

PREDICTING TYPESCRIPT TYPE ANNOTATIONS AND
DEFINITIONS WITH MACHINE LEARNING

MING-HO YEE

Doctor of Philosophy (Ph.D.)
Khoury College of Computer Sciences
Northeastern University

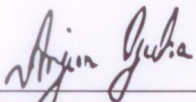
April 2024

Thesis Title: Predicting TypeScript Type Annotations and Definitions with Machine Learning

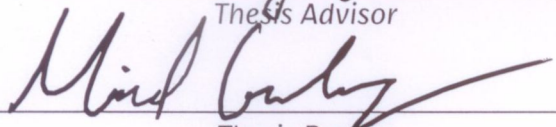
Author: Ming-Ho Yee

PhD Program: Computer Science Cybersecurity Personal Health Informatics

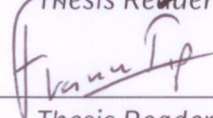
PhD Thesis Approval to complete all degree requirements for the above PhD program.


Thesis Advisor

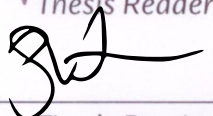
March 29 2024
Date


Thesis Reader

2024.03.29
Date


Thesis Reader

3/29/24
Date

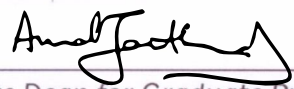

Thesis Reader

3/29/24
Date

Thesis Reader


Date

KHOURY COLLEGE APPROVAL:


Associate Dean for Graduate Programs

4/8/2024
Date

COPY RECEIVED BY GRADUATE STUDENT SERVICES:


Recipient's Signature

4/2/2024
Date

ABSTRACT

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 this dissertation, I propose to use machine learning to partially migrate JavaScript programs to TypeScript, by *predicting type annotations* and *generating type definitions*. To support this 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 (LLM) with fill in the middle (FIM) 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.

ACKNOWLEDGMENTS

This has been a very long journey, and there have been so many days, weeks, and months where I did not think I would ever reach the end. But I made it, thanks to everyone who has supported and encouraged me.

First, I have to thank my advisor, Arjun Guha. I genuinely believe I would not have completed my Ph.D. without Arjun. My committee, Michael Greenberg, Frank Tip, and Steven Holtzen, also helped me get over the finish line, with their thoughtful feedback and questions.

I had the opportunity to mentor and advise several undergraduate and master's students. Thank you for being great collaborators. I learned a lot from Ayaz Badouraly, Jakob Hain, Federico Cassano, Noah Shinn, and Sydney Nguyen. In particular, [Chapter 4](#) of my dissertation would not have been possible without the help from Federico and Noah.

The faculty, former faculty, and visiting faculty of the Programming Research Laboratory have always provided advice and valuable insight. Thank you Mitch Wand, Karl Lieberherr, Matthias Felleisen, Olin Shivers, Amal Ahmed, Ben Lerner, Jan Vitek, Frank Tip, Arjun Guha, Jon Bell, Steven Holtzen, Chris Martens, Francesco Zappa Nardelli, John Boyland, Heather Miller, Will Clinger, and Jason Hemann.

I could not imagine doing a Ph.D. without the community of students and postdocs of the Programming Research Laboratory. We spent so much time together, we talked about research, and we talked about everything else. I am fortunate to have been able to meet so many people. Thank you Tony Garnock-Jones, Kevin Clancy, William Bowman, Andrew Cobb, Celeste Hollenbeck, Di Zhong, Hyeyoung Shin, Jonathan Schuster, Ben Greenman, Justin Slepak, Max New, Aaron Weiss, Daniel Patterson, Leif Andersen, Oli Flückiger, Ellen Arteca, Alexi Turcotte, Artem Pelenitsyn, Julia Belyakova, Aviral Goel, Ben Chung, Sam Caldwell, Michael Ballantyne, Olek Gierczak, Cameron Moy, Sam Stites, Nate Yazdani, Andrew Wagner, Lucy Menon, Katie Hough, Donald Pinckney, Michelle Thalakottur, John Gouwar, John Li, Satya Gokhale, James Perretta, Yangtian Zi, Max Bernstein, Minsung Cho, Gwen Lincroft, Francesca Lucchetti, Farideh Khalili, Brianna Marshall, Liam DeVoe, Filippo Ghibellini, Ayaz Badouraly, Paul Laforge, Borja Lorente Escobar, Lionel Zoubritzky, Jakob Hain, Petr Maj, Jan Ječmen, Sebastián Krynski, Federico Cassano, Dan Nguyen, Sydney Nguyen, Anders Freeman, Claire Schlesinger, Filip Křikava, Gabriel Scherer, Konrad Siek, Paley Li, Ryan Culpepper, Saba Alimadadi, Ashish Mishra, Stephen Chang, Guido Chari, Pierre Donat-Bouillud, Dimi Racordon and Ryan Doenges. I especially want to thank Celeste, Leif, Ellen,

Alexi, Artem, Julia, Aviral, and Ben C. for helping me through all the challenging times.

I am grateful to the friends who have supported me through the ups and downs over the years. Thank you Aaron Dos Remedios, Adam Swift, Akhil Garg, Andrew Yim, Brian Luong, Hugh Hamilton, Isaac Renert, Jenny Mao, Leslie Newcombe, Luka Obrovac, Meghan Torchia, Milan Fernandez, Raymond Chen, Rein Otsason (rest in peace), Simone Valade, and Zhenglin Liu.

Last but not least: my mother, Wai-Sin; my father, Siu-Pok; and my sister, Ming-Cee, have supported me for as long as I can remember. Thank you.

CONTENTS

1	Introduction	1
1.1	Thesis and Contributions	3
2	Background	5
2.1	Type Migration	5
2.2	Machine Learning for Type Prediction	9
2.2.1	Large Language Models	10
2.2.2	Fill in the Middle	12
3	Evaluating Type Prediction Models	15
3.1	Type Prediction Models and Accuracy	16
3.1.1	DeepTyper	16
3.1.2	LambdaNet	17
3.1.3	InCoder	18
3.1.4	StarCoder	19
3.1.5	Evaluating on Accuracy	19
3.2	Approach	21
3.2.1	CommonJS to ECMAScript Module Conversion	21
3.2.2	Type Annotation Prediction	23
3.2.3	Type Weaving	25
3.2.4	Type Checking	26
3.3	Evaluation	27
3.3.1	Dataset	27
3.3.2	TypeScript Built-in Type Inference	29
3.3.3	Success Rate of Type Checking	30
3.3.4	Error Analysis	35
3.3.5	ECMAScript Module Conversion	37
3.3.6	Case Studies	39
3.4	Discussion	45
4	Training Type Prediction Models	47
4.1	Overview	48
4.2	Program Typedness	52
4.3	Tree-Based Program Decomposition	53
4.3.1	Decomposing the Program	53
4.3.2	Traversing the Tree	54
4.3.3	Ranking Candidate Solutions	56
4.4	Fine-Tuning for Fill in the Type	57
4.5	Evaluation	58
4.5.1	Dataset	58
4.5.2	Experiments	61

4.5.3	Case Studies	64
4.6	Summary	68
5	Generating Type Definitions	73
5.1	Approach	74
5.1.1	Single-step migration	75
5.1.2	Multi-step migration	76
5.2	Training	79
5.3	Evaluation	80
5.3.1	Dataset	80
5.3.2	Inference	82
5.3.3	Results	83
5.3.4	Case Studies	94
5.4	Discussion	97
6	Discussion	101
6.1	Performance Bottleneck	101
6.2	Refactoring	101
6.3	Limitations	105
7	Related Work	107
7.1	Gradual Typing for JavaScript	107
7.2	Constraint-Based Type Inference	107
7.2.1	Inferring TypeScript Type Declarations	108
7.3	Deep Type Prediction	109
7.3.1	JavaScript and TypeScript	109
7.3.2	Python	110
7.3.3	Evaluation Datasets	111
7.4	Code Generation	111
8	Future Work	113
9	Conclusion	115
	Bibliography	117

LIST OF FIGURES

Figure 2.1	Classification approach to type prediction	9
Figure 2.2	Fill in the middle approach to type prediction . . .	10
Figure 2.3	Example of a large language model generating text	11
Figure 3.1	TYPEWEAVER workflow	21
Figure 3.2	CommonJS vs. ECMAScript modules	22
Figure 3.3	Generating types with InCoder	24
Figure 3.4	eCDF of lines of code per package	27
Figure 3.5	Packages that type check	31
Figure 3.6	Error-free files per package	32
Figure 3.7	Files with no compilation errors	33
Figure 3.8	Trivial type annotations	34
Figure 3.9	Type annotation accuracy	36
Figure 3.10	eCDF of errors per package	37
Figure 3.11	Distribution of the most common error codes	38
Figure 3.12	Module conversion and type checking	40
Figure 3.13	Case study: decamelize	42
Figure 3.14	Case study: ieee754	43
Figure 3.15	Case study: @gar/promisify	44
Figure 3.16	Case study: array-unique	45
Figure 4.1	A program and its tree representation	49
Figure 4.2	Prompt and candidate solution for helloHelper . .	50
Figure 4.3	Prompt and type annotations for helloGen	50
Figure 4.4	Both candidate solutions for the program	51
Figure 4.5	Two ways of assigning types to a function	52
Figure 4.6	The PSM and SPM formats	57
Figure 4.7	An example transformation to PSM format	58
Figure 4.8	An example of an index signature	60
Figure 4.9	Fill in the middle generates extraneous code	64
Figure 4.10	Files excluded from the dataset	65
Figure 4.11	A low quality TypeScript file	66
Figure 4.12	A high quality TypeScript file	67
Figure 4.13	Comparing the baseline to OPENTAU	69
Figure 4.14	Comparing OPENTAU without and with usages . . .	70
Figure 5.1	A prompt for adding type annotations	74
Figure 5.2	A training example for single-step migration	75
Figure 5.3	Training examples for multi-step migration	78
Figure 5.4	Packages and files that parse	84
Figure 5.5	Packages and files that type check	85

Figure 5.6	Trivial type annotations	87
Figure 5.7	Type annotation sites	89
Figure 5.8	Type definitions added	90
Figure 5.9	Type definitions used	91
Figure 5.10	Similarity of input and output code	93
Figure 5.11	Files that migrated correctly	95
Figure 5.12	Case study: tokenizer	96
Figure 5.13	Case study: nonsense output	99

LIST OF TABLES

Table 2.1	The StarCoder family of models	12
Table 3.1	Summary of TYPEWEAVER dataset categories	27
Table 3.2	Packages that type check	31
Table 3.3	Files with no compilation errors	33
Table 3.4	Trivial type annotations	34
Table 3.5	Type annotation accuracy	36
Table 3.6	The top 10 most common error codes	39
Table 3.7	Module conversion and type checking	40
Table 3.8	Module conversion and error-free files	41
Table 3.9	Module conversion and type annotation accuracy	41
Table 4.1	Typedness scores	53
Table 4.2	Factors used to filter the evaluation dataset	59
Table 4.3	OPENTAU evaluation results	62
Table 4.4	Comparing the TypeScript compiler to OPENTAU	63
Table 4.5	Effectiveness of the type parser	63
Table 5.1	Summary of STENOTYPE training datasets	79
Table 5.2	Summary of STENOTYPE evaluation datasets	82
Table 5.3	Inference time for evaluation	83
Table 5.4	Packages and files that parse	84
Table 5.5	Packages and files that type check	85
Table 5.6	Trivial type annotations	87
Table 5.7	Average number of errors	88
Table 5.8	Type annotation sites	89
Table 5.9	Type definitions added	90
Table 5.10	Type definitions used	91
Table 5.11	Similarity of untyped code	92
Table 5.12	Similarity of input and output code	93
Table 5.13	Files that migrated correctly	95

ACRONYMS

AST	abstract syntax tree
CSV	comma-separated values
DOM	Document Object Model
FIM	fill in the middle
FIT	fill in the type
LLM	large language model
LOC	lines of code
LoRA	Low Rank Adaptation
PSM	prefix-suffix-middle
SPM	suffix-prefix-middle

INTRODUCTION

Type information is useful for developing large-scale software. Types help prevent bugs, provide documentation, and improve support for editors and development tools. Furthermore, a static type checker can identify type errors at compile time, early in the development cycle, rather than much later at run time. On the other hand, 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 typed and untyped code and choose the level of type safety they wish to opt into [Guha et al., 2011; Siek and Taha, 2006; Tobin-Hochstadt and Felleisen, 2008; Tobin-Hochstadt, Felleisen, et al., 2017]. This makes it possible to prototype with untyped code, and then incrementally add static types to an existing codebase without requiring a complete rewrite 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 [Bierman et al., 2014; Bonnaire-Sergeant et al., 2016; Cassola et al., 2020; Chaudhuri et al., 2017; Lu et al., 2022; Meta Platforms, Inc., 2019; Ottoni, 2018; Tobin-Hochstadt and Felleisen, 2008; Rossum et al., 2014; Zimmerman, 2022].

TYPESCRIPT. One success story is TypeScript, a widely used gradually typed language that extends JavaScript syntax with optional type annotations [Bierman et al., 2014]. Both TypeScript and JavaScript are immensely popular: JavaScript is the most popular programming language on GitHub, while TypeScript has risen from eighth in 2017 to third in 2023 [GitHub, Inc., 2023]. JavaScript is also the most popular language on Stack Overflow, while TypeScript is fifth [Stack Overflow, 2023]. Programmers can write their code in TypeScript (assigning or omitting type annotations as desired), benefit from static type checking, 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: programmers must tediously annotate their programs with types, and code may even need to be rewritten in some cases. For example, Airbnb engineers took more than two years to migrate 6 million lines of JavaScript [Rudenko, 2020], and there are several other accounts of multi-year JavaScript-to-TypeScript migration efforts [Autry, 2019; Burgess et al., 2022; Moore, 2019; Parparita, 2020; Rieseberg, 2017].

CONSTRAINT-BASED TYPE INFERENCE. One approach to automate type migration is constraint-based type inference. Anderson et al. [2005] presented the first type inference algorithm for JavaScript, but only for a fragment of the language. Other, more recent approaches are similarly limited to fragments or dialects of JavaScript, or are used for performance optimizations rather than type migration [Hackett and S.-y. Guo, 2012; Rastogi, Chaudhuri, et al., 2012; Chandra et al., 2016].

There are also type inference tools that generate type annotations and type definitions, but not TypeScript code [Naus, 2015; Kahlert, 2018]. For example, the TypeScript compiler can be configured to generate interface declaration (.d.ts) files.¹ These files provide type annotations for exported functions, but do not actually insert type annotations into the original JavaScript code. Instead, the purpose of these type declaration files is to allow JavaScript packages to be imported into TypeScript projects. While this is useful as an intermediate migration step, it is not an end-to-end solution for JavaScript-to-TypeScript type migration.

TYPE PREDICTION. An alternative approach towards effective automated type migration is to use machine learning techniques to attack the narrower problem of *type prediction*. Compared to type migration, which may involve refactoring code that is not well typed, type prediction is concerned only with predicting the most likely type annotations for a given code fragment [Allamanis et al., 2020; Cassano, Yee, et al., 2023b; Hellendoorn et al., 2018; Jesse, Devanbu, and Ahmed, 2021; Jesse, Devanbu, and Sawant, 2022; Malik et al., 2019; Mir, Latoškinas, Proksch, et al., 2022; Pandi et al., 2021; Peng, Gao, et al., 2022; Peng, Wang, et al., 2023; Pradel et al., 2020; Wei et al., 2020b; Z. Xu et al., 2016; Yee and Guha, 2023a].

Type prediction is appealing because machine learning models can take into account the linguistic context of the code fragment, such as comments and identifier names, while traditional, constraint-based approaches have difficulty accommodating language features such as `eval`. Furthermore, there is a significant quantity of high-quality, open-source JavaScript and TypeScript code that is available to serve as training data [Husain et al., 2020; Jesse and Devanbu, 2022; Kocetkov et al., 2022; Mir, Latoškinas, and Gousios, 2021; F. F. Xu et al., 2022]. In particular, large language models (LLMs) have been successful in a variety of code generation tasks [Athiwaratkun et al., 2023; Austin et al., 2021; Ben Allal et al., 2023; Cassano, Gouwar, et al., 2023; Chen et al., 2021; Christopoulou et al., 2022; Izadi et al., 2022; Nijkamp et al., 2023; F. F. Xu et al., 2022].

However, there are limitations in the existing literature:

¹ The compiler flags are `--declaration --allowJs`

- 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.
- LLMs have neither been trained for nor evaluated on the type prediction task, other than small-scale evaluations [Fried et al., 2023; Li et al., 2023a].
- The related problem of *type definition generation* 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.

I address these limitations in my thesis.

1.1 THESIS AND CONTRIBUTIONS

This dissertation focuses 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 problem of type annotation prediction and type definition generation, and leave refactoring for future work. Therefore, my thesis is:

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

To elaborate on my thesis statement, my work uses *machine learning* models, specifically LLMs, and in particular, open such as SantaCoder [Ben Allal et al., 2023] and StarCoder [Li et al., 2023a]. I focus on a *partial migration*, acknowledging that some manual refactoring may be required, and I restrict my work to *JavaScript and TypeScript*. I do not believe a fully automated migration is currently possible, especially when handling popular programming languages with challenges not present in smaller languages like the gradually typed lambda calculus and its extensions. Finally, I study the specific tasks of *type annotation prediction* and *type definition generation*, which are part of type migration.

To support my thesis, I make three contributions:

1. I propose evaluating type prediction systems by *type checking* the generated types, rather than using the typical metric of accuracy. As

part of this work, I present an evaluation dataset and TYPEWEAVER, a system for evaluating type prediction systems.

I discuss this work in [Chapter 3](#), which is based on the ECOOP 2023 paper “Do Machine Learning Models Produce TypeScript Types That Type Check?” [Yee and Guha, 2023a].

2. Because LLMs have not been extensively evaluated for the type prediction task, I created an evaluation dataset and worked on a new fine-tuning approach called *fill in the type (FIT)* for type prediction.

I discuss this work in [Chapter 4](#), which is based on the preprint *Type Prediction With Program Decomposition and Fill-in-the-Type Training* [Cassano, Yee, et al., 2023b].

3. Finally, taking lessons from the previous two contributions, I train and evaluate STENOTYPE, an LLM that generates type definitions for TypeScript, in addition to predicting type annotations for untyped JavaScript programs.

I discuss this work in [Chapter 5](#).

The remaining chapters of my dissertation cover background material ([Chapter 2](#)), overall discussion ([Chapter 6](#)), related work ([Chapter 7](#)), future work ([Chapter 8](#)), and conclusions ([Chapter 9](#)).

BACKGROUND

In this chapter, I provide background information on *type migration*, contrasting it to type inference, as well as machine learning techniques for type migration, such as large language models (LLMs) and fill in the middle (FIM) inference.

2.1 TYPE MIGRATION

I use the term *type migration* to describe the problem of migrating a program from an untyped language to a typed language, e. g., from JavaScript to TypeScript. This process mainly involves adding type annotations to untyped code, but also has additional challenges. For example, multiple types may be valid for a program, the inserted type annotations may refer to types that need to be defined, a program can be trivially migrated with unhelpful type annotations, new errors can be introduced by type annotations, and dynamic features like `eval` cannot easily be handled without escape hatches.

In contrast, *type inference*, or *type reconstruction* is a narrower problem, where the goal is to compute the types of expressions and functions when some or all of the type annotations are missing [Pierce, 2002, ch. 22]. Furthermore, type inference typically requires a statically typed language with implicit type information in the program, and the type inference algorithm computes the missing type annotations. As a result, the inferred type annotations refer to well-defined type definitions that already exist, so there is no need to add new type definitions. Additionally, inference can frequently compute *principal types*, i. e., the most general types, meaning there is a single correct type annotation for a given expression. These properties make type inference a more straightforward problem to solve than general type migration.

The following examples illustrate some of the challenges encountered when migrating JavaScript programs to TypeScript.

Multiple types for a program

TypeScript does not have *principal types*, so a program may have multiple valid types. For example, in the following program, the parameter `x` could

be annotated as either `number` or `string`, as `x + x` could either be addition or string concatenation:

```
1 function f(x) { return x + x; }
2 f(1);      // returns 2
3 f("a");   // returns "aa"
```

Furthermore, since `f` is called with both numeric and string arguments, the only valid type annotation for `x` is `any`.¹

The actual culprit in this example is that `+` is overloaded to mean both addition and string concatenation. However, it is also possible to encounter the same problem of multiple valid type annotations, even without operator overloading. Consider the following program, which adapts an example from Migeed and Palsberg [2020, sec. 2.3]:

```
4 function id1(x) { return x; }
5 function id2(y) { return y; }
6 id2(id1(true)) * 2;
```

This program defines two identity functions, `id1` and `id2`. It calls `id1` with the argument `true`, and then passes that result to `id2`. Finally, the result is multiplied by 2. There are three valid migrations for this program:

```
7 // Migration 1
8 function id1(x: any) { return x; }
9 function id2(y: any) { return y; }
10 id2(id1(true)) * 2;
11
12 // Migration 2
13 function id1(x: any) { return x; }
14 function id2(y: number) { return y; }
15 id2(id1(true)) * 2;
16
17 // Migration 3
18 function id1(x: boolean) { return x; }
19 function id2(y: any) { return y; }
20 id2(id1(true)) * 2;
```

In other words, there are multiple valid type annotations for the original program, but none of them is the best: at least one of the annotations must be `any`.

Types need to be defined

Type inference reconstructs missing type annotations that refer to existing type definitions in the program. However, this is not the case with type

¹ The union type `number | string` produces a type error in the function.

migration, where type definitions do not already exist. Consider the following example, which initializes `point` to an empty object and then sets its `x` and `y` properties:²

```
21 let point = {}
22 point.x = 42;
23 point.y = 54;
```

A possible type annotation for `point` is `{x?: number, y?: number}`, because `point` is initialized as an empty object, so its `x` and `y` properties must be optional. Adding the type annotation results in the following program:

```
24 let point: {x?: number, y?: number} = {}
25 point.x = 42;
26 point.y = 54;
```

TypeScript has a structural type system [Bierman et al., 2014], meaning type annotations can become verbose and difficult to read, or even impossible to write in the case of recursive types. To avoid this situation, the programmer can write explicit type definitions, but now the type migration involves adding type definitions as well as type annotations:

```
27 interface Point { x?: number, y?: number };
28 let point: Point = {}
29 point.x = 42;
30 point.y = 54;
```

A more idiomatic solution would be to refactor the code as follows, by defining a class and constructor, and making `x` and `y` required:

```
31 class Point {
32   x: number;
33   y: number;
34   constructor(x: number, y: number) {
35     this.x = x;
36     this.y = y;
37   }
38 }
39 let point: Point = new Point(42, 54);
```

Trivial type annotations

The type annotation `any` will always pass the type checker, but is imprecise and provides little benefit to the programmer. For example, the following program type checks, but is no better than an untyped program:

² Prior to the introduction of classes in ECMAScript 6, this was a common JavaScript idiom.

```

40 function sumThree(x: any, y: any, z: any) {
41   return x + y + z;
42 }

```

The use of `any` can also lead to unintuitive behaviours. For example, in the following program, a programmer might expect `g` to have the same type as `f`, and therefore reject the string argument `"hi"`:

```

43 function f(x: number) { return x + 1; }
44 function id(y: any) { return y; }
45 let g = id(f);
46 g("hi") // returns "hi1"

```

However, the code type checks and the `+` operator in `f` is treated as string concatenation instead of addition, despite both of its arguments having the type `number`. This is because of type erasure: the TypeScript compiler erases types when emitting JavaScript, so there is no run-time type representation or run-time type checking [Bierman et al., 2014].³

Adding types can introduce new errors

An untyped program may be valid with no run-time errors, but adding type annotations can introduce new static errors. For example, consider the following program, which is adapted from Phipps-Costin et al. [2021, Fig. 2]:

```

47 function f(x) {
48   if (false) {
49     let y = x + 10;
50     x();
51   } else {
52     x();
53   }
54 }

```

In this program, the function `f` branches on the literal `false`, meaning the path where `x` is used as both a number and a function is unreachable. Thus, no run-time error can occur when calling this function. However, annotating `x` as `(number) => number` or `number` causes the program to fail to type check. In other words, this valid program is now ruled out by the type checker.

³ Chung et al. [2018] describes this as the *optional* approach to gradual typing, while Greenman and Felleisen [2018] calls it the *erasure* approach.

	Type of x	Probability
58 <code>function f(x) {</code>	number	0.4221
59 <code> return x + 1;</code>	any	0.2611
60 <code>}</code>	string	0.2558
	<i>other</i>	

Figure 2.1: An example of using a classification approach for type prediction. The function on the left is given as input, the model predicts types for `x`, and the predictions are shown in the table on the right.

Dynamic language features

Dynamic features like `eval` cannot be easily handled without escape hatches, such as any:

```
55 function f(s: string): any {
56   return eval(s);
57 }
```

In the example above, `eval` takes an arbitrary string argument and executes it, meaning the return type of `f` cannot be determined at compile time. Therefore, the only type annotation that captures all potential behaviours is `any`.

Summary

These examples highlight some of the challenges that arise from JavaScript to TypeScript type migration, and show that it is a more difficult problem than type inference. Ultimately, type migration is a refactoring problem.

2.2 MACHINE LEARNING FOR TYPE PREDICTION

Because of the numerous challenges of type migration, recent work has focused on the narrower problem of assigning type annotations to TypeScript code, in particular, using machine learning approaches. Machine learning is an appealing choice for the *type annotation prediction* task, because of the availability of high-quality training data, such as ManyTypes4TS [Jesse and Devanbu, 2022] and The Stack [Kocetkov et al., 2022]. This has led to a proliferation of machine learning models that can roughly be categorized into two approaches: classification approaches and code LLM approaches.

Classification approaches are an older approach where a model is trained specifically for type prediction: given a code fragment, for each

<pre> 61 function f(x: _hole_) { 62 return x + 1; 63 } </pre>	<pre> 64 function f(x: number) { 65 return x + 1; 66 } </pre>
--	--

Figure 2.2: An example of using a FIM approach for type prediction. The function on the left contains a hole and is given as input, and the LLM fills in the type annotation number.

identifier, the model produces a probability distribution of the most likely type annotations. For example, in Figure 2.1, the code on the left is given as input to a model, which predicts types for the highlighted parameter x . The predictions are shown in the table on the right, where `number` is the most likely type annotation, followed by `any`. In other words, each identifier is assigned a list of likely type annotations and their probabilities. In an interactive setting, a user could examine the predicted type annotations and choose the best one, while in an automated setting, the most likely annotation is taken. Some examples of classification approaches for TypeScript type prediction include DeepTyper [Hellendoorn et al., 2018] and LambdaNet [Wei et al., 2020b], which are discussed in Sections 3.1.1 and 3.1.2.

Today, most approaches use code LLMs, also known as large language models for code. These models are trained on vast amounts of data for general-purpose code generation: given a code fragment, they predict what code comes next.⁴ Additionally, some models support fill in the middle (FIM) inference, which allows code generation to occur at arbitrary locations, i. e., *holes* that the user inserts into the code, rather than at the end. Figure 2.2 shows a function on the left, where a hole has been inserted at the type annotation site. The LLM generates code at the hole, conditioned on the surrounding context, and the result is the function on the right, with `number` as the type annotation. LLMs that support FIM include Codex [Bavarian et al., 2022], InCoder [Fried et al., 2023], SantaCoder [Ben Allal et al., 2023], StarCoder [Li et al., 2023a], Code Llama [Rozière et al., 2023], and DeepSeekCoder [D. Guo et al., 2024].

2.2.1 Large Language Models

Large language models (LLMs) were originally trained for generating natural language text. For example, given a *prompt*, i. e. a fragment of text

⁴ Technically, LLMs are instances of classification approaches: predicting the next code token is a classification problem where the categories are all the tokens in the vocabulary. In this dissertation, it is convenient to contrast general-purpose code LLMs with older models that are only trained for type prediction, which assign type annotation labels to identifiers in code.

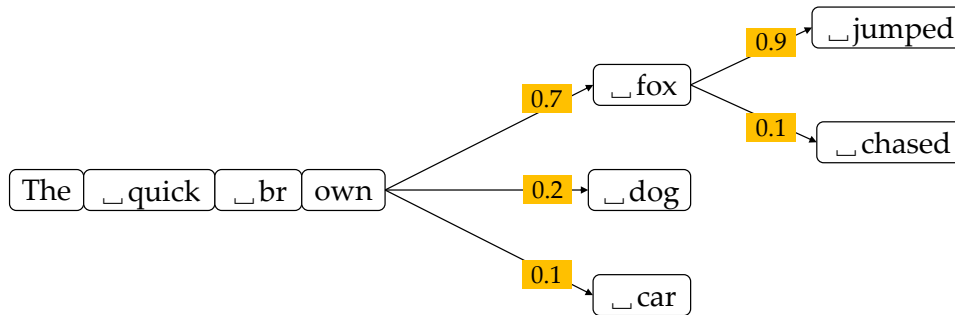


Figure 2.3: An example of an LLM generating text.

as input, the model predicts the text that follows. The model operates on *tokens*, which are small units of text that may include non-word characters like whitespace and punctuation, and may also be smaller than individual words. For example, the prompt `The quick brown` could be tokenized into four tokens: `The quick br own`. When given to a model, the model predicts the most likely token to follow. For example, Figure 2.3 shows the prompt on the left and three possible tokens that could follow: `fox` with a probability of 0.7, `dog` with a probability of 0.2, and `car` with a probability of 0.1. If `fox` is selected, it is appended to the input to create a new prompt, `The quick br own fox`, and subsequent tokens are generated from this input, for example `jumped` or `chased`. The process repeats until a fixed number of tokens is generated, or one of the special predefined “stop” tokens is generated. In this example, the most likely token was selected at each step, but there are a variety of different strategies, and in practice, some form of sampling is used, so the results will be non-deterministic.

LLMs are typically implemented as neural networks. While the details are not necessary for understanding this dissertation, the general idea is that a neural network is a graph of nodes and edges, where each node represents a weight and a bias, collectively called *parameters*. Parameters are the values that affect the output of a model, and are adjusted during training. In general, the more parameters a model has, the more powerful and capable it is, at the cost of requiring more resources to train and use. A language model is considered large when it has millions or billions of parameters. For example, GPT-3 has roughly 175 billion parameters and was trained on 570 GB of data [Brown et al., 2020].

A code LLM is similar to a natural language LLM, but trained on code examples. Some models are trained directly on code, while others are based on prior natural language LLMs. For example, the Codex model was produced by fine-tuning GPT-3 on code, and is now used to provide code suggestions for the GitHub Copilot service [Chen et al., 2021]. However,

Table 2.1: The StarCoder family of models, and the resources required to store and run inference with them.

Parameters	Disk space	GPU memory
1 B	4.3 GB	4 GB
3 B	11.5 GB	8 GB
7 B	28.0 GB	17 GB
15 B	60.0 GB	34 GB

GPT-3 and Codex are proprietary, so the model architectures, parameters, and training data are not public. As a result, there has been interest in creating “open” code LLMs, where the parameters and training data are openly available. These include SantaCoder [Ben Allal et al., 2023] and its successor, StarCoder [Li et al., 2023a].

The full StarCoder model has 15 billion parameters, which requires about 60 GB of disk for storage and a data centre GPU to run inference, such as the NVIDIA A100 80GB or NVIDIA H100. Alternatively, there are smaller versions of StarCoder available: StarCoder-1B, StarCoder-3B, and StarCoder-7B, with 1, 3, and 7 billion parameters, respectively. For comparison, StarCoder-1B and StarCoder-3B are small enough to run on consumer hardware. Table 2.1 lists the StarCoder family of models and the resources required to use them.

2.2.2 Fill in the Middle

Traditional LLMs perform left-to-right generation, where text or code is generated at the end of the input prompt. However, some models support fill in the middle (FIM), where generation occurs at arbitrary locations within the prompt. This style of generation is a natural fit for type prediction, since the predicted type annotation depends on the surrounding context. For example, suppose we would like to type annotate the parameter x of the given code fragment:

```
67 function f(x) {
68     return x + 1;
69 }
```

Conceptually, we insert a special token, representing a *hole*, at the location where a type annotation should be generated:

```
70 function f(x: _hole_) {
71     return x + 1;
72 }
```


The model then uses the surrounding code context to predict the type annotation:

```
73 function f(x: number) {
74   return x + 1;
75 }
```

TRAINING. The FIM training process uses a special format that ensures the model can generate code in the middle of a program, conditioned on the surrounding context, while preserving its ability for left-to-right code generation [Bavarian et al., 2022; Fried et al., 2023]. For example, suppose we wish to transform the following factorial function into a training example, where the model learns to generate the highlighted line of code:

```
76 function fact(n) {
77   if (n == 0) return 1;
78   return n * fact(n - 1);
79 }
```

We insert special training tokens that denote the prefix, middle, and suffix spans of code:

```
80 <fim_prefix>function fact(n) {
81   <fim_middle>if (n == 0) return 1;
82   <fim_suffix>return n * fact(n - 1);
83 }
```

Then, we move the middle code span to the very end:

```
84 <fim_prefix>function fact(n) {
85   <fim_suffix>return n * fact(n - 1);
86 }<fim_middle>if (n == 0) return 1;
```

This transforms the original surrounding context into a single prefix: in other words, FIM inference has been transformed into a left-to-right generation problem. More generally, the training procedure randomly selects one or more non-overlapping, contiguous middle spans from the training data, applies the FIM transformation, and then trains as a left-to-right language model.

INFERENCE. Once trained, inference uses the same format and special tokens to generate text. Returning to our previous example, we would like the model to replace `_hole_` with a type annotation:

```
87 function f(x: _hole_) {
88   return x + 1;
89 }
```

We insert the special prefix, suffix, and middle tokens, and rearrange the text so that the middle token is at the end:

```
90 <fim_prefix>function f(x: <fim_suffix>) {  
91     return x + 1;  
92 }<fim_middle>
```

The model generates text at the very end, after the special middle token:

```
93 <fim_prefix>function f(x: <fim_suffix>) {  
94     return x + 1;  
95 }<fim_middle> number
```

To get the final result, we extract the code after the special middle token and reverse the transformation:

```
96 function f(x: number) {  
97     return x + 1;  
98 }
```

EVALUATING TYPE PREDICTION MODELS

Over the last few years, advances in model architectures and high-quality training data have led to type annotation prediction with high accuracy on individual type annotations [Hellendoorn et al., 2018; Jesse, Devanbu, and Sawant, 2022; Jesse, Devanbu, and Ahmed, 2021; Pandi et al., 2021; Wei et al., 2020b]. However, in this chapter, I argue that accuracy can be misleading, and that predicting individual type annotations is just the first step of migrating a codebase from JavaScript to TypeScript. Instead, evaluation should ask a different question: *can an automatic type migration tool produce code that type checks?* If so, type annotations should be non-trivial and useful (i. e., annotations that are not just any). On the other hand, if the code does not type check, it may have too many errors, which can 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.

To answer the type checking question, I present `TYPEWEAVER`, a TypeScript type migration tool that can be used with an arbitrary type prediction model.¹ In this chapter, I evaluate four models from the literature: `DeepTyper` [Hellendoorn et al., 2018], a recurrent neural network; `LambdaNet` [Wei et al., 2020b], a graph neural network; and `InCoder` [Fried et al., 2023] and `StarCoder` [Li et al., 2023a], two LLMs that support `FIM`, which I discussed in [Section 2.2.2](#).

`TYPEWEAVER` automates several steps that are necessary for using a type prediction model, including:

IMPORTING TYPE DEPENDENCIES Before migrating a JavaScript project, its dependencies must be typed. This means transitively migrating dependencies, or ensuring that the dependencies have TypeScript interface declaration (`.d.ts`) files available.

MODULE CONVERSION JavaScript code written for Node.js may use either the CommonJS or ECMAScript module system. However, when migrated to TypeScript, only ECMAScript modules preserve type information. Thus, to fully benefit from static type checking, code written with CommonJS modules should be refactored to use ECMAScript modules.

¹ This work was published as Yee and Guha [2023a] and is available as an artifact [Yee and Guha, 2023b]

TYPE WEAVING Models that assign type labels to variables do not update the JavaScript source to include type annotations. Therefore, to type check a program, the predicted type annotations must be “woven” into the original JavaScript source to produce TypeScript.

REJECTING NON-TYPE PREDICTIONS Models that predict type annotations as in-filled sequences of tokens can easily produce token sequences that are not syntactic types. These predictions need to be rejected or cleaned for type prediction to work.

After completing these tasks, it is then possible to type check the resulting TypeScript program and evaluate the effectiveness of `TYPEWEAVER`.

3.1 TYPE PREDICTION MODELS AND ACCURACY

In this section, I provide some background on the four type prediction models that I evaluate: DeepTyper [Hellendoorn et al., 2018], LambdaNet [Wei et al., 2020b], InCoder [Fried et al., 2023], and StarCoder [Li et al., 2023a]. I also describe their training and evaluation datasets.

DeepTyper was the first deep neural network for TypeScript type prediction, and uses a *bidirectional recurrent neural network* architecture. LambdaNet was another early approach, and it uses a *graph neural network* architecture. InCoder and StarCoder are recent LLMs that predict arbitrary code completions, and while not trained specifically to predict type annotations, their support for FIM make them ideal for that task. All four models use training data from public code repositories.

`TYPEWEAVER` requires a system that takes a JavaScript project as input and outputs a type-annotated TypeScript project. DeepTyper and LambdaNet output a probability distribution of types for each identifier, which must be then “woven” into the original JavaScript source to produce TypeScript; I describe this technique in Section 3.2.3. InCoder and StarCoder are LLMs, which require a front end to predict type annotations and output TypeScript. The front end only supports type predictions for function parameters, and its implementation is described in Section 3.2.2.1.

My approach can be adapted to work with any type prediction model. Older models may require some work to adapt their outputs, but the InCoder front end has been easily extended to support other fill-in-the-type models, such as SantaCoder [Ben Allal et al., 2023] and StarCoder [Li et al., 2023a].

3.1.1 DeepTyper

DeepTyper [Hellendoorn et al., 2018] predicts types for variables, function parameters, and function results using a fixed vocabulary of types, i. e., it

cannot predict types declared by the program under analysis unless those types were observed during training. DeepTyper treats type inference as a machine translation problem from one language (unannotated TypeScript) to another (annotated TypeScript). Specifically, it uses a model based on a *bidirectional recurrent neural network* architecture to translate a sequence of tokens into a sequence of types: for each identifier in the source program, DeepTyper returns a probability distribution of predicted types. Because the input token sequence is perfectly aligned with the output type sequence, this task can also be considered a sequence annotation task, where an output type is expected for every input token.² However, this approach treats each input token as independent from the others, i. e., a source variable may be referenced multiple times and each occurrence may have a different type. To mitigate this, DeepTyper adds a consistency layer to the neural network, which encourages—but cannot enforce—the model to treat multiple occurrences of the same identifier as related.

DeepTyper’s dataset is based on the top 1,000 most starred TypeScript projects on GitHub, as of February 2018. After cleaning to remove large files (those with more than 5,000 tokens) and projects that contained only TypeScript declaration files, the dataset was left with 776 TypeScript projects (containing about 62,000 files and about 24 million tokens), which were randomly split into 80% (620 projects) training data, 10% (78 projects) validation data, and 10% (78 projects) test data. Further processing and cleaning of rare tokens resulted in a final vocabulary of 40,195 source tokens and 11,830 types.

The final training dataset contains both identifiers and types, where each identifier has an associated type annotation; this includes annotations inferred by the TypeScript compiler that were not manually annotated by a programmer. The testing dataset contains type annotations and no identifiers; specifically, the type annotations added by programmers are associated with their declaration sites, and all other sites are associated with “no-type.” As a result, DeepTyper’s predictions are evaluated against the handwritten type annotations, rather than all types in a project.

3.1.2 *LambdaNet*

Like DeepTyper, LambdaNet [Wei et al., 2020b] predicts type annotations for variables, function parameters, and function returns: it takes an unannotated TypeScript program and outputs a probability distribution of predicted types for each declaration site. LambdaNet improves upon two limitations of DeepTyper. First, it predicts from an open vocabulary,

² The DeepTyper architecture must classify *every* input token, including ones where an output type does not make sense, such as `if`, `(`, `)`, and even whitespace. DeepTyper filters out these predictions, so a user will never observe these meaningless types.

beyond the types that were observed during training; i. e., it can predict user-defined types from within a project. Second, it only produces type predictions at declaration sites, rather than at every variable occurrence; in other words, multiple uses of the same variable will have a consistent type.

LambdaNet uses a *graph neural network* architecture and represents a source program as a so-called *type dependency graph*, which is computed from an intermediate representation of TypeScript that names each sub-expression. The type dependency graph is a hypergraph that encodes program type variables as nodes, and relationships between those variables as labeled edges. By encoding type variables, LambdaNet makes a single prediction over all occurrences of a variable, rather than a prediction for each instance of a variable. Furthermore, the edges encode logical constraints and contextual hints. Logical constraints include subtyping and assignment relations, functions and calls, objects, and field accesses, while contextual hints include variable names and usages. Finally, LambdaNet uses a *pointer network* to predict type annotations.

LambdaNet’s dataset takes a similar approach to DeepTyper: it consists of the 300 most popular TypeScript projects from GitHub that contained 500–10,000 lines of code, and had at least 10% of type annotations that referred to user-defined types. The dataset has about 1.2 million lines of code, and only 2.7% of the code is duplicated. The 300 projects were split into three sets: 200 (67%) for training data, 40 (13%) for validation data, and 60 (20%) for test data. The vocabulary was split into library types, which consist of the top 100 most common types in the training set, and user-defined types, which are all the classes and interfaces defined in source projects. Similar to DeepTyper, LambdaNet’s predictions are evaluated against the handwritten annotations that were added by programmers.

3.1.3 InCoder

InCoder [Fried et al., 2023] is a 6 billion parameter LLM for generating arbitrary code that is trained with a FIM objective on a corpus of several programming languages, including TypeScript. InCoder’s corpus consists of permissively licensed, open-source code from GitHub and GitLab, as well as Q&A and comments from Stack Overflow. This raw data is filtered to exclude:

- code that is duplicated;
- code that is not written in one of 28 languages;
- files that are extremely large or contain very few alphanumeric characters;

- code that is likely to be compiler generated; and
- certain code generation benchmarks.

The result is about 159 GB of code, which is dominated by Python and JavaScript. TypeScript is approximately 4.5 GB of the training data.

I describe InCoder in more depth in [Section 3.2.2.1](#), where I present what is necessary to use it as a type annotation prediction tool for TypeScript.

3.1.4 *StarCoder*

StarCoder and StarCoderBase are 15 billion parameter LLMs that also support FIM [Li et al., 2023a]. StarCoderBase was trained on The Stack [Kocetkov et al., 2022], a dataset of code from permissively licensed GitHub repositories. The Stack contains code from over 300 programming languages, but only 86 languages were selected to train StarCoderBase. Furthermore, The Stack also includes natural language text from GitHub issues and pull requests, as well as Git commits. After de-duplicating, filtering, and cleaning, the final training dataset consists of 306 million files, or 816 GB of data. About 65 GB (8%) was JavaScript and 27 GB (3%) was TypeScript. StarCoderBase was trained on a total of one trillion tokens from the training dataset. Afterwards, StarCoder was created by further fine-tuning StarCoderBase on 35 billion tokens from the Python subset of the training dataset.

In this dissertation, I use StarCoderBase, since StarCoder is specialized for Python. From this point on, I refer specifically to StarCoderBase as the model I work with.

3.1.5 *Evaluating on Accuracy*

The main evaluation criteria for the type annotation prediction task is 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.

DEEPTYPYER. DeepTyper’s test dataset makes up 10% (78 projects) of its original corpus and contains only the annotations that were manually added by programmers. Predictions are compared against this ground truth dataset, and DeepTyper reports a top-1 accuracy of 56.9%. Because DeepTyper may predict different types for multiple occurrences of the

same variable, Hellendoorn et al. also report an inconsistency metric: 15.4% of variables had multiple type predictions.

LAMBDA NET. LambdaNet also compares predictions against a ground truth of handwritten type annotations, but Wei et al. use a different corpus and split 20% (60 projects) for the test dataset. LambdaNet can predict user-defined types, so the evaluation reports two sets of results: a top-1 accuracy of 75.6% when predicting only common library types, and a top-1 accuracy of 64.2% when predicting both library and user-defined types.

INCODER. InCoder was not designed specifically to predict TypeScript type annotations, but Fried et al. report an experiment to predict only the result types for Python functions. For this task, InCoder was evaluated on a test dataset of 469 functions, which was constructed from the CodeXGLUE dataset; InCoder achieved an accuracy of 58.1%.

STARCODERBASE. Li et al. perform two experiments to evaluate type prediction with [FIM](#). The first experiment uses the benchmarks of Fried et al. and evaluates Python return type prediction. The results show that StarCoder and StarCoderBase achieve accuracies of 66.9% and 77.4% respectively, which outperforms InCoder and SantaCoder.

The second experiment does not use accuracy, and is based on a revision of my benchmark that was published in Yee and Guha [2023a]. That is, the experiment evaluates TypeScript type prediction and evaluates with the TypeScript type checker. However, the experiment is only based on the “never typed” subset ([Section 3.3.1](#)) of my dataset, i. e., the packages that have never been type annotated before. In this chapter, I provide the results of the evaluation on the full dataset.

LIMITATIONS OF ACCURACY. I believe accuracy is not the right metric for evaluating a type prediction model. As a first step, I would like to type check the TypeScript project. Additionally, when migrating a JavaScript project to TypeScript, there is frequently no ground truth of handwritten type annotations; instead, the ground truth is what the compiler accepts. This condition is much stronger than accuracy, as even a single, incorrect type annotation causes a package to fail to type check. On the other hand, less precise type annotations (e. g., `any`) and equivalent annotations (e. g., `number | string` vs. `string | number`) may be accepted, despite not matching the ground truth exactly.

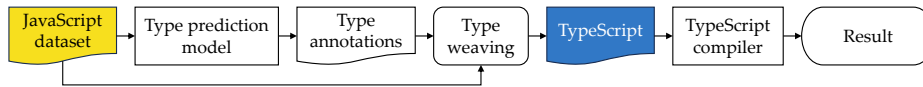


Figure 3.1: TYPEWEAVER workflow: a JavaScript dataset is given to a type prediction model, which returns type annotations; next, type weaving merges the type annotations with the original JavaScript code and produces TypeScript; finally, the TypeScript is type checked by the TypeScript compiler.

3.2 APPROACH

I built TYPEWEAVER to evaluate TypeScript type prediction models. The workflow for TYPEWEAVER is illustrated in Figure 3.1 and starts from an untyped JavaScript project and finishes with a type-annotated TypeScript project that can be given to the TypeScript type checker. The first step, which is optional, is to convert from CommonJS modules to ECMAScript modules. Next, the JavaScript code is given to one of the supported type prediction models: DeepTyper, LambdaNet, InCoder, or StarCoder-Base. DeepTyper and LambdaNet produce type predictions rather than TypeScript code, so a step called *type weaving* is needed to combine the type predictions with the original JavaScript code to produce TypeScript. Finally, TYPEWEAVER invokes the TypeScript compiler to type check the now-migrated TypeScript project.

3.2.1 CommonJS to ECMAScript Module Conversion

The first step is to convert projects from CommonJS module notation to ECMAScript module notation. This step is not necessary for type prediction, but is important for the type checking evaluation, as only ECMAScript modules preserve type information across module boundaries. Because this step is optional, my dataset has two versions: the original projects, which may use CommonJS or ECMAScript modules, and a final version that only uses ECMAScript modules.

Most Node.js packages use the CommonJS module system, which was the original module system for Node.js and remains the default. Figures 3.2a and 3.2b show an example of the CommonJS module system, where files `a.js` and `b.js` implement separate modules. In this example, `a.js` sets the `foo` and `f` properties of the special `module.exports` object. Local variables like `x` are private and not exported. On line 104, `b.js` uses the Node.js function `require` to load module `a.js` into the local variable `a`. As a result, `a` takes on the value of the `module.exports` object set by `a.js`, and both `foo` and `f` are available as properties of `a`.

<hr/> <pre> 99 // a.js 100 var x = 2; // private 101 module.exports.foo = 42; 102 module.exports.f = (i) => i+x; </pre> <hr/> <p>(a) CommonJS: a.js exports foo and f, but not x.</p> <hr/>	<hr/> <pre> 103 // b.js 104 var a = require('./a.js'); 105 console.log(a.foo); // 42 106 console.log(a.f(1)); // 3 </pre> <hr/> <p>(b) CommonJS: b.js imports the module a.js, and can access a.foo and a.f.</p> <hr/>
<hr/> <pre> 107 // a.mjs 108 var x = 2; // private 109 export var foo = 42; 110 export var f = (i) => i+x; </pre> <hr/> <p>(c) ECMAScript: a.mjs exports foo and f, but not x.</p> <hr/>	<hr/> <pre> 111 // b.mjs 112 import {foo,f} from './a.mjs'; 113 console.log(foo); // 42 114 console.log(f(1)); // 3 </pre> <hr/> <p>(d) ECMAScript: b.mjs imports foo and f from the module a.mjs.</p> <hr/>

Figure 3.2: An example comparing imports and exports with the CommonJS and ECMAScript module systems.

ECMAScript 6 introduced a new module system, referred to as ECMAScript modules. Node.js supports ECMAScript modules when using the `.mjs` extension or setting a project-wide configuration in the `package.json` file. Figures 3.2c and 3.2d show the same program as before, but rewritten to use ECMAScript modules. In this example, `a.mjs` directly exports `foo` and `f`, rather than writing to a special `module.exports` object. Then, `b.mjs` directly imports the names with the `import` statement instead of loading an object.

TypeScript supports both CommonJS and ECMAScript modules, depending on the project configuration. However, CommonJS modules in TypeScript are untyped; specifically, `require` is typed as a function that returns `any`. 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, with ECMAScript modules, the `import` statement preserves the type annotations of names it imports.

In order to make use of the most type information available, I prefer using ECMAScript modules in my evaluation. To ensure this, I use the `cjs-to-es6` tool [Lawson, 2016] to transform my dataset to use ECMAScript modules. The conversion tool is not perfect, and in particular has difficulty when `require` is used to dynamically load a module. Some of these cases could be fixed manually, but many are genuine uses of dynamic loading in JavaScript.

3.2.2 Type Annotation Prediction

The next step is to invoke a deep learning model to predict type annotations for a JavaScript project. DeepTyper and LambdaNet require an additional step, which I call *type weaving*, to produce TypeScript, while InCoder and StarCoderBase, with a front end, output TypeScript directly.

I used the pretrained DeepTyper model available from its GitHub repository, which is not identical to the model used in the DeepTyper paper [Helleendoorn et al., 2019]. DeepTyper reads in a JavaScript file, and for each identifier, predicts the top five most likely types, outputting the result in comma-separated values (CSV) format.

I used the pretrained LambdaNet model available from its GitHub repository, specifically the model that supports user-defined types [Wei et al., 2020a]. The model reads in a directory containing a JavaScript project, and predicts the top five most likely types for each variable and function declaration. I modified LambdaNet to output in CSV format.

DeepTyper predicts types for all identifiers in the program, including program locations that do not allow type annotations. Therefore, type weaving must also ensure that type annotations are applied correctly, i. e., only to variable declarations, function parameters, and function results. LambdaNet predicts types for variable and function declarations, and in the correct locations; however, type weaving is still required to produce TypeScript code. The InCoder and StarCoderBase front ends do not require type weaving, but only support type predictions for function parameters. I used the pretrained InCoder [Fried et al., 2022] and StarCoderBase [Li et al., 2023b] models, both available from Hugging Face.

3.2.2.1 Type Prediction Front End

INCODER. InCoder is trained to generate code in the middle of a program, conditioned on the surrounding code. To train on a single example (i. e., a file of code), the training procedure replaces a randomly selected contiguous span of tokens with a *mask sentinel* token. It appends the mask sentinel to the end of the example, followed by the tokens that were replaced and a special *end-of-mask* token. The model is then trained as a left-to-right language model. This approach generalizes to support several, non-overlapping masked spans, and its training examples have up to 256 randomly selected masked spans, though the majority have just a single masked span.

In principle one could give InCoder a program with up to 256 types to generate at once. However, I found that InCoder is more successful

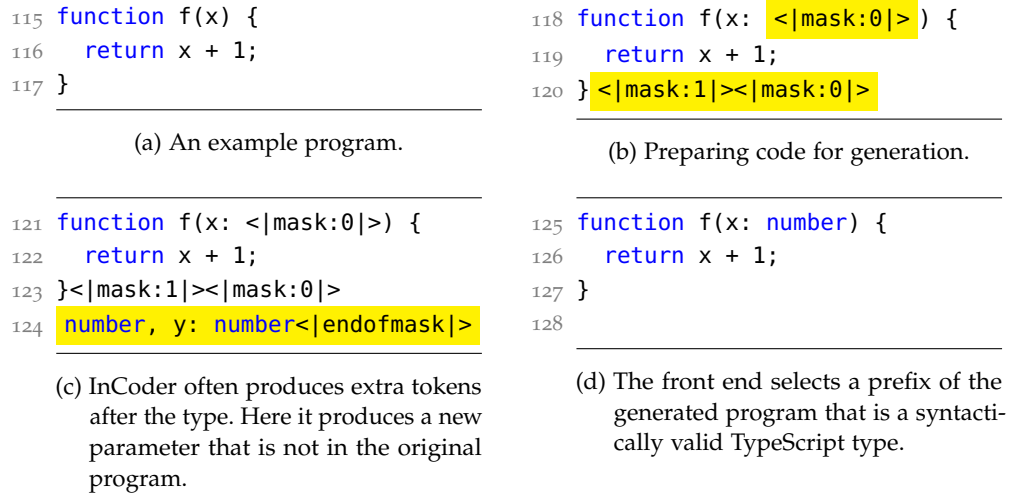


Figure 3.3: Generating types with InCoder.

generating a single type at a time, and with a limited amount of context. Generating a type annotation with InCoder involves the following steps:

1. insert the mask sentinel token at the insertion point;
2. add the mask sentinel to the end of the file;
3. generate at the end of the file until the model produces the end-of-mask token;
4. move the generated text to the insertion point; and
5. remove all sentinels.

Figure 3.3 shows an example of generating a type annotation. However, InCoder will frequently generate more than just a single type. For instance, Figure 3.3c shows an example where InCoder generates a new parameter that is not in the original program. The simplest approach is to reject this result and get InCoder to re-generate completions until it produces a type. However, it is far more efficient to accept a prefix of the generated code if it is a syntactically valid type, which is checked with a TypeScript parser in the generation loop.

STARCODERBASE. The FIM training procedure for StarCoderBase is similar to InCoder; however, the sentinel tokens are different. As a result, the type prediction front end must be adapted to use those tokens, but otherwise uses the same strategy to generate and parse type annotations.

3.2.3 *Type Weaving*

To produce type-annotated TypeScript code, I use a process I call *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 CSV file with type predictions. The type weaving program parses the JavaScript source into an abstract syntax tree (AST), and then traverses the AST and CSV files simultaneously, using the TypeScript compiler to insert type annotations into the program AST. Both DeepTyper and LambdaNet require type weaving, but their CSV files are in different formats. My type weaving program can be extended to support custom CSV formats.

3.2.3.1 *DeepTyper*

Each row of a DeepTyper CSV file represents a lexical token from the source program. Rows with non-identifier tokens, such as keywords and symbols, contain two columns: the token text and the token type. Rows with identifiers contain columns for the token text, token type, as well as the top five most likely types and their probabilities.

The DeepTyper implementation has a few limitations that are handled during type weaving. First, the implementation uses regular expressions instead of a parser to tokenize JavaScript code. This results in some tokens that are missing or incorrectly classified as identifiers. Second, DeepTyper provides type predictions for every occurrence of an identifier, so type weaving must use only the predictions for declarations. Finally, DeepTyper often predicts `complex` as a type; these do not appear to refer to a complex number type, so `any` is used instead.

My type weaving algorithm works as follows: as it traverses the program AST, if it encounters a declaration node, it queries the CSV file for a type prediction. However, the DeepTyper format does not record source location information and the token classification is brittle, so it is not straightforward to identify which rows are actually declarations and which rows should be skipped. My algorithm searches the CSV file for a short sequence of rows that corresponds to the declaration node in the AST. This algorithm works well in practice, but does not handle optional parameters or statements that declare multiple variables.

3.2.3.2 *LambdaNet*

For each declaration, LambdaNet prints the source location of the identifier (start line, start column, end line, and end column), followed by the top five most likely types and their probabilities. I modified LambdaNet to output in CSV format.

LambdaNet frequently predicts the following types: `Number`, `Boolean`, `String`, `Object`, and `Void`. The first four are valid TypeScript types, but are non-primitive boxed types distinct from `number`, `boolean`, `string`, and `object`. The TypeScript documentation strongly recommends using the lowercase type names [Microsoft Corp., 2019], so type weaving normalizes those types. Furthermore, `Void` is not a valid type, so type weaving normalizes it to `void` instead. Finally, LambdaNet does not support generic types, but will predict them without type arguments, which is not valid in TypeScript. While type weaving cannot fix every generic type, it normalizes `Array` to `any[]`, which is shorthand for `Array<any>`.

As the type weaving program traverses the program `AST`, if it encounters a declaration node, it computes the node's source location information, and uses that to query the `CSV` file for a type prediction. However, the type annotation cannot be applied directly to the declaration node, as this modifies the `AST` and invalidates source location information. Therefore, type weaving for LambdaNet occurs in two phases. In the first phase, the traversal does not modify the `AST`, but saves the declaration node and type prediction in a map. Then, in the second phase, type weaving iterates over the map and updates the `AST`.

3.2.4 Type Checking

In the final step, `TYPEWEAVER` runs the TypeScript compiler to type check the migrated projects. The compiler runs on each project, providing all the TypeScript input files as arguments, and setting the following compiler flags:

```
--noEmit    Type check only, do not emit JavaScript
--esModuleInterop  Improve handling of CommonJS and ECMAScript
                  modules
--moduleResolution node  Explicitly set the module resolution strategy
                          to Node.js
--target es6    Enable ECMAScript 6 features, which are used by some
                packages
--lib es2021,dom  Include ECMAScript 2021 library definitions and
                  browser Document Object Model (DOM) definitions
```

`TYPEWEAVER` does not set the `--strict` flag, allowing the type checker to be more lenient in certain situations, which should already be a significant challenge for automated type migration. Furthermore, I ensure that package dependencies are properly included in the dataset so that the compiler can resolve them.

Table 3.1: Summary of TYPEWEAVER dataset categories: number of packages, files, and lines of code (LOC).

Dataset category	Packages	Files	LOC
DefinitelyTyped, no dependencies	282	2,653	121,137
DefinitelyTyped, with dependencies	83	652	61,229
Never typed, no dependencies	101	250	19,565
Never typed, with dependencies	40	544	19,189
<i>Overall</i>	506	4,099	221,120

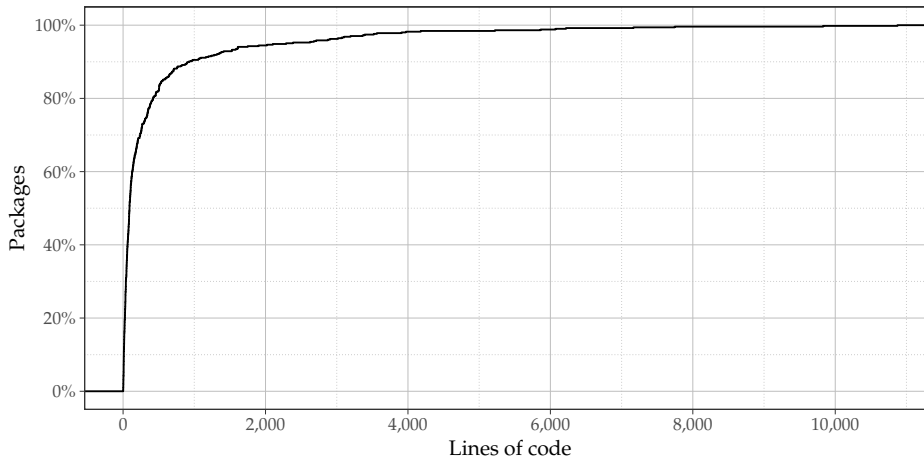


Figure 3.4: Empirical cumulative distribution function of LOC per package, over all datasets. The x-axis shows LOC and the y-axis shows the proportion of packages with fewer than x lines of code.

3.3 EVALUATION

3.3.1 Dataset

The TYPEWEAVER evaluation dataset consists of 506 JavaScript packages. To build this dataset, I start from the top 1,000 most downloaded packages from the npm Registry (as of August 2021) and narrow and clean as follows:

1. I add any transitive dependencies that are not in the original set of packages, to ensure that the dataset is closed.
2. I try to fetch the original source code of every package, and eliminate any package where this is not possible (e. g., the package did not provide a repository URL or was deleted from GitHub). Fetching the

source helps obtain original code, and not compiled or “minified” JavaScript.

3. I remove packages that were built from a “monorepo,” i. e., a single repository containing multiple packages that are published separately. For example, the Babel JavaScript compiler has over 100 separate packages, but all share the same monorepo; fetching each source package meant downloading the entire monorepo multiple times and including unnecessary packages.
4. I remove packages that were not implemented in JavaScript, do not contain code, or have more than 10,000 lines of code. The size limit helps avoid timeouts, and mostly excludes large toolchains and standard libraries, such as the TypeScript compiler and `core-js` standard library.
5. I remove testing code from every package. Tests frequently require extra dependencies, and different frameworks set up the test environment in different ways, which makes large-scale evaluation harder. To remove testing code, I deleted directories named `test`, `tests`, `__tests__`, or `spec`, and files named `test.js`, `tests.js`, `test-*.js`, `*-test.js`, `*.test.js`, or `*.spec.js`.
6. Finally, I ensure that every package has *no dependencies*, or that *all its dependencies are typed*, meaning the dependencies have TypeScript type declaration (`.d.ts`) files available. (I do not require that *packages* are typed, but only that their *dependencies* are.) 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 [DefinitelyTyped contributors, 2012] contains interface type declarations for many popular JavaScript packages, and a handful of packages include their own. I download type declarations of project dependencies and include them in the dataset for evaluation purposes—they are not used for type prediction.

After filtering and cleaning the dataset, I classify each package with two criteria: (1) whether the package has type declarations available; and (2) whether the package has dependencies.

If a package has type declarations available, I say it is “DefinitelyTyped” and use its type annotations as ground truth in the evaluation.³ Otherwise, I use the term “never typed”: these packages *have never been type annotated*

³ However, there is evidence that some of these type annotations are incorrect [Feldthaus and Møller, 2014; Kristensen and Møller, 2017a; Kristensen and Møller, 2017b; Williams et al., 2017; Hoeflich et al., 2022].

and thus no ground truth exists, so machine learning models have never been evaluated on these packages before. If a package has dependencies, I classify it as “with dependencies” (and from the filtering, this means every dependency is typed); otherwise, I classify the package as “no dependencies.” Thus, there are four dataset categories; I list them in [Table 3.1](#) along with the number of packages, files, and lines of code for each category.

[Figure 3.4](#) is an empirical cumulative distribution function of the lines of code per package: the x -axis shows lines of code in a package and the y -axis shows the proportion of packages with fewer than x lines of codes. From the graph, we observe that approximately 90% of packages have fewer than 1,000 lines of code, and approximately 95% of packages have fewer than 2,000 lines of code.

3.3.2 TypeScript Built-in Type Inference

The TypeScript compiler can be configured to generate `.d.ts` TypeScript type declaration files. This uses the compiler’s built-in type inference to generate type annotations for module interfaces. In other words, it only annotates functions and constants that are explicitly exported by the module, and none of the internal, private definitions. Furthermore, the compiler attempts to infer type-correct annotations, even if it means inferring any.

Importantly, the TypeScript compiler does not generate actual TypeScript code, i. e. `.ts` files, that can be type checked.⁴ Therefore, the TypeScript compiler cannot be used for a full JavaScript-to-TypeScript migration, and is not comparable to the type prediction models I evaluate.

Nevertheless, for certain experiments, I compare the type prediction models to the TypeScript compiler baseline. These experiments involve trivial type annotations ([Section 3.3.3.3](#)) and accuracy ([Section 3.3.3.4](#)).

To generate type declaration files, `TYPEWEAVER` invokes the TypeScript compiler on each JavaScript project, with the following flags:

- `--declaration` Generate `.d.ts` files for each JavaScript (and TypeScript) file given as input
- `--allowJs` Allow JavaScript files to be processed by the compiler
- `--emitDeclarationOnly` Output `.d.ts` files and not JavaScript files (which would overwrite the existing JavaScript files)
- `--esModuleInterop`

⁴ For a file `a.js`, the TypeScript compiler generates `a.d.ts`, which cannot be type checked on its own. However, if a file `b.js` imports `a.js`, then the type-annotated declarations in `a.d.ts` are checked against calls in `b.js`.

```
--moduleResolution node
--target es6
--lib es2021,dom
```

The last four flags are the same as the ones described in [Section 3.2.4](#). Running the TypeScript compiler with these flags will output `file.d.ts` in the same directory as `file.js`, where `file.js` is given to the compiler.

3.3.3 Success Rate of Type Checking

3.3.3.1 Do Migrated Packages Type Check?

The first question to ask is whether entire packages type check after automated migration from JavaScript to TypeScript. However, not all packages successfully translate to TypeScript with every migration tool; some packages cause the type migration tool to time out or error. Thus, I report the success rate of type checking as a fraction of packages that successfully translate to TypeScript.

[Table 3.2](#) and [Figure 3.5](#) show the fraction of packages that type check with each tool. DeepTyper and InCoder perform similarly (22–24% success rate), LambdaNet performs worse (10% success rate), and StarCoder-Base performs best (31% success rate). Across all tools, packages without dependencies type check at a higher rate than packages with dependencies.

These package-level type checking results are disappointing—but this is a very high standard to meet. Even a single incorrect type annotation causes the entire package to fail. Therefore, I next consider a finer-grained metric that is still useful.

3.3.3.2 How Many Files are Error Free?

As an alternate measure, I consider the percentage of *files with no compilation errors*. Instead of a binary pass/fail outcome, this is a more fine-grained result for a package, which is motivated by observing that TypeScript files are modules with explicit imports and exports. If a file type checks without errors, then it is using all of its internal and imported types consistently. Thus, when triaging type errors, a programmer may (temporarily) set these files aside and focus on the files with compilation errors. However, the programmer may later need to return to a file with no type errors and adjust its type annotations, for example, if a consumer of that file expects a different interface. Examples of this are presented in the case studies, specifically [Sections 3.3.6.2](#) and [3.3.6.3](#).

[Table 3.3](#) and [Figure 3.7](#) present the fraction of files with no compilation errors. The results are more encouraging: using StarCoderBase, 69% of files

Table 3.2: Number and percentage of packages that type check.

- (1) DefinitelyTyped, no dependencies
 (2) DefinitelyTyped, with dependencies
 (3) Never typed, no dependencies
 (4) Never typed, with dependencies
 ✓ = number of packages that type check
 # = total number of packages
 % = percentage of packages that type check

Dataset	DeepTyper			LambdaNet			InCoder			StarCoderBase		
	✓	#	%	✓	#	%	✓	#	%	✓	#	%
(1)	54	226	23.9	24	213	11.3	56	245	22.9	77	245	31.4
(2)	5	53	9.4	1	50	2.0	9	65	13.8	9	65	13.8
(3)	31	86	36.0	11	79	13.9	26	91	28.6	41	91	45.1
(4)	5	36	13.9	3	32	9.4	4	37	10.8	8	37	21.6
<i>Overall</i>	95	401	23.7	39	374	10.4	95	438	21.7	135	438	30.8

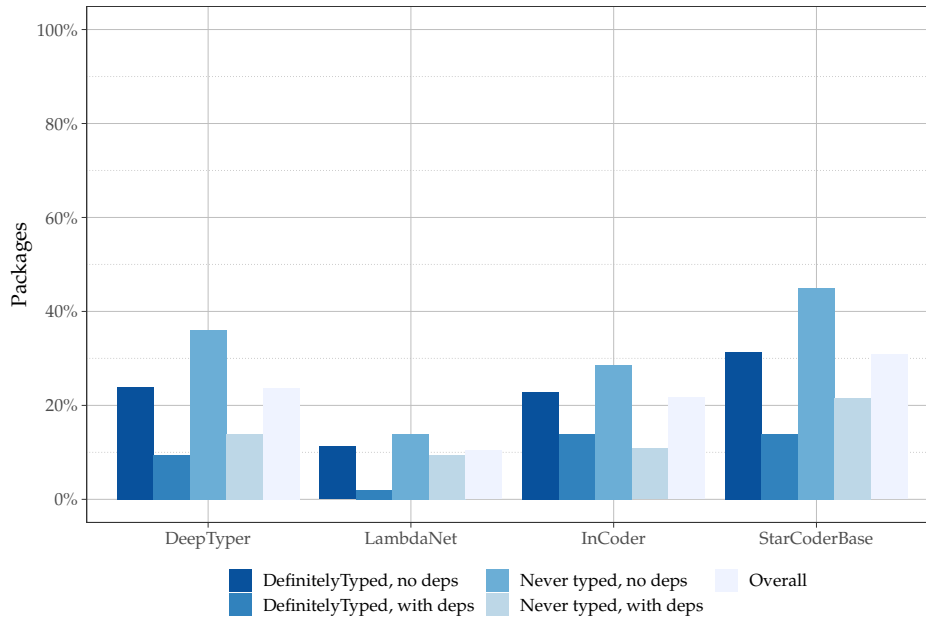


Figure 3.5: Percentage of packages that type check.

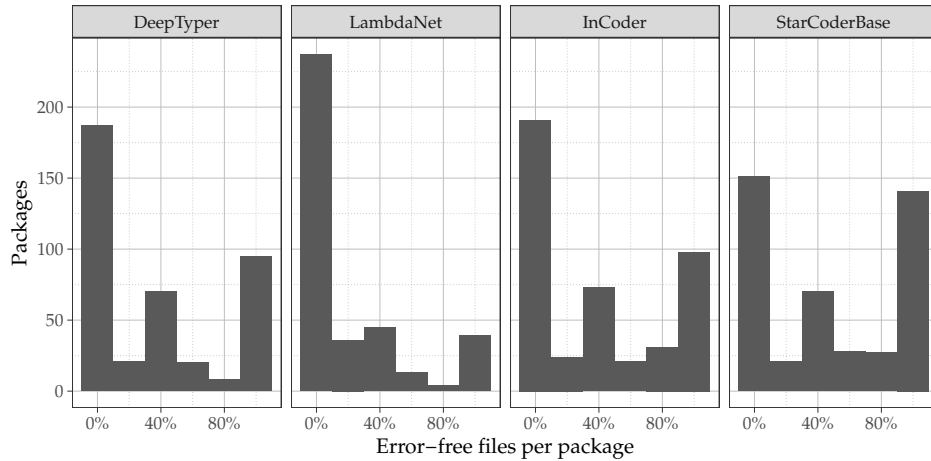


Figure 3.6: Number of packages vs. percentage of error-free files per package.

are error free, and the other models also show improvement. With these results, it is not clear that packages without dependencies outperform packages with dependencies. Finally, [Figure 3.6](#) shows the percentage of error-free files for each package, and plots histograms of the distribution. Across all tools, most packages have type errors in most or all files.

3.3.3.3 What Percentage of Type Annotations Are Trivial?

Next, I examine what percentage of type annotations, *within the error-free files*, are trivial, i.e., what percentage are any, any[] (array of anys), or Function (function that accepts any arguments and returns anything). These annotations can hide type errors and allow more code to type check; however, they provide little value to the programmer.

[Table 3.4](#) and [Figure 3.8](#) shows the percentage of trivial type annotations within error-free files. DeepTyper produces the most (about 60%), LambdaNet and the TypeScript compiler produce the least (about 30%), while InCoder, and StarCoderBase are in between (45–48%).

Comparing to the percentage of files with no compilation errors ([Figure 3.7](#)), DeepTyper produces more type-correct code than LambdaNet, but it also generates more trivial type annotations. The TypeScript compiler, InCoder, and StarCoderBase produce the most type-correct code, while generating a moderate percentage of trivial type annotations.

However, the TypeScript compiler result is misleading: [Table 3.4](#) shows that the TypeScript compiler generates an order of magnitude more type annotations than any other system. This is because many packages export dictionaries of values, which are normally not annotated. However, when the TypeScript compiler generates type definitions, it converts dictionary values into exported constants and infers types for those constants. Since

Table 3.3: Number and percentage of files with no compilation errors.

(1) DefinitelyTyped, no dependencies
 (2) DefinitelyTyped, with dependencies
 (3) Never typed, no dependencies
 (4) Never typed, with dependencies
 ✓ = number of files with no compilation errors
 # = total number of files
 % = percentage of files with no compilation errors

Dataset	DeepTyper			LambdaNet			InCoder			StarCoderBase		
	✓	#	%	✓	#	%	✓	#	%	✓	#	%
(1)	337	853	39.5	364	1,328	27.4	913	1,523	59.9	1,049	1,523	68.9
(2)	73	195	37.4	52	257	20.2	219	424	51.7	220	424	51.9
(3)	91	185	49.2	60	198	30.3	97	221	43.9	116	221	52.5
(4)	42	116	36.2	25	529	4.7	474	539	87.9	477	539	88.5
<i>Overall</i>	543	1,349	40.3	501	2,312	21.7	1,703	2,707	62.9	1,862	2,707	68.8

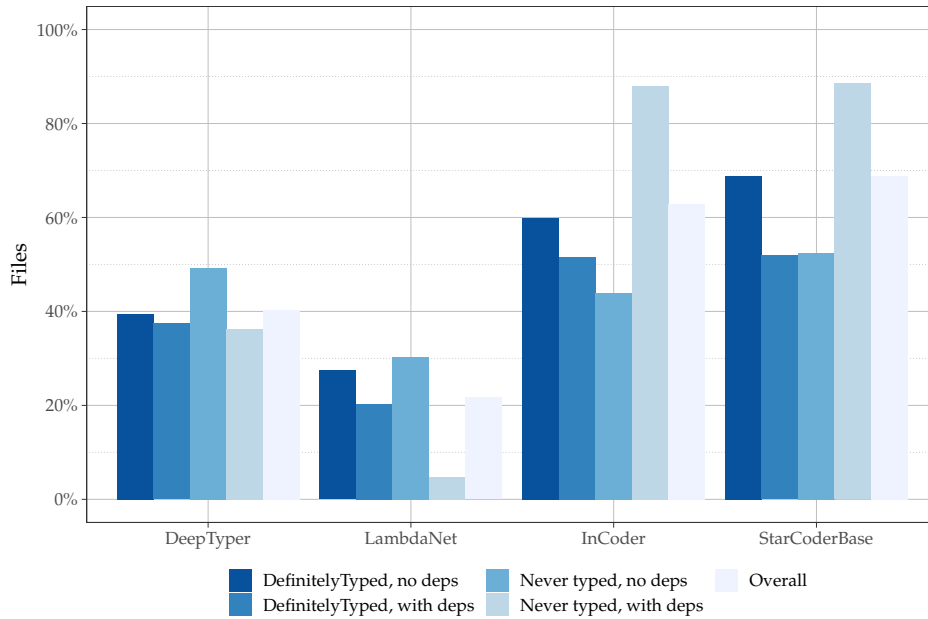


Figure 3.7: Percentage of files with no compilation errors.

Table 3.4: Number and percentage of type annotations that are any, any[], or Function, in files with no errors.

(1) DefinitelyTyped, no dependencies
 (2) DefinitelyTyped, with dependencies
 (3) Never typed, no dependencies
 (4) Never typed, with dependencies
 ✓ = number of trivial type annotations
 # = total number of type annotations
 % = percentage of trivial type annotations

Dataset	tsc			DeepTyper			LambdaNet			InCoder			StarCoderBase		
	✓	#	%	✓	#	%	✓	#	%	✓	#	%	✓	#	%
(1)	2,480	6,199	40.0	749	1,189	63.0	279	968	28.8	664	1,439	46.1	1,032	1,866	55.3
(2)	357	883	40.4	101	164	61.6	9	43	20.9	288	696	41.4	211	692	30.5
(3)	317	2,807	11.3	204	331	61.6	21	82	25.6	42	67	62.7	117	237	49.4
(4)	131	677	19.4	106	152	69.7	15	66	22.7	14	50	28.0	18	62	29.0
<i>Overall</i>	<i>3,285</i>	<i>10,566</i>	<i>31.1</i>	<i>1,160</i>	<i>1,836</i>	<i>63.2</i>	<i>324</i>	<i>1,159</i>	<i>28.0</i>	<i>1,008</i>	<i>2,252</i>	<i>44.8</i>	<i>1,378</i>	<i>2,857</i>	<i>48.2</i>

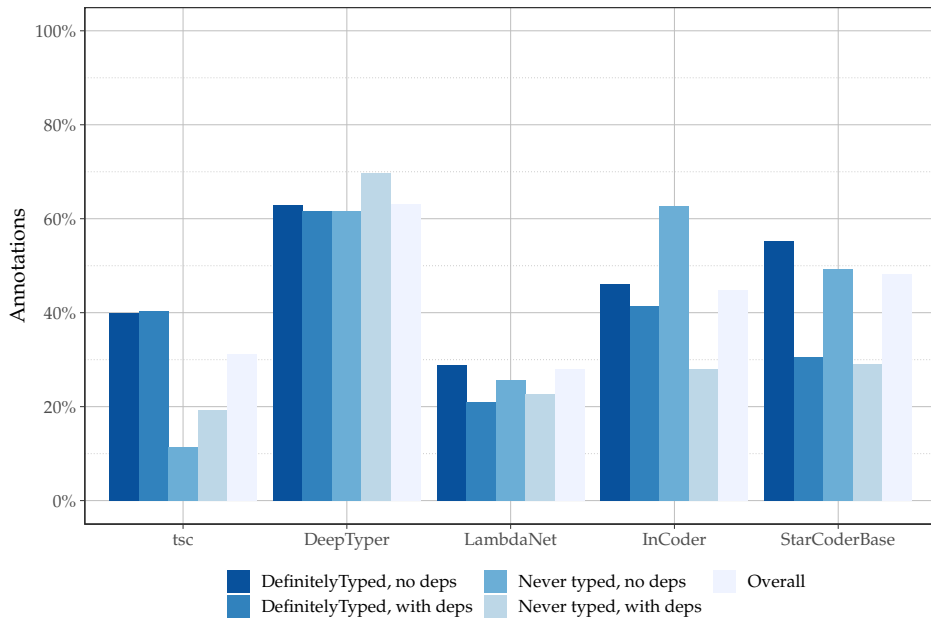


Figure 3.8: Percentage of type annotations that are any, any[], or Function, in files with no errors.

the values are known, the type annotations will not be trivial, which artificially suppresses the percentage of trivial type annotations.

3.3.3.4 *Do Migrated Types Match Human-Written Types (When Available)?*

Since my dataset is constructed from JavaScript packages instead of TypeScript packages, it does not have fully type-annotated files as ground truth; instead, I use declaration files provided by the DefinitelyTyped repository or package author. I configure the TypeScript compiler to emit declarations during type checking, which it can do even if the whole package does not type check. Thus, I compare handwritten, ground truth declarations against declarations generated from migrated packages.

I extract function signatures from declaration files and only compare a signature if it is in both the ground truth and generated declaration. I compare the function parameter types and return types one-to-one, ignoring modifiers (e.g., `readonly`), and require an exact string match (i.e. `string | number` and `number | string` are considered different types). Following the literature, I skip a comparison if the ground truth is the any annotation.

The results are presented in [Table 3.5](#) and [Figure 3.9](#): accuracy is generally better for packages without dependencies. Additionally, these results follow the same pattern in prior work, where LambdaNet has better accuracy than DeepTyper, despite performing worse in other metrics. However, StarCoderBase has the best accuracy, while the TypeScript compiler and InCoder perform poorly.

3.3.3.5 *How Many Errors Occur in Each Package?*

[Figure 3.10](#) shows an empirical cumulative distribution function of errors: the x -axis shows the number of errors and the y -axis shows the proportion of packages with fewer than x errors. For example, when migrating the “DefinitelyTyped, with dependencies” dataset with LambdaNet, approximately 80% of packages have fewer than 250 errors each. Additionally, all of DeepTyper’s packages and almost all of InCoder’s packages have fewer than 500 errors each.

3.3.4 *Error Analysis*

Next, I consider the kinds of errors that arise during migration. Every TypeScript compiler error has an associated code [TypeScript contributors, 2022], making categorization straightforward. [Figure 3.11](#) summarizes the top 10 most common errors and [Table 3.6](#) provides the corresponding messages.

Table 3.5: Accuracy of type annotations, compared to non-any ground truth.

(1) DefinitelyTyped, no dependencies

(2) DefinitelyTyped, with dependencies

✓ = number of matching type annotations

= total number of type annotations

% = percentage of matching type annotations

Dataset	tsc			DeepTyper			LambdaNet			InCoder			StarCoderBase		
	✓	#	%	✓	#	%	✓	#	%	✓	#	%	✓	#	%
(1)	130	436	29.8	85	209	40.7	103	227	45.4	34	105	32.4	72	144	50.0
(2)	41	155	26.5	15	44	34.1	18	40	45.0	5	33	15.2	17	31	54.8
<i>Overall</i>	171	591	28.9	100	253	39.5	121	267	45.3	39	138	28.3	89	175	50.9

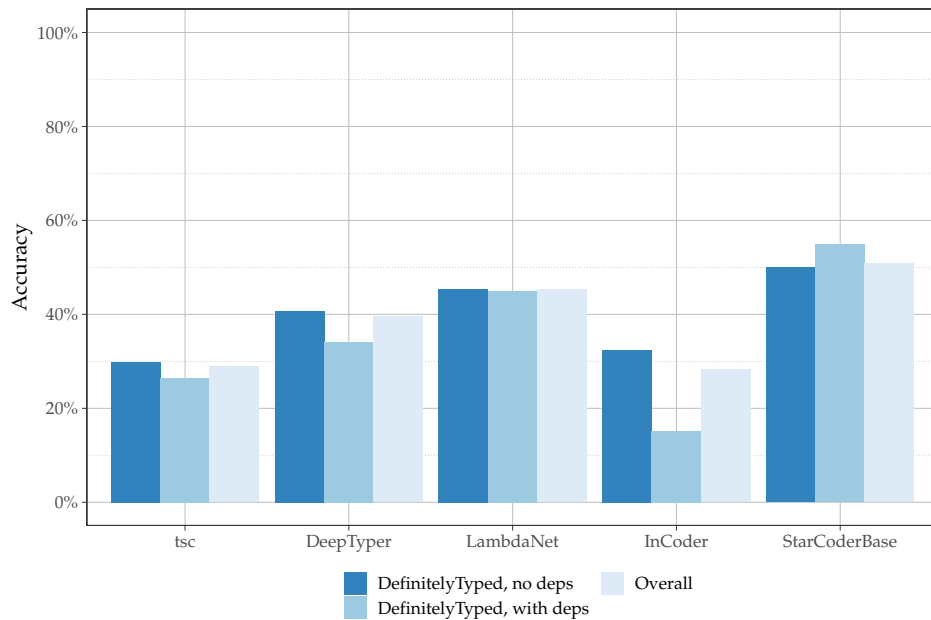


Figure 3.9: Accuracy of type annotations, compared to non-any ground truth.

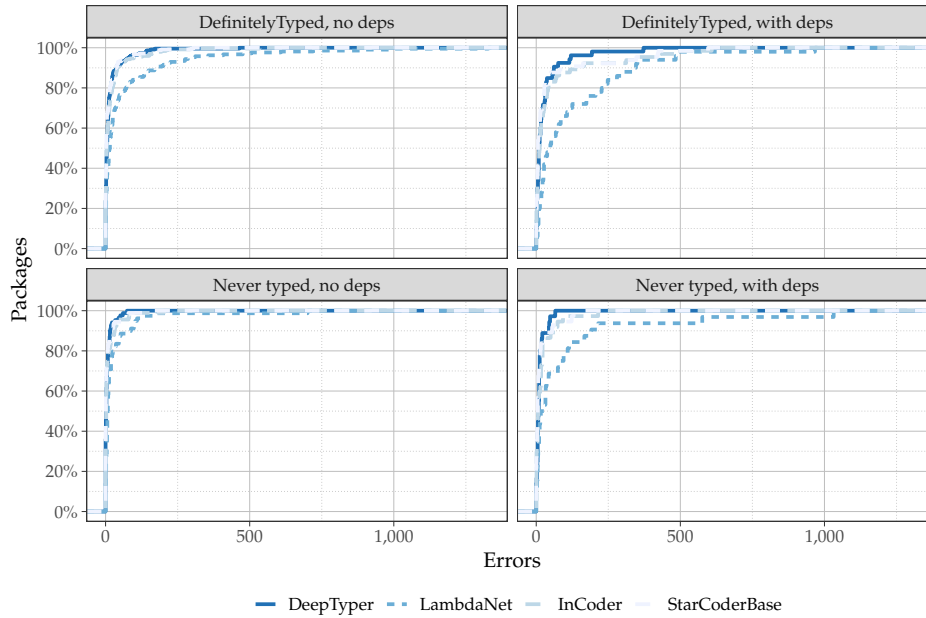


Figure 3.10: Empirical cumulative distribution function of errors per package. The x -axis shows the number of errors and the y -axis shows the proportion of packages with fewer than x errors.

Most of the errors relate to types. These errors are the following: a property not existing on a type (TS2339 and TS2551); an assignment with mismatched types (TS2322); a function call with mismatched parameter and argument types (TS2345); calling a function that was assigned a non-function type annotation (TS2349); and a conditional that compares values from different types (TS2367).

The remaining errors are not directly related to types. TS2304 and TS4078 refer to an unknown name, which may not necessarily be a type. TS2554 is emitted because TypeScript requires the number of call arguments to match the number of function parameters, but JavaScript does not. TS2339 includes cases where an empty object is initialized by setting its properties, but TypeScript requires that the object’s properties are declared in its type. Finally, TS2307 indicates that the ECMAScript module conversion produced incorrect code.

3.3.5 ECMAScript Module Conversion

Recall that there is an optional step before evaluation: converting packages to use ECMAScript modules. In this section, I re-run the evaluation—predicting types, weaving types, and type checking—to compare the results before and after the conversion step. Specifically, I examine the

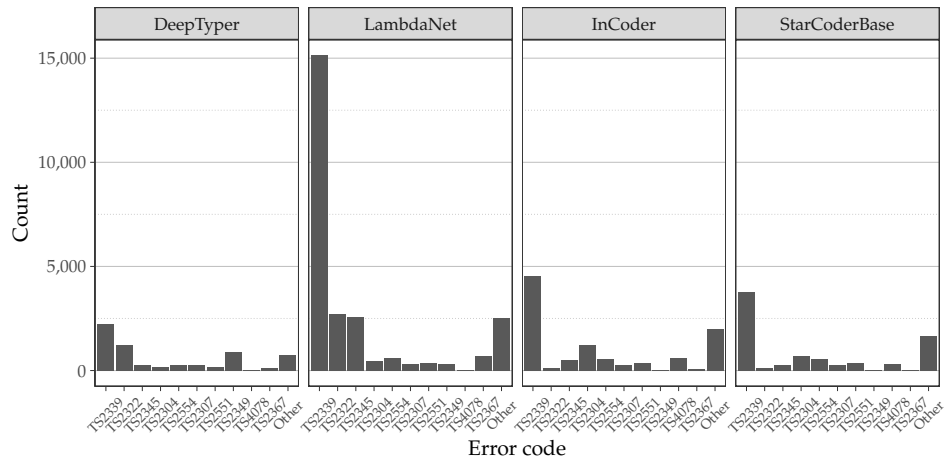


Figure 3.11: Distribution of the top 10 most common error codes, over all datasets.

percentage of packages that type check (Table 3.7), the percentage of files with no errors (Table 3.8), and accuracy (Table 3.9). However, this is not a direct comparison between CommonJS and ECMAScript modules, as some of the original packages were already using ECMAScript modules. Furthermore, the conversion affected a small handful of packages: some packages successfully migrated to TypeScript after the conversion but failed before, and the inverse was true for other packages.

Table 3.7 shows that ECMAScript module conversion makes fewer packages type check for DeepTyper, but slightly improves the results for LambdaNet. For InCoder and StarCoderBase, ECMAScript module conversion has little effect. Figure 3.12 compares packages that type checked before or after the ECMAScript module conversion; packages that never type checked were excluded. In general, if a package type checked before the conversion, it likely type checked after the conversion. However, if a package failed to type check before the conversion, it was unlikely to type check afterwards; in fact, this never happened for a package with dependencies.

Table 3.8 compares the percentage of files with no compilation errors. The conversion improves the results for InCoder and StarCoderBase, but makes the results slightly worse for LambdaNet and much worse for DeepTyper.

The conversion improves the results for LambdaNet and InCoder, but makes the results worse for DeepTyper. One dramatic result is the change for InCoder and the “never typed, with dependencies” dataset, where the ECMAScript module conversion results in 88% of files type checking, when it was only 12% before. The difference is caused by a single package, `regenerate-unicode-properties`, which has over 400 files. With

Table 3.6: The top 10 most common error codes and their messages.
 DT = DeepTyper; LN = LambdaNet; IC = InCoder; SC = StarCoderBase

Error	Message	DT	LN	IC	SC
TS2339	Property 'o' does not exist on type '1'.	2,222	15,130	4,520	3,760
TS2322	Type 'o' is not assignable to type '1'.	1,240	2,692	112	103
TS2345	Argument of type 'o' is not assignable to parameter of type '1'.	241	2,581	522	287
TS2304	Cannot find name 'o'.	166	440	1,225	717
TS2554	Expected 0 arguments, but got 1.	241	582	560	561
TS2307	Cannot find module 'o' or its corresponding type declarations.	269	304	271	271
TS2551	Property 'o' does not exist on type '1'. Did you mean '2'?	173	354	343	337
TS2349	This expression is not callable.	895	292	15	10
TS4078	Parameter 'o' of exported function has or is using private name '1'.	17	18	616	302
TS2367	This comparison appears to be unintentional because the types 'o' and '1' have no overlap.	103	676	91	25
<i>Other</i>		738	2,537	2,000	1,679
<i>Total</i>		6,305	25,606	10,275	8,052

CommonJS modules, each file produces an error; however, with ECMAScript modules, those files type check successfully.

Finally, [Table 3.9](#) compares the accuracy of type annotations, before and after the ECMAScript module conversion. Recall that for accuracy, type annotations are compared against the ground truth of handwritten TypeScript declaration files; these are the “DefinitelyTyped” datasets. Accuracy improves for DeepTyper and LambdaNet, but worsens slightly for InCoder and StarCoderBase.

3.3.6 Case Studies

In this section, I examine how the models performed on four packages, whether the packages type check, and what steps are left to migrate the packages to TypeScript.

Table 3.7: Percentage of packages that type check, before and after ECMAScript module conversion.

- (1) DefinitelyTyped, no dependencies
- (2) DefinitelyTyped, with dependencies
- (3) Never typed, no dependencies
- (4) Never typed, with dependencies

Dataset	DeepTyper		LambdaNet		InCoder		StarCoderBase	
	Before	After	Before	After	Before	After	Before	After
(1)	25.7	23.9	9.0	11.3	21.3	22.9	28.4	31.4
(2)	11.9	9.4	2.9	2.0	14.5	13.8	15.7	13.8
(3)	34.7	36.0	11.6	13.9	26.7	28.6	43.6	45.1
(4)	28.2	13.9	8.8	9.4	22.5	10.8	37.5	21.6
<i>Overall</i>	25.8	23.7	8.5	10.4	21.3	21.7	30.0	30.8

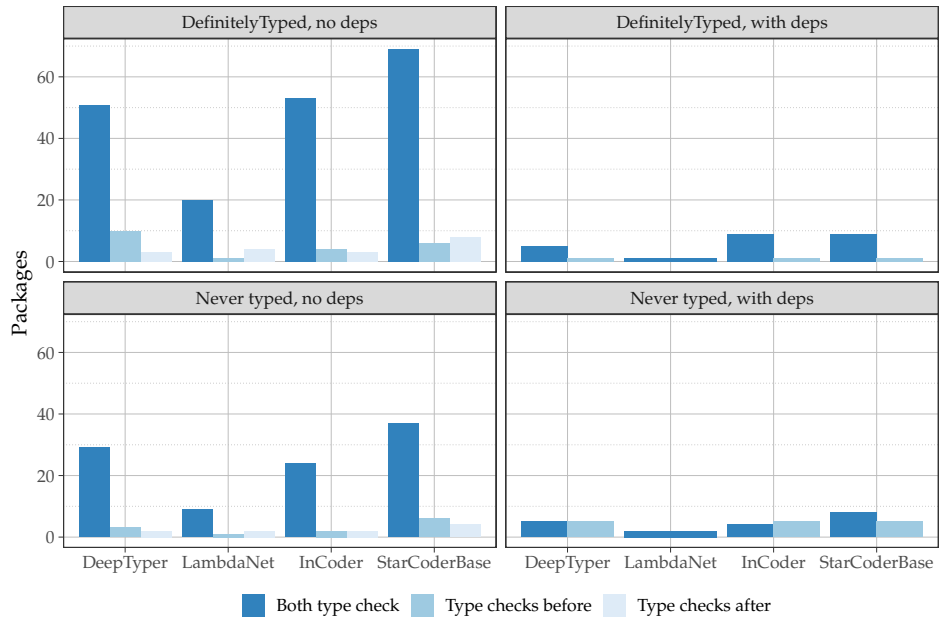


Figure 3.12: Packages that type check before and after ECMAScript module conversion; before but not after conversion; and after but not before conversion.

Table 3.8: Percentage of files with no compilation errors, before and after ECMAScript module conversion.

- (1) DefinitelyTyped, no dependencies
- (2) DefinitelyTyped, with dependencies
- (3) Never typed, no dependencies
- (4) Never typed, with dependencies

Dataset	DeepTyper		LambdaNet		InCoder		StarCoderBase	
	Before	After	Before	After	Before	After	Before	After
(1)	43.7	39.5	26.6	27.4	69.7	59.9	75.3	68.9
(2)	51.2	37.4	24.9	20.2	50.0	51.7	50.9	51.9
(3)	48.8	49.2	25.4	30.3	41.6	43.9	50.4	52.5
(4)	64.5	36.2	8.6	4.7	11.6	87.9	12.3	88.5
<i>Overall</i>	47.5	40.3	22.8	21.7	57.2	62.9	61.6	68.8

Table 3.9: Accuracy of type annotations, before and after ECMAScript module conversion.

- (1) DefinitelyTyped, no dependencies
- (2) DefinitelyTyped, with dependencies

Dataset	DeepTyper		LambdaNet		InCoder		StarCoderBase	
	Before	After	Before	After	Before	After	Before	After
(1)	35.6	40.7	44.4	45.4	33.6	32.4	55.5	50.0
(2)	25.0	34.1	35.7	45.0	19.0	15.2	51.9	54.8
<i>Overall</i>	31.7	39.5	42.3	45.3	28.6	28.3	54.5	50.9

```

129 // Original
130 const handlePreserveConsecutiveUppercase =
131   (decamelized, separator) => {
132     // code omitted and simplified
133     return decamelized.replace(
134       /([A-Z]+)([A-Z][a-z]+)/gu,
135       (_, $1, $2) => $1 + separator + $2.toLowerCase(),
136     );
137   }
138
139 // Elsewhere in the package; str and sep are both strings
140 return handlePreserveConsecutiveUppercase(str, sep);
141
142 // DeepTyper solution
143 const handlePreserveConsecutiveUppercase: string =
144   (decamelized: string, separator: string) => { ... }
145
146 // LambdaNet solution
147 const handlePreserveConsecutiveUppercase: Function =
148   (decamelized: string, separator: number) => { ... }

```

Figure 3.13: The `handlePreserveConsecutiveUppercase` function adapted from the `decamelize` package. The DeepTyper and LambdaNet solutions are also shown.

3.3.6.1 Error Message Does Not Refer to Incorrect Type Annotation

`decamelize` is a package for converting strings in camel case to lowercase.⁵ It is in the “DefinitelyTyped, no dependencies” dataset, as it ships with a `.d.ts` declaration file and has no dependencies. Figure 3.13 shows a simplified version of the function `handlePreserveConsecutiveUppercase`. This function is not exported, thus there are no programmer-written type annotations. Line 131 uses JavaScript’s arrow function notation to define a function that takes two arguments, and assigns it to the constant on line 130. Elsewhere in the package (line 140 in the listing), the helper function is called with `str` and `sep` string arguments.

A programmer inspecting the function can reason that line 133 is a call to a string method that uses a regular expression on line 134 to replace text in `decamelized` with the result on line 135, where `separator` is concatenated with the regular expression match. Therefore, both the `decamelized` and `separator` parameters on line 131 should be annotated as `string`.

The DeepTyper solution is listed on line 143: it correctly annotates both function parameters as `string`, but incorrectly annotates `handlePreserveConsecutiveUppercase` as `string`. The compiler emits errors for lines 140

⁵ <https://www.npmjs.com/package/decamelize>

```

149 // Original
150 export const write =
151   function (buffer, value, offset, isLE, mLen, nBytes) { ... }
152
153 // Ground truth signature
154 export function write(
155   buffer: Uint8Array, value: number, offset: number, isLE: boolean,
156   mLen: number, nBytes: number): void;
157
158 // DeepTyper solution
159 export const write: void = function (
160   buffer: Buffer, value: number, offset: number, isLE: number,
161   mLen: number, nBytes: number) { ... }
162
163 // InCoder solution
164 export const write = function (
165   buffer: Buffer, value: any, offset: number, isLE: boolean,
166   mLen: number, nBytes: number) { ... }

```

Figure 3.14: The write function adapted from the `ieee754` package. The ground truth signature is also shown, along with the DeepTyper and InCoder solutions.

and [line 144](#), because [line 140](#) is attempting to call a non-function, and [line 144](#) is attempting to assign a function to a non-function variable. However, the fix must be applied to the annotation on [line 143](#).

3.3.6.2 *Incorrect Type Annotation Can Type Check Successfully*

The LambdaNet solution on [line 147](#) correctly annotates `handlePreserveConsecutiveUppercase` as `Function`, but it incorrectly annotates the separator parameter on [line 148](#) as `number`. A programmer might expect the compiler to emit a type error, since [line 140](#) calls the function with string arguments. However, the code type checks successfully, because the generic `Function` type on [line 147](#) accepts any number of arguments of any type. The `Function` type annotation is similar to `any`, in that it enables more code to type check, but at the cost of fewer type guarantees.

Another example of this problem is the `ieee754` package, which reads and writes floating point numbers to and from buffers.⁶ It is categorized as “DefinitelyTyped, no dependencies,” since it provides a `.d.ts` declaration file and has no dependencies. [Figure 3.14](#) shows the original declaration for the `write` function on [line 150](#), and the handwritten, ground truth signature on [line 154](#).

⁶ <https://www.npmjs.com/package/ieee754>

```

167 // LambdaNet solution
168 export default function (thingToPromisify: string) {
169   if (typeof thingToPromisify === 'function') {
170     return promisify(thingToPromisify)
171   }
172   throw new TypeError('Can only promisify functions or objects')
173 };

```

Figure 3.15: The LambdaNet solution for a function adapted from the @gar/promisify package.

Consider the DeepTyper solution: the compiler emits an error on [line 159](#), because a function is being assigned to a variable of type `void`. However, even if that error is fixed, there is another, more subtle error not detected by the compiler: the `isLE` parameter on [line 160](#) is incorrectly annotated as `number`, not `boolean`. Because this is compatible with the body of the function, there is no error.⁷

The InCoder solution on [line 164](#) type checks successfully. It also uses the `Buffer` type for `buffer`, and it uses `any` instead of `number` for the `value` parameter on [line 165](#). However, the `any` annotation may cause run-time errors if the function is called with arguments of the wrong type.

3.3.6.3 Run-Time Type Assertions

The @gar/promisify package,⁸ simplified and shown in [Figure 3.15](#), is another example where a program type checks, but is incorrect. The example exports a function that takes an argument `thingToPromisify`, type annotated as `string` by LambdaNet. [Line 169](#) performs a run-time type check with the `typeof` operator. This ensures that `thingToPromisify` is a function on [line 170](#), which is what the `promisify` function, defined by Node.js, expects. If `thingToPromisify` is not a function, the exception on [line 172](#) is thrown.

The example type checks successfully, because the TypeScript compiler treats the `typeof` check as a type guard, and reasons that on [line 170](#), the `thingToPromisify` variable has been narrowed [Microsoft Corp., 2020] to a more specific type. However, because `thingToPromisify` is annotated as `string`, the type guard always returns `false`. Therefore, [line 170](#) is actually unreachable, so the exception on [line 172](#) is always thrown.

⁷ The `Buffer` annotation is valid, despite not matching the ground truth `Uint8Array`, because `Buffer` is defined by the Node.js standard library as a subtype of `Uint8Array`.

⁸ <https://www.npmjs.com/package/@gar/promisify>

```

174 export default function(arr: any[]) {
175   var len: number = arr.length;
176   var o: object = {};
177   var i: number;
178
179   for (i = 0; i < len; i += 1) { ... }
180
181   for (i in o) { ... }
182 };

```

Figure 3.16: The LambdaNet solution for a function adapted from the array-unique package.

3.3.6.4 Variable Used as Two Different Types

The example in Figure 3.16 is adapted from the array-unique package.⁹ The example contains two for loops: a traditional, counter-based for loop on line 179, and a for...in loop on line 181 that iterates over all enumerable string properties of an object. Both loops share the same loop variable, *i*, defined on line 177 and annotated as *number* by LambdaNet.

The use of *i* on line 181 causes a type error, as for...in loops require the loop variable to be *string*. However, changing the annotation on line 177 to *string* causes a type error on line 179, as counter-based for loops require the loop variable to be *number*. One solution is to use the *any* annotation, and another is to use the union type *number | string*. Ultimately, the correct solution is to define separate loop variables; this example highlights that code written in JavaScript may need to be refactored for TypeScript.

3.4 DISCUSSION

HOW SHOULD TYPE PREDICTION MODELS BE EVALUATED? Prior work has used accuracy to evaluate type prediction models, but I argue that we should instead type check the generated code. My methodology makes it possible to evaluate performance on code without known type annotations, i. e., code that has never been typed before. In contrast, prior work required the benchmarks to have ground truth type annotations. My approach also reduces the likelihood of training data leaking into the test set.

One limitation of my approach is that code may type check with trivial type annotations (e. g., *any* or *Function*) that provide little benefit to the programmer. Furthermore, type correctness does not necessarily mean the type annotations are correct: *any* can hide type errors that are only

⁹ <https://www.npmjs.com/package/array-unique>

encountered at run time. Therefore, it is also important to measure the proportion of trivial annotations.

CAN SLIGHTLY WRONG TYPE ANNOTATIONS BE USEFUL? A type prediction model may suggest types that are slightly wrong and easily fixable by a programmer, but fail to type check. However, a tool that produces hundreds or thousands of slightly wrong type annotations would overwhelm the programmer, and I believe it is important to build tools that try to produce fewer errors. On the other hand, slightly wrong type annotations may still provide value, but “slightly wrong” needs to be defined and measured. Without a tool like `TYPEWEAVER`, which weaves type annotations into code and type checks the result, it would not be possible to ask these questions.

SHOULD EVALUATION USE JAVASCRIPT OR TYPESCRIPT PROGRAMS? In this chapter, I chose to evaluate on JavaScript programs, so my dataset deviates from prior work, which only considered TypeScript. The motivating problem is not to recover type annotations for TypeScript programs that already type check, but to migrate untyped JavaScript programs to type-annotated TypeScript. For this problem, type prediction on its own is not enough, and other steps and further refactoring may be required.

However, there may be scenarios where a type prediction model is used to generate type annotations for a partially annotated TypeScript project. In these situations, type migration would likely not require additional refactoring steps.

CAN TYPE MIGRATION BE FULLY AUTOMATED? My results show that automatically predicting type annotations is a challenging task and much work remains to be done. Furthermore, migrating JavaScript to TypeScript involves more than just adding type annotations: the two languages are different and some refactoring may be required. The models I evaluate in this dissertation do not refactor code, and I believe it is unlikely for automated type migration to be perfect. Thus, some manual refactoring will always be necessary for certain kinds of code, but I believe that tools can reduce the overall burden on programmers.

TRAINING TYPE PREDICTION MODELS

Large language models (LLMs) have been successful at a variety of code generation tasks, and fill in the middle (FIM), where the model generates code in the middle of a program, conditioned on the surrounding context, is a natural fit for type prediction. However, my colleagues and I find several challenges that prevent these models from working out of the box [Cassano, Yee, et al., 2023b]. First, FIM 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 [Yee and Guha, 2023a; Pradel et al., 2020]. 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 [Shi et al., 2023; Sun et al., 2021].

There are two problems specific to LLMs that we must address. First, a language model can only accept a limited number of tokens as input. (Recall that a tokens are the basic units of input and output for an LLM, 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, FIM can generate unwanted code, as discussed in Section 3.2.2.1. For example, consider a single-parameter function that is given to a model:

```
183 function f(x) {  
184     return x + 1;  
185 }
```

However, the model produces the following:

```
186 function f(x: number, y: number) {  
187     return x + 1;  
188 }
```

In the output, `x` is correctly annotated as `number`, but an additional parameter `y` is generated.

To address these issues, my colleagues and I built `OPENTAU` [Cassano, Yee, et al., 2023b; Cassano, Yee, et al., 2023a], which handles the large context problem by recursively decomposing a program into smaller contexts, and then running inference on the respective subprograms. `OPENTAU` handles the combinatorial explosion problem that naturally arises from deep and wide trees, and uses local type inference for simple variable declarations. Furthermore, our work proposes a new evaluation methodology for gradual type migration that measures program *typedness*, the degree to which migrated programs contain type information. Additionally, we introduce fill in the type (`FIT`), a new fine-tuning approach that adapts `FIM` for training type prediction.

4.1 OVERVIEW

Programs are often large and complex, and may not fit into a model’s context window. Even in emerging models with larger context windows, performance may be poor as the relevant context for an arbitrary type can be spread across long sequences within a program. Furthermore, a model may predict multiple annotations for each type annotation site, leading to a combinatorial explosion.

To make the problem tractable, `OPENTAU` decomposes the input program into a tree, with each node representing a code block. Next, it traverses the tree in bottom-up level order, visiting child nodes before their parents. It generates candidate solutions for each node, where a candidate is a type-annotated code block. This step includes child candidates as context for type prediction of the parent node. The traversal continues until reaching the root node, where it produces a collection of fully typed program candidates. Finally, `OPENTAU` scores and ranks the program candidates, returning the best solution as the final, fully typed program.

DECOMPOSITION. [Figure 4.1a](#) shows a TypeScript program with type annotation sites denoted by `_hole_`. The tree representation is shown in [Figure 4.1b](#) and follows the structure of the program. Functions `hello` and `helloGen` are defined at the top level, so their nodes are under the root. `helloHelper` is nested within `helloGen`, so it is a child node of `helloGen`. Finally, variable declarations `greeting` and `suffix` are grouped into `varNode1`.

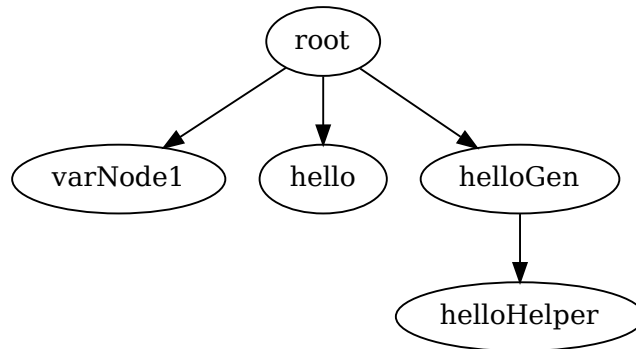
TREE TRAVERSAL. After decomposing the program, `OPENTAU` traverses the tree representation and generates type predictions for each code block. It starts with `helloHelper` (a leaf node) and builds a prompt for the model. The prompt is composed of the original text of `helloHelper` with the type annotations masked with `_hole_`, as shown in [Figure 4.2a](#). Then, the model

```

189 let greeting: _hole1_ = "Hello";
190 let suffix: _hole2_ = "!";
191
192 // Produces a greeting for the given name
193 const hello = (name: _hole3_): _hole4_ => {
194   return greeting + " " + name;
195 };
196
197 function helloGen(name: _hole5_): _hole6_ {
198   const helloHelper = (): _hole7_ => {
199     return hello(name) + suffix;
200   };
201   return helloHelper;
202 }

```

(a) An example TypeScript program, with holes inserted.



(b) Tree representation of the program.

Figure 4.1: A TypeScript program and its tree representation. The unannotated program is provided as input to OPENTAU.

infers a set of type annotations for the node and infills each `_hole_` with its corresponding type annotation, labeling this result the candidate solution, which is shown in [Figure 4.2b](#).

Next, the traversal continues one level up and produces candidate solutions for `hello`, `helloGen`, and `varNode1`. `hello` is a leaf node, so OPENTAU infers type annotations in the same fashion as `helloHelper`. `varNode1` contains variable declarations, which will be handled in the parent node.

`helloGen` is treated differently, as it contains `helloHelper` as a child node and must consider its candidate solutions as context. In this example, there is only one candidate. OPENTAU incorporates `helloHelper`'s candidate solution into `helloGen`'s prompt, resulting in [Figure 4.3a](#). In this example, the model generates two candidate solutions for `helloGen`. For brevity,

```

203 const helloHelper = (): _hole_ => {
204   return hello(name) + "!";
205 };

```

(a) Prompt.

```

206 const helloHelper = (): string => {
207   return hello(name) + suffix;
208 };

```

(b) Candidate solution.

Figure 4.2: Prompt and candidate solution for helloHelper.

<pre> 209 function helloGen(name: _hole5_): _hole6_ { 210 const helloHelper = (): string => { 211 return hello(name) + suffix; 212 }; 213 214 return helloHelper; 215 } </pre>	<pre> // Solution 1 string // _hole5_ () => string // _hole6_ // Solution 2 string // _hole5_ Function // _hole6_ </pre>
---	--

(a) Prompt.

(b) Type annotations.

Figure 4.3: Prompt and type annotations for helloGen.

only the type annotations are shown in [Figure 4.3b](#); they are substituted for the holes in [Figure 4.3a](#) to produce helloGen’s candidate solutions.

Finally, the traversal reaches the root node. To produce candidate solutions for the entire program, OPEN_{TAU} considers the candidate solutions from varNode1, hello, and helloGen. varNode1 contains variable declarations, so OPEN_{TAU} invokes the TypeScript compiler and determines that both greeting and suffix have type string. hello has only one candidate solution, but helloGen has two candidate solutions. Therefore, OPEN_{TAU} composes a set of root candidate solutions from the combination set of varNode1, hello, and helloGen’s candidate solutions, which results in a total of two candidate solutions for the program. [Figure 4.4](#) shows both solutions for the program, with the difference being the highlighted annotations, () => string in [Figure 4.4a](#) and Function in [Figure 4.4b](#).

RANKING. Given a set of typed programs, OPEN_{TAU} scores and ranks candidate solutions and selects the best one. The evaluation methodology consists of two components: the number of type errors present and a *typedness* score that measures the overall type precision of the candidate solution. OPEN_{TAU} returns the program with the fewest type errors with ties broken by typedness.

```

223 let greeting: string = "Hello";
224 let suffix: string = "!";
225
226 // Produces a greeting for the given name
227 const hello = (name: string): string => {
228     return "Hello " + name;
229 };
230
231 function helloGen(name: string): () => string {
232     const helloHelper = (): string => {
233         return hello(name) + suffix;
234     };
235     return helloHelper;
236 }

```

(a) Candidate solution #1.

```

237 let greeting: string = "Hello";
238 let suffix: string = "!";
239
240 // Produces a greeting for the given name
241 const hello = (name: string): string => {
242     return "Hello " + name;
243 };
244
245 function helloGen(name: string): Function {
246     let suffix: string = "!";
247     const helloHelper = (): string => {
248         return hello(name) + suffix;
249     };
250     return helloHelper;
251 }

```

(b) Candidate solution #2.

Figure 4.4: Both candidate solutions for the program, with the different type annotation highlighted.

<pre> 252 function h(name: Name): string { 253 return "Hello " + name; 254 }</pre>	<pre> 255 function h(name: any): any { 256 return "Hello " + name; 257 }</pre>
--	--

(a) A function that may have high accuracy, but fails to type check.

(b) A function that is effectively untyped, as all type annotations are any.

Figure 4.5: Two ways of assigning type annotations to a TypeScript function.

In this case, both candidate solutions in [Figure 4.4](#) type check, so they have zero type errors each. However, the solution in [Figure 4.4a](#) is returned to the user, because `() => string` is more precise than the generic `Function` type.

In this example, we walked through a type prediction procedure given a simple program. Real programs, however, are generally more complex and longer in token size, often resulting in wider, deeper trees that can lead to combinatorial explosion. I discuss each component of `OPENTAU` in detail in the following sections, and describe how it handles very large programs.

4.2 PROGRAM TYPEDNESS

Type prediction systems are typically evaluated on accuracy: predicted types are compared to handwritten, ground truth type annotations [Helledoorn et al., 2018; Wei et al., 2020b; Jesse, Devanbu, and Ahmed, 2021; Jesse, Devanbu, and Sawant, 2022; Pradel et al., 2020]. However, this approach requires labeled data and ignores program semantics—the predicted types may not type check, requiring the programmer to manually resolve type errors. In [Chapter 3](#), I proposed type checking the generated program, which does not require ground truth type annotations. However, trivial type annotations (e. g., `any`) will always type check, but provide little benefit to the programmer. For example, [Figure 4.5](#) shows two assignments of type annotations to a function. In [Figure 4.5a](#), the type annotations have high accuracy but fail to type check, while in [Figure 4.5b](#), the function type checks but is effectively untyped, because all type annotations are `any`.

We would like to combine the strengths of both approaches and define 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.

To compute the typedness score of a program, we:

Table 4.1: Typedness scores 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

1. count the number of undesirable type annotations, i. e., annotations that are trivial or cause type errors;
2. assign a score to each annotation as specified in [Table 4.1](#);
3. sum the scores; and finally
4. normalize the score by the number of types encountered.

The program score is normalized to a number between 0 and 1,000, where lower scores are preferred. For example, a program with a score of 1,000 contains only unknown types, while a program with a score of 0 contains only descriptive names, e. g., `number` or `string[]`.

The typedness metric counts only *leaf* types in the abstract syntax tree, i. e., the types that are being applied to the program. For example, `Array<any>` is scored as 0.5, since `any` is the type argument.

4.3 TREE-BASED PROGRAM DECOMPOSITION

4.3.1 *Decomposing the Program*

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. For example, the comment ([line 192](#)) in [Figure 4.1a](#) is included in the `hello` node.

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.

EXAMPLE. The `hello` function (line 193) in Figure 4.1a is used by `helloHelper` on line 199. `OPENTAU` generates the following comment and includes it in the `hello` node:

```
258 /* Example usages of 'hello' are shown below:
259    hello(name) + suffix; */
```

This comment provides additional context for both the parameter and return type of `hello`, as it shows that the return value can be used with the `+` operator, i. e., numeric addition or string concatenation. Furthermore, the identifiers `name` and `suffix` suggest that they are strings, so the return value of `hello` is likely a string that is concatenated with `suffix`.

4.3.2 Traversing the Tree

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. For example, the tree in Figure 4.1b is traversed in the following order: `helloHelper`, `varNode1`, `hello`, `helloGen`, root.

To generate a candidate solution for a node, i. e., a fully typed node, `OPENTAU` uses a combination of type annotations predicted by an LLM that supports FIM, and type annotations computed by the TypeScript compiler through a process called local type inference. Local type inference is *sound* (it produces types that will always type check) but *conservative* (it may give up and produce any). `OPENTAU` uses local type inference for variable declarations (i. e., `const`, `let`, and `var`) and model-generated predictions for everything else (e. g., function parameters and returns, and class and interface properties). Local type inference is practical for variable declarations because the compiler can inspect the right-hand side of the assignment (if present).

4.3.2.1 Visiting Leaf Nodes

The traversal starts at a leaf node, i. e., a node with no children. To create a prompt for the model, `OPENTAU` uses the TypeScript compiler to identify type annotation sites in the node’s code block and inserts the special token `_hole_` into the first annotation site; passes the prompt to the model, which returns a completion that contains the predicted type; updates the prompt by replacing `_hole_` with the type prediction; and repeats the process with `_hole_` in the next type annotation site of the updated prompt. This fills in the type annotations from left to right.¹

When using the model, its context window size is set to a fixed number of tokens, which is the maximum number of tokens it can read. If the input prompt is larger than the context window, `OPENTAU` truncates the prompt to fit into the context window, removing tokens from both the beginning and end of the prompt. In practice, when the program is decomposed, code blocks generally fit into the context window, so truncation is only necessary for very large code blocks.²

The model can be configured to generate `num_comps` completions for a single hole, and `OPENTAU` can use those completions to generate `num_comps` prompts for the second hole. However, this could lead to a combinatorial explosion of num_comps^n candidate solutions, where n is the number of type annotation sites to be filled in. This is not practical, so `OPENTAU` takes a different approach: it asks the model to generate `num_comps` for the first hole, but only one completion for subsequent holes. This results in `num_comps` candidate solutions (each with n type annotations).

Once candidate solutions have been generated for a node, `OPENTAU` removes duplicates and stores the unique candidates in the node as metadata. Later, when the node’s parent is visited, the candidates will be incorporated into the parent prompt.

4.3.2.2 Visiting Internal Nodes

An internal tree node, i. e., a node with children, can only be processed after its children. This is because an internal node represents a code block that contains other declarations, i. e., those represented by its child nodes, whose candidate solutions must be included in the parent node’s prompt. The child candidates provide additional context to the model when predicting types for a code block, which may reference those child declarations.

To incorporate a child node’s candidate solution into the parent node’s prompt, `OPENTAU` *transplants* type annotations. The key idea is that the

¹ Some models, such as InCoder [Fried et al., 2023], support filling in multiple holes at a time.

² This applies to only 2% of functions in our evaluation dataset.

parent node contains an unannotated version of the child node’s candidate solution. Thus, `OPENTAU` traverses over the candidate’s `AST`, building a dictionary that maps identifiers to type annotations. Next, it traverses over the corresponding `AST` in the parent node, using the dictionary to apply type annotations to the appropriate identifiers. If a type annotation is any or missing, the algorithm uses the TypeScript compiler’s local type inference to compute a type annotation.

Because there may be multiple child nodes, each containing multiple candidates, `OPENTAU` takes all combinations of the child candidates to create prompts for the parent node. However, this may lead to another combinatorial explosion, so the number of combinations is limited to `stop_at`, a user configurable parameter. `OPENTAU` sorts the combinations by their typedness score (Section 4.2); assigns the k -th combination a weight from the Poisson distribution, with `index = k` and $\lambda = 0.7$; and samples for `stop_at` combinations. The Poisson distribution skews the sampling towards the beginning of the list, where the combinations have better typedness scores. Once the combinations are sampled and the prompts are created, `OPENTAU` treats the parent node as a leaf node, and applies the procedure described in Section 4.3.2.1.

EXAMPLE. If a node has two children with m_1 and m_2 candidate solutions respectively, `OPENTAU` generates m_1m_2 prompts for that node. If $m_1m_2 > \text{stop_at}$, `OPENTAU` samples `stop_at` combinations. Then, for each prompt, it generates at most `num_comps` candidates, since the model may return duplicates. This results in at most $\min(m_1m_2, \text{stop_at}) \times \text{num_comps}$ candidate solutions.

4.3.3 Ranking Candidate Solutions

The tree traversal continues until it reaches the root node, and returns at most `stop_at` candidate solutions for the entire program. `OPENTAU` runs the TypeScript compiler’s type checker on each candidate and extracts the number of type errors. If there are no type errors, then the solution type checks. `OPENTAU` additionally computes the typedness score for each candidate solution.

Finally, `OPENTAU` sorts the candidates 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. This solution is presented to the user, with the other solutions available for inspection.

$$\begin{aligned} \langle \text{fim_prefix} \rangle p \langle \text{fim_suffix} \rangle s \langle \text{fim_middle} \rangle m & \quad (\text{PSM}) \\ \langle \text{fim_prefix} \rangle \langle \text{fim_suffix} \rangle s \langle \text{fim_middle} \rangle p m & \quad (\text{SPM}) \end{aligned}$$

Figure 4.6: p , s , and m are the encoded prefix, suffix, and middle spans. $\langle \text{fim_prefix} \rangle$, $\langle \text{fim_suffix} \rangle$, and $\langle \text{fim_middle} \rangle$ are special sentinel tokens defined during the pre-training phase.

4.4 FINE-TUNING FOR FILL IN THE TYPE

We present fill in the type ([FIT](#)), adapting the technique of Bavarian et al. [2022] and Fried et al. [2023] to fine-tune an LLM to predict TypeScript type annotations. 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 [FIM](#) code generation [Ben Allal et al., 2023]. Then, we fine-tune SantaCoder using the TypeScript subset of the near-deduplicated version of The Stack, a dataset of permissively licensed source code [Kocetkov et al., 2022]. We set December 31, 2021 as the training cutoff. Files in The Stack have multiple timestamps for different events, and if the *earliest* timestamp is *after* the cutoff, we set the file aside for evaluation and leave the remaining files for training. This results in a dataset of 12.1 million TypeScript files, with over 1.1 billion lines of code, including comments.

Following Bavarian et al. [2022], we split inputs into prefix, middle, and suffix spans; however, we split on *type annotation* location indices rather than arbitrary code sequences, and select 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.

Next, we transform the spans into prefix-suffix-middle ([PSM](#)) or suffix-prefix-middle ([SPM](#)) formats, as defined in [Figure 4.6](#). We set a 50/50 split for joint training on [PSM](#) and [SPM](#), and train using a left-to-right training objective. Intuitively, the model learns to connect the prefix to the suffix with a single type annotation. [Figure 4.7](#) shows an example of transforming an input into [PSM](#) format.

TRAINING. We trained [FIT](#) for three days, using two NVIDIA H100 GPUs. We set the sequence length to 2,048 tokens and the learning rate to 5×10^{-5} , following SantaCoder [Ben Allal, 2023]. We trained the model

```

260 function sumThree(a: number, b: number, c: number): number {
261   return a + b + c;
262 }

```

(a) A fully typed program with four type annotations: three for function parameters and one for the return type.

```

263 function sumThree(a: number, b: // prefix
264 number // middle
265 , c) {\n return a + b + c;\n} // suffix

```

(b) We select the second type annotation as the middle span, then split the code into prefix, middle, and suffix spans. We remove type annotations from the suffix span.

```

266 (<fim_prefix>function sumThree(a: number, b: (<fim_suffix>), c) {
267   return a + b + c;
268 }(<fim_middle>)

```

(c) The example transformed into [PSM](#) format for training. Although both [SPM](#) and [PSM](#) are used for training, we only use [PSM](#) for inference.

Figure 4.7: An example function, split and transformed into the [PSM](#) context format.

for 59,500 iterations, and roughly 487 million tokens were seen during training.

INFERENCE. We employ the [PSM](#) transformation, which we observed to perform better than [SPM](#). We sample the middle sequence until reaching an end-token or the maximum number of tokens.

4.5 EVALUATION

4.5.1 Dataset

As part of the evaluation, I contribute 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 sites). Focusing on files rather than packages (as I did in [Chapter 3](#)) 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.

Table 4.2: 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
LOC per function	0.11
Function usages	0.11

I construct a dataset of 744 TypeScript files, totalling 77,628 lines of code (excluding blanks and comments). I derive this dataset by filtering the near-deduplicated version of The Stack [Kocetkov et al., 2022], which contains roughly 12.8 million TypeScript files. First, I remove files that depend on external modules and do not type check: this guarantees that all files in the dataset have valid type annotations, and that all files are actually TypeScript and not other files that were incorrectly classified. Next, I remove:

- files that have no type annotation sites (these files are typically only data or comments and will trivially type check);
- have fewer than 50 lines of code (small files are often trivial and uninteresting to evaluate);
- have no functions (these files are typically data or contain only type definitions, which provide little context for type prediction besides the names of identifiers); or
- average fewer than five lines of code per function (short functions are often trivial, e. g. getters and setters).

These filtering steps reduce the dataset to 21,464 files.

Then, I compute a weighted quality score for each file, with the weights shown in Table 4.2. The score is based on several factors, most of them being *density* metrics that normalize by the number of tokens in a file, to avoid bias from very large files. These metrics are:

- higher function and variable annotation density (so files have more type annotation sites);

<pre> 269 export interface IParseOptions { 270 filename?: string; 271 startRule?: string; 272 tracer?: any; 273 [key: string]: any; 274 } </pre>	<pre> 275 export interface IParseOptions { 276 filename?; 277 startRule?; 278 tracer?; 279 // what goes here? 280 } </pre>
--	--

(a) The original interface, which defines an interface with three properties and an index signature.

(b) Removing type annotations; however, it is not clear how to handle the index signature.

Figure 4.8: A TypeScript file that was removed from the dataset, because the type definition contains an index signature.

- higher type definition density (so there are more user-defined types);
- fewer occurrences of dynamic features like `eval` (because these features are difficult to migrate);
- lower trivial types density (so the dataset contains fewer annotations with little type information);
- lower predefined types density (to encourage more user-defined types);
- more lines of code per function (to encourage more complex files); and
- more function usages (since function call sites provide additional context).

After computing each metric separately, I convert them to standard scores (i. e., the number of standard deviations above or below the mean) and normalize to a value between 0 and 1. Then, I use the weights to compute a single, combined quality score, and remove any file that is one or more standard deviations below the mean score. This leaves 17,254 files in the dataset.

To minimize test-train overlap, I apply the 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. However, this process does not always succeed, in particular, when types use *index signatures*. For example, [Figure 4.8](#) declares a type that uses an index signature on [line 273](#): this means that values of the `IParseOptions` type can be indexed with a string, with the result having type `any`. However, it is not clear how this index signature can be removed, nor how type prediction should fill in an index signature when there is nothing to annotate ([line 279](#)). Therefore, I remove

files that use index signatures. The results in the final evaluation dataset of 744 files.

4.5.2 Experiments

We evaluate OPENTAU to determine the effectiveness of FIT and its *tree-based program decomposition*, using three metrics: the percentage of files that type check, the average typedness score for files that type check, and the average number of type 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 FIT for TypeScript (SantaCoder-FIT). We compare OPENTAU’s program decomposition with a baseline that treats the entire file as a single tree node. For all experiments, we set `temperature = 0.75`, `stop_at = 400`, and `num_comps = 3`. We use a default context window size of 640 tokens, but run additional experiments on context window sizes of 160, 320, and 1280 tokens.

OPENTAU and the baseline experiments use SantaCoder to infer type annotations for function parameters, return types, class and interface fields, and lambda functions. However, the completion that SantaCoder returns is parsed to extract the first plausible type annotation, e.g., if the completion is `stringstringstring`, the type parser returns `string`. Variable declarations are handled differently: OPENTAU uses TypeScript’s local type inference to compute their type annotations, but they are ignored in the baseline experiments, which is equivalent to treating them as any.

Inference on a single hole takes an average 1.6 seconds on an NVIDIA RTX 2080 Ti GPU. A full experiment can take 10–30 hours on eight 2080 Tis. Smaller context window sizes and using OPENTAU’s program decomposition can significantly decrease the execution time.

Table 4.3 shows our results. OPENTAU with usages 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.

FILL IN THE TYPE. We run the baseline configuration (no program decomposition) to compare the effectiveness of FIT. 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.

CONTEXT WINDOW SIZE. To evaluate the impact of context window size, we run additional experiments with SantaCoder-FIT on window sizes of 160, 320, and 1,280. We observe that a larger context window size results in more files that type check, while maintaining similar average typedness scores.

Table 4.3: Experimental results of evaluating OPENTAU. We measure *files that type check*, which is more rigorous than measuring individually correct type annotations. The number of files that type check is reported, out of a dataset of 744 files. All numbers are rounded to the nearest tenth.

TS = SantaCoder-TS, i. e., fine-tuned on TypeScript but not FIT

FIT = SantaCoder-FIT, i. e., fine-tuned on TypeScript for FIT

Configuration	Window	Type checks	Typedness	Type errors
TS baseline	640	245 (32.9%)	200.7	4.7
FIT baseline	1,280	353 (43.3%)	210.0	4.1
FIT baseline	640	297 (39.9%)	200.9	5.2
FIT baseline	320	248 (33.3%)	200.7	5.1
FIT baseline	160	178 (23.9%)	201.2	6.3
FIT baseline, usages	640	280 (37.6%)	206.0	4.9
FIT OPENTAU, no usages	640	274 (36.8%)	168.4	3.7
FIT OPENTAU, usages	640	353 (47.4%)	154.6	3.3

TREE-BASED PROGRAM DECOMPOSITION. 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.

USAGE COMMENTS. Next, we compare configurations with usage comments enabled and disabled. Recall that when predicting types for functions, OPENTAU searches the program for usages of that function, generates a comment containing those usage statements, and prepends it to the function’s prompt (Section 4.3.1). OPENTAU without usages performs similar to the baseline with usages, and both perform slightly worse than the baseline without usages. However, enabling usages for OPENTAU results in the largest jump in performance, while enabling usages for the baseline results in diminished performance. This experiment shows that long-range context is helpful for type prediction.

TYPESCRIPT COMPILER. We compare our best OPENTAU configuration (which uses FIT and tree-based program decomposition on a context window size of 640 tokens) to the TypeScript compiler’s builtin type inference. Table 4.4 shows the results. Although the TypeScript compiler allows significantly more files to type check, its results have a much higher (worse) typedness score, more type errors, and more trivial type annotations.

Table 4.4: Comparing the TypeScript compiler to OPENTAU, on the evaluation dataset with 744 files.

Errors = average number of type errors per file
 # = average number of trivial annotations per file
 % = percentage of trivial type annotations

System	Type checks	Typedness	Errors	Trivial	
				#	%
tsc	553 (74.3%)	309.4	6.0	13.1	29.4
OPENTAU	353 (47.4%)	154.6	3.3	4.3	10.4

Table 4.5: Comparing the effectiveness of the type parser on both models. All experiments were run with a context window of 640 tokens, and on the baseline, i. e., without program decomposition.

TS = SantaCoder-TS, i. e., fine-tuned on TypeScript but not FIT
 FIT = SantaCoder-FIT, i. e., fine-tuned on TypeScript for FIT

Configuration		Type checks			Typedness	Errors	
		✓	#	%		Type	Syntax
TS	No parser	1	50	2.0	0.0	121.2	42.1
FIT	No parser	25	50	50.0	230.0	4.6	0.2
TS	Parser	245	744	32.9	200.7	4.7	0.0
FIT	Parser	297	744	39.9	200.9	5.2	0.0

These trivial type annotations, i. e., `any`, `any[]`, or `Function`, *exclude* the ones that were already present in the original, type-annotated files, since those annotations were handwritten and likely necessary to ensure the file type checks.

TYPE PARSER. Finally, 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. Table 4.5 shows that FIT 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. However, the type parser is helpful, as FIT can still produce syntax errors, and it practically eliminates all syntax errors—the results round to 0.00, even with two decimal places of precision.

<pre> 281 function sum_list(l: _hole_) { 282 let sum = 0; 283 for (let i=0;i<l.length;i++) { 284 sum += l[i]; 285 } 286 287 return sum; 288 } </pre>	<pre> 289 any[]: number { 290 if (l.length === 0) { 291 throw 'Empty list!'; 292 } 293 if (l.length === 1) { 294 return l[0]; 295 } 296 return sum </pre>
<p>(a) A function where <code>_hole_</code> should be filled in with a type annotation.</p>	<p>(b) The “type annotation” provided by a FIM model.</p>

Figure 4.9: An example of how FIM generates extraneous code. The expected type annotation is `number[]`.

4.5.3 Case Studies

4.5.3.1 Fill in the Type

Figure 4.9 shows an example of how FIM performs poorly, which motivates our FIT method. Figure 4.9a is an input function where `_hole_` should be replaced by a type annotation, which is expected to be `number[]`. However, FIM generates the code in Figure 4.9b: it generates the imprecise type `any[]`, along with most of a body function. We require a model that fills in only the type annotation.

4.5.3.2 Evaluation Dataset

Constructing the evaluation dataset involved two filtering stages. First, undesirable files were excluded, then a quality score was computed and used to exclude lower quality files. Figure 4.10 shows examples of files that were excluded at the first stage. These include files that are entirely commented out (Figure 4.10a), files that have no type annotation sites (Figure 4.10b), files with trivial functions (Figure 4.10c), and files with no functions (Figure 4.10d).

In the second stage of filtering, lower quality files are removed. Figure 4.11 shows an example of a file with a low quality score: it has only one type annotation location with type annotation `any` (line 326), and only one function (lines 327–363), which is a constructor with no parameters. The majority of the file is a single configuration object (lines 328–362). On the other hand, Figure 4.12 shows an example of a file with a high quality score: it defines a type (lines 365–369) that is used in two locations (lines 381 and 394), four functions with multiple function parameters (lines 371, 389, 397 and 405), and one usages of the `splitKey` function (line 406).

```

297 //// global app config
298 //declare type appConfigType = {
299 //  baseUrl: string
300 //  debounceTime: number
301 //}

```

(a) A TypeScript file with zero lines of code (and therefore zero type annotation sites), because everything is commented out.

```

302 export default {
303   group: "typography",
304   pagination: {
305     currentPage: 2,
306     prevPagePath: "/typography/page/1",
307     nextPagePath: "/typography/page/3",
308     hasNextPage: true,
309     hasPrevPage: true,
310   },
311 };

```

(b) A TypeScript file that exports constants, but has zero type annotation sites, so there is nothing to migrate.

```

312 export const TabIcons = [
313   'tab', 'code-braces', 'tags', 'target'
314 ];
315
316 export function getTabIcon(tabType: number): string {
317   return TabIcons[tabType];
318 }

```

(c) A short TypeScript file with an even shorter function that is not doing anything interesting, and has very little context for type prediction.

```

319 export interface Log {
320   error: (text: any) => void
321   warn: (text: any) => void
322   info: (msg: any, ...optionalParams: any[]) => void
323   log: (text: any) => void
324 }

```

(d) A TypeScript file that defines an interface with several type annotation sites; however, there are no function bodies and very little context for type prediction. In particular, it is not obvious what types should annotated for error, warn, info, and log.

Figure 4.10: Files that were excluded from the dataset by the thresholds.

```
325 export class EventsConfig {
326   public config: any = {};
327   constructor() {
328     this.config = {
329       items: [
330         {
331           id: 1,
332           name: 'New Year Party',
333           image: './assets/images/background/horizontal/1.jpg',
334           date: '04/14/2020 00:00:00',
335           price: 100,
336           address: '2102 Tennessee Avenue, Plymouth MI - 48170',
337           phone: '734-637-0374',
338           email: 'y65nl6lt7pf@payspun.com',
339           description: '' // omitted string
340         },
341         {
342           id: 2,
343           name: 'Dance with DJ Nowan',
344           image: './assets/images/background/horizontal/2.jpg',
345           date: '12/31/2019 00:00:00',
346           address: '2102 Tennessee Avenue, Plymouth MI - 48170',
347           phone: '734-637-0374',
348           email: 'y65nl6lt7pf@payspun.com',
349           description: '' // omitted string
350         },
351         {
352           id: 3,
353           name: 'Move You\'s Legs',
354           image: './assets/images/background/horizontal/3.jpg',
355           date: '12/31/2019 00:00:00',
356           address: '2102 Tennessee Avenue, Plymouth MI - 48170',
357           phone: '734-637-0374',
358           email: 'y65nl6lt7pf@payspun.com',
359           description: '' // omitted string
360         }
361       ]
362     };
363   }
364 }
```

Figure 4.11: A TypeScript file with a low quality score, because it has only one type annotation site (with type annotation any), and the majority of the file is data.

```
365 export type EntityId = {
366   prefix: string;
367   id: string;
368   key: string;
369 };
370
371 export const generateEntityId = (prefix: string, length: number) => {
372   const base62Chars =
373     '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';
374   let id = '';
375
376   for (let i: number = 0; i < length; i++) {
377     const random = Math.floor(Math.random() * 62);
378     id = id.concat(base62Chars[random]);
379   }
380
381   const entityId: EntityId = {
382     prefix: prefix,
383     id: id,
384     key: `${prefix}:${id}`
385   };
386   return entityId;
387 };
388
389 export const getEntityIdfromID = (prefix: string, id: string) => {
390   return {
391     prefix,
392     id,
393     key: `${prefix}:${id}`
394   } as EntityId;
395 };
396
397 const splitKey = (key: string) => {
398   const [prefix, id] = key.split(':');
399   return {
400     prefix,
401     id
402   };
403 };
404
405 export const getIdFromKey = (key: string) => {
406   const splittedKey = splitKey(key);
407   return splittedKey.id;
408 };
```

Figure 4.12: A TypeScript file with a high quality score, because it defines a type, several functions, and has a call to one of those functions (`splitKey`).

4.5.3.3 *Tree-based Program Decomposition*

Figure 4.13 compares a prediction given by the baseline (with a context window of 160 tokens) to an OPENTAU prediction (tree-based program decomposition with usages). The baseline predicts number for the min parameter (line 410), which seems reasonable for a parameter that is likely to be a “minimum,” but OPENTAU correctly predicts that min has type number[] (line 423). The baseline also predicts ZPoint as the return type (line 413), while OPENTAU correctly predicts void (line 426). Finally, the baseline skips the type annotations for local variables x, y, and z (lines 415–417), as it is unlikely to predict the correct types from the given context. On the other hand, OPENTAU invokes the TypeScript compiler, which deduces that morton3 returns number, so the local variables are correctly annotated (lines 428–430).

4.5.3.4 *Usage Comments*

Figure 4.14 compares a prediction given by OPENTAU, without and with usages. There is a critical usage of the _preparePaper method in an adjacent method, as the any[] type annotation is given to the return value of _preparePaper. Furthermore, the second argument to _preparePaper is a call to find, which returns an array. This information is not available in Figure 4.14a, which does not have a usages comment, so the model predicts number for the firstYFold parameter and a return type of boolean (line 439). On the other hand, the usages comment is available in Figure 4.14b (lines 443–445), so the model predicts number[] for the firstYFold parameter and a return type of number[][] (line 448). Indeed, the body of _preparePaper accesses firstYFold as an array (lines 462 and 463).

4.6 SUMMARY

MACHINE LEARNING AND SEARCH. LLMs have shown promise in code generation tasks, but they are limited by not being able to take advantage of program semantics. One way to mitigate this is by combining machine learning and search: the model generates a probability distribution of type annotations, and an algorithm searches for the “best” type annotation. In OPENTAU’s case, this search finds the type annotations with the best typedness score, i. e., the annotations with the most type information. The experiments show that this search is successful in returning solutions that have a higher typedness score, and more importantly, solutions that have fewer trivial type annotations than would be inferred by the TypeScript

```

409 public toPoint(
410   min: number,
411   step: number,
412   buffer: Uint8Array,
413   pos: number): ZPoint
414 {
415   let x = this.morton3(this.lo, this.hi >>> 1);
416   let y = this.morton3(this.lo >>> 1, this.hi >>> 2);
417   let z = this.morton3(/* omitted */, this.hi >>> 3);
418   buffer[pos + 0] = (x + min[0]) * step;
419   buffer[pos + 1] = (y + min[1]) * step;
420   buffer[pos + 2] = (z + min[2]) * step;
421 }

```

(a) Baseline type prediction. Note that baseline type prediction skips the local variable declarations `x`, `y`, and `z`.

```

422 public toPoint(
423   min: number[],
424   step: number,
425   buffer: number[],
426   pos: number): void
427 {
428   let x: number = this.morton3(this.lo, this.hi >>> 1);
429   let y: number = this.morton3(this.lo >>> 1, this.hi >>> 2);
430   let z: number = this.morton3(/* omitted */, this.hi >>> 3);
431   buffer[pos + 0] = (x + min[0]) * step;
432   buffer[pos + 1] = (y + min[1]) * step;
433   buffer[pos + 2] = (z + min[2]) * step;
434 }
435 // morton3 has signature:
436 // public morton3(lo: number, hi: number): number;

```

(b) Type prediction with OPEN_{TAU}'s tree-based program decomposition. OPEN_{TAU} uses the TypeScript compiler to infer type annotations for the local variable declarations `x`, `y`, and `z`.

Figure 4.13: Comparing the baseline to OPEN_{TAU}: type prediction for `toPoint`, a class method. Type annotations that are different are highlighted.

```

437 private _preparePaper(
438   coords: number[],
439   firstYFold: number): boolean
440 {
441   // omitted
442 }

```

(a) Type prediction without usages. OPENTAU does not predict the correct type annotation for the `firstYFold` parameter.

```

443 /* Example usages of '_preparePaper' are shown below:
444   let paper: any[] =
445     this._preparePaper(coords, folds.find(f => f[0] === 'y')); */
446 private _preparePaper(
447   coords: number[][],
448   firstYFold: number[]): number[][]
449 {
450   let maxY: number = 0; let maxX: number = 0;
451   for (const coord of coords) {
452     if (coord[1] > maxY) { maxY = coord[1]; }
453     if (coord[0] > maxX) { maxX = coord[0]; }
454   }
455   const paper: any[] = [];
456   for (let y: number = 0; y <= maxY; y++) {
457     paper.push(new Array(maxX + 1).fill(false));
458   }
459   for (const coord of coords) {
460     paper[coord[1]][coord[0]] = true;
461   }
462   if (paper.length <= (firstYFold[1] * 2) {
463     const toAdd: number = firstYFold[1] * 2 - paper.length + 1;
464     for (let i: number = 0; i < toAdd; i++) {
465       paper.push(new Array(maxX + 1).fill(false));
466     }
467   }
468   return paper;
469 }

```

(b) Type prediction with usages. OPENTAU identifies a usage of the `_preparePaper` method, and uses it to provide additional context to the model.

Figure 4.14: Comparing OPENTAU without and with usages, when predicting types for `_preparePaper`, a class method. The relevant type annotation is highlighted.

compiler. I leave alternate definitions of the typedness metric and alternate search algorithms to future work.

PROGRAM DECOMPOSITION. If a program is too large to fit into the context window, decomposing the program into smaller functions can help. This allows OPENTAU to migrate a program one function at a time, and use a fully type-annotated function as context when migrating another function. This strategy allows OPENTAU to use the structure and implicit dependencies that already exist in a program. Additionally, program decomposition allows OPENTAU to identify all call sites of a function and uses this to generate usage comments, which serve as additional context. The experiments show that usage comments are effective, and suggest that other kinds of context could be explored in future work.

FINE-TUNING FOR TYPE PREDICTION. Fill in the middle (FIM) is a natural fit for type prediction, since it allows generating type annotations in the middle of a program. However, FIM is trained for general-purpose code generation, and not type prediction. By using a fill in the type (FIT) training procedure that is aware of type annotations, the experiments show that this approach significantly improves a model's ability to infill syntactically valid type annotations. I believe that when using a model for a specific task, it is worthwhile to consider whether that model can be fine-tuned for that task, instead of relying on the model's general-purpose capabilities.

GENERATING TYPE DEFINITIONS

Up until now, all prior work has focused solely on *type annotation prediction* and the problem of *type definition generation* has not been studied. In this chapter, I describe the type definition generation problem and my approach to solving it by fine-tuning an [LLM](#).

To motivate the problem, consider the following example:

```
470 function dist(p1, p2) {  
471   const dx = p2.x - p1.x;  
472   const dy = p2.y - p1.y;  
473   return Math.sqrt(dx*dx + dy*dy);  
474 }
```

This function takes two points, `p1` and `p2`, and computes the distance between those points. A type prediction system might suggest the following annotations, which are highlighted:

```
475 function dist(p1: Point, p2: Point): number {  
476   const dx = p2.x - p1.x;  
477   const dy = p2.y - p1.y;  
478   return Math.sqrt(dx*dx + dy*dy);  
479 }
```

This is a reasonable and correct type annotation;¹ however, the code does not type check because `Point` is not defined. To fully migrate this function, the type definition needs to either be manually defined by a programmer, or automatically generated, e. g., by an [LLM](#). The type definition could be a class definition (which also requires defining a constructor), an interface definition, or a type alias for an object type:

```
480 // Class definition  
481 class Point {  
482   x: number;  
483   y: number;  
484   constructor(x: number, y: number) {  
485     this.x = x;  
486     this.y = y;  
487   }  
488 }
```

¹ Another possible type annotation for `p1` and `p2` is `{x: number, y: number}`. This annotation type checks successfully, but is more verbose than `Point`.

```

498 <commit_before>function dist(p1, p2) {
499   const dx = p2.x - p1.x;
500   const dy = p2.y - p1.y;
501   return Math.sqrt(dx*dx + dy*dy);
502 }
503 <commit_msg> Add type annotations
504 <commit_after>

```

Figure 5.1: A prompt used at inference time to add type annotations to `dist`, with the instruction highlighted.

```

489
490 // Interface definition
491 interface Point {
492   x: number;
493   y: number;
494 }
495
496 // Type alias
497 type Point = {x: number, y: number};

```

Generating type definitions is a much more difficult problem than predicting type annotations, because an LLM has fewer constraints, i. e., it must generate significantly more code than a single type annotation. Therefore, when training an LLM to generate type definitions, it is important to determine which dataset and training format work best.

5.1 APPROACH

My approach is to fine-tune one of the smaller StarCoderBase models, such as StarCoderBase-7B, similar to how FIM is trained. StarCoderBase is an open code LLM, meaning its parameters and training data are openly available, and it has been trained on a variety of formats, which extends its capabilities. For example, StarCoderBase was trained on Git commit data using a format with the special tokens `<commit_before>`, `<commit_after>` and `<commit_msg>`. These tokens denote code before a commit, code after a commit, and the commit message describing the changes, which allows the model to learn to edit code by following natural language instructions. Reusing this training format with StarCoderBase is more practical than training a new model from scratch or fine-tuning StarCoderBase on a different format.

To use this format at inference time, StarCoderBase is provided a prompt that consists of `<commit_before>`, the original code to edit, `<commit_msg>`,

```

505 <commit_before>function dist(p1, p2) {
506   const dx = p2.x - p1.x;
507   const dy = p2.y - p1.y;
508   return Math.sqrt(dx*dx + dy*dy);
509 }
510 <commit_msg> Add type annotations and interfaces
511 <commit_after>interface Point {
512   x: number;
513   y: number;
514 }
515
516 function dist(p1: Point, p2: Point): number {
517   const dx = p2.x - p1.x;
518   const dy = p2.y - p1.y;
519   return Math.sqrt(dx*dx + dy*dy);
520 }

```

Figure 5.2: A training example for single-step migration.

an instruction to edit that code, and finally `<commit_after>`. The code generated after `<commit_after>` is then extracted and taken as the result of editing the original code. Figure 5.1 shows a prompt that is used to add type annotations to the `dist` function: the original function is between the `<commit_before>` and `<commit_msg>` tokens on lines 498–502, the edit instruction is after the `<commit_msg>` token on line 503, and the model generates code after the `<commit_after>` token on line 504.

5.1.1 Single-step migration

The Git commit format can be used with StarCoderBase to migrate a program in a single step, from untyped code to fully typed code. At inference time, the prompt will be similar to Figure 5.1, but with the instruction `Add type annotations and interfaces`. From informal testing with a StarCoderBase model that had not been fine-tuned, I found this instruction performed better than alternatives that mentioned classes or TypeScript.

Figure 5.2 shows a training example for this approach, with both the code before and after the change. To construct a training example, I start with a TypeScript program that has type definitions and type annotations: this will be the code after the change, so I mark it with `<commit_after>` (lines 511–520). Next, I delete types to get an untyped program, i. e., the code before the change, and mark it with `<commit_before>` (lines 505–509). Deleting types means removing non-class type definitions, type annotations, and type assertions; class definitions are preserved because they contain additional non-type code in methods. Finally, I use the instruction

Add type annotations and interfaces and mark it with `<commit_msg>` (line 510). This format trains the model to associate the fully typed code with the untyped version and an instruction to add type annotations and interfaces.

5.1.2 Multi-step migration

An alternative migration is to take a multi-step approach that first adds type annotations and then adds type definitions one at a time. For example, consider the two following functions:

```
521 function circleArea(c) {
522   return Math.PI * c.radius * c.radius;
523 }
524 function rectangleArea(r) {
525   return r.width * r.height;
526 }
```

This code is given to the model with an instruction to add type annotations:

```
527 <commit_before>function circleArea(c) {
528   return Math.PI * c.radius * c.radius;
529 }
530 function rectangleArea(r) {
531   return r.width * r.height;
532 }
533 <commit_msg> Add type annotations
534 <commit_after>
```

The model generates code after the `<commit_after>` token, adding the Circle and Rectangle type annotations, which are highlighted:

```
535 function circleArea(c: Circle) {
536   return Math.PI * c.radius * c.radius;
537 }
538 function rectangleArea(r: Rectangle) {
539   return r.width * r.height;
540 }
```

The types Circle and Rectangle are not defined, so the next step is to add those type definitions, one at a time:

```
541 <commit_before>function circleArea(c: Circle) {
542   return Math.PI * c.radius * c.radius;
543 }
544 function rectangleArea(r: Rectangle) {
545   return r.width * r.height;
```



```

546 }
547 <commit_msg> Add a type alias or interface for Circle
548 <commit_after>

```

The instruction specifically mentions “type alias” and “interface,” and does not mention “class,” because that could lead the model to generate a class with extraneous methods. In this example, the model generates an interface definition for `Circle`, but `Rectangle` is still undefined and now `Point` is a new type annotation that is undefined:

```

549 interface Circle {
550   position: Point;
551   radius: number;
552 }
553
554 function circleArea(c: Circle) {
555   return Math.PI * c.radius * c.radius;
556 }
557 function rectangleArea(r: Rectangle) {
558   return r.width * r.height;
559 }

```

To finish migrating the example, the model needs to be instructed to generate type definitions for `Point` and `Rectangle`, as well as any further types that might be introduced. The model can be repeatedly invoked with a new instruction for each type to be generated until all type definitions are present or a limit has been reached, which avoids infinite loops.

This strategy requires fine-tuning `StarCoderBase` on a mix of two different training formats, which are shown in [Figure 5.3](#). The first format ([Figure 5.3a](#)) trains the model to add type annotations, while the second format ([Figure 5.3b](#)) trains the model to add a single type definition.

To construct a training example, I again start with a TypeScript program. If the program has no type definitions, then I use the first training format ([Figure 5.3a](#)): the original program will be the code after adding type annotations, so I mark it with `<commit_after>` (lines 567–572). Next, I delete type all type annotations to get the code before the change, and mark it with `<commit_before>` (lines 560–565). Finally, I use the instruction `Add type annotations` and mark it with `<commit_msg>` (line 566).

If the original program contains type definitions, then I use the second training format ([Figure 5.3b](#)). First, I randomly delete some of the type definitions—this is so the training example resembles what the model would observe in practice, where some of the type definitions may be missing. For example, lines 585–600 shows code where the type definitions for `Rectangle` and `Circle` have been inserted, but the type definition for `Point` is still missing. Next, I select one of the present type definitions,

```
560 <commit_before>function circleArea(c) {  
561   return Math.PI * c.radius * c.radius;  
562 }  
563 function rectangleArea(r) {  
564   return r.width * r.height;  
565 }  
566 <commit_msg> Add type annotations  
567 <commit_after>function circleArea(c: Circle) {  
568   return Math.PI * c.radius * c.radius;  
569 }  
570 function rectangleArea(r: Rectangle) {  
571   return r.width * r.height;  
572 }
```

(a) Adding type annotations.

```
573 <commit_before>interface Circle {  
574   position: Point;  
575   radius: number;  
576 }  
577  
578 function circleArea(c: Circle) {  
579   return Math.PI * c.radius * c.radius;  
580 }  
581 function rectangleArea(r: Rectangle) {  
582   return r.width * r.height;  
583 }  
584 <commit_msg> Add a type alias or interface for Rectangle  
585 <commit_after>interface Rectangle {  
586   position: Point;  
587   width: number;  
588   height: number;  
589 }  
590 interface Circle {  
591   position: Point;  
592   radius: number;  
593 }  
594  
595 function circleArea(c: Circle) {  
596   return Math.PI * c.radius * c.radius;  
597 }  
598 function rectangleArea(r: Rectangle) {  
599   return r.width * r.height;  
600 }
```

(b) Adding a type definition.

Figure 5.3: Training examples for multi-step migration.

Table 5.1: Summary of STENO`TYPE` training datasets.

Dataset	Examples	LOC	Tokens	Size
OPEN <code>TAU</code> training	9.7M	564M	7.4 B	26.4 GB
STENO <code>TYPE</code> training	7.2M	984M	12.1 B	44.7 GB
STENO <code>TYPE</code> annotations	5.3M	700M	8.0 B	28.8 GB
STENO <code>TYPE</code> definitions	1.9M	284M	4.1 B	15.9 GB

e. g., `Rectangle`, and delete it to get the code before, which is shown on [lines 573–583](#). Finally, I add an instruction on [line 584](#) that explicitly refers to `Rectangle`, i. e., `Add a type alias or interface for Rectangle`.

5.2 TRAINING

Based on preliminary experiments, I fine-tuned `StarCoderBase-7B` on the multi-step migration format to produce the `STENOTYPE` model for migrating JavaScript to TypeScript. Smaller `StarCoderBase` models and the single-step migration format were not as effective at generating type definitions. To fine-tune, I used a training dataset based on the one used to train `OPENTAU` ([Section 4.4](#)). That dataset consists of the TypeScript files from near-deduplicated version of The Stack [Kocetkov et al., 2022], with a training cutoff of December 31, 2021; files in The Stack have multiple timestamps for different events, so if the *earliest* timestamp is *after* the cutoff, I exclude it from training. Furthermore, I exclude files that are syntactically invalid. This results in a dataset of 9.67 million files with 564 million lines of code (excluding comments and whitespace), or 7.38 billion tokens taking up 26.4 GB of storage.

However, this dataset contains only TypeScript files, so it must be transformed into the multi-step Git commit format for training. This transformation can be done online, during training, or it can be done offline, ahead of time, which speeds up training at the cost of requiring more disk space. I chose the latter and preprocessed the dataset: the multi-step migration dataset contains 12.1 billion tokens taking up 44.7 GB of storage. Within the multi-step migration dataset, 73.4% of files (which make up 66.2% of tokens) contain no type definitions and are used to train the model to add type annotations, while the remaining 26.6% of files (which comprise 33.8% of tokens) are used to train the model to add type definitions. [Table 5.1](#) summarizes the training datasets.

I trained `STENOTYPE` for 28 hours, using four NVIDIA A100 80GB GPUs. The sequence length was set to 8,192 tokens, i. e., the full context window

size, with multiple training examples concatenated to improve efficiency. The batch size was set to 1, gradient accumulation to 4 steps, and learning rate to 2.5×10^{-5} . Additionally, I train with the Low Rank Adaptation (LoRA) technique [Hu et al., 2022], with a rank of 16, a scaling factor of 32, and a dropout probability of 0.05. I trained for 1,000 steps, or approximately 0.01 epochs, and roughly 131 million tokens were seen during training.

5.3 EVALUATION

5.3.1 Dataset

I use two datasets to evaluate STENOTYPE, which I call TS-SOURCED and JS-SOURCED. TS-SOURCED is based on the TypeScript files I used to evaluate OPENTAU (Section 4.5.1), while JS-SOURCED is based on the JavaScript packages I used to evaluate TYPEWEAVER (Section 3.3.1).

Constructing TS-SOURCED is similar to the process I used for the OPENTAU dataset: I start with TypeScript files from the near-deduplicated version of The Stack and keep only the files that are syntactically valid and type check. Next, I filter and keep files that:

- have at least one type annotation site;
- have at least one (non-class) type definition;
- have at least 50 lines of code (excluding comments and whitespace), outside of type definitions;
- have at least one function;
- have at least five lines of code per function;
- have at most 4,096 tokens; and
- are after the December 31, 2021 training cutoff.

This leaves 338 files after filtering, which I sample to get 50 files. The filtering process removes files that cannot be migrated (e.g., because they are not valid TypeScript), files that can be trivially migrated (e.g., because they require no type annotations or no type definitions), and files that are too simple (e.g., too short, have no functions, or have trivial functions). The 4,096 token limit ensures that the full prompt and output (which roughly doubles the number of tokens from the original TypeScript file) can fit into STENOTYPE’s 8,192-token context window. Finally, taking files after the training cutoff minimizes test/train overlap.

As the final step, I remove all types to get the untyped code for evaluation. Specifically, I remove type annotations, type assertions (i. e., casts), type aliases, and type interface definitions. I do not remove *class definitions*: although a class definition is a type, it also defines fields and methods, and removing those would fundamentally change the file being evaluated. However, I still remove *type annotations* from class definitions.

JS-SOURCED is a more challenging dataset than TS-SOURCED for two reasons. First, TS-SOURCED is built from TypeScript files with the types removed, which means a migration to typed code exists. On the other hand, JS-SOURCED is built from JavaScript files, with no guarantee that the files can be typed. Second, TS-SOURCED is built from single files, while JS-SOURCED is built from multi-file projects. This introduces inter-file dependencies.

To construct JS-SOURCED, I start with the TYPEWEAVER dataset, which consists of 506 JavaScript packages from the top 1,000 most downloaded packages from the npm Registry (as of August 2021). These packages contain the original source code from GitHub, with testing code removed and type declarations (.d.ts) from DefinitelyTyped included. To prepare this dataset for evaluating STENOTYPE, I perform additional processing and filtering. First, because STENOTYPE is designed for *single-file* inputs but the TYPEWEAVER dataset consists of *multi-file packages*, I use Rollup [Rollup contributors, 2015] to compile a multi-file package into a single file. Additionally, I implemented a Rollup plugin to insert the name of each constituent file as a comment in the output, preceding the contents of that file, in the output; this will be useful during evaluation to map lines from the output back to the original files. Second, similar to TS-SOURCED dataset, I filter and keep files that:

- have at least one type annotation site;
- have at least 50 lines of code (excluding comments and whitespace);
- have at least one function;
- have at least five lines of code per function; and
- have at most 4,096 tokens.

This leaves 177 packages, which I sample to get 50 packages for the final dataset. Unlike TS-SOURCED, I do not remove files that fail to type check: this is because most JavaScript code will fail to type check with the TypeScript compiler,² especially before adding type annotations. Additionally,

² Out of the 506 packages from the original TYPEWEAVER dataset, 469 bundled successfully. Of those packages, only 192 (40.9%) type check. Out of the 177 packages after filtering, only 39 (22.0%) type check. Within the 50-package sample, only 7 (14%) type check.

Table 5.2: Summary of STENOTYPE evaluation datasets.

(a) Number of packages, files, lines of code (LOC), tokens, and size.

Dataset	Packages	Files	LOC	Tokens	Size
TS-SOURCED	50	50	6,339	65.9K	225 KB
JS-SOURCED	50	91	7,645	82.0K	287 KB

(b) Number of functions, annotation sites, and type definitions.

Dataset	Functions	Annotation sites	Type definitions
TS-SOURCED	455	1,823	190
JS-SOURCED	723	2,874	4

I do not apply a training cutoff, since STENOTYPE was fine-tuned only on TypeScript code, which should not overlap with the JS-SOURCED dataset. Even if StarCoderBase was trained on this dataset, it was not trained on the specific format that I use for inference.

Table 5.2 shows a summary of the TS-SOURCED and JS-SOURCED evaluation datasets. JS-SOURCED contains a non-zero number of type definitions because modern JavaScript supports class definitions.

5.3.2 Inference

I evaluated single-step migration with StarCoderBase-7B, and multi-step migration with both StarCoderBase-7B and STENOTYPE. I call these configurations StarCoder-7B SINGLE, StarCoder-7B MULTI, and STENOTYPE-7B MULTI. These configurations were evaluated on both the TS-SOURCED and JS-SOURCED evaluation datasets, resulting in a total of six evaluation configurations.

For all configurations, I use a context window size of 8,192 tokens and the parameters `temperature = 0.2` and `top_p = 0.95`, following the convention used by Chen et al. [2021]. For the multi-step migration, I configure the model to generate up to five type definitions, to avoid timeouts or infinite loops, and I generate 20 samples for each problem. I ran inference on an NVIDIA H100 GPU, using the vLLM library [Kwon et al., 2023]. Inference took between 40–154 minutes, with the multi-step migrations taking more time. The inference times for each configuration are shown in Table 5.3.

Table 5.3: Inference time for each evaluation configuration, in minutes.
(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Inference time (minutes)
(1)	StarCoder-7B SINGLE	40
	StarCoder-7B MULTI	131
	STENOTYPE-7B MULTI	154
(2)	StarCoder-7B SINGLE	77
	StarCoder-7B MULTI	59
	STENOTYPE-7B MULTI	72

5.3.3 Results

5.3.3.1 Does Migrated Code Parse?

In the migration approaches I use, StarCoderBase and STENOTYPE may generate syntactically invalid code. This is different from the type prediction models I evaluated, which generate syntactically valid type annotations. Thus, in this first experiment, I determine how many packages and files parse after migration. Table 5.4 and Figure 5.4 show the number and percentage of packages and files that parse as valid TypeScript. For the TS-SOURCED dataset, every package consists of a single file, so the numbers are the same for packages and files. From this experiment, both the multi-step approach and fine-tuning help the model generate syntactically valid code.

Interestingly, when evaluated on JS-SOURCED, StarCoder-7B MULTI almost always produces code that parses, while STENOTYPE *always* produces code that parses. This is because the models make very few changes to JavaScript input: usually the models do not generate any code and simply return the same code as the input, or they only add or delete single, small functions.

5.3.3.2 How Many Packages and Files Type Check?

Table 5.5 and Figure 5.5 show the number and percentage of packages and files that type check. Each package was migrated 20 times, so there are 1,000 packages in total, and each TS-SOURCED package contains only one file, so the file-level results for TS-SOURCED are identical to the package-level results. For this evaluation, a package type checks if the TypeScript compiler returns without errors. However, requiring an entire package

Table 5.4: Number and percentage of packages and files that parse. Each of the 50 packages was migrated 20 times, resulting in 1,000 packages total. Each TS-SOURCED package consists of a single file, so its results are not repeated in the table.

✓ = number of packages or files that parse

= total number of packages or files

% = percentage of packages or files that parse

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Packages			Files		
		✓	#	%	✓	#	%
(1)	StarCoder-7B SINGLE	938	1,000	93.8			
	StarCoder-7B MULTI	940	1,000	94.0			
	STENOType-7B MULTI	960	1,000	96.0			
(2)	StarCoder-7B SINGLE	890	1,000	89.0	2,401	2,511	95.6
	StarCoder-7B MULTI	992	1,000	99.2	2,582	2,590	99.7
	STENOType-7B MULTI	1,000	1,000	100.0	2,760	2,760	100.0

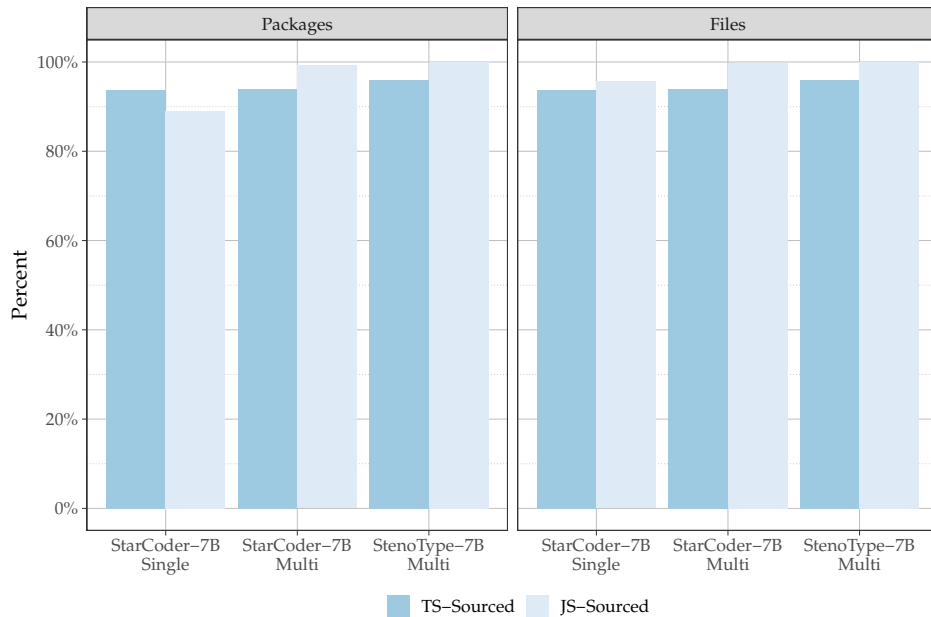


Figure 5.4: Percentage of packages and files that parse.

Table 5.5: Number and percentage of packages and files that type check. Each of the 50 packages was migrated 20 times, resulting in 1,000 packages total. Each TS-SOURCED package consists of a single file, so its results are not repeated in the table.

✓ = number of packages or files that type check

= total number of packages or files

% = percentage of packages or files that type check

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Packages			Files		
		✓	#	%	✓	#	%
(1)	StarCoder-7B SINGLE	254	1,000	25.4			
	StarCoder-7B MULTI	351	1,000	35.1			
	STENOType-7B MULTI	472	1,000	47.2			
(2)	StarCoder-7B SINGLE	113	1,000	11.3	1,404	2,511	55.9
	StarCoder-7B MULTI	128	1,000	12.8	1,520	2,590	58.7
	STENOType-7B MULTI	140	1,000	14.0	1,720	2,760	62.3

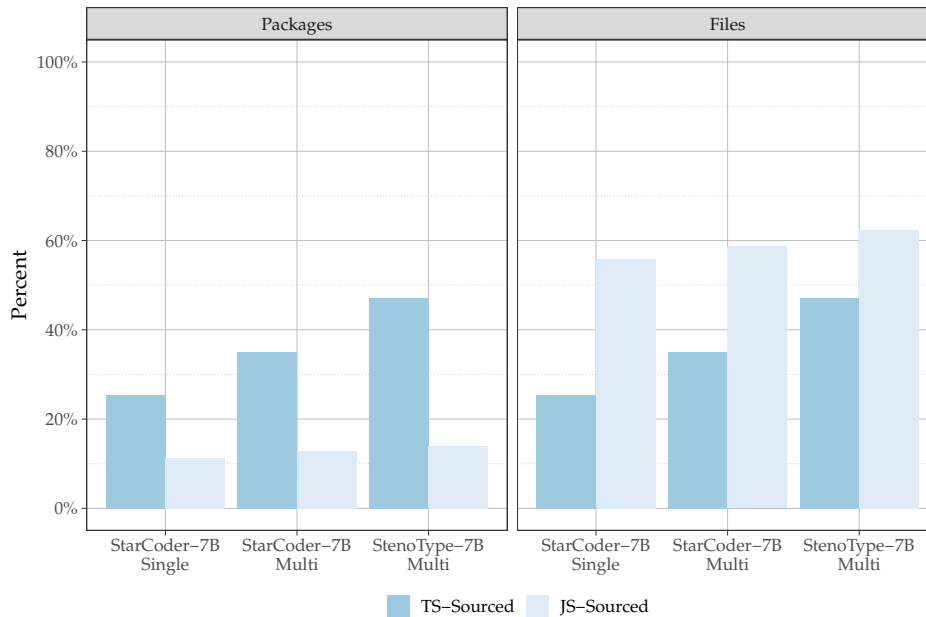


Figure 5.5: Percentage of packages and files that type check.

to type check means every file within that package must type check. Therefore, the table also shows the number of files that type check, i. e., there are no compiler errors attributed to that file. This is the same metric I used in [Section 3.3.3.2](#).

The results show that both the multi-step approach and fine-tuning improve the number of packages and files that type check. With TS-SOURCED, STENOTYPE has a 22% absolute improvement over the baseline, StarCoder-7B SINGLE. STENOTYPE also shows an improvement on JS-SOURCED; however, the 14% of packages that type check is the same set of packages that type checked before migration.

This is actually an example of a trivial migration: STENOTYPE simply returned the input without changing it. Other kinds of trivial migrations are possible; for example, the model could generate trivial type annotations and not require any new type definitions, or the model could delete code from the input.

5.3.3.3 *What Percentage of Type Annotations Are Trivial?*

The total number of trivial type annotations, in files with no errors, is shown in [Table 5.6](#) and [Figure 5.6](#). For TS-SOURCED, the multi-step approach generates fewer type annotations and significantly more trivial type annotations. With STENOTYPE, the number of annotations increases, but the percentage of trivial type annotations decreases. On average, each file has 9–13 type annotations, of which 1–2 are trivial. This shows that, for files with no errors, STENOTYPE produces mostly non-trivial type annotations. The table also shows the number of trivial type annotations in the *original* source files, i. e., before the types were removed for evaluation. For StarCoder-7B SINGLE and StarCoder-7B MULTI, there were more trivial type annotations in the original source files than were added by the migration.

For JS-SOURCED, the multi-step approach generates only four type annotations (none of which is trivial), while STENOTYPE generates mostly trivial type annotations. The original source files do not contain any handwritten type annotations, so they contain no trivial type annotations.

5.3.3.4 *Errors*

[Table 5.7](#) shows the total number of errors, as well as the average number of errors per package and per file. Each TS-SOURCED package contains only one file, so its per-file results are identical to the per-package results. For TS-SOURCED, StarCoder-7B MULTI produces more errors than StarCoder-7B SINGLE, but STENOTYPE produces the fewest. On the other hand, the number of errors for JS-SOURCED is more consistent.

Table 5.6: Number and percentage of type annotations that are any, any[], or Function, in files with no errors. The number of files with no errors is also shown.

Original = number of trivial type annotations in original source files

✓ = number of trivial type annotations

= total number of type annotations

% = percentage of trivial type annotations

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Files	Trivial annotations			
			Original	✓	#	%
(1)	StarCoder-7B SINGLE	254	1,288	276	3,370	8.2
	StarCoder-7B MULTI	351	1,010	848	3,291	25.8
	STENOType-7B MULTI	472	812	1,179	5,870	20.1
(2)	StarCoder-7B SINGLE	1,404		89	902	9.9
	StarCoder-7B MULTI	1,520		0	4	0.0
	STENOType-7B MULTI	1,720		150	217	69.1

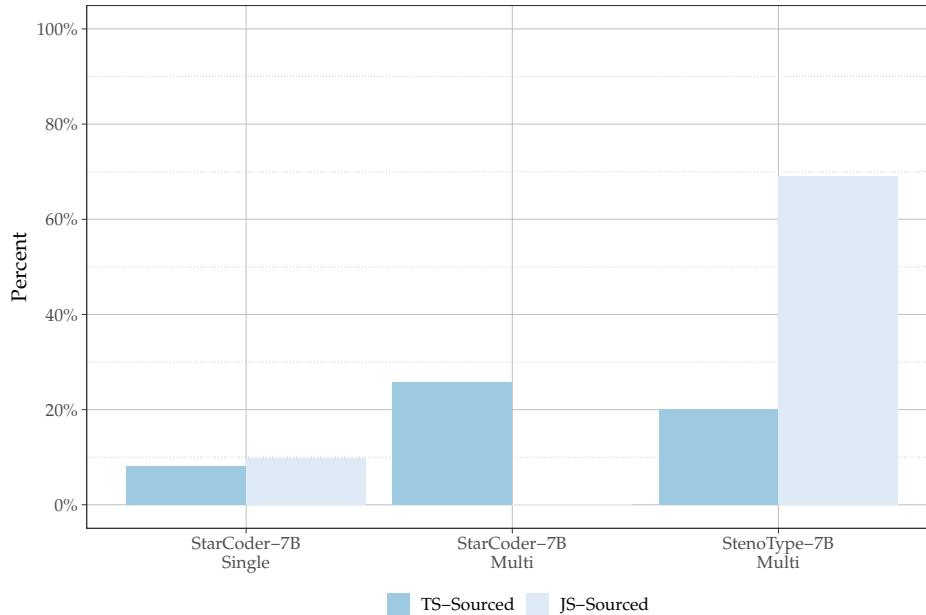


Figure 5.6: Percentage of type annotations that are any, any[], or Function, in files with no errors.

Table 5.7: Average number of errors per file and per package. Each TS-SOURCED package consists of a single file, so its results are not repeated in the table. The total number of errors is also shown.

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Errors		
		Total	Per package	Per file
(1)	StarCoder-7B SINGLE	3,105	3.1	
	StarCoder-7B MULTI	4,135	4.1	
	STENOTYPE-7B MULTI	2,498	2.5	
(2)	StarCoder-7B SINGLE	9,249	9.2	3.7
	StarCoder-7B MULTI	9,733	9.7	3.8
	STENOTYPE-7B MULTI	9,730	9.7	3.5

5.3.3.5 What Percentage of Annotation Sites Were Filled?

Table 5.8 and Figure 5.7 show the number and percentage of type annotation sites that have been filled with a type annotation. For TS-SOURCED, about 33–46% of type annotation sites are filled by type migration. The numbers are different for each configuration, because adding type definitions can add more type annotation sites. For JS-SOURCED, there are many more type annotation sites, but significantly fewer sites are filled by the model.

Not every type annotation site in a program needs to be filled. For example, the statement `const f = function() { return 1; }` has an annotation site for `f` and another for the anonymous function, but filling both sites would be verbose and redundant.

5.3.3.6 How Many Type Definitions Were Added and Used?

Table 5.9 and Figure 5.8 show the number of type definitions that were *added*, as a percentage of the total number of type definitions, within files that parse. Table 5.10 and Figure 5.9 are similar, but shows the number of type definitions that were *used*. Recall from Section 5.3.1 that I remove types to create the TS-SOURCED dataset, but I do not remove class definitions because they contain fields and methods.

For TS-SOURCED, STENOTYPE generates fewer type definitions; however a much higher percentage of those type definitions are actually used, i. e., they are referenced by type annotations. This is unsurprising, as the multi-step migration explicitly instructs STENOTYPE to generate definitions for

Table 5.8: Number and percentage of type annotation sites that were filled with a type annotation, in files that parse. The number of files that parse is also shown.

✓ = number of type annotation sites that were filled

= total number of type annotation sites

% = percentage of type annotation sites that were filled

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Files	Annotation sites		
			✓	#	%
(1)	StarCoder-7B SINGLE	938	14,669	36,123	40.6
	StarCoder-7B MULTI	940	11,711	35,242	33.2
	STENOType-7B MULTI	960	16,203	35,364	45.8
(2)	StarCoder-7B SINGLE	2,401	4,464	54,720	8.2
	StarCoder-7B MULTI	2,582	275	55,236	0.5
	STENOType-7B MULTI	2,760	1,662	57,483	2.9

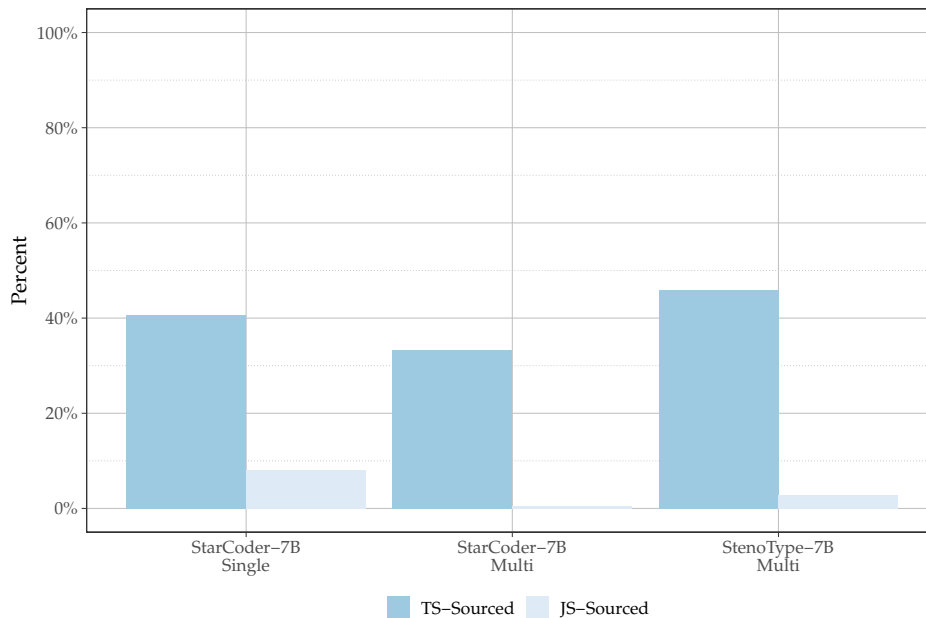


Figure 5.7: Percentage of type annotation sites that were filled with a type annotation, in files that parse.

Table 5.9: Number and percentage of type definitions that were added, out of the total number of type definitions present, since the input may contain class definitions. These results are only for files that parse. The number of files that parse is also shown.

+ = number of type definitions that were added

= total number of type definitions

% = percentage of type definitions that were added

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Files	Type definitions		
			+	#	%
(1)	StarCoder-7B SINGLE	938	1,226	2,259	54.3
	StarCoder-7B MULTI	940	1,218	2,087	58.4
	STENOTYPE-7B MULTI	960	802	1,447	55.4
(2)	StarCoder-7B SINGLE	2,401	406	529	76.7
	StarCoder-7B MULTI	2,582	23	103	22.3
	STENOTYPE-7B MULTI	2,760	3	83	3.6

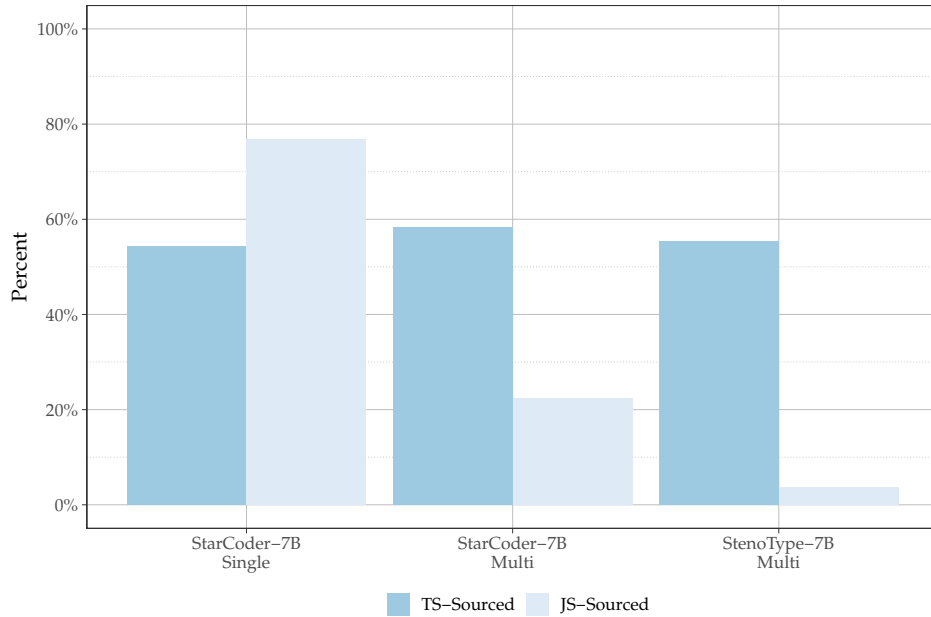


Figure 5.8: Percentage of type definitions that were added, out of the total number of type definitions present.

Table 5.10: Number and percentage of type definitions that were used, out of the total number of type definitions present, within the files that parse.

The number of files that parse is also shown.

✓ = number of type definitions that were used

= total number of type definitions

% = percentage of type definitions that were used

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Files	Type definitions		
			✓	#	%
(1)	StarCoder-7B SINGLE	938	776	2,259	34.3
	StarCoder-7B MULTI	940	908	2,087	43.5
	STENOTYPE-7B MULTI	960	844	1,447	58.3
(2)	StarCoder-7B SINGLE	2,401	277	529	52.4
	StarCoder-7B MULTI	2,582	13	103	12.6
	STENOTYPE-7B MULTI	2,760	12	83	14.5

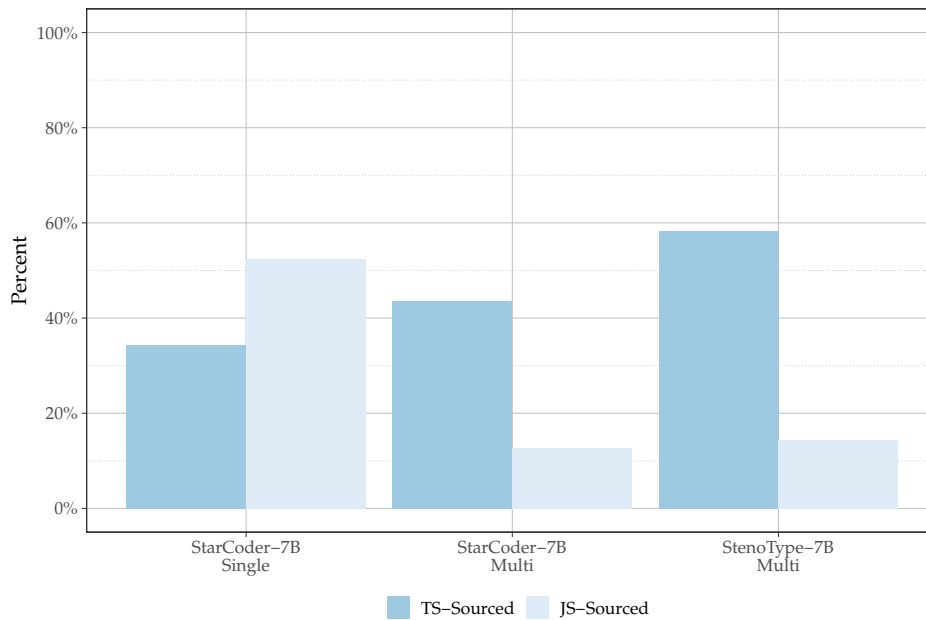


Figure 5.9: Percentage of type definitions that were used, out of the total number of type definitions present.

Table 5.11: Similarity between the (untyped) input code and the untyped output code, i. e., with type annotations and definitions removed.

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Similarity
(1)	StarCoder-7B SINGLE	0.926
	StarCoder-7B MULTI	0.957
	STENOTYPE-7B MULTI	0.982
(2)	StarCoder-7B SINGLE	0.886
	StarCoder-7B MULTI	0.963
	STENOTYPE-7B MULTI	0.9998

types that are actually used. On the other hand, for JS-SOURCED, STENOTYPE generates almost no type definitions.

5.3.3.7 Does the Code Change, Beyond Adding Types?

Ideally, when the model generates type annotations and type definitions, it does not change the existing code. Otherwise, automated migration could insert or delete code and affect program functionality. To measure this, I compute the normalized similarity between the original code (which is untyped) and the generated code with type annotations and type definitions removed. This is calculated as $1 - \frac{\text{distance}}{\text{len1} + \text{len2}}$, where distance is the Levenshtein distance and len1 and len2 are the number of characters of the input and output code (with types removed). In other words, if the model *only* adds annotations and definitions, then the similarity should be 1.0, since the two strings are identical after removing types.

Table 5.11 shows these results. Both multi-step migration and fine-tuning lead to better results, with the model being less likely to add or delete non-type code. STENOTYPE performs best on the JS-SOURCED dataset, but in the previous results, STENOTYPE was also unlikely to add type annotations or definitions to JS-SOURCED. This suggests that when given JavaScript, STENOTYPE does not change the input very much, and just returns the code as-is.

5.3.3.8 Does the Model Leave the Code Unchanged?

To determine whether the model makes any changes to the program during migration, I compare the input and output programs, requiring an exact match, and count the number of packages and files that are identical. The results are shown in Table 5.12 and Figure 5.10. For TS-SOURCED,

Table 5.12: Number and percentage of files and packages that were unchanged after migration. Each TS-SOURCED package consists of a single file, so its results are not repeated in the table.

✓ = number of packages or files that were unchanged

= total number of packages or files

% = percentage of packages or files that were unchanged

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Packages			Files		
		✓	#	%	✓	#	%
(1)	StarCoder-7B SINGLE	23	1,000	2.3			
	StarCoder-7B MULTI	45	1,000	4.5			
	STENOType-7B MULTI	0	1,000	0.0			
(2)	StarCoder-7B SINGLE	369	1,000	36.9	1,739	2,511	69.3
	StarCoder-7B MULTI	639	1,000	63.9	2,204	2,590	85.1
	STENOType-7B MULTI	852	1,000	85.2	2,583	2,760	93.6

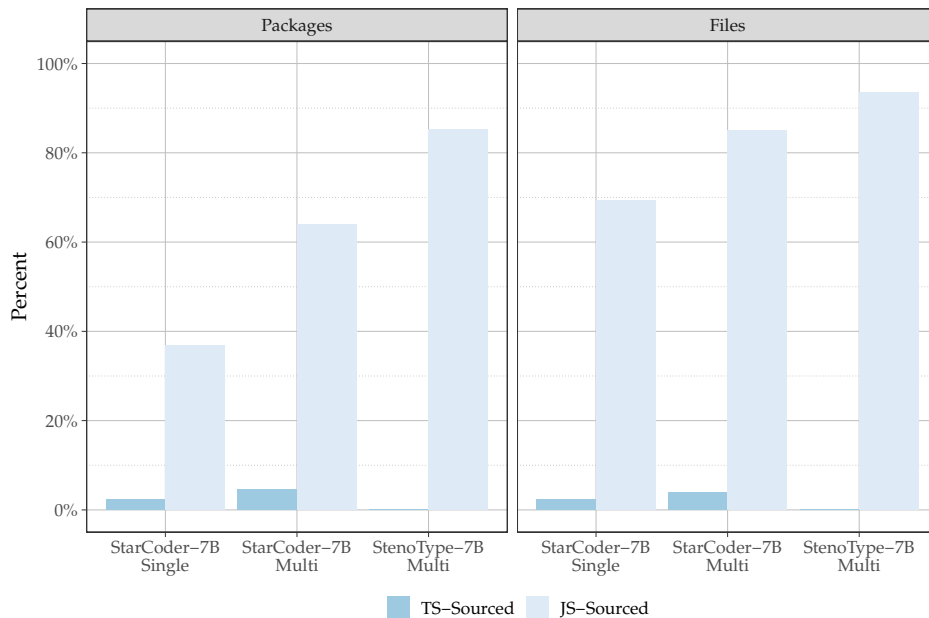


Figure 5.10: Percentage of files and packages that were unchanged after migration.

each package consists of a single file, so the results are not repeated. The results show that the model makes changes to almost every package, and with STENO`TYPE`, every package has some change. However, the opposite situation occurs with JS-SOURCED: most code is unchanged by the model, especially when moving to multi-step migration and STENO`TYPE`.

5.3.3.9 *What is the Success Rate of Migration, in the Pessimistic Case?*

Finally, I consider a pessimistic view of migration: a file successfully migrates if:

1. there are no compiler errors, i. e. the file type checks;
2. the input *exactly* matches the output with all types removed, i. e. the model did not add or delete code beyond type annotations and definitions; and
3. at least one type annotation or type definition was added, i. e. the model did not simply return the input.

These criteria are strict; however, they exclude several undesirable cases, such as migrations with incorrect types, migrations that modified the original code beyond adding types, and migrations that did nothing. The results are shown in [Table 5.13](#) and [Figure 5.11](#). The success rate for TS-SOURCED improves significantly when using multi-step migration, and fine-tuning STENO`TYPE` also helps. However, none of the models was able to correctly migrate a single file from the JS-SOURCED dataset.

5.3.4 *Case Studies*

5.3.4.1 *Generating Interfaces*

[Figure 5.12](#) shows an example of the output produced by STENO`TYPE`. The original file defines a `Tokenizer` class as well as several interfaces, and has been simplified for exposition. Before providing the file to STENO`TYPE`, all type annotations and interfaces are removed, but the class definition is kept, otherwise the entire file would be blank.

STENO`TYPE` adds the type annotation `IToken` on [line 631](#), as well as the interface definitions `IToken` ([lines 601–607](#)) and `ITokenizer` ([lines 608–614](#)), and the result type checks. The definition for `IToken` is type correct and consistent with the call on [line 623](#) and the definition on [line 631](#). However, compared to original definition shown in [Figure 5.12b](#), `value` ([line 603](#)) should have type `any` ([line 636](#)), and all the properties of `IToken` should be read only. On the other hand, the definition for `ITokenizer` matches the original definition, except that the original definition contains comments.

Table 5.13: Number and percentage of files that were correctly migrated, in the pessimistic case.

✓ = number of files that migrated correctly

= total number of files

% = percentage of files that migrated correctly

(1) = TS-SOURCED; (2) = JS-SOURCED

Dataset	Configuration	Correct		
		✓	#	%
(1)	StarCoder-7B SINGLE	48	1,000	4.8
	StarCoder-7B MULTI	210	1,000	21.0
	STENOTYPE-7B MULTI	257	1,000	25.7
(2)	StarCoder-7B SINGLE	0	2,511	0.0
	StarCoder-7B MULTI	0	2,590	0.0
	STENOTYPE-7B MULTI	0	2,760	0.0

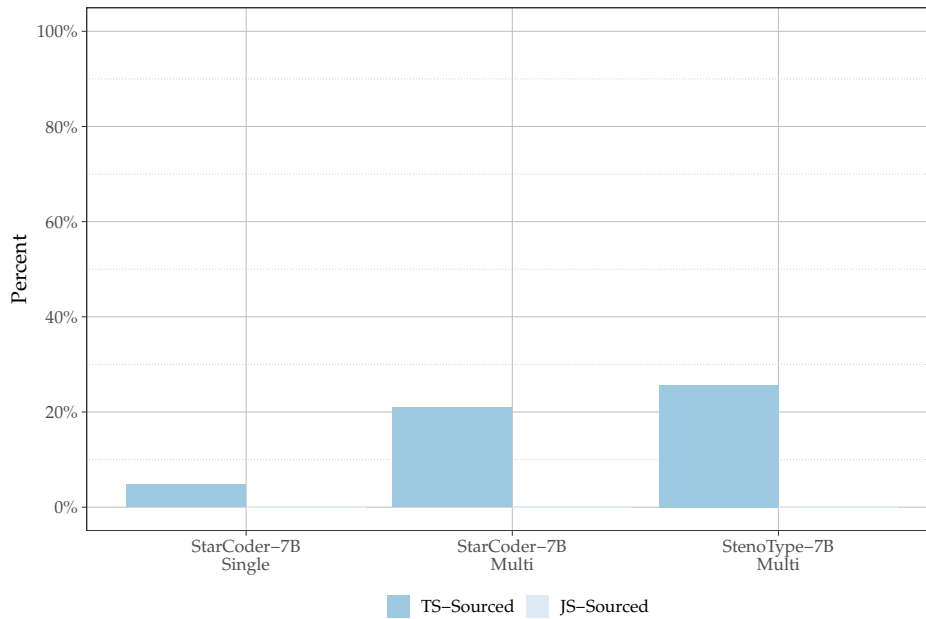


Figure 5.11: Percentage of files that were correct migrated, in the pessimistic case.

```

601 export interface IToken {
602     type: TokenType;
603     value?: number | string;
604     line: number;
605     column: number;
606     string: string;
607 }
608 export interface ITokenizer {
609     readNext(): void;
610     getCurrent(): IToken | undefined;
611     isAtEnd(): boolean;
612     getLine(): number;
613     getColumn(): number;
614 }
615
616 export class Tokenizer implements ITokenizer {
617     // many fields and methods omitted
618     private readNumber() {
619         while (this.isDigit(this.peek())) {
620             this.advance()
621         }
622         const stringValue = ... // omitted
623         this.setCurrent({
624             type: TokenType.NUMBER,
625             value: parseFloat(stringValue),
626             line: this.curTokenLine!,
627             column: this.curTokenColumn!,
628             string: stringValue,
629         });
630     }
631     private setCurrent(token: IToken) {
632         this.curToken = token;
633     }

```

(a) The highlighted type definitions and type annotation were added by STENO_{TYPE}.

```

634 export interface IToken {
635     readonly type: TokenType;
636     readonly value?: any;
637     readonly line: number;
638     readonly column: number;
639     readonly string: string;
640 }

```

(b) The original definition for IToken.

Figure 5.12: A (simplified) tokenizer class from a compiler.

Interestingly, the StarCoder-7B MULTI configuration is also able to generate type definitions and type annotations; however, it also deletes some code, which makes the output incorrect.

5.3.4.2 *Generating Nonsense*

In some cases, the model can go completely off track and generate nonsense. Figure 5.13 shows an example. The input (Figure 5.13a) is a class with several methods, which are omitted in this example. In the first step (Figure 5.13b), the model is instructed to add type annotations, and it does so on lines 651–653. However, it also tries to add definitions for those types, but instead of adding definitions, it adds imports from made-up modules on lines 645–647. Then, in the second step (Figure 5.13c), the model is prompted to add a type definition for `UnleashConfig`, but it generates meaningless code and continues doing so until it reaches the maximum number of tokens for its context window.

5.4 DISCUSSION

PROMPTS. My approach for the type definition generation problem is to use the Git commit prompt format, which provides code and an edit instruction to the model. In these experiments, as well as my preliminary experiments, I found that a simpler prompt, instructing the model to make smaller changes, was more effective than a complicated prompt that instructed the model to make significant changes to the code. Additionally, I found that it was more effective to instruct the model to make specific changes, i. e., adding a specific type definition. Intuitively, a simpler and more specific instruction makes the problem easier for the model, as fewer changes are required. This suggests further refinements to the prompt, for example, instructing the model to add a type definition with specific properties. However, one downside is that simpler prompts usually require more steps, which increases inference time.

JS-SOURCED DATASET. The JS-SOURCED dataset is more challenging than TS-SOURCED for three reasons. First, JS-SOURCED is constructed from JavaScript projects, rather than TypeScript files. Second, JS-SOURCED consists of multi-file projects, rather than single-file examples. Third, the multiple files in a project were bundled together for JS-SOURCED, but the bundling process introduces compilation artifacts that do not appear in handwritten code. Therefore, the JS-SOURCED consists of code and patterns that the model never observed during training. This explains the results I observed: most packages do not type check after migration, few anno-

tations and type definitions are added, and most of the time the model makes no changes to the code at all.

While I believe it is important to evaluate type migration systems on multi-file JavaScript projects, to better represent how type migration would occur in practice, I also believe the model needs to be trained on JavaScript as well as TypeScript. I leave this to future work.

```

641 export class UnleashFlags {
642   private constructor (features, config, context) {}
643   // omitted
644 }

```

(a) The original input is a class, `UnleashFlags`, which is mostly omitted in this example.

```

645 import { UnleashConfig } from './unleash-config'
646 import { UnleashContext } from './unleash-context'
647 import { UnleashFeature } from './unleash-feature'
648
649 export class UnleashFlags {
650   private constructor (
651     features: UnleashFeature[],
652     config: UnleashConfig,
653     context: UnleashContext
654   ) {}
655 }

```

(b) In the first step, StarCoder-7B MULTI adds type annotations, but also adds meaningless imports.

```

656 import { UnleashConfig } from './unleash-config'
657 import { UnleashContext } from './unleash-context'
658 import { UnleashFeature } from './unleash-feature'
659
660 export interface UnleashConfig {
661   unleashAPI: string
662   unleashAPIPath: string
663   unleashAPIVersion: string
664   unleashAPIMetricsPath: string
665   unleashAPIMetricsVersion: string
666   unleashAPIMetricsInterval: number
667   unleashAPIMetricsIntervalTime: number
668   unleashAPIMetricsResetInterval: number
669   unleashAPIMetricsResetIntervalTime: number
670   // over 900 lines
671 }

```

(c) In the second step, StarCoder-7B MULTI rewrites the class and generates nonsense.

Figure 5.13: StarCoder-7B MULTI generates nonsense for this example.

DISCUSSION

6.1 PERFORMANCE BOTTLENECK

There are several ways to improve the performance of these models, including better search, more training, and better datasets. I believe the performance bottleneck lies in better search, followed by better datasets. The search approaches I have been using are initial steps, and I believe there is much work that can be done here. Programs contain significant amounts of information that is not visible at the token level, such as type constraints and hierarchies, syntax, program structure, resolved names and identifiers, static call graphs, and so on. Exposing this information to a model and requiring the output to satisfy program constraints should yield better performance.

As a second step, I believe improving the quality of training data should also help. The training data I used is from The Stack, and there are no guarantees on code quality or complexity: most code appears to be small programs or examples, rather than large-scale applications. Training on higher quality code or synthetic data should improve performance.

I do not believe additional training, without improving the training dataset, will help. In fact, additional training is likely to lead to overfitting, as the model simply reproduces its training data as output.

6.2 REFACTORING

The type migration problem is ultimately a *refactoring* problem, as type migration involves more than simply adding type annotations to code. Programmers writing in an untyped language will use idioms and patterns that are different and not compatible with a typed version of that language. As Chung [2023, p. 75] writes, "in practice few untyped programs are actually typable without modification . . . [f]ew programmers write perfectly typable untyped code without the aid of a type checker."

Object Prototypes and Classes

Prior to the introduction of classes in ECMAScript 6, JavaScript objects used prototypes. For example, here is how a `Circle` class and `area` method could be defined in JavaScript:

```

672 function Circle(x, y, r) {
673   this.x = x;
674   this.y = y;
675   this.r = r;
676 }
677
678 Circle.prototype.area = function() {
679   return Math.PI * this.r * this.r;
680 }

```

With strict mode disabled, this program type checks with the TypeScript compiler. However, strict mode is often encouraged, as it allows more bugs to be detected. In this case, strict mode issues errors for [line 672](#) because the parameters implicitly have the any type, and [lines 673–675](#) because this implicitly has the any type.

These issues could be resolved by adding type annotations, but the more idiomatic TypeScript code is to declare a class with properties `x`, `y`, and `r`, and a constructor with type-annotated arguments:

```

681 class Circle {
682   x: number;
683   y: number;
684   r: number;
685
686   constructor(x: number, y: number, r: number) {
687     this.x = x;
688     this.y = y;
689     this.r = r;
690   }
691
692   area() {
693     return Math.PI * this.r * this.r;
694   }
695 }

```

Variable Used as Two Different Types

In [Section 3.3.6.4](#), I discussed an example where a variable `i` is used as a number in one part of the program, but then the same `i` is used as a string. A simplified version of that program is shown below:

```

696 let i;
697
698 for (i = 0; i < 5; i += 1) {

```

```

699     console.log(i)
700 }
701
702 const o = {a: 1, b: 2, c: 3};
703 for (i in o) {
704     console.log(i);
705 }

```

Neither number nor string type annotations work for `i` on [line 696](#), as this is code that relies on the dynamic type of variables. Although the TypeScript type system can accommodate this with the union type `number | string` or the trivial type `any`, a better solution would be to refactor the code to use two different variables.

Eval

`eval` is problematic because it allows arbitrary code to be evaluated, so it may not be possible to statically assign types. For example, in the code below, the `getProp` function on [line 707](#) returns either the string `"foo"` or `"bar"`, which is then used on [line 708](#) to access a property of `obj`:

```

706 const obj = {foo: 1, bar: "2"};
707 const prop = getProp(); // returns "foo" or "bar"
708 const v = eval("obj." + prop);

```

The only valid annotations for `v` are `any` or the union type `number | string`, but in general, it may not be possible to determine the type of the value returned by `eval`. In this case, a better approach would be to refactor the code to avoid `eval`, so that at least `number | string` could be used.

Sequentially Added Properties

In [Section 2.1](#), I discussed the example of dynamically adding properties to an empty object:

```

709 let point = {}
710 point.x = 42;
711 point.y = 54;

```

This was a common JavaScript idiom before the introduction of classes. There are several verbose type annotations that could be used for `point`, a new type could be defined, or the code could be refactored to define a class and use a constructor. Yet another approach is to initialize the properties from within the object:

```

712 let point = {

```

```

713   x: 42,
714   y: 54,
715 }

```

Incorrect Number of Function Arguments

JavaScript allows functions to be called with any number of arguments. Typically, this can be used with the special arguments object to implement variadic functions. For example, the following function adds its arguments and returns the sum:

```

716 function sum() {
717   let result = 0;
718   for (let i of arguments) {
719     result += i;
720   }
721   return result;
722 }
723 sum(1, 2, 3);

```

However, the call on [line 723](#) causes an error in TypeScript, because the type checker requires the number of arguments to match with the function definition, in this case, zero arguments. To make this valid TypeScript, the function must be refactored to take *rest parameters*, e. g. `...nums`:

```

724 function sum(...nums: number[]): number {
725   let result = 0;
726   for (let i of arguments) {
727     result += i;
728   }
729   return result;
730 }
731 sum(1, 2, 3);

```

DYNAMIC EVALUATION

All of the evaluation methods discussed in this dissertation are static: only the output code is examined, and the dynamic behaviour of the program is not considered. However, as type migration is a refactoring problem, these static metrics (such as accuracy, type checking, and counting trivial type annotations) may not be enough, and it may be useful to consider a dynamic evaluation, to ensure that the program behaviour does not change after migration.

One approach for a dynamic evaluation is to consider JavaScript programs with test suites. After migrating the program to TypeScript, we could compile back to JavaScript and then run the test suite. If the test suite passes both before and after migration, then we have some confidence that the program behaviour has not changed. However, this approach requires high quality test suites, so a tool like *npm-filter* [Arteca and Turcotte, 2022] could be used to automate the process of downloading, building, and testing JavaScript packages.

6.3 LIMITATIONS

Some limitations of my work are that I do not consider recursive types or generic types. In [Chapter 3](#), I evaluate DeepTyper and LambdaNet, which do not support generic types at all.

I also evaluate two models that support FIM, InCoder and StarCoder, and in [Chapter 4](#) I present OPENTAU, which supports FIT. These models may be able to predict type annotations that involve generics, e. g., `Array<string>`, if generics are present in the training data. However, these models cannot migrate an untyped function to a typed generic function:

```

732 // Untyped
733 function identity(x) { return x; }
734
735 // Generic function
736 function identity<T>(x: T): T { return x; }

```

Similarly, in [Chapter 5](#), STENOTYPE may be able to generate recursive types and generic types if they were present in the training data. An analysis of the training data (which contains roughly 9.7 million TypeScript files) shows that there are roughly 410,000 occurrences of generic functions and generic classes, and roughly 217,000 files (2.25% of the dataset) contain at least one generic.

Finally, as TypeScript evolves, new language features are being introduced, but these models are trained on programs from older versions of TypeScript. Therefore, a model may not be able to migrate code written in a newer version of TypeScript than the model was trained on.

RELATED WORK

7.1 GRADUAL TYPING FOR JAVASCRIPT

Although my dissertation focuses on type migration for TypeScript, there are several other gradual type systems for JavaScript and TypeScript. Thiemann [2005] and Anderson et al. [2005] present the earliest type systems for JavaScript, and later works present type systems and type checkers for JavaScript [Guha et al., 2011; Chugh et al., 2012; Lerner et al., 2013; Vekris et al., 2015; Chaudhuri et al., 2017], as well as sound, gradual type systems for TypeScript [Rastogi, Swamy, et al., 2015; Richards et al., 2015]. However, none of these systems support type inference, nor do they provide tools for type migration. Instead, like Typed Racket [Tobin-Hochstadt and Felleisen, 2008], they require programmers to manually migrate their code to add types.

7.2 CONSTRAINT-BASED TYPE INFERENCE

There are many constraint-based approaches to type migration for the gradually typed lambda calculus and some modest extensions. The earliest approach was a variation of unification-based type inference [Siek and Vachharajani, 2008], and more recent work uses a wide range of techniques [Campora et al., 2018; Castagna et al., 2019; Garcia and Cimini, 2015; Migeed and Palsberg, 2020; Miyazaki et al., 2019; Phipps-Costin et al., 2021; Campora et al., 2022; Mahmoud, 2023]. 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.

There are also several constraint-based approaches to type inference for larger languages. Anderson et al. [2005] present the first type inference algorithm for a fragment of JavaScript, but it does not support gradual typing. Furr, An, Foster, and Hicks [2009] infer types for Ruby and treat type annotations in a novel way: inference assumes that annotations are correct, and defers checking them to run time. Rastogi, Chaudhuri, et al. [2012] infer gradual types for ActionScript to improve performance, and Chandra et al. [2016] infer types for JavaScript programs with the goal of compiling them to run efficiently on resource-constrained devices, but their approach is not gradual and deliberately rejects certain programs.

Hassan et al. [2018] uses an SMT solver to infer types for Python programs, but their approach is also not gradual.

An alternative to purely static constraint-based approaches is to use dynamic approaches, or a combination of both static and dynamic techniques. For example, Furr, An, and Foster [2009] and An et al. [2011] use dynamic profiles to infer types for Ruby, while Saftoiu [2010] and Naus [2015] use a similar technique for JavaScript, and Hackett and S.-y. Guo [2012] and Kedlaya et al. [2013] use hybrid analyses to infer types to optimize code.

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 [Cartwright and Fagan, 1991] infers types for Scheme programs, but Flanagan [1997, p. 41] reports that it produces unintuitive types. For Ruby, Kazerounian, Ren, et al. [2020] use hand-coded heuristics to infer more natural types, and Kazerounian, Foster, et al. [2021] use machine learning to predict equalities between structural types and more natural types. Similarly, Pandi et al. [2021] infers more natural types for TypeScript, by combining logical constraints from the type system and natural constraints from a deep learning model that learns naming conventions from code examples.

A related problem to type inference is inferring the structure of ad hoc data, e. g. log files, based on observing example data. Fisher et al. [2008] propose an inference algorithm that discovers the structure of provided data and outputs a format specification that can then be used to generate further tools for data analysis. The algorithm considers base types and complex types built from base types, and tries to improve and simplify specifications by searching the description space for types that minimize an *information-theoretic* score, i. e., the search prefers descriptions that are compact and precise. This is particularly relevant to *type prediction*, which I discuss below (Section 7.3), where it is useful to search the space of type predictions for type correct and precise type annotations.

7.2.1 Inferring TypeScript Type Declarations

As an alternative to fully migrating a JavaScript project to TypeScript, TypeScript type declaration (`.d.ts`) files can be used. These files provide type annotations for functions and constants exported by JavaScript modules, so that type information is available when those modules are imported into TypeScript projects. I use type declaration files when I evaluate `TYPEWEAVER` and `STENOTYPE` on JavaScript packages with dependencies.

Although many type declaration files are available in the community-maintained `DefinitelyTyped` repository, there has been research in automatically generating these declarations with dynamic analysis [Kristensen

and Møller, 2017a; Kahlert, 2018; Cristiani and Thiemann, 2021]. Additionally, there is evidence that some of the DefinitelyTyped annotations are incorrect, and there are a variety of techniques to identify these bugs. For example, Feldthaus and Møller [2014] and Kristensen and Møller [2019] use static analysis, Williams et al. [2017] develop a tool based on the polymorphic blame calculus, and Kristensen and Møller [2017b] use feedback-directed random testing. More recently, Hoeflich et al. [2022] identifies mismatches between JavaScript code and their type declarations in DefinitelyTyped, by converting types into contracts and then checking them.

7.3 DEEP TYPE PREDICTION

7.3.1 *JavaScript and TypeScript*

One of the first works to study probabilistic type inference for JavaScript was JSNice [Raychev et al., 2015], which predicts program properties, including types, but it only supports a limited set of type names. DeepTyper [Hellendoorn et al., 2018] and LambdaNet [Wei et al., 2020b] are two different approaches for predicting types for TypeScript and JavaScript programs, which I discuss at length in Sections 3.1.1 and 3.1.2. DeepTyper was the first deep neural network for TypeScript type prediction, and uses a bidirectional recurrent neural network architecture, while LambdaNet uses a graph neural network architecture, which allows it to better represent relationships between type variables. Unlike DeepTyper, LambdaNet can predict types that were not observed during training, and ensures that multiple uses of the same variable have a consistent type.

NL2Type [Malik et al., 2019] is another system for predicting JavaScript types that improves on DeepTyper by considering comments as well as the names of identifiers. Khaled Saifullah et al. [2020] propose a technique that captures locally specific context and achieves similar accuracy as DeepTyper and NL2Type, but with significantly fewer resources. Like LambdaNet, Ye, Zhao, and Sarkar [2021] also use graph neural networks for type prediction, and their best model achieves higher accuracy than DeepTyper and LambdaNet. Stallenberg et al. [2022] apply an unsupervised probabilistic type inference approach to improve unit test generation for JavaScript.

OptTyper [Pandi et al., 2021] takes a different approach to the type inference problem: it extracts the logical constraints of a type system and the natural constraints learnt from a code base, combining both kinds of constraints into a single, continuous optimization problem. This allows OptTyper to predict types that respect (local) type constraints; however, this does not guarantee that the entire program type checks. OptTyper

achieves higher accuracy than DeepTyper and LambdaNet, but is limited to a vocabulary of 100 library types, and does not predict user-defined types at all.

TypeBert [Jesse, Devanbu, and Ahmed, 2021] predicts built-in types for TypeScript, and was followed up by DiverseTyper [Jesse, Devanbu, and Sawant, 2022], which can also predict user-defined types and achieves state-of-the-art accuracy on type prediction. Both models are BERT-style pre-trained models, which are closely related to transformer-based models like InCoder [Fried et al., 2023], i. e., they work by being trained on large amounts of data rather than using custom model architectures and training data. CodeTIDAL5 [Seidel et al., 2024] is another transformer model, based on T5, and is able to predict user-defined types. Unlike other works, Seidel et al. include a small evaluation on JavaScript, which is manually reviewed, but they do not use a type checker.

Other than TYPEWEAVER, there are few end-to-end deep type prediction tools for TypeScript. A recent one is FlexType [Voruganti et al., 2023], an editor plug-in that can take a type prediction model and automatically or interactively add type annotations to a JavaScript file. Additionally, FlexType uses smaller models that can run without a GPU.

7.3.2 Python

There are also several type prediction systems for Python. A distinction between Python type systems and TypeScript type systems is that Python code is predominantly nominally typed: the type of a variable is either a built-in type or a class, whereas TypeScript uses structural types.

The earliest type prediction system for Python is by Z. Xu et al. [2016], which collects type hints (e. g., from data flow, attribute accesses, variable names, and explicit type checks) and uses probabilistic inference. DLTPy [Boone et al., 2019] uses a recurrent neural network to make predictions based on comments and the names of identifiers.

Typilus [Allamanis et al., 2020] uses a graph neural network, can predict rare and user-defined types, and uses a type checker to filter out obviously incorrect type predictions. Typilus also includes an evaluation with a type checker, which tests each type prediction one at a time: the predicted type is inserted into a ground truth, annotated program, possibly replacing an existing annotation, and then type checked.

TypeWriter [Pradel et al., 2020] combines learning-based probabilistic type prediction with search-based type validation. First, a model predicts a ranked list of type annotations for each function argument and result type, using context from comments and identifier names. Then, TypeWriter searches the space of type assignments, using a type checker to validate type annotations and guarantee that no type assignment introduces a type

error. However, TypeWriter is limited to a fixed vocabulary of types, and cannot predict rare or user-defined types.

PYInfer [Cui et al., 2021] uses static analysis to generate a labeled dataset on which the model is trained, and can predict user-defined types. Type4Py [Mir, Latoškinas, Proksch, et al., 2022] is trained on a type-checked dataset, and also provides an editor plug-in for interactive use. HiTyper [Peng, Gao, et al., 2022] uses a combination of both static type inference and type prediction: when a type cannot be statically inferred, a deep learning model predicts a type that is validated by a type checker. Thus, HiTyper guarantees type correctness; however, it does not output type-annotated Python code, so additional work is required to use HiTyper as a type migration tool. TypeGen [Peng, Wang, et al., 2023] predicts types by generating *chain-of-thought* prompts that capture some of the type constraints in natural language prose. DLInfer [Y. Yan et al., 2023] uses static slicing to isolate variable usages before training a deep neural network. Ye, Zhao, Shirako, et al. [2023] use a combination of machine learning and SMT constraint solving to infer types, but their end goal is code optimization, not type migration.

7.3.3 Evaluation Datasets

Several of the previously mentioned works for type prediction also provide evaluation datasets [Hellendoorn et al., 2018; Jesse, Devanbu, and Ahmed, 2021; Allamanis et al., 2020; Pradel et al., 2020; Mir, Latoškinas, Proksch, et al., 2022; Y. Yan et al., 2023]. In addition, there are standalone datasets such as ManyTypes4Py [Mir, Latoškinas, and Gousios, 2021] and ManyTypes4TypeScript [Jesse and Devanbu, 2022], and Abdelaziz et al. [2022] present techniques for generating high quality type data for Python. However, many of these datasets are incomplete (e. g., they only contain URLs to repositories and not source code), or they contain only preprocessed data and not the original source files. Furthermore, their metrics are based on the accuracy of individual type annotations. TypeEvalPy [Venkatesh et al., 2024] is notable for presenting a handwritten micro-benchmark of Python programs and extensive evaluation scripts for type prediction, but their metrics are also based on accuracy and they do not do type migration or type checking.

7.4 CODE GENERATION

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 [Chen et al., 2021],

InCoder [Fried et al., 2023], SantaCoder [Ben Allal et al., 2023], and StarCoder [Li et al., 2023a]. For code generation tasks that require edit-style generation, *fill-in-the-middle* training and inference strategies have been proposed [Ben Allal et al., 2023; Fried et al., 2023; Bavarian et al., 2022].

Code LLMs are typically evaluated on benchmarks such as HumanEval [Chen et al., 2021], which evaluate the functional correctness of small functions; however, there is interest in constructing benchmarks that evaluate a code LLM's ability on a larger variety and higher difficulty of tasks, such as generating classes and translating code. These include CrossCodeBench [Niu et al., 2023], ClassEval [Du et al., 2023], and CodeScope [W. Yan et al., 2024]. The type definition generation task is an example of a more challenging task that has not been studied or evaluated extensively. However, the more general problem of type migration can be considered an instance of *code translation*, i. e., a translation from JavaScript to TypeScript. Both CrossCodeBench and CodeScope include code translation tasks in their benchmarks.

FUTURE WORK

DATASET QUALITY. There is always room to improve dataset quality, both for training and evaluation datasets. For training, my datasets have been constructed from The Stack, a collection of permissively licensed source code. However, there are no guarantees on code quality: the code may not type check or even parse, may be throwaway code, and may not even have any useful type annotations or type definitions to train on. Finding or manually constructing a source of high quality code should help with training.

There are similar issues with evaluation datasets, as well as additional challenges. First, when using publicly available code for evaluation, there is always the potential for the evaluation code to leak into the training dataset. Second, most dataset infrastructure and models expect a single file as output, but a more realistic evaluation should use multi-file projects. Although there are tools that can bundle multi-file projects into a single file, they can introduce compilation artifacts that confuse the model. Therefore, a different approach for multi-file projects should be explored.

EVALUATION CRITERIA. I believe that type checking the output of type prediction models is an important first step. However, there are other evaluation criteria that could be explored, such as the precision of a type annotation, partial typedness (e. g., inheritance), permitting “slightly wrong” type annotations, and even examining the run-time behaviour of migrated programs.

TYPE PREDICTION, REVISITED. Currently, `TYPEWEAVER` invokes a type prediction model and chooses the top, most likely prediction. On the other hand, `OPENTAU` incorporates search, and tries to find type predictions with more type information. There is potential to explore different search strategies, such as type checking predictions. However, this is not straightforward: type checking an expression may depend on the type of another (unannotated) expression, and sometimes the TypeScript compiler infers a type that is too precise and fails to type check. Therefore, the type checking procedure would need to be relaxed.

Another avenue to explore in `OPENTAU` is to incorporate additional context in the prompt. Currently, `OPENTAU` inserts usage comments, which show examples of a function being invoked. However, there may be other useful context that could be included.

Finally, similar to `STENOTYPE`, type prediction could take place over several steps. For example, once the initial type predictions are assigned, a type checker could be invoked. Then, if there are any type errors, a single type error could be extracted and used to construct a new prompt for the model to revise its type predictions.

GENERATING TYPE DEFINITIONS, REVISITED. While `STENOTYPE` is successful in generating type definitions for TypeScript code, the results are mixed for JavaScript. One potential reason is that `STENOTYPE` was trained on TypeScript code and not JavaScript. Therefore, an alternate approach is to train a model on JavaScript, specifically examples of JavaScript being migrated to TypeScript. One way is to manually construct these training examples, or compile a TypeScript program to JavaScript, and then train the model to essentially decompile JavaScript back to TypeScript. However, one potential challenge is that the compilation process introduces artifacts that do not commonly occur in handwritten JavaScript.

An alternative approach is to generate type definitions first, which reflects how a programmer writes in a statically typed programming language: defining types before use. The idea is to use constraint-based type inference (potentially using an unsound analysis, to avoid the challenges of a sound analysis) to generate a possibly verbose structural type definition, and then use machine learning to generate a name for that type. However, the challenge then would be to evaluate the quality of a type name.

Finally, there is an approach that does not require machine learning. The idea is to process the training dataset and create a database of type definitions. Thus, if a type-annotated program refers to an undefined type `T`, the database is queried for `T`'s definition, which is then inserted into the program. However, the challenge here is that there may be multiple definitions for `T`, so the right one must be selected.

FULLY AUTOMATED TYPE MIGRATION. My dissertation focuses on *partial* type migration, as a full migration may involve refactoring code. I intentionally leave this as future work, as the scope of the problem is too large for a single dissertation. Current `LLMs` may not be able to fully migrate code perfectly, but the hope is that they can still reduce the overall burden on programmers.

END-TO-END TOOLING. Currently, there are few end-to-end migration tools, or even end-to-end type prediction tools. While research prototypes are important, the desired outcome is to have end-to-end tools that programmers can use. I believe that it is important to build these tools, not only so they can be used, but also because the process of building them will uncover new problems to study.

CONCLUSION

In this dissertation, I set out to show that machine learning can be used to partially migrate JavaScript programs to TypeScript, by predicting type annotations and generating type definitions.

First, in [Chapter 3](#), I show that we should evaluate type prediction systems by *type checking* the generated types, instead of using accuracy. As part of this work, I built `TYPEWEAVER`, an end-to-end system for evaluating type prediction systems, and presented a dataset of 506 widely used JavaScript packages that are suitable for type migration. I show that StarCoder-Base can migrate 68.8% of files in the dataset, and observe that certain patterns in JavaScript do not make sense in TypeScript, so a migration may require manual refactoring of the code.

Next, in [Chapter 4](#), I present `OPENTAU`, a search-based approach for type prediction that uses `LLMs` and a new training technique called *fill in the type (FIT)*. I show that `OPENTAU` outperforms simpler approaches for type prediction that do not exploit *program decomposition*, and that the *typedness* metric allows `OPENTAU` to search for more precise types.

Finally, in [Chapter 5](#), I present `STENOTYPE`, an `LLM` fine-tuned to generate TypeScript type definitions. I describe the training format for `STENOTYPE`, which involves multiple steps of prompting with specific instructions, and show that `STENOTYPE` is effective at generating TypeScript type definitions.

BIBLIOGRAPHY

- Ibrahim Abdelaziz, Julian Dolby, and Kavitha Srinivas (2022). *Large Scale Generation of Labeled Type Data for Python*. DOI: [10.48550/arXiv.2201.12242](https://doi.org/10.48550/arXiv.2201.12242) (cited on p. 111).
- Miltiadis Allamanis, Earl T. Barr, Soline Ducouso, and Zheng Gao (2020). “Typilus: Neural Type Hints.” In: *Programming Language Design and Implementation (PLDI)*. DOI: [10.1145/3385412.3385997](https://doi.org/10.1145/3385412.3385997) (cited on pp. 2, 110–111).
- 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)*. DOI: [10.1145/1926385.1926437](https://doi.org/10.1145/1926385.1926437) (cited on p. 108).
- Christopher Anderson, Paola Giannini, and Sophia Drossopoulou (2005). “Towards Type Inference for JavaScript.” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.1007/11531142_19](https://doi.org/10.1007/11531142_19) (cited on pp. 2, 107).
- Ellen Arteca and Alexi Turcotte (2022). “npm-filter: Automating the mining of dynamic information from npm packages.” In: *Mining Software Repositories (MSR)*. DOI: [10.1145/3524842.3528501](https://doi.org/10.1145/3524842.3528501) (cited on p. 105).
- 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, Baisakhi 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)*. DOI: [10.48550/arXiv.2210.14868](https://doi.org/10.48550/arXiv.2210.14868) (cited on p. 2).
- 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*. DOI: [10.48550/arXiv.2108.07732](https://doi.org/10.48550/arXiv.2108.07732) (cited on p. 2).
- Luke Autry (2019). *How we failed, then succeeded, at migrating to TypeScript*. <https://heap.io/blog/migrating-to-typescript>. Accessed: 2022-12-01 (cited on p. 1).
- 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*. DOI: [10.48550/arXiv.2207.14255](https://doi.org/10.48550/arXiv.2207.14255) (cited on pp. 10, 13, 57, 112).

- Loubna Ben Allal (2023). *Fine-tuning SantaCoder for Code/Text Generation*. <https://github.com/loubnabnl/santacoder-finetuning/>. Accessed: 2023-04-21 (cited on p. 57).
- 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!* DOI: [10.48550/arXiv.2301.03988](https://doi.org/10.48550/arXiv.2301.03988) (cited on pp. 2–3, 10, 12, 16, 57, 112).
- Gavin Bierman, Martín Abadi, and Mads Torgersen (2014). “Understanding TypeScript.” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.1007/978-3-662-44202-9_11](https://doi.org/10.1007/978-3-662-44202-9_11) (cited on pp. 1, 7–8).
- Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt (2016). “Practical Optional Types for Clojure.” In: *European Symposium on Programming (ESOP)*. DOI: [10.1007/978-3-662-49498-1_4](https://doi.org/10.1007/978-3-662-49498-1_4) (cited on p. 1).
- Casper Boone, Niels de Bruin, Arjan Langerak, and Fabian Stelmach (2019). *DLTPy: Deep Learning Type Inference of Python Function Signatures using Natural Language Context*. DOI: [10.48550/arXiv.1912.00680](https://doi.org/10.48550/arXiv.1912.00680) (cited on p. 110).
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (2020). *Language Models are Few-Shot Learners*. DOI: [10.48550/arXiv.2005.14165](https://doi.org/10.48550/arXiv.2005.14165) (cited on p. 11).
- Ryan Burgess, Joe King, Stacy London, Sumana Mohan, and Jem Young (2022). *TypeScript migration - Strict type of cocktails*. <https://frontend.happyhour.com/episodes/typescript-migration-strict-type-of-cocktails>. Accessed: 2022-12-01 (cited on p. 1).
- John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw (2018). “Migrating Gradual Types.” In: *Proc. ACM Program. Lang.* 2.POPL. DOI: [10.1145/3158103](https://doi.org/10.1145/3158103) (cited on p. 107).

- John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw (2022). “Migrating gradual types.” In: *Journal of Functional Programming (JFP)* 32. DOI: [10.1017/S0956796822000089](https://doi.org/10.1017/S0956796822000089) (cited on p. 107).
- Robert Cartwright and Mike Fagan (1991). “Soft Typing.” In: *Programming Language Design and Implementation (PLDI)*. DOI: [10.1145/113445.113469](https://doi.org/10.1145/113445.113469) (cited on p. 108).
- 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.” In: *IEEE Transactions on Software Engineering (TSE)* 49.7. DOI: [10.1109/TSE.2023.3267446](https://doi.org/10.1109/TSE.2023.3267446) (cited on p. 2).
- Federico Cassano, Ming-Ho Yee, Noah Shinn, Arjun Guha, and Steven Holtzen (2023a). *OpenTau*. <https://github.com/GammaTauAI/opentau>. Accessed: 2023-08-01 (cited on p. 48).
- Federico Cassano, Ming-Ho Yee, Noah Shinn, Arjun Guha, and Steven Holtzen (2023b). *Type Prediction With Program Decomposition and Fill-in-the-Type Training*. DOI: [10.48550/arXiv.2305.17145](https://doi.org/10.48550/arXiv.2305.17145) (cited on pp. 2, 4, 47–48).
- 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)*. DOI: [10.1145/3427081.3427084](https://doi.org/10.1145/3427081.3427084) (cited on p. 1).
- Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek (2019). “Gradual Typing: A New Perspective.” In: *Proc. ACM Program. Lang.* 3.POPL. DOI: [10.1145/3290329](https://doi.org/10.1145/3290329) (cited on p. 107).
- 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)*. DOI: [10.1145/2983990.2984017](https://doi.org/10.1145/2983990.2984017) (cited on pp. 2, 107).
- Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi (2017). “Fast and Precise Type Checking for JavaScript.” In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: [10.1145/3133872](https://doi.org/10.1145/3133872) (cited on pp. 1, 107).
- 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*. DOI: [10.48550/arXiv.2107.03374](https://doi.org/10.48550/arXiv.2107.03374) (cited on pp. 2, 11, 82, 111–112).
- 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*. DOI: [10.48550/arXiv.2207.11280](https://doi.org/10.48550/arXiv.2207.11280) (cited on p. 2).
- Ravi Chugh, David Herman, and Ranjit Jhala (2012). “Dependent Types for JavaScript.” In: *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. DOI: [10.1145/2384616.2384659](https://doi.org/10.1145/2384616.2384659) (cited on p. 107).
- Benjamin Chung (2023). “A Type System for Julia.” PhD thesis. URL: <https://doi.org/10.17760/D20531813> (cited on p. 101).
- Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek (2018). “KafKa: Gradual Typing for Objects.” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.4230/LIPIcs.ECOOP.2018.12](https://doi.org/10.4230/LIPIcs.ECOOP.2018.12) (cited on p. 8).
- Fernando Cristiani and Peter Thiemann (2021). “Generation of TypeScript declaration files from JavaScript code.” In: *International Conference on Managed Programming Languages and Runtimes (MPLR)*. DOI: [10.1145/3475738.3480941](https://doi.org/10.1145/3475738.3480941) (cited on p. 109).
- Siwei Cui, Gang Zhao, Zeyu Dai, Luochao Wang, Ruihong Huang, and Jeff Huang (2021). *PYInfer: Deep Learning Semantic Type Inference for Python Variables*. DOI: [10.48550/arXiv.2106.14316](https://doi.org/10.48550/arXiv.2106.14316) (cited on p. 111).
- DefinitelyTyped contributors (2012). *DefinitelyTyped*. <https://github.com/DefinitelyTyped/DefinitelyTyped/>. Accessed: 2022-06-01 (cited on p. 28).
- Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou (2023). *ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation*. DOI: <https://doi.org/10.48550/arXiv.2308.01861> (cited on p. 112).
- Asger Feldthaus and Anders Møller (2014). “Checking Correctness of TypeScript Interfaces for JavaScript Libraries.” In: *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. DOI: [10.1145/2660193.2660215](https://doi.org/10.1145/2660193.2660215) (cited on pp. 28, 109).

- 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)*. DOI: [10.1145/1328438.1328488](https://doi.org/10.1145/1328438.1328488) (cited on p. 108).
- Cormac Flanagan (1997). “Effective Static Debugging via Componential Set-based Analysis.” PhD thesis. Rice University. URL: <https://users.soe.ucsc.edu/~cormac/papers/thesis.pdf> (cited on p. 108).
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis (2022). *InCoder 6B*. <https://huggingface.co/facebook/incoder-6B>. Accessed: 2022-06-01 (cited on p. 23).
- 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)*. DOI: [10.48550/arXiv.2204.05999](https://doi.org/10.48550/arXiv.2204.05999) (cited on pp. 3, 10, 13, 15–16, 18, 20, 55, 57, 110, 112).
- 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)*. DOI: [10.1145/1640089.1640110](https://doi.org/10.1145/1640089.1640110) (cited on p. 108).
- Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks (2009). “Static Type Inference for Ruby.” In: *Symposium on Applied Computing (SAC)*. DOI: [10.1145/1529282.1529700](https://doi.org/10.1145/1529282.1529700) (cited on p. 107).
- Ronald Garcia and Matteo Cimini (2015). “Principal Type Schemes for Gradual Programs.” In: *Principles of Programming Languages (POPL)*. DOI: [10.1145/2676726.2676992](https://doi.org/10.1145/2676726.2676992) (cited on p. 107).
- GitHub, Inc. (2023). *The most popular programming languages*. <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/#the-most-popular-programming-languages>. Accessed: 2024-02-01 (cited on p. 1).
- Ben Greenman and Matthias Felleisen (2018). “A Spectrum of Type Soundness and Performance.” In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: [10.1145/3236766](https://doi.org/10.1145/3236766) (cited on p. 8).
- Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi (2011). “Typing Local Control and State Using Flow Analysis.” In: *European Symposium on Programming (ESOP)*. DOI: [10.1007/978-3-642-19718-5_14](https://doi.org/10.1007/978-3-642-19718-5_14) (cited on pp. 1, 107).
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang (2024). *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. DOI: [10.48550/arXiv.2401.14196](https://doi.org/10.48550/arXiv.2401.14196) (cited on p. 10).

- Brian Hackett and Shu-yu Guo (2012). “Fast and Precise Hybrid Type Inference for JavaScript.” In: *Programming Language Design and Implementation (PLDI)*. DOI: [10.1145/2254064.2254094](https://doi.org/10.1145/2254064.2254094) (cited on pp. 2, 108).
- Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller (2018). “MaxSMT-Based Type Inference for Python 3.” In: *Computer Aided Verification (CAV)*. DOI: [10.1007/978-3-319-96142-2_2](https://doi.org/10.1007/978-3-319-96142-2_2) (cited on p. 108).
- 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)*. DOI: [10.1145/3236024.3236051](https://doi.org/10.1145/3236024.3236051) (cited on pp. 2, 10, 15–16, 20, 52, 109, 111).
- Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis (2019). *DeepTyper*. <https://github.com/DeepTyper/DeepTyper/tree/master/pretrained>. Accessed: 2022-06-01 (cited on p. 23).
- 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.” In: *Proc. ACM Program. Lang.* 6.OOPSLA2. DOI: [10.1145/3563305](https://doi.org/10.1145/3563305) (cited on pp. 28, 109).
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen (2022). “LoRA: Low-Rank Adaptation of Large Language Models.” In: *International Conference on Learning Representations (ICLR)*. DOI: [10.48550/arXiv.2106.09685](https://doi.org/10.48550/arXiv.2106.09685) (cited on p. 80).
- Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt (2020). *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. DOI: [10.48550/arXiv.1909.09436](https://doi.org/10.48550/arXiv.1909.09436) (cited on p. 2).
- 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)*. DOI: [10.1145/3510003.3510172](https://doi.org/10.1145/3510003.3510172) (cited on p. 2).
- Kevin Jesse and Premkumar T. Devanbu (2022). “ManyTypes4TypeScript: A Comprehensive TypeScript Dataset for Sequence-Based Type Inference.” In: *Mining Software Repositories (MSR)*. DOI: [10.1145/3524842.3528507](https://doi.org/10.1145/3524842.3528507) (cited on pp. 2, 9, 111).
- 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)*. DOI: [10.1145/3468264.3473135](https://doi.org/10.1145/3468264.3473135) (cited on pp. 2, 15, 52, 110–111).
- Kevin Jesse, Premkumar T. Devanbu, and Anand Ashok Sawant (2022). “Learning To Predict User-Defined Types.” In: *IEEE Transactions on Software Engineering (TSE)*. DOI: [10.1109/TSE.2022.3178945](https://doi.org/10.1109/TSE.2022.3178945) (cited on pp. 2, 15, 52, 110).

- Tobias Kahlert (2018). “Data Flow Based Type Inference for JavaScript.” MA thesis. Karlsruhe Institute of Technology. URL: <https://pp.ipd.kit.edu/uploads/publikationen/kahlert18masterarbeit.pdf> (cited on pp. 2, 109).
- Milod Kazerounian, Jeffrey S. Foster, and Bonan Min (2021). “SimTyper: Sound Type Inference for Ruby Using Type Equality Prediction.” In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: [10.1145/3485483](https://doi.org/10.1145/3485483) (cited on p. 108).
- Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster (2020). “Sound, Heuristic Type Annotation Inference for Ruby.” In: *Dynamic Languages Symposium (DLS)*. DOI: [10.1145/3426422.3426985](https://doi.org/10.1145/3426422.3426985) (cited on p. 108).
- Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf (2013). “Improved Type specialization for Dynamic Scripting Languages.” In: *Dynamic Languages Symposium (DLS)*. DOI: [10.1145/2508168.2508177](https://doi.org/10.1145/2508168.2508177) (cited on p. 108).
- C. M. Khaled Saifullah, Muhammad Asaduzzaman, and Chanchal K. Roy (2020). “Exploring Type Inference Techniques of Dynamically Typed Languages.” In: *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. DOI: [10.1109/SANER48275.2020.9054814](https://doi.org/10.1109/SANER48275.2020.9054814) (cited on p. 109).
- 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*. DOI: [10.48550/arXiv.2211.15533](https://doi.org/10.48550/arXiv.2211.15533) (cited on pp. 2, 9, 19, 57, 59, 79).
- Erik Krogh Kristensen and Anders Møller (2017a). “Inference and Evolution of TypeScript Declaration Files.” In: *Fundamental Approaches to Software Engineering (FASE)*. DOI: [10.1007/978-3-662-54494-5_6](https://doi.org/10.1007/978-3-662-54494-5_6) (cited on pp. 28, 108).
- Erik Krogh Kristensen and Anders Møller (2017b). “Type Test Scripts for TypeScript Testing.” In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: [10.1145/3133914](https://doi.org/10.1145/3133914) (cited on pp. 28, 109).
- Erik Krogh Kristensen and Anders Møller (2019). “Reasonably-Most-General Clients for JavaScript Library Analysis.” In: *International Conference on Software Engineering (ICSE)*. DOI: [10.1109/ICSE.2019.00026](https://doi.org/10.1109/ICSE.2019.00026) (cited on p. 109).
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica (2023). “Efficient Memory Management for Large Language Model Serving with PagedAttention.” In: *Symposium on Operating Systems Principles (SOSP)*. DOI: <https://doi.org/10.1145/3600006.3613165> (cited on p. 82).
- Nolan Lawson (2016). *cjs-to-es6*. <https://github.com/nolanlawson/cjs-to-es6>. Accessed: 2022-06-01 (cited on p. 22).

- Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi (2013). “TeJaS: Retrofitting Type Systems for JavaScript.” In: *Dynamic Languages Symposium (DLS)*. DOI: [10.1145/2578856.2508170](https://doi.org/10.1145/2578856.2508170) (cited on p. 107).
- 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 (2023a). *StarCoder: may the source be with you!* DOI: [10.48550/arXiv.2305.06161](https://doi.org/10.48550/arXiv.2305.06161) (cited on pp. 3, 10, 12, 15–16, 19–20, 112).
- 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 (2023b). *StarCoderBase*. <https://huggingface.co/bigcode/starcoderbase>. Accessed: 2023-06-01 (cited on p. 23).
- Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi (2022). “Gradual Soundness: Lessons from

- Static Python.” In: *The Art, Science, and Engineering of Programming* 7.1. DOI: [10.22152/programming-journal.org/2023/7/2](https://doi.org/10.22152/programming-journal.org/2023/7/2) (cited on p. 1).
- Zeina Mohamed Magdy Abdelmigeed Mahmoud (2023). “Theory and Applications for Gradual Type Migration.” PhD thesis. University of California Los Angeles. URL: <https://www.proquest.com/dissertations-theses/theory-applications-gradual-type-migration/docview/2885427272/se-2> (cited on p. 107).
- 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)*. DOI: [10.1109/ICSE.2019.00045](https://doi.org/10.1109/ICSE.2019.00045) (cited on pp. 2, 109).
- Meta Platforms, Inc. (2019). *Pyre: A performant type-checker for Python 3*. <https://pyre-check.org/>. Accessed: 2022-12-01 (cited on p. 1).
- Microsoft Corp. (2019). *TypeScript Documentation: Do’s and Don’ts*. <https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html#number-string-boolean-symbol-and-object>. Accessed: 2022-12-01 (cited on p. 26).
- Microsoft Corp. (2020). *TypeScript Documentation: Narrowing*. <https://www.typescriptlang.org/docs/handbook/2/narrowing.html>. Accessed: 2022-12-01 (cited on p. 44).
- Zeina Migeed and Jens Palsberg (2020). “What Is Decidable about Gradual Types?” In: *Proc. ACM Program. Lang.* 4.POPL. DOI: [10.1145/3371097](https://doi.org/10.1145/3371097) (cited on pp. 6, 107).
- Amir M. Mir, Evaldas Latoškinas, and Georgios Gousios (2021). “Many-Types4Py: A Benchmark Python Dataset for Machine Learning-Based Type Inference.” In: *Mining Software Repositories (MSR)*. DOI: [10.1109/MSR52588.2021.00079](https://doi.org/10.1109/MSR52588.2021.00079) (cited on pp. 2, 111).
- 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)*. DOI: [10.1145/3510003.3510124](https://doi.org/10.1145/3510003.3510124) (cited on pp. 2, 111).
- Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi (2019). “Dynamic Type Inference for Gradual Hindley–Milner Typing.” In: *Proc. ACM Program. Lang.* 3.POPL. DOI: [10.1145/3290331](https://doi.org/10.1145/3290331) (cited on p. 107).
- 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 (cited on p. 1).
- Nico Naus (2015). “Dynamic type inference for JavaScript.” MA thesis. Utrecht University. URL: <https://studenttheses.uu.nl/handle/20.500.12932/22912> (cited on pp. 2, 108).
- 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)*. DOI: [10.48550/arXiv.2203.13474](https://doi.org/10.48550/arXiv.2203.13474) (cited on p. 2).
- Changan Niu, Chuanyi Li, Vincent Ng, and Bin Luo (2023). *CrossCodeBench: Benchmarking Cross-Task Generalization of Source Code Models*. DOI: <https://doi.org/10.48550/arXiv.2302.04030> (cited on p. 112).
- Guilherme Ottoni (2018). “HHVM JIT: A Profile-Guided, Region-Based Compiler for PHP and Hack.” In: *Programming Language Design and Implementation (PLDI)*. DOI: [10.1145/3192366.3192374](https://doi.org/10.1145/3192366.3192374) (cited on p. 1).
- Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton (2021). *OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints*. DOI: [10.48550/arXiv.2004.00348](https://doi.org/10.48550/arXiv.2004.00348) (cited on pp. 2, 15, 108–109).
- 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 (cited on p. 1).
- 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)*. DOI: [10.1145/3510003.3510038](https://doi.org/10.1145/3510003.3510038) (cited on pp. 2, 111).
- Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu (2023). “Generative Type Inference for Python.” In: *Automated Software Engineering (ASE)*. DOI: [10.1109/ASE56229.2023.00031](https://doi.org/10.1109/ASE56229.2023.00031) (cited on pp. 2, 111).
- Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha (2021). “Solver-Based Gradual Type Migration.” In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: [10.1145/3485488](https://doi.org/10.1145/3485488) (cited on pp. 8, 107).
- Benjamin C. Pierce (2002). *Types and Programming Languages*. The MIT Press (cited on p. 5).
- 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)*. DOI: [10.1145/3368089.3409715](https://doi.org/10.1145/3368089.3409715) (cited on pp. 2, 47, 52, 110–111).
- Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer (2012). “The Ins and Outs of Gradual Type Inference.” In: *Principles of Programming Languages (POPL)*. DOI: [10.1145/2103656.2103714](https://doi.org/10.1145/2103656.2103714) (cited on pp. 2, 107).
- Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris (2015). “Safe & Efficient Gradual Typing for TypeScript.” In: *Principles of Programming Languages (POPL)*. DOI: [10.1145/2676726.2676971](https://doi.org/10.1145/2676726.2676971) (cited on p. 107).

- Veselin Raychev, Martin Vechev, and Andreas Krause (2015). “Predicting Program Properties from “Big Code”.” In: *Principles of Programming Languages (POPL)*. DOI: [10.1145/2676726.2677009](https://doi.org/10.1145/2676726.2677009) (cited on p. 109).
- Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek (2015). “Concrete Types for TypeScript.” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.4230/LIPIcs.ECOOP.2015.76](https://doi.org/10.4230/LIPIcs.ECOOP.2015.76) (cited on p. 107).
- Felix Rieseberg (2017). *TypeScript at Slack*. <https://slack.engineering/typescript-at-slack/>. Accessed: 2022-12-01 (cited on p. 1).
- Rollup contributors (2015). *Rollup*. <https://rollupjs.org/>. Accessed: 2024-02-01 (cited on p. 81).
- Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa (2014). *PEP 484 - Type Hints*. <https://peps.python.org/pep-0484/>. Accessed: 2023-04-01 (cited on p. 1).
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rabin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve (2023). *Code Llama: Open Foundation Models for Code*. DOI: [10.48550/arXiv.2308.12950](https://doi.org/10.48550/arXiv.2308.12950) (cited on p. 10).
- 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 (cited on p. 1).
- Claudiu Saftoiu (2010). “JSTrace: Run-time Type Discovery for JavaScript.” MA thesis. Brown University. URL: <https://cs.brown.edu/research/pubs/theses/ugrad/2010/saftoiu.pdf> (cited on p. 108).
- Lukas Seidel, Sedick David Baker Effendi, Xavier Pinho, Konrad Rieck, Brink van der Merwe, and Fabian Yamaguchi (2024). “Learning Type Inference for Enhanced Dataflow Analysis.” In: *European Symposium on Research in Computer Security (ESORICS)*. DOI: [10.1007/978-3-031-51482-1_10](https://doi.org/10.1007/978-3-031-51482-1_10) (cited on p. 110).
- 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.” In: *International Conference on Machine Learning (ICML)*. DOI: [10.48550/arXiv.2302.00093](https://doi.org/10.48550/arXiv.2302.00093) (cited on p. 47).
- Jeremy G. Siek and Walid Taha (2006). “Gradual Typing for Functional Languages.” In: *Scheme and Functional Programming Workshop*. URL: <http://schemeworkshop.org/2006/13-siek.pdf> (cited on p. 1).

- Jeremy G. Siek and Manish Vachharajani (2008). “Gradual Typing with Unification-Based Inference.” In: *Dynamic Languages Symposium (DLS)*. DOI: [10.1145/1408681.1408688](https://doi.org/10.1145/1408681.1408688) (cited on p. 107).
- Stack Overflow (2023). *Stack Overflow Developer Survey 2023*. <https://survey.stackoverflow.co/2023/#most-popular-technologies-language>. Accessed: 2023-08-01 (cited on p. 1).
- Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella (2022). “Guess What: Test Case Generation for JavaScript with Unsupervised Probabilistic Type Inference.” In: *Search-Based Software Engineering (SBSE)*. DOI: [10.1007/978-3-031-21251-2_5](https://doi.org/10.1007/978-3-031-21251-2_5) (cited on p. 109).
- 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)*. DOI: [10.48550/arXiv.2109.09115](https://doi.org/10.48550/arXiv.2109.09115) (cited on p. 47).
- Peter Thiemann (2005). “Towards a Type System for Analyzing JavaScript Programs.” In: *European Symposium on Programming (ESOP)*. DOI: [10.1007/978-3-540-31987-0_28](https://doi.org/10.1007/978-3-540-31987-0_28) (cited on p. 107).
- Sam Tobin-Hochstadt and Matthias Felleisen (2008). “The Design and Implementation of Typed Scheme.” In: *Principles of Programming Languages (POPL)*. DOI: [10.1145/1328438.1328486](https://doi.org/10.1145/1328438.1328486) (cited on pp. 1, 107).
- 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)*. DOI: [10.4230/LIPIcs.SNAPL.2017.17](https://doi.org/10.4230/LIPIcs.SNAPL.2017.17) (cited on p. 1).
- TypeScript contributors (2022). *TypeScript v4.9.3 – diagnosticMessages.json*. <https://github.com/Microsoft/TypeScript/blob/v4.9.3/src/compiler/diagnosticMessages.json>. Accessed: 2022-12-01 (cited on p. 35).
- 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)*. DOI: [10.4230/LIPIcs.ECOOP.2015.52](https://doi.org/10.4230/LIPIcs.ECOOP.2015.52) (cited on p. 107).
- Ashwin Prasad Shivarpatna Venkatesh, Samkuty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden (2024). *TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools*. DOI: [10.48550/arXiv.2312.16882](https://doi.org/10.48550/arXiv.2312.16882) (cited on p. 111).
- Sivani Voruganti, Kevin Jesse, and Premkumar Devanbu (2023). “FlexType: A Plug-and-Play Framework for Type Inference Models.” In: *Automated Software Engineering (ASE)*. DOI: [10.1145/3551349.3559527](https://doi.org/10.1145/3551349.3559527) (cited on p. 110).

- Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig (2020a). *LambdaNet*. <https://github.com/MrVPlusOne/LambdaNet/tree/ICLR20>. Accessed: 2022-06-01 (cited on p. 23).
- Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig (2020b). “LambdaNet: Probabilistic Type Inference using Graph Neural Networks.” In: *International Conference on Learning Representations (ICLR)*. DOI: [10.48550/arXiv.2005.02161](https://doi.org/10.48550/arXiv.2005.02161) (cited on pp. 2, 10, 15–17, 20, 52, 109).
- Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski (2017). “Mixed Messages: Measuring Conformance and Non-Interference in TypeScript.” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.4230/LIPIcs.ECOOP.2017.28](https://doi.org/10.4230/LIPIcs.ECOOP.2017.28) (cited on pp. 28, 109).
- 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)*. DOI: [10.1145/3520312.3534862](https://doi.org/10.1145/3520312.3534862) (cited on p. 2).
- 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)*. DOI: [10.1145/2950290.2950343](https://doi.org/10.1145/2950290.2950343) (cited on pp. 2, 110).
- Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, and Hari Sundaram (2024). *CodeScope: An Execution-based Multilingual Multitask Multidimensional Benchmark for Evaluating LLMs on Code Understanding and Generation*. DOI: <https://doi.org/10.48550/arXiv.2311.08588> (cited on p. 112).
- Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu (2023). “DLInfer: Deep Learning with Static Slicing for Python Type Inference.” In: *International Conference on Software Engineering (ICSE)*. DOI: [10.1109/ICSE48619.2023.00170](https://doi.org/10.1109/ICSE48619.2023.00170) (cited on p. 111).
- Fangke Ye, Jisheng Zhao, and Vivek Sarkar (2021). *Advanced Graph-Based Deep Learning for Probabilistic Type Inference*. DOI: [10.48550/arXiv.2009.05949](https://doi.org/10.48550/arXiv.2009.05949) (cited on p. 109).
- Fangke Ye, Jisheng Zhao, Jun Shirako, and Vivek Sarkar (2023). “Concrete Type Inference for Code Optimization using Machine Learning with SMT Solving.” In: *Proc. ACM Program. Lang.* 7.OOPSLA2. DOI: [10.1145/3622825](https://doi.org/10.1145/3622825) (cited on p. 111).
- Ming-Ho Yee and Arjun Guha (2023a). “Do Machine Learning Models Produce TypeScript Types That Type Check?” In: *European Conference on Object-Oriented Programming (ECOOP)*. DOI: [10.4230/LIPIcs.ECOOP.2023.37](https://doi.org/10.4230/LIPIcs.ECOOP.2023.37) (cited on pp. 2, 4, 15, 20, 47).
- Ming-Ho Yee and Arjun Guha (2023b). “Do Machine Learning Models Produce TypeScript Types That Type Check? (Artifact).” In: *Dagstuhl Artifacts Series* 9.2. DOI: [10.4230/DARTS.9.2.5](https://doi.org/10.4230/DARTS.9.2.5) (cited on p. 15).

Jake Zimmerman (2022). *Sorbet: Stripe's type checker for Ruby*. <https://stripe.com/blog/sorbet-stripes-type-checker-for-ruby>. Accessed: 2022-12-01 (cited on p. 1).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst’s seminal book on typography “*The Elements of Typographic Style*.” `classicthesis` is available for both \LaTeX and \LyX :

<https://bitbucket.org/amiede/classicthesis/>

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>

Final Version as of April 16, 2024 (`classicthesis v4.6 – commit 92oadfo`).