

Novice Type Error Diagnosis with Natural Language Models

Chuqin Geng^{1,2}[0000-0002-3563-1596], Haolin Ye¹[0000-0002-7402-617X], Yixuan Li¹[0000-0001-9349-5476], Tianyu Han¹[0000-0001-6582-165X], Brigitte Pientka¹[0000-0002-2549-4276], and Xujie Si^{1,2,3}[0000-0002-3739-2269]

¹ McGill University

² Mila - Quebec Institute

³ CIFAR AI Research Chair

{chuqin.geng,haolin.ye,yixuan.li,tianyu.han2}@mail.mcgill.ca

{bpientka,xsi}.cs.mcgill.ca

Abstract. Strong static type systems help programmers eliminate many errors without much burden of supplying type annotations. However, this flexibility makes it highly non-trivial to diagnose ill-typed programs, especially for novice programmers. Compared to classic constraint solving and optimization-based approaches, the data-driven approach has shown great promise in identifying the root causes of type errors with higher accuracy. Instead of relying on hand-engineered features, this work explores natural language models for type error localization, which can be trained in an end-to-end fashion without requiring any features. We demonstrate that, for novice type error diagnosis, the language model-based approach significantly outperforms the previous state-of-the-art data-driven approach. Specifically, our model could predict type errors correctly 62% of the time, outperforming the state-of-the-art NATE’s data-driven model by 11%, in a more rigorous accuracy metric. Furthermore, we also apply structural probes to explain the performance difference between different language models.

Keywords: Type Error Diagnosis · Language Model · Natural Language Processing · Type System

1 Introduction

Diagnosing type errors has received much attention from both industry and academia due to its potential of reducing efforts in computer software development. Existing approaches such as standard compilers with type systems, report type errors through type checking and constraint analysis. Thus, they merely point to locations where the constraint inconsistencies can occur and such locations might be far away from the true error source. Moreover, type error localization would require programmers to understand the functionality of type systems and check which part of the code contradicts their intent. Languages such as C and Java force programmers to write annotations which make the

code neat. It also makes it easier to find the roots of type errors. Strongly typed functional languages such as OCaml and Haskell, however, programmers need not bother with annotations, since type systems automatically synthesize the types. The absence of type annotation comes at a price: novices could easily get lost in debugging their programs and locations of constraint inconsistencies from error messages can be misleading.

Joosten et al [7] suggests that beginners usually pay more attention to underlined error locations rather than error messages themselves when fixing programs. Therefore it is necessary to ameliorate the localizing performance of these type systems. Let us consider an OCaml ill-type program in 1a. Although the programmer intends to write a function that sums up all numbers from a list, they mistakenly put the empty list, [], at line 3 as a base case. This should instead be 0 as shown in Fig 1b. The compiler identifies the type error in line 5 saying that the head of the list, h, has type list rather than integer as required by the integer addition operator.

```

1         let rec sumList xs =
2             match xs with
3             | [] -> [] (* root cause *)
4             | h :: [] -> h
5             | h :: t -> h + sumList t (* misleading complaint *)
6         this expression has type 'a list but was expected of type int
7

```

(a) an ill-typed OCaml program that aims to sum all the elements from a list

```

1         match xs with
2         | [] -> 0 (* <= correct fix *)
3         | h :: [] -> h
4         | h :: t -> h + sumList t

```

(b) the fixed version of the OCaml code above

Fig. 1: A simple example of OCaml type error and its relevant fix.

This illustrates that programmer’s intent plays an important role in localizing type errors. To tackle this issue, NATE [18] proposes to use data-driven models to diagnose type errors. In this way, programmers’ intent can be learned and incorporated into machine learning models. NATE’s best model could achieve over 90% accuracy in diagnosing type errors. Although this is an exciting result, NATE’s models are evaluated with a rather loose metric and heavily rely on a considerable amount of hand-designed feature engineering. In addition, these features are designed in an ad-hoc fashion which prevents them from being directly applied to other language compilers.

Our approach adopts transformer-based language models to avoid considerable feature engineering. As we treat programs as natural language texts, these models do not rely on any knowledge or features about the specific programming language, thus they can be easily applied on any language. This method may seem to ignore the syntactic structure of a given programming language. However, we use structural probes [11] to demonstrate the structure is embedded implicitly in the deep learning models’ vector geometry in Section 4. We also propose a more rigorous metric, and show language models outperform not only standard OCaml compiler and constraints-based approaches but also the state-of-the-art NATE’s models under the new metric.

Transformer-based models have achieved great success in a wide range of domains in computer science including natural language processing. BERT [4] and GPT [15,1], popular transformer variants, have shown incredible capability of understanding natural languages. Together with its pre-training and fine-tuning paradigm, these models can transfer knowledge learned from a large text corpus to many downstream tasks such as token classification and next sentence prediction. Empirical results suggest that the performance of these language models even exceeds the human level in several benchmarks. In this work, we show how to take advantage of these powerful language models to localize type errors. First, we process programs as if they were natural language text and decompose the processed programs at the term or subterm level into token sequences so that they can be fed to language models. This allows us to turn the type error diagnosis problem into a token classification problem. In this way, language models can learn how to localize type errors in an end-to-end fashion.

Contributions We propose a natural language model-based approach to the type error localization problem. Our main contributions are as follows:

- Without any feature engineering or constraints analysis, we apply different language models including BERT, CodeBERT, and Bidirectional LSTM to type error localization.
- We study training methodology such as positive/negative transfer to improve our models’ performance. Instead of using a loose evaluation metric as proposed in previous work, we define a more rigorous, yet realistic, accuracy metric of type error diagnosis.
- Empirical results suggest that our best model can correctly predict expressions responsible for type error 62% of the time, 24 points higher than SHERLoc and 11 points higher than the state-of-the-art NATE tool.
- We study the interpretability of our models using structural probes and identify the link between language models’ performance with their ability of encoding structural information of programs such as AST.

We start by presenting the baseline, our model architecture and the structural probe in Section 2. Section 3 introduces the dataset and evaluation metric, while Section 4 presents the experiential results and our discussion. Then, Section 5 gives an overview of related work. Finally, Section 6 concludes the whole paper and proposes some directions for future work.

2 Approach

In this section, we introduce deep learning-based language models including RNN, BERT, and CodeBERT. We take advantage of the pre-training and fine-tuning paradigm of language models and show how to transform the type error diagnosis problem to a token classification problem, a common downstream task in fine-tuning. We also present the structural probe which allows us to find the embedded structural information of programs from models’ vector geometry.

2.1 Language models

Deep learning has achieved great success in modelling languages since the invention of recurrent neural networks (RNNs) [9]. RNNs adopt “internal memory” to retain information of prior states to facilitate the computation of the current state. Unlike traditional deep neural networks, the output of RNNs depends on the prior elements within the sequence which make them ideal for processing sequential inputs such as natural languages and programs.

In this study, we also choose a bidirectional long-short term memory (Bidirectional LSTM) [16] as our baseline model. However, RNNs are known to have several drawbacks such as a lack of parallelization and weak long-range dependencies. These two limitations are later addressed by the self-attention mechanism introduced by the transformer. Self attention [20] is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. Transformers also follow an encoder-decoder architecture as other successful neural sequential models. Both its encoder and decoder have been studied and shown great capabilities for modelling natural languages and solving many downstream tasks.

BERT, which stands for Bidirectional Encoder Representations from Transformers takes advantage of the encoder part of the transformer while the GPT-n series are based on the decoder. In this work, we focus on BERT rather than GPT-3 [1] for several reasons. First, BERT requires a fine-tuning process which alters the pre-trained model for specific downstream tasks. This fits our formalization of treating type error diagnosis as a downstream task. Second, the size of GPT-3 is enormous compared to BERT, making it hard to train and infer. Third, BERT is an open-source tool and easily available for users to access while GPT-3 is not open-sourced.

2.2 The pre-training and fine-tuning scheme

The pre-training and fine-tuning scheme allows machine learning models to apply knowledge gained from solving one task to different yet related tasks. Compared to fine-tuning, pre-training is more essential as it determines what knowledge is learned and stored in machine learning models. As a result, there are some recent works on improving the pre-training scheme of language models.

BERT stands out by proposing two critical unsupervised tasks during pre-training - Masked Language Modeling (MLM) and Next Sentence Prediction

(NSP) [4]. MLM requires the model to predict masked-out tokens conditioned on other tokens within sentences whereas NSP forces the model to predict if the input two sentences are next to each other in the original document. These two training tasks or objectives allow the model to understand the natural language from a statistical perspective, and empirical results of BERT have shown that pre-training on large text corpus using these two objectives facilitates a wide range of downstream tasks.

2.3 Type error diagnosis as token classification

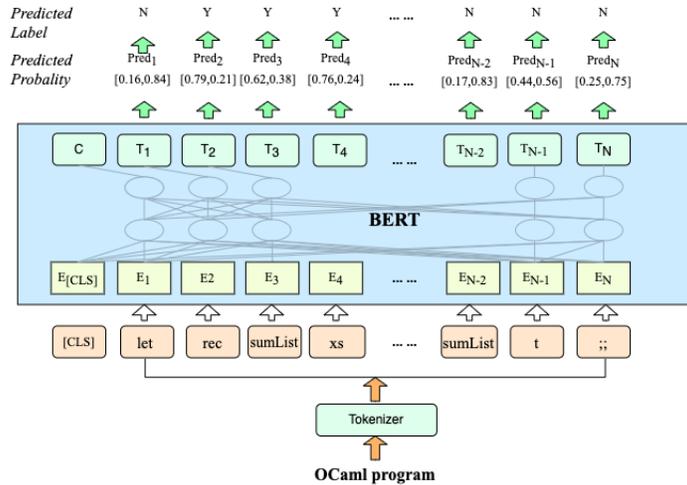


Fig. 2: The type error diagnosis as a token classification task. After an input program is split into a sequence of tokens by a tokenizer, each token is fed into BERT to get an embedding representation. A simple classification head takes each token representation and outputs a predicted probability which indicates the model’s belief of the current token being related to type error.

Token classification [4] is a downstream task which uses a pre-trained Bert model with a token classification head on top to make a prediction for each token in a sentence. One of the most common token classification tasks is Named Entity Recognition (NER). The goal of NER is to find a label for each entity in a sentence, such as a person, location or organization. Type error diagnosis can be naturally viewed as a token classification problem. Note that type error diagnosis attempts to find type error locations within a piece of code, so we can reformulate it to a token classification task if we assign label 1 to all the tokens that contribute to the type error and label 0 to those tokens that are unrelated to type error. Figure 2 gives an overview of using token classification to achieve

type error diagnosis. As a fine-tuning task, token classification requires labelled data to provide ground truth to help the model learn. In the context of type error diagnosis, this means we need to have a dataset consisting of many ill-typed programs along with their true type error locations. We will discuss our dataset more in Section 3.

Given that large language models are extremely expensive to train, even for industrial companies, a common practice is to fine-tune pre-trained large language models on a new dataset. In our case, we choose different configurations of BERT to explore the optimal model for type error diagnosis. Our models are as follows:

- **Bidirectional LSTM:** The model is trained directly on the fine-tuned dataset from scratch. This model serves as our baseline.
- **BERT from scratch:** To compare with Bidirectional LSTM, we train the BERT to do token classification from scratch, without any pre-training process.
- **BERT Small, BERT Medium, BERT Base, and BERT Large:** As the name suggests, these four models are different in terms of size. Although they are pre-trained on the same dataset, we hypothesize that the size would affect the representation power of models and therefore would affect the performance of type error diagnosis.
- **CodeBERT:** CodeBert [5] is a pre-trained bimodal model for programming language (PL) and natural language (NL). It is pre-trained on several programming languages including Ruby, Javascript, Go, Python, Java, and PHP. As it is pre-trained on such programming languages that ask programmers to specify the type, we postulate that it may not work well on OCaml programs. However, we are still curious to see if these programming languages may share some patterns with OCaml which can enhance its error localization ability during the fine-tuning process.
- **BERT pre-trained on OCaml:** Since BERT is pre-trained on natural language texts which do not contain OCaml programs, we collect two datasets of OCaml programs, one from industry and the other one from students' homework submission. Then we pre-train BERT Base and BERT Large on them with the same training objectives. This technique is also called domain shift which could help the model perform better on downstream tasks which have different data distribution from that of the original input. Together with CodeBERT, these models allow us to explore how domain shift affects models' performance.

2.4 Structural probe

We attempt to use the structural probe method [11] to find structural information embedded implicitly in the deep learning models' vector geometry.

In deep learning, each token has a vector representation after feeding into the model. The method finds a distance metric that can approximate the result of

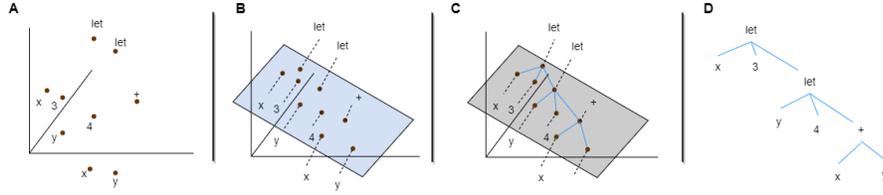


Fig. 3: Syntax tree of example OCaml program “let x = 3 in let y = 4 in x + y”

the distance metric defined by the syntax tree from applying to any two tokens of a program. More specifically, it defines a linear transform of the space in which squared L_2 distance between vectors best reconstructs tree path distance, and thus the structure of the tree is demonstrated by the geometry of the vector space. Figure 3 gives an overview of using the structural probe to reconstruct tree structure information of programs.

3 Dataset and evaluation metric

In this section, we present the datasets and the evaluation metric that we use in our work. The training datasets that we use are the same ones used in the baseline. However, we propose a different metric of accuracy. Notice this metric has been applied to every model, so data produced by NATE’s models may look different from their original paper. To explore the capability of language models, we also create two pre-training datasets consisting of over 370,000 OCaml programs in total.

	Num. of programs	Average num. of tokens	Has ground-truth label	Usage
NATE SP14	2,712	136	Yes	Fine-tuning (training and testing)
FA15	2,365	133	Yes	
GitHub	350,000	121	N/A	Pre-training
Homework	20,000	99	N/A	

Table 1: Statistics of pre-training and fine-tuning datasets

3.1 Pre-training dataset

The pre-training procedure plays an important role in transformer-based language models. The purpose of pre-training is to train the model on large-text corpus in an unsupervised fashion. After pre-training, models should have weights that encode the probabilities of a given sequence of words occurring in sentences.

The success of modern language models is often attributed to large pre-training datasets. Motivated by this observation, we collect the following two datasets.

- **GitHub-OCaml dataset** Based on the default ranking configuration provided by GitHub, we collected top-500 OCaml GitHub repositories, which has been identified as an instance of “fair use” [13]. Given that most samples of our GitHub dataset are from the industry, we perform some post-processing work by filtering out programs that are automatically generated by lexer and parser. Even though, some of them are still quite different from programs written by novice programmers. The resulting dataset contains over 350,000 OCaml programs.
- **Student-OCaml dataset** To collect OCaml programs written by novice programmers, we collected around 2, 000 homework submissions made by over 350 students from an undergraduate programming languages course taught at McGill University. Each homework submission consists of 10 sub-tasks (e.g., functions for a specific coding question) on average. This gives us 20,000 OCaml data samples.

In Section 4, we study how pre-training on these two datasets affects the performance of type error diagnosis.

3.2 Fine-tuning dataset

Fine-tuning, on the other hand, requires labelled data as supervision to facilitate model learning. This means we need a set of ill-typed programs along with the correct locations of type errors as ground truth. Manual ground truth annotation and ill-typed program collection can be troublesome. Fortunately, NATE provides a dataset consisting of 5,000 labelled programs that cover many types and locations of the errors that beginners make in practice, together with the corresponding fixes.

NATE’s dataset was collected from an undergraduate Programming Languages course at UC San Diego in Spring 2014 and Fall 2015, which are named SP14 and FA15 respectively. Besides providing supervision, NATE’s dataset can also be used as a test-bed so we can compare our models with NATE’s fairly.

3.3 Evaluation metric

NATE processes programs to sub-tree sets and then filters the sub-trees that are not related to type errors to get true error locations — the ground truths. As a consequence, if a large sub-tree, T , is the ground truth, many of its sub-trees are treated as true error locations as well. Then, if a model predicts any of these sub-trees, the prediction accuracy would be 100% under the NATE’s metric. We illustrate this using an example as shown in Fig 4. As we can see in 4a, the model blames the token, `clone`, which happens to match the error token, `clone`, in 4b. It is not easy to see why `clone` is an error location because we only highlighted the union of all error spans. It ends up to be one since it is a strongly-related

```

1 let rec clone x n = if n <= 0 then []
2                       else x :: (clone x (n - 1))
3 let rec helper x = if x = 0 then 1 else 10 * helper (x - 1)
4 let padZero l1 l2 =
5   if (List.length l1) < (List.length l2)
6   then ((clone "0" List.length l2) - (List.length l1)) :: l1
7   else ((clone "0" List.length l1) - (List.length l2)) :: l2

```

(a) An ill-typed OCaml program. Highlighted tokens on line 6 and 7 are predictions made by NATE’s model.

```

1 let rec clone x n = if n <= 0 then []
2                       else x :: (clone x (n - 1))
3 (*let rec helper x = if x = 0 then 1 else 10 * helper (x -
4   1) *)
5 let padZero l1 l2 =
6   if (List.length l1) < (List.length l2)
7   then (((clone 0 ((List.length l2) - (List.length l1))) @ l1), l2)
8   else (((clone 0 ((List.length l1) - (List.length l2))) @ l2), l1)

```

(b) Fixed version of the OCaml program above. Highlighted tokens on line 6 and 7 form the ground truth, whereas the change on line 3 does not.

Fig. 4: An ill-typed OCaml program and its corresponding fix.

subtree of the ground truth highlighted in blue at line 6. Therefore, the prediction matches to the ground truth, resulting in an accuracy of 100% under NATE’s metric. However, such prediction is merely a tiny portion of the union of all error spans highlighted in blue which makes this an over-evaluation.

As a result, NATE’s metric overestimates the prediction accuracy of not only its own machine learning models but also our language models. To visualize the over-evaluation from a data point of view, we test NATE’s models and some of our models under NATE’s metric. We use BERT Small, Base and Large models, and they achieve Top-3 accuracies of 80%, 84% and 87% respectively. Generally speaking, they are comparable to NATE’s models, whose accuracies range from 84% to 90%.

We solve the overestimation issue by treating programs as consecutive token sequences rather than trees. Hence by counting the number of correctly predicted tokens, we can get a more precise and strict accuracy between 0% and 100% rather than just 0 (miss) or 1 (hit). To be more specific, our models estimate the probabilities of type error blame for each token in a binary classification setting. By converting predicted probabilities to label 0 or 1 using a default threshold value of 0.5, gives us a collection of predicted token sequences, P . By transforming the ground truth denoted by L to token sequences as well, the correctly predicted token set is simply the intersection of them, $P \cap L$. However, a trivial prediction which simply predicates each token as type error, i.e. $P =$

$\{1, 1, 1, \dots, 1, 1\}$, could achieve 100% accuracy due to $P \wedge L = L$. To prevent this from happening, we divide the size of $P \wedge L$ by that of $P \vee L$.

$$Accuracy(P, L) = \frac{|P \wedge L|}{|P \vee L|}$$

4 Evaluation

In this section, we empirically evaluate several approaches to type error diagnosis with particular focus on the following research questions:¹

RQ1: How well do language models and other baseline methods perform on type error diagnosis?

RQ2: To what extent do model size and transfer learning affect models’ performance?

RQ3: How well do language models generalize to unseen data?

RQ4: Does the model’s ability of encoding structure information contribute to prediction accuracy?

Implementation and training. We implement our experiments using PyTorch, Tensorflow and HuggingFace library. We use a batch size of 32 for both pre-training and fine-tuning processes. For pre-training, we pre-train BERT Base and BERT Large on both pre-training datasets for 10 epochs. For fine-tuning, all BERT models are fine-tuned on NATE’s training dataset for 30 epochs. We set the initial learning rate to 0.00003 and use a scheduler to alter the learning rate during fine-tuning. To avoid stochasticity, we run each experiment three times and take the average. All our models are trained on a Tesla P100-PCIE-16GB GPU.

Configurations of language models. We explore different configurations of BERT to find the best model. We call BERT Base and BERT Large (BERT+) pre-trained on the homework OCaml dataset OCamlBERT Base (OBERT) and OCamlBERT Large (OBERT+). As for the BERT Base pretrained on the GitHub OCaml dataset, we call it BERT pre-GitHub (PBERT). We also trained a BERT Base from scratch without leveraging pre-trained weights and name it BERT Init (IBERT).

Baselines. We compare our language model-based approach with three baselines as follows:

- OCAML, which extracts the type error location from the error message from the standard OCaml compiler. It is worth noting that the standard OCaml compiler stops compiling immediately when any type check fails and thus cannot report multiple errors.
- SHERRLOC, which identifies the minimum set of locations to patch a type error using Bayesian inference [10].

¹ Our artifact is available at [6].

- NATE, which predicts the top-K most likely ASTs that contribute to the type errors based on 282 hand-designed features [18]. Specifically, NATE uses five different machine learning models — logistic regression (LOGISTIC), decision tree (TREE), random forests (FOREST), and two multi-layer perception models (MLP-10 and MLP-500) with a single hidden layer of 10 and 500 neurons, respectively.

4.1 Performance of different models (RQ1)

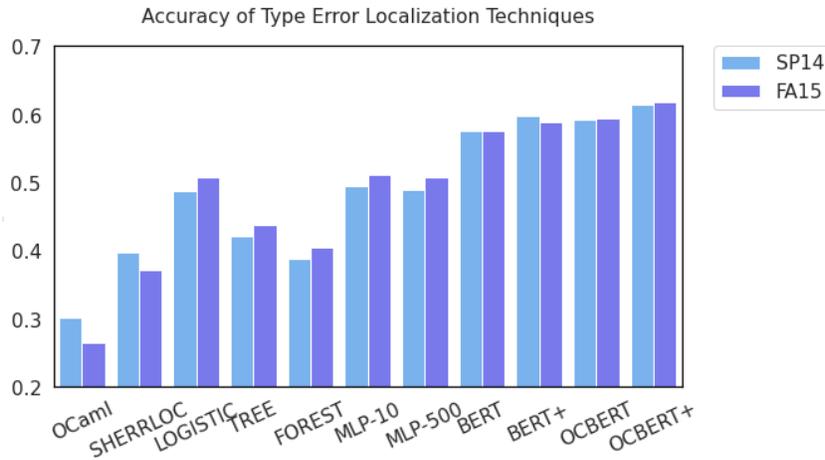


Fig. 5: Comparison of accuracy of type error diagnosis methods.

Fig 5 summarizes our main results. We observe that both optimization-based approaches like SHERRLOC and data-driven based approaches like NATE and various language models (i.e., BERT, BERT+, OCBERT, OCBERT+) outperform the standard OCaml compiler in terms of localizing root causes of type errors. Furthermore, data-driven approaches generally outperform optimization-based approaches, indicating that data plays a more important role compared to the pure optimization algorithm as adopted by SHERRLOC. Among the five models used by NATE, it is a bit surprising that LOGISTIC achieves similar performance as multi-layer perceptrons. We believe this is due to the rich hand-designed features which make simple models like logistic regression very effective.

Nevertheless, we observe that our language model-based approaches significantly outperform NATE, which suggests that the embeddings learned in an end-to-end fashion are more effective than hand-designed features.

Both NATE’s and BERTs’ output can be interpreted as a probability. Normally, we set the threshold to be 0.5, so if the output probability is greater than 0.5, the prediction will be 1, and 0 otherwise. The change in accuracy of

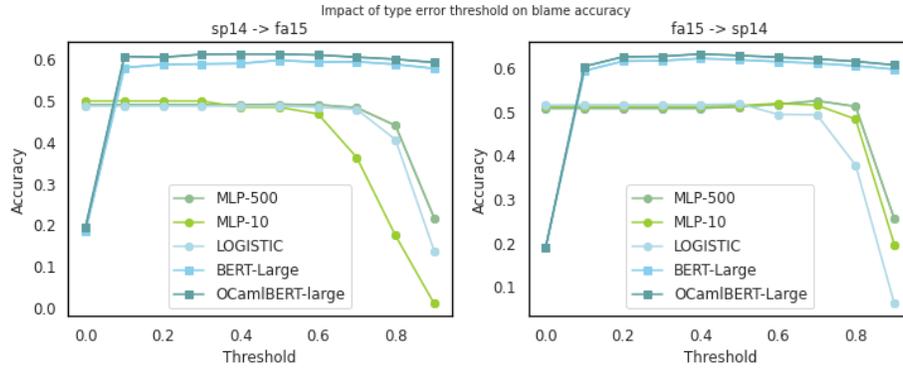


Fig. 6: Impact of different thresholds on accuracy.

models along with varying thresholds is reported in Fig 6. We notice that if we increase the threshold, BERTs’ accuracy is robust whereas NATE drops significantly. This suggests that BERT models are much more confident with their predictions compared to NATE.

4.2 Effectiveness of model sizes and transfer learning (RQ2)

Larger model leads to higher accuracy (not overfitting). To study the effectiveness of different model sizes, we evaluate four models of different sizes — Small (L=4, H=256), Medium (L=8, H=512), Base (L=12, H=768), and Large (L=16, H=1024). Fig 7(a) presents training loss curves of the four models of different sizes. This is somewhat expected since larger models usually tend to have lower training loss but may have an overfitting concern. This then may not lead to better accuracy.

We further evaluate the testing accuracies of models of different sizes, which is summarized in Table 2. The top half of Table 2 shows the testing accuracies of four BERT models on four different train/test setups. The accuracy increases consistently when the model size increases on all train/test setups. This is very interesting because the train/test dataset is fixed with only the model size increasing, that is, with the same dataset, the larger model usually leads to higher accuracy instead of overfitting.

Positive/Negative transfer of learning To study this objective, we focus on BERT, BERT pre-GitHub, CodeBERT, and OCamlBERT. Fig 7(b) presents the training loss curves of these four models. Since OCamlBERT is pretrained on only 20k code samples whereas BERT pre-GitHub is pretrained on 350k samples, it is quite surprising to notice that OCamlBERT has the lowest training loss whereas BERT pre-GitHub has the highest one. This is kind of counter-intuitive as models usually perform better when trained on a larger dataset. The bottom part of Table 2 shows the testing accuracies of four BERT models on four different train/test setups. BERT pre-GitHub reports 51% accuracy, 8 points lower

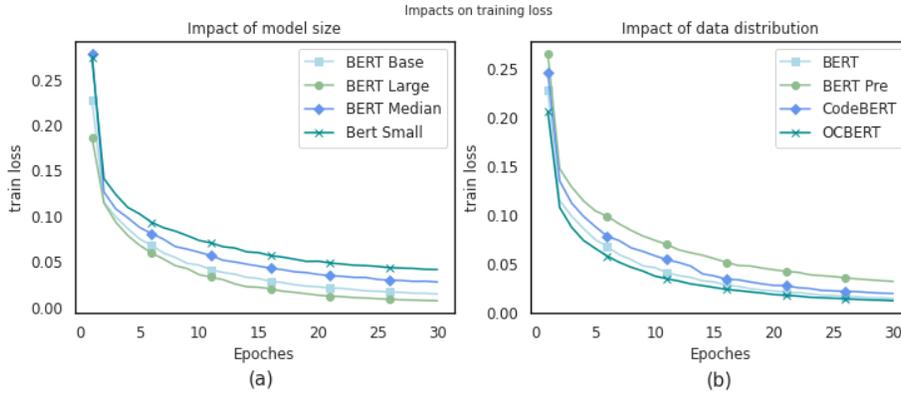


Fig. 7: Impact on training loss. (a) shows how model size affects training loss while (b) illustrates how data distribution affects training loss.

than OCamlBERT which is consistent with training loss curves. The difference in accuracy could be explained by the positive/negative transfer of learning effect. As OCamlBERT is pretrained on code samples written by students/novice programmers (although from different universities working on different programming assignments), the similar data distribution in the fine-tuning process affects positively on accuracy [22]. On the other hand, transfer learning impacts CodeBERT’s and BERT pre-GitHub’s accuracy due to disparate data distribution.

	SP14/SP14 (Acc)	FA15/FA15 (Acc)	SP14/FA15 (Acc)	FA15/SP14 (Acc)
BERT Small	68.83	63.05	50.45	51.37
BERT Medium	71.53	65.08	53.39	53.52
BERT Base	74.10	69.89	57.62	57.52
BERT Large	77.57	74.36	59.72	58.89
Bi-LSTM	44.15	40.51	7.25	8.79
BERT Init	60.70	57.02	45.37	43.98
BERT pre-GitHub	68.52	59.59	51.84	51.43
CODEBERT	71.94	69.35	56.40	55.98
OCamlBERT Base	74.72	70.11	59.24	59.34
OCamlBERT Large	78.76	74.78	61.40	61.84

Table 2: Accuracies of different models evaluated on four train/test setups.

4.3 Generalization ability of language models (RQ3)

The generalization ability is an important property of our models as it measures how well a trained model performs on unseen program questions [12].

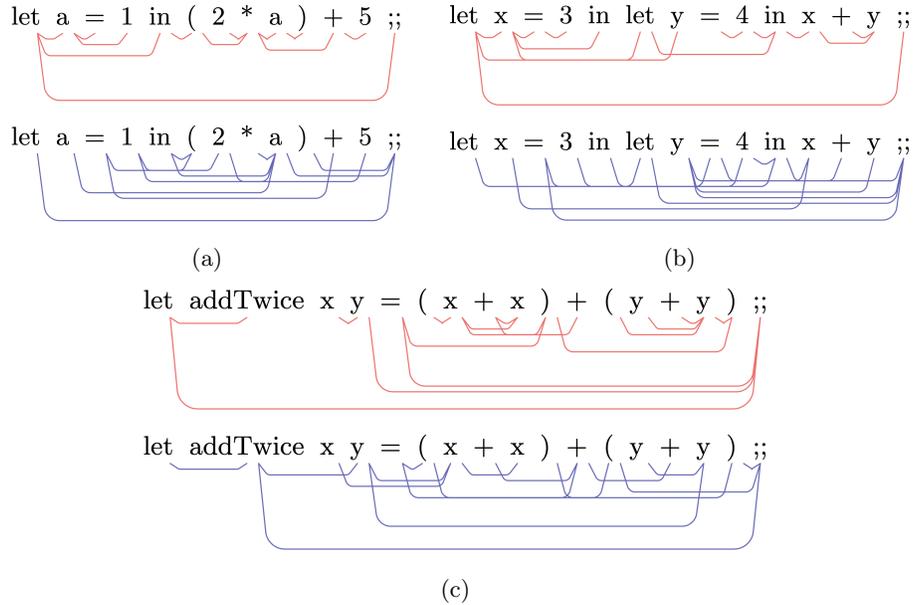


Fig. 8: AST edge reconstruction from learned embeddings (edges in red are reconstructed by BERT Large model; edges in blue are reconstructed by Bidirectional LSTM model).

To study this property, we focus on accuracy drops when evaluating different program problem sets, for instance, training on SP14 yet evaluating on FA15. We calculate accuracy drops using the difference between the first and last two columns of Table 2. We observe that relatively simple language models such as the Bidirectional LSTM model (Bi-LSTM) experience a large accuracy drop of over 30% on unseen data. In addition, it achieves only 7.25% and 8.79% accuracies on the generalization tasks, which makes it almost useless in the real world. In contrast, the severest accuracy drop of BERTs accuracy is merely 17%. This indicates that BERTs may have learned more robust and critical features which facilitate localizing type errors compared to Bi-LSTM.

In short, we should always consider large and powerful language models rather than small and simple ones when solving difficult tasks such as type error diagnosis.

4.4 Explaining performance difference by the structural probe (RQ4)

We use the structural probe to reconstruct structural information of programs based on both BERT’s and Bi-LSTM’s embedding representation. We hypothesize the difference in the ability of encoding structure information of programs could explain the performance gap between these two models. To gain some

insights, we conduct a number of qualitative case studies. Fig 8 shows three examples of reconstructed AST of OCaml program using the structural probe.

Although the reconstructed ASTs are not adequate, we observe AST reconstructed from BERT embeddings turns to have many more meaningful structures than the AST reconstructed from Bi-LSTM embeddings. To be more specific, in Fig 8(a), BERT’s AST connects edge (+, 5), (2, *) and (*, a) whereas Bi-LSTM’s AST only has one meaningful edge (*, a). Similarly, in Fig 8(b), BERT’s AST has an edge (+, y) while Bi-LSTM’s AST fails to do so. In example (c), BERT’s AST has an edge (+, x) and (+, y) which Bi-LSTM’s AST omits. Compared to Bi-LSTM, BERT is able to encode richer structural information, which may explain the huge 54% performance gap between these two models.

5 Related Work

Some recent works on type error diagnosis, such as SHErrLoc [10] and Mycroft [21], aim to analyze a set of typing constraints to find their minimum weight subsets [8]. A minimum weight subset means omitting this subset will make the remaining constraints satisfiable and the subterms yielding the minimum weight subset inherit the blame. However, this approach has a few disadvantages. First, they are limited in terms of language choice. Different languages tend to employ different type systems and constraints. Thus, an approach designed for one type system can be hard to transfer to others. Secondly, the weights assigned are based on researchers’ prior knowledge of the most likely errors instead of the most likely mistakes in practice [10]. Moreover, constraint-based approaches could blame a number of locations equally without taking the author’s intent into account.

In contrast, data-driven approaches such as NATE employ machine learning models to learn to localize type error from a large data set. While constraint analysis is not mandatory, NATE’s machine learning models require considerable feature engineering. To be more specific, NATE employs over 282 hand-designed features annotated by human experts which are then fed into machine learning models to make the final prediction. However, NATE and other data-driven approaches still suffer from some disadvantages mentioned above. Although NATE doesn’t perform constraint analysis, feature engineering also requires prior knowledge, making it difficult to transfer to other type systems. In addition, data-driven methods may implicitly consider the programmer’s intent when making predictions, but there is no guarantee that such intent can be understood by models. In our study, we also show that the accuracy metric of NATE can be problematic in certain conditions.

There are also approaches that provide instructions to help novice programmers debug. Seidel et al. creates a dynamic model that generates counterexample witness inputs to show how the program goes wrong [17]. When given a function with type errors, the algorithm symbolically executes the program and synthesizes witness the wrong values. Then the procedure is extended to a graph that

shows the witness execution. Experimental results suggest their algorithm can generate witnesses 88% of the time and in these successful programs, the algorithm yields counterexample successfully 81% of the time. The advantage of this algorithm is that by using graphs and counterexamples, students can learn how to write code easily and understand the logic of the programming language. However, people who are familiar with the language but not that skilled, do not need such detailed suggestions. All they need is the precise location of the error. Chen et al. develops a type debugging system that asks programmers to provide type specifications during the debugging process and then generate suggestions that help to fix the type error [2]. The advantage of this system is instead of aiming exclusively at the removal of type errors, it collects user feedback about result types to give useful suggestions, which include almost all possible corrections. This will help novices to debug more easily as they only need to choose from options given by the system. However, to achieve their goal, the authors systematically generate all potential type changes, which, when compared to our model, is more time-consuming in construction and needs more human judgment to make corrections.

There are also approaches which adopt SAT and SMT solvers to solve the type error localization. Pavlinovic et al. designs an algorithm that finds all minimum error sources, where 'minimum' is defined in terms of a compiler-specific ranking criterion [14]. With these error sources, a compiler is able to offer more useful reports. Then the authors try to reduce the search for minimum error sources to an optimization problem by implementing weighted maximum satisfiability modulo theories (MaxSMT). In this way, they leverage SMT solvers, making it easier to extend to multiple type systems and abstract from the concrete criterion that is used for ranking the error sources. The evaluation results on existing OCaml benchmarks for type error localization are also quite promising. In another work [8], Jose et al. aims to reduce the error localization problem to a maximal satisfiability problem (MAX-SAT), which finds the maximum number of clauses that are simultaneously satisfied by an assignment. Three steps are involved when an error should be reported. First, it encodes the denotation of a bounded unrolling of the program to a boolean formula. Then they construct an unsatisfiable formula for the failing program execution. In the last step, a MAX-SAT solver is used to find the largest set of clauses that can be satisfied at the same time, after which they output complement set as result, which is treated as potential locations of type error. Experimental results suggest the algorithm can find a few lines of code that are probable to be blamed for type error. Compared with our algorithm, the location it gives is too general. For novices, it is difficult for them to find the precise location of the type error when given such a large span of possible locations.

There are some other works that aim to diagnose the root causes of programs with typing errors. Chitil et al. uses a compositional type explanation graph created based on the Hindley-Milner type system [3]. More specifically, this work relies on structural type information such as trees with principal typings. Tsushima et al. builds a type debugger without implementing any dedicated

type inferencer [19]. The type debugger avoids re-implementing an independent type inference algorithm by leveraging the compiler’s type inference engine. In contrast to their work, we train natural language models to capture patterns in code changes. Our models do not require additional information beyond the code and can predict multiple error locations simultaneously. Our approach provides an orthogonal angle for (novice) type error diagnosis, and we believe that incorporating explicit type information can further improve our current approach.

6 Conclusions and Future Work

Many techniques have been developed to address the type error localization problem. While most of them employ static analysis on programs such as SHERRLOC, NATE’s success suggests that data-driven methods are also promising. Our experimental results suggest transformer-based language models outperform the state-of-art NATE and SHERRLOC under a stricter yet more realistic accuracy metric.

Although being a black-box model, we show that language models can encode structural information of programs which may explain their better performance. Moreover, our models simply view a program as a sequence of tokens, thus they do not rely on any special knowledge of OCaml. It is the large amount of data (programs in our context) that plays an essential role in the performance. Since no feature engineering and constraints analysis are required, our approach can be transferred to other programming languages easily. We plan to investigate the effectiveness of our model on new languages like Go and Rust in the future. Through experiments, we identify several factors which help improve model accuracy such as size and positive transfer. We believe these factors may also be beneficial to solving other programming language-related tasks using language models.

In this work, our approach treats programs as natural language texts. This, however, fails to utilize the structural information of programs. Although we show language models could encode some structures, it is unclear how the encoded structural information leads to the final prediction. In contrast, constraint-based approaches such as SHERRLOC take advantage of structural information and have much better interpretability. We plan to explore how to combine language models and the structural information of programs in our future work.

Acknowledgement

We thank the anonymous reviewers for their insightful comments. This work was supported, in part, by Individual Discovery Grants from the Natural Sciences and Engineering Research Council of Canada, the Canada CIFAR AI Chair Program, and Social Sciences and Humanities Research Council (SSHRC).

References

1. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. *CoRR* **abs/2005.14165** (2020), <https://arxiv.org/abs/2005.14165>
2. Chen, S., Erwig, M.: Guided type debugging. In: Codish, M., Sumii, E. (eds.) *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings. Lecture Notes in Computer Science*, vol. 8475, pp. 35–51. Springer (2014). https://doi.org/10.1007/978-3-319-07151-0_3
3. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. *SIGPLAN Not.* **36**(10), 193–204 (oct 2001). <https://doi.org/10.1145/507669.507659>, <https://doi.org/10.1145/507669.507659>
4. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. pp. 4171–4186. Association for Computational Linguistics (2019). <https://doi.org/10.18653/v1/n19-1423>
5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages. In: Cohn, T., He, Y., Liu, Y. (eds.) *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020. Findings of ACL*, vol. EMNLP 2020, pp. 1536–1547. Association for Computational Linguistics (2020). <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
6. Geng, C., Ye, H., Li, Y., Han, T., Pientka, B., Si, X.: Novice Type Error Diagnosis with Natural Language Models - Artifacts (Aug 2022). <https://doi.org/10.5281/zenodo.7055133>, <https://doi.org/10.5281/zenodo.7055133>
7. Joosten, S., van den Berg, K., van Der Hoeven, G.: Teaching functional programming to first-year students. *J. Funct. Program.* **3**(1), 49–65 (1993). <https://doi.org/10.1017/S0956796800000599>
8. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Hall, M.W., Padua, D.A. (eds.) *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. pp. 437–446. ACM (2011). <https://doi.org/10.1145/1993498.1993550>
9. Li, H.: Deep learning for natural language processing: Advantages and challenges. *National Science Review* **5**(1), 24–26 (2017). <https://doi.org/10.1093/nsr/nwx110>
10. Loncaric, C., Chandra, S., Schlesinger, C., Sridharan, M.: A practical framework for type inference error explanation. In: Visser, E., Smaragdakis, Y. (eds.) *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. pp. 781–799. ACM (2016). <https://doi.org/10.1145/2983990.2983994>

11. Manning, C.D., Clark, K., Hewitt, J., Khandelwal, U., Levy, O.: Emergent linguistic structure in artificial neural networks trained by self-supervision. *Proc. Natl. Acad. Sci. USA* **117**(48), 30046–30054 (2020). <https://doi.org/10.1073/pnas.1907367117>
12. Neyshabur, B., Bhojanapalli, S., McAllester, D., Srebro, N.: Exploring generalization in deep learning. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. pp. 5947–5956 (2017), <https://proceedings.neurips.cc/paper/2017/hash/10ce03a1ed01077e3e289f3e53c72813-Abstract.html>
13. O’Keefe, C., Lansky, D., Clark, J., Payne, C.: Comment regarding request for comments on intellectual property protection for artificial intelligence innovation (2019), <https://perma.cc/ZS7G-2QWF>
14. Pavlinovic, Z., King, T., Wies, T.: Finding minimum type error sources. In: Black, A.P., Millstein, T.D. (eds.) *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. pp. 525–542. ACM (2014). <https://doi.org/10.1145/2660193.2660230>
15. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019)
16. Schuster, M., Paliwal, K.: Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* **45**(11), 2673–2681 (1997). <https://doi.org/10.1109/78.650093>
17. Seidel, E.L., Jhala, R., Weimer, W.: Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In: Garrigue, J., Keller, G., Sumii, E. (eds.) *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. pp. 228–242. ACM (2016). <https://doi.org/10.1145/2951913.2951915>
18. Seidel, E.L., Sibghat, H., Chaudhuri, K., Weimer, W., Jhala, R.: Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.* **1**(OOPSLA), 60:1–60:27 (2017). <https://doi.org/10.1145/3138818>
19. Tsushima, K., Asai, K.: An embedded type debugger. In: Hinze, R. (ed.) *Implementation and Application of Functional Languages*. pp. 190–206. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
20. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. pp. 5998–6008 (2017), <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>
21. Zhang, D., Myers, A.C.: Toward general diagnosis of static errors. In: Jagannathan, S., Sewell, P. (eds.) *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*. pp. 569–582. ACM (2014). <https://doi.org/10.1145/2535838.2535870>
22. Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., Xiong, H., He, Q.: A comprehensive survey on transfer learning. *Proc. IEEE* **109**(1), 43–76 (2021). <https://doi.org/10.1109/JPROC.2020.3004555>