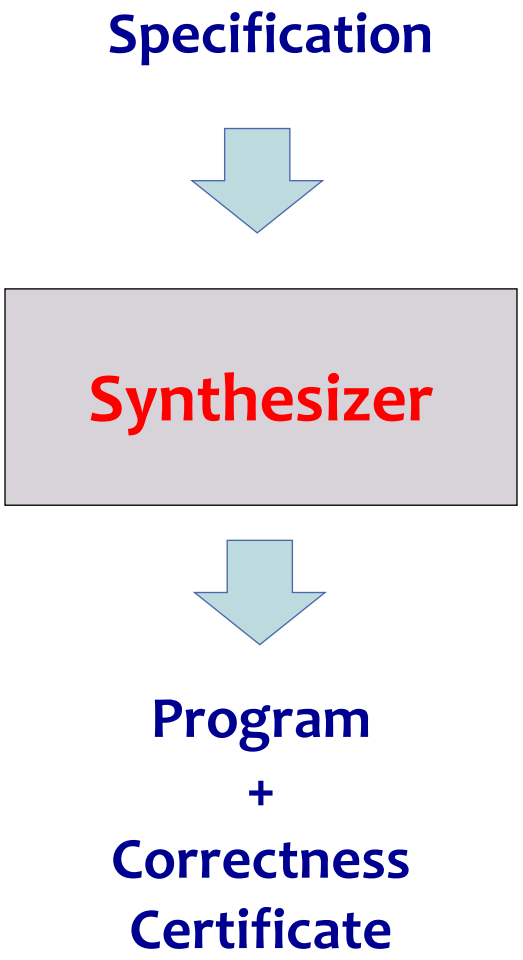
Deep Learning for Program Synthesis

Swarat Chaudhuri
Rice University

Guest lecture in Reliable and Interpretable Artificial Intelligence, Fall 2017

Program synthesis

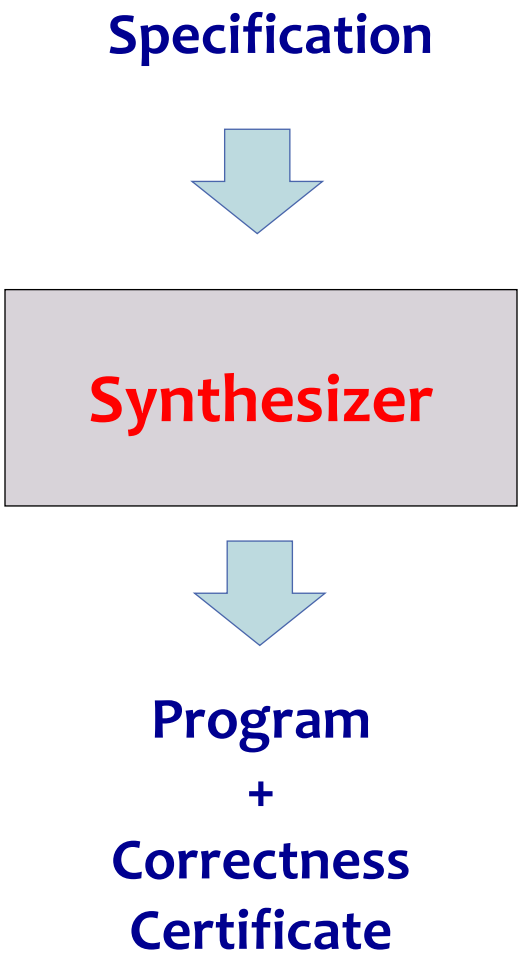
[Simon 1963, Summers 1977, Manna-Waldinger 1977, Pnueli-Rosner 1989]



Specification: Logical constraint that must be satisfied exactly

Algorithm: Search for a program that satisfies the specification.

Program synthesis



Main questions:

1. How to search vast, combinatorial spaces of programs

2. How to express intent and embed algorithms into a larger design process

An old problem in artificial intelligence

[Simon 1963, Summers 1977, Manna-Waldinger 1977, Pnueli-Rosner 1989]

What's different this time?

- More powerful hardware
- New applications
- Diligent engineering
- …but also, new algorithmic insights.
  - In this lecture: union of ideas from the symbolic and connectionist traditions in AI.

Example: Program Sketching

[Solar-Lezama et al. 2005, 2006]

```
void find (Set S, int key, ref Node prev, ref Node cur) {
  while (cur.key < key) {
        reorder {
            prev = cur;
            cur = cur.next;
            if (??) { lock (??); }
            if (??) { unlock (??); }
        }
}
```

Insight: Synthesizing small parts of programs can still be useful

Question: How to reduce human involvement?

Example: Flash Fill [Gulwani, 2011]

Input	Output
(425)-706-7709	425-706-7709
510.220.5586	510-220-5586
1 425 235 7654	425-235-7654
425 745-8139	425-745-8139

Synthesis of Excel macros from input-output examples.

"One of the shock-and-awe features of Excel 2013." — Ars Technica

Insight: Synthesizing simple programs can still be useful

Question: How to extend to more complex programs?

This lecture: Combining search/logic and learning for program synthesis

- Combinatorial program synthesis
  - The benefits of language abstractions [Feser et al. 2015]
  - The power of deduction [Feser et al. 2015; Feng et al. 2017]
- How learning helps [Murali et al. 2017]

Language abstractions

Example 1: Data structure transformation

Transforming nested lists

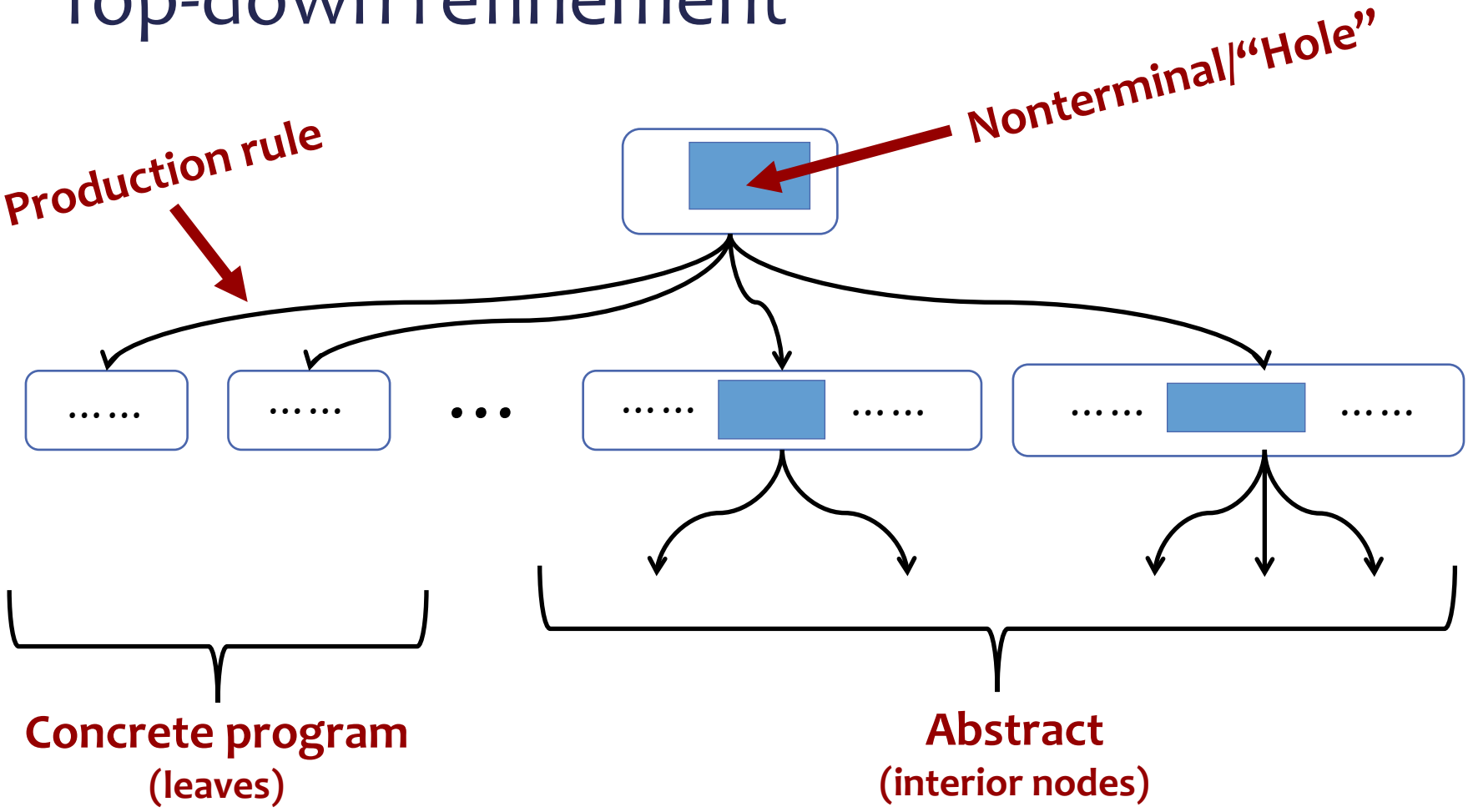
- Teacher has list of scores for each student
- Single nested list, where each sub-list contains a single student's scores
- Want to drop each student's lowest score.

Given a set of input-output examples

$$\texttt{dropmins}\ [[1,3,5],[6,4,2]] = [[3,5],[6,4]]$$

Synthesizing Data Structure Transformations from Input-Output Examples. Feser, Chaudhuri, Dillig. PLDI 2015.

Top-down refinement



Functional representations

Many benefits

- Easy composition of simpler programs into larger ones
- Abstract common recursion schemes as higher-order *combinators*
- More common for representations to be canonical.

More complex programs with a smaller amount of search.

$$\lambda x.\ \texttt{map}\ x\ (\lambda y.\ g^*)$$

vs.

```
it = x.iterator();
while (it.hasNext()) {
      y = it.next();
      out.add(g*(y));
}
return out;
```

Transforming nested lists

$$\texttt{dropmins} :: \texttt{list}[\texttt{list}[\texttt{int}]] \rightarrow \texttt{list}[\texttt{list}[\texttt{int}]]$$

$$\texttt{dropmins}\ [[1,3,5],[6,4,2]] = [[3,5],[6,4]]$$

```
dropmins x = map dropMin x
    where dropMin y = filter isMin y
          where isMin z = foldl h False y
                where h t w = t || (w < z)
```

Synthesis:

- Assume a fixed set of first-order API procedures and higher-order combinators
- Search the space of type-safe compositions.

Logic and deduction

Problem: The space of programs is still too large

- More pruning needed
- Logical deduction can help.

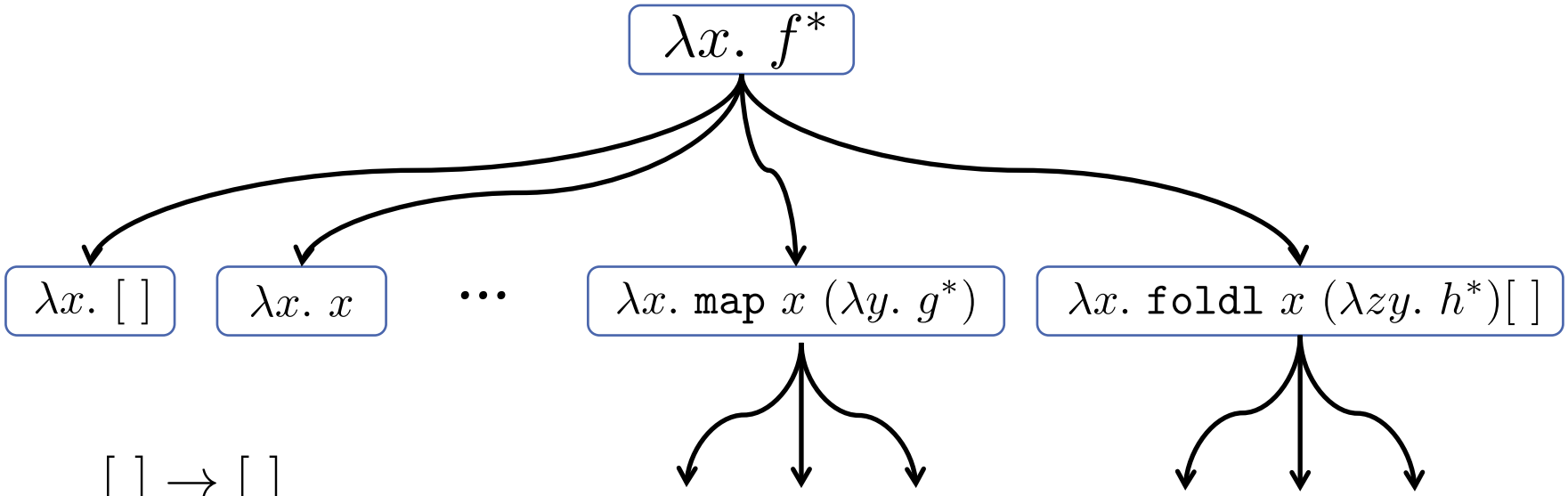
Example 1.1: list reverse

$$\texttt{reverse} :: \texttt{list}[\texttt{int}] \to \texttt{list}[\texttt{int}]$$

$$\begin{aligned}
\texttt{reverse}[\,] &= [\,] \\
\texttt{reverse}[2] &= [2] \\
\texttt{reverse}[2\ 1] &= [1\ 2] \\
\texttt{reverse}[2\ 1\ 3] &= [3\ 1\ 2]
\end{aligned}$$

Solution: $\lambda x.\ \texttt{foldl}\ x\ (\lambda z y.\ y : z)\ [\,]$

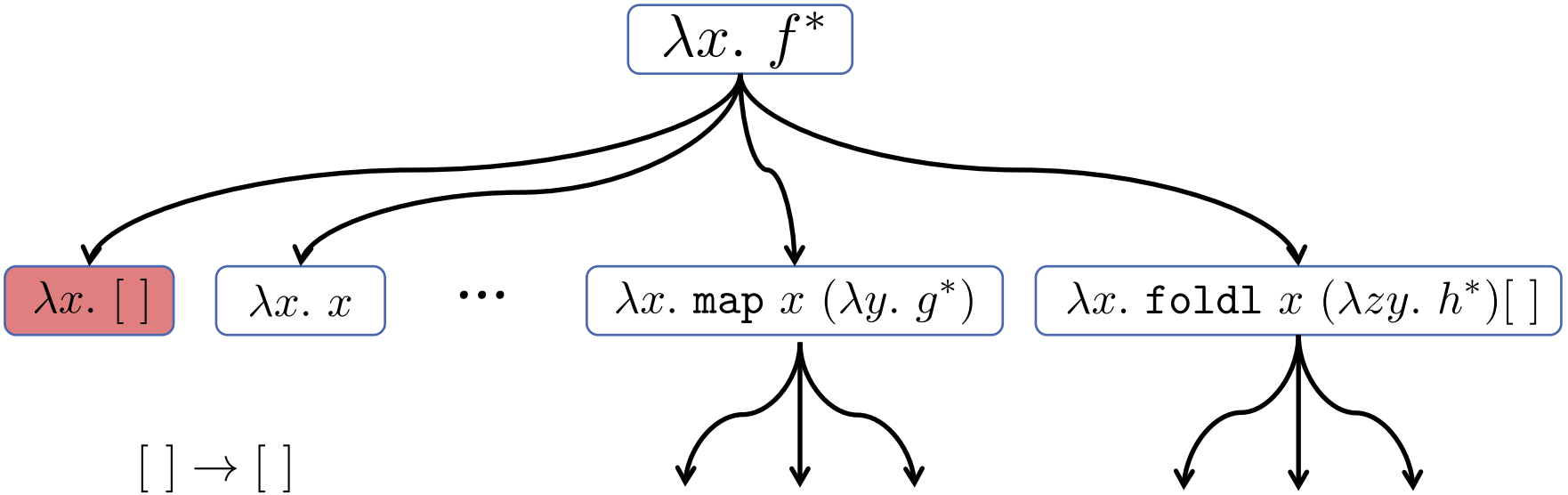
Checking concrete programs



$$[\,] \rightarrow [\,]$$
$$[2] \rightarrow [2]$$
$$[2\ 1] \rightarrow [1\ 2]$$
$$[2\ 1\ 3] \rightarrow [3\ 1\ 2]$$

Evaluate hypothesis on each input,
check that the
result equals the expected output

Checking concrete programs



$$[\,] \rightarrow [\,]$$
$$[2] \rightarrow [2]$$
$$[2\ 1] \rightarrow [1\ 2]$$
$$[2\ 1\ 3] \rightarrow [3\ 1\ 2]$$

Evaluate program on each input

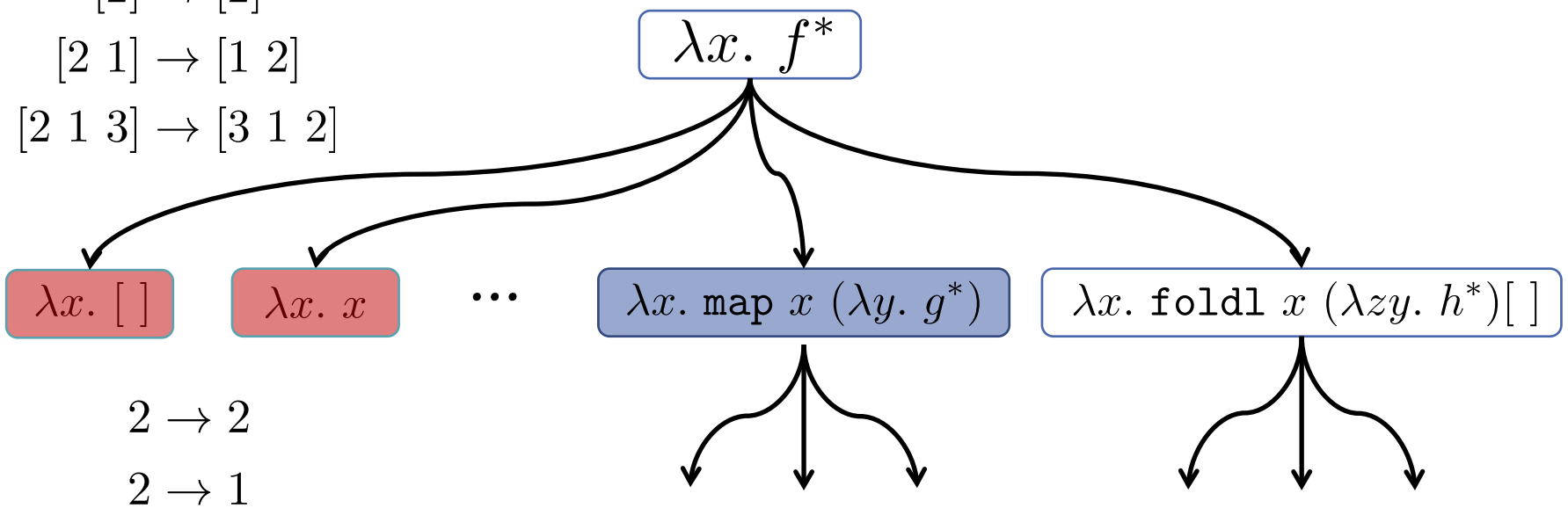
Checking abstract programs

$[\,] \rightarrow [\,]$

$[2] \rightarrow [2]$

$[2\ 1] \rightarrow [1\ 2]$

$[2\ 1\ 3] \rightarrow [3\ 1\ 2]$



$2 \rightarrow 2$

$2 \rightarrow 1$

$1 \rightarrow 2$

$2 \rightarrow 3$

$1 \rightarrow 1$

$3 \rightarrow 2$

Abstract programs are checked by deducing examples for their holes

Generate examples for g^*

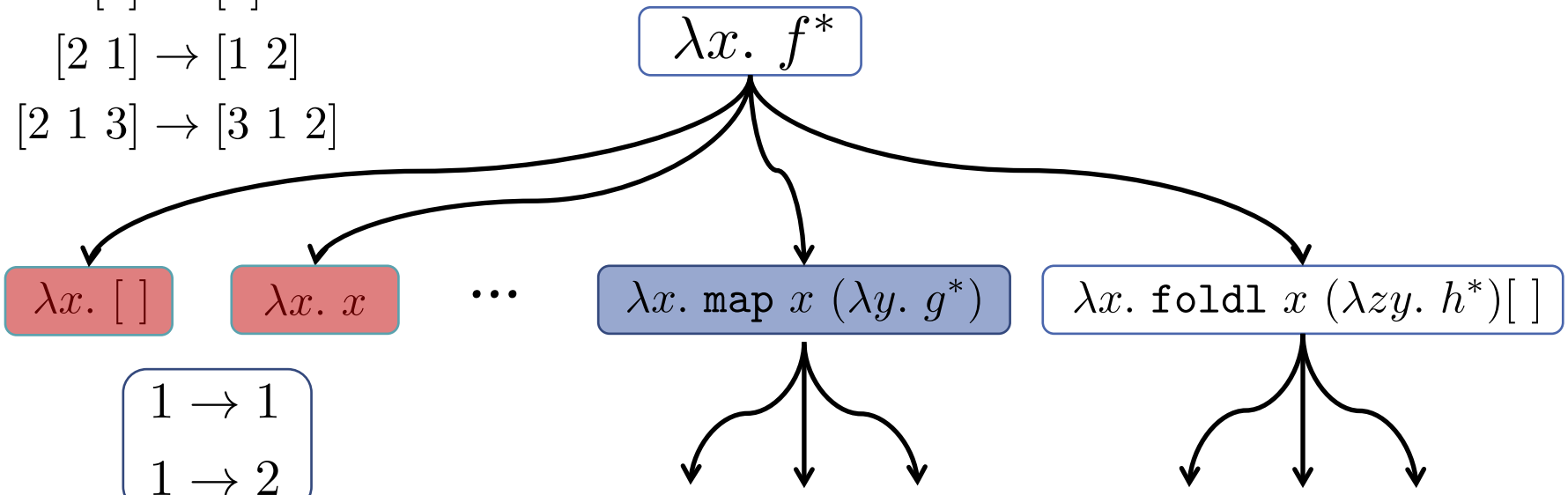
Conflict detection

$[\,] \to [\,]$

$[2] \to [2]$

$[2\ 1] \to [1\ 2]$

$[2\ 1\ 3] \to [3\ 1\ 2]$



$1 \to 1$

$1 \to 2$

$2 \to 1$

$2 \to 2$

$2 \to 3$

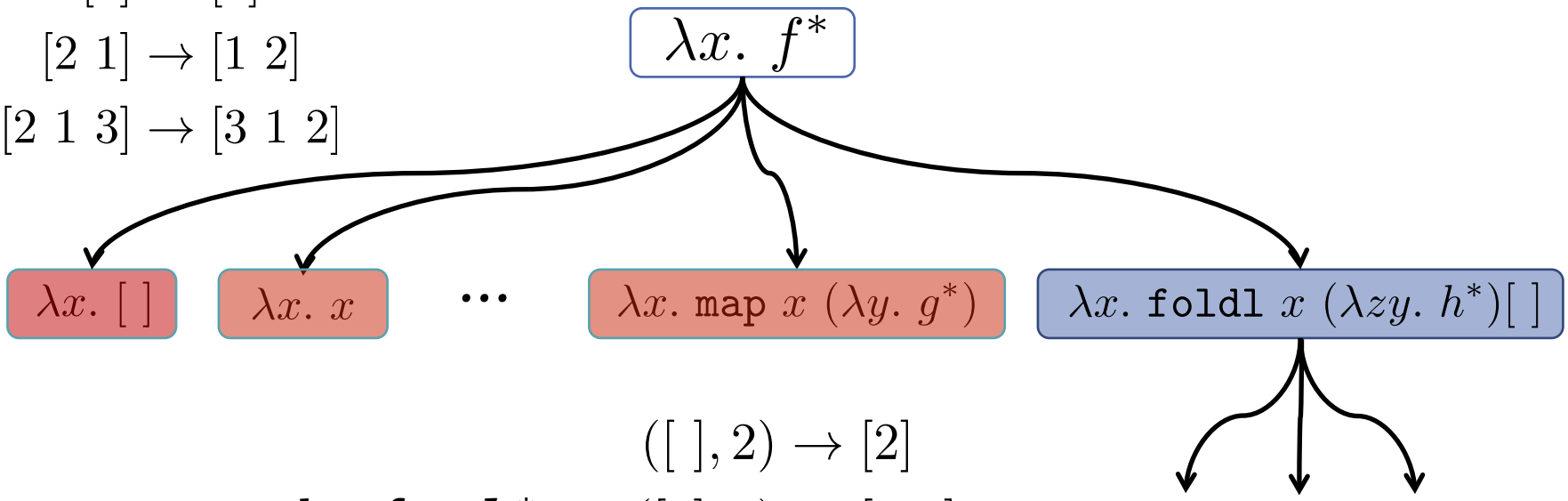
$3 \to 2$

$\lambda x.\ f^* :: \texttt{list[int]} \to \texttt{list[int]}$

$\lambda y.\ g^* :: \texttt{int} \to \texttt{int}$

Subgoal generation

$$[\,] \to [\,]$$
$$[2] \to [2]$$
$$[2\ 1] \to [1\ 2]$$
$$[2\ 1\ 3] \to [3\ 1\ 2]$$



Examples for h^*:

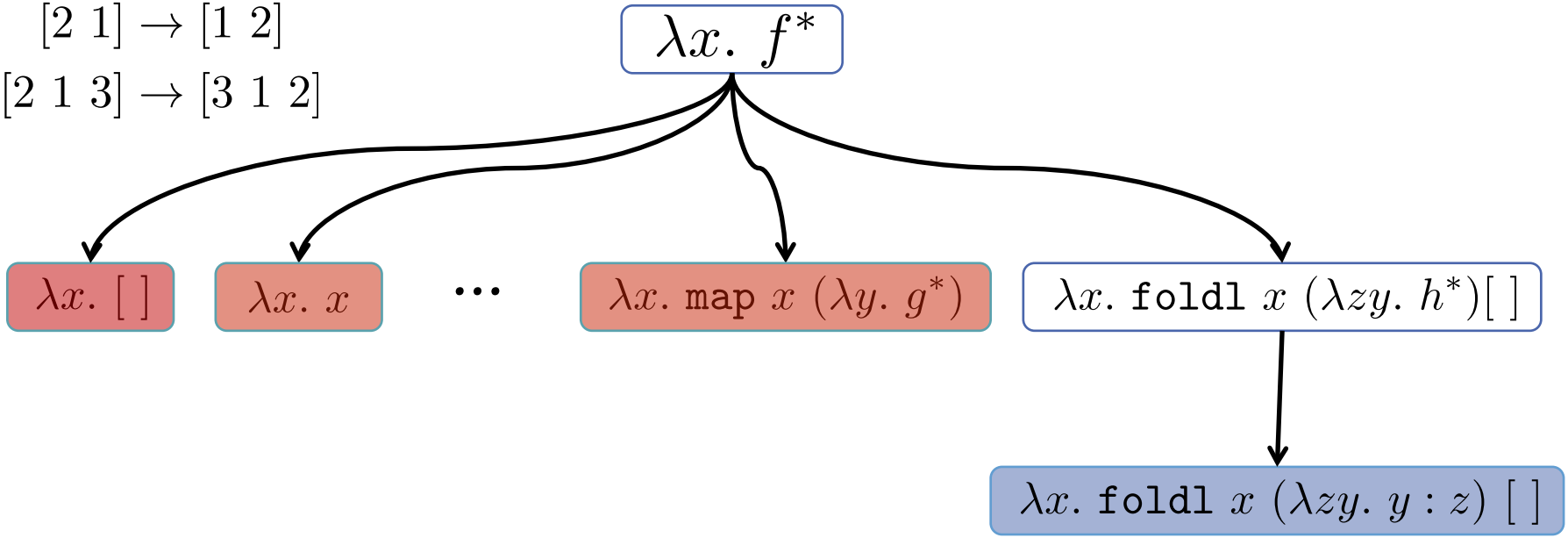
$$([\,], 2) \to [2]$$
$$([2], 1) \to [1\ 2]$$
$$([1\ 2], 3) \to [3\ 1\ 2]$$

$[\,] \to [\,]$

$[2] \to [2]$

$[2\ 1] \to [1\ 2]$

$[2\ 1\ 3] \to [3\ 1\ 2]$



SMT-solvers for deduction

SMT solvers: Automatic satisfiability checkers for quantifier-free first-order logic

- Major advances in the last 20 years
- Numerous applications in formal methods

Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples.
Feng, Martins, Van Geffen, Dillig, and Chaudhuri. PLDI 2017.

SMT-solvers for deduction

Logical specs for API procedures

- `filter`(t, p): given a table t, select the maximal subset of rows satisfying a given predicate p

$$t_{out}.rows < t_{in}.rows$$

$$t_{out}.columns = t_{in}.columns$$

When considering an abstract program

- Compose API procedure specs to produce constraint that specifies the abstract program
- Check if this constraint is consistent with goal (examples)
- If not consistent, prune.

Results: Search + functional abstractions + deduction

λ^2 [Feser et al., 2015]

- Many Functional Programming 101 examples
- The "first functional pearl": Strachey's Cartesian product of lists
- Median number of examples = 4; Median runtime < 1 sec;

Morpheus [Feng et al., 2017]

- Common data preparation tasks in R, picked up from online fora
- API procedures from two popular R libraries
- From a single example transformation; median runtime < 5 sec

Strachey's Cartesian product of lists

```
cprod xss =
    foldr f [[]] xss
       where f xs yss = foldr g [] xs
           where g x zss = foldr h zss yss
                 where h ys qss = cons (cons (x, ys), qss)
```

Danvy & Spivey wrote a paper [ICFP07] just to explain the program.

$\boldsymbol{\lambda}^2$ [Feser et al., 15] can synthesize it from four input-output examples.

How machine learning helps

Problem 1: Scalability

How do we go from functional programs over two APIs to programs with over hundreds of APIs and complex control flow?

- Manipulate files using the Java.io package
- Open a bluetooth connection
- Create a dialog box
- Read a CSV file, parse it, then copy the contents into a dictionary
- …

Problem 2: Specification

As the targets of synthesis become more complex, how do you specify them?

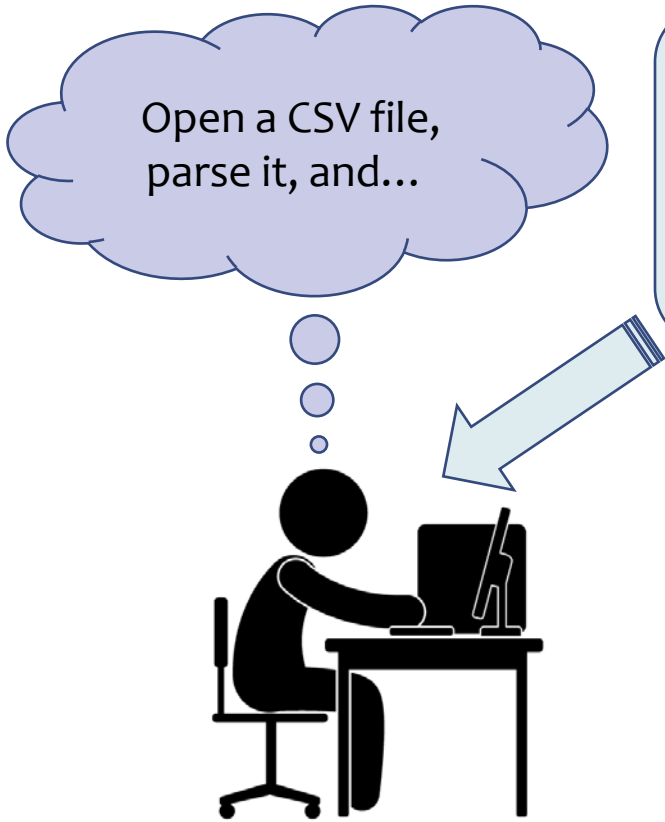
- Manipulate files using the Java.io package
- Open a bluetooth connection
- Create a dialog box
- Read a CSV file, parse it, then copy the contents into a dictionary
- …

A deeper problem

- In practical program synthesis, we do not have full formal specifications, but underspecifications.
- How do you rule out meaningless output?
  - ✓ Switch-statements that trivially match a set of input-output examples
- Usual solution: ad hoc cost heuristics.
  - ✓ For example, impose high cost on switches with many branches.

Can we do better?

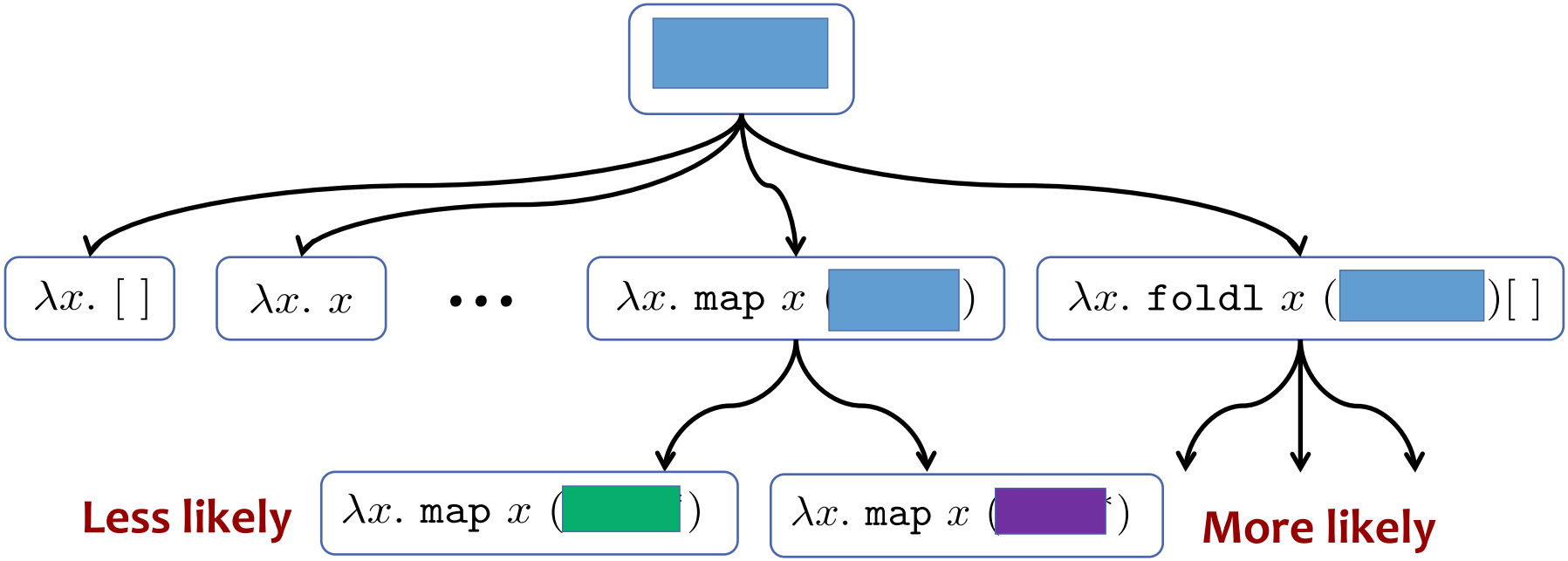
The “human” solution: Use data!



- Textbooks, documentation
- Forums, chats
- Other people’s code
- Personal experience

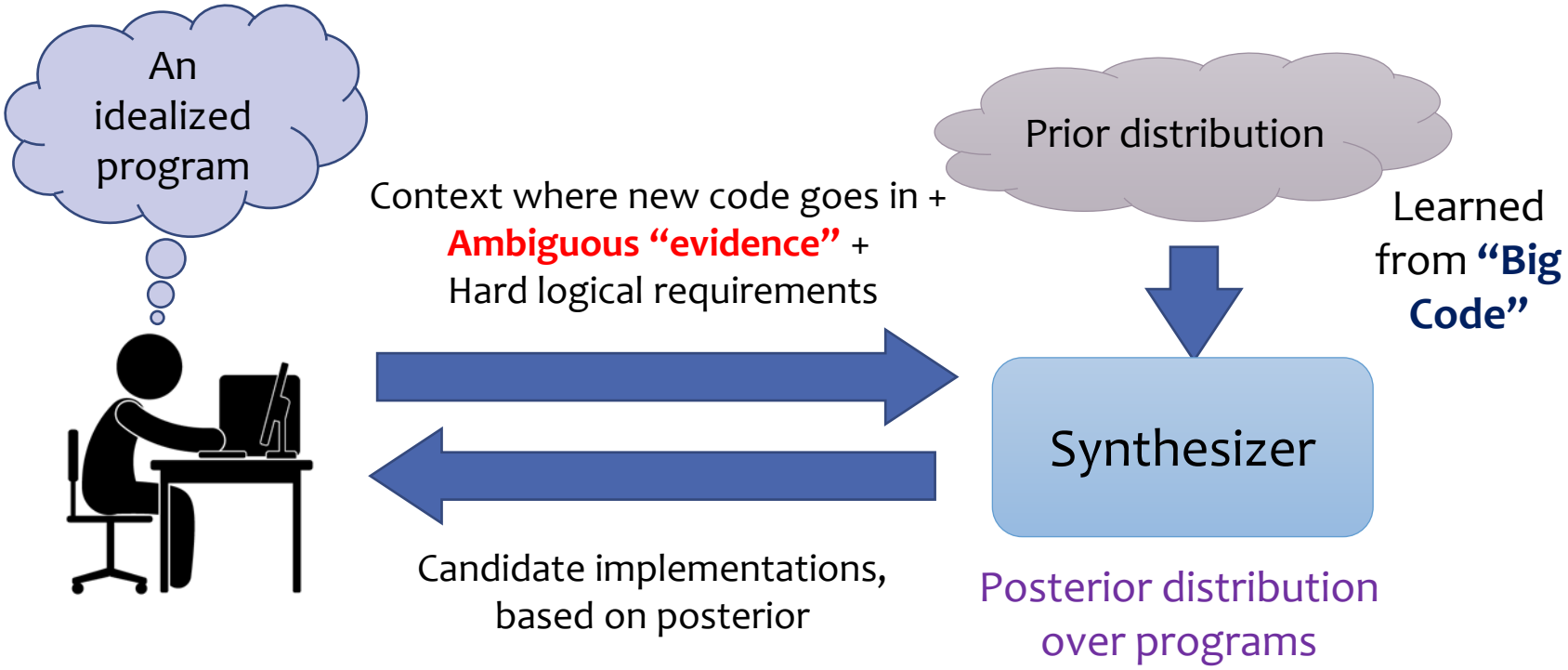
Humans use models learned from data to interpret ambiguous and incomplete specifications.

Data can help with scalability too



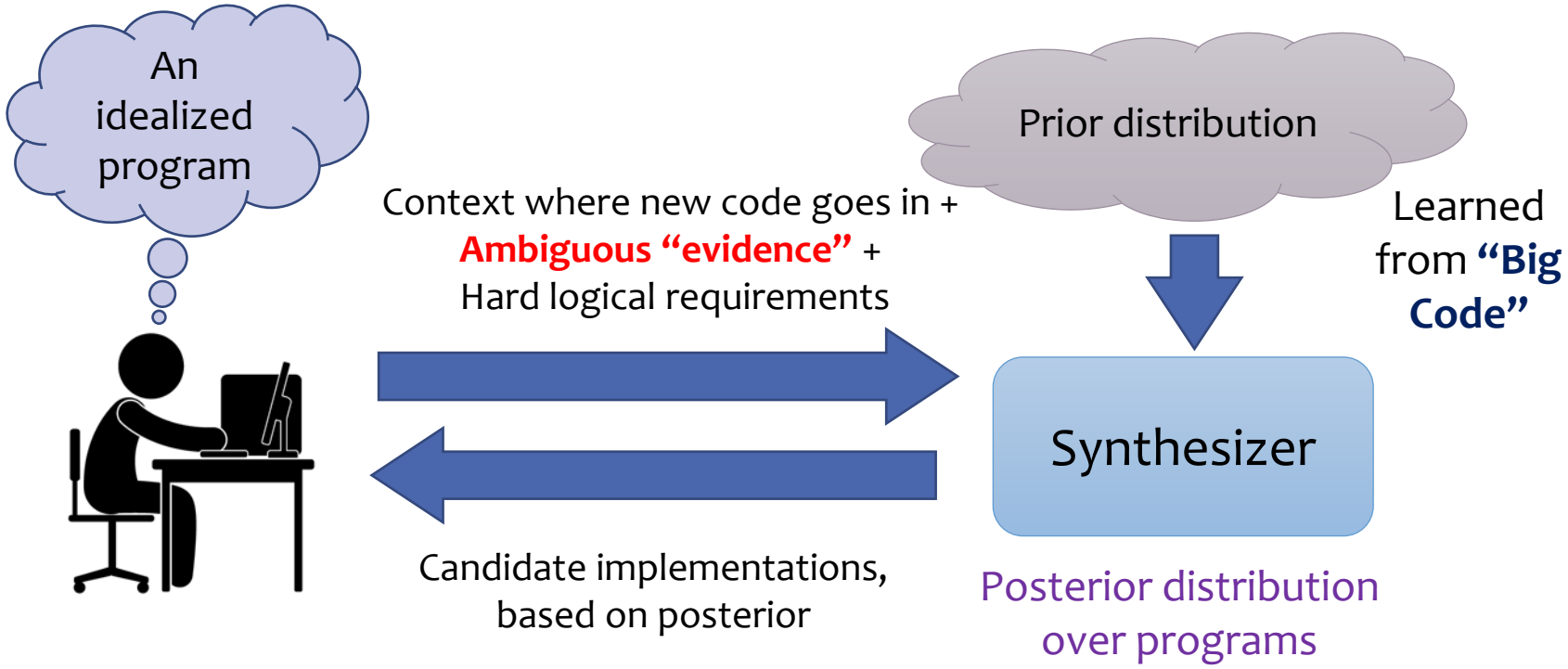
Structural patterns in code, learned from data, let us avoid "unreasonable" programs

Synthesis from ambiguous evidence



Neural Sketch Learning for Conditional Program Generation. Murali, Qi, Chaudhuri, and Jermaine. Arxiv 2017.

Synthesis from ambiguous evidence



- API calls or types that the program uses
- Traces or I/O examples that capture the program's behavior
- Natural language description of what the program does
- …

Example 2: File manipulation

Input

/// readline
/// FileReader

Output

```
void read(String file) {
   String s;
   BufferedReader br;
   FileReader fr;
    try {
       fr = new FileReader(file);
       br = new BufferedReader(fr);
       while ((s = br.readLine()) != null) {}
      br.close();
    }
   catch (FileNotFoundException _e) {}
   catch (IOException _e) {}
}
```

The Bayou synthesizer: A demo

http://bit.ly/2zZ5aE8

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.

$$
\begin{array}{lcl}
\text{Prog} & ::= & \mathbf{skip} \mid \text{Prog}_1 ; \text{Prog}_2 \mid \mathbf{call}\ \text{Call} \mid \\
 & & \mathbf{let}\ x = \text{Call} \mid \\
 & & \mathbf{if}\ \text{Exp}\ \mathbf{then}\ \text{Prog}_1\ \mathbf{else}\ \text{Prog}_2 \mid \\
 & & \mathbf{while}\ \text{Exp}\ \mathbf{do}\ \text{Prog}_1 \mid \mathbf{try}\ \text{Prog}_1\ \text{Catch} \\
\text{Exp} & ::= & \text{Sexp} \mid \text{Call} \mid \mathbf{let}\ x = \text{Call} : \text{Exp}_1 \\
\text{Sexp} & ::= & c \mid x \\
\text{Call} & ::= & \text{Sexp}_0.a(\text{Sexp}_1, \ldots, \text{Sexp}_k) \\
\text{Catch} & ::= & \mathbf{catch}(x_1)\ \text{Prog}_1\ \ldots\ \mathbf{catch}(x_k)\ \text{Prog}_k
\end{array}
$$

API call

API method name

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

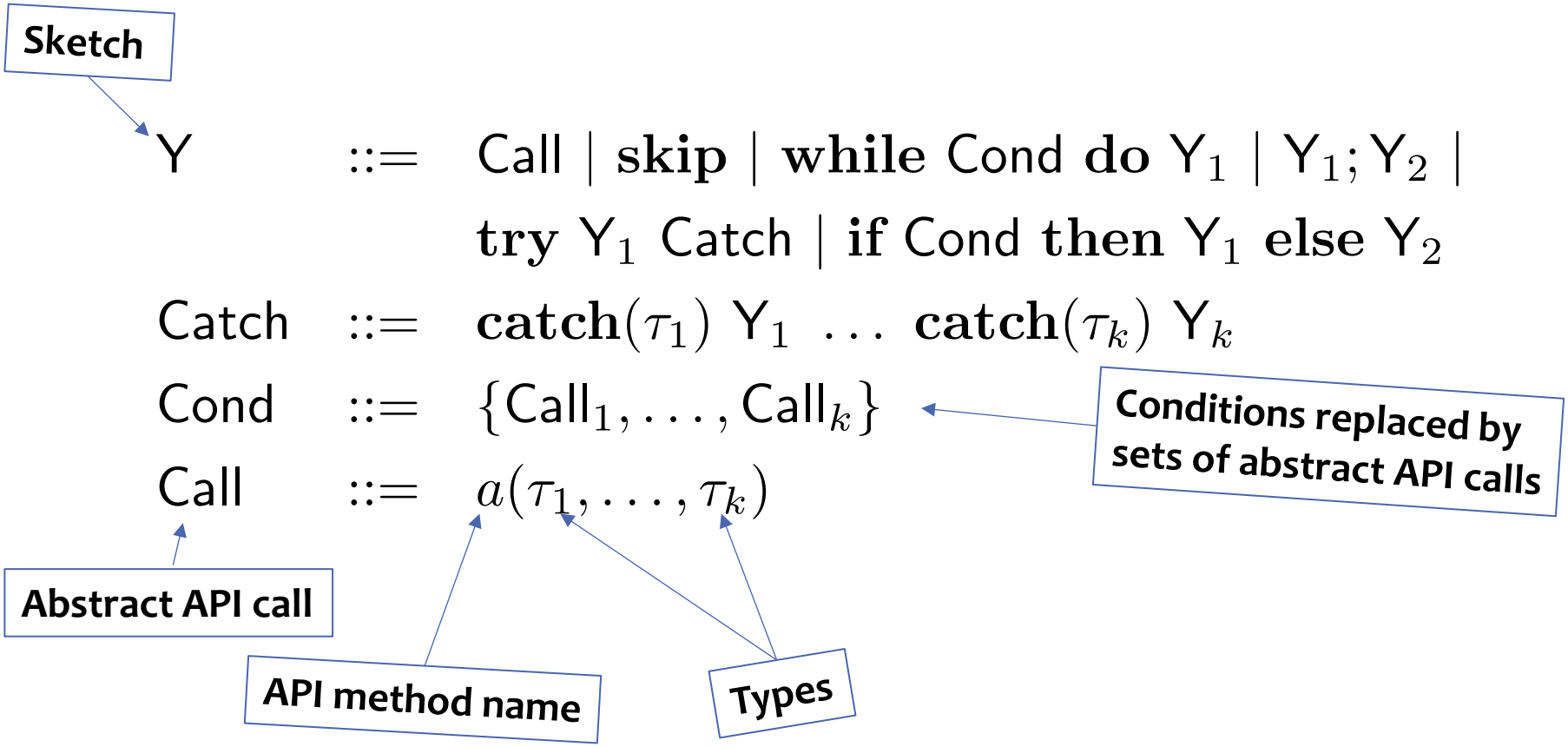
The sketch of a program is obtained by applying an **abstraction function** α.

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

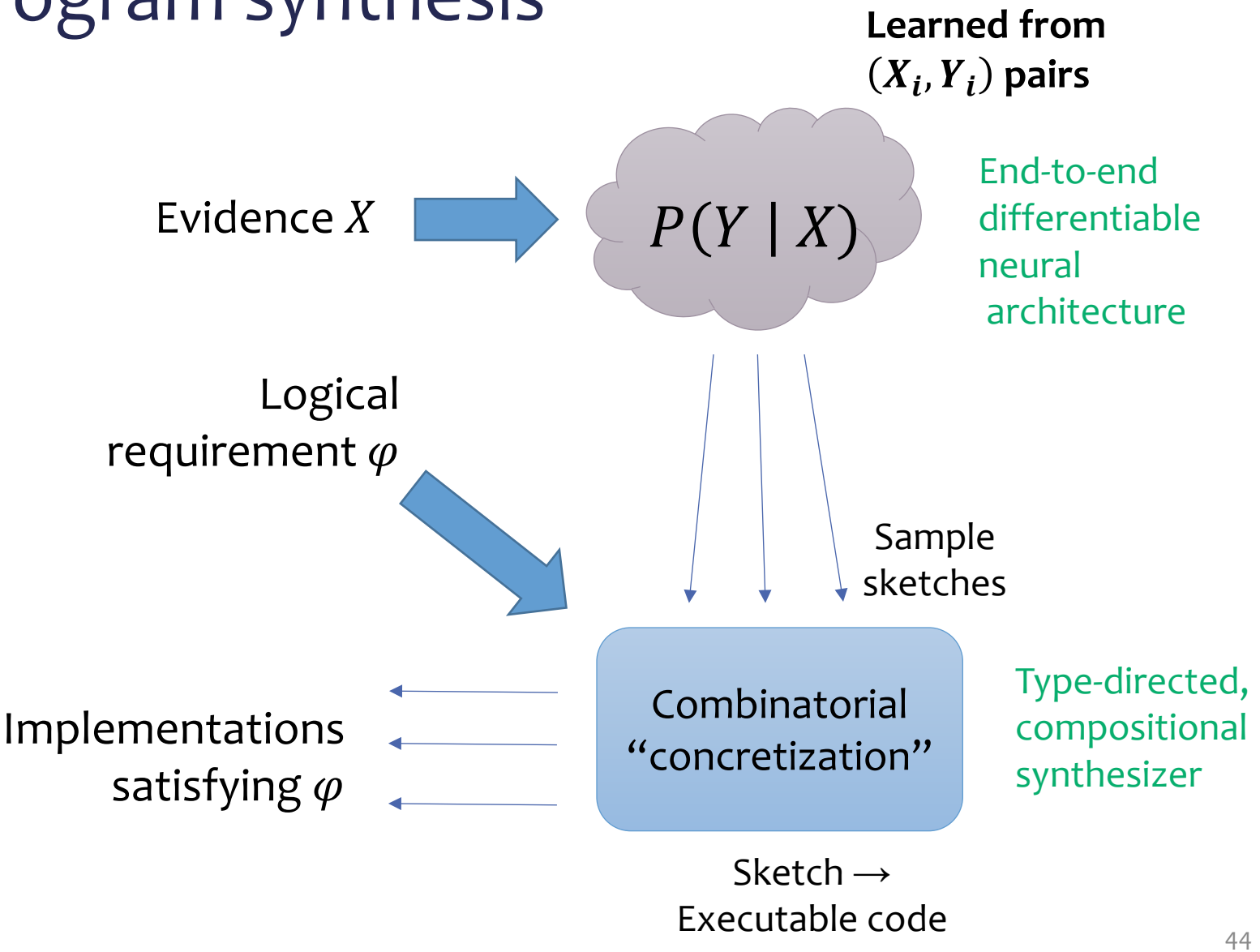
Learning goal changes to

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

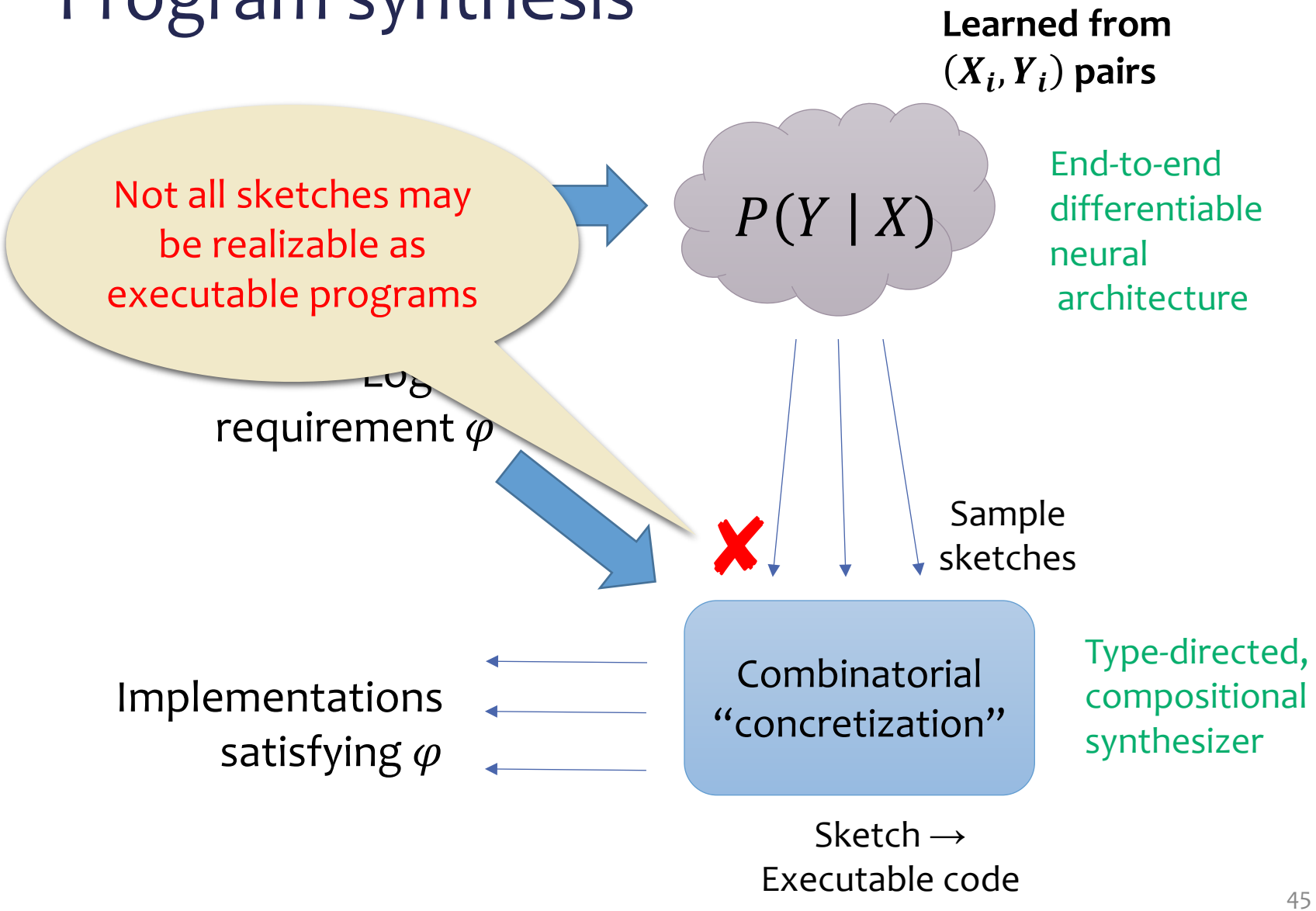
Sketches


Sketch
Y ::= Call | skip | while Cond do Y1 | Y1; Y2 |
try Y1 Catch | if Cond then Y1 else Y2
Catch ::= catch(τ1) Y1 ... catch(τk) Yk
Cond ::= {Call1, ..., Callk}
Call ::= a(τ1, ..., τk)
Conditions replaced by sets of abstract API calls
Abstract API call
API method name
Types

Program synthesis

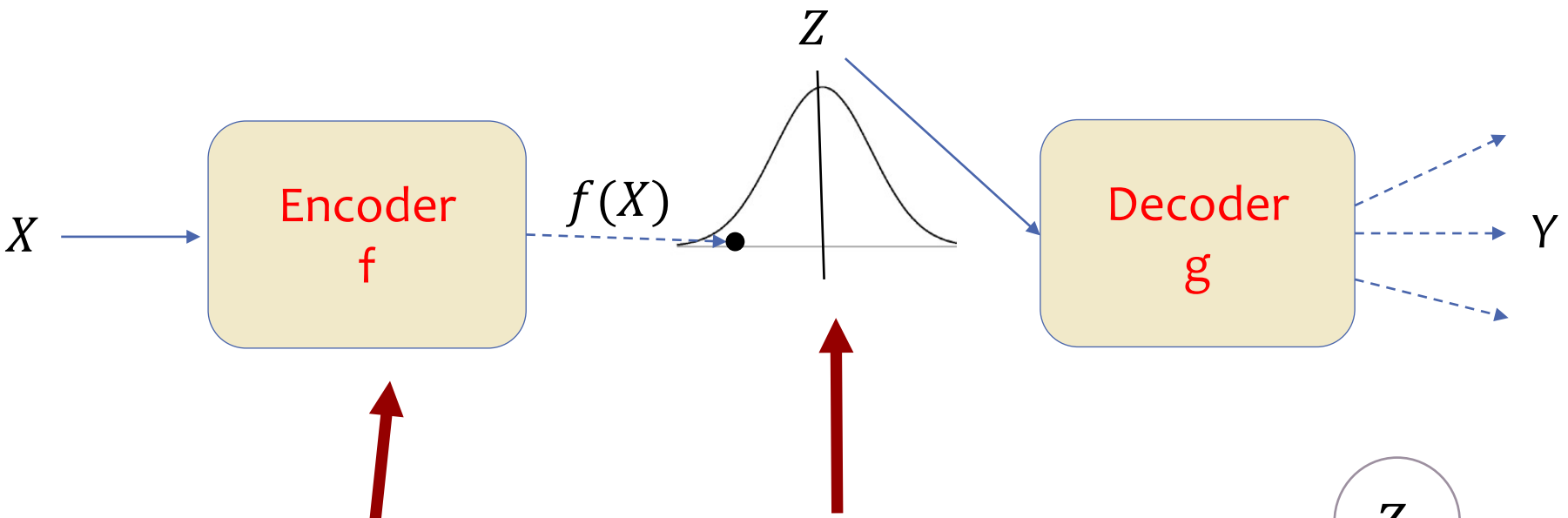


Program synthesis



Learning using Gaussian encoder-decoders

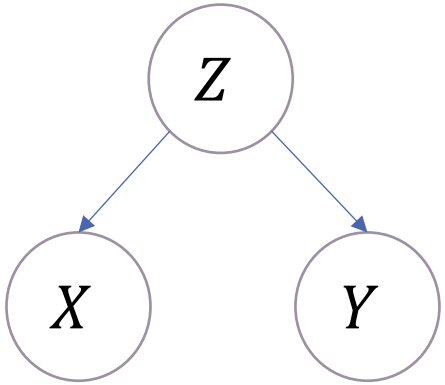
$$X = \langle X_{Types}, X_{Calls}, X_{Keys} \rangle$$



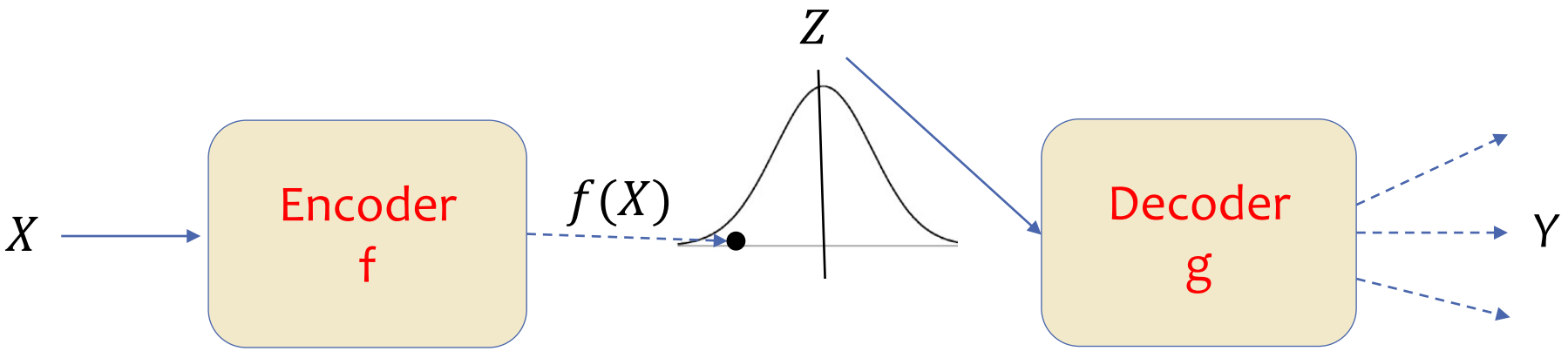
Feedforward neural net

Representation of hidden intent

Prior for regularization



Learning using Gaussian encoder-decoders



$$P(Z) = Normal(0, I)$$

$$P(f(X) \mid Z) = Normal(Z, \sigma^2 I)$$

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

$$\sum_i \log P(Y_i|X_i,\theta) = \sum_i \log \int_{Z\in\mathbb{R}^m} P(Z|X_i,\theta)P(Y_i|Z,\theta)dZ$$

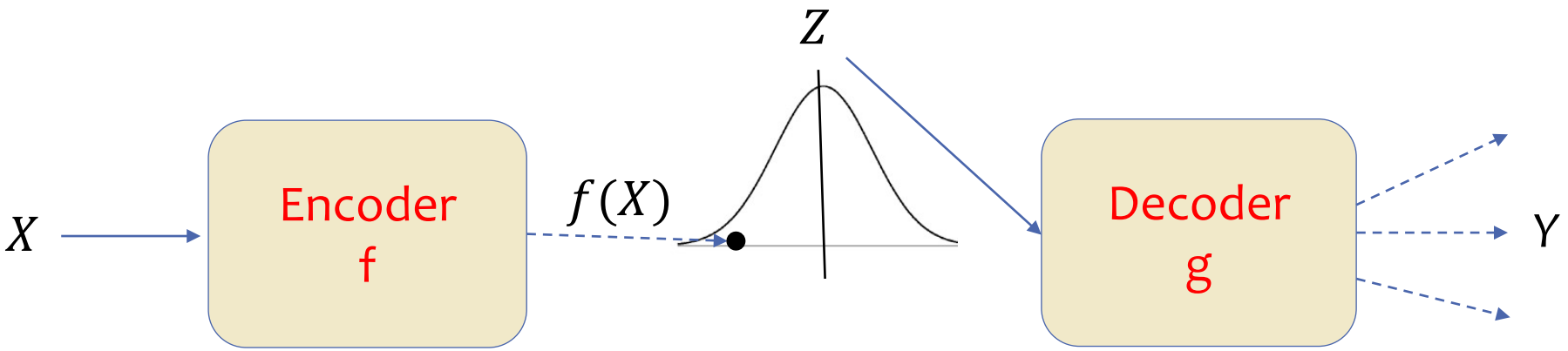
$$= \sum_i \log \mathbf{E}_{Z\sim P(Z|X_i,\theta)}[P(Y_i|Z,\theta)]$$

$$\geq \sum_i \mathbf{E}_{Z\sim P(Z|X_i,\theta)}[\log P(Y_i|Z,\theta)]$$

$$= \mathcal{L}(\theta).$$

Learning using Gaussian encoder-decoders

X: Evidence
Y: Sketches
Z: Latent "intent"



$$P(Z) = Normal(0, I)$$

$$P(f(X) \mid Z) = Normal(Z, \sigma^2 I)$$

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

$$
\begin{aligned}
P(X|Z,\theta) \quad = \quad & \left(\prod_j \text{Normal}(f(X_{Calls,j})|Z, \mathbf{I}\sigma^2_{Calls}) \right) \\
& \left(\prod_j \text{Normal}(f(X_{Types,j})|Z, \mathbf{I}\sigma^2_{Types}) \right) \\
& \left(\prod_j \text{Normal}(f(X_{Keys,j})|Z, \mathbf{I}\sigma^2_{Keys}) \right) .
\end{aligned}
$$

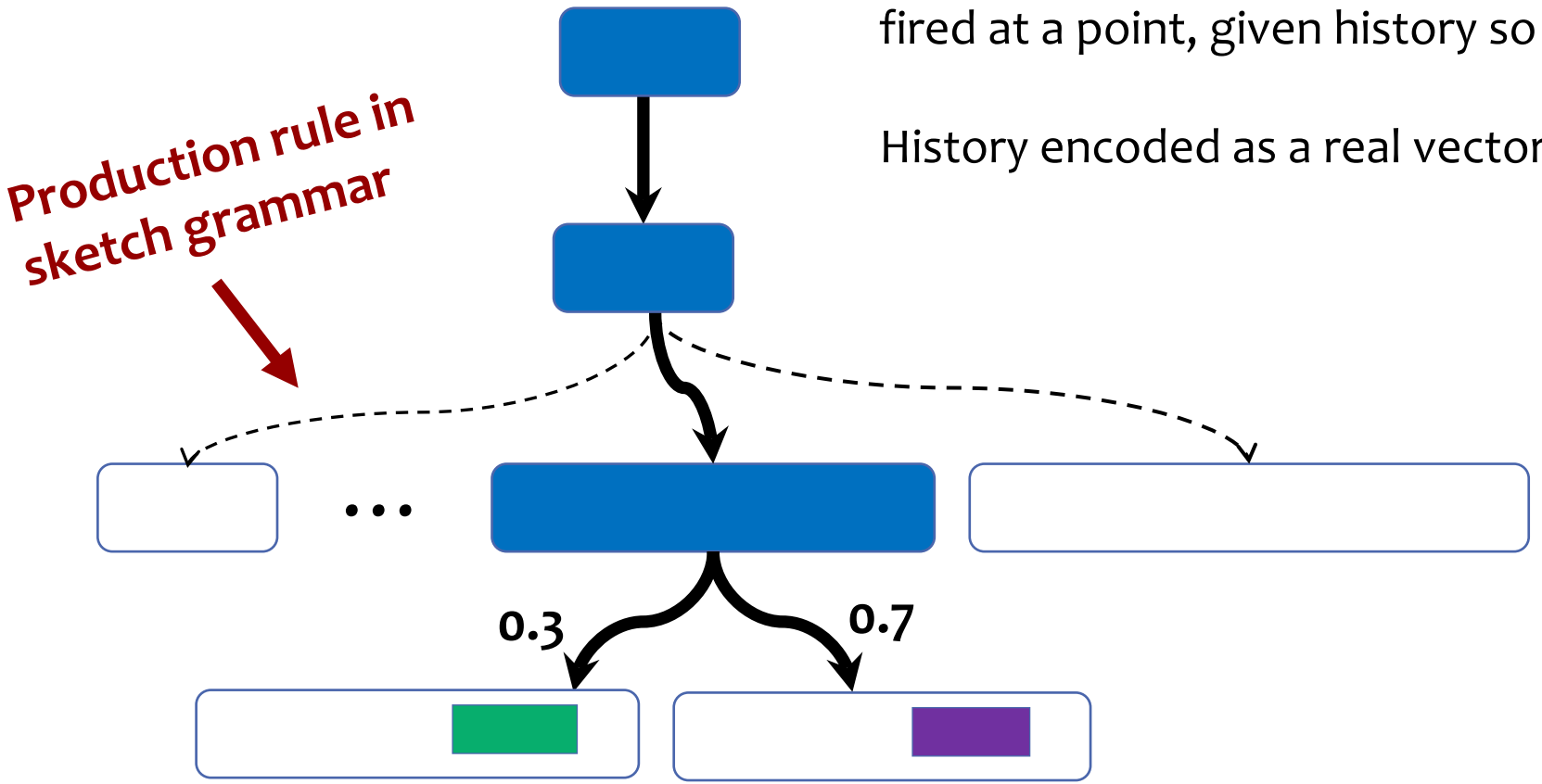
$$P(Z|X) = \text{Normal}\left(Z \mid \frac{\bar{\mathsf{X}}}{1+n}, \frac{1}{1+n}\mathbf{I}\right)$$

where

$$\bar{\mathsf{X}} = \left(\sigma_{Types}^{-2} \sum_j f(\mathsf{X}_{Types,j})\right) + \left(\sigma_{Calls}^{-2} \sum_j f(\mathsf{X}_{Calls,j})\right) + \left(\sigma_{Keys}^{-2} \sum_j f(\mathsf{X}_{Keys,j})\right)$$

$$n = n_{Types}\sigma_{Types}^{-2} + n_{Calls}\sigma_{Calls}^{-2} + n_{Keys}\sigma_{Keys}^{-2}$$

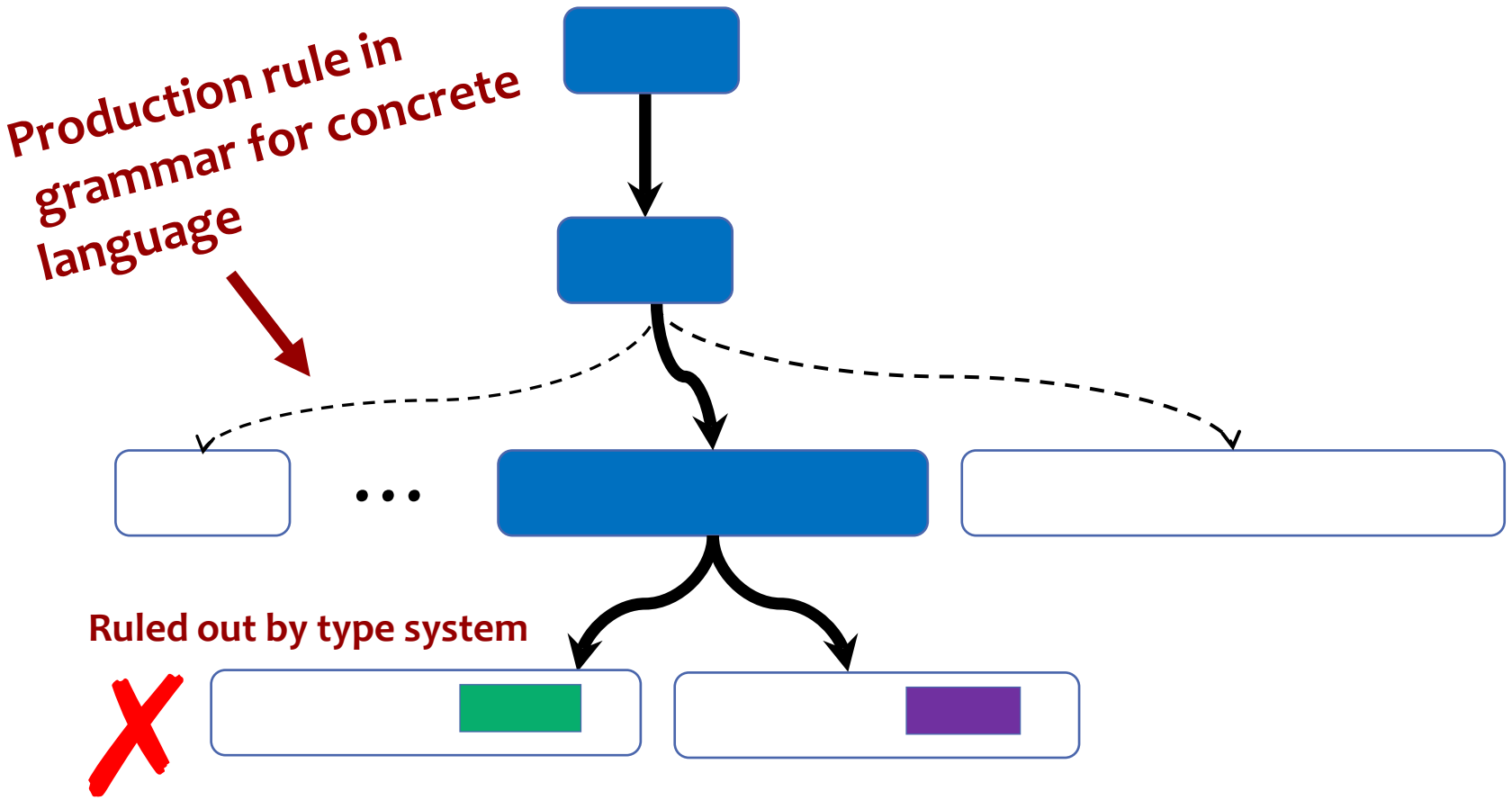
Neural decoder



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

History encoded as a real vector.

Concretization



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.

Implications

1. Uncertainty matters

Formal methods and programming systems research typically ignore uncertainty in intent and incompleteness of knowledge.

This is unfortunate, as programming is a human process.

2. Background knowledge matters

In formal methods and programming systems, one typically solves each problem from scratch.

We can do much better by considering the context in which program development happens.

Statistical learning techniques can help discover this background knowledge.

Deep API Learning. Gu, Zhang, Zhang, and Kim. FSE 2016.

Learning Programs from Noisy Data. Raychev, Bielik, Vechev, Krause. POPL 2016.

Code Completion from Statistical Language Models. Raychev, Vechev, Yahav. PLDI 2014.

Learning Natural Coding Conventions. Allamanis, Barr, Bird, and Sutton. FSE 2014.

Neurosymbolic Program Synthesis. Parisotto, Mohamed, Singh, Li, Zhou, and Kohli. ICLR 2017.

DeepCoder: Learning to Write Programs. Balog, Gaunt, Brockschmidt, Nowozin, Tarlow. ICLR 2017.

3. Human computation matters

Statistical learning depends on quality data. “Big Code” is a start, but by no means the end.

We could do much better with richer information about what programmers do, especially when interacting with a programming tool.

Human computation techniques, by now ubiquitous in data science, can help produce this sort of data.

4. Programming languages matter

Programs and proofs are different from images and natural language in that they have crisp requirements.

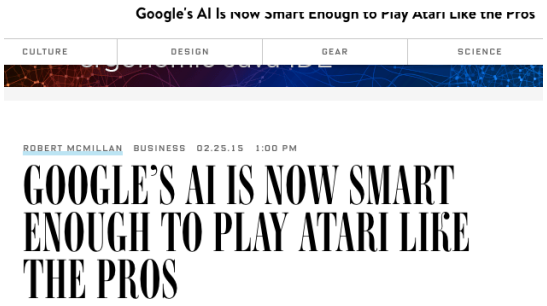
At best, statistical reasoning can get us “close” to proofs and programs. Combinatorial methods must do the rest.

Types and compositionality are critical to controlling the complexity of combinatorial reasoning.

Parting thought


CADE METZ BUSINESS 01.31.17 7:00 AM

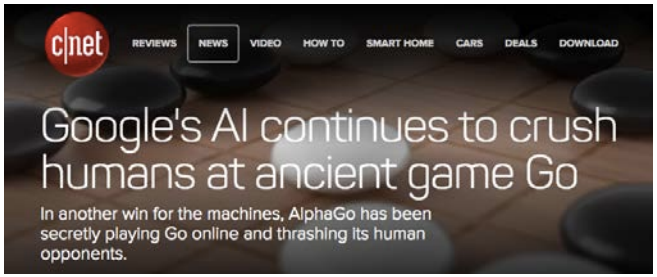
A MYSTERY AI JUST CRUSHED THE BEST HUMAN PLAYERS AT POKER


Google's AI Is Now Smart Enough to Play Atari Like the Pros

CULTURE DESIGN GEAR SCIENCE

ROBERT MCMILLAN BUSINESS 02.25.15 1:00 PM

GOOGLE'S AI IS NOW SMART ENOUGH TO PLAY ATARI LIKE THE PROS



Can we write a computer program that wins a programming contest?