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

Rohit Singh      Armando Solar-Lezama
Massachusetts Institute of Technology

Mountain View, CA, USA
General Constraint Solvers

Solver
General Constraint Solvers

- Interprocedural Analysis
- Test Case Generation
- Software Replay
- Synthesis

Solver

Snugglebug
Klee
BBR
Synquid
General Constraint Solvers

Interprocedural Analysis

Test Case Generation

Software Replay

Synthesis

Snugglebug

Simplifier

Klee

Simplifier

BBR

Simplifier

Synquid

Solver
General Constraint Solvers

- Interprocedural Analysis
- Test Case Generation
- Software Replay
- Synthesis

- Snugglebug
- Klee
- BBR
- Synquid

Solver

- Simplifier
General Constraint Solvers

Interprocedural Analysis
Test Case Generation
Software Replay
Synthesis

Snugglebug
Simplifier
Klee
Simplifier
BBR
Simplifier
Synquid
Simplifier

Solver
Simplifier

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 \( \exists \forall \) formulas
Target: Sketch Solver

Sketch
\( \exists \forall \text{ formulas} \)

Simplifier
Target: Sketch Solver

AutoGrader (Python, edX)

SyGuS Competition

SAT Solver Encodings

Sketch $\exists \forall$ formulas

Simplifier
Target: Sketch Solver

AutoGrader (Python, edX)

SyGuS Competition

SAT Solver Encodings

\[ \exists \forall \text{ formulas} \]

Simplifier

Simplifier

Simplifier

\vdots

Auto-generate efficient domain-specific simplifiers
Sketch Simplifier

Messy low-level C++ code
Sketch Simplifier

Messy low-level C++ code

Employs simple declarative \textit{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

\[ \text{or} (\text{lt}(a, b), \text{lt}(a, d)) \]

Directed Acyclic Graphs
Conditional Rewrite Rules
Conditional Rewrite Rules

\[ \text{or}(\text{lt}(a,b), \text{lt}(a,d)) \quad \xrightarrow{b<d} \quad \text{lt}(a,d) \]
Conditional Rewrite Rules

\[ \text{or}(\text{lt}(a, b), \text{lt}(a, d)) \quad \overset{b<d}{\longrightarrow} \quad \text{lt}(a, d) \]
Conditional Rewrite Rules

\[ \text{or}(\text{lt}(a,b), \text{lt}(a,d)) \xrightarrow{b<d} \text{lt}(a,d) \]

- **Pattern**
- **Assumptions**
- **Pattern**
Conditional Rewrite Rules

Pattern: $\text{or}(\text{lt}(a, b), \text{lt}(a, d)) \quad \text{Assumptions} \quad \text{Pattern}$

Inputs: $a \quad b \quad d$
Conditional Rewrite Rules

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

- **Pattern** \xrightarrow{Assumptions} Pattern
- **Inputs**: a, b, d
- $\forall x \; \text{pred}(x) \Rightarrow (LHS(x) == RHS(x))$
Conditional Rewrite Rules

Pattern → Pattern

Inputs: a, b, d

∀x \text{pred}(x) \Rightarrow (LHS(x) == RHS(x))

Sketch Simplifier: apply in order at each node
Code for implementing Rewrite Rules

\[ \text{and}(\text{lt}(\text{plus}(a,e),x), \text{lt}(\text{plus}(e,b),x)) \]

\[ b < a \quad \rightarrow \quad \text{lt}(\text{plus}(a,e),x) \]
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;
            }
        }
    }
}

\[ \text{and}(\text{lt}(\text{plus}(a,e),x),
\text{lt}(\text{plus}(e,b),x)) \]
\[ \text{b}<\text{a} \quad \rightarrow \quad \text{lt}(\text{plus}(a,e),x) \]
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

Corpus of Benchmarks
SWAPPER framework

Corpus of Benchmarks → Pattern Finding → Patterns
SWAPPER framework

- Corpus of Benchmarks
- Pattern Finding
- Patterns
- Rule Generation (Synthesis)
- Rules
- Auto-tuning (Machine Learning)
- Simplifier
- Subset of Rules
- Simplifier Generation (Compilation)
Related Work

- Peephole optimizations
  - 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
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 $\Rightarrow$ More applicable rules
Pattern Finding

Corpus of Formulas (DAGs)

Patterns (Sub-formulas)

Commonly Occurring Patterns $\Rightarrow$ More applicable rules
Different from Motif Discovery
Representative Sampling

Formula Trees
Representative Sampling

Formula Trees

\[ \frac{1}{\text{nodes}} \]
Representative Sampling

Formula Trees
Representative Sampling

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

Formula Trees
Representative Sampling

\[ \frac{1}{|nodes|} \times \frac{1}{2} \times \frac{1}{3} \]

Formula Trees
Representative Sampling

Formula Trees

\[ \frac{1}{|\text{nodes}|} \times \frac{1}{2} \times \frac{1}{3} \times \frac{1}{4} \ldots \]
Representative Sampling

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)

```
5 + a
```
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)

```
5 + a
```

- 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
**Issues:**

- Dealing with missing edges (e.g. reaching top-most nodes or variable arity)
- For DAGs, Finding same pattern in multiple ways

**Tree Construction**

**BFS Ordering**
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

![Diagram showing tree construction and BFS ordering]

Use Rejection Sampling
Pattern Finding: Book-keeping

AND

OR

OR
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()

\[
\text{static}(b) = (-\infty, 0]
\]

\[
\text{static}(d) = (0, \infty)
\]
Contextual Information: \textit{static( )}

\[
\begin{align*}
\text{static}(b) &= (-\infty, 0] \\
\text{static}(d) &= (0, \infty)
\end{align*}
\]

Can infer strong assumptions like:
\[
\begin{align*}
&b < d \\
&b \neq d \\
&b \leq 0 \\
&0 < d
\end{align*}
\]
SWAPPER framework

Corpus of Benchmarks → Pattern Finding → Patterns → Rule Generation (Synthesis) → Rules → Simplifier Generation (Compilation) → Subset of Rules

Optimal Simplifier → Auto-tuning (Machine Learning) → Simplifier
Conditional Rewrite Rules

- **Inputs** \( (x) \):  \( a \),  \( b \),  \( d \)

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

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

| $x_i$ | $\neg x_j$ for boolean $x_i, x_j$ |

$| True$

$$x = (x_1, x_2, ..., x_n) \text{ and } 1 \leq i \neq j \leq n$$
Rule Generation: Grammars

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

\[
\text{pred}(x) \equiv x_i \ \text{binop} \ x_j \text{ for integer } x_i, x_j \\
\quad \mid x_i \mid \neg x_j \text{ for boolean } x_i, x_j \\
\quad \mid \text{True} \\
\text{x} = (x_1, x_2, ..., x_n) \text{ and } 1 \leq i \neq j \leq n \\
\text{binop} \equiv \langle | > | \leq | \geq | == | \neq
\]
Rule Generation: Grammars

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

  \[ \text{pred}(x) \equiv x_i \ \text{binop} \ x_j \ 	ext{for integer } x_i, x_j \]
  \[
  \mid x_i \mid \neg x_j \ 	ext{for boolean } x_i, x_j
  \]
  \[
  \mid \text{True}
  \]

  \[x = (x_1, x_2, ..., x_n) \ 	ext{and } 1 \leq i \neq j \leq n\]

  \[\text{binop} \equiv \langle |> | \leq | \geq | == | \neq\]

- **Grammar for \textit{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:

- $\forall x : static(x) \Rightarrow pred(x)$
- $\forall x : pred(x) \Rightarrow (LHS(x) == RHS(x))$
- $size(RHS) < size(LHS)$
Rule Generation: Hybrid approach

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

- \( \forall x: \text{static}(x) \Rightarrow \text{pred}(x) \)
- \( \forall x: \text{pred}(x) \Rightarrow (LHS(x) == RHS(x)) \)
- \( \text{size}(RHS) < \text{size}(LHS) \)

\[
\exists c_p c_r \forall x \left[ (\text{static}(x) \Rightarrow \text{pred}(x)) \wedge \text{pred}(x, c_p) \Rightarrow (LHS(x) = RHS(x, c_r)) \right]
\]

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

$$\exists c_p c_r \forall x \left[ (static(x) \Rightarrow pred(x)) \land pred(x, c_p) \Rightarrow (LHS(x) = RHS(x, c_r)) \right]$$

Classic Syntax-guided synthesis problem (Sketch)

+ Enumerative predicate refinement
Rule Generation: Example
Rule Generation: Example

\[
\begin{align*}
  a[j] &= y \\
  a[i + n] &= a[i] \\
  \text{if}(a[i] &= c) \\
  \text{then } x \\
  \text{else } a[i]
\end{align*}
\]
Rule Generation: Example

\[
\begin{align*}
  & a[j] = y \\
  & a[i + n] = a[i] \\
  & if(a[i] == c) then x else a[i]
\end{align*}
\]
Rule Generation: Example

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

\[ i == j \]
\[ \text{if} (y == c) \text{ then } x \text{ else } y \]

\[ \text{True} \]
\[ a[j] = y \]
\[ \text{if} (a[i] == c) \text{ then } x \text{ else } a[i] \]
SWAPPER framework

Corpus of Benchmarks → Pattern Finding → Patterns → Rule Generation (Synthesis) → Rules

Optimal Simplifier → Auto-tuning (Machine Learning) → Simplifier → Subset of Rules → Simplifier Generation (Compilation) →
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

- Matching and removing sub-patterns
Simplifier Generation: Rule Generalization

- Matching and removing sub-patterns

\[ a: \text{INT} \quad b: \text{INT} \quad c: \text{INT} \]

\[ \text{PLUS} \quad \text{MULT} \quad \text{DIV} \]

\[ c == 1 \]

\[ a: \text{INT} \quad b: \text{INT} \]

\[ \text{PLUS} \quad \text{DIV} \]
Simplifier Generation: Rule Generalization

- Matching and removing sub-patterns
Replace sub-patterns by inputs recursively
Replace sub-patterns by inputs recursively
Replace sub-patterns by inputs recursively
SWAPPER framework

Corpus of Benchmarks -> Pattern Finding -> Patterns -> Rule Generation (Synthesis) -> Rules

Optimal Simplifier

Auto-tuning (Machine Learning) -> Simplifier -> Subset of Rules -> Auto-tuning (Machine Learning)
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
<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark DAGs Used</th>
<th>Avg. Number of Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>45</td>
<td>23289</td>
</tr>
<tr>
<td>Sygus</td>
<td>22</td>
<td>68366</td>
</tr>
<tr>
<td>SAT Encodings</td>
<td>70</td>
<td>6504</td>
</tr>
</tbody>
</table>
## Domains & Benchmarks

<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark DAGs Used</th>
<th>Avg. Number of Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>45</td>
<td>23289</td>
</tr>
<tr>
<td>Sygus</td>
<td>22</td>
<td>68366</td>
</tr>
<tr>
<td>SAT Encodings</td>
<td>70</td>
<td>6504</td>
</tr>
</tbody>
</table>

- Full evaluation was done on **AutoGrader** and **Sygus**
## Domains & Benchmarks

<table>
<thead>
<tr>
<th>Domain</th>
<th>Benchmark DAGs Used</th>
<th>Avg. Number of Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>45</td>
<td>23289</td>
</tr>
<tr>
<td>Sygus</td>
<td>22</td>
<td>68366</td>
</tr>
<tr>
<td>SAT Encodings</td>
<td>70</td>
<td>6504</td>
</tr>
</tbody>
</table>

- Full evaluation was done on **AutoGrader** and **Sygus**
- Performed a validation case study on **SAT Encodings** benchmarks
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
Divided the corpus as \((\text{SEARCH,TRAIN,TEST})\)
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
Divided the corpus as \((\text{SEARCH,TRAIN,TEST})\)

- \text{SEARCH}: Pattern Finding
- \text{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

<table>
<thead>
<tr>
<th></th>
<th>AutoGrader</th>
<th>Sygus</th>
<th>SAT Encodings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtained</td>
<td>201</td>
<td>163</td>
<td>117</td>
</tr>
<tr>
<td>Optimal</td>
<td>135</td>
<td>65</td>
<td>68</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>Domain</th>
<th>Pattern Finding</th>
<th>Rule Generation</th>
<th>Auto-Tuning</th>
<th>Total Time (in hours)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>3 hours</td>
<td>1 hour $\times 5$</td>
<td>0.08 $\times 150$</td>
<td>20</td>
<td>$21.28$</td>
</tr>
<tr>
<td>Sygus</td>
<td>2 hours</td>
<td>1 hour $\times 5$</td>
<td>0.1 $\times 150$</td>
<td>22</td>
<td>$23.42$</td>
</tr>
</tbody>
</table>
Realistic Time and Costs

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

<table>
<thead>
<tr>
<th>Domain</th>
<th>Pattern Finding</th>
<th>Rule Generation</th>
<th>Auto-Tuning</th>
<th>Total Time (in hours)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>3 hours</td>
<td>1 hour × 5</td>
<td>0.08 × 150</td>
<td>20</td>
<td>$21.28</td>
</tr>
<tr>
<td>Sygus</td>
<td>2 hours</td>
<td>1 hour × 5</td>
<td>0.1 × 150</td>
<td>22</td>
<td>$23.42</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Domain</th>
<th>Pattern Finding</th>
<th>Rule Generation</th>
<th>Auto-Tuning</th>
<th>Total Time (in hours)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoGrader</td>
<td>3 hours</td>
<td>1 hour × 5</td>
<td>0.08 × 150</td>
<td>20</td>
<td>$21.28</td>
</tr>
<tr>
<td>Sygus</td>
<td>2 hours</td>
<td>1 hour × 5</td>
<td>0.1 × 150</td>
<td>22</td>
<td>$23.42</td>
</tr>
</tbody>
</table>

Costs less than an hour’s work of a good developer
Can reduce time by increasing parallelism or smarter evaluations with timeouts
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!