Learning Type Annotation: Is Big Data Enough?

Kevin Jesse
University of California, Davis
Davis, USA

Premkumar T. Devanbu
University of California, Davis
Davis, USA

Toufique Ahmed
University of California, Davis
Davis, USA

ABSTRACT

TypeScript is a widely used optionally-typed language where developers can adopt "pay as you go" typing: they can add types as desired, and benefit from static typing. The "type annotation tax" or manual effort required to annotate new or existing TypeScript can be reduced by a variety of automatic methods. Probabilistic machine-learning (ML) approaches work quite well. ML approaches use different inductive biases, ranging from simple token sequences to complex graphical neural network (GNN) models capturing syntax and semantic relations. More sophisticated inductive biases are hand-engineered to exploit the formal nature of software. Rather than deploying fancy inductive biases for code, can we just use "big data" to learn natural patterns relevant to typing? We find evidence suggesting that this is the case. We present **TypeBert**, demonstrating that even with simple token-sequence inductive bias used in BERT-style models and enough data, type-annotation performance of the most sophisticated models can be surpassed.

CCS CONCEPTS

• **Theory of computation** → *Type structures*; *Type theory*; • **Computing methodologies** → *Machine learning*; *Transfer learning*.

KEYWORDS

Type inference, deep learning, transfer learning, TypeScript

ACM Reference Format:
Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3468264.3473135

1 INTRODUCTION

Gradual typing [6, 23, 24] is gaining popularity, in programming languages like Python and JavaScript. Developers can *incrementally* type-annotate identifiers to better document, check, and maintain code [14]. Type annotation promotes error-detection, [9, 19] while enabling more optimizations, and better IDE support. However, with type declarations existing in various library packages and project-specific locations, migrating dynamically typed software to gradually-typed paradigms is a non-trivial task, often requiring considerable human effort.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473135

TypeScript *transpiles* type-annotated code into JavaScript (JS) which provides the benefits of typing wherever traditional JS is used [5]. A lot of TypeScript annotated code is available; this raises the opportunity to train *probabilistic type annotators* to help with type annotation. This idea of training a type annotator using data from manually annotated code, has been widely applied [11, 17, 21, 25]; except for Raychev *et al* [21], most use deep-learning methods. Each probabilistic annotator features a specific choice of representation, viz., *inductive bias,* that characterizes what and how they learn. Inductive biases are important, consequential, and well-studied. But do more complex inductive biases help? Do they perform better?

Recently, in NLP [7] and code [8, 12], an alternative paradigm has emerged, to the ongoing quest for better inductive-biases: *let high-capacity models learn representations on their own* which capture the deeper statistical structure of the data, directly from very large dataset, using a form of self-supervision. In the case of NLP, Devlin *et al* [7] exploit giga-token textual corpora to construct a vector representation of token sequence patterns, by learning to reconstruct artificially masked-out tokens. This approach elegantly bypasses the debates on inductive-bias engineering, and simply lets high-capacity neural models *autonomously learn the statistical structure of the data* via simple, giga-scale self-supervision.

Autonomous representation-learning (aka pre-training) has been used for code. Feng *et al.* [8] used pre-training to improve performance on code-natural language bi-modal datasets (*e.g.* code with comments) and Kanade *et al.* [12] used pre-training to help with retrieval-like tasks. Type annotation is of particular interest: types are a subtle semantic property of code; one might reasonably expect that complex inductive biases leveraging syntax & semantics would be very helpful. Prior work has indeed heavily leveraged increasingly sophisticated inductive biases, with better and better results. But are these really necessary? Can models learn good enough representations *on their own?* This motivates our RQs.

RQ1: Does BERT-style pre-training work for type inference, and how does the performance compare with models that use sophisticated, custom-designed inductive biases? Pre-training helps our TypeBert reach 89.51% accuracy on common (top-100) types compared to the state-of-the-art LambdaNet (66.9% for the same types). Furthermore, despite the limits of a closed type vocabulary, TypeBert does surprisingly well on user defined types. Overall, TypeBert achieved an overall accuracy of 71.12% to LambdaNet's 64.2% across both common and user-defined types.

RQ2: Qualitatively, what cues does TypeBert appear to use for its inferences, and what inferences does it make? TypeBert seems to use multiple features for type inferences. Like TypeWriter [20], it appears to leverage names of identifiers; like LambdaNet *etc.* [21, 25] it appears to use data and control-flow information. Using these cues, TypeBert predicts types with specificity, *i.e.* `tf.Tensor` rather than `Tensor`. Overall, our qualitative analysis

suggests that TypeBert implicitly learns complex inductive biases like data/control flow, even without explicit graph representations.

Our models and datasets are publicly available[1].

2 RELATED WORK

LambdaNet [25] (like other recent works) used sophisticated inductive biases [3, 4, 10] to achieve state-of-the-art (SOTA) type inference, improving substantially upon earlier approaches like Hellendoorn [11] and Malik *et al.* [17]. LambdaNet uses graph neural networks (GNN) to model abstract dependency graphs, derived by analysis of the code. Typilus [4] also uses GNNs, but includes vector embeddings to allow an open type vocabulary (for Python). Pradel [20] combines a probabilistic guessing component with a typechecker that verifies the proposed annotations. OPTTyper [18] achieves performance close to LambdaNet, by optimizing formal type constraints that are first "slackened" into numerical constraints; however OPTTyper is limited to the top 100 most frequent types ignoring the challenge of annotating user defined types.

Related to this work are highly parametrized pre-trained transformer models like CodeBert [8] and PLBART [1]. These models have successfully achieved SOTA on code-related tasks by pre-training on large code corpora and fine-tuning on the specific tasks. CodeBert's effectiveness suggests that self-supervised pre-training followed by fine-tuning may also work for type inference. Our approach differs in that our pre-training is mono-lingual; we don't use any natural language, and is pretrained on a type-free dialect (JavaScript) of the target language (TypeScript). Our goal was also to evaluate if pre-training could learn representations powerful enough for type inference.

3 TYPEBERT

TypeBert uses pre-training to learn JavaScript syntax and semantics by modeling token co-occurrences.

3.1 *Pre-Training TypeBert*

Pre-Training Corpus TypeBert is pre-trained on a large corpus of JavaScript. We collected the most-starred 25,000 Github JavaScript projects, using the GraphQL[2]. To avoid bias, we remove duplicate snippets using Allamanis's method [2]. We remove non-code related entities like block comments and copyright blocks. The corpus is tokenized with a SentencePiece model [16] with a vocabulary of 16k subtokens. Tokenizing with Byte Pair Encoding (BPE)[22] or with a unigram language model like SentencePiece [15] is a common approach to manage large code vocabularies [13].

Architecture TypeBert uses the same architecture as BERT_{large} [7]. TypeBert has 24 layers of encoder with model dimension of 1024 and 16 attention heads (340M parameters). Finally, we add an output classification layer for the type inference task (after pre-training).

Noise functions Pre-training is self-supervised; the task is reconstructing noised-up text sequences. By training on this task, the model learns prevalent syntactic and semantic forms. TypeBert follows BERT [7] where "noising" consists of randomly masking, replacing, or retaining sub-tokens. We uniformly sample (sub)tokens with a 15% probability and perform noising. Noising operations are weighted thus: 80% are masked, 10% are replaced with a random token, 10% are left alone. We allow up to 20 noise operations per token sequence. The noise function performs whole-word masking (*viz.*, all subtokens of a particular word are all masked if one subtoken is selected) so as to not provide too easy hints to the model. TypeBert is jointly pre-trained with a next sentence prediction (NSP) task which is to predict if a code sequence b follows another code sequence a. For this, we select in-order or random pairs (in equal proportion) and train the model to label them correctly.

Input/Output Format The input format for the pre-training step is two concatenated, randomly sampled, code sequences with a separator token [*CLS*], a_1, a_2, …,a_n , [*SEP*], b_1, b_2, …, b_n, [*SEP*]. Sometimes the a and b code sequences are contiguous, sometimes not; the NSP task is to distinguish these cases. For the random masking, any a_i or b_i may be noised. The [*CLS*] token's embedding is used as an aggregated sequence representation for tasks at the sequence level. As in BERT [7] the two sequences are separated by a [*SEP*] token.

Optimization We train TypeBert on 6 Nvidia Titan RTX GPUs for 200K steps. We use Adam with polynomial weight decay starting at 5e-5 and 10K warm-up. We use a dropout of 0.1 on hidden and attention layers. Pre-training takes about 160 hours (6.33 days) and was done using a modified version of Tensorflow's Model Garden. This pre-training is a one-time cost, followed by on-task fine-tuning.

Table 1: Type Annotation Datasets

Dataset	Projects	Files
TypeBert	20,860	1,473,418
LambdaNet | OPTTyper	275	91,228

* LambdaNet / OptTyper parsed projects and files for comparison.

3.2 *Fine-Tuning TypeBert*

Type Inference Dataset We collected 20,860 most-starred Github TypeScript projects (Table 1). This code contains human-annotated types within variable, parameter, function and method declarations. These types range from frequent types like `int` and `string` to library and user-defined types like `tf.Tensor` and `CoffeeFlavor`. We use LambdaNet's type parser to process type annotations, gathering both human annotated and compiler-inferred types. Following LambdaNet[25] and OPTTyper, we only use the human annotated locations for evaluation but include the compiler-inferred types in training data. Furthermore, as in prior work, we ignore locations with the uninformative "`any`" type. Of LambdaNet's full 300 project dataset, 275 could be found on Github. From these, we sample 60 projects for testing, and just add the other 215 to our training set. This results in a total of 20,384 projects for training, 416 for validation (2%) and 60 projects for test. We report results on these 60 projects from the original LambdaNet/OPTTyper dataset.

Type Inference We add a dense, softmax layer for the most frequent 40k TypeScript types and the `UNK` type for all types greater than rank 40k. The type vocabulary is closed, and restricted to these 40,001 types. TypeBert does not handle a open vocabulary, but `UNK` occurs <8% in the test set see Figure 1. TypeBert is fine-tuned on

[1] https://github.com/typebert/typebert
[2] https://graphql.org

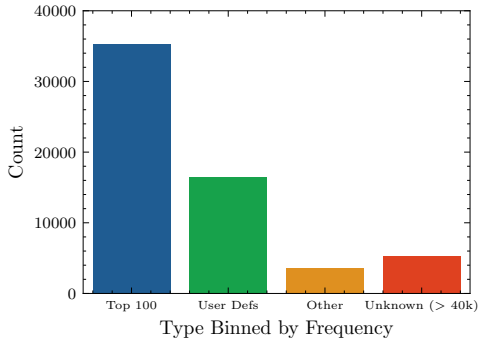

Figure 1: Frequency of types in bins. Types that exceed the Top 40,000 are marked `UNK` and scored incorrect in metrics.

Table 2: Accuracy Comparison across Various Sets of Types.

Model	Top 1 Acc %				Top 5 Acc %			
	User Def	Other	Top 100	Overall	User Def	Other	Top 100	Overall
LambdaNet [25]	**53.4**	N/R	66.9	64.2	**77.7**	N/R	86.2	**84.5**
OPTTyper [18]	N/R	N/R	76	N/R	N/R	N/R	N/R	N/R
TypeBert	41.40	50.49	**89.51**	**71.12**	55.02	70.34	**98.51**	81.88

* `UNK` (OOV) annotations are always counted incorrect for TypeBert. Overall includes User Def and Top 100 and reported directly from [25] and [18]. N/R → Not Reported in the original paper. Allamanis [2] deduplication method applied on train and test sets for TypeBert results.

our data set consisting of > 2 million type annotations. We use de-duplication [2] to avoid risk of leakage from training to test.

4 EVALUATION METRICS

We report Top 1 Accuracy (Exact Match) and Top 5 Accuracy (correct prediction in top 5 guesses) for several subsets of types exactly as with previous works.

User Defined Types are type labels corressponding to class, enum, or type interface, where the type was defined within the same project scope (as in [25]). A class defined within the project would be considered a user-defined type; an imported library would not.

Top 100 Types are highly frequent types (such as `int` and `string`) that are not user-defined, and are within the top 100 ranks [18, 25].

Other Types are types occurrences that do not occur within the top 100 most frequent types and are not user defined. Examples would be library functions like `tf.Tensor4D`. This set of locations *were ignored* in previous works [18, 25] and are not included in their reported results. We consider them, and report it separately.

Overall is a weighted average of user-defined type occurrences and top 100; this is calculated exactly as in [25], and is strictly comparable (See Table 2).

Unknown (OOV) type occurrences which are types with rank < 40,000. We score occurrences of `UNK` (<8%) as incorrect predictions. `UNK` locations are comprised of a mixture of user-defined and other non user-defined types. Counting `UNK` occurrences as wrong for user definition, other, and overall is conservative but appropriate.

5 RESULTS

RQ1: Comparing TypeBert's type inference accuracy to SOTA.

Table 2 shows results on the test set of projects in LambdaNet/OPTTyper dataset. LambdaNet [25] serves as our baseline because it is evaluated on both the top-100 most frequent types and on user-defined types. Its use of a sophisticated graph inductive bias makes it a good contrast for our pre-training/fine-tuning approach, with a very simple token-sequence basis. TypeBert betters LambdaNet

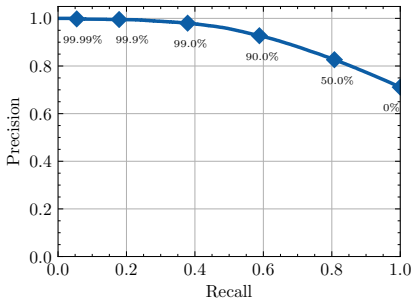

Figure 2: Recall vs. Precision of TypeBert on test data subject to probability thresholds reflecting the models confidence.

on the top-100 (89.51% Top 1 vs 66.9%), despite the disadvantage of a much simpler inductive bias. This suggests that large scale pre-training helps TypeBert autonomously learn nuanced, rich contextual representations that rival LambdaNet's complex hand-engineered hyper-edges. LambdaNet's does excel on user defined types; still, TypeBert (even without any mechanism for user definitions) achieves a higher Top 1 overall accuracy (71.3% vs 64.2%). TypeBert's Top 5 accuracy on the top 100 types (98.5%), compared to LambdaNet's (86.2%) is significantly better demonstrating more relevant predictions across a set of five recommendations; for developers this means more relevant choices to choose from.

While TypeBert demonstrates high Top 1 and Top 5 accuracy, it also performs well with high confidence. Figure 2 shows the trade-off in precision and recall when varying the confidence threshold. Precision exceeds 92.72% with a threshold of 90% with a recall of 58%. At a threshold of 99%, precision exceeds 98% with a recall rate of 38%. TypeBert could add ca. 22,000 of the ca. 58,000 annotations across the 60 test projects with very high precision. TypeBert is quite fast: on a single Nvidia Titan RTX can perform type inference on 16,384 locations in a batch of 64 sequences of length 256 in just 1.28s or .02s per sequence. LambdaNet, we note, requires a dependency hypergraph: computing which correctly is limited by missing dependencies, libraries, and type definitions; thus it cannot perform accurately at such locations. This is not a problem for TypeBert. It's important to note that TypeBert performs creditably on "Other" types, (50.5% for Top 1, 70.3% Top 5); this category is not dealt with by LambdaNet. Finally, we note that OPTTyper works only for for top-100 types, and we improve upon it as well.

RQ2: Qualitative analysis of TypeBert.

What evidence does TypeBert use to make type predictions? We examine this with an example. Figure 3 shows a function signature (top), with correct annotations; inferences from TypeBert and LambdaNet below (correct inferences shown in yellow). This file imports `tfjs-core` with `import * as tf`. Thus syntactically correct types from `@tensorflow/tfjs-core` must have a prefix of "`tf.`". TypeBert recognizes this context and correctly infers `Tensor4D` with the appropriate prefix i.e `tf.Tensor4D`. TypeBert maintains consistency in this example and types variable `x` as `tf.Tensor4D`; the same type and appropriate prefix. As another example, to infer boolean type for `withRelu`, TypeBert appears to take cues from the return statement; to get array type for `strides`, it appears to be using the call to `convLayer` within `conv`. These control and data-flow oriented semantic cues are being learned implicitly from

A: https://github.com/justadudewhohacks/face-api.js/blob/master/src/faceRecognitionNet/convLayer.ts

```
function convLayer(
  x: tf.Tensor4D, params: ConvLayerParams, strides: array,
withRelu: boolean, padding: string
): tf.Tensor4D {
  const { filters, bias } = params.conv;
  let out = tf.conv2d(x, filters, strides, padding);
  out = tf.add(out, bias);
  out = scale(out, params.scale);
  return withRelu ? tf.relu(out) : out
}

export function conv(x, params) {
  return convLayer(x, params, [1, 1], true)
}
```

TypeBert

convLayer		x		params	
tf.Tensor4D	84.36%	tf.Tensor4D	99.12%	UNK	55.22%
Tensor	2.73%	tf.Tensor2D	0.39%	ConvParams	0.78%
tf.Tensor2D	2.37%	TensorInfo	0.07%	object	0.44%
object	2.30%	Tensor4D	0.05%	SeparableConvParams	0.20%
Tensor4D	2.21%	tf.Tensor3D	0.03%	Params	0.13%

strides		withRelu		padding	
array	98.93%	boolean	99.9886%	string	98.911%
Array	0.39%	Boolean	0.0095%	UNK	0.356%
ReadonlyArray	0.27%	string	0.0008%	PaddingMode	0.016%
number	0.15%	number	0.0005%	number	0.013%
Set	0.07%	UNK	0.0003%	String	0.010%

LambdaNet

strides		withRelu	
Function	12.45%	MtcnnBox	9.64%
String	10.14%	BoundingBox	5.44%
Number	8.44%	ComputeAllFaceDescriptorsTask	5.34%
NetInput	6.68%	FaceLandmarkNet	5.31%
Object	5.70%	DetectAllFaceLandmarksTask	4.86%

Figure 3: Qualitative evaluation of TypeBert and LambdaNet.

lexical sequences. LambdaNet fails to infer the return value type and cannot provide type recommendations for 4 other locations; this is likely the result of types existing outside of LambdaNet's prediction space both top 100 and its pointer mechanism.

What kinds of inferences does TypeBert make? A characteristic of TypeBert's top-k type guesses for an annotation are lexical and semantic similarities (Figure 3). This is due to the contextual usage of similar types i.e `tf.Tensor4D` and `tf.Tensor2D` and an alignment of meaning representation i.e `array` and `Set`. While TypeBert seems highly confident when it is correct, the other alternatives tend to be relevant, and sometimes even partially-correct *e.g.* `Array` (.39%) and `Boolean` (.0095%) for `array` and `boolean`.

Finally, TypeBert is strongly confident when the answer is outside its closed vocabulary (UNK). This confident `UNK` prediction has value: in future work, we hope to use open vocabulary mechanisms such as pointer networks or metric similarity functions to search for a better answer in such cases. This example (Figure 3) is typical; most often, TypeBert's offers correct inferences with high confidence and with highly similar alternatives.

6 CONCLUSION

We present a "big-data" alternative to the type inference problem: we use a pre-trained BERT-style model rather than custom-engineering a complex, specialized inductive bias and training dataset. TypeBert uses the simplest inductive bias: considering code as a sequence of tokens. The lack of input structure is overcome by increased learning capacity of the BERT-style approach. TypeBert leverages pre-training on 25,000 JavaScript projects, and fine-tuning on 20,800 TypeScript projects. We find that TypeBert is competitive with SOTA approaches which use much fancier inductive biases. It infers the exact type in common locations almost 90% of the time beating the SOTA models by a significant margin. Additionally, TypeBert's Top 1 accuracy overall is better than the the SOTA, at 71.12%. Our findings suggest that TypeBert *implicitly* learns the statistics of the semantic relationships, relevant to typing, that are made explicit in the graph-based static analysis products (*e.g.*, those used by LambdaNet). It is intriguing to contemplate that generic, automated methods can utilize additional model capacity to "learn" to do some sort of static analysis.

ACKNOWLEDGMENTS

We gratefully acknowledge support from NSF CISE (SHF) Grant No. 1414172.

REFERENCES

[1] Wasi Uddin Ahmad et al. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]

[2] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 143–153.

[3] Miltiadis Allamanis et al. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*.

[4] Miltiadis Allamanis et al. 2020. Typilus: neural type hints. *arXiv preprint arXiv:2004.10657* (2020).

[5] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding typescript. In *European Conference on Object-Oriented Programming*. Springer.

[6] Benjamin Chung et al. 2018. KafKa: gradual typing for objects.

[7] Jacob Devlin et al. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Zhangyin Feng et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[9] Z. Gao, C. Bird, and E. T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 758–769.

[10] Justin Gilmer et al. 2017. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212* (2017).

[11] Vincent J. Hellendoorn et al. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[12] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code.

[13] Rafael-Michael Karampatsis et al. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *International Conference on Software Engineering (ICSE)*.

[14] S. Kleinschmager, R. Robbes, et al. 2012. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 153–162.

[15] Taku Kudo. 2018. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. arXiv:1804.10959 [cs.CL]

[16] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[17] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[18] Irene Vlassi Pandi et al. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. *arXiv preprint arXiv:2004.00348* (2020).

[19] Jihyeok Park. 2014. JavaScript API misuse detection by using typescript. In *Proceedings of the companion publication of the 13th international conference on Modularity*.

[20] Michael Pradel et al. 2019. TypeWriter: Neural Type Prediction with Search-based Validation. *arXiv preprint arXiv:1912.03768* (2019).

[21] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

[22] Rico Sennrich et al. 2015. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909* (2015).

[23] Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

[24] Michael M Vitousek et al. 2014. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic languages*. 45–56.

[25] Jiayi Wei et al. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. *arXiv preprint arXiv:2005.02161* (2020).