TYPE5: SEQ2SEQ TYPE INFERENCE USING STATIC ANALYSIS

Jiayi Wei, Greg Durrett, Isil Dillig
Department of Computer Science
University of Texas at Austin
{jiayi, gdurrett, isol}@cs.utexas.edu

ABSTRACT

There has been growing interest in automatically predicting missing type annotations in programs written in Python and JavaScript. While prior methods have achieved impressive accuracy when predicting the most common types, they often perform poorly on rare or complex types. In this paper, we present a new type inference method that treats type prediction as a code infilling task by leveraging CodeT5, a state-of-the-art seq2seq pre-trained language model for code. Our method uses static analysis to construct dynamic contexts for each code element whose type signature is to be predicted by the model. We also propose an iterative decoding scheme that incorporates previous type predictions in the model’s input context, allowing information exchange between related code elements. Our evaluation shows that the proposed approach, TypeT5, not only achieves a higher overall accuracy (particularly on rare and complex types) but also produces more coherent results with fewer type errors—while enabling easy user intervention.

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 & Taha, 2007) was proposed to allow type annotations 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 (Raychev et al., 2015; Hellendoorn et al., 2018; Wei et al., 2020; Pradel et al., 2020; Allamanis et al., 2020; Pandi et al., 2020; Jesse et al., 2021; Mir et al., 2022; Jesse et al., 2022; Peng et al., 2022).

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., 2020), completion (Wang et al., 2021; Ahmad 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 & 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 correct one in every five model predictions.

To summarize, this paper 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.

2 Overview

In this section, we motivate the design of TypeT5 using the example shown in Figure 1. This example features a method predict and two functions eval_on_dataset and chunk_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]].

Figure 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.
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 Encoding (Radford et al.; Gage, 1994) (see Figure 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_seq (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 its 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, 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 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. 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 Methods

3.1 Using CodeT5 for Type Prediction

We formulate type prediction as a sequence-to-sequence (seq2seq) task. Let \( u = (u_1, \ldots, u_n) \) represent a sequence of code tokens, where each token is a single untyped code element \( e \) (function or variable). We insert into \( u \) indexed marker tokens (\( \langle \text{extra}_i \rangle \)) at each point where we wish to predict types and let the model predict \( t = (t_1, \ldots, t_m) \), the token sequence that encodes types for the marked locations in \( u \). Note that \( t \) only contains the types, no other code tokens, in the format \( \langle \text{extra}_1 \rangle \text{[type 1 tokens]} \langle \text{extra}_2 \rangle \text{[type 2 tokens]} \), 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 \( P(t | u, \bar{u}) \) placing a distribution over type sequences \( t \) given the raw code sequence \( u \) and its surrounding code \( \bar{u} \). This process is shown in Figure 2, and \( \bar{u} \) is omitted for clarity.

Our improved model, TypeT5, replaces the surrounding code \( \bar{u} \) with a context constructed from static analysis, which we denote as \( s \). Thus, we can write the model as \( P(t | u, s) \). 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 \( s \) is constructed from static analysis.

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 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 \( \text{chunk} \) is directly used by \( \text{eval} \) on dataset, and we also know that \( \text{predict_on_batch} \) is called by \( \text{predict} \) (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 more details of usage graph construction in subsection A.1.

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 \( s_{\text{pre}} \), \( \text{uses} \) \( s_{\text{uses}} \), main code \( u \), and \( \text{users} \) \( s_{\text{user}} \), which are constructed as follows (see subsection A.9 for a concrete example).
• **Preamble:** The main purpose of $s_{\text{preamble}}$ 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 $u$. If $e$ is a function, $u$ is just its source code, and if $e$ is a variable, $u$ 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 $u$ and prepend it with the header of $C$, as shown in Figure 7.

• **Users:** Given the usage graph $G$, we construct $s_{\text{user}}$, by including the source tokens of all elements from $\text{users}(G, e)$. 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 3.4.

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

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.

### 3.4 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 $P(t \mid u, \bar{u})$, we can infer a most likely set of types $t$ for $u$ (with surrounding context $\bar{u}$) using beam search. Our implementation performs joint prediction of the output types for a single code block $u$, since later types in the block condition on the predictions of earlier types. However, note that both $u$ and $\bar{u}$ 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.

**TypeT5 iterative decoding:** Part of our motivation for including the context $s$ is to exploit its type information at inference time. Crucially, this requires $s$ to be typed. However, the contexts that are drawn from the original codebase are not typed, so TypeT5 iteratively adds types 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 $t_e$, and denote $\mathcal{M}(s)$ as the context obtained by annotating the elements in $s$ according to $\mathcal{M}$. Starting with an empty $\mathcal{M}$ (which leaves any context $s$ unchanged), TypeT5 then iteratively updates $\mathcal{M}$ using an iterative decoding scheme that traverses the usage graph $G$ twice, as shown in Figure 3. The first prediction pass follows the usee-to-user order, 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 $P(t_e \mid u_e, \mathcal{M}(s_e))$, and the predicted types $t'_e$ are then used to update $\mathcal{M}$ such that $\mathcal{M}(e) = t'_e$.

### 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., $\log P(t^*_e \mid u_e, \mathcal{M}^*(s_e))$, for every element $e$ where $t^*_e$ is available, using teacher-forcing. We train the CodeT5 baseline model similarly on the same dataset.

---

1 In our experiments, including predicted types actually hurts the performance due to exposure bias.
2 This requires performing a topological sort over $G$. When $G$ 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.
Table 1: Basic statistics of our two datasets.

<table>
<thead>
<tr>
<th></th>
<th>BetterTypes4Py</th>
<th>InferTypes4Py</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>train</td>
<td>valid</td>
</tr>
<tr>
<td>Projects</td>
<td>573</td>
<td>40</td>
</tr>
<tr>
<td>Nonempty files</td>
<td>16.5K</td>
<td>1098</td>
</tr>
<tr>
<td>Lines of code</td>
<td>2.4M</td>
<td>174K</td>
</tr>
<tr>
<td>Top-level type slots</td>
<td>541K</td>
<td>38.2K</td>
</tr>
<tr>
<td>Top-level user-added types</td>
<td>275K</td>
<td>19.3K</td>
</tr>
<tr>
<td>Rare type ratio</td>
<td>25.7%</td>
<td>23.3%</td>
</tr>
<tr>
<td>Complex type ratio</td>
<td>20.4%</td>
<td>16.6%</td>
</tr>
<tr>
<td>Average type size</td>
<td>1.42</td>
<td>1.33</td>
</tr>
</tbody>
</table>

4 EXPERIMENTS

We implement TypeT5 in Python, whose the source code and model weights can be found on GitHub[1]. We list the hyperparameters and hardware details in subsection A.6. Below, we first compare 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]. We then present ablation studies to evaluate different factors contributing to the model’s performance.

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 the 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 (Pradel et al., 2020; Mir et al., 2022), 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 Many-Types4Py dataset (Mir et al., 2021), which was used to train Type4Py. We describe the selection criteria in subsection A.2. 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. We further discuss the potential code duplication issue (subsection A.3) and label quality issue (subsection A.4) in the appendix.

We summarize key properties of both datasets in Table 1. We define the size of a type as the number of type constructors in its body[5] 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 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[6], 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 subsection A.5 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 T since they

---

[1] Available at [https://github.com/utopia-group/TypeT5](https://github.com/utopia-group/TypeT5).

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

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

### 4.2 Comparing TypeT5 with other approaches

We compare TypeT5 with the basic CodeT5 model described in subsection 3.4 as well as the released versions of three other state-of-the-art approaches from prior work. 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-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 2 (on common types) and Table 3 (on rare types), and we put the combined results in appendix (Table 6). Since HiTyper was only able to make a prediction for about 67% of all labels, we report its performance on this subset. From Table 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 Table 6, 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 subsection A.9.

### Table 2: Accuracy comparison on common types.

<table>
<thead>
<tr>
<th></th>
<th>BetterTypes4Py</th>
<th>InferTypes4Py</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>full</td>
<td>adjusted</td>
</tr>
<tr>
<td>Typhilus</td>
<td>n/a</td>
<td>50.05</td>
</tr>
<tr>
<td>Type4Py</td>
<td>n/a</td>
<td>50.34</td>
</tr>
<tr>
<td>HiTyper</td>
<td>59.20</td>
<td>54.28</td>
</tr>
<tr>
<td>CodeT5</td>
<td>76.74</td>
<td>78.04</td>
</tr>
<tr>
<td>TypeT5</td>
<td>79.24</td>
<td>81.43</td>
</tr>
</tbody>
</table>

### Table 3: Accuracy comparison on rare types.

<table>
<thead>
<tr>
<th></th>
<th>BetterTypes4Py</th>
<th>InferTypes4Py</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>full</td>
<td>adjusted</td>
</tr>
<tr>
<td>Type4Py</td>
<td>n/a</td>
<td>12.37</td>
</tr>
<tr>
<td>HiTyper</td>
<td>10.30</td>
<td>25.51</td>
</tr>
<tr>
<td>CodeT5</td>
<td>49.47</td>
<td>52.95</td>
</tr>
<tr>
<td>TypeT5</td>
<td>58.56</td>
<td>61.47</td>
</tr>
</tbody>
</table>

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.

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. The accuracies reported in the original Type4Py paper are much higher than we measured here. We analyze this discrepancy in subsection A.7. 

---

7. 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.

---

9. The accuracies reported in the original Type4Py paper are much higher than we measured here. We analyze this discrepancy in subsection A.7.
Table 4: Performance of different model modifications. All models are retrained with the corresponding inputs.

<table>
<thead>
<tr>
<th>Modification</th>
<th>Accuracy</th>
<th>Type Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Preamble</td>
<td>64.20</td>
<td>6067</td>
</tr>
<tr>
<td>No Users</td>
<td>71.20</td>
<td>7053</td>
</tr>
<tr>
<td>No Usees</td>
<td>67.25</td>
<td>7332</td>
</tr>
<tr>
<td>Nonincremental</td>
<td>72.52</td>
<td>5720</td>
</tr>
<tr>
<td>Original (TypeT5)</td>
<td>73.02</td>
<td>5087</td>
</tr>
</tbody>
</table>

Table 5: Performance of different decoding strategies. The same TypeT5 model weights are used for different decoding strategies.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Accuracy</th>
<th>Type Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent</td>
<td>71.68</td>
<td>6876</td>
</tr>
<tr>
<td>Random</td>
<td>71.66</td>
<td>6215</td>
</tr>
<tr>
<td>UserToUsee</td>
<td>70.67</td>
<td>7415</td>
</tr>
<tr>
<td>UseeToUser</td>
<td>72.65</td>
<td>6402</td>
</tr>
<tr>
<td>TwoPass (TypeT5)</td>
<td>73.02</td>
<td>5087</td>
</tr>
</tbody>
</table>

4.3 ABLATIONS ON TYPE T5

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. Note that we only report errors that are directly related to type coherence (since the type checker also reports errors unrelated to type coherence, such as unresolved import statements. We give more details about this metric in subsection A.8).

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 s_preamble, s_user, and s_usee context elements (introduced in subsection 3.3) from the model’s input, respectively.

The Nonincremental 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 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 4 (and show other accuracy metrics in Table 7). 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 3.4, 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 Nonincremental 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 5 (and show other accuracy metrics in Table 8). 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.

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 $M^*$, 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 $t_e$ with the corresponding human annotations $M^*(e)$ if $e \in M^*$. On the BetterTypes4Py dataset, this interactive decoding process achieves a full accuracy of 78.04% and an adjusted accuracy of 79.37%—meaning that

---

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 accuracy in expectation.
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.

5 RELATED WORK

Deep Learning Type Inference Most prior approaches predict user-defined types via some form of type embedding matching. For example, 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 analysis. 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 (Chu 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 (Lazicard & 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 & Barto, 2018) may help further boost the performance of iterative decoding.

6 CONCLUSION

In this work, we showed that TypeT5, a system integrating CodeT5 with lightweight static analysis and a new decoding scheme, is effective at predicting types for untyped codebases. We show that fine-tuning a pre-trained code model already results in a strong baseline, outperforming systems from past work, and that incorporating the right context with static analysis is particularly helpful to enable accurate prediction on more challenging cases. We believe that even larger models can further improve the accuracy, but they will still need to leverage the methods in this work to achieve high accuracy on rare and complex types.
ACKNOWLEDGMENT

This work is partly funded by NSF Award CCF-1918889. We would like to express our sincere gratitude to Miltiadis Allamanis and Amir Mir for generously helping us in setting up their tools and completing the comparison experiments. We would also like to thank members of the UTOPIA group, especially Jocelyn Chen, for their assistance in managing the GPU server for the experiments.

REFERENCES


A APPENDIX

A.1 CONSTRUCTING THE USAGE GRAPH

To resolve the user-usee relations at the project level (used in subsection 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 straightforward 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.

A.2 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.

A.3 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 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.

A.4 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 subsection A.8) 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.

11 https://github.com/kucherenko/jscpd
Table 6: Accuracy comparison on all types (common + rare).

<table>
<thead>
<tr>
<th></th>
<th>BetterTypes4Py</th>
<th>InferTypes4Py</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>full adjusted base</td>
<td>full adjusted base</td>
</tr>
<tr>
<td></td>
<td>all simple complex</td>
<td>all simple complex</td>
</tr>
<tr>
<td>Type4Py</td>
<td>n/a 34.52 35.87 19.68</td>
<td>35.61 n/a 21.11 22.35 9.61</td>
</tr>
<tr>
<td>HiTyper</td>
<td>42.53 41.95 44.86 19.02</td>
<td>47.88 34.83 32.50 35.11 11.40</td>
</tr>
<tr>
<td>CodeT5</td>
<td>67.07 67.47 72.12 44.05</td>
<td>73.44 68.80 68.95 75.14 54.04</td>
</tr>
<tr>
<td>TypeT5</td>
<td>71.89 73.02 77.07 49.72</td>
<td>78.87 71.98 73.15 77.16 62.66</td>
</tr>
</tbody>
</table>

A.5  **Type Normalization**

To compute the accuracy metrics in subsection 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`.

A.6  **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 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., UseToUser) 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.

A.7  **Why Type4Py has much lower performance on our dataset**

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 inferable from the type checker, which explain the performance drop when evaluated on our datasets.

A.8  **Measuring Type Coherence using Type Errors**

To estimate the type coherence (subsection 4.3), we call the type checker `Mypy` on codebases annotated with the types predicted by the model. Since not all errors reported by `Mypy` are type errors, we filter out warnings and style errors. We consider an error to be relevant if it is either in the same file as the type prediction or within 10 lines above or below it. We report the number of relevant errors as a measure of type coherence.

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 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 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 iterative decoding model.

## A.9 Real Examples Produced by TypeT5

We run TypeT5 on the actual code corresponding to the example shown in Figure 1 and show the obtained preamble (Figure 5), usee context (Figure 6), main code (Figure 7), and user context (Figure 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 iterative decoding process (described in subsection 3.4), so all the elements have already been annotated at least once (but some may have not been annotate the second time).
Figure 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 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 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 use 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 4.4).
Figure 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.