Send paper preference via email by **midnight Sept 4**

First paper presentation

First review

Schedule will be updated with speakers
  - There may be new papers
  - Paper orders may change
  - Check after Sept 5!
Three pillars: intention, invention, adaptation

This course: different techniques for different specifications in different applications
Today’s lecture

- Syntax-guided synthesis (SyGuS): a framework to study program synthesis
- Programming-by-example (PBE): an instance of SyGuS
- Two PBE techniques: search-based and representation-based
Syntax-guided synthesis
Syntax-guided synthesis (SyGuS)

• SyGuS is a general formulation of program synthesis problems
  • Not a program synthesis technique

• Idea: search over space of programs

• Specification → Search
  Program written according to syntax (context-free grammar)

  semantic constraint
  (what should this program do)

  syntactic constraint
  (what should this program look like)

• Advantages: synthesis becomes more tractable
An example SyGuS problem

- Find a program $f(x)$ in the following grammar

$$e ::= x \mid 1 \mid e + e$$

such that $f(1) = 2$

- A solution: $f(x) = x + 1$
- Another solution: $f(x) = x + x$
Formal definition of SyGuS

- Given a first-order formula $\phi$ in a background theory $T$ and a context-free grammar $L$, the syntax-guided synthesis problem is to find an expression $e \in L$ such that formula $\phi[f/e]$ is valid in theory $T$.

- In previous example:
  - Find a program in the following grammar, such that $f(1) = 2$.
    
    $e ::= x \mid 1 \mid e + e$

    context-free grammar $L$

- $x + 1$ is a solution, because $1 + 1 = 2$ is valid in theory of Linear Integer Arithmetic, intuitively, this means $1 + 1 = 2$ is correct.
**Context-free grammar (CFG)**

- CFG defines **syntax** of a programming language
  - A set of programs

- More formally: $(T, N, P, S)$
  - $T$: set of terminal symbols
  - $N$: set of non-terminal symbols
  - $P$: set of productions of the form $s \rightarrow f(s_1, \cdots, s_n)$
  - $S \in N$: start symbol
An example CFG

- CFG is defined as $(T, N, P, S)$
  
  $e ::= x \mid 1 \mid e + e$

- $T$: set of terminal symbols
  
  $\{x, 1\}$

- $N$: set of non-terminal symbols
  
  $\{e\}$

- $P$: set of productions of the form $s \rightarrow f(s_1, \ldots, s_n)$
  
  $\{e \rightarrow x, e \rightarrow 1, e \rightarrow e + e\}$

- $S \in N$: start symbol
  
  $e$
Write programs according to CFG

- Step 1: begin with the start symbol
- Step 2: pick a non-terminal in current result and replace it with one of its productions
- Step 3: continue step 2 until no more non-terminal remains (i.e., only terminals)
An example

• CFG:  \[ e ::= x \mid 1 \mid e + e \]

• Step 1: begin with the start symbol
• Step 2: pick a non-terminal in current result and replace it with one of its productions
• Step 3: continue step 2 until no more non-terminal remains (i.e., only terminals)
SyGuS recap

- Given a first-order formula $\phi$ in a background theory $T$ and a context-free grammar $L$, the syntax-guided synthesis problem is to find an expression $e \in L$ such that formula $\phi[f/e]$ is valid in theory $T$.

SyGuS: $< T, \phi, L >$

- Theory: What does a program mean
- Specification: What program we want
- Grammar: What programs we consider
Programming-by-Example
Programming-by-example (PBE)

- PBE is a specific kind of SyGuS
  - Specification $\phi$ encodes a set of input-output examples
  - Goal of PBE: find a program in CFG that satisfies a given set of I/O examples
  - A more ambitious goal: not only satisfy examples, but actual intent (mind reading!)
- E.g., FlashFill
Why is PBE important

• Simplest kind of specification (arguably)

• Useful in practice
  • Even non-programmers can provide examples

• Technically fundamental
  • Underly many program synthesis techniques using other specs
  • Will cover these techniques in Module 2 (and Module 3)
Challenges of PBE

• Scalability
  • Search space defined by syntax is huge (although examples are simple)
  • How to find a program w/o waiting too long?

• Ambiguity
  • Examples are ambiguous
  • How to guess the right program w/o one telling you everything about it?

• Usability — how to make PBE systems useful and usable in practice?

• Applicability — how to create PBE systems widely applicable to many domains?
  • ...
An example PBE problem

• Syntax

\[
\begin{align*}
\text{e} & : = \text{f} \mid \text{concat}(\ \text{f}, \ \text{e}) \\
\text{f} & : = \text{s} \mid \text{substr}(\ \text{x}, \ \text{p}, \ \text{p}) \\
\text{p} & : = \text{k} \mid \text{position}(\ \text{x}, \ \text{r}, \ \text{k}) \\
\text{r} & : = \text{t} \mid \text{seq}(\ \text{t}, \ \ldots, \ \text{t}) \\
\text{t} & : = \text{<num>} \mid \text{<let>} \mid \text{<ws>} \mid \text{<any>} \mid \ldots
\end{align*}
\]

s is string constant, k is int constant, 
x is input variable

• Semantics

• Specification

“Bill Gates” \rightarrow “BG”

Some sample programs in this language:

\[
\begin{align*}
\text{concat}(\ “a”, \ “b” ) \\
“12ab” \rightarrow ??? \\
\text{concat}(\ “a”, \ \text{substr}(\ \text{x}, \ 0, \ 1)) \\
“12ab” \rightarrow ??? \\
\text{concat}(\ “a”, \ \text{substr}(\ \text{x}, \ 0, \ \text{position}(\ \text{x}, \ <\text{num}>, \ 1))) \\
“12ab” \rightarrow ??? \\
\text{concat}(\ \text{substr}(\ \text{x}, \ 0, \ 1), \\
\quad \text{substr}(\ \text{x}, \ \text{position}(\ \text{x}, \ <\text{ws}>, \ 1), \ \text{position}(\ \text{x}, \ <\text{cap}>, \ 2)))
\end{align*}
\]

What does this program do?
Solve PBE problems

• “Bill Gates” → “BG”

    “B”

    concat substr(x, 0, 1),

    substr(x, position(x, <ws>, 1), position(x, <cap>, 2))

• Solution:

    “Bill Gates”

    e := f | concat(f, e)

    f := s | substr(x, p, p)

    p := k | position(x, r, k)

    r := t | seq(t, ..., t)

    t := <num> | <let> | <ws> | <any> | ...
Solve PBE problems (cont’d)

• “Bill Gates” —> “BG”

• Given solution, simple to check correctness

• ... but we do not have solution a priori (only spec!)

• How to find the solution?

e := f | \texttt{concat}( f, e )
f := s | \texttt{substr}( x, p, p )
p := k | \texttt{position}( x, r, k )
r := t | \texttt{seq}( t, ..., t )
t := <\texttt{num}> | <\texttt{let}> | <\texttt{ws}> | <\texttt{any}> | ...
PBE challenges

• Huge search space (easily $>10^{20}$ in simplified FlashFill language!) — how to scale?

  \[
e := f \mid \text{concat}( f, e )
  
  f := s \mid \text{substr}( x, p, p )
  
  p := k \mid \text{position}( x, r, k )
  
  r := t \mid \text{seq}( t, \ldots, t )
  
  t := <\text{num}> \mid <\text{let}> \mid <\text{ws}> \mid <\text{any}> \mid \ldots
  \]

• Ambiguity — how to find the desired program w/o too many examples?

  \[
  \text{concat}( \text{substr}( x, 0, 1 ), \text{substr}( x, \text{position}( x, <\text{ws}>, 1 ), \text{position}( x, <\text{cap}>, 2 )))
  \]

  \[
  \text{concat}( \text{substr}( x, 0, 1 ), \text{substr}( x, 5, 6 )))
  \]

  \[
  \text{concat}( "B", "G" )
  \]

  \[
  \text{concat}( "B", \text{substr}( x, \text{position}( x, <\text{ws}>, 1 ), \text{position}( x, <\text{cap}>, 2 )))
  \]

  \[
  \ldots
  \]
PBE techniques
Many PBE techniques

- Search-based
- Representation-based

- Using constraint solving
- Stochastic
- Neural approaches
- ...

23
PBE technique 1: search-based

- Idea: enumerate programs from grammar systematically and test them on examples
- Observation 1: exhaustive, exponential
- Observation 2: random enumeration order may not work well
- Challenges — scalability & ambiguity
  - Today’s lecture: two systematic search-based approaches (top-down & bottom-up)
  - Subsequent paper presentations: scale, resolve ambiguity
Top-down search

• We have already seen how this works

• CFG: $e ::= x \mid 1 \mid e + e$

• Idea: start from start symbol, expand non-terminal symbols according to production rules, until reaching a program that satisfies examples
Top-down search (cont’d)

- Algorithm skeleton
  
  top-down-search((T, N, P, S), E):
  
  worklist := { S };
  
  while ( worklist is not empty ):
    pp := worklist.remove();
    if ( pp is complete & pp satisfies E ):
      return pp;
    worklist.addAll(expand(pp));

  return more partial programs by replacing a non-terminal in pp

T: terminal symbols
N: non-terminal symbols
P: productions
S: start symbol
An example

top-down-search( (T, N, P, S), E ):
worklist := { S };
while ( worklist is not empty ):
    pp := worklist.remove();
    if ( pp is complete & pp satisfies E ): return pp;
worklist.addAll( expand(pp) );

• CFG: e := x | 1 | e + e
• Example: (1,2)
• Worklist (at end of iterations)
  iter 0: e
  iter 1: x 1 e + e
  iter 2: 1 e + e
  iter 3: e + e
  iter 4: x + e 1 + e e + e + e
      e + x e + 1 e + e + e
  iter 5: x + x x + 1 x + e + e
      1 + e e + e + e
      e + x e + 1 e + e + e
  iter 6: return x + x
Bottom-up search

- Idea: start with terminal symbols, combine smaller programs into bigger programs according to production rules, until reaching a program that satisfies examples

- Algorithm skeleton

```java
bottom-up-search( (T, N, P, S), E ):
  worklist := { t | t ∈ T };
  while ( true ):
    foreach p in worklist: if ( p is complete & p satisfies E ): return p;
    worklist.addAll( grow(worklist) );

  return more programs by applying production rules to programs in worklist
```
An example

bottom-up-search( (T, N, P, S), E ):

    worklist := { t | t ∈ T };

    while ( true ):
        foreach p in worklist: if ( p is complete & p satisfies E ): return p;
        worklist.addAll( grow(worklist) );

• CFG:  e := x | 1 | e + e
• Example: (1,2)
• Worklist (at end of iterations)
  - iter 0:  x  1
  - iter 1:  x  1  x + x  x + 1  1 + x  1 + 1
  - iter 2:  return x + x
Top-down vs. bottom-up

Top-down

- Generate programs top-down
- Candidates in worklist are partial programs

Bottom-up

- Generate programs bottom-up
- Candidates are concrete programs

Both exhaustive and brute-force procedures (both can be implemented using worklist algorithm)
Search-based approaches: scalability & ambiguity

• Scalability — how to make search faster?
  • Pruning
    • Top-down: eliminate “incorrect” partial programs
    • Bottom-up: discard “unpromising” sub-programs
  • Prioritization
    • Better order of candidates in worklist

• Ambiguity — how to find intended program (not arbitrary one satisfying examples)?
  • Ranking (similar idea to prioritization)

• Will talk more in paper presentations
PBE technique 2: representation-based

- Idea: represent search space **explicitly**, then use representation to better guide search

- CFG: \( e := x \mid 1 \mid e + e \)

- Challenge: how to construct representation efficiently, how to use it for synthesis
Different representations

- Version space algebras (VSAs) [Gulwani et al. 11]
- Finite tree automata (FTAs) [Wang et al. 17]
- Petri nets [Feng et al. 17]
- Type-transitions nets [Guo et al. 20]
Version space algebra

- Idea: construct a compact data structure (i.e., an VSA) that succinctly represents all programs consistent with examples

- Construction is top-down
  - FlashFill paper [Gulwani 11] has more details (will discuss in presentation)
  - [Polozov et al. 15] — VSA-based program synthesis framework
Finite tree automaton

- Idea: construct a compact data structure (i.e., an FTA) that succinctly represents all programs consistent with examples
  - Same idea as VSA, but different data structure

- Construction is bottom-up
  - Dace paper [Wang et al. 17] has more details (will discuss in presentation)
  - [Wang et al. 18] — FTA-based program synthesis framework
Summary of this lecture

• Syntax-guided synthesis (SyGuS): both semantic and syntactic constraints

• Programming-by-example (PBE): examples as spec

• Two PBE techniques: search-based & representation-based