The Dissertation Committee for Jiayi Wei
certifies that this is the approved version of the following dissertation:

Combining Static Analysis with Deep Learning for Type Inference and Code Editing

Committee:

Isil Dillig, Supervisor

Greg Durrett, Co-Supervisor

Raymond J. Mooney

Miltiadis Allamanis

Combining Static Analysis with Deep Learning for Type Inference and Code Editing

by

Jiayi Wei

DISSERTATION

Presented to the Faculty of the Graduate School
of the University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

The University of Texas at Austin
August 2023

Acknowledgments

Reflecting on my past six years at UT Austin, I am filled with immense gratitude toward the incredible colleagues, friends, and family who have supported me. Their contributions have made this journey not just possible, but truly memorable.

Firstly, I owe a huge debt of gratitude to my advisor, Isil, who ignited my curiosity in programming language theories and saw my potential early on. Her guidance has been instrumental in my growth, and the lessons I've learned from her, particularly about communicating effectively and staying goal-oriented, will forever guide me forward. I deeply appreciate her understanding and support, even when I shifted my research directions multiple times. I am fortunate and privileged to have an advisor who consistently prioritized my interests above her own.

My co-advisor Greg also deserves a special mention. He guided me in the areas of deep learning and NLP and taught me how to approach challenging problems with simple steps—in fact, the type inference work in this thesis began as a final project in his NLP class. He patiently taught me how to make compelling presentations and effectively promote our work. And more importantly, his kindness and support, in both good times and bad, for both things big and small, has made this journey much more enjoyable and colorful.

I am also thankful to my committee members, Ray Mooney and Miltos Allamanis. Ray's insightful feedback has significantly improved this thesis, while Miltos, a pioneer in the field of AI4Code, has been a constant source of inspiration. I look forward to our paths crossing again in the future.

I was fortunate to work with many great collaborators when I first joined UT Austin. Yu Feng, whose optimism and hard work ethic had a deep influence on my early PhD years and played a crucial role in helping me successfully complete my initial projects. Jia Chen, who made an excellent duo with me that made my first projects much smoother. Osbert Bastani, for providing valuable experience and insights on reinforcement learning. Maruth Goyal, for his assistance on the type inference projects.

I also want to express my gratitude to my fellow members of the UTOPIA research group: To Yuepeng Wang, for inspiring me to apply to UT Austin and for his help on countless occasions. To Xinyu Wang, for his early guidance and wise advices. To Shankara Pailoor, with whom I had many fanscinating discussions during our ping pong games and movie nights. To Jocelyn Chen, for being a wonderful classmate, labmate, and friend. And to Kostas Ferles, whose humor often brightened my days.

Lastly, I thank my family for their unconditional love and support. Their encouragement has been a cornerstone of my progress. I express my profound gratitude to Yuege Xie. Meeting you has brought about the most beautiful transformation in my life and guided me on the path I'm on today.

Combining Static Analysis with Deep Learning for Type Inference and Code Editing

by

Jiayi Wei, Ph.D.
The University of Texas at Austin, 2023
Supervisor: Isil Dillig, Greg Durrett

For many programming tasks, state-of-the-art machine learning techniques treat programs as sequences of tokens and encode only local syntactic information. While this approach has achieved impressive results on tasks such as code autocompletion and program synthesis, many other tasks require analyzing programs at the project level. In this thesis, we propose techniques that combine lightweight static analysis and code transformations with machine learning to tackle two challenging problems from this category.

We first focus on probabilistic type inference, where the goal is to predict missing type annotations for programs written in a gradually typed language like JavaScript and Python. Global information is essential for this task as the model needs to consider how a function is used throughout the project and be aware of the new types defined elsewhere. Our first approach, LambdaNet, uses lightweight static analysis to generate a program abstraction called

a type dependency graph, which is then processed by a graph neural network to make type predictions. Our more recent work, TypeT5, models type inference as a code-infilling task and fine-tunes a pre-trained code-infilling model on type annotation labels. To best utilize the transformer model's limited receptive field, TypeT5 uses static analysis to construct a dynamic context for each code element. During inference time, we also propose a sequential decoding scheme to incorporate previously predicted types into the dynamic context, allowing information exchange between distant but related code elements.

We then focus on contextual code change prediction, where the goal is to predict how to edit a piece of code based on other relevant changes made elsewhere in the same project. We introduce Coeditor, a fine-tuned CodeT5 model specifically designed for code editing tasks. We again model this task as code infilling using a line-diff-based code change encoding scheme and employ static analysis to form large customized model contexts, ensuring appropriate information for prediction. Coeditor significantly outperforms the best code completion approach in a simplified single-round, single-edit task. In the proposed multi-round, multi-edit setting, Coeditor demonstrates substantial gains by iteratively conditioning on additional user edits. To encourage future research, we open-source our code, data, and model weights, and release a VSCode extension powered by our model for interactive usage.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, machine learning (ML) has played a significant role in various programming-related tasks such as code autocompletion (Chen et al., 2021, Raychev et al., 2014) and program synthesis (Li et al., 2022). These tasks often involve modeling programs as a sequence of tokens and using only local syntactical information. This approach has achieved impressive results. e.g., Copilot, the code completion system developed by GitHub, has helped millions of developers with their everyday tasks, and AlphaCode, the program synthesis system developed by DeepMind, has been able to match the programming skills of average coders in online programming competitions. Despite these successes, there are many other programming tasks that require a more holistic analysis of the entire program.

This thesis presents techniques that combine static analysis, code transformations, and machine learning to tackle two challenging problems that fall into this category, namely, probabilistic type inference and code auto-editing.

1.1 A Brief History of Probabilistic Type Inference

As gradual typing (Siek and Taha, 2007a) becomes increasingly popular in languages like TypeScript[1] and Python[2], there is a growing need to infer type annotations automatically: while type annotations benefit tasks like code completion and analysis, these annotations cannot be fully determined by compilers and are tedious to annotate by hand. Probabilistic type inference techniques aim to automatically predict these missing type annotations using ML-based techniques.

The first probabilistic type inference system we are aware of is JSNice (Raychev et al., 2015a), which uses probabilistic graphical models such as conditional random fields to infer JavaScript program properties like variable names and type annotations. Similarly, Xu et al. (2016) developed a system for inferring Python type annotations by modeling programs as a factor graph and applying the sum-product belief propagation algorithm. More recently, DeepTyper (Hellendoorn et al., 2018a) introduced the use of deep learning for type inference, using an LSTM to process program tokens and training the model on open-source TypeScript programs.

To address the challenge of predicting types from an open vocabulary and make use of more global information, we proposed LambdaNet (Wei et al.,

[1]TypeScript is a programming language that is a strict syntactical superset of JavaScript, with additional features such as optional static typing and object-oriented programming constructs.

[2]Type annotations were introduced in Python 3.5 as a way to provide hints about the expected type of a variable or expression to static type checkers and IDEs.

2020), a system that combines lightweight static analysis with a graph neural network and a pointer network layer to predict JavaScript types. We describe LambdaNet in detail in chapter 2. Typilus (Allamanis et al., 2020) is a similar system that uses graph neural networks to predict Python type annotations, but instead of using a pointer network, it uses nearest neighbor search in the type embedding space to predict user-defined types. Both LambdaNet and Typilus are able to predict user-defined types that were not seen during training and achieved new state-of-the-art performance on JavaScript and Python type inference tasks, respectively.

In addition to using source code, other forms of information have been utilized to improve probabilistic type inference. OptTyper (Pandi et al., 2020) explicitly models type constraints as a relaxed continuous optimization objective. TypeWriter (Pradel et al., 2020) and HiTyper (Peng et al., 2022) both combine rule-based type checking with inference-time search to prevent generating type errors. These approaches have shown to be effective in improving the accuracy and reliability of type prediction systems.

Meanwhile, with the advent of large-scale pretraining and the explosion of transformer architectures, in TypeBert (Jesse et al., 2021), the authors proposed the first type inference method based on a pre-trained transformer model and demonstrated significant improvement over LambdaNet in predicting common JavaScript types. Jesse et al. (2022) later improved TypeBert by using deep similarity learning (Allamanis et al., 2020, Mir et al., 2022) to better support user-defined types. These approaches showcase the potential

of using pre-training and modern transformer architectures for type inference.

Lastly, in our latest work, TypeT5 (Wei et al., 2023), we view type inference as a masked span infilling task and leverage a pre-trained seq2seq code completion model for the task. One particularly attractive feature of using such models is that, due to the use of subword tokenization (Gage, 1994, Schuster and Nakajima, 2012, Sennrich et al., 2016), they can generate arbitrary code expressions—including novel identifier names and AST structures—at test time, providing a simple and flexible solution to predict user-defined types and parametric types.[3] To optimize the use of the transformer model's limited receptive field, TypeT5 uses static analysis to construct a dynamic context for each code element. During inference time, we also propose a sequential decoding scheme to incorporate previously predicted types into the dynamic context, allowing information exchange between distant but related code elements. Our evaluation shows that TypeT5 outperforms prior approaches by a large margin and drastically improves the accuracy on rare and complex types. We provide a detailed description of TypeT5 in chapter 3.

1.2 From Type Inference to Code Auto-Editing

Code editing and refactoring are crucial aspects of software development, enabling programmers to continuously enhance and maintain their code.

[3]It is worth noting that none of the aforementioned approaches can handle the unbounded prediction space induced by parametric types, so complex parametric types have to be projected into simpler types as a preprocessing step.

However, there is a lack of tools available to assist with these tasks. Code completion and generation techniques may be helpful for writing new code, but they are not ideal for editing and refactoring existing code. These techniques cannot predict where or how to make changes and do not account for changes made by the programmer elsewhere in the project.

In this thesis, we address this gap by developing a technique for *code auto-editing*. We propose Coeditor, a Copilot-style tool that functions as follows: as the programmer makes changes to a function, the tool automatically suggests additional changes to the remaining code based on partial changes made to the function thus far and other relevant changes made elsewhere in the same project. This problem also requires a global view of the program, as the model must be aware of all relevant changes, such as a signature change to a used function, even if they are far away. The model must also understand what APIs are available and how to use them to produce correct code rewrites. We model this task as a masked span infilling task using a line-diff-based code change encoding scheme and employ lightweight static analysis to form large customized model contexts, ensuring appropriate information for prediction. Our Coeditor model, fine-tuned from CodeT5, significantly outperforms the best code completion approach in a simplified single-round, single-edit task, nearly doubling its exact-match accuracy, despite using a much smaller model. In a multi-round, multi-edit setting, we also observe substantial gains by iteratively prompting the model with additional user edits. We present Coeditor in detail in chapter 4.

Chapter 2

LambdaNet: Probabilistic Type Inference using Graph Neural Networks

2.1 Introduction

Dynamically typed languages like Python, Ruby, and Javascript have gained enormous popularity over the last decade, yet their lack of a static type system comes with certain disadvantages in terms of maintainability (Hanenberg et al., 2013), the ability to catch errors at compile time, and code completion support (Gao et al., 2017). *Gradual typing* can address these shortcomings: program variables have *optional* type annotations so that the type system can perform static type checking whenever possible (Chung et al., 2018, Siek and Taha, 2007b). Support for gradual typing now exists in many popular programming languages (Bierman et al., 2014, Vitousek et al., 2014), but due to their heavy use of dynamic language constructs and the absence of principal types (Ancona and Zucca, 2004), compilers cannot perform type inference using standard algorithms from the programming languages community (Bierman et al., 2014, Pierce and Turner, 2000, Traytel et al., 2011),

[0]An early version of this chapter appeared in Wei et al. (2020). Jiayi Wei is the first author of the paper, and with the help from other authors, he developed the research idea, wrote the code, performed the experiments and analysis, and wrote the paper.

and manually adding type annotations to existing codebases is a tedious and error-prone task. As a result, legacy programs in these languages do not reap all the benefits of gradual typing.

To reduce the human effort involved in transitioning from untyped to statically typed code, this work focuses on a learning-based approach to automatically inferring likely type annotations for untyped (or partially typed) codebases. Specifically, we target TypeScript, a gradually-typed variant of Javascript for which plenty of training data is available in terms of type-annotated programs. While there has been some prior work on inferring type annotations for TypeScript using machine learning (Hellendoorn et al., 2018a, Raychev et al., 2015a), prior work in this space has several shortcomings. First, inference is restricted to a finite dictionary of types that have been observed during training time—i.e., they cannot predict any user-defined data types. Second, even without considering user-defined types, the accuracy of these systems is relatively low, with the current state-of-the-art achieving 56.9% accuracy for primitive/library types (Hellendoorn et al., 2018a). Finally, these techniques can produce inconsistent results in that they may predict different types for different token-level occurrences of the same variable.

In this work, we propose a new probabilistic type inference algorithm for TypeScript to address these shortcomings using a graph neural network architecture (GNN) (Li et al., 2016, Mou et al., 2016, Veličković et al., 2018). Our method uses lightweight source code analysis to transform the program into a new representation called a *type dependency graph*, where nodes repre-

sent type variables and labeled hyperedges encode relationships between them. In addition to expressing logical constraints (e.g., subtyping relations) as in traditional type inference, a type dependency graph also incorporates contextual hints involving naming and variable usage.

Given such a type dependency graph, our approach uses a GNN to compute a vector embedding for each type variable and then performs type prediction using a pointer-network-like architecture (Vinyals et al., 2015). The graph neural network itself requires handling a variety of hyperedge types—some with variable numbers of arguments—for which we define appropriate graph propagation operators. Our prediction layer compares the vector embedding of a type variable with vector representations of candidate types, allowing us to flexibly handle user-defined types that have not been observed during training. Moreover, our model predicts consistent type assignments by construction because it makes variable-level rather than token-level predictions.

We implemented our new architecture as a tool called LambdaNet and evaluated its performance on real-world TypeScript projects from Github. When only predicting library types, LambdaNet has a top1 accuracy of 75.6%, achieving a significant improvement over DeepTyper (61.5%). In terms of overall accuracy (including user-defined types), LambdaNet achieves a top1 accuracy of around 64.2%, which is 55.2% (absolute) higher than the TypeScript compiler.

Contributions. This work makes the following contributions: (1) We pro-

```
class MyNetwork {
   name: string;  time: number;
   forward(x: Tensor, y: Tensor): Tensor {
      return x.concat(y) * 2;
   }
}
// more classes ...
function restore (network: MyNetwork): void {
   network.time = readNumber("time.txt");
   // more code...
}
```

Figure 2.1: A motivating example: Given an unannotated version of this TypeScript program, a traditional rule-based type inference algorithm cannot soundly deduce the true type annotations (shown in green).

pose a probabilistic type inference algorithm for TypeScript that uses deep learning to make predictions from the type dependency graph representation of the program. (2) We describe a technique for computing vector embeddings of type variables using GNNs and propose a pointer-network-like method to predict user-defined types. (3) We experimentally evaluate our approach on hundreds of real-world TypeScript projects and show that our method significantly improves upon prior work.

2.2 Motivating Example and Problem Setting

Figure 2.1 shows a (type-annotated) TypeScript program. Our goal in this work is to infer the types shown in the figure, given an *unannotated* version of this code. We now justify various aspects of our solution using this example.

Typing constraints. The use of certain functions/operators in Figure 2.1

imposes hard constraints on the types that can be assigned to program variables. For example, in the `forward` function, variables `x`, `y` must be assigned a type that supports a `concat` operation; hence, `x`, `y` could have types like `string`, `array`, or `Tensor`, but not, for example, `boolean`. This observation motivates us to incorporate typing constraints into our model.

Contextual hints. Typing constraints are not always sufficient for determining the intended type of a variable. For example, for variable `network` in function `restore`, the typing constraints require `network`'s type to be a class with a field called `time`, but there can be *many* classes that have such an attribute (e.g., `Date`). However, the similarity between the variable name `network` and the class name `MyNetwork` hints that `network` might have type `MyNetwork`. Based on this belief, we can further propagate the return type of the library function `readNumber` (assuming we know it is `number`) to infer that the type of the `time` field in `MyNetwork` is likely to be `number`.

Need for type dependency graph. There are many ways to view programs, e.g., as token sequences, abstract syntax trees, control flow graphs, etc. However, none of these representations is particularly helpful for inferring the most likely type annotations. Thus, our method uses static analysis to infer a set of *predicates* that are relevant to the type inference problem and represents these predicates using a program abstraction called the *type dependency graph*.

Handling user-defined types. As mentioned in Section 2.1, prior techniques can only predict types seen during training. However, the code from Figure 2.1 defines its own class called `MyNetwork` and later uses a variables

of type `MyNetwork` in the `restore` method. A successful model for this task therefore must dynamically make inferences about user-defined types based on their definitions.

2.2.1 Problem Setting

Our goal is to train a type inference model that can take as input an entirely (or partially) unannotated TypeScript project g and output a probability distribution of types for each missing annotation. The prediction space is $\mathcal{Y}(g) = \mathcal{Y}_{\text{lib}} \cup \mathcal{Y}_{\text{user}}(g)$, where $\mathcal{Y}_{\text{user}}(g)$ is the set of all user-defined types (classes/interfaces) declared within g, and $\mathcal{Y}_{\text{lib}}$ is a fixed set of commonly-used library types.

Following prior work in this space (Hellendoorn et al., 2018a, Raychev et al., 2015a, Xu et al., 2016), we limit the scope of our prediction to non-polymorphic and non-function types. That is, we do not distinguish between types such as `List<T>`, `List<number>`, `List<string>` etc., and consider them all to be of type `List`. Similarly, we also collapse function types like `number` $\rightarrow$ `string` and `string` $\rightarrow$ `string` into a single type called `Function`. We leave the extension of predicting structured types as future work.

2.3 Type Dependency Graph

A type dependency graph $\mathcal{G} = (N, E)$ is a hypergraph where nodes N represent type variables and labeled hyperedges E encode relationships between them. We extract the type dependency graph of a given TypeScript

```
1       var c1: τ8 = class MyNetwork {
2           name: τ1;  time: τ2;
3           var m1: τ9 = function forward(x: τ3, y: τ4):τ5 {
4               var v1: τ10 = x.concat;  var v2: τ11 = v1(y);
5               var v3: τ12 = v2.TIMES_OP; var v4: τ13 = v3(NUMBER);
6               return v4;
7           }
8       }  // more classes...
9       var f1:τ14 = function restore (network: τ6): τ7 {
10        var v3: τ15 = network.time;
11        var v4: τ16 = readNumber(STRING);
12        network.time = v4;  // more code...
13      }
```

Figure 2.2: An intermediate representation of the (unannotated version) program from Figure 2.1. The τ_i represent type variables, among which τ_8–τ_{16} are newly introduced for intermediate expressions.

Table 2.1: Different types of hyperedges used in a type dependency graph.

Type	**Edge**	**Description**
	Logical	
FIXED	$\text{Bool}(\alpha)$	α is used as boolean
FIXED	$\text{Subtype}(\alpha, \beta)$	α is a subtype of β
FIXED	$\text{Assign}(\alpha, \beta)^\dagger$	β is assigned to α
NARY	$\text{Function}(\alpha, \beta_1, \ldots, \beta_k, \beta^*)$	$\alpha = (\beta_1, \ldots, \beta_k) \to \beta^*$
NARY	$\text{Call}(\alpha, \beta^*, \beta_1, \ldots, \beta_k)$	$\alpha = \beta^*(\beta_1, \ldots, \beta_k)$
NARY	$\text{Object}_{l_1,\ldots,l_k}(\alpha, \beta_1, \ldots, \beta_k)$	$\alpha = \{l_1 : \beta_1, \ldots, l_k : \beta_k\}$
FIXED	$\text{Access}_l(\alpha, \beta)$	$\alpha = \beta.l$
	Contextual	
FIXED	$\text{Name}_l(\alpha)$	α has name l
FIXED	$\text{NameSimilar}(\alpha, \beta)$	α, β have similar names
NPAIRS	$\text{Usage}_l((\alpha^*, \beta^*), (\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$	usages involving name l

† Although assignment is a special case of a subtype constraint, we differentiate them because these edges appear in different contexts and having uncoupled parameters for these two edge types is beneficial.

program by performing static analysis on an *intermediate representation* of its source code, which allows us to associate a unique variable with each program sub-expression. As an illustration, Figure 2.2 shows the intermediate

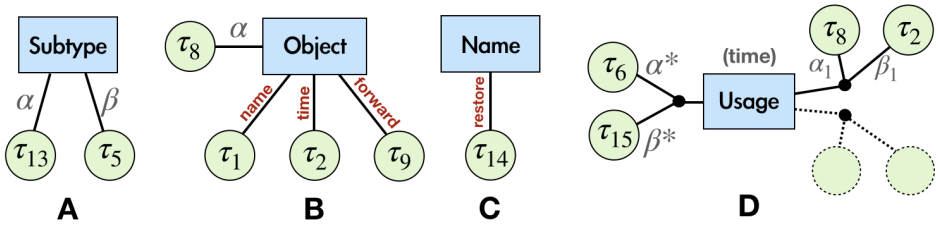

Figure 2.3: Example hyperedges for Figure 2.2. Edge labels in gray (resp. red) are positional arguments (resp. identifiers). **(A)** The return statement at line 6 induces a subtype relationship between τ_{13} and τ_5. **(B)** `MyNetwork` τ_8 declares attributes `name` τ_1 and `time` τ_2 and method `forward` τ_9. **(C)** τ_{14} is associated with a variable whose named is `restore`. **(D)** Usage hyperedge for line 10 connects τ_6 and τ_{15} to all classes with a `time` attribute.

representation of the code from Figure 2.1.

Intuitively, a type dependency graph encodes properties of type variables as well as relationships between them. Each hyperedge corresponds to one of the predicates shown in Table 2.1. We partition our predicates (i.e., hyperedges) into two classes, namely *Logical* and *Contextual*, where the former category can be viewed as imposing hard constraints on type variables and the latter category encodes useful hints extracted from names of variables, functions, and classes.

Figure 2.3 shows some of the hyperedges in the type dependency graph $\mathcal{G}$ extracted from the intermediate representation in Figure 2.2. As shown in Figure 2.3(A), our analysis extracts a predicate $\text{Subtype}(\tau_{13}, \tau_5)$ from this code because the type variable associated with the returned expression `v4` must be a subtype of the enclosing function's return type. Similarly, as shown in Figure 2.3(B), our analysis extracts a predicate $\text{Object}_{\text{name,time,forward}}(\tau_8, \tau_1, \tau_2, \tau_9)$

because τ_8 is an object type whose `name`, `time`, and `forward` members are associated with type variables τ_1, τ_2, τ_9, respectively.

In contrast to the Subtype and Object predicates that impose hard constraints on type variables, the next two hyperedges shown in Figure 2.3 encode contextual clues obtained from variable names. Figure 2.3(C) indicates that type variable τ_{14} is associated with an expression named `restore`. While this kind of naming information is invisible to TypeScript's structural type system (Bierman et al., 2014), it serves as a useful input feature for our GNN architecture described in Section 2.4.

In addition to storing the unique variable name associated with each type variable, the type dependency graph also encodes similarity between variable and class names. The names of many program variables mimic their types: for example, instances of a class called `MyNetwork` might often be called `network` or `network1`. To capture this correspondence, our type dependency graph also contains a hyperedge called NameSimilar that connects type variables α and β if their corresponding tokenized names have a non-empty intersection.[1]

As shown in Table 2.1, there is a final type of hyperedge called Usage that facilitates type inference of object types. In particular, if there is an object access `y = x.l`, we extract the predicate $\text{Usage}_l((\tau_x, \tau_y), (\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$

[1]During tokenization, we split identifier names into tokens based on underscores and camel case naming. More complex schemes are possible, but we found this simple method to be effective.

to connect `x` and `y`'s type variables with *all* classes α_i that contain an attribute/method β_i whose name is `l`. Figure 2.3 shows a Usage hyperedge extracted from the code in Figure 2.2. As we will see in the next section, our GNN architecture utilizes a special attention mechanism to pass information along these usage edges.

2.4 Neural Architecture

Our neural architecture for making type predictions consists of two main parts. First, a graph neural network passes information along the type dependency graph to produce a vector-valued embedding for each type variable based on its neighbors. Second, a pointer network compares each variable's type embedding to the embedding vectors of candidate types (both computed from the previous phase) to place a distribution over possible type assignments.

Given a type dependency graph $\mathcal{G} = (N, E)$, we first to compute a vector embedding $\mathbf{v}_n$ for each $n \in N$ such that these vectors implicitly encode type information. Because our program abstraction is a graph, a natural choice is to use a graph neural network architecture. From a high level, this architecture takes in initial vectors $\mathbf{v}_n^0$ for each node n, performs K rounds of message-passing in the graph neural network, and returns the final representation for each type variable.

In more detail, let $\mathbf{v}_n^t$ denote the vector representation of node n at the tth step, where each round consists of a *message passing* and an *aggregation* step. The message passing step computes a vector-valued update to send to

the jth argument of each hyper-edge $e \in E$ connecting nodes $p_1, \ldots, p_a$. Then, once all the messages have been computed, the *aggregation* step computes a new embedding $\mathbf{v}_n^t$ for each n by combining all messages sent to n:

$$\mathbf{m}_{e,p_j}^t = \text{Msg}_{e,j}(\mathbf{v}_{p_1}^{t-1}, \ldots, \mathbf{v}_{p_a}^{t-1}) \qquad \mathbf{v}_n^t = \text{Aggr}(\mathbf{v}_n^{t-1}, \{\mathbf{m}_{e,n}^t | e \in \mathcal{N}(n)\})$$

Here, $\mathcal{N}$ is the neighborhood function, and Msg_e denotes a particular neural operation that depends on the type of the edge (FIXED, NARY, or NPAIRS), which we will describe later.

Initialization. In our GNN, nodes correspond to type variables and each type variable is associated either with a program variable or a constant. We refer to nodes representing constants (resp. variables) as *constant (resp. variable) nodes*, and our initialization procedure works differently depending on whether or not n is a constant node. Since the types of each constant are known, we set the initial embedding for each constant node of type τ (e.g., `string`) to be a trainable vector $\mathbf{c}_\tau$ and do not update it during GNN iterations (i.e., $\forall t, \mathbf{v}_n^t = \mathbf{c}_\tau$). On the other hand, if n is a variable node, then we have no information about its type during initialization; hence, we initialize all variable nodes using a generic trainable initial vector (i.e., they are initialized to the *same* vector but updated to different values during GNN iterations).

Message passing. Our Msg operator depends on the category of edge it corresponds to (see Table 2.1); however, weights are shared between all instances of the same hyperedge type. In what follows, we describe the neural layer that is used to compute messages for each type of hyperedge:

- FIXED: Since these edges correspond to fixed arity predicates (and the position of each argument matters), we compute the message of the jth argument by first concatenating the embedding vector of all arguments and then feed the result vector to a 2-layer MLP for the jth argument. In addition, since hyperedges of type *Access* have an identifier, we also embed the identifier as a vector and treat it as an extra argument. (We describe the details of identifier embedding later in this section.)

- NARY: Since NARY edges connect a variable number of nodes, we need an architecture that can deal with this challenge. In our current implementation of LAMBDANET, we use a simple architecture that is amenable to batching. Specifically, given an NARY edge $E_{l_1,\ldots,l_k}(\alpha, \beta_1, \ldots, \beta_k)$ (for *Function* and *Call*, the labels l_j are argument positions), the set of messages for α is computed as $\{\text{MLP}_\alpha(\mathbf{v}_{l_j} \parallel \mathbf{v}_{\beta_j}) \mid j = 1 \ldots k\}$, and the message for each β_j is computed as $\text{MLP}_\beta(\mathbf{v}_{l_j} \parallel \mathbf{v}_\alpha)$. Observe that we compute k different messages for α, and the message for each β_j only depends on the vector embedding of α and its position j, but not the vector embeddings of other β_j's.[2]

- NPAIRS: This is a special category associated with the usage relation $\text{Usage}_l((\alpha^*, \beta^*), (\alpha_1, \beta_1), \ldots, (\alpha_k, \beta_k))$. Recall that this kind of edge arises from expressions of the form $b = a.l$ and is used to connect a and b's type

[2]In our current implementation, this is reducible to multiple FIXED edges. However, NARY edges could generally use more complex pooling over their arguments to send more sophisticated messages.

variables with all classes α_i that contain an attribute/method β_i with label l. Intuitively, if a's type embedding is very similar to a type C, then b's type will likely be the same as $C.l$'s type. Following this reasoning, we use dot-product based attention to compute the messages for α^* and β^*. Specifically, we use α^* and α_j's as attention keys and β_j's as attention values to compute the message for β^* (and switch the key-value roles to compute the message for α^*):

$$\mathbf{m}^t_{e,\beta^*} = \sum_j w_j \mathbf{v}^{t-1}_{\beta_j} \qquad \mathbf{w} = \text{softmax}(\mathbf{a}) \qquad a_j = \mathbf{v}_{\alpha_j} \cdot \mathbf{v}_{\alpha^*}$$

Aggregation. Recall that the aggregation step combines all messages sent to node n to compute the new embedding v^t_n. To achieve this goal, we use a variant of the attention-based aggregation operator proposed in graph attention networks (Veličković et al., 2018).

$$v^t_n = \text{Aggr}(v^{t-1}_n, \{m^t_{e,n} | e \in \mathcal{N}(n)\}) = v^{t-1}_n + \sum_{e \in \mathcal{N}(n)} w_e \mathbf{M}_1 m^t_{e,n} \tag{2.1}$$

where w_e is the attention weight for the message coming from edge e. Specifically, the weights w_e are computed as $\text{softmax}(\mathbf{a})$, where $a_e = \text{LeakyReLu}(v^{t-1}_n \cdot \mathbf{M}_2 m^t_{e,n})$, and $\mathbf{M}_1$ and $\mathbf{M}_2$ are trainable matrices. Similar to the original GAT architecture, we set the slope of the LeakyReLu to be 0.2, but we use dot-product to compute the attention weights instead of a linear model.

Identifier embedding. Like in Allamanis et al. (2017), we break variable names into word tokens according to camel case and underscore rules and assign a trainable vector for all word tokens that appear more than once in

the training set. For all other tokens, unlike Allamanis et al. (2017), which maps them all into one single `<Unknown>` token, we randomly mapped them into one of the `<Unknown-i>` tokens, where i ranges from 0 to 50 in our current implementation. This mapping is randomly constructed every time we run the GNN and hence helps our neural networks to distinguish different tokens even if they are rare tokens. We train these identifier embeddings end-to-end along with the rest of our architecture.

Prediction Layer. For each type variable n and each candidate type $c \in \mathcal{Y}(g)$, we use a MLP to compute a compatibility score $s_{n,c} = \text{MLP}(\mathbf{v}_n, \mathbf{u}_c)$, where $\mathbf{u}_c$ is the embedding vector for c. If $c \in \mathcal{Y}_{\text{lib}}$, $\mathbf{v}_c$ is a trainable vector for each library type c; if $c \in \mathcal{Y}_{\text{user}}(g)$, then it corresponds to a node n_c in the type dependency graph of g, so we just use the embedding vector for n_c and set $\mathbf{u}_c = \mathbf{v}_{n_c}$. Formally, this approach looks like a pointer network (Vinyals et al., 2015), where we use the embeddings computed during the forward pass to predict "pointers" to those types.

Given these compatibility scores, we apply a softmax layer to turn them into a probability distribution. i.e., $P_n(c|g) = \exp(s_{n,c}) / \sum_{c'} \exp(s_{n,c'})$. During test time, we max over the probabilities to compute the most likely (or top-N) type assignments.

2.5 Evaluation

In this section, we describe the results of our experimental evaluation, which is designed to answer the following questions: (1) How does our approach

compare to previous work? (2) How well can our model predict user-defined types? (3) How useful is each of our model's components?

Dataset. Similar to Hellendoorn et al. (2018a), we train and evaluate our model on popular open-source TypeScript projects taken from Github. Specifically, we collect 300 popular TypeScript projects from Github that contain between 500 to $10,000$ lines of code and where at least 10% of type annotations are user-defined types. Note that each project typically contains hundreds to thousands of type variables to predict, and these projects in total contain about 1.2 million lines of TypeScript code. Among these 300 projects, we use 60 for testing, 40 for validation, and the remainder for training.

Code Duplication. We ran jscpd[3] on our entire data set and found that only 2.7% of the code is duplicated. Furthermore, most of these duplicates are intra-project. Thus, we believe that code duplication is not a severe problem in our dataset.

Preprocessing. Because some of the projects in our benchmark suite are only sparsely type annotated, we augment our labeled training data by using the forward type inference functionality provided by the TypeScript compiler.[4]

[3] A popular code duplication detection tool, available at `https://github.com/kucherenko/jscpd`.

[4] Like in many modern programming languages with forward type inference (e.g., Scala, C#, Swift), a TypeScript programmer does not need to annotate every definition in order to fully specify the types of a program. Instead, they only need to annotate some "key places" (e.g., function parameters and return types, class members) and let the forward inference algorithm to figure out the rest of the types. Therefore, in our training set, we can keep the user annotations on these key places and run the TS compiler to recover these implicitly specified types as additional labels.

The compiler cannot infer the type of every variable and leaves many labeled as `any` during failed inference; thus, we exclude `any` labels in our data set. Furthermore, at test time, we evaluate our technique only on annotations that are manually added by developers. This is the same methodology used by Hellendoorn et al. (2018a), and, since developers often add annotations where code is most unclear, this constitutes a challenging setting for type prediction.

Prediction Space. As mentioned in Section 2.2.1, our approach takes an entire TypeScript project g as its input, and the corresponding type prediction space is $\mathcal{Y}(g) = \mathcal{Y}_{\text{lib}} \cup \mathcal{Y}_{\text{user}}(g)$. In our experiments, we set $\mathcal{Y}_{\text{user}}(g)$ to be all classes/interfaces defined in g (except when comparing with DeepTyper, where we set $\mathcal{Y}_{\text{user}}(g)$ to be empty), and for $\mathcal{Y}_{\text{lib}}$, we select the top-100 most common types in our training set. Note that this covers 98% (resp. 97.5%) of the non-`any` annotations for the training (resp. test) set.

Hyperparameters We selected hyperparameters by tuning on a validation set as we were developing our model. We use 32-dimensional type embedding vectors, and all MLP transformations in our model use one hidden layer of 32 units, except the MLP for computing scores in the prediction layer, which uses three hidden layers of sizes 32,16, and 8 (and size 1 for output). GNN message-passing layers from different time steps have independent weights.

We train our model using Adam (Kingma and Ba, 2014) with default parameters ($\alpha = 0.9$, $\beta = 0.999$) and set the learning rate to be 10^{-3} initially but linearly decrease it to 10^{-4} until the 30th epoch. We use a weight decay of 10^{-4} for regularization and stop the training once the loss on validation set

starts to increase (which usually happens around 30 epochs). We use the type annotations from a single project as a minibatch and limit the maximal batch size (via downsampling) to be the median of our training set to prevent any single project from having too much influence.

Implementation Details. We implemented LambdaNet in Scala, building on top of the Java high-performance Tensor library *Nd4j*(K.K.), and used a custom automatic differentiation library to implement our GNN. Our GNN implementation does not use an adjacency matrix to represent GNN layers; instead, we build the hyperedge connections directly from our type dependency graph and perform batching when computing the messages for all hyperedges of the same type.

Code Repository. We have made our code publicly available on Github.[5]

2.5.1 Comparison with DeepTyper

In this experiment, we compare LambdaNet's performance with DeepTyper (Hellendoorn et al., 2018a), which treats programs as sequences of tokens and uses a bidirectional RNN to make type predictions. Since DeepTyper can only predict types from a fixed vocabulary, we fix both LambdaNet and DeepTyper's prediction space to $\mathcal{Y}_{\text{lib}}$ and measure their corresponding top-1 accuracy.

The original DeepTyper model makes predictions for each variable oc-

[5] `https://github.com/MrVPlusOne/LambdaNet`.

Table 2.2: Comparing LambdaNet with DeepTyper on library types.

Model	**Top1 Accuracy (%)**	
	Declaration	*Occurrence*
DeepTyper	61.5	67.4
LambdaNet$_{\text{lib}}$ (K=6)	75.6	77.0

currence rather than declaration. In order to conduct a meaningful comparison between DeepTyper and LambdaNet, we implemented a variant of DeepTyper that makes a single prediction for each variable (by averaging over the RNN internal states of all occurrences of the same variable before making the prediction). Moreover, for a fair comparison, we made sure both DeepTyper and LambdaNet are using the same improved naming feature that splits words into tokens.

Our main results are summarized in Table 2.2, where the Declaration (resp. Occurrence) column shows accuracy per variable declaration (resp. token-level occurrence). Note that we obtain occurrence-level accuracy from declaration-level accuracy by weighting each variable by its number of occurrences.

As we can see from the table, LambdaNet achieves significantly better results compared to DeepTyper. In particular, LambdaNet outperforms DeepTyper by 14.1% (absolute) for declaration-level accuracy and by 9.6% for occurrence-level accuracy.

Note that the accuracy we report for DeepTyper (67.4%) is not directly comparable to the original accuracy reported in Hellendoorn et al. (2018a)

Table 2.3: Accuracy when predicting all types.

Model	**Top1 Accuracy (%)**			**Top5 Accuracy (%)**		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
TS Compiler	2.66	14.39	8.98	-	-	-
SimilarName	24.1	0.78	15.7	42.5	3.19	28.4
LambdaNet (K=6)	53.4	66.9	64.2	77.7	86.2	84.5

(56.9%) for the following reasons. While we perform static analysis and have a strict distinction of library vs. user-defined types and only evaluate both tools on library type annotations in this experiment, their implementation treat types as tokens and does not have this distinctions. Hence, their model also considers a much larger prediction space consisting of many user-defined types—most of which are never used outside of the project in which they are defined—and is also evaluated on a different set of annotations than ours.

2.5.2 Predicting User-Defined Types

As mentioned earlier, our approach differs from prior work in that it is capable of predicting user-defined types; thus, in our second experiment, we extend LambdaNet's prediction space to also include user-defined types. However, since such types are not in the prediction space of prior work (Hellendoorn et al., 2018a), we implemented two simpler baselines that can be used to calibrate our model's performance. Our first baseline is the type inference performed by the TypeScript compiler, which is sound but incomplete (i.e., if it infers a type, it is guaranteed to be correct, but it infers type `any` for most

variables).[6] Our second baseline, called SimilarName, is inspired by the similarity between variable names and their corresponding types; it predicts the type of each variable v to be the type whose name shares the most number of common word tokens with v.

The results of this experiment are shown in Table 2.3, which shows the top-1 and top-5 accuracy for both user-defined and library types individually as well as overall accuracy. In terms of overall prediction accuracy, LambdaNet achieves 64.2% for top-1 and 84.5% for top-5, significantly outperforming both baselines. Our results suggest that our fusion of logical and contextual information to predict types is far more effective than rule-based incorporation of these in isolation.

2.5.3 Ablation Study

Table 2.4 shows the results of an ablation study in which (a) we vary the number of message-passing iterations (left) and (b) disable various features of our architecture design (right). As we can see from the left table, accuracy continues to improve as we increase the number of message passing iterations as high as 6; this gain indicates that our network learns to perform inference over long distances. The right table shows the impact of several of our design choices on the overall result. For example, if we do not use *Contextual* edges (resp. *Logical* edges), overall accuracy drops by 14.5% (resp. 25.8%). These

[6]For inferring types from the TypeScript compiler, we use the code provided by Hellendoorn et al. (2018a). We found this method had a slightly lower accuracy than reported in their work.

Table 2.4: Performance of different GNN iterations (left) and ablations (right).

K	**Top1 Accuracy (%)**		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
6	53.4	66.9	64.2
4	48.4	65.5	62.0
2	47.3	61.7	58.8
1	16.8	48.2	41.9
0	0.0	17.0	13.6

Ablation (K = 4)	**Top1 Accuracy (%)**		
	$\mathcal{Y}_{\text{user}}$	$\mathcal{Y}_{\text{lib}}$	*Overall*
LambdaNet	48.4	65.5	62.0
No NPair Attention	44.1	57.6	54.9
No *Contextual*	27.2	52.6	47.5
No *Logical**	24.7	39.2	36.2
Simple Aggregation	40.2	66.9	61.5

* Training was unstable and experienced gradient explosion.

drops indicate that both kinds of predicates are crucial for achieving good accuracy. We also see that the attention layer for NPair makes a significant difference for both library and user-defined types. Finally, Simple Aggregation is a variant of LambdaNet that uses a simpler aggregation operation which replaces the attention-based weighed sum in Eq 2.1 with a simple average. As indicated by the last row of Table 2.4 (right), attention-based aggregation makes a substantial difference for user-defined types.

2.5.4 Comparison with JSNice

Since JSNice (Raychev et al., 2015a) cannot properly handle class definitions and user-defined types, for a meaningful comparison, we compared both tools' performance on top-level functions randomly sampled from our test set. We filtered out functions whose parameters are not library types and manually ensured that all all the dependency definitions are also included. In this way, we constructed a small benchmark suite consisting of 41 functions.

Among the 107 function parameter and return type annotations, LambdaNet correctly predicted 77 of them, while JSNice only got 48 of them right. These results suggest that LambdaNet outperforms JSNice, even when evaluated only on the places where JSNice is applicable.

2.6 Related Work

Type Inference using Statistical Methods. There are several previous works on predicting likely type annotations for dynamically typed languages: Raychev et al. (2015a) and Xu et al. (2016) use structured inference models for Javascript and Python, but their approaches do not take advantage of deep learning and are limited to a very restricted prediction space. Hellendoorn et al. (2018a) and Jangda and Anand (2019) model programs as sequences and AST trees and apply deep learning models (RRNs and Tree-RNNs) for TypeScript and Python programs. Malik et al. (2019) make use of a different source of information and take documentation strings as part of their input. However, all these previous works are limited to predicting types from a fixed vocabulary.

Graph Embedding of Programs. Allamanis et al. (2017) are the first to use GNNs to obtain deep embedding of programs, but they focus on predicting variable names and misuses for C♯ and rely on static type information to construct the program graph. Wang et al. (2017) use GNNs to encode mathematical formulas for premise selection in automated theorem proving. The way we encode types has some similarity to how they encode quantified

formulas, but while their focus is on higher-order formulas, our problem requires encoding object types. Veličković et al. (2018) are the first to use an attention mechanism in GNNs. While they use attention to compute node embeddings from messages, we use attention to compute certain messages from node embeddings.

Predicting from an Open Vocabulary. Predicting unseen labels at test time poses a challenge for traditional machine learning methods. For computer vision applications, solutions might involve looking at object attributes (Farhadi et al., 2017) or label similarity Wang et al. (2018); for natural language, similar techniques are applied to generalize across semantic properties of utterances (Dauphin et al., 2013), entities (Eshel et al., 2017), or labels (Ren et al., 2016). Formally, most of these approaches compare an embedding of an input to some embedding of the label; what makes our approach a pointer network (Vinyals et al., 2015) is that our type encodings are derived during the forward pass on the input, similar to unknown words for machine translation (Gulcehre et al., 2016).

Chapter 3

TypeT5: Seq2seq Type Inference using Static Analysis

As seen in the previous chapter, LambdaNet addresses the challenges of type inference in TypeScript by leveraging graph neural networks and a novel program representation called type dependency graph. While this approach provides significant improvements over prior work, it still exhibits some critical limitations, such as the inability to predict parametric types and the lack of large-scale pre-training. In this chapter, we aim to overcome these limitations by presenting a new approach, TypeT5, which integrates static analysis techniques with the powerful seq2seq transformer model, CodeT5, to predict type annotations in Python code.

3.1 Introduction

In languages like Python and JavaScript, the lack of a static type system makes it harder to maintain and analyze codebases. To address this issue, *gradual typing* (Siek and Taha, 2007a) was proposed to allow type annotations

[0]An early version of this chapter appeared in Wei et al. (2023). Jiayi Wei is the first author of the paper, and with the help from other authors, he developed the research idea, wrote the code, performed the experiments and analysis, and wrote the paper.

to be incrementally added to untyped codebases, thereby marrying the benefits of static typing with the convenience of easy prototyping. As a result, many mainstream programming languages, including Python and JavaScript, have already adopted this idea, and researchers have also developed learning-based techniques to predict missing type annotations (Allamanis et al., 2020, Hellendoorn et al., 2018b, Jesse et al., 2021, 2022, Mir et al., 2022, Pandi et al., 2020, Peng et al., 2022, Pradel et al., 2020, Raychev et al., 2015b, Wei et al., 2020).

Meanwhile, with the advent of large-scale pretraining and the explosion of transformer architectures, seq2seq models have proven to be very effective for programming tasks like code comments generation (Panthaplackel et al., 2020b), completion (Ahmad et al., 2021, Wang et al., 2021), and synthesis (Li et al., 2022). One particularly attractive feature of such models is that, due to the use of subword tokenization (Gage, 1994, Schuster and Nakajima, 2012, Sennrich et al., 2016), they can generate arbitrary code expressions—including novel identifier names and AST structures—at test time. However, unlike code completion tasks that can often work well with just the surrounding code as context, effective type inference generally requires non-local information, including code fragments that may belong to an entirely different file. For instance, consider a function f that passes a generically named parameter x directly into another function g. It can be hard to figure out the type of x by just looking at f's body. When programmers find themselves in such a situation, they often inspect the callers and callees of f, sometimes even

transitively, in order to figure out the intended type of x. Thus, in many cases, looking at the immediate context of a given variable may be insufficient for accurately predicting its type.

Our approach, TypeT5, solves this challenge by using static analysis to identify which parts of the codebase are useful for each prediction. In particular, we construct a so-called *usage graph*, where nodes correspond to code elements (i.e., functions or variables whose types we want to predict) and edges denote a potential user-usee relation between them. Given such a graph, we then encode the users and usees of a given code element in a form that resembles normal code and feeds them as additional contexts to the transformer model. To take full advantage of the seq2seq paradigm, we also propose an iterative decoding scheme that pass in previous type predictions using the contexts, allowing information to be propagated between distant code elements across the entire codebase.

We have implemented TypeT5 on top of the popular CodeT5 model and use it to synthesize type annotations for untyped Python code. Our evaluation compares TypeT5 with three state-of-the-art type inference tools (Allamanis et al., 2020, Mir et al., 2022, Peng et al., 2022) and a CodeT5 baseline that does not leverage static analysis. The results show that TypeT5 outperforms all baselines by a large margin, while drastically improving the accuracy on rare and complex types. Our ablation studies confirm the benefits of the various modifications we made to the CodeT5 baseline, while an additional type checking experiment shows that the proposed iterative decoding scheme

also improves the coherence of the produced type assignments, resulting in fewer type constraint violations. Finally, we explore an alternative use case of our model, where the user interactively inspects the model's predictions and makes necessary corrections. The result demonstrates the usefulness of our approach as a developer tool to annotate entirely untyped projects—on average, the user only needs to to correct one in every five model predictions.

To summarize, this work makes the following contributions:

- We apply CodeT5 to infer Python type annotations and show significant improvement over prior approaches. To our knowledge, this is the first ML-based technique capable of predicting both parametric and user-defined types.
- We improve the vanilla CodeT5 model by applying static analysis techniques to help the model reason about information beyond local contexts, further boosting its performance.
- We propose an iterative decoding scheme that particularly helps with *coherence*, as measured by the number of type errors reported by the type checker. We additionally propose the novel setting that combines the seq2seq decoding scheme with user intervention.

3.2 Overview

In this section, we motivate the design of TypeT5 using the example shown in Figure 3.1. This example features a method `predict` and two

functions `eval_on_dataset` and `chuck_srcs`, each of which is implemented in a different file. Given an untyped version of this code, our goal is to automatically infer the type annotations (highlighted in green). This example is challenging for existing type inference techniques due to the heavy use of user-defined types (such as `ChunkedDataset`, `PythonType`, and `ModelWrapper`) and complex parametric type like `dict[int,list[PythonType]]`.

Type inference as code infilling In this work, we advocate a new approach that views type inference as an instance of code infilling. Because type annotations can be viewed as missing code fragments, we fine-tune a state-of-the-art code infilling model, namely CodeT5, as our starting point. Since CodeT5 produces sequences of subword tokens using Byte Pair Encod-

model.py

```python
class ModelWrapper:
    ... # other methods and attributes omitted

    def predict(
        self,
        data: ChunkedDataset,
        n_seqs: Optional[int] = None,
    ) -> dict[int, list[PythonType]]:
        pred_types = dict()
        for batch in data.data:
            batch["input_ids"] = batch["input_ids"].to(device)
            preds, _ = self.predict_on_batch(batch, n_seqs)
            for i, c_id in enumerate(batch["chunk_id"]):
                if n_seqs is None:
                    pred_types[c_id] = preds[i]
                else:
                    span = i * n_seqs : (i + 1) * n_seqs
                    pred_types[c_id] = preds[span]
        return pred_types
```

eval.py

```python
def eval_on_dataset(
    model: ModelWrapper,
    data: TokenizedSrcSet,
    window_size: Optional[int] = None,
) -> dict[int, list[PythonType]]:
    window = copy(model.DefaultWindow)
    if scale_window is not None:
        window.left_tokens = window_size
        window.right_tokens = window_size

    chunks = chunk_srcs(data, window)
    return model.predict(chunks, n_seqs=None)
```

data.py

```python
def chunk_srcs(
    data: TokenizedSrcSet,
    window: WindowArgs
) -> ChunkedDataset:
    ...  # implementation omitted
    return ChunkedDataset(...)
```

Figure 3.1: Simplified code snippets taken from our own codebase. The `eval_on_dataset` function first calls the `chunk_srcs` function to convert the given textual data into equally sized chunks, and it then feed them into the `ModelWrapper.predict` method.

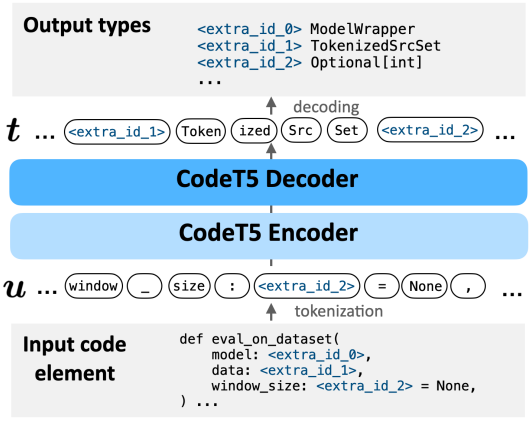

Figure 3.2: How CodeT5 encodes and decodes source code using BPE. Marker tokens (highlighted in blue) indicate gaps (input) and their corresponding fillers (output).

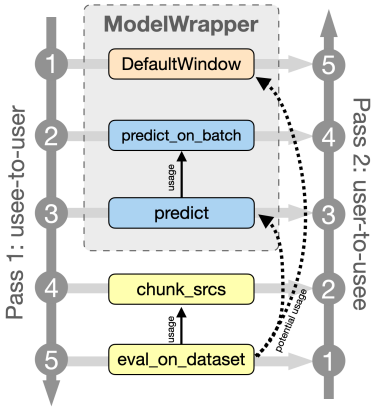

Figure 3.3: The two-pass iterative decoding process.

ing (Gage, 1994, Radford et al.) (see Figure 3.2), it can, in principle, predict arbitrary code snippets to fill masked gaps—including type annotations with complex parametric types and user-defined classes.

Incorporating context through static analysis By using surrounding code as the prediction context, our fine-tuned version of CodeT5 can relatively easily predict the correct type annotations of some of the variables. For example, based on the names and default values of `n_seqs` (model.py, line 7) or `window_size` (eval.py, line 4), CodeT5 can figure out the correct types of these parameters. However, for other parameters such as `model` in line 2 of eval.py, the surrounding context does not contain enough information to make a reasonable prediction. To see why, observe that `ModelWrapper` is a new class defined in a separate file, so, (1) it has never been seen during training, and

(2) its definition is not available as part of the context. Similarly, it is also very difficult to predict the return type of `eval_on_dataset` since it directly returns the result of `model.predict`, whose definition is also not available to the model.

To address this issue, our approach enhances the prediction context of CodeT5 using static analysis. The details of how we construct the context from static analysis will be described in subsection 3.3.3, but, in a nutshell, our approach analyzes the user-usee relations among code elements and pulls in the relevant definitions into the context. For example, when the model is making a prediction for `eval_on_dataset`, the context includes the definitions of `DefaultWindow` and `predict`, which are defined in `ModelWrapper` and invoked at line 6 and line 12 of eval.py, respectively.

TypeT5 architecture As there are many dependencies between different code elements, making *independent* type predictions for each code element is not ideal. For example, to infer the return type of `eval_on_dataset`, we would need to know the return type of `ModelWrapper.predict`, which depends on the return type of the `self.predict_on_batch` (model.py, line 12). However, since there is limited context that can be fed to the model, it is not feasible to include all transitive users and usees. To deal with this problem, TypeT5 utilizes an iterative decoding scheme that allows conditioning on previous type predictions. In more detail, our decoding scheme first sorts the user-usee graph topologically and then performs two sequential prediction passes, first

from usee to users and then going in the reverse direction, as illustrated in Figure 3.3. To see why both of these directions are useful, observe that the return type of `eval_on_dataset` depends on `predict`, which in turn depends on `predict_on_batch`. Thus, propagating information from callees to callers is clearly useful. Conversely, consider predicting the type of `data`, the first parameter of `predict`. Since the return value of `chunk_srcs` is passed as the first argument of `predict`, propagating information in the reverse direction can also be helpful.

3.3 Methods

3.3.1 Using CodeT5 for type prediction

We formulate type prediction as a sequence-to-sequence (seq2seq) task. Let $\boldsymbol{u} = (u_1, \ldots, u_n)$ represent a sequence of code tokens, where each token is a single untyped code element e (function or variable). We insert into $\boldsymbol{u}$ indexed marker tokens (`<extra_id_i>`) at each point where we wish to predict types and let the model predict $\boldsymbol{t} = (t_1, \ldots, t_m)$, the token sequence that encodes types for the marked locations in $\boldsymbol{u}$. Note that $\boldsymbol{t}$ only contains the types, no other code tokens, in the format `<extra_id_1> [type 1 tokens] <extra_id_2> [type 2 tokens]`, etc. We use the same tokenizer as CodeT5, which allows encoding any Python expression as a (typically short) sequence of subword tokens.

Our baseline CodeT5 model is a Transformer seq2seq model $P(\boldsymbol{t} \mid \boldsymbol{u}, \bar{\boldsymbol{u}})$ placing a distribution over type sequences $\boldsymbol{t}$ given the raw code sequence $\boldsymbol{u}$ and

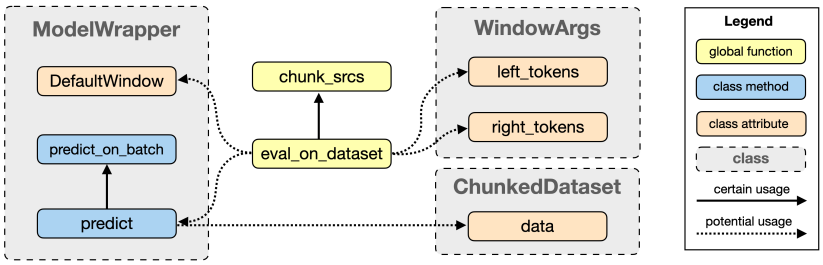

Figure 3.4: The usage graph corresponding to the code snippets in Figure 3.1. `WindowArgs` and `ChunkedDataset` are assumed to be defined elsewhere in the same project.

its surrounding code $\bar{\boldsymbol{u}}$. This process is shown in Figure 3.2, and $\bar{\boldsymbol{u}}$ is omitted for clarity.

Our improved model, TypeT5, replaces the surrounding code $\bar{\boldsymbol{u}}$ with a context constructed from static analysis, which we denote as $\boldsymbol{s}$. Thus, we can write the model as $P(\boldsymbol{t} \mid \boldsymbol{u}, \boldsymbol{s})$. Note that, in both models, we remove all comments and Python docstrings as a pre-processing step, as was done in prior work. We now show how $\boldsymbol{s}$ is constructed from static analysis.

3.3.2 Building the Usage Graph

To overcome the limitations of using *only* the surrounding code as context, our approach relies on static analysis to extract relevant global information about the codebase. In particular, our analysis constructs a *usage graph*, whose nodes correspond to code elements and edges represent a direct usage. For example, if x and y are both functions, an edge from x to y means that x calls y. Similarly, for variables, an edge from x to y means that the value of x

depends on y.

The usage graph for our previous example is shown in Figure 3.4. We show two types of usages: a certain usage, shown as solid arrows, and a potential usage, shown as dotted arrows. A certain usage is one that can be statically determined without knowing the types of the involved variables. For example, the global function `chunk_srcs` is directly used by `eval_on_dataset`, and we also know that `predict_on_batch` is called by `predict` (`self` method call on line 12). For other cases like a non-`self` attribute access or method call, the target depends on the type of the receiver, so we first collect all attributes and methods with a matching name from the current project and then generate a potential usage for each of them. We give the details of how we construct usage graphs for Python programs below.

Analyzing Python usage graphs To resolve the user-usee relations at the project level (used in subsection 3.3.2), we built a custom static analysis pipeline on top of the `libcst` library. We use `libcst` to parse Python files and also rely on its utilities to resolve symbol references within the same file (i.e., it tells us whether a local name refers to a function defined in the current file or comes from an import statement, etc.) We then implement custom import resolution logic (following Python's module rules) and combine it with `libcst` to resolve all certain usages within the current project. For unresolved usages with a syntactic form matching a class attribute/method usage, we generate a potential usage to each class members with a matching name. Since the

involved static analysis operations are fairly lightweight and we parallelize the construction of different graphs on CPUs, the time spent on constructing the usage graphs only makes up a tiny fraction of the total training or inference time. For example, using 28 Python processes, it only takes about 8 minutes to process the entire BetterTypes4Py training set, including the time spent on other tasks such as parsing, code transformation, and tokenization. While our implementation limits the analysis scope to the current project, it is straight-forward to extend the analysis to also include usages involving library definitions, which may help the model see type signatures of uncommon library APIs. We did not do this in our experiments mainly due to the manual effort it would require to install the correct library dependencies for all the projects in our datasets.

3.3.3 Constructing Model Inputs

TypeT5 leverages the usage graph $\mathcal{G}$ to construct the inputs to our model. In particular, we define the model input for a code element e to be a concatenation of four parts: the preamble $\boldsymbol{s}_{\text{pre}}$, usees $\boldsymbol{s}_{\text{usee}}$, main code $\boldsymbol{u}$, and users $\boldsymbol{s}_{\text{user}}$, which are constructed as follows (see section 3.6 for a concrete example).

- **Preamble**: The main purpose of $\boldsymbol{s}_{\text{pre}}$ is to help the model map each local name to the definition that it is referring to, so the preamble includes all import statements. Additionally, to help the model see which types are available in the current scope, the preamble also includes the headers of all

class definitions as well as all the type variable declarations from the current file.

- **Main code**: For the code element e of interest, we use its definition to construct $\boldsymbol{u}$. If e is a function, $\boldsymbol{u}$ is just its source code, and if e is a variable, $\boldsymbol{u}$ includes all top-level assignment statements in which e appears as the left hand side. Additionally, if e is a method or attribute of a class C, we also indent $\boldsymbol{u}$ and prepend it with the header of C, as shown in Figure 3.7.

- **Users**: Given the usage graph $\mathcal{G}$, we construct $\boldsymbol{s}_{\text{user}}$ by including the source tokens of all elements from $\text{users}(\mathcal{G}, e)$. Since these elements can come from different source files, we also prepend each element with a special comment denoting the Python module it comes from. Note that these elements can optionally contain the types predicted by our TypeT5 model, as described later in subsection 3.3.4.

- **Usees**: $\boldsymbol{s}_{\text{usee}}$ contains not just the direct users of e, but also anything that is used in the user context $\boldsymbol{s}_{\text{user}}$. i.e., $\boldsymbol{s}_{\text{usee}}$ contains the elements from $\text{usees}(\mathcal{G}, e) \cup \bigcup_{e' \in \text{users}(\mathcal{G}, e)} \text{usees}(\mathcal{G}, e')$. Since this generally contains many more elements than $\boldsymbol{s}_{\text{user}}$, we only use the (typed or untyped) *signatures* of the elements to construct $\boldsymbol{s}_{\text{usee}}$.

We limit the total input size to be at most 4096 subword tokens and cut off any exceeding tokens from both left and right, centered around the main code. Context elements involving certain usages are arranged to be closer to the center so that they are always prioritized over potential usages.

3.3.4 Iterative Decoding Inference

We now describe how to conduct inference in our base model as well as our context-augmented model using an iterative decoding scheme.

CodeT5 decoding: Given our trained model $P(\boldsymbol{t} \mid \boldsymbol{u}, \bar{\boldsymbol{u}})$, we can infer a most likely set of types $\boldsymbol{t}$ for $\boldsymbol{u}$ (with surrounding context $\bar{\boldsymbol{u}}$) using beam search. Our implementation performs joint prediction of the output types for a single code block $\boldsymbol{u}$, since later types in the block condition on the predictions of earlier types. However, note that both $\boldsymbol{u}$ and $\bar{\boldsymbol{u}}$ are always *completely untyped* code: while we condition on previous types as we predict, these are not inserted into the prediction context for the next element.[1]

TypeT5 iterative decoding: Part of our motivation for including the context $\boldsymbol{s}$ is to exploit its type information at inference time. Crucially, this requires $\boldsymbol{s}$ to be *typed*. However, the contexts that are drawn from the original codebase are not typed, so TypeT5 iteratively adds type signatures to these contexts using its own predictions. Let $\mathcal{M}$ be the type assignment produced by the model, which maps each code element e to its predicted type $\boldsymbol{t}_e$, and denote $\mathcal{M}(\boldsymbol{s})$ as the context obtained by annotating the elements in $\boldsymbol{s}$ according to $\mathcal{M}$. Starting with an empty $\mathcal{M}$ (which leaves any context $\boldsymbol{s}$ unchanged), TypeT5 then iteratively updates $\mathcal{M}$ using an iterative decoding scheme that traverses the usage graph $\mathcal{G}$ twice, as shown in Figure 3.3. The

[1] In our experiments, including predicted types actually hurts the performance due to exposure bias.

first prediction pass follows the usee-to-user order,[2] while the second pass goes in the reverse direction to allow for bidirectional information flow. At any given time step, we can denote the model's prediction for element e as drawn from $P(\boldsymbol{t}_e \mid \boldsymbol{u}_e, \mathcal{M}(\boldsymbol{s}_e))$, and the predicted types $\boldsymbol{t}'_e$ are then used to update $\mathcal{M}$ such that $\mathcal{M}(e) = \boldsymbol{t}'_e$.

3.3.5 Training

To save computation and improve parallelism during training, we use the available human annotations as a gold type assignment $\mathcal{M}^*$ instead of letting the model condition on its own prediction. Note that this type assignment is generally incomplete and may not contain a label for every missing type. We train the model to maximize the log likelihood of predicting the ground truth, i.e., $\log P(\boldsymbol{t}^*_e \mid \boldsymbol{u}_e, \mathcal{M}^*(\boldsymbol{s}_e))$, for every element e where $\boldsymbol{t}^*_e$ is available, using teacher-forcing. We train the CodeT5 baseline model similarly on the same dataset.[3]

3.4 Experiments

We implement TypeT5 in Python, whose the source code and model weights can be found on GitHub[4]. Below, we first describe our evaluation setup and list the hyperparameters and hardware details. We then compare TypeT5

[2]This requires performing a topological sort over $\mathcal{G}$. When $\mathcal{G}$ is not a DAG, we break the cycles arbitrarily.

[3]Note that without this training (fine-tuning) step, the original CodeT5 model performs very poorly as it tends to predict arbitrary Python expressions that are not types.

[4]Available at `https://github.com/utopia-group/TypeT5`.

against three state-of-the-art type inference systems for Python, namely Typilus (Allamanis et al., 2020), Type4Py (Mir et al., 2022), and HiTyper (Peng et al., 2022). Finally, we present ablation studies to evaluate different factors contributing to the model's performance.

3.4.1 Evaluation Setup

Datasets In our evaluation, we predict the type annotations for *top-level* code elements of each Python project. These include all class attributes, methods, top-level functions, and global variables. We treat any existing user-added type annotations as the ground truth, and we use per type accuracy as the main performance metric. Following a setting similar to that of Allamanis et al. (2020) and Wei et al. (2020), we split our dataset per Python project. This way, types newly defined in the test projects will not have been seen during training. Such a setting is more challenging than splitting the dataset per file, as is done in Type4Py and other work (Mir et al., 2022, Pradel et al., 2020), but more closely reflects the model's real-world performance (Gruner et al., 2022).

Our main dataset, **BetterTypes4Py**, is constructed by selecting a high-quality subset from the ManyTypes4Py dataset (Mir et al., 2021), which was used to train Type4Py. We describe the selection criteria in the next paragraph. Since our model is fine-tuned from the CodeT5 model (which may have already been pre-trained on some of the test repositories in the aforementioned dataset), we additionally construct **InferTypes4Py**, a test

set derived from the source code of Typilus, Type4Py, and our own tool, none of which were used as CodeT5's (pre-)training data.

Constructing the BetterTypes4Py dataset Our dataset is constructed from the ManyTypes4Py dataset as follows: we first filter out the GitHub repositories that are no longer accessible or that fail to download within 10 seconds. This leaves us with 4890 out of the 5996 original projects. Then, we discard projects that have not been updated for more than one year (to avoid outdated library APIs), reducing the number of repositories to 1218. To limit the influence of any particular project, we also filter out the 37 projects with more than 50K lines of code. Finally, to exclude those projects that have very few type annotations, we compare the number of type annotations n_t of each project with its lines of code n_c and filter out those with a n_t-to-n_c ratio less than 1:20. This gives us our final set of 663 projects. We then randomly select 50 test and 40 validation projects and use the rest for training.

Code Duplication Code duplication (Allamanis, 2019) has been shown to have adverse effects on the performance of ML models and can blur the evaluation results. Hence, prior work like Type4Py applies file-level deduplication to remove duplicated source files. However, this is hard to do in our project-based setting since we need all files to be present during inference. To address this, we have manually verified that our InferTypes4Py dataset does not contain files that are copy-pasted from elsewhere. We also run the popular

code duplication tool `jscpd`[5] on our test set to detect duplicated code blocks. The analysis shows that there is relatively little duplication in the dataset (only around 4% of duplicated lines), and the majority of these duplications came from the same project rather than across projects, so we believe code duplication is not a major issue under our by-project evaluation.

Label Quality In our evaluation, developer-provided type annotations are used as the ground truth, which are not always accurate or coherent (Ore et al., 2018). This partially motivated us to construct the InferTypes4Py dataset, which consists of high-quality type annotations with a relatively low error rate. In particular, our own codebase (which are included in InferTypes4Py) makes heavy use of type annotations throughout the development process and is continuously type-checked by VSCode. As a very rough metric to approximate label quality, we report the coherence error (defined in section 3.4.3) of the human labels on both datasets: On BetterTypes4Py and InferTypes4Py, the average coherence error per human annotation is 0.019 and 0.045, respectively.

We summarize key properties of both datasets in Table 3.1. We define the size of a type as the number of type constructors in its body[6] and categorize a type as **simple** if its size is 1, and **complex** otherwise. We also categorize a type as **rare** or **common** depending on whether it contains a rare type

[5]https://github.com/kucherenko/jscpd

[6]e.g., both `int` and `foo.Bar` has a size of 1, whereas `dict[str, foo.Bar]` has a size of 3.

Table 3.1: Basic statistics of our two datasets.

	BetterTypes4Py			**InferTypes4Py**
	train	*valid*	*test*	*test*
Projects	573	40	50	3
Nonempty files	16.5K	1098	949	99
Lines of code	2.4M	174K	139K	21K
Top-level type slots	541K	38.2K	28.4K	4.6K
Top-level user-added types	275K	19.3K	15.8K	2.7k
Rare type ratio	25.7%	23.3%	35.0%	33.8%
Complex type ratio	20.4%	16.6%	20.8%	33.4%
Average type size	1.42	1.33	1.43	1.72

constructor that is not from the top-100 most frequent type constructors in our training set.

Accuracy metrics. Since Python allows the same type to be written in different syntactic forms,[7] we first perform type normalization to convert both the predicted and ground-truth types into a canonical form. The details of this normalization step can be found in the next paragraph. and we use the term **full accuracy** to refer to the accuracy against all human annotations after normalization. To better compare with prior work, we also define **adjusted accuracy** (our main metric), which (1) filters out all `None` and `Any` labels (as in prior work); (2) converts fully qualified names to simple names (e.g., `Tensor` instead of `torch.Tensor`) since some prior approach does not output correctly qualified types; (3) rewrites any outermost `Optional[T]` and `Final[T]` into

[7] e.g., both `Union[int,None]` and `Optional[int]` refer to an integer that can also be `None`, and both `list` and `List[Any]` refer to a python list with untyped elements

`T` since they tend not to be used consistently across programmers.[8] Finally, we also define a **base accuracy** metric that is the same as adjusted accuracy except that it only checks the outermost type (e.g., `Dict[str,List]` will match any `Dict` but, for example, not `Mapping`.)

Type Normalization To compute the accuracy metrics in subsection 3.4.1, we recursively apply the following steps to normalize a Python type:

1. Rewrite any `Optional[T]` to `Union[T,None]`.
2. Sort the arguments of `Union` types and flatten any nested `Union`s.
   e.g., rewrite `Union[B,Union[C,A]]` into `Union[A,B,C]`.
3. If all type arguments are `Any`, drop them all. e.g., rewrite `List[Any]` to `List`.
4. Capitalize the names of basic types. e.g., rewrite `list` to `List`.

Hyperparameters and Running times We initialize our model's weights from the CodeT5 model provided by Huggingface Transformers library (Wolf et al., 2020) and train our model for exactly one epoch using the library's default optimizer configuration (AdamW with a base learning rate of 2e-5 and a weight decay of 0.01.) During inference, we use beam search with a beam width of 16 and diversity penalty of 1.0. We adaptively set the maximal output

[8] e.g., the type checker `mypy` has an option to enable implicit `Optional` types, so it would not be possible for the model to know if it should output `Optional[T]` or `T` just from the untyped code.

sequence length to be $16n+10$, where n is the number of types to be predicted in the input. We set the size limit of preamble, usee Context, main code, and user Context to 1000, 2048, 512, 1536, respectively, both during training and test time. Note that preamble uses the space within usee context, so the total maximal input size is 4096. Training the model took about 11 hours on a single Quadro RTX 8000 GPU with 48GB memory. Performing the two-pass inference on the BetterTypes4Py test set takes about 4 hours, whereas performing a single-pass inference (e.g., UseeToUser) takes about half the time. For comparison, training CodeT5 model on the same machine takes about 3.7 hours, and the corresponding evaluation takes about 0.5 hour.

3.4.2 Comparing TypeT5 with other approaches

We compare TypeT5 with the basic CodeT5 model described in subsection 3.3.4 as well as the released versions of three other state-of-the-art approaches from prior work.[9] Typilus (Allamanis et al., 2020) models the program as a graph and applies a graph neural network to compute the types of variables. The original Typilus model can predict from a set of common types as well as (nonparametric) user-defined types, but their released model can only predict common types, so we only evaluate its performance on common types. Type4Py (Mir et al., 2022) uses variable names and the surrounding code to compute an embedding and performs type prediction via a nearest-

[9]Since our approach benefits from CodeT5's large-scale pre-training across 8 different programming languages, we use a smaller training set than prior work and do not retrain these prior approaches on our dataset.

Table 3.2: Accuracy comparison on **common** types.

	BetterTypes4Py				
	full	**adjusted**			**base**
	all	all	simple	complex	all
Typilus	n/a	54.05	55.12	33.23	60.37
Type4Py	n/a	50.34	51.91	32.14	47.51
HiTyper	59.20	54.28	57.70	26.44	59.01
CodeT5	76.74	78.04	82.43	53.03	82.44
TypeT5	**79.24**	**81.43**	**85.69**	**56.75**	**84.82**

	InferTypes4Py				
	full	**adjusted**			**base**
	all	all	simple	complex	all
Typilus	n/a	52.33	52.19	53.91	64.67
Type4Py	n/a	32.08	33.47	16.54	29.83
HiTyper	45.67	43.54	46.00	19.27	47.99
CodeT5	77.83	78.06	85.31	63.41	81.87
TypeT5	**81.75**	**82.95**	**87.62**	**72.78**	**84.17**

Table 3.3: Accuracy comparison on **rare** types.

	BetterTypes4Py				
	full	**adjusted**			**base**
	all	all	simple	complex	all
Type4Py	n/a	12.37	13.17	4.05	14.15
HiTyper	10.30	25.51	27.59	9.79	29.33
CodeT5	49.47	52.95	57.28	34.26	57.65
TypeT5	**58.56**	**61.47**	**65.21**	**40.22**	**68.44**

	InferTypes4Py				
	full	**adjusted**			**base**
	all	all	simple	complex	all
Type4Py	n/a	0.25	0.14	0.98	0.17
HiTyper	9.36	9.36	10.79	1.19	12.33
CodeT5	51.64	53.28	59.97	30.62	66.47
TypeT5	**53.44**	**56.27**	**61.50**	**36.92**	**69.23**

neighbor search in the type embedding space. We run the released Type4Py model via its web interface. HiTyper (Peng et al., 2022) combines the strengths of a rule-based type inference algorithm with ML type inference models by only invoking the ML model on places where the inference algorithm gets stuck and deducing the types elsewhere using typing constraints. Its implementation uses Type4Py as the default ML backend, which we use to run HiTyper.

We show each tool's accuracy in Table 3.2 (on common types) and Table 3.3 (on rare types). Since HiTyper was only able to make a predition for about 67% of all labels, we report its performance on this subset. From Table 3.2, we can make the following observations. (1) Our CodeT5 baseline model already outperforms all prior approaches by a large margin, demonstrating the advantage of using a seq2seq pre-trained language model. (2) TypeT5 further improves the adjusted accuracy by 3.4% and 4.9% on the two datasets. (3) Type4Py's performance on InferTypes4Py dataset is significantly lower than on BetterTypes4Py, likely because Type4Py was originally trained on some of the test files in BetterTypes4Py. Looking at **??**, we see that both CodeT5 and TypeT5 dramatically outperform the other approaches on rare types. Moreover, TypeT5 achieves the largest improvements on types that are both rare and complex, improving upon CodeT5 by about 6% on both datasets, suggesting that that global information is especially important in these cases. For a qualitative analysis, we also show TypeT5's outputs on a real-world example in section 3.6.

Why Type4Py has much lower performance on our dataset The accuracies reported in the original Type4Py paper are much higher than we measured here. We have discussed our experiments with the authors of Type4Py and believe that the discrepancy we observe is likely due to the combination of two reasons: First, while our evaluation only counts the type annotations on all top-level APIs, Mir et al. (2022) includes all local variables in their evaluation as well. Second, while we only evaluate on human annotations, they also include machine-inferred type annotations (via the type checker Pyre) as ground-truth labels. As a result, the distributions of labels reported by the two papers are significantly different: in our setting, function annotations (parameters + return types) constitute the majority (87%) of the labels, whereas in Mir et al. (2022), their portion is significantly smaller, merely 18%. This suggests that their label set is likely inflated by simple labels inferrable from the type checker, which explain the performance drop when evaluated on our datasets.

3.4.3 Ablations on TypeT5

We next present a series of ablations that evaluate how various factors contribute to TypeT5's performance. In addition to accuracy (on all types), we also report the **type error count** as a way to estimate how *coherent* the model's predictions are.[10]

[10] Accuracy does not always correlate with coherency. e.g., when we have `x = y`, and `x` and `y` are equally likely to be a `str` or an `int`, a coherent type assignment needs to ensure that `x` and `y` always have the same type, even if this requirement does not lead to a higher

Measuring Type Coherence using Type Errors To estimate the type coherence (subsection 3.4.3), we call the type checker `MyPy`[11] on codebases annotated with the types predicted by the model. Since not all errors reported by `MyPy` are type errors or are related to type coherence, we only count the errors with the following 5 error codes, whose meaning according to `MyPy`'s documentation are:

- `attr-defined` checks that an attribute is defined in the target class or module when using the dot operator.
- `arg-type` checks that argument types in a call match the declared argument types in the signature of the called function.
- `return-value` checks that the returned value is compatible with the type signature of the function.
- `assignment` checks that the assigned expression is compatible with the assignment target.
- `name-defined` checks that a name is defined in the current scope.

Note that this metric does have its limitations. One undesired property we found is that it can favor predicting a non-existing type over an incorrect type. For instance, when the model predicts an incorrect type on a function, the type

accuracy in expectation.

[11] http://mypy-lang.org/.

Table 3.4: Performance of different model modifications. All models are retrained with the corresponding inputs.

Modification	Accuracy	Type Error
No Preamble	64.20	6067
No Users	71.20	7053
No Usees	67.25	7332
Nonincremental	72.52	5720
Original (TypeT5)	**73.02**	**5087**

checker will check all the usages of that function against this declared type and will thus likely report multiple errors. However, if the model predicts a non-existing type, the type checker will only report a single `name-defined` error at the declaration site and will skip checking its usages. This effect has caused the No Preamble variant in Table 3.4 to have a lower error count than other two other variants since it tends to predict a lot more non-existing types. But we have verified that it was not the cause of the type error count improvement by our two-pass sequential decoding model.

How do different components contribute to the model's performance? To evaluate the impact of each context element, we remove one component at a time and *retrain the model* accordingly. In particular, the **No Preamble**, **No Users**, and **No Usees** ablations correspond to removing the $\boldsymbol{s}_{\text{pre}}$, $\boldsymbol{s}_{\text{user}}$, and $\boldsymbol{s}_{\text{usee}}$ context elements (introduced in subsection 3.3.3) from the model's input, respectively. The **Noniterative** model does not perform iterative decoding and is trained to condition on an untyped context. We use the same input size limit (4096 subword tokens) for all models, so a model that does

Table 3.5: Performance of different decoding strategies. The same TypeT5 model weights are used for different decoding strategies.

Strategy	**Accuracy**	**Type Error**
Independent	71.68	6876
Random	71.66	6215
UserToUsee	70.67	7415
UseeToUser	72.65	6402
TwoPass (TypeT5)	**73.02**	**5087**

not utilize one kind of information have more space for other kinds. We show both the adjusted accuracy on all types and type error count in Table 3.4. We can see that (1) all components improve the overall accuracy, with the preamble having the largest impact, and (2) while the iterative decoding scheme only improves overall accuracy slightly, it significantly improves the model's coherence, resulting in 12% fewer type errors.

How do different decoding strategies compare? In addition to the two-pass iterative decoding strategy introduced in subsection 3.3.4, we also test four other decoding strategies: (1) **Independent**, which independently predicts the type signature for each element without conditioning on the model's own prediction (same as the Noniterative model except not retrained); (2) **Random**, which visits each element once following a random order; (3) **UserToUsee**, which visits the users before the usees; (4) **UseeToUser**, which visits the usees before the users. The results are shown in Table 3.5. We can see that our proposed TwoPass decoding scheme yields the largest accuracy and type error improvement over Independent; whereas UserToUsee performs

worse than Independent, suggesting that bad decoding ordering can have adverse effects on the model's performance.

3.4.4 User-Guided Interactive Decoding

Compared to prior work, our approach has a unique strength: because the model can condition on previous types, it allows easy user intervention by conditioning on any corrections made by the user. We thus explore an alternative use case of our model where the user interactively inspects each type signature predicted by the model and makes any necessary corrections before the model moves on to the next prediction. We emulate this interactive process using the ground-truth human annotations $\mathcal{M}^*$, and we modify the usee-to-user decoding process to let the model predict the types of each element e (as before), but then override the predicted types $\boldsymbol{t}_e$ with the corresponding human annotations $\mathcal{M}^*(e)$ if $e \in \mathcal{M}^*$. On the BetterTypes4Py dataset, this interactive decoding process achieves a full accuracy of **78.04%** and an adjusted accuracy of **79.37%**—meaning that on average, it only requires the user to correct one in every five model-predicted types to fully annotate an entirely untyped project from scratch. Note that only 59% of the test set contains a user type annotation, so some incorrect type annotations may not be corrected immediately and can get propagated to other places. Thus, in an interactive real-world setting, we can expect the actual accuracy to be even higher.

3.5 Related Work

Deep Learning Type Inference Most prior approaches predict user-defined types via some form of type embedding matching. e.g., LambdaNet (Wei et al., 2020) tackles this challenge by combining graph neural networks with a pointer network. Typilus (Allamanis et al., 2020) performs nearest-neighbor search in the type embedding space. However, neither approach can handle the unbounded type space induced by parametric types. TypeBert (Jesse et al., 2021) is the first type inference method based on pre-trained transformer models and has demonstrated superior performance in predicting common types. Jesse et al. (2022) later improved TypeBert using deep similarity learning to better support user-defined types. Different from our work, TypeBert does not use a seq2seq model or construct context from static analaysis. Apart from just using the source code, other forms of information have also been utilized for type prediction. Both TypeWriter (Pradel et al., 2020) and HiTyper (Peng et al., 2022) combines type checking with inference-time search to avoid generating type errors. OptTyper (Pandi et al., 2020) explicitly models type constraints by turning them into a continuous optimization objective. NL2Type (Malik et al., 2019) focuses on predicting types from natural language information such as code comments and documentation.

Retrieval-based models in NLP Similar to our context augmentation with static analysis, a number of methods have been developed to augment pre-trained models with context in natural language processing (NLP). For

open-domain question answering (Chen et al., 2017), approaches like REALM Guu et al. (2020) and DPR (Karpukhin et al., 2020) can retrieve knowledge relevant to a query, and retrieval-augmented generation (Lewis et al., 2020) and its extensions (e.g., Fusion-in-Decoder (Izacard and Grave, 2020)) have shown that it is possible to generate longer outputs using this information. RETRO (Borgeaud et al., 2021) and WebGPT (Nakano et al., 2021) take this to a web-scale extreme. However, we are also able to leverage static analysis based on the usage graph, which has no analogue for text.

Linking with T5 While our use of CodeT5 for type prediction is novel to our knowledge, T5 (Raffel et al., 2019) has been applied to a range of NLP tasks like summarization. Most similar to ours is its use for entity linking (Petroni et al., 2021): Systems in this vein generate names of entities token by token (De Cao et al., 2020) and are able to generalize to new entities like TypeT5. However, the presence of ad hoc types for each new context and the types of context clues needed are very different in the code setting than in natural language.

Structured Prediction Our iterative decoding process can be viewed as applying a learned policy to perform structured prediction (BakIr et al., 2007, Daumé et al., 2009). In this work, our training scheme can be viewed as behavior cloning since the model directly conditions on ground-truth human annotations, which can lead to distributional mismatch between training and

inference time. Applying more advanced training schemes such as Scheduled Sampling (Bengio et al., 2015), DAgger (Ross et al., 2011), or reinforcement learning (Sutton and Barto, 2018) may help further boost the performance of iterative decoding.

3.6 Real Examples Produced by TypeT5

We run TypeT5 on the actual code corresponding to the example shown in Figure 3.1 and show the obtained preamble (Figure 3.5), usee context (Figure 3.6), main code (Figure 3.7), and user context (Figure 3.8) for the `ModelWrapper.predict` element. For each type predicted by the model, we indicate whether it is correct with a green or red marker, and if not, also show the ground truth type in red. Note that these outputs were generated by the model in the second pass of the sequential decoding process (described in subsection 3.3.4), so all the elements have already been annotated at least once (but some may have not been annotate the second time).

model_input.py **Preamble**

```python
import random
from copy import copy, deepcopy
import numpy as np
from collections import Counter
from mypy_extensions import mypyc_attr
from torch import Tensor
from transformers.data.data_collator import DataCollatorForSeq2Seq
from typing import NamedTuple, overload
from datasets.arrow_dataset import Dataset
from torch.utils.data import DataLoader, RandomSampler
from.data import (
    ChunkedDataset,
    CtxArgs,
    TokenizedSrcSet,
    output_ids_as_types,
    preds_to_accuracies,
)
from.type_env import AccuracyMetric, PythonType
from.utils import *
@dataclass
class DecodingArgs:
    ...
@dataclass
class DatasetPredResult(Generic[T1]):
    ...
@dataclass
class ModelWrapper:
    ...
```

Figure 3.5: The preamble gathers all the important statements and class headers from the current file. This helps the model see which types are available and where each symbol comes from.

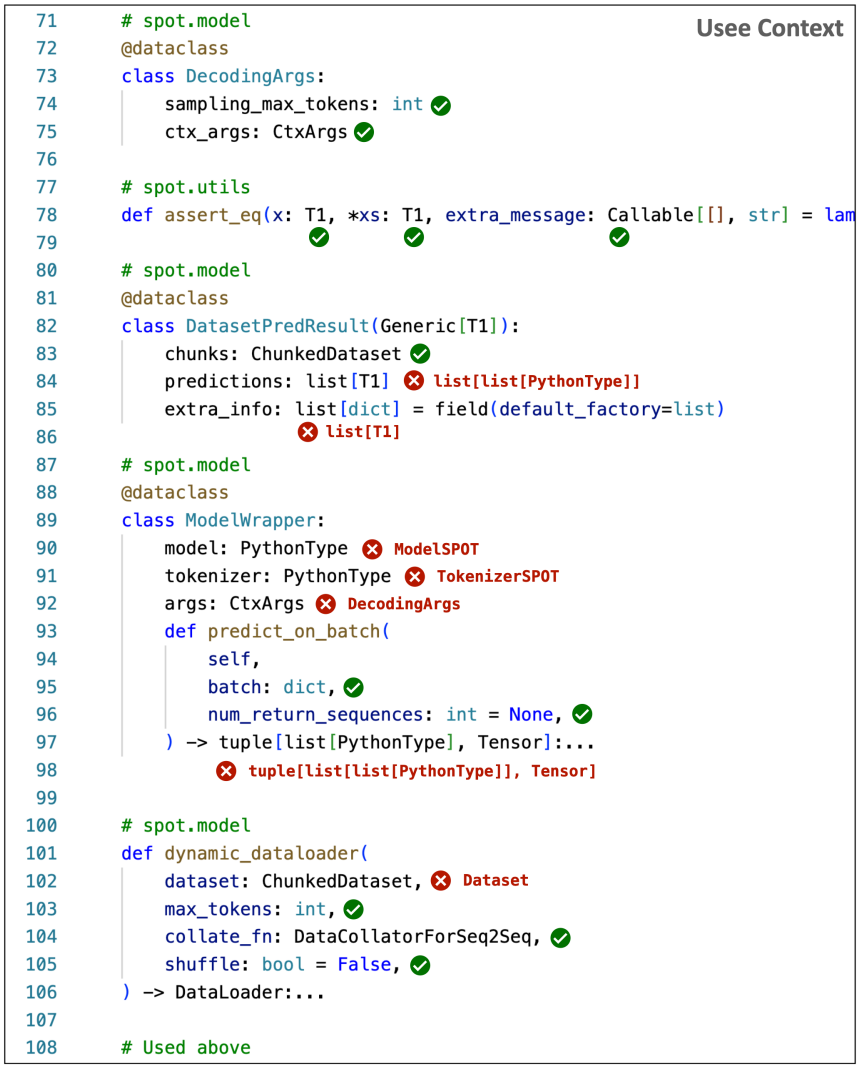

Figure 3.6: The usee context shows the signature of the elements that are used by the main code or by elements from the user context. By seeing their predicted type signatures, the model can understand the type-level behavior of these definitions without having to dive into their implementation.

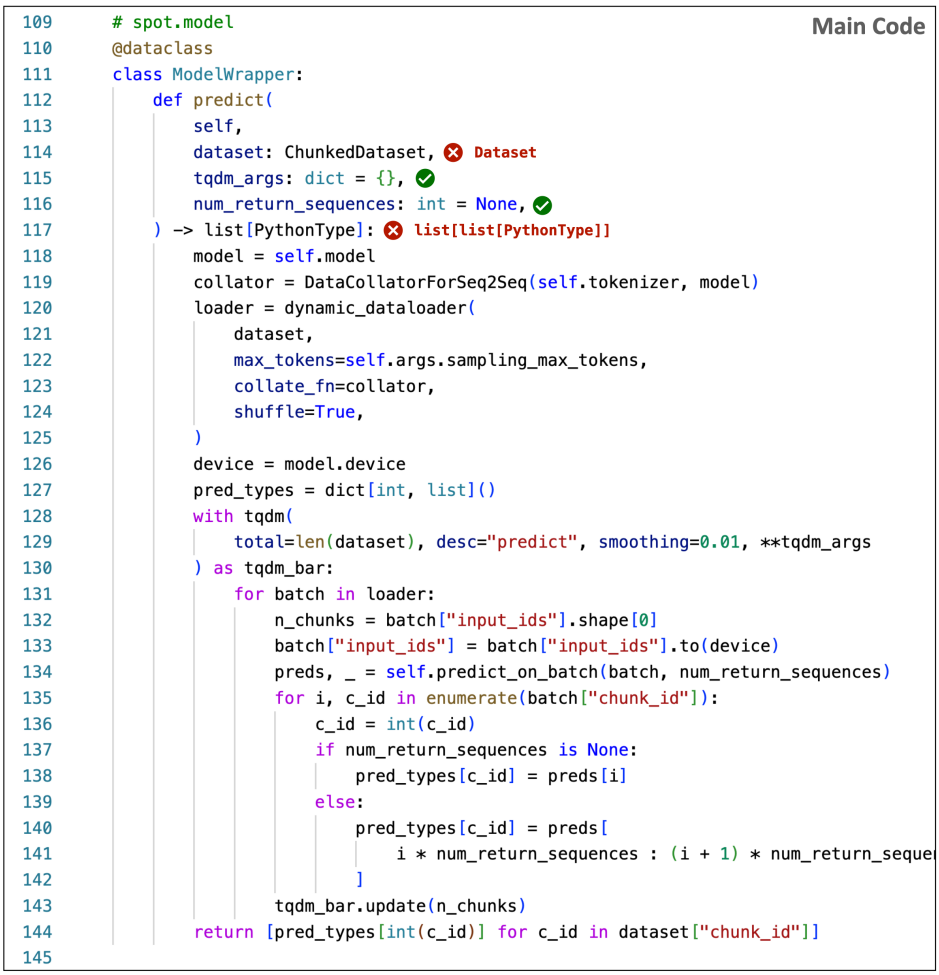

Figure 3.7: The main code is the element that is being annotated by the model at the current decoding step. The model has made two errors in this example, both of which can be directly attributed to the previous two errors made in the usee context (line 102 and line 97). This shows that the model is making coherent predictions according to the context, and such errors can be avoided if the user has corrected the previous errors (as described in subsection 3.4.4).

User Context, first half

```python
    # Users below
    # spot.model
    @dataclass
    class ModelWrapper:
        def eval_on_dataset(
            self,
            src_data: TokenizedSrcSet,
            max_labels: int = None,
            tqdm_args: dict = {},
        ) -> DatasetPredResult:
            ctx_args = self.args.ctx_args
            if max_labels is not None:
                ctx_args = copy(ctx_args)
                ctx_args.max_labels = max_labels

            chunks = src_data.to_chunks(ctx_args, tqdm_args=tqdm_args)
            preds = self.predict(
                chunks.data, num_return_sequences=None, tqdm_args=tqdm_args
            )
            return DatasetPredResult(chunks, preds)

```

User Context, second half

```python
    # spot.decode
    def sample_candidates(
        wrapper: ModelWrapper,
        src_data: TokenizedSrcSet,
        n_samples: int,                tuple[ChunkedDataset, list[list[list[PythonType]]]]
    ) -> tuple[ChunkedDataset, list[PythonType]]:
        ctx_args = wrapper.args.ctx_args
        do_sample = wrapper.args.do_sample
        if not do_sample:
            assert wrapper.args.num_beams is not None, "num_beams needs to be set"
            assert n_samples <= wrapper.args.num_beams

        chunks = src_data.to_chunks(ctx_args)
        n_chunks = len(chunks.data)

        if do_sample:
            samples = [
                wrapper.predict(chunks.data, tqdm_args={})
                for _ in tqdm(range(n_samples), desc="Sampling")
            ]
        else:⋯

        def get_preds(chunk_id, sample_id):
            return (
                samples[sample_id][chunk_id] if do_sample else samples[chunk_id][sampl
            )

        pred_candidates = [
            [get_preds(cid, sid) for sid in range(n_samples)] for cid in range(n_chunk
        ]
        return chunks, pred_candidates

```

Figure 3.8: The user context shows two callers of the `predict` method from the main code. We see that the model successfully predicts all user-defined types, despite the fact that these are all new classes defined in the current project.

Chapter 4

Coeditor: Leveraging Contextual Changes for Multi-round Code Auto-editing

Building on our previous work on probabilistic type inference, we now shift our focus to a more general and important problem: contextual code editing prediction. While both tasks require analyzing programs at the project level and leveraging global information, the latter aims to predict how to edit a piece of code based on other relevant changes made elsewhere in the same project—in some sense, type inference can be viewed as predicting a special type of code edits. To address this challenge, we reuse some of the ideas from our type inference work, such as using lightweight static analysis to construct dynamic contexts and modeling our problem as masked span infilling.

4.1 Introduction

In recent years, there has been enormous interest in applying transformer models for code generation (Ahmad et al., 2021, Allal et al., 2023, Chen et al., 2021, Feng et al., 2020, Fried et al., 2022, Wang et al., 2021),

[0]An early version of this chapter was submitted to NeurIPS 2023. Jiayi Wei is the first author of the paper, and with the help from other authors, he developed the research idea, wrote the code, performed the experiments and analysis, and wrote the paper.

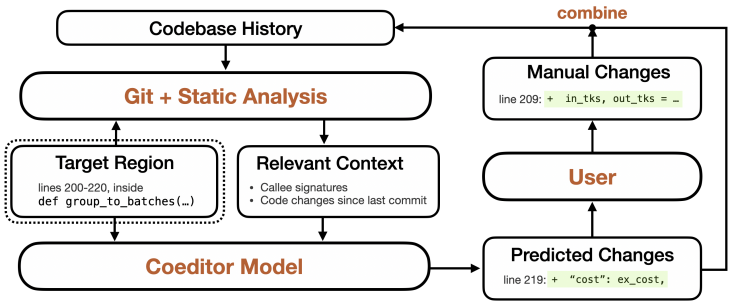

Figure 4.1: The multi-round auto-editing task. The user inspects the model output in each editing round and can optionally perform manual editing.

which has led to impressive performance on tasks such as program synthesis (Li et al., 2022, Nijkamp et al., 2022), program translation (Lachaux et al., 2020, Szafraniec et al., 2022), type inference (Jesse et al., 2022, Wei et al., 2023), and code auto-completion (Guo et al., 2021, Nguyen and Nadi, 2022, Svyatkovskiy et al., 2021, Zhang et al., 2023).

While these approaches effectively help programmers *creating* new code, they are not as adept at assisting with *revising* existing code. Code completion tools like GitHub Copilot do not track programmers' changes and cannot predict where and how to make additional modifications. However, during a software project's development cycle, developers often spend significant time editing code—changes made to one part of the codebase typically affect many others, and manually propagating these changes can be tedious and time-consuming.

In this work, we introduce a task that we call (multi-round) *auto-editing* where the goal is to predict edits to code conditioned on the user's previous

edits. In particular, given an original codebase U and a set of code changes $\Delta_1, \ldots, \Delta_k$ that are semantically related (like those forming part of a commit), the auto-editing problem is to predict how to modify a specified region of code $u \in U$ by learning the following distribution:

$$P(\Delta u \mid \Delta_k \ldots \Delta_1, U) \ . \tag{4.1}$$

Importantly, we allow the target region region u to overlap with any previous modifications $\Delta_1, \ldots, \Delta_k$ to support repeated editing to the same region. This formulation enables the workflow illustrated in Figure 4.1, where a user can work alongside the model in multiple editing rounds, accepting suggestions matching the user's intent and making additional edits manually if necessary.

To solve this problem, we propose a new model called Coeditor that builds on top of the established CodeT5 (Wang et al., 2021) model architecture and pre-trained checkpoint. Coeditor is based on two key ideas. First, it encodes all prior code edits $\Delta_1, \ldots, \Delta_k$ using a line-based diffing scheme and decodes Δu using masked span infilling; and, second, it uses lightweight static analysis to pull in relevant parts of the codebase U. To effectively handle large contexts with numerous code changes, we also replace CodeT5's dense attention with a block-sparse attention pattern, allowing us to reduce the computation cost while maintaining the ability to attend to all relevant code changes.

Another challenge in developing Coeditor is the lack of suitable training data for multi-round auto-editing. We address this issue by collecting a new

dataset, PyCommits, from the commit histories of 1650 open-source Python projects on GitHub. We compute tree-differences between adjacent codebase versions to identify modifications to the same Python function and randomly split some changes into the model input for training in repeated editing scenarios. During testing, we use ground truth code changes to simulate user decisions regarding when to accept partial changes suggested by the model and when to manually perform edits missed by the model.

We compare our approach against existing code infilling models and show that, even in a simplified setting that requires predicting a *single* edited line in isolation, they severely lag behind our change-aware model: our method achieves 60.4% exact match accuracy, almost twice that of the best performing code infilling model despite using a model that is 30x smaller. In the full multi-round setting, we found that Coeditor automates editing 46.7% of the changed lines, saving the user 28.6% of keystrokes measured by an edit distance metric that accounts for cursor movement.

In summary, this work presents the following main contributions:

- We introduce the multi-round code editing suggestion task, along with the corresponding PyCommits dataset and evaluation framework.
- We introduce a new code editing model derived from CodeT5, using a line diff-based encoding scheme and enhancements that enable the model to condition on long contexts and appropriate other parts of the codebase, addressing key challenges in this setting.

- We release our source code, dataset, model, as well as a VSCode extension that supports interactive usage to foster future research.

4.2 Motivating Example

In this section, we illustrate our technique using the example in Figure 4.2, showcasing a two-round interaction between the user and our Coeditor model. Subfigures (a) and (b) display two initial user changes, while subfigures (c) and (d) illustrate two sequential Coeditor invocations with inlined model suggestions. We further analyze this example in detail below.

First, the user modifies the `pack_batch` function in subfigure (a) to read a new dictionary key, `‘‘cost’’`, from each row in the input. The extracted values are used to compute the total cost of the batch and added to the output. Next, the user removes three lines at the top of the `group_to_batches` function in subfigure (b). By removing these three lines, the user wants to avoid creating these lists beforehand and instead plans to call the `process_edit` function inside the for loop below.

The user then scrolls down and invokes Coeditor at the bottom half of the same function (subfigure c). Here, the modified `pack_batch` function is called at lines 225 and 228 in subfigure (c), and its argument `current_batch` is iteratively constructed from `row`, which is a dictionary defined at line 215. Hence, the model correctly infers that `row` should be updated to include a `‘‘cost’’` key. Examining the surrounding context, the model also identifies that the `ex_cost` variable (defined at line 209) should be used as the inserted

Figure 4.2: An example usage of Coeditor. (a) The user first edits the `pack_batch` function to read an additional dictionary key, `‘‘cost’’`, from each row in the input. (b) The user then removes 3 lines at the top of the `group_to_batches` function. (c) The user now invokes Coeditor at the bottom half of the same function. Coeditor correctly suggests adding a `‘‘cost’’` key to the dictionary variable `row`, but it fails to address the now undefined variables underlined in red. (d) However, if the user accepts the suggested change and manually introduces two new variables at line 209, Coeditor can then suggest the correct changes accordingly.

dictionary value.[1]

While Coeditor makes some useful editing suggestions so far, it does not address the now-undefined variables underlined in red by the IDE in subfigure (c). In particular, as there are no obvious alternatives nearby to replace these variables, Coeditor is unable to automatically fix these errors. Such a situation is common when the surrounding changes alone do not provide sufficient information to derive a complete solution. Therefore, the user can accept the partial changes suggested by the model and then manually introduce two new variables at line 209, as shown in subfigure (d). Coeditor can then leverage these new variables to suggest the correct changes needed to fix the errors.

This iterative approach enables Coeditor to adapt and refine its suggestions based on additional user edits, providing a more efficient and flexible code editing experience compared to existing code completion techniques. By incorporating the editing history into the prediction context, Coeditor demonstrates its potential to assist developers in a wide range of code editing tasks, from simple modifications and refactoring to more complex codebase-wide updates.

[1]TypeT5 would produce the same result even if `pack_batch` were defined far away or in a different file, as Coeditor tracks all changes Δ_i the user has made since the last commit and incorporates them into the prediction context.

4.3 Methods

Recall from the introduction that we wish to model the distribution $P(\Delta u \mid \Delta_k \dots \Delta_1, U)$. To this end, we first describe how to encode the target change Δu and contextual changes $\Delta_1 \dots \Delta_k$ (subsection 4.3.1). We then describe how to form the context from the codebase U using function signatures (subsection 4.3.2). These choices naturally lead to a model compatible with fine-tuning CodeT5, which was pre-trained on the masked span infilling task (subsection 4.3.3). Finally, we describe our new dataset that is used to fine-tune this model (subsection 4.3.5).

4.3.1 Representing Code Changes

A suitable format is required to map code changes into token sequences that can be processed by a seq2seq transformer language model. In our setting, we want to select a format that encodes and decodes code changes in a uniform manner while minimizing the number of tokens the model needs to produce. Hence, we adopt a line-diff-based format, enabling us to convert auto-editing into a masked span infilling problem (Wang et al., 2021).[2]

Consider a block of code u to be made up of lines $l_1, \dots, l_m$ and a user-specified edit region that spans between line a and $a+n$, where $1 \leq a \leq a+n \leq m$. Moreover, each line is associated with a status variable s_i indicating what

[2]Prior work has proposed various methods to produce code changes. e.g., Zhang et al. (2022) learns the distribution $P(u' \mid u)$ and Reid and Neubig (2022) tags each input token with a label indicating deletion, insertion, or replacement. However, these methods require more copying or tagging, resulting in longer output sequences compared to our approach.

type of change (if any) has already been made; $s_i \in \{\texttt{(empty)}, \texttt{<add>}, \texttt{<del>}\}$.[3] We represent the input code by a function EncInput that (optionally) prepends status tokens $s_1 \ldots s_m$ and placeholder tokens $\texttt{<1>} \ldots \texttt{<n>}$ at the start of each line:

$$\text{EncInput}(u) = s_1 l_1 s_2 l_2 \ldots \texttt{<1>} s_a l_a \texttt{<2>} s_{a+1} l_{a+1} \ldots \texttt{<n>} s_{a+n} l_{a+n} \ldots s_m l_m \ .$$

For contextual changes $\Delta_1 \ldots \Delta_k$, we can represent them using the same format but with an empty edit region. When the target change Δu contains line additions, denoting the jth line to be inserted before line i as l'_{ij}, we can represent Δu using the following expression,

$$\begin{aligned} \text{EncOutput}(\Delta u) &= \texttt{<1>} I_a D_a \texttt{<2>} I_{a+1} D_{a+1} \ \ldots \ \texttt{<n>} I_{a+n} D_{a+n} \ , \\ \text{where } I_i &= \texttt{<add>} \ l'_{i1} \texttt{<add>} \ l'_{i2} \ \ldots \ \texttt{<add>} \ l'_{i|I_i|} \ , \\ D_i &= \text{if } l_i \textit{ is to be deleted} \text{ then } \texttt{<del>} \text{ else } \texttt{(empty)} \ . \end{aligned}$$

Note that we add a further restriction that forbids D_i from being `<del>` if s_i is `<add>` in order to prevent the model from modifying a line that has just been added; we discuss this in more detail in **??**. Figure 4.3 illustrates this line-diff-based encoding scheme using the example from Figure 4.2. This format ensures that if we replace the placeholder tokens in the input with the corresponding changes specified in the output sequence, we obtain the total change that combines u and Δu.

[3] We represent edits as line diffs output by `Differ.compare` using the standard `difflib` library.

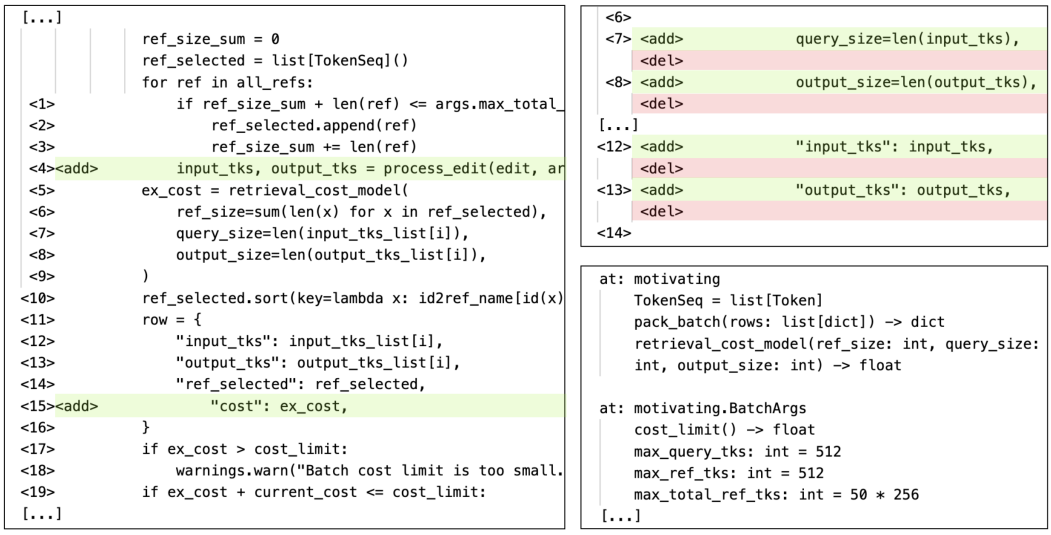

Figure 4.3: Coeditor encoding format. (Left) the input sequence adds placeholder tokens to indicate code region to edit. (Top right) the output sequence specifies further changes at each placeholder token. (Bottom right) relevant signatures are retrieved from the codebase and added to the context. (In this example, the Python module is called `motivating`).

4.3.2 Analyzing Relevant Signatures

Having described how we represent code changes, we must also establish a method for feeding U, the remaining codebase, to the model. Simply inputting the entire codebase as is would result in an excessive number of tokens, overwhelming the context. Instead, inspired by the ideas proposed in previous type inference work (Pradel et al., 2020, Wei et al., 2020, 2023), we employ lightweight static analysis to extract the most relevant information into the context, as outlined below.

For each target code region u, we analyze its pre-edit code and generate a list of its usages.[4] In the case of a function usage, we retrieve its function signature; for a variable or class member usage, we retrieve the first statement in which it was assigned. We then concatenate all these usages into a single "document", as shown at the bottom right of Figure 4.3, which serves as additional input context. This approach allows the model to access the most pertinent information about the current code region and significantly improves model performance (Table 4.5), while generating only a small number of extra tokens in the context (Table 4.2).

4.3.3 Adapting CodeT5

Our model is based on the architecture and pre-trained weights of CodeT5 (Wang et al., 2021). CodeT5 was pre-trained on a large corpus of code

[4]We use the Jedi package for this purpose: `https://github.com/davidhalter/jedi`.

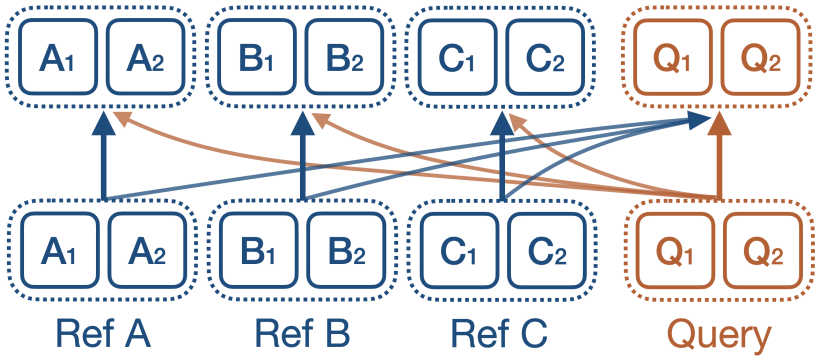

Figure 4.4: Coeditor encoder sparse attention pattern. All attention between the reference blocks are skipped to avoid the quadratic cost of dense attention.

data using the masked span infilling objective, making it a suitable choice for our problem. We employ the CodeT5-base model, containing 220M parameters, and fine-tune it for our code auto-editing setting. Although the original CodeT5 model was pre-trained with a small sequence length of 512, its relative positional encoding scheme allows us to fine-tune it on much longer sequences for our problem.

Considering that a single commit may contain numerous code changes, concatenating all changes into a single input can lead to long token sequences that are difficult for the CodeT5 model to process with dense attention. To mitigate this issue, we replace the full attention in its encoder with a block-sparse attention pattern, illustrated in Figure 4.4. This pattern divides the input sequences into multiple reference blocks and a query block. The query block contains the code to be edited, whereas each reference block encodes a contextual unit change Δu_j or a chunk of the signature document. We limit the sequence length of each block to 512 tokens for references and 1024 for the

query, dividing longer blocks into multiple ones if necessary. The self-attention within each block is performed as usual, but the attention between different reference blocks is skipped to save computation, similar to other retrieval-augmented models (Izacard and Grave, 2021). However, we still allow the query block to attend to and be attended by all reference blocks (a global attention block (Beltagy et al., 2020, Zaheer et al., 2020)). We also set the relative distance between each reference and the query to be infinite when computing the relative positional encoding, making the model is insensitive to the ordering of the references. We are able to use a total of 16.4K reference tokens at test time, which is sufficient to cover 88.8% of problem instances in our test set without truncating the context (Table 4.2).

4.3.4 Discussion of Sparse Attention Mechanisms

The block-sparse attention pattern we described in subsection 4.3.3 follows past work on retrieve-and-read models for natural language question answering. Specifically, it resembles Fusion-in-Decoder (Izacard and Grave, 2021) with three changes. First, we have no notion of a question that is jointly encoded with each retrieved snippet. Second, our target code block u is given special status in the encoder and can globally attend to each retrieved snippet. Third, we modify the relative positional encoding to make each query "infinitely" far from the reference.

Our approach also resembles Longformer (Beltagy et al., 2020) or Big-Bird (Zaheer et al., 2020), most notably in how our query block's cross-

attention with the references can be viewed as an instance of global attention. However, our segments do not come from a coherent context, so our local attention component is a block-diagonal sparse matrix rather than a sliding window as in those methods. Our modification to relative position encoding also makes our model invariant to the ordering of references, helping it generalize to different editing orders at inference time.

4.3.5 The PyCommits Dataset

To train our model, we gather real-world code changes from the commit histories of open-source Python projects, a dataset we call PyCommits. For each commit, we first identify which changes are made to the same code unit (a unit can be either a function, a region of a class, or a region of a module) and subsequently separate the commit into a list of unit additions, unit deletions, or unit modifications. As our work primarily focuses on code editing, only unit modifications are used as training labels, while the other two types of changes remain visible to the model as context.

For each unit modification, we create a training problem instance that instructs the model to predict the code change based on all prior (but not future) changes from the same commit. Git does not record the editing order of changes within the same commit, so we employ a simple heuristic that sorts unit changes according to their source code locations and the import order between modules. Specifically, we assume that units within the same file are modified from top to bottom, and if a module imports another module, changes

Table 4.1: General statistics of the PyCommits dataset.

	train	valid	test
projects	1550	50	50
used commits	217K	5006	5854
modified files	501K	10.1K	11.1K
modified functions	958K	20.1K	22.5K
modified lines	7.10M	143K	169K

Table 4.2: Additional statistics specific to our technique, computed over the test set.

	definition	median	mean	max	$\geq$ max
query tokens	EncInput(u)	258	361.9	1024	7.8%
output tokens	EncOutput(Δu)	60	89.7	512	1.3%
prev change tokens	EncInput($\Delta_1 \dots \Delta_k$)	1625	4.14K	16.4K	11.2%
signature tokens	$\{\text{signature}(v)\}_{v \in \text{usages}(u)}$	313	515.5	15.9K	0.0%

in the imported module occur before those in the importing module.[5] To train our model for the proposed multi-round editing setting, we generate synthetic data demonstrating repeated editing to the same code unit as follows: for those code change involving least two changed lines, we randomly sample a subset of the changes as the prediction target and line the remaining changes into the input. For example, the problem instance shown in Figure 4.3 can be generated by inlining 2 of the 6 changed lines in the input.

We construct a new code editing dataset using the commit history of 1,650 Python projects with permissive licenses (MIT, Apache, and BSD) sourced from GitHub. We use 50 of the projects for testing and 50 for val-

[5]Note that this ordering mainly affects how we generate the training and testing data. At test time, our model can condition on changes both above and below the edit region.

idation and use the remaining 1,550 projects for training. We use at most 1000 commits per project per project to ensure that the model is trained on a diverse set of code changes. We show the general statistics in Table 4.1 and the statictics that are specific to our technique in Table 4.2. Tokenization is performed using the CodeT5 tokenizer.

4.4 Evaluation

In this section, we first compare Coeditor with prior code completion approaches on a simplified version of the editing task. We then report Coeditor's performance on the proposed multi-round editing task and conduct ablation studies. Example model outputs are included in the end of this chapter.

Training Setup We initialize Coeditor with the CodeT5-base checkpoint (220M parameters) and train the model on our training set for 1.75 epoch, gradually increasing the model reference context size from 2048 tokens to 4096 tokens (at epoch 1) and then to 8192 tokens (at epoch 1.5). We use Huggingface's Trainer implementation and the AdamW optimizer, with a linear learning rate schedule with a starting learning rate of 2e-5 and 0.01 weight decay. We train the model with a fixed batch size of 1 and a total of 1.34 million training steps. Training took about 5 days on a single NVIDIA Quadro RTX 8000 GPU with 48 GB memory.

4.4.1 Comparison with Code Completion Approaches

Baselines We compare Coeditor with 3 open-source code generation models: InCoder-1B, InCoder-6B (Fried et al., 2022), and SantaCoder (Allal et al., 2023). All three code generation models are trained with the Fill-in-the-middle pre-training objective (Aghajanyan et al., 2022) and use a context size of 2048 tokens.

Creating test instances We generate code completion problem instances from real commits as follows. For each code change in PyCommits, we take the last changed line as the completion target. If the last change is a modification, we delete the modified line and let the model fill in the new version of the line. If the last change is a deletion, we simply discard the change. We then inline all changes before the target into the prediction context. This inlining process is implemented differently for each model: for our Coeditor model, the inlined changes are visible to our model following the encoding scheme described in subsection 4.3.1; for the code completion models, we simply apply the inlined changes to the original code and use the resulting state as the model input. Also note that while our model constructs its prediction context using relevant changes and static analysis (as described in subsection 4.3.2), the code completion models (which are unaware of code changes) only use the code surrounding the completion target as the prediction context. We call this test dataset, derived from our PyCommits test set, PyCommits-OneLine.

Table 4.3: Performance on 5000 code completion instances extracted from edits (PyCommits-OneLine). Add EM and Replace EM are the (enhanced) exact-match accuracies on addition and replacement change, respectively.

Model	Parameters	Add EM (%)	Replace EM (%)	Overall EM (%)
InCoder1B	1.3B	29.0	25.2	26.2
InCoder6B	6.7B	34.0	30.4	31.3
SantaCoder	1.1B	31.0	28.1	28.8
Coeditor	220M	**47.1**	**64.9**	**60.4**

Results We report the performance (without fine-tuning on this task) of all approaches in Table 4.3. We use an enhanced exact-match (EM) accuracy metric that performs semantic-preserving code normalization before checking for string equivalence.[6] The results are measured on 5000 code completion problems sampled from our test set. We see that Coeditor significantly outperforms the other code generation models for both addition and replacement changes. Coeditor achieves an overall EM of 60.4%, which is almost twice as high as the best performing code completion model (31.3%), despite using a 30 times smaller model, demonstrating the significant benefits of incorporating editing history for code completion. We also include 3 example model outputs on this task in section 4.6.

[6]We normalize Python code by (1) parsing the code into a syntax tree using the `ast` library, (2) removing any comments and docstrings, (3) sorting all keyword arguments in function calls, and (4) un-parsing the syntax tree.

4.4.2 Multi-round Editing

This evaluation focuses on the editing assistant use case where we assume the user has some desired code changes in mind, and we aim to evaluate how much the model can save the user's effort by automating as much changes as possible, potentially under the guidance of the user. The user can accept partial changes suggested by the model and make additional changes manually if needed.

Evaluation workflow To evaluate the above use case automatically, we use the ground-truth code changes to simulate the user's actions. In particular, when the model predicts a list of changes, we compare the predicted changes against the ground truth changes line-by-line and accept any line change that exactly matches the ground truth. If none of the suggested changes match the ground truth, we assume the user will manually perform the first remaining change. In both cases, after the additional changes, we rerun the model to obtain new suggestions and repeat until all desired changes have been performed or the round limit = 6 has been reached. In the end, we compute the total gain using the difference between the editing cost of the ground truth and the accumulative editing cost of all manually performed edits.

Measuring editing cost There are multiple ways to measure the cost of performing a code change. Since there is no consensus on the best metric, we report 3 metrics in our results. Prior work (Lavazza et al., 2023) suggests that

for code understanding tasks, simple line counts-based metrics are almost as good as more complex metrics, hence our first metric, **Lines**, simply measures the number of changed lines before and after the edit. We also report **Levenshtein**, the classic Levenshtein editing distance metric that measures the minimal number of character addition, deletion, and substitution needed to transform one string into another. Although simple, the Levenshtein distance doesn't model many important aspects of code editing, such as the cost associated with cursor movement, and it also under-count the cost of substitution and over-count the cost of large deletions. Hence, we propose an additional metric, **Keystrokes**, that aims to better approximate the number of needed user keystrokes than Levenshtein by allowing for batch deletion and accounting for the cost of cursor movements. We describe this metric in detail below. Note that the total gain can be negative when measured with Levenshtein and Keystrokes.[7]

Keystroke Distance We developed a string distance metric incorporating the cost of cursor movement, approximating the number of keystrokes needed to transform an input string into an output string.

Given the initial state with `i=len(input)`, `j=len(output)`, `cursor_dis=init_cursor_dis`, and `deleting=False`, the cost is calculated using dynamic programming with the optimal combination of the following operations:

[7]For example, the Levenshtein distance of modifying a sentence is lower than the total of first deleting the sentence and then adding a new one.

- M: Match character (cost=0), requires `input[-i] == output[-j]` and `not deleting`, results in `i -= 1`, `j -= 1`, and `cursor_dis += 1`.
- D: Delete input character (cost=1), requires `cursor_dis == 0` and `not deleting`, results in `i -= 1`.
- A: Add output character (cost=1), requires `cursor_dis == 0` and `not deleting`, results in `j -= 1`.
- C: Move cursor to current position (cost=min(`cursor_dis`, `jump_cost`)), requires no conditions, results in `cursor_dis = 0`.
- S: Begin deletion (cost=1), requires `cursor_dis == 0` and `not deleting`, results in `deleting = True`.
- K: Continue deletion (cost=0), requires `deleting`, results in `i -= 1`.
- E: End deletion (cost=1), requires `cursor_dis == 0` and `deleting`, results in `deleting = False`.

Where `jump_cost` is a constant that we set to 4 when reporting our results. Note that this model does not consider copying and pasting operations. The worst-case complexity of this algorithm is $O($`len(input)` $\times$ `len(output)` $\times$ `jump_cost`$)$.

Results We report the evaluation results on 5000 problems sampled from our test set in Table 4.4, and we also report the single-round performance for

Table 4.4: Multi-round evaluation results measured on 5000 problems from the PyCommits test set. Lines, Levenshtein, and Keystrokes are the average total gains in the corresponding metrics. Rounds is the average number of rounds needed to complete all desired changes.

Setting	Lines (%)	Levenshtein (%)	Keystrokes (%)	Rounds
SingleRound	28.5	23.1	19.2	1
MultiRound	46.7	25.9	28.6	2.43

reference. We see that Coeditor achieves much larger total gains under the multi-round setting, especially when measured with the Lines and Keystrokes metric (which we believe more accurately captures the user editing effort than Levenshtein). We also show 3 examples of the model's suggestions in section 4.7.

4.4.3 Ablation Studies

We retrain the model with various components disabled to study their impact on the overall model performance. We report the (single-round) exact match performance of each variation on the entire PyCommits validation set in Table 4.5. The results show that removing any of the components leads to a decrease in performance, highlighting the importance of each component in the overall model. Specifically, when we remove the explicit feeding of code changes (No Diffs), the EM drops the most, from 42.1% to 26.1%. When we disable the static analysis component (No Signatures), the EM decreases to 33.3%. Using a smaller limit of reference tokens impacts the model performance the least, reducing EM to 39.8%. All results reported in Table 4.5 were obtained

Table 4.5: Ablation results on the entire validation set (PyCommits). All pairwise differences are statistically significant with $p < 0.05$ using a paired bootstrap test.

Ablation	Description	EM (%)
No Diffs	Feeding the same input to the model except that all changes are replaced with their post-edit results alone.	26.1
No Signatures	Disabling the static analysis component and removing function and class signatures from the prediction context.	33.3
Small Context	Reducing the max number of reference tokens from 16K to 2048.	39.8
No Ablation	Model trained with our default settings.	**42.1**

by training the model for half amount of training steps to save compute.

4.5 Related Work

The past work most similar to our setting is that of Brody et al. (2020), which also targets a contextual code editing setting. However, it can only predict a restrictive set of code changes expressable as moving, deleting, or copying existing AST nodes and cannot generate novel expressions that are not present in the input. It also doesn't make use of modern transformer architecture or pre-training techniques. In contrast, Ni et al. (2021) takes a rule-based approach, using program synthesis methods to distill similar change patterns in the context and make editing suggestions accordingly.

There is also prior work on non-contextual code change prediction settings. In Chakraborty et al. (2020), the authors use past code patches to train the model to perform similar edits and evaluate it on future edits in the same

codebase. However, since the model does not condition on relevant changes, their technique requires retraining the model for new types of edit patterns. Panthaplackel et al. (2020a) proposes augmenting the decoder with a direct copying mechanism to help a encoder-decoder model perform editing tasks. Zhang et al. (2022) proposes a de-noising pre-training scheme, in which they randomly corrupt actual code snippets and train the model to predict the uncorrupted version from the corrupted version. Tufano et al. (2021) focuses on predicting code review changes using developer discussions. Reid and Neubig (2022) focuses on modeling the iterative editing process of texts and code and proposes a different change encoding scheme that represents edits at the word-level based on the Levenshtein algorithm.

Lastly, there is prior work that focuses on on learning to update code comments (Panthaplackel et al., 2020b) or generating natural language descriptions (Panthaplackel et al., 2022) from code changes. This work, we focus on measuring our model's ability to make correct code changes and remove comments and doc-strings before measuring the exact accuracy.

4.6 Code Completion Examples

To help the reader see why including contextual changes can be beneficial for (editing-related) code completion problems, we compare Coeditor and InCoder6B's outputs on 3 example problems from our test set in the next few pages (Figure 4.5–Figure 4.10). These examples are sampled from a subset that are small enough to be presentable within one or two pages and in which

Coeditor outperforms InCoder6B.

4.7 Multi-round Editing Examples

We show 3 multi-round editing examples from our test set in Figure 4.11–Figure 4.15. These examples are sampled from a subset that are small enough to be presentable within two pages and in which Coeditor achieved 50–100 total keystrokes edit gain.

```
--------------------------------------------------------------------------------
      # module: anyio._core._testing
      def get_running_tasks() -> list[TaskInfo]:
          """
          Return a list of running tasks in the current event loop.

          :return: a list of task info objects

          """
    -     return get_asynclib().get_running_tasks()
     <infill here>

==================================
Ground Truth:
  return get_async_backend().get_running_tasks()
Coeditor Prediction:
  return get_async_backend().get_running_tasks()
Incoder Prediction:
  return get_asynclib().get_running_tasks()
```

Figure 4.5: Code completion example 1. Coeditor sees from the relevant contextual changes (shown in Figure 4.6) that some `get_asynclib()` calls should be replaced with `get_async_backend()`, so it correctly suggested the change based on the deletion before the infilling point. InCoder was not able to see the deletion and infilled the original code given only the surrounding code.

```
===========changed ref 0===========
    # module: anyio._core._testing
    def get_current_task() -> TaskInfo:
        """
        Return the current task.

        :return: a representation of the current task

        """
  +     return get_async_backend().get_current_task()
  -     return get_asynclib().get_current_task()

===========changed ref 1===========
    # module: anyio.from_thread
    class BlockingPortal:
        def __new__(cls) -> BlockingPortal:
  +         return get_async_backend().create_blocking_portal()
  -         return get_asynclib().BlockingPortal()

===========changed ref 2===========
    # module: anyio._core._eventloop
    def get_cancelled_exc_class() -> type[BaseException]:
        """Return the current async library's cancellation exception class."""
  +     return get_async_backend().cancelled_exception_class()
  -     return get_asynclib().CancelledError
```

Figure 4.6: Code completion example 1: relevant contexts. The changes highlighted in orange tell Coeditor that some `get_asynclib()` calls should be replaced with `get_async_backend()`.

```
--------------------------------------------------------------------------------
    # module: instructor.oracle_data.seqgan_instructor
    class SeqGANInstructor(BasicInstructor):
        def __init__(self, opt):
            super(SeqGANInstructor, self).__init__(opt)

            # generator, discriminator
            self.gen = SeqGAN_G(cfg.gen_embed_dim, cfg.gen_hidden_dim,
                                cfg.vocab_size, cfg.max_seq_len,
                                cfg.padding_idx, cfg.temperature, gpu=cfg.CUDA)
            self.dis = SeqGAN_D(cfg.dis_embed_dim, cfg.vocab_size,
                                cfg.padding_idx, gpu=cfg.CUDA)
            self.init_model()

            # Optimizer
            self.gen_opt = optim.Adam(self.gen.parameters(), lr=cfg.gen_lr)
            self.dis_opt = optim.Adam(self.dis.parameters(), lr=cfg.dis_lr)

            # Criterion
            self.mle_criterion = nn.NLLLoss()
            self.dis_criterion = nn.CrossEntropyLoss()

            # DataLoader
  +         self.oracle_samples = torch.load(cfg.oracle_samples_path)
    <infill here>
            self.gen_data = GenDataIter(self.gen.sample(cfg.batch_size, cfg.batch_size))

================================
Ground Truth:
  self.oracle_data = GenDataIter(self.oracle_samples)
Coeditor Prediction:
  self.oracle_data = GenDataIter(self.oracle_samples)
Incoder Prediction:
  self.oracle = Oracle(self.oracle_samples)
```

Figure 4.7: Code completion example 2. Coeditor was able to suggest the correct code based on a similar change from another file (Figure 4.8, highlighted in orange), whereas InCoder was not able to see the change and suggested a wrong statement.

```
===========changed ref 19===========
    # module: instructor.oracle_data.relgan_instructor
    class RelGANInstructor(BasicInstructor):
        def __init__(self, opt):
            super(RelGANInstructor, self).__init__(opt)

            # generator, discriminator
            self.gen = RelGAN_G(cfg.mem_slots, cfg.num_heads, ↩
cfg.head_size, cfg.gen_embed_dim, cfg.gen_hidden_dim,
                                cfg.vocab_size, cfg.max_seq_len, ↩
cfg.padding_idx, gpu=cfg.CUDA)
            self.dis = RelGAN_D(cfg.dis_embed_dim, cfg.max_seq_len, ↩
cfg.num_rep, cfg.vocab_size, cfg.padding_idx,
                                gpu=cfg.CUDA)

            self.init_model()

            # Optimizer
            self.gen_opt = optim.Adam(self.gen.parameters(), lr=cfg.gen_lr)
            self.gen_adv_opt = optim.Adam(self.gen.parameters(), lr=cfg.gen_adv_lr)
            self.dis_opt = optim.Adam(self.dis.parameters(), lr=cfg.dis_lr)

            # Criterion
            self.mle_criterion = nn.NLLLoss()

            # DataLoader
  +         self.oracle_samples = torch.load(cfg.oracle_samples_path)
  +         self.oracle_data = GenDataIter(self.oracle_samples)
            self.gen_data = GenDataIter(self.gen.sample(cfg.batch_size, cfg.batch_size))
```

Figure 4.8: Code completion example 2: relevant contexts.

```
--------------------------------------------------------------------------------
def get_config(dataset: str,
               num_states: Optional[int] = None,
               shots: Optional[int] = 0,
               with_bow: Optional[bool] = True,
               encoder_embedding_type: Optional[str] = model_config.GLOVE_EMBED,
               decoder_embedding_type: Optional[str] = model_config.GLOVE_EMBED,
               shared_embedding: Optional[bool] = False,
               bert_embedding_type: Optional[str] = 'base',
               bert_dir: Optional[str] = '') -> config_dict.ConfigDict:
  """Returns the configuration for this experiment.
  […]
  """
  config = config_dict.ConfigDict()
  config_dir = get_config_dir(dataset)

  # config.max_per_task_failures = -1
  # config.max_task_failures = 10

  config.platform = 'jf'
  config.tpu_topology = '2x2'

  config.seed = 8

  config.dataset = dataset
  config.dataset_dir = data_utils.get_dataset_dir(dataset)
<infill here>
  config.train_epochs = 10
  config.train_batch_size = 16
  config.eval_batch_size = 16
  # Batch size for inference. Predicting in batches in case of OOM.
  config.inference_batch_size = 300
  # Seed used to generate datasets for inference.
  config.inference_seed = 9527
  # Directory storing the saved model and model prediction outputs.
  config.model_base_dir = None
  # Directory or checkpoint to initalize the model from. The initialize priority
  # will be:
  # -init_checkpoint
  # -latest checkpoint in init_dir
  # -latest checkpoint in output_dir
  config.init_checkpoint = None
  config.init_dir = None
  […]

==================================
Ground Truth:
  config.domain_adaptation = False
Coeditor Prediction:
  config.domain_adaptation = False
Incoder Prediction:
  config.train_split = 'train'
```

Figure 4.9: Code completion example 3. Coeditor was able to suggest adding the correct attribute initialization based on the new usage highlighted in Figure 4.10, whereas InCoder was not able to see the new usages and hallucinated a new attribute.

```
==========changed ref 15==========
    # module: experimental.language_structure.vrnn.train
    def run_experiment(config: config_dict.ConfigDict, output_dir: str):
    # offset: 1
    <s>model_dir, 'labeled_dialog_turn_ids.txt'), 'w') as f:
            f.write('\n'.join(
                str(id) for id in labeled_dialog_turn_ids.numpy().tolist()))
      else:
        labeled_dialog_turn_ids = None

  +   if config.domain_adaptation:
  +     inputs = preprocessor.get_full_dataset_outputs(train_dataset_builder)
  +     # Notice domain label id 0 is also treated as in-domain: ood should have
  +     # a different id from it.
  +     in_domains, _ = tf.unique(tf.reshape(inputs[_DOMAIN_LABEL_NAME], [-1]))
  +     metric_namespaces = [
  +         _metric_namespace(_TRAIN),
  +         _metric_namespace(_TEST, True),
  +         _metric_namespace(_TEST, False)
  +     ]
  +     fewshot_metric_namespaces = [
  +         _metric_namespace(_FEWSHOT_NAMESPACE, True),
  +         _metric_namespace(_FEWSHOT_NAMESPACE, False)
  +     ]
  +   else:
  +     in_domains = None
  +     metric_namespaces = [_metric_namespace(split) for split in _SPLITS]
  +     fewshot_metric_namespaces = [_metric_namespace(_FEWSHOT_NAMESPACE)]
  +
  +   data_preprocessor = _build_data_processor(config, labeled_dialog_turn_ids,
  +                                             in_domains)
  -   data_preprocessor = _build_data_processor(config, labeled_dialog_turn_ids)
      preprocess_fn = data_preprocessor.create_feature_and_label

      # Load PSL configs
      psl_learning = config.psl_constraint_learning_weight > 0
      psl_inference = config.psl_constraint_inference_weight > 0
      if psl_learning or psl_inference:
        with tf.io.gfile.GFile(
            config.model.vae_cell.decoder_embedding.vocab_file_path, 'r') as f:
          vocab = f.read()[:-1].split('\n')
        preprocess_fn = psl_utils.psl_feature_mixin(preprocess_fn, config.dataset,
                                                    config.psl, vocab)

      # Load datasets
      # TODO(yquan): invesigate why distributed training fails in *fish TPU
      # Failure example: https://xm2a.corp.google.com/experiments/33275459
      distributed_training = False
      train_dataset = preprocessor.create_dataset(train_dataset_builder,
```

Figure 4.10: Code completion example 3: relevant contexts.

```
project: google/uncertainty-baselines
commit: '3e74384539 Adding support for tb.dev h…'
path: uncertainty_baselines.experiments.deterministic.eval/run_eval_epoch
{'n_references': 6, 'changed_reference_tks': 3247, 'unchanged_reference_tks': 37}
--------------------------------------------------------------------------------
editing round: 1
========Ground Truth========
 <1>:    +   val_outputs_np = None
 <5>:    +       if hparams:
         +         hp.hparams(hparams)
<13>:    +     if hparams:
         +       hp.hparams(hparams)
<15>:    +   return val_outputs_np, {k: v.numpy() for k, v in test_outputs.items()}

========Main Code========
      <s>baselines.experiments.deterministic.eval
      def run_eval_epoch(
          val_fn: EvalStepFn,
          val_dataset: tf.data.Dataset,
          val_summary_writer: tf.summary.SummaryWriter,
          test_fn: EvalStepFn,
          test_dataset: tf.data.Dataset,
          test_summary_writer: tf.summary.SummaryWriter,
    +     current_step: int,
    -     current_step: int):
    +     hparams: Optional[Dict[str, Any]]):
 <0>    """Run one evaluation epoch on the test and optionally validation splits."""
 <1>    if val_dataset:
 <2>      val_iterator = iter(val_dataset)
 <3>      val_outputs = val_fn(val_iterator)
 <4>      with val_summary_writer.as_default():
 <5>        for name, metric in val_outputs.items():
 <6>          tf.summary.scalar(name, metric, step=current_step)
 <7>      val_outputs_np = {k: v.numpy() for k, v in val_outputs.items()}
 <8>      logging.info(
 <9>          'Validation metrics for step %d: %s', current_step, val_outputs_np)
<10>    test_iterator = iter(test_dataset)
<11>    test_outputs = test_fn(test_iterator)
<12>    with test_summary_writer.as_default():
<13>      for name, metric in test_outputs.items():
<14>        tf.summary.scalar(name, metric, step=current_step)
<15>

========Predicted Changes========
 <5>:    +       if hparams:
         +         hp.hparams(hparams)
<13>:    +     if hparams:
         +       hp.hparams(hparams)

========Accepted gains========
keystrokes gain: 94
diff-lines gain: 4
levenshtein gain: 88
```

Figure 4.11: Multi-round editing example 1. Coeditor correctly suggested a subset of the ground-truth changes. Contextual changes omitted for this example.

```
project: facebookresearch~pytorchvideo
commit: 'b0ae784244 Get audio samples only from…'
path: pytorchvideo.data.video/VideoPathHandler.video_from_path
{'n_references': 1, 'changed_reference_tks': 0, 'unchanged_reference_tks': 167}
--------------------------------------------------------------------------------
editing round: 3
========Ground Truth========
 <0>:    +                 decode_video=decode_video,
         +                 decode_audio=decode_audio,
         +                 decoder=decoder,
         +             )
         -             return EncodedVideo.from_path(filepath, decode_audio, decoder)

========Main Code========
      # module: pytorchvideo.data.video
      class VideoPathHandler(object):
    +     def video_from_path(
    +         self, filepath, decode_video=True, decode_audio=False, decoder="pyav", fps=30
    +     ):
    -     def video_from_path(self, filepath, decode_audio=False, decoder="pyav", fps=30):
              try:
                  is_file = g_pathmgr.isfile(filepath)
                  is_dir = g_pathmgr.isdir(filepath)
              except NotImplementedError:

                  # Not all PathManager handlers support is{file,dir} functions,
                  # when this is the case, we default to assuming the path is a file.
                  is_file = True
                  is_dir = False

              if is_file:
                  from pytorchvideo.data.encoded_video import EncodedVideo

    +             return EncodedVideo.from_path(
    +                 filepath,
 <0>              return EncodedVideo.from_path(filepath, decode_audio, decoder)
 <1>          elif is_dir:
 <2>              from pytorchvideo.data.frame_video import FrameVideo
 <3>
 <4>              assert not decode_audio, "decode_audio must be False when using FrameVideo"
 <5>              return FrameVideo.from_directory(
 <6>                  filepath, fps, path_order_cache=self.path_order_cache
 <7>              )
 <8>          else:
 <9>              raise FileNotFoundError(f"{filepath} not found.")
<10>

========Predicted Changes========
 <0>:    +                 decode_audio,
         +                 decoder,
         +                 fps,
         +                 video_frame_to_path_fn=self.video_frame_to_path_fn,
         +             )
         -             return EncodedVideo.from_path(filepath, decode_audio, decoder)
```

Figure 4.12: Multi-round editing example 2 (round 3). Coeditor misunderstood the user's intention and suggested adding two more arguments to the `EncodedVideo.from_path` function call. Under our multi-round evaluation strategy, we assume the user would then manually add the next line from the ground truth changes (see the next figure).

```
editing round:4
========Ground Truth========
 <0>:    +                 decode_audio=decode_audio,
         +                 decoder=decoder,
         +             )
         -             return EncodedVideo.from_path(filepath, decode_audio, decoder)

========Main Code========
      # module: pytorchvideo.data.video
      class VideoPathHandler(object):
    +     def video_from_path(
    +         self, filepath, decode_video=True, decode_audio=False, decoder="pyav", fps=30
    +     ):
    -     def video_from_path(self, filepath, decode_audio=False, decoder="pyav", fps=30):
              try:
                  is_file = g_pathmgr.isfile(filepath)
                  is_dir = g_pathmgr.isdir(filepath)
              except NotImplementedError:

                  # Not all PathManager handlers support is{file,dir} functions, when this is the
                  # case, we default to assuming the path is a file.
                  is_file = True
                  is_dir = False

              if is_file:
                  from pytorchvideo.data.encoded_video import EncodedVideo

    +             return EncodedVideo.from_path(
    +                 filepath,
    +                 decode_video=decode_video,
 <0>              return EncodedVideo.from_path(filepath, decode_audio, decoder)
 <1>          elif is_dir:
 <2>              from pytorchvideo.data.frame_video import FrameVideo
 <3>
 <4>              assert not decode_audio, "decode_audio must be False when using FrameVideo"
 <5>              return FrameVideo.from_directory(
 <6>                  filepath, fps, path_order_cache=self.path_order_cache
 <7>              )
 <8>          else:
 <9>              raise FileNotFoundError(f"{filepath} not found.")
<10>

========Predicted Changes========
 <0>:    +                 decode_audio=decode_audio,
         +                 decoder=decoder,
         +             )
         -             return EncodedVideo.from_path(filepath, decode_audio, decoder)
========Accepted gains========
keystrokes gain: 50
diff-lines gain: 4
levenshtein gain: 47
```

Figure 4.13: Multi-round editing example 2 (round 4). With the next line change from the ground truth added, Coeditor understood that the user intended to only change the calling style and was thus able to suggest the correct change.

```
project: qiandao-today~qiandao
commit: [8d0f0e04df Cursor操作结束后自动关闭; 统一DB连接操作]
path: db.basedb/BaseDB._insert
{'n_references': 5, 'changed_reference_tks': 957, 'unchanged_reference_tks': 168}
--------------------------------------------------------------------------------
editing round: 1
========Ground Truth========
<13>:    +         lastrowid = dbcur.lastrowid
         +         dbcur.close()
         +         return lastrowid
         -         return dbcur.lastrowid

========Main Code========
      # module: db.basedb


      class BaseDB(object):

          def _insert(self, tablename=None, **values):
 <0>          tablename = self.escape(tablename or self.__tablename__)
 <1>          if values:
 <2>              _keys = ", ".join((self.escape(k) for k in values.keys()))
 <3>              _values = ", ".join([self.placeholder, ] * len(values))
 <4>              sql_query = "INSERT INTO %s (%s) VALUES (%s)" % (tablename, _keys, _values)
 <5>          else:
 <6>              sql_query = "INSERT INTO %s DEFAULT VALUES" % tablename
 <7>          logger.debug("<sql: %s>", sql_query)
 <8>
 <9>          if values:
<10>              dbcur = self._execute(sql_query, list(values.values()))
<11>          else:
<12>              dbcur = self._execute(sql_query)
<13>          return dbcur.lastrowid
<14>


========Predicted Changes========
<13>:    +         lastrowid = dbcur.lastrowid
         +         dbcur.close()
         +         return lastrowid
         -         return dbcur.lastrowid


========Accepted gains========
keystrokes: 50
diff-lines: 4
levenshtein: 47
```

Figure 4.14: Multi-round editing example 3. Coeditor was able to predict the correct change in the first editing round by identifying a similar change inside a different function (see Figure 4.15, highlighted in orange).

```
===========signatures ref 0===========
    at: db.basedb
        logger = logging.getLogger('qiandao.basedb')

    at: db.basedb.BaseDB
        placeholder = "%s" # mysql

        _execute(sql_query, values=[])
        _execute(self, sql_query, values=[])

    at: db.basedb.BaseDB._replace
        tablename = self.escape(tablename or self.__tablename__)
[More sinagures …]

===========changed ref 0===========
    # module: db.basedb


    class BaseDB(object):
  +     # placeholder = '?' # sqlite3
  +
  +     def __init__(self, host=config.mysql.host, port=config.mysql.port,
  +             database=config.mysql.database, user=config.mysql.user, passwd=↩
config.mysql.passwd, auth_plugin=config.mysql.auth_plugin):
  +         import mysql.connector
  +         self.conn = mysql.connector.connect(user=user, password=passwd, host=host, port=port,
  +                 database=database, auth_plugin=auth_plugin, autocommit=True)
  +

===========changed ref 1===========
    # module: db.basedb


    class BaseDB(object):

        def _replace(self, tablename=None, **values):
            tablename = self.escape(tablename or self.__tablename__)
            if values:
                _keys = ", ".join(self.escape(k) for k in values.keys())
                _values = ", ".join([self.placeholder, ] * len(values))
                sql_query = "REPLACE INTO %s (%s) VALUES (%s)" % (tablename, _keys, _values)
            else:
                sql_query = "REPLACE INTO %s DEFAULT VALUES" % tablename
            logger.debug("<sql: %s>", sql_query)

            if values:
                dbcur = self._execute(sql_query, list(values.values()))
            else:
                dbcur = self._execute(sql_query)
  +         lastrowid = dbcur.lastrowid
  +         dbcur.close()
  +         return lastrowid
  -         return dbcur.lastrowid
[More changed references…]
```

Figure 4.15: Multi-round editing example 3 (reference blocks). The bottom changes highlighted in orange are similar to the changes needed in Figure 4.14.

Chapter 5

Conclusion and Future Work

In this thesis, we presented techniques that combine static analysis, code transformations, and machine learning to tackle two challenging problems in the domain of programming languages: probabilistic type inference and contextual code change prediction.

We first introduced LambdaNet, a neural architecture for type inference that combines explicit program analysis with graph neural networks. LambdaNet demonstrated improved performance over state-of-the-art tools in predicting library types and effectively predicted user-defined types not encountered during training. Despite its success, LambdaNet had limitations in handling function types and generic types.

To address these limitations, we developed TypeT5, a system that integrates a pre-trained code completion model (CodeT5) with lightweight static analysis and a new decoding scheme. TypeT5 outperformed previous systems and showed that incorporating the right context with static analysis significantly improved the prediction of rare and complex types. This work highlights the potential of leveraging pre-trained code models and static analysis for probabilistic type inference tasks.

Finally, we introduced Coeditor, an approach for multi-round code auto-editing based on the CodeT5 architecture. Coeditor incorporated line diff format and static analysis to create large customized model contexts, resulting in substantial performance improvements over existing code completion methods. We also demonstrated the practical application of Coeditor by releasing a VSCode extension for interactive model usage.

Our work emphasizes the importance of combining static analysis with machine learning models to increase their effectiveness in tackling complex programming tasks. The techniques presented in this thesis have the potential to be applied to various programming tasks beyond type inference and code change prediction. For instance, they could help developers with code refactoring, code generation, and unit testing by providing context-aware and edit-sensitive suggestions. We hope that our work will inspire future research in this area, ultimately leading to more powerful and versatile tools for enhancing developer productivity.

Looking ahead, there are several potential improvements and extensions to our type inference systems that could be explored for future work. One critical area for advancement lies in enforcing hard constraints during inference could provide consistent and reliable type assignments, enhancing the quality of the predictions. Another crucial direction for future work is to align the training and inference processes more closely. Our TypeT5 system creates a distribution discrepancy as it conditions only on previously predicted types during inference, despite incorporating user annotations from real code into

the context during model training. This discrepancy could lead to suboptimal performance. Advanced training schemes such as reinforcement learning could be utilized to better align these processes, potentially boosting the performance of the system. Moreover, enhancing the static analysis-based retrieval methods to obtain additional types of relevant context information beyond user-usee relations could further increase the accuracy of our models.

In terms of Coeditor, a promising direction for future work would be extending the model to help users identify regions of code that need to be changed within the entire codebase. This feature could enable new types of usage such as auto-refactoring. Furthermore, it would be beneficial to explore methods for allowing the model to interactively collaborate with users in editing partially modified lines. While our current design prevents the model from modifying lines that the user has already changed, this restriction can also limit the practical usage of our tool.

Bibliography

Armen Aghajanyan, Bernie Huang, Candace Ross, Vladimir Karpukhin, Hu Xu, Naman Goyal, Dmytro Okhonko, Mandar Joshi, Gargi Ghosh, Mike Lewis, et al. CM3: A Causal Masked Multimodal Model of the Internet. *arXiv eprint arxiv:2201.07520*, 2022.

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, 2021.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

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

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *ICLR*, 2017.

Miltiadis Allamanis, Earl T Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In *PLDI*, 2020.

Davide Ancona and Elena Zucca. Principal typings for java-like languages. In *ACM SIGPLAN Notices*, volume 39, pages 306–317. ACM, 2004.

G. BakIr, Neural Information Processing Systems Foundation, T. Hofmann, A.J. Smola, B. Schölkopf, and B. Taskar. *Predicting Structured Data*. Advances in neural information processing systems. MIT Press, 2007. ISBN 9780262026178. URL `https://books.google.com/books?id=b1EFKUoFF8IC`.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL `https://proceedings.neurips.cc/paper/2015/file/e995f98d56967d946471af29d7bf99f1-Paper.pdf`.

Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Program-*

ming, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George van den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, Diego de Las Casas, Aurelia Guy, Jacob Menick, Roman Ring, Tom Hennigan, Saffron Huang, Loren Maggiore, Chris Jones, Albin Cassirer, Andy Brock, Michela Paganini, Geoffrey Irving, Oriol Vinyals, Simon Osindero, Karen Simonyan, Jack W. Rae, Erich Elsen, and Laurent Sifre. Improving language models by retrieving from trillions of tokens. In *arXiv*, 2021. URL `https://arxiv.org/abs/2112.04426`.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4 (OOPSLA):1–28, November 2020. ISSN 2475-1421. doi: 10.1145/3428283. URL `https://dl.acm.org/doi/10.1145/3428283`.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. ISSN 0098-5589, 1939-3520, 2326-3881. doi: 10.1109/TSE.2020.3020502. URL `http://arxiv.org/abs/1810.00314`. arXiv: 1810.00314.

Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading Wikipedia to Answer Open-Domain Questions. In *Proceedings of the 55th*

Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2017. URL https://aclanthology.org/P17-1171.

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

Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. Kafka: Gradual typing for objects. In *ECOOP 2018-2018 European Conference on Object-Oriented Programming*, 2018.

Hal Daumé, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine learning*, 75(3):297–325, 2009.

Yann Dauphin, Gokhan Tur, Dilek Z. Hakkani-Tur, and Larry P. Heck. Zero-shot learning for semantic utterance classification. In *ICLR*, 2013.

Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2020. URL https://arxiv.org/abs/2010.00904.

Yotam Eshel, Noam Cohen, Kira Radinsky, Shaul Markovitch, Ikuya Yamada, and Omer Levy. Named entity disambiguation for noisy text. In *CoNLL*, 2017.

Ali Farhadi, Ian Endres, Derek Hoiem, and David Forsyth. Describing objects by their attributes. In *CVPR*, 2017.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2): 23–38, 1994.

Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: Quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 758–769, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.75. URL `https://doi.org/10.1109/ICSE.2017.75`.

Bernd Gruner, Tim Sonnekalb, Thomas S Heinze, and Clemens-Alexander Brust. Cross-domain evaluation of a deep learning-based type inference system. *arXiv preprint arXiv:2208.09189*, 2022.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the unknown words. In *Proceedings of the ACL*, 2016.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. Learning to complete code with sketches. In *International Conference on Learning Representations*, 2021.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. REALM: Retrieval-Augmented Language Model Pre-Training. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19:1335–1382, 2013.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 152–162, New York, NY, USA, 2018a. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236051. URL `http://doi.acm.org/10.1145/3236024.3236051`.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th acm joint*

meeting on european software engineering conference and symposium on the foundations of software engineering, pages 152–162, 2018b.

Gautier Izacard and Edouard Grave. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. In *arXiv*, 2020. URL `https://arxiv.org/abs/2007.01282`.

Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 874–880, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.74. URL `https://aclanthology.org/2021.eacl-main.74`.

Abhinav Jangda and Gaurav Anand. Predicting variable types in dynamically typed programming languages. *arXiv preprint arXiv:1901.05138*, 2019.

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

Kevin Jesse, Premkumar Devanbu, and Anand Ashok Sawant. Learning to predict user-defined types. *IEEE Transactions on Software Engineering*, 2022.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense Passage Retrieval for Open-Domain Question Answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020. URL `https://aclanthology.org/2020.emnlp-main.550`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2014.

Konduit K.K. Deeplearning4j. `https://github.com/eclipse/deeplearning4j`. Accessed: 2019-09-24.

Marie-Anne Lachaux, Baptiste Rozière, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *ArXiv*, abs/2006.03511, 2020.

Luigi Lavazza, Abedallah Zaid Abualkishik, Geng Liu, and Sandro Morasca. An empirical evaluation of the "cognitive complexity" measure as a predictor of code understandability. *Journal of Systems and Software*, 197:111561, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. *ICLR*, abs/1511.05493, 2016.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering*, pages 304–315. IEEE Press, 2019.

A. M. Mir, E. Latoskinas, and G. Gousios. Manytypes4py: A benchmark python dataset for machine learning-based type inference. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 585–589. IEEE Computer Society, May 2021. doi: 10.1109/MSR52588.2021.00079.

Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, volume 2, page 4, 2016.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. WebGPT: Browser-assisted question-answering with human feedback. In *arXiv*, 2021. URL `https://arxiv.org/abs/2112.09332`.

Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Ines Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SOAR: A Synthesis Approach for Data Science API Refactoring. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 112–124, Madrid, ES, May 2021. IEEE. ISBN 978-1-66540-296-5. doi: 10.1109/ICSE43902.2021.00023. URL `https://ieeexplore.ieee.org/document/9402016/`.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. A conversational paradigm for program synthesis. *arXiv e-prints*, pages arXiv–2203, 2022.

John-Paul Ore, Sebastian Elbaum, Carrick Detweiler, and Lambros Karkazis. Assessing the type annotation burden. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 190–201, 2018.

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

Sheena Panthaplackel, Miltiadis Allamanis, and Marc Brockschmidt. Copy that! Editing Sequences by Copying Spans, December 2020a. URL `http://arxiv.org/abs/2006.04771`. arXiv:2006.04771 [cs, stat].

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020b.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Ray Mooney. Learning to describe solutions for bug reports based on developer discussions. *Findings of the Association for Computational Linguistics: ACL 2022*, 2022.

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. doi: 10.1145/3510003.3510038. URL `https://doi.org/10.1145/3510003.3510038`.

Fabio Petroni, Aleksandra Piktus, Angela Fan, Patrick Lewis, Majid Yazdani, Nicola De Cao, James Thorne, Yacine Jernite, Vladimir Karpukhin,

Jean Maillard, Vassilis Plachouras, Tim Rocktäschel, and Sebastian Riedel. KILT: a Benchmark for Knowledge Intensive Language Tasks. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021. URL `https://aclanthology.org/2021.naacl-main.200`.

Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. In *arXiv*, 2019. URL `https://arxiv.org/abs/1910.10683`.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, pages 419–428, 2014.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015a. ACM.

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

Machel Reid and Graham Neubig. Learning to Model Editing Processes, May 2022. URL `http://arxiv.org/abs/2205.12374`. arXiv:2205.12374 [cs].

Xiang Ren, Wenqi He, Meng Qu, Clare R Voss, Heng Ji, and Jiawei Han. Label noise reduction in entity typing by heterogeneous partial-label embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1825–1834. ACM, 2016.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In

2012 IEEE international conference on acoustics, speech and signal processing (ICASSP), pages 5149–5152. IEEE, 2012.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL `https://aclanthology.org/P16-1162`.

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

Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, 2007b.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 329–340. IEEE, 2021.

Marc Szafraniec, Baptiste Rozière, Hugh Leather Francois Charton, Patrick Labatut, and Gabriel Synnaeve. Code translation with compiler representations. *ArXiv*, abs/2207.03578, 2022.

Dmitriy Traytel, Stefan Berghofer, and Tobias Nipkow. Extending hindley-milner type inference with coercive structural subtyping. In *Asian Symposium on Programming Languages and Systems*, pages 89–104. Springer, 2011.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018. accepted as poster.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NeurIPS*, 2015.

Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *ACM SIGPLAN Notices*, volume 50, pages 45–56. ACM, 2014.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

Xiaolong Wang, Yufei Ye, and Abhinav Gupta. Zero-shot recognition via semantic embeddings and knowledge graphs. In *CVPR*, 2018.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=Hkx6hANtwH`.

Jiayi Wei, Greg Durrett, and Isil Dillig. TypeT5: Seq2seq Type Inference using Static Analysis. In *International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=4TyNEhI2GdN`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 607–618. ACM, 2016.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems*, 2020.

Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: Pretraining for Source Code and Natural Language Editing, September 2022. URL `http://arxiv.org/abs/2208.05446`. arXiv:2208.05446 [cs].

Vita

Jiayi Wei was born in Sichuan, China, and holds a Bachelor of Science degree in Physics from the University of Science and Technology of China. He commenced his graduate studies at the University of Texas at Austin in August 2018 under the supervision of Isil Dillig. Later, he was co-advised by Joydeep Biswas (2021-2022) and Greg Durrett (2022-2023). During his PhD, Jiayi has conducted research in software security, robotics, and machine learning for code, focusing on type inference and code editing. He has a keen interest in software tools and programming languages, with a background in functional programming and six years of experience in writing Scala, before he was converted to Julia and now Python. Jiayi has also gained industry experience as a PhD intern at Facebook (2020) and JetBrains (2021).

Address: MrVPlusOne@gmail.com

This dissertation was typeset with LaTeX by the author.