PyTy: Repairing Static Type Errors in Python

Yiu Wai Chow  
University of Stuttgart  
Germany  
victorcwai@gmail.com

Luca Di Grazia  
University of Stuttgart  
Germany  
work@lucadigrazia.com

Michael Pradel  
University of Stuttgart  
Germany  
michael@binaervarianz.de

ABSTRACT

Gradual typing enables developers to annotate types of their own choosing, offering a flexible middle ground between no type annotations and a fully statically typed language. As more and more code bases get type-annotated, static type checkers detect an increasingly large number of type errors. Unfortunately, fixing these errors requires manual effort, hampering the adoption of gradual typing in practice. This paper presents PyTy, an automated program repair approach targeted at statically detectable type errors in Python. The problem of repairing type errors deserves specific attention because it exposes particular repair patterns, offers a warning message with hints about where and how to apply a fix, and because gradual type checking serves as an automatic way to validate fixes. We address this problem through three contributions: (i) an empirical study that investigates how developers fix Python type errors, showing a diverse set of fixing strategies with some recurring patterns; (ii) an approach to automatically extract type error fixes, which enables us to create a dataset of 2,766 error-fix pairs from 176 GitHub repositories, named PyTyDefects; (iii) the first learning-based repair technique for fixing type errors in Python. Motivated by the relative data scarcity of the problem, the neural model at the core of PyTy is trained via cross-lingual transfer learning. Our evaluation shows that PyTy offers fixes for ten frequent categories of type errors, successfully addressing 85.4% of 281 real-world errors. This effectiveness outperforms state-of-the-art large language models adopted optional type annotations, enabling developers to annotate types of their choosing.

CCS CONCEPTS

- Software and its engineering → Software maintenance tools.

KEYWORDS

Automatic Program Repair, Type Annotation, Transfer Learning

ACM Reference Format:


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION

Dynamically typed languages, such as Python and JavaScript, have become very popular.1 One reason is their lightweight syntax, which does not require developers to specify types for parameters, return values, or variables. Because this flexibility may negatively affect the maintainability and robustness of code, in 2015, Python adopted optional type annotations, enabling developers to annotate types of their choosing.

Context. Since their introduction into the Python language, type annotations have been getting increasingly popular [9]. To support developers, several automated approaches for adding type annotations to existing code bases have been proposed, e.g., Type-Wraper [34], DeepTyper [17], Typilus [1], and work by Xu et al. [51]. While adding type annotations is generally considered a step forward, newly added annotations often reveal previously unnoticed type errors, which can be easily detected with a static type checker. Unfortunately, developers commonly lack the time to fix these errors [9], which hampers the usefulness of gradual typing.

Figure 1 shows two real-world, statically detectable type errors along with their fixes, as performed by developers. The error presented in Figure 1a is caused by passing the arguments to a function in the wrong order [35], i.e., a kind of problem that in statically typed languages is often can be prevented by the type system. The developers fix the problem by swapping the arguments.2 The error presented in Figure 1c is caused by annotating a parameter to be a string, while at the same time, initializing it to None, which is type incompatible with str.3 To fix this error, the developer modifies the type annotation to Optional[str]. As illustrated by these examples, there may be many ways of addressing different type errors in Python, and finding the right fix for a given error is non-trivial.

1https://octoverse.github.com/#top-languages-over-the-years
2https://github.com/pythonarcade/arcade/commit/c6aL883
3https://github.com/awslabs/aws-lambda-powertools-python/3898e55
Significance. Organizations with large Python code bases invest significant efforts toward using type annotations and type checkers. For example, Google’s Python style guide mentions that developers are “strongly encouraged to enable Python type analysis” because “The type checker will convert many runtime errors to build-time errors.” Likewise, Dropbox type-checked over four million lines of Python in 2019, because “A type checker will find many subtle (and not so subtle) bugs. A typical example is forgetting to handle a None value or some other special condition.” Finally, Meta use[s] it extensively to maintain the codebases of Facebook and Instagram.

To handle type errors in legacy code and type errors revealed by adding type annotations to previously unannotated code, an automated technique to help developers fix such errors would be desirable. However, despite the increasing popularity of automated program repair (APR) [25], there currently is no APR approach targeting static type errors in Python. Compared to repair scenarios targeted by existing APR approaches, fixing type errors in Python differs in three important ways, making the problem particularly amenable to automated repair. First, type errors require specific fix patterns, which an approach specifically targeting such errors can exploit. Second, when a gradual type checker reports a type error, the report includes an error message that may offer hints about the location and nature of the problem. Third, the gradual type checker also offers an automatic oracle, which an APR technique can use to validate candidate fixes.

Approach. This paper introduces PyTy, the first APR approach for static type errors in Python. To guide the design of PyTy, we investigate in a preliminary study how developers typically fix type errors. The study investigates (i) how repetitive type errors and their fixes are, (ii) how difficult it is to localize the fix location, and (iii) to what extent the error message provided by a type checker helps in finding the fix. In short, the results show that there are recurring fix patterns, but ambiguous rules for when to apply them, and that the locations and error messages provided by the type checker are valuable information.

Based on the results of the preliminary study, we design PyTy as a data-driven approach. This kind of approach requires a dataset for training and evaluation. However, automatically collecting a large-scale dataset of type error fixes is challenging because it requires identifying relevant commits and isolating the type error fixes in these commits. We address these challenges through an automated approach that combines gradual type checking and delta debugging [54]. Using this approach, we obtain 2,766 real-world pairs of type errors and corresponding single-hunk fixes from 176 GitHub repositories. To the best of our knowledge, our PyTyDefects dataset is the first of its kind.

The core of PyTy is a neural type error repair model. Motivated by the relative data scarcity of the problem, we present a cross-lingual transfer learning approach. Specifically, we base PyTy on the existing APR system TFix [5], which has been trained to fix lint errors in JavaScript code. By fine-tuning the TFix model with PyTyDefects, we retain the knowledge learned from fixing JavaScript code and apply it to fixing Python type errors. To ensure that every fix suggested by PyTy indeed fixes the targeted type error, the approach checks candidate fixes with a gradual type checker, and returns a fix only if it removes the error.

Results. Our evaluation on a held-out subset of 281 type error fixes shows that PyTy finds a fix that removes type errors for 85.4% of all errors. Moreover, 54.4% of the predicted fixes exactly match the developer’s fix. Comparing PyTy with previous work, we find that it clearly outperforms several state-of-the-art large language models (text-davinci-003, gpt-3.5-turbo, and gpt-4) asked to repair type errors (54.4% vs. 26.4% exact matches) and complements a technique aimed at type errors that manifest at runtime [32]. As evidence of the usefulness of PyTy in practice, 20 out of 30 GitHub pull requests with PyTy-suggested fixes have been merged by the developers. Finally, we also validate the automatically gathered PyTyDefects dataset underlying our approach, and find that almost all gathered fixes are minimal and correct.

Contributions. In summary, the contributions of this paper are:

• An empirical study of how developers fix type errors.
• A technique to extract type errors and corresponding fixes through a combination of gradual type checking and delta debugging, which yields the first dataset of its kind, with 2,766 type error fix pairs from 176 GitHub repositories.
• Cross-language transfer learning that uses a model pre-trained on JavaScript to repair type errors in Python.
• Empirical evidence of the effectiveness of the approach when being applied to real-world type errors.

2 BACKGROUND ON PYTHON TYPE CHECKERS

In 2015, Python introduced a syntax for type annotations. These annotations are optional and not checked at runtime. The Python language also does not define a static type system, but leaves type checking to third-party tools. In response, the Python community has developed several type checkers that perform gradual type checking [40], i.e., a form of type checking aimed at exposing incompatibilities between the provided type annotations while allowing parts of the program to remain unannotated. Popular type checkers include Pyre, Mypy, Pytype, and Pyright. The type systems implemented by checkers differ, and hence, different type checkers may reveal different type errors [38]. Conceptually, the approach described in this paper is independent of a specific type checker and could be adapted to any of the popular checkers. Our implementation builds upon Pyre because it is widely used, available as open-source, backed by a major tech company, and has been the basis of recent work on studying Python type annotation practices [9]. Pyre reports a wide range of type-related problems, such as incompatible variable, parameter, and return types, uses of unbound names, unsupported operands, and inconsistent method overrides. We use Pyre’s default configuration, i.e., it runs only on functions that are at least partially type-annotated. In the remainder of the paper, we refer to Pyre using the term type checker.

https://google.github.io/styleguide/pyguide.html#2212-pros
https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python
https://developers.facebook.com/blog/post/2021/05/10/eli5-pyre-fast-error-flagging-python-codebases/
3 PRELIMINARY STUDY

To guide important design decisions of our approach, we perform a preliminary empirical study that investigates three questions (PQs):

PQ1 How repetitive are real-world type errors and type error fixes? Answering this question is useful for deciding about the kind of technique, e.g., rule-based vs. data-driven, to build for automatically repairing type errors.

PQ2 How difficult is identifying the fix location for a given type error? Answering this question helps us decide how PyTy can effectively determine where in the given code to fix a type error.

PQ3 How useful for fixing type errors are the error messages provided by a type checker? Answering this question is useful to determine if and how a repair technique will be able to benefit from error messages.

3.1 Data Collection

To address the above questions, we systematically study type error fixes in the version histories of popular projects. We apply three strategies to select commits with type error fixes. First, we search for GitHub issues that call for help in fixing type errors. Second, we search for commits on GitHub via the keywords: “type+fix”, “pyre” and “mypy” in Python repositories with more than 100 stars. Third, we use a dataset extracted from the top 10,000 Python repositories [9], which contains commits with edits related to inserting, removing, or updating a type annotation.

After collecting the commits, we clone the repositories and run the type checker before and after each commit. During our manual inspection, we observe that some warnings and fixes are not useful for our study towards building an APR tool, and hence, we remove (i) fixes that delete entire functions or files, without actually fixing a type error, (ii) import-related warnings, as they are often due to libraries missing in the type checker’s search path, and (iii) fixes that add comments # pyre-ignore or # type: ignore to suppress warnings from the type checker. Overall, for the preliminary study, we collect 125 type error fixes from 14 GitHub repositories.

3.2 Results

3.2.1 PQ1: Repetitiveness of Type Errors and Fixes. We analyze the most frequent classes of type errors fixed by developers, which helps understand which errors concern developers the most, and hence, should be the focus of an APR technique. Figure 2 (left) shows the distribution of the most frequently fixed classes of type errors.

We take the classes of type errors from the Pyre documentation. The most frequent classes are incompatible return, variable, and parameter types, which together account for 64.8% of the dataset. For example, one such fix is for a function expected to return a str but that actually returns int due to a statement return −1. The error is fixed by changing the return statement to return “X”.10

We also analyze the most frequent types involved in the fixes. We observe that Python’s built-in types occur frequently, e.g., str (23.9%) and int (22.4%). Also relatively frequent are types related to optional values, such as Optional (9.3%) and None (5.6%), and other types from the typing library, e.g., Union (7.1%).

We study how type errors are fixed by manually categorizing the error-fixing code changes into 17 classes. This categorization was performed by one of the authors based on grounded theory [14], i.e., we discovered and refined fix patterns until they sufficiently covered the studied examples. Figure 2 (right) shows the distribution of the identified fix patterns. Beyond the distribution, the figure shows that there is no simple mapping from classes of type errors to fix patterns. The most frequent relationships are Incompatible return type fixed with the pattern Modify function return type (14.4%) and Incompatible parameter type fixed with the pattern Modify function parameter type (13.6%). However, the same class of type error may also get addressed by applying several other fix patterns.

Answer to PQ1: A few kinds of type errors account for most fixed errors, and the fixes often involve Python’s built-in data types. The fixes expose some recurring patterns, but only an ambiguous mapping from classes of type errors to fix patterns.

3.2.2 PQ2: Difficulty of Identifying the Fix Location. To assess the difficulty of localizing where to fix type errors, we start by investigating how much code developers typically change to fix a type error. Based on the categorization of fix patterns in Figure 2, we see that most fixes are single-line edits, such as modifying a type annotation from one type to another, changing an operator, removing a type annotation, or adding a cast. Next, we study the location of the fixes (Figure 4a). More than half of the fixes happen exactly in the line where the type error is reported. Other locations include the function parameters, return annotations, and function callees (i.e., the functions that are called).

Answer to PQ2: Most fixes of type errors affect only a single line of code, which often is the line where the type checker reports the type error.
def is_valid_public_key_static(
    local_private_key_str: str, remote_public_key_str: str, prime: int
) -> bool:
    ...)

[Error message] Expected 'int' for 1st parameter but got 'str'.
if pow(remote_public_key_str, (prime - 1) // 2, prime) == 1:
    ...

(a) Commit with a type error.

def get_model_for_finetuning(
    previous_model_file: Optional[Union[Path, Text]]
) -> Optional[Text]:
    ...)

[Error message] Expected 'Optional[Union[Path, Text]]' but got 'Union[None, Path, Text]'.
) -> Optional[Text]:

(c) Commit with a type error.

def get_model_for_finetuning(
    previous_model_file: Optional[Union[Path, Text]]
) -> Optional[Union[Path, Text]]

[Fix pattern] Modify function return type.
if pow(remote_public_key_str, (prime - 1) // 2, prime) == 1:
    ...

(b) Commit that applies the "Use expected type" pattern.

def get_model_for_finetuning(
    previous_model_file: Optional[Union[Path, Text]]
) -> Optional[Union[Path, Text]]

[Fix pattern] Modify function parameter type.

(d) Commit that applies the "Do not use expected type" pattern.

Figure 3: Examples of fixing type errors based on error messages.

3.2.3 PQ3: Usefulness of Error Messages and Locations. Finally, we
want to understand how useful the error messages provided by a
type checker are for fixing type errors. To this end, we extract from
the error message the kind of error, the types involved, and any
hinds about the location and the fix. We classify an error message
as correctly hinted if the message contains the type that the developer
uses to fix the error. Figure 3a shows an example, where the
type checker returns the following error message: "Incompatible
parameter type [6]: Expected int for 1st positional only parameter
to call pow but got str", where the message hints at replacing
str with the correct type int.11 Note that the hinted type might not
be an exact match to the newly annotated type. For example, we
consider an error message that suggests str as correctly hinted also
if the developer fixes the error using Optional[str].

Given the above definition, we find that 89 out of 125 (71.2%)
type fixes in our study are correctly hinted by the type checker.
These hinted types can serve as a reference for APR tools to narrow
down the search space. The types of code changes correctly hinted
by the checker are shown in Figure 4b.

Figure 4c shows how the developers use the correctly hinted
types. As an example, in Figure 3c, the type checker returns the
following error message: "Incompatible return type [7]: Expected
Optional[Text] but got Union[None, Path, Text]".12 The developer
does not fix the error by using the suggested type Union[None, Path,
Text], but instead uses Optional[Union[Path, Text]]. In contrast,
the example in Figure 3b shows a case where the developer uses
the type suggested by the type checker. We find that for 64 out of
the 89 hints (71.9%) the type used by the developer is exactly as
suggested in the error message. It is also common to introduce a
value of the suggested type, e.g., by adding return -1 to a function
supposed to return int. Besides the 89 error messages that correctly
hint at the correct type, most of the remaining messages (24 out
of 36) give no hint at all. For example, this is the case for the error
classes "Undefined type", "Invalid type", and "Undefined attribute".

Answer to PQ3: Most types used in fixes (71.2%) are correctly
hinted by the type checker, and developers often follow these hints.

3.3 Implications

The three main findings of the preliminary study guide the design of
our approach as follows. (PQ1) We observe that type errors and their
fixes expose recurring patterns, which might suggest an approach
based on manually designed rules and heuristics for selecting them.
However, we also find that there is only an ambiguous mapping
from errors to fix patterns, making a rule-based approach laborious
and fragile. As a result, we decide against a rule-based and in favor
of a data-driven approach, aiming for a model that learns when
to apply which fix pattern from fixes performed by developers.
(PQ2) We find that most type errors are fixed by editing a single line,
and that this line is often localized correctly by the type checker.
Hence, we focus our work on fixing type errors in single-hunk
edits13 and exploit the localization hint given by the type error
location. (PQ3) We find that the error message provided by the type
checker often gives valuable hints for finding the fix, e.g., which
type to use. As a result, we provide the error message as an input
to our approach.

4 APPROACH

Based on the findings of our preliminary study, we design PyTy,
a data-driven approach to automatically fix static type errors in
Python using a cross-language transfer learning approach. Figure 5
shows an overview of the approach, which consists of two phases.
First, during the offline phase, we automatically collect a dataset
of type error fixes from GitHub, which we call PyTyDefects, by
combining delta debugging and gradual type checking, followed
by fine-tuning a pre-trained model [5] with PyTyDefects. Second,
during the online phase, PyTy receives code with a type error as the
input and then queries the model for fix candidates. The approach
uses the type checker to validate that the type error gets resolved
when applying a fix candidate, and then reports only fixes that are
guaranteed to remove the targeted type error.

4.1 Automated Data Gathering

To build a learning-based APR model, we must first collect a relevant
dataset as our training data. As a first step, we search for Python
repositories that are popular (≥ 100 stars), have a manageable size
(≤ 5GB), and were created between 2010 and 2021 on GitHub. We

11https://github.com/TheAlgorithms/Python/commit/6089536
12https://github.com/RasaHQ/rasa/commit/1ded5ef
13Hunks may be larger than single lines, allowing PyTy to predict some fixes that
involve multiple lines.
use the keywords “fixing+typing”, “fixing+pyre”, “fixing+mypy”, “typing+bug”, and “typing+error” to search for commits that possibly remove type errors. Next, we run the type checker before and after each such commit to find commits that indeed remove type errors. As a result, we obtain 32,330 type errors that are removed by 4,515 commits in 176 GitHub repositories.

Many of the extracted commits contain changes not directly related to fixing type errors. Moreover, a single commit often fixes multiple type errors. Figure 6 illustrates these problems with an example.14 To isolate individual type error fixes, we present a delta debugging-inspired algorithm that reduces commits into small code changes that fix a single type error. The basic idea is to iteratively reduce the set of code hunks while preserving the fact that the code change fixes a particular type error.

Algorithm 1 summarizes our approach for reducing a commit to a small set of code changes that fix the given type error. We illustrate

![Algorithm 1](image)

the algorithm using the example in Figure 6. We focus on the type error “Unbound name: basestring is used but not defined in the current scope”, reported for the line in hunk H1. The error gets fixed by changing the base class to object. Our approach considers

14Simplified from https://github.com/jazzband/django-redis/commit/5f6f383

Figure 4: Fix locations and usefulness of error messages.

(a) Top seven locations for fix patterns. (b) Top seven fix patterns correctly hinted by the type checker. (c) How developers use type checker hints.

Figure 5: Overview of the approach.
the four code hunks (H1, H2, H3, and H4) of this commit, and determines which hunks are relevant for fixing the type error with the following steps, where line numbers refer to Algorithm 1:

1. The algorithm splits the set of hunks into $H1+H2$ and $H3+H4$ with granularity two (line 8):
   a. The algorithm patches the code with only hunks $H1+H2$, which yields parsable code (lines 9 and 10).
   b. The type error disappears and there are no new errors (lines 11 and 12).
   c. There are still two hunks $H1+H2$ (line 13).
2. The algorithm splits the code hunks $H1+H2$ into $H1$ and $H2$ (line 8):
   a. The algorithm patches the code with only hunk $H1$, which yields parsable code (lines 9 and 10).
   b. The type error disappears and there are no new errors (lines 11 and 12).
3. The algorithm returns $H1$ as a minimal code change to fix the type error (line 14).

To properly track the error location while reducing the hunks, we need to keep track of how the line numbers change. To this end, we calculate the new line number based on how many lines are inserted or removed in each code hunk. We consider the error fixed if the error no longer exists at the corresponding line and column. If the error is located inside a code hunk, i.e., the code with the error is being modified, we consider the error as fixed only if all lines in the code hunk are free of errors after the change.

To ensure the quality of PyTyDefects, we apply additional filtering steps. Algorithm 1 checks that there are no new errors introduced by the code changes (line 12). The algorithm also rejects any set of code hunks that result in parsing failures (line 10). The space complexity of the algorithm is $O(2*N)$, where $N = \text{size}(D_{original})$ and the time complexity is $O(N \times \log N)$.

Running the algorithm on the 32,330 type errors gives 11,955 examples of reduced error fixes. We further filter them by keeping only fixes that (i) are relatively small ($\leq$ 512 characters and at most three changed lines), which is motivated by limitations of the neural model (Section 4.2); (ii) do not contain any error suppression; (iii) are not only deletion; (iv) are located close (i.e., within the same hunk) to the reported bug location. Finally, after applying these filters, PyTyDefects has 2,766 entries that cover ten frequent categories of type errors listed in Table 1. We select 10% (always rounding up to the nearest integer) of the entries of each error class in this final dataset as a test set, and then split the remaining fixes into 90% for training and 10% for validation.

4.2 Neural Type Error Fixing

Given the automatically extracted dataset of type error fixes, PyTy trains a neural model that predicts how to fix type errors. The input to the model is a sequence of one or more lines, i.e., the size of a single hunk, that contains a type error. The output of the approach is a fix that removes the targeted type error.

4.2.1 Base Model. Instead of learning a model from scratch, we fine-tune a model pre-trained on another APR task. Building on a pre-trained model is motivated by the fact that PyTyDefects, with 2,766 examples, is relatively small. As our base model, we use TFix [5], a learning-based APR technique trained to fix linter errors in JavaScript. We select TFix for three reasons: (i) it is already trained on a bug fix dataset of 104,804 samples; (ii) it accepts error messages as input; and (iii) the TFix authors used it to predict single line fixes, which resembles our single-hunk setup. By fine-tuning TFix, PyTy transfers the already learned knowledge to the related but different domain of Python type errors (Section 6.3). TFix itself is based on T5 [37], a transformer-based model that maps sequences of input tokens to sequences of output tokens. The simple input and output structure eliminates the need for implementing a static analysis tool to transform our code into a specific structure, such as graphs [2, 10]. Furthermore, having an unconstrained token sequence may enable the model to fix errors missed by a template-based APR approach, which is inherently limited to its set of templates (Section 6.4.2).

4.2.2 Fine-Tuning for Python Type Error Fixing. To fine-tune TFix with PyTyDefects, we follow the same input format as TFix: “fix” $\omega$ $t$ $\omega$ $m$ $\omega$ $l_k$ $\omega$; “$C$”, where “fix” and “;” are literals, $t$ is the class of type error, $m$ is the error message, $l_k$ is the line of code with the type error, $\omega$ represents a space, and $C$ represents the buggy lines of code (i.e., the single-hunk we extracted in Section 4.1). In the T5 framework, the string “fix” $\omega$ $t$ $\omega$ $m$ $\omega$ $l_k$ $\omega$; “$C$” represents the current task, and $C$ represents the input of this task. The model outputs $C'$, which we use as a replacement for $C$ to fix the type error.

4.2.3 Python Code Pre- and Post-Processing. We use the tokenizer from the Python standard library to pre-process the source code and inject special tokens for indention and dedentation. TFix uses SentencePiece [22] as its tokenizer. However, SentencePiece does not take the number of whitespaces into account, as it escapes all whitespaces into a single “$\_” symbol. Since the amount of white-space carries semantics in Python, we preserve this information by adding special tokens “<IND>” and “<DED>” into the source code before passing it to the model. Given a prediction by the model, PyTy replaces the special tokens in a post-processing step to obtain syntactically correct Python code.

4.2.4 Validating Fixes via Type Checking. Once trained, we query the model for a ranked list of the $k$ most likely fixes. To ensure that a fix suggestion given to a user indeed removes the targeted type errors.
type error, PyTy validates all candidate fixes by running the type checker on them. If and only if the targeted type error disappears and no new errors appear, the fix is suggested to the user.

5 IMPLEMENTATION

We fine-tune the t5-base (220M parameters) model of TFix for 30 epochs with a batch size of 32, and then evaluate the model that has the lowest validation loss on the validation set. The model converges at the 17th epoch. We follow the default hyperparameters of TFix [5]. When validating candidate fixes using the type checker, we sample from the model up to $k = 50$ predictions to be validated. Since PyTy validates fix candidates automatically, a user does not have to inspect these 50 suggestions, but only the first one found to successfully remove the type error. To fix 281 type errors (i.e., our test set, which amounts to a total of 174,586 lines of code) and automatically check whether the targeted type errors disappear, PyTy takes in total six hours and 44 minutes, i.e., an average of 86.2 seconds per type error fix. We perform all experiments on a server with 48 Intel Xeon CPU cores clocked at 2.2GHz, 250GB of RAM, one NVIDIA Tesla V100 GPU, running Ubuntu 18.04. Most of the time is spent on running the type checker for validating candidate fixes.

6 EVALUATION

We evaluate our PyTyDefects dataset and PyTy, our learning-based type error repair approach, focusing on the following research questions (RQs):

RQ1 How effective is our automated data gathering at producing minimal code changes that fix type errors?
RQ2 How effective is PyTy at fixing type errors?
RQ3 How do variants of PyTy compare to the full approach?
RQ4 How does PyTy compare to state-of-the-art APR techniques?

6.1 RQ1: Effectiveness of Automatic Data Gathering

6.1.1 Data analysis. To validate the effectiveness of automatically gathering PyTyDefects, two of the authors independently annotate a random sample of 100 of the 2,766 entries in the dataset. The sample contains at least one error from each class of type errors except for “call error”. Each entry is assigned one of three labels: minimal if the extracted code change fixes a type error and cannot be further reduced, correct but not minimal if the extracted code change correctly fixes a type error but is not minimal, and wrong otherwise.

6.1.2 Results. After independently labeling the 100 entries, the two annotators initially agree on 89 labels. After discussing the divergent labels and refining the labels of some entries, there is a final agreement on 94/100 minimal, 3/100 correct but not minimal, and 0/100 wrong entries. The remaining three entries with divergent labels are due to hunks that fix two type errors at once. These entries are minimal in the sense that a hunk-based reduction algorithm cannot further reduce them, but they could be further reduced by a more fine-grained reduction algorithm [18, 41]. The inter-rater agreement, as given by Cohen’s kappa coefficient [8] is 0.651, which means a substantial agreement [23].

As an example of a minimal type error fix, recall hunk H1 from the previously discussed commit in Figure 6. All changes in hunk H1 are necessary for fixing the type error. Figure 7 shows an example of a correct but not minimal commit, which includes some changes not relevant to fixing the type error.16 A single hunk updates multiple parameter type annotations of the same function. However, only one code change is relevant to fixing the type error reported for basis_to, which should be annotated Optional[Basis] instead of Basis, as it is initialized to None.

Answer to RQ1: The automated data gathering yields type error fixes that are mostly correct (97/100) and minimal (94/100), i.e., PyTyDefects provides a solid basis to train and validate PyTy.

6.2 RQ2: Effectiveness of PyTy

We evaluate the effectiveness of PyTy on all ten classes of type errors covered by our test set. We configure the approach to consider up to $k = 50$ candidate fixes. Note that users do not have to manually check all candidate fixes, but only see the first successful fix.

6.2.1 Metrics. We use two metrics to evaluate the effectiveness of PyTy. First, we compute the error removal rate, i.e., how often the approach succeeds at finding a fix that removes the targeted type error without introducing new type errors. Second, we compute the exact match rate, i.e., how often the model output is identical to the fix committed by the developer. This metric underapproximates

---

16https://github.com/kinnala/scikit-fem/commit/a555ca3
The example in Figure 9 fixes the type error in a way that matches the developer-provided fix for type error "Incompatible variable type: string is declared to have type str but is used as type bytes".

Figure 10: Fix predicted by the neural model, but not suggested to the user, as the type error "Unbound name: Name F5_DEVICE_TYPE is used but not defined in the current scope" would still exist for DEVICE_TYPE.

The error is because the variable constrained used in the format string is not defined. PyTy successfully fixes the mistake by replacing constrained with constraint, which exactly matches the developer’s fix.

The example in Figure 9 fixes the type error in a way that matches the intention of the developer but differs from the original fix. The developer fix directly passes the byte string _fmt(string) as an argument to the function lib.TCOD_console_printf_ex, avoiding the error caused by re-assigning the byte string to the variable string, which is previously annotated as type str. The PyTy-suggested fix instead declares a new variable byte_string for the byte string, and passes it to lib.TCOD_console_printf_ex as an argument.

Figure 10 shows a predicted fix that fails to remove the type error.19 The developer fix uses a variable (FS_API_DEVICE_TYPE) imported from another package. However, since the context code and the error message do not give any hint about the identifier to use, the model simply replaces it with DEVICE_TYPE. Because PyTy validates that a fix candidate removes the type error before reporting the fix to the user, this fix suggestion is not shown to users, highlighting the importance of validating fix candidates.

Finally, Figure 11 fixes the type error but changes the semantics of the code in an unintended way.20 The error is because Bot and new, which is a variable, have incompatible types. The developer fixes the error by asserting that new is of type BotUser. PyTy instead suggests a fix that declares a new variable new_Bot, which however fails to update the global Bot variable. We include this example to show that PyTy is limited by relying on the type checker as the only

---

17https://www.github.com/DragonMinded/bemaninutils/commit/438a3da
18https://github.com/libtcod/python-tcod/commit/60066f3
19https://github.com/networktocode/pyntc/commit/ebb35344e0121
20https://www.github.com/lykoss/lykos/commit/abbd35c
Table 2: Ablation study and comparison with LLMs.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Error removal (%)</th>
<th>Exact match (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-1</td>
<td>Top-5</td>
</tr>
<tr>
<td>No pre-training</td>
<td>47.3</td>
<td>57.3</td>
</tr>
<tr>
<td>Vanilla TFix</td>
<td>4.6</td>
<td>11.0</td>
</tr>
<tr>
<td>No preprocessing</td>
<td>17.8</td>
<td>23.5</td>
</tr>
<tr>
<td>Small TFix model</td>
<td>43.1</td>
<td>63.3</td>
</tr>
<tr>
<td>text-davinci-003</td>
<td>21.7</td>
<td>27.8</td>
</tr>
<tr>
<td>gpt-3.5-turbo</td>
<td>21.9</td>
<td>23.8</td>
</tr>
<tr>
<td>gpt-4</td>
<td>34.1</td>
<td>36.7</td>
</tr>
<tr>
<td>Full PyTy</td>
<td>50.9</td>
<td>66.2</td>
</tr>
</tbody>
</table>

We perform an ablation study to evaluate the effectiveness of PyTy in different configurations. The upper part of Table 2 summarizes the results discussed in the following.

No pre-training. We train the T5 model directly on PyTyDefects, i.e., without pre-training the model on the JavaScript APR tasks. The purpose of this experiment is to check if the knowledge of fixing JavaScript errors helps in fixing Python type errors. We use the same experimental setup as discussed in Section 5, except that training continues beyond 30 epochs because the evaluation loss keeps decreasing. We train the model for 100 epochs and pick the model with the lowest validation loss, which is at the 32nd epoch. The results show that pre-training the model on the JavaScript validation mechanism. Future work could address this limitation by further validating fixes by running a test suite.

6.4 RQ4: Comparison with Prior Work
6.4.1 RQ4a: PyTy vs. Large Language Models. Fixing type errors relates to general-purpose APR [25]. The following compares PyTy with large language models (LLMs), which have been shown to yield state of the art results [19, 48, 50]. PyTy and LLMs fundamentally differ in the sense that PyTy is designed and fine-tuned specifically for type error repair, whereas LLMs are trained in a task-independent manner, but typically on much more data.
Experimental Setup. We compare PyTy with three recent models offered by OpenAI: text-davinci-003, gpt-3.5-turbo, and gpt-4. Our prompt consists of five parts: a description of the task, the buggy code snippet, the type checker’s error message, the line containing the error, and a description of the expected output format.

Results. The lower part of Table 2 shows the effectiveness of different models. PyTy clearly outperforms all LLMs in terms of error removal and finding the exact developer fix. The gpt-4 model, as the most recent and largest model, is the most effective LLM. The text-davinci-003 model is slightly more effective than gpt-3.5-turbo, which may be because the latter is optimized for chat. Manually analyzing the successful fixes, we notice that the LLMs mostly fix those errors that can be fixed with a single-token edit. Instead, PyTy can fix more complex type errors.

Answer to RQ4a: PyTy is more effective than prompting general-purpose LLMs (54.4% vs. 26.4%).

6.4.2 RQ4b: PyTy vs. PyTER. Instead of targeting statically detected type errors, the recent PyTER [32] approach repairs bugs that manifest through a TypeError exception. For a comparison, consider the two subproblems that both approaches address. Subproblem 1 is detecting a type error, done by the static type checker in our approach and observing a runtime exception in PyTER. Subproblem 2 is fixing a detected type error, done by a neural model in our approach and by applying a set of repair templates in PyTER. How PyTy and PyTER address subproblem 1 differs fundamentally. While static type errors manifest without running the code, revealing a runtime type error require tests cases or a production run that triggers the error. Moreover, a single type-related problem may manifest at different locations. For example, a function that returns an incorrect value will manifest as a static type error at the return statement, but as a runtime type error at a code location that uses the value. Because of these differences, performing a direct, end-to-end comparison is neither possible nor meaningful. Instead, we quantify the overlap of the two approaches in terms of the errors they address and the fixes that they could potentially find, which answers four questions.

PyTER on PyTyDefects. 1) How many of the errors in PyTyDefects manifest via a runtime type error? We pick a random sample of 30 of all 281 fixes in our test set and inspect their commit messages. The inspection shows that for 16/30 fixes, the problem was certainly found via static type checking, e.g., because the message mentions the type checker, and for 27/30 fixes, the problem was certainly not found via a TypeError thrown at runtime. 2) How many of the type errors in PyTyDefects are in the scope of PyTER’s fix templates? The repair templates cover three kinds of fixes: adding an instanceof check, adding a type conversion, e.g., via a call to int(), adding code to catch and handle a TypeError exception. We check for each type error in our test set whether PyTER’s repair templates can be instantiated into the fix, which shows that 15/281 type errors are in scope for PyTER, whereas the remaining 266 errors are not covered by any repair template. Examples of fixes that are out-of-scope for PyTER are: (i) fixes that change a value, e.g., by modifying a string “a b c” into an array of strings [“a”, “b”, “c”], (ii) fixes that change a type annotation, e.g., from T to Optional[T], and (iii) fixes that add a call to typing.cast(). In summary, PyTER address only a small fraction of the type errors in our dataset.

PyTy on PyTER’s dataset. 3) How many of the errors in PyTER’s dataset manifest via a static type error? The Pyre type checker that PyTy builds on checks (partially) type-annotated code only. Among the 93 errors in PyTER’s dataset, 16 are in a type-annotated function, and hence, checked at all, but the type checker does not find the errors fixed by PyTER. 4) How many of the type errors in PyTER’s dataset are in the scope of PyTy’s neural model? Our approach focuses on single-hunk fixes where the type error location is inside the hunk that needs to be changed. While these assumptions commonly hold for static type errors (Section 3), only 11/93 errors in PyTER’s dataset match our assumptions. In summary, PyTy addresses only a small fraction of the type errors in the PyTER dataset.

Answer to RQ4b: Our approach and PyTER [32] are complementary in the sense that they address type errors that manifest in different ways and that they apply different kinds of fixes.

7 DISCUSSION AND THREATS TO VALIDITY

Python repositories. We select popular projects for our dataset, because recent work finds such projects to contain type annotations and type errors [9]. A different set of repositories could yield different results, in particular for the preliminary study (Section 3).

Limitations of static type checking. PyTy builds upon the Pyre type checker, which, as all static type checkers, may suffer from false positives and false negatives. A false positive, where the type checker incorrectly reports a type error in correct code, may lead to unnecessary code modifications by PyTy. Conversely, a false negative, where an error goes unnoticed by the type checker, may cause PyTy to suggest a fix that does not really solve the problem, or even worse, introduces a new problem. As a lower bound on PyTy’s effectiveness despite these limitations, we find that 54.4% of the predicted fixes exactly match the developer’s fix. Other type checkers than Pyre may find different kinds of type errors and provide different kinds of hints for fixing them. Because our approach uses the type checker as a black-box, adapting our implementation to support another type checker seems straightforward.

Type annotations. Because the type checker reports errors only in functions that are at least partially type-annotated, PyTy cannot fix errors in completely unannotated code. Despite this limitation, there is evidence that more and more code gets type-annotated, and hence, is in scope for PyTy. For example, a recent study on the evolution of type annotations [9] finds 50 type annotations per 1,000 lines of code and an increasing trend on the adoption of type annotations. Moreover, our dataset of thousands of real-world commits that address type errors shows that developers care about such errors. Finally, as described in Section 1, large companies, such as Google, Dropbox, and Meta, are actively working toward type-annotating their Python code bases.

Type errors. PyTyDefects, containing 2,766 real-world type error fixes, is filtered to contain only errors fixable with a single-hunk code change, and we cannot draw any conclusions about more complex fixes. As shown in Section 3.2.2, many real-world fixes are
local edits, which has motivated our design decision to focus on single-hunk fixes. The distribution of error classes in PyTyDefects reflects the errors that occur in practice, and does not cover all error classes that the type checker may find. Thanks to the data-driven design of PyTy, the approach should be able to fix further classes of type errors when given corresponding training data.

Future work. We plan to improve the error localization and will try different prompts to improve the performance of LLMs. Moreover, we plan to fine-tune different models beside TFix and apply PyTy to more classes of type errors. Finally, we plan to integrate our approach into an IDE.

8 RELATED WORK

Automated program repair. Earlier APR approaches [25] can be classified into heuristic repair, e.g., based on the generate-and-validate method [21, 24], and constraint-based repair, which synthesizes a patch based on constraints [29, 31]. Both techniques rely on test suites, and hence, may suffer from overfitting [36]. PyTy belongs to a more recent stream of work on learning-based repair. In contrast to the above techniques, PyTy does not require tests but uses a static type checker to validate candidate fixes. Other learning-based approaches include DrRepair [52], which fixes C compilation errors, Hoppyty [10], which represents fixes as a sequence of graph edits, and Recoder [56], based on TreeGen [42], which proposes a syntax-guided edit decoder. Compared to these GNN-based models, our approach uses a text-to-text transformer, which is easy to apply to any language. Other text-based models include SequenceR [6] and work by Tufano et al. [43], Vasic et al. [44] propose to jointly localize and repair bugs. These approaches neither benefit from pre-training nor target Python type errors. CoCoNut [26] combines multiple models using ensemble learning. Instead, our approach learns how to fix all error types in one model. Ye et al. incorporate feedback from compiling and executing tests to train a repair model [53], an idea that could also be adapted to type error repair. Finally, motivated by recent results that show general-purpose LLMs to provide competitive results [19, 48–50], we empirically compare PyTy with three LLMs (Section 6.4.1).

We are aware of two APR approaches that target type errors. Rite [39] is a template-based, data-driven approach for type errors in OCaml. Their approach builds on a specifically designed, AST-based representation of fixes, while our approach uses textual inputs and outputs. PyTER [32] is a test-based APR approach to fix runtime type errors in Python, which we empirically compare with in Section 6.4.2. Their work and ours address related but ultimately different problems: PyTER requires test cases that trigger a runtime type error, but tests may not exist at all or have low coverage (e.g., Gruber et al. [15] report a median coverage of 3.7% across 22k Python projects). In contrast, our work addresses statically detectable errors, and hence, is limited to errors that are statically detectable. In practice, we expect PyTER and PyTy to complement each other. Beyond type errors, several techniques for fixing other kinds of static analysis warnings have been proposed [3, 12, 28], which are also complementary to our work.

Type annotations and type errors. Several techniques predict types via deep learning [1, 17, 27, 30], sometimes augmented with search-based validation of predicted types [34] or static type inference [33]. While these approaches focus on predicting correct types, PyTy addresses the complementary problem of fixing type-related errors. Di Grazia and Pradel [9] show that adding type annotations to a code base often reveals statically detectable type errors, but that developers often do not find time to fix these errors. Other studies investigate the impact of using type checkers in Python [20], compare different type checkers with each other [38], and provide evidence that static typing may reduce the bug fixing effort [55]. Both Di Grazia and Pradel [9] and Rak-annouykit et al. [38] report that type-annotated repositories rarely type-check, showing the need for an APR tool for Python type errors, such as PyTy.

Transfer learning on code. Pre-trained models for source code, e.g., CodeBERT [13], GraphCodeBERT [16], CodeT5 [46] and CodeTrans [11], achieve promising results [45]. Following this paradigm, we build on TFix because it is already trained on an APR task. Another work that applies transfer learning in language models of code is VRepair [7]. They pre-train a transformer model on a large bug fix dataset for C, and then fine-tune it with a vulnerability fix dataset for C. Our work shows that the benefits of transferring knowledge not only between different fixing tasks, but also between different programming languages.

Delta debugging. Delta debugging [54] finds failure-inducing code hunks in a commit, and is widely used, e.g., for fault localization [47]. We treat hunks that fix type errors as “failure-inducing”, which is similar to prior work [4] but adopted to type errors.

9 CONCLUSION

This paper presents PyTy, the first automated repair technique targeted specifically at statically detectable type errors in Python. The design of the approach is motivated by the findings of a preliminary study. To generate a relevant dataset, we apply a combination of delta debugging and type checking, which results in PyTyDefects, containing 2,766 Python type errors and fixes. We then present cross-lingual transfer learning, which addresses the problem of having a small dataset for a deep learning model by fine-tuning an existing APR model originally trained for another task and language. Our evaluation shows the effectiveness of PyTy, e.g., by providing a fix that removes the targeted type error for 85.4% of the studied errors. Finally, as of this writing, 20 out of 30 GitHub pull requests based on PyTy-generated type error fixes have been merged by developers, demonstrating the usefulness of PyTy in practice.

DATA AVAILABILITY

The PyTyDefects dataset, the PyTy implementation, and our experimental results are available online: https://github.com/sola-st/PyTy and https://zenodo.org/records/10441045.

ACKNOWLEDGMENT

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects. We would like to thank the reviewers for their valuable feedback to improve the paper.
PyTy: Repairing Static Type Errors in Python


