Predicting TypeScript Type Annotations and Definitions With Machine Learning

Thesis Proposal

MING-HO YEE, Northeastern University, USA

Type information is useful for developing large-scale software systems. Types help prevent bugs, but may be inflexible and hamper quick iteration on early prototypes. TypeScript, a syntactic superset of JavaScript, brings the best of both worlds, allowing programmers to freely mix statically and dynamically typed code, and choose the level of type safety they wish to opt into. However, *type migration*, the process of migrating an untyped program to a typed version, has remained a labour-intensive manual effort in practice. As a first step towards automated effective type migration, there has been interest in applying machine learning to the narrower problem of *type prediction*.

In my thesis, I propose to use machine learning to partially migrate JavaScript programs to TypeScript, by *predicting type annotations* and *generating type definitions*. To support my thesis, I make three contributions. First, I propose evaluating type prediction by type checking the generated annotations instead of computing accuracy. Second, I fine-tune a large language model with fill-in-the-middle capability to *fill-in-the-type* and predict type annotations. Finally, I use a similar approach to fine-tune a large language model to generate missing type definitions.

1 INTRODUCTION

Type information is useful for developing large-scale software systems. Types help prevent bugs, provide documentation, and improve support for editors and development tools. At the same time, types can be inflexible and may hamper quick iteration on early prototypes. Gradual typing brings the best of both worlds, allowing programmers to freely mix statically and dynamically typed code, and choose the level of type safety they wish to opt into [31, 66, 70, 71]. This makes it possible to incrementally add static types to a large program without requiring a complete rewrite of an existing codebase at once. As a result, gradual typing has proliferated over the past decade, and there are now gradually typed versions of several mainstream programming languages [9, 10, 16, 19, 45, 47, 54, 70, 72, 81].

One success story is TypeScript, a syntactic superset of JavaScript that supports optional type annotations [9]. Both languages are very popular: JavaScript is listed as the top programming language on GitHub and Stack Overflow, while TypeScript ranks in the top five [30, 68]. Programmers can write their code in TypeScript, benefit from static typing, and then compile to JavaScript. However, *type migration*, the process of migrating an untyped JavaScript program to TypeScript, has remained a labour-intensive manual effort in practice. For example, Airbnb engineers took more than two years to migrate 6 million lines of JavaScript [63], and there are several other accounts of multi-year TypeScript migration efforts [6, 11, 52, 56, 62].

As a first step towards effective automated type migration, there has been significant research interest in using machine learning to attack the narrower problem of *type prediction*. Compared to type migration, which may involve refactoring code that is not well typed, the objective of type prediction is to maximize the likelihood of a correct type prediction given a code fragment [1, 32, 37, 39, 46, 50, 55, 57, 58, 60, 74, 76, 79]. Type prediction is appealing because machine learning models can take into account the linguistic context of the code fragment. Furthermore, there is a significant quantity of high-quality, open-source JavaScript and TypeScript code that is available to serve as training data [35, 38, 42, 49, 75]. In particular, *large language models* (LLMs) are successful at a variety of code generation tasks [4, 5, 8, 14, 20, 21, 36, 53, 75], and recent work presents

Author's address: Ming-Ho Yee, Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA.

fill-in-the-middle (FIM) inference in which the model learns editing tasks while still performing left-to-right token generation [7, 26].

However, there are limitations in existing work, which I aim to address in my dissertation research:

1. Machine learning models for type prediction are typically evaluated on accuracy, which is the proportion of type predictions that are correct. However, calculating accuracy requires a ground truth of existing type annotations, which is not available when migrating JavaScript code, and accuracy says nothing about whether the migrated code will type check. I proposed *type checking* the migrated programs [79], and as part of this work I created an evaluation dataset and built TypeWeaver, a system for evaluating type prediction systems.
2. Large language models with fill-in-the-middle capabilities have neither been trained for nor evaluated on the type prediction task, other than small-scale evaluations [26, 44]. I created an improved evaluation dataset and worked on a new fine-tuning approach called *fill-in-the-type* that adapts fill-in-the-middle for type prediction [15].
3. The related problem of *generating type definitions* has not been studied. This is particularly relevant when migrating JavaScript to TypeScript, as TypeScript has a structural type system and the migrated code may refer to types that need to be defined. Working on this problem is the last piece of my thesis work and is ongoing: to generate *type definitions* in addition to predicting type annotations for untyped JavaScript programs.

In this document, I state my proposed thesis (section 2) and describe how my completed and future work supports that thesis; namely, evaluating type prediction (section 3), training and prompting for type prediction (section 4), and generating type definitions (section 5). I conclude with a proposed schedule (section 6) and a discussion of related work (section 7).

2 THESIS

My dissertation research has focused on how machine learning can be used to migrate JavaScript programs to TypeScript. However, the scope of that problem is too large for a single dissertation, so I focus on the narrower problems of type annotation prediction and type definition generation. Therefore, my thesis is:

> *Machine learning can be used to partially migrate JavaScript programs to TypeScript, by predicting type annotations and generating type definitions.*

Below, I elaborate on each component of the thesis statement.

Machine learning. My work uses machine learning models, specifically large language models, and in particular, open-source models such as SantaCoder [8] and StarCoder [44].

Partial migration. I do not believe it is currently possible to *fully* migrate a JavaScript program to TypeScript using machine learning. Therefore, I focus on aspects of migration, and acknowledge that some manual refactoring may be required.

JavaScript to TypeScript. I restrict my work to JavaScript and TypeScript, which are listed within the top five programming languages on GitHub and Stack Overflow [30, 68]. Handling a popular programming language involves challenges not present for smaller languages like the gradually typed lambda calculus and its extensions.

Predicting type annotations and generating type definitions. My dissertation studies these two specific tasks, which are part of type migration.

To support my thesis, I make three contributions:

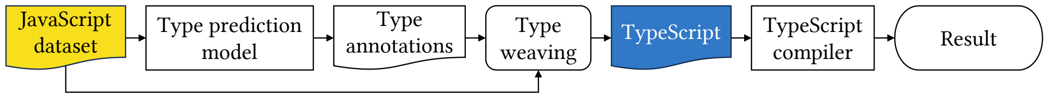

Fig. 1. TypeWeaver workflow: a JavaScript dataset is given to a type prediction model, which returns type annotations; next, type weaving merges the type annotations with the original JavaScript code and produces TypeScript; finally, the TypeScript is type checked by the TypeScript compiler.

1. Before working on the problem, it is important to define how it will be measured. In Yee and Guha [79], which was published at ECOOP 2023, I proposed evaluating type prediction systems by *type checking* the generated types, rather than using the typical metric of accuracy.
2. I approach the task of predicting type annotations in Cassano et al. [15], which was submitted to NeurIPS 2023. In this paper, we adapt the *fill-in-the-middle* training technique to fine-tune a model that fills in type annotations.
3. Finally, taking lessons from the previous two papers, I propose work on the final part of my thesis, which is using large language models to generate type definitions.

3 EVALUATING TYPE PREDICTION MODELS

In prior work, the usual evaluation criteria for the type annotation prediction task has been accuracy: what is the likelihood that a predicted type annotation is correct? Correct means the prediction *exactly* matches the ground truth, which is the handwritten type annotation at that location. Accuracy is typically measured as top-k accuracy, where a prediction is deemed correct if any of the top k most probable predictions is correct. Thus, a top-1 accuracy is the likelihood that the top prediction is correct.

However, in Yee and Guha [79], I argued that accuracy is not the right metric for evaluating a type prediction model and can be misleading. First, when migrating a JavaScript project to TypeScript, there is frequently no ground truth of handwritten type annotations. Second, equivalent annotations that are not exact textual matches (e.g., the union types `number | string` vs. `string | number`) are counted as incorrect. Third, the predicted type annotations may not type check, requiring the programmer to manually resolve type errors.

As a first step towards better evaluation criteria, I proposed *type checking* the predicted type annotations. This condition is much stronger than accuracy, as even a single, incorrect type annotation causes a package to fail to type check, and too many errors may overwhelm a user who may just turn off the tool. Moreover, it may not be feasible to fix the type errors automatically, since type errors refer to code locations whose typed terms are used, and not necessarily to faulty annotations. On the other hand, type checking may allow less precise type annotations that are trivial and unhelpful to a user, e.g. `any`.

3.1 Approach

I built TypeWeaver, a framework for evaluating TypeScript type prediction models [80], and evaluated three systems: DeepTyper, a bidirectional recurrent neural network [32]; LambdaNet, a graph neural network [74]; and InCoder, a general-purpose, multi-language transformer that supports fill-in-the-middle tasks [26].

The workflow for TypeWeaver is illustrated in Figure 1. First, the process starts with a JavaScript dataset that I created. Second, the JavaScript code is given to one of the supported type prediction models: DeepTyper, LambdaNet, and InCoder. Finally, the TypeScript code is type checked by the TypeScript compiler and the results are recorded.

3.1.1 Dataset. I created a JavaScript dataset of 513 packages, taken from the top 1,000 most downloaded packages from the npm Registry. This was because existing datasets for type prediction, including ManyTypes4TypeScript [38], were in TypeScript, not JavaScript, and I believe the evaluation should reflect what would occur in practice, i.e., migrating from an untyped JavaScript codebase to TypeScript. Additionally, ManyTypes4TypeScript is given as pairs of tokens and labels, which is not suitable for a type checking metric.

In constructing my dataset, I ensured that every package has *no dependencies*, or that *all its dependencies are typed*, meaning the dependencies have TypeScript type declaration (`.d.ts`) files available. This requirement is necessary because a JavaScript package can only be imported into a TypeScript project if its interface has TypeScript type declarations. The DefinitelyTyped repository[1] contains interface type declarations for many popular JavaScript packages, and a handful of packages include their own. I collected the type declarations of project dependencies and include them in the dataset.

Next, I converted projects from CommonJS module notation to ECMAScript module notation. This step is important for the type checking evaluation, as most packages use the CommonJS module system, but CommonJS modules in TypeScript are untyped. Therefore, even if a module has type annotations for the variables and functions it exports, those annotations are lost when the module is imported. On the other hand, ECMAScript modules preserve type information across module boundaries. Thus, I used the `cjs-to-es6` tool[2] off-the-shelf to transform the dataset to use ECMAScript modules.

3.1.2 Type Prediction. Given JavaScript input, DeepTyper and LambdaNet output type annotations: DeepTyper predicts types for all identifiers in the program, including at program locations that do not allow type annotations, while LambdaNet predicts types for variable and function declarations, but only in the correct locations. However, an additional step I call *type weaving* is required to filter and merge the type annotations with the original JavaScript, in order to produce valid TypeScript code. On the other hand, InCoder outputs TypeScript directly, but it requires a type prediction front end that we implemented. Currently, our front end only supports type predictions for function parameters.

Type weaving. To produce type-annotated TypeScript code, I use a process called *type weaving* to combine type predictions with the original JavaScript code. Type weaving takes two files as input: a JavaScript source file and an associated comma-separated values (CSV) with type predictions. The type weaving program parses the JavaScript source into an abstract syntax tree (AST) and then traverses the AST and CSV file simultaneously, using the TypeScript compiler to insert type annotations into the program AST. To my knowledge, this is the first time such a tool has been implemented for DeepTyper or LambdaNet, meaning that this is the first time those tools have been used to output TypeScript, which can then be type checked.

Type prediction front end. InCoder is trained to generate code in the middle of a program, conditioned on the surrounding code. This is useful for type prediction, as “holes” can be inserted at type annotation sites. However, this required implementing a type prediction front end, which transforms the JavaScript input into the appropriate format for InCoder to generate type annotations. Furthermore, we observed that InCoder frequently generates more than just a single type, so our front end also parses the generated response from InCoder and extracts the first syntactically valid type.

---

[1] https://github.com/DefinitelyTyped/DefinitelyTyped/

[2] https://github.com/nolanlawson/cjs-to-es6

Table 1. Number and percentage of packages and files that type check.

	DeepTyper			LambdaNet			InCoder		
	✓	Total	%	✓	Total	%	✓	Total	%
Packages	95	460	20.7	39	439	8.9	94	493	19.1
Files	746	1,720	43.4	731	2,905	25.2	2,569	3,710	69.2

Table 2. Number and percentage of type annotations that are `any`, `any[]`, or `Function`.

	DeepTyper			LambdaNet			InCoder		
	Trivial	Total	%	Trivial	Total	%	Trivial	Total	%
Annotations	2,045	3,435	59.5	580	2,363	24.5	987	2,418	40.8

3.1.3 Type Checking. The final step is to type check to generated TypeScript code. I use the TypeScript compiler, configured to accept all TypeScript input files as arguments, and with all package dependencies resolvable by the compiler. I save all results and compiler messages for further analysis.

3.2 Results

The first question to ask is *do migrated packages type check*? However, not all packages successfully translate to TypeScript with every type migration tool; some packages cause the tool to time out or error. Thus, I report the success rate of type checking as a fraction of the packages that successfully translate to TypeScript. The "packages" row of Table 1 summarizes the result: DeepTyper and InCoder perform similarly, with about 20.7% and 19.1% of packages that type check, while only 8.9% of packages migrated by LambdaNet type check.

However, requiring an entire package to type check is a very high standard to meet. Even a single incorrect type annotation causes the entire package to fail to type check. Therefore, as a more fine-grained metric, the second question asks *do migrated files type check*? The "files" row of Table 1 summarizes the result: now InCoder performs the best with 69.2% of its files type checking successfully, whereas the results are 43.4% for DeepTyper and 25.2% for LambdaNet.

One limitation of type checking as a metric is that trivial type annotations, such as, `any`, `any[]` (array of `any`s), and `Function` (a function that accepts any number of arguments of any type and returns `any`), can hide type errors and allow more code to type check, but provide little value to a programmer. Therefore, *within the files that type check*, I count the number of type annotations that are `any`, `any[]`, or `Function`. Table 2 summarizes the result: DeepTyper produces the most (59.5%), LambdaNet produces the fewest (24.5%), while InCoder is in between (40.1%).

3.3 Takeaways

There are several lessons and takeaways from this work:

- I argue that when evaluating type prediction systems, *type checking* is an improvement over accuracy; however, type checking has its own limitations. For example, requiring an entire package to type check is significant burden, so it may be better to type check individual files. Additionally, trivial type annotations like `any` can hide type errors and allow more code to type check, but are imprecise and provide little value to a programmer.

```
1 function f(x) {
2   return x + 1;
3 }
```

(a) Input function

```
4 function f(x: number, y: number) {
5   return x + 1;
6 }
```

(b) The highlighted "type annotation" is filled in

Fig. 2. A example where fill-in-the-middle generates additional code.

- InCoder was trained to infill code that typically spans multiple lines; as a result, InCoder frequently generates more than a single type. I believe that InCoder could be improved if it was fine-tuned specifically for type prediction.
- The dataset could be improved: I observed packages that were trivially typable, either because there were very few declarations to annotate, or because the type annotations were mostly primitive types. Additionally, there are also JavaScript files that cannot be typed in TypeScript without some amount of refactoring. Finally, it may be worthwhile to consider the run-time behaviour of migrated programs, since it is possible for a package to type check but still have run-time errors.

I address some of these issues in a follow-up paper with Cassano et al. [15].

3.4 Publication Status

This work was submitted to, accepted at, and presented at ECOOP 2023, the European Conference on Object-Oriented Programming [79]. Additionally, the artifact was submitted, evaluated, and awarded the Available and Reusable badges [80].

4 TRAINING TYPE PREDICTION MODELS

Large language models are successful at a variety of code generation tasks, and *fill-in-the-middle* is a natural fit for type prediction. However, in Cassano et al. [15], we find several challenges that prevent these models from working out-of-the-box. First, fill-in-the-middle models are trained to infill code that typically spans multiple lines, which inhibits their ability to infer end tokens after short token sequences such as type annotations. Second, models generally do not understand the implicit type constraints within a program, which produces programs that may not type check [60, 79]. These errors are tedious for human programmers to manually resolve. Third, entire programs are often very large and may not fit within a context window. This problem exists more broadly in code generation models, and even more broadly in almost every transformer-based language model. Even in emerging models with larger context windows, the relevant context for an arbitrary type may be spread over long sequences within a program. This problem becomes more apparent in larger context models that trade adequate attention for performance [65, 69].

Why can't a language model solve this? There are two problems specific to language models that we must address. First, a language model can only accept a limited number of tokens as input. Tokens are the basic units of input and output for a language model, and may not necessarily correspond to lexical tokens of a program. This token limit is called the *context window*, and programs can be arbitrarily large and therefore cannot fit within a context window. Second, fill-in-the-middle can generate unwanted code. For example, Figure 2a shows a single-parameter function that is given to a model; however, the model produces the result in Figure 2b, where `x` is correctly annotated as `number`, but an additional parameter `y` is generated.

4.1 Approach

Our work has four contributions: a new evaluation methodology that measures program *typedness*, the degree to which migrated programs contain type information; OpenTau, an implementation that takes a *tree-based program decomposition* approach to fit large programs within context windows; a *fill-in-the-type* fine-tuning approach based on fill-in-the-middle; and an evaluation on a new dataset of TypeScript files.

4.1.1 Program Typedness. In previous work, I proposed type checking the program instead of using accuracy as a metric [79]. However, one of the lessons of that work is that trivial type annotations (e.g., `any`) will always type check, but provide little benefit to the programmer.

We would like a metric that captures type information but is also amenable to type checking and does not require ground truth data. As a first step, we propose a *typedness* metric that measures the degree to which a program contains type information. Intuitively, this rewards type annotations that are informative but restrictive, which allow the type checker to catch more errors.

Table 3. Score for each type encountered. A type that is not in the table is scored as 0.

Type annotation	Score
`unknown`	1.0
`any` (or missing)	0.5
`Function`	0.5
`undefined`	0.2
`null`	0.2

To compute the typedness score of a program, we count the number of undesirable type annotations, assign a score to each annotation as specified in Table 3, sum the scores, and normalize the score by the number of types encountered. The score is normalized to a number between 0 and 1000, where lower scores are preferred. The typedness metric counts only *leaf* types in an abstract syntax tree, e.g., `Array<any>` is scored as 0.5, since `any` is the type argument.

4.1.2 Tree-Based Program Decomposition. Programs are hierarchical in structure: the top-level code block contains declarations and each declaration creates a code block that may contain nested declarations, e.g., functions may contain nested functions and classes may contain methods. OpenTau reuses this structure for type prediction by representing the program as a tree, with the top level as the root node, declarations as non-root nodes, nested declarations as child nodes, and top-level variable declarations grouped into a single node under the root. OpenTau also ensures that comments appearing directly before a declaration are included in that declaration's node, as comments may contain additional context.

The tree representation also allows long-range context to be included in a node. For instance, if a node represents a function definition, OpenTau scans the parent node's code block for statements that use that function. Then, it generates a comment containing usage information and prepends it to the node's declaration. Thus, the prompt to the model contains the full text of the node's function definition, as well as a comment containing usages of that function.

The tree representation also encodes dependencies between nodes: nested declarations must be fully typed before their enclosing declarations, so child nodes are visited before their parents. Additionally, a fully annotated child node provides context when predicting types for the parent node. This induces a bottom-up, level-order traversal that starts from the deepest level of the tree and finishes at the root. Then, at each node, OpenTau uses a large language model with fill-in-the-middle to predict type annotations.

At the end of the tree traversal, OpenTau may have generated multiple typings for the program. Each typing is type checked by the TypeScript compiler, and the number of type errors is logged. If there are no type errors, then the solution type checks. OpenTau additionally computes the typedness score for each typing. Finally, the typings are sorted by the number of type errors, with

ties broken by the typedness score. The best solution has the fewest type errors—ideally zero—but the most type information.

4.1.3 Fill-in-the-Type. We adapt the fill-in-the-middle training technique [7, 26] to fine-tune fill-in-the-type for TypeScript type prediction. We use SantaCoder as the base model, an open-source model with 1.1 billion parameters that was pre-trained on Python, JavaScript, and Java for left-to-right and fill-in-the-middle code generation [8]. We fine-tune SantaCoder using the TypeScript subset of the near-deduplicated version of The Stack, a dataset of permissively licensed source code [42].

The original fill-in-the-middle technique splits an input into prefix, middle, and suffix spans; however, our approach splits on *type annotation* location indices rather than arbitrary code sequences, and selects a type annotation as the middle span rather than a multi-line span of code. Furthermore, to closely resemble the context format that the model sees at inference time, we ensure type annotations are present in the prefix, but absent from the suffix 90% of the time, i.e., we allow type annotations to be present in the suffix 10% of the time to handle inputs that may be partially type annotated.

As a result, fill-in-the-type addresses a limitation we observed when using InCoder's fill-in-the-middle for type prediction [79].

4.1.4 Dataset. As part of the evaluation, I created a new dataset for evaluating type migration of TypeScript files. This new dataset satisfies certain properties: for instance, dataset files should not be trivially incorrect (e.g., syntactically invalid or requiring external modules) or trivial to migrate (e.g., files that are too short or have no type annotation locations). Focusing on files rather than packages (as I did for the TypeWeaver dataset [79]) makes it easier to use the dataset and avoids requiring an entire package to type check. Focusing on TypeScript rather than JavaScript avoids code that cannot be migrated without refactoring, but has the disadvantage of not reflecting the actual practice of migrating JavaScript to TypeScript. These choices address limitations I observed in prior work [79].

Table 4. Factors and their weights, used to compute a quality score for filtering the evaluation dataset.

Factor	Weight
Function annotations	0.25
Variable annotations	0.25
Type definitions	0.11
Dynamic features	0.01
Trivial type annotations	0.11
Predefined type annotations	0.05
Lines of code per function	0.11
Function usages	0.11

I constructed a dataset of 744 TypeScript files by filtering the near-deduplicated version of The Stack [42], which contains roughly 12.8 million TypeScript files. Filtering removes files that depend on external modules, do not type check, have no type annotation locations, have fewer than 50 lines of code, have no functions, or average fewer than five lines of code per function. These filtering steps reduce the dataset to 21,464 files.

Next, I computed a weighted quality score for each file, preferring files with (1) more function and parameter annotation sites, (2) more variable annotation sites, (3) more type definitions, (4) fewer instances of dynamic features (e.g., `eval`), (5) fewer trivial type annotations (e.g., `any`), (6) fewer predefined type annotations (e.g., `string`), (7) more lines of code per function, and (8) more function usages. The weights are shown in Table 4. After computing scores, I removed files that were one or more standard deviations below the mean score, leaving 17,254 files in the dataset.

Next, to minimize test-train overlap, I apply a December 31, 2021 cutoff, consistent with the training cutoff used for fine-tuning. This results in 867 files after the cutoff. Finally, I process the filtered, high-quality TypeScript dataset to remove type annotations. This procedure does not always succeed, so I discard files where it fails, resulting in the final evaluation dataset of 744 files.

Table 5. Experimental results of evaluating OpenTau. We measure *files that type check*, which is more rigorous than measuring individually correct type annotations.
All numbers are rounded to the nearest tenth.
TS = TypeScript; FIT = fill-in-the-type; ✓ denotes the number of files that type check.

		Type checks				Errors	
Model	Configuration	✓	Total	%	Typedness	Type	Syntax
TS	baseline, no parser	1	50	2.0	0.0	121.2	42.1
FIT	baseline, no parser	25	50	50.0	230.0	4.6	0.2
TS	baseline	245	744	32.9	200.7	4.7	0.0
FIT	baseline	297	744	39.9	200.9	5.2	0.0
FIT	OpenTau	353	744	**47.4**	**154.6**	**3.3**	0.0

4.2 Results

We evaluate OpenTau to determine the effectiveness of *fill-in-the-type* and its *tree-based program decomposition*, using four metrics: the percent of files that type check, the typedness score for files that type check, the average number of type errors, and the average number of syntax errors.

We compare two SantaCoder models: one that has been fine-tuned for TypeScript code generation (SantaCoder-TS), and one that has been fine-tuned for fill-in-the-type for TypeScript (SantaCoder-FIT). We compare OpenTau's program decomposition with a baseline that treats the entire file as a single tree node. We use a type parser to extract the first plausible type annotation returned by SantaCoder.

Table 5 shows our results. OpenTau significantly outperforms the baseline: 47.4% of files type check (14.5% absolute improvement) with a much lower typedness score. We discuss our experiments below.

Type parser. We conduct a small experiment that compares SantaCoder-TS and SantaCoder-FIT with the type parser disabled, on a random sample of 50 files from the dataset. Our results show that fill-in-the-type significantly helps with predicting syntactically valid type annotations, and is effective without the type parser: 50% of files type check with an average rate of 0.2 syntax errors per file, compared to 2% of files that type check and 42.1 syntax errors per file.

Fill-in-the-type. We repeat the experiment on the full dataset with the type parser enabled. SantaCoder-FIT outperforms SantaCoder-TS in the percentage of files that type check (32.9% vs. 39.9%), while maintaining a similar average typedness score.

Tree-based program decomposition. Finally, we compare OpenTau's program decomposition to the baseline, and show that it outperforms the baseline in all metrics. In particular, the typedness score is much lower, suggesting that OpenTau is successful in searching for more precise type annotations.

4.3 Takeaways

Our work in Cassano et al. [15] builds on Yee and Guha [79] in several respects:

- We introduced a *program typedness* metric that measures the degree to which a program contains type information. This metric can be used alongside a type checking metric while quantifying the type information available, i.e., trivial type annotations like `any` are penalized.

```
7  function dist(p1, p2) {
8    const dx = p2.x - p1.x;
9    const dy = p2.y - p1.y;
10   return Math.sqrt(dx*dx + dy*dy);
11 }
```

(a) JavaScript

```
12 function dist(p1: Point, p2: Point) {
13   const dx = p2.x - p1.x;
14   const dy = p2.y - p1.y;
15   return Math.sqrt(dx*dx + dy*dy);
16 }
```

(b) TypeScript

Fig. 3. A function that computes the distance between two points.

- We designed a new *fill-in-the-type* training technique that adapts fill-in-the-middle for TypeScript type prediction. This significantly improves the ability of a large language model to fill in a syntactically valid type annotation, rather than a multi-line span of code.
- I created an improved dataset that excludes files that are trivially incorrect or depend on external modules, and prioritizes files that are not trivial to migrate. Furthermore, the dataset is based on individual files rather than packages, which makes the dataset more convenient to use and avoids the burden of requiring the entire package to type check. Finally, in choosing to use TypeScript instead of JavaScript, I avoid dataset files that cannot be migrated without refactoring; however, the trade-off is that this no longer reflects the use case of migrating JavaScript to TypeScript.

In particular, our fill-in-the-type technique demonstrates that SantaCoder can significantly improve at type prediction if it is fine-tuned properly. Additionally, OpenTau shows that constructing an appropriate prompt format also helps improve type prediction performance. Together, this suggests that specialized fine-tuning and prompting for a specific task, e.g., type prediction, can improve performance.

4.4 Publication Status

This work was submitted to NeurIPS 2023, the Conference on Neural Information Processing Systems [15], and a final decision is expected in late September 2023.

5 PROPOSED WORK: GENERATING TYPE DEFINITIONS

My previous work on type migration has only focused on type prediction. Now, I would like to return to one of the other type migration problems, which is *generating type definitions*. To my knowledge, this problem has not been studied before, and prior work has only examined predicting library or user-defined types.

5.1 Problem Definition

Consider Figure 3, which shows a function that computes the distance between two points. A type prediction system may take the unannotated JavaScript version of Figure 3a as input and return the TypeScript version of Figure 3b, where the inserted type annotations for `Point` are highlighted. However, the resulting program fails to type check, because `Point` is not defined.

The task is to generate the definition for `Point` and insert it into the program so that the program type checks. This assumes that type annotations have already been inserted into the program.

5.2 Assumptions

For training and evaluation, I plan to use self-contained, single-file TypeScript programs; these are the datasets [77, 78] I created for Cassano et al. [15]. This approach avoids the problem of handling inter-file dependencies or inter-project dependencies. However, it also assumes that files in the dataset define and use types; otherwise, there is no need to generate type definitions. If the

```
17 <commit_before>function dist(p1, p2) {
18   const dx = p2.x - p1.x;
19   const dy = p2.y - p1.y;
20   return Math.sqrt(dx*dx + dy*dy);
21 }
22 <commit_msg>add TypeScript annotations to function
23 <commit_after>
```

(a) Input code and edit instruction, formatted as a Git commit for StarCoder.

```
24 function dist(p1: Point, p2: Point): number {
25   const dx = p2.x - p1.x;
26   const dy = p2.y - p1.y;
27   return Math.sqrt(dx*dx + dy*dy);
28 }
```

(b) Output code, with type annotations added.

Fig. 4. Adding type annotations by providing an edit instruction to StarCoder. Special tokens are highlighted.

assumption does not hold, then I will need a dataset of *projects*, rather than individual files. For example, I could use the JavaScript dataset from Yee and Guha [79], or an unused JavaScript dataset I had prepared for Cassano et al. [15].

Another assumption is that a TypeScript dataset is appropriate for evaluation. This approach avoids evaluating on files that cannot be migrated without refactoring, but does not reflect the use case of migrating from JavaScript to TypeScript.

5.3 Approach

I propose fine-tuning a large language model to generate type definitions, specifically, one of the StarCoder models [44]. This approach draws from the lessons of Cassano et al. [15], where we fine-tuned a large language model for type prediction and determined an effective way of using that model during inference.

The StarCoder family of models builds on the work of SantaCoder: StarCoderBase was trained on 86 programming languages from The Stack, including JavaScript and TypeScript, and has 15.5 billion parameters. This model is large, requiring around 60 GB to store and a data centre GPU to run. There are also smaller versions of StarCoderBase, such as StarCoderBase-1B, which has 1 billion parameters and requires 5 GB to store and a consumer GPU to run.

The training data and format for StarCoder included Git commits, allowing the model to learn to edit code based on natural language prompts. Figure 4 shows an example of adding type annotations to a function by providing a natural language prompt to the model: StarCoder was trained on the format in Figure 4a, so it learns to associate a commit message with the code before and after the commit.

I do not believe it is feasible to train a new model from scratch, or to fine-tune StarCoder on a new format. Therefore, I believe the best approach is to re-use the Git commit format and try different edit instructions and code examples, e.g. untyped to fully typed, partially typed to more typed, typed but missing definitions to fully typed with definitions, etc.

5.3.1 *Alternative Approaches.*

Database of types. This approach does not involve machine learning. The idea is to process source code that was used for training and create a database of type definitions. Thus, if a type-annotated

program refers to an undefined type `T`, then the database is queried for `T`'s definition, which is then inserted into the program. However, there may be multiple definitions for `T`, so the challenge is to determine the right one.

This approach assumes that an untyped program is first migrated to a typed program that may be missing type definitions. One way to identify missing type annotations is to parse the error messages for unresolved type names, transform the unresolved names to `any`, and then type check. If the program type checks, then the type errors are due to missing type definitions and not some other issue.

Type definitions first. In contrast to all other approaches discussed, this approach flips the order by generating type definitions first, and then adding type annotations to the program. This reflects how a programmer writes in a statically typed programming language: defining types before use. The idea would be to use some kind of constraint-based type inference (e.g., the TypeScript compiler) to generate a possibly verbose structural type definition, and then use machine learning to generate a name for that type.

However, this approach seems impractical, due to two challenges. First, it is not clear how to evaluate the quality of a type name: there may be no ground truth for comparison, the name does not matter when type checking, and name quality is ultimately a subjective measure. Second, the process of generating type definitions may be brittle: the TypeScript type system is complex, and a conservative type inference scheme may produce imprecise and unhelpful types, e.g., `any`. As a result of these challenges, I will be prioritizing the other approaches previously discussed.

5.4 Plan

5.4.1 *Preliminary Work.*

1. I implemented a test harness to evaluate the task of generating type definitions. As a baseline, the harness uses StarCoderBase-1B, the smaller, 1 billion parameter version of StarCoderBase.
2. I ran the baseline experiments on my TypeScript test dataset [77], with type annotations and definitions removed. After some experimentation, I decided to use the edit instruction "`Add type annotations and interfaces`."
3. As an initial evaluation metric, I used accuracy and measured an accuracy of 15% on the baseline experiment. Although accuracy has its limitations, it does not make sense to invoke the type checker during the training loop. I plan to use type checking as a metric in the full evaluation.
4. I prepared, on-the-fly, a training dataset based on Yee [78], that removes all type definitions and annotations, using the StarCoder Git commit format, with the edit instruction "`Add type annotations and interfaces`." This format trains the model to go from an untyped program to a fully typed program with type definitions.
5. I fine-tuned StarCoderBase-1B on the training dataset, for 24 hours on two NVIDIA H100 GPUs.

5.4.2 *Planned Work.*

6. Evaluate the fine-tuned model on accuracy.
7. Try different fine-tuning formats, e.g., partially typed to more typed, typed but missing definitions to fully typed with definitions, interactive chat-like format, fill-in-the-middle for type definitions, etc.
8. Perform a more rigorous evaluation, e.g. using the TypeScript type checker instead of accuracy, using projects instead of files, and using JavaScript instead of TypeScript. This step may involve preparing a new or adapted dataset.

Table 6. Schedule

	Sep	Oct	Nov	Dec
Type definitions	X			
Paper	X	X		
Dissertation	X	X	X	
Defence				X

6 PROPOSED SCHEDULE

I am currently working on the type definitions problem and plan to finish soon. After my proposal defence, I expect to start writing the paper and my dissertation. Depending on completion status and fit, I see several potential venues for submitting the paper: ETAPS (expected October deadline) PLDI (expected November deadline), ECOOP (expected December deadline), and ICML (expected January deadline). Table 6 summarizes my proposed timeline.

Concurrently, I expect a decision for the NeurIPS paper [15] in late September. If the paper is accepted, we will need to work on final revisions and a talk for the conference in December. If the paper is not accepted, then I expect we will resubmit to ICLR or PLDI.

7 RELATED WORK

Constraint-based type inference. There are many constraint-based approaches to type migration for the gradually typed lambda calculus (GTLC) and some modest extensions. The earliest approach was a variation of unification-based type inference [67], and more recent work uses a wide range of techniques [12, 17, 29, 48, 51, 59]. Since these approaches are based on programming language semantics, they produce sound results, which is their key advantage over learning-based approaches. However, these would require significant work to scale to complex programming languages such as JavaScript.

Constraint-based type inference for larger languages. There are also several constraint-based approaches to type inference for larger languages. Anderson et al. [3] presents type inference for a small fragment of JavaScript, but is not designed for gradual typing. Rastogi et al. [61] infer gradual types for ActionScript to improve performance. More recently, Chandra et al. [18] infer types for JavaScript programs with the goal of compiling them to run efficiently on low-powered devices; their approach is not gradual by design and deliberately rejects certain programs. DRuby [28] infers types for Ruby and treats type annotations in a novel way: inference assumes that annotations are correct, and defers checking them to runtime.

There are also several other gradual type systems for JavaScript [22, 31, 43, 73]. These languages do not have support for type inference and do not provide tools for type migration. Instead, like Typed Racket [70], they require programmers to manually migrate their code to add types. However, there are tools that use dynamic profiles to infer types for these type systems [2, 27, 64].

Even when constraint-based type inference succeeds in a gradually typed language, it can fail to produce the kinds of types that programmers write, e.g., named types, instead of the most general structural type for every annotation. Soft Scheme [13] infers types for Scheme programs, but Flanagan [25, p. 41] reports that it produces unintuitive types. For Ruby, InferDL [41] uses hand-coded heuristics to infer more natural types, and SimTyper [40] uses machine learning to predict equalities between structural types and more natural types; however, the type names refer to existing type definitions seen during training or present in user code, and SimTyper does not generate type definitions.

Deep type prediction and code generation. Several earlier works have proposed using deep learning to predict types for JavaScript and TypeScript. DeepTyper [32] and NL2Type [46] use recurrent neural networks, LambdaNet [74] uses a graph neural network, and TypeBert [39] and DiverseTyper [37] use BERT-style architectures.

There have also been works to predict types for Python [1, 23, 50, 57, 58, 76]; in particular, TypeWriter [60] uses a type checker to search the space of type predictions. A distinction between Python type systems and TypeScript is that Python code is predominantly nominally typed: the type of a variable is either a builtin type or a class, whereas TypeScript uses structural types.

Recently, decoder-only transformer neural networks have been widely used for general code generation, which in extension are capable of type prediction. Notable among these works are Codex [20], InCoder [26], SantaCoder [8], and StarCoder [44]. For code generation tasks that require edit-style generation, *fill-in-the-middle* training and inference strategies have been proposed [7, 8, 26, 34].

Evaluation datasets. ManyTypes4TypeScript [38] is a comprehensive dataset of TypeScript type annotations for training and evaluation, including evaluation scripts; however, the metrics are based on accuracy of individual type annotations. There are also datasets for Python deep learning type inference [1, 49].

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3385997

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1926385.1926437

[3] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/11531142_19

[4] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. 2023. Multi-lingual Evaluation of Code Generation Models. In *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.2210.14868

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. https://doi.org/10.48550/arXiv.2108.07732

[6] Luke Autry. 2019. How we failed, then succeeded, at migrating to TypeScript. https://heap.io/blog/migrating-to-typescript. Accessed: 2022-12-01.

[7] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. https://doi.org/10.48550/arXiv.2207.14255

[8] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! https://doi.org/10.48550/arXiv.2301.03988

[9] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-662-44202-9_11

[10] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-662-49498-1_4

[11] Ryan Burgess, Joe King, Stacy London, Sumana Mohan, and Jem Young. 2022. TypeScript migration - Strict type of cocktails. https://frontendhappyhour.com/episodes/typescript-migration-strict-type-of-cocktails. Accessed: 2022-12-01.

[12] John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. *Proc. ACM Program. Lang.* 2, POPL (2018). https://doi.org/10.1145/3158103

[13] Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/113445.113469

[14] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering (TSE)* 49, 7 (2023). https://doi.org/10.1109/TSE.2023.3267446

[15] Federico Cassano, Ming-Ho Yee, Noah Shinn, Arjun Guha, and Steven Holtzen. 2023. Type Prediction With Program Decomposition and Fill-in-the-Type Training. https://doi.org/10.48550/arXiv.2305.17145

[16] Mauricio Cassola, Agustín Talagorria, Alberto Pardo, and Marcos Viera. 2020. A Gradual Type System for Elixir. In *Brazilian Symposium on Context-Oriented Programming and Advanced Modularity (SBLP)*. https://doi.org/10.1145/3427081.3427084

[17] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290329

[18] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2983990.2984017

[19] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). https://doi.org/10.1145/3133872

[20] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374

[21] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, Hao Yu, Li Yan, Pingyi Zhou, Xin Wang, Yuchi Ma, Ignacio Iacobacci, Yasheng Wang, Guangtai Liang, Jiansheng Wei, Xin Jiang, Qianxiang Wang, and Qun Liu. 2022. PanGu-Coder: Program Synthesis with Function-Level Language Modeling. https://doi.org/10.48550/arXiv.2207.11280

[22] Ravi Chugh, David Herman, and Ranjit Jhala. 2012. Dependent Types for JavaScript. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/2384616.2384659

[23] Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang. 2021. PYInfer: Deep Learning Semantic Type Inference for Python Variables. https://doi.org/10.48550/arXiv.2106.14316

[24] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. 2008. From Dirt to Shovels: Fully Automatic Tool Generation from Ad Hoc Data. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1328438.1328488

[25] Cormac Flanagan. 1997. *Effective Static Debugging via Componential Set-based Analysis.* Ph. D. Dissertation. Rice University. https://users.soe.ucsc.edu/~cormac/papers/thesis.pdf

[26] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.2204.05999

[27] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. https://doi.org/10.1145/1640089.1640110

[28] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. 2009. Static Type Inference for Ruby. In *Symposium on Applied Computing (SAC)*. https://doi.org/10.1145/1529282.1529700

[29] Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2676726.2676992

[30] GitHub, Inc. 2020. The top programming languages. https://octoverse.github.com/2022/top-programming-languages. Accessed: 2023-08-01.

[31] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/978-3-642-19718-5_14

[32] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3236024.3236051

[33] Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. 2022. Highly Illogical, Kirk: Spotting Type Mismatches in the Large despite Broken Contracts, Unsound Types, and Too Many Linters. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022). https://doi.org/10.1145/3563305

[34] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-Tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *Automated Software Engineering (ASE)*. https://doi.org/10.1145/3551349.3556912

[35] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. https://doi.org/10.48550/arXiv.1909.09436

[36] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-Token Code Completion by Jointly Learning from Structure and Naming Sequences. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510172

[37] Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. 2022. Learning To Predict User-Defined Types. *IEEE Transactions on Software Engineering (TSE)* (2022). https://doi.org/10.1109/TSE.2022.3178945

[38] Kevin Jesse and Premkumar T. Devanbu. 2022. ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference. In *Mining Software Repositories (MSR)*. https://doi.org/10.1145/3524842.3528507

[39] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3468264.3473135

[40] Milod Kazerounian, Jeffrey S. Foster, and Bonan Min. 2021. SimTyper: Sound Type Inference for Ruby Using Type Equality Prediction. *Proc. ACM Program. Lang.* 5, OOPSLA (2021). https://doi.org/10.1145/3485483

[41] Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. Sound, Heuristic Type Annotation Inference for Ruby. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/3426422.3426985

[42] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. https://doi.org/10.48550/arXiv.2211.15533

[43] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/2578856.2508170

[44] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! https://doi.org/10.48550/arXiv.2305.06161

[45] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2022. Gradual Soundness: Lessons from Static Python. *The Art, Science, and Engineering of Programming* 7, 1 (2022). https://doi.org/10.22152/programming-journal.org/2023/7/2

[46] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00045

[47] Meta Platforms, Inc. 2019. Pyre: A performant type-checker for Python 3. https://pyre-check.org/. Accessed: 2022-12-01.

[48] Zeina Migeed and Jens Palsberg. 2020. What Is Decidable about Gradual Types? *Proc. ACM Program. Lang.* 4, POPL (2020). https://doi.org/10.1145/3371097

[49] Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-Based Type Inference. In *Mining Software Repositories (MSR)*. https://doi.org/10.1109/MSR52588.2021.00079

[50] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510124

[51] Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290331

[52] Thomas Moore. 2019. How We Completed a (Partial) TypeScript Migration In Six Months. https://blog.abacus.com/how-we-completed-a-partial-typescript-migration-in-six-months/. Accessed: 2022-12-01.

[53] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.2203.13474

[54] Guilherme Ottoni. 2018. HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3192366.3192374

[55] Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2021. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. https://doi.org/10.48550/arXiv.2004.00348

[56] Mihai Parparita. 2020. The Road to TypeScript at Quip, Part Two. https://quip.com/blog/the-road-to-typescript-at-quip-part-two. Accessed: 2022-12-01.

[57] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/3510003.3510038

[58] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. https://doi.org/10.48550/arXiv.2307.09163

[59] Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-Based Gradual Type Migration. *Proc. ACM Program. Lang.* 5, OOPSLA (2021). https://doi.org/10.1145/3485488

[60] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3368089.3409715

[61] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103656.2103714

[62] Felix Rieseberg. 2017. TypeScript at Slack. https://slack.engineering/typescript-at-slack/. Accessed: 2022-12-01.

[63] Sergii Rudenko. 2020. ts-migrate: A Tool for Migrating to TypeScript at Scale. https://medium.com/airbnb-engineering/ts-migrate-a-tool-for-migrating-to-typescript-at-scale-cd23bfeb5cc. Accessed: 2022-12-01.

[64] Claudiu Saftoiu. 2010. *JSTrace: Run-time Type Discovery for JavaScript*. Master's thesis. Brown University. https://cs.brown.edu/research/pubs/theses/ugrad/2010/saftoiu.pdf

[65] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. *International Conference on Machine Learning (ICML)* (2023). https://doi.org/10.48550/arXiv.2302.00093

[66] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. http://schemeworkshop.org/2006/13-siek.pdf

[67] Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-Based Inference. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/1408681.1408688

[68] Stack Overflow. 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#most-popular-technologies-language. Accessed: 2023-08-01.

[69] Simeng Sun, Kalpesh Krishna, Andrew Mattarella-Micke, and Mohit Iyyer. 2021. Do Long-Range Language Models Actually Use Long-Range Context?. In *Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.48550/arXiv.2109.09115

[70] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1328438.1328486

[71] Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *Summit on Advances in Programming Languages (SNAPL)*. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

[72] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 - Type Hints. https://peps.python.org/pep-0484/. Accessed: 2023-04-01.

[73] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2015. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2015.52

[74] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.2005.02161

[75] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Machine Programming Symposium (MAPS)*. https://doi.org/10.1145/3520312.3534862

[76] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python Probabilistic Type Inference with Natural Language Support. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/2950290.2950343

[77] Ming-Ho Yee. 2023. ts-eval. https://huggingface.co/datasets/nuprl/ts-eval. Accessed: 2023-08-01.

[78] Ming-Ho Yee. 2023. ts-training. https://huggingface.co/datasets/nuprl/ts-training. Accessed: 2023-08-01.

[79] Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types That Type Check?. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.37

[80] Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types That Type Check? (Artifact). *Dagstuhl Artifacts Series* 9, 2 (2023). https://doi.org/10.4230/DARTS.9.2.5

[81] Jake Zimmerman. 2022. Sorbet: Stripe's type checker for Ruby. https://stripe.com/blog/sorbet-stripes-type-checker-for-ruby. Accessed: 2022-12-01.

A APPENDIX

This appendix was added after the thesis proposal defence.

A.1 Thesis Committee

My thesis committee consists of:

- Arjun Guha (advisor)
- Steven Holtzen
- Frank Tip
- Michael Greenberg (external)

Steven Holtzen works on machine learning and programming languages, and was a collaborator on one of the papers in my thesis proposal. Frank Tip works on software engineering, program analysis, and JavaScript. Michael Greenberg works on scaling programming language techniques to real systems, and has also worked on gradual typing.

A.2 Thesis Committee Feedback

After the thesis proposal defence, my thesis committee provided feedback and require the following before my thesis defence:

1. Resubmit the OpenTau paper [15] if needed.
2. Add related work, specifically Fisher et al. [24], Hoeflich et al. [33], Pradel et al. [60].
3. Present the alternative approaches as "Future Work."
4. Present the most naïve baseline, which is using `tsc` itself to migrate JavaScript code.
5. Evaluate type migration approaches on 5–10 larger projects that span multiple files. Join all the files into one file when needed, and feel free to do so by hand.
6. Complete the type declaration inference project as outlined. (This work does not need to be submitted as a paper.)