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.

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

[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 DeepTyper [**?**]. 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. TypeScipt'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 $\Gamma = \{x_v : t_v \mid v \in 1 \dots V\}$ 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.
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 where column τ has the maximum probability in the vth probability vector (the one for identifier x_v).
6. The outcome is the environment $\Gamma = \{x_v : t_v \mid v \in 1 \dots V\}$.

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

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

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 $P = \begin{bmatrix} \boldsymbol{p}_1 & \dots & \boldsymbol{p}_V \end{bmatrix}^\mathsf{T}$, where each row $\boldsymbol{p}_v = \begin{bmatrix} p_{v,1} & \dots & p_{v,T} \end{bmatrix}^\mathsf{T}$ 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 $[\![E]\!]_P$ is a function $\Pi^{V\times T} \times \mathcal{E} \to [0,1]$, defined as

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

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 $P \in \arg\max_{P\in\Pi^{V\times T}} [\![E]\!]_P$.*

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

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 $\text{LSTM}(\boldsymbol{h}_{i-1}, z_i)$.

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

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

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

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, 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

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

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

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

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

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

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 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.

A Appendix

Text of appendix …