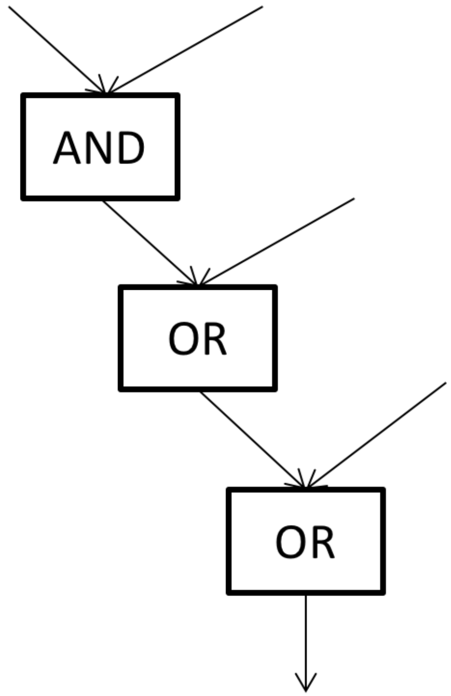
  - For DAGs, Finding same pattern in multiple ways



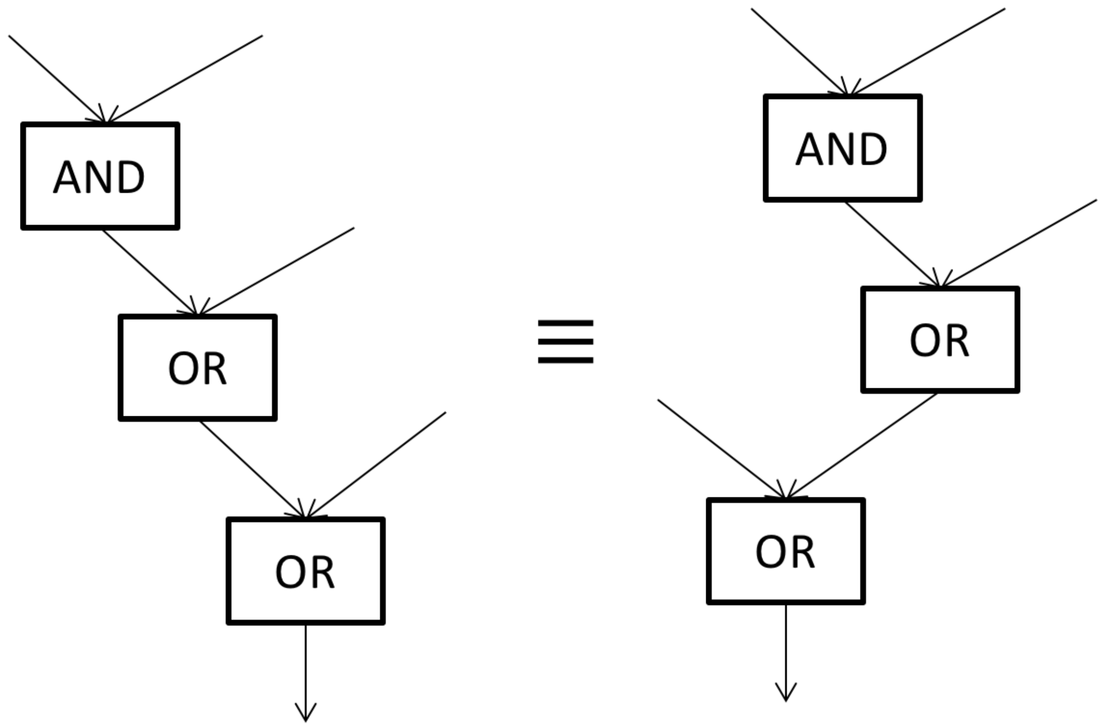
BFS Ordering

Use Rejection Sampling

Pattern Finding: Book-keeping

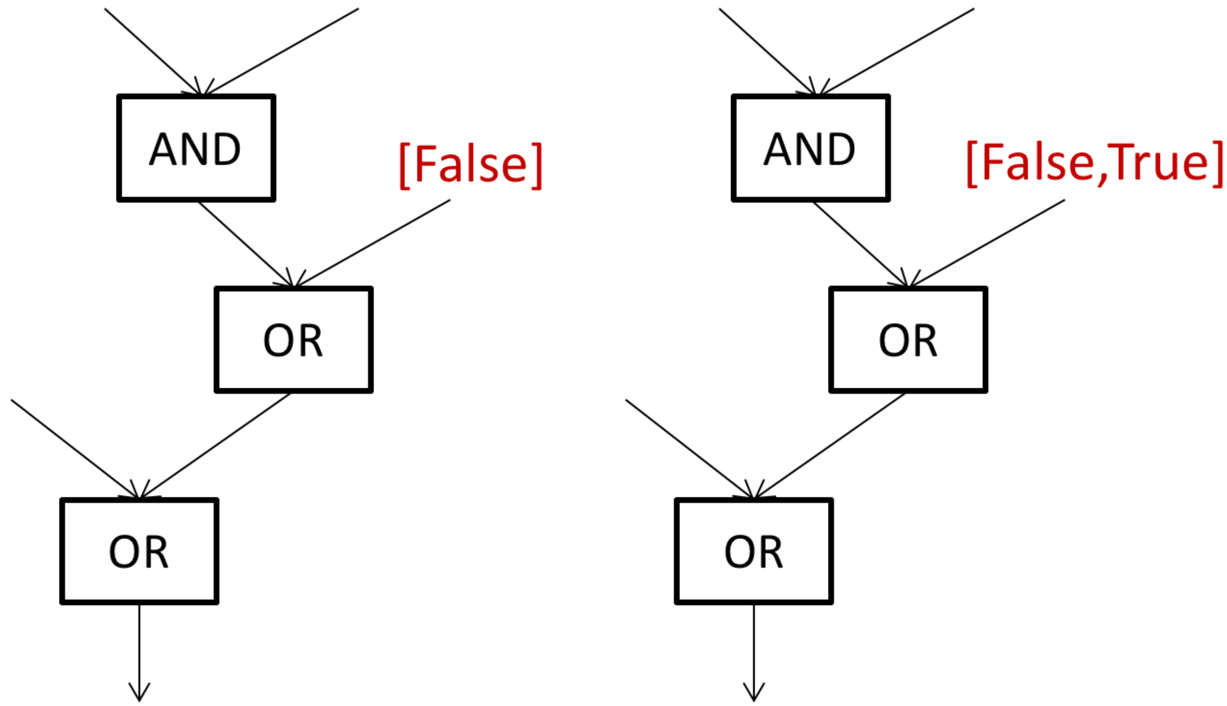


Pattern Finding: Book-keeping



- Aggregation modulo symmetries

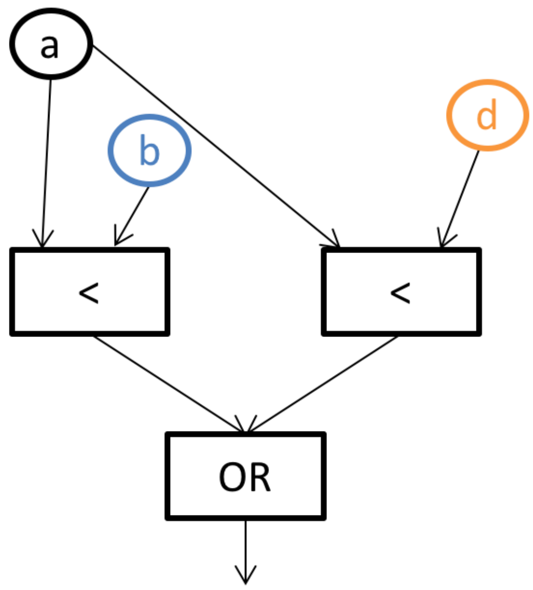
Pattern Finding: Book-keeping



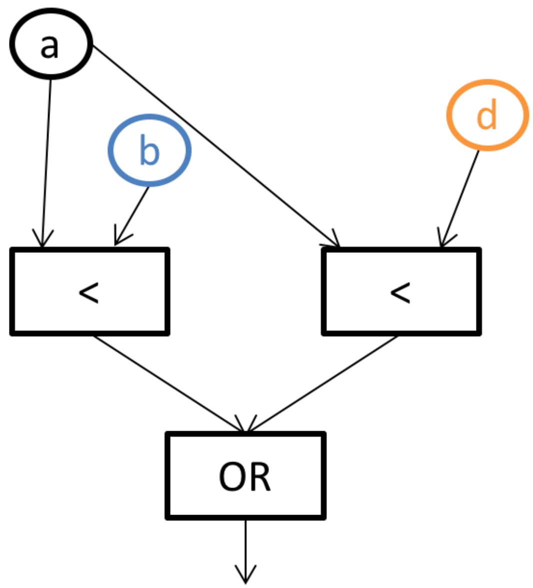
- Aggregation modulo symmetries
- Handling contextual information around formulas

Contextual Information: *static()*

Contextual Information: *static()*



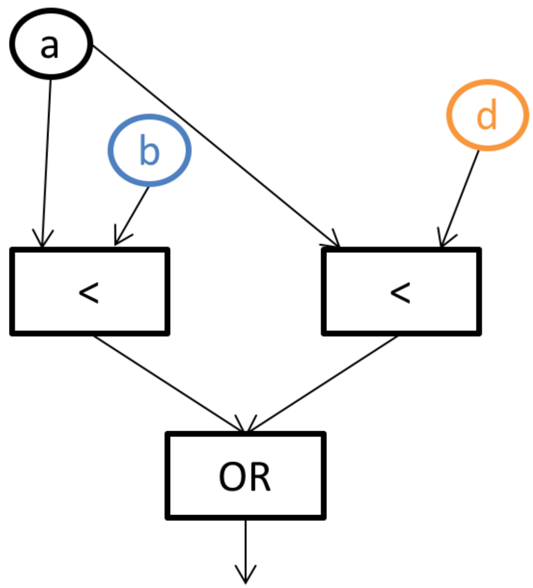
Contextual Information: *static()*



$$\boldsymbol{static(b) = (-\infty, 0]}$$

$$\boldsymbol{static(d) = (0, \infty)}$$

Contextual Information: *static()*



$$static(b) = (-\infty, 0]$$

$$static(d) = (0, \infty)$$

Can infer strong assumptions like:

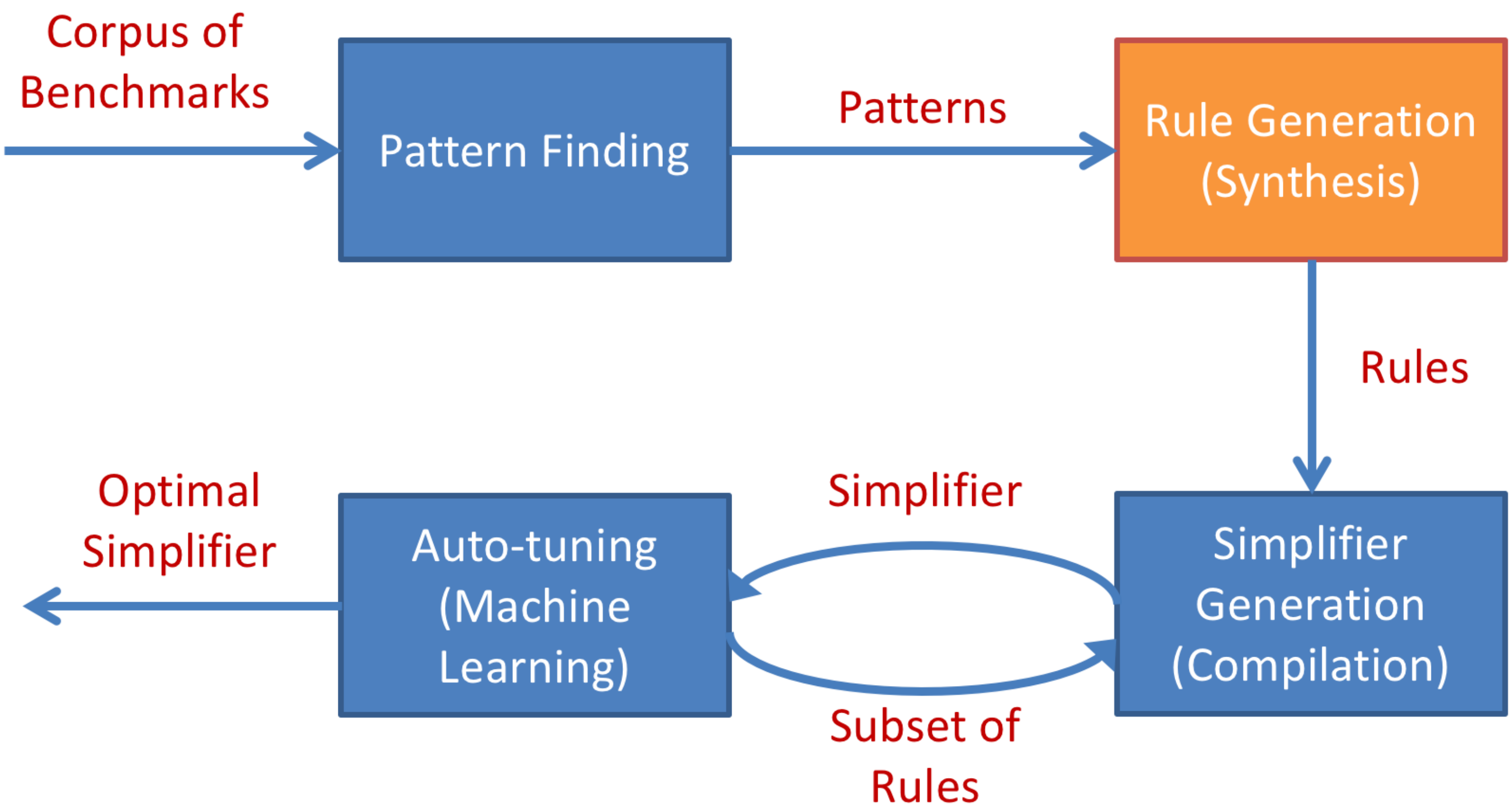
$b < d$

$b \neq d$

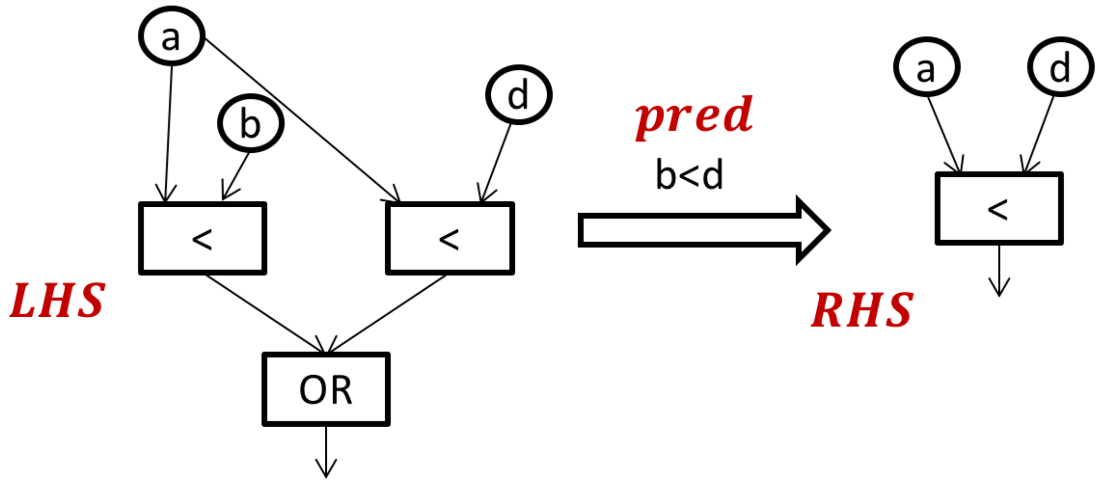
$b \leq 0$

$0 < d$

SWAPPER framework



Conditional Rewrite Rules



- **Inputs (x) :** ⓐ ⓑ ⓓ
- $\forall x \; pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$

Rule Generation: The Problem

- Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:

Rule Generation: The Problem

- Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:
  - $\forall x : static(x) \Rightarrow pred(x)$

Rule Generation: The Problem

- Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:
  - $\forall x : static(x) \Rightarrow pred(x)$
  - $\forall x : pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$

Rule Generation: The Problem

- Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:
  - $\forall x : static(x) \Rightarrow pred(x)$
  - $\forall x : pred(x) \Rightarrow \left(LHS(x) == RHS(x)\right)$
  - $size(RHS) < size(LHS)$

Rule Generation: The Problem

- Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:
  - $\forall x : static(x) \Rightarrow pred(x)$
  - $\forall x : pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$
  - $size(RHS) < size(LHS)$
  - $pred(x)$ is one of the weakest (most permissive) candidates

Rule Generation: Grammars

Rule Generation: Grammars

- **Grammar for $pred(x)$:**

Rule Generation: Grammars

- **Grammar for $\boldsymbol{pred(x)}$:**

$$pred(x) \equiv x_i \ \mathbf{binop}\ x_j \text{ for integer } x_i, x_j$$
$$| \ x_i \ | \ \neg x_j \ \text{ for boolean } x_i, x_j$$
$$| \ \mathbf{True}$$
$$x = (x_1, x_2, \ldots, x_n) \text{ and } 1 \leq i \neq j \leq n$$

Rule Generation: Grammars

- **Grammar for $\boldsymbol{pred(x)}$:**

$$pred(x) \equiv x_i \ \textbf{binop} \ x_j \text{ for integer } x_i, x_j$$

$$\mid x_i \mid \neg x_j \ \text{ for boolean } x_i, x_j$$

$$\mid \textbf{True}$$

$$x = (x_1, x_2, \dots, x_n) \text{ and } 1 \le i \ne j \le n$$

$$\textbf{binop} \equiv < \mid > \mid \le \mid \ge \mid == \mid \ne$$

Rule Generation: Grammars

- **Grammar for $\boldsymbol{pred(x)}$:**

$$pred(x) \equiv x_i \ \mathbf{binop} \ x_j \text{ for integer } x_i, x_j$$
$$\mid x_i \mid \neg x_j \ \text{ for boolean } x_i, x_j$$
$$\mid \mathbf{True}$$
$$x = (x_1, x_2, \ldots, x_n) \text{ and } 1 \leq i \neq j \leq n$$
$$\mathbf{binop} \equiv < \mid > \mid \leq \mid \geq \mid == \mid \neq$$

- **Grammar for $\boldsymbol{RHS(x)}$:** complete DAGs

Rule Generation: Hybrid approach

Rule Generation: Hybrid approach

Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:

Rule Generation: Hybrid approach

Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:

- $\forall x : static(x) \Rightarrow pred(x)$
- $\forall x : pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$
- $size(RHS) < size(LHS)$

Rule Generation: Hybrid approach

Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:

- $\forall x : static(x) \Rightarrow pred(x)$
- $\forall x : pred(x) \Rightarrow \big(LHS(x) == RHS(x)\big)$
- $size(RHS) < size(LHS)$

$$\exists c_p c_r \forall x \begin{bmatrix} \big(static(x) \Rightarrow pred(x)\big) \\ \wedge\, pred(x, c_p) \Rightarrow \big(LHS(x) = RHS(x, c_r)\big) \end{bmatrix}$$

Classic Syntax-guided synthesis problem (Sketch)

Rule Generation: Hybrid approach

Given a **pattern** $LHS(x)$, **assumptions** $static(x)$ and **grammars** for $pred$ and RHS, find $pred(x)$, $RHS(x)$ such that:

- $\forall x : static(x) \Rightarrow pred(x)$
- $\forall x : pred(x) \Rightarrow (LHS(x) == RHS(x))$
- $size(RHS) < size(LHS)$
- $pred(x)$ is one of the weakest (most permissive) candidates

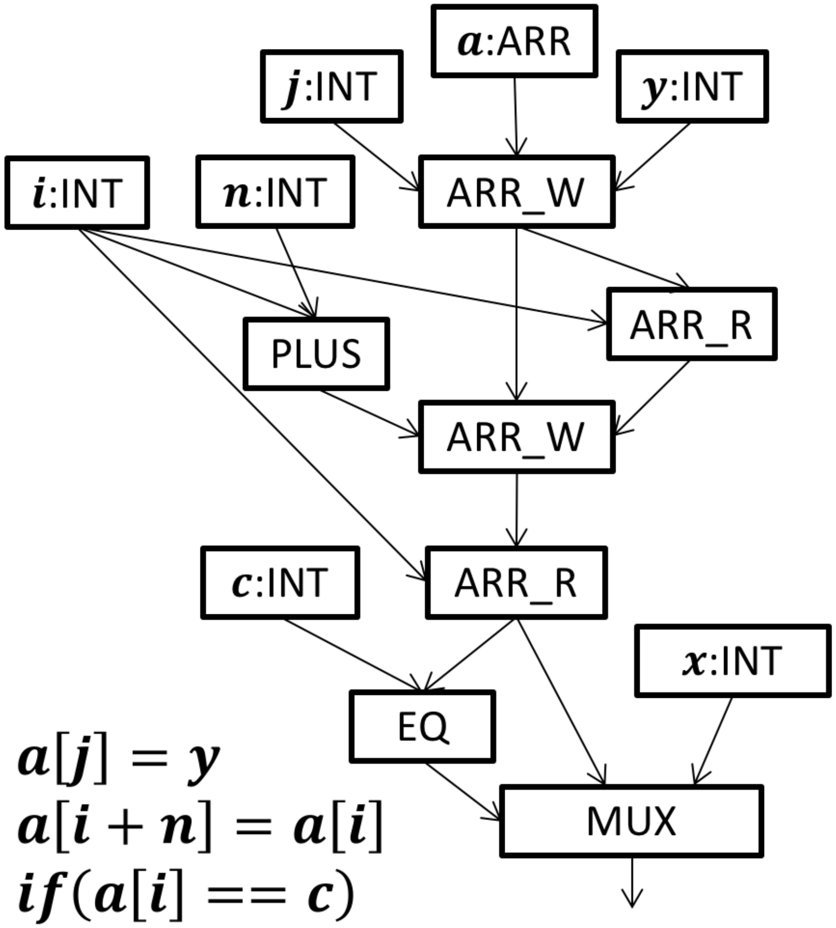
$$\exists c_p c_r \forall x \begin{bmatrix} (static(x) \Rightarrow pred(x)) \\ \wedge\, pred(x, c_p) \Rightarrow (LHS(x) = RHS(x, c_r)) \end{bmatrix}$$

Classic Syntax-guided synthesis problem (Sketch)

+ Enumerative predicate refinement

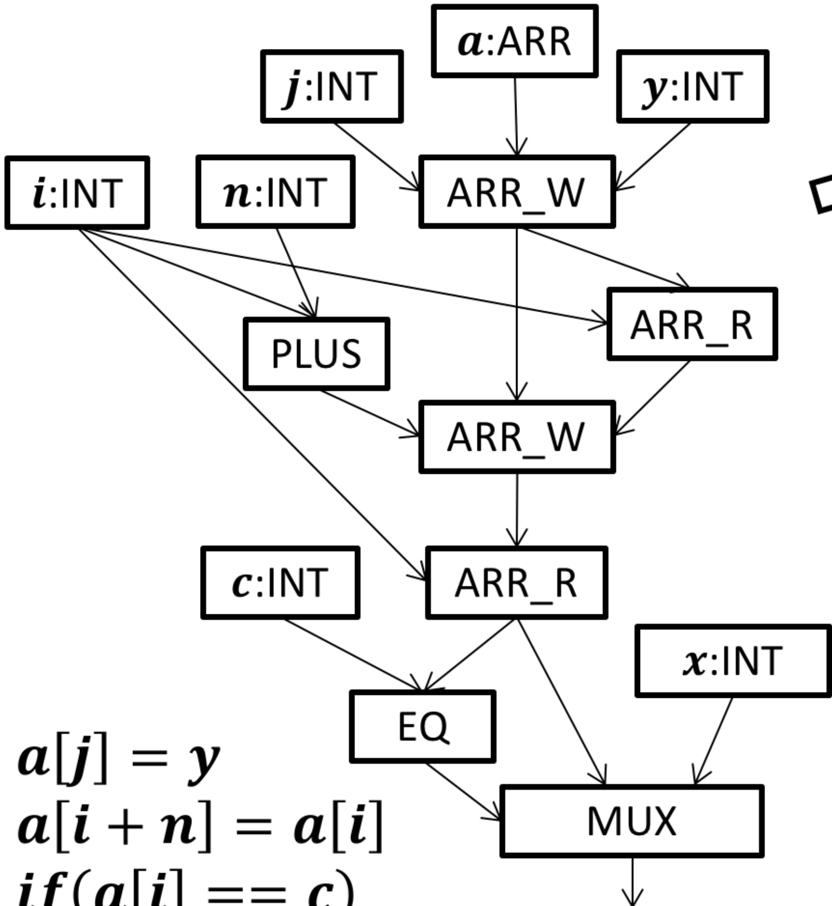
Rule Generation: Example

Rule Generation: Example



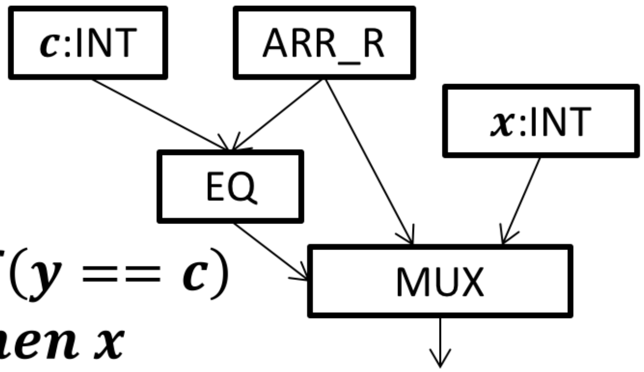
$a[j] = y$
$a[i+n] = a[i]$
$if(a[i] == c)$
$then\ \ x$
$else\ \ a[i]$

Rule Generation: Example



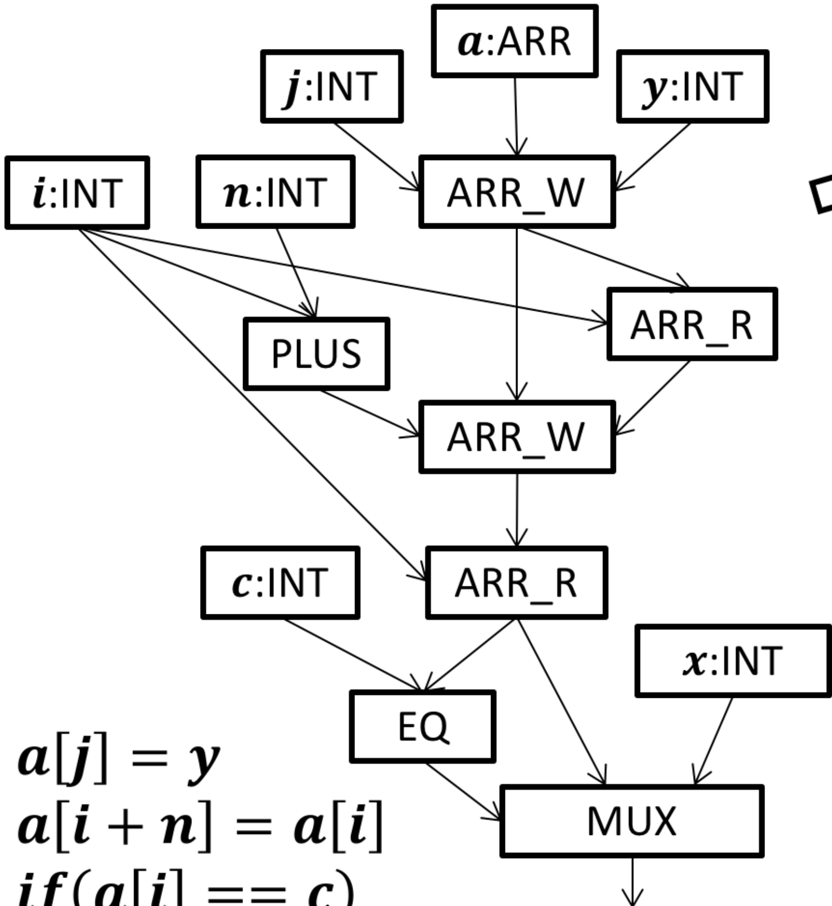
$a[j] = y$
$a[i+n] = a[i]$
$if (a[i] == c)$
$then\ x$
$else\ a[i]$

$i == j$



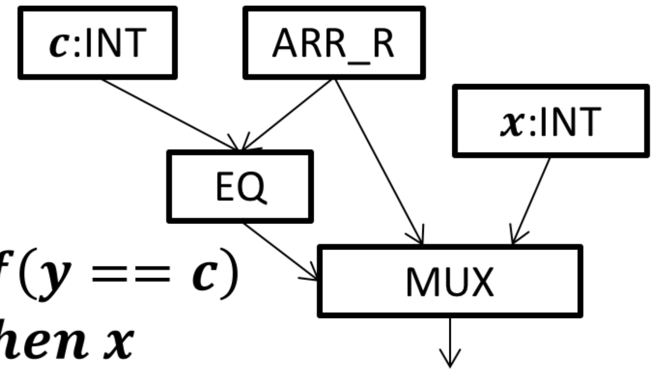
$if (y == c)$
$then\ x$
$else\ y$

Rule Generation: Example



$a[j] = y$
$a[i+n] = a[i]$
$if(a[i] == c)$
$then\ x$
$else\ a[i]$

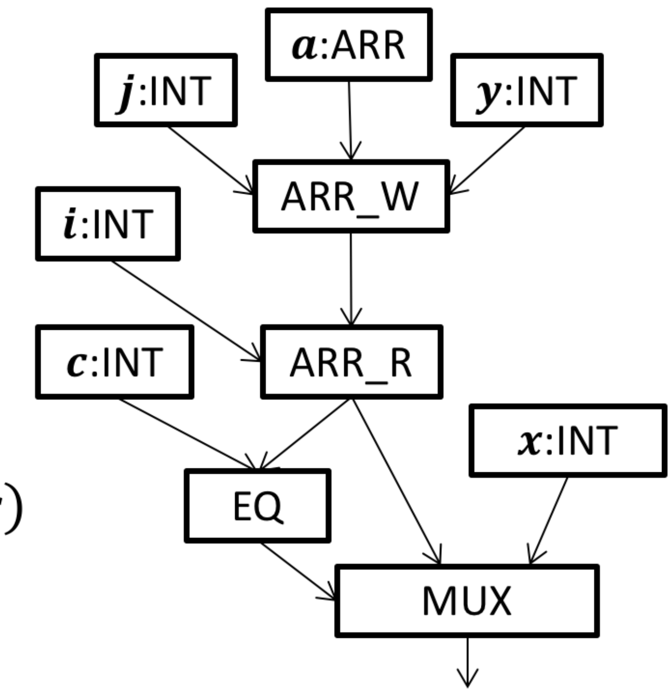
$i == j$



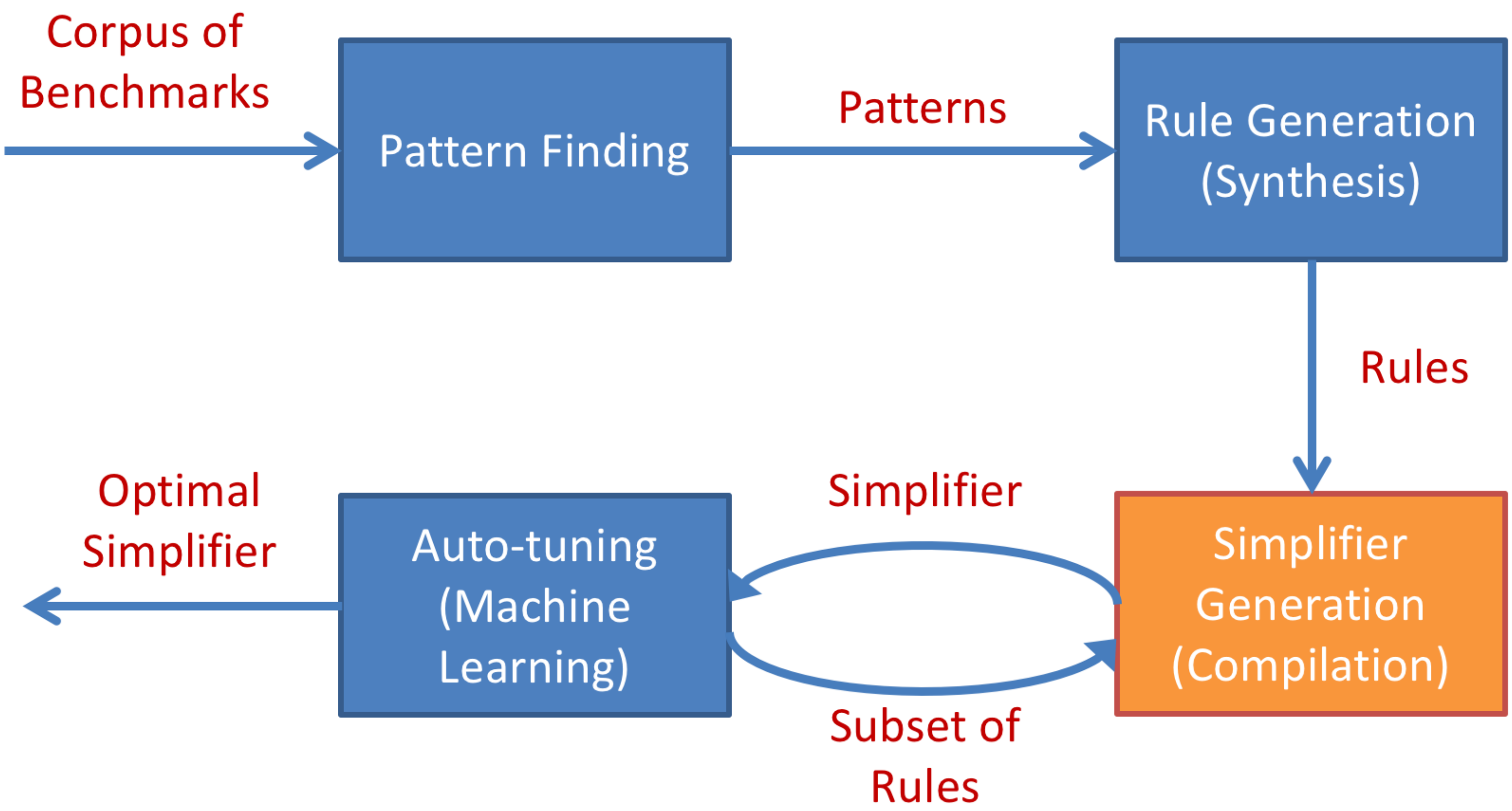
$if(y == c)$
$then\ x$
$else\ y$

True

$a[j] = y$
$if(a[i] == c)$
$then\ x$
$else\ a[i]$



SWAPPER framework



Simplifier Generation

- Default node traversal and rule application strategy
- Generate efficient C++ code

Simplifier Generation

- Default node traversal and rule application strategy
- Generate efficient C++ code
  - *Rule generalization*

Simplifier Generation

- Default node traversal and rule application strategy
- Generate efficient C++ code
  - *Rule generalization*
  - Incorporate *symmetries* of the rules

Simplifier Generation

- Default node traversal and rule application strategy
- Generate efficient C++ code
  - *Rule generalization*
  - Incorporate *symmetries* of the rules
  - Share pattern matching cost across rules

Simplifier Generation

- Default node traversal and rule application strategy
- Generate efficient C++ code
  - *Rule generalization*
  - Incorporate *symmetries* of the rules
  - Share pattern matching cost across rules
  - Fully verifying the rule at each stage

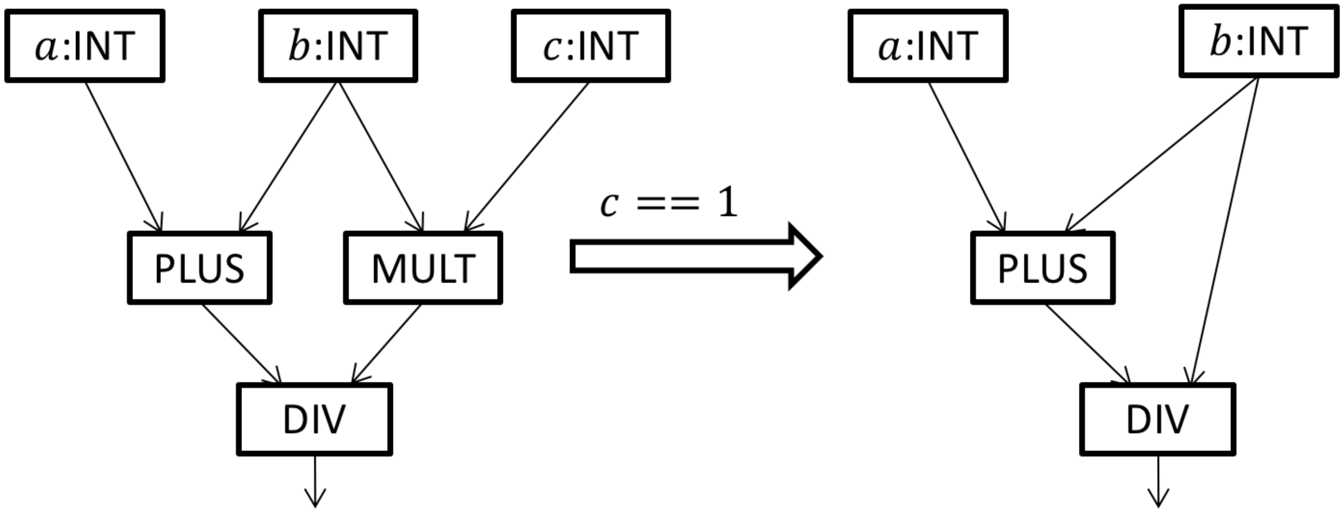
Simplifier Generation: Rule Generalization

Simplifier Generation: Rule Generalization

- Matching and removing sub-patterns

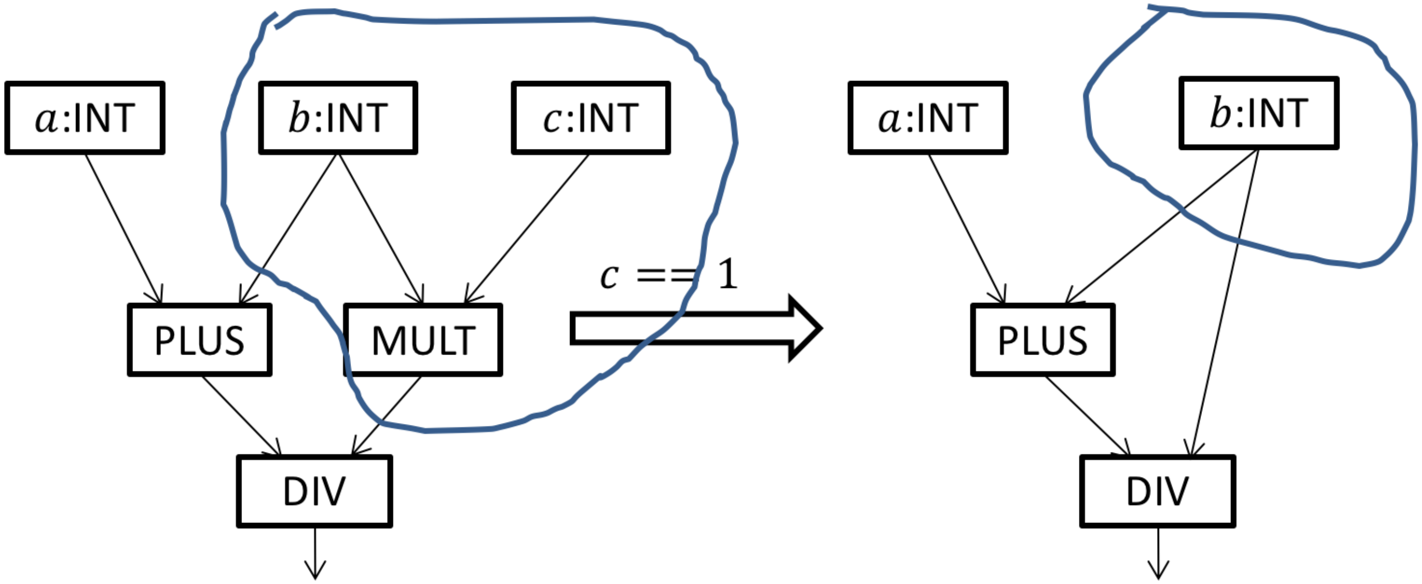
Simplifier Generation: Rule Generalization

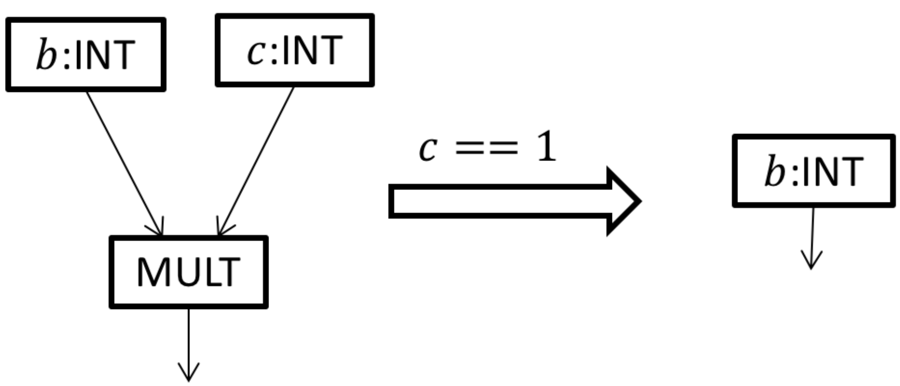
- Matching and removing sub-patterns



Simplifier Generation: Rule Generalization

- Matching and removing sub-patterns



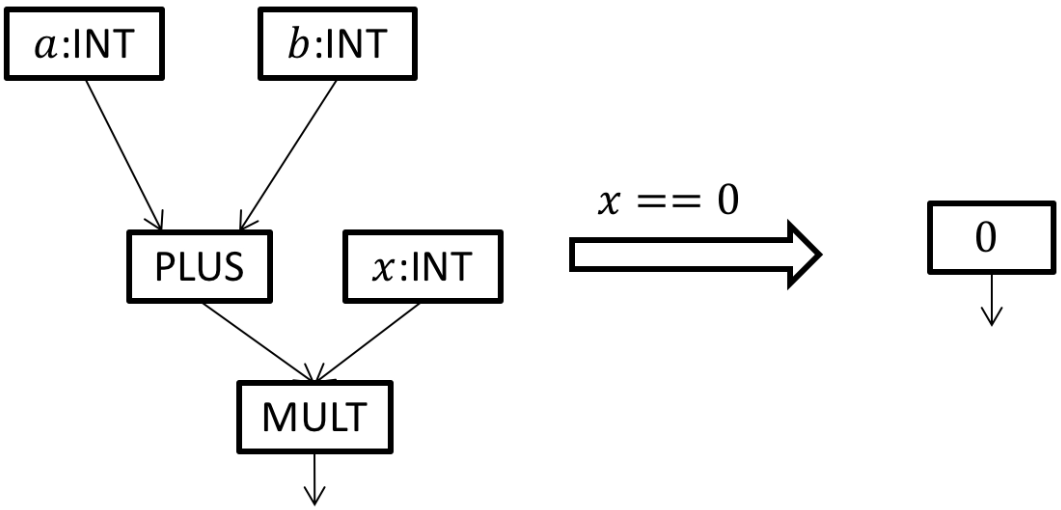


Simplifier Generation: Rule Generalization

- Replace sub-patterns by inputs recursively

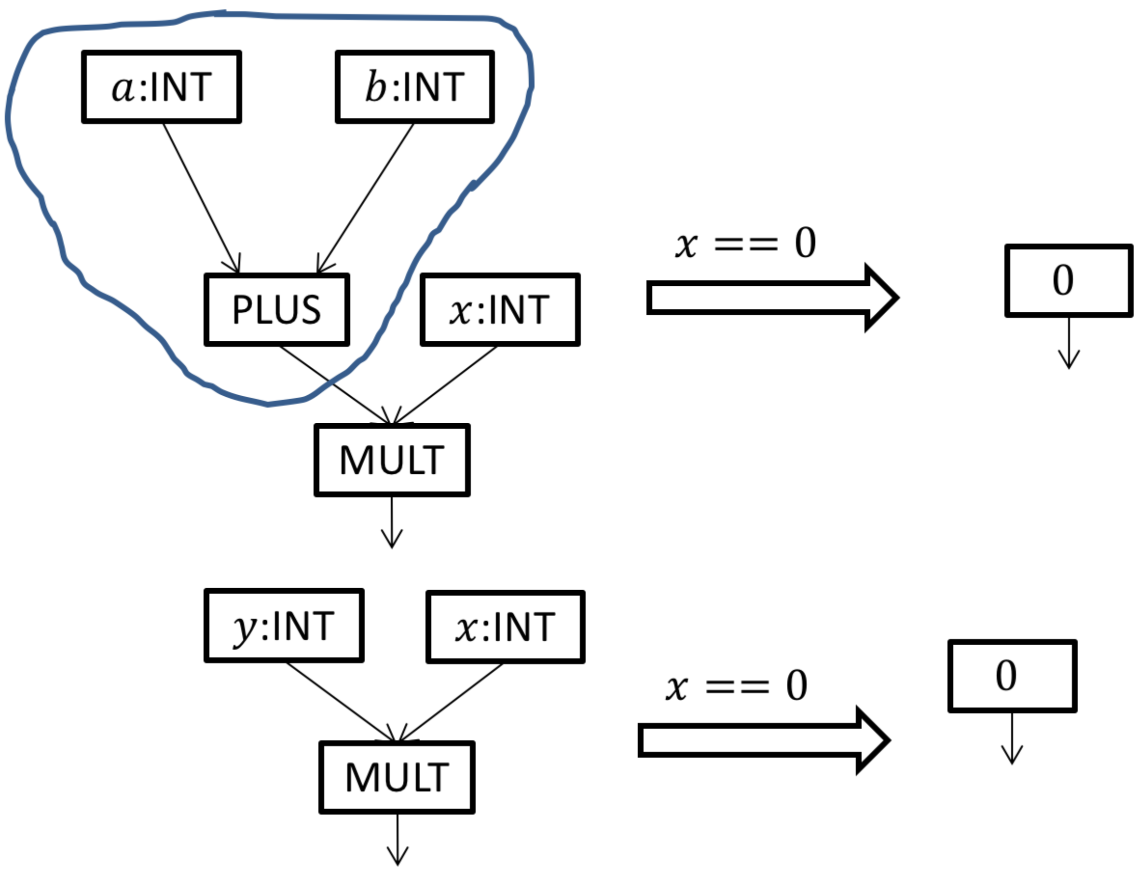
Simplifier Generation: Rule Generalization

- Replace sub-patterns by inputs recursively

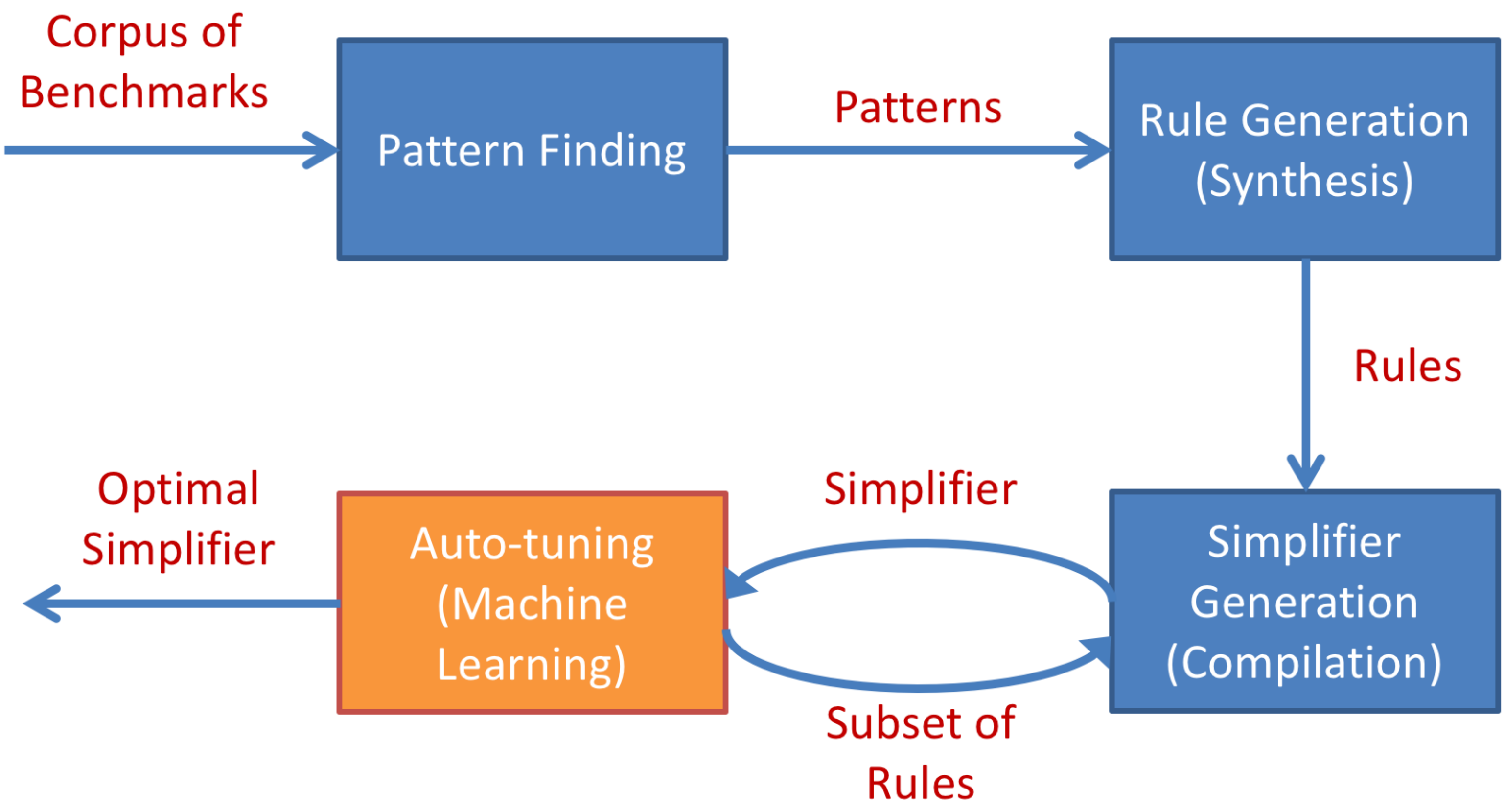


Simplifier Generation: Rule Generalization

- Replace sub-patterns by inputs recursively



SWAPPER framework



Auto tuning

Auto tuning

- Identifies the *best* subset of rules

Ansel et al, PACT 2014
http://opentuner.org

Auto tuning

- Identifies the *best* subset of rules
- Problem Setup:

Ansel et al, PACT 2014
http://opentuner.org

Auto tuning

- Identifies the *best* subset of rules
- Problem Setup:
  - Search space parameters:
    - Permutation of rules
    - Number of rules to be used
  - Optimization Function: Weighted Solution time

Ansel et al, PACT 2014
http://opentuner.org

Experiments

Domains & Benchmarks

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366
SAT Encodings	70	6504

Domains & Benchmarks

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366
SAT Encodings	70	6504

- Full evaluation was done on **AutoGrader** and **Sygus**

Domains & Benchmarks

Domain	Benchmark DAGs Used	Avg. Number of Terms
AutoGrader	45	23289
Sygus	22	68366
SAT Encodings	70	6504

- Full evaluation was done on **AutoGrader** and **Sygus**
- Performed a validation case study on **SAT Encodings** benchmarks

Comparing Simplifiers

Comparing Simplifiers

- We compare the following simplifiers:

Comparing Simplifiers

- We compare the following simplifiers:
  - **Hand-coded**: default in Sketch

Comparing Simplifiers

- We compare the following simplifiers:
  - **Hand-coded**: default in Sketch
  - **Baseline**: disables all rules except constant propagation

Comparing Simplifiers

- We compare the following simplifiers:
  - **Hand-coded**: default in Sketch
  - **Baseline**: disables all rules except constant propagation
  - **Auto-generated**: Baseline + generated rules

SWAPPER performance

SWAPPER performance

- Divided the corpus as (SEARCH,TRAIN,TEST)

SWAPPER performance

- Divided the corpus as (SEARCH,TRAIN,TEST)
  - SEARCH: Pattern Finding

SWAPPER performance

- Divided the corpus as (SEARCH,TRAIN,TEST)
  - SEARCH: Pattern Finding
  - TRAIN,TEST: Rule generation, Auto-tuning

SWAPPER performance

- Divided the corpus as (SEARCH,TRAIN,TEST)
  - SEARCH: Pattern Finding
  - TRAIN,TEST: Rule generation, Auto-tuning
  - Two-fold cross validation to avoid over-fitting

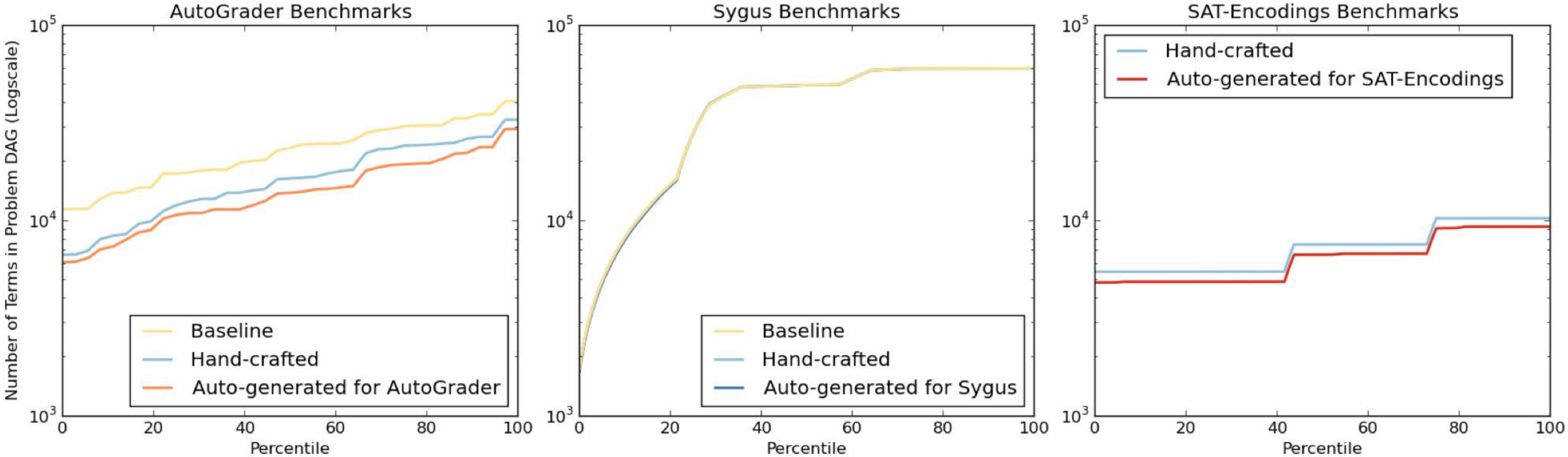
SWAPPER performance

- Divided the corpus as (SEARCH,TRAIN,TEST)
  - SEARCH: Pattern Finding
  - TRAIN,TEST: Rule generation, Auto-tuning
  - Two-fold cross validation to avoid over-fitting
    - For **AutoGrader** and **Sygus** domains

SWAPPER : Generated Rules

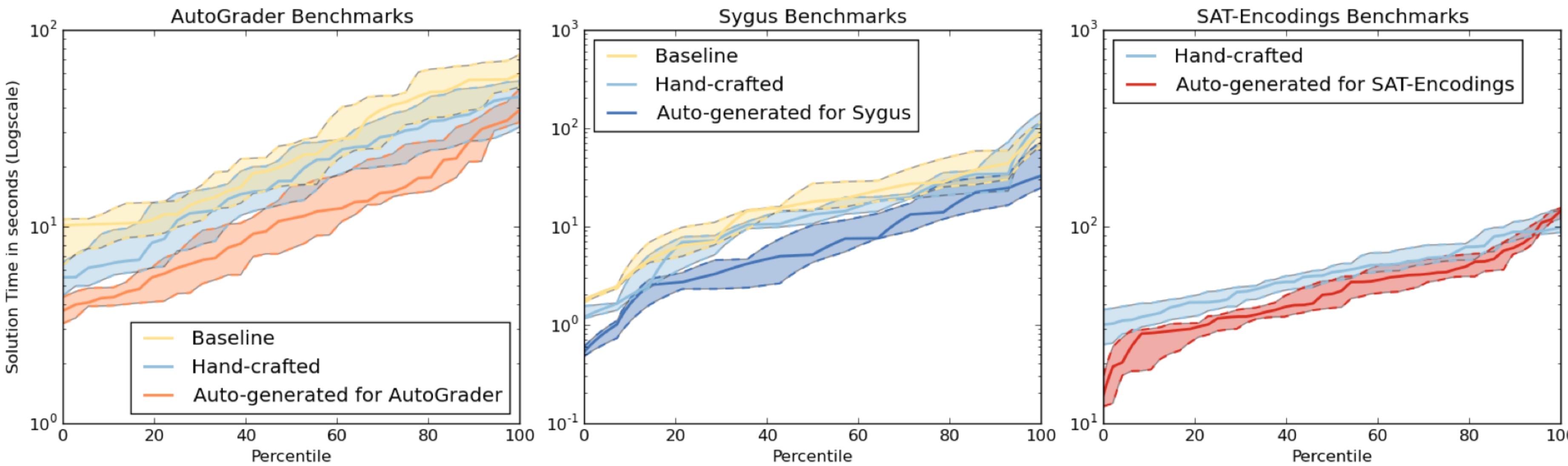
	AutoGrader	Sygus	SAT Encodings
Obtained	**201**	**163**	**117**
Optimal	**135**	**65**	**68**

Impact on Sizes



- **AutoGrader**: 13.8% reduction
- **Sygus**: 1.1% reduction
- **SAT Encodings**: 11% reduction

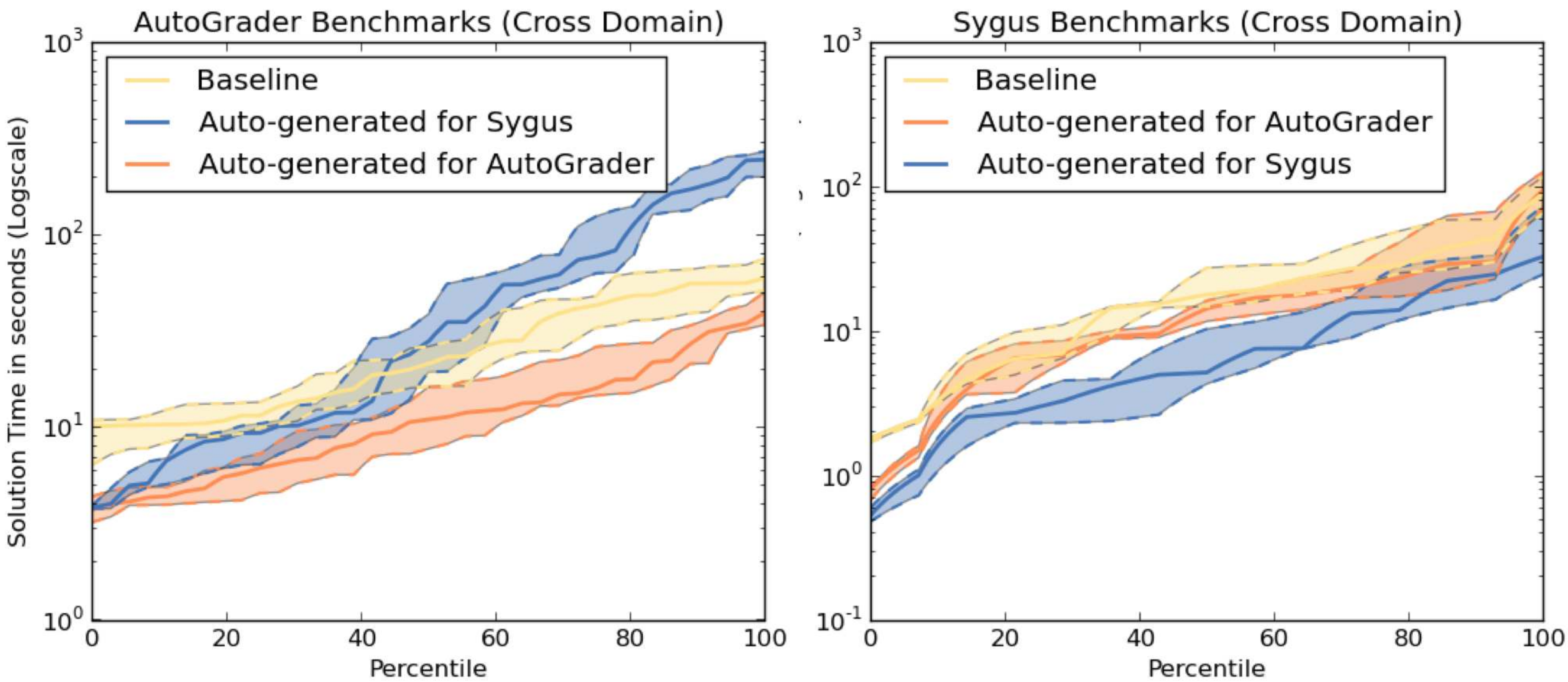
Impact on Running times



- Medians with quartile confidence intervals
- **AutoGrader**: 21s → 13s average times
- **Sygus**: 20s → 8s average times
- **SAT Encodings**: 59s → 51s average times

Domain Specificity

Impact on times across domains



Realistic Time and Costs

Realistic Time and Costs

Time and Cost Estimation (on AWS, parallelism of 40 threads)

Domain	Pattern Finding	Rule Generation	Auto-Tuning	Total Time (in hours)	Cost
AutoGrader	3 hours	1 hour $\times$ 5	0.08×150	20	$21.28
Sygus	2 hours	1 hour $\times$ 5	0.1×150	22	$23.42

Realistic Time and Costs

Time and Cost Estimation (on AWS, parallelism of 40 threads)

Domain	Pattern Finding	Rule Generation	Auto-Tuning	Total Time (in hours)	Cost
AutoGrader	3 hours	1 hour $\times$ 5	0.08×150	20	$21.28
Sygus	2 hours	1 hour $\times$ 5	0.1×150	22	$23.42

Costs less than an hour's work of a good developer

Realistic Time and Costs

Time and Cost Estimation (on AWS, parallelism of 40 threads)

Domain	Pattern Finding	Rule Generation	Auto-Tuning	Total Time (in hours)	Cost
AutoGrader	3 hours	1 hour $\times$ 5	0.08×150	20	$21.28
Sygus	2 hours	1 hour $\times$ 5	0.1×150	22	$23.42

Costs less than an hour's work of a good developer

Can reduce time by increasing parallelism or smarter evaluations with timeouts

Conclusion

Conclusion

- SWAPPER can generate good simplifiers in reasonable time and low cost.

Conclusion

- SWAPPER can generate good simplifiers in reasonable time and low cost.
- SWAPPER generated simplifiers perform better than hand written simplifier in Sketch.

Conclusion

- SWAPPER can generate good simplifiers in reasonable time and low cost.
- SWAPPER generated simplifiers perform better than hand written simplifier in Sketch.
- SWAPPER generated Simplifiers are highly domain specific

Conclusion

- SWAPPER can generate good simplifiers in reasonable time and low cost.
- SWAPPER generated simplifiers perform better than hand written simplifier in Sketch.
- SWAPPER generated Simplifiers are highly domain specific
- Part of a broader agenda to automatically generate parts of constraint solvers

Conclusion

- SWAPPER can generate good simplifiers in reasonable time and low cost.
- SWAPPER generated simplifiers perform better than hand written simplifier in Sketch.
- SWAPPER generated Simplifiers are highly domain specific
- Part of a broader agenda to automatically generate parts of constraint solvers

Thank You!