DISS. ETH NO. 23746

LEARNING FROM LARGE CODEBASES

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES OF ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

VESELIN RAYCHEV
Master in Informatics
Sofia University "St. Kliment Ohridski"
born on 23.04.1984
citizen of Bulgaria

accepted on the recommendation of

Prof. Dr. Martin Vechev
Prof. Dr. Eran Yahav
Prof. Dr. Armando Solar-Lezama
Prof. Dr. Charles Sutton

2016

ABSTRACT

As the size of publicly available codebases has grown dramatically in recent years, so has the interest in developing programming tools that solve software tasks by learning from these codebases. Yet, the problem of learning from programs has turned out to be harder than expected and thus, up to now, there has been little progress in terms of practical tools that benefit from the availability of these massive datasets.

This dissertation focuses on addressing this problem: we present new techniques that learn probabilistic models from large datasets of programs as well as new tools based on these probabilistic models which improve software development.

The thesis presents three new software systems (JSNice, Slang and DeepSyn) that learn from large datasets of programs and provide likely solutions to previously unsolved programming tasks including deobfuscation, static type prediction for dynamic languages, and code synthesis. All three of these systems were trained on thousands of open source projects and answer real-world queries in seconds and with high precision. One of these systems, JSNice, was publicly released and is already widely used in the JavaScript community.

An important ingredient of the thesis is leveraging static analysis techniques to extract semantic representations of programs and building powerful probabilistic models over these semantics (e.g., conditional random fields). Working at the semantic level also allows us to enforce important constraints on the predictions (e.g. typechecking). The net result is that our tools make predictions with better precision than approaches whose models are learned directly over program syntax.

Finally, the dissertation presents a new framework for addressing the problem of program synthesis with noise. Using this framework, we show how to construct programming-by-example (PBE) engines that handle incorrect examples, and introduce a new learning approach based on approximate empirical risk minimization. Based on the framework, we developed a new code synthesis system (DeepSyn) which generalizes prior work and provides state-of-the-art precision.

ZUSAMMENFASSUNG

So wie die Größe der öffentlich zugänglichen Codebasen in den letzten Jahren dramatisch zugenommen hat, so hat auch das Interesse an der Entwicklung von Programmier-Tools zugenommen, die Software-Probleme lösen indem sie von diesen Codebasen lernen. Doch das Problem des Lernens von Programmen hat sich als schwieriger als erwartet herausgestellt und bisher hat es wenig Fortschritt bei praktischen Tools gegeben, die von massiven Datenmengen profitieren. Die vorliegende Arbeit konzentriert sich auf die Lösung dieses Problems: Wir präsentieren neue Techniken, die Wahrscheinlichkeitsmodelle von großen Datensätzen von Programmen lernen, sowie neue Tools, die Software-Entwicklung verbessern.

Diese Doktorarbeit präsentiert drei neue Software-Systeme (JSNice, Slang und DeepSyn), die von großen Datenmengen von Programmen lernen und Lösungen für bisher ungelöste Programmierprobleme bieten, unter anderem Deobfuscation, statische Typinferenz für dynamische Sprachen und Programmsynthese. Alle drei dieser Systeme wurden mit Tausenden von Open-Source-Projekten trainiert und beantworten reale Abfragen in Sekunden und mit hoher Präzision. Eines dieser Systeme, JSNice wurde veröffentlicht und in großem Umfang in der JavaScript-Community verwendet.

Ein wichtiger Bestandteil der Arbeit ist die Verwendung von Techniken der statischen Analyse zur Extraktion semantischer Repräsentationen von Programmen und der Erzeugung von mächtigen Wahrscheinlichkeitsmodellen anhand dieser Semantiken (z.B. Conditional Random Fields). Auf semantischer Ebene zu arbeiten erlaubt es auch wichtige Einschränkungen auf die Vorhersagen zu erzwingen (z.B. Typkorrektheit). Das Endergebnis ist, dass unsere Tools Vorhesagen mit einer höheren Präzision machen als Ansätze, deren Modelle direkt von Programmsyntax lernen.

Schließlich stellt die Dissertation einen neuen Framework für die Behandlung des Problems der Programmsynthese mit Rauschen vor. Mit diesem Framework zeigen wir, wie man Programming-by-Example-Systeme (PBE) konstruiert, die falsche Beispiele verstehen. Wir führen einen neuen Lernansatz basierend auf approximierter empirischer Risikominimierung (ERM) ein. Basierend auf dem Framework haben wir ein neues Programmsynthesesystem (DeepSyn) entwickelt, welches vorherige Resultate verallgemeinert und Präzision auf dem Stand der Technik bietet.

PUBLICATIONS

This thesis is based on the following publications:

- Veselin Raychev, Martin Vechev, and Eran Yahav.
  "Code Completion with Statistical Language Models."
  ACM PLDI 2014. [110].
- Veselin Raychev, Martin Vechev, and Andreas Krause.
  "Predicting Program Properties from "Big Code"."
  ACM POPL 2015. [107].
- Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause.
  "Learning Programs from Noisy Data."
  ACM POPL 2016. [104].

The following publications were part of my PhD research and present results that are supplemental to this work or build upon results of this thesis:

- Pavol Bielik, Veselin Raychev, and Martin Vechev.
  "Programming with "Big Code":
  Lessons, Techniques and Applications."
  SNAPL 2015. [18].
- Pavol Bielik, Veselin Raychev, and Martin Vechev.
  "PHOG: Probabilistic Model for Code."
  ICML 2016 [20]
- Veselin Raychev, Pavol Bielik, and Martin Vechev.
  "Probabilistic Model for Code with Decision Trees."
  ACM OOPSLA 2016 [103]
- Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev.
  "Statistical Deobfuscation of Android Applications."
  ACM CCS 2016 [17]

The remaining publications were part of my PhD research, but are not covered in this thesis. The topics of these publications are outside of the scope of the material covered here.

- Veselin Raychev, Martin Vechev, and Eran Yahav.
  "Automatic Synthesis of Deterministic Concurrency."
  SAS 2013 [109].

- Veselin Raychev, Martin Vechev, and Manu Sridharan.
  "Effective Race Detection for Event-driven Programs."
  ACM OOPSLA 2013 [108].

- Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev.
  "Refactoring with Synthesis."
  ACM OOPSLA 2013 [106].

- Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and
  Eric Koskinen.
  "Commutativity Race Detection."
  ACM PLDI 2014 [37].

- Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev.
  "Phrase-Based Statistical Translation of Programming Languages."
  Onward! 2014 [70].

- Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz.
  "Parallelizing User-defined Aggregations Using Symbolic Execution."
  ACM SOSP 2015 [105].

- Pavol Bielik, Veselin Raychev, and Martin Vechev.
  "Scalable Race Detection for Android Applications."
  ACM OOPSLA 2015. [19].

- Casper S. Jensen, Anders Møller, Veselin Raychev,
  Dimitar Dimitrov, and Martin Vechev.
  "Stateless Model Checking of Event-driven Applications."
  ACM OOPSLA 2015. [66].

ACKNOWLEDGMENTS

First, I would like to thank my professor Martin Vechev for the continuous support during my PhD studies at ETH. The provided scientific freedom and guidance were instrumental to the success of this thesis. I would also like to express my gratitude to the reviewers: Armando Solar-Lezama, Eran Yahav and Charles Sutton, who provided valuable feedback which I incorporated in the final version of the dissertation. I would like to also thank Sasa Misailovic and Otmar Hilliges for their valuable suggestions.

I would like to acknowledge my co-workers and co-authors of papers published during my PhD. It was a great experience working with all of you: Anders Møller, Andreas Krause, Benjamin Bichsel, Casper S. Jensen, Christine Zeller, Dimitar Dimitrov, Eran Yahav, Eric Koskinen, Madanlal Musuvathi, Manu Sridharan, Max Schäfer, Pascal Roos, Pavol Bielik, Petar Tsankov, Svetoslav Karaivanov and Todd Mytkowicz. Thank you!

Many thanks to my Bulgarian friends in Switzerland who supported me in my pursuit for academic excellence. Without you guys, I would still be in the potato fields of the software industry, roaming for the clicks of yet another million users.

Special thanks to my mother, father and brother for their encouragement and love throughout these years. Last but not least, there is a special place in my heart and in my mind for Elena.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Learning from large datasets (also called "Big Data") using powerful probabilistic models has transformed a number of areas such as natural language processing [25, 74], computer vision [115], recommendation systems [59] and many others. Prominent examples include automatic machine translation systems such as Google Translate [48] that learn a probabilistic model of a natural language from existing documents and use that model to translate sentences from one natural language to another. Accurate face detection for photo sharing services such as Facebook [39] is another example of successful learning from a large dataset of images.

In the meantime, there has been significant progress in the area of programing languages driven by advances in type systems, constraint solving and other program analysis techniques. These advances have enabled a range of powerful specialized programming tools for bug finding [10, 108], model checking [43, 66], verification [44, 67, 116, 131], code completion [86, 99], program synthesis [51, 68, 81, 98, 120, 121, 122, 133] and others. However, these tools are nearing their inherent limits in terms of scalability meaning that a significant improvement is likely to come from a more disruptive change. For example, current synthesis approaches start from scratch considering each task in isolation [6, 52] and as a result, their focus is on discovering small (albeit tricky) programs. At the same time, significant advances in this area have the potential to transform software engineering as a whole [51].

Despite the overwhelming success of "Big Data" in a variety of application domains, up to now, learning from big datasets has not had tangible impact on programming tools. Meanwhile, the last few years have seen a dramatic increase in the amount of available source code typically found in public repositories such as GitHub [46], BitBucket [22] and others (termed "Big Code" by a recent initiative [34]).

The question then is: can one learn from "Big Code" in a way that the effort spent in designing existing programs is leveraged to help the development of new software? Addressing this question is challenging, because as opposed to other kinds of data, programs are data transformers with complex structure and semantics that should be captured in the learned model. Existing naïve approaches which treat programs as pure syntax [61] (and ignore semantics) are too imprecise leading to limited practical applicability. Thus, the main question we investigate in this thesis is:

How to leverage large datasets of code to build practical programming tools?

A NEW RESEARCH DIRECTION Leveraging "Big Code" to create programming tools is a new research direction we explore with two main goals in mind: usability of the resulting tools (i.e. scalability and precision) and generality of the underlying techniques upon which the tools are built. For example, as we will describe later in the thesis, our JavaScript deminification tool JSNice is used by thousands of people with the feedback being overwhelmingly positive. A search on Twitter[1] shows remarks such as:

"I've been looking for this for years" *"This is magic!"*
"Tell me how this works. Impressive!"

Further, we developed frameworks such as Nice2Predict[2] which enable a number of interesting new programming tools for deobfuscation, program understanding, type analysis and others. In this thesis, we cover a wide range of applications including learning program invariants and predicting the most likely snippets for code synthesis. Finally, we provide important guarantees (e.g. semantic equivalence or typechecking) for our predictions to ensure their correctness and connect core techniques from statistical learning (e.g. empirical risk minimization [84]) and program synthesis (e.g. counterexample guided inductive synthesis [121]), enabling new applications.

The thesis focuses on two core research challenges that affect the prediction accuracy of a "Big Code" system: creating a suitable probabilistic model for programs and learning from a large corpus of training data. We address the first challenge by carefully designing program analyses that capture semantic information and feed it into a probabilistic model. This is an essential design decision with significant impact on the overall system precision. Consider the following example demonstrating this point: Fig. 1.1 presents several configurations of our Slang code completion system (discussed in Chapter 3) trained

1 https://twitter.com/search?f=tweets&q=jsnice.org&src=typd
2 http://nice2predict.org/

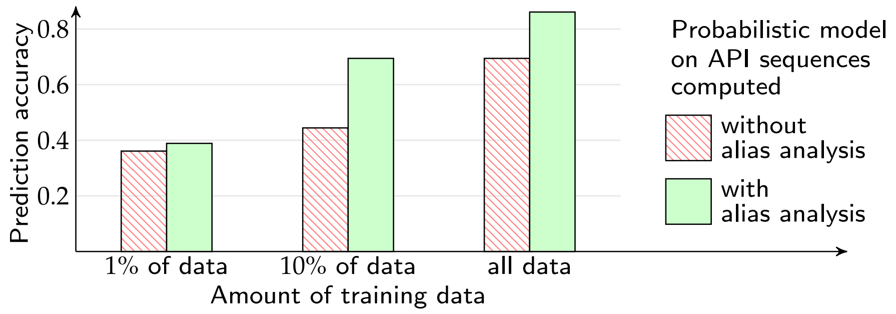


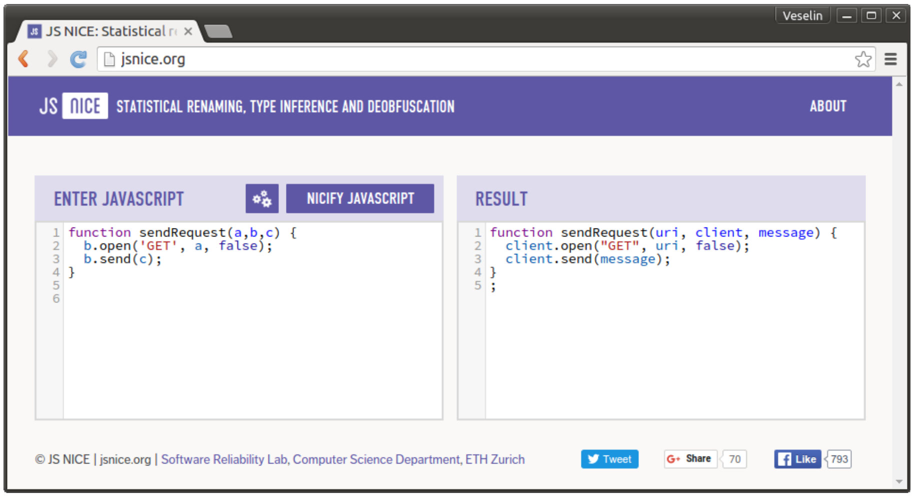
Figure 1.1: Impact of the amount of training data and probabilistic model on the accuracy of the SLANG code completion system.

on different amounts of data and using different program analysis abstractions. It is notable that using more semantic static analysis (i.e. a more precise abstraction with alias analysis) improves the accuracy of the predictions as much as an order of magnitude more data. This observation motivated us to formulate the problem of synthesizing the best (with respect to some user-defined metric) program analysis for a probabilistic model. We investigate this problem in Chapter 5.

Even with a well chosen probabilistic model, efficiently training that model on a large dataset is pivotal. Thus, we design learning procedures that take less than a day on datasets consisting of hundreds of thousands source code files. Importantly, user queries are also answered quickly. For example, a key factor for the success of JSNICE is that it can analyze thousands of lines of JavaScript code from real-world web applications (e.g. CNN, Facebook, Google Maps) in a few seconds. This was enabled by our scalable inference algorithms.

1.1 TOOLS SHOWCASE

Next, we give a brief informal overview of the probabilistic tools presented in this thesis. The goal is to provide an intuition for the capabilities of these tools from the perspective of their users and to discuss the challenges and choices when designing these systems. The general approach underlying these tools and formal definitions are provided later in the thesis.


JS NICE: Statistical r
jsnice.org
Veselin
JS NICE STATISTICAL RENAMING, TYPE INFERENCE AND DEOBFUSCATION
ABOUT
ENTER JAVASCRIPT
NICIFY JAVASCRIPT
RESULT
© JS NICE | jsnice.org | Software Reliability Lab, Computer Science Department, ETH Zurich
Tweet
Share 70
Like 793

```
function sendRequest(a,b,c) {
  b.open('GET', a, false);
  b.send(c);
}
```

```
function sendRequest(uri, client, message) {
  client.open("GET", uri, false);
  client.send(message);
}
;
```

Figure 1.2: A screenshot of the JSNice website. On the left side: minified code that is to be deobfuscated. On the right side: code with names inferred by the JSNice tool.

JSNICE Fig. 1.2 shows a screenshot of `http://jsnice.org/` which hosts the JSNice service. JSNice takes as input JavaScript code and renames its local variables and function parameters. It outputs JavaScript code that is semantically equivalent to the input, but is more human readable. In essence, JSNice reverses the obfuscation process done by JavaScript minifiers that rename all variables to shorter, but meaningless names. In the example in Fig. 1.2, the code on the left is minified with the parameter names set to `a`, `b` and `c`. Only the function name is not shortened to preserve the external interface of the function. The code on the right is produced by JSNice and includes meaningful identifier names that make the code significantly easier for a human to read and understand. The names suggested by JSNice were learned from a large corpus of JavaScript code that we collected and used as training data.

Several authors studied the importance of meaningful identifier names [26, 126] and some of those works [27, 64, 112] attempt to improve identifier names by enforcing coding conventions. These works, however, can only apply certain predefined fixes to the names and are not applicable to arbitrary name predictions like JSNice. A recent work [1] uses a probabilistic model to predict identifier names. However, it can only make predictions of one identifier name at a time and only in the context of other already present good identifier names

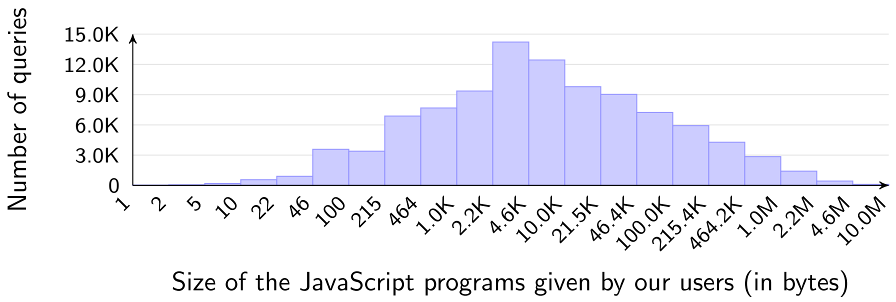


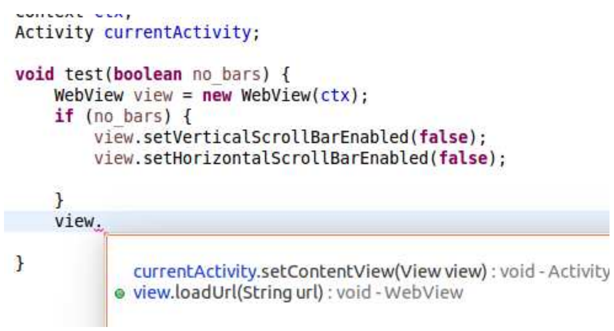
Figure 1.3: Histogram of query sizes to `http://jsnice.org/` sent by its users in the period May 10, 2015 – May 10, 2016.

and thus, it is not directly applicable to the deobfuscation problem considered by JSNice.

To construct JSNice, we framed the name prediction task as a structured prediction problem, developed a fast inference algorithm that assigns labels such that an optimization function is maximized and used an efficient learning algorithm that generalizes support vector machines to train a prediction model. The general approach and the resulting system are discussed in Chapter 2.

JSNice is widely used in the JavaScript community (it has users from every country in the world). In a period of a year, our users have deobfuscated over 9 gigabytes of unique (non-duplicate) JavaScript programs. Fig. 1.3 shows a histogram of the size of these programs, showing that users often query it with large code fragments, with average size of 91.7 kilobytes.

JSNice is not only applicable to name inference, but also to other settings. For instance, JSNice automatically predicts optional type annotations in JavaScript code. Several JavaScript extensions (e.g. TypeScript [130] and Google Closure Compiler [47]) add optional type annotations to program variables using a gradual type system. These extensions help discover type errors and improve code documentation. Despite their advantages, all of these techniques require manual effort to provide the initial type annotations. In this thesis, we developed a type prediction engine for a portion of the type system in the Google Closure Compiler. This engine learns type annotation rules from existing code and uses these rules to annotate new, unseen code. Overall, JSNice generates annotations that typecheck and often agree



```
Activity currentActivity;

void test(boolean no_bars) {
    WebView view = new WebView(ctx);
    if (no_bars) {
        view.setVerticalScrollBarEnabled(false);
        view.setHorizontalScrollBarEnabled(false);

    }
    view.

}
          currentActivity.setContentView(View view) : void - Activity
        ● view.loadUrl(String url) : void - WebView
```

Figure 1.4: A code completion plugin capable to predict multiple statements at once.

with annotations manually provided by developers. As a result, our type prediction engine may help adoption of gradual type systems.

slang The thesis also explores the problem where the number of predicted program elements is not known in advance – a setting not supported by the JSNice model. We handle this new setting with different statistical models and learning techniques. Based on these models, we propose an API code completion system called Slang, capable of predicting one or multiple method invocations at once. Fig. 1.4 shows an experimental user interface of such a system for Eclipse. In this case, the system suggests a code snippet consisting of multiple Android API invocations that complete the given program so to display a webpage. We present the techniques that enable such multi-API code completion in Chapter 3. In fact, our approach is applicable beyond the case of API "dot" code completion and allows completions of a partial program with multiple APIs at different program locations simultaneously. This extended setting is similar to the one suggested in Sketch synthesis [120], but instead of user-provided specifications, the completions are guided by a probability distribution. Finally, Slang does not return only the best ranked solution, but can efficiently compute and return multiple completions sorted according to their probabilities.

The effectiveness of Slang comes from the insight that predicting an API call on an object can be effectively done if conditioned on the APIs previously called on the same object. To build a model that captures this insight, we propose a scalable static analysis that combines Steensgaard style alias analysis [123] and typestate analysis [41] in order to perform this conditioning. Once sequences of API invocations are obtained for all the programs in the training data, a probabilistic model predicts the probability of an API invocation x based on previous API invocations $x_1 \cdots x_{n-1}$. Here, we leverage state-of-the-art probabilistic models from natural language processing such as n-gram models with smooting [114] and recent advances in recurrent neural networks [89].

In contrast to the traditional use of static analysis for verification and bug finding, the static analysis we developed to extract the sequences upon which the probabilistic model is built, need not be sound. The reason is that many probabilistic models provide smooth probability estimates (i.e. assign non-zero probabilities to any values outside of the training data) and thus, strict constraints (e.g. sound typechecking) are enforced separately. That interaction between the static analysis and the probabilistic model raises several key research questions:

1. What is the best program analysis for making predictions about programs?

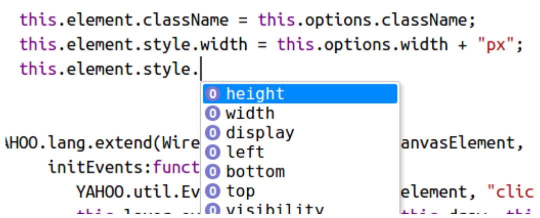
2. How will the analysis look like in programming languages which lack static type information (e.g. JavaScript)?

3. Can we use similar techniques to perform completion of other program elements that are not API calls?

DEEPSYN To answer these questions, in Chapter 5 we developed a systematic, general technique that automatically synthesizes code completion systems described by a domain specific language (DSL).

This technique effectively searches and evaluates thousands of possible completion systems and returns the most precise one according to an empirical risk metric. We applied this idea in the context of JavaScript where it is notoriously hard to perform static analysis due to lack of static type information. Despite this difficulty, our algorithm synthesized a code completion system for JavaScript that predicts around half of JavaScript API calls and almost 40% of field accesses correctly. Fig. 1.5 shows a possible use-case of such a system.

In addition to providing several programming tools, this thesis also discusses a number of new research directions such as extending the ideas to new programming languages (e.g. functional), applying our techniques to new areas such as security (e.g. deobfuscating Android application using techniques from JSNice), integrating statistical syn-



```
this.element.className = this.options.className;
this.element.style.width = this.options.width + "px";
this.element.style.

\HOO.lang.extend(Wire                    anvasElement,
    initEvents:funct
        YAHOO.util.Ev                    element, "clic
```

Figure 1.5: The DeepSyn tool for completing JavaScript code. In this snippet of code, standard type analysis cannot resolve the types of the used variables. Our statistical model predicts that if `width` was set, `height` may need to be set as well.

thesis with traditional synthesizers (combining probabilities with constraints), as well as making connections between traditional program synthesis, program analysis and statistical learning.

1.2 ARCHITECTURE OF STATISTICAL PROGRAMMING TOOLS

In Fig. 1.6, we show the general architecture of a statistical tool which learns from programs, consisting of two phases: a training phase and a query phase. In the training phase, for every program in "Big Code", program analysis is applied to convert it into a specialized intermediate representation. That representation aims to capture an abstraction of the training data suitable for the task solved by the particular tool. For example, a code completion tool which tries to complete a program using date and time APIs, should be able to learn from programs that use the same API (e.g. calendar applications) even if these programs are not identical to the query program or use the API in different contexts. The intermediate representation in this case would capture rich sequences of APIs from the training data. Once extracted, these intermediate representations are used to train a probabilistic model. We advocate that statistical tools which learn from programs should use intermediate representations and probabilistic models that are able to generalize beyond what is seen in the training data. That is, the predictions made by these tools need not be present at training time.

At query time, the user provides an application-specific input. For example, in code completion, the input is typically a partial program and a position in the program to be completed. This user-provided input is then processed with program analysis and again converted

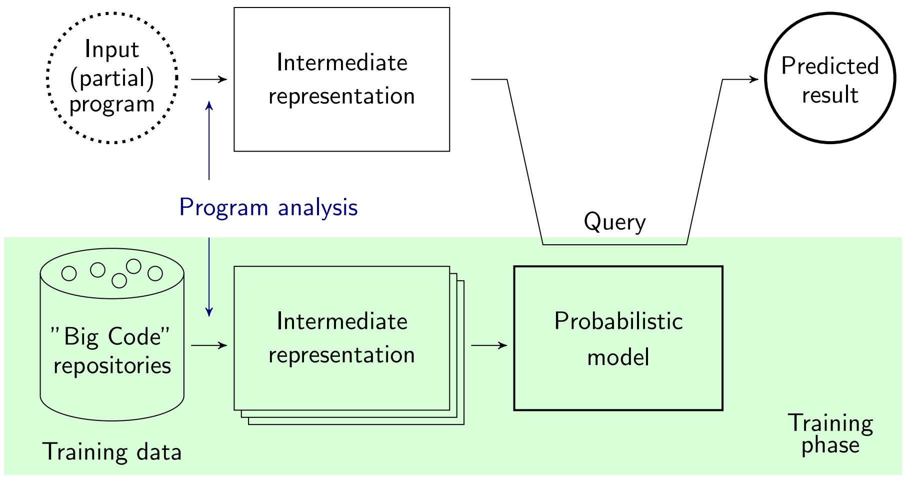


Figure 1.6: General architecture of "Big Code" tools.

into an intermediate representation suitable to query the probabilistic model. Finally, the probabilistic model returns the most likely output to the user – e.g. a synthesized program.

1.3 PROBLEM DIMENSIONS

In Fig. 1.7 (left column) we present several dimensions that a statistical tool needs to consider. In this thesis, we consider several different instantiations of these dimensions summarized in the right column of the figure. For instance, we consider the applications of JavaScript deobfuscation, property prediction and code synthesis. Although there are multiple dimensions in the design space of "Big Code" systems, some of the dimensions are correlated.

First, the intermediate representations must match the corresponding probabilistic models. For example, sequences are indexed in statistical language models (this model comes from natural language processing and its goal is to estimate probabilities of sequences of words). Conditional Random Fields (CRF) [79] and Structured Support Vector Machine (SSVM) [127, 129] models match with the intermediate representation of a factor graph.

Second, the corresponding program analysis must produce an intermediate representation that matches the particular application. For example, when renaming variables for deobfuscation, all references to

Dimensions	Instantiations in this thesis
Application	Deobfuscation (§2) Type Prediction (§2) Code Synthesis (§3, §5)
Program analysis	Scope Analysis, Type Analysis (§2) TypeState and Alias Analysis (§3) Domain Specific Languages (§5)
Intermediate representation	Factor Graphs (§2) Sequences (§3, §5)
Probabilistic model	CRF, Structured SVM (§2) Statistical Language Models (§3, §5)
Query	MAP Inference (§2, §3)

Figure 1.7: Dimensions and instantiations of statistical programming tools. that learn from "Big Code"

a local variable must be renamed to the same name in order to preserve the program semantics and thus scope analysis of local variables is used. Still, when mapping an application to an intermediate representation, there are multiple possible viable choices of program analysis and in Chapter 5, we present an approach that searches a space of such analyses defined by a domain specific language.

Finally, for our applications we consider *MAP inference* (Maximum APosteriori) queries. MAP inference means predicting multiple values at once such that the score of the whole assignment according to the probabilistic model is maximized. For example, when performing type predictions, all types of a program are predicted simultaneously enabling the inference to typecheck the predictions. To the best of our knowledge, this work is the first to use MAP inference in the context of programming problems. For two of our applications (JSNice, Slang), we include efficient procedures for MAP inference, while for DeepSyn we do not discuss it, even though MAP inference is also applicable in that setting.

1.4 challenges

In designing the systems presented in this thesis, we had to address a number of challenges.

vast number of labels For our applications, the number of labels is typically very large. Thus, trying every possible label at query time is practically infeasible. For example, when predicting identifier names, the range of labels are all possible identifier names – hundreds of thousands of names in our training data. A similar issue arises in API code completion with tens of thousands of possible APIs.

dependent predictions The problem with a large number of labels is further exacerbated by the fact that multiple predictions are performed simultaneously, leading to a combinatorial explosion. For example, to predict the names of n variables, we need to consider up to n^k possible name assignments if each variable name ranges over a set of size k. We address this problem by designing an approximate greedy algorithm that is fast and precise for our applications. Similarly, we designed an algorithm for predicting sequences of multiple API calls subject to additional constraints.

estimating probabilities and learning Some machine learning techniques compute predictions and return the associated probability with each prediction. Valid probabilities, however, need to be positive and sum to one over all possible outcomes. With the large number of predictions and specifically the presence of constrains (e.g. non-duplicate names, type checking rules, API restrictions), it becomes intractable to even count the number of possible predictions (i.e. possible outcomes). For some probabilistic models such an intractable computation appears in computing the so called *partition function* [93] that makes probabilities sum to one. In our approach, we avoid computing expensive partition functions by carefully selecting our models and training algorithms. One such model is the structured support vector machine (SSVM) trained using the MAP inference procedure [102]. This model is *discriminative*, which means that it models the conditional probabilities of one set of properties given another set of properties. For example, in name prediction, we learn the names of local variables given other values such as field names, strings and other constants.

FEATURE ENGINEERING Finally, many machine learning models can be expressed in terms of a linear combination of feature functions. These feature functions are typically defined by experts that design the specific tool. In our JSNice and Slang tools we manually define feature functions that parametrize the probabilistic model. This effort, however, does not scale as experts are needed to fine-tune every specific tool variant. We address this challenge with a general program synthesis technique over a domain-specific language. The synthesis technique behind our DeepSyn system replaces the feature engineering process and even leads to models that exceed the precision of current models manually tailored by experts.

1.5 RELATED WORK

The developments presented in this thesis have already led to various subsequent works by other groups [23, 50, 71, 83, 97]. Here we list some of these works, other relevant works are mentioned within the appropriate chapter.

A recent work of Katz et al. [71] applies a language model similar to the one we introduce in Chapter 3, to reverse engineer binaries where types of registers are estimated. The works of Oh et al. [97] and Grigore and Yang [50] use learned models for abstraction refinement in order to speed-up a program verifier. Another recent work applies a model similar to the one we use in JSNice to triage false positives in static analysis [87]. The work of Long and Rinard [83] uses probabilistic models to sometimes improve the number of correct patches over their prior patch generation system. However, most of these works are less scalable, train on orders of magnitude smaller datasets than ours, learn only a small number of parameters and as a result provide relatively small improvements over non-machine learning approaches.

MACHINE LEARNING WITHIN A SINGLE PROGRAM Several recent works use machine learning techniques such as support vector machines [116] or decision trees [44] to discover inductive invariants for program verification. These works are orthogonal to the direction explored in this thesis and have a different focus: they use machine learning only within a program and do not "transfer" learned facts from one program to another.

1.6 THESIS CONTRIBUTIONS

We next summarize the main contributions of this thesis:

- In Chapter 2 we describe a new approach for probabilistic prediction of program properties and the JSNICE system based on this approach. That chapter is the first to connect conditional random fields [79] to the problem of learning from programs.
- In Chapter 3 we connect static analysis with statistical language models and present the SLANG API code completion tool based on this approach. That chapter also illustrates how to apply recent advances in deep learning (recurrent neural networks [89]) to the problem of learning from programs.
- In Chapter 4, we develop a new framework for program synthesis with noise, connecting traditional program synthesis with statistical learning. This framework: (i) enables existing programming-by-example engines (PBE) to deal with noise, (ii) provides a fast procedure for approximate empirical risk minimization in machine learning, and (iii) serves as a basis for developing learning procedures from "Big Code" with high precision.
- In Chapter 5, we introduce a new learning approach based on the ideas from Chapter 4, generalizing many prior works [61, 94, 95, 110]. We implemented this approach in the DEEPSYN system and showed that its precision significantly improves over existing approaches.

We believe this thesis provides an in-depth investigation of constructing programming tools based on probabilistic models learned from a large set of programs, and opens several promising directions for future research, both practical and theoretical. We discuss some of these in Chapter 6.

2 DISCRIMINATIVE MODELS FOR PREDICTING PROGRAM PROPERTIES

In this chapter we focus on the problem of inferring program properties. We introduce a novel statistical approach which predicts properties of a given program by learning a probabilistic model from existing codebases already annotated with such properties. We use the term program property to encompass both: classic semantic properties of programs (*e.g.* type annotations) as well as syntactic program elements (*e.g.* identifiers or code).

The core technical insight is transforming the input program into a representation that enables us to formulate the problem of inferring program properties (be it semantic or syntactic) as structured prediction with conditional random fields (CRFs) [79], a powerful undirected graphical model successfully used in a wide variety of applications including computer vision, information retrieval, and natural language processing [16, 60, 79, 100, 101].

To our knowledge, this is the first work which shows how to leverage CRFs in the context of programs. This work also presents a class of relevant feature functions that allow for efficient MAP inference in a graphical model. By connecting programs to CRFs, we can reuse state-of-the-art learning algorithms [75] to the domain of programs. Another benefit of this encoding is that we can make multiple predictions at once – a desired feature of the predictions about program elements when there are multiple possibly dependent ones.

Fig. 2.1 illustrates our two-phase structured prediction approach. In the prediction phase (upper part of figure), we are given an input program for which we are to infer properties of interest. In the next step, we convert the program into a representation which we call a dependency network. The essence of the dependency network is to capture relationships between program elements whose properties are to be predicted with elements whose properties are known. Once the network is obtained, we perform structured prediction and in particular, a query referred to as *Maximum a Posteriori (MAP)* inference [75]. This query makes a *joint* prediction for all elements together by optimiz-

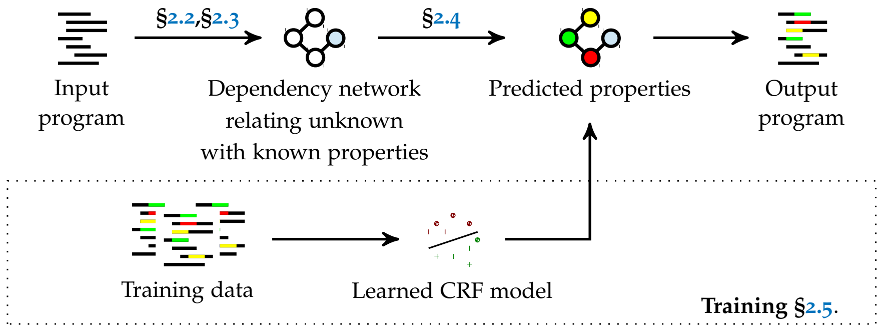


Figure 2.1: Statistical Prediction of Program Properties.

ing a scoring function based on the learned CRF model. Making a joint prediction which takes into account structure and dependence is particularly important as properties of different elements are often related. A useful analogy is the ability to make joint predictions in image processing where the prediction of a pixel label is influenced by the predictions of neighboring pixels. To achieve good performance for the MAP inference, we developed a new algorithmic variant which targets the domain of programs (existing inference algorithms cannot efficiently deal with the combination of unrestricted network structure and massive number of possible predictions per element). Finally, we output a program where the newly predicted properties are incorporated. In the training phase (lower part of Fig. 2.1), we find a good scoring function by learning a CRF model from a large training set of programs. Here, because we deal with CRFs and MAP inference queries, we are able to leverage state-of-the-art max-margin CRF training methods [127].

TRAINING DATA An important requirement of our statistical approach is relevant training data. In our case, the data consists of *programs* with their corresponding *program properties* that were already computed or manually annotated. The amount of data should be large enough to enable learning of a sophisticated model that makes interesting predictions. For our applications, such a training corpus is obtained by downloading a large number of code repositories from GitHub [46].

APPLICATIONS This work focuses on applications in the context of Software Engineering, Program Analysis and Security. In one of our applications we perform identifier renaming, which is a form of semantic preserving transformation (refactoring) that improves the code readability. In another application, we propose to add type annotations to code. This is, we learn a program analyzer based on a probabilistic model. Finally, inferring name and type information for programs is a form of decompilation or deobfuscation which is an interesting problem in the context of security.

To build these individual applications, we first phrase them in terms of CRF models and then apply the techniques presented in this chapter. Phrasing a problem into a CRF involves defining feature functions for each individual problem as we show in Section 2.2.2. To facilitate faster creation of new applications, we open sourced a framework called Nice2Predict [1], which includes all the components discussed in this chapter except defining the feature functions. Nice2Predict enables applications such as code deobfuscation (an example is Android deobfuscation [17] where all method, class and field names are changed by ProGuard [118]) or learning other types of program analyzers beyond type annotations. Nice2Predict can be directly applied to tasks where the set of possible predicted labels is finite, e.g. for the applications in this chapter, we predict from a finite set of names and types.

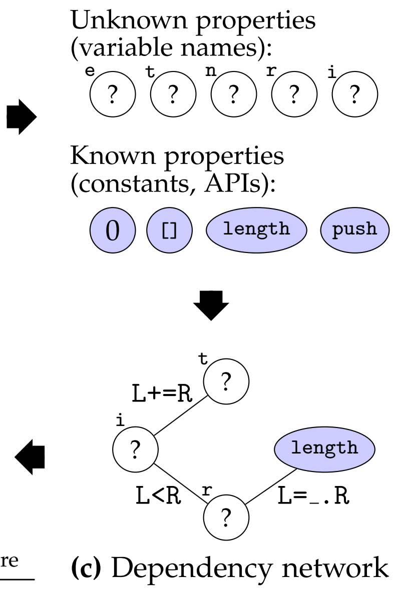
JSNICE: NAME AND TYPE INFERENCE FOR JAVASCRIPT As an example of our approach, we built a system called JSNice which addresses two important challenges in JavaScript: predicting (syntactic) identifier names and predicting (semantic) type annotations of variables. We focused on JavaScript for three reasons. First, in terms of type inference, recent years have seen extensions of JavaScript that add type annotations such a Google Closure Compiler [47] and TypeScript [130]. However, these extensions rely on traditional type inference which does not scale to realistic programs that make use of dynamic evaluation and complex libraries (e.g. jQuery) [67]. Our work predicts likely type annotations for real world programs which can then be provided to the programmer or to a standard type checker. Second, much of JavaScript code found on the Web is obfuscated, making it difficult to understand what the code is doing. Our approach recovers likely identifier names thereby making the code readable again. Finally, JavaScript programs are readily available in source code repositories (e.g. GitHub) meaning that we can obtain a large set of high quality training programs.

1 https://github.com/eth-srl/Nice2Predict

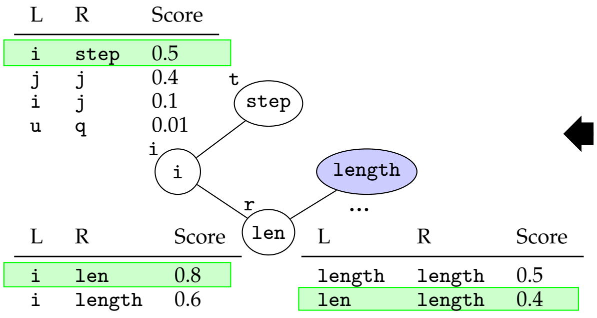
(a) JavaScript program with minified identifier names

```
function chunkData(e, t) {
  var n = [];
  var r = e.length;
  var i = 0;
  for (; i < r; i += t) {
    if (i + t < r) {
      n.push(e.substring(i, i + t));
    } else {
      n.push(e.substring(i, r));
    }
  }
  return n;
}
```

(b) Known and unknown properties



(c) Dependency network



L	R	Score
i	step	0.5
j	j	0.4
i	j	0.1
u	q	0.01

L	R	Score
i	len	0.8
i	length	0.6

L	R	Score
length	length	0.5
len	length	0.4

(d) Result of MAP inference

(e) JavaScript program with new identifier names (and type annotations)

```
/* str: string, step: number, return: Array */
function chunkData(str, step) {
  var colNames = [];
/* colNames: Array */
  var len = str.length;
  var i = 0;  /* i: number */
  for (;i < len;i += step) {
    if (i + step < len) {
      colNames.push(str.substring(i, i + step));
    } else {
      colNames.push(str.substring(i, len));
    }
  }
  return colNames;
}
```

Figure 2.2: A JavaScript program with new names and type annotations, along with an overview of the name inference procedure.

The JSNice service is publicly available at http://jsnice.org and was used by more than 100, 000 JavaScript developers from every country worldwide. As of writing this thesis, JSNice also has thousands of active users and according to the statistics in Google Analytics [2], the ratio of returning users increases over time.

2.1 OVERVIEW

In this section we provide an informal description of our statistical inference approach on a running example. Consider the JavaScript program shown in Fig. 2.2(a). This is a program which has short, non-descriptive identifier names. Such names can be produced by both a novice inexperienced programmer or by an automated process known as minification (a form of obfuscation) which replaces identifier names with shorter names. In the case of client-side JavaScript, minification is a common process on the Web and is used to reduce the size of the code being transferred over the network and/or to prevent users from understanding what the program is actually doing. In addition to obscure names, variables in this program also lack annotated type information. The net effect is that it is difficult to understand what the program actually does, which is that it partitions an input string into chunks of given sizes and stores those chunks into consecutive entries of an array.

Given the program in Fig. 2.2(a), our system produces the program in Fig. 2.2(e). The output program has new identifier names and is annotated with predicted types for the parameters, the local variables and the return statement. Overall, it is easier to understand what that program does when compared to the input program. Next, we provide a step by step overview of the prediction procedure that performs that program transformation. We focus on predicting names (reversing minification), but the process for predicting types is identical.

STEP 1: DETERMINE KNOWN AND UNKNOWN PROPERTIES Given the program in Fig. 2.2(a), using a simple analysis for the scope of the variables, we first determine the set of program elements for which we would like to infer properties. These are elements whose properties are *unknown* in the input (i.e. are affected by minification). For example, in

2 http://analytics.google.com/

the case of name inference, this set consists of all the local variables and function parameters in the input program: `e`, `t`, `n`, `r`, and `i`. We also determine the set of elements whose properties are *known* (not affected by minification). One such element is the name of the field `length` in the input program or the names of the methods. Both kinds of elements are shown in Fig. 2.2(b). The goal of the prediction task is to predict the *unknown* properties based on: i) the obtained *known* properties, and ii) the relationship between various elements (discussed below).

STEP 2: BUILD DEPENDENCY NETWORK Next, using features that we later define in Section 2.2.2, we build a dependency network capturing various kinds of relationships between program elements. The dependency network is key to capturing structure when performing predictions and intuitively captures how properties which are to be predicted influence each other. For example, the link between known and unknown properties allows us to leverage the fact that many programs use common anchors (e.g. common API's such as JQuery) meaning that the unknown quantities we aim to predict are influenced by the way the known elements are used by the program. Further, the link between two unknown properties signifies that the prediction for the two properties is related in some way. Dependencies are triplets of the form $\langle n,m,rel \rangle$ where n and m are program elements and rel is the particular relationship between the two elements. In our work all dependencies are triplets, but in general, they can be extended to other more complex relationships that relate more than two elements.

In Fig. 2.2(c), we show three example dependencies between the program elements. For instance, the statement `i += t` generates a dependency ⟨`i`,`t`,`L+=R`⟩, because `i` and `t` are on the left and right side of a `+=` expression. Similarly, the statement `var r = e.length` generates several dependencies including ⟨`r`,`length`,`L=_.R`⟩ which designates that the left part of the relationship, denoted by `L`, appears before the dereference of the right side denoted by `R` (we elaborate on the different types of relationships later in the chapter). For clarity, in Fig. 2.2(c) we include only some of the relationships.

STEP 3: MAP INFERENCE After obtaining the dependency network of a program, the next step is to infer the most likely values (according to a probabilistic model learned from data) for the nodes of the network, a query referred to as MAP inference [75]. As illustrated in Fig. 2.2(d), for the network of Fig. 2.2(c), our system infers the new names `step` and `len`. It also inferred that the previous name `i` was most likely.

Let us consider how we predicted the names `step` and `len`. Consider the network in Fig. 2.2(d). This is the same network as in Fig. 2.2(c) but with additional tables we elaborate on now (these tables are produced as an output of the training phase). Each table is a function that scores the assignment of properties for the nodes connected by the corresponding edge. The function takes as input two properties and returns the score for the pair (intuitively, how likely is the particular pair). In Fig. 2.2(d), each table shows possible functions for the three kinds of relationships we have.

Now, consider the topmost table in Fig. 2.2(d). The first row says that the assignment of `i` and `step` is scored with 0.5. The MAP inference tries to find an assignment of properties to the nodes so that the assignment maximizes the *sum* of the given scoring functions shown in the tables. For the two nodes `i` and `t`, the inference ends up selecting the highest score from that table (i.e., the values `i` and `step`). Similarly for the nodes `i` and `r`. However, for nodes `r` and `length`, the inference *does not* select the topmost row but selects values from the second row. The reason is that if it had selected the topmost row, then the only viable choice (in order to match the value `length`) for the remaining relationship is the second row of that table (with value 0.6). However, the assignment 0.6 leads to a lower *combined* overall score. That is, the MAP inference must take into account the *structure* and dependencies between the nodes and cannot simply select the maximal score of each function and then stop.

OUTPUT PROGRAM Finally, after the new names are inferred, our system transforms the original program to use these names. The output of the entire inference process is captured in the program shown in Fig. 2.2(e). Notice how in this output program, the names tend to accurately capture what the program does.

PREDICTING TYPE ANNOTATIONS Even though we illustrated the inference process for variables names, the overall flow for predicting type annotations is identical. First, using program analysis, we define the program elements with unknown properties to infer type annotations for. Then, we define elements with known properties such as API names or variables with known types. Next, we build the dependency network (some of the relationships overlap with those for names) and finally we perform MAP inference and output a program annotated with the predicted type annotations. One can then run a standard type checker to check whether the predicted types are valid for that program. In our example program shown in Fig. 2.2(e), the predicted type annotations (shown in comments) are indeed valid. In general, when automatically trying to predict semantic properties (such as types) where soundness is required, the approach presented here will have value as part of a guess-and-check loop.

INDEPENDENCE FROM THE MINIFIER USED We note that our name inference process is independent of what the minified names are. In particular, the process will return the same names regardless of which minifier was used to obfuscate the original program (provided these minifiers always rename the *same set* of variables).

2.2 STRUCTURED PREDICTION FOR PROGRAMS

In this section we introduce our approach for predicting program properties. The key idea is to formulate the problem of inferring program properties as structured prediction with *conditional random fields* (CRFs). We first introduce CRFs, then show how the framing is done in a step-by-step manner, and finally discuss the specifics of inference and learning in the context of programs. The prediction framework presented in this section is fairly general and can potentially be instantiated to many different kinds of problems. In Section 2.3 of this work, we instantiate it for name and type annotation inference.

NOTATION: PROGRAMS, LABELS, PREDICTIONS Let $x \in X$ be a program. As with standard program analysis, we will infer properties about program statements or expressions (referred to as program elements). For a program x, each element (e.g. a variable) is identified with an index (a natural number). We will usually need to separate the elements into two kinds: i) elements for which we are interested in inferring properties and ii) elements for which we already know their properties (e.g. these properties may have been obtained via standard program analysis or via manual annotation). We use two helper functions $n, m \colon X \to \mathbb{N}$ to return the appropriate number of program elements for a given program x: $n(x)$ returns the total number of elements of the first kind and $m(x)$ returns the total number of elements of the second kind. To avoid clutter, when x is clear from the context, we write n instead of $n(x)$ and m instead of $m(x)$.

We use the set $Labels_U$ to denote all possible values that a predicted property can take. For instance, in type prediction, $Labels_U$ contains all possible basic types (e.g. number, string, etc). Then, for a program x, we use the notation $\boldsymbol{y} = (y_1, ..., y_{n(x)})$ to denote a vector of predicted program properties. Here, $\boldsymbol{y} \in Y$ where $Y = (Labels_U)^*$. That is, each entry y_i in the vector $\boldsymbol{y}$ ranges over $Labels_U$ and denotes that program element i has a property y_i.

Note that we fixed the number of the inferred program elements $\boldsymbol{y}$ to be $n(x)$. With the approach we show here we determine n and m after reading the input program and not as we make predictions. In later chapters we will remove this limitation with other probabilistic models.

For a program x, we define the vector $\boldsymbol{z}^x = \{z^x_1, ..., z^x_m\}$ to capture the set of properties that are already known; that is, each element in V^x_K is assigned a property from $\boldsymbol{z}^x$. Each z^x_i ranges over a set of properties $Labels_K$ which could potentially differ from the properties $Labels_U$ that we use for inference. For example, if the known properties are integer constants, $Labels_K$ will be all valid integers. To avoid clutter where x is clear from the context, we use $\boldsymbol{z}$ instead of $\boldsymbol{z}^x$. We use $Labels = Labels_U \cup Labels_K$ to denote the set of all properties.

PROBLEM DEFINITION Let $D = \{\langle x^{(j)}, \boldsymbol{y}^{(j)}\rangle\}^t_{j=1}$ denote the training data: a set of t programs each annotated with corresponding program properties. Our goal is to learn a model that captures the conditional probability $Pr(\boldsymbol{y} \mid x)$. Once the model is learned, we can predict properties of new programs by posing the following query (also known as MAP or Maximum a Posteriori query):

Given a **new** program x, find $\boldsymbol{y} = \arg\max_{\boldsymbol{y}' \in \Omega_x} Pr(\boldsymbol{y}' \mid x)$

That is, for a new program x, we aim to find the most likely assignment of program properties $\boldsymbol{y}$ according to the probabilistic distribution. Here, $\Omega_x \subseteq Y$ describes the set of possible assignments of properties $\boldsymbol{y}'$ for the program elements of x. The set Ω_x is important as it allows restricting the set of possible properties and is useful for encoding problem-specific constraints. For example, in type annotation inference, the set Ω_x may restrict resulting annotation to types that make the resulting program typecheck.

2.2.1 *Conditional Random Fields (CRFs)*

We now describe CRFs, a particular model defined in [79] and previously used for a range of tasks such as natural language processing, image processing and others. CRFs represent the conditional probability $Pr(\boldsymbol{y} \mid x)$. We consider the case where the factors are positive in which case, without loss of generality, any conditional probability of properties $\boldsymbol{y}$ given a program x can be encoded as follows:

$$Pr(\boldsymbol{y} \mid x) = \frac{1}{Z(x)} \exp(score(\boldsymbol{y}, x))$$

where *score* is a function that returns a real number indicating the score of an assignment of properties $\boldsymbol{y}$ for a program x. Assignments with higher score are more likely than assignments with lower score. $Z(x)$, called the *partition function*, ensures that the above expression does in fact encode a conditional distribution. It returns a real number depending only on the program x, such that the probabilities over all possible assignments $\boldsymbol{y}$ sum to 1, i.e.:

$$Z(x) = \sum_{\boldsymbol{y} \in \Omega_x} \exp(score(\boldsymbol{y}, x))$$

Without loss of generality, *score* can be expressed as a composition of a sum of k feature functions f_i associated with weights w_i:

$$score(\boldsymbol{y}, x) = \sum_{i=1}^{k} w_i f_i(\boldsymbol{y}, x) = \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{y}, x)$$

Here, $\boldsymbol{f}$ is a vector of functions f_i and $\boldsymbol{w}$ is a vector of weights w_i. The feature functions $f_i \colon Y \times X \to \mathbb{R}$ are used to score assignments of program properties. This representation of score functions is particularly suited for learning (as the weights $\boldsymbol{w}$ can be learned from data). Based on the definition above, we can now define a *conditional random field*.

Definition 2.1 (Conditional Random Field (CRF))**.** A model for the conditional probability of labels $\boldsymbol{y}$ given observations x is called (log-linear) conditional random field, if it is represented as:

$$Pr(\boldsymbol{y} \mid x) = \frac{1}{Z(x)} \exp(\boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{y}, x))$$

FEATURES AS CONSTRAINTS Feature functions are key to controlling the likelihood of an assignment of properties $\boldsymbol{y}$ for a program x. For instance, a feature function can be defined in a way which prohibits or lowers the score of undesirable predictions: say if $f_i(\boldsymbol{y}^B, x) = -\infty$, the feature function f_i (with weight $w_i > 0$) disables an assignment $\boldsymbol{y}^B$, thus resulting in $Pr(\boldsymbol{y}^B \mid x) = 0$.

We discuss how the feature functions are defined in the next subsection. Note that feature functions are defined *independently of the program* being queried, and are only based on the particular prediction problem we are interested in. For example, when we predict types of a program, we define one set of feature functions and when we predict identifier names, we define another set. Once defined, the feature functions are reused for predicting the particular kind of property we are interested in for *any* input program.

2.2.2 *Making Predictions for Programs*

We next describe a step-by-step method for predicting program properties using CRFs. We first give definitions, then show how to build a network between elements, then describe how to build the feature functions f_i based on that network and finally illustrate how to score a prediction.

DEFINING PROGRAM ELEMENT RELATIONS Before we start with the problem of making predictions, we need to define what kind of relations between program elements are going to be used for the predictions. Let the set of all element relations be *Rels*. In practice, the set *Rels* is specific to the application that is solved with CRFs. An example relation that we considered in our running example is `L+=R` as seen in Fig. 2.2(c) where the relation relates the variable `i` to the variable `t` because there is an expression `i+=t` in the code. Detailed concrete instantiations for the relations we use in our applications for JavaScript are given in Section 2.3.

DEFINING PAIRWISE FEATURE FUNCTIONS Let $\{\psi_i\}_{i=1}^{k}$ be a set of pairwise feature functions where each ψ_i: $\mathit{Labels} \times \mathit{Labels} \times \mathit{Rels} \rightarrow \mathbb{R}$

scores a *pair* of program properties when they are related with the given relation from *Rels*. As an example feature function, consider:

$$\psi_{example}(l_1, l_2, e) = \begin{cases} 1 & \textbf{if } l_1 = \texttt{i} \textbf{ and } l_2 = \texttt{step} \textbf{ and } e = \texttt{L+=R} \\ 0 & \textbf{otherwise} \end{cases}$$

In the example from Fig. 2.2, this feature function scores with 1 an assignment of names of `i` and `step` to the variables participating in a "+=" expression and scores with 0 any other assignment of names. We discuss the particular kinds of pairwise feature functions for JavaScript in Section 2.3.4. Although in this work we use pairwise feature functions, there is nothing specific in our approach which precludes us from using functions with higher arity. Using the definitions of the set of possible program element relations and feature functions were defined, we proceed to the step-by-step prediction method.

STEP 1: BUILD DEPENDENCY NETWORK The first step to make predictions for an input program x is to build what we refer to as a dependency network $G^x = \langle V^x, E^x \rangle$. This network captures dependencies between the predictions made for the program elements of interest. Here, $V^x = V^x_U \cup V^x_K$ denotes the set of program elements (e.g. variables) and consists of elements for which we would like to predict properties V^x_U and elements whose properties we already know V^x_K. The set of edges $E^x \subseteq V^x \times V^x \times Rels$ denotes the fact that there is a relationship between two program elements and describes what that relationships is. This definition of network is also called multi-graph because there is no restriction on having only a single edge between a pair of edges – our definition permits multiple dependencies with different *Rels* between a pair of program elements.

Recall that the number of known and unknown properties for a program x were defined to be $n(x)$ and $m(x)$. Thus, the resulting multi-graph G^x consists of $n(x) + m(x)$ nodes.

STEP 2: DEFINE FEATURE FUNCTIONS OVER ENTIRE PROGRAMS x Recall that the definition of a CRF (Definition 2.1) uses a vector features functions $\boldsymbol{f}(\boldsymbol{y}, x)$ defined over all unknown program elements $\boldsymbol{y}$ in a program x. In contrast, our pairwise feature functions only apply on two program properties. In this step, we define feature functions f_i from ψ_i using the network G^x.

Let the assignment vector $A = (\boldsymbol{y}, \boldsymbol{z}^x)$ be a concatenation of two assignments: the unknown properties $\boldsymbol{y}$ and the known properties $\boldsymbol{z}^x$ in x. As usual, the property of the j'th element of the vector A is accessed via A_j. Then, the feature function f_i is defined as the sum of the applications of its corresponding pairwise feature function ψ_i over the set of all network edges in G^x as follows:

$$f_i(\boldsymbol{y}, x) = \sum_{\langle a,b,rel \rangle \in E^x} \psi_i((\boldsymbol{y}, \boldsymbol{z}^x)_a, (\boldsymbol{y}, \boldsymbol{z}^x)_b, rel)$$

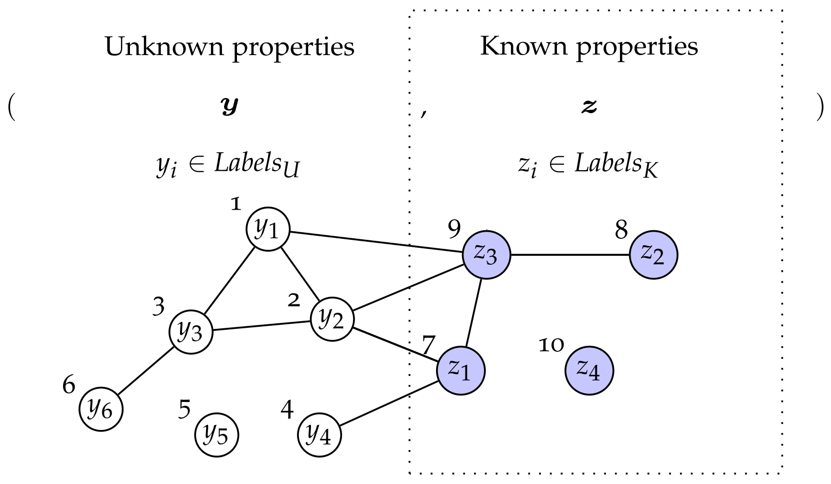
STEP 3: SCORE A PREDICTION $\boldsymbol{y}$ Based on the above definition of feature functions, we can now define how to obtain a total score for a prediction $\boldsymbol{y}$. According to the definition of *score* and by substitution, we obtain:

$$score(\boldsymbol{y}, x) = \sum_{i=1}^{k} w_i f_i(\boldsymbol{y}, x) = \sum_{\langle a,b,rel \rangle \in E^x} \sum_{i=1}^{k} w_i \psi_i((\boldsymbol{y}, \boldsymbol{z})_a, (\boldsymbol{y}, \boldsymbol{z})_b, rel)$$

That is, for a program x and its dependency network G^x, by using the pairwise functions ψ_i and the learned weights w_i associated with each ψ_i, we can obtain the score of a prediction $\boldsymbol{y}$.

EXAMPLE Let us illustrate the above steps as well as some key points about G^x on the simple example in Fig. 2.3. Here we have 6 program elements for which we would like to predict program properties. We also have 4 program elements whose properties we already know. Each program element is a node with an index shown outside the circles. The edges indicate relationships between the nodes and the labels inside the nodes are the predicted program properties or the already known properties. As explained earlier, the known properties $\boldsymbol{z}$ are fixed before the prediction process begins. In a structured prediction problem, the properties $y_1, \ldots, y_6$ of program elements $1 \ldots 6$ are predicted such that $Pr(\boldsymbol{y} \mid x)$ is maximal.

EDGES AND DEPENDENCIES OF PROGRAM PROPERTIES The shape of the dependency network G^x for a program x determines several important properties about the predictions $\boldsymbol{y}$ that we illustrate in Fig. 2.3. First, predictions for a node (e.g. 5) disconnected from all other nodes in the network can be made independently of the predictions made for the other nodes. Second, nodes 2 and 4 are connected but only



Prediction: $\boldsymbol{y} = \arg\max_{\boldsymbol{y'} \in \Omega_x} Pr(\boldsymbol{y'} \mid x)$

Figure 2.3: A general schema for building a network for a program x and finding the best scoring assignment of properties $\boldsymbol{y}$.

via nodes with known predictions. Therefore, the properties for nodes 2 and 4 can be assigned independently of one another. That is, the prediction y_2 of node 2 will not affect the prediction y_4 of node 4 with respect to the total score function and vice versa. The reason why this is the case is due to a property in CRFs known as *conditional independence*. We say that the prediction for a pair of nodes a and b is conditionally independent given a set of nodes C if the predictions for the nodes in C are fixed and all paths between a and b go through a node in C. This is why the predictions for nodes 2 and 4 are conditionally independent of node 7. Conditional independence is an important property of CRFs and is leveraged to speed up both the inference and the learning algorithms. We do not discuss conditional independence further but refer the reader to a standard reference [75]. Finally, a path between two nodes (not involving known nodes) means that the predictions for these two nodes may (and generally will) be dependent on one another. For example, nodes 2 and 6 are transitively connected (without going through known nodes) meaning that the prediction for node 2 can influence the scores of predictions for node 6.

2.2.3 *MAP Inference*

Recall that the key query we perform is MAP inference:

Given a program x, find $\boldsymbol{y} = \arg\max_{\boldsymbol{y}' \in \Omega_x} \; Pr(\boldsymbol{y}' \mid x)$

In a CRF, this amounts to the query:

$$\boldsymbol{y} = \underset{\boldsymbol{y}' \in \Omega_x}{\arg\max} \; \frac{1}{Z(x)} \exp(score(\boldsymbol{y}', x))$$

where:

$$Z(x) = \sum_{\boldsymbol{y}'' \in \Omega_x} \exp(score(\boldsymbol{y}'', x))$$

Note that $Z(x)$ does not depend on $\boldsymbol{y}'$ and as a result it does not affect the final choice for the prediction $\boldsymbol{y}$. This is an important observation, because computing $Z(x)$ is generally very expensive as it may need to sum over all possible assignments $\boldsymbol{y}''$. Therefore, we can exclude $Z(x)$ from the maximized formula. Next, we take into account the fact that exp is a monotonically increasing function enabling us to remove exp from the equation. This leads to an equivalent simplified query:

$$\boldsymbol{y} = \underset{\boldsymbol{y}' \in \Omega_x}{\arg\max} \, score(\boldsymbol{y}', x)$$

This is, an algorithm answering the MAP inference query must ultimately maximize the *score* function. For instance, for the example in Fig. 2.3, once we fix the known labels $\boldsymbol{z}$, we need to find labels $\boldsymbol{y}$ such that *score* is maximized.

In principle, at this stage one can use any algorithm to answer the MAP inference query. For instance, a naïve but inefficient way to solve this query is by trying all possible outcomes $\boldsymbol{y}' \in \Omega_x$ and scoring each of them to select the highest scoring one. Other exact and inexact [75] inference algorithms exist if the network G^x and the outcomes set Ω_x have certain restrictions (e.g. G^x is a tree).

SPECIFICS OF PROGRAMS Unfortunately, the problem with existing inference algorithms is that they are too slow to be usable for our problem domain (i.e. programs). For example, in typical applications of CRFs [75] such as text parsing [79], it is unusual to have more than a handful of possible assignments for an element (e.g. 10), while in our case there could potentially be thousands of possible assignments per element. To address this issue, in Section 2.4 we present a fast and approximate MAP inference algorithm that is tailored to the specifics of dealing with programs: the shape of the feature functions, the unrestricted nature of G^x and the massive set of possible assignments.

2.2.4 *Learning*

The weights $\boldsymbol{w}$ used in the scoring function *score* cannot be directly obtained by means of counting in the training data [75, § 20.3.1]. Instead, we use stochastic gradient descent and learning technique from online support vector machines: given a training dataset $D = \{\langle x^{(j)}, \boldsymbol{y}^{(j)}\rangle\}_{j=1}^{t}$ of t samples, the goal is to find $\boldsymbol{w}$ such that the given assignments $\boldsymbol{y}^{(j)}$ are the highest scoring assignments in as many training samples as possible subject to additional learning constraints. We discuss the learning procedure in detail in Section 2.5.

Before we give more details about the inference and learning procedures, we describe a specific application in the context of the JavaScript programming language.

2.3 JSNICE: PREDICTING NAMES AND TYPE ANNOTATIONS FOR JAVASCRIPT

In this section we present an example of using our structured prediction approach presented in Section 2.2 for inferring two kinds of properties: (i) predicting names of local variables, and (ii) predicting type annotations of function arguments. We investigate the above challenges in the context of JavaScript, a popular language where addressing the above two questions is of significant importance. We do note however that much of the machinery discussed in this section applies as-is to other programming languages.

PRESENTATION FLOW Recall that in Section 2.2.2, we defined a method to score predictions of program properties of a program x by defining a network $G^x = \langle V^x, E^x\rangle$ and pairwise feature functions $\{\psi_i\}_{i=1}^{k}$. In what follows, we first present the probabilistic name prediction and define V^x for that problem. We then present the probabilistic type prediction and define V^x in that context. Then, we define E^x: which program elements from V^x are related as well as how they are related (that is, *Rels*). Some of these relationships are similar for both prediction problems and hence we discuss them in the same section. Finally, we discuss how to obtain the pairwise feature functions ψ_i.

2.3.1 *Probabilistic Name Prediction*

The goal of our name prediction task is to predict the (most likely) names of local variables in a given program x. The way we proceed to solve this problem is as follows. First, as outlined in Section 2.2, we identify the set of known program elements, referred to as V_K^x, as well as the set of unknown program elements for which we will be predicting new names, referred to as V_U^x.

For the name prediction problem, we take V_K^x to be all constants, objects properties, methods and global variables of the program x. Each program element in V_K^x can be assigned values from the set $Labels_K = JSConsts \cup JSNames$, where $JSNames$ is a set of all valid identifier names, and $JSConsts$ is a set of possible constants. We note that object property names and API names are modeled as constants, as the dot (`.`) operator takes an object on the left-hand side and a string constant on the right-hand side. We define the set V_U^x to contain all local variables of a program x. Here, a variable name belonging to two different scopes leads to two program elements in V_U^x. Finally, $Labels_U$ ranges over $JSNames$.

To ensure the newly predicted names are semantic preserving, we ensure that the prediction satisfies the following constraints:

1. *All* references to a renamed local variable must be renamed to the same name.
2. The predicted identifier names must not be reserved keywords.
3. The prediction must not suggest the same name for two different variables in the same scope.

The first property is naturally enforced in the way we define V_U^x where each element corresponds to a local variable as opposed to having a unique element for every variable occurrence in the program. The second property is enforced by making sure the set $Labels_U$ from which predicted names are drawn does not contain keywords. Finally, we enforce the third constraint by restricting Ω_x so that predictions with conflicting names are prohibited.

2.3.2 *Probabilistic Type Annotation Prediction*

Our second application involves probabilistic type annotation inference of function parameters. Focusing on function parameters is particularly important for JavaScript, a duck-typed language lacking type annotations. Without knowing the types of function parameters, a forward type inference analyzer will fail to derive precise and meaningful types (except the types of constants and those returned by common APIs such as DOM APIs). As a result, real-world programs using libraries cannot be analyzed precisely [67].

Instead, we propose to probabilistically predict the type annotations of function parameters. Here, our training data consists of a set of JavaScript programs that have already been annotated with types for function parameters. In JavaScript, these annotations are provided in a specially formatted comments known as JSDoc[3].

The simplified language over which we predict type annotations is defined as follows:

$$\begin{array}{rll} expr ::= & val \mid var \mid expr_1(expr_2) \mid expr_1 \circledast expr_2 & \textit{Expression} \\ val ::= & \lambda var : \tau.expr \mid n & \textit{Value} \end{array}$$

Here, n ranges over constants ($n \in JSConsts$), var is a meta-variable ranging over the program variables, $\circledast$ ranges over the standard binary operators (+, -, *, /, ., <, ==, ===, etc.), and τ ranges over all possible variable types. That is, $\tau = \{?\} \cup L$ where L is a set of types (we discuss how to instantiate L below) and ? denotes the unknown type. To be explicit, we use the set $JSTypes$ where $JSTypes = \tau$. We use the function:

$$[]_x : expr \rightarrow JSTypes$$

to obtain the type of a given expression in a given program x. This map can be manually provided or built using program analysis. When the program x is clear from the context we use $[e]$ as a shortcut for $[]_x(e)$.

3 https://developers.google.com/closure/compiler/docs/js-for-compiler

DEFINING KNOWN AND UNKNOWN PROGRAM ELEMENTS As usual, our first step is to define the two sets of known and unknown elements. We define the set of unknown program elements as follows:

$$V_U^x = \{e \mid e \textbf{ is } var, [e] = ?\}$$
$$Labels_U = JSTypes$$

That is, V_U^x contains variables whose type is unknown. We differentiate between the type $\top$ and the unknown type ? in order to allow for finer control over which types we would like to predict. For instance, a type may be $\top$ if a classic type inference algorithm fails to infer more precise types (usually, standard inference only discovers types of constants and values returned by common APIs, but fails to infer types of function parameters because it is unable to approximate all the possible callers of a function). A type may be denoted as unknown (i.e. ?) if the type inference did not even attempt to infer types for the particular expression (e.g. function parameters). Of course, in the above definition of V_U^x we could also include $\top$ and use our approach to potentially refine the results of classic type inference.

Next, we define the set of known elements V_K^x. Note that V_K^x can contain any expression, not just variables like V_U^x above:

$$V_K^x = \{e \mid e \textbf{ is } expr, [e] \neq ?\} \cup \{n \mid n \textbf{ is } constant\}$$
$$Labels_K = JSTypes \cup JSConsts$$

That is, V_K^x contains both, expressions whose types are known as well as constants. Currently, we do not apply any global restriction on the set of possible assignments Ω_x, that is, $\Omega_x = (JSTypes)^n$ (recall that n is a function which returns the number of elements whose property is to be predicted). This means that we rely entirely on the learning to discover the rules that will produce non-contradicting types. The only restriction (discussed below) that we apply is constraining *JSTypes* when performing predictions.

DEFINING ***JSTypes*** So far, we have not discussed the exact contents of the set *JSTypes* except to state that $JSTypes = \{?\} \cup L$ where L is a set of types. The set L can be instantiated in various ways. In this work, we chose to define L as $L = \mathcal{P}(T)$ where $\langle T, \sqsubseteq \rangle$ is a complete lattice of types with T and $\sqsubseteq$ as defined in Fig. 2.4. In the figure we use "..." to denote a potentially infinite number of user-defined object types.

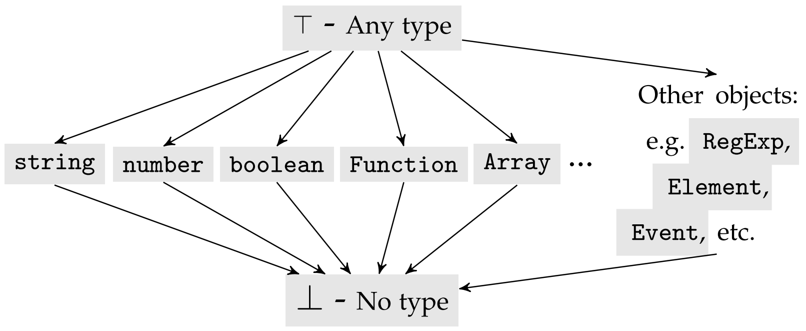


Figure 2.4: The lattice of types over which prediction occurs.

OBTAINING *JSTypes* We note several important points here. First, the set *JSTypes* is built during training from a finite set of possible types that are already manually provided or are inferred by the classic type inference. Therefore, for a given training data, *JSTypes* is necessarily a finite set. Second, because *JSTypes* may contain a subset of types $O \subseteq JSTypes$ specific to a particular program in the training data, it may be the case that when we are considering a new program whose types are to be predicted, the types found in O are simply not relevant to that new program (for instance, the types in O refer to names that do not appear in the new program). Therefore, when we perform prediction, we filter irrelevant types from the set *JSTypes*. This is the only restriction we consider when performing type predictions. Finally, because L is defined as a powerset lattice, it encodes (in this case, a finite number of) disjunctions. That is, a variable whose type is to be predicted ranges over exponentially many subsets allowing many choices for the type. For example, a variable can have a type {`string`, `number`} which for convenience can also be written as `string` $\vee$ `number`.

2.3.3 *Relating Program Elements*

We next describe the relationships we introduce between program elements. These relationships define how to build the set of edges E^x of a program x. Since the program elements for both prediction tasks are similar (e.g. they both contain JavaScript constants, variables and expressions), we discuss the relationships we use for each task together. If a relationship is specific to a particular task, we explicitly state so when describing it.

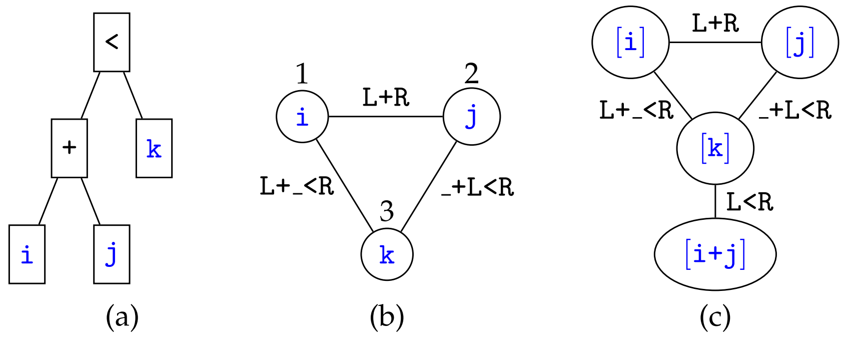


Figure 2.5: (a) the AST of expression `i+j<k`, and two dependency networks built from the AST relations: (b) for name predictions, and (c) for type predictions.

2.3.3.1 *Relating Expressions*

The first relationship we discuss is syntactic in nature: it relates two program elements based on the their syntactic relationship in the program's Abstract Syntax Tree (AST). Let us consider how we obtain the relationships for the expression `i+j<k`. First, we build the AST of the expression shown in Fig. 2.5 (a). Suppose we are interested in performing name prediction for variables `i`, `j` and `k` (denoted by program properties with indices 1, 2 and 3 respectively), that is, $V_U^x = \{1, 2, 3\}$. Then, we build the dependency network as shown in Fig. 2.5 (b) to indicate that the prediction for the three elements are dependent on one another (with the particular relationship shown over the edge). For example, the edge between `1` and `2` represents the relationships that these nodes participate in an expression `L+R` where `L` is a node for `1` and `R` is a node for `2`.

The relationships are defined using the following grammar:

$$\begin{aligned} rel_{ast} &::= rel_L(rel_R) \;\mid\; rel_L \circledast rel_R \\ rel_L &::= \mathtt{L} \;\mid\; rel_L(\_) \;\mid\; \_(rel_L) \;\mid\; rel_L \circledast \_ \;\mid\; \_ \circledast rel_L \\ rel_R &::= \mathtt{R} \;\mid\; rel_R(\_) \;\mid\; \_(rel_R) \;\mid\; rel_R \circledast \_ \;\mid\; \_ \circledast rel_R \end{aligned}$$

All relationships rel_{ast} are part of *Rels*, that is, $rel_{ast} \in Rels$. Here, as discussed earlier, $\circledast$ ranges over binary operators. All relationships derived using the above grammar have exactly one occurrence of `L` and `R`. For a relationship $r \in rel_{ast}$, let $r[x/\mathtt{L}, y/\mathtt{R}, e/\_]$ denote the expression where x is substituted for `L`, y is substituted for `R` and the expression e is substituted for $\_$. Then, given two program elements a and b and a relationship $r \in rel_{ast}$, a match is said to exist if $r[a/\mathtt{L}, b/\mathtt{R}, [expr]/\_] \cap Exp(x) \neq \emptyset$ (here, $[expr]$ denotes all possible expressions in the programming language and $Exp(x)$ is all expressions of program x). An edge $(a, b, r) \in E^x$ between two program elements a and b exists if there exists a match between a, b and r.

Note that for a given pair of elements a and b there could be more than one relationship which matches, that is, both $r_1, r_2 \in rel_{ast}$ match where $r_1 \neq r_2$ (therefore, there could be multiple edges between a and b with different relationships). Here, we leverage the fact that by definition G^x is a multi-graph.

The relationships described above are useful for both name and type inference. In the case of predicting names, the expressions being related are always variables, while for type annotations, the expressions need not be restricted to variables. For example, in Fig. 2.5(c) there is a relationship between the types of `k` and `i+j` via `L<R`. Note that our rules do not directly capture relationships between [`i`] and [`i+j`], but they are transitively dependent. Still, many useful and interesting direct relationships for type inference are present. For instance, in classic type inference, the relationship `L=R` implies a constraint rule $[\mathtt{L}] \sqsupseteq [\mathtt{R}]$ where $\sqsupseteq$ is the super-type relationship (indicated in Fig. 2.4). Interestingly, our inference model can learn such rules instead of providing them explicitly.

2.3.3.2 *Aliasing Relations*

Another kind of (semantic) relationship we introduce is that of aliasing. Let $alias(e)$ denote the set of expressions that may alias with the expression e (this information can be determined via standard alias analysis [123]).

ARGUMENT-TO-PARAMETER We introduce the `ARG_TO_PM` relationship which relates arguments of a function invocation (the arguments can be arbitrary expressions) with parameters in the function declaration (variables whose names or types are to be inferred). Let $e_1(e_2)$ be an invocation of the function captured by the expression e_1. Then, for all possible declarations of e_1 (those are an over-approximation), we relate the argument of the call e_2 to the parameter in the declaration. That is, for any $v \in \{p \mid (\lambda p : \tau.e) \in alias(e_1)\}$, we add the edge

$(e_2, v, \texttt{ARG\_TO\_PM})$ to E^x. When predicting names e_2 is always a variable, while when predicting types e_2 is not restricted to variables.

TRANSITIVE ALIASING Second, we introduce a transitive aliasing relationship referred to as $(r, \texttt{ALIAS})$ between variables which may alias. This is a relationship that we introduce only when predicting types. Let a and b be related via the relationship r where r ranges over the grammar defined earlier. Then, for all $c \in \mathit{alias}(b)$ where c is a variable, we include the edge $(a, c, (r, \texttt{ALIAS}))$.

2.3.3.3 *Function name relationships*

We introduce two relationships referred to as MAY_CALL and MAY_ACCESS. These relationships are only used when predicting names and are particularly useful for predicting function names. The reason is that in JavaScript many of the local variables are function declarations. The MAY_CALL relationship relates a function name f with names of other functions g that f may call (this semantic information can be obtained via program analysis). That is, if a function f may call function g, we add the edge $(f, g, \texttt{MAY\_CALL})$ to the set of edges E^x. Similarly, if in a function f, there is an access to an object field named fld, we add the edge $(f, fld, \texttt{MAY\_ACCESS})$ to the set E^x. Naturally, f and g are allowed to only range over variables (as when predicting names the nodes represent variables and not arbitrary expressions), and the name of an object field fld is a string constant.

2.3.4 *Obtaining Pairwise Feature Functions*

Finally, we describe how to define the pairwise feature functions $\{\psi_i\}_{i=1}^{k}$. We obtain these functions as a pre-processing step before the *training phase* begins. Recall that our training set $D = \{\langle x^{(j)}, \boldsymbol{y}^{(j)}\rangle\}_{j=1}^{t}$ consists of t programs where for each program x we are given the corresponding properties $\boldsymbol{y}$. For each tuple $(x, \boldsymbol{y}) \in D$, we define the set of features as follows:

$$\mathit{features}(x, \boldsymbol{y}) = \{((\boldsymbol{y}, \boldsymbol{z})_a, (\boldsymbol{y}, \boldsymbol{z})_b, \mathit{rel}) \mid (a, b, \mathit{rel}) \in E^x\}$$

Then, for the entire training set we obtain all features as follows:

$$all\_features(D) = \bigcup_{j=1}^{t} features(x^{(j)}, \boldsymbol{y}^{(j)})$$

We then define the pairwise feature functions to be indicator functions of each feature triple in *all_features*(D). Let the *all_features*$(D) = \{\ \langle l_i^1, l_i^2, rel_i\rangle\ \}_{i=1}^{k}$. Then, we define the feature functions as follows:

$$\psi_i(l^1, l^2, rel) = \begin{cases} 1 & \textbf{if } l^1 = l_i^1 \textbf{ and } l^2 = l_i^2 \textbf{ and } rel = rel_i \\ 0 & \textbf{otherwise} \end{cases}$$

In addition to indicator functions, we have features for equality of program properties $\psi_=(l_1, l_2, rel)$ that return 1 if and only if the two related labels are equal. Our feature functions are fully inferred from the available training data and the network G^x of each program x. After the feature functions are defined, in the training phase (discussed later), we learn their corresponding weights $\{w_i\}_{i=1}^{k}$ (k is the number of pairwise functions). Note that the weights and the feature functions can vary depending on the training data D, but both are independent of the program for which we are trying to predict properties.

2.4 PREDICTION ALGORITHM

In this section we present our inference algorithm for making predictions (also referred to as MAP inference). Recall that predicting properties $\boldsymbol{y}$ of a program x involves finding a $\boldsymbol{y}$ such that:

$$\boldsymbol{y} = \arg\max_{\boldsymbol{y}' \in \Omega_x} Pr(\boldsymbol{y}' | x) = \arg\max_{\boldsymbol{y}' \in \Omega_x} score(\boldsymbol{y}', x) = \arg\max_{\boldsymbol{y}' \in \Omega_x} \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{y}', x)$$

When designing our inference algorithm, a key objective was optimizing the speed of prediction. There are two reasons why speed is critical. First, we expect prediction to be done interactively, as part of a program development environment or as a service (e.g., via a public web site such as JSNice). This requirement renders any inference algorithm that takes more than a few seconds unacceptable. Second (as we will see later), the prediction algorithm is part of the inner-most loop of training, and hence its performance directly impacts an already costly and work-intensive training phase.

EXACT ALGORITHMS Exact inference in CRFs is generally NP-hard and computationally prohibitive in practice. This problem is well known and hard specifically for denser networks with no predefined shape like the ones we obtain from programs [75].

APPROXIMATE ALGORITHMS Previous studies [42, 69] for MAP inference in networks of arbitrary shapes discuss loopy-belief propagation, greedy algorithms, combination approaches or graph-cut based algorithms. In their results, they show that advanced approximate algorithms may result in higher precision for the inference and the learning, however they also come at the cost of significantly more computation. Their experiments confirm that techniques such as belief propagation are consistently at least an order of magnitude slower than greedy algorithms.

As our focus is on performance, we proceeded with a greedy approach (also known as iterated conditional modes [16]). Our algorithm is tailored to the nature of our prediction task (especially when predicting names where we have a massive number of possible assignments for each element) in order to significantly improve the computational complexity over a naïve greedy approach. In particular, our algorithm leverages the shape of the feature functions discussed in Section 2.3.4. In essence, the approach works by selecting candidate assignments from a beam of s-best possible labels leading to significant gains in performance at the expense of slightly higher chance of obtaining non-optimal assignments.

2.4.1 *Greedy Inference Algorithm*

Algorithm 1 illustrates our greedy inference procedure. The inference algorithm has four inputs: (i) a network G^x obtained from a program x, (ii) an initial assignment of properties for the unknown elements $\boldsymbol{y}_0$, (iii) the obtained known properties $\boldsymbol{z}$, and (iv) the pairwise feature functions and their weights. The way these inputs are obtained was already described earlier in Section 2.2. The output of the algorithm is an approximate prediction $\boldsymbol{y}$ which also conforms to the desired

```
Input: network $G^x = \langle V^x, E^x \rangle$ of program $x$,
       initial assignment of $n$ unknown properties $\boldsymbol{y}_0 \in \Omega_x$,
       known properties $\boldsymbol{z}$
       pairwise feature functions $\psi_i$ and their learned weights $w_i$
Output: $\boldsymbol{y} \approx \arg\max_{\boldsymbol{y}' \in \Omega_x} \big(score(\boldsymbol{y}', x)\big)$
begin
    $\boldsymbol{y} \leftarrow \boldsymbol{y}_0$
    for $pass \in [1..num\_passes]$ do
        // for each node with unknown property in the graph $G^x$
        for $v \in [1..n]$ do
            $E_v \leftarrow \{(v, \_, \_) \in E^x\} \cup \{(\_, v, \_) \in E^x\}$
            $score_v \leftarrow scoreEdges\big(E_v, (\boldsymbol{y}, \boldsymbol{z})\big)$
            for $l' \in candidates\big(v, (\boldsymbol{y}, \boldsymbol{z}), E_v\big)$ do
                $l \leftarrow y_v$      // get current label of $v$
                $y_v \leftarrow l'$     // change label of $v$ in $\boldsymbol{y}$
                $score'_v \leftarrow scoreEdges\big(E_v, (\boldsymbol{y}, \boldsymbol{z})\big)$
                if $\boldsymbol{y} \in \Omega_x \wedge score'_v > score_v$ then
                    $score_v \leftarrow score'_v$
                else
                    $y_v \leftarrow l$      // no score improvement: revert label.
                end
            end
        end
    end
    return $\boldsymbol{y}$
end
```

Algorithm 1: Greedy Inference Algorithm

constraints Ω_x. The algorithm also uses an auxiliary function called *scoreEdges* defined as follows:

$$scoreEdges(E, A) = \sum_{(a,b,rel)\in E} \sum_{i=1}^{k} w_i \psi_i(A_a, A_b, rel)$$

The $scoreEdges(E, A)$ function is the same as *score* defined earlier except that *scoreEdges* works on a subset of the network edges $E \subseteq E^x$. Given a set of edges E and an assignment of elements to properties A, *scoreEdges* iterates over E, applies the appropriate feature functions to each edge and sums up the results.

The basic idea of the algorithm is to start with an initial assignment $\boldsymbol{y}_0$ (Line 2) and to make a number of passes over all nodes in the network, attempting to improve the score of the current prediction $\boldsymbol{y}$. The algorithm works on a node by node basis: it selects a node $v \in [1..n]$ and then finds a label for that node which improves the score of the assignment. That is, once the node v is selected, the algorithm first obtains the set of edges E_v in which v participates (shown on Line 6) and computes via *scoreEdges* the contribution of the edges to the total score. Then, the inner loop starting at Line 8 tries new labels for the element v from a set of candidate labels and accepts only labels that lead to a score improvement.

TIME COMPLEXITY The time complexity for one iteration of the prediction algorithm depends on the number of nodes, the number of adjacent edges for each node and the number of candidate labels. Since the total number of edges in the graph $|E^x|$ is a product of the number of nodes and the number of edges per node, then one iteration of the algorithm has $O(d|E^x|)$ time complexity, where d is the total number of possible candidate assignment labels for a node (obtained on Line 8).

2.4.2 *Obtaining Candidates*

Our algorithm does not try all possible labels for a node. Instead, we define the function $candidates(v, A, E)$ which suggests candidate labels given a node v, assignment A, and a set of edges E. Recall that $all\_features(D)$ is a (large) set of triples (l^1, l^2, r) obtained from the training data D relating labels l^1 and l^2 via r. Further, our pairwise feature functions $\{\psi_i\}_{i=1}^{k}$ (where $k = |all\_features(D)|$) defined earlier are indicator functions meaning there is a one to one correspondence between a triple (l^1, l^2, r) and a pairwise function. In the training phase (discussed later), we learn a weight w_i associated with each function ψ_i (and because of the one-to-one mapping, with each triple (l^1, l^2, r)). We use these weights in order to restrict the set of possible assignments we consider for a node v. Let top_s be a function which given a set of features (triples) returns the top s triples based on the respective weights. Let for convenience $F = all\_features(D)$. Then, we define the following auxiliary functions:

$$topL_s(lbl, rel) = top_s(\{t \mid t_l = lbl \wedge t_{rel} = rel \wedge t \in F\})$$
$$topR_s(lbl, rel) = top_s(\{t \mid t_r = lbl \wedge t_{rel} = rel \wedge t \in F\})$$

The above functions can be easily pre-computed for a fixed beam size s and all triples in the training data F. Finally, we define:

$$\begin{aligned} candidates(v, A, E) = \\ = \bigcup_{\langle a,v,rel\rangle \in E} \{l^2 \mid \langle l^1, l^2, r\rangle \in topL_s(A_a, rel)\} \quad \cup \\ \bigcup_{\langle v,b,rel\rangle \in E} \{l^1 \mid \langle l^1, l^2, r\rangle \in topR_s(A_b, rel)\} \end{aligned}$$

The meaning of the above function is that for every edge adjacent to v, we consider at most s of the highest scoring triples (according to the learned weights). This results in a set of possible assignments for v used to drive the inference algorithm. The beam parameter s controls a trade-off between precision and running time. Lower values of s decrease the chance of predicting a good candidate label, while higher s make the algorithm consider more labels and run longer. Our experiments show that good candidate labels can be obtained with fairly low values of s such as 32. Thanks to this observation, the prediction runs orders of magnitude faster than naïve algorithms that try all possible labels.

MONOTONICITY At each pass of our algorithm, we iterate over the nodes of G^x and update the label of each node only if this leads to a score improvement (at Line 12). Since we always increase the score of the assignment $\boldsymbol{y}$, after a certain number of iterations, we reach a fixed point assignment $\boldsymbol{y}$ that can no longer be improved by the algorithm. The local optimum however, is not guaranteed to be a global optimum. Since we cannot give a complete optimality guarantee, to achieve further speed ups, we also cap the number of algorithm passes at a constant *num_passes*.

2.4.3 *Additional Improvements*

To further decrease the computation time and possibly increase the precision of our algorithm, we made two improvements. First, if a node has more than a certain number of adjacent nodes, we decrease the size of the beam s. In our implementation we decrease the beam size by a factor of 16 if a node has more than 32 adjacent nodes. At almost no computation cost, we also perform optimizations on pairs of nodes in addition to individual nodes. In this case, for each edge in G^x, we use the s best scoring features on the same type of edge in the training set and attempt to set the labels of the two elements connected by the edge to the values in each triple.

2.5 LEARNING

In this section we discuss our learning procedure for obtaining the weights $\boldsymbol{w}$ of the model $Pr(\boldsymbol{y} \mid x)$ from a data set. We assume that there is some underlying joint distribution $P(\boldsymbol{y}, x)$ from which the data set $D = \{\langle x^{(j)}, \boldsymbol{y}^{(j)} \rangle\}_{j=1}^{t}$ of t programs is drawn independently and identically distributed and from which new programs for which we want to infer program properties are drawn. In addition to programs $x^{(j)}$, we assume a given assignment of labels $\boldsymbol{y}^{(j)}$ (names or type annotations in our case) is provided as well. We perform discriminative training (i.e., estimate $Pr(\boldsymbol{y} \mid x)$ directly) rather than generative training (i.e., estimate $Pr(\boldsymbol{y}, x)$, and deriving $Pr(\boldsymbol{y} \mid x)$ from this joint model), since the latter requires estimating a distribution over programs x – a challenging, and for the purposes of predicting program properties, unnecessary task.

The goal of learning is to estimate the parameters $\boldsymbol{w}$ to achieve *generalization*: we wish that for a *new program* x drawn from the same distribution P – but generally *not* contained in the data set D – its properties $\boldsymbol{y}$ are predicted accurately (using the prediction algorithm from Section 2.4). Several approaches to accomplish this task exist and can potentially be used.

One approach is to fit parameters in order to maximize the (conditional) likelihood of the data, that is, try to choose weights such that the estimated model $Pr(\boldsymbol{y} \mid x)$ accurately fits the true conditional distribution $P(\boldsymbol{y} \mid x)$ associated with the data-generating distribution P. Unfortunately, this task requires computation of the partition function $Z(x)$ which is a formidable task [75].

Instead, we perform what is known as *max-margin training*: we learn weights $\boldsymbol{w}$ such that the training data is classified correctly subject to additional constraints like margin and regularization. For this task, powerful learning algorithms are available [127, 129]. In particular, we use a variant of the *Structured Support Vector Machine (SSVM)*[4] [129] and we train it efficiently with the scalable subgradient descent algorithm proposed in [102].

STRUCTURED SUPPORT VECTOR MACHINE The goal of SSVM learning is to find $\boldsymbol{w}$ such that for each training sample $\langle x^{(j)}, \boldsymbol{y}^{(j)} \rangle$ ($j \in [1, t]$), the assignment found by the classifier (i.e., maximizing the score) is equal to the given assignment $\boldsymbol{y}^{(j)}$, and there is a *margin* between the correct classification and any other classification:

$$\forall j, \forall \boldsymbol{y}' \in \Omega_{x^{(j)}} \;\; score(\boldsymbol{y}^{(j)}, x^{(j)}) \geq score(\boldsymbol{y}', x^{(j)}) \; + \; \Delta(\boldsymbol{y}^{(j)}, \boldsymbol{y}') \qquad (2.1)$$

Here, $\Omega_{x^{(j)}}$ is the set of all possible assignments of predicted program properties for $x^{(j)}$ and $\Delta\colon \mathit{Labels}^* \times \mathit{Labels}^* \to \mathbb{R}$ is a distance function (non-negative and satisfying triangle inequality). One can interpret $\Delta(\boldsymbol{y}^{(j)}, \boldsymbol{y}')$ as a (safety) margin between the given assignment $\boldsymbol{y}^{(j)}$ and any other assignment $\boldsymbol{y}'$, w.r.t. the score function. Δ is chosen such that slight mistakes (e.g., incorrect prediction of few properties) lead to less margin than major mistakes. For our applications, we took Δ to return the number of different labels between the reference assignment $\boldsymbol{y}^{(j)}$ and any other assignment $\boldsymbol{y}'$.

The weights $\boldsymbol{w}$ in (2.1) are not shown explicitly, but they are part of the *score* function. Recall that by definition $score(\boldsymbol{y}, x) = \boldsymbol{w}^T \boldsymbol{f}(\boldsymbol{y}, x)$.

Generally, it may not be possible to find weights achieving the above constraints. Hence, SSVMs attempt to find weights $\boldsymbol{w}$ that minimize the violation of the margin (i.e., maximize the goodness of fit to the data). At the same time, SSVMs control model complexity via *regularization*, penalizing the use of large weights. This is done to facilitate better generalization to unseen test programs.

4 Structured Support Vector Machines generalize classical Support Vector Machines to predict many interdependent labels at once, as necessary when analyzing programs.

2.5.1 *Learning with Stochastic Gradient Descent*

Achieving the balance of data-fit and model complexity leads to a natural optimization problem for the vector of weights $\boldsymbol{w}$:

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w} \in \mathcal{W}_\lambda} \sum_{j=1}^{t} \ell(\boldsymbol{w}; x^{(j)}, \boldsymbol{y}^{(j)}) \tag{2.2}$$

where

$$\ell(\boldsymbol{w}; x^{(j)}, \boldsymbol{y}^{(j)}) = \max_{\boldsymbol{y}' \in \Omega_{x^{(j)}}} \left(\boldsymbol{w}^T [\boldsymbol{f}(\boldsymbol{y}', x^{(j)}) - \boldsymbol{f}(\boldsymbol{y}^{(j)}, x^{(j)})] + \Delta(\boldsymbol{y}^{(j)}, \boldsymbol{y}') \right) \tag{2.3}$$

is called the *structured hinge loss*. This nonnegative loss function measures the violation of the margin constraints caused for the j-th program, when using a particular set of weights $\boldsymbol{w}$. Thus, if the objective (2.2) reaches zero, all margin constraints are respected, i.e., accurate labels are returned for all training programs. Furthermore, the set $\mathcal{W}_\lambda$ encodes some constraints on the weights in order to control model complexity and avoid overfitting. In our work, we regularize by requiring all weights to be nonnegative and bounded by $1/\lambda$, hence we set

$$\mathcal{W}_\lambda = \{\boldsymbol{w} : w_i \in [0, 1/\lambda] \text{ for all } i\}.$$

The SSVM optimization problem (2.2) is *convex* (since the structured hinge loss is a pointwise maximum of linear functions, and $\mathcal{W}_\lambda$ is convex), suggesting the use of gradient descent optimization. In particular, we use a technique called *projected stochastic gradient descent*, which is known to converge to an optimal solution, while being extremely scalable for structured prediction problems [102].

The algorithm proceeds iteratively; in each iteration, it picks a random program with index $j \in [1..t]$ from D, computes the gradient (w.r.t. $\boldsymbol{w}$) of the loss function $\ell(\boldsymbol{w}; x^{(j)}, \boldsymbol{y}^{(j)})$ and takes a step in the negative gradient direction. If it ends up outside the feasible region $\mathcal{W}_\lambda$, it projects $\boldsymbol{w}$ to the closest feasible point.

The gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \ell(\boldsymbol{w}; x^{(j)}, \boldsymbol{y}^{(j)})$ at $\boldsymbol{w}$ w.r.t. the j-th program is

$$\boldsymbol{g} \leftarrow \boldsymbol{f}(\boldsymbol{y}_{best}, x^{(j)}) - \boldsymbol{f}(\boldsymbol{y}^{(j)}, x^{(j)})$$

where $\boldsymbol{y}_{best} \in \Omega_{x^{(j)}}$ maximizes the value inside the max term of equation (2.3). Thus

$$\boldsymbol{y}_{best} \leftarrow \arg\max_{\boldsymbol{y}' \in \Omega_{x^{(j)}}} \left(\, score(\boldsymbol{y}', x^{(j)}) + \Delta(\boldsymbol{y}^{(j)}, \boldsymbol{y}') \, \right), \tag{2.4}$$

Hence, computing the gradient requires solving the *loss-augmented inference* problem (2.4). This problem can be (approximately) solved using the algorithm presented in Section 2.4 by modifying the formula *scoreEdges* there to also include the Δ term.

After finishing the gradient computation, the weights $\boldsymbol{w}$ (used by *score*) are updated as follows:

$$\boldsymbol{w} \leftarrow \mathrm{Proj}_{\mathcal{W}_\lambda}(\boldsymbol{w} - \alpha \boldsymbol{g})$$

where α is a learning rate constant and $\mathrm{Proj}_{\mathcal{W}_\lambda}$ is a projection operation determined by the regularization described below.

2.5.2 *Regularization*

The function $\mathrm{Proj}_{\mathcal{W}_\lambda} : \mathbb{R}^k \to \mathbb{R}^k$ projects its arguments to the point in $\mathcal{W}_\lambda$ that is closest in terms of Euclidean distance. This operation is used to place restrictions on the weights $\boldsymbol{w} \in \mathbb{R}^k$ such as non-negativity and boundedness. The goal of this procedure, known as *regularization*, is to avoid a problem known as *overfitting* – a case where $\boldsymbol{w}$ is good in predicting training data, but fails to generalize to unseen data. In our case, we perform ℓ^{inf} regularization, which can be efficiently done in closed form as follows:

$$\mathrm{Proj}_{\mathcal{W}_\lambda}(\boldsymbol{w}) = \boldsymbol{w}' \quad \text{such that} \quad w'_i = \max(0, \min(1/\lambda, w_i))$$

This projection ensures that the learned weights are non-negative and never exceed a value $1/\lambda$, limiting the ability to learn too strong feature functions that may not generalize to unseen data. Our choice of ℓ^{inf} has the additional benefit that it operates on vector components independently. This allows for efficient implementation of the learning where we regularize only vector components that changed or components for which the gradient $\boldsymbol{g}$ is non-zero. This enables us to use a sparse representation of the vectors $\boldsymbol{g}$, avoiding iteration over all components of the vector $\boldsymbol{w}$ when projecting.

2.5.3 *Complete Training Phase*

In summary, our training procedure first iterates once over the training data and selects a set of pairwise feature functions (as described in Section 2.3.4) used to compute $\boldsymbol{f}$. We initialize the weight of each feature function with $w_i = 1/(2\lambda)$. Then, we start with a learning rate of $\alpha = 0.1$ and iterate in multiple passes to learn the weights with stochastic gradient descent. In each pass, we compute gradients via inference, and apply regularization as described before. Additionally, we count the number of wrong labels in the pass and compare it to the number of mispredicted labels in the previous pass. If we do not observe improvement, we decrease the learning rate α by one half. In our implementation, we iterate over the data up to 24 times.

To speed up the training phase, we also parallelized the stochastic gradient descent on multiple threads as described in [141]. At each pass, we randomly split the data to d threads where each thread t_i updates its own version of the weights $\boldsymbol{w}^{t_i}$. At the end of each pass, the final weights $\boldsymbol{w}$ is set to the average of the weights $\{\boldsymbol{w}^{t_i}\}_{i=1}^{d}$.

2.5.4 *Example of Learning Weights for Type Annotations*

Next, we give a geometric interpretation of the learning algorithm with an example on learning type annotation rules. The idea is to give an intuition for the previously described learning procedure. The following example shows predictions for a single program property that is the type of a variable. The case for multiple properties and structured prediction has a similar geometric interpretation.

2.5.4.1 *Multi-class SVM example*

Consider the following JavaScript program `i = 5` that assign a constant value of type `number` to the variable `i`. Let this program be part of our training data. For demonstration purposes, assume that using existing rule-based type inference approaches can infer the type of the constant `5` to be `number`, but the type of `i` cannot be precisely inferred. A probabilistic model will learn rules that predict the type of `i`. We show the example in Fig. 2.6.

In Fig. 2.6 (b) we define a dependency network where the type of `i` is unknown and two indicator features with weights w_1 and w_2 predict the type to be either `number` or `string` respectively. These weights w_1 and w_2 are learned from the training data. The training data is manually labeled and includes the correct type annotation `number` for i. Ideally, the learned weights after training will be such that for the program `i = 5`, the predicted type for the variable `i` will be `number`.

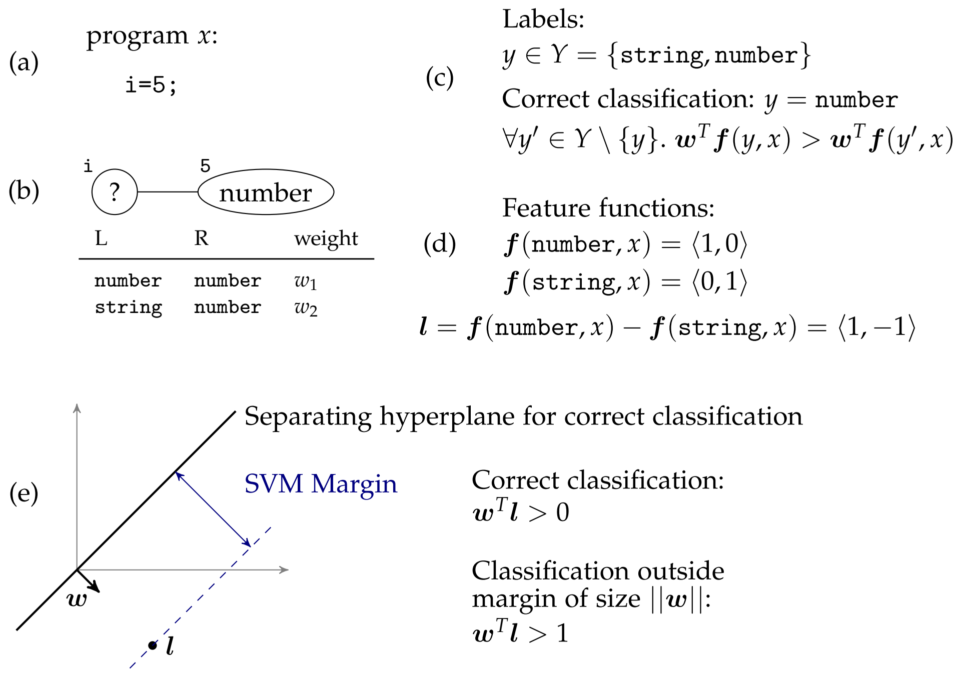

(a)
program x:
i=5;
(b)
i
5
?
number
L
R
weight
number number w_1
string number w_2
(c)
Labels:
$y \in Y = \{\texttt{string}, \texttt{number}\}$
Correct classification: $y = \texttt{number}$
$\forall y' \in Y \setminus \{y\}.\ \boldsymbol{w}^T \boldsymbol{f}(y, x) > \boldsymbol{w}^T \boldsymbol{f}(y', x)$
(d)
Feature functions:
$\boldsymbol{f}(\texttt{number}, x) = \langle 1, 0 \rangle$
$\boldsymbol{f}(\texttt{string}, x) = \langle 0, 1 \rangle$
$\boldsymbol{l} = \boldsymbol{f}(\texttt{number}, x) - \boldsymbol{f}(\texttt{string}, x) = \langle 1, -1 \rangle$
(e)
Separating hyperplane for correct classification
SVM Margin
$\boldsymbol{w}$
$\boldsymbol{l}$
Correct classification:
$\boldsymbol{w}^T \boldsymbol{l} > 0$
Classification outside margin of size $||\boldsymbol{w}||$:
$\boldsymbol{w}^T \boldsymbol{l} > 1$

Figure 2.6: Example of structured/multi-class SVM on one training sample. (a) Training data program x. (b) Formulation of a type prediction problem for x as a dependency graph. (c) The linear classification task that says that the correct label scores better than any other label. (d) Feature functions for x. (e) Visualization of finding weights $\boldsymbol{w} = \{w_1, w_2\}$ that make correct classification and also have maximize margin like in SVM. Note that correct classification here means to be on the right size of the hyperplane defined by $\boldsymbol{w}$.

This translates to the requirement that assigning $y = $ `number` produces higher score than assigning any other label that is not `number`. i.e. $\forall y' \in Y \setminus \{y\}.\ \boldsymbol{w}^T \boldsymbol{f}(y,x) > \boldsymbol{w}^T \boldsymbol{f}(y',x)$ as shown in Fig. 2.6 (c).

Traditionally, linear classifiers such as support vector machines are described in terms of finding a separating hyperplane between positive and negative training data samples and in the context of two-class (binary) classification [21, §4.1.1]. While this view gives an intuition of the underlying algorithms, it does not directly generalize to multiple classes or structured prediction like in our setting. Trying to compose multiple binary classifiers by multiple lines separating labels one-versus-one or one-versus-the-rest leads to ambiguity in the classification [21, §4.1.2] so we handle multiple classes differently. Instead, we use one linear classifier that finds the class with the highest score as shown in Fig. 2.6 (c).

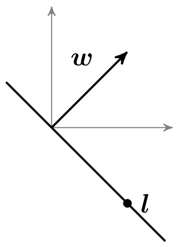
To preform multi-class classification, the vector of feature functions $\boldsymbol{f}$ is dependent not only on the input program x, but on the predicted label y. In Fig. 2.6 (b) we show the values of the features in a tabular form. This table corresponds to the feature function $\boldsymbol{f}$ described in Fig. 2.6 (d). If the predicted label y is `number`, then f_1 is 1 and f_2 is 0. If the predicted label is `string`, f_1 is zero and f_2 – one. Now consider the difference in features between the correct label $y = $ `number` and another label $y' = $ `string`. This is recorded by the point $\boldsymbol{l}$. Our goal to have the correct label score better than the label `string` is expressed in $\boldsymbol{w}^T \boldsymbol{l} > 0$. This is shown in Fig. 2.6 (e) where the point l is on the right side of the hyperplane $\boldsymbol{w}$ going through the center of the coordinate system.

Finally, once the requirements for correct classification are established, a margin is introduced. This is, from all possible hyperplanes for which $\boldsymbol{l}$ is on the right size of the hyperplane described by $\boldsymbol{w}$, the parameters are chosen to maximize the distance between $\boldsymbol{l}$ and the hyperplane. The margin is shown in Fig. 2.6 (e).

2.5.4.2 *Stochastic gradient descent*

Next, we visualize the technique to find the optimal parameters $\boldsymbol{w}$. Fig. 2.7 shows two iterations of running the stochastic gradient descent algorithm starting with initial weights $\boldsymbol{w} = \langle 1, 1 \rangle$, updating weights with learning rate $\alpha = 0.5$ and using regularization constant $\lambda = 1$. At each step of the algorithm, the given correct label $y = $ `number` is compared to the label y' that maximizes the measure $\boldsymbol{w}^T \boldsymbol{f}(y',x) +$

(a) First iteration. Initial weights $\boldsymbol{w} = \langle 1, 1\rangle$. Then from (2.4):



$$y_{best} \leftarrow \arg\max_{y' \in Y} \quad \boldsymbol{w}^T \boldsymbol{f}(y', x) + \Delta(y, y')$$

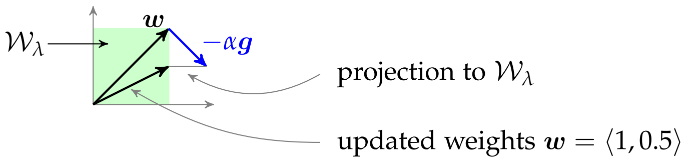
$$\begin{aligned} &\text{for } y' = \texttt{number:} && \langle 1,1\rangle^T \langle 1,0\rangle + 0 = 1 \\ &\text{for } y' = \texttt{string:} && \langle 1,1\rangle^T \langle 0,1\rangle + 0 = 2 \end{aligned}$$

$$\begin{aligned} &\Rightarrow \; y_{best} = \texttt{string} \\ &\Rightarrow \; \boldsymbol{g} = \boldsymbol{f}(y_{best}, x^{(j)}) - \boldsymbol{f}(y, x^{(j)}) = \\ &\qquad = \boldsymbol{f}(\texttt{string}, x) - \boldsymbol{f}(\texttt{number}, x) = \langle -1, 1\rangle \end{aligned}$$

Perform gradient descent with $\alpha = 0.5$:

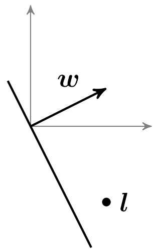
$\boldsymbol{w} \leftarrow \text{Proj}_{\mathcal{W}_\lambda}(\boldsymbol{w} - \alpha \boldsymbol{g}) = \text{Proj}_{\mathcal{W}_\lambda}(\langle 1,1\rangle - 0.5\langle -1,1\rangle) = \text{Proj}_{\mathcal{W}_\lambda}(\langle 1.5, 0.5\rangle)$

Projection for $\lambda = 1$ is:



$\mathcal{W}_\lambda$
$\boldsymbol{w}$
$-\alpha\boldsymbol{g}$
projection to $\mathcal{W}_\lambda$
updated weights $\boldsymbol{w} = \langle 1, 0.5\rangle$

(b) Second iteration:



$$y_{best} \leftarrow \arg\max_{y' \in Y} \quad \boldsymbol{w}^T \boldsymbol{f}(y', x) + \Delta(y, y')$$

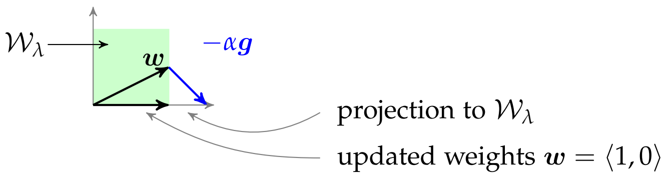
$$\begin{aligned} &\text{for } y' = \texttt{number:} && \langle 1,0.5\rangle^T \langle 1,0\rangle + 0 = 1.0 \\ &\text{for } y' = \texttt{string:} && \langle 1,0.5\rangle^T \langle 0,1\rangle + 0 = 1.5 \end{aligned}$$

$$\begin{aligned} &\Rightarrow \; y_{best} = \texttt{string} \\ &\Rightarrow \; \boldsymbol{g} = \boldsymbol{f}(y_{best}, x^{(j)}) - \boldsymbol{f}(y, x^{(j)}) = \\ &\qquad = \boldsymbol{f}(\texttt{string}, x) - \boldsymbol{f}(\texttt{number}, x) = \langle -1, 1\rangle \end{aligned}$$

Perform gradient descent with $\alpha = 0.5$:

$\boldsymbol{w} \leftarrow \text{Proj}_{\mathcal{W}_\lambda}(\boldsymbol{w} - \alpha \boldsymbol{g}) = \text{Proj}_{\mathcal{W}_\lambda}(\langle 1,0.5\rangle - 0.5\langle -1,1\rangle) = \text{Proj}_{\mathcal{W}_\lambda}(\langle 1.5, 0\rangle)$

Projection for $\lambda = 1$ is:



$\mathcal{W}_\lambda$
$-\alpha\boldsymbol{g}$
$\boldsymbol{w}$
projection to $\mathcal{W}_\lambda$
updated weights $\boldsymbol{w} = \langle 1, 0\rangle$

Figure 2.7: Two iterations of stochastic gradient descent steps done in SSVM learning on the example from Fig. 2.6.

$\Delta(y, y')$ as in formula (2.4). The first term of this measure checks if the classification would score y' higher than y and the second term introduces a margin by giving score of 1 to any label $y' \neq y$ and score 0 otherwise.

Consider the first iteration of the algorithm in Fig. 2.7 (a). The label $y =$ `number` does not score better than the label $y' =$ `string` and as a result $y_{best} =$ `string`. Then, a gradient $\boldsymbol{g}$ is computed to update the weights $\boldsymbol{w}$. Following the update of $\boldsymbol{w}$, the point $\boldsymbol{l}$ is on the right size of hyperplane described by $\boldsymbol{w} = \langle 1, 0.5 \rangle$ as shown in Fig. 2.7 (b).

The vector $\boldsymbol{w} = \langle 1, 0.5 \rangle$ already classifies $y =$ `number` correctly for our training data sample. This is, as shown in Fig. 2.6 (c), for any $y' \in Y \setminus \{y\}$. $\boldsymbol{w}^T \boldsymbol{f}(y, x) > \boldsymbol{w}^T \boldsymbol{f}(y', x)$. However, the hyperplane described by $\boldsymbol{w}$ in Fig. 2.7 (b) does not maximize the margin to $\boldsymbol{l}$. In this case, note that the term $\Delta(y, y')$ contributes in setting $y_{best} =$ `string`. Due to this, the second iteration of the algorithm still updates the hyperplane $\boldsymbol{w}$ to be further away from $\boldsymbol{l}$. Any further iterations after the second one do not update the weights $\boldsymbol{w}$.

At each step, regularization is performed by projecting the update of the vector $\boldsymbol{w}$ into the area $\mathcal{W}_\lambda$. Our regularization disallows negative weights and limits their magnitude to 1. As a result, the final learned weights are $\boldsymbol{w} = \langle 1, 0 \rangle$ which describe the hyperplane with the widest possible margin to $\boldsymbol{l}$ subject to the regularization constraints. Finally, the obtained model learns that the predicted type annotation for `i` in code like `i = 5` should be `number` with weight 1 and `string` with weight 0. The learned weights correspond to the following learned type inference rules:

- assigning a value of type `number` to a variable `i` sets the type of `i` to `number` with a very high confidence of 1.
- assigning a value of type `number` to a variable `i` sets the type of `i` to `string` with a very low confidence of 0.

2.6 IMPLEMENTATION AND EVALUATION

We implemented our approach in an end-to-end production quality interactive tool, called JSNice, which targets name and type annotation prediction for JavaScript. JSNice is a modification of the Google Closure Compiler project [47]. In standard operation, Google Closure Compiler takes human-readable JavaScript with optional type annotations and typechecks it. It then returns an optimized, minified and human-unreadable JavaScript with stripped annotations.

In our system, we added a new mode to the compiler that aims to reverse its operation: given an optimized minified JavaScript code, JSNice generates JavaScript code that is well annotated (with types) and as human-readable as possible (with useful identifier names). Our two applications for names and types were implemented as two models that can be run separately.

JSNICE: IMPACT ON DEVELOPERS A week after JSNice was made publicly available, it was used by more than 100, 000 developers, with the vast majority of feedback left in blogs and tweets being very positive (those can be found by a simple web search). We believe the combination of high speed and high precision achieved by the structured prediction approach were the main reasons for this positive reception.

EXPERIMENTAL EVALUATION We next present a detailed experimental evaluation of our statistical approach and demonstrate that the approach can be successfully applied to the two prediction tasks we described. Further, we evaluate how various knobs of our system affect the overall performance and precision of the predictions.

We collected two disjoint sets of JavaScript programs to form our training and evaluation data. For training, we downloaded 10, 517 JavaScript projects from GitHub [46]. For evaluation, we took the 50 JavaScript projects with the highest number of commits from BitBucket [22]. By taking projects from different repositories, we decrease the likelihood of overlap between training and evaluation data. We also searched in GitHub to check that the projects in the evaluation data are not included in the training data. Finally, we implemented a simple checker to detect and filter out minified and obfuscated files from the training and the evaluation data. After filtering minified files, we ended up with training data consisting of 324, 501 files and evaluation data of 2, 710 files. Next, we discuss how we trained and evaluated our system: first, we discuss parameter selection (Section 2.6.1), then precision (Section 2.6.2) and model sizes (Section 2.6.3), and finally the running times (Section 2.6.4).

2.6.1 *Parameter Selection*

We used 10-fold cross-validation to select the best learning parameters of the system only based on the training data and not biased by any test set [93]. Cross-validation works by splitting the training data into 10 equal pieces called folds and evaluating the ratio of mispredicted program properties on each fold by training a model on the data in the other 9 folds. Then, we trained and evaluated on a set of different training parameters and selected the parameters with the lowest number of mispredictions.

We tuned the values of two parameters that affect the learning: regularization constant λ, and presence of margin. Higher values of λ mean that we regularize more, i.e. add more restrictions on the feature weights by limiting their maximal value to a lower value $1/\lambda$. The margin parameter determines if the margin function Δ (see Section 2.5) should return zero or return the number of different labels between the two assignments. To reduce computation time (since we must train and test a large number of values for the parameters), we performed cross-validation on only 1% sample of the training data. Using subsample here means that we bias towards more regularization than necessary and overfit less to the data [84]. As a result, the reported precision may not be the highest possible. This cross-validation procedure determined that the best value for λ is 2.0 for names, 5.0 for types, and margin Δ should be applied to both tasks.

2.6.2 *Precision*

After choosing the parameters, we evaluated the precision of our system for predicting names and type annotations. Our experiments were performed by predicting the names and types in isolation on each of the 2,710 testing files. To evaluate precision, we first minified all 2,710 files with UglifyJS [5], but any other sound minifier should produce input that is equivalent for the purposes of using JSNice to reconstruct variable names and type annotations. The minification process renames local variable identifiers to meaningless short names and removes whitespaces and type annotations. Each minified program is semantically equivalent (except when using `with` or `eval`) to the original program.

5 https://github.com/mishoo/UglifyJS

Then, we used JSNice on the minified programs to evaluate its capabilities to precisely reconstruct name and type information. We compared the precision of the following configurations:

- The most powerful system works with all of the training data and performs structured prediction as described so far.
- Two systems using a fraction of the training data – one on 10% and one on 1% of the files.
- To evaluate the effect of structure when making predictions, we disabled relationships between unknown properties and performed predictions on that network (the training phase still uses structure).
- A naïve baseline which does no prediction: it keeps names the same and sets all types to the most common type `string`.

2.6.2.1 *Name predictions*

To evaluate the accuracy of name predictions, we took each of the minified programs and used the name inference in JSNice to rename its local variables. Then, we compared the new names to the original names (before obfuscation) for each of the tested programs. The results for the name reconstruction are summarized in the second column of Table 2.1. Overall, our best system produces code with 63.4% of identifier names exactly equal to their original names. The systems trained on less data have significantly lower precision showing the importance of the amount of training data.

Not using structured prediction also drops the accuracy significantly and has about the same effect as an order of magnitude less data. Finally, not changing any identifier names produces accuracy of 25.3% – this is because minifying the code may not rename some variables (e.g. global variables) in order to guarantee semantic preserving transformations and occasionally one-letter local variable names stay the same (e.g. induction variable of a loop).

2.6.2.2 *Type annotation predictions*

Out of the 2,710 test programs, 396 have type annotations for functions in a JSDoc. For these 396, we took the minified version with no type

System	Names Accuracy	Types Precision	Types Recall
all training data	**63.4%**	**81.6%**	**66.9%**
10% of training data	54.5%	81.4%	64.8%
1% of training data	41.2%	77.9%	62.8%
all data, no structure	54.1%	84.0%	56.0%
baseline - no predictions	25.3%	37.8%	100%

Table 2.1: Precision and recall for name and type reconstruction of minified JavaScript programs evaluated on our test set.

annotations and tried to rediscover all types in the function signatures. We first ran the closure compiler type inference, which produces no types for the function parameters. Then, we ran and evaluated JSNice on inferring these function parameter types.

JSNice does not always produce a type for each function parameter. For example, if a function has an empty body, or a parameter is not used, we often cannot relate the parameter to any known program properties and as a result, we make no prediction and return the unknown type (?). To take this effect into account, we do not report accuracy like for names, but compute recall and precision. Recall is the percentage of function parameters in the evaluation for which JSNice made a prediction other than ?. Precision refers to the percentage of cases – among the ones for which JSNice made a prediction – where it was exactly equal to the manually provided JSDoc annotation of the test programs. We note that the manual annotations are not always correct, and as a result 100% precision is not necessarily a desired outcome.

We present our evaluation results for types in the last two columns of Table 2.1. Since we evaluate on production JavaScript applications that typically have short methods with complex relationships, the recall for predicting program types is only 66.9% for our best system. However, we note that none of the types we infer are inferred by state-of-the-art forward type analysis (e.g. Facebook Flow [6]).

6 https://github.com/facebook/flow

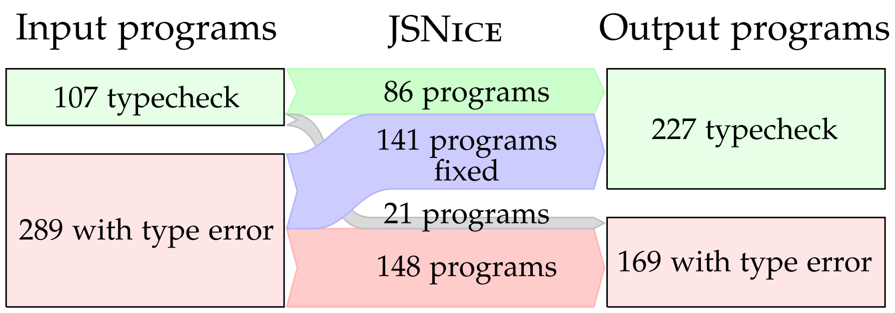


Figure 2.8: Evaluation results for the number of typechecking programs with manually provided types and with predicted types.

Since the total number of commonly used types is not as high as the number of names, the amount of training data has less impact on the system precision and recall. To further increase the precision and recall of type prediction, we hypothesize that adding more (semantic) relationships between program elements would be of higher importance than adding more training data. Dropping structure increases the precision of the predicted types slightly, but at the cost of a significantly reduced recall. The reason is that some types are related to known properties only transitively via other predicted types – relationships that non-structured approaches cannot capture. On the other end of the spectrum is a prediction system that suggests the most likely type in JavaScript programs – `string`. Such a system produces a type for every variable (100% recall), but its precision is only 37.8%.

USEFULNESS OF TYPE ANNOTATIONS To see if the predicted type annotations are useful, we compared them to the original types provided in the evaluated programs. First, we note that our evaluation data has 3,505 type annotations for function parameters in 396 programs. After removing these annotations and reconstructing them with JSNice, the number of annotations that are not `?` increased to 4,114 for the same programs. The reason JSNice produces more types than originally present despite having only 66.3% recall is that not all functions in the original programs had manually provided type annotations.

Despite annotating more functions than in the original code, the output of JSNice has fewer type errors. We summarize these findings in Fig. 2.8. For each of the 396 programs, we ran the typechecking pass of Google's Closure Compiler to discover type errors. Among others, this pass checks for incompatible types, calling into a non-function, conflicting and missing types, and non-existent properties on objects. For our evaluation, we kept all checks except the inexistent property check, which fails on almost all (even valid) programs, because it depends on annotating all properties of the JavaScript classes – annotations that almost no program in the training or evaluation data possesses.

When we ran typechecking on the input programs, we found the majority (289) to have typechecking errors. While surprising, this can be explained by the fact that JavaScript developers typically do not typecheck their annotations. Among others, we found the original code to have misspelled type names. Most typecheck errors occur due to missing or conflicting types. In a number of cases, the types provided were interesting for documentation, but were semantically wrong - e.g. a parameter is a `string` that denotes function *name*, but the manual annotation designates its type to be `Function`. In contrast, the types reconstructed by JSNice make the majority (227) of the programs typecheck. In 141 of the programs that originally did not typecheck, JSNice was able to infer correct types. On the other hand, JSNice introduced type errors in 21 programs. We investigated some of these errors and found that not all of them were due to wrong types – in several cases the predicted types were rejected due to inability of the type system to precisely express the desired program properties without also manually providing type cast annotations.

2.6.3 *Model Sizes*

Our models contain 7,627,484 features for names and 70,052 features for types. Each feature is stored as a triple, along with its weight. As a result we need only 20 bytes per feature, resulting in a 145.5MB model for names and 1.3MB model for types. The dictionary which stores all names and types requires 16.8MB. As we do not data compress our model, the memory requirements for query processing are about as much as the model size.

2.6.4 *Running Times*

We performed our performance evaluation on a 32-core machine with four 2.13GHz Xeon processors and running Ubuntu 12.04 with 64-Bit

OpenJDK Java 1.7.0_51. The training phase for name prediction took around 10 hours: 57 minutes to compile the input code and generate networks for the input programs and 23 minutes per SSVM (sub-) gradient descent optimization pass. Similarly for types, the compilation and network construction phase took 57 minutes and then we needed 2 minutes and 16 seconds per SSVM (sub-)gradient descent optimization pass. For all our training, we ran 24 gradient descent passes on the training data. All the training passes used 32 threads to utilize the cores of our machine.

RUNNING TIMES OF PREDICTION We evaluated the effect of changing the beam size parameter s of our MAP inference algorithm (from Section 2.4), and the effect s has on the prediction time. The average prediction times per program are summarized in Table 2.2. Each query is performed on a single core of our test machine. As expected, increasing s improves prediction accuracy but requires more time. Removing the beam altogether and running naïve greedy iterated conditional modes [16] leads to running times of around two minutes per program for name prediction, unacceptable for an interactive tool such as JSNice. Also, its precision trails some of the beam-based systems, because it does not perform optimization per pair of nodes, but only a node at a time. Due to the requirements for high performance, in our main evaluation and for our live server at `http://jsnice.org/`, we chose the value $s = 64$. This value provides a good balance between performance and precision suitable for an interactive system.

EVALUATION DATA METRICS Our evaluation data consists of $381,243$ lines of JavaScript code with the largest file being $3,055$ lines. For each of the evaluated files, the constructed dependency network for name prediction has on average 383.5 arcs and 29.2 random variables. For the type prediction evaluation tasks, each network has on average 109.5 arcs and 12.6 random variables.

2.7 LESSONS AND DESIGN DECISIONS

The problem of effectively learning from existing code and precisely answering interesting questions on new programs is non-trivial and requires careful interfacing between programs and sophisticated probabilistic models. As the overall machine can be fairly complex, here we

Beam size parameter	**Name prediction**		**Type prediction**	
b	Accuracy	Time	Precision	Time
4	57.9%	43ms	80.6%	36ms
8	59.2%	60ms	80.9%	39ms
16	62.8%	62ms	81.6%	33ms
32	63.2%	80ms	81.3%	37ms
64 **(JSNice)**	63.4%	114ms	81.6%	40ms
128	63.5%	175ms	82.0%	42ms
256	63.5%	275ms	81.6%	50ms
Naïve greedy, no beam	62.8%	115.2 s	81.7%	410ms

Table 2.2: Trade-off between precision and runtime for the name and type predictions depending on beam search parameter s.

state the important design choices that we made in order to arrive at the solution presented here.

FROM PROGRAMS TO RANDOM VARIABLES When predicting program properties, one needs to account for both, the program elements whose properties are to be predicted and the set of available properties from which we draw predictions. In JSNICE, the elements are local variables (for name prediction) and function parameters (for type prediction). In general however, one could instantiate our approach with any program element that ranges over program properties for which sufficient amount of training data is available.

We note that for our instantiation, JSNICE, a predicted program property always exists in the training data. In the case of names, large amounts of training data still allow us to predict meaningful and useful names. Because of the assumption that the property exists in the training data, we create one random variable per local variable of a program with the name to predict and the feature functions as described in Section 2.3.4. However, the framework from Section 2.2 can be instantiated (with different feature functions) to cases where a predicted variable name does not exist in the training data (e.g. is a concatenation of several existing words).

THE NEED FOR STRUCTURE When predicting facts and properties of programs, it is important to observe that these properties are usually *dependent* on one another. This means that any predictions of these properties should be done *jointly* and not independently in isolation.

GRAPHICAL MODELS: UNDIRECTED OVER DIRECTED A family of probabilistic models able to capture complex structural dependencies are graphical models such as Bayesian networks (directed models) and Markov networks (undirected models) [75]. In graphical models, nodes represent random variables and edges capture dependencies between these random variables. Therefore, graphical models are a natural fit for our problem domain where program elements are random variables and the edges capture a particular dependency between the properties to be inferred for these program elements. While we do need to capture dependence between two (or more) random variables, it is often not possible to decide a priori on the exact order (direction of the edges) of that dependence. In turn, this means that undirected models are a better match for our setting as they do not require specifying a direction of the dependence.

INFERENCE: MAP INFERENCE OVER MARGINALS For a given program x and a probabilistic model P, we are fundamentally interested in finding the most likely properties which should be assigned to program elements V_U^x *given* the known, fixed values for the elements V_K^x. What is the right query for computing this most likely assignment? Should we try to find the value v of each random variable $r_i \in V_U^x$ which maximizes its marginal probability $P(r_i = v)$ separately, or should we try to find the values $v_0, v_1, \ldots, v_n$ for *all* random variables $r_0, \ldots, r_n \in V_U^x$ *together* so that the joint probability is maximized, that is, $P(r_0 = v_0, r_1 = v_1, \ldots, r_n = v_n)$ (called MAP inference)?

We decided to use MAP inference over marginals for several reasons. First, we ultimately aim to find the best *joint* assignment and not make independent, potentially contradicting predictions. Second, it is easy to show that maximizing the marginal probability of each variable separately *does not* lead to the optimum assignment computed by MAP inference. Third, when computing marginals, it is difficult to enforce deterministic constraints such as A $\neq$ B. It is often easier to incorporate such constraints with MAP inference. Finally, and as we discuss below, the decision to perform MAP inference over marginals enjoys a substantial benefit when it comes to training the corresponding probabilistic model.

An interesting point is that standard MAP inference algorithms are computationally expensive when the number of possible properties is large. Hence, we had to develop new algorithmic variants which can effectively deal with thousands of possible properties for a given random variable (e.g. many possible names discussed in Section 2.4.1).

DISCRIMINATIVE OVER GENERATIVE TRAINING An important question when dealing with probabilistic models is deciding how the model should be trained. One approach is to train an undirected model in a generative way, thus obtaining a *joint* probability distribution over *both* V^x_K and V^x_U (this model is sometimes referred to as Markov Random Field). While possible in theory, this has a serious practical disadvantage: it requires placing prior probabilities on the known variables V^x_K, which can be very difficult in practice. For our applications this would mean providing probability estimates not only for the names and type annotations that we predict, but also for the facts we condition on such as API names, numeric and string constants, etc.

However, recall that our MAP inference query is in fact *conditional* on an existing assignment of the known elements V^x_K. This means that the underlying probabilistic model must only capture the *conditional* probability distribution of V^x_U *given* V^x_K and *not* the joint distribution. An undirected graphical model able to capture such conditional distributions is referred to as a Conditional Random Field (CRF) [79]. The CRF model admits *discriminative* training where priors on V^x_K are no longer necessary, a key reason for why this model is effective and popular in practice.

TRAINING WITH MAX-MARGIN OVER MAXIMUM LIKELIHOOD After deciding to use CRFs, we still need to find a scalable method for training and obtaining such CRF models from available data (e.g. "Big Code" in our setting). Training these models (say via maximum likelihood) is possible but also requires estimating marginal probabilities and this can be computationally expensive (see Ch.20, [75]).

Recall that we are mainly interested in MAP inference queries where we do not need the exact probability value for the predicted assignment of V^x_U. Because of that, we are now able to leverage recent advances in scalable training methods for CRFs and in particular max-margin training [102, 127], a method geared towards answering MAP inference queries on the trained CRF model.

CHOICE OF MODEL: SUMMARY In summary, based on the above reasoning, we arrive at using MAP inference queries with Conditional Random Fields (CRFs), a probabilistic, undirected graphical model which we learn from the available data via efficient max-margin training. We do note however that the translation of programs to networks and the feature functions we provide are directly reusable if one is interested in performing marginal queries and quantifying the uncertainty associated with the solutions.

2.7.1 *Clustering vs. Probabilistic Models*

It is instructive to understand that our approach is fundamentally *not* based on clustering. Given a program x, we do not try to find a similar (for some definition of similarity) program s in the training corpus and then extract useful information from s and integrate that information into x. Such an approach would be limiting as often there is not even a single program in the training corpus which contains all of the information we need to predict for program x. In contrast, with the approach presented here, it is possible to build a probabilistic model from multiple programs and then use that information to predict properties about a single program x.

2.8 RELATED WORK

Several works have used graphical models in the context of programs [13, 54, 77, 78, 82]. All of these works phrase the prediction problem as computing marginals. As we already discussed in Section 2.7, we find MAP inference to be the conceptually preferred problem formulation over marginals. Except for [77], none of these works *learn* the probabilistic models from data. If one is to pursue learning in their context, then this would essentially require new features which keep information common among programs (need some form of "anchors"). Further, because they frame the problem as computing marginal probabilities, these approaches do not allow for learning with loss functions, meaning that one has to model probabilities and compute the partition function at learning time. This would be prohibitively expensive for large datasets (such as ours) and does not allow for leveraging advanced methods for MAP inference based on structured prediction (where one need not compute the partition function). Indeed, the learning method of [77] is extremely inefficient and suffers from the need to compute expectations w.r.t to the model which is generally very expensive and requires sampling, a less scalable procedure than MAP inference.

In terms of particular applications, [78] aims to infer ownership values for pointers which range over a very small domain of 4 values: `ro`, `co`, and their negation. In contrast, we efficiently handle variables with thousands of possible assignments (for names) and even for types, the size of our domain is much greater. Further, their selection of feature functions makes the inference process extremely slow. The reason is that one of the features (called the check factor, critical to the precision of their approach) requires running program analysis essentially on every iteration of the inference: this is impractical in an interactive setting (even with optimizations). Indeed, their follow-up work (section 3, [77]) mentions that the authors would like to find a better solution to this problem (unfortunately, pre-computing this feature is also practically infeasible due to the large numbers of possible combinations of values/variables). Similarly, [82] focuses on inferring likely tags for String methods (e.g. source, sink, regular, sanitizer), where values range over a small domain. The basic idea is to convert a graph extracted from a program (called the propagation graph) into a factor graph and to then perform marginal inference on that graph. This work does a little more than computing marginals: it selects the best marginals and conditions on them for re-computation of the rest, meaning that it can potentially encode constraints by conditioning on what is already computed. This approach is computationally inefficient: it essentially requires running inference N times for N variables, instead of once. Further, the approach cannot compute optimal MAP inference. The paper of Beckman et al.[13] is similar to [82] in the sense that it infers permission annotations again ranging over a small set of values and both build factor graphs from programs. However, the inference algorithm in their paper just computes marginals directly and is simpler than the one in [82] (as it does not even iterate).

Overall, we believe that the problems these works address may benefit from considering MAP inference instead of computing marginals. Also, it seems that even with computing marginals, these works are not using state of the art inference methods (e.g. variational methods instead of the basic sum-product belief propagation which has no guarantees on convergence).

2.9 DISCUSSION

In this chapter, we presented a new statistical approach for predicting program properties by learning from massive codebases (aka "Big Code"). The core idea is to formulate the problem of property inference as structured prediction with conditional random fields (CRFs), enabling joint predictions of program properties. As an example of our approach, we built a system called JSNice which is able to predict names and type annotations for JavaScript. The JSNice tool became popular in the JavaScript community.

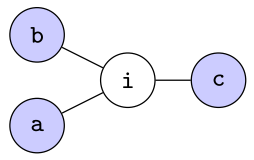
MAKING THE CRF MODEL TRACTABLE The model we presented in this chapter is a variant of a log-linear CRF that assigns probabilities proportional to a score that is a weighted sum of feature functions $score(y, x) = \boldsymbol{w}^T \boldsymbol{f}(y, x)$. However, the CRF model is too general, and thus, to ensure practical applicability, it has to be restricted in some way. For example, solving MAP inference (i.e. computing $\arg\max_y score(y, x)$) may be undecidable if there are no restrictions on the predictions y or the feature functions $\boldsymbol{f}$ (e.g. maximizing a feature function f_i may require satisfying predicates from undecidable theories [24]). In order to transition from an undecidable problem to a tractable solution, in this chapter, we imposed a number of additional restrictions:

1. We fixed the predictions y to be a vector of a given size $n(x)$ known before any predictions are made (in Section 2.2).

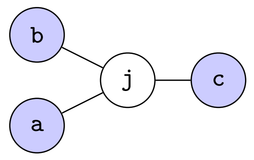
2. We introduced feature functions that relate only *pairs* of program properties and then defined $\boldsymbol{f}$ over them (in Section 2.2.2).

3. We restricted the pairwise feature function to be (mostly) indicator functions in order to enable fast inference (in Section 2.3.4).

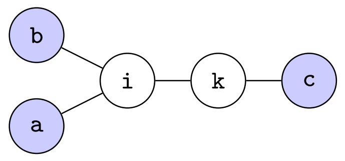
Relaxing these restrictions is non-trivial and typically comes at significant computational cost. For example, with factor graphs [75], restriction 2 can be slightly relaxed to feature functions that relate tuples



(a) Initial configuration



(b) A possible candidate configuration in the search space



(c) Configuration outside of the search space

Figure 2.9: Illustration of configurations that are inside or outside the search space of a MAP inference procedure starting from an initial configuration (a).

of size n as opposed to pairwise functions. Restriction 3 is dictated by the need for a fast inference algorithm. If it was not for the need of fast inference, we could have encoded JSNice into existing frameworks [40, 113] or as a probabilistic program [49]. However, these existing tools do not come with utilities to search only in a small space of label *candidates* (as defined in Section 2.4.2) for the MAP inference algorithm.

Restriction 1 is probably the most challenging to relax. Thanks to this restriction, our MAP inference procedure first builds a fixed graph and then performs a search only over labels associated with every node in that graph. We illustrate the effects on this restriction on a prediction task with one unknown property initially assigned to `i` as in Fig. 2.9 (a). In this example, the search procedure may explore configurations such as Fig. 2.9 (b) that assign different values to nodes associated with unknown program properties.

However, configurations that involve different graph structures such as the one in Fig. 2.9 (c) are not included in the search space. Thus, restriction 1 is important for the decidability and the scalability of the algorithm, because it avoids exploring the potentially infinite space of graphs. In practice, this means that the model is directly applicable to problems such as deobfuscation or type prediction, but pushing this model to other problems such as synthesis of complex program expressions (e.g. synthesizing expressions that are defined recursively or making completions with unbounded number of completed elements) would also require other MAP inference procedures, which is an open problem.

IMPROVING THE MODEL EXPRESSIVENESS CRF is a powerful model leveraged in multiple fields such as computer vision, natural language processing and others [60, 79, 101]. In this chapter, we presented an instantiation of CRF that pushed its scalability and enabled new programming applications such as JSNice. That model represents a solution based on traditional machine learning techniques – a log-linear model based on feature functions trained with a gradient descent procedure. It is also a *discriminative* model, which means that for each program $\boldsymbol{x}$ in the training data, certain elements $\boldsymbol{y}$ are selected for prediction and then a model is trained to predict $\boldsymbol{y}$ given $\boldsymbol{x}$. In probabilistic terms, this is learning a probability distribution $P(\boldsymbol{y} \mid \boldsymbol{x})$. These techniques are powerful, but they come with their limitations as discussed before. Further, there is need to *manually* define a vector of feature functions, for which the training procedure finds weights.

Next, we present another approach that is based on *generative* probabilistic models of programs. In contrast to the discriminative models, these *generative* models capture probabilities of programs $P(\boldsymbol{x})$ by modeling them as sequences of events and thus allow us to compare probabilities of programs of different length. In the next chapter, we develop such state-of-the-art generative models applicable to code completion and program synthesis.

Later in Chapter 5, we remove the requirement to provide feature functions. Instead, we directly synthesize a program (i.e. a feature function) that defines a generative probabilistic model. In a recent experiment for code completion [20], we has shown that leveraging such synthesis techniques leads to higher precision than using pure machine learning techniques such as support vector machines on simple, manually defined feature functions.

3 GENERATIVE MODELS FOR API COMPLETION

In this chapter, we present a new approach for creating probabilistic models of code and use these models for the task of API code completion. The chapter establishes a link between statistical models for code and statistical models for natural languages. We show an implementation of our approach in a tool for code completion of Java programs called Slang and an experimental evaluation on a number of real world programming scenarios. Our results show that Slang is fast and effective. Virtually all completions synthesized by Slang typecheck, and the desired completion appears in the top 3 results in more than 90% of the cases.

The problem considered here differs from the one discussed in the previous chapter in multiple ways. First, we solve a problem where sequences of unknown length (of API calls) may be predicted as opposed to a number of elements determined in advance. Second, in multiple cases we are interested not only in the best completion, but in a ranked list of solutions. Indeed, with the approach proposed in this chapter, we can rank solutions and report probability estimates along with each solution. Finally, we learn from a corpus where the actual completion positions (holes) are not available at learning time, but only at query time. This is in contrast to the type and name predictions whose positions are known both at query and at learning time.

Our approach is based on two ingredients: (i) a powerful static analysis that extracts sequences of method calls from large codebases, and (ii) a statistical language model such as *N-gram* or *(RNN) Recurrent Neural Networks* [38] to index the resulting sequences. We show how to reduce the problem of code completion to a natural-language processing problem of predicting probabilities of sentences. We show that careful design of the static analysis contributes to higher accuracy of the code completion system. That is, specialized static analysis establishes the link between statistical models for code and statistical models for natural languages.

In Table 3.1 we show two dimensions in the design of probabilistic code completion systems with statistical language models.

		Language model	
		n-gram	RNN [38]
Intermediate representation	Syntactic (tokens)	[61]	future
	Semantic (Sec. 3.3)	this work	this work

Table 3.1: Dimensions in encoding the probabilistic code completion problem into a language model.

In the first dimension, we consider an intermediate representation (IR) which captures sequences extracted from code. For example, the work of Hindle et al. [61] proposes to use the n-gram language model, but on sequences of tokens found in the source code. Their model only makes syntactic suggestions and according to a number of studies is very imprecise for API predictions [61, 95]. An evaluation of this IR in Section 5.3.4 also confirms that such a naïve model is imprecise. Instead of using a shallow representation of the code as tokens, in this work we propose program analysis that constructs sequences based on a semantically meaningful program representation that captures API call invocations.

Another dimension in designing the code completion system is choosing the right statistical model. In our experiments, we use an n-gram language model, as well as a deep learning-based language model called RNN [89]. To the best of our knowledge, this work is the first to apply deep learning techniques to code synthesis. Finally, we show that the combination of a suitable intermediate representation with state-of-the-art statistical language model leads to high accuracy for the problem of Java API code completion that we consider here.

3.1 MOTIVATION

To accomplish many common tasks, programmers increasingly rely on the rich functionality provided by numerous libraries and frameworks. Unfortunately, a typical API can involve hundreds of classes with dozens of methods each, and often requires specific sequences of operations to be invoked to perform a single task [12, 135, 136]. Even experienced programmers might spend hours trying to understand how to use a simple API [86]. To address this challenge, recent years have seen increasing interest in code search, recommendation and completion systems [5, 56, 63, 86, 90, 99, 111, 117, 128, 140].

Despite significant progress, existing techniques cannot *synthesize* usable code beyond simple sequences required for instantiation of library objects. No existing technique can generate code of the complexity found in real tutorials and code examples. In fact, most existing approaches to code completion target completion based on shallow semantic information, and cannot capture the temporal information required for predicting correct code using a library. Some specification-mining techniques do capture rich temporal information (see Sec. 3.8), but do not attempt to synthesize usable code.

OUR APPROACH: USING STATISTICAL LANGUAGE MODELS Statistical language models have been successfully used to model regularities in natural languages and applied to problems such as speech recognition, optical character recognition, and others [114].

Our main idea is to reduce the problem of code completion to a *natural-language processing* problem of predicting probabilities of sentences: we use regularities found in sequences of method invocations to predict and synthesize a likely method invocation sequence for code completion. This reduction is in fact building a probability distribution over method invocation sequences such that method invocation sequences can be predicted for new programs.

BIG DATA, SMALL PROGRAMS To construct the statistical language model, we use static analysis to extract a large number of *histories* of API method calls from a massive number of *code snippets* obtained from GitHub [46] and other repositories. The extracted histories are used as training sentences for the statistical language model. We show how to use this statistical model in order to generate completions in code. Then, we show that the quality of the synthesized completions depends on the precision of the static analysis and whether we use aliasing information during the history extraction. This is, the static analysis is used to extract relevant information for predicting API method calls. Overall, our synthesizer represents a new *combination* of statistical language models with program analysis techniques.

The synthesizer takes as input a partial program with holes and outputs a program where all of the holes are filled in with (sequences of) method invocations. Computing the "small program" required for

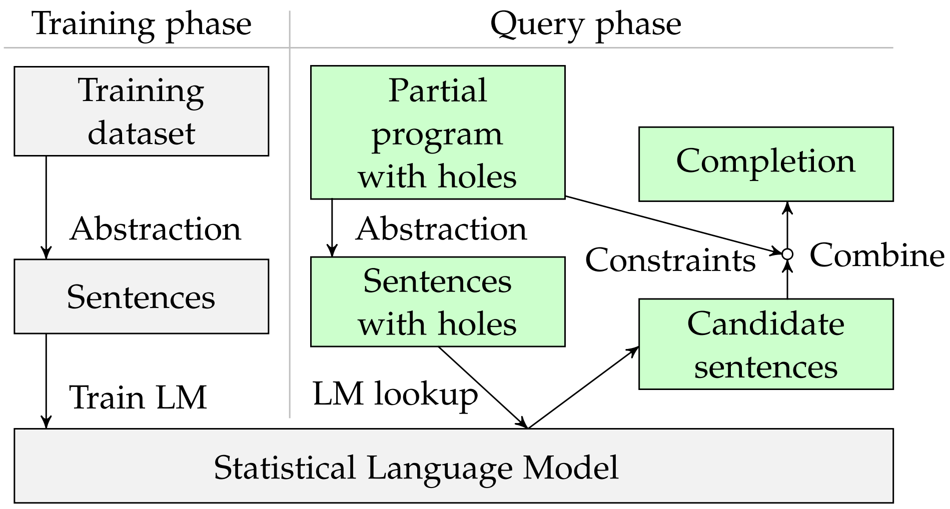


Figure 3.1: The architecture of SLANG.

code completion, is based on the language model constructed from "big data". Specifically, we employ the language model to find the highest ranked sentences, and use them to synthesize a code completion.

The synthesizer can:

1. discover sequences of invocations across multiple types,
2. complete both invocations *and* arguments of invocations,
3. complete multiple holes as well as each hole with a sequence of invocations, and
4. infer fused completions which do not exist in the training set.

3.2 OVERVIEW

The overall flow of SLANG is shown in Fig. 3.1. During the training phase, we use program analysis to extract sequences of API calls from the entire code base. Then, a statistical language model is trained on this extracted data. In this work we use the N-gram model, Recurrent Neural Networks and a combination of these two. The result of the training phase is a probability associated with each of the extracted sequences of method invocations. To interact with SLANG, the programmer provides a partial program with holes. Our program analysis extracts the sequences from this partial program, and uses the statistical language model to compute a set of candidate completion sequences. The final completion for all the holes is selected based on the highest probability and on whether the completion satisfies the constraints posed by each hole.

The effectiveness of SLANG is due to a careful *combination* of statistical models with program analysis. In particular, we use a form of alias and history analysis to extract sequences of method invocations from the code base, which are then used to train the language model. Training on sequences extracted *without* performing program analysis produces poor results and fails to produce completions (let alone desired ones) for many examples. Our combination of program analysis and language models makes the difference between not obtaining any solution at all versus obtaining the desired solution at the top of the list.

LEARNING CORPUS A key enabler for SLANG is the availability of large code repositories to train on. This code, however, is not necessarily easily executable. First, compiling, resolving dependencies and linking is a challenge (sometimes compiling instructions are only available in plain text). Then, many programs do not include inputs that would make them readily executable. Finally, even if we execute some programs, it is unclear if these executions will produce interesting traces to learn from. In contrast, *static* program analysis does not suffer from these limitations and is available even for partial programs [33].

EXAMPLE To illustrate our SLANG system, consider the Android example shown in Fig. 3.2. The Android `MediaRecorder` API is known to be quite involved. The official documentation for this API includes a state-machine with 7 different states[1], corresponding to internal states of the media recorder.

Consider a programmer trying to work with the `MediaRecorder` API and interested in combining this API with other APIs from classes such as `Camera` and `SurfaceHolder`. The programmer may have partial knowledge about `MediaRecorder`, for instance, she may know that she needs to set an audio and video source as well as the exact API calls and parameters for doing so. However, she may still be missing some of the details.

1 see `http://developer.android.com/reference/android/media/MediaRecorder.html`

```
void exampleMediaRecorder() throws IOException {
  Camera camera = Camera.open();
  camera.setDisplayOrientation(90);
   ?  // (H1)
  SurfaceHolder holder = getHolder();
  holder.addCallback(this);
  holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
  MediaRecorder rec = new MediaRecorder();
   ?  // (H2)
  rec.setAudioSource(MediaRecorder.AudioSource.MIC);
  rec.setVideoSource(MediaRecorder.VideoSource.DEFAULT);
  rec.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
   ? {rec} // (H3)
  rec.setOutputFile("file.mp4");
  rec.setPreviewDisplay(holder.getSurface());
  rec.setOrientationHint(90);
  rec.prepare();
   ? {rec} // (H4)
}
```

(a)

```
void exampleMediaRecorder() throws IOException {
  Camera camera = Camera.open();
  camera.setDisplayOrientation(90);
  camera.unlock(); // (H1)
  SurfaceHolder holder = getHolder();
  holder.addCallback(this);
  holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
  rec = new MediaRecorder();
  rec.setCamera(camera); // (H2)
  rec.setAudioSource(MediaRecorder.AudioSource.MIC);
  rec.setVideoSource(MediaRecorder.VideoSource.DEFAULT);
  rec.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
  rec.setAudioEncoder(1); // (H3) - completed with two methods
  rec.setVideoEncoder(3);
  rec.setOutputFile("file.mp4");
  rec.setPreviewDisplay(holder.getSurface());
  rec.setOrientationHint(90);
  rec.prepare();
  rec.start(); // (H4)
}
```

(b)

Figure 3.2: (a) A partial program using `MediaRecorder` and other APIs, and (b) its completion as synthesized by Slang.

Using SLANG, a programmer can write the partial program of Fig. 3.2(a), and rely on the synthesizer to complete the missing details. The partial program uses the statement "?" to denote a "hole", missing code to be completed by the synthesizer. The program of Fig. 3.2(a) has four different holes, marked with comments (H1)-(H4). Each hole is a query to the synthesizer asking it to infer a sequence of method invocations using (some of) the variables that are in scope. The hole can be constrained to only use certain variables by specifying a set of variable names. In this example, holes (H1)-(H2) are not bound to specific variables, while (H3)-(H4) are limited to only infer invocations that use the variable `rec` (either passed in as an argument or as the receiver). In Section 3.5, we describe other forms of queries that can relay additional information to the synthesizer. Given the partial program of Fig. 3.2(a), SLANG automatically synthesizes completions for the holes using the most likely sequences of method invocations, shown in bold in Fig. 3.2(b).

KEY ASPECTS This example highlights four key aspects of SLANG:

- **Completion across multiple types:** the completion of (H1) as `camera.unlock()` is an invocation on an object of type `Camera`, the completions for (H2-H4) are invocations on an object of type `MediaRecorder`. Further, the completion of (H2) as `rec.setCamera(camera)` uses parameter of type `Camera` in a completed invocation for `MediaRecorder`.

- **Completion of parameters:** the completion of (H3) using the two invocations `rec.setAudioEncoder(1)` and `rec.setVideoEncoder(3)` includes not only the invocations, but also the required parameters.

- **Holes as sequences:** The completion of (H3) uses a sequence of two invocations to complete a single hole. In general, our approach can generate a sequence of invocations to complete a hole (up to some specified length).

- **New fused completions:** Our system can infer fused sequences that *did not exist* before. Neither of the sequences for `Camera` or `MediaRecorder` were in the training set, yet SLANG successfully synthesized an invocation that involves both of these in order to complete the hole (H2).

3.3 FORMAL SEMANTICS

In this section, we provide basic definitions of an *event* and a sequence of events (history) that we use in the rest of the paper. In Section 3.3.1, we provide a simple instrumented semantics for tracking sequences of events over objects. Because there is no a priori bound on the number of dynamically allocated objects, and no a priori bound on the length of a history, the concrete semantics is generally non-computable. In Section 3.3.2, we present an abstract semantics that provides a bounded representation for histories, and tracks a bounded set of bounded histories for each (abstract) object.

3.3.1 *Concrete Semantics*

We define an instrumented concrete semantics that tracks the concrete sequence of events for each concrete object. We refer to the concrete sequence of events as the *concrete history* of the concrete object. We start with a standard concrete semantics for an imperative object-oriented language, defining a program state and evaluation of an expression in a program state.

OBJECTS AND PROGRAM STATE Restricting attention to reference types, the semantic domains are defined as follows:

$$\begin{array}{lcl}
L^\natural & \in & \mathcal{P}(objects^\natural) \\
v^\natural & \in & Val = objects^\natural \cup \{null\} \\
\rho^\natural & \in & Env = VarIds \to Val \\
\pi^\natural & \in & Heap = objects^\natural \times FieldId \to Val \\
state^\natural = \langle L^\natural, \rho^\natural, \pi^\natural \rangle & \in & States = \mathcal{P}(objects^\natural) \times Env \times Heap
\end{array}$$

where $objects^\natural$ is an unbounded set of dynamically allocated objects, *VarIds* is a set of local variable identifiers, and *FieldId* is a set of field identifiers. A *program state* tracks the set $L^\natural$ of allocated objects, an *environment* $\rho^\natural$ mapping local variables to (reference) values, and a heap $\pi^\natural$ mapping fields of allocated objects to values.

INSTRUMENTED SEMANTICS: EVENTS AND HISTORIES In our instrumented semantics, each concrete object is mapped to a "concrete history" that records the sequence of *events* that has occurred for that object. That is, we employ a form of per-object cartesian abstraction. An *event* for an object o corresponds to an invocation of an API method involving the object o: o can either be the receiver object (`this`), the return value returned by the API invocation, or one of the arguments to the method invocation.

More formally, an *event* is a pair $\langle m(t_1, \ldots, t_k), p \rangle$, of a method signature $m(t_1, \ldots, t_k)$, and a position argument p denoting the position of the object o in the invocation of m. The position p can be 0 denoting `this`, or a value denoting one of the positions $1, \ldots, k$. We also use a designated position value `ret` to denote the case where o is a new object returned from the invocation of m. To simplify presentation, we assume that an object appears in at most one position of a given method invocation, and that methods are not invoked with a *null* argument. Our implementation deals with the more general case where an object can appear in multiple positions (by replacing the position argument p to be a set of positions), and correctly handles invocations with *null* arguments.

Given an API A with methods $m_1, \ldots, m_n$, we use Σ_A to denote the set of all events over the API. When the API is clear from context, we omit the subscript A. We define the notion of a *concrete history* for an API simply as a sequence of events Σ^*. We denote the empty concrete history by ϵ and denote the set of all concrete histories by $\mathcal{H}$. The instrumented semantics is obtained by augmenting every concrete state $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$ with an additional mapping that maps each allocated object to its concrete history, that is $his^\natural : L^\natural \rightharpoonup \mathcal{H}$. Given a state $\langle L^\natural, \rho^\natural, \pi^\natural, his^\natural \rangle$, the semantics generates a new state $\langle L^{\natural\prime}, \rho^{\natural\prime}, \pi^{\natural\prime}, his^{\natural\prime} \rangle$ when evaluating each statement. We assume a standard interpretation for statements updating $L^\natural$, $\rho^\natural$, and $\pi^\natural$. The $his^\natural$ component changes on object allocations and method invocations:

- *Object Allocation:* The statement `x = new T()` allocates a fresh object $o_{new} \in objects^\natural \setminus L^\natural$ initialized with the empty-sequence history $his^{\natural\prime}(o_{new}) = \epsilon$.

- *Method Invocation:* For an invocation x_0.`m` $(x_1, \ldots, x_n)$ of a method with signature $\mathtt{m}(t_1, \ldots, t_n)$, the history of every object $o = \rho^\natural(x_i)$ (where $0 \leq i \leq n$) is extended with an event $e = \langle \mathtt{m}(t_1, \ldots, t_n), i \rangle$, that is: $his^{\natural\prime}(o) = his^\natural(o) \cdot e$, adding an event to the history to reflect the invocation of `m`. If the invocation of `m` returns an object r, its history $his^\natural(r)$ is extended with $\langle \mathtt{m}(t_1, \ldots, t_n), \mathtt{ret} \rangle$.

3.3.2 *Abstract Semantics*

The instrumented concrete semantics is generally non-computable as there are no a priori bounds on the number of dynamically allocated objects, or on the length of histories. We now present an abstract semantics that provides a bounded representation.

HEAP ABSTRACTION We use a flow-insensitive and field-sensitive Steensgaard style [123] points-to analysis to partition the $objects^\natural$ set into a bounded set of abstract objects called *objects*.

HISTORY ABSTRACTION Our goal is to extract a set of *sentences* that can be given as input to language models (see Section 3.4). Towards that end, we bound the number of loop iterations in our analysis to guarantee that collected histories are of bounded length. We define the notion of an *abstract history* as a set of concrete histories of bounded length, namely an abstract history $h \subseteq \mathcal{H}$. That is, while a concrete history describes a unique sequence of events, an abstract history represents potentially many concrete histories capturing the different control flows through the program.

ABSTRACT STATE The tuple $\langle L, \rho, \pi, his \rangle$ denotes an instrumented abstract program state consisting of the set of allocated abstract objects, the local variables which point to abstract objects, the abstract heap and the abstract history for each abstract object. The definition of the first three components is computed in a standard way. We next discuss the definition of *his* which is now lifted to abstract objects and abstract histories as follows: $his\colon L \rightharpoonup \mathcal{P}(\mathcal{H})$.

ABSTRACT SEMANTICS OF *his* The abstract semantics for updating *his* follow the structure of the concrete semantics except that it is lifted to deal with abstract objects, and abstract histories.

- *Object Allocation:* The statement `x = new T()` results in an abstract object $a_{new} \in objects$ with a set containing a new empty history: $his(a_{new}) \cup = \{\epsilon\}$.
- *Method Invocation:* For an invocation x_0.m $(x_1, \ldots, x_n)$ of a method with signature $\mathtt{m}(t_1, \ldots, t_n)$, the abstract history of every abstract object $o = \rho(x_i)$ $(0 \leq i \leq n)$ is extended with $e = \langle \mathtt{m}(t_1, \ldots, t_n), i \rangle$,

that is, $his'(o) = \{h \cdot e \mid h \in his(o)\}$, adding an event to each concrete history of the abstract history. If the invocation returns an object r, the abstract history $his(r)$ is extended with $\langle \mathtt{m}(t_1, \ldots, t_n), \mathtt{ret} \rangle$.

JOINS Whenever a join of the control-flow occurs, the new history for each abstract object is computed by combining the histories for that abstract object arriving from each of the branches (by applying union to the corresponding sets). As long as the domain of abstract histories is bounded, the analysis is guaranteed to terminate. However, in practice, it can suffer from an exponential blowup due to branching control flow. To mitigate potential exponential blowup, we limit the number of collected histories by some threshold. Once that threshold has been met, we randomly evict older histories to make room for new ones. In our experiments, we used the threshold 16 which was sufficient for 99.5% of the analyzed methods.

3.4 STATISTICAL LANGUAGE MODELS

Statistical language models have been used to model the regularities in natural languages and improve the performance of problems such as speech recognition, statistical machine translation, optical character recognition, and others [114]. In this work, we use regularities found in sequences of method invocations to predict and synthesize a likely method invocation sequence in the context of code completion. In this section, we first define the necessary statistical language modeling background, and then show how language models can be leveraged for synthesis of code completions.

Statistical language models are based on the concepts of words and sentences, where each sentence is an ordered sequence of words. Every word w is taken from a set D also called a dictionary. A language is informally defined as all sentences used in some particular domain. The goal of a language model is to build a probabilistic distribution over all possible sentences in a language. This is, given a sentence s, the language model estimates its probability $Pr(s)$. For a sentence

$s = w_1 \cdot w_2 \cdot ... \cdot w_m$, many language modeling approaches estimate its probability as follows:

$$Pr(s) = \prod_{i=1}^{m} Pr(w_i \mid h_{i-1})$$

where we refer to the sequence $h_i = w_1 \cdot w_2 \cdot ... \cdot w_i$ as a history. That is, the probability of a sentence can be calculated by generating it word by word using conditional probabilities on the already generated words. Furthermore, language models are usually constructed on a finite amount of training data that is used to estimate the actual probabilities of sentences. Because not all possible sentences in the language or their prefixes will be in the training data (also referred to as the problem of sparse data [114]), the model uses other statistical techniques to estimate probabilities.

3.4.1 *N-gram Language Models*

In order to deal with the sparseness of the data, an N-gram data model estimates the probability of a sentence of m words by modeling a language as a Markov source of order $n - 1$:

$$Pr(s) = \prod_{i=1}^{m} Pr(w_i \mid w_{i-n+1} \cdot ... \cdot w_{i-1})$$

That is, the probability of the next word w_i depends only on the previous $n - 1$ words. Note that this definition also refers to words with negative indices w_a for $a < 0$, for example when looking the words preceding the first word. In this case, the language model uses a special "start of sentence" word $w_a =$ `<s>` to denote conditional probability on beginning a sentence. In our work we use the trigram language model where the probability of a word depends on a pair of previous words. That is, for the trigram language model, we have that:

$$Pr(s) = \prod_{i=1}^{m} Pr(w_i \mid w_{i-2} \cdot w_{i-1}).$$

Such probabilities are estimated by counting the number of occurrences of trigrams and bi-grams in the training data.

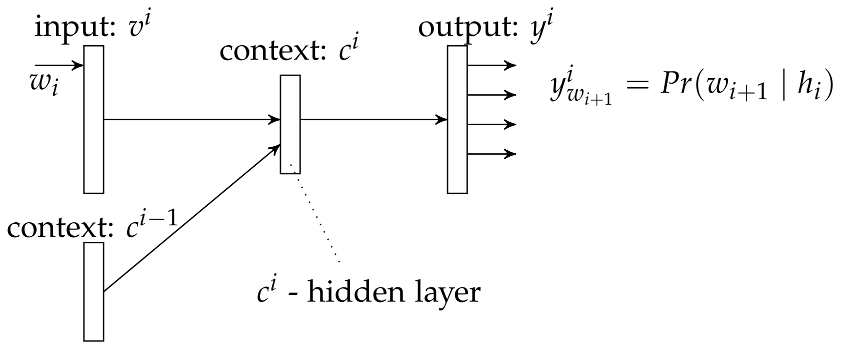


Figure 3.3: A scheme of a recurrent neural network (RNN). The input is a word vector for the i-th word in a sentence, the output is probabilities for different possible words at position $i+1$.

SMOOTHING Even for small n, these models can still suffer from the problem of data sparseness. For example, some n-grams may only occur once or not at all in the training data and yet their probabilities must be estimated. To mitigate this problem, practical n-gram language models use counts for n-grams, $(n-1)$-grams and for all lengths down to unigrams. Further, they smooth the probability by using models of lower length [72, 73] when sparseness problems in estimating sentence probabilities occur. In our work, we use Witten-Bell backoff smoothing [137], which is applicable even when we remove rare words from the training data.

3.4.2 *Recurrent Neural Networks (RNNs)*

In recent years, the increased availability of computational resources for training led to wider adoption of neural networks for predicting probabilities of sentences [76]. These approaches are conceptually interesting in the fact that they do not capture only regularities between a word and a fixed number of predecessor words, but may also capture longer distance relations between words. Initially proposed by Elman [38], recurrent neural networks (RNNs) predict probabilities of the $(i+1)$-st word according to the scheme in Fig. 3.3.

In the schema, v^i and y^i are vectors of $|D|$ real numbers, where D is the dictionary such that every possible word $x \in D$ has a corresponding index in v^i and y^i (referred as v^i_x and y^i_x). Let c^i (for every i) be a vector of p real numbers. The number p is also called the size of the hidden layer and the entire network is referred to as RNN-p. RNN uses two functions f and g and estimates word probabilities iteratively on a sentence $s = w_1 \cdot ... \cdot w_m$ by performing the following actions for every word $w_i \in s$: i) set all positions of v^i to zeros, except position $v^i_{w_i}$ to one; ii) compute $c^i = f(v^i, c^{i-1})$ and $y^i = g(c^i)$. Then the vector y^i is used as an estimator of the probabilities for the next word w_{i+1}:

$$Pr(w_{i+1} \mid w_1 \cdot ... \cdot w_i) \approx y^i_{w_{i+1}}$$

During training, the functions f and g are learned from data to minimize the error rate of the estimates y^i (details are in [76]). What is essential for RNNs, however, is that they can capture long distance regularities in the language via the hidden layer c^i. Intuitively, the values in c^i act as an internal state of an automaton and at every step i, the previous internal state c^{i-1} is used for computing c^i.

In SLANG, we use RNNME-p – a faster variant of RNN with a hidden layer size of p that combines RNN-p with a class-based maximum entropy model [89]. To the best of our knowledge, this work was the first one to apply deep learning to probabilistic models for programs.

COMBINATION MODELS Due to the different nature of the models based on n-grams and RNNs, it is possible that averaging the probabilities returned by two probabilistic models performs better than using the probabilities of the individual models separately. In our experiments in Section 3.7, we show that a combined model between a 3-gram and a RNNME-40 language model ranks the correct completion as a first result in more cases than the two base models individually.

FUTURE WORK There is an ongoing effort to develop even more precise neural language models than recurrent neural networks. One issue of RNNs is that while they can theoretically express arbitrary long distance relationships, learning these relationships is hard [14]. To address this problem, Gers et al. [45] proposed variations of RNNs called Long Short-Term Memory (LSTM) language models. We leave exploring LSTMs as a future work item, but since our abstraction limits the length of the sentences to only 16 words, we do not expect LSTMs alone to bring significant improvements.

3.4.3 *Sentence Completion with Language Models*

In addition to computing probabilities for single sentences, we can leverage a language model to complete missing holes in a sentence (with the most likely completions). As a simple example, consider the following natural language sentence with a missing word:

```
The quick brown ? jumped .
```

If the word `?` is replaced with an actual natural language word from the dictionary of words D, a statistical language model is useful as a scoring function of the most probable completion. However, certain language models are also useful to suggest very likely completions of the holes. For example, a bigram model keeps all pairs of sequential words that are present in the training data. In our example, these could be the pairs $\langle \texttt{brown}, \texttt{fox} \rangle$, $\langle \texttt{brown}, \texttt{dog} \rangle$, etc. Then, if the word preceding the hole is a (e.g., $a = \texttt{brown}$), we can suggest filling the hole only with words x, such that $\langle a, x \rangle$ are bigrams in the training data. This procedure significantly reduces the set of words that are candidate completions of the sentence holes by producing candidates which a language model may score high.

Note that in natural languages punctuation signs such as "`.`" denote end of a sentence. For the purposes of language models these signs can be treated as words. They play an important role in giving high probabilities to complete sentences and giving low probabilities to incomplete sentences. Consider the following pair of sentences:

```
The quick .
The quick brown fox jumped .
```

Despite the fact, that the first sentence is shorter, it likely has low overall probability because it is highly unlikely that "`.`" appears after the words "`The quick`" somewhere in the training data and $Pr(\texttt{ . } \mid \texttt{ The quick })$ would be very low. On the other hand, the second sentence is longer, but may have higher probability, because every word in the sentence should have reasonable probability to follow the words preceding it. This distinction of long and complete versus short and incomplete sentences would not be possible without the "`.`" marker. In our language model, to make sure such end-of-sentence marker is always present, we always append a special `</s>` word at the end every sentence (although we omit it in the presentation to keep the notation short). Using such a marker is also standard in existing language implementations such as SRILM [125].

3.4.4 *Training on Programs*

Recall that in Section 3.3, we presented a history abstraction that maps every (abstract) object to a set of histories (i.e., sentences). These sentences can be automatically extracted via program analysis and then fed to a statical language model which can train on this data.

This abstraction nicely matches the two worlds of program analysis and language models: an event in the semantics corresponds to a language word and a history sequence $h \in \mathcal{H}$ corresponds to a language sentence. To train a language model on a large set of programs, we: i) use program analysis to extract the abstract objects and their corresponding (history) sequences; and ii) discard the abstract objects, treat the extracted histories as sentences in the language, and train a statistical language model over this data.

3.5 SYNTHESIS

So far we discussed the training phase of SLANG. We next discuss how code completion works. The synthesizer takes as input a partial program (augmented with holes) and outputs a program where the holes are filled with (sequences of) method invocations. To enable programmers to use our approach and specify partial programs, we introduce the following construct for specifying holes:

```
? lvars:l:u
```

where `lvars` $\in \mathcal{P}(VarIds)$ is a set of (reference) local variables and `l` and `u` are natural numbers which constrain the length of the sequence from below and from above. All of these are optional parameters which are provided as a convenience to the programmer in case she would like to constrain the possible completions. Informally, this construct directs the synthesizer to search for a valid replacement of `? lvars:l:u` with a sequence of method invocations where `lvars` participates in each invocation and where the length of the sequence is between `l` and `u`. For example, the hole `?` directs the synthesizer to look for the most likely sequence of method invocations of *any* length. A more restrictive hole would be `?{x}` which instructs the synthesizer to find sequences where variable `x` participates in the method invocation: either a method on `x` was invoked *or* `x` is passed in as an argument to some other method. That is, in the sequence, for each of the method invocations, the variable `x` should participate in some form. The meaning of a query such as `?{x,y}:1:1` is that the suggested sequence must consist of exactly `1` method invocation where *both* `x` and `y` participate in that invocation.

CODE COMPLETION: STEP-BY-STEP We now present the procedure which takes as input a partial program that may contain multiple holes and infers the most likely completions for the holes. To avoid clutter, we describe the case where all of the holes require `l` and `u` to be equal to `1`, that is, all holes have the shape: `?lvars:1:1`. This means that every hole has to be replaced with exactly one method invocation (there could be multiple variables constraining the hole). We can translate holes of the more general shape `?lvars:l:u` to $\mathtt{u} - \mathtt{l} + 1$ separate queries: for every $i \in [\mathtt{l}, \mathtt{u}]$, perform a query with i sequentially placed holes where each hole has the shape `?lvars:1:1`.

Before we explain the steps of our algorithm, let us introduce some necessary notation. Recall that a concrete history is a sequence of events where each event (see Section 3.3.1) represents a method invocation. However, with partial programs, we now have hole statements which are to be replaced with sequences of events. Therefore, we define a set of histories with holes $H^\circ = (\Sigma \cup G)^*$ where G represents all possible holes. Next, we explain our algorithm and illustrate each step on the example in Fig. 3.4. The example is based on a question from StackOverflow at http://stackoverflow.com/questions/14452808/sending-and-receiving-sms-and-mms-in-android. Here, we have a partial program Fig. 3.4(a), for which SLANG must synthesize the completion in Fig. 3.4(b). That is, the tool must infer that if the message was divided into parts, the most likely method to call is `sendMultipartTextMessage`, while otherwise it is `sendTextMessage`. The first and the second hole are assigned unique identifiers `H1` and `H2` respectively.

STEP 1: EXTRACT ABSTRACT HISTORIES WITH HOLES Given a partial program, for each abstract object, we automatically extract its abstract histories with holes (as described in Section 3.3.2, except that we

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  ? {smsMgr, msgList}  // (H1)
} else {
  ? {smsMgr, message}  // (H2)
}
```

(a)

```
SmsManager smsMgr = SmsManager.getDefault();
int length = message.length();
if (length > MAX_SMS_MESSAGE_LENGTH) {
  ArrayList<String> msgList =
      smsMgr.divideMsg(message);
  smsMgr.sendMultipartTextMessage(...msgList...);  // (H1)
} else {
  smsMgr.sendTextMessage(...message...);  // (H2)
}
```

(b)

Figure 3.4: (a) A partial program, and (b) its completion as automatically synthesized by SLANG (the full list of parameters is omitted for clarity). The example is based on a question from StackOverflow.

now also have holes appearing in abstract histories). The output of this step is a function $his^{pt} : L \rightharpoonup \mathcal{P}(H^{\circ})$. For our running example, the output of this step will be a map his^{pt} defined as follows:

$$\begin{aligned}
\texttt{smsMgr} \quad &\mapsto \{\langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{H2}\rangle\ , \\
&\quad \langle \texttt{getDefault}, \texttt{ret}\rangle \cdot \langle \texttt{divideMsg}, 0\rangle \cdot \langle \texttt{H1}\rangle\} \\[1em]
\texttt{message} \quad &\mapsto \{\langle \texttt{length}, 0\rangle\ , \\
&\quad \langle \texttt{length}, 0\rangle \cdot \langle \texttt{H2}\rangle\} \\[1em]
\texttt{msgList} \quad &\mapsto \{\langle \texttt{divideMsg}, \texttt{ret}\rangle \cdot \langle \texttt{H1}\rangle\}
\end{aligned}$$

STEP 2: COMPUTE CANDIDATE COMPLETIONS We next compute the set of candidate completions for all of the abstract histories obtained from Step 1. For our example, this set of partial histories is shown in the first column of Fig. 3.5. To aid the subsequent completion, we slightly overload the notation for holes and to each hole, we also add the abstract object for which the partial abstract history was built. For instance, if SLANG suggests $\langle \texttt{sendTextMessage}, 0\rangle$ for replacing $\langle \texttt{H2}, \texttt{smsMgr}\rangle$, then `smsMgr` will be placed at position 0, essentially denoting the invocation `smsMgr.sendTextMessage(...)`.

For each partial abstract history, we compute a *sorted* list of possible histories *without* holes. The way we do that is via a two-step approach. In the first step, we use the bigram model in order to suggest candidate completions to the holes and obtain histories without holes (as discussed in Section 3.4.3). Then, in the second step, we use an N-gram language model or an RNN model to rank these completed candidate histories.

Finally, we end up with a map $\mathit{candidates} : H^{\circ} \rightharpoonup \mathcal{H}^{*}$ where for a partial abstract history, the list $\mathit{candidates}(h)$ is sorted by the probability of the sequence (history without holes). That is, more likely sequences appear ahead of less likely sequences. For our example, the candidate completions together with the probability of each sequence are shown in the last two columns of Fig. 3.5.

Partial History	Id	Candidate Completions	*Pr*
$\langle$`getDefault,ret`$\rangle \cdot \langle$`H2,smsMgr`$\rangle$	11	$\langle$`getDefault,ret`$\rangle \cdot \langle$`sendTextMessage`,0$\rangle$	0.0073
	12	$\langle$`getDefault,ret`$\rangle \cdot \langle$`sendMultipartTextMessage`,0$\rangle$	0.0010
$\langle$`getDefault,ret`$\rangle \cdot \langle$`divideMsg`,0$\rangle \cdot \langle$`H1,smsMgr`$\rangle$	21	$\langle$`getDefault,ret`$\rangle \cdot \langle$`divideMsg`,0$\rangle \cdot \langle$`sendMultipartTextMessage`,0$\rangle$	0.0033
	22	$\langle$`getDefault,ret`$\rangle \cdot \langle$`divideMsg`,0$\rangle \cdot \langle$`sendTextMessage`,0$\rangle$	0.0016
$\langle$`length`,0$\rangle \cdot \langle$`H2,message`$\rangle$	31	$\langle$`length`,0$\rangle \cdot \langle$`length`,0$\rangle$	0.0132
	32	$\langle$`length`,0$\rangle \cdot \langle$`split`,0$\rangle$	0.0080
	33	$\langle$`length`,0$\rangle \cdot \langle$`sendTextMessage`,3$\rangle$	0.0017
	34	$\langle$`length`,0$\rangle \cdot \langle$`sendMultipartTextMessage`,1$\rangle$	0.0001
$\langle$`divideMsg,ret`$\rangle \cdot \langle$`H1,msgList`$\rangle$	41	$\langle$`divideMsg,ret`$\rangle \cdot \langle$`sendMultipartTextMessage`,3$\rangle$	0.0821

Figure 3.5: The partial sequences extracted from the program in Fig. 3.4 and their candidate completions (with probabilities).

STEP 3: COMPUTE AN OPTIMUM AND CONSISTENT SOLUTION Finally, in step 3 we compute the map $completion\colon H^\circ \rightharpoonup \mathcal{H}$. That is, for each partial abstract history $h \in H^\circ$, we need to select a history from $candidates(h)$ which completes h. However, even though the list $candidates(h)$ is sorted by probability, we may not always pick the first sequence in that list. The reason we cannot always pick the first sequence is because we need to make a *global* decision for *all* suggested completions, rather than a *local* per-history choice. In our algorithm, we iterate over the map *candidates* (over the sorted lists in *candidates*), following the sorted priority order and build a map *completion* for each abstract history. In particular, the *completion* which we return satisfies two criteria:

- *Global optimality*: Let $T = his^{pt}(L)$ denote all partial abstract histories. Then, the returned *completion* should maximize the score:
$$\frac{\sum_{h \in T}(Pr(completion(h)))}{|T|}$$
- *Consistency*: A proposed *completion* should also be consistent: we make sure that the completion satisfies certain constraints imposed by the programming language and by the constraints of the hole. First, if a hole appears multiple times (e.g., due to loop unrolling), then we make sure that the hole is always filled in with the same completion in every history of *completion*'s range (to yield a syntactically valid program). Second, if we have a hole of type `?{x,y,...}:1:1` which involves more than one variable (which do not alias), we make sure that the variables `x`, `y`,..., appear as parameters at different positions in the corresponding suggestion.

Since our completion algorithm starts with the highest scoring completion and exhaustively generates candidates in reverse score order until a consistent completion is obtained, our procedure is guaranteed to always find the best scoring *completion*. Finally, given a *completion*, we extract the methods found for each hole and suggest those to the developer.

COMPLETIONS FOR OUR EXAMPLE Back to our example, if we choose the completions 11, 21, 31, and 41 in Fig. 3.5, we get the highest probability according to the Global optimality equation above. However the combination of these sentences is *inconsistent* because completion 11 suggests that we fill the hole `H2` with method `sendTextMessage` while 31 suggests that we use `sendTextMessage`. This is clearly impossible when the hole is of size one. Thus, we continue to generate candidate completions in the order of their probabilities until we find the first consistent completion – using sentences 11, 21, 33, and 41. According to this choice of sentences, `H1` is filled with `sendMultiPartTextMessage`, and `H2` is filled with `sendTextMessage`. This is the completion returned to the developer, also shown in Fig. 3.4 (b).

3.6 IMPLEMENTATION

We implemented SLANG as a series of utilities that train statistical language models on massive codebases and perform completions on partial programs with holes. SLANG is implemented in Java and C++ and depends on a Java compiler for compiling the code, the Soot [131] framework for obtaining an intermediate representation (we work with Jimple) useful for program analysis, SRILM [125] for n-gram language models, and RNNLM [2] for recurrent neural networks. We have designed SLANG to be scalable and efficient: it can handle most queries in few seconds. Next, we discuss the implementation of the different components of SLANG.

3.6.1 *Program Analysis: Heap and Sequences*

We aimed at a simple, fast and scalable program analysis that can quickly process massive amounts of data. To abstract the heap, we implemented an intra-procedural Steensgaard-style alias analysis [123] due to its near linear time complexity and the fact that it can process classes and methods independently. At the start of every method, we assume that all reference arguments in the method do not alias. Generally, an assumption of this (or similar) kind is required, because at both training time and query time we do not have the entire context in which the method will execute. Further, for our problem of suggesting code completions, we are not limited to only consider over-approximations.

In our implementation of the history abstraction, we bound the number of loop iterations L in order to avoid exponential blowup in

2 http://www.fit.vutbr.cz/ imikolov/rnnlm/

space and time (in the number of generated sequences). Further, we do not consider extracted sequences with more than K words (invocations) per abstract object. We can easily vary both L and K, though in our experiments we set those to 2 and 16 respectively.

3.6.2 *Language Models: Preprocessing*

Once the sentences (histories) from the training data are obtained via the program analysis, we index them into a language model. As with natural languages, sentences include some commonly occurring words, but there is a heavy long tail of very rare words. However, the rarely occurring words are of little value for our code completion problem. The reason is that these words are likely to represent events that are specific to only a few projects in our index. Thus, we have added a preprocessing step that replaces words that occur less than a certain number of times in the training corpus with placeholder unknown words. This replacement has no observable effect on the availability of results other than for very rare API calls. However, it enables us to obtain compact n-gram language models and a small dictionary is essential for RNNs [15].

Once the preprocessing step is complete, SLANG invokes a language modeling system in order to generate an N-gram language model or an RNN model of the training data and in addition also builds a bigram model of the training data in order to create candidate completions as described in Section 3.4.3. These two steps are independent and can be performed in any order.

3.6.3 *Query Processing*

To perform a query in SLANG, the user provides a partial program with holes which are to be filled-in by the tool. Given a query, SLANG discovers a mapping from holes to (sequences of) method invocations. The completions include method names, as well as non-constant parameters given to the method call. That is, SLANG can infer both method invocations as well as the reference arguments passed to the invocation. To infer constants, we train a simple, but effective model that given a method call and a parameter position, returns the most likely constant to pass as a parameter.

CONSTANT MODEL We estimate the probability of a constant value as a parameter of a method *m* by counting the number of times each constant was given as a parameter to *m* in the training data and dividing it by the total number of calls to *m* in the training data. This simple model assumes that the constant values are independent of the context of the method or other parameters, yet the approach is fast, feasible and enables our completion to include complete method invocations.

3.7 EVALUATION

In this section we discuss an experimental evaluation of SLANG. Our main objective was to study how effective the combination of a statistical language model with a history abstraction is for code completion purposes. Towards that, we have obtained 3, 090, 194 Android methods used them as training data. Our training data consists of source code of Android applications collected from various source repositories. We compiled these sources using a specially modified version of the partial compiler [33], extended to handle more cases. Then we analyzed the compiled programs with Soot [131] to convert them into Jimple bytecode and then fed the bytecode as training data into SLANG.

3.7.1 *Training Parameters*

To evaluate the effect of various parameters on the quality of code completion, we experimented with three knobs: the size of the data set, the precision of the program analysis abstraction, and the different choices for the language models.

For the size of the training data set, we considered three choices. The first data set includes the entire codebase we have collected. The second (smaller) data set contains 10% of the files of the codebase. The third (smallest) data set contains 1% of the files. For the program analysis abstraction, we experimented with both options: enabling or disabling the alias analysis. Finally, we experimented with the following options for training the statistical language model:

1. A 3-gram language model with Witten-Bell backoff smoothing,
2. A RNNME-40 recurrent neural network language model,
3. A combination of the previous two language models.

Phase	**Running time on dataset**		
	1%	**10%**	**all data**
Training without alias analysis			
Sequence extraction	4.682s	54.187s	9m 3s
3-gram language model construction	0.352s	2.366s	10.187s
RNNME-40 model construction	5m 46s	0h 53m	5h 31m
Training with alias analysis			
Sequence extraction	3.556s	34.846s	5m 34s
3-gram language model construction	0.442s	3.239s	13.510s
RNNME-40 model construction	8m 42s	2h 16m	9h 34m

Table 3.2: Training phase running times.

3.7.2 *Training Phase*

We ran our experiments on a standard desktop workstation with a 3.5GHz Core i7 2700K processor, 16GB RAM, a solid state drive storage, and running 64-bit Ubuntu 12.04 with OpenJDK 1.7. Our system takes the Jimple input data and produces a language model as an output. Our system can parallelize some steps of the computation by performing the analysis on multiple cores, however we report runtimes only using a single thread.

Running times of our training phase are summarized in Table 3.2. First, we provide the time to extract the abstract histories (i.e., sequences) from the training data. Next, we provide running times for constructing each of the corresponding language models. We provide two pairs of numbers - without heap abstraction (assuming that no two pointers alias), and with a Steensgaard style alias analysis. In all cases, the training phase processes more than 5000 methods per second on average and the main slowdown occurs when we train the neural network. In our experiments, performing the alias analysis did not significantly affect the training time.

Table 3.3 provides statistics for the precomputation phase. As seen, by using alias analysis, the data size of the produced sentences increases by around 20%, and average sentence length increases by around 0.45 words. Importantly, the alias analysis enables extraction of more precise histories. All of this reduces noise in the training data and helps the

Data statistics		**Dataset**	
	1%	**10%**	**all data**
Training without alias analysis			
Sequences (file size as text)	7.2MiB	46.5MiB	597.4MB
Number of generated sentences	74979	759434	6989349
Number of generated words	188668	1864402	16430269
Average words per sentence	2.5163	2.4549	2.3508
3-gram language model file size	11.1MiB	50.9MiB	72.2MiB
RNNME-40 language model file size	19.3MiB	41.8MiB	29.7MiB
Training with alias analysis			
Sequences (file size as text)	9.3MiB	89.1MiB	761MiB
Number of generated sentences	81477	805578	7435307
Number of generated words	241004	2358302	20751368
Average words per sentence	2.9579	2.9275	2.7909
3-gram language model file size	14.6MiB	69.6MiB	108.1MiB
RNNME-40 language model file size	22.2MiB	51.1MiB	36.0MiB

Table 3.3: Data size statistics used for evaluation of SLANG.

language model learn longer and more precise event sequences from the training data.

In terms of language models, the RNNME-40 language model is significantly slower to train than the 3-gram model (the reason is that the time complexity per processed word in 3-gram is constant, while in RNN, it is linear to the size of the dictionary), but on the other hand the RNN index with all the data is smaller in size.

3.7.3 *Code Completion*

We designed three different kinds of code completion tasks for evaluating our system:

1. *Single object single-method completion*: this task is characterized by a *single* hole of type `?{x}:1:1` placed at the end of a method, meaning that given a local reference variable `x`, the task of the synthesizer is to discover exactly one method invocation which uses `x`. That is, the tool predicts the next method call to be performed involving `x`.

Id	Description
1	Registering a event listener to read the accelerometer
2	Add an account
3	Take a picture with the camera
4	Disable the lock screen
5	Get Battery Level
6	Get free memory card space
7	Get the name of the currently running task
8	Get the ringer volume
9	Get the SSID of the current WiFi network
10	Read GPS location
11	Record a video using `MediaRecorder`
12	Create a notification
13	Set display brightness
14	Change the current wallpaper
15	Display the onscreen keyboard
16	Register an SMS receiver
17	Send SMS
18	Load a sound resource to play in `SoundPool`
19	Display a web page in a `WebView` control
20	Toggle WiFi enabled/disabled

Table 3.4: Description on the examples from task 1 on which we perform prediction.

2. *General completion*: this task is characterized by *multiple* holes and includes examples like Fig. 3.2 and Fig. 3.4.

3. *Random completion*: this task completes methods from large programs where one or more holes were introduced at random.

The first task is similar to functionality provided by many IDEs where when `dot` is pressed, the IDE displays a complete list of all methods associated with the object on the left of the `dot`. In our case however, we only display a partial list of methods for which we have confidence given the training data.

EVALUATION DATA To evaluate task 1, we came up with 20 tasks that a programmer may want to accomplish. Solving these tasks requires usage of various Android APIs. We then inspected some of the popular solutions available on the Web, typically provided in the form of a code snippet. We summarized this set of examples in Table 3.4. To evaluate task 2, we selected 14 code snippets from task 1 where we believed it makes sense to extend the snippet to contain more than one hole and with more complex constraints. For both tasks, we introduced holes in the code snippets accordingly. We made sure to not include the evaluation data into the training data in order to avoid statistical problems such as overfitting.

For task 3, we took code from open source projects and randomly introduced holes in 50 methods with objects interacting with multiple Android APIs. For 23 of the random tests, multiple holes need to be completed. We ensured that the projects we evaluate on were not included in the training data.

Column (1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Analysis	No alias analysis			With alias analysis			With alias analysis	
Language model type	**3-gram**			**3-gram**			**RNNME-40**	**Combination**
Training dataset	1%	10%	all data	1%	10%	all data	all data	all data
Task 1 (20 examples)								
Desired completion in top 16	11	16	18	12	18	**20**	**20**	**20**
Desired completion in top 3	10	12	16	11	15	**18**	**18**	**18**
Desired completion at position 1	7	8	12	7	10	**15**	**14**	**15**
Task 2 (14 examples)								
Desired completion in top 16	3	5	7	10	10	**13**	**13**	**13**
Desired completion in top 3	3	4	6	8	8	**13**	**12**	**13**
Desired completion at position 1	3	3	5	6	6	**11**	**11**	**12**
Task 3 (50 random examples)								
Desired completion in top 16	13	27	36	21	43	**48**	**48**	**48**
Desired completion in top 3	13	23	32	18	34	**44**	**40**	**45**
Desired completion at position 1	13	16	25	14	25	**31**	**27**	**31**

Table 3.5: Accuracy of SLANG on the test datasets depending on the amount of training data, the analysis and the language model.

EXPERIMENTS We studied how the different knobs in our system affect the quality of code completion. We considered three accuracy metrics, based on the number of examples for which the:

1. desired method invocation is found in the list of results (we limit the size of the list to 16),
2. desired method invocation was found in the top 3 results, and
3. desired method invocation was ranked first in the suggested candidates list.

We evaluated a number of parameter choices and summarized the results in Table 3.5. Columns 2-7 contain the effect of the abstraction and the training data size. The system trained on the complete dataset with alias analysis is able to predict all examples in our first task, and the correct completion is in the first 3 results for 90% of the examples. Without alias analysis and with decrease of the training data, the accuracy significantly decreases and we can roughly quantify that using a better program analysis component has the same effect as adding an order of magnitude more data.

For our second task, one example could not be solved even by our best system, because SLANG was unable to collect sufficient information for the `Notification.Builder` class during training. The reason for this is that developers may use the class via a chain of calls `builder.setSmallIcon(_).setAutoCancel(_)` that make it difficult for an intra-procedural analysis to discover calls on the same object. We believe that adding a more advanced (inter-procedural) analysis could lead to further improvements of SLANG.

Two tests from the random tests task could not be solved by our best system. We believe one of them is due to a limitation of the partial compiler that prevented us to collect data for a class at training time, while the other is a completion that scores below the top 16 results.

LANGUAGE MODEL TYPES Columns 7, 8 and 9 of Table 3.5 summarize the effect of the language models (we compare the models with the full data size and with alias analysis). As discussed in Section 3.4, n-gram and RNN differ significantly in the way they express histories of sequences. While n-gram discover regularities between the last $n-1$ method calls, RNN is capable of discovering longer distance relations.

In our experiments, the two models differ in how they rank the completions for some tests: while RNNME-40 is better in examples with long distance relations similar to the one in Fig. 3.2, it occasionally misses some short distance relations. On the other hand, the 3-gram language model consistently finds all short distance relations. Because most histories per object are short, the 3-gram model outperforms the RNNME-40 model. The highest precision, however, is achieved with a combination model that averages the probabilities of these two models.

TYPE CHECKING ACCURACY To evaluate how many completions typechecked, we took our best combined system and manually inspected all of the 1032 possible completions that SLANG returned (for all of our examples). In this experiment, we found only 5 completions which did not typecheck and they were always among the worst ranked completions for the examples. We believe one of the reasons for such outlier completions to appear in the results is imprecision of the alias analysis at training time, which leads to impossible sequences in the model. To guarantee no type errors, one can add a typechecker on the results of SLANG that discards the bad solutions.

Part of the reason for the high typechecking accuracy is in the words of the model itself. The words in our language model are pairs of API names and positions. The API names for Java classes are in fact fully qualified names (i.e. include a package name, class name and a method name). Since our training data consists of sequences that typecheck and the words in the language model also include class names, our completions are heavily biased towards APIs that make the completions typecheck.

CONSTANT MODEL Our constant model worked reasonably well. Out of the 41 constants that needed to be inferred in the first two tasks, 25 were produced by SLANG as the first result and 3 as the second result. However, the prediction of certain constants is very difficult with statistical techniques: e.g., guessing URLs, passwords, etc.

COMPLETION SPEED Our query time was dominated by the time necessary to load the language model files. For our best system which combines the two language models, we observed average time per example of 2.78 seconds. To allow for interactive completions within an IDE, we plan to load language models only once at startup.

SUMMARY We have shown that SLANG is effective in completing partial programs with holes. Our experiments show that using alias analysis is important and has the effect of an order of magnitude more training data. Combining language models has positive effect on the ranking of the completions in our tests and with our best system, we return the correct completion as a first result in 58 out of 84 test cases.

3.8 RELATED WORK

Our work investigates the potential of techniques from natural language processing (n-gram language model and recurrent neural networks) in the context of programming tasks such as code completion of API calls. We show that language models alone are sub-optimal for consistently producing quality sequences of completions and show how to combine these ideas with classic programming languages concepts such as alias analysis. This combination significantly improves the quality of the result. We believe that such combinations of statistical methods with programming language techniques hold a great promise and are worth further exploration.

The work of Hindle et al. [61] is prior to ours and uses natural language techniques for code completion, but without using a static analysis component to capture semantic information in their model. While showing interesting experimental results, their evaluation confirms that a probabilistic model on a purely syntactic program representation as tokens is interesting, but insufficient to perform high accuracy predictions for programming tasks.

CODE COMPLETION AND SYNTHESIS The last few years have seen a renewed interest in various synthesis techniques which promise to simplify various software development tasks. Many of these techniques deal with some form of program "completion", typically by combining a predefined set of building blocks (e.g., expressions of some kind). For a broader survey in recent program synthesis techniques, see Gulwani [52]. Below, we briefly discuss the approaches that deal with completion of general user-level code. Prospector [86] is an approach which automatically discovers a sequence of API calls that transform an object of a given input type into an object of a given output type. PARSEWeb [128] also suggests a sequence of API calls but this time the search for the sequence is guided by the source code available on the

Web, thus helping to eliminate many otherwise undesirable sequences. Other works [56, 57, 99] focus on code completion by (statically) synthesizing expressions of a given type at a particular program point (these works examine the program context around that point). To find the most likely expressions desired by the programmer, these approaches also rely on ranking algorithms to handle the large numbers of potential candidates. As opposed to these (static) approaches, MatchMaker [139] synthesizes code based on observed API usage in dynamic executions of real-world programs.

code search and specification mining There has been a lot of work on *dynamic* specification mining (e.g., [7, 31, 138]), most of it for extracting various forms of temporal specifications. As always with dynamic analyses, the barrier to wide application of these approaches is the ability to execute code samples, and to obtain workloads that provide reasonable coverage. However, when they are applicable, our approach can benefit from such dynamic methods as an additional source of sentences provided to the training phase.

MAPO [140] uses static analysis to extract common API usage patterns. MAPO employs a simple static analysis followed by an algorithm for finding common sequences, which are later used for recommending code snippets to users. In contrast, our goal is to synthesize code completions, and we do so directly based on probability of sequences. The Strathcona [62] code recommendation system matches the structure of the code under development to the code in the examples. The query in this case is implicit and consists of the prefix of the currently written code. The search is performed over a sample repository (e.g., the existing project). Temporal information such as the order of method invocations is not considered.

Recently, [90] presented a typestate-based code search technique that is able to perform limited code completion. Their approach is based on an inherently expensive and limited abstract representation of automata. For instance, on 1% of the training data, it took [90] about 3 hours to complete (our system takes 5 seconds with a 3-gram model and 9 minutes with RNN). Further, Slang can complete parameters of method calls whereas [90] can only produce completions of method names. Upon manual inspection of the resulting automata mined by [90], 10 of the 20 examples in our set 1 were not even accepted by their automata, let alone ranked.

Technically, a key shortcoming of these clustering approaches is their limited ability to generalize to sequences that did not exist in the training data.

SYNTHESIS WITH PARTIAL PROGRAMS The concept of a partial program has proven effective in various synthesis contexts. Partial programs allow users to naturally express the parts about the program that they know, while leaving parts they are not sure about, empty. The synthesizer then automatically figures out how to complete the holes in a way that some property of the resulting program holds. Examples where partial programs are used heavily include the sketching approach [120] to program synthesis. In this line of work, the partial program is referred to as a "sketch", where typically, the programmer specifies a space of possible expressions which can be used to fill in the holes. The synthesizer then searches for completions that satisfy a given property. Partial programs, or templates, have also been effectively used for synthesis of various problems including classic sequential algorithms [122], bit-ciphers [121], and concurrent algorithms [132].

In this work, we also leverage partial programs as we believe they are an effective mechanism for capturing programmer's intent. However, fundamentally, unlike all of these works, we learn the candidate completions of a hole in the partial program by examining and leveraging the vast amount of data available on the Web (in our case, in the form of API usage). In the future, we believe that it will be fruitful to combine these two approaches: for instance, by leveraging the power of SMT solvers to infer fine-grained numerical expressions with our approach which can predict likely API completions and their parameters.

4 PROGRAM SYNTHESIS WITH NOISE

So far, we described several intermediate representations used for probabilistic models (e.g. graphs and sequences). These representations were manually designed based on knowledge about program analysis, intuition and experimental evidence. A key problem, however, is to determine if a representation is optimal and if it satisfies some desired properties such as high precision of the predictions. Later in Chapter 5, we will show an approach that synthesizes the best intermediate representation for a probabilistic model with respect to an entropy metric – a metric that is correlated with the amount of errors that a model will make. However, before we dive into the concrete problem for probabilistic models, we present a general framework for program synthesis with noise. This framework was motivated by our investigation in the "Big Code" space, but is applicable beyond that domain. For example, the framework can be used to enable existing program synthesizers such as [68] to handle noise.

programming by example The main idea of *programming by example* (PBE) is that instead of directly providing a program, the user provides a number of examples (e.g. by demonstrating the desired output for a given input). A key assumption of PBE is that the provided examples are sufficient to properly determine the desired program. One common way to satisfy this assumption is by letting the user interactively provide examples until the desired program is produced.

Indeed, in recent years there has been substantial interest in learning programs from examples (e.g., [6, 68, 80, 121]) with a vast amount of interesting new applications such as completing spreadsheet data [51], structuring data [11], generating bit manipulating programs [68], and others. These applications were successful as they give the power of a specialized programming language to users that only need to demonstrate the desired outputs of a program on a number of inputs.

However, many of these PBE techniques cannot adequately deal with incorrect examples as they attempt to satisfy *all* given examples, thus overfitting to the data. This means that when the user makes a mistake

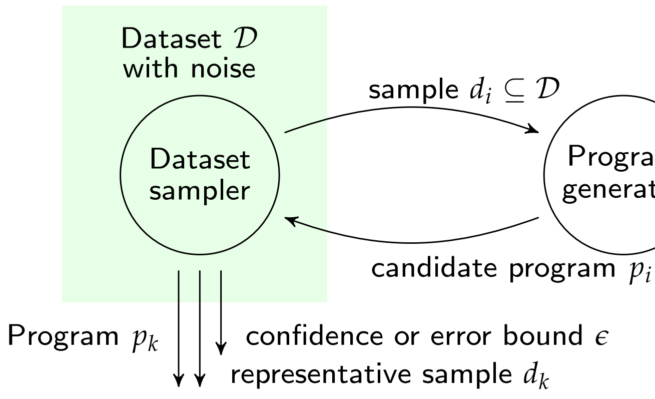

Dataset $\mathcal{D}$ with noise
Dataset sampler
sample $d_i \subseteq \mathcal{D}$
Program generator
candidate program p_i
Program p_k
confidence or error bound ϵ
representative sample d_k

Figure 4.1: General approach of program synthesis with noise.

while providing the examples, they either fail to return a program or produce the wrong program. Post-mortem ranking techniques do not help as they simply end up ranking incorrect solutions. Some synthesizers [51] have limited support to discover incorrect examples, but only if synthesis on all examples fails and with solutions specific to a particular domain of programs. Handling noise in the general case improves both existing PBE systems and enables synthesizing a model from a "Big Code" dataset that may contain errors.

THIS CHAPTER In this chapter, we propose a new approach for creating synthesizers that deal with noise, which also generalizes *counter-example guided inductive synthesis* (CEGIS) [121]. In the standard setting of noise-free synthesis, one is given a dataset $\mathcal{D}$ of examples that is very large (or even infinite) and the synthesizer selects a small set $d \subseteq \mathcal{D}$ such that synthesizing on d generates the desired program that could have been produced if it synthesized directly on $\mathcal{D}$. The dataset d is gradually built generating candidate programs p_i and by adding examples d that are not satisfied by the last generated program p_i. This procedure terminates when it generates a program that satisfies *all* input/output examples in $\mathcal{D}$.

As opposed to the standard setting, where all examples in $\mathcal{D}$ must be satisfied, our approach can handle incorrect input/output examples. Our technique is based on a loop that consists of: i) *dataset sampler* that carefully selects a small number of examples with specific properties from $\mathcal{D}$; ii) *program generator* that produces a program given the selected sample in a way which controls the complexity of the solution and avoids over-fitting. These two components are linked together in a feedback loop as shown in Fig. 4.1, iterating until the desired solution p_k is found. We show how this approach serves as a basis for constructing prediction engines given a noisy dataset $\mathcal{D}$.

In this chapter, we develop the general technique that extends the capabilities of PBE systems. This technique will then enable the synthesis of intermediate representation for "Big Code" where the input/output examples are a large corpus of partial programs and their completions (discussed later in Chapter 5).

QUANTIFYING NOISE To systematically instantiate our approach, we consider both cases for quantifying the noise in the dataset: the case where we have a bound on the noise and the case where the noise is arbitrary. In the first case, we also provide optimality guarantees on the learned program. In the second case, we approach the learning problem with a fast, scalable algorithm for performing approximate empirical risk minimization (ERM) [84], bridging the fields of applied program synthesis and machine learning. Our setting in fact raises new challenges from a machine learning perspective as here, ERM is performed not on the whole data at once as in traditional ML, but on carefully selected (small) samples of it.

SYNTHESIS WITH NOISE We illustrate how our concepts apply with synthesizers targeting the different noise settings. First, we present a synthesizer for bit-stream programs, called BitSyn, where we dynamically add examples, possibly with some errors, until we produce the desired program. This synthesizer shows that the approach is applicable outside of area of "Big Code" and shows a recipe to add noise handling to other existing PBE systems. Second, in Chapter 5, we present DeepSyn, a statistical synthesizer that learns probabilistic models from a dataset of programs and makes predictions (i.e., code completion) based on this model. Our engine generalizes several existing efforts (e.g., [61, 110] and Chapter 3) and is able to make predictions beyond the capability of these systems. Importantly, as our predictions are conditioned on program elements, they are easy to understand and justify to a programmer using DeepSyn, a capability missing in approaches where the prediction is based on weights and feature functions, and is not human understandable.

DETECTING NOISE While not the primary goal, this work represents a new way for performing *anomaly detection*: besides the learned program, our approach can return the set of examples d_k the program does not satisfy (these represent the potential anomalies). Our approach is unlike prior works which either assume the program is already provided [10] or make statistical assumptions on the data [29].

4.1 PROBLEM FORMULATION

Let $\mathcal{D}$ be a dataset consisting of a set of examples and $\mathbb{P}$ be the set of all possible programs. The objective is to discover a program in $\mathbb{P}$ which satisfies the examples in $\mathcal{D}$. In practice however, the dataset $\mathcal{D}$ may be imperfect and contain errors, that is, contain examples which the program should not attempt to satisfy. These errors can arise for various reasons, for instance, the user inadvertently provided an incorrect example in the dataset $\mathcal{D}$, or the dataset already came with noise (of which the user may be unaware of).

Because we are not dealing with the binary case of correct/incorrect programs and need to deal with errors, we introduce a form of a cost (risk) function associated with the program to be learned from the noisy dataset. Let $r \colon \mathcal{P}(\mathcal{D}) \times \mathbb{P} \to \mathbb{R}$ be a cost function that given a dataset and a program, returns a non-negative real value that determines the inferiority of the program on the dataset. In machine learning terms, we can think of this function as a generalized form of empirical risk (e.g., error rate) associated with the data and the function. In the special case typically addressed by PBE systems (e.g., [68, 80]), the function returns either 0 or 1, that is, the program either produces the desired output for all inputs in the given dataset, or it does not. Later in the thesis, we discuss several possibilities for the r function depending on the particular application.

PROBLEM STATEMENT The synthesis problem is the following:

$$\text{find a program} \quad p_{best} = \arg\min_{p \in \mathbb{P}} r(\mathcal{D}, p)$$

That is, the goal is to find a program whose cost on the entire dataset is lowest (e.g., makes the least number of errors, or minimizes empirical risk as in Section 5.1). We note that while in general there could be many programs with an equal (lowest) cost, for our purposes it suffices to find one of these. It is easy to instantiate this problem formulation to the specific binary case of synthesis from examples, where r returns 1 if some example in $\mathcal{D}$ is not satisfied and 0 otherwise. A challenge which occurs in solving the above problem is that the dataset $\mathcal{D}$ may be prohibitively large, or simply infinite (e.g., may need to continually ask a user for samples of the dataset) and thus, trying to directly learn the optimal program p_{best} that satisfies the dataset $\mathcal{D}$ may be infeasible.

Another issue in comparison to the traditional PBE formulation is that we may not be able to show that a candidate program p' is optimal, unless we can rank it with respect to *all* possible programs in $\mathbb{P}$, which could be prohibitively expensive. To mitigate this problem, we work with a more relaxed program statement where we search for a *satisfactory* program $P^{\approx best}$ that has a cost close to the cost of the best program p_{best} or is better (has lower cost) than a given noise bound.

4.2 ITERATIVE SYNTHESIS ALGORITHM

The key idea of our solution is to start with a small sample of the dataset $\mathcal{D}$ and to iteratively and carefully modify this sample in a way which allows finding a good solution with a few and small-sized samples. Our solution consists of two separate components: a *program generator* and a *dataset sampler*. We continually iterate between these two components until we reach a fixed point and the desired program is found.

PROGRAM GENERATOR For a finite dataset $d \subseteq \mathcal{D}$, a program generator is a function $gen \colon \mathcal{P}(\mathcal{D}) \to \mathbb{P}$ defined as follows:

$$gen(d) = \arg\min_{p \in \mathbb{P}} r(d, p)$$

We assume that invocations to $gen(d)$ are expensive and in our prediction algorithm we aim for a size of the dataset d that is as small as possible.

DATASET SAMPLER The second component of our approach is what we refer to as the *dataset sampler* $ds \colon \mathcal{P}(\mathbb{P}) \times \mathbb{N} \to \mathcal{P}(\mathcal{D})$:

$$ds(progs, n) = d' \texttt{ with } |d'| \geq n$$

That is, a dataset sampler takes as input a set of programs (and a bound on the minimum size of the returned sample) and produces a

```
   **Input**: Dataset $\mathcal{D}$, initial (e.g. random) dataset $\emptyset \subset d_1 \subseteq \mathcal{D}$
   **Output**: Program $p$
1  begin
2      $progs \leftarrow \emptyset$
3      $i \leftarrow 0$
4      repeat
5          $i \leftarrow i + 1$
6          // Dataset sampling step
7          if $i > 1$ then
8              $d_i \leftarrow ds(progs, |d_{i-1}| + 1)$
9          end
10         // Program generation step
11         $p_i \leftarrow gen(d_i)$
12         if $found\_program(p_i)$ then
13             return $p_i$
14         end
15         $progs \leftarrow progs \cup \{p_i\}$
16     until $d_i = \mathcal{D}$;
17     return "No such program exists"
18 end
```

Algorithm 2: Program Synthesis with Noise

set of examples which are then fed back into the generator. We will see several instantiations of the data sampler later in the thesis.

ITERATIVE SAMPLING We connect the program generator and data sampler components in an iterative loop. The resulting algorithm is shown in Algorithm 2. At every iteration of the loop, the algorithm checks if the current program p_i is a satisfactory solution and can be returned (in later sections, we discuss instantiations of *found_program*). If the current program p_i is not the right one, we sample from the dataset $\mathcal{D}$ using the current set of explored programs *progs*, obtaining the next dataset d_i. Note that while the size of the sample d_i is greater than the size of the previous sample d_{i-1}, there is no requirement that d_i is a superset of d_{i-1} (i.e., the sets may be non-comparable). Once we have obtained our new sample d_i, we use it to generate the new candidate program p_i. In case the program p_i is not the desired one, the algorithm continues and adds p_i to the *progs* set and continues iterating.

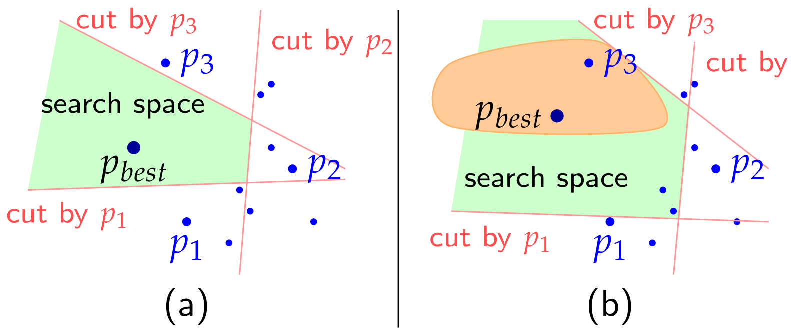


Figure 4.2: Trimming the space of programs (a) for noise-free synthesis and (b) for synthesis with noise.

Technically, CEGIS [121] is an instantiation of Algorithm 2 where the program generator $gen(d_i)$ invocation on line 11 always returns a program that satisfies all examples in d_i and the dataset sampler invocation on line 8 is such that the dataset $d_i \leftarrow d_{i-1} \cup \{x\}$ where x is an example not satisfied by the last generated program p_{i-1}.

4.2.1 *Reduction of Search Space*

First, note that Algorithm 2 always terminates if the dataset $\mathcal{D}$ is finite. This is because the size of the dataset d_i increases at every step until it eventually reaches the full dataset. However, our goal is to discover a good program using only a small dataset d_i. To achieve this, we leverage the dataset sampler to carefully pick small datasets that trim the space of possible programs as illustrated in Fig. 4.2.

NOISE-FREE SEARCH SPACE PRUNING Consider an initial dataset d_1. Since this dataset may be random, let us assume that any possible program p_1 can be returned as a result. If p_1 was not the desired program, we would like to select the next dataset d_2 to be such that p_1 cannot be returned at the next step by $gen(d_2)$. In general, we would like that at step i, $gen(d_i) \notin \{p_j\}_{j=1}^{i-1}$. This is, the program generated at step i is different from the previously generated programs (p_j for $j \in 1 \cdots i-1$). We illustrate this scenario in Fig. 4.2 (a). Here, we have three explored programs p_1, p_2 and p_3 which are pruned away by the current dataset. The figure also shows the space of remaining candidate programs (in 🟩) that can possibly be generated by *gen*. This space excludes all three generated programs as well as any programs removed as a result of pruning these three. Indeed, existing synthesis approaches that do not deal with noise (e.g., [68, 121]) typically prune the search space as shown in Fig. 4.2(a).

PRUNING SEARCH SPACE WITH NOISE Unlike the noise-free setting where a binary criteria for pruning a generated program p exists, when the data contains noise, we cannot immediately decide whether to prune p. The reason is that even though p may make a mistake on a given example, at an intermediate point in the algorithm we may not know whether another, even better program on $\mathcal{D}$ exists. What this uncertainty means is that we may need to keep p in the candidate program space for longer than a single algorithmic iteration. This raises the following question: which programs are kept and which ones are removed from the candidate set?

To address this question, at every iteration of the synthesis algorithm, we aim to prune some of the generated programs (and conversely, keep the remaining ones). In particular, we introduce a margin ϵ and we keep a generated program p if it is within ϵ distance of p_{best}. This is:

$$r(\mathcal{D}, p) \leq r(\mathcal{D}, p_{best}) + \epsilon \; . \tag{4.1}$$

In Fig. 4.2(b), the area shaded with 🟧 around p_{best} denotes the set of programs within distance ϵ of p_{best}. The space reduction process is illustrated in Fig. 4.2(b). Here, the first two explored programs p_1 and p_2 fall outside the accepted area and are thus permanently pruned from the candidate space. The score of the latest generated program p_3, however, is within ϵ of p_{best} and is thus kept as a viable candidate to be returned. At this point, the algorithm can return p_3 or keep searching further, hoping to find better scoring programs than p_3. We also require that $\epsilon \geq 0$ ensuring completeness: we always keep the best program p_{best} in the candidate space of programs.

In what follows, we describe dataset samplers which enable pruning of the search space in the manner described above.

4.2.2 *Hard Dataset Sampler (ds^H)*

We introduce an instance of the dataset sampler *ds* used in Algorithm 2 as follows:

Definition 4.1 (Hard dataset sampler)**.** A hard dataset sampler is a function ds^H such that for $Q \subseteq \mathbb{P}$, $d' = ds^H(Q, min\_size)$, it holds that $\forall p \in Q.\ r(\mathcal{D}, p) \leq r(d', p)$ and $|d'| \geq min\_size$.

Note that the hard dataset sampler always exists as we can trivially set $d' = \mathcal{D}$. In Algorithm 2, we always invoke the hard dataset sampler with $Q = progs$ (the current set of generated programs). The meaning of the hard dataset sampler is that for all programs in Q, the cost on the returned dataset d is higher or equal than on the full dataset $\mathcal{D}$.

There are two ways in which this definition generalizes the concept of providing more examples in CEGIS. This means that selecting a dataset in CEGIS is a hard dataset sampler. First, since CEGIS does not handle noise, $r(d, p)$ simply returns 0 if the program p satisfies all examples in d and 1 if p does not satisfy some example in d. The hard dataset sampler in the noise-free case generates (e.g., by asking questions to the user) a dataset d' such that for all explored programs $progs$, an unsatisfied example is in d' if an unsatisfied example exists in $\mathcal{D}$. Second, CEGIS works by building datasets $d_i \leftarrow d_{i-1} \cup \{x\}$ where x is one input/output example not satisfied by the last generated program p_{i-1}. In contrast, Definition 4.1 is more permissive allowing any dataset d_i to be returned by calling $d_i \leftarrow ds^H(progs, min\_size)$, not necessarily a superset of the dataset d_{i-1} from the previous iteration of Algorithm 2.

Using the hard dataset sampler, we now state a theorem which ensures that the generated program p_i at step i does not appear in a subset of the explored programs outside certain range beyond p_{best}. That is, p_i cannot be the same as any previously generated p_j that is outside of the 🟧 area in Fig. 4.2(b).

Theorem 4.2. *Let $Q = \{p_1, \ldots, p_{i-1}\}$ be the set of programs generated up to iteration i of Algorithm 2, where the dataset sampler ds satisfies Definition 4.1. If $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$, then $p_i = gen(d_i) \notin Q'$ where:*

$$Q' = \{\ p \in Q\ \mid\ r(\mathcal{D}, p) > r(\mathcal{D}, p_{best}) + \epsilon\ \}$$

Proof. First note that the set Q' includes all generated programs in Q that are outside the 🟧 area in Fig. 4.2(b), because the condition for a program from Q to remain in Q' is the inverse of (4.1).

Let $p \in Q'$. Then $r(\mathcal{D}, p) > r(\mathcal{D}, p_{best}) + \epsilon$.

From the definition of $\epsilon : \epsilon + r(\mathcal{D}, p_{best}) \geq r(d_i, p_{best})$,

Then $r(\mathcal{D}, p) > r(d_i, p_{best})$.

Because $d_i = ds(Q, \_)$ and ds satisfies Definition 4.1, $r(d_i, p) \geq r(\mathcal{D}, p)$. From the last two statement follows that $r(d_i, p) > r(d_i, p_{best})$.

Because $p_i = gen(d_i)$ and $gen(d_i) = \arg\min_{p' \in \mathbb{P}} r(d_i, p')$, it follows that $p_i \neq p$. Thus, we prove that $p_i \notin Q'$. □

The above theorem is useful if we know the bound ϵ on the best program p_{best} for any dataset $d \subseteq \mathcal{D}$ as required in the theorem precondition. The smaller the value of ϵ that we can show, the smaller the areas marked in 🟧 and 🟩 around p_{best} will be. Furthermore, from this theorem follows that if we stop the synthesis algorithm as soon as we generate a program p_i that is already in the explored set, the program p_i will be within a distance of at most ϵ from p_{best}.

In Section 4.3 we consider a scenario with $\epsilon = 0$. Then, Theorem 4.2 provides even stronger guarantees at every step of Algorithm 2: all previously generated candidate programs that are not p_{best} are eliminated from future consideration. For cases where we cannot obtain bounds on the best program p_{best}, we next define a different dataset sampler.

4.2.3 *Representative Dataset Sampler (ds^R)*

First, we define a measure of representativeness for dataset d with respect to the full dataset $\mathcal{D}$ on a set of programs $Q \subseteq \mathbb{P}$.

Definition 4.3 (Representativeness measure)**.**

$$repr(Q, \mathcal{D}, d) = \max_{p \in Q} |r(\mathcal{D}, p) - r(d, p)|$$

The measure of representativeness says how close are the costs of the programs in Q on the dataset d with respect to their costs on the full dataset $\mathcal{D}$. The metric is set to the maximum difference in costs since our goal for the dataset d is to be a representative of $\mathcal{D}$ for *all* programs. Then, we define a dataset sampler as follows:

Definition 4.4 (Representative dataset sampler)**.**

$$ds^R(Q, size) = \operatorname*{arg\,min}_{d \subseteq \mathcal{D}, |d| = size} repr(Q, \mathcal{D}, d)$$

We have defined the representative dataset sampler to return a dataset of exactly the size given by the *size* parameter, which for step i of Algorithm 2 is $|d_{i-1}| + 1$. Also, in Algorithm 2, a dataset sampler is always used with a set of programs $Q = progs$.

ANALYSIS Note that the *repr* measure is a non-negative function that is minimized by ds^R. If $d' = ds^R(Q, size)$ is such that $repr(Q, \mathcal{D}, d') = 0$ then the produced dataset is perfectly representative. In this case ds^R is also a hard dataset sampler, because $\forall p \in Q.\ r(\mathcal{D}, p) = r(d', p)$.

The question then is: why do we attempt to achieve $r(\mathcal{D}, p) = r(d', p)$ instead of $r(\mathcal{D}, p) \leq r(d', p)$ as in Definition 4.1? If we perform our analysis using Theorem 4.2, then we must find as small as possible value $\epsilon \geq 0$ such that $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$. In ds^R, instead of minimizing $r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$, we minimize $|r(d_i, p_j) - r(\mathcal{D}, p_j)|$ for programs p_j already explored up to step i (i.e. $j \in 1..i-1$).

The intuition of this requirement is that if an example is incorrect, it will likely behave similarly on all programs (e.g. it will be more difficult to satisfy and most programs will fail to satisfy it). Thus, if we find a small ϵ on several already explored programs, a similar bound may be true for all programs and for p_{best}. Next, we also give a formal argument about eliminating *some*, but not *all* of the already explored programs in the search process.

Theorem 4.5. *Let $Q = \{p_1, \ldots, p_{i-1}\}$ be the set of programs generated up to iteration i of Algorithm 2. Let $p_k = \arg\min_{p' \in Q} r(\mathcal{D}, p')$ be the best program explored so far. By definition, $p_k \in Q$. Let $\delta = repr(Q, \mathcal{D}, d_i)$ be the representativeness measure of d_i and d_i is obtained from a representative dataset sampler as $d_i \leftarrow ds^R(Q, size)$. Then $p_i = gen(d_i) \notin Q'$ where:*

$$Q' = \{\ p \in Q \ \mid\ r(\mathcal{D}, p) > r(\mathcal{D}, p_k) + 2\delta\ \}$$

Proof. Let $p \in Q'$. Because $Q' \subseteq Q$, $p \in Q$. Then, from Definition 4.3 because $\delta = repr(Q, \mathcal{D}, d_i)$ follows that $|r(d_i, p) - r(\mathcal{D}, p)| \leq \delta$. Then:

$$r(\mathcal{D}, p) \leq r(d_i, p) + \delta \tag{4.2}$$

Similarly, $p_k \in Q$. From Definition 4.3, $|r(d_i, p_k) - r(\mathcal{D}, p_k)| \leq \delta$. Then:

$$r(d_i, p_k) \leq r(\mathcal{D}, p_k) + \delta \tag{4.3}$$

Because $p_k = \arg\min_{p' \in Q} r(\mathcal{D}, p')$ and $p \in Q$, then

$$r(\mathcal{D}, p_k) < r(\mathcal{D}, p) \tag{4.4}$$

Then, we obtain that:

$$r(d_i, p_k) \overset{\text{from (4.3)}}{\leq} r(\mathcal{D}, p_k) + \delta \overset{\text{from (4.4)}}{<} r(\mathcal{D}, p) + \delta \overset{\text{from (4.2)}}{\leq} r(d_i, p) + 2\delta$$

Finally, from $r(d_i, p_k) < r(d_i, p)$ follows that $p \neq \arg\min_{p' \in \mathbb{P}} r(d_i, p')$ and as a result $p \neq p_i = gen(d_i)$. □

Note that the set Q' has the same shape as in Theorem 4.2 except that here we consider p_k (best program so far) instead of p_{best} (best program globally), and instead of ϵ we have 2δ.

What this theorem says is that the programs $Q' \subseteq Q$ that were already generated and are worse than $p_k \in Q$ by more than twice the representativeness measure δ of the dataset d_i cannot be generated at step i of Algorithm 2.

We can also instantiate the condition for cutting the space discussed earlier: $r(\mathcal{D}, p) \le r(\mathcal{D}, p_{best}) + \epsilon$ and visualize Theorem 4.5 in Fig. 4.2(b) as follows: take $p_3 = p_k$ and let $x = r(p_3, \mathcal{D}) - r(p_{best}, \mathcal{D})$ be the distance between p_k and p_{best}. Then take $\epsilon = x + 2\delta$. Thus, programs p_1 and p_2 are worse than p_{best} by more than ϵ and are permanently removed from the program search space.

In case $\delta = 0$, we can see that all programs in Q worse than the (locally) best program $p_k \in Q$ will be eliminated. Still, this is a weaker guarantee than for the case where $\epsilon = 0$ in Theorem 4.2. Later we will show that ds^R works well in practice, but in general it is theoretically possible that Algorithm 2 with ds^R makes no progress until a dataset of a certain size is accumulated.

4.2.4 *Cost Functions and Regularization*

So far, we have placed few restrictions on the cost function r and we defined the synthesis problem to minimize the cost of a program p on a dataset d. We now list concrete cost functions that we consider later in the thesis:

- $num\_errors(d, p)$ returns the number of errors a program p does on a dataset of examples d.

- $error\_rate(d, p) = \frac{num\_errors(d,p)}{|d|}$ is the fraction of the examples with an error. A related metric used in machine learning is the *accuracy*, which is $1 - error\_rate$.

- Other measures weight the errors done by the program p on the dataset d according to their kind (e.g., entropy is one possible such measure).

The choice of cost metric is key for the usefulness of the theorems stated before. For example, if the metric is *num_errors* and the dataset

$\mathcal{D}$ includes n incorrect examples, a representative (or hard) dataset sampler that returns up to n elements may simply return a set with only incorrect examples. A sample of size $n+1$ is required to ensure that at least one correct example is returned. In contrast, metrics like error rate may not suffer from this problem with a representative dataset sampler, because the ratio of errors in the small dataset should be approximately the same as the ratio of errors on the full dataset.

REGULARIZATION We also use a class of cost functions known as regularized cost metrics. If r is a cost metric, its regularized version is $r_{reg}(d, p) = r(d, p) + \lambda \cdot \Omega(p)$. Here, λ is a real-valued constant and $\Omega(p)$ is a function referred to as a regularizer. The goal of the regularizer is to penalize programs which are too complex and prevent overfitting to the data. Note that the regularizer *does not* have access to the dataset d, but only to the given program p. In practice, using regularization means we may not necessarily return the program with the least number of errors if a much simpler program with slightly more errors exists. In Section 5.1.1, we justify the use of regularization in the context of empirical risk minimization. To the best of our knowledge, this is the first work that uses regularization for program synthesis.

4.3 THE CASE OF BOUNDED NOISE

In this section, we show how to instantiate Algorithm 2 for the case where we can define a bound on the noise that the best program p_{best} exhibits.

Definition 4.6 (Noise Bound)**.** We say that ϵ_k is a noise bound for samples of size k if for the program p_{best}:

$$\forall d \subseteq \mathcal{D}. |d| = k \implies \epsilon_k \geq r(d, p_{best}) - r(\mathcal{D}, p_{best})$$

For example, if $r \triangleq \mathit{error\_rate}$ and $\mathcal{D}$ contains at most one incorrect example, then $\epsilon_{10} = 0.1$ is a noise bound, because for any sample $d \subseteq \mathcal{D}$ of size $|d| = 10$, the error rate is at most 0.1. Note that the error rate on the entire dataset $\mathcal{D}$ or on larger datasets is lower than 0.1. Another interesting case is if $r \triangleq \mathit{num\_errors}$ and p_{best} has at most K errors on the examples in $\mathcal{D}$. Then a noise bound for any k is $\epsilon_k = 0$ because no dataset $d \subseteq \mathcal{D}$ has more errors than the full dataset $\mathcal{D}$. Note that using regularization is an orthogonal issue and does not affect the noise bound, because the regularizer $\Omega(p_{best})$ cancels out in the inequality of Definition 4.6.

We can easily instantiate Theorem 4.2 when a noise bound ϵ_k is available by setting $\epsilon = \epsilon_k$ in the theorem's precondition $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$ (here, $k = |d_i|$).

DERIVED TERMINATION CRITERION Using Theorem 4.2 and the hard dataset sampler allows us to derive a possible termination criterion for Algorithm 2. In particular, if a satisfactory program is such that $r(\mathcal{D}, p_{satisfactory}) \leq r(\mathcal{D}, p_{best}) + \epsilon_{satisfactory}$ (i.e., it is worse than the best program by at most $\epsilon_{satisfactory}$), then the following stopping criterion:

$$\mathit{found\_program}(p_i) \triangleq (p_i \in \mathit{progs}) \wedge \epsilon_{|d_i|} \leq \epsilon_{satisfactory}$$

in Algorithm 2 will produce a satisfactory program. Here $\epsilon_{|d_i|}$ is a bound known according to Definition 4.6 for the dataset d_i at the i-th iteration of the algorithm. This criterion follows from Theorem 4.2 because if a program $p_i \in \mathit{progs}$ (i.e., it was already explored previously), then p_i was not excluded from the search space and thus it must be that:

$$r(\mathcal{D}, p_i) \leq r(\mathcal{D}, p_{best}) + \epsilon_{|d_i|} \qquad \text{(i.e., } p_i \in \text{🟧).}$$

Then, the returned program p_i is satisfactory because $\epsilon_{|d_i|} < \epsilon_{satisfactory}$.

BOUND ON THE NUMBER OF ERRORS We next consider an interesting special case where we know that for the best program p_{best}, there are at most K incorrect examples in $\mathcal{D}$ that it does not satisfy. Note that we only need to know a bound, not the exact number of errors the best program makes. In this case, we propose to use the following cost function:

$$r_K(d, p) = \min(\mathit{num\_errors}(d, p), K + 1)$$

That is, we count the number of unsatisfied examples and cap the cost at $K + 1$, thus we do not distinguish programs or datasets with more than K errors. Since we know that the best program has at most K errors, in Definition 4.6, we can show that $\epsilon_k = 0$ (for any k) is a valid bound. In this case, we can also obtain a stopping criterion with $\epsilon_{satisfactory} = 0$ by using:

$$\mathit{found\_program}(p_i) \triangleq p_i \in \mathit{progs}$$

Thus, we get the stronger guarantees as in Fig. 4.2 (a) and ensure that upon termination the algorithm produces the program p_{best}.

DISCUSSION We note the meaning of the hard dataset sampler when $r \triangleq num\_errors$. According to Definition 4.1, $r(d, p) \geq r(\mathcal{D}, p)$ holds for all p generated up to iteration i of Algorithm 2. This means that the sample d_i must contain *all* errors in $\mathcal{D}$ – naturally, this may lead to a dataset that is too big. If we know an upper bound K on the number of errors of the best program p_{best} and use a cost function $r \triangleq r_K$, then the sampler will need to include exactly $K + 1$ unsatisfied examples in order to eliminate p_i as a candidate for the next step. In this setting, knowing a bound K of p_{best} in advance enables a cost metric that makes the procedure of the algorithm with hard dataset sampler more efficient.

Next, we illustrate the concepts by showing how existing synthesizers can be extended to deal with noise. We present a synthesizer for bitstream programs with a bound on the number of errors.

4.4 BITSYN: BITSTREAM PROGRAMS FROM NOISY DATA

In this section we instantiate the approach presented earlier to the problem of building a programming-by-example (PBE) engine able to deal with up to K incorrect input/output examples in its input data set for the best program p_{best}. To illustrate the process, we chose the domain of bitstream programs as they are well understood and easy to implement, allowing us to focus on studying the behavior of Algorithm 2 in a clean manner. We believe many synthesis engines are good candidates for being extended to deal with noise (e.g., synthesis of floating point functions [98] or data extraction [81]).

THE SETTING We consider two scenarios: (1) the dataset $\mathcal{D}$ is obtained dynamically and the noise is bounded (i.e., up to K errors), and (2) the dataset $\mathcal{D}$ is present in advance and may contain an unknown number of errors. Interestingly, the second scenario is useful beyond synthesizing programs, in this case, for anomaly detection.

We created a synthesizer called BitSyn that generates loop-free bit manipulating code from input/output examples. The programs generated by BitSyn are similar to those produced in Jha et al [68]. We use a library of instructions for addition, bitwise logical operations, equality, less than comparison and combine them into a program that takes 32-bit integers as input and outputs one 32-bit integer. The program may use registers to store intermediate values. The goal of the synthesizer is to take a number of input/output examples and generate a program.

	Number of instructions	**Number of errors (K)**																
		0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	9
Program	↓	Number of input/output examples needed										Synthesis time (seconds)						
P1	2	4	4	10	7	9	11	14	16	17	22	1.11	1.17	1.98	1.51	1.80	7.33	102.76
P2	2	5	6	6	7	11	12	15	19	20	22	1.21	1.48	1.79	2.70	2.45	12.96	72.37
P3	3	4	4	9	10	8	13	15	16	17	21	1.75	1.81	4.42	8.63	9.20	40.62	156.09
P4	2	2	4	7	8	9	10	13	15	17	19	1.05	1.19	1.56	3.07	4.01	11.34	12.30
P5	2	3	3	9	9	10	10	14	16	20	22	1.08	1.10	1.84	3.45	9.38	11.64	139.75
P6	2	4	5	10	9	10	11	13	17	20	22	1.18	1.51	2.70	3.50	10.60	12.44	91.49
P7	3	5	5	7	9	11	12	15	19	20	22	1.80	2.20	2.77	5.15	12.65	21.62	117.16
P8	3	5	5	10	10	8	12	13	16	20	19	1.90	2.44	4.41	4.47	5.15	26.62	41.46
P9	3	3					timeout					2.58	timeout	timeout	timeout	timeout	timeout	timeout

Table 4.1: Number of input/output examples needed by BitSyn to synthesize the correct program (program taken from [68, 134]) depending on the number of errors in the examples as well as the synthesis time with the respective number of errors.

4.4.1 *Program Generator with Errors*

A key quality of BitSyn is that it includes a program generator that may not satisfy all provided input/output examples. This may serve multiple purposes as we discuss later. Let us consider the following example input/output pairs:

$$d_1 = \{\{2 \to 3\}, \{5 \to 6\}, \{10 \to 11\}, \{15 \to 16\}, \{-2 \to -2\}\}$$

All examples except for $\{-2 \to -2\}$ describe a function that increments its input. A problem with existing PBE engines in this case is that they *succeed* in generating a program even if it was not the desired one, e.g. by producing the following code:

$p_a =$ `return input + 1 + (input >> 8)`

Note that providing more examples would not necessarily help discover or solve this problem. The user may in fact get *lucky* by getting into a situation where the synthesizer fails to produce a program, however if the hypothesis space of programs (e.g., which operators is the engine allowed to use) is not very constrained, this program can overfit to the incorrect examples. Later we quantify this problem. The problem of overfitting to the data (i.e., input/output examples) occurs in multiple mathematical and machine learning problems where the provided specification does not permit exactly one solution, for example when dealing with noise.

We combat overfitting by introducing regularization to the cost. We define a function $\Omega\colon \mathbb{P} \to \mathbb{R}^+$ that punishes overly complex programs p by returning the number of instructions used. For our example, $\Omega(p_a) = 3$ since p_a has three instructions (two + and one >>). Then, we create a program generator that minimizes:

$$r_{reg}(d, p) = error\_rate(d, p) + \lambda \cdot \Omega(p).$$

The value $\lambda \in \mathbb{R}$ is a *regularization constant* that we choose in evaluation. The higher the regularization constant is, the more importance we place on producing small programs. In our example, if $\lambda > 0.1$, the cost r_{reg} of the following p_b program will be lower than the cost of p_a on the dataset d_1:

$p_b =$ `return input + 1`

IMPLEMENTATION OF BITSYN We implemented BitSyn using the Z3 SMT solver [35]. At each query to the SMT solver, we encode the set of input/output examples $d = \{x_i\}_{i=1}^{n}$ in a formula based on the techniques described in [68]. In each formula given to the SMT solver, we encode the length of the output program and we additionally encode a constraint for the number of allowed errors. Let χ_i be a formula that is true iff example $x_i \in d$ is satisfied. Then, to encode a constraint that allows up to T errors, we must satisfy the following formula:

$$\Upsilon \quad \equiv \quad T \geq \sum_{i=1}^{n} \text{if } \chi_i \text{ then } 0 \text{ else } 1$$

To find the best scoring solution, we make multiple calls to the SMT solver to satisfy Υ by iterating over the lengths of programs and the number of allowed incorrect input/output examples T ordered according to the cost of the solution and then return the first obtained satisfiable assignment of instructions.

4.4.2 *Case 1: Examples in $\mathcal{D}$ are provided dynamically*

A common scenario for programming-by-example engines (and for CEGIS) is when the input/output examples are obtained dynamically, either by interactively asking the user or by querying an automated reasoning engine (e.g. oracle-guided [68]). Ultimately, this means that the entire dataset $\mathcal{D}$ is not directly observable by the program generator (in fact, the dataset may be infinite). In this case, the synthesizer starts with a space of candidate programs and narrows that space by dynamically obtaining more examples.

For this setting, we designed a *hard dataset sampler* using the cost function r_K as described in Section 4.3. Our dataset sampler attempts to create a dataset d_{i+1} with $K+1$ unsatisfied examples for each program in the set of candidate programs explored so far $progs = \{p_j\}_{j=1}^{i-1}$. Generally, incorrect examples can be readily obtained automatically from an SMT solver (or another tool). When the tool is used interactively, the user needs to answer questions until the desired number of errors is reached (that is, here, the user takes an active part in the work of the dataset sampler).

EVALUATION Our goal was to check if BitSyn can synthesize the correct program in the presence of errors. Towards this, we implemented a simulated user that provides examples using a hard dataset sampler with a known bound on the incorrect examples. We aimed to answer the following research questions:

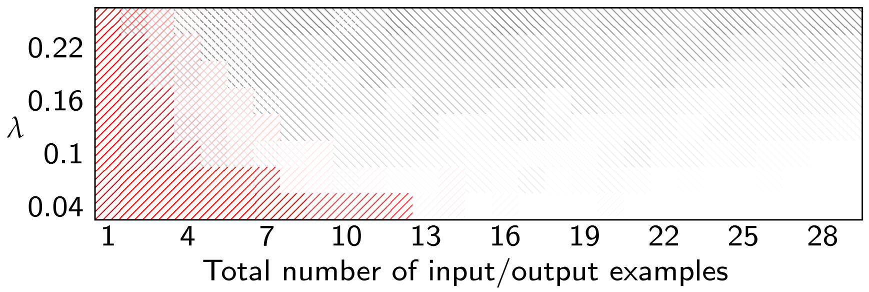
- Up to how many errors does BitSyn scale for synthesizing solutions?
- How many (more) examples does BitSyn need in order to compensate for the incorrect examples?

For our evaluation, we took a number of programs from [68], on which an existing synthesizer without noise could generate solutions within a few seconds. These are the programs P1-P9 from the Hacker's Delight book [134], also evaluated in previous works [53, 68].

We summarize our results in Table 4.1. For each program, we tried settings with different numbers of incorrect examples. We first supplied the incorrect examples and then started supplying correct examples. In each cell in the left part of Table 4.1, we list the *total* number of input/output examples needed to obtain the correct result. In the right part of Table 4.1, we list the time needed to complete each synthesis task. Our results can be summarized in two areas: (1) overall, adding incorrect input/output examples complicates the program synthesis task. For tasks $P1 - P8$, each synthesis task completes within our timeout of 300 seconds. Task $P9$ did not scale since it needs bit-shift operations and their presence leads to difficult formulas for the Z3 solver, and (2) the number of necessary input/output examples overall increases with an increased number of errors, but only slightly. Further, in some cases, the number of needed examples stays constant when introducing more errors. This motivated us to ask the question explored next, which is whether the tool is useful beyond synthesis, but also for detecting incorrect examples (i.e., anomaly detection).

4.4.3 *Case 2: All examples in $\mathcal{D}$ are given in advance*

As a side question not directly related to Algorithm 2, we wanted to understand how well the regularized program generator in BitSyn detects incorrect examples in the setting where the dataset $\mathcal{D}$ is fully available. Here, we provide all our examples to a regularized generator and ask if the unsatisfied examples are exactly the incorrect ones. Such a setting of finding a model that describes data and then detects outliers in the



▨ No example removed (overfitting to incorrect example)

▧ Some correct examples removed (too simple program)

☐ Noisy example correctly detected

Figure 4.3: Ability of BitSyn to detect an incorrect example for programs (P1-P9) depending on total number of examples and regularization constant λ.

model is called anomaly detection [29]. Note that our approach is very different from recent work [10] which considers a more restricted case where the program is already fully available before the process of anomaly detection starts.

EVALUATION We evaluate the anomaly detection capability of BitSyn depending on the regularization constant λ and the number of samples present in $\mathcal{D}$. For this experiment, we created data sets $\mathcal{D}$ of various sizes and introduced an incorrect example in each of them. Then, we looked at how often the synthesized program did not satisfy exactly the incorrect examples. We summarize the results in Fig. 4.3. The figure visualizes the conditions under which the anomaly detection is effective on our test programs (P1-P9). Every cell in the diagram of Fig. 4.3 says how often a given output occurs. The cells with ▨ denote that the synthesizer successfully satisfied the provided examples, including the incorrect one. This case typically occurs with no or low regularization. This means that synthesizers that fail to take noise into account will easily overfit to the incorrect example and return valid, but incorrect programs.

On the other hand, using too much regularization may bias BitSyn towards producing too simple programs that do not satisfy even some of the correct examples. We denote this with the pattern ▧. The darker a pattern, the more often the corresponding issue occurs. In the white areas of the graph, BitSyn reliably discovers the incorrect input/output example. For this to happen reliably, our results show that we need a dataset with more than 10 examples and a regularization constant λ between 0.05 and 0.1.

4.5 RELATED WORK

Below, we survey works related to ours.

BOOLEAN PROGRAM SYNTHESIS Over the last few years, there has been an increased interest in various forms of synthesis. Examples of recent techniques include synthesis from examples [68], partial programs [121] and synchronization [133]. A more detailed survey of the various approaches can be found here [52]. Generally, however, these are approaches which attempt to satisfy *all* provided examples and constraints. Thus, they typically overfit to the data, and as a result, incorrect examples in the data set will lead to incorrectly learned programs (or no programs at all). Some approaches come with various fine-tuned ranking functions which assign preference to the (potentially many) synthesized programs. However, regardless of how good the ranking function is, if the data set contains even one wrong example, then the ranking function will be of little use, as it will simply rank incorrect programs. In contrast, our approach deals with noise and is able to synthesize desirable programs even in the presence of incorrect examples. It achieves that via both, the usage of regularizers which combat over-fitting, and smart, iterative sampling of the entire data set. As we showed in Chapter 4, a standard all-or-nothing synthesis approach can be extended to incorporate and benefit from our techniques.

The work of Menon et al. [88] proposes to speed-up the synthesis process by using machine learning features that guide the search over candidate programs. This approach enables a faster decision procedure for synthesizing a program, but requires all provided input/examples to be satisfied.

QUANTITATIVE PROGRAM SYNTHESIS Another line of work is that of synthesis with quantitative objectives [28, 30]. Here, it is possible to specify a quantitative specification (e.g., a probabilistic assertion) and to synthesize a program that satisfies that weaker specification while maximizing some quantitative objective. In our setting, one can think of the dataset $\mathcal{D}$ as being the specification, however, we essentially learn how to relax the spec, and do not require the user to provide it (which can be difficult). Further, our entire setting is very different, from the iterative sampling loop, to the fact that even if the specification can be fully satisfied, our approach need not satisfy it (e.g., due to regularization constraints). In the future, it may be useful to think of ways to bridge these directions.

4.6 SUMMARY

In this chapter, we introduced a program synthesis approach that can deal with incorrect examples. This approach is based on a feedback loop between a dataset sampler and a program generator. We have shown an instantiation of the program generator that deals with noise using a regularization function. We introduced two variations of the dataset sampler and analyzed the case where there is a known bound on the errors of the best program. In this case, we have shown that our algorithm terminates early and always returns an optimal solution.

Our approach is applicable not only for counting incorrect input/output examples, but also for other metrics (e.g. error rate, entropy, metrics where the examples are weighted, etc.). In this case, we provide optimality guarantees, if we know a bound on the cost function for the best program (Definition 4.6). Finally, in the most general case where there are no error bounds, we do not provide an early termination condition for the algorithm, but we have shown that some suboptimal candidate programs are removed from the search space (Theorem 4.5).

We believe this is the first comprehensive work that deals with the problem of learning programs from noisy datasets, and represents an important step in understanding the trade-offs arising when trying to build program synthesis engines that deal with incorrect examples. Based on the techniques presented here, one can also investigate how to adapt and extend many of the existing programming-by-example and synthesis engines to deal with noise. We provide additional resources such as source code and test data online at

`http://www.srl.inf.ethz.ch/noise`.

5 LEARNING A SYNTHESIZER WITH "BIG CODE"

In previous chapters, we manually designed intermediate representations tailored to specific tasks and programming languages. When developing a name or type annotation predictors, we motivated the need to use conditional random fields and factor graphs. However, the specifics of the factor graphs such as their edges and feature functions were empirically designed as described in Section 2.3. There, we listed a number of features that relate program elements according to their distance in the program abstract syntax trees. Similarly, for code completion we proposed a statistical n-gram language model that parametrizes a predicted method call on the $n-1$ method calls preceding the predicted position. There, we defined program analysis (Section 3.3) to determine and extract those sequences using program analysis. Since there was no requirement for soundness or precision on the program analysis, the space of possible analyses was essentially unrestricted and we picked the best model empirically. We could not know that our choice of intermediate representation for these problems was optimal for the given task. In fact, a number of recent works keep proposing new intermediate representations without asking this question [55, 61, 83, 85, 94, 95, 110].

In this chapter, we address the problem of choosing an intermediate representation for a "Big Code" system and formulate it as a program synthesis problem. To solve the synthesis problem, we apply the concepts developed in Chapter 4 to:

- handle noise in the training data (e.g. some programs from the "Big Code" contain bugs or are badly written and should be ignored), and
- perform fast and scalable search for the best intermediate representation.

We then combine these concepts with the probabilistic models defined in the previous chapters.

Previously, the training procedure with hard-coded intermediate representations followed the procedure shown in Fig. 5.1 (a). There, a

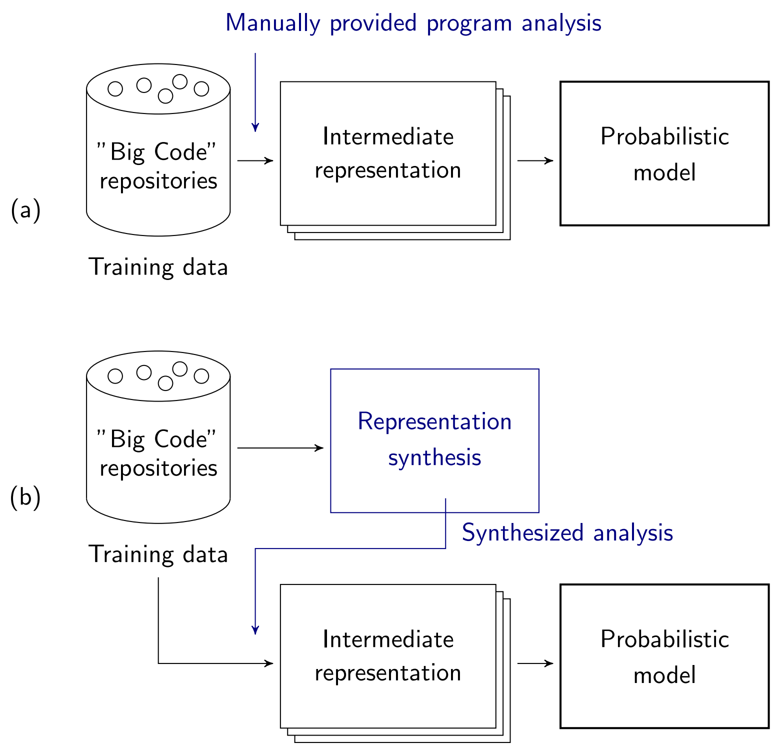


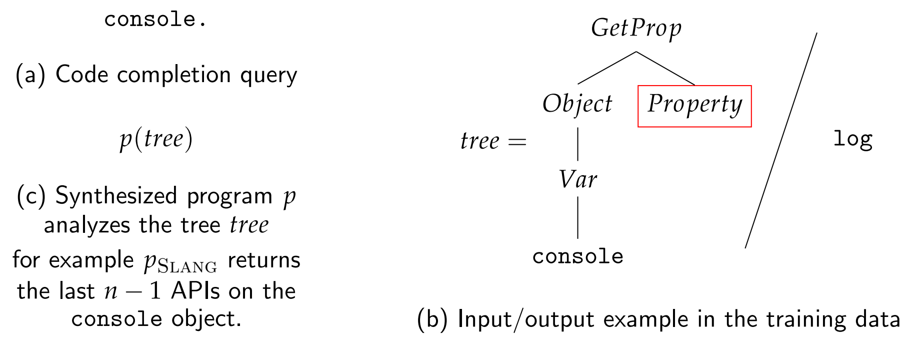
Figure 5.1: Architecture variants of tools learning tools from "Big Code". (a) Learning with a manually designed analysis. (b) Learning with synthesized analysis.

manually provided analysis determines the actual intermediate representation that is then fed to a machine learning model. In this chapter we propose to add a component that would synthesize the best intermediate representation such that the overall precision of the model is maximized. Overall, we show the high-level architecture of a system with such a component in Fig. 5.1 (b). Typically, the space of possible intermediate representations is very large in order to capture interesting insights that help improve the precision of the systems, and not just tuning parameters. As a result, we define each such intermediate representation by a program p drawn from a domain-specific language, which defines a set of programs $\mathbb{P}$. Essentially, p is the program that performs program analysis on the "Big Code" and then on each prediction query.

In this chapter, we instantiate this idea to the same code completion problem as in Chapter 3. For that application, the SLANG API code completion system is essentially described by one possible $p_{\textsc{Slang}} \in \mathbb{P}$. As a result, we find even better intermediate representation $p_{\approx best}$ and show it to be far more precise than $p_{\textsc{Slang}}$ and other models for the challenging task of JavaScript code completion. This is a particularly hard task since existing approached based on types, or other static analyses fail to produce results due to the dynamic nature of the language (which makes static analyses unsound and intractable).

To estimate the precision of an intermediate representation and a model, we take part of the training data, build a model and then evaluate on the rest of the training data. Ideally, we want to find an intermediate representation $p \in \mathbb{P}$ such that the predictions done in our evaluation are correct. That is, finding an intermediate representations $p \in \mathbb{P}$ now becomes a program synthesis problem.

TERMINOLOGY Throughout this chapter, we will synthesize intermediate representations about programs from a "Big Code" corpus in order to statistically solve a programming problem. Since the word *program* is highly overloaded, in the context of code completion we use the terminology illustrated in Fig. 5.2. When we refer to the "Big Code" repository, we say it consists of input/output examples as illustrated in Fig. 5.2 (b). Every input of these examples is a *tree* (since we use an abstract syntax tree representation of the code). When we refer to a *program p*, this is a program that analyzes trees and computes an intermediate representation used for a probabilistic model. Then, the


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.
GetProp
Object
Property
tree =
Var
console
log
(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`.

Figure 5.2: Terminology for "Big Code" systems based on program synthesis.

model m_p based on that intermediate representation for the training data can answer code completions queries such as Fig. 5.2 (a) as shown in Fig. 5.2 (d). Note that the program p operates on trees, but does not directly return the completion for the query in Fig. 5.2 (b). Instead, it parametrizes a probabilistic model that returns the actual completion.

We next proceed with defining the program synthesis problem and then in Section 5.2 we discuss the full code completion solution called DeepSyn. In Section 5.3 we evaluate DeepSyn on a large corpus of JavaScript code.

5.1 INDUCTIVE SYNTHESIS FOR EMPIRICAL RISK MINIMIZATION

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

Let $\mathcal{D}$ be the set of input/output examples from the "Big Code" as previously defined for our application. Recall that $\mathbb{P}$ is the set of programs that describe intermediate representations for our probabilistic model. Our key idea is to view the problem of learning a program $p \in \mathbb{P}$ from a noisy dataset $\mathcal{D}$ as an empirical risk minimization (ERM) task over a discrete search space of programs. Then, we will perform approximate ERM by formulating it as a program synthesis problem.

Next, we first review the concept of ERM and the guarantees provided by statistical learning theory. In this case, we restrict $\mathcal{D}$ to be a finite set obtained from a probability distribution $\mathcal{S}$ over examples $\mathcal{X}$, that is, $\mathcal{D} \subseteq \mathcal{X}$. Intuitively, $\mathcal{X}$ is the full specification of a problem as a possibly infinite set of input/output examples and $\mathcal{D}$ is a finite set that we use to observe $\mathcal{X}$. In what follows, our cost function r is set to be the empirical risk function (discussed below). Then, in Section 5.1.2, we present our (novel) approach to performing approximate ERM.

5.1.1 *Empirical Risk Minimization*

Let $\ell \colon \mathbb{P} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ be a function, such that $\ell(p, x)$ quantifies the loss (amount of inaccuracy) when applying program p to example x. Later in Section 5.2.4 we show an example of a loss function. Our task is to synthesize a program $p^* \in \mathbb{P}$ that minimizes the expected loss on example x drawn i.i.d. from distribution $\mathcal{S}$ (we assume w.l.o.g. $\mathcal{S}(x) > 0$ for all $x \in \mathcal{X}$). 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:

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

As a concrete example, $\ell(p, x)$ could be 0 if p produces the "correct" output for x and 1 if it is incorrect. In this setting, $R(p)$ corresponds to the expected number of mistakes that p makes on random example x. Moreover, $R(p) = 0$ iff it is "correct" (i.e., produces the correct behavior on all examples in $\mathcal{S}$). As another example, p could produce real-valued outputs, and $\ell(p, x)$ could measure the squared error between the correct output and the actual output.

There are two problems with computing p^* using the above approach. First, since $\mathcal{S}$ is unknown, the risk $R(p)$ cannot even be evaluated. Second, even if we could evaluate it, finding the best program is generally intractable. To address these concerns, we make two assumptions. First, we assume we are given a *dataset* $\mathcal{D}$ of examples drawn i.i.d. from $\mathcal{S}$. 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

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

The above equation is in fact an instance of the problem stated in Section 4.1.

GUARANTEES Standard arguments from statistical learning theory [84] now guarantee that, for any $\varepsilon, \delta > 0$, if our dataset of examples $\mathcal{D}$ is big enough with respect to the space of programs ([91, Chapters 7.3,7.4]), then it holds for the solution p_{best} that $R(p_{best}) \leq R(p^*) + \varepsilon$, with probability at least $1 - \delta$ (over the random sampling of the dataset) [91]. Hence, the best-performing program on the dataset is close (in risk) to the best program over all of $\mathcal{S}$. This is because under the above conditions, the empirical risk approximates the true risk uniformly well, i.e., for all $p \in \mathbb{P}$ it holds that $|R(p) - r_{emp}(\mathcal{D}, p)| \leq \varepsilon$.

REGULARIZATION ERM solution can overfit if the dataset $\mathcal{D}$ is not large enough [84]. Overfitting means that $R(p^*) \ll R(p_{best})$, i.e., the ERM solution has much higher risk than the optimal program. This often happens when the space of programs $\mathbb{P}$ under consideration is very complex, i.e., the solution could overfit by "memorizing" the training data and fail on other examples. 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, $\Omega : \mathbb{P} \to \mathbb{R}_{\geq 0}$ is a function (called *regularizer*), which prefers "simple" programs. For example, $\Omega(p)$ could count the number of instructions in p, i.e., a program is "simpler" if it contains fewer instructions. Note that the regularizer does not depend on the data set, it only depends on the program. The *regularization parameter* λ, which controls the strength of our simplicity bias, is usually optimized over using a process called *cross-validation*. In the following, we use the notation r_{emp} and refer to minimizing it as ERM, whether or not we are applying regularization.

5.1.2 *Using Representative Dataset Sampler*

The complexity of solving ERM is heavily dependent on the size of the dataset $\mathcal{D}$. This is due to the fact that evaluating r_{emp} or r_{reg} gets more expensive (since we need to sum over more examples).

To enable ERM on the large dataset $\mathcal{D}$, we use Algorithm 2 with a representative dataset sampler ds^R and a program generator that solves ERM on small datasets. Our goal here is to sample subsets $d_1, d_2, \ldots d_m \subseteq \mathcal{D}$, with the property that solving ERM on these subsets leads to good solutions in terms of the (intractably large) dataset $\mathcal{D}$. Our goal upon termination of the synthesis procedure from Algorithm 2 is to obtain a program p_m for which:

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

Obtaining such a bound ε' of the produced solution p_m with respect to p_{best} is what we investigated in Chapter 4. Then, recall that $R(p_{best}) \leq R(p^*) + \varepsilon$ to obtain that the resulting solution p_m will have risk at most $\varepsilon + \varepsilon'$ more than p^*. On the other hand, 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 $\mathcal{D}$ (which can also be practically infeasible). This instantiation is a new approach of performing approximate ERM over discrete search spaces.

5.2 deepsyn: learning statistical code completion systems

In this section we present a new approach for constructing statistical code completion systems. Such statistical code completion systems are typically trained on a large corpus of programs and are used to generate a (probabilistically) likely completion of a given input program. Currently, the predictions made by existing systems (e.g., [61, 94] or Slang from Chapter 3) are "hard-wired" (see Section 5.4 for further discussion) with certain insight that determines their expressiveness and precision. Improving the precision of these systems requires changes to this hard-wired intermediate representation for different kinds of predictions. The ideas presented here for automatically synthesizing an intermediate representation cleanly generalize these existing approaches.

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. However, unlike the first-order setting described in Section 4.4 where the data is simply a set of input/output examples and the learned program tries to explain these examples and predict new examples, the learned program in this section is second-order. 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.

5.2.1 *Preliminaries*

Recall that we will refer to the program that is to be completed as a *tree* (a shortcut for Abstract Syntax Trees). The reason we choose trees as a representation of the program is because trees provide a reasonable way to navigate over the program elements. 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, $s \in N$ is a start symbol, R is a finite set of production rules of the form $\alpha \to \beta_1 ... \beta_n$ with $\alpha \in N$ and $\beta_i \in N \cup \Sigma$ for $i \in [1..n]$.

In the whole exposition, we will assume that we are given a fixed CFG: $G = (N, \Sigma, s, R)$.

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 $\xi\colon X \to X^*$ 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.

Definition 5.3 (Partial parse tree)**.** A partial parse tree is a triple (T, G, σ) where $T = (X, x_0, \xi)$ is a tree, $G = (N, \Sigma, s, R)$ is a CFG, and $\sigma\colon X \to \Sigma \cup N$ attaches a terminal or non-terminal symbol to every node of the tree such that: if $\xi(x) = x_{a_1} ... x_{a_n}$ $(n > 1)$, then $\exists(\alpha \to \beta_1 ... \beta_n) \in R$ with $\sigma(x) = \alpha$ and $\forall i \in 1..n . \sigma(x_{a_i}) = \beta_i$.

Note that the condition for a partial parse tree requires that the tree follows the grammar production rules, but does not require all leaves to be terminal symbols (if σ attaches terminal symbols to all leaves, the tree will not be partial). Let the set of all partial parse trees be PT. Next, we define tree completion queries.

Definition 5.4 (Tree completion query)**.** A tree completion query is a triple $(ptree, x_{comp}, rules)$ where $ptree = (T, G, \sigma)$ is a partial parse tree with $T = (X, x_0, \xi)$, $x_{comp} \in X$ is a node labeled with a non-terminal symbol ($\sigma(x_{comp}) \in N$) where a completion will be performed, and $rules = \{\sigma(x_{comp}) \to \beta^i\}_{i=1}^n$ is the set of available rules that one can apply at the node x_{comp}.

Using the above definitions, we can now state the precise problem that is solved by this section.

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.

For the completions we consider, the right hand side β of each rule is a terminal symbol (e.g., a single API). In principle, one can make longer completions by iteratively chaining smaller ones. Our follow-up work defines a grammar called PHOG [20] using these rules.

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):

$$\begin{array}{rcl} GetProp & \to & Object\ \ Property \\ Object & \to & Var \mid GetProp \\ Var & \to & \texttt{console} \mid \texttt{document} \mid \text{... (other variables)} \\ Property & \to & \texttt{info} \mid \texttt{log} \mid \text{... (other properties incl. APIs)} \end{array}$$

The tree completion query for this example is illustrated in Fig. 5.3.

5.2.2 *Our Method: Second-order learning*

The key idea of our solution is to *synthesize a program which conditions the prediction*. That is, rather than statically hard-wiring the context

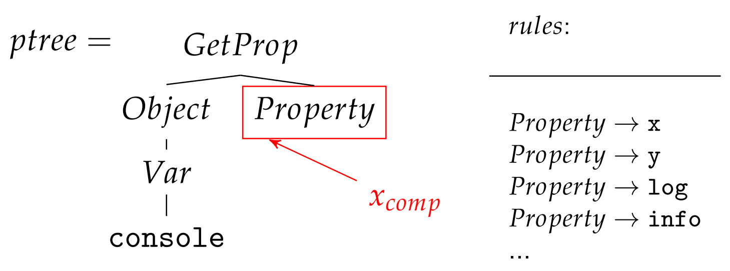


Figure 5.3: A tree completion query $(ptree, x_{comp}, rules)$ corresponding to completion for the code: "`console`".

on which the prediction depends on as in prior work, we use the program to *dynamically* determine the context for the particular query. For our example, given a partial parse tree *ptree* and a position x_{comp}, the program determines that the prediction of the API call should depend on the context `console`. Note that the node that holds the value `console` is not parent of the completion node and thus the completion does not only depend on the rules in the CFG.

In our setting, a context $c \in Context$ is a sequence ranging over terminal and non-terminal symbols seen in the tree, as well as integers. That is, $Context = (N \cup \Sigma \cup \mathbb{N})^*$. We next describe our method in a step-by-step manner following the learning architecture from Fig. 5.1 (b) and then elaborate on some of the steps in more detail.

STEP 1: REPRESENTATION SYNTHESIS The goal of the first step is to learn a conditioning program $p_{\approx best} \in \mathbb{P}$. In this step, we will apply the techniques for approximate empirical risk minimization discussed in Section 5.1.2.

Let $\mathcal{D} = \{X^i, Y^i\}_{i=1}^n$ be a training dataset of tree completions queries $X^i = (ptree^i, x^i_{comp}, rules)$ along with their corresponding completions $Y^i \in rules$. We assume that all examples in $\mathcal{D}$ solve the same task (e.g., API completion) and thus they share the CFG production *rules*. The goal of this step is to synthesize the (approximately) best conditioning program $p_{\approx best} \in PT \times X \to Context$ 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`$]$. In Section 5.2.3, we present a domain-specific language from which the conditioning program is drawn while in Section 5.2.4 we elaborate on this step in detail.

STEP 2: LEARN A PROBABILISTIC MODEL $p(rule \mid ctx)$ Once the conditioning program $p_{\approx best}$ is learned, we use that program to train a probabilistic model. Given our training dataset $\mathcal{D}$ as described above, we next apply $p_{\approx best}$ to every query in the training data, obtaining a new data set:

$$H(\mathcal{D}, p_{\approx best}) = \{(p_{\approx best}(Q_i), Y^i) \mid ((Q_i, rules), Y^i) \in \mathcal{D}\}$$

where $Q_i = (ptree^i, x^i_{comp})$. 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 $c_i \in Context$. Based on this derived set, we can now train a probabilistic model using MLE training (maximum likelihood estimation) which estimates the true probability $P(r \mid c)$. The MLE estimation is standard and is computed as follows:

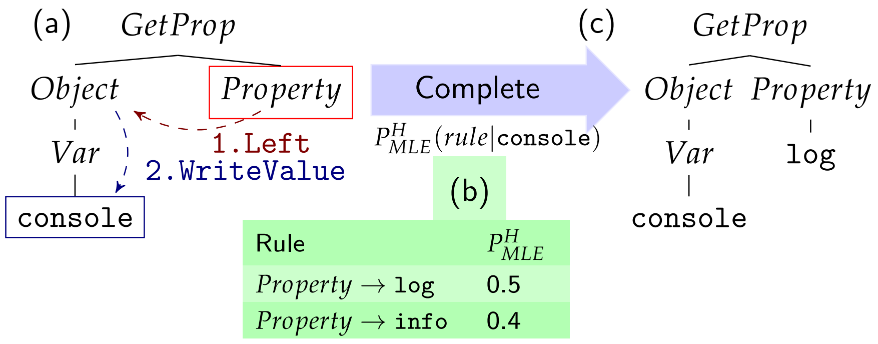
$$P^H_{MLE}(r \mid c) = \frac{|\{i \mid (c_i, r_i) \in H, c_i = c, r_i = r\}|}{|\{i \mid (c_i, r_i) \in H, c_i = c\}|}$$

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. As we will see later in Section 5.2.4, MLE learning as described above is also used in step 1.

STEP 3: PERFORM PREDICTIONS Once we have learned the conditioning program $p_{\approx best}$ and the probabilistic model $P(rule \mid ctx)$, 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:

$$rule = \arg\max_{r \in rules} P^H_{MLE}(r \mid ctx)$$

To illustrate the prediction on an example, consider the query shown in Fig. 5.4 (a) (this is the same query as in Fig. 5.3, repeated for convenience). In this example, the program $p_{\approx best}$ consists of two instructions: one moves left in the tree and the other one writes the element at the current position (we will see exact semantics of these instructions in Section 5.2.3). When applied to the given query, the program produces



Rule	P^H_{MLE}
Property → `log`	0.5
Property → `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.

$$\begin{aligned}
\texttt{Ops} &::= \epsilon \mid \texttt{Op Ops}\\
\texttt{Op} &::= \texttt{WriteOp} \mid \texttt{MoveOp}\\
\texttt{WriteOp} &::= \texttt{WriteValue} \mid \texttt{WritePos} \mid \texttt{WriteAction}\\
\texttt{MoveOp} &::= \texttt{Up} \mid \texttt{Left} \mid \texttt{DownFirst} \mid \texttt{DownLast} \mid \texttt{PrevDFS} \mid\\
&\quad\ \texttt{PrevLeaf} \mid \texttt{PrevNodeType} \mid \texttt{PrevActor}
\end{aligned}$$

Figure 5.5: The TCond language for extracting context from trees.

the context *ctx* consisting of the sole symbol `console`. Once the context is obtained, we can simply look up the probabilistic model P^H_{MLE} to find the most likely rule given the context (we list some of the rules and their probability in Fig. 5.4 (b)). Finally, we complete the query as shown in Fig. 5.4 (c).

5.2.3 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. We next provide an informal introduction to TCond. 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 partial tree is not modified during program execution but position and the context may be.

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.

Move instructions change the node in a state as follows:

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

Depending on the operation, $node'$ is set to either the node on the left of $node$ (for `Left`), to the parent of $node$ (for `Up`), to the first child of $node$ (for `DownFirst`), to the last child of $node$ (for `DownLast`), to the last leaf node in the tree on the left of $node$ (for `PrevLeaf`), to the previous node in depth-first search traversal order of the tree (for `PrevDFS`). When $node$ is a non-terminal symbol, the `PrevNodeType` instruction moves to the previous non-terminal symbol of the same type that is left of $node$.

Write instructions update the context of a state as follows:

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

Depending on the instruction, different value x is appended to the context. For the `WriteValue` instruction, the value of the terminal symbol below $node$ is written (if there is one, otherwise $x = -1$). For the `WritePos` instruction, if $parent$ is the parent node of $node$, then x is set to the index of $node$ in the list of the children of $parent$.

The `PrevActor` and `WriteAction` instructions use a simple lightweight 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. Our static analysis ignores loops, branches and function calls thus previous here refers to the occurrence of the memory location on the left of $node$ 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. An example of a program execution was already discussed for the example in shown in Fig. 5.4 (more examples can be seen in Fig. 5.8 (c), discussed later).

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

5.2.4 *Learning $p_{\approx best}$*

We next describe step 1 of our method in greater detail. The key objective of this step is to synthesize a conditioning program $p_{\approx best}$. Our DSL $\mathbb{P}$ is general enough such that several existing code completion systems [61, 110] can be seen as hard-wired programs $p \in \mathbb{P}$ for specific tasks. In order to pick a best program, we first define what it means for a synthesized program p to perform well (i.e. we define our cost function r). Then we describe the program generator and the representative dataset sampler ds^R that we use for Algorithm 2.

BUILDING A PROBABILISTIC MODEL As described earlier, we are given a dataset $\mathcal{D} = \{X^i, Y^i\}_{i=1}^n$ of queries $X^i = (ptree^i, x^i_{comp}, rules)$, along with their corresponding completions $Y^i \in rules$. For a program p and a dataset $\mathcal{D}$ we can then derive a new data set by applying the program to every query in $\mathcal{D}$, obtaining the resulting context, and storing that context and the given prediction together as a tuple, i.e., $Q_i = (ptree^i, x^i_{comp})$:

$$H(\mathcal{D}, p) = \{(p(Q_i), Y^i) \mid ((Q_i, rules), Y^i) \in \mathcal{D}\}$$

Let us partition the given dataset $\mathcal{D}$ into two non-overlapping parts – $\mathcal{D}_{train}$ and $\mathcal{D}_{eval}$. We then obtain the derived (training) set $H^t(\mathcal{D}, p) = H(\mathcal{D}_{train}, p)$ from which we build a probability distribution $P_{MLE}^{H^t(\mathcal{D},p)}$ as outlined earlier.

SCORING A PROBABILISTIC MODEL To evaluate a probabilistic model, we use an entropy measure of the derived (evaluation) set $H^e(\mathcal{D}, p) = H(\mathcal{D}_{eval}, p)$ on the learned model $P_{MLE}^{H^t(\mathcal{D},p)}$. The cross-entropy is a measure of how many bits we need to encode the evaluation data with the model build from the training data and provides insight not only on the error rate, but also how often a result is at a high rank and is produced with high confidence. In an empirical risk minimization setting, we use the cross-entropy to define the loss function on a single example $(ctx, rule)$:

$$\ell_{ent}(p, (ctx, rule)) = -\log_2 P_{MLE}^{H^t(\mathcal{D},p)}(rule \mid ctx)$$

Based on this loss function, we now define regularized empirical risk r_{regent} as in Section 5.1.1:

$$r_{regent}(\mathcal{D}, p) = \frac{1}{|H^e(\mathcal{D}, p)|} \sum_{x \in H^e(\mathcal{D},p)} \ell_{ent}(p, x) + \lambda \cdot \Omega(p),$$

where Ω is a regularizer function that returns the number of instructions in p. For all our experiments, we use $\lambda = 0.05$.

PROGRAM GENERATORS AND DATASET SAMPLERS Using the cost function $r_{regent}(d, p)$, we can now define the rest of the components and plug them into Algorithm 2. For our implementation, we use approximate versions of a program generator and a representative dataset sampler realized with random mutations and genetic programming.

An approximate *program generator* $gen_{\approx}$ takes a dataset d_i and an initial program p_{i-1}. Then, the program generator keeps a list L of candidate programs and iteratively updates the list in the following manner. First, $gen_{\approx}$ takes one candidate program, then performs random mutations on the instructions of the program, scores the modified program on the given dataset d_i according to the r_{regent} measure and then it adds it to the list L. Our mutations are chosen randomly among: (i) replacing one random instruction from the candidate program with a random instruction from the TCond language, (ii) removing a random instruction from the candidate program, and (iii) inserting an instruction from the TCond language at a random location of the candidate program. Using a genetic-programming like procedure, $gen_{\approx}$ randomly removes from the list candidate some of the programs that score worse than another candidate program. This is done to keep the list of candidate programs L small. After a fixed number of iterations, $gen_{\approx}$ returns the best scoring program from the list $p_i = \arg\min_{p \in L} r_{regent}(d_i, p) \approx \arg\min_{p \in \mathbb{P}} r_{regent}(d_i, p)$.

Our approximate *dataset sampler* $ds^R_{\approx}$ keeps tracks of the costs on the full dataset $\mathcal{D}$ for all programs $\{p_j\}_{j=1}^{i-1}$ generated by $gen_{\approx}$. Once a new program p_j is generated, $ds^R_{\approx}$ computes $r_{regent}(\mathcal{D}, p_j)$. Then, using a genetic-programming like procedure $ds^R_{\approx}$ keeps a list of candidate dataset samples and iteratively updates the list. At each iteration, $ds^R_{\approx}$ takes a dataset and randomly resamples its elements (such that the mutated dataset is $\subseteq \mathcal{D}$), scores the dataset and adds it to the list. Then $ds^R_{\approx}$ randomly removes from the list candidate datasets

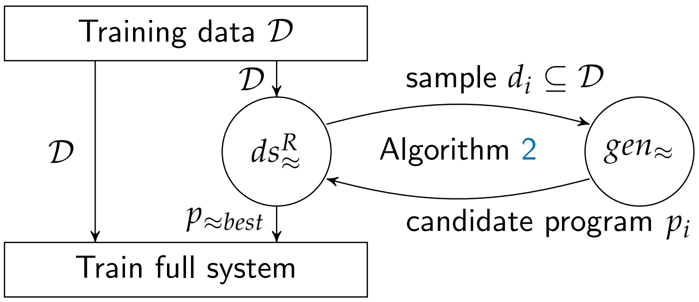


Figure 5.6: Overall diagram of learning a statistical code completion system using DeepSyn.

that are less representative (according to the *repr* measure defined in Definition 4.3) than another candidate dataset. After a fixed number of iterations $ds^R_\approx$ returns the dataset $d_i \subseteq \mathcal{D}$ which is approximately the most representative for $progs = \{p_j\}_{j=1}^{i-1}$ according to Definition 4.4.

TERMINATION CONDITION We have chosen a time-based termination condition. After some fixed time limit for training expires, we return the (approximately) best learned program $p_{\approx best}$ obtained up to that moment. This is, from the programs $p_1, ..., p_m$ produced up to the time limit by Algorithm 2, we return as $p_{\approx best}$ the one that has the best cost on the full dataset $\mathcal{D}$.

SMOOTHING An important detail for increasing the precision of the system is using smoothing when computing the maximum likelihood estimate at any stage of the system. In our implementation, we use Witten-Bell interpolation smoothing [137]. Smoothing ensures our system performs well even in cases when a context was not seen in the training data. Consider for example looking for the probability of $P_{MLE}(rule \mid c_1c_2c_3)$ which is a complex context of three observations in the tree. If such a context was not seen in the training data, to estimate reasonable probabilities we backoff to a simpler context with only c_1c_2 in it. Then, we expect to see more dense data in estimating $P_{MLE}(rule \mid c_1c_2)$. To enable such backoff in $P_{MLE}(rule \mid c_1c_2)$, at training time P_{MLE} counts the events conditioned on the full context and conditioned on all the prefixes of the full context.

5.2.5 *Summary of Approach*

Our learning procedure consists of two steps shown in Fig. 5.6. In the first step, we learn the conditioning program $p_{\approx best}$. That is, given a dataset $\mathcal{D}$, we first use Algorithm 2, $gen_{\approx}$ and $ds^{R}_{\approx}$ to find a program $p_{\approx best}$ for which the cost $r_{regent}(\mathcal{D}, p)$ is minimized. Then, in the second step, we a learn a probabilistic model $P_{MLE}^{H(\mathcal{D}, p_{\approx best})}$ over the full dataset $\mathcal{D}$ and the program $p_{\approx best}$ (discussed earlier).

This model, together with $p_{\approx best}$ can then be used to answer field/API completion queries from the programmer. We show in Section 5.3.1 that using Algorithm 2 equipped with a representative dataset sampler leads to high accuracy of the resulting statistical synthesizer.

DISCUSSION At its core, DEEPSYN includes a probabilistic model for predicting program elements that is parametrized by a program from a DSL. Each program from that DSL takes as an input a code snippet and traverses it in order to define this probabilistic model.

In our implementation of DEEPSYN, a program in the TCOND DSL takes its input represented as an AST (abstract syntax tree) and executes instructions that are easily described as tree movements. This representation, however, is not a conceptual limitation of our approach. For example, a future more advanced version of the DSL may include instructions that use other representations of the code such as program traces, control-flow graphs, value dependence graphs, call graphs and others [92]. Alternatively, the DSL may remain simple, but the program generator can be improved in order to synthesize more complex programs such that some lightweight program analysis is learned as part of the synthesized program [103]. It is important to note that as the synthesized program (or the DSL) complexity increases, so does the necessary work per data sample in the training dataset. This makes the program synthesis procedure directly on the entire training set even more prohibitive. In contrast, we propose to use small representative samples which enable scalable program synthesis.

Once the synthesis is complete, the final probabilistic model is trained on the entire dataset and thus, there is no loss of precision of the model due to the sampling. In this work, we illustrate such a model on all samples for one type of program elements (APIs/fields). However, a model can be learned for each program element and these models can then be combined into one [20].

5.3 EVALUATION OF DEEPSYN

Based on Section 5.2, we created a statistical code completion system for JavaScript programs. This is a particularly hard setting for code completion systems as unlike in other languages (e.g., Java) where type information is easily available (e.g., as in SLANG), in JavaScript, obtaining precise aliasing and type information is a difficult task (stressing the prediction capabilities of the system in the presence of noise, even further). The concrete task we consider is "dot" completion: given a receiver object, the completion should predict the field name or the API name that should be used on the object. All our experiments were performed on a 2.13 GHz Intel Xeon E7-4830 32-core machine with 256 GB of RAM, running 64-bit Ubuntu 14.04. To benefit from the amount of cores, we implemented the $gen_{\approx}$ and $ds^{R}_{\approx}$ procedures to evaluate multiple candidate programs and datasets in parallel.

To train and evaluate our system, we collected JavaScript programs from GitHub [46], removed duplicate files or project forks (copy of another existing repository) and kept only programs that parse and are not obfuscated. As a result, we obtained $150,000$ JavaScript files (trees). We used two thirds of the data for learning, and the last one third only for evaluation. Our dataset, split into training and evaluation portion, is available at `http://www.srl.inf.ethz.ch/js150.php`.

5.3.1 *Learning $p_{\approx best}$*

We now discuss our training procedure. The first question we may ask is if using Algorithm 2 with a representative dataset sampler ($ds^{R}_{\approx}$) is of any benefit to the speed or precision of the system. To answer this question, we designed a number of system variants:

- DEEPSYN ($gen_{\approx}$ with $ds^{R}_{\approx}$) is our system as described Fig. 5.6. For this system, we start with a random sample of 100 trees from the training dataset and then through the loop of Algorithm 2, we modify the sample to be more representative.

- $gen_{\approx}$ on full dataset is a system that directly optimizes the program on the full training dataset $\mathcal{D}$. Because evaluating each candidate programs takes a long time, this system can only try a smaller number of candidate programs and results in an imprecise probabilistic model.

- $gen_{\approx}$ on fixed sample is a system that starts with a random sample of 100 trees from the training dataset and optimizes the program only on this small sample. This policy has the time budget to explore a large number of candidate programs, but is unable to provide reasonable score estimates for them. It quickly reaches a cap on what can be learned on the small dataset.

- $gen_{\approx}$ with randomly increasing sample is a system that starts with a small random sample of trees from the training dataset and iteratively increases the size of the sample. It performs no optimization of the sample for representativeness, just adds more elements to it.

The results summarized in Fig. 5.7 illustrate the effectiveness of each of the four approaches. Each plot gives the r_{regent} of the best candidate program $p_{\approx best}$ found by a system at a given time. Note that the lower values of cross-entropy, the better. In fact a small decrease in cross-entropy typically leads to a significantly better model, because entropy is measured in bits (with equally likely choices, reduction of one bit means there are 2 times less possible choices to make). The graph shows that our full DeepSyn system initially spends time to find the best sample, but then reaches the best program of all systems. If provided with infinite time, $gen_{\approx}$ on full data will approach the optimal program, but in practice it is prohibitively slow and has far worse performance.

We note that the synthesizer with randomly increasing sample appears to find a reasonable program faster than when using $ds_{\approx}^{R}$. The reason for this is that this procedure did not evaluate its result on the full dataset *dataset* (we evaluated the programs after the procedure completed). If we include the time to evaluate the candidate programs, this setting would not be as fast as it appears.

Next, we take the best program obtained after one hour of computation by DeepSyn and analyze it in detail.

5.3.2 *Precision of* DeepSyn

Once we obtain the best TCond program, we create a completion system based on it, train it on our learning dataset and evaluate it on the evaluation dataset. For each file in the evaluation dataset we

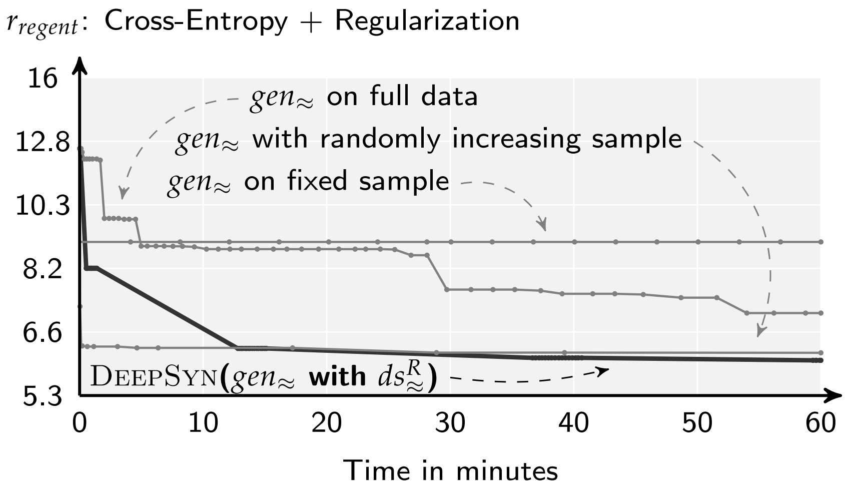


Figure 5.7: Effect of various data sampling policies used to find TCond programs.

randomly selected 100 method calls and queried our system to predict the correct completion one at a time. Given the evaluation dataset of 50, 000 programs, this resulted in invoking the prediction of 2, 537, 415 methods for any API and 48, 390 methods when predicting method calls on DOM document object.

The accuracy results are summarized in Table 5.1. The columns of the table represent systems trained on different amounts of training data. The right-most column trains on all 100, 000 JavaScript programs in the training set and the columns on the left use a subset of this data. Different rows on Table 5.1 include information for different tasks. On the task of predicting DOM APIs on the document object, the APIs are shared across all projects and the accuracy is higher – the correct completion is the first suggestion in 77% of the cases. When we extend the completion to any APIs, including APIs local to each project that the model may not know about, the accuracy drops to 50.4%. When used on non-API property completions, our completion system predicts the correct field name as a first suggestion in 38.9% of the cases.

	Size of training data (files)		
Task	1K	10K	100K
DOM APIs on document object			
correct completion at position 1	63.2%	69.2%	**77.0%**
correct completion in top 3	90.1%	84.6%	89.9%
correct completion in top 8	83.5%	88.6%	92.9%
Unrestricted API completion			
correct completion at position 1	22.6%	34.2%	**50.4%**
correct completion in top 3	30.8%	44.5%	61.9%
correct completion in top 8	33.6%	47.7%	64.9%
Field (non-API) completion			
correct completion at position 1	21.0%	29.7%	**38.9%**
correct completion in top 3	26.3%	37.0%	48.9%
correct completion in top 8	28.0%	38.8%	51.4%

Table 5.1: Accuracy of API method and object field completion depending on the task and the amount of training data.

5.3.3 *Interpreting $p_{\approx best}$*

A useful aspect of TCond programs is that they are easily readable by a human. The best program is listed with its instructions in Fig. 5.8 (a). Let us illustrate what this program does on a JavaScript code completion query from Fig. 5.8 (b). Going instruction by instruction, Fig. 5.8 (c) shows the execution. First, the program moves to the left of the completion position in the tree (i.e., to the receiver object of the completion). Then, it moves to the previous usage of the same object (`PrevActor`), writes the action being done on the project (i.e., the name of the field of API invoked). Next, it writes the name of the variable (if any) and moves to the previous usage of the same object, etc. Finally, at instruction 9, it cannot move anymore and the program stops and returns the accumulated sequence so far. For our example, the program accumulates the following context:

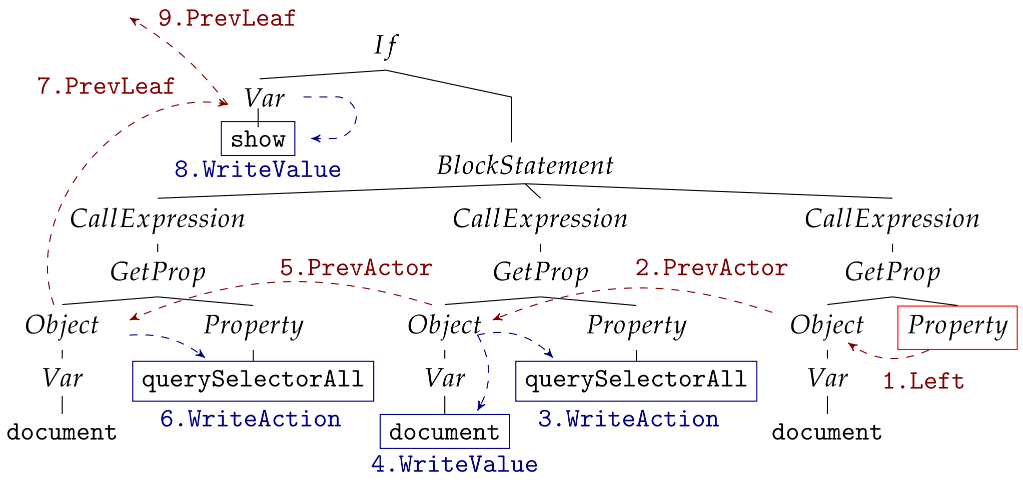
```
querySelectorAll document querySelectorAll show
```

$p_{\approx best} =$ `Left PrevActor WriteAction WriteValue`
`PrevActor WriteAction PrevLeaf`
`WriteValue PrevLeaf WriteValue`

(a) TCond program

```
if (show) {
 var cws = document.querySelectorAll(...);
 for (var i = 0, slide; slide = cws[i]; i++) {
   slide.classList.add("hidden");
 }
 var iap = document.querySelectorAll(...);
 for (var i = 0, slide; slide = iap[i]; i++) {
   slide.classList.add("hidden");
 }
 var dart = document.
 ...                        └──── Completion position
}
```

(b) JavaScript code snippet



(c) Execution of $p_{\approx best}$ on the AST representation of the code snippet from (b)

Figure 5.8: The (a) $p_{\approx best}$ TCond program describing which parts of the code to condition on for an API method completion task, (b) a JavaScript code snippet on which we want to perform API completion, and (c) a relevant part of the AST representation of (b) where the execution of the TCond program from (a) is illustrated.

5.3.4 *Comparison to Existing Systems*

We note that previous works essentially use a hard-coded tree conditioning program. In some of the works immediate context is used – the non-terminals directly preceding the completion location [3, 61]. The conditioning of the SLANG system described in Chapter 3 can also be described by a TCOND program.

Note that for SLANG, our statistical code completion systems targeted Java which makes it easier to perform static program analysis (e.g., type analysis, getting fully qualified names, alias analysis) which explains why the SLANG program performs far worse in the case of JavaScript API completions. Next, we review these programs and compare their accuracy to DEEPSYN using the best program $p_{\approx best}$ in Fig. 5.8 (a). As previously shown in Table 5.1, the accuracy on the task of predicting JavaScript APIs for this program is 50.4%.

- The work of Hindle at al. [61] uses an 3-gram language model on the tokenized program without performing any semantic analysis. This results in a predictor for JavaScript APIs with accuracy of 22.2%. The program that corresponds to this model is:

  ```
  PrevDFS WriteValue PrevDFS WriteValue
  ```

- The language model of SLANG performs a conditioning that depends on the previous APIs of the same object as well as whether the object was used as a receiver or as a method parameter. This conditioning is captured in the following program:

  ```
  Left  PrevActor WriteAction WritePos
        PrevActor WriteAction WritePos
  ```

  The accuracy of this program for JavaScript APIs is 30.4%.

In addition to the best possible program $p_{\approx best}$, we include the programs generated by our tool with different data sampling procedures.

- $\underline{gen_{\approx} \text{ on full data}}$ operates on the full data and within one hour it could not learn to condition on previous APIs. The program learned by this synthesis procedure is

  ```
  Left DownLast WriteValue PrevNodeType DownLast
  PrevDFS PrevLeaf WriteValue DownLast PrevLeaf
  Left WriteAction PrevNodeType WriteValue
  ```

  and results in 46.3% accuracy.

- $\underline{gen_{\approx}\text{ on fixed sample}}$ only learns a program from a small sample of 100 programs and results in a simple conditioning that only depends on a single previous API. The resulting accuracy for JavaScript API completion is 18.8% with the program

  ```
  PrevDFS PrevActor WriteAction
  ```

- $\underline{gen_{\approx}\text{ with randomly increasing sample}}$ iteratively increases the sample and results in a program that has accuracy of 47.5% which is lower than our best accuracy of 50.4%. The program generated by randomly increasing the sample size conditions on one previous API, but the rest of the program is not optimal

  ```
  Left PrevActor WriteAction
  DownLast PrevActor PrevNodeType WriteAction
  WriteValue Left PrevLeaf WriteValue
  ```

SUMMARY Overall, we have shown that our approach of learning a tree conditioning program in a domain specific language (TCond) exceeds the accuracy of systems where an expert hard-wired the prediction, as used in previous systems. At the same time, our approach includes a domain specific language that is flexible and extensible to further increase the accuracy and the capabilities of the system. The high accuracy of this work is achieved thanks to our general approach – a program generator that predicts a program from a small dataset ($gen_{\approx}$) and a representative dataset sampler that makes the small set behave similarly to the large training data ($ds^{R}_{\approx}$).

5.4 RELATED WORK

As our work touches on several areas, below we survey some of the existing results that are most closely related to ours.

CORE SETS A core set is a concept in machine learning used to summarize a large data set into a smaller data set that preserves its properties. Core sets were successfully applied in the context of clustering such as k-means [58]. Obtaining core sets from a data set is a procedure that is manually tailored to the particular problem at hand (e.g., k-means), that is, there are currently no universal techniques for constructing core sets for arbitrary programs. In contrast, our work is not based on a specific algorithm or property such as k-means. In fact, an ideal outcome of our sampling step is to compute or approximate a core set of the training data. An interesting question for future work is to explore the connection between core sets and our iterative sampling algorithm.

DATASET CLEANING A different method for dealing with large noisy data is to clean up the data beforehand either with a statistical model [29], or with an already given program [10]. These approaches, however, need additional statistical assumptions about the data or specification of another program. In contrast, in this chapter, we simultaneously build the cleaned dataset and the program.

Another promising direction has been proposed in the context of debugging probabilistic models [36]. The idea is to detect outlier examples by observing if flipping a label in the training data leads to higher accuracy of the model. Similar to our work, their approach may discover a sample that is representative for the learned task. However, they do not attempt to reduce the size of the sample, to speed-up learning or to use this sample to discover deeper models or features.

GENETIC ALGORITHMS Genetic programming has been proposed as a general approach to explore a large set of candidates in order to discover a solution that maximizes an objective function [9, 119]. The work or Cramer [32] discusses the language design decisions to encode a program synthesis problem from input/output examples into genetic programming. Some of the problems studied here in terms of selecting subset of the evaluation data to score instances were considered in the context of genetic programming. A technique known as stochastic sampling [8] reduces the number of evaluations by only considering random subsets of the training data. In our experiments, however, we show that using our strategy of representative sampling is superior than using random sampling.

PROBABILISTIC PROGRAMS A recent synthesizer PSketch [96] performs synthesis of probabilistic programs by approximating the program via a mixture of Gaussians. Fundamentally, probabilistic programs interpret program executions as distributions and the synthesis task fits parameters to these distributions. Instead, with our approach we learn deterministic programs that approximate a dataset well. In general, our idea of a dataset sampler should be applicable also to learning probabilistic programs from data, but we leave this as a future work item.

automatic configuration of algorithms ParamILS [65] picks a configuration of an algorithm based on its performance on a dataset. Similar to our approach, ParamILS attempts to speed-up the evaluation of a configuration by running on a small subset of the full dataset, but only does so by selecting a random sample.

summary In this chapter, we presented a new procedure for learning synthesizers with "Big Code". An interesting future direction is to explore other domain-specific languages with richer instruction sets (e.g. containing branches and/or loops). Such languages could enable even higher expressivity and better precision.

6 CONCLUSION AND FUTURE WORK

In this dissertation, we demonstrated a new approach for building probabilistic systems aimed at solving programming tasks. The core idea is to use program analysis to transform the problem into a suitable intermediate representation and to then learn a probabilistic model over this intermediate representation in order to make predictions. We discussed two kinds of probabilistic models – a *discriminative* conditional random field model with queries represented as graphs and a *generative* model with queries represented as sequences. Both of these models were shown to be scalable to train and efficient to query. We discussed several manually designed program analyses to transform a problem into one of these intermediate representations. We also presented a general approach that uses a domain-specific language to automatically synthesize code completion systems. Using this approach, we built a state-of-the-art code completion system for JavaScript.

PROBABILISTIC SYSTEMS We described three scalable systems trained on a large corpus of code. The JSNICE system predicts names and type annotations for JavaScript. JSNICE is practically useful: it has become a popular tool in the JavaScript developer community and it has been used by more than 100′000 developers. The SLANG system provides state-of-the-art accuracy for Java Android API completion and an Eclipse plug-in that displays SLANG predictions is in development. The DEEPSYN system provides JavaScript code completion with far higher accuracy than that of previous works.

6.1 FUTURE WORK

We foresee a number of directions for extending these tools and for applying the techniques in this dissertation to new applications. Next, we discuss some possible extensions.

6.1.1 *Creating Probabilistic Tools on Top of Our Models*

The success of the JSNice tool in the JavaScript developer community led us to separate its probabilistic model and open source it as a separate project called Nice2Predict available at `http://nice2predict.org/`. Nice2Predict should be applicable to a range of applications for which there is a need to make joint predictions of multiple program properties that satisfy a given set of constraints.

REVERSE ENGINEERING TOOLS Code transformation tools such as compilers take a human-readable representation of a program and convert it into a representation that is less human readable, but often more efficient for machine processing. In the process, these tools may remove information from the original program description such as names, type annotations, coding style, comments and others. We foresee a range of tools that reverse such transformations by using "Big Code" to reconstruct the removed information. These tools could benefit from the models developed by JSNice and many of them can be directly encoded in our Nice2Predict framework. Next, we give two examples of such tools.

Similar to JavaScript code, it is common for Android developers to reduce the size of their applications by performing minification. The most common tool for Android minification is called ProGuard [118] and ProGuard renames application class, method and field names to meaningless, but shorter names. To reverse the obfuscation, a "Big Code" tool may build a probabilistic model akin to the model of JSNice such that the shortened names in an Android application are predicted based on other non-shortened names such as Android APIs. We have recently started investigating this direction [17].

The work of Katz et al. [71] is another instance of this direction and focuses on recovering types in compiled binaries. Their solution is also similar to the one we propose in Chapter 3, but with differences in the selected program analysis component that computes the features. Unfortunately, their approach predicts each type in isolation and as a result the predictions cannot be guaranteed to satisfy type-checking rules. A possible way to address this limitation is by framing the problem as a MAP inference query as in our work and using our learning procedure from Section 2.5.

IMPROVING THE ACCURACY OF PREDICTIONS Our current solution for name prediction in JSNice only predicts identifier names that were seen in the training data. However, many identifier names are not shared across projects. Yet, some names are a concatenation of several common English words. Using a model that performs prediction of each word in a name separately will enable a name predictor that can suggest names not seen in the training data (a promising work in this direction is [2]). Similar arguments may be made about predictions of other program properties such as template types or invariants – while the entire type may be project-specific, it can be split into components that range from a set of types shared among projects in the training data.

6.1.2 *Universal Model for Code*

In natural language processing tasks, modeling probabilities of (arbitrary) text enables a range of applications including statistical machine translation, speech recognition and others [114]. Similarly, a universal probabilistic model that assigns probabilities to all program elements enables new statistical programming tools. So far, these statistical programming tools model the probability of code using a probabilistic context-free grammar [55] or a language model on code tokens [4]. While these models are universal, they are a poor fit for code semantics [61, 95]. With the techniques from Chapter 5, we have shown how to accurately predict code and how to assign probabilities to our predictions. Based on this idea, we can build a new probabilistic model for code that assigns high probabilities to correct programs and low probabilities to incorrect programs, i.e. a *good* probabilistic model for code. We have recently started exploring this direction in [20].

A universal probabilistic model for code would not only improve on existing applications, but may enable new applications. Automatic generation of refactorings has been an active research area in recent years [106, 124]. For example, the ReSynth tool [106] asks the user to demonstrate a code modification on part of a program and then the tool performs refactorings on the entire program that include the desired modifications. To handle the case in which the user demonstration is ambiguous, we suggest to return the solution that is scored highest by the probabilistic model. In an interesting use case of such a tool, the user is not even required to provide a demonstration, yet the tool can automatically suggest a refactoring that improves the overall likelihood of a program according to a probabilistic model.

Another area of interest is automatic bug location and bug fixing. Similar to the refactoring setting [106], a tool may search through a space of programs by performing modifications on a given input program. In contrast to the refactoring setting, however, these modifications are not semantic-preserving and an attempt to fix a defect in the input program. Using a probabilistic model, we foresee a tool that can localize and fix defects in programs. This enables improvements on top of current techniques [83] which only find a fix once the defective line has already been localized. More generally, probabilistic models for code can be used in a number of other applications such as fuzzing, program synthesis and others.

6.1.3 *Statistical Program Synthesis*

In this thesis, we investigated the problem of program synthesis from two perspectives. First, we introduced statistical synthesizers that complete partial programs based on probabilistic models and subject to semantic constraints. Second, we formulated the problem of program synthesis with noise such that incorrect input/output examples are handled by the synthesizer. Future work items in this area include combining these ideas into one common synthesizer that takes into account both: (possibly noisy) input/output examples and likelihood measures from a probabilistic model.

COMBINING PROBABILISTIC SEARCH WITH REASONING Another interesting future research direction is to combine probabilistic synthesis with other synthesis techniques such as deductive synthesis, version spaces and others [52]. One possible approach here is that part of the program is synthesized with probabilistic models and the rest is completed with other techniques.

BIBLIOGRAPHY

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. „Learning Natural Coding Conventions.“ In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 281–293. URL: http://doi.acm.org/10.1145/2635868.2635883 (cit. on p. 4).

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. „Suggesting Accurate Method and Class Names.“ In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 38–49. URL: http://doi.acm.org/10.1145/2786805.2786849 (cit. on p. 151).

[3] Miltiadis Allamanis and Charles Sutton. „Mining Source Code Repositories at Massive Scale Using Language Modeling.“ In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 207–216. URL: http://dl.acm.org/citation.cfm?id=2487085.2487127 (cit. on p. 145).

[4] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. „Bimodal Modelling of Source Code and Natural Language.“ In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, pp. 2123–2132. URL: http://jmlr.org/proceedings/papers/v37/allamanis15.html (cit. on p. 151).

[5] A. Alnusair, Tian Zhao, and E. Bodden. „Effective API navigation and reuse.“ In: *2010 IEEE International Conference on Information Reuse and Integration (IRI)*. Aug. 2010, pp. 7–12. URL: http://dx.doi.org/10.1109/IRI.2010.5558972 (cit. on p. 69).

[6] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. „Syntax-Guided Synthesis.“ In: *Proceedings of the IEEE Interna-*

tional Conference on Formal Methods in Computer-Aided Design (FMCAD). Oct. 2013, pp. 1–17 (cit. on pp. 1, 101).

[7] Glenn Ammons, Rastislav Bodík, and James R. Larus. „Mining Specifications.“ In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 4–16. URL: `http://doi.acm.org/10.1145/503272.503275` (cit. on p. 99).

[8] James E. Baker. „Reducing Bias and Inefficiency in the Selection Algorithm.“ In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Cambridge, Massachusetts, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21. URL: `http://dl.acm.org/citation.cfm?id=42512.42515` (cit. on p. 147).

[9] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998 (cit. on p. 147).

[10] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. „CheckCell: Data Debugging for Spreadsheets.“ In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &#38; Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 507–523. URL: `http://doi.acm.org/10.1145/2660193.2660207` (cit. on pp. 1, 104, 120, 147).

[11] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. „FlashRelate: extracting relational data from semi-structured spreadsheets using examples.“ In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 2015, pp. 218–228. URL: `http://doi.acm.org/10.1145/2737924.2737952` (cit. on p. 101).

[12] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. „An Empirical Study of Object Protocols in the Wild.“ In: *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. Springer-Verlag, 2011, pp. 2–26. URL: `http://dl.acm.org/citation.cfm?id=2032497.2032501` (cit. on p. 68).

[13] Nels E. Beckman and Aditya V. Nori. „Probabilistic, Modular and Scalable Inference of Typestate Specifications.“ In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 211–221. URL: http://doi.acm.org/10.1145/1993498.1993524 (cit. on pp. 62, 63).

[14] Y. Bengio, P. Simard, and P. Frasconi. „Learning Long-term Dependencies with Gradient Descent is Difficult.“ In: *Trans. Neur. Netw.* 5.2 (Mar. 1994), pp. 157–166. URL: http://dx.doi.org/10.1109/72.279181 (cit. on p. 80).

[15] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. „A Neural Probabilistic Language Model.“ In: *Journal of Machine Learning Research* 3 (Mar. 2003), pp. 1137–1155. URL: http://dl.acm.org/citation.cfm?id=944919.944966 (cit. on p. 89).

[16] Julian Besag. „On the Statistical Analysis of Dirty Pictures.“ In: *Journal of the Royal Statistical Society. Series B (Methodol.)* 48.3 (1986), pp. 259–302. URL: http://dx.doi.org/10.2307/2345426 (cit. on pp. 15, 39, 58).

[17] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. „Statistical Deobfuscation of Android Applications.“ In: CCS 2016. to appear, 2016 (cit. on pp. vii, 17, 150).

[18] Pavol Bielik, Veselin Raychev, and Martin Vechev. „Programming with "Big Code": Lessons, Techniques and Applications.“ In: *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 2015, pp. 41–50. URL: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.41 (cit. on p. vii).

[19] Pavol Bielik, Veselin Raychev, and Martin Vechev. „Scalable Race Detection for Android Applications.“ In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. ACM, 2015, pp. 332–348. URL: http://doi.acm.org/10.1145/2814270.2814303 (cit. on p. viii).

[20] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. „PHOG: Probabilistic Model for Code.“ In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City,*

NY, USA, June 19-24, 2016. 2016, pp. 2933–2942 (cit. on pp. vii, 66, 131, 139, 151).

[21] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006 (cit. on p. 49).

[22] *BitBucket*. https://bitbucket.org/ (cit. on pp. 1, 52).

[23] James Bornholt and Emina Torlak. „Scaling program synthesis by exploiting existing code.“ In: *Machine Learning for Programming Languages (ML4PL)*. 2015 (cit. on p. 12).

[24] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007 (cit. on p. 64).

[25] Thorsten Brants and Alex Franz. „Web 1T 5-gram, Version 1.“ In: *Linguistic Data Consortium* (2006). URL: http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13 (cit. on p. 1).

[26] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. „Exploring the Influence of Identifier Names on Code Quality: An Empirical Study.“ In: *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 156–165. URL: http://dx.doi.org/10.1109/CSMR.2010.27 (cit. on p. 4).

[27] Bruno Caprile and Paolo Tonella. „Restructuring Program Identifier Names.“ In: *Proceedings of the International Conference on Software Maintenance (ICSM'00)*. ICSM '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 97–. URL: http://dl.acm.org/citation.cfm?id=850948.853439 (cit. on p. 4).

[28] Pavol Černý and Thomas A. Henzinger. „From Boolean to Quantitative Synthesis.“ In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT '11. Taipei, Taiwan: ACM, 2011, pp. 149–154. URL: http://doi.acm.org/10.1145/2038642.2038666 (cit. on p. 121).

[29] Varun Chandola, Arindam Banerjee, and Vipin Kumar. „Anomaly Detection: A Survey.“ In: *ACM Comput. Surv.* 41.3 (July 2009), 15:1–15:58. URL: http://doi.acm.org/10.1145/1541880.1541882 (cit. on pp. 104, 120, 147).

[30] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. „Bridging Boolean and Quantitative Synthesis Using Smoothed Proof Search.“ In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 207–220. URL: `http://doi.acm.org/10.1145/2535838.2535859` (cit. on p. 121).

[31] Jonathan E. Cook and Alexander L. Wolf. „Discovering Models of Software Processes from Event-based Data.“ In: *ACM Transactions on Software Engineering and Methodology* 7.3 (July 1998), pp. 215–249. URL: `http://doi.acm.org/10.1145/287000.287001` (cit. on p. 99).

[32] Nichael Lynn Cramer. „A Representation for the Adaptive Generation of Simple Sequential Programs.“ In: *Proceedings of the 1st International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1985, pp. 183–187. URL: `http://dl.acm.org/citation.cfm?id=645511.657085` (cit. on p. 147).

[33] Barthélémy Dagenais and Laurie Hendren. „Enabling Static Analysis for Partial Java Programs.“ In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 313–328. URL: `http://doi.acm.org/10.1145/1449764.1449790` (cit. on pp. 71, 90).

[34] DARPA. *Mining and Understanding Software Enclaves (MUSE)*. `http://www.darpa.mil/news-events/2014-03-06a`. 2014 (cit. on p. 1).

[35] Leonardo De Moura and Nikolaj Bjørner. „Z3: An Efficient SMT Solver.“ In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. URL: `http://dl.acm.org/citation.cfm?id=1792734.1792766` (cit. on p. 118).

[36] „Debugging machine learning models.“ In: *ICML Workshop on Reliable Machine Learning in the Wild*. 2016 (cit. on p. 147).

[37] Dimitar Dimitrov, Veselin Raychev, Martin Vechev, and Eric Koskinen. „Commutativity Race Detection.“ In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. ACM, 2014, pp. 305–315. URL: http://doi.acm.org/10.1145/2594291.2594322 (cit. on p. viii).

[38] Jeffrey L. Elman. „Finding structure in time.“ In: *Cognitive Science* 14.2 (1990), pp. 179–211. URL: http://groups.lis.illinois.edu/amag/langev/paper/elman90findingStructure.html (cit. on pp. 67, 68, 79).

[39] *Facebook*. http://facebook.com/ (cit. on p. 1).

[40] *Factorie*. https://github.com/factorie/factorie (cit. on p. 65).

[41] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. „Effective Typestate Verification in the Presence of Aliasing.“ In: *ACM Trans. Softw. Eng. Methodol.* 17.2 (May 2008), 9:1–9:34. URL: http://doi.acm.org/10.1145/1348250.1348255 (cit. on p. 6).

[42] Thomas Finley and Thorsten Joachims. „Training Structural SVMs when Exact Inference is Intractable.“ In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: ACM, 2008, pp. 304–311. URL: http://doi.acm.org/10.1145/1390156.1390195 (cit. on p. 39).

[43] Cormac Flanagan and Patrice Godefroid. „Dynamic Partial-order Reduction for Model Checking Software.“ In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 110–121. URL: http://doi.acm.org/10.1145/1040305.1040315 (cit. on p. 1).

[44] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. „Learning Invariants Using Decision Trees and Implication Counterexamples.“ In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 499–512. URL: http://doi.acm.org/10.1145/2837614.2837664 (cit. on pp. 1, 12).

[45] Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. „Learning to Forget: Continual Prediction with LSTM.“ In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. URL: http://dx.doi.org/10.1162/089976600300015015 (cit. on p. 80).

[46] *GitHub*. http://github.com/ (cit. on pp. 1, 16, 52, 69, 140).

[47] *Google Closure Compiler*. https://developers.google.com/closure/compiler/ (cit. on pp. 5, 17, 51).

[48] *Google Translate*. http://translate.google.com/ (cit. on p. 1).

[49] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. „Probabilistic Programming.“ In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: ACM, 2014, pp. 167–181. URL: http://doi.acm.org/10.1145/2593882.2593900 (cit. on p. 65).

[50] Radu Grigore and Hongseok Yang. „Abstraction Refinement Guided by a Learnt Probabilistic Model.“ In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 485–498. URL: http://doi.acm.org/10.1145/2837614.2837663 (cit. on p. 12).

[51] Sumit Gulwani. „Automating String Processing in Spreadsheets Using Input-output Examples.“ In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 317–330. URL: http://doi.acm.org/10.1145/1926385.1926423 (cit. on pp. 1, 101, 102).

[52] Sumit Gulwani. „Dimensions in Program Synthesis.“ In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP '10. Hagenberg, Austria: ACM, 2010, pp. 13–24. URL: http://doi.acm.org/10.1145/1836089.1836091 (cit. on pp. 1, 98, 121, 152).

[53] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. „Synthesis of Loop-free Programs.“ In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 62–73. URL: http://doi.acm.org/10.1145/1993498.1993506 (cit. on p. 119).

[54] Sumit Gulwani and Nebojsa Jojic. „Program Verification As Probabilistic Inference.“ In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: ACM, 2007, pp. 277–289. URL: http://doi.acm.org/10.1145/1190216.1190258 (cit. on p. 62).

[55] Tihomir Gvero and Viktor Kuncak. „Synthesizing Java Expressions from Free-form Queries.“ In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 416–432. URL: http://doi.acm.org/10.1145/2814270.2814295 (cit. on pp. 123, 151).

[56] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. „Complete Completion Using Types and Weights.“ In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. ACM, 2013, pp. 27–38. URL: http://doi.acm.org/10.1145/2491956.2462192 (cit. on pp. 69, 99).

[57] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. „Interactive Synthesis of Code Snippets.“ In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 418–423. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_33 (cit. on p. 99).

[58] Sariel Har-Peled and Soham Mazumdar. „On Coresets for K-means and K-median Clustering.“ In: *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: ACM, 2004, pp. 291–300. URL: http://doi.acm.org/10.1145/1007352.1007400 (cit. on p. 146).

[59] F. Maxwell Harper and Joseph A. Konstan. „The MovieLens Datasets: History and Context.“ In: *ACM Trans. Interact. Intell. Syst.* 5.4 (Dec. 2015), 19:1–19:19. URL: http://doi.acm.org/10.1145/2827872 (cit. on p. 1).

[60] Xuming He, Richard S. Zemel, and Miguel Á. Carreira-Perpiñán. „Multiscale Conditional Random Fields for Image Labeling.“ In: CVPR '04. Washington, D.C., USA. URL: http://dl.acm.org/citation.cfm?id=1896300.1896400 (cit. on pp. 15, 66).

[61] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. „On the naturalness of software." In: *ICSE 2012*. 2012. URL: http://dl.acm.org/citation.cfm?id=2337223.2337322 (cit. on pp. 2, 13, 68, 98, 103, 123, 129, 136, 145, 151).

[62] Reid Holmes and Gail C. Murphy. „Using Structural Context to Recommend Source Code Examples." In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: ACM, 2005, pp. 117–125. URL: http://doi.acm.org/10.1145/1062455.1062491 (cit. on p. 99).

[63] Reid Holmes, Robert J. Walker, and Gail C. Murphy. „Strathcona Example Recommendation Tool." In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. ACM, 2005, pp. 237–240. URL: http://doi.acm.org/10.1145/1081706.1081744 (cit. on p. 69).

[64] Einar W. Høst and Bjarte M. Østvold. „Debugging Method Names." In: *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Genoa. Italy: Springer-Verlag, 2009, pp. 294–317. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_14 (cit. on p. 4).

[65] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. „ParamILS: An Automatic Algorithm Configuration Framework." In: *J. Artif. Int. Res.* 36.1 (Sept. 2009), pp. 267–306. URL: http://dl.acm.org/citation.cfm?id=1734953.1734959 (cit. on p. 148).

[66] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. „Stateless Model Checking of Event-driven Applications." In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. ACM, 2015, pp. 57–73. URL: http://doi.acm.org/10.1145/2814270.2814282 (cit. on pp. viii, 1).

[67] Simon Holm Jensen, Anders Møller, and Peter Thiemann. „Type Analysis for JavaScript." In: *Proceedings of the 16th International Symposium on Static Analysis*. SAS '09. Los Angeles, CA: Springer-

Verlag, 2009, pp. 238–255. URL: http://dx.doi.org/10.1007/978-3-642-03237-0_17 (cit. on pp. 1, 17, 32).

[68] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. „Oracle-guided Component-based Program Synthesis.“ In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 215–224. URL: http://doi.acm.org/10.1145/1806799.1806833 (cit. on pp. 1, 101, 104, 108, 115, 116, 118, 119, 121).

[69] Jorg H. Kappes, Bjoern Andres, Fred A. Hamprecht, Christoph Schnorr, Sebastian Nowozin, Dhruv Batra, Sungwoong Kim, Bernhard X. Kausler, Jan Lellmann, Nikos Komodakis, and Carsten Rother. „A Comparative Study of Modern Inference Techniques for Discrete Energy Minimization Problems.“ In: *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1328–1335. URL: http://dx.doi.org/10.1109/CVPR.2013.175 (cit. on p. 39).

[70] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. „Phrase-Based Statistical Translation of Programming Languages.“ In: Onward! '14. ACM, 2014. URL: http://doi.acm.org/10.1145/2661136.2661148 (cit. on p. viii).

[71] Omer Katz, Ran El-Yaniv, and Eran Yahav. „Estimating Types in Binaries Using Predictive Modeling.“ In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 313–326. URL: http://doi.acm.org/10.1145/2837614.2837674 (cit. on pp. 12, 150).

[72] Slava M. Katz. „Estimation of probabilities from sparse data for the language model component of a speech recognizer.“ In: *IEEE Transactions on Acoustics, Speech and Singal processing*. Vol. ASSP-35. 3. Mar. 1987 (cit. on p. 79).

[73] Reinhard Kneser and Hermann Ney. „Improved backing-off for m-gram language modeling.“ In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. Vol. I. May 1995 (cit. on p. 79).

[74] Philipp Koehn. „Europarl: A Parallel Corpus for Statistical Machine Translation.“ In: *Conference Proceedings: the tenth Machine Translation Summit*. AAMT. Phuket, Thailand: AAMT, 2005, pp. 79–86. URL: http://mt-archive.info/MTS-2005-Koehn.pdf (cit. on p. 1).

[75] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009 (cit. on pp. 15, 20, 28–30, 39, 44, 60, 61, 64).

[76] Stefan Kombrink, Tomas Mikolov, Martin Karafiát, and Lukás Burget. „Recurrent Neural Network Based Language Modeling in Meeting Recognition.“ In: *INTERSPEECH*. 2011, pp. 2877–2880 (cit. on pp. 79, 80).

[77] Ted Kremenek, Andrew Y. Ng, and Dawson Engler. „A Factor Graph Model for Software Bug Finding.“ In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 2510–2516. URL: http://dl.acm.org/citation.cfm?id=1625275.1625680 (cit. on pp. 62, 63).

[78] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. „From Uncertainty to Belief: Inferring the Specification Within.“ In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pp. 161–176. URL: http://dl.acm.org/citation.cfm?id=1298455.1298471 (cit. on pp. 62, 63).

[79] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. „Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data.“ In: ICML '01. San Francisco, CA, USA, 2001, pp. 282–289. URL: http://dl.acm.org/citation.cfm?id=645530.655813 (cit. on pp. 9, 13, 15, 24, 29, 61, 66).

[80] Tessa Ann Lau. „Programming by Demonstration: A Machine Learning Approach.“ AAI3013992. PhD thesis. 2001 (cit. on pp. 101, 104).

[81] Vu Le and Sumit Gulwani. „FlashExtract: A Framework for Data Extraction by Examples.“ In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 542–

553. URL: http://doi.acm.org/10.1145/2594291.2594333 (cit. on pp. 1, 115).

[82] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. „Merlin: Specification Inference for Explicit Information Flow Problems.“ In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland: ACM, 2009, pp. 75–86. URL: http://doi.acm.org/10.1145/1542476.1542485 (cit. on pp. 62, 63).

[83] Fan Long and Martin Rinard. „Automatic Patch Generation by Learning Correct Code.“ In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 298–312. URL: http://doi.acm.org/10.1145/2837614.2837617 (cit. on pp. 12, 123, 152).

[84] Ulrike von Luxburg and Bernhard Schoelkopf. *Statistical Learning Theory: Models, Concepts, and Results*. Version 1. Oct. 27, 2008. arXiv: 0810.4752 [stat]. URL: http://arxiv.org/abs/0810.4752 (cit. on pp. 2, 53, 103, 128).

[85] Chris J. Maddison and Daniel Tarlow. „Structured Generative Models of Natural Source Code.“ In: *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. 2014, pp. 649–657 (cit. on p. 123).

[86] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. „Jungloid Mining: Helping to Navigate the API Jungle.“ In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. ACM, 2005, pp. 48–61. URL: http://doi.acm.org/10.1145/1065010.1065018 (cit. on pp. 1, 68, 69, 98).

[87] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. „A User-guided Approach to Program Analysis.“ In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 2015, pp. 462–473. URL: http://doi.acm.org/10.1145/2786805.2786851 (cit. on p. 12).

[88] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. „A Machine Learning Framework for Programming by Example." In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. 2013, pp. 187–195. URL: http://jmlr.org/proceedings/papers/v28/menon13.html (cit. on p. 121).

[89] Tomas Mikolov, Anoop Deoras, Daniel Povey, Lukás Burget, and Jan Cernocký. „Strategies for training large scale neural network language models." In: *2011 IEEE Workshop on Automatic Speech Recognition & Understanding, ASRU 2011, Waikoloa, HI, USA, December 11-15, 2011*. 2011, pp. 196–201. URL: http://dx.doi.org/10.1109/ASRU.2011.6163930 (cit. on pp. 7, 13, 68, 80).

[90] Alon Mishne, Sharon Shoham, and Eran Yahav. „Typestate-based Semantic Code Search over Partial Programs." In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '12. ACM, 2012, pp. 997–1016. URL: http://doi.acm.org/10.1145/2384616.2384689 (cit. on pp. 69, 99).

[91] Thomas M. Mitchell. *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997 (cit. on p. 128).

[92] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997 (cit. on p. 139).

[93] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012 (cit. on pp. 11, 53).

[94] Anh Tuan Nguyen and Tien N. Nguyen. „Graph-based Statistical Language Model for Code." In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 858–868. URL: http://dl.acm.org/citation.cfm?id=2818754.2818858 (cit. on pp. 13, 123, 129).

[95] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. „A Statistical Semantic Language Model for Source Code." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 532–542. URL: http://doi.

acm.org/10.1145/2491411.2491458 (cit. on pp. 13, 68, 123, 151).

[96] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. „Efficient Synthesis of Probabilistic Programs.“ In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, 2015, pp. 208–217. URL: http://doi.acm.org/10.1145/2737924.2737982 (cit. on p. 147).

[97] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. „Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation.“ In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 572–588. URL: http://doi.acm.org/10.1145/2814270.2814309 (cit. on p. 12).

[98] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. „Automatically improving accuracy for floating point expressions.“ In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 2015, pp. 1–11. URL: http://doi.acm.org/10.1145/2737924.2737959 (cit. on pp. 1, 115).

[99] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. „Type-directed Completion of Partial Expressions.“ In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. ACM, 2012, pp. 275–286. URL: http://doi.acm.org/10.1145/2254064.2254098 (cit. on pp. 1, 69, 99).

[100] David Pinto, Andrew McCallum, Xing Wei, and W. Bruce Croft. „Table Extraction Using Conditional Random Fields.“ In: SIGIR '03. Toronto, Canada, 2003, pp. 235–242. URL: http://doi.acm.org/10.1145/860435.860479 (cit. on p. 15).

[101] Ariadna Quattoni, Michael Collins, and Trevor Darrell. „Conditional random fields for object recognition.“ In: *NIPS*. 2004, pp. 1097–1104 (cit. on pp. 15, 66).

[102] Nathan D. Ratliff, J. Andrew Bagnell, and Martin Zinkevich. „(Approximate) Subgradient Methods for Structured Predic-

tion.“ In: *AISTATS*. 2007, pp. 380–387 (cit. on pp. 11, 44, 45, 62).

[103] Veselin Raychev, Pavol Bielik, and Martin Vechev. „Probabilistic Model for Code with Decision Trees.“ In: OOPSLA 2016. to appear, 2016 (cit. on pp. vii, 139).

[104] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. „Learning Programs from Noisy Data.“ In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 761–774. URL: http://doi.acm.org/10.1145/2837614.2837671 (cit. on p. vii).

[105] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. „Parallelizing User-defined Aggregations Using Symbolic Execution.“ In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. ACM, 2015, pp. 153–167. URL: http://doi.acm.org/10.1145/2815400.2815418 (cit. on p. viii).

[106] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. „Refactoring with Synthesis.“ In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM, 2013, pp. 339–354. URL: http://doi.acm.org/10.1145/2509136.2509544 (cit. on pp. viii, 151, 152).

[107] Veselin Raychev, Martin Vechev, and Andreas Krause. „Predicting Program Properties from "Big Code".“ In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. ACM, 2015, pp. 111–124. URL: http://doi.acm.org/10.1145/2676726.2677009 (cit. on p. vii).

[108] Veselin Raychev, Martin Vechev, and Manu Sridharan. „Effective Race Detection for Event-driven Programs.“ In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. ACM, 2013, pp. 151–166. URL: http://doi.acm.org/10.1145/2509136.2509538 (cit. on pp. viii, 1).

[109] Veselin Raychev, Martin Vechev, and Eran Yahav. „Automatic Synthesis of Deterministic Concurrency.“ In: *Static Analysis*. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, 2013, pp. 283–303. URL: http://dx.doi.org/10.1007/978-3-642-38856-9_16 (cit. on p. viii).

[110] Veselin Raychev, Martin Vechev, and Eran Yahav. „Code Completion with Statistical Language Models.“ In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. ACM, 2014, pp. 419–428. URL: http://doi.acm.org/10.1145/2594291.2594321 (cit. on pp. vii, 13, 103, 123, 136).

[111] Steven P. Reiss. „Semantics-based Code Search.“ In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. IEEE Computer Society, 2009, pp. 243–253. URL: http://dx.doi.org/10.1109/ICSE.2009.5070525 (cit. on p. 69).

[112] Phillip A. Relf. „Tool assisted identifier naming for improved software readability: an empirical study.“ In: *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*. 2005, pp. 53–62. URL: http://dx.doi.org/10.1109/ISESE.2005.1541814 (cit. on p. 4).

[113] Matthew Richardson and Pedro M. Domingos. „Markov logic networks.“ In: *Machine Learning* 62.1-2 (2006), pp. 107–136. URL: http://dx.doi.org/10.1007/s10994-006-5833-1 (cit. on p. 65).

[114] Ronald Rosenfeld. „Two decades of statistical language modeling: Where do we go from here.“ In: *Proceedings of the IEEE (Volume:88, Issue: 8)*. 2000, pp. 1270–1278. URL: http://dx.doi.org/10.1109/5.880083 (cit. on pp. 7, 69, 77, 78, 151).

[115] Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. „LabelMe: A Database and Web-Based Tool for Image Annotation.“ In: *Int. J. Comput. Vision* 77.1-3 (May 2008), pp. 157–173. URL: http://dx.doi.org/10.1007/s11263-007-0090-8 (cit. on p. 1).

[116] Rahul Sharma, Aditya V. Nori, and Alex Aiken. „Interpolants As Classifiers.“ In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV'12. Berkeley, CA: Springer-Verlag, 2012, pp. 71–87. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_11 (cit. on pp. 1, 12).

[117] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. „Static Specification Mining Using Automata-based Abstractions.“ In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA '07. ACM, 2007, pp. 174–184. URL: http://doi.acm.org/10.1145/1273463.1273487 (cit. on p. 69).

[118] *Shrink Your Code and Resources*. ProGuard for Android Applications: https://developer.android.com/studio/build/shrink-code.html (cit. on pp. 17, 150).

[119] Stephen Frederick Smith. „A Learning System Based on Genetic Adaptive Algorithms.“ AAI8112638. PhD thesis. Pittsburgh, PA, USA, 1980 (cit. on p. 147).

[120] Armando Solar-Lezama. „The Sketching Approach to Program Synthesis.“ In: *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. 2009, pp. 4–13. URL: http://dx.doi.org/10.1007/978-3-642-10672-9_3 (cit. on pp. 1, 6, 100).

[121] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. „Combinatorial Sketching for Finite Programs.“ In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA, 2006, pp. 404–415. URL: http://doi.acm.org/10.1145/1168857.1168907 (cit. on pp. 1, 2, 100–102, 107, 108, 121).

[122] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. „From Program Verification to Program Synthesis.“ In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 313–326. URL: http://doi.acm.org/10.1145/1706299.1706337 (cit. on pp. 1, 100).

[123] Bjarne Steensgaard. „Points-to Analysis in Almost Linear Time.“ In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: ACM, 1996, pp. 32–41. URL: http://doi.acm.org/10.1145/237721.237727 (cit. on pp. 6, 36, 76, 88).

[124] Friedrich Steimann and Jens von Pilgrim. „Refactorings Without Names." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 290–293. URL: http://doi.acm.org/10.1145/2351676.2351726 (cit. on p. 151).

[125] Andreas Stolcke. „SRILM - an extensible language modeling toolkit." In: *7th International Conference on Spoken Language Processing, ICSLP2002 - INTERSPEECH 2002, Denver, Colorado, USA, September 16-20, 2002*. 2002. URL: http://www.isca-speech.org/archive/icslp_2002/i02_0901.html (cit. on pp. 82, 88).

[126] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. „The effects of comments and identifier names on program comprehensibility: an experimental investigation." In: *J. Prog. Lang.* 4.3 (1996), pp. 143–167. URL: http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html (cit. on p. 4).

[127] Benjamin Taskar, Carlos Guestrin, and Daphne Koller. „Max-Margin Markov Networks." In: *Advances in Neural Information Processing Systems 16*. NIPS' 03. 2003, pp. 25–32 (cit. on pp. 9, 16, 44, 62).

[128] Suresh Thummalapenta and Tao Xie. „Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web." In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. ACM, 2007, pp. 204–213. URL: http://doi.acm.org/10.1145/1321631.1321663 (cit. on pp. 69, 98).

[129] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. „Large Margin Methods for Structured and Interdependent Output Variables." In: *J. Mach. Learn. Res.* 6 (Dec. 2005), pp. 1453–1484. URL: http://dl.acm.org/citation.cfm?id=1046920.1088722 (cit. on pp. 9, 44).

[130] *TypeScript*. https://www.typescriptlang.org/ (cit. on pp. 5, 17).

[131] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. „Soot - a Java Bytecode Optimization Framework." In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL:

http://dl.acm.org/citation.cfm?id=781995.782008 (cit. on pp. 1, 88, 90).

[132] Martin Vechev and Eran Yahav. „Deriving Linearizable Fine-grained Concurrent Objects." In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA, 2008, pp. 125–135. URL: http://doi.acm.org/10.1145/1375581.1375598 (cit. on p. 100).

[133] Martin Vechev, Eran Yahav, and Greta Yorsh. „Abstraction-guided Synthesis of Synchronization." In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: ACM, 2010, pp. 327–338. URL: http://doi.acm.org/10.1145/1706299.1706338 (cit. on pp. 1, 121).

[134] Henry S. Warren. *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on pp. 116, 119).

[135] Andrzej Wasylkowski and Andreas Zeller. „Mining Temporal Specifications from Object Usage." In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. IEEE Computer Society, 2009, pp. 295–306. URL: http://dx.doi.org/10.1109/ASE.2009.30 (cit. on p. 68).

[136] Westley Weimer and George C. Necula. „Mining Temporal Specifications for Error Detection." In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'05. Springer-Verlag, 2005, pp. 461–476. URL: http://dx.doi.org/10.1007/978-3-540-31980-1_30 (cit. on p. 68).

[137] Ian H. Witten and Timothy C. Bell. „The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression." In: *IEEE Transactions on Information Theory* 37.4 (1991), pp. 1085–1094 (cit. on pp. 79, 138).

[138] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. „Perracotta: Mining Temporal API Rules from Imperfect Traces." In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 282–291. URL: http://doi.acm.org/10.1145/1134285.1134325 (cit. on p. 99).

[139] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. „Data-driven Synthesis for Object-oriented Frameworks.“ In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA, 2011, pp. 65–82. URL: http://doi.acm.org/10.1145/2048066.2048075 (cit. on p. 99).

[140] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. „MAPO: Mining and Recommending API Usage Patterns.“ In: *Proceedings of the 23rd European Conference on Object-Oriented Programming*. ECOOP' 09. Springer-Verlag, 2009, pp. 318–343. URL: http://dx.doi.org/10.1007/978-3-642-03013-0_15 (cit. on pp. 69, 99).

[141] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. „Parallelized stochastic gradient descent.“ In: *NIPS*. 2010, pp. 2595–2603 (cit. on p. 47).

COLOPHON

This document was typeset in LaTeX using the typographical look-and-feel `classicthesis`. Most of the graphics in this thesis are generated using `pgfplots` and `pgf/tikz`. The bibliography is typeset using `biblatex`.