Probabilistic Type Inference by Optimizing Logical and Natural Constraints

Anonymous Author(s)

Abstract

We present a new approach to the type inference problem for dynamic languages. Our goal is to combine logical constraints, that is, deterministic information from a type system, with natural constraints, uncertain information about types from sources like identifier names. To this end, we introduce a framework for probabilistic type inference that combines logic and learning: logical constraints on the types are extracted from the program, and deep learning is applied to predict types from surface-level code properties that are statistically associated, such as variable names. The main insight of our method is to constrain the predictions from the learning procedure to respect the logical constraints, which we achieve by relaxing the logical inference problem of type prediction into a continuous optimization problem. To evaluate the idea, we build a tool called PRODTS to predict a TypeScript declaration file for a JavaScript library. PRODTS combines a continuous interpretation of logical constraints derived by a simple augmented static analysis of the JavaScript code, with natural constraints obtained from a deep learning model, which learns naming conventions for types from a large code base. We evaluate PRODTS on a data set of 5,800 open source JavaScript projects which have type annotations in the well-known DefinitelyTyped repository. We find that combining logical and natural constraints yields a large improvement in performance over either kind of information individually, and produces 50% fewer incorrect type predictions than previous approaches from the research literature.

Keywords Type Inference, Dynamic Languages, TypeScript, Continuous Relaxation, Numerical Optimization, Deep Learning

1 Introduction

Statically-typed programming languages aim to enforce correctness and safety properties on programs by guaranteeing constraints on program behaviour. Large scale user-studies suggest that programmers benefit from type safety in tasks such as class identification and type error fixing [?]. However, type safety comes at a cost: these languages require explicit type annotations, this comes with additional cost for the programmer who needs to declare and maintain the annotations. Strongly statically-typed, usually functional languages, like Haskell or ML, offer strong type inference procedures that reduce the cost of explicitly writing types but come with a steep learning curve. On the other hand, dynamically typed languages are intuitive and popular [?], at the expense of being more error-prone. To compromise between static and dynamic typing, the programming language community has developed hybrid approaches for type systems such as gradual typing systems [?] or rich type inference procedures, see **??**. Although these approaches literally provide static and dynamic typing in the same program, they also do require adding some optional type annotations to enable more precise inference. As a result, performing type inference for dynamic languages without having the programmer provide at a least a subset of the type annotations is still an open challenge.

Probabilistic type inference has recently been proposed as an attempt to reduce the burden of writing and maintaining type annotations [? ? ?]. Just as the availability of large data sets has transformed artificial intelligence, in a similar way the increased amount of publicly available source code, through code repositories like GitHub[1] or GitLab[2], enables a new class of applications that leverage statistical patterns in large codebases. For type inference, machine learning allows us to developing less strict type inference systems, that learn to predict types from probabilistic information, such as comments, names, and lexical context, even in cases where traditional type inference procedures fail to infer a useful type. For instance, JSNice [?] uses probabilistic graphical models to statistically infer types of identifiers in programs written in JavaScript, while DeepTyper [?] targets TypeScript [?] via deep learning techniques. These approaches all use machine learning to capture the structural similarities between typed and untyped source code and extract a statistical model for the text. However, none of them takes into account explicitly the underlying type inference rules, and thus their prediction loses important flow information. This is a missed opportunity that we aim to address in this work.

1.1 Our Contribution

Current type inference systems rely on one of two sources of information:

(I) *Logical constraints* on type annotations that follow from the type system. These are the constraints used by standard deterministic approaches for static type inference.

---

[1]https://github.com
[2]https://gitlab.com

(II) *Natural constraints* are statistical constraints on type annotations which can be inferred from relationships between types and surface-level properties such as names, types, and lexical context. These constraints can be learned by applying machine learning to large code bases. These are the constraints that are currently employed by probabilistic typing systems.

Our goal is to improve the accuracy of probabilistic type inference by combining both kinds of constraints into a single analysis, unifying logic and learning. To do this, we define a new probabilistic type inference procedure that combines programming language and machine learning techniques into a single framework. We start with a formula that defines the logical constraints on the types of a set of identifiers in the program, and a machine learning model, such as a deep network, that makes a probabilistic prediction of the type of each identifier.

Our method is based on two key ideas. First, we relax the logical formula into a continuous function by relaxing type environments to probability matrices and defining a continuous semantic interpretation of logical expressions; the relaxed logical constraints are now commensurate with the predicted probability distribution. This allows us to define a continuous function over the continuous version of the type environment that sums the logical and natural constraints. Second, once we have a continuous function, we can optimize it: we set up an optimization problem that returns the most natural type assignment for a program while at the same time respecting the logical constraints. To the best of our knowledge, no prior work has applied machine learning to infer types while simultaneously taking into account logical constraints extracted from a static type analysis.

We investigate the above challenge in a real-world language by building ProdTS, which is a tool that mitigates the effort of generating a TypeScript declaration files for existing JavaScript libraries. TypeScript is a superset of JavaScript that adds static typing to the language. In fact, as JavaScript libraries and frameworks are very popular, many TypeScript applications need to use untyped JavaScript libraries. To support static type checking of such applications the typed APIs of the libraries are expressed as separate TypeScript declaration files *(.d.ts)*. Although this manual approach has been proven effective, it raises the challenge of how to automatically maintain valid declaration files as library implementations evolve.

Our contributions can be summarized as:

- We introduce a principled framework to combine logical and natural constraints for type inference, based on transforming a type inference procedure into a numerical optimization problem.
- As an instantiation of this framework, we implement ProdTS, a tool to generate probabilistic type signatures on TypeScript from JavaScript libraries. ProdTS aims at predicting types for methods that are declared in a TypeScript declaration file.
- We evaluate ProdTS on a corpus of 5,800 JavaScript libraries that have type annotations on the Definitely-Typed repository. We find that combining natural and logical constraints yields improved performance over either alone. Further, ProdTS outperforms previous state-of-the-art research systems, JSNice [?] and Deep-Typer [?]. ProdTS achieves a *50% reduction in error* (measured relatively) over these previous systems.

1.2 Our Framework via an Example

Before formalizing our framework, we pictorially illustrate our approach in ??. Our input is a minimal TypeScript function without any type annotations on its parameters. TypeScript's compiler will by default consider the parameters as the top-level type any. Our goal is by exploiting different sources of information to suggest to the programmer a type lower in the type lattice. To do so we correspond a type parameter for each of the formal parameters and the return type of the function and inject them in the definition of the function, see Fig. 1(a). This step limits the scope of the parameters to this function, but now we can extract more elaborate information about the usage of the parameters in this function. For instance, we can now capture the fact that the parameters are used as terms in a binary operation. This could give us a complex set of logical constraint about the possible combinations of the two operands, however for the purpose of this paper we limit ourselves in a minimal set of constraints shown in Fig. 1(c), where both the two operands should have the same type. In this stage a classical type inference approach would not be able to give us back a user-friendly type and would probably roll back to its top-level type. To alleviate that we are willing to take into account other sources of information, which try to capture the human intuition about the source code itself. As an instantiation of this scenario we use a machine learning model to capture naming conventions over types, see Fig. 1(d). Intuitively, what the table in Fig. 1(d) is showing is that a programmer is much more likely to name a variable start or end if she intents to use it as a number than as a string. This kind of information is given to us not with a binary probability, but as a fraction of probability that describes the confidence we have for this information. To combine the two sources of information we present the logical constraints as a matrix of probabilities vectors shown in Fig. 1(b). Doing that, enable us to relax the boolean operators to a numerical operators as shown in Fig. 1(e). When we numerically optimize the resulting expression we obtain that both variables are strings with high probability. At the end we setup and solve an optimization problem that combines both sources of information into one and tries to satisfy the constraints from both sources as much as possible to improve the results. This is shown in Fig. 1(f), where the predicted type is now

Figure 1. An overview of the three type inference procedures via a minimal example.

Figure 2. Overview of general framework that combines logical and natural constraints in a single optimization problem.

number. Finally, we obtain the correctly annotated function signature shown in Fig. 1(g).

2 General Framework for Probabilistic Type Inference

In this section we introduce our general framework, shown in ??, which we instantiate in the next section by building a tool for predicting types in TypeScript. Our goal is to enhance the type inference procedure for dynamic languages by incorporating into a single engine both information learned from a corpus of typed code as well as information derived directly from the code that is to be typed. We distinguish between two main kinds of constraints that we eventually combine in an optimization problem. The next few subsections formalize this approach.

2.1 An Outline of Probabilistic Type Inference

We consider a dynamic language of untyped programs that is equipped with an existing deterministic type system, that requires type annotations on identifiers. Given a program U plus a type environment Γ let $\Gamma \vdash U$ mean that the program U is well-typed according to the (deterministic) type system, given types for identifiers provided by Γ. The environment takes the form $\Gamma = \{x_v : t_v \mid v \in 1 \ldots V\}$ where each x is an identifier and each t is a literal type. Given an untyped program U, let *probabilistic type inference* consist of these steps:

1. We choose a finite universe consisting of T distinct concrete types.
2. We compute a set $\{x_v \mid v \in 1 \ldots V\}$ of a number V of distinct identifiers in U that need to be assigned types.
3. We extract a set of constraints from U.
4. By optimizing these constraints, we construct a matrix M with V rows and T columns, such that each row is a probability vector (over the T concrete types).
5. For each identifier x_v, we set type t_v to the concrete type l_τ where column τ has the maximum probability in the vth probability vector (the one for identifier x_v).
6. The outcome is the environment $\Gamma = \{x_v : t_v \mid v \in 1 \ldots V\}$.

We say that probabilistic type inference is *successful* if $\Gamma \vdash U$, that is, the untyped program U is well-typed according to the deterministic type system. Since several steps may involve approximation, the prediction Γ may only be partially correct. Still, given a known $\hat{\Gamma}$ such that $\hat{\Gamma} \vdash U$ we can measure how well Γ has predicted the identifiers and types of $\hat{\Gamma}$. A key idea is that there are two sorts of constraints generated in step (??): logical constraints and natural constraints.

A logical constraint is a formula E that describes necessary conditions for U to be well-typed. In principle, E can be any formula such that if $\Gamma \vdash U$, then Γ satisfies E. Thus, the logical constraints do not need to uniquely determine Γ. For this reason, the natural constraints encode less-certain information about Γ, for example, based on comments or names. Just as we can conceptualize the logical constraints as a function to $\{0, 1\}$, we can conceptualize the natural constraints as functions that map Γ to $[0, 1]$, which can be interpreted as a prediction of the probability that Γ would be successful. To combine these two constraints, we relax the boolean operations to continuous operators on $[0, 1]$. Since we can conceptualize E as a function that maps Γ to a boolean value $\{0, 1\}$, we relax this function to map to $[0, 1]$, using a continuous interpretation of the semantics of E. Similarly, we relax Γ to a $V \times T$ matrix of probabilities. Having done this, we formalize the type inference problem as a problem in numerical optimization, in which the goal to find a relaxed type assignment that satisfies as much as possible both sort of constraints. The result of this optimization procedure is the M matrix of probabilities described in step (??). We explain the above formalization in more detail in the remainder of this section.

2.2 Logical Constraints in Continuous Space

The first source of information concerns classical and deterministic sources of information about types, which we abstract as logical relationships between type parameters. A *logical constraint* is the kind of constraint that arises from classical type inference rules and consist of logical formulas about the type assignment. The logical constraints restrict the space of valid type annotations. However, especially for untyped programs, the resulting space might be large, Therefore, instead of solving the problem with classical approaches, like a SAT solver, we interpret the boolean type expressions as numerical expressions in a continuous space. This interpretation enables us to mix together the logical constraints with information coming from statistical analysis in a constructive way and hence to narrow down the predicted type. Logical constraints can be extracted from U using standard program analysis techniques. We employ an augmented static program analysis (*Aug-Static*) that takes into account a set of rules that the type system enforces and generates a corresponding boolean expression for them. We will refer to this mechanism as the *Constraints Generator*, see ??.

In this work, we consider the following simple grammar of logical constraints:

Definition 2.1 (*Grammar of Logical Constraints*). A logical constraint is an expression of the form

$$E ::= x_v \text{ is } l_\tau$$
$$| \ not \ E \qquad\qquad (1)$$
$$| \ E \ and \ E$$
$$| \ E \ or \ E$$

We use $\mathcal{E}$ to denote the set of all syntactically valid logical constraints.

Continuous Relaxation We explain how to specify a *continuous relaxation* of the discrete logical semantics. A formula E can be viewed as a boolean function $f_E : \{0,1\}^{V \times T} \to \{0,1\}$ that maps binary matrices to $\{0,1\}$. To see this, we can convert an environment Γ into a $V \times T$ binary matrix M by setting $m_{v\tau} = 1$ if $(x_v, l_\tau) \in \Gamma$, and 0 otherwise. Let $M(\Gamma)$ be the binary matrix corresponding to Γ. Also, define $\Pi^{V \times T}$ to be the set of all probability matrices of size $V \times T$, that is matrices of the form $P = \begin{bmatrix} \boldsymbol{p}_1 & \dots & \boldsymbol{p}_V \end{bmatrix}^\top$, where each row $\boldsymbol{p}_v = \begin{bmatrix} p_{v,1} & \dots & p_{v,T} \end{bmatrix}^\top$ is a vector that defines probability distribution over concrete types. Finally, a *relaxed semantics* is a continuous function that always agrees with the logical semantics, that is, a relaxed semantics is a function $\tilde{f}_E : \Pi^{V \times T} \to [0,1]$ such that all formulas E and environments Γ, $\tilde{f}_E(M(\Gamma)) = f_E(M(\Gamma))$.

To define a relaxed semantics, we introduce a continuous semantics of E based on generalizations of two-valued logical conjuctions to many-valued [?]. In specific, we use the product t-norm, because the binary operation associated with it is smooth and fits with our optimization-based approach. Product t-norm has already been used for obtaining continuous semantics in machine learning, for example by [?].

The continuous semantics $\llbracket E \rrbracket_P$ is a function $\Pi^{V \times T} \times \mathcal{E} \to [0,1]$, defined as

$$\llbracket x_v \text{ is } l_\tau \rrbracket_P = p_{v,\tau}$$
$$\llbracket not \ E \rrbracket_P = 1 - \llbracket E \rrbracket_P \qquad (2)$$
$$\llbracket E_1 \ and \ E_2 \rrbracket_P = \llbracket E_1 \rrbracket_P \cdot \llbracket E_2 \rrbracket_P$$
$$\llbracket E_1 \ or \ E_2 \rrbracket_P = \llbracket E_1 \rrbracket_P + \llbracket E_2 \rrbracket_P - \llbracket E_1 \rrbracket_P \cdot \llbracket E_2 \rrbracket_P$$

We note that in the actual implementation we use logits instead of probabilities for numerical stability, see **??**.

To motivate this continuous semantics, recall that in our setting, we know E but do not know P. We argue that the continuous semantics, when considered as a function of P, can serve as a sensible objective for an optimization problem to infer P. This is because it relaxes the deterministic logical semantics of E, and it is maximized by probability matrices P which correspond to satisfying type environments. This is stated more formally in the following theorem.

Theorem 2.1. *For any E, if $P = M(\Gamma)$ for some Γ that satisfies E, then $P \in \arg\max_{P \in \Pi^{V \times T}} \llbracket E \rrbracket_P$.*

To sketch the proof, two facts can be seen immediately. First, for any formula E, the function $\tilde{f}(P) = \llbracket E \rrbracket_P$ is a relaxation of the true logical semantics. That is, for any environment Γ, we have that $\tilde{f}(M(\Gamma)) = \llbracket E \rrbracket_{M(\Gamma)} = 1$ if and only if Γ satisfies E. This can be shown by induction. Second, for any matrix $P \in \Pi^{V \times T}$, we have the bound $\tilde{f}(P) \leq 1$. Putting these two facts together immediately yields the theorem.

2.3 Natural Constraints via Machine Learning

A complementary source of information about types arises from statistical dependencies in the source code of the program. For example, names of variables provide information about their types [?], natural language in method-level comments provide information about function types [?], and lexically nearby tokens provide information about a variable's type [?]. This information is indirect, and extremely difficult to formalize, but we can still hope to exploit it by applying machine learning to large scale corpora of source code.

Recently, the software engineering community has adopted the term *naturalness of source code* to refer to the concept that programs have statistical regularities because they are written by humans to be understood by humans [?]. Following the idea that the naturalness in source code may be in part responsible for the effectiveness of this information, we refer generically to indirect, statistical constraints about types as *natural constraints*. Because natural constraints are uncertain, they are naturally formalized as probabilities. A *natural constraint* is a mapping from a type variable to a vector of probabilities over possible types.

Definition 2.2 (*Natural Constraints*). For each identifier x_v in a program U, a *natural constraint* is a probability vector $\boldsymbol{\mu}_v = [\mu_{v1}, \dots, \mu_{vT}]^\top$. Correspondingly, we aggregate the probability vectors of the learning model in a matrix defined as $\mathcal{M} = \begin{bmatrix} \boldsymbol{\mu}_1 & \dots & \boldsymbol{\mu}_V \end{bmatrix}^\top$.

In principle, naturalness constraints could be defined based on any property of U, including names and comments. In this paper, we consider a simple but practically effective example of naturalness constraint, namely, a deep network that predicts the type of a variable from the characters in its name. For the the type variable x_v, denote the name of the associated identifier as $w_v = (c_{v1} \dots c_{vN})$, where each c_{vi} is a character. (This instantiation of the naturalness constraint is defined only for those each type variable x_v corresponds to an identifier in the source code, such as a function identifier or a parameter identifier.) This is a classification problem, where the input is w_v, and the output classes are the set of T concrete types. Ideally, the classifier would learn that identifier names that are lexically similar tend to have similar types, and specifically which subsequences of the character

names, like lst, are highly predictive of the type, and which subsequences are less predictive. One simple way to do this is to use a recurrent neural network (RNN).

For our purposes, an RNN is simply a function $(\boldsymbol{h}_{i-1}, z_i) \mapsto \boldsymbol{h}_i$ that maps a so-called state vector $\boldsymbol{h}_{i-1} \in \mathbb{R}^H$ and an arbitrary input z_i to an updated state vector $\boldsymbol{h}_i \in \mathbb{R}^H$. (The dimension H is one of the hyperparameters of the model, which can be tuned to obtain the best performance.) The RNN has continuous parameters that are learned to fit a given data set, but we elide these parameters to lighten the notation, because they are trained in a standard way. We use a particular variant of an RNN called a long-short term memory network (LSTM) [?], which has proven to be particularly effective both for natural language and for source code [? ? ? ?]. We write the LSTM as LSTM$(\boldsymbol{h}_{i-1}, z_i)$.

With this background, we can describe the specific natural constraint that we use. Given the name $w_v = (c_{v1} \ldots c_{vN})$, we input each character c_{vi} to the LSTM, obtaining a final state vector $\boldsymbol{h}_N$, which is then passed as input to a small neural network that outputs the naturalness constraint $\boldsymbol{\mu}_v$. That is, we define

$$\boldsymbol{h}_i = \text{LSTM}(\boldsymbol{h}_{i-1}, c_{vi}) \qquad i \in 1, \ldots, N \qquad (3a)$$
$$\boldsymbol{\mu}_v = F(\boldsymbol{h}_N), \qquad\qquad\qquad\qquad (3b)$$

where $F : \mathbb{R}^H \to \mathbb{R}^T$ is a simple neural network. In our instantiation of this natural constraint, we choose F to be a feedforward neural network with no additional hidden layers, as defined in (??). We provide more details regarding the particular structure of our neural network in ??.

This network structure is by now a fairly standard architectural motif in deep learning. More sophisticated networks could certainly be employed, which are left to future work.

2.4 Combining Logical and Natural Constraints to an Optimization Problem

The logical constraints pose challenges to the probabilistic world of machine learning. It is not straightforward to incorporate them along with the logical rules that they should follow to a probabilistic model. To combine the logical and the natural constraints we define a continuous optimization problem.

Intuitively, we design the optimization problem to be over probability matrices $P \in \Pi^{V \times T}$; we wish to find P that is as close as possible to the natural constraints M, subject to the logical constraints being satisfied. A simple way to quantify distance via the *euclidean norm* $||\cdot||_2$, i.e., the square root of the sum of the squares of its elements. This leads to the

optimization problem

$$\min_{P \in \mathbb{R}^{V \times T}} \sum_v ||\boldsymbol{p}_v - \boldsymbol{\mu}_v||_2^2$$
$$\text{subject to } p_{v\tau} \in [0, 1] \qquad \forall v, \tau \qquad (4)$$
$$\sum_{\tau=1}^{T} p_{v\tau} = 1 \qquad \forall v$$
$$[\![E]\!]_P = 1,$$

This is a constrained optimization problem. Although there is an extensive literature on constrained optimization, it is often the case that the most effective way to solve a constrained optimization problem is to transform it into an equivalent unconstrained one. This can be done in two steps. First we reparameterize the problem to remove the probability constraints. The softmax function (??) maps real-valued vectors to probability vectors. Thus we define

$$\min_{Y \in \mathbb{R}^{V \times T}} \sum_v ||\sigma(\boldsymbol{y}_v)^\mathsf{T} - \boldsymbol{\mu}_v||_2^2 \qquad (5)$$
$$\text{subject to } [\![E]\!]_{[\sigma(\boldsymbol{y}_1), \ldots, \sigma(\boldsymbol{y}_V)]^\mathsf{T}} = 1.$$

It is easy to see that if Y minimizes (??), then $P = [\sigma(\boldsymbol{y}_1), \ldots, \sigma(\boldsymbol{y}_V)]^\mathsf{T}$ minimizes (??). To remove the final constraint, we introduce a Lagrange multiplier $\lambda > 0$ to weight the two terms, yielding our final optimization problem

$$\min_{Y \in \mathbb{R}^{V \times T}} \sum_v ||\sigma(\boldsymbol{y}_v)^\mathsf{T} - \boldsymbol{\mu}_v||_2^2 - \lambda[\![E]\!]_{[\sigma(\boldsymbol{y}_1), \ldots, \sigma(\boldsymbol{y}_V)]^\mathsf{T}}. \qquad (6)$$

This can now be solved numerically using standard optimization techniques, such as gradient descent. The parameter λ trades off the importance of the two different kinds of constraints. In the limiting case where $\lambda \to \infty$, the second term in the objective function (??) is dominant and we obtain the solution that best satisfies the relaxed logical constraints. If these constraints are consistent then the obtained probability vectors correspond to one-hot vectors. Similarly, for $\lambda \to 0$ the first term dominates and we obtain the solution that best matches the natural constraints, which is naturally M itself. By choosing λ well, we can trace the Pareto frontier between the two types of constraints, and identify a value that minimizes the original problem (??).

To obtain a final hard assignment Γ, we first solve (??) to obtain the optimal Y, compute the associated probability vector $P = [\sigma(\boldsymbol{y}_1), \ldots, \sigma(\boldsymbol{y}_V)]^\mathsf{T}$. Then, for each identifier x_v, we select the element of the corresponding probability vector which is closest to one.

3 PRODTS: Predict TypeScript Type Signatures for JavaScript Libraries

To evaluate our type inference approach in a real-word scenario we implement an end-to-end application, called PRODTS, which aims to alleviate the process of inferring a TypeScript declaration file for an underlying JavaScript library.

3.1 Introduction: TypeScript's Type System

Syntactically, TypeScript [?] is a typed superset of JavaScript that allows us to develop large-scale, stable applications. To leverage the fact that JavaScript is the only true cross-platform language that runs in any browser, any host, and any OS the TypeScript's compiler typechecks TypeScript programs and eventually emits plain JavaScript. To compromise between static and dynamic typing, TypeScript supports a structural type system. In structural type systems record types (classes) whose fields or members have the same names and types are considered to be equal. This way TypeScript's static type system can work well with a lot of JavaScript code that is dynamically typed. The aforementioned implementation decisions reveal that one of the main intentions of TypeScript's designers is to provide a smooth transition from JavaScript. As a result TypeScript's type system is deliberately unsound [?]. TypeScript uses the any type as an intermediate step in cases it needs to statically assign a type to variables whose type is determined at runtime or is otherwise unknown at compile time. In fact, TypeScript applications and libraries commonly take advantage of JavaScript's flourishing ecosystem and use untyped JavaScript libraries. To support static type checking of such applications, the types of such JavaScript libraries' APIs are expressed as separate TypeScript *declaration files (.d.ts)*. The TypeScript community has already made a huge effort to support this process by manually writing and maintaining declaration files for over five thousand of the most popular JavaScript libraries. These files are available on the DefinitelyTyped [?] repository. Although this approach has proven useful, it raises the challenge of keeping declaration files in sync with the library implementations.

Ideally we would like to automatically infer the typed APIs of such libraries. TypeScript supports a peculiar flavor of type inference: TypeScript's soft type system [?] defaults to the any type. Consequently, in the case where a parameter does not have a type annotation, the type system assumes that it is of type any and does not try to infer it further. The type system is allowed to infer more precisely, lower on the type lattice, the return type of a function, for which the scope is clear. To make things more complex, it is impractical for TypeScript code to implement run-time casts—typical for gradual type systems—due to necessary type erasure to transform back to plain JavaScript [?].

Therefore traditional static or dynamic analyses are not adequate to generate ready-to-use definition files. Towards this direction DefinitelyTyped community officially suggests a tool called dts-gen [?]. The dts-gen tool uses runtime information to produce a .d.ts file that gives a clear shape of the input API but does not provide type information for function arguments and return types. As suggested in the instructions, it is only meant to be used as a starting point for writing a high-quality declaration file. Dts-gen only collects dynamic

```
1    exports.byteLength = byteLength
2    ...
3
4    function getLens (b64) {
5    ... return validLen, placeHoldersLen]
6    }
7
8    function byteLength (b64) {
9        var lens = getLens(b64)
10       var validLen = lens[0]
11       var placeHoldersLen = lens[1]
12       return ((validLen + placeHoldersLen) * 3 / 4)
13            - placeHoldersLen
14   }
15   ...
16
```

```
export function byteLength(b64: any): any;
```

```
export function byteLength(b64: string): number;
```

Figure 3. Center: Example of untyped JavaScript code. This an excerpt from the library `base64-js`. Left: Declaration file after the *Structure* stage, Right: Declaration file after the *Type* stage.

information and as a result it is bound to produce a file full of anys that the developer has to fill by hand. ?] created the TSINFER and TSEVOLVE tools with the same goal that is assist programmers to create and maintain TypeScript declaration files. The tools are designed to exploit information from a static analysis performed in a recorded snapshot of a concretely initialized library. The results, as before, are quite good—although less user-friendly— at capturing the structure of the file but often do not predict readable types.

3.2 Problem Statement

Specifically, we consider the specific problem of predicting a TypeScript declaration file for an underlying JavaScript library.

Input: Our implementation takes as input three files:

1. A JavaScript library file.
2. A declaration file containing the exported functions with every type annotated to any, which we call as *Structure* stage. This file comes from a third party tool and thus why we use as an input in the formal problem statement, in the actual implementation this stage is completely integrated to the process.

3. The TypeScript's default library declaration file [3], that is the built-in types defined TypeScript's standard library.

Output: A corresponding TypeScript declaration file for the JavaScript library, that we from now on we call *predicted.d.ts* and includes type predictions for

- the return type of the function, denoted as *fnRet*, and
- type annotations for the function's arguments, denoted as *param*.

The output of our tool is essentially the output of the *Structure* stage with some, or ideally all, of the any types substituted with built-in types—that is non user-defined types—and sum/product types of those.

We will refer to the process of filling up the typing holes from the previous stage with the correct typing information as *Type* stage. In other words, our main focus on this work is to apply our probabilistic type inference of the types in the *Structure* stage to meliorate the *Type* process. For detailed metrics and evaluation see ??.

3.3 Logical Constraints for TypeScript

To generate the logical constraints of ??, we exploit the information that the TypeScript compiler provides [?] in conjunction with a refining static analysis. We include all the return types that the compiler is able to infer given a JavaScript function. The compiler does not try to infer the parameter types, as mentioned before, but rather emits some flow information it gets from the static analysis of the code. To overcome this obstacle and prompt the compiler to reveal us more, we transform each function's definition by explicitly assigning each formal parameter to a new type parameter. Unconstrained type parameters default to the bottom of the type lattice. We consider a type constraint, according to ??, to be a disjunction between one or more types. For example a valid constraint on type *l* would be equivalent to $l = String$ or $l = \text{Array<any>}$[4] but not *l has member .length*. We initially use unconstrained type parameters and parse the resulting errors to introduce constraints.

We can identify 5 common type errors (??) that the `tsc` compiler emits, and turn those into type constraints. The main problem with using this approach in isolation is that we produce an average total of only around 10 constraints per library. As this information is quite limited and incomplete, our work focuses on utilizing machine learning to augment the quality and amount of type constraints.

---

[3]https://github.com/Microsoft/TypeScript/blob/master/lib/lib.d.ts

[4]There is a subtle difference between JavaScript primitive types, written in lowercase, and their corresponding TypeScript primitive types, which start with a capital letter, and being impleemted as interfaces TypeScript's default declaration file. For this project we consider them to be the same type.

```
1   function toByteArray<B64, TOBYTEARRAY>(b64: B64): TOBYTEARRAY {
2       ...
3       var len = b64.length; // B64<:.length
4       ...
5       tmp = (revLookup[b64.charCodeAt(i)] << 18) // B64<:.charCodeAt()
6   }
```

Figure 4. TypeScript code snippet of base64-js library augmented with type parameters for each of the formal variables and the return type, along with two constraint on lines 3 and 5.

The parsing of the first type error—constraint kind pair of ?? refers to identifying properties/methods that a type parameter should implement and using them to generate a type constraint. *lib.d.ts* includes interfaces for the built-in types of the language. These interfaces contain the typed signatures of each property and method that a built-in type implements. We use these interfaces to construct a set of possible built-in types we could assign to the type parameter and therefore a proper type constraint. For a concrete example of this procedure, see ????.

3.4 Natural Constraints for TypeScript

We now focus our attention on extracting natural constraints for this particular problem. The TypeScript community has already made a huge effort to support this process by writing and maintaining the declaration files for over four thousand of the most popular JavaScript libraries. These files are available on the [?] repository. The declaration files on this repository provide an excellent opportunity for statistical learning algorithms. We choose to use a Char-Level LSTM trained on *(id, type)* pairs to obtain knowledge about naming conventions for identifiers. The main intuition behind the particular choice of the LSTM is that developers commonly use multiple abbreviations of the same word. Thus, this family of abbreviations share type and a Char-Level LSTM is well-suited to predict the type for every identifier in the family.

Our universe of types consists of 78 built-in types or union of them appeared at least 10000 times in our training set. We choose to handle only built-in types to ensure that we do not introduce types that are not available to the compiler.

Regarding the implementation details of the LSTM network, for the *F* in (??), we use a feedforward neural network

$$F(\boldsymbol{h}) = \log\left(\sigma\left(\boldsymbol{h}A^T + b\right)\right), \tag{7}$$

where the log function is applied componentwise, *A* and *b* are learnable weights and bias. The softmax function is

```
LSTMClassifier(
(embedding): Embedding(90, 128)
(lstm): LSTM(128, 64)
(hidden2out): Linear(in_features=64, out_features=78, bias=True)
(softmax): LogSoftmax()
(optimization fun): ADAM)
```

Figure 5. Pipeline of learning naming conventions with a Char-Level *LSTM*, represented by a probability vector for each identifier, including function and parameter identifiers.

defined as

$$\sigma(\boldsymbol{x}) = \left[\frac{\exp\{x_1\}}{\sum_i \exp\{x_i\}}, \frac{\exp\{x_2\}}{\sum_i \exp\{x_i\}}, \cdots \right]^\top. \quad (8)$$

The softmax function corresponds to the last layer of our neural network and essentially maps the values of the previous layer to $[0, 1]$, while the sum of all values is 1; as expected by a probability vector. The log function is applied for numerical stability since computing directly (**??**) can be problematic. As a result, F outputs values in $[-\infty, 0]$.

The model is trained based on sequences of variable names of known type. This is done by minimizing a loss function. Our particular choice of loss function is the negative log likelihood function—conveniently combined with our log output—defined as

$$L(\boldsymbol{y}) = - \sum_i log(\boldsymbol{y}_i). \quad (9)$$

Essentially, we select during training the element that corresponds to the correct label from the output F and sum all the values of the correct labels for the entire test set.

Regarding our optimization algorithm we use ADAM, which is an extension of stochastic gradient descent [**?**]. The main difference between ADAM and classical stochastic gradient descent is the use of adaptive instead of fixed learning rates. Although there exist other algorithms with adaptive learning rates like ADAGRAD [**?**] and RMSPROP [**?**], ADAM tends to have better convergence [**?**].

We have used all the available projects $5,800$ from the DefinitelyTyped repository on a particular date (17th February 2019). We have trained our model for $1,000$ epochs, for 78 different types, and obtained a validation accuracy of *validation acc:* 0.79. This information is also summarized in **??**. Our dataset was randomly split by project into 80% training data, 10% validation data and 10% test data. To split the available data in these three different sets is a common practice in machine learning which ensures that the learning model doesn't simply memorize the training data but is able to generalize to unseen inputs.

3.5 Combining Logical and Natural Constraints for TypeScript

We now describe the particular implementations details used to combine the two sources of information. Most of our design choices here are empirical based on what we have noticed maximizes the performance of the framework.

We implement the optimization problem described in (**??**) for initial $\lambda = 10$. The adaptive optimization algorithm updates its value in subsequent iterations. To evaluate the minimum of the generated functions we use a different adaptive optimization algorithm, that is RMSPROP [**?**]. We set the maximum number of iterations to $2,000$ where we have noticed that the loss stabilizes.

3.6 Experimental Setup

Both the code for the deep learning and the optimization part is written in PyTorch [**?**]. All experiments are conducted on an NVIDIA Titan Xp with 12GB VRam, in combination with an 2-core Intel Core i5 CPU with 8GB of RAM. Our resulting model requires ca. 400MB of RAM to be loaded into memory and can be run on both a GPU and CPU. It can provide type annotations on average for 58 files in approximately 60 sec for solving logical constraints and natural constraints and about 65 sec for the combined optimization.

4 Evaluation of PRODTS

To measure the performance of our tool PRODTS we make the following assumption: The existing declaration files on the DefinitelyTyped repository define the *gold standard* for our predictions. This is an assumption in the sense that some files may contain errors [**?**]. Based on it, we measure the performance of our tool as well as other related tools by comparing the output declaration file for a given input JavaScript library with the corresponding gold declaration file included in the DefinitelyTyped repository.

Next, we define the metrics that we use to perform this comparison. Traditionally, there have been two measures when comparing a result set (here the output declaration file) with human judgement (the "gold" declaration file): *Precision* and *Recall* [**?**]. We use these two metrics to evaluate the output structure and types for each of the exported functions in a library. Here, we focus solely on functions while we discard the other exported entities, such as variables, methods, and properties; the metrics presented next can be extended to evaluate all exported entities straightforwardly. The following definitions formalize these metrics.

4.1 Precision and Recall for TypeScript Declaration Files

Definition 4.1 (*Paths*). Let a *path* be either a *structural path*, or a *type path*. A *structural path* is a fully-qualified name *n*. A *type path* is a pair *(n,ty)* of a fully-qualified name *n* and a type *ty*. Let the variable X range over sets of paths.

Let S be the set of all structural paths, which is partitioned into the following subsets:

- function identifiers S_{fnRet}
- function parameters S_{param}
- both function and parameters identifiers $S_{\text{total}} = S_{\text{fnRet}} \cup S_{\text{param}}$

Let T be the set of all type paths, which is partitioned into the following subsets:

- type returned by each function T_{fnRet}
- type of each function parameter T_{param}
- type for both function's return and parameter $T_{\text{total}} = T_{\text{fnRet}} \cup T_{\text{param}}$

We call the predicted TypeScript declaration file *predicted.d.ts* while for the gold standard file we use the term *gold.d.ts*.

Definition 4.2 (Functions *Paths* & *Filter$_X$*). To capture the contents of a file we define the function *Paths(*.d.ts)*, which takes as an input a TypeScript declaration file and returns a set of paths to represent the structure and types of the file. In the next step we apply a *Filter$_X$* function to filter the output of the *Paths* function to keep only the paths in the set X, that is, *Filter$_X$(Y) = X ∩ Y*.

As we are interested in evaluating both *Structure* and *Type*—the core elements of our pipeline as analyzed in ??— and not only the final output, we measure the precision and recall for both stages independently.

Precision for *Structure* measures the proportion of entities found in *predicted.d.ts* that are also included in *gold.d.ts*. For example, if we get an 80% *Structure* precision that means that on average out of the 10 entities that we found, 8 exist in *gold.d.ts* while 2 do not. Recall for *Structure* measures the proportion of entities that exist in *gold.d.ts* that we found in *predicted.d.ts*. For example, if we compute a 70% *Structure* recall that means that on average out of 10 entities in *gold.d.ts* we successfully identified 7 of them while 3 were not recognized.

For measuring *Type* we exclude all entities that we found in *predicted.d.ts* but not found in *gold.d.ts* since we do not have a means to evaluate their validity. Precision for *Type* measures the proportion of correct types found in *predicted.d.ts* with respect to the types defined in *gold.d.ts*. For example, we interpret a 90% *Type* precision as finding on average 9 correct types out of the 10 predicted. Recall for *Type* measures the percentage of correct types found in *gold.d.ts* with respect to the total number of types defined in *gold.d.ts*. For example, we interpret a 60% *Type* recall as correctly predicting on average 6 types out of the 10 found in *gold.d.ts*. We compute the precision and recall for both stages using the formulae defined next. To do so, we define X which symbolises the stage for which we perform measurements; here it belongs to a binary set $X \in \{S, T\}$ for *Structure* and *Type*, respectively. In general, X is not restricted to the binary set mentioned

before. For example, we could perform a finer analysis if $X = S_{\text{fnRet}}$ for function identifiers.

Definition 4.3 (Precision & Recall for X). Given a declaration file *predicted.d.ts*, an ideal declaration file *gold.d.ts*, and a class of paths X, we define precision and recall for X as:

$$P(X) = \frac{|Filter_X(Paths(predicted.d.ts)) \cap Filter_X(Paths(gold.d.ts))|}{|Filter_X(Paths(predicted.d.ts))|}, \tag{10}$$

$$R(X) = \frac{|Filter_X(Paths(predicted.d.ts)) \cap Filter_X(Paths(gold.d.ts))|}{|Filter_X(Paths(gold.d.ts))|}. \tag{11}$$

The maximum precision and recall that we can obtain is equal to 1. For *Structure*, the larger the precision the more relevant entities are returned than irrelevant ones, while the larger the recall the more existing entities are discovered. Similarly, for *Type*, the larger the precision the more precise are the identified types, while the larger the recall the more existing types are discovered.

Our contributions presented in ???? concern the second stage of the workflow, that is the type prediction phase. Here we describe an integrated, end-to-end approach so we focus on presenting a holistic evaluation of the problem. Thus, we will present results from the first stage too, although our method does not contribute to the improvement of it. For the same reason, we start by presenting the results related to the *Type* stage.

4.2 Evaluation of *Type* Stage

For evaluating the predicted *Types*, we perform a comparison of the augmented static analysis, and the Char-Level LSTM, the output of the optimiser. Two kinds of input might be required from any of these packages: the JavaScript library file or a declaration file containing the exported functions. We summarise below the characteristics of each package and the provided input.

- *Aug-Static*: Using the JavaScript library and the declaration file as an input, we utilise the compiler in a pragmatic way allowing us to generate logical constraints as logical formulas and then solve them using fuzzy logic (??).
- *LSTM*: We query our pre-trained Char-Level *LSTM* to give us predictions for every id found in the declaration file (??). It requires as input the declaration file only.
- PRODTS: The output of the optimization problem for the corresponding combination of the Character Level and fuzzy tool (??). Since this tool depends on Aug-Static and Char-Level, we provide both the JavaScript library and the declaration file as input.

Furthermore, we report separately in ?? the precision and recall for predicting the function return types (first and second column) and the types of the functions' parameters (third and fourth column).

Table 1. Aggregate *Type* precision and recall for 58 JavaScript libraries with 1272 identifiers in total (610 *funRet*, 662 *param*). We use the notation $P(T_{\mathrm{fnRet}}^N)$ for $P(T_{\mathrm{fnRet}} \cap N)$, where N is the number of types in our universe of types.

Since we consider 78 types in total and the *gold.d.ts* contains a larger number, we first compute the maximum achievable precision and recall even if all our type predictions were correct. The precision for the function return types is 0.61 and the recall 0.59, while the precision for the parameters' types is 0.63 and the recall 0.58. The combined precision and recall are 0.62 and 0.58, respectively. The results presented in **??** are normalized based on these upper precision and recall limits.

Regarding the augmented static analysis, the results are significantly better for function return types rather than parameters' types. This happens because the TypeScript compiler generates much richer constraints for all—exported or not—return types of functions. For the parameters' types, the compiler disregards any information and infers all types as any; a situation that produces no useful logical constraints. To mitigate this issue, we define some simple heuristics that allow the augmented static analysis to generate some non-trivial constraints. A more principled approach could greatly improve the results regarding the parameters' type inference, albeit outside the scope of this work.

Furthermore, we observe for all tools that the precision and recall for both task are close. By comparing these two metric in **??**, their similarity can be traced to similarity of their denominators. As it will be analyzed in **??**, the *Structure* stage is able to identify the majority (over 80%) of the function signatures; this gives rise to the closeness of the denominators and subsequently of the precision and recall.

Finally, for PRODTS we notice that the combination of the output of the augmented static analysis and the Char-Level LSTM improves our precision and recall for both tasks. A certain level of robustness can be also identified, because the low results of the augmented static analysis for parameters' types positively contributes to the optimizer results. This means that on top of the predictions of the neural network, some constraints of the augmented static analysis tip the balance towards more reasonable predictions. *Overall, the combination of the logical (Aug-Static) and natural (LSTM) constraints in PRODTS greatly improves our type inference capabilities.*

4.3 Comparison of *Type* Stage with Existing Tools

To evaluate PRODTS we compare against two state-of-the-art tools that utilize machine learning techniques, both aiming to give type suggestions to the programmer. None of these tools was made explicitly for predicting TypeScript declaration files, so we made the necessary adjustments to meet the

Table 2. Aggregate *Type* precision and recall across all evaluated modules for DeepTyper and JSNice; as input we use 41 JavaScript libraries with 860 identifiers in total (270 funRet, 590 param). The superscript N has the same meaning as in **??**, where $N = 6$ consists of the six types predicted by JSNice. Boldface indicates the best results of the full set of $N = 78$ types.

requirements of our setting. For all of our evaluations we have used the same Structure file, that it the one that dts-gen gives us. The two tools that we compare against are:

- DeepTyper: A tool based on deep learning which learns types for every identifier. The training of the network is performed using already annotated TypeScript code.
- JSNice: A tool based on graphical machine learning which analyzes relationships between program elements to infer types for JavaScript files.

We summarize the results of our comparisons in **??**.

Comparison with DeepTyper DeepTyper is essentially a tool that outputs a type vector for every identifier based on information from the source-code context. That is, it utilizes the information in the vicinity of the identifier to predict its type. This is in stark contrast with our method of handling pairs of identifiers and types; we focus of obtaining information based on the identifier alone and not its context.

For the results shown in **??**, it is usually the case that DeepTyper returns any as the most relevant suggestion for the type of an identifier. In this case, to keep the comparison meaningful, we select the second best candidate; DeepTyper is a deep learning framework and it outputs a probability vector similar to our LSTM network.

A particularly favourable fact of our results is that even though LSTM takes as input identifiers and types only—while DeepTyper utilizes more context—predictions for both tools are comparable for parameters' types. For function return types, the Char-Level LSTM clearly outperforms DeepTyper. This shows that taking into account information in the vicinity can be problematic; function definitions might be placed relatively far away from their calls and as a result the context might be difficult to utilize properly.

Finally, it is worth pointing out that DeepTyper has a type vocabulary of size $N = 11,000$, because it includes user-defined types too; much larger than our vocabulary of size $N = 78$. This suggests that if DeepTyper was trained on a smaller size vocabulary the results—at least for the predicted types of parameters—might be improved. Alternatively, this suggests that taking into account user-defined types needs extra considerations; simply learning user-defined identifier and type pairs might not be adequate and in fact it might even worsen the tool's performance.

Comparison with JSNice JSNice is a tool that learns statistical correlations between program elements by shallowly exploiting their relationship. It was introduced as an approach for (among others) type inference on JavaScript using statistics from dependency graphs learned on a large corpus. The evaluation had to be performed manually because it provides a website interface only.

JSNice has a type lattice that contains only primitive types $N = 6$ and exploits the relationships between types of shallow depth. Therefore, the tool does not consider at all the additional issues involved with predicting precise types, in which the system must infer what level of generality is most appropriate for the predicted types. Moreover, it does not entirely capture the flow and dependencies among typings.

Due to the previous issue, we report results for two instantiations of JSNice: for the first one we calculated the results for the 6 primitive types only, while for the second one we included the results of rest $N = 78$ wrongly identified types. We notice that regarding the type recall of the parameters in **??** for the $N = 6$ case, JSNice scored better results than our tool. This might be the case due to the property that we discussed already regarding the use of a small size vocabularies.

Overall, we believe that this performance drop for DeepTyper and JSNice happens for the following reasons:

(i) The identifiers for functions are not as repeatable as the identifiers for parameters; DeepTyper and JSNice have less training data for function identifiers, while our Char-Level LSTM can captures variations of the same identifier irrespectively of its source.

(ii) The return type of a function is usually related to code at the bottom of the function's body. As a result, a learning approach which takes into account a limited amount of context can miss relationships that are not spatially close.

(iii) JSNice only captures shallow dependencies between identifiers and thus ther are occasionally not enough to capture the information flow, especially for the function's return types.

NL2Type a tool by [**?**] also uses a deep learning approach to the problem which considers JSDoc comments as an additional type hint, we could not compare directly to them as we do not have many example with both JSDoc and available declaration files in the DefinitelyTyped repository. Finally, it is worth pointing out that it would be fairly simple to extend the method to handle constraints generated by DeepTyper, NL2Type or in fact any other deep learning approach that give similar kind of information. For example, we could simply add more terms to the combined objective function, including an extra term for every source of natural constraint that we would like to add.

Table 3. Aggregate *Structure* precision and recall across all modules. Our type universe consists of 78 types, while we use 48 JavaScript libraries as input with 2012 identifiers in total.

4.4 Comparison for *Structure*

We now turn our attention on the first stage that predicts the structure of our JavaScript library. Here, we compare two tools that can output a structure file: *declFlag* and *dts-gen*. We summarise each tool as follows:

- *declFlag*: The TypeScript compiler, when called with the flag `--declaration`, generates statically some of the exported definitions.
- *dts-gen*: A tool that attempts to address this problem by examining JavaScript objects that appear during runtime, rather than analysing the code statically.

The result of their comparison is shown in **??**. It is clear that *dts-gen* outperforms *declFlag* across all tasks. As a result, we used the output of *dts-gen* as input in our type prediction stage.

5 Related Work

5.1 Classical Type Inference

Rich type inference mitigates the cost of explicitly annotating types. This feature is an inherent trait of strongly, statically-typed, functional languages (like Haskell or ML). In this direction, some procedural languages attempt to include type inference as a feature. For instance, in C++ programmers can use the auto keyword to avoid writing the type in the definition of a variable with an explicit initialization, while in C# (starting with version 3) the var keyword can be used as a convenient syntactic sugar for shorter local variable declarations. Nevertheless, C# is still a statically typed language. These enhancements are implemented via compiler tricks and thus are considered as a small step towards a world of static typing where possible, and dynamic typing when needed.

Dynamic languages have also started to pay more attention to typings. Several JavaScript extensions, like Closure Compiler [**?**] and TypeScript, see **??** add optional type annotations to program variables using a gradual type system. In JavaScript, these annotations are provided by specially formatted comments known as JSDoc [**?**]. However, these extensions often fail to scale to realistic programs that make use of dynamic evaluation and complex libraries,for example jQuery, which cannot be analyzed precisely [**?**]. In line with this approach we have seen extensions for other popular scripting languages, like [**?**], which is an optional static type checker for Python, or RuboCop [**?**], which serves as a static

analyzer for Ruby by enforcing many of the guidelines out-lined in the community Ruby Style Guide [?] and performing various check types known as cops.

The quest for more modular and extensible static analysis techniques has resulted in the development of richer type systems. Refinement types, that is subsets of types that satisfy a logical predicate (like Boolean expression), constrain the set of values described by the type, and hence allow the use of modern logic solvers (aka SAT and SMT engines) to dramatically extend the scope of invariants that can be statically verified. An implementation of this concept comes with Logically Qualified Data Types, abbreviated to Liquid Types. DSOLVE is an early application of liquid type inference in OCAML [?]. A type-checking algorithm, which relies on an SMT solver to compute subtyping efficiently for a core, first order functional language enhanced with refinement types [?], provides a different approach. LiquidHaskell [?] is a static verifier of Haskell based on Liquid Types via SMT and predicate abstraction. The brainy and expressive type system of functional languages made the task of adding refinement types easier to achieve. For instance, these languages take as primitive the useful idea of data tagged with data constructors by providing kinds, and algebraic types as built-in notion. Although modern scripting languages have popularised the use of higher-order constructs, attempts to apply refinement typing to scripts have mostly proven to be impractical [?]. However, the case is that as each programming language has developed its own characteristics, all of the above solutions are entailed to a specific language.

5.2 Machine Learning Over Source Code

Although the interdisciplinary field between machine learning and programming languages is quite young, some complete reviews of this area are already available. ?] in their survey give an extensive synopsis of works that model source code in a probabilistic way by containing a learning component and using complex representations of the underlying code. A detailed description of the area is also given by ?] in their related article, whilst ?] in their position paper also examine this research area by categorizing the challenges involved in three main, overlapping pillars. A sub-field of this emerging area applies probabilistic models from machine learning to infer semantic properties of programs, such as types. ?] use control and data flow analyses to extract the desired statistical graphical model. ?] also use probabilistic graphical models to statistically infer types of identifiers in programs written in Python. Their tool trains the classification model for each type in the domain and uses a different approach to build the graphical model as it allows to leverage type hints derived from data flow, attribute accesses, and naming conventions for types.

6 Conclusion

This paper addresses the lack of rich type inference process for dynamically typed languages. To tackle this problem we define a general probabilistic framework that combines information from traditional analyses with statistical reasoning for source code text, and thus enable us to to predict natural occurring types. To evaluate our framework we build PRODTS, a tool to generate typed TypeScript declaration files for untyped JavaScript libraries. Our experiments show that PRODTS predicts functions types signatures with a precision and recall score of almost 70% for the top-most prediction. We believe that the probabilistic type inference approach presented in this work can serve as a basis for exploring ways of constructively combining different type analyses by using numerical methods.

A Appendix

A.1 Continuous Relaxation in the Logit Space

In ??, we present the continuous interpretation based on probabilities. As already mentioned, in the actual implementation we use logit instead for numerical stability. Logit is defined as the inverse of the softmax function; that is, an element of a probability vector $p \in [0, 1]$ corresponds to

$$\pi = \log \frac{p}{1 - p}.$$

It allows us to map probability values from $[0, 1]$ to $[-\infty, \infty]$.

Given the matrix $\mathcal{L}$, which corresponds to the logit of the matrix P in ??, we interpret an expression E as a number $[\![E]\!]_P \in \mathbb{R}$ as follows:

$$[\![x_v \ is \ l_\tau]\!]_{\mathcal{L}} = \pi_{v,\tau}$$
$$[\![not \ E]\!]_{\mathcal{L}} = \log(1 - \text{sigmoid}([\![E]\!]_{\mathcal{L}})$$
$$[\![E_1 \ and \ E_2]\!]_{\mathcal{L}} = [\![E_1]\!]_{\mathcal{L}} + [\![E_2]\!]_{\mathcal{L}}$$
$$[\![E_1 \ or \ E_2]\!]_{\mathcal{L}} = \text{LogSumExp}([\![E_1]\!]_{\mathcal{L}} + [\![E_2]\!]_{\mathcal{L}} - [\![E_1]\!]_{\mathcal{L}} \cdot [\![E_2]\!]_{\mathcal{L}}).$$

The sigmoid function is defined as

$$\text{sigmoid}(a) = \frac{\exp\{a\}}{1 + \exp\{a\}},$$

while the LogSumExp function is defined as

$$\text{LogSumExp}(\boldsymbol{x}) = \log \left(\sum_i \exp\{x_i\} \right).$$