Program Synthesis

A Tutorial

Yingfei Xiong
Peking University
ISSTA Summer School 2019

Can grandmas program?

- The development of programming languages is to raise the level of abstraction

What is the next?

Haskell (1990), Prolog (1972)

Java

C

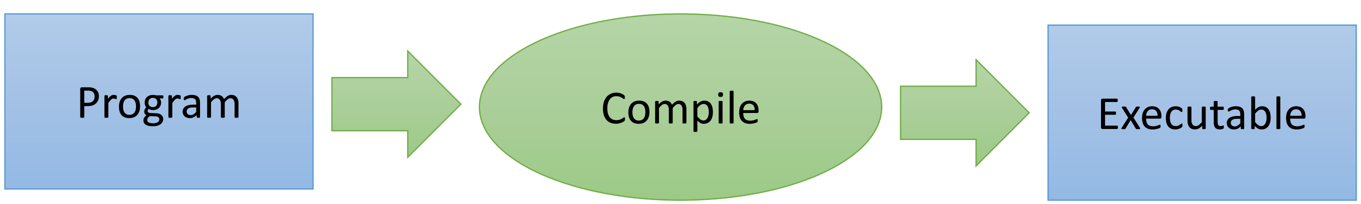
Assembly

Level of Abstraction

Why cannot?

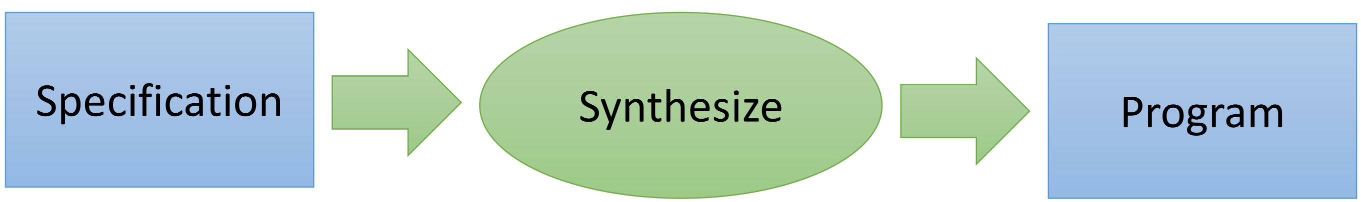
- Programming languages come with many guarantees
  - Well-typed programs are guaranteed to compile
  - Compiled programs have clear, well-defined semantics

- It is difficult to further raise the level of abstraction



Program Synthesis saves grandmas

- Generate a program from a specification
  - Specification can be fuzzy
  - Generation is not guaranteed



“**One of the most central problems in the theory of programming.**”
----Amir Pneuli
Turing Award Recipient

“**The fundamental way to improve software productivity**.”
----Jiafu Xu
Founder of Software Research in China

History of Program Synthesis

1957

- Start of program synthesis
- Circuit synthesis problem by Alonzo Church

Before 2000

- Deductive Synthesis

After 2000

- Inductive Synthesis

Application – Data Wrangling

	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

Application – Superoptimization

```
i=round(i);
```

→

```
a = 6755399441055744.0;
i=(i+a)-a;
```

Application – Reducing Duplicated Programming


```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
                         CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
                         battlecry=Battlecry(Destroy(),
                                   WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```

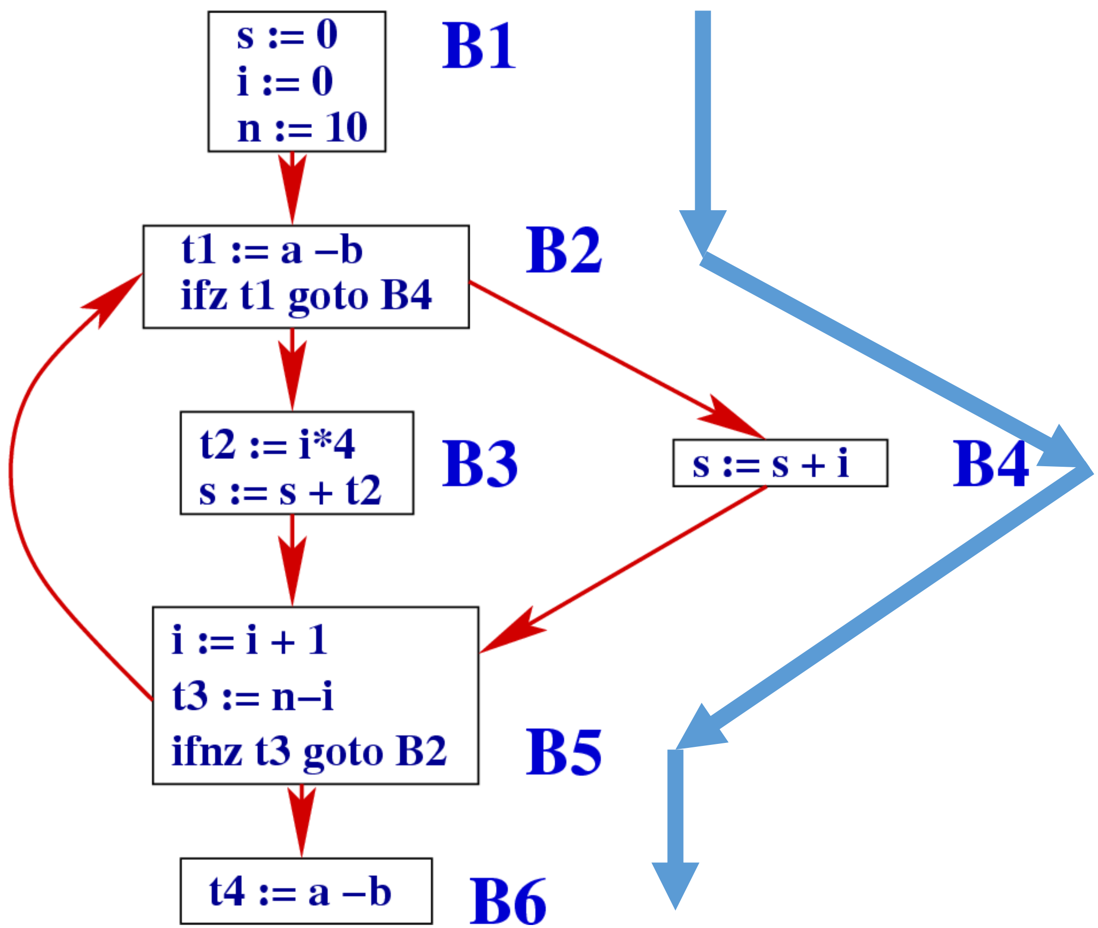
Application – Program Repair

```java
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```

Synthesize an expression to replace the buggy one

Application – Testing

Synthesize a unit test to cover a path



Application – Analysis

SMT Solver

Apply Tactic 1
If formula is long
  Apply Tactic 2
Else
  Apply Tactic 3

Strategies

Synthesize a strategy for a class of problems

Defining Program Synthesis

Classic Synthesis	Program Optimization	Program Estimation

• Input:
  • A specification
• Output: A program that
  • meets the specification
	• Input:
	  • A specification
	  • A cost function
	• Output: A program that
	  • meets the specification, and
	  • maximizes the cost function
		• Input:
		  • A specification
		  • A dataset for target distribution
		• Output: A program that
		  • meets the specification and
		  • maximizes the probability represented by the dataset

Test Generation	Superoptimization	Program Repair

This Lecture

Classic Synthesis

- Problem Definition
- Enumerative
- Presentation-based
- Constraint-based

Program Estimation

- Problem Definition
- Estimating Probabilities
- Locating the most-likely one

SyGuS: Syntax-Guided Synthesis

- A standardization of classic program synthesis problem.
- Input:
  - grammar G
  - specification S
- Output:
  - program P
  - such that $P \in G \land P \mapsto S$

Example: max

- Grammar:

$$
\begin{array}{rcl}
\texttt{Expr} & ::= & x \quad | \quad y \\
& | & \texttt{Expr} + \texttt{Expr} \\
& | & (\texttt{ite BoolExpr Expr Expr}) \\
\texttt{BoolExpr} & ::= & \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\
& | & \neg\texttt{BoolExpr} \\
& | & \texttt{Expr} \leq \texttt{Expr}
\end{array}
$$

- Specification:

$$
\begin{array}{ll}
\forall x, y : \mathbb{Z}, & max_2(x, y) \geq x \wedge max_2(x, y) \geq y \\
& \wedge (max_2(x, y) = x \vee max_2(x, y) = y)
\end{array}
$$

- Expected answer： ite (x <= y) y x

SyGuS format: Synth-Lib

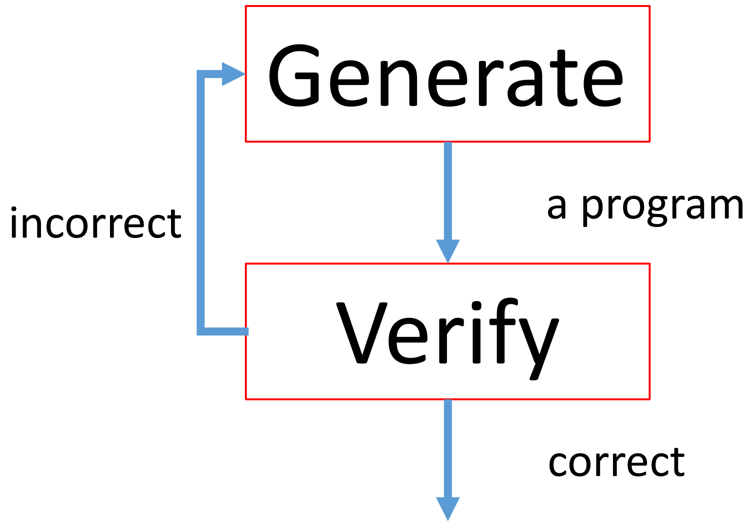
- Synth-Lib uses a format similar to SMT-Lib
  - http://sygus.seas.upenn.edu/files/SyGuS-IF.pdf

```
(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int
    ((Start Int (x y
                 (+ Start Start)
                 (ite StartBool Start Start)))……))

(declare-var x Int)
(declare-var y Int)
(constraint (>= (max2 x y) x))
……

(check-synth)
```

Program Synthesis as a Search Problem



Q1: How to generate the next program to be verified?

Q2: How to verify the correctness?

Q1: How to verify correctness?

- If the specification includes only tests,
  - test the program.

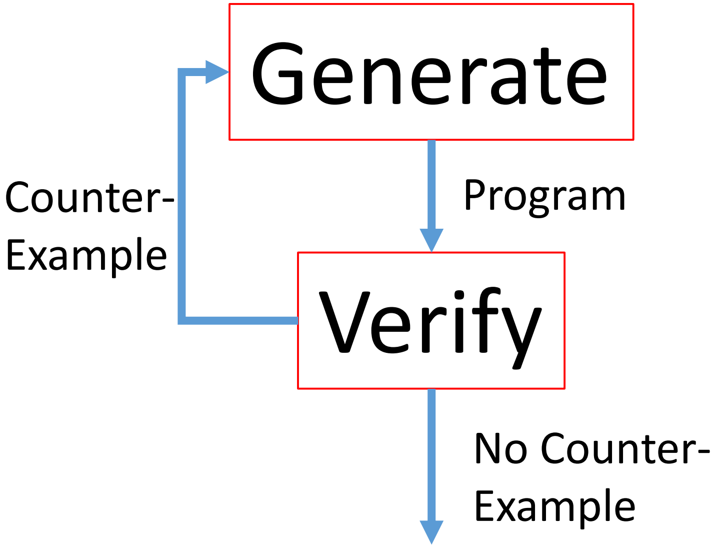
Fast

- If the specification is a logic constraint S,
  - verify $Program \rightarrow S$ by an SMT solver.
  - Synth-lib directly supports this

Slow

Can we combine the two?

CEGIS: Counter-Example Guided Inductive Synthesis



- Constraint solvers give counter-examples
- Save counter-examples as tests
- First use tests to validate programs

Q2: How to generate the next program to be verified?

- Enumerative – exhaustive search
- Representation-based – manipulate sets of programs instead of single programs
- Constraint-based – convert to an SMT problem

Top-Down Enumeration

- Expand according to the grammar
  - Expr
  - x, y, Expr+Expr, if(BoolExpr, Expr, Expr)
  - y, Expr+Expr, if(BoolExpr, Expr, Expr)
  - Expr+Expr, if(BoolExpr, Expr, Expr)
  - x+Expr, y+Expr, Expr+Expr+Expr, if(BoolExpr, Expr, Expr)+Expr, if(BoolExpr, Expr, Expr)
  - …

Bottom-Up Enumeration

- Combine expressions from small to big
  - size=1
    - x, y
  - size=2
  - size=3
    - x+y
  - size=4
  - size=5
    - x+(x+y), (x+y)+y
  - size=6
    - if(x<=y, x, y), …

Optimization

- Discard a partial program early
- Pruning
  - None of the expansions could satisfy the specification
  - ~~Ite BoolExpr x x~~
- Equivalence reduction
  - Equivalent to a previous program
  - Expr+x, ~~x+Expr~~

Pruning

- Generate constraints from the partial program

Ite BoolExpr x x →

```
(declare-fun boolExpr () Int)
(declare-fun max2 ((x Int) (y Int)) Int
   (ite boolExpr x x))
```

- Generate constraints from each test

max2(1,2)=2 →

```
(assert (= (max2 1 2) 2))

(check-sat)
```

Needs to balance between the benefit and the cost.

Equivalence reduction: How to determine equivalence?

- With an SMT solver
  - Check satisfiability of $f(x,y) \neq f'(x,y)$
  - The cost may not pay off

- With tests
  - Check if $f = f'$ on all tests
  - Not safe for logic specifications
  - Does not work on partial programs

- With predefined-rules
  - e.g Expr+x and x+Expr
  - Needs customization for each domain

How to generate the next program to be verified?

- Enumerative – exhaustive search
- Representation-based – manipulate sets of programs instead of single programs
- Constraint-based – convert to an SMT problem

Representation-based

- Enumerative approaches manipulates single programs
  - Inefficient: too many in number
- Can we manipulate sets of programs? e.g.
  - Find a set that satisfies a specification
  - Intersects sets for a conjunction of specifications
  - Combine sets with program constructs to satisfy more complex specifications
- Representation-based
  - Use data structures to represent such a set
  - E.g. Grammars, Automata, Logic Formulas

FlashMeta: Basic Idea

- Grammar is a representation of sets
  - Size of a grammar = O(log(#Represented Program))
- The original grammar is too coarse-grained

- Idea: Annotate a non-terminal with a synthesis goal
  - [2]Expr – expressions that evaluates to 2

FlashMeta: Single Test

- Pick a test
  - $\max2(1,2) = 2$
- Refine the grammar
  - $[2]\text{Expr} \rightarrow y \mid [1]\text{Expr} + [1]\text{Expr}$
    $\mid \text{ite } [\text{true}]\text{BoolExpr } [2]\text{Expr } [*]\text{Expr}$
    $\mid \text{ite } [\text{false}]\text{BoolExpr } [*]\text{Expr } [2]\text{Expr}$
  - $[1]\text{Expr} \rightarrow x \mid \cdots$
  - $[\text{true}]\text{BoolExpr} \rightarrow \neg[\text{false}]\text{BoolExpr}$
    $\mid [\text{true}]\text{BoolExpr} \land [\text{true}]\text{BoolExpr}$
    $\mid [2]\text{Expr} \leq [2]\text{Expr} \mid \cdots$
  - ...
  - Assume a user-provided operation to perform the refinement
- Any program represented by the grammar passes the test

Intersection of grammars

- Suppose
  - $N \to P_1 \mid \cdots \mid P_k$
  - $N' \to P'_1 \mid \cdots \mid P'_{k'}$

- $N \cap N' = P_1 \cap P'_1 \mid P_1 \cap P'_2 \mid \cdots \mid P_1 \cap P'_{k'}$
  $\mid P_2 \cap P'_1 \mid P_2 \cap P'_2 \mid \cdots \mid P_2 \cap P'_{k'}$
  $\mid \cdots$
  $\mid P_k \cap P'_1 \mid P_k \cap P'_2 \mid \cdots \mid P_k \cap P'_{k'}$
- $P_1 \cap P_2 = \emptyset$ if P_1 and P_2 are of different types
- $f(N_1, \ldots, N_k) \cap f(N'_1, \ldots, N'_k) = f(N_1 \cap N'_1, \ldots, N_k \cap N'_k)$

FlashMeta: Multiple Tests

- Produce a grammar for each test
- Intersects the grammars

FlashMeta: Discussion

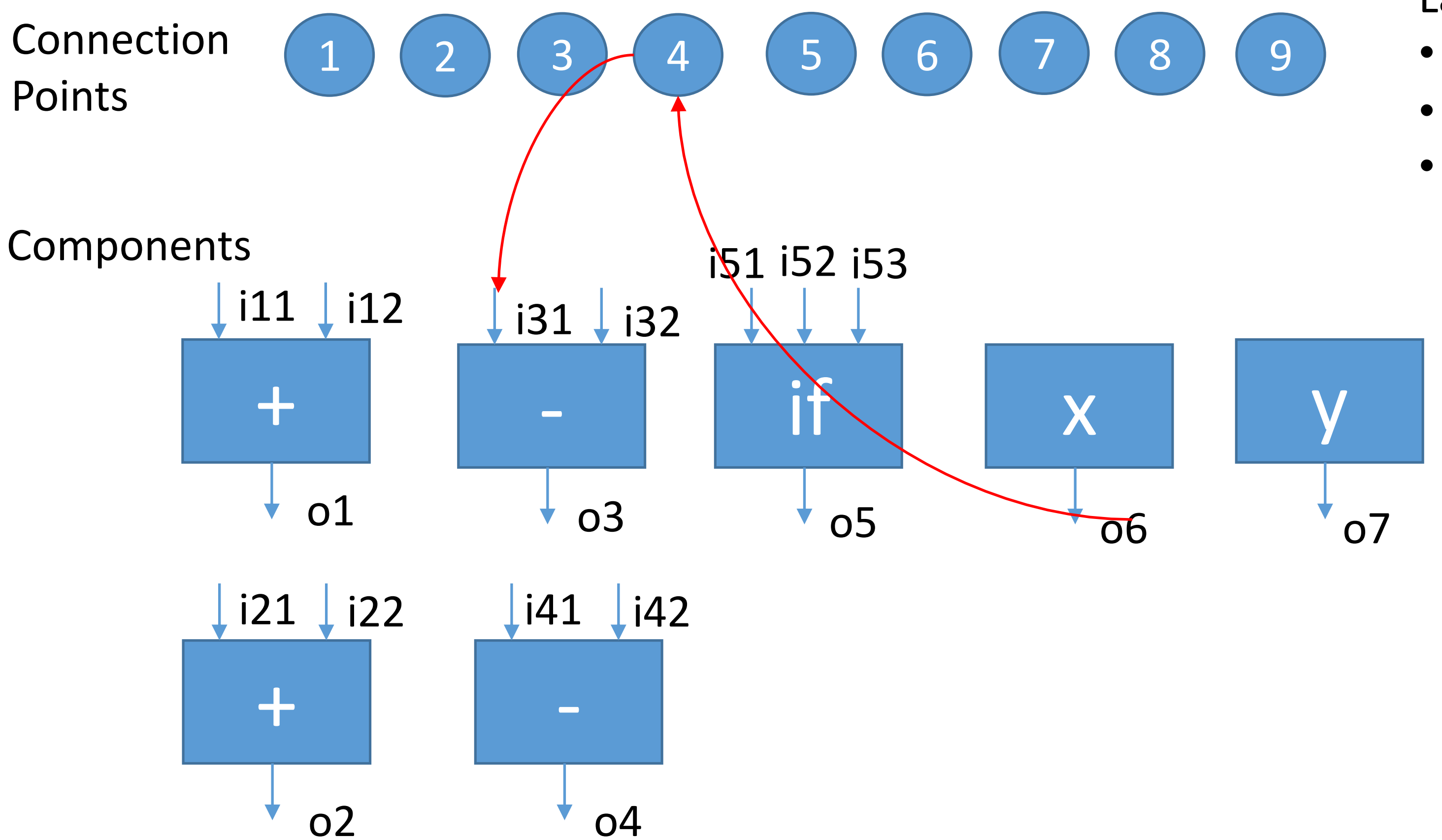
- Avoids duplicated computation
  - $[1]\mathrm{Expr} + [1]\mathrm{Expr}$
  - $[1]\mathrm{Expr}$ is explored only once in FlashMeta
- Pruning is naturally included
  - $[1]\mathrm{Expr} \rightarrow \cancel{\mathrm{Expr} + \mathrm{Expr}}$
- Needs user-provided operation for refinement
  - [65536]Expr

- Trivia: original paper uses version space algebra, which is essentially grammar

How to generate the next program to be verified?

- Enumerative – exhaustive search
- Representation-based – manipulate sets of programs instead of single programs
- Constraint-based – convert to an SMT problem

Component-Based Program Synthesis



Label variables：

- $l_{i11}, l_{i22}, \ldots$
- $l_{o1}, l_{o2}, \ldots$
- l_o: program output

$l_{o6} = l_{i31} = 4$

Generate constraints

- Test
  - $o6 = 1 \wedge o7 = 2$
  - $o \geq 1 \wedge o \geq 2 \wedge (o = 1 \vee o = 2)$
- Component Semantics
  - $o1 = i11 + i12$
- Label Semantics
  - $l_{o1} = l_{i11} \rightarrow o_1 = i_{11}$
- Label Range
  - $l_{o1} \geq 1 \wedge l_{o1} \leq 9$
- Uniqueness of Output
  - $l_{o1} \neq l_{o2}$
- No Cycle
  - $l_{i11} < l_{o1}$

Why use connection points? What if we remove connection points and output label $l_{o\mathrm{x}}$, and use l_{ixx} to represent the index of the output?

This Lecture

Classic Synthesis

- Problem Definition
- Enumerative
- Presentation-based
- Constraint-based

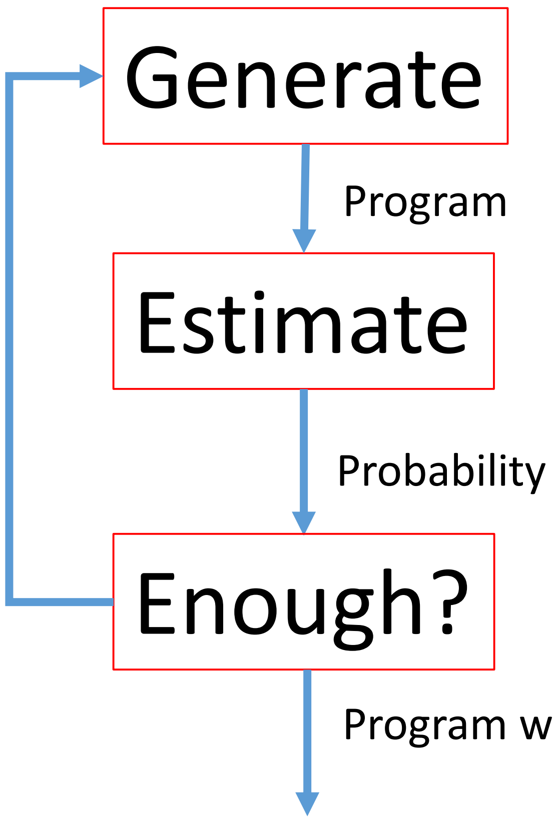
Program Estimation

- Problem Definition
- Estimating Probabilities
- Locating the most-likely one

Program Estimation

- Input:
  - program space G
  - specification S
  - context C
  - a training set T of context-program pairs
- Output:
  - program P
  - such that $P \in G \wedge P \mapsto S \wedge \Pr(P \mid C)$
  - where $\Pr$ represents the probability learned from T

Program Estimation as an Search Problem



Q3: How to estimate the probability $\Pr(P \mid C)$?

Q4: How to find program P such that $\Pr(\,prog \mid context\,)$ is the largest?

Learning to synthesis (L2S)

- A general framework to address program estimation
- Combining four tools
  - **Rewriting rules**: defining a search problem
  - **Constraint solving**: pruning off invalid choices in each step
  - **Machine-learned models**: estimating the probabilities of choices in each step
  - **Search algorithms**: solving the search problem

Example: Condition Completion

- Given a program without a conditional expression, completing the condition

```
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

E → E ">12"
  | E ">0"
  | E "+" E
  | *"hours"*
  | *"value"*
  | …

Space of Conditions

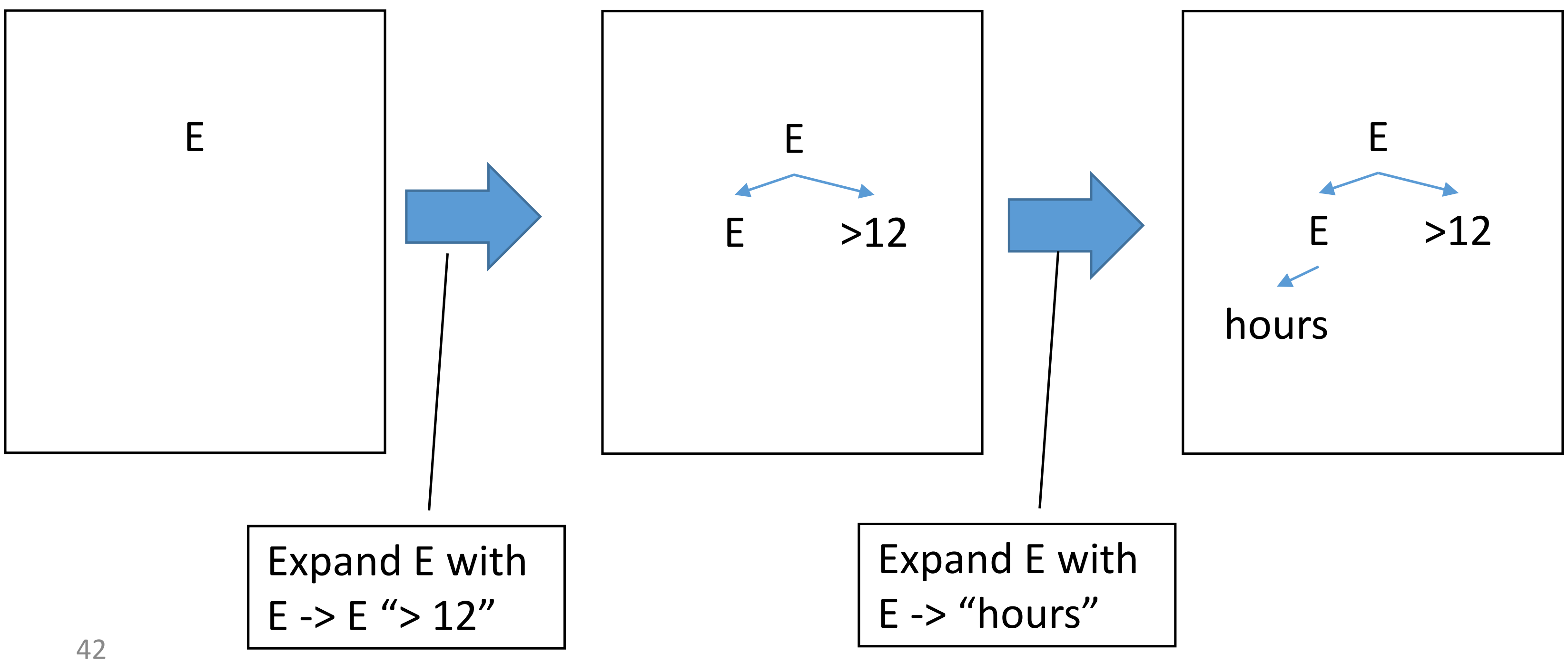
- Useful in program repair
  - Many bugs are caused by incorrect conditions
  - Existing work could localize the faulty condition
  - Can we generate a correct condition to replace the incorrect one?

Q3: Estimating the Probability

- Idea: Using machine learning
  - To train over a set of programs and their contexts
- Problem: machine learning usually works for classification problems
  - where the number of classes are usually small
- Idea: turn the generation problem into a set of classification problem along the grammar

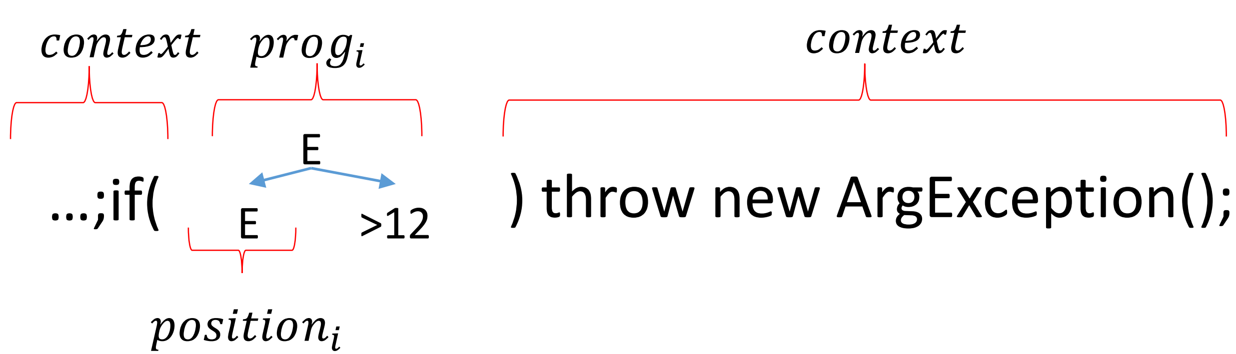
Decomposing Generation

- In each step, we estimate the probabilities of the rules to expand the left-most non-terminal
  - A classification problem



Probability of the program

- $P(prog \mid context) = \prod_i P(\,rule_i \mid context, prog_i, position_i\,)$
    - $context$: The context of the program
    - $prog_i$: The AST generated at the ith step
    - $position_i$: The non-terminal to be expanded at the ith step
    - rule: the chosen rule at the ith step
    - $prog$: the complete program

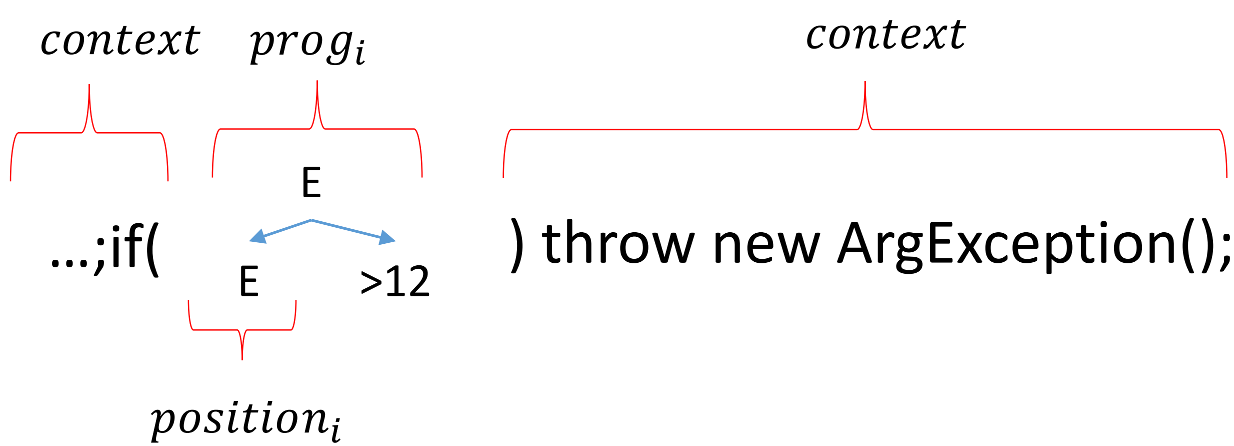


Training models

- Train a model for each non-terminal
  - to classify rules expanding this non-terminal
- Training set preparation
  - The original training set:
    - A set of programs
    - Their contexts
  - Decomposing the training set:
    - Parse the programs
    - Extract the rules chosen for each non-terminal

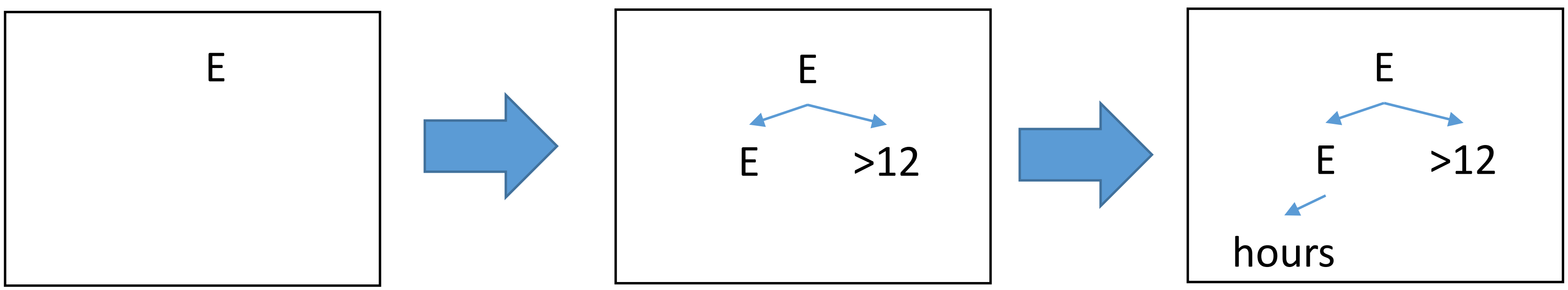
Feature Engineering

- Extract features from
  - $context$: The context
  - $prog_i$: The generated partial AST
  - $position_i$: The position of the node to be expanded

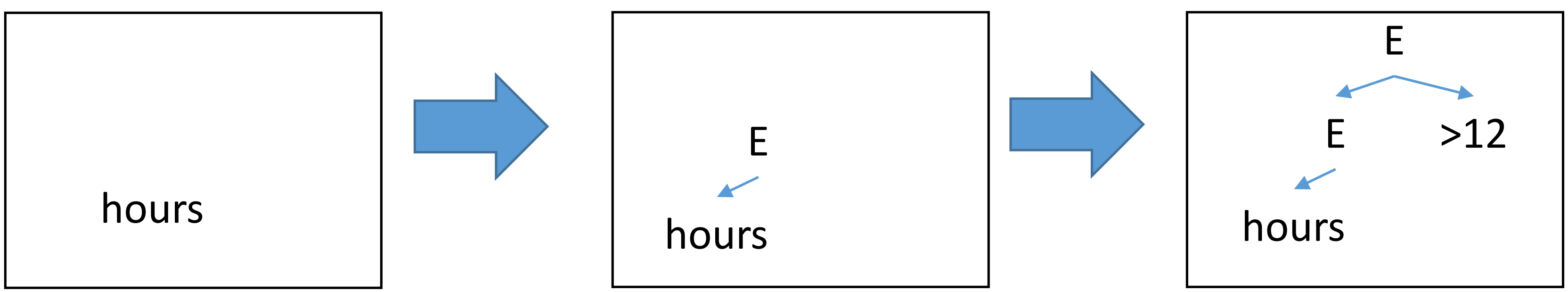


Can we use a different expansion order?

- Top-down



- Bottom-up



The order may greatly affect the performance of L2S.

Annotations

- Introduce annotations to symbols
  - E^D indicates E can be expanded downward
  - E^U indicates E can be expanded upward
  - E^{UD} indicates E can be expanded in both directions

From Grammar to Rewriting Rules

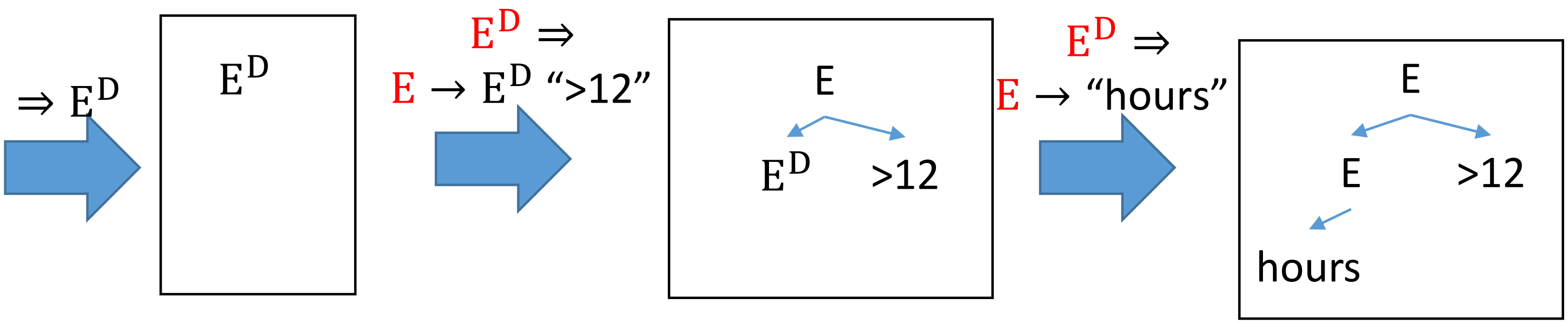
Grammar	Top-down Rules	Bottom-up Rules
$E \rightarrow E$ "+" E	$E^D \Rightarrow E \rightarrow E^D$ "+" E^D	$E^U \Rightarrow E^U \rightarrow E$ "+" E^D $E^U \Rightarrow E^U \rightarrow E^D$ "+" E
$E \rightarrow E$ ">12"	$E^D \Rightarrow E \rightarrow E^D$ ">12"	$E^U \Rightarrow E^U \rightarrow E$ ">12"
$E \rightarrow$ "hours"	$E^D \Rightarrow E \rightarrow$ "hours"	"hours"$^U \Rightarrow E^U \rightarrow$ "hours"

Creation Rules	
$\Rightarrow E^D$	// starting from the root
$\Rightarrow E^{DU}$	// starting from a middle node
$\Rightarrow$ "hours"^{U}	// starting from a leaf

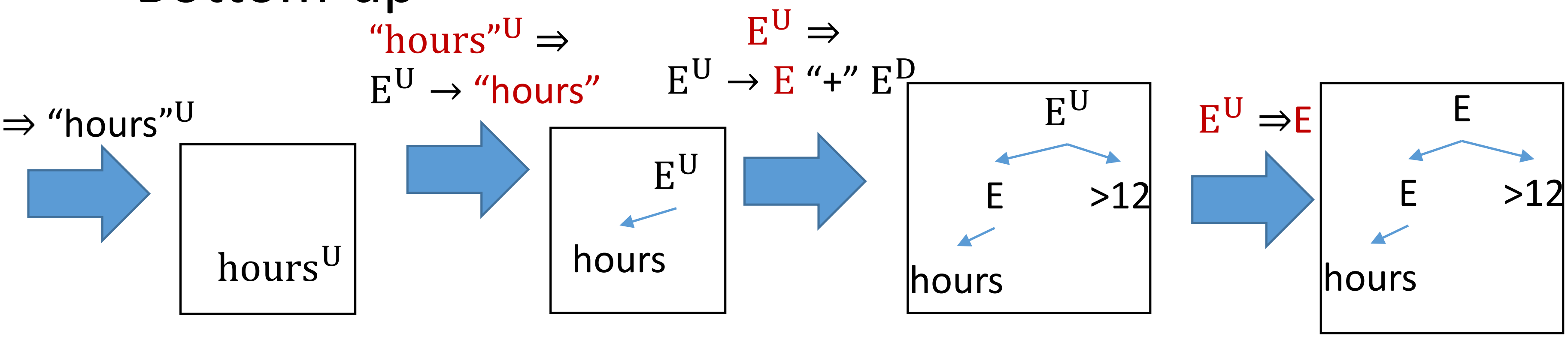
Ending Rule	$E^U \Rightarrow E$

Example

- Top-down



- Bottom-up



Unambiguity

- A set of rewriting rules are unambiguous if
  - there is at most one unique set of rule applications to construct any program.

- When the rule set is unambiguous, we still have
  - $P(prog \mid context) = \prod_i P(\, rule_i \mid context, prog_i, position_i \,)$

Q4: How to find the most probable program?

- Local Optimal ≠ Global Optimal

E_0

E → E " > 12"	0.3
E → E " > 0"	0.6

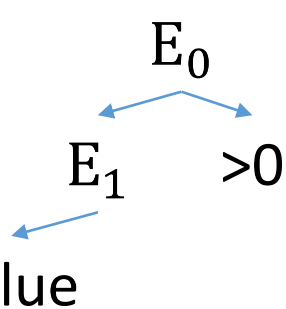
E_1

E → "hours"	0.1
E → "value"	0.2
E → E " + " E	0.05

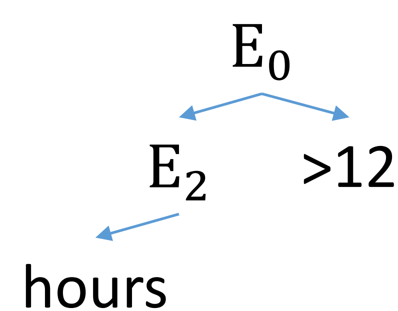
E_2

E → "hours"	0.8
E → "value"	0.1
E → E " + " E	0.05

0.6 * 0.2
= 0.12



0.3 * 0.8
= 0.24



Use Metaheuristic Search

- Beam Search:
  - Keep n most probable partial programs
  - Expand the programs to get new programs
- Genetic Search:
  - Keep n most probably complete programs
  - Mutate the programs to get new programs

Applications

- Application 1:
  - Repairing Conditional Expressions
- Application 2:
  - Generating Code from Natural Language Expression

Repairing Conditional Expressions

- Condition bugs are common

```
  lcm = Math.abs(a+b);
+ if (lcm == Integer.MIN_Value)
+   throw new ArithmeticException();
```

Missing boundary checks

```
-  if (hours >= 24)
+ if (hours > 24)
     withinOneDay=true;
```

Conditions too weak or too strong

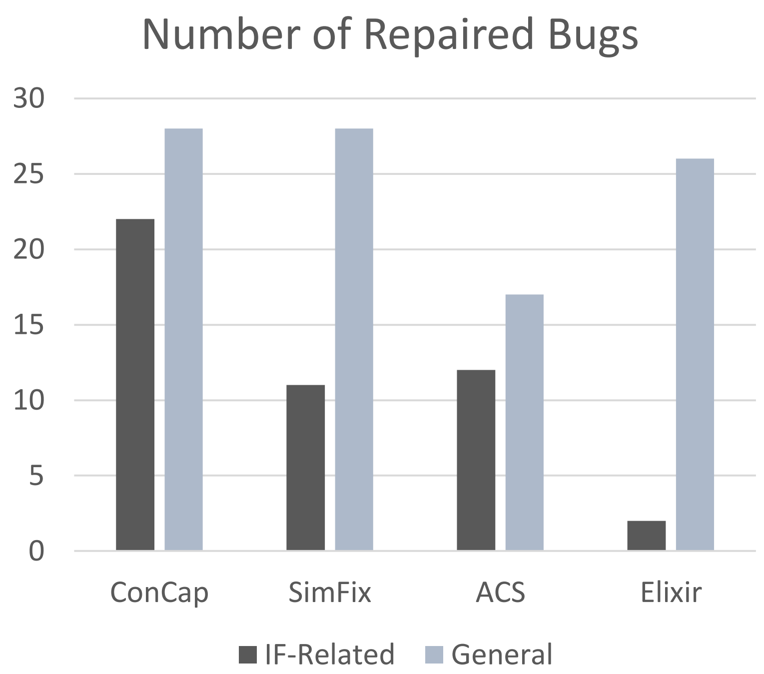
- Steps:
  1. Localize a buggy if condition with SBFL and predicate switching
  2. Synthesize an if condition to replace the buggy one
  3. Validate the new program with tests

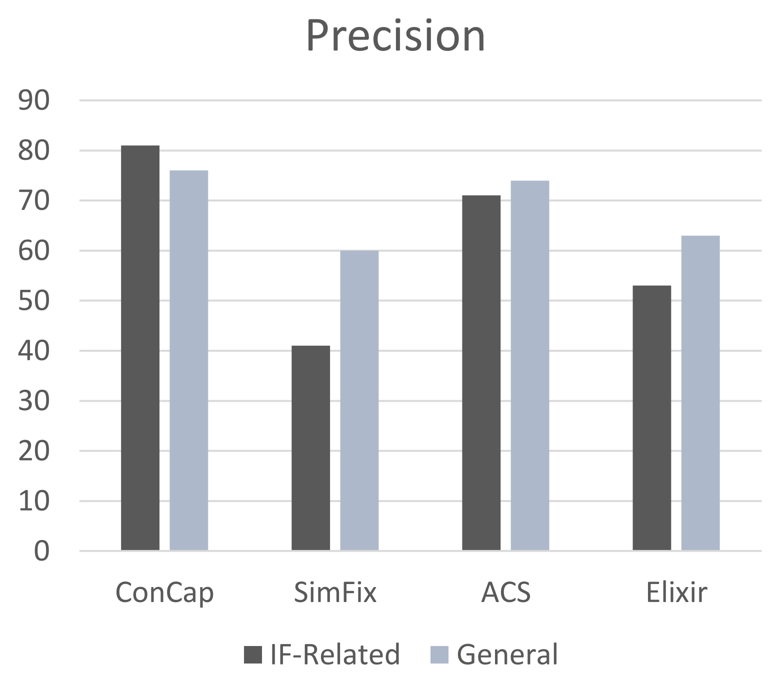
L2S Configuration

- Rewriting rules
  - Bottom-up
  - Estimate the leftmost variable first
- Machine learning
  - Xgboost
  - Manually designed features
- Constraints
  - Type constraints & size constraints
- Search algorithm
  - Beam search

Results

Benchmark: Defects4J





Also repaired 8 unique bugs that have never been repaired by any approach.

Generating Code from Natural Language Expression

- Can we generate code automatically to avoid repetitive coding?
- Existing approaches use RNN to translate natural language descriptions to programs
  - **Long dependency problem**: work poorly on long programs


```
[NAME]
Acidic Swamp Ooze
[ATK] 3
[DEF] 2
[COST]  2
[DUR]  -1
[TYPE]  Minion
[CLASS] Neutral
[RACE] NIL
[RARITY] Common
[DESCRIPTION]
"Battlecry: Destroy Your Opponent's Weapon"
```

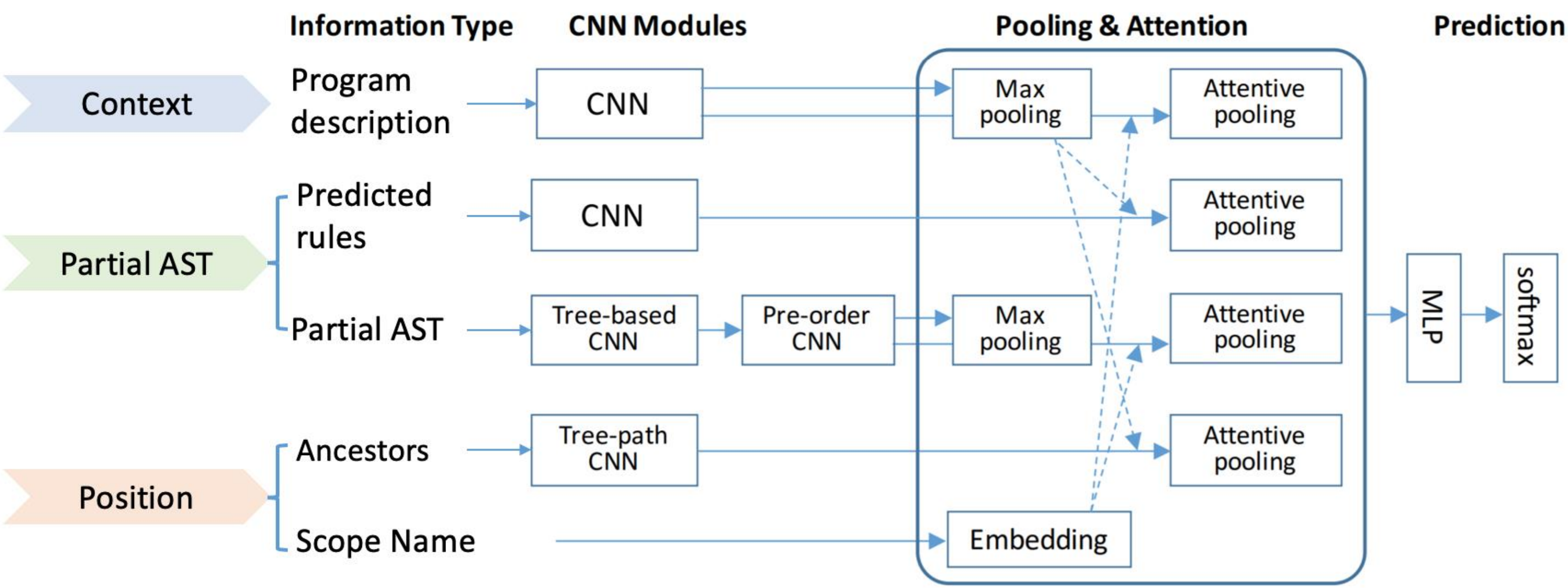
```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry=Battlecry(Destroy(), WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```

L2S Configuration

- Rewriting rules
  - Top-down
- Machine learning
  - A CNN-based network
- Constraints
  - Size constraints
- Search algorithm
  - Beam search

A CNN-based Network Architecture



Results

Benchmark: HearthStone

Model	**StrAcc**	**Acc+**	**BLEU**
LPN (Ling et al. 2016)	6.1	–	67.1
SEQ2TREE (Dong and Lapata 2016)	1.5	–	53.4
SNM (Yin and Neubig 2017)	16.2	∼18.2	75.8
ASN (Rabinovich, Stern, and Klein 2017)	18.2	–	77.6
ASN+SUPATT (Rabinovich, Stern, and Klein 2017)	22.7	–	79.2
Our system	**27.3**	**30.3**	**79.6**

Newest Results

- Replacing CNN with Transformer
  - Transformer: a new neural architecture at 2017
  - The flexibility of L2S allows to easily utilize new models

	Model	StrAcc	Acc+	BLEU
Plain	LPN (Ling et al., 2016)	6.1	–	67.1
	SEQ2TREE (Dong and Lapata, 2016)	1.5	–	53.4
	YN17 (Yin and Neubig, 2017)	16.2	∼18.2	75.8
	ASN (Rabinovich et al., 2017)	18.2	–	77.6
	ReCode (Hayati et al., 2018)	19.6	–	78.4
	CodeTrans-A	**25.8**	**25.8**	**79.3**
Structured	ASN+SUPATT (Rabinovich et al., 2017)	22.7	–	79.2
	SZM19 (Sun et al., 2019)	27.3	30.3	79.6
	CodeTrans-B	**31.8**	**33.3**	80.8

Future Learning

- Surveys:
  - Sumit Gulwani, Oleksandr Polozov, Rishabh Singh: Program Synthesis. Foundations and Trends in Programming Languages 4(1-2): 1-119 (2017)
  - Rajeev Alur, Rastislav Bodík, et al.: Syntax-guided synthesis. FMCAD 2013: 1-8
- Tools:
  - sygus.org – the SyGuS competition, a good place to look at
  - Some tools we recently used
    - EUSolver
    - CVC4
    - Second-Order Solver
- Course:
  - Program Synthesis by Nadia Polikarpova@UCSD
  - https://github.com/nadia-polikarpova/cse291-program-synthesis/

Reference

- Enumerative
  - Sumit Gulwani, Oleksandr Polozov, Rishabh Singh: Program Synthesis. Foundations and Trends in Programming Languages 4(1-2): 1-119 (2017)
  - Rajeev Alur, Rastislav Bodík, et al.: Syntax-guided synthesis. FMCAD 2013: 1-8
- FlashMeta
  - Oleksandr Polozov, Sumit Gulwani: FlashMeta: a framework for inductive program synthesis. OOPSLA 2015: 107-126
- Componen-Based Program Synthesis
  - Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, Ashish Tiwari: Oracle-guided component-based program synthesis. ICSE (1) 2010: 215-224
- L2S
  - Yingfei Xiong, Bo Wang, et al.: Learning to Synthesize. GI'18: Genetic Improvment Workshop, May 2018