HIGHER SCHOOL OF ECONOMICS

NATIONAL RESEARCH
UNIVERSITY

Doctoral School of Computer Science

LEARNING A SYNTHESIZER WITH “BIG CODE”

Dmitry Aldunin

Moscow, 2018

LEARNING A SYNTHESIZER WITH “BIG CODE”

Contents

1. Previously on the show
2. Architecture variants of tools learning tools from “Big Code”
3. Terminology
4. Inductive synthesis for empirical risk minimization
5. Deepsyn: learning statistical code completion systems

LEARNING A SYNTHESIZER WITH “BIG CODE”

Contents

1. **Previously on the show**
2. Architecture variants of tools learning tools from “Big Code”
3. Terminology
4. Inductive synthesis for empirical risk minimization
5. Deepsyn: learning statistical code completion systems

1.Previously on the show

Previously we:

- manually designed intermediate representations tailored to specific tasks and programming languages
- motivated the need to use conditional random fields and factor graphs when developing a name or type annotation predictors
- listed a number of features that relate program elements according to their distance in the program abstract syntax trees
- proposed a statistical n-gram language model that parametrizes a predicted method call on the $n - 1$ method calls preceding the predicted position
- defined program analysis to determine and extract those sequences using program analysis

LEARNING A SYNTHESIZER WITH "BIG CODE"

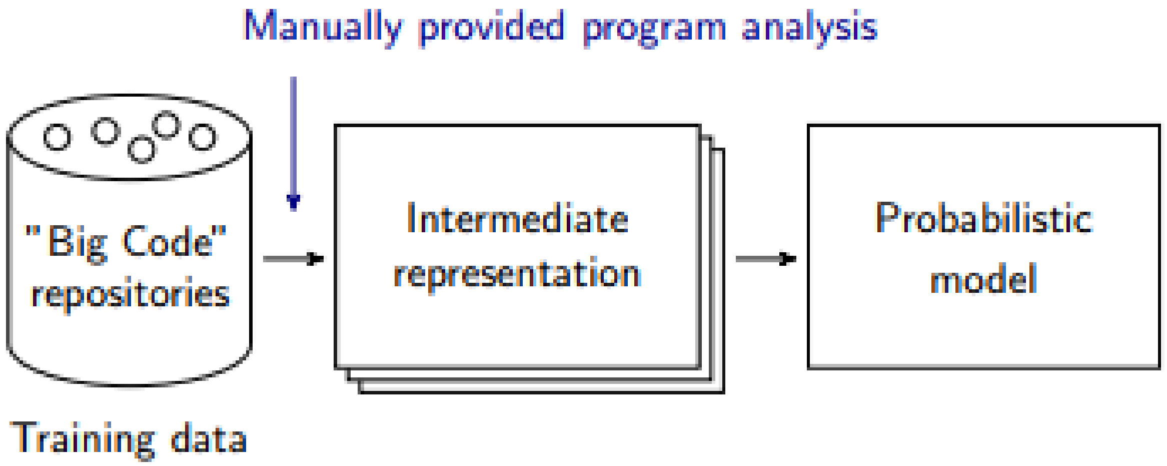
Contents

1. Previously on the show
2. **Architecture variants of tools learning tools from "Big Code"**
3. Terminology
4. Inductive synthesis for empirical risk minimization
5. Deepsyn: learning statistical code completion systems

ARCHITECTURE VARIANTS OF TOOLS LEARNING TOOLS FROM “BIG CODE”

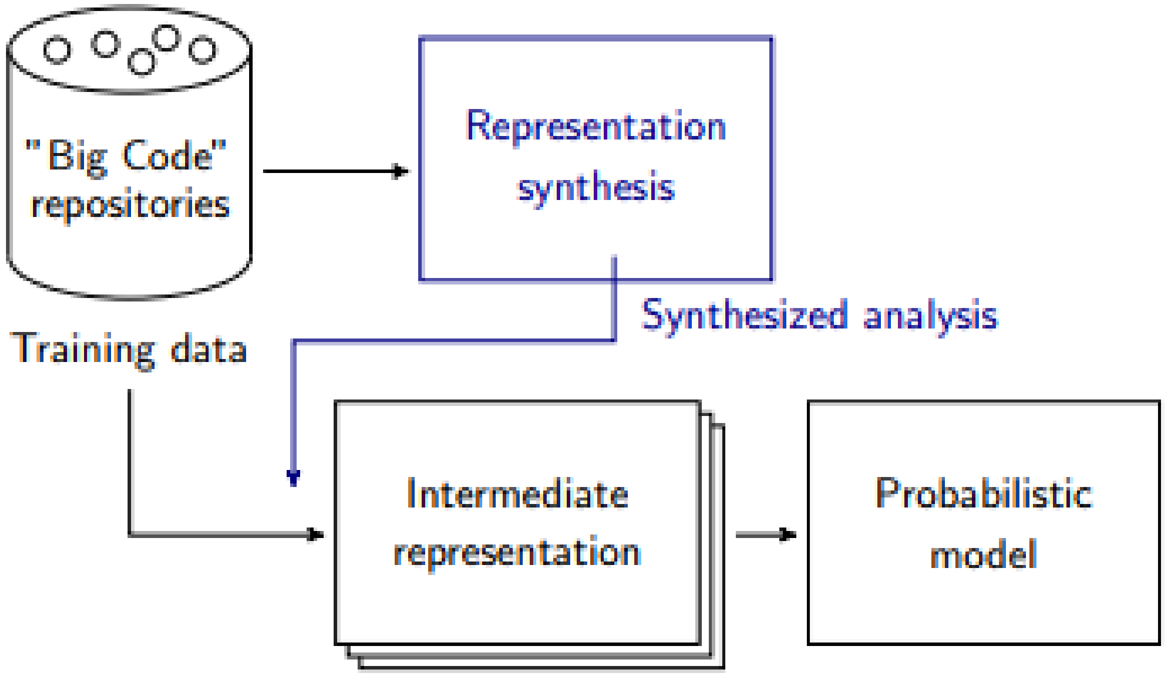
(a) Learning with a manually designed analysis

(a)


Manually provided program analysis
"Big Code" repositories
Training data
Intermediate representation
Probabilistic model

(b) Learning with synthesized analysis.

(b)



LEARNING A SYNTHESIZER WITH “BIG CODE”

Contents

1. Previously on the show
2. Architecture variants of tools learning tools from “Big Code”
3. **Terminology**
4. Inductive synthesis for empirical risk minimization
5. Deepsyn: learning statistical code completion systems

TERMINOLOGY FOR “BIG CODE” SYSTEMS BASED ON PROGRAM SYNTHESIS

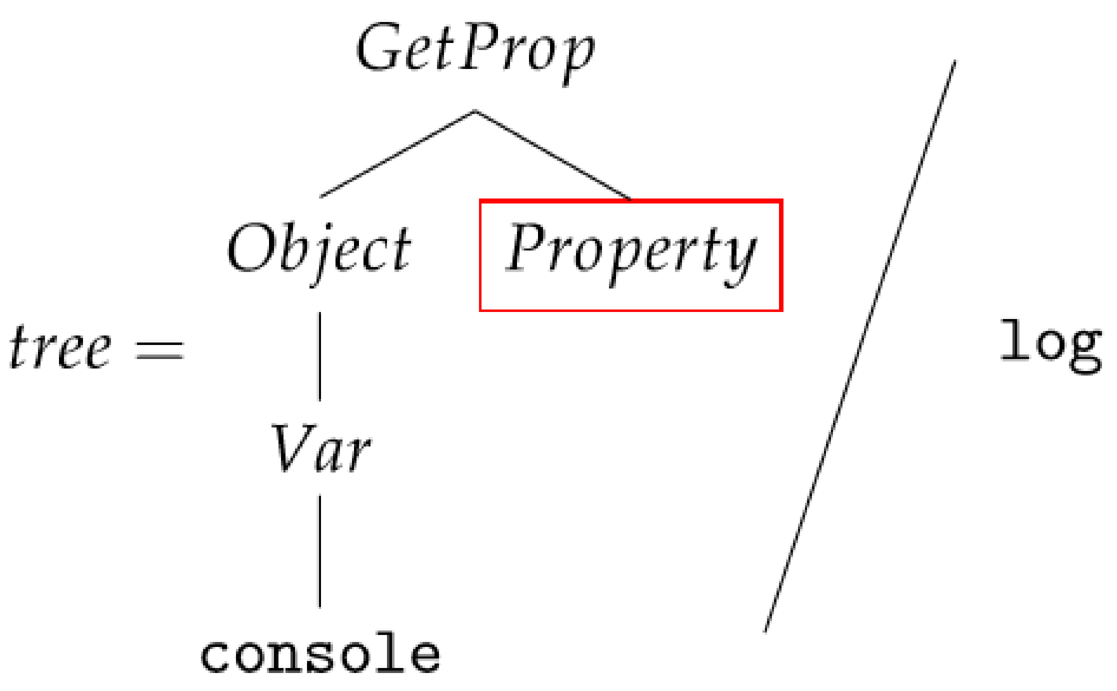
`console.`

(a) Code completion query

$p(tree)$

(c) Synthesized program p analyzes the tree $tree$

for example p_{SLANG} returns the last $n-1$ APIs on the `console` object.



(b) Input/output example in the training data

(d) Synthesis goal: find p such that for a probabilistic model m_p : $m_p(tree) =$ `log`.

LEARNING A SYNTHESIZER WITH “BIG CODE”

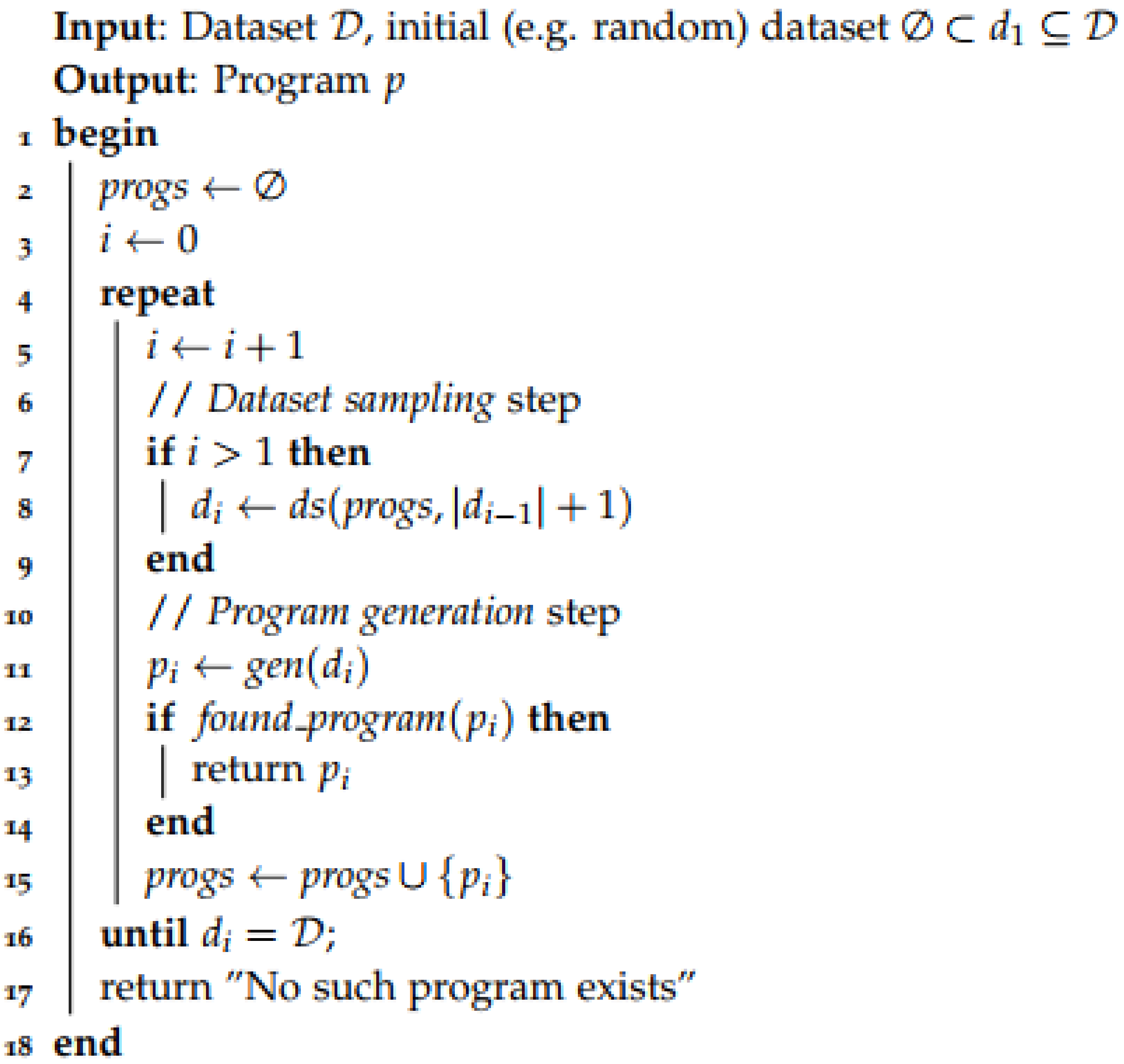
Contents

1. Previously on the show
2. Architecture variants of tools learning tools from “Big Code”
3. Terminology
4. **Inductive synthesis for empirical risk minimization**
5. Deepsyn: learning statistical code completion systems

INDUCTIVE SYNTHESIS FOR EMPIRICAL RISK MINIMIZATION

We connect the program generator and data sampler components in an iterative loop.

In this section, we show how to leverage Algorithm 2 from Chapter 4 to perform fast approximate empirical risk minimization.



```
Input: Dataset $\mathcal{D}$, initial (e.g. random) dataset $\emptyset \subset d_1 \subseteq \mathcal{D}$
Output: Program $p$
begin
    $progs \leftarrow \emptyset$
    $i \leftarrow 0$
    repeat
        $i \leftarrow i + 1$
        // Dataset sampling step
        if $i > 1$ then
            $d_i \leftarrow ds(progs, |d_{i-1}| + 1)$
        end
        // Program generation step
        $p_i \leftarrow gen(d_i)$
        if $found\_program(p_i)$ then
            return $p_i$
        end
        $progs \leftarrow progs \cup \{p_i\}$
    until $d_i = \mathcal{D}$;
    return "No such program exists"
end
```

Algorithm 2: Program Synthesis with Noise

EMPIRICAL RISK MINIMIZATION

Let $l : P \times X \rightarrow R_{\geq 0}$ be a function, such that l(p, x) quantifies the loss (amount of inaccuracy) when applying program p to example x.

Our task is to synthesize a program $p^{*} \in$ P that minimizes the expected loss on example x drawn i.i.d. (independent and identically-distributed) from distribution S.

I.e., we seek to minimize the risk (defined in terms of the expectation of the function):

$$R(p) = \mathbb{E}_{x \sim \mathcal{S}}[\ell(p, x)],$$

i.e. find the program:

$$p^{*} = \arg\min_{p \in \mathbb{P}} R(p)$$

EMPIRICAL RISK MINIMIZATION

There are two problems with computing p^*using the above approach:

- First, since S is unknown, the risk R(p) cannot even be evaluated.

- Second, even if we could evaluate it, finding the best program is generally intractable.

EMPIRICAL RISK MINIMIZATION

To address these concerns, we make two assumptions. First, we assume we are given a dataset D of examples drawn i.i.d. from S.
We can approximate the risk R(p) by the empirical risk, i.e.,

$$r_{emp}(\mathcal{D}, p) = \frac{1}{|\mathcal{D}|} \sum_{x \in d} \ell(\mathcal{D}, x)$$

Then, we assume (for now) that we have an “oracle”, an algorithm that can solve the *empirical risk minimization* (ERM) problem

$$p_{best} = \arg\min_{p \in \mathbb{P}} r_{emp}(\mathcal{D}, p).$$

GUARANTEES

For any ε, δ > 0, if our dataset of examples D is big enough with respect to the space of programs, then it holds for the solution p_{best} that R(p_{best}) ≤ R(p^*) + ε, with probability at least 1 − δ.

Hence, the best-performing program on the dataset is close (in risk) to the best program over all of S. This is because for all p ∈ P it holds that |R(p) − r_{emp} (D, p)| ≤ ε.

REGULARIZATION

ERM solution can overfit if the dataset D is not large enough. Overfitting means that R(p^*) ≪ R(p_{best}). As a remedy, a common approach is to apply regularization: i.e., instead of minimizing the empirical risk, one modifies the objective function by:

$$r_{reg}(\mathcal{D}, p) = r_{emp}(\mathcal{D}, p) + \lambda\Omega(p)$$

Hereby, Ω : P → R≥0 is a function (called regularizer), which prefers “simple” programs.

USING REPRESENTATIVE DATASET SAMPLER

The complexity of solving ERM (empirical risk minimization) is heavily dependent on the size of the dataset D. To enable ERM on the large dataset D, we use Algorithm 2 with a representative dataset sampler ds^R and a program generator that solves ERM on small datasets (sample subsets d_1, d_2, . . . d_m ⊆ D). Our goal upon termination of the synthesis procedure from Algorithm 2 is to obtain a program pm for which:

$$r_{emp}(\mathcal{D}, p_m) \in [r_{emp}(\mathcal{D}, p_{best}), r_{emp}(\mathcal{D}, p_{best}) + \varepsilon']$$

USING REPRESENTATIVE DATASET SAMPLER

$$r_{emp}(\mathcal{D}, p_m) \in [r_{emp}(\mathcal{D}, p_{best}), r_{emp}(\mathcal{D}, p_{best}) + \varepsilon']$$

Recall that R(pbest) ≤ R(p^*) + ε to obtain that the resulting solution pm will have risk at most ε + ε' more than p^*.

By exploiting the fact that we can solve ERM much faster on small datasets d_i, we can find such a solution much more efficiently than solving the ERM problem on the full dataset D.

This instantiation is a new approach of performing approximate ERM over discrete search spaces.

LEARNING A SYNTHESIZER WITH “BIG CODE”

Contents

1. Previously on the show
2. Architecture variants of tools learning tools from “Big Code”
3. Terminology
4. Inductive synthesis for empirical risk minimization
5. **Deepsyn: learning statistical code completion systems**

DEEPSYN: LEARNING STATISTICAL CODE COMPLETION SYSTEMS

In this section we present a new approach for constructing statistical code completion systems.

While not obvious, we show that the problem of synthesizing a program from noisy data appears in this setting as well, and thus the general framework of synthesis with noise (Chapter 4) applies here.

This means that the learned program does not predict its output directly from the input, but instead is used as part of a probabilistic model that performs the final prediction seen by the developer.

PRELIMINARIES

We begin with a standard definition of context-free grammars (CFGs), trees and parse trees.

Definition 5.1 (CFG). A context-free grammar (CFG) is the quadruple (N, Σ,s, R) where N is a set of non-terminal symbols, Σ is a set of terminal symbols, s ∈ N is a start symbol, R is a finite set of production rules of the form α → β_1... β_n with α ∈ N and β_i ∈ N ∪ Σ for i ∈ [1..n].
In the whole exposition, we will assume that we are given a fixed CFG:
G = (N, Σ,s, R).

PRELIMINARIES

Definition 5.2 (Tree). A tree T is a tuple (X, x_0, ξ) where X is a finite set of nodes, $x_0 \in X$ is the root node and $\xi : X \longrightarrow X^*$ is a function that given a node returns a list of its children. A tree is acyclic and connected: every node except the root appears exactly once in all the lists of children. This means that there is a path from the root to every node. Finally, no node has the root as a child.

PRELIMINARIES

Definition 5.3 (Partial parse tree). A partial parse tree is a triple (T, G, σ) where T = (X, x_0 , ξ) is a tree, G = (N, Σ,s, R) is a CFG, and σ : $X \longrightarrow$ Σ ∪ N attaches a terminal or non-terminal symbol to every node of the tree such that: if ξ(x) = x_{a1}... x_{an} (n > 1), then ∃(α ⟶ β_1... β_n) ∈ R with σ(x) = α and ∀i ∈ 1..n.σ(x_{ai}) = β_i.

PRELIMINARIES

Definition 5.4 (Tree completion query). A tree completion query is a triple (p_{tree}, x_{comp},*rules*) where p_{tree} = (T, G, σ) is a partial parse tree with T = (X, x_0, ξ), x_{comp} ∈ X is a node labeled with a non-terminal symbol (σ(x_{comp}) ∈ N) where a completion will be performed, and *rules* = {σ(x_{comp}) ⟶ β$_i$} n_i = 1 is the set of available rules that one can apply at the node x_{comp}.

PRELIMINARIES

Problem statement The code completion problem we are solving can now be stated as follows:

Given a tree completion query, select the most likely rule from the set of available rules and complete the partial parse tree with it.

EXAMPLE: FIELD/API COMPLETION

Consider the following partial JavaScript code ”console.” which the user is interested in completing. The goal of a completion system is to predict the API call log, which is probably the most likely one for console. Now consider a simplified CFG that can parse such programs (to avoid clutter, we only list the grammar rules):

GetProp $\longrightarrow$ Object Property
Object $\longrightarrow$ Var | GetProp
Var $\longrightarrow$ console | document | ... (other variables)
Property $\longrightarrow$ info | log | ... (other properties incl. APIs)

SECOND-ORDER LEARNING

The key idea of our solution is to synthesize a program which conditions the prediction.

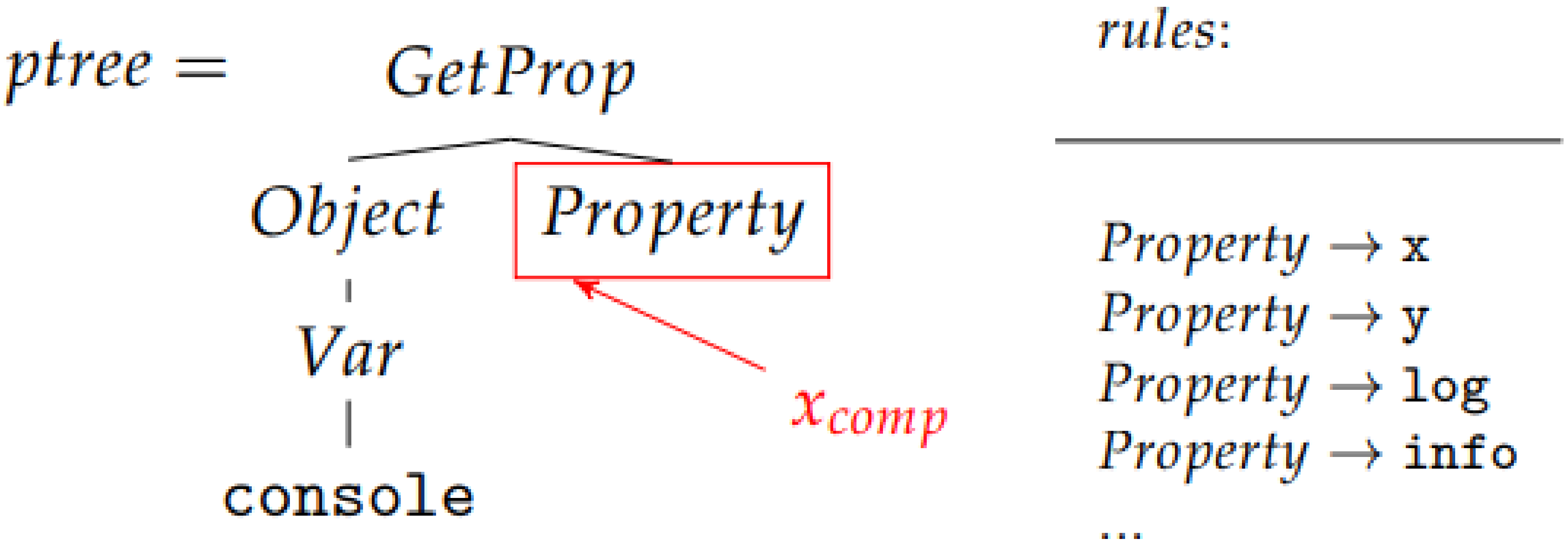


Figure 5.3: A tree completion query $(ptree, x_{comp}, rules)$ corresponding to completion for the code: "console".

SECOND-ORDER LEARNING

In our setting, a context c ∈ $Context$ is a sequence ranging over terminal and non-terminal symbols seen in the tree, as well as integers. That is, $Context = (N \cup \Sigma \cup \mathbb{N})^*$.

STEP 1: REPRESENTATION SYNTHESIS

The goal of the first step is to learn a conditioning program $p_{\approx best} \in$ P. In this step, we will apply the techniques for approximate empirical risk minimization. Let D = $\{X^i, Y^i\}_{i=1}^n$ be a training dataset of tree completions queries X^i = $(ptree,\ x_{comp}^i, rules)$ along with their corresponding completions $Y^i \in rules$. We assume that all examples in D solve the same task and thus they share the CFG production rules.

The goal of this step is to synthesize the (approximately) best conditioning program $p_{\approx best}\in$ PT × $X \rightarrow Context$ that given a query returns the context on which to condition the prediction. For instance, for the example in Fig. 5.3, a possible program p could produce p($ptree$, x_{comp}) = [console].

STEP 2: LEARN A PROBABILISTIC MODEL P(RULE | CTX)

Once the conditioning program $p_{\approx best}$ is learned, we use that program to train a probabilistic model. We next apply $p_{\approx best}$ to every query in the training data, obtaining a new data set: H(D, $p_{\approx best}$) = {($p_{\approx best}(Q_i)$, Y_i) | ((Q_i,$rules$), Y_i) ∈ D} where Q_i = ($ptree$, x^i_{comp}). The derived data set consists of a number of pairs where each pair {(c_i,r_i)} indicates that rule r_i is triggered by context c_i ∈ $Context$. Based on this derived set, we can now train a probabilistic model using MLE training (maximum likelihood estimation) which estimates the true probability P(r | c). The MLE estimation is standard and is computed as follows:

$$P^{H}_{MLE}(r \mid c) = \frac{|\{i \mid (c_i, r_i) \in H, c_i = c, r_i = r\}|}{|\{i \mid (c_i, r_i) \in H, c_i = c\}|}$$

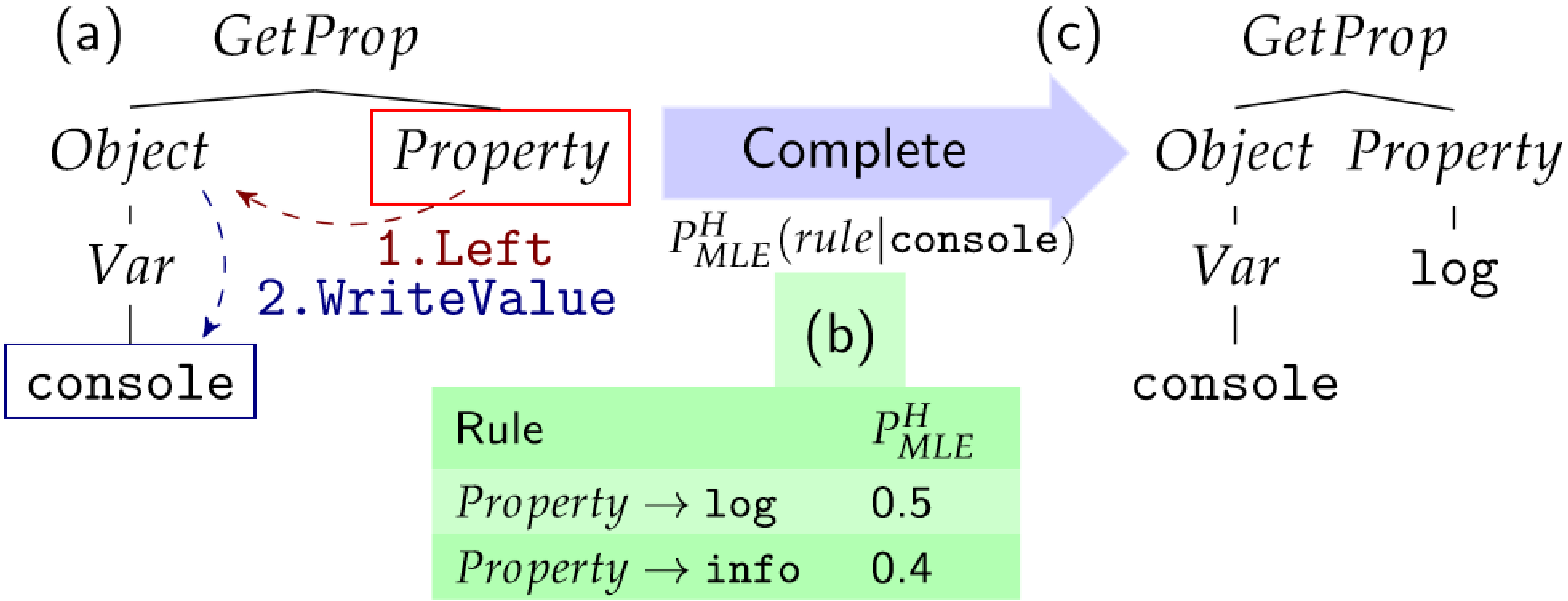
The MLE simply counts the number of times rule r appears in context c and divides it by the number of times context c appears.

STEP 3: PERFORM PREDICTIONS

Once we have learned the conditioning program $p_{\approx best}$ and the probabilistic model P(rule | ctx), we use both components to perform prediction. That is, given a query $(ptree,\ x_{comp}, rules)$, we first compute the context ctx = $p_{\approx best}(ptree,\ x_{comp})$. Once the context is obtained, we can use the trained probabilistic model to select the best completion (i.e., the most likely rule) from the set of available rules:

$$rule = \underset{r \in rules}{\arg\max}\ P^{H}_{MLE}(r \mid ctx)$$

STEP 3: PERFORM PREDICTIONS

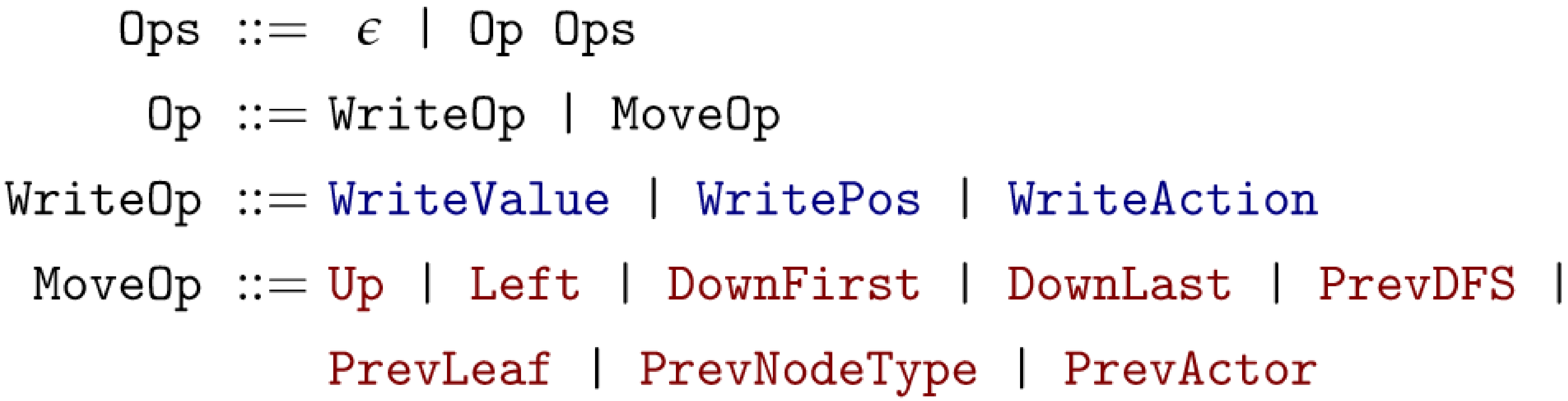


Rule	P^{H}_{MLE}
$Property \rightarrow$ log	0.5
$Property \rightarrow$ info	0.4

Figure 5.4: (a) TCond program p^a = Left WriteValue executed on a partial tree producing [console], (b) rules with their probabilities conditioned on [console], (c) the final completion.

TCOND: DOMAIN SPECIfiC LANGUAGE FOR TREE CONTEXTS

We now present a domain specific language, called TCond, for expressing the conditioning function p. The language is loop-free and is summarized in Fig. 5.5.



```
     Ops ::=  ϵ | Op Ops
      Op ::= WriteOp | MoveOp
 WriteOp ::= WriteValue | WritePos | WriteAction
  MoveOp ::= Up | Left | DownFirst | DownLast | PrevDFS |
             PrevLeaf | PrevNodeType | PrevActor
```

TCOND: DOMAIN SPECIfiC LANGUAGE FOR TREE CONTEXTS

Every statement of the language transforms a state $v \in PT \times X \times Context$. The state contains a partial tree, a position in the partial tree and the (currently) accumulated context.

The language has two types of instructions: movement (MoveOp) and write instructions (WriteOp). The program is executed until the last instruction and the accumulated context is returned as the result of the program.

TCOND: DOMAIN SPECIfiC LANGUAGE FOR TREE CONTEXTS

Move instructions change the node in a state as follows:

$$(ptree, node, ctx) \xrightarrow{\texttt{MoveOp}} (ptree, node', ctx)$$

TCOND: DOMAIN SPECIfiC LANGUAGE FOR TREE CONTEXTS

Write instructions update the context of a state as follows:

$$(ptree, node, ctx) \xrightarrow{\texttt{WriteOp}} (ptree, node, ctx \cdot x)$$

TCOND: DOMAIN SPECIfiC LANGUAGE FOR TREE CONTEXTS

The PrevActor and WriteAction instructionsuseasimplelightweight static analysis. If node denotes a memory location (field, local or global variable, that we call actor), PrevActor moves to the previous mention of the same memory location in the tree.

WriteAction writes the name of the operation performed on the object referred by node. In case the object referred by node is used for a field access, WriteAction will write the field name being read from the object. In case the object node is used with another operation (e.g., +), the operation will be recorded in the context.

For a program $p \in \mathtt{Ops}$, we write $p(ptree, x_{comp}) = ctx$ to denote that $(ptree, x_{comp}, \varepsilon) \xrightarrow{p} (ptree, node', ctx)$.

HIGHER · SCHOOL · OF · ECONOMICS ·

NATIONAL RESEARCH
UNIVERSITY

Thanks for your attention!