Predicting Program Properties from "Big Code"

Veselin Raychev
Department of Computer Science
ETH Zürich
veselin.raychev@inf.ethz.ch

Martin Vechev
Department of Computer Science
ETH Zürich
martin.vechev@inf.ethz.ch

Andreas Krause
Department of Computer Science
ETH Zürich
andreas.krause@inf.ethz.ch

Abstract

We present a new approach for predicting program properties from massive codebases (aka "Big Code"). Our approach first learns a probabilistic model from existing data and then uses this model to predict properties of new, unseen programs.

The key idea of our work is to transform the input program into a representation which allows us to phrase the problem of inferring program properties as structured prediction in machine learning. This formulation enables us to leverage powerful probabilistic graphical models such as conditional random fields (CRFs) in order to perform *joint* prediction of program properties.

As an example of our approach, we built a scalable prediction engine called JSNICE[1] for solving two kinds of problems in the context of JavaScript: predicting (syntactic) names of identifiers and predicting (semantic) type annotations of variables. Experimentally, JSNICE predicts correct names for 63% of name identifiers and its type annotation predictions are correct in 81% of the cases. In the first week since its release, JSNICE was used by more than 30,000 developers and in only few months has become a popular tool in the JavaScript developer community.

By formulating the problem of inferring program properties as structured prediction and showing how to perform both learning and inference in this context, our work opens up new possibilities for attacking a wide range of difficult problems in the context of "Big Code" including invariant generation, de-compilation, synthesis and others.

1. Introduction

The increased amounts of freely available high quality programs in code repositories such as GitHub[2] (a situation termed "Big Code" [7] by a recent initiative) creates a unique opportunity for new kinds of programming tools based on statistical reasoning. These tools will extract useful information from existing codebases and will use that information to provide statistically likely solutions to problems that are difficult or impossible to solve with traditional rule based techniques.

[1] http://jsnice.org

[2] http://github.com

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2677009

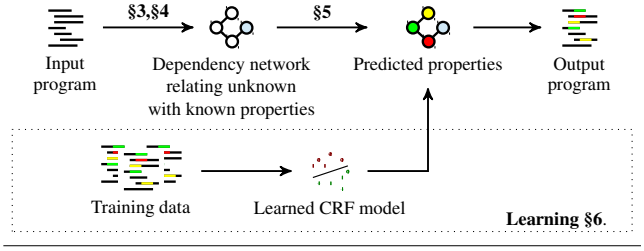

Figure 1. Statistical Prediction of Program Properties.

In this work we focus on the general 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). Our approach is fairly general and can serve as a basis for various kinds of statistical invariant predictions and statistical code synthesizers [24].

The core technical insight of our work 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) [17], a powerful undirected graphical model successfully used in a wide variety of applications including computer vision, information retrieval, and natural language processing [4, 10, 17, 21, 22].

To our knowledge, this is the first work which shows how to leverage CRFs in the context of programs. By connecting programs to CRFs, our work also enables immediate reuse of state-of-the-art learning and inference algorithms [14] to the domain of programs.

Fig. 1 illustrates our two-phase structured prediction approach. In the prediction phase (shown on top), 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 [14]. This query makes a *joint* prediction for all elements together by optimizing 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 learning phase, 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 [26].

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 [6, 28]. 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) [11]. 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.

Main contributions The contributions of this paper are:

- A new approach for inferring likely program facts based on structured prediction with conditional random fields (CRFs).
- A new framework consisting of fast and approximate inference and learning algorithms tailored to structured prediction tasks in programs.
- A new system, JSNICE, which is an instance of our approach focusing on predicting names and type annotations of JavaScript programs. A week after its release[3], JSNICE was used by $> 30,000$ developers and has since become a popular tool in the JavaScript developer community.
- An evaluation on a range of real-world JavaScript programs. The experimental results indicate that JSNICE successfully predicts correct names for 63.4% of program identifiers and 81.6% of the guessed type annotations are correct.

Paper structure The paper is organized as follows. Section 2 illustrates our approach on an example. Section 3 introduces the general prediction framework, while Section 4 describes JSNICE, an instantiation of that framework. Section 5 and Section 6 discuss our structured prediction and learning procedures. Section 7 presents a detailed experimental evaluation of JSNICE. In Section 8 we elaborate on our design choices and explain why and how we arrived at using structured prediction with CRFs. Finally, in Section 9 we discuss related work in detail and conclude in Section 10.

2. 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(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(a), our system produced the program in Fig. 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 an overview of the prediction procedure. We focus on predicting names, but the process for predicting types is identical.

Determine known and unknown properties Given the program in Fig. 2(a), we first determine the set of program elements for which we would like to infer properties. These are elements for which the properties to be inferred are currently *unknown*. For example, in the case of name inference, this set of elements includes the local variables of the input program: `e`, `t`, `n`, `r`, and `i`. We also determine the set of elements whose properties are *known*. 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(b). The goal of our prediction task is to predict the *unknown* properties based on: i) the obtained *known* properties, and ii) the relationship between various elements (discussed below).

Build dependency network Next, 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.

In Fig. 2(c), we show three example dependencies between the program elements. For instance, the statement `i += t` generates a dependency $\langle$`i,t,`**`L+=R`**$\rangle$, 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 $\langle$`r,length,`**`L=_.R`**$\rangle$ which designates that the left part of the relationship, denoted by **`L`**, appears before the de-reference of the right side denoted by **`R`** (we elaborate on the different types of relationships later in the paper). For clarity, in Fig. 2(c) we include only some of the relationships.

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 [14]. As illustrated in Fig. 2(d), for the network of Fig. 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(d). This is the same network as in Fig. 2(c) but with additional tables we elaborate on now (these tables are produced as an output of the learning 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

[3] http://jsnice.org

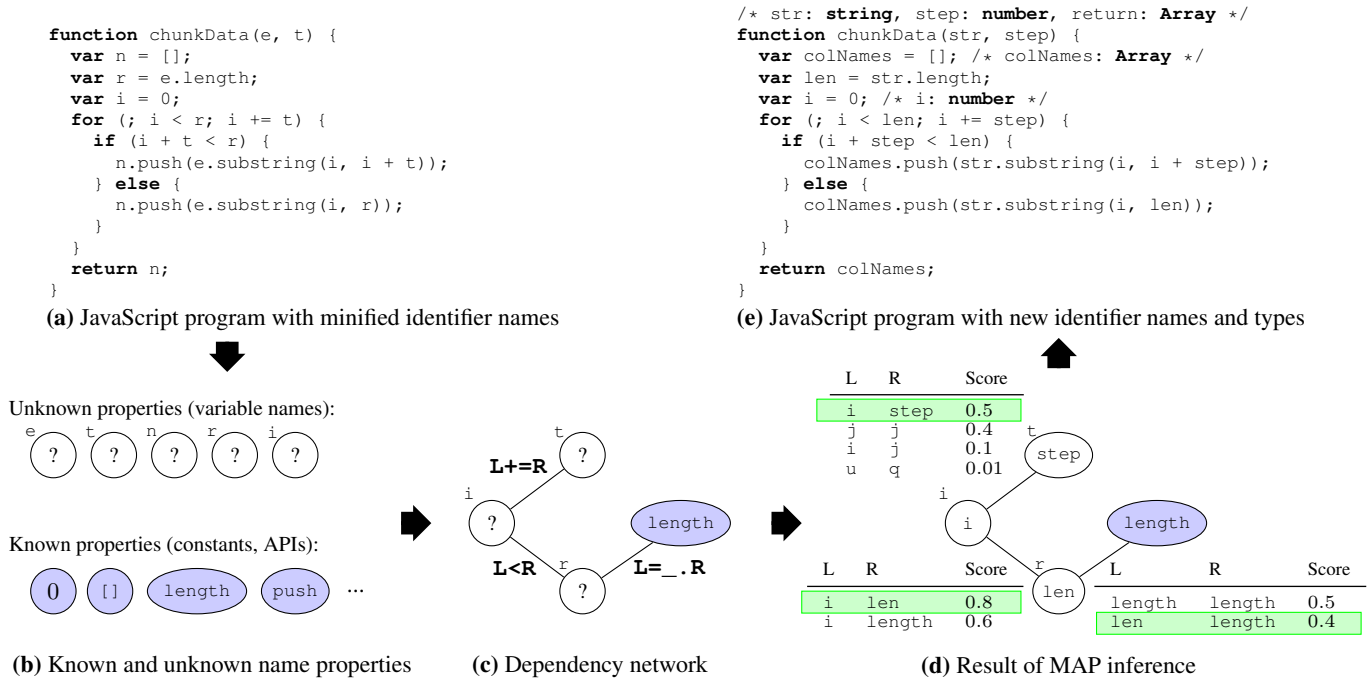

Figure 2. A JavaScript program with new names and type annotations, along with an overview of the name inference procedure.

two properties and returns the score for the pair (intuitively, how likely is the particular pair). In Fig. 2(d), each table shows possible functions for the three kinds of relationships we have.

Let us consider the topmost table. 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 a particular scoring function. For the two nodes `i` and `t`, the inference ends up selecting the highest score from the 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(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, 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(e), the predicted type annotations 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.

A note on name inference 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).

3. 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 challenges (we instantiate it for two challenges in Section 4).

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. For convenience, we assume that elements of the first kind are indexed in the range $[1..n(x)]$ and elements of the second kind are indexed in the range

$[n(x)+1, n(x)+m(x)]$. 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 property can take. For instance, in type prediction, $Labels_U$ contains all possible types (e.g. number, string, etc). Then, for a program x, we use the notation $\mathbf{y} = (y_1, ..., y_{n(x)})$ to denote a vector of predicted program properties. Here, $\mathbf{y} \in Y$ where $Y = (Labels_U)^*$. That is, each entry y_i in the vector $\mathbf{y}$ ranges over $Labels_U$ and denotes that program element i has a property y_i.

Problem definition Let $D = \{\langle x^{(j)}, \mathbf{y}^{(j)}\rangle\}_{j=1}^{t}$ denote the training data: a set of t programs each with corresponding program properties. Our goal is to learn a model that captures the conditional probability $Pr(\mathbf{y} \mid x)$. 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 $\mathbf{y} = \text{argmax}_{\mathbf{y}' \in \Omega_x} Pr(\mathbf{y}' \mid x)$

That is, for a new program x, we aim to find the most likely assignment of program properties $\mathbf{y}$ according to the probabilistic distribution. Here, $\Omega_x \subseteq Y$ describes the set of possible assignments of properties $\mathbf{y}'$ 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.

3.1 Conditional Random Fields (CRFs)

We now describe CRFs, a particular model for representing the conditional probability $Pr(\mathbf{y} \mid x)$. We consider the case where the factors are positive in which case, without loss of generality, any conditional probability of properties $\mathbf{y}$ given a program x can be encoded as follows:

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

where $score$ is a function that returns a real number indicating the score of an assignment of properties $\mathbf{y}$ 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 $\mathbf{y}$ sum to 1, i.e.:

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

We consider $score$ functions that can be expressed as a composition of a sum of k feature functions f_i associated with weights w_i:

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

Here, $\mathbf{f}$ is a vector of functions f_i and $\mathbf{w}$ is a vector of weights w_i. The feature functions $f_i \colon Y \times X \rightarrow \mathbb{R}$ are used to score assignments of program properties. This representation of score functions is particularly suited for learning (as the weights $\mathbf{w}$ can be learned from data). Based on the definition above, we can now define a *conditional random field* [17].

Definition 3.1 (Conditional Random Field (CRF))**.** *A model for the conditional probability of labels $\mathbf{y}$ given observations x is called (log-linear) conditional random field, if it is represented as:*

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

A note on feature functions Feature functions are key to controlling the likelihood of an assignment of properties $\mathbf{y}$ 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 $f_i(\mathbf{y}^B, x) = -\infty$, the feature function f_i (with weight $w_i > 0$) disables an assignment $\mathbf{y}^B$, thus resulting in $Pr(\mathbf{y}^B \mid x) = 0$.

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 a program's types, we define one set of feature functions and when we predict identifier names, we define another set. Once defined, the feature functions are re-used for predicting the particular kind of property we are interested in for *any* input program.

3.2 Making Predictions for Programs

We next describe a step by step process for predicting program properties using CRFs where program elements are related with pairwise functions. We first 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.

Step 1: Build dependence network G^x The first step in defining $f_i(\mathbf{y}, x)$ is to build what we refer to as a dependency network $G^x = \langle V^x, E^x\rangle$ from the input program x. This network captures dependencies between the predictions made for the program elements of interest. Here, $V^x = V_U^x \cup V_K^x$ denotes the set of program elements (e.g. variables) and consists of elements for which we would like to predict properties V_U^x and elements whose properties we already know V_K^x. 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.

For a program x, we define the vector $\mathbf{z}^x = \{z_1^x, ..., z_m^x\}$ to capture the set of properties that are already known, that is, each element in V_K^x is assigned a property from $\mathbf{z}^x$. Here, z_i^x denotes the property of program element $n + i$. Each z_i^x 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 $\mathbf{z}$ instead of $\mathbf{z}^x$. We use $Labels = Labels_U \cup Labels_K$ to denote the set of all properties.

Step 2: Define feature functions Once the network G^x for a program x is obtained, we use it to define the shape of the feature functions. We define the assignment vector $A = (\mathbf{y}, \mathbf{z})$ which is a concatenation of two assignments: the unknown properties $\mathbf{y}$ and the known properties $\mathbf{z}$. As usual, we access the property of the j'th element of the vector A via A_j. We define a feature function f_i as the sum of the applications of its corresponding pairwise feature function ψ_i over the set of network edges obtained from step 1. That is, the formula below allows us to compute the value of a particular feature function for a given prediction $\mathbf{y}$:

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

Here, each $\psi_i \colon Labels \times Labels \times Rels \rightarrow \mathbb{R}$ is a pairwise feature function relating a *pair* of program properties as opposed to a larger number like f_i. Recall that each edge $(a, b, rel) \in E^x$ represents a pair of elements a and b and the kind of relationship $rel \in Rels$ between them. Then, every time we encounter an edge between two program elements, we apply the pairwise feature function passing in the appropriate relationship rel as an argument. Conversely, if two program elements are unrelated there is no need to invoke the pairwise feature function for these two elements. Focusing on pairwise feature functions allows us to define f_i directly on the network obtained from the program. Note again that pairwise feature functions ψ_i are pre-defined once and for all *independently* of the program for which we are predicting properties. We will see particular instantiations of pairwise feature functions in later sections.

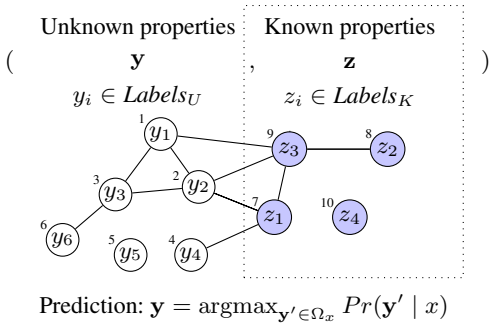

Figure 3. A general schema for building a network for a program x and finding the best scoring assignment of program properties $\mathbf{y}$.

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.

Step 3: Score a prediction $\mathbf{y}$ Based on the above definition of a feature function, we can now define how to obtain a total score for a prediction $\mathbf{y}$. By substitution, we obtain:

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

That is, for a program x and its dependence 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 $\mathbf{y}$.

Example Let us illustrate the above steps as well as some key points on the simple example in Fig. 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 $\mathbf{z}$ 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 $Pr(\mathbf{y} \mid x)$ is maximal.

Key Points Let us note three important points. 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 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 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 by both the inference and the learning algorithms. We do not discuss conditional independence further but refer the reader to a standard reference [14]. 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 prediction for node 6 and vice versa.

3.3 MAP inference

Recall that the key query we perform is MAP inference:

Given a program x, find $\mathbf{y} = \text{argmax}_{\mathbf{y}' \in \Omega_x} Pr(\mathbf{y}' \mid x)$

In a CRF, this amounts to the query:

$$\mathbf{y} = \underset{\mathbf{y}' \in \Omega_x}{\text{argmax}} \frac{1}{Z(x)} \exp(score(\mathbf{y}', x))$$

where:

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

Note that $Z(x)$ does not depend on $\mathbf{y}'$ and as a result it does not affect the final choice for the prediction $\mathbf{y}$. This is an important observation, because computing $Z(x)$ is generally very expensive as it may need to sum over all possible assignments $\mathbf{y}''$. 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:

$$\mathbf{y} = \underset{\mathbf{y}' \in \Omega_x}{\text{argmax}}\, score(\mathbf{y}', x)$$

This means that an algorithm answering the MAP inference query must ultimately maximize the $score$ function. For instance, for the example in Fig. 3, once we fix the labels z_i, we need to find labels y_i 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 $\mathbf{y}' \in \Omega_x$ and scoring each of them to select the highest scoring one. Other exact and inexact [14] 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 [14], 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. Towards that, in Section 5 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.

3.4 Learning

We briefly discuss how we learn the weights $\mathbf{w}$ that describe the scoring function $score$. To learn $\mathbf{w}$, we use an advanced learning technique that generalizes support vector machines. Given the training data $D = \{\langle x^{(j)}, \mathbf{y}^{(j)} \rangle\}_{j=1}^{t}$ of t samples, our goal is to find $\mathbf{w}$ such that the given assignments $\mathbf{y}^{(j)}$ 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 6.

4. JSNICE: Predicting Names and Type Annotations for JavaScript

In this section we present an example of using our structured prediction approach presented in Section 3 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 almost as-is to other languages.

Presentation Flow Recall that in Section 3, we defined a three-step process to obtaining a total score for a prediction, where the first step is to define the network $G^x = \langle V^x, E^x \rangle$ and the second step is to define the pairwise feature functions ψ_i. The combination of these two fully defines the feature functions f_i. 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.

4.1 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 in our framework is as follows. First, as outlined in Section 3, 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 $\mathit{Labels}_K = JSConsts \cup JSNames$, 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 size and a string constant on the right-hand size. 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.

4.2 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 [11].

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[4].

[4] https://developers.google.com/closure/compiler/docs/js-for-compiler

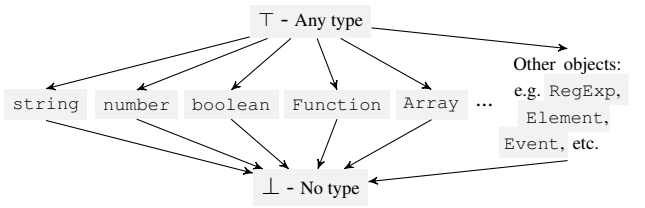

Figure 4. The lattice of types over which prediction occurs.

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

$$expr ::= val \mid var \mid expr_1(expr_2) \mid expr_1 \circledast expr_2 \quad \textit{Expression}$$
$$val ::= \lambda var : \tau.expr \mid n \quad \textit{Value}$$

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:

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

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 $[]_x(e)$.

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:

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

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). 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:

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

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

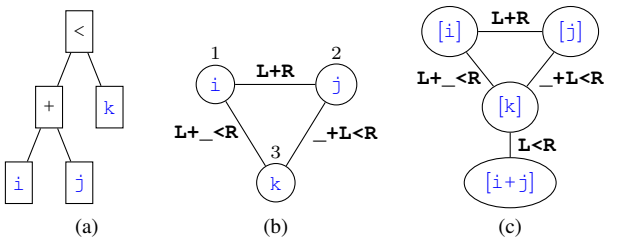

Figure 5. (a) the AST of expression `i+j<k`, and two dependency networks built from the AST relations: (b) for name predictions, and (c) for type predictions.

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. 4. In the figure we use "..." to denote a potentially infinite number of user-defined object types.

Key points 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` ∨ `number`.

4.3 Relating program elements

We next describe the relationships that 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.

4.3.1 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. 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. 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:

$$rel_{ast} ::= rel_L(rel_R) \mid rel_L \circledast rel_R$$
$$rel_L ::= \mathtt{L} \mid rel_L(\_) \mid \_(rel_L) \mid rel_L \circledast \_ \mid \_ \circledast rel_L$$
$$rel_R ::= \mathtt{R} \mid rel_R(\_) \mid \_(rel_R) \mid rel_R \circledast \_ \mid \_ \circledast rel_R$$

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).

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. 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 `[L]` ⊒ `[R]` where ⊒ is the super-type relationship (indicated in Fig. 4). Interestingly, our inference model can learn such rules instead of providing them explicitly.

4.3.2 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 [25]).

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. In the case of predicting names e_2 is always a variable, while with 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 $c \in alias(b)$ where c is a variable, we include the edge $(a, c, (r, \texttt{ALIAS}))$.

4.3.3 Function name relationships

We also 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.

4.4 Pairwise feature functions

Finally, we describe how to obtain and 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 $D = \{\langle x^{(j)}, \mathbf{y}^{(j)}\rangle\}_{j=1}^t$ consists of t programs where for each program x we are given the corresponding properties $\mathbf{y}$. For each tuple $(x, \mathbf{y}) \in D$, we define the set of features as follows:

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

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

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

We then define the pairwise feature functions to be indicator functions of each feature triple $\langle l_i^1, l_i^2, rel_i\rangle \in all\_features(D)$:

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

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.

5. Prediction Algorithm

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

$$\mathbf{y} = \operatorname*{argmax}_{\mathbf{y}' \in \Omega_x} Pr(\mathbf{y}'|x) = \operatorname*{argmax}_{\mathbf{y}' \in \Omega_x} score(\mathbf{y}', x) = \operatorname*{argmax}_{\mathbf{y}' \in \Omega_x} \mathbf{w}^T \mathbf{f}(\mathbf{y}', x)$$

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 [14].

Approximate algorithms Previous studies [8, 12] 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 more advanced 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 [4]). 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 4.4. 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.

Algorithm 1: Greedy Inference Algorithm

Input: network $G^x = \langle V^x, E^x\rangle$ of program x,
initial assignment of n unknown properties $\mathbf{y}_0 \in \Omega_x$,
known properties $\mathbf{z}$
pairwise feature functions ψ_i and their learned weights w_i
Output: $\mathbf{y} \approx \operatorname{argmax}_{\mathbf{y}' \in \Omega_x} \big(score(\mathbf{y}', x)\big)$

```
1  begin
2      y ← y_0
3      for pass ∈ [1..num_passes] do
4          // for each node with unknown property in the graph G^x
5          for v ∈ [1..n] do
6              E_v ← {(v, _, _) ∈ E^x} ∪ {(_, v, _) ∈ E^x}
7              score_v ← scoreEdges(E_v, (y, z))
8              for l' ∈ candidates(v, (y, z), E_v) do
9                  l ← y_v        // get current label of v
10                 y_v ← l'       // change label of v in y
11                 score'_v ← scoreEdges(E_v, (y, z))
12                 if y ∈ Ω_x ∧ score'_v > score_v then
13                     score_v ← score'_v
14                 else
15                     y_v ← l    // no score improvement: revert label.
16     return y
```

5.1 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 $\mathbf{y}_0$, iii) the obtained known properties $\mathbf{z}$, and iv) the pairwise feature functions and their weights. The way these inputs are obtained was already described earlier in Section 3. The output of the algorithm is an approximate prediction $\mathbf{y}$ which also conforms to the desired 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 function to each edge and sums up the results.

The basic idea of the algorithm is to start with an initial assignment $\mathbf{y_0}$ (Line 2) and to make a number of passes over all nodes in

the network, attempting to improve the score of the current prediction $\mathbf{y}$. 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).

5.2 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. Recall that 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:

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

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. Thanks to this observation, the prediction runs orders of magnitude faster than a naïve greedy algorithms that tries 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 $\mathbf{y}$, after a certain number of iterations, we reach a fixed point assignment $\mathbf{y}$ 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.

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.

6. Learning

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

The goal of learning is to then estimate the parameters $\mathbf{w}$ 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 $\mathbf{y}$ are predicted accurately (using the prediction algorithm from Section 5). 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 $Pr(\mathbf{y} \mid x)$ accurately fits the true conditional distribution $P(\mathbf{y} \mid x)$ associated with the data-generating distribution P. Unfortunately, this task requires computation of the partition function $Z(x)$ which is a formidable task [14].

Instead, we perform what is known as *max-margin training*: we learn weights $\mathbf{w}$ such that the training data is classified correctly subject to additional constraints like margin and regularization. For this task, powerful learning algorithms are available [26, 27]. In particular, we use a variant of the *Structured Support Vector Machine (SSVM)*[5] [27] and we train it efficiently with the scalable subgradient descent algorithm proposed in [23].

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

$$\forall j, \forall \mathbf{y}' \in \Omega_{x^{(j)}}\ score(\mathbf{y}^{(j)}, x^{(j)}) \geq score(\mathbf{y}', x^{(j)}) + \Delta(\mathbf{y}^{(j)}, \mathbf{y}')$$

Here, Δ: *Labels** $\times$ *Labels** $\rightarrow \mathbb{R}$ is a distance function (non-negative and satisfying triangle inequality). One can interpret $\Delta(\mathbf{y}^{(j)}, \mathbf{y}')$ as a (safety) margin between the given assignment $\mathbf{y}^{(j)}$ and any other assignment $\mathbf{y}'$, w.r.t. the score function. Δ is chosen such that slight mistakes (e.g., incorrect prediction of few properties) require less margin than major mistakes. For our appli-

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

cations, we took Δ to return the number of different labels between the reference assignment $\mathbf{y}^{(j)}$ and any other assignment $\mathbf{y}'$.

Generally, it may not be possible to find weights achieving the above constraints. Hence, SSVMs attempt to find weights $\mathbf{w}$ 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.

6.1 Learning with stochastic gradient descent

Achieving the balance of data-fit and model complexity leads to a natural optimization problem:

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} \sum_{j=1}^{t} \ell(\mathbf{w}; x^{(j)}, \mathbf{y}^{(j)}) \text{ s.t. } \mathbf{w} \in \mathcal{W}_\lambda \qquad (1)$$

where

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

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 $\mathbf{w}$. Thus, if the objective (1) 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

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

The SSVM optimization problem (1) 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 [23].

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. $\mathbf{w}$) of the loss function $\ell(\mathbf{w}; x^{(j)}, \mathbf{y}^{(j)})$ and takes a step in the negative gradient direction. If it ends up outside the feasible region $\mathcal{W}_\lambda$, it projects $\mathbf{w}$ to the closest feasible point.

In order to compute the gradient $\mathbf{g} = \nabla_{\mathbf{w}} \ell(\mathbf{w}; x^{(j)}, \mathbf{y}^{(j)})$ at $\mathbf{w}$ w.r.t. the j-th program, we must solve the problem

$$\mathbf{y}_{best} \leftarrow \operatorname*{argmax}_{\mathbf{y}' \in \Omega_{x^{(j)}}} \big(\mathit{score}(\mathbf{y}', x^{(j)}) + \Delta(\mathbf{y}^{(j)}, \mathbf{y}') \big), \qquad (2)$$

resulting in the gradient $\mathbf{g}$

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

Hence, computing the gradient requires solving the *loss-augmented inference* problem (2). This problem can be (approximately) solved using the algorithm presented in Section 5.

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

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

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

6.2 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 $\mathbf{w} \in \mathbb{R}^k$ 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 $\mathbf{w}$ is good in predicting training data, but fails to generalize to unseen data. In our case, we perform $\ell^{\inf}$ regularization, which can be efficiently done in closed form as follows:

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

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 $\ell^{\inf}$ 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 $\mathbf{g}$ is non-zero. This enables us to use a sparse representation of the vectors $\mathbf{g}$, avoiding iteration over all components of the vector $\mathbf{w}$ when projecting.

6.3 Complete training phase

In summary, our training procedure first iterates once over the training data and extracts features. We initialize each weight with $w_i = 1/(2\lambda)$. Then, we start with a learning rate of $\alpha = 0.1$ and iterate in multiple passes to learn their 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 wrong 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 [29]. At each pass, we randomly split the data to threads where each thread t_i updates its own version of the weights $\mathbf{w}^{t_i}$. At the end of each pass, the weights $\mathbf{w}^{t_i}$ are averaged to obtain the final weights $\mathbf{w}$.

7. 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 integrated within the Google Closure Compiler [6], a tool which takes human-readable JavaScript with optional type annotations and typechecks it. It then returns an optimized, minified and human-unreadable JavaScript with stripped annotations.

To implement 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 $30,000$ 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. For evaluation, we took the 50 JavaScript projects with the highest number of commits

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

from BitBucket[6]. 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 7.1), then precision (Section 7.2) and model sizes (Section 7.3), and finally the running times (Section 7.4).

7.1 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 [20]. Cross-validation works by splitting the training data into 10 equal pieces called folds and evaluating the error rate 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 error rate.

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 6) should return zero or the number of different labels between the two assignments. To reduce computation (since we must train and test a large number of parameters), we performed cross-validation on only 1% sample of the training data. The 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.

7.2 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 [7]. The 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. Then, we used JSNICE to 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 learning 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`.

7.2.1 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 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).

7.2.2 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 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 present two metrics for types: 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 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 can be inferred by regular forward type analysis.

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 increase the precision and recall of type prediction, adding more (semantic) relationships between program elements will 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

[6] http://bitbucket.org

[7] https://github.com/mishoo/UglifyJS

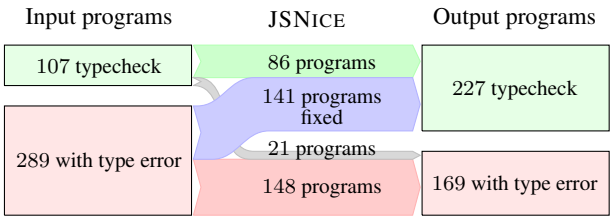

Figure 6. Evaluation results for the number of typechecking programs with manually provided types and with predicted types.

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. 6. 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 types – annotations that almost no program 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 types were rejected due to imprecision of the type system.

7.3 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.

7.4 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.

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

Running times of prediction We evaluated the effect of changing the beam size s of our MAP inference algorithm (from Section 5), and the effect s has on the prediction time. The average prediction times per program are summarized in Table 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 [4] 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, we chose the value $s = 64$. This value provides a good balance between performance and precision suitable for our live 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 CRF for name prediction has on average 383.5 arcs and 29.2 random variables. For the type prediction evaluation tasks, each CRF has on average 109.5 arcs and 12.6 random variables.

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

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 4.4. However, the framework from Section 3 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) [14]. 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).

Training a Markov Network: Discriminative over Generative 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_K^x and V_U^x (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_K^x. Providing such a prior distribution can however be very difficult in practice.

However, recall that our MAP inference query is in fact *conditional* on an existing assignment of the known elements V_K^x. This means that the underlying probabilistic model need only capture the *conditional* probability distribution of V_U^x *given* V_K^x and *not* the joint distribution. An undirected graphical model able to capture such conditional distributions is referred to as a Conditional Random Field (CRF) [17]. The CRF model admits *discriminative* training where priors on V_K^x are no longer necessary, a key reason for why this model is so effective and popular in practice.

Training a CRF: 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 can be computationally expensive (see Ch.20, [14]).

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_U^x. Because of that, we are now able to leverage recent advances in scalable training methods for CRFs and in particular max-margin training [23, 26], a method geared towards answering MAP inference queries on the trained CRF 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 an 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.

8.1 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.

9. Related Work

We next discuss some of the work most closely related to ours.

Probabilistic models for code Recently, there has been interest in creating probabilistic models of programs based on large codebases [1, 19, 24]. In contrast to such *generative* models, in this work we model a conditional distribution over likely properties for any given program. Such *discriminative* approaches are often more scalable, in terms of their data requirements and computational efficiency [20], and enable us to capture complex relationships between program properties. While there are other probabilistic models [1, 2, 5, 13], they typically serve different purposes. CRFs, as we use here, are particularly well suited for making multiple, dependent predictions for complex, structured data like programs.

Graphical models in programming Several works have used graphical models in the context of programs [3, 9, 15, 16, 18]. All of these works phrase the prediction problem as computing marginals. As we already discussed in Section 8, we find MAP inference to be the conceptually preferred problem formulation over marginals. Except for [15], 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 structured prediction (where one need not compute the partition function). Indeed, the learning method of [15] 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 procedure that is a lot less scalable than MAP inference.

In terms of particular applications, [16] 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 handle efficiently 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 make 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 devastating in an interactive setting (even with optimizations). Indeed, their follow-up work (section 3, [15]) 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, [18] 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 very inefficient: it essentially requires running inference N times for N variables, instead of once. Further, the approach cannot compute optimal MAP inference. The paper [3] is similar to [18] 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 [18] (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).

10. Conclusion

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. JSNICE is practically effective: it was used by more than $30,000$ developers a week after its release and has now become a popular tool in the JavaScript developer community.

We believe the structured prediction approach presented in this work can serve as a basis for exploring approximate solutions to a variety of analysis and synthesis tasks in the context of "Big Code".

References

[1] ALLAMANIS, M., AND SUTTON, C. Mining source code repositories at massive scale using language modeling. In *MSR* (2013).

[2] ANDRZEJEWSKI, D., MULHERN, A., LIBLIT, B., AND ZHU, X. Statistical debugging using latent topic models. In *ECML* (2007).

[3] BECKMAN, N. E., AND NORI, A. V. Probabilistic, modular and scalable inference of typestate specifications. PLDI '11, pp. 211–221.

[4] BESAG, J. On the Statistical Analysis of Dirty Pictures. *Journal of the Royal Statistical Society. Series B (Methodol.) 48*, 3 (1986), 259–302.

[5] BLEI, D., AND LAFFERTY, J. Topic models. In *Text Mining: Classification, Clustering, and Applications*. 2009.

[6] Closure compiler. https://developers.google.com/closure/compiler/.

[7] Mining big code to improve software reliability and construction. http://www.darpa.mil/NewsEvents/Releases/2014/03/06a.aspx.

[8] FINLEY, T., AND JOACHIMS, T. Training structural svms when exact inference is intractable. In *ICML* (2008), pp. 304–311.

[9] GULWANI, S., AND JOJIC, N. Program verification as probabilistic inference. POPL '07, ACM, pp. 277–289.

[10] HE, X., ZEMEL, R. S., AND CARREIRA-PERPIÑÁN, M. A. Multiscale conditional random fields for image labeling. CVPR '04.

[11] JENSEN, S. H., MØLLER, A., AND THIEMANN, P. Type analysis for javascript. In *SAS'09*.

[12] KAPPES, J. H., ET AL. A comparative study of modern inference techniques for discrete energy minimization problems. CVPR'13.

[13] KARAIVANOV, S., RAYCHEV, V., AND VECHEV, M. Phrase-based statistical translation of programming languages. Onward! '14.

[14] KOLLER, D., AND FRIEDMAN, N. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[15] KREMENEK, T., NG, A. Y., AND ENGLER, D. A factor graph model for software bug finding. IJCAI'07.

[16] KREMENEK, T., TWOHEY, P., BACK, G., NG, A., AND ENGLER, D. From uncertainty to belief: Inferring the specification within. OSDI '06, USENIX Association, pp. 161–176.

[17] LAFFERTY, J. D., MCCALLUM, A., AND PEREIRA, F. C. N. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. ICML '01, pp. 282–289.

[18] LIVSHITS, B., NORI, A. V., RAJAMANI, S. K., AND BANERJEE, A. Merlin: Specification inference for explicit information flow problems. PLDI '09, ACM, pp. 75–86.

[19] MADDISON, C. J., AND TARLOW, D. Structured generative models of natural source code.

[20] MURPHY, K. P. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[21] PINTO, D., MCCALLUM, A., WEI, X., AND CROFT, W. B. Table extraction using conditional random fields. SIGIR '03, pp. 235–242.

[22] QUATTONI, A., COLLINS, M., AND DARRELL, T. Conditional random fields for object recognition. In *NIPS* (2004), pp. 1097–1104.

[23] RATLIFF, N. D., BAGNELL, J. A., AND ZINKEVICH, M. (approximate) subgradient methods for structured prediction. In *AISTATS* (2007), pp. 380–387.

[24] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language models. PLDI '14, ACM, pp. 419–428.

[25] STEENSGAARD, B. Points-to analysis in almost linear time. POPL '96, pp. 32–41.

[26] TASKAR, B., GUESTRIN, C., AND KOLLER, D. Max-margin markov networks. In *NIPS* (2003).

[27] TSOCHANTARIDIS, I., JOACHIMS, T., HOFMANN, T., AND ALTUN, Y. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research 6, 2005*, 1453–1484.

[28] Typescript language. http://www.typescriptlang.org/.

[29] ZINKEVICH, M., WEIMER, M., LI, L., AND SMOLA, A. J. Parallelized stochastic gradient descent. In *NIPS* (2010), pp. 2595–2603.