TYPET5: SEQ2SEQ TYPE INFERENCE USING STATIC ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

There has been growing interest in automatically predicting missing type annotations in programs written in Python and JavaScript. Prior methods have achieved impressive accuracy when predicting the most common types, but they often perform poorly on rare or complex types (or do not support them at all). In this paper, we present a new type inference method that treats type prediction as a code completion task by leveraging CodeT5, a state-of-the-art seq2seq pre-trained language model for code. Our method uses static analysis to construct context for each code element whose type is to be predicted by the model. We also propose a sequential decoding scheme that incorporates previous type predictions in the model's input context, allowing information exchange between related code elements. Our evaluation shows that our 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 code and catch errors. To address this issue, *gradual typing* (Siek & Taha, 2007) allows programmers to incrementally add type annotations to previously 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 there have been several research efforts that automatically predict missing type annotations using deep learning. (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)

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 solves this challenge by using static analysis to identify what parts of the codebase are useful for type prediction. In particular, our method constructs 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 extracted via static analysis, our method then encodes the users and usees of a given code element in a form that resembles normal code and feeds it as additional context to the transformer model. with the goal of facilitating knowledge transfer. Furthermore, to take full advantage of the seq2seq paradigm,

TypeT5 uses an incremental decoding scheme that conditions on prior type predictions, allowing information to be propagated across distant code elements.

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 two state-of-the-art type inference tools (Allamanis et al., 2020; Mir 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 over the CodeT5 baseline, while an additional type checking experiment shows that the proposed sequential 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. Our result demonstrates the usefulness of our approach as a developer tool to annotate entirely untyped projects—requiring the user to correct only one in every five model predictions.

To summarize, this papers 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 a sequential decoding scheme that particularly helps with *coherence*, as measured by the number of type errors reported by the type checker.

2 Overview

In this section, we motivate the design of TypeT5 with the aid of the motivating example shown in Figure 1. This example features a class 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 completion In this work, we advocate a new approach that views type inference as an instance of code completion. Because type annotations can be viewed as missing

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

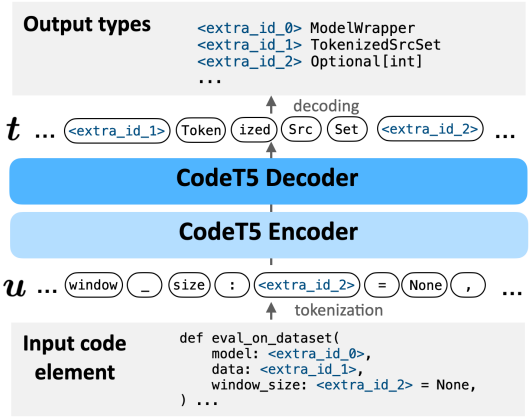

Figure 2: How CodeT5 encodes and decodes source code using BPE. Marker tokens (highlighted in blue) indicate gaps (input) and their corresponding fillers (output).

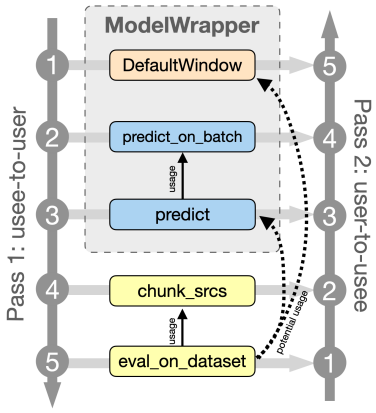

Figure 3: The two-pass sequential decoding process.

code fragments, we fine-tune a state-of-the-art code completion 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_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, 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 a sequential 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.

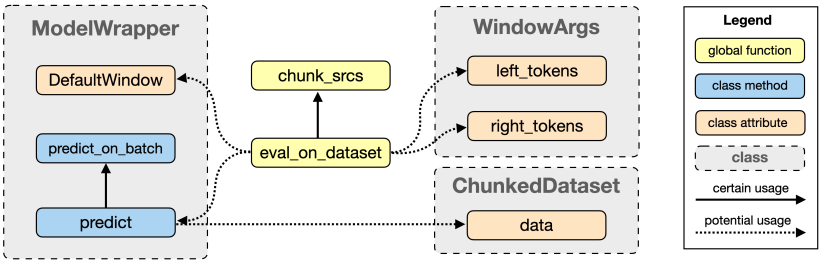

Figure 4: The usage graph corresponding to the code snippets in Figure 1. `WindowArgs` and `ChunkedDataset` are assumed to be defined elsewhere in the same project.

3 METHODS

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 its surrounding code $\bar{\boldsymbol{u}}$. This process is shown in Figure 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.

3.2 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, and 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 `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 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 $\boldsymbol{s}_{\text{pre}}$, usees $\boldsymbol{s}_{\text{usee}}$, main code $\boldsymbol{u}$, and users $\boldsymbol{s}_{\text{user}}$, which are constructed as follows (see subsection A.7 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 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 3.4.
- **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.

3.4 Sequential Decoding Inference

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

Basic CodeT5. 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 method 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 sequential 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 a sequential decoding scheme that traverses the usage graph $\mathcal{G}$ twice, as shown in Figure 3. The 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 $\mathcal{M}(e) = \boldsymbol{t}'_e$.

3.5 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.

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

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

4 EXPERIMENTS

In this section, we first compare TypeT5 against two state-of-the-art type inference systems for Python, namely Typilus (Allamanis et al., 2020) and Type4Py (Mir et al., 2022). We then present ablation studies to evaluate different factors contributing to the model's performance.

We implement TypeT5 in Python and plan to release its source code and model weights upon paper acceptance. We list the hyperparameters and hardware details in subsection A.4.

4.1 EVALUATION SETUP

Datasets In our evaluation, we predict the type annotations for *public* APIs of an entire Python project, including class attributes, methods, top-level functions, and global variables. We treat any existing user-added type annotations as the ground truth, and we measure accuracy against these annotations 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 that are newly defined in the test projects will not be 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 data cleaning details in subsection A.3. 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, TypeT5, none of which were used as CodeT5's (pre-)training data.

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[3] 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,[4] 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.2, 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`); (3) rewrites any outermost `Optional[T]` and `Final[T]` into `T` since they tend not to be used consistently across programmers.[5] 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 two other state-of-the-art approaches described in prior work as well as the basic CodeT5 model described in subsection 3.4. 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 trained Type4Py model via its web interface.

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

[4] 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

[5] e.g., the type checker `mypy` allows the user to configure whether it should assume implicit Optional types to permit common patterns like `n:int = None`.

Table 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
API type slots	541K	38.2K	28.4K	4.6K
API 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

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

	BetterTypes4Py					InferTypes4Py				
	full	**adjusted**			**base**	**full**	**adjusted**			**base**
	all	all	simple	complex	all	all	all	simple	complex	all
Typilus	n/a	54.05	55.12	33.23	60.37	n/a	52.33	52.19	53.91	64.67
Type4Py	n/a	50.34	51.91	32.14	47.51	n/a	32.08	33.47	16.54	29.83
CodeT5	76.74	78.04	82.43	53.03	82.44	77.83	78.06	85.31	63.41	81.87
TypeT5	79.24	81.43	85.69	56.75	84.82	81.75	82.95	87.62	72.78	84.17

We show each tool's accuracy in Table 2 (on common types only) and Table 3 (on all types). From Table 2, we can make the following observations. First, our CodeT5 baseline model already outperforms both Typilus and Type4Py by a large margin, demonstrating the advantage of using a seq2seq pre-trained language model. Second, TypeT5 further improves the adjusted accuracy by 3.4% and 4.9% on the two datasets. Third, Type4Py's performance on InferTypes4Py dataset is significantly lower than on BetterTypes4Py, likely because some of the test files in BetterType4Py were also used to train Type4Py.[6] Looking at Table 3, we additionally see that TypeT5 significantly outperforms others when rare types are considered, with TypeT5 improving upon CodeT5 by 5.6% and 4.2% on the two datasets. Moreover, TypeT5 achieves the largest improvements on complex types, improving upon CodeT5 by 5.7% and 8.6%. These results suggest that global information is especially important in situations that involve both rare and complex types.

We also show the outputs of TypeT5 on a real-world example in subsection A.7.

4.3 Ablations on TypeT5

We next present a series of ablations that evaluate how various factors contribute to the performance of TypeT5. In addition to full accuracy, we also report the **type error count** as a way to estimate how *coherent* the model's predictions are.[7] Note that we only report errors that are directly related to type coherence (since the type checker also reports errors that are not related to type coherence,

[6]The accuracies reported in the original Type4Py paper are much higher. We analyze this discrepency in subsection A.5.

[7]Accuracy does not always correlate with coherency. e.g., when we have `x = y` and `x` is 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.

Table 3: Accuracy comparison on all types (**common** + **rare**).

	BetterTypes4Py					InferTypes4Py				
	full	**adjusted**			**base**	**full**	**adjusted**			**base**
	all	all	simple	complex	all	all	all	simple	complex	all
Type4Py	n/a	34.52	35.87	19.68	35.61	n/a	21.11	22.35	9.61	22.64
CodeT5	67.07	67.47	72.12	44.05	73.44	68.80	68.95	75.14	54.04	77.78
TypeT5	71.89	73.02	77.07	49.72	78.87	71.98	73.15	77.16	62.66	80.20

Table 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

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

such as unresolved import statements). We provide more details about how we compute this metric in subsection A.6.

How do different components contribute to the model's performance? Recall that TypeT5 augments context information using static analysis. To evaluate the relative 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) from the model's input. The **Nonincremental** model does not perform the sequential 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 type of information will have more space for other types of information. We show both the adjusted accuracy and number of type errors in Table 4. We can see that (1) all components improve the overall accuracy, with the preamble having the largest impact (2) while the sequential decoding scheme only improves overall accuracy by 0.5%, 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 sequential 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. 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 allow 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 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.

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 and trains the model with contrastive learning. 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, they do 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. TypeWriter (Pradel et al., 2020) combines a type checker with inference-time search to avoid generating type errors, and OptTyper (Pandi et al., 2020) explicitly models type constraints by turning them into a continuous optimization objective. Finally, unlike aforementioned approaches, 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 & 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, these applications have different properties from ours: knowing what kind of encyclopedic knowledge is needed to answer a question is a very different problem than knowing what contexts in code are needed to infer type information. We are also able to leverage static analysis based on the usage graph which has no analogue for text; however, we are not able to rely on relatively unambiguous identifiers like names of real-world entities. Finally, our sequential decoding scheme is needed to reckon with the fact that our added context itself contains untyped code that we need to run inference on.

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 sequential 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 sequential 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 pre-trained code models are already a strong baseline, outperforming systems from past work, and that incorporating the right context with static analysis is particularly helpful to enable accurate prediction. We believe that even larger models can improve the accuracy of our predictions further, but will still need to leverage the methods in this work to achieve high accuracy on user-defined types and complex types.

REFERENCES

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*, pp. 2655–2668, 2021.

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

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.

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 (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/e995f98d56967d946471af29d7bf99f1-Paper.pdf.

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.

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.

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

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.

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

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.

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.

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*, pp. 152–162, 2018.

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.

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*, pp. 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.

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, 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 304–315. IEEE, 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)*, pp. 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*, pp. 2241–2252, 2022.

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.

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, 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*, pp. 1853–1868, 2020.

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.

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*, pp. 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 Andreas Krause. Predicting program properties from" big code". *ACM SIGPLAN Notices*, 50(1):111–124, 2015.

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*, pp. 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)*, pp. 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)*, pp. 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*, pp. 2–27. Springer, 2007.

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

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*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

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.

While our implementation limits the analysis scope to the current project, it is straight-forward to extend this 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.

A.2 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.3 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.4 HYPERPARAMETERS

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.) Training the model took about 11 hours on a single Quadro RTX 8000 GPU with 48GB memory. Performing the two-pass inference on the entire test set takes about 1.7 hours.

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.

A.5 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 public 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.

A.6 MEASURING TYPE COHERENCE USING TYPE ERRORS

To estimate the type coherence (subsection 4.3), we call the type checker `MyPy`[8] 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.

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

model_input.py **Preamble**

```
1   import random
2   from copy import copy, deepcopy
3   import numpy as np
4   from collections import Counter
5   from mypy_extensions import mypyc_attr
6   from torch import Tensor
7   from transformers.data.data_collator import DataCollatorForSeq2Seq
8   from typing import NamedTuple, overload
9   from datasets.arrow_dataset import Dataset
10  from torch.utils.data import DataLoader, RandomSampler
11  from.data import (
12      ChunkedDataset,
13      CtxArgs,
14      TokenizedSrcSet,
15      output_ids_as_types,
16      preds_to_accuracies,
17  )
18  from.type_env import AccuracyMetric, PythonType
19  from.utils import *
20  @dataclass
21  class DecodingArgs:
22      ...
23  @dataclass
24  class DatasetPredResult(Generic[T1]):
25      ...
26  @dataclass
27  class ModelWrapper:
28      ...
```

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.

- **`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 types and will thus likely raise multiple errors. However, if the model predicts a non-existing type, the type checker will only raise 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 sequential decoding model.

A.7 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 sequential 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).

```
Usee Context
# spot.model
@dataclass
class DecodingArgs:
    sampling_max_tokens: int ✓
    ctx_args: CtxArgs ✓

# spot.utils
def assert_eq(x: T1, *xs: T1, extra_message: Callable[[], str] = lam
                  ✓       ✓                  ✓

# spot.model
@dataclass
class DatasetPredResult(Generic[T1]):
    chunks: ChunkedDataset ✓
    predictions: list[T1] ✗ list[list[PythonType]]
    extra_info: list[dict] = field(default_factory=list)
                  ✗ list[T1]
# spot.model
@dataclass
class ModelWrapper:
    model: PythonType ✗ ModelSPOT
    tokenizer: PythonType ✗ TokenizerSPOT
    args: CtxArgs ✗ DecodingArgs
    def predict_on_batch(
        self,
        batch: dict, ✓
        num_return_sequences: int = None, ✓
    ) -> tuple[list[PythonType], Tensor]:...
         ✗ tuple[list[list[PythonType]], Tensor]

# spot.model
def dynamic_dataloader(
    dataset: ChunkedDataset, ✗ Dataset
    max_tokens: int, ✓
    collate_fn: DataCollatorForSeq2Seq, ✓
    shuffle: bool = False, ✓
) -> DataLoader:...

# Used above
```

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.

Main Code

```
109     # spot.model
110     @dataclass
111     class ModelWrapper:
112         def predict(
113             self,
114             dataset: ChunkedDataset,  Dataset
115             tqdm_args: dict = {},
116             num_return_sequences: int = None,
117         ) -> list[PythonType]:  list[list[PythonType]]
118             model = self.model
119             collator = DataCollatorForSeq2Seq(self.tokenizer, model)
120             loader = dynamic_dataloader(
121                 dataset,
122                 max_tokens=self.args.sampling_max_tokens,
123                 collate_fn=collator,
124                 shuffle=True,
125             )
126             device = model.device
127             pred_types = dict[int, list]()
128             with tqdm(
129                 total=len(dataset), desc="predict", smoothing=0.01, **tqdm_args
130             ) as tqdm_bar:
131                 for batch in loader:
132                     n_chunks = batch["input_ids"].shape[0]
133                     batch["input_ids"] = batch["input_ids"].to(device)
134                     preds, _ = self.predict_on_batch(batch, num_return_sequences)
135                     for i, c_id in enumerate(batch["chunk_id"]):
136                         c_id = int(c_id)
137                         if num_return_sequences is None:
138                             pred_types[c_id] = preds[i]
139                         else:
140                             pred_types[c_id] = preds[
141                                 i * num_return_sequences : (i + 1) * num_return_seque
142                             ]
143                     tqdm_bar.update(n_chunks)
144             return [pred_types[int(c_id)] for c_id in dataset["chunk_id"]]
145
```

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 actually 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 4.4).

User Context, first half

```
147     # Users below
148     # spot.model
149     @dataclass
150     class ModelWrapper:
151         def eval_on_dataset(
152             self,
153             src_data: TokenizedSrcSet,
154             max_labels: int = None,
155             tqdm_args: dict = {},
156         ) -> DatasetPredResult:
157             ctx_args = self.args.ctx_args
158             if max_labels is not None:
159                 ctx_args = copy(ctx_args)
160                 ctx_args.max_labels = max_labels
161
162             chunks = src_data.to_chunks(ctx_args, tqdm_args=tqdm_args)
163             preds = self.predict(
164                 chunks.data, num_return_sequences=None, tqdm_args=tqdm_args
165             )
166             return DatasetPredResult(chunks, preds)
167
```

User Context, second half

```
169     # spot.decode
170     def sample_candidates(
171         wrapper: ModelWrapper,
172         src_data: TokenizedSrcSet,
173         n_samples: int,                tuple[ChunkedDataset, list[list[list[PythonType]]]]
174     ) -> tuple[ChunkedDataset, list[PythonType]]:
175         ctx_args = wrapper.args.ctx_args
176         do_sample = wrapper.args.do_sample
177         if not do_sample:
178             assert wrapper.args.num_beams is not None, "num_beams needs to be set"
179             assert n_samples <= wrapper.args.num_beams
180
181         chunks = src_data.to_chunks(ctx_args)
182         n_chunks = len(chunks.data)
183
184         if do_sample:
185             samples = [
186                 wrapper.predict(chunks.data, tqdm_args={})
187                 for _ in tqdm(range(n_samples), desc="Sampling")
188             ]
189         else: …
197
198         def get_preds(chunk_id, sample_id):
199             return (
200                 samples[sample_id][chunk_id] if do_sample else samples[chunk_id][sampl
201             )
202
203         pred_candidates = [
204             [get_preds(cid, sid) for sid in range(n_samples)] for cid in range(n_chunk
205         ]
206         return chunks, pred_candidates
207
```

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.