Probabilistic Type Inference by Optimizing Logical and Natural Constraints*

Irene Vlassi Pandi[1], Earl T. Barr[2], Andrew D. Gordon[1,3], and Charles Sutton[1,4,5]

[1] University of Edinburgh, Edinburgh, UK
[2] University College London, London, UK
[3] Microsoft Research Cambridge, Cambridge, UK
[4] The Alan Turing Institute, London, UK
[5] Google AI, Mountain View, USA

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, that is, uncertain statistical information about types learnt 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. The foremost 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 optimisation problem. We build a tool called *OptTyper* to predict missing types for TypeScript files. *OptTyper* combines a continuous interpretation of logical constraints derived by classical static analysis of TypeScript code, with natural constraints obtained from a deep learning model, which learns naming conventions for types from a large codebase. By evaluating *OptTyper*, we show that the combination of logical and natural constraints yields a large improvement in performance over either kind of information individually and achieves a 4% improvement over the state-of-the-art. The full paper of this work is available on arxiv [1].

In our view, probabilistic type inference should be considered as a constrained problem, as it makes no sense to suggest types that violate type constraints. To respect this principle, a principled framework for probabilistic type inference that couples hard, *logical* type constraints with soft constraints drawn from structural, *natural* patterns into a single optimisation problem. While, in theory, there is the option of filtering out the incorrect predictions, our framework goes beyond that; our composite optimisation serves as a communication channel between the two different sources of information.

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.

* This work was supported by Microsoft Research through its PhD Scholarship Programme.

(II) *Natural* Constraints are statistical constraints on type annotations that can be inferred from relationships between types and surface-level properties such as names and lexical context. These constraints can be learned by applying machine learning to large codebases.

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 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 probabilistically predicts the type of each identifier.

The key idea behind our methods is a *continuous relaxation* of the logical constraints. This means that 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 relaxation has a special property, namely, that when this continuous function is maximised with respect to the relaxed type environment, we obtain a discrete type environment that satisfies the original constraints. The benefit of this relaxation is that logical constraints can now be combined with the probabilistic predictions of type assignments that are produced by machine learning methods.

More specifically, this allows us to define a continuous function over the continuous version of the type environment that sums the logical and natural constraints. And once we have a continuous function, we can optimise it: we set up an optimisation problem that returns the most natural type assignment for a program, while at the same time respecting type constraints produced by traditional type inference. Our main contributions follow:

- We introduce a general, principled framework that uses soft logic to combine logical and natural constraints for type inference, based on transforming a type inference procedure into a single numerical optimisation problem.
- We instantiate this framework in *OptTyper*, a probabilistic type inference tool for TypeScript.
- We evaluate *OptTyper* and find that combining logical and natural constraints has better performance than either alone. Further, *OptTyper* outperforms state-of-the-art systems, LambdaNet, DeepTyper and JSNice.
- We show how *OptTyper* achieves its high performance by combining *Logical* and *Natural* constraints as an optimisation problem at test time.

Fig. 1 illustrates our general framework through a simplified running example of predicting types. Our input is a minimal function with no type annotations on its parameters or result. To begin, in Box (a), we propose fresh type annotations `START` and `END` for each parameter and `ADDNUM` for the return type. We insert these annotations into the function's definition. Our *logical constraints* on these types represent knowledge obtained by a symbolic analysis of the code in the function's body. 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

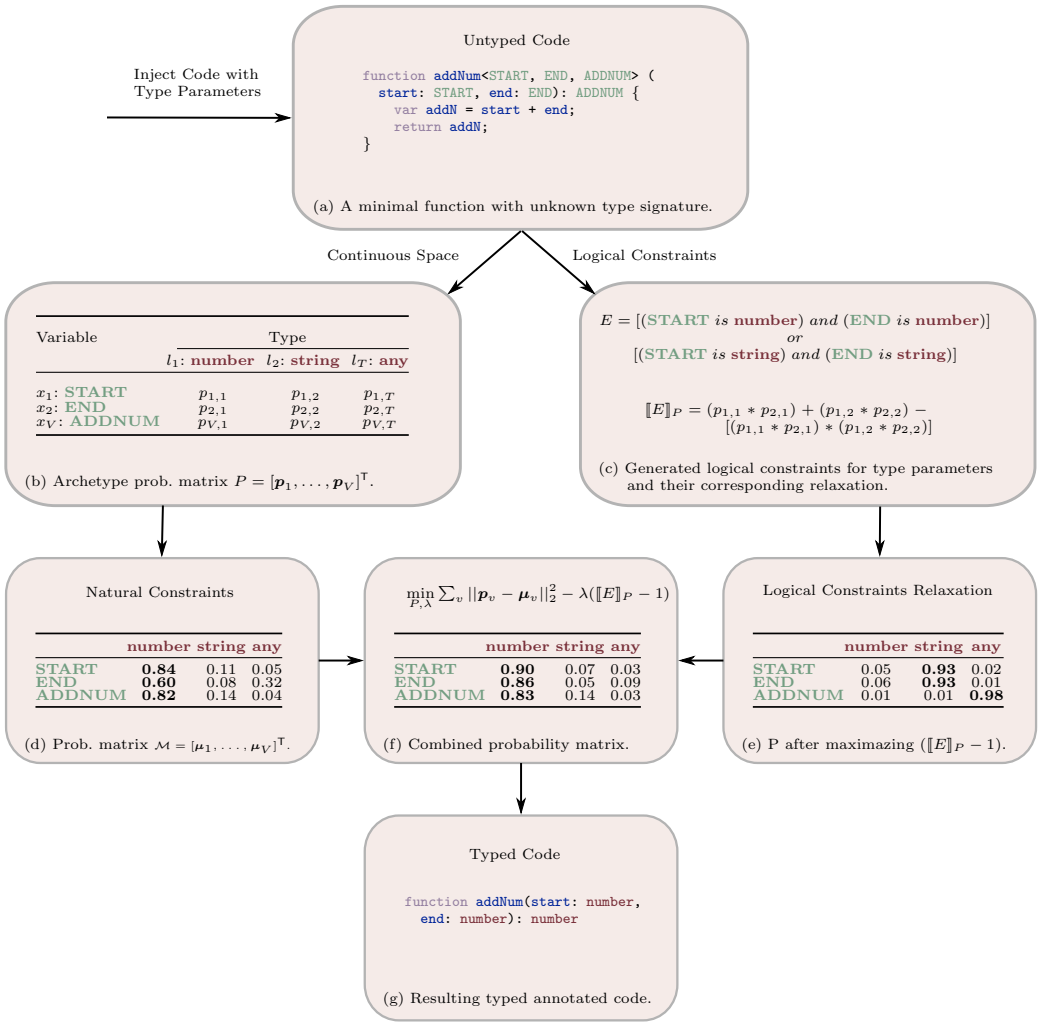

Fig. 1: An overview of the three type inference procedures via a minimal example.

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 the function name itself. 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`. We can relax the boolean constraint to a numerical function on probabilities as shown in Box (c). When we numerically optimise the resulting expression, we obtain the matrix in Box (e); it predicts that both variables are strings with high probability. Although the objective function is symmetric between `string` and `number`, the solution in (e) is asymmetric because it depends on the initialisation of the optimiser. Finally, Box (f) shows an optimisation 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 predicted to be `number`.

References

1. Pandi, I.V., Barr, E.T., Gordon, A.D., Sutton, C.: Probabilistic type inference by optimising logical and natural constraints (2020), https://arxiv.org/abs/2004.00348