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 that 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.

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

ADG says: Each paper should have no more than 12 pages, excluding bibliography, in 10pt font

1 Introduction

Statically-typed programming languages aim to enforce correctness and safety properties on programs by guaranteeing constraints on program behaviour. A large scale user-study suggests 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, which imposes the burden of declaring and maintaining these annotations on the programmer. Strongly statically-typed, usually functional languages, like Haskell or ML, offer 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 [**?**]. 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 provide both static and dynamic typing in the same program, they also require adding some optional type annotations to enable more precise inference. Hence, inferring types for dynamic languages without needing the programmer to provide at a least a subset of the type annotations remains 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, the increased volume 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 develop less strict type inference systems that learn to predict types from uncertain information, such as comments, names, and lexical context, even when 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 to extract a statistical model for the text. However, none explicitly models the underlying type inference rules, and thus their predictions ignore useful logical information.

We sieze the opportunity to plug this gap.

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.

(II) *Natural constraints* are statistical constraints on type annotations which can be inferred from relationships between types and surface-level properties such as

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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

names and lexical context. These constraints can be learned by applying machine learning to large code bases. They 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 neural 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 compatible 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 file 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 seeks to predict types for methods that are declared in a TypeScript declaration file.
- We evaluate ProdTS on a corpus of 5800 JavaScript libraries for which the DefinitelyTyped repository provides type declaration files. We find that combining natural and logical constraints has better performance than either alone. Further, ProdTS outperforms state-of-the-art systems, JSNice [**?**] and DeepTyper [**?**]. ProdTS achieves a *50% reduction in error* (measured relatively) over these previous systems.

2 General Framework for Probabilistic Type Inference

This section introduces our general framework, which we instantiate in the next section by building a tool for predicting types in TypeScript. **??** illustrates our general framework through an example of predicting TypeScript types.

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 $\Gamma = \{x_v : t_v \mid v \in 1 \dots V\}$ 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:

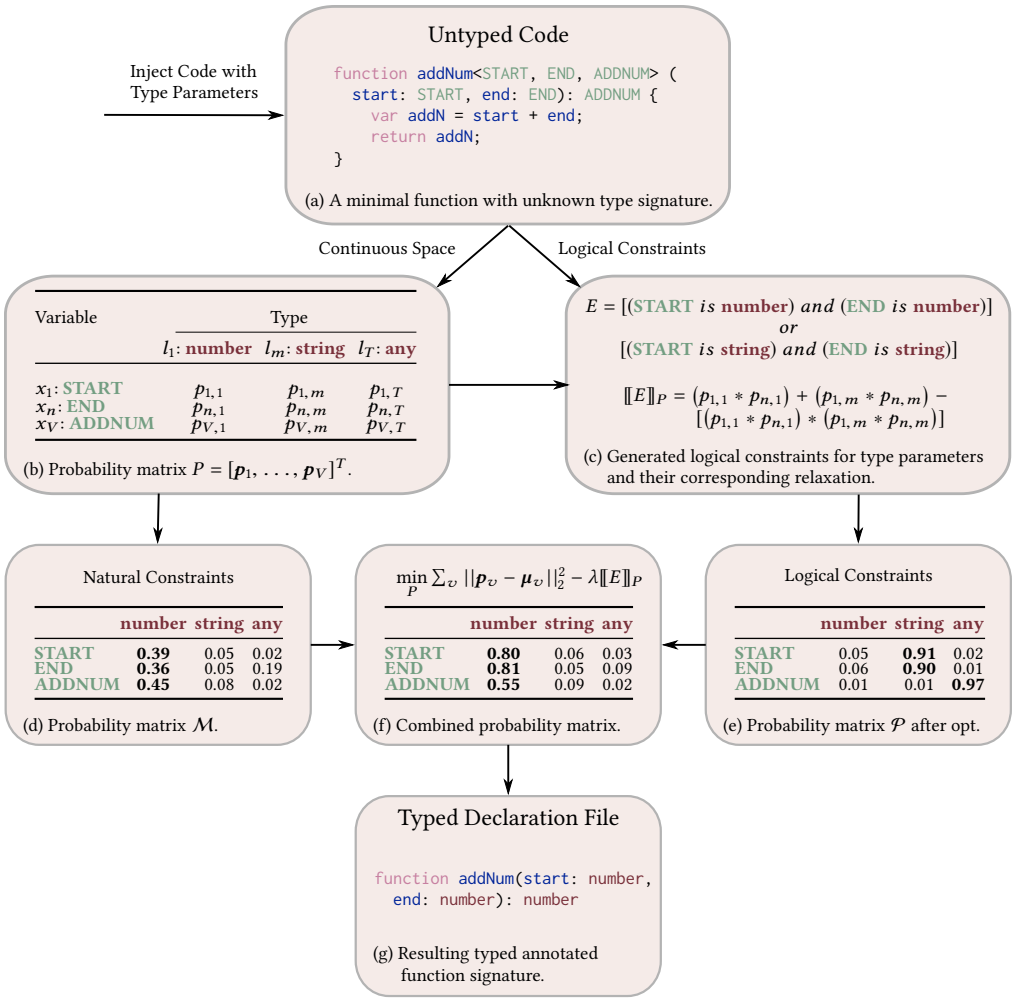
1. We choose a finite universe consisting of T distinct concrete types $\{l_\tau \mid \tau \in 1 \dots T\}$.
2. We compute a set $\{x_v \mid v \in 1 \dots V\}$ of a number V of distinct identifiers in U that need to be assigned types.
3. We extract a set of constraints from U.
4. By optimizing these 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 which we compute from the vth probability vector (the one for identifier x_v). In this work, we pick the column τ that has the maximum probability in x_v's probability vector.
6. The outcome is the environment $\Gamma = \{x_v : t_v \mid v \in 1 \dots V\}$.

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 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,

ADG says: Irene: please mention the μ_v in Box(d) because we later refer to μ_v in Box(f).

ADG says: This is the statement of our general framework. We should tighten the connection to Figure 1. eg $V = 3$ and $T = 3$



Our input is a minimal JavaScript function with no type annotations on its parameters or result. By default, TypeScript's compiler assigns its wildcard type `any` to parameters. Our goal is to exploit both logical and natural constraints to suggest more specific types. To begin, in Box (a), we propose fresh type annotations `START` and `END` (uppercasing the identifier) for each parameter and `ADDNUM` for the return type. We insert these annotations into the definition of the function. Our *logical constraints* on these types represent knowledge obtained by a symbolic analysis of the code in the body of the function. In our example, the use of a binary operation implies that the two parameter types are equal. Box (c) shows a minimal set of logical constraints that state that `addNum`'s two operands have the same type. In general, the logical constraints can be much more complex than our simple example. If we only have logical constraints, we cannot tell whether `string` or `number` is a better solution, and so may fall back to the type `any`. The crux of our approach is to take into account *natural constraints*; that is, statistical properties learnt from a source code corpus that seek to capture human intention. In particular, we use a machine learning model to capture naming conventions over types. We represent the solution space for our logical or natural constraints or their combination as a $V \times T$ matrix P of the form in Box (b): each row vector is a discrete probability distribution over our universe of $T = 3$ concrete types (`number`, `string`, and `any`) for one of our $V = 3$ identifiers. Box (d) shows the natural constraints $\mathcal{M}$ induced by the identifier names for the parameters and return types of our function. Intuitively, Box (d) shows that a programmer is more likely to name a variable `start` or `end` if she intends to use it as a `number` than as a `string`. Returning to the logical constraints, we can relax the boolean constraint of Box (c) to a numerical function on probabilities as shown in Box (e). When we numerically optimize the resulting expression, we obtain the matrix in Box (e); it predicts that both variables are strings with high probability. Finally, Box (f) shows an optimization objective that combines both sources of information: E consists of the logical constraints and each probability vector μ_v (the row of $\mathcal{M}$ for v) is the natural constraint for variable v. Box (f) also shows the solution matrix and Box (g) shows the induced type annotations, now all equal to `number`.

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

the logical constraints do not need to uniquely determine Γ. For this reason, a *natural constraint* encodes 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 type inference as a problem in numerical optimization, with the goal to find a relaxed type assignment that satisfies as much as possible both sorts of constraints. The result of this optimization procedure is the M matrix of probabilities described in step (**??**).

2.2 Logical Constraints in Continuous Space

Logical constraints are extracted from our untyped input program U using standard program analysis techniques. We employ an augmented static 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*.

In this work, we consider the following logical constraints.

Definition 2.1 (*Grammar of Logical Constraints*)**.** A logical constraint is an expression of the following form. Let $\mathcal{E}$ be the set of all logical constraints, while E be their grammar:

$$\begin{aligned} E ::= & \; x_v \text{ is } l_\tau \\ \mid & \; \textit{not } E \\ \mid & \; E \textit{ and } E \\ \mid & \; E \textit{ or } E \end{aligned} \tag{1}$$

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 $P = \begin{bmatrix} \boldsymbol{p}_1 & \dots & \boldsymbol{p}_V \end{bmatrix}^\intercal$, where each $\boldsymbol{p}_v = \begin{bmatrix} p_{v,1} & \dots & p_{v,T} \end{bmatrix}^\intercal$ is a vector that defines a 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 for 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 [**?**]. Specifically, we use the product t-norm, because the binary operation associated with it is smooth and fits with our optimization-based approach. The product t-norm has already been used for obtaining continuous semantics in machine learning, for example by **?**].

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

$$\begin{aligned} [\![x_v \textit{ is } l_\tau]\!]_P &= p_{v,\tau} \\ [\![\textit{not } E]\!]_P &= 1 - [\![E]\!]_P \\ [\![E_1 \textit{ and } E_2]\!]_P &= [\![E_1]\!]_P \cdot [\![E_2]\!]_P \\ [\![E_1 \textit{ or } E_2]\!]_P &= [\![E_1]\!]_P + [\![E_2]\!]_P - [\![E_1]\!]_P \cdot [\![E_2]\!]_P \end{aligned} \tag{2}$$

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. The reason is that it relaxes the deterministic logical semantics of E, and it is maximized by probability matrices P which correspond to satisfying type environments. The following theorem formalizes the idea:

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

To sketch the proof, two facts can be seen immediately. First, for any formula E, the function $\tilde{f}(P) = [\![E]\!]_P$ is a relaxation of the true logical semantics. That is, for any environment Γ, we have that $\tilde{f}(M(\Gamma)) = [\![E]\!]_{M(\Gamma)} = 1$ if and only if Γ satisfies E. This can be shown by induction on the structure of E. Second, for any matrix $P \in \Pi^{V\times T}$, we have the bound $\tilde{f}(P) \le 1$. 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 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

ADG says: Cut for now but restore somewhere. 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

Charles says: This bit is unimportant. It's a pretty simple fact, though, does it merit a theorem?

Charles says: It would be better to have the 'only if' direction as well, and I think that there is a sense in which this is indeed true but there are some more tricky corner cases, e.g., if E does not depend on all of the variables in Γ.

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 $\boldsymbol{\mu}_v = [\mu_{v1}, \ldots, \mu_{vT}]^\mathsf{T}$. We aggregate the probability vectors of the learning model in a matrix defined as $\mathcal{M} = \begin{bmatrix}\boldsymbol{\mu}_1 & \ldots & \boldsymbol{\mu}_V\end{bmatrix}^\mathsf{T}$.

ADG says: changing "naturalness constraints" to "natural constraints" as per the definition.

In principle, natural constraints can be defined based on any property of U, including names and comments. In this paper, we consider a simple but practically effective example of natural constraint, namely, a deep network that predicts the type of a variable from the characters in its name. We consider each variable identifier x_v to be a character sequence $(c_{v1} \ldots c_{vN})$, where each c_{vi} is a character. (This instantiation of the natural constraint is defined only on types for identifiers that occur in the source code, such as a function identifier or a parameter identifier.) This is a classification problem, where the input is x_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 so is to use a recurrent neural network (RNN).

ADG says: I am identifying each variable x_v with the character identified w_v

For our purposes, an RNN is simply a function $(\boldsymbol{h}_{i-1}, z_i) \mapsto \boldsymbol{h}_i$ that maps a 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 $\text{LSTM}(\boldsymbol{h}_{i-1}, z_i)$.

With this background, we can describe the specific natural constraint that we use. Given the name $x_v = (c_{v1} \ldots c_{vN})$, 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 natural constraint $\boldsymbol{\mu}_v$. That is, we define

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

$$\boldsymbol{\mu}_v = F(\boldsymbol{h}_N), \tag{3b}$$

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, but are left to future work.

2.4 Combining Logical and Natural Constraints to Form an Optimization Problem

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

IVP says: Brief motivation of Lagrange multiplier/ridge regression

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 $\mathcal{M}$ subject to the logical constraints being satisfied. A simple way to quantify the distance is via the *Euclidean norm* $||\cdot||_2$ of a vector, that is, the square root of the sum of the squares of its elements. Hence, we obtain the constrained optimization problem:

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

Charles says: This could alternately be formalized as a multi-objective problem. I do not know what is best. Maybe it is better to talk about maximizing [[ ]], because our theorem talks about maximization.

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

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

It is easy to see that if Y minimizes (**??**), then $P = [\sigma(\boldsymbol{y}_1), \ldots, \sigma(\boldsymbol{y}_V)]^\mathsf{T}$ minimizes (**??**).

We choose to use Mean Squared Error (MSE), also called the Brier score in statistics [**? ?**] instead of Cross Entropy (CE) which is is another common choice for probabilities. MSE is a proper scoring rule [**?**], which intuitively means that this loss function will indeed encourage the model to use the correct probabilities. We do not claim any particular advantage to the Brier score versus CE. To remove the final constraint, we introduce a Lagrange multiplier $\lambda > 0$ to weight the two terms, yielding our final optimization problem

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

IVP says: polish this insertion from rebuttal

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 that is closest to one.

3 ProdTS: Predict TypeScript Type Signatures for JavaScript Libraries

To evaluate our approach in a real-world scenario, we implement an end-to-end application, called ProdTS, which aims to infer TypeScript declaration files for an underlying JavaScript library.

3.1 Background: TypeScript's Type System

Syntactically, TypeScript [?] is a typed superset of JavaScript designed for developing large-scale, stable applications. TypeScript's compiler typechecks TypeScript programs then emits plain JavaScript to leverage the fact that JavaScript is the only cross-platform language that runs in any browser, any host, and any OS. Structural type systems consider record types (classes), whose fields or members have the same names and types, to be equal. To compromise between static and dynamic typing, TypeScript supports a structural type system because it permits TypeScript to handle many JavaScript idioms that depend on dynamic typing. One of the main goals 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.

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 manual 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's soft type system [?] defaults to the any type. So, when a parameter has no type annotation, TypeScript assumes that it has type any and does not infer a more specific type. TypeScript does, however, seek to infer more specific types for the return type of a function. It is impractical for TypeScript code to implement run-time casts — typical for gradual type systems — due to the type erasure that necessarily occurs when translating to plain JavaScript [?].

For generating definition files for existing JavaScript libraries, the DefinitelyTyped community officially recommends dts-gen [?]. This tool uses runtime information to produce a `.d.ts` file that clearly defines the shape of the input API but does not provide type information for function arguments and returns. Dts-gen only collects dynamic information. As a result it emits many any types that the developer must refine manually. It is only meant to be used as a starting point for writing a high-quality declaration file. ?] created the TSINFER and TSEVOLVE tools to address the same problem. These tools work by analyzing a recorded snapshot of a concretely initialized library. Like dts-gen, they are good at capturing the structure of the definition file, but often fail to capture readable forms for argument and result types.

3.2 Problem Statement

We consider the 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 as any. We call this step the *Structure* stage. (We obtain this file from dts-gen.)
3. TypeScript's default library declaration file, that is, the built-in types defined by the standard library.[3]

Output: A 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 declaration file from the *Structure* stage but with some, or ideally all, of the any types substituted with built-in types—not including user-defined types—and sum/product types of those.

Let the *Type* stage be the process of predicting types to fill the holes left in the declaration files following the *Structure*.

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

ADG says: Am I correct?

ADG says: Is the Structure versus Type stage helpful? Do we use it later? Will continue to read.

ADG says: is that named interface or class types? first mention of sum/product types

In **??**, we present metrics on the performance of tools, including ProdTS, on the *Type* stage, given skeleton definition files produced by the *Structure* stage.

3.3 Logical Constraints for TypeScript

To generate the logical constraints of **??**, we exploit the information from `tsc`, the TypeScript compiler [**?**]. Instead of modifying `tsc` to emit logical constraints, a substantial engineering effort especially given the speed of `tsc`'s evolution, we devised the following technique to obtain constraints from the unmodified compiler. We harvest type constraints from type errors `tsc` generates. We trigger these errors by explicitly assigning a fresh generic type variable to each formal of each function, then invoking `tsc`. We parse the error messages to construct our logical constraints. We also include return types inferred by the compiler as logical constraints.

We identify 5 common type errors (**??**) that the `tsc` compiler emits, and turn them into type constraints. The main problem with this approach in isolation is that it produces 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.

The first row of **??** refers to identifying properties/methods that a type parameter should implement and using them to generate a type constraint. The file `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 that we could assign to the type parameter and therefore a proper type constraint.

Our purpose is to establish the principle that a combination of natural and logical constraints can outperform either on its own, and outperforms the state-of-the-art. To measure our method versus other tools on our gold files, we had to find a way to generate logical constraints from `tsc`. We judged it better to generate a limited set of constraints by processing the TypeScript error messages than to attempt to modify `tsc`, a highly optimized, complex, and quickly evolving piece of software. Our logical constraints include propositional logic, and therefore seem able to express a wide range of interesting type constraints. This technique seems a useful device that could be employed in other situations, and serves our purpose. Having established the general principle, we will aim to show how to modify a type-checker to omit constraints directly in future work.

3.4 Natural Constraints for TypeScript

We now focus our attention on extracting natural constraints for our 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 use a Char-Level LSTM trained on *(id, type)* pairs to learn naming conventions for identifiers, treated as sequences of characters. The main intuition behind this choice is that developers commonly use multiple abbreviations for the same word and this family of abbreviations shares a type. A Char-Level LSTM is well-suited to predict the type for any identifier in an abbreviation families.

Our universe of types consists of the 78 built-in types or union of them that appear at least 10,000 times in our training set. We consider only built-in types to ensure that we do not introduce types that are not available to the compiler. We leave a consideration of user-defined types to future work.

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, and A and b are learnable weights and bias. The softmax function is defined as

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

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 for a probability vector. We work in log space to help numerical stability since computing (**??**) directly can be problematic. As a result, F outputs values in $[-\infty, 0]$.

We train the model by supplying sets of variable identifiers together with their known types, and minimizing a loss function. Our loss function is the negative log likelihood function—conveniently combined with our log output—defined as:

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

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 training set.

We use Adam, an extension of stochastic gradient descent [**?**], as our optimization algorithm. 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 used all 5,800 projects available from the DefinitelyTyped repository on 17th February 2019. We trained our model for 1,000 epochs, for 78 different types, and obtained a validation accuracy of 0.79. Our dataset was randomly split by project into 80% training data, 10% validation data and 10% test data. Splitting the available data in these three different sets is a common practice in machine learning to ensure that

ADG says: this example is rather out of context: For example a valid constraint on type l would be equivalent to l = *String or* l = Array<any>[4] but not l *has member .length.*

IVP says: is that right??

IVP says: Irene: I changed test to training; can you check?

IVP says: From Charles: We should say where we obtained the data set from, i.e., these are all of the projects on Definitely-

Table 1. The five different type errors from which generate the *logical constraints*.

Constraint-Id	Description
Property	Property X does not exist in type Y.
Binop	Operator $\oplus$ cannot be applied to types X and Y.
Index	Type X cannot index type Y.
ArithLHS	The left-hand side of an arithmetic operation must be an `enum` type or have type `any` or `number`.
ArithRHS	The right-hand side of an arithmetic operation must be an `enum` type or have type `any` or `number`.

the learning model does not simply memorize the training data but is able to generalize to unseen inputs.

3.5 Combining Logical and Natural Constraints

We describe the particular implementation details used to combine the two sources of information. Most of our design choices here are made empirically 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, known as RMSprop [**?**]. We set the maximum number of iterations to 2,000, which suffices in practice for the loss to stabilize.

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 a 2-core Intel Core i5 CPU with 8GB of RAM. Our resulting model requires about 400MB of RAM to be loaded into memory and can be run on both a GPU and CPU. It computes type annotations on average for 58 files in about 60 seconds for solving logical constraints and natural constraints, and in about 65 seconds 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. It 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 used 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 and discard the other exported entities, such as variables, methods, and properties. The metrics presented next can easily be extended to evaluate to those exported entities we currently exclude. The following definitions formalize these metrics.

4.1 Precision and Recall for 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}
- either function or parameter 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 either function result or parameter $T_{\text{total}} = T_{\text{fnRet}} \cup T_{\text{param}}$

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

Definition 4.2 (Functions *Paths* and $Filter_X$)**.** 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, $Filter_X(Y) = X \cap Y$.

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

ADG says: Irene, I removed quotes from "gold". In general, avoid using quotations as a form of emphasis.

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 the corresponding *predicted.d.ts* file 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 formulas defined next. We define $X \in \{S, T\}$ (*Structure* and *Type*) to denote the stage for which we perform measurements. X's range could be extended to include, for instance, S_{fnRet} (function identifiers) to perform a finer analysis.

Definition 4.3 (Precision and 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:

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

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

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 also present results from the first stage, although our method does not contribute to its improvement. For the same reason, we start by presenting the results related to the *Type* stage.

4.2 Evaluation of *Type* Stage

For evaluating type predictions, we compare both the augmented static analysis and the Char-Level LSTM on their own, and then their combination. Two kinds of input may be required by 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 identifier 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).

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 if only 78 types can be predicted, even if all the 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 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 tasks are close. By comparing these two metrics in **??**, their similarity can be traced to the similarity of their denominators. As we discuss in **??**, the *Structure* stage can identify the majority (over 80%) of the function signatures; this causes 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

ADG says: Here we use N for the size of the type universe but earlier we use T for the size. We should change N to T, unless that creates another clash.

ADG says: Are these 1272 identifiers for all types, or just for the universe of 78 types? for all types Best to give both totals.

Table 2. Aggregate *Type* precision and recall for 58 JavaScript libraries with 1272 identifiers in total (610 $funRet$, 662 $param$). We use the notation $P(T^N_{\text{fnRet}})$ for $P(T_{\text{fnRet}} \cap N)$, where N is the number of types in our universe of types.

Tool	$P(T^N_{\text{fnRet}})$	$R(T^N_{\text{fnRet}})$	$P(T^N_{\text{param}})$	$R(T^N_{\text{param}})$	$P(T^N_{\text{total}})$	$R(T^N_{\text{total}})$
Aug-Static (N = 78)	0.42	0.41	0.15	0.15	0.29	0.28
LSTM (N = 78)	0.48	0.48	0.61	0.56	0.55	0.52
ProdTS (N = 78)	**0.61**	**0.59**	**0.63**	**0.58**	**0.62**	**0.58**

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.

ADG says: rephrase this sentence

4.3 ProdTS at the *Type* Stage

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. Neither tool was designed explicitly for predicting TypeScript declaration files, so we made the necessary adjustments to meet the requirements of our setting. For all of our evaluations we have used the same Structure file, that is, the one produced by dts-gen. The two tools that we compare against are:

- DeepTyper [**?**]: A tool based on deep learning which learns types for every identifier. The network is trained using previously annotated TypeScript code.
- JSNice [**?**]: A tool based on probabilistic graphical models which analyzes relationships between program elements to infer types for JavaScript files.

We have tried specifically to address any unfair comparison. Since DeepTyper has a vocabulary of 11000 types, we measure the results on our set of 78 types and give every other type we encounter *OutOfVoc*, so that it does not contribute to the final result. JSNice's vocabulary is smaller than ours that's why we have two rows for JSNice on Table 4, for N=6, which is the original implementation, the results for the types of the parameters are almost identical to the ones of the original JSNice paper. **??** summarizes the results of our comparisons.

The *relative error reduction* is a standard way to express performance improvement in machine learning. In our case, it is how many fewer errors ProdTS makes as a proportion of the errors that JSNice makes. The JSNice N=78 precision of 0.37 implies an error rate 0.63, whereas the ProdTS N=78 precision of 0.68 implies an error rate 0.32. So the relative reduction in error is $(0.63 - 0.32)/0.63 = 49.2\%$.

Comparison with DeepTyper DeepTyper is a deep learning framework that outputs a type vector for every identifier based on information from the source-code context. It utilizes information in the vicinity of the identifier to predict the type. In contrast, our LSTM is trained on identifiers and types; we focus on obtaining information based on the identifier alone and not its context.

For the results shown in **??**, DeepTyper often 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.

Even though our LSTM is trained only on identifiers and types—while DeepTyper utilizes more context—predictions for both tools are comparable for parameters' types. For function return types, our Char-Level LSTM clearly outperforms DeepTyper. This shows that taking into account information in the vicinity can be problematic; function definitions may be placed relatively far away from their calls and hence the context is not very informative.

Finally, it is worth pointing out that DeepTyper has a type vocabulary of size $N = 11000$, much larger than our vocabulary of size $N = 78$, because it includes user-defined types too. If DeepTyper were trained on a smaller size vocabulary the results—at least for the predicted types of parameters—might improve. Further, perhaps taking into account user-defined types needs extra consideration; simply learning user-defined identifier and type pairs, as DeepTyper does, might not be adequate and may, in fact, even worsen its performance.

ADG says: Irene, roughly how often does DeepTyper return any? Saying just "often" is not very informative!

Comparison with JSNice JSNice is a tool that learns statistical correlations between program elements by shallowly exploiting their relationship. Its purpose is for type inference (and other tasks) on JavaScript using statistics from dependency graphs learned on a large corpus. The evaluation on our data had to be performed manually because JSNIce is only available via a website interface.

JSNice has a type lattice that contains only $N = 6$ primitive types and exploits the relationships between types of shallow depth. Therefore, the tool does not consider the additional issues involved with predicting precise types, in which the system must infer what level of generality is most

ADG says: "shallowly exploitin" doesn't sound like a Good Thing— can we rephrase?

Table 3. 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.

Tool	$P(T^N_{\text{fnRet}})$	$R(T^N_{\text{fnRet}})$	$P(T^N_{\text{param}})$	$R(T^N_{\text{param}})$	$P(T^N_{\text{total}})$	$R(T^N_{\text{total}})$
DeepTyper($N = 78$)	0.20	0.20	0.50	0.52	0.35	0.36
JSNice ($N = 6$)	0.15	0.15	0.78	0.80	0.47	0.48
JSNice ($N = 78$)	0.12	0.12	0.62	0.64	0.37	0.38
ProdTS ($N = 78$)	**0.68**	**0.68**	**0.67**	**0.67**	**0.68**	**0.68**

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. Regarding the type recall of the parameters in **??** for the $N = 6$ case, JSNice scored better results than our tool. This may be because of the property that we discussed already regarding the use of a small size vocabularies.

Overall, we conjecture that the 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 regardless 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 that 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, especially for function return types, there is occasionally insufficient information to capture the information flow.

NL2Type, a tool by [**?**], also uses a deep learning approach to the problem, and relies on JSDoc comments as an additional type hint. We could not compare directly to ML2Type because there are few examples with both JSDoc and available declaration files in the DefinitelyTyped repository. Finally, it would be fairly simple to extend our method to include natural constraints generated by DeepTyper, NL2Type or indeed any other deep learning approach that offers similar information. For example, we could simply add more terms to the combined objective function, including an extra term for every additional source of natural constraints.

4.4 Comparison for *Structure*

We return to 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`, statically generates some exported definitions.
- *dts-gen*: A tool that dynamically examines JavaScript objects at runtime to generate exported definitions.

?? shows the results of the comparison. Clearly *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

ProdTS is a new form of probabilistic type inference that optimises over both logical and natural constraints. Related spans classical, deterministic type inference and earlier machine learning approaches.

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).

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 [**?**]. There are similar extensions for other popular scripting languages, like [**?**], an optional static type checker for Python, or RuboCop [**?**], which serves as a static analyzer for Ruby by enforcing many of the guidelines outlined in the community Ruby Style Guide [**?**].

The quest for more modular and extensible static analysis techniques has resulted in the development of richer

IVP says: add description of jsnice, 2 cases, one with 6 types only the one that jsnice is trained(results comparable with the result reported on jsnice for parameters but not for return type) 78 our set of types we do better in each case, main critique the fail to capture return types

ADG says: perhaps this discussion should come before the discussion of the type stage?

Earl says: This must come first! It's too confusing to report the first stage second!

ADG says: There are purely statically typed languages with powerful inference so that annotations are infrequent. Any work to date on ML for statically typed languages? Not AFAIK

Earl says: There is work on ML for stat-

Table 4. 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.

Tool	$P(S_{\text{fnRet}})$	$R(S_{\text{fnRet}})$	$P(S_{\text{param}})$	$R(S_{\text{param}})$	$P(S_{\text{total}})$	$R(S_{\text{total}})$
declFlag	0.09	0.11	0.08	0.11	0.08	0.11
dts-gen	0.88	0.84	0.83	0.79	0.83	0.79

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 (such as SAT and SMT engines) to 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. DependentJS [**?**] incorporates dependent types into JavaScript.

5.2 Machine Learning Over Source Code

Although the interdisciplinary field between machine learning and programming languages is still young, complete reviews of this area are already available. **?**] extensively survey work that probabilistically model source code via a learning component and complex representations of the underlying code. **?**] give a detailed description of the area, whilst **?**]'s position paper examines 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. Two earlier approaches for probabilistic typing are JSNice [**?**] and DeepTyper [**?**]. Indeed, they serve as baselines in our evaluation, where we discuss them in detail (**??**).

6 Conclusion and Future Work

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 function 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 here is a basis for constructively combining different type analyses by using numerical methods.

Our system can be trained on the most common types occurring in any codebase. We limit ourselves to simple common types and products of them, primarily because we did not want to introduce types unknown to the compiler imported from other libraries. We leave the challenging task of introducing user defined types from scratch as future research. It is straightforward though to extend our set of types to whatever type as long as we provide also its interface. We consider as a more challenging problem the extension to user-defined types that emerge as logical constraints and to predict natural names for them.

A Appendix: 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. The *logit* of a probability is the logarithm of the odds ratio. It 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:

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

ADG says: Irene, I have trimmed the discussion of dependent types as it's not the focus of the paper.

IVP says: Arguably this can claim to be doing this, but much more loosely compared to you. (but since you might get the authors of the papers to be your reviewers it might be worth mentioning them)

https://ml4code.github.io/publications/raychev2015scalable/

https://pdfs.semanticscholar.org/cfd1/b5c47aee34173033d7fcc4dda03ab4889e98.pdf

(<- I have not yet

The sigmoid function is defined as

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

while the LogSumExp function is defined as

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