Conditional Program Generation for Bimodal Program Synthesis

Swarat Chaudhuri
Rice University
www.cs.rice.edu/~swarat

(Joint work with Chris Jermaine, Vijay Murali, and Letao Qi)
Program synthesis

**Specification**

**Synthesizer**

**Program**

+ **Correctness Certificate**

**Specification**: Logical constraint that must be satisfied exactly

**Algorithm**: Search for a program that satisfies the specification.
“Bimodal” program synthesis

An idealized program

Ambiguous “evidence” + Logical requirements

Candidate implementations

Prior distribution

Learned from a real-world code corpus

Synthesizer

Posterior distribution over programs

“Bimodal” program synthesis

- API calls or types that the program uses
- “Soft” I/O examples or constraints
- Natural language description of what the program does
- ...

An idealized program

Ambiguous “evidence” + Logical requirements

Prior distribution

Learned from a real-world code corpus

Synthesizer

Candidate implementations

Posterior distribution over programs
The Bayou synthesizer: A demo

Conditional program generation

Assume random variables $X$ and $Prog$, over labels and programs respectively, following a joint distribution $Q(X, Prog)$.

**Offline:**
- You are given a set $\{(X_i, Prog_i)\}$ of samples from $Q(X, Prog)$. From this, learn a function $g$ that maps evidence to programs.
- **Learning goal:** maximize $E_{(X, Prog) \sim Q}[I]$, where

$$I = \begin{cases} 
1 & \text{if } g(X) \equiv Prog \\
0 & \text{otherwise.}
\end{cases}$$

**Online:** Given $X$, produce $g(X)$. 

In what we actually do

The map $g$ is probabilistic.

Learning is maximum conditional likelihood estimation:

- Given $\{(X_i, Prog_i)\}$, solve $\arg\max_{\theta} \sum_i \log P(Prog_i | X_i, \theta)$. 
Programs

Language capturing the essence of API usage in Java.

\[
\text{Prog} ::= \text{skip} \mid \text{Prog}_1; \text{Prog}_2 \mid \text{call Call} \mid \\
\quad \text{let } x = \text{Call} \mid \\
\quad \text{if Exp then Prog}_1 \text{ else Prog}_2 \mid \\
\quad \text{while Exp do Prog}_1 \mid \text{try Prog}_1 \text{ Catch} \\
\text{Exp} ::= \text{Sexp} \mid \text{Call} \mid \text{let } x = \text{Call} : \text{Exp}_1 \\
\text{Sexp} ::= c \mid x \\
\text{Call} ::= \text{Sexp}_0.\text{a}(\text{Sexp}_1, \ldots, \text{Sexp}_k) \\
\text{Catch} ::= \text{catch}(x_1) \text{ Prog}_1 \ldots \text{catch}(x_k) \text{ Prog}_k
\]
Labels

Set of API calls
• readline, write,...

Set of API datatypes
• BufferedReader, FileReader,...

Set of keywords that may appear while describing program actions in English
• read, file, write,...
• Obtained from API calls and datatypes through a camel case split
Challenges

Directly learning over source code simply doesn’t work

• Source code is full of low-level, program-specific names and operations.

• Programs need to satisfy structural and semantic constraints such as type safety. Learning to satisfy these constraints is hard.
Language abstractions to the rescue!

Learn not over programs, but typed, syntactic models of programs.
Sketches

The sketch of a program is obtained by applying an abstraction function $\alpha$.

From sketch $Y$ to program $Prog$: a fixed concretization distribution $P(Prog \mid Y)$.

Learning goal changes to
• Given $\{(X_i, Y_i)\}$, solve $\arg \max_{\theta} \sum_i \log P(Y_i \mid X_i, \theta)$.
Sketches

\[ Y ::= \text{Call} \mid \text{skip} \mid \text{while Cond do } Y_1 \mid Y_1; Y_2 \mid \text{try } Y_1 \ \text{Catch} \mid \text{if Cond then } Y_1 \ \text{else } Y_2 \]

\[ \text{Catch} ::= \text{catch}(\tau_1) Y_1 \ldots \text{catch}(\tau_k) Y_k \]

\[ \text{Cond} ::= \{\text{Call}_1, \ldots, \text{Call}_k\} \]

\[ \text{Call} ::= a(\tau_1, \ldots, \tau_k) \]

Conditions replaced by sets of abstract API calls

Abstract API call

API method name

Types
Program synthesis

Evidence $X$ $\xrightarrow{}$ $P(Y \mid X)$

Logical requirement $\varphi$

Learned from $(X_i, Y_i)$ pairs

Sample sketches

Implementations satisfying $\varphi$

Type-directed, compositional synthesizer

Sketch $\rightarrow$ Executable code
Program synthesis

Implementations satisfying \( \varphi \)

*Sketch \( \rightarrow \) Executable code*

**Combinatorial “concretization”**

Sample sketches

\( P(Y \mid X) \)

End-to-end differentiable neural architecture

Learned from \((X_i, Y_i)\) pairs

Type-directed, compositional synthesizer

Not all sketches may be realizable as executable programs

\( \text{Log } \) requirement \( \varphi \)
Learning using a probabilistic encoder-decoder

$X$: Evidence
$Y$: Sketches
$Z$: Latent “intent”

Encoder $f$

Decoder $g$

Feedforward neural net

Representation of hidden intent

Prior for regularization
Learning using a probabilistic encoder-decoder

\[ \begin{align*}
X & \rightarrow \text{Encoder } f \\
& \rightarrow f(X) \\
& \rightarrow \text{Decoder } g \\
& \rightarrow Y
\end{align*} \]

\[ P(Z) = \text{Normal}(0, I) \]
\[ P(f(X) | Z) = \text{Normal}(Z, \sigma^2 I) \]

During learning, use Jensen’s inequality to get smooth loss function

\[ X: \text{Evidence} \]
\[ Y: \text{Sketches} \]
\[ Z: \text{Latent “intent”} \]
Learning using a probabilistic encoder-decoder

\[ X: \text{Evidence} \]
\[ Y: \text{Sketches} \]
\[ Z: \text{Latent “intent”} \]

\[ P(Z) = \text{Normal}(0, I) \]
\[ P(f(X) | Z) = \text{Normal}(Z, \sigma^2 I) \]

During inference, get \( P(Z | X) \) using normal-normal conjugacy
Neural decoder

Distribution on rules that can be fired at a point, given history so far.

History encoded as a real vector.
Concretization

Production rule in grammar for concrete language

Ruled out by type system
Results

• Trained method on 100 million lines of Java/Android code. ~2500 API methods, ~1500 types.

• Synthesis of method bodies from scratch, given 2-3 API calls and types.

• Sketch learning critical to accuracy.

• Good performance compared to GSNNs (state of the art conditional generative model).

• Good results on label-sketch pairs not encountered in training set.
Thank you!

Questions?

swarat@rice.edu
http://www.cs.rice.edu/~swarat

(Research funded by the DARPA MUSE award #FA8750-14-2-0270)