FlexType: A Plug-and-Play Framework for Type Inference Models

Sivani Voruganti[†]
svoruganti@ucdavis.edu
University of California, Davis
Davis, CA, USA

Kevin Jesse[†]
krjesse@ucdavis.edu
University of California, Davis
Davis, CA, USA

Premkumar T. Devanbu
ptdevanbu@ucdavis.edu
University of California, Davis
Davis, CA, USA

ABSTRACT

Types in TypeScript play an important role in the correct usage of variables and APIs. Type errors such as variable or function misuse can be avoided with explicit type annotations. In this work, we introduce FlexType, an IDE extension that can be used on both JavaScript and TypeScript to infer types in an interactive or automatic fashion. We perform experiments with FlexType in JavaScript to determine how many types FlexType could resolve if it were to be used to migrate top JavaScript projects to TypeScript. FlexType is able to annotate 56.69% of all types with high precision and confidence including native and imported types from modules. In addition to the automatic inference, we believe the interactive Visual Studio Code extension is inherently useful in both TypeScript and JavaScript especially when resolving types is taxing for the developer.

The source code is available at GitHub[1] and a video demonstration at https://youtu.be/4dPV05BWA8A.

CCS CONCEPTS

• **Computing methodologies → Machine learning**; • **Theory of computation →** *Type structures*; • **Software and its engineering → Integrated and visual development environments**; Source code generation.

KEYWORDS

optional typing, type systems, type inference, deep learning

ACM Reference Format:
Sivani Voruganti, Kevin Jesse, and Premkumar T. Devanbu. 2022. FlexType: A Plug-and-Play Framework for Type Inference Models. In *Proceedings of ASE '22: 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Type inference for dynamically typed programming languages, like Python and TypeScript, can help developers improve code quality. By foregoing type annotations, developers coding in dynamically typed languages gain additional flexibility. This flexibility helps developers and designers avoid committing to particular design decisions regarding types. On the other hand, static typing helps detect bugs before execution, and supports both compilation performance and program understanding [7, 30]. Developers have viewed the benefits of static typing as the most desired feature in languages like Python [19]. Leading technology companies have developed their own type systems for various languages; Microsoft's TypeScript, Facebook's Flow, and Google's Closure with TypeScript and Flow being syntactic supersets of JavaScript and Python respectively. TypeScript has exploded in popularity over the last few years jumping to the fourth most used language according to GitHub's Octoverse [12] in 2020 and 2021. While JavaScript remains the top language, it is a reasonable expectation for TypeScript to further increase in popularity since it can be applied to any JavaScript project with few modifications. TypeScript inherits JavaScript's long standing popularity and widespread adoption so tools built for TypeScript often benefit the JavaScript community as well.

Unlike JavaScript, TypeScript calls for a set of types (either explicitly annotated or inferred) that type the program consistently. Defining a set of types and annotating with said types is not a trivial task for developers; this is called the *type annotation tax*. Type declaration files (.d.ts) and repositories like DefinitelyTyped[2] help alleviate the typing cost by defining general, high quality types which are included automatically by the compiler. The convenience of importing existing types does not supplant the action of annotating the code elements. Moreover, the compiler cannot synthesize types where static constraints or dependencies are not satisfied in the type dependency graph. Frequently, existing tools like TypeScript's type checker are unable to infer types more specific than the generic "any" because it fails to find type hints from static type constraints or package dependencies. Type ambiguity often exists in dynamic typing, because the compiler has too few type constraints to resolve [8]. Type ambiguity is more prevalent in languages like JavaScript, than in explicitly typed languages like TypeScript, where developers have no explicit annotations and must rely on interpretation, documentation, and surrounding expressions to determine the likely types. Thus, developers could benefit from tools that recommend likely types and insert types with little to no effort.

For these reasons, the *type inference* task has been well studied, in the software engineering research community [3, 14, 18, 26, 27, 29, 31, 33, 37]. Most of these works in type inference are a result of the abundance of code and the success of deep learning for software engineering. The abundance of patterns in code warrants probabilistic models to exploit the regularity of code; in type inference it is the regularity of how types are used. The newest advances in machine learning [10, 22–24, 32] often come with downstream

[†]Equal Contribution
[1]https://github.com/vsiv16/typescriptsuggestions
[2]https://github.com/DefinitelyTyped/DefinitelyTyped

ASE 2022, October 10-14, 2022, Ann Arbor, MI, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

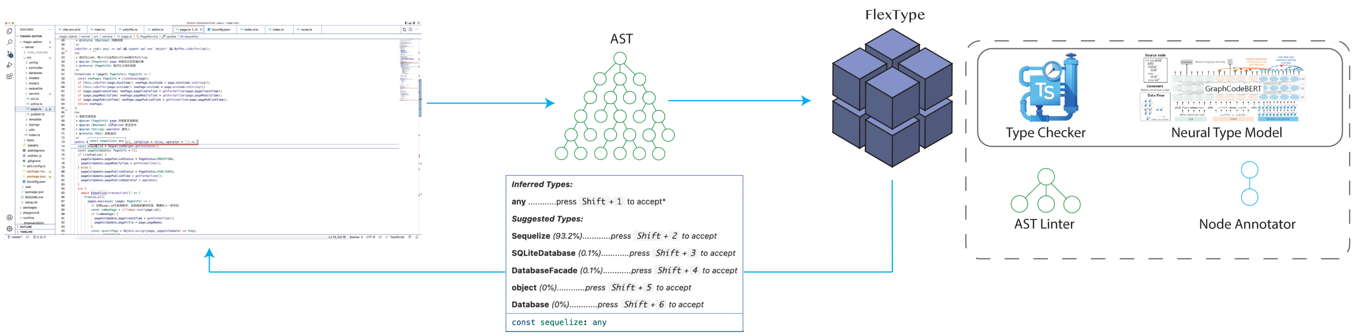


Figure 1: An overview workflow of the FlexType framework. To determine the type, the framework parses JavaScript or TypeScript ASTs and passes AST or token information to type checker and open source type inference neural model. The type is converted to a type node and added to the type attribute in the AST. Finally, FlexType converts the AST to a token sequence for the IDE.

improvements to software engineering models [1, 4, 11, 13, 21, 36], but in practice, these improvements have not been tangible to developers as most published models stop short of publishing IDE tools. We argue that the gap between model development and model deployment in integrated development environments is worthwhile, but challenging.

To address this gap, in this paper, we present our tool FlexType, a plug-and-play framework for any new state-of-the-art type inference model in a VSCode environment for TypeScript *and* JavaScript. JetBrains found that 60% of JavaScript and TypeScript developers use Visual Studio or Visual Studio Code as their preferred IDE [20]. The core idea behind FlexType is the integration of such models in an interactive and automatic way that complements existing static type checking capabilities, even in dynamically typed languages like JavaScript. To evaluate our idea, we have implemented an extension for Visual Studio Code, a popular IDE from Microsoft, using one of the of several type inference models from ManyTypes4TypeScript [17]. Our contributions are as follows,

- An interactive, model-agnostic framework for type inference in Visual Studio Code.
- A tool that uses the AST to correctly insert type elements from sequence-based or graph-based models.
- A use-case experiment evaluating the effectiveness of FlexType in migrating JavaScript projects to TypeScript.

2 RELATED WORK

The landscape of type completion tools ranges significantly in capability from static checking [6], neural type inference [3, 14, 17, 18, 26, 31, 37], and code completion-like type generation [1, 5, 9, 35, 36, 38].

Static type checking from the TypeScript compiler occurs when the TypeScript compiler transpiles TypeScript to JavaScript. The TypeScript type checker can be accessed through a shipped version of TypeScript installed with the IDE. The IntelliSense feature in Visual Studio and Visual Studio Code can provide underlying types by relying on the internal type checker for TypeScript. The type checker is capable of performing type inference from the variable's value as long as the type constraints exist. For example, the variable `i` in `var i = 0` can be inferred as a number from the value in the assignment expression. Any high-level interpretation of `i`, such as the use of `i` as an iterator, cannot be inferred by the type checker without a higher order type indicating such functionality. In JavaScript, the IntelliSense method signature information shows the uninformative "any" type for the method parameters because JavaScript is dynamic and does not enforce types [25]; this is not particularly helpful for a developer wishing to pass the correct type to the function.

Neural type inference and code completion aim to model attributes of source code probabilistically by exploiting the regularity of software [2, 16] and an abundance of existing typed code on open source repositories. In contrast to static type inference, neural type models rely on large code corpora and can suggest richer, more contextualized type annotations overcoming the lack of existing type constraints realized when the compiler predicts "any"; in our experiments this occurs 63.46% of all typeable identifiers. Our goal here is to build a flexible way to integrate neural type inference models into an IDE, to make these models more accessible.

Some published neural type inference models Typilus [3], HiTyper [29], and LambdaNet [37] expose inference methods where the model can be called on a set of source code files and the appropriate annotations are logged in an output file; this is impractical for the typical developer and such models often require computing not found on a laptop. One model cites the need of a "high-end Nvidia GPU with at least 8GB of RAM" and "a CPU with 16 threads or higher" [26]. The requirements for running massive code generation models like Codex [9] (Copilot), Google's 137B parameter model [5], and PolyCoder [38] is further beyond any consumer PC, thus access to such models must be by remote API. Remote API access is viable for many developers, but communicating large token windows of proprietary software introduces valid security and privacy concerns [9, 28, 34]; a local type inference model is ideal. In contrast, FlexType uses local models that can run efficiently on a laptop CPU. The user can simply hover over the variable, parameter, function or method to get a drop down list of types including the compiler inferred type, if any, and see the type properly inserted.

In the following sections, we present our approach, implementation, and evaluation of FlexType.

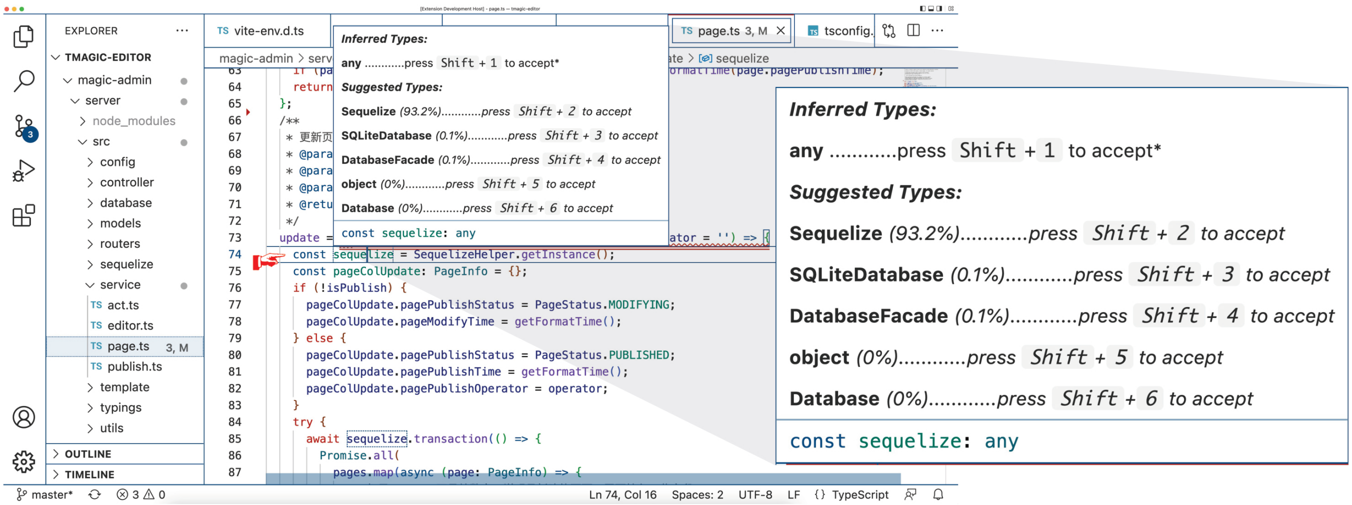


Figure 2: A snapshot of the FlexType VSCode extension.

3 APPROACH

Figure 1 shows how FlexType interactively works with the developer to recommend types. When the developer toggles the VSCode extension, FlexType activates the mouse hover action which pops up a list of types. By default, VSCode provides existing prototype information with type annotations that are written in the code such as `const sequelize: any` in Figure 2. FlexType presents an informative list to the developer integrating *compiler inferred types* (often useful for native types and user-defined types) with *the neural type suggestions*. The neural type suggestions can be quite useful to the developer, because the type recommendations derive from large corpora training, and elucidate types that local constraints often cannot resolve.

Figure 2 illustrates a key situation where the type assistant shines. With the current type constraints, the compiler cannot resolve what type `sequelize` is. The term "sequelize" in itself is a natural language hint, one that hints at it connecting to a SQL database often pronounced "see-kwl". While these natural language hints are not always readily available, the syntax and usage of function calls are, which deep learning models capture. The resulting list of contextually derived types, as seen in Figure 2, is helpful in understanding the likely functionality of such APIs. The developer has the liberty to choose which type annotations are useful with the model's perceived probabilities. This feature is available for TypeScript *and* JavaScript files as TypeScript data transpiles into JavaScript code and thus captures otherwise implicit type information in JavaScript. JavaScript syntax does not permit types, so types are not "insertable" when interacting with JavaScript. We believe FlexType can help both TypeScript and JavaScript developers, as type information improves code readability, comprehension, and proper usage of code elements. In the following text, we discuss the details of the approach within the framework's pipeline.

FlexType starts with an incremental compilation[3] of the program, targeting just the current editor file for AST parsing. Then the framework digests the developers current word token index[4] and finds the character offset of the token which aligns best with the pos (position) field in the parsed AST. FlexType uses an AST linter to traverse the AST in *preorder*, filtering only valid typeable locations. For each leaf node (indicating a code token) the corresponding code token is appended to a list of tokens which will serve as the tokenized input to the machine learning model; tokenizing from the AST has the benefit of filtering out non-code related tokens such as comment blocks. In the traversal, the framework keeps a cache of the parent type because the parent node is where the type annotation is located, specifically, in a variable, parameter, function, or method declaration syntax node. Finally, when the identifier syntax node corresponding to the identifier of interest is visited, which is a child to the typed parent, this token is aligned to the cached AST type and to the current token index. The cached type node is fed to the type checker which returns the result of any the static type constraints, if any, for that identifier. Finally, the token sequence, inferred type, and token index is returned. If the developer's cursor location is not at a typeable variable, parameter, function, or method declaration, the AST linter is immediately returned with null values.

The token sequence and token index is passed through a localhost port to a WSGI Flask[5] server started as a background task when the extension is enabled. This server encapsulates the neural inference model. The token sequence is subtokenized using the neural model's tokenizer and the new subtoken index is calculated. The framework then determines an optimal context window around the identifier of interest; this is necessary for long files as a model's sequence-based input is limited. The type inference model in our demo, is a Huggingface type inference model based on the popular GraphCodeBert [13]. Here, we emphasize the "plug-and-play" dynamic where neural type inference models, such as our `from_pretrained('microsoft/graphcodebert-base')`, is

[3] An incremental compilation saves compute resources when previous changes are minimal across a set of files and project dependencies.

[4] The term *position* is usually synonymous for token indexes in sequences across NLP literature, but is confusing in the context of the AST, thus we only use it when referring to the AST.

[5] https://flask.palletsprojects.com/en/2.1.x/

amenable with alternative choices. With respect to future proofing our design, our GraphCodeBert [13] type inference model improves upon CodeBert, namely, where data flow awareness is principle to performance. For type inference, GraphCodeBert increases performance, likely due to the role data flow plays in types. Finally, these neural suggestions are serialized and returned to the VSCode portion of the framework where the types are displayed to the developer.

For a TypeScript (.ts) file, the framework presents the recommended types with keystrokes to embed the types as formal type annotations. For a JavaScript (.js) file, the framework shows the developer the type recommendations only. If the developer chooses a type, the framework performs a *postorder* AST traversal to return to the identifier's parent node, generate a type node from the type, and assign the type node to the parent's `type` field. The traversal is immediately returned, returning the root node of the sourcefile which is then used to synthesize the file with the type annotation in the correct location; this insertion technique is guaranteed by the compiler's printer to work for any valid type node. Since the type node's synthesis is independent of the actual type value, FlexType can always guarantee correct type placement. Finally, the VSCode editor is updated with the new type embedded sequence. In the next section, we discuss the high level implementation design.

4 IMPLEMENTATION

We implemented our approach as an extension to Microsoft's *Visual Studio Code*, Figure 2, which is the most adopted TypeScript/-JavaScript IDE according to a JetBrains survey [20]. We implement the client in TypeScript where VSCode can pass actions such as hover, click and drag, and keyboard strokes to the client. Upon a hover over a type permissive location (variable, parameter, function, method), FlexType performs static and neural type inference and recommends types. The modularity of the static type checker and the neural type model permits the interchange of a variety of models with minimal changes. While sequence-based methods (RNN, Transformer, Pretrained Language Models) are very popular, there is an increasing demand for models that capture code structure (GNN [4], Hybrid [15]). In addition to the "plug-and-play" neural type architecture, FlexType re-synthesizes the snippet of code with the TypeScript Compiler API[6]. By altering the AST, rather than the code sequence itself, the framework is compatible with graph-based methods. Finally, for best results, we apply graph optimization and quantization to the neural type inference model, which results in blazing quick inference times under .4 seconds on a Intel 8th generation Coffee Lake and even faster on Apple M1. In the next section, we perform an experiment to simulate the impact of our tool for developers migrating from JavaScript to TypeScript and equivalently coding only in JavaScript.

5 EVALUATION

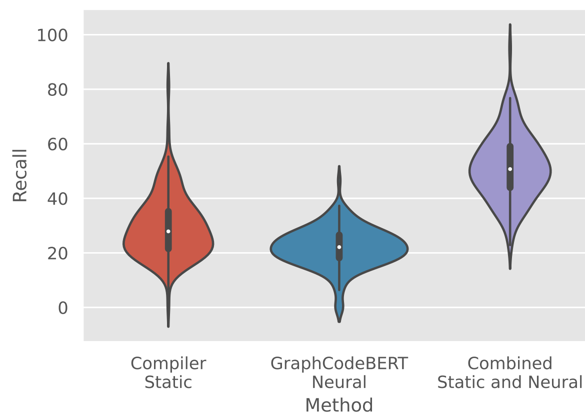
FlexType uses both type checker and neural type inference models. To evaluate FlexType's effectiveness migrating JavaScript to TypeScript, we checkout over 150 most-starred Javascript repositories and let FlexType annotate them as best as it can. For brevity, we have only included 12 of these projects in Table 1 with the full results available at our GitHub.

Table 1: Recall Percentage of Types Across Top 12 Projects

Repo	Stars	TC (%)	TC + NN (%)
goldbergyoni/nodebestpractices	77728	33.73	60.0
Dogfalo/materialize	38682	29.02	52.6
yangshun/front-end-interview-handbook	33963	19.83	45.69
quilljs/quill	32667	26.28	55.01
marktext/marktext	31921	33.1	56.63
MostlyAdequate/mostly-adequate-guide	21732	17.69	30.2
Kong/insomnia	20947	34.81	55.0
pugjs/pug	20753	26.46	53.32
wekan/wekan	17988	28.52	49.42
louislam/uptime-kuma	17288	31.25	56.65
mysqljs/mysql	17177	15.6	43.24
alsotang/node-lessons	16408	22.73	52.44

FlexType recall using only the static type checker (TC) and FlexType using both the type checker and neural type inference model (TC+NN).

Figure 3: Static, neural, and combined recall of FlexType components per project.



The 150 JS projects have no human annotated types, so performance must be evaluated with an oracle. The neural type model *per se* can serve as an oracle if it's confidence threshold is set such that precision remains very high; only if a type prediction is above this threshold can it be labeled as correct. To calibrate, we measure the precision-recall curve of GraphCodeBERT on the ManyTypes4TypeScript [17] test set. This is a dataset of manually-annotated Typescript projects allowing direct performance evaluation. GraphCodeBERT achieves a precision of 89.10% and recall of 53.83% across 224,415 types with a 90% confidence threshold. Thus, we can use GraphCodeBERT's confidence threshold with a precision of 89.10% as a proxy to the number of types that can be resolved. In other words, 89.10% of JavaScript types with a confidence of 90% or greater is a reasonable metric for evaluation. While this method is effective, it is important for us to calculate how much we are potentially underestimating our model's performance.

The recall of 53.83% means that 46.17% of types fall below the confidence threshold. We can calculate the precision across the 46.17% set of types to determine how many types were missed. This precision is 31.51% and so the model's recall is underestimated at most by 15% (31.51% of 46.17%).

We emphasize that this performance is for the top-1 (the model's best guess), and ignores selecting the 2nd or 3rd best choice in the

[6] https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

interactive dialog seen in Figure 2. In the interactive setting with 5 choices, the recall is naturally higher than in the top-1 setting. We use top-1 in our automatic evaluation of FlexType to estimate a lower bound of performance in a common use case, migrating JavaScript to TypeScript.

RQ1: What is the recall of types for FlexType across the top 150 starred JavaScript projects?

Across the set of 150 projects, 56.69% of types are resolved by FlexType. The recall of the compiler is 36.54% and the neural type inference model provides the additional 20.15% recall. On a per project evaluation, the mean project recall is 51.44% with the compiler providing 29.49% of the types and the neural type inference model providing an additional 21.95% recall. The per project recall distribution of each component can be seen in Figure 3.

This evaluation suggests that FlexType helps annotate a good fraction of type locations (56.69%) in JavaScript; this reduces the annotation burden in JavaScript to TypeScript migration. Moreover, we argue that developers will use the tool in an interactive fashion using the drop down menu in Figure 2. This should further increase the recall which represents the number of type constraints the developer could reasonably add with minimal effort.

6 CONCLUSION

As a development tool, FlexType can help increase the volume of type annotations. We also see an opportunity to use FlexType in an automated setting to improve type annotation coverage in existing and new projects. We hope the adoption of this framework can reduce the burden of adding type annotations in TypeScript and the reduce the misuse of variables and APIs in both TypeScript and JavaScript, thus improving software development and maintenance.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*. 91–105.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[6] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*.

[7] Luca Cardelli. 1996. Type systems. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 263–264.

[8] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Github. 2021. The 2021 state of the octoverse, 2021. (2021). https://octoverse.github.com

[13] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[14] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 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*. 152–162.

[15] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.

[16] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[17] Kevin Jesse and Premkumar T Devanbu. 2022. ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference. (2022).

[18] Kevin Jesse, Premkumar T Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1483–1486.

[19] JetBrains and Contributors. 2020. Python developer survey conducted by Jetbrains and python software foundation, 2020. (2020). https://www.jetbrains.com/lp/python-developers-survey-2020

[20] JetBrains and Contributors. 2020. TypeScript developer survey conducted by Jetbrains, 2020. (2020). https://www.jetbrains.com/lp/devecosystem-2020/javascript

[21] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[22] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[23] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[24] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[25] Microsoft and Contributors. 2022. IntelliSense, 2022. (2022). https://code.visualstudio.com/docs/editor/intellisense

[26] Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2021. Type4py: Deep similarity learning-based type inference for python. *arXiv preprint arXiv:2101.04470* (2021).

[27] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. 2020. Opttyper: Probabilistic type inference by optimising logical and natural constraints. *arXiv preprint arXiv:2004.00348* (2020).

[28] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *arXiv e-prints* (2021), arXiv–2108.

[29] Yun Peng, Zongjie Li, Cuiyun Gao, Bowei Gao, David Lo, and Michael Lyu. 2021. HiTyper: A Hybrid Static Type Inference Framework with Neural Prediction. *arXiv preprint arXiv:2105.03595* (2021).

[30] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[31] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 209–220.

[32] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[33] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[34] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning. *Proceedings of the ACM Web Conference 2022* (2022).

[35] Lewis Tunstall, Leandro von Werra, Thomas Wolf, and Aurélien Géron. 2022. *Natural language processing with transformers: Building language applications with hugging face*. O'Reilly Media.

[36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[37] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. Lambdanet: Probabilistic type inference using graph neural networks. *arXiv preprint arXiv:2005.02161* (2020).

[38] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *arXiv preprint arXiv:2202.13169* (2022).