Statistical Type Inference for Incomplete Programs

Yaohui Peng
pengyaohui@whu.edu.cn
School of Computer Science, Wuhan University
China

Jing Xie
xj199704@whu.edu.cn
School of Computer Science, Wuhan University
China

Qiongling Yang
yangqiongling@whu.edu.cn
School of Computer Science, Wuhan University
China

Hanwen Guo
hanwen.guo@whu.edu.cn
School of Computer Science, Wuhan University
China

Qingan Li
qingan@whu.edu.cn
School of Computer Science, Wuhan University
China

Jingling Xue
j.xue@unsw.edu.au
School of Computer Science and Engineering, University of New South Wales
Australia

Mengting Yuan*
ymt@whu.edu.cn
School of Computer Science, Wuhan University
China

ABSTRACT

We propose a novel two-stage approach, Stir, for inferring types in incomplete programs that may be ill-formed, where whole-program syntactic analysis often fails. In the first stage, Stir predicts a type tag for each token by using neural networks, and consequently, infers all the simple types in the program. In the second stage, Stir refines the complex types for the tokens with predicted complex type tags. Unlike existing machine-learning-based approaches, which solve type inference as a classification problem, Stir reduces it to a sequence-to-graph parsing problem. According to our experimental results, Stir achieves an accuracy of 97.37% for simple types. By representing complex types as directed graphs (type graphs), Stir achieves a type similarity score of 77.36% and 59.61 % for complex types and zero-shot complex types, respectively.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → Language features.

KEYWORDS

Type inference, deep learning, structured learning, graph generation

ACM Reference Format:
Yaohui Peng, Jing Xie, Qiongling Yang, Hanwen Guo, Qingan Li, Jingling Xue, and Mengting Yuan. 2023. Statistical Type Inference for Incomplete Programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616283

*Corresponding Author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616283

1 INTRODUCTION

Type inference, as an essential part of type systems, is widely used in program analysis, program abstraction, program optimization, and language security (among others). For whole programs, traditional type systems rely on syntax rules and type rules [7, 10, 37]. Therefore, well-typed terms (e.g., code snippets) must be well-formed.

Problem Statement. For software engineering tasks such as code search [6, 45, 46, 50], code mining [33], code review [47] and code summarization [18], which focus on code snippets retrieved from programming forums or code repositories, programs may be incomplete or even ill-formed. In addition, real-time program analysis tasks, such as code completion [19] in source code editors, also tend to parse incomplete programs. For example, a simple C/C++ program in several lines of code containing an include directive to a header file in SDKs may be expanded by a preprocessor into millions of lines of code, whose behavior can be hard to predict [28]. In this paper, we aim to infer types from such incomplete or ill-formed programs, for which whole-program syntactic analysis is often inapplicable. Software engineering tasks for incomplete programs will benefit from such inferred type information, as they are no longer plain text.

Prior Work. Recently, machine learning has been adopted to perform type inference in whole programs. Some reason about type information probabilistically [39, 40, 42, 51] while others resort to deep learning [3, 26, 36, 38, 49]. These efforts require whole-program syntactic analysis to convert a program into features before machine learning is applied. SnowWhite [22] predicts types for function parameters and return values in WebAssembly binary, yet still well-formed, programs. PsycheC [29] can handle ambiguous

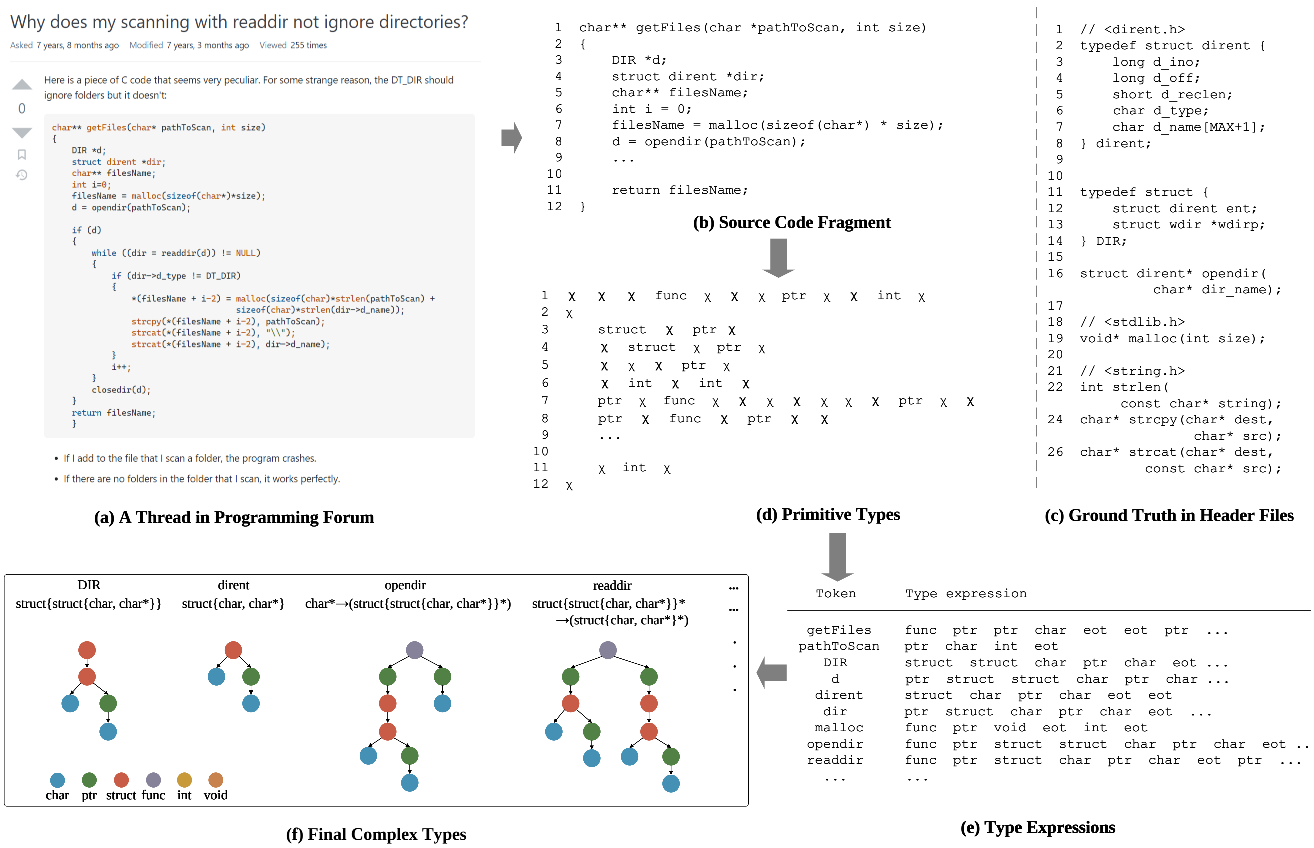


Figure 1: A motivating example. For a thread, shown in (a), posted in Stack Overflow, STIR analyzes its code snippet shown in (b) without considering the relevant ground truth types that are available in the omitted header files shown in (c). In the prediction stage ((b)→(d)), STIR predicts a type tag for each identifier token, and consequently, all the simple types, where χ stands for "type information is not applicable". In the refinement stage ((d)→(e)→(f)), STIR first translates a sequence of tokens with inferred type tags into type expressions ((d)→(e)) and then refines complex types from type expressions ((e)→(f)).

(with T representing possibly either a type or a variable) yet well-formed incomplete C programs, but cannot infer types in ill-formed programs. To the best of our knowledge, DeepTyper [14] is the only one that requires no syntactic analysis but applies NLP to infer types from well-formed programs (complete or not). However, all these prior approaches solve type inference as a classification problem, where types are mapped to type tags. Such a formulation has two undesirable consequences. First, it is impossible to infer fresh types, as they are unavailable in the training stage. Second, type correlation is lost, since types are treated as discrete labels, making it difficult to infer complex types. In practice, a type system often provides complex types, e.g., functions, tuples, and references to characterize high-level objects. Therefore, types are structural objects and cannot be converted into a finite set of discrete labels. For the traditional rule-based type inference on a whole program, types can be deduced inductively in a syntax-directed manner. For incomplete or ill-formed programs, however, learning complex structures is much harder than solving a classification problem.

Our Solution. We introduce STIR, a statistical approach to inferring types in incomplete or ill-formed programs. In the absence of syntactic knowledge, a program is literally plain text. Our first insight is that even plain text can still provide hints for type inference. These hints can be *local* (e.g., the type of an identifier "count" has a great chance of being int), *contextual* (e.g., the left-hand side and the right-hand side of the operator '=' may share the same type), or *global* (e.g., the implication between the declaration of an identifier and its later uses). Given enough training data, it is possible to capture such hints. Our second insight is that although learning structures directly from plain text is hard (as demonstrated in, for example, protein prediction [23]), we can transform the type inference problem into a sequence-to-graph parsing problem, which entails translating a type from a sequential representation to a graphical representation. Assisted further by a probabilistic model, STIR can infer types more effectively than the prior work that relies on learning types directly from plain text (e.g., DeepTyper).

STIR infers types in two stages. In the prediction stage, we use a BiLSTM-CRF model to predict type tags for individual tokens. Tokens are transformed into vectors to be fed to the model in order to learn local type hints, BiLSTM (Bidirectional Long Short-Term

Memory [16]) is used to capture contextual clues, and CRF (Conditional Random Field [21]) is designed to learn global knowledge. At the end of this first stage, a source program, which is a sequence of tokens, is translated into a sequence of type tags. The tokens tagged with primitive type tags are classified as simple type tokens, and therefore will not be refined in the second stage.

In the refinement stage, we refine or infer the actual complex types for the tokens with complex type tags. We first apply multi-head attention [48] to capture the type correlation among the tokens. We then use a generative model to obtain a type expression per token. For the type expressions that are incomplete or ill-formed (due to the nature of machine learning), we have designed a probabilistic model to recover their complex types. Instead of repairing an incomplete program directly, repairing type expressions achieves higher accuracy as the domain space has been massively reduced.

Contributions. To the best of our knowledge, STIR is the first to infer complex types (including zero-shot types) in incomplete or ill-formed programs with an infinite type vocabulary.

- We introduce STIR, a novel two-stage approach for inferring types in incomplete or ill-formed programs statistically.
- We propose a BiLSTM-CRF model that can generate highly precise type tags for simple and complex types.
- We reduce the type inference problem for constructing complex types to a sequence-to-graph parsing problem.
- We show experimentally that STIR advances the state of the art in performing type inference in incomplete programs.

The rest of the paper is organized as follows. Section 2 motivates STIR with an example. Section 3 introduces STIR. Section 4 describes and analyzes our experimental results. Section 5 discusses the related work. Section 6 concludes the paper.

2 A MOTIVATING EXAMPLE

We use an example to show how STIR infers complex types in incomplete programs. Figure 1(a) depicts a thread from Stack Overflow[1]. Figure 1(b) gives the code snippet in C extracted from the corresponding question. Figure 1(c) gives the ground-truth types available in the header files that are omitted but would otherwise be included in the corresponding complete program. Although programmers who are familiar with Linux file systems may correctly deduce the type for each user-defined identifier in the code snippet, traditional rule-based type inference approaches will fail, as the header files (Figure 1(c)) are missing in the thread.

Given the code snippet as an incomplete program, STIR infers the complex types for its identifiers as shown in Figure 1(f) according to the workflow Figure 1(b)⟶Figure 1(d)⟶Figure 1(e)⟶Figure 1(f). Each inferred complex type may be incomplete if some of its parts are never used in the code snippet. However, the inferred complex types are expected to make the code snippet well-typed.

STIR infers types in the following two stages:

• **Prediction.** In this first stage, STIR acts as a classifier to predict type tags for the tokens that represent user-defined identifiers. It takes as input the code snippet in Figure 1(b) as a sequence of tokens and produces as output a sequence of type tags in Figure 1(d). For this example, the five relevant type tags are "χ", "func", "ptr", "struct", and "int", which stand for "type information is not applicable", "function", "pointer", "structure" and "int", respectively. Consider the 12 type tags in Figure 1(d) predicted for the 12 tokens at line 1 in Figure 1(b). The three identifiers, `getFiles`, `pathToScan`, and `size` are tagged with "func", "ptr' and "int", respectively, and all the others including built-in types `char` and `int` are tagged with "χ". Note that all the tokens will also be fed to the second stage as well. In this first stage, STIR only predicts the type tags for the identifier tokens. Consider `opendir` with its ground truth type being "char*⟶ (struct { struct {..., char, char* }, struct { ... } }*)" (Figure 1(c)). STIR identifies it as a function in this first stage and will infer its complex type in the second stage. For many classifier-based approaches like [3, 14, 26, 38], it will be impossible to predict such complex types unless they appear in the training set.

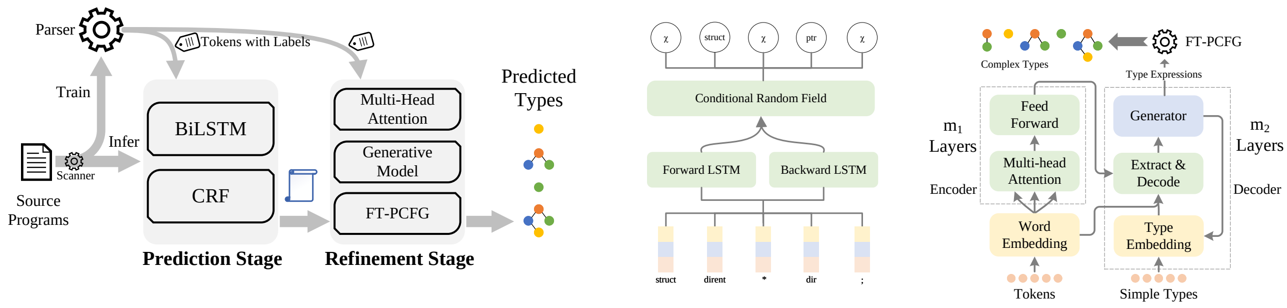
• **Refinement.** In this second stage, given a sequence of tokens (Figure 1(b)), together with their type tags (Figure 1(d)), STIR refines the complex types for identifier tokens with complex type tags (e.g., "struct" and "ptr"), as depicted in Figure 1(f). The tokens with primitive type tags (e.g., "χ" and "int") will not be refined.

Constructing structural types (i.e., type graphs) directly from sequential tokens is non-trivial. In this case, we make use of type expressions, each of which is a string of type symbols, as an intermediate representation to smooth the process of constructing type graphs. STIR leverages a generative model with multi-head attention to generate type expressions, as shown in Figure 1(e). For example, the type expression for `opendir` is "func ptr struct struct char ptr char eot ...", where "eot" (end of type) marks the end of a subexpression. Type expressions are generated in an iterative way by using a trained sequential decision model. At each iteration, the model generates a new type symbol for each token based on the context (i.e., the input token sequence) and the history (i.e., the portions of type expressions already generated). This strategy has the advantage of being able to update the types for correlated identifiers simultaneously. Although simple type tokens will not be refined in this stage, STIR still generates type expressions for them so as to simplify the neural network architecture.

Let us consider the term "`dir = readdir(d)`" in Figure 1(b), for example. Here, `dir` and `readdir` are correlated as `dir` may have the same type as the return value of `readdir`. When STIR finds that `*dir` has an array member according to the term `dir->c_name`, this member can be used to update the type expression of `opendir` at the next iteration. Given the type expressions constructed in Figure 1(e), STIR can finally turn them into types as shown in Figure 1(f). During this process, a complex type is treated as a graph (or a tree if it is non-recursive (Section 4.4.2)). For example, `opendir` is a function. Therefore, the root of its type tree is a "func" node in gray color, with its first child representing the return type, and the remaining children representing the types of the parameters. Type graphs are constructed from type expressions by following a trained PCFG (Probabilistic Context Free Grammar) model. We have designed a set of semantic rules for specifying type expressions, so that parsing a type expression produces a type graph.

Due to the probabilistic nature of neural networks, a type expression may still be incomplete or ill-formed. Thus, we have added a fault-tolerant mechanism to PCFG to enable STIR to recover type

[1]https://stackoverflow.com/questions/30544500/why-does-my-scanning-with-readdir-not-ignore-directories



(a) Architecture of Stir (b) Prediction Stage (c) Refinement Stage

Figure 2: Overview of Stir. In the prediction stage, we train a "BiLSTM-CRF" model to infer type tags for tokens, and consequently, determine all the simple types. In the refinement stage, we use a generation-based model to refine complex types.

graphs from incomplete or ill-formed type expressions. For example, the type expression generated initially for `pathToScan` is "ptr char int eot", which is ill-formed. To parse it, our fault-tolerant mechanism finds two matching productions in which either "char" or "int" is a fault (or error). The trained PCFG model then chooses the one that treats "int" as a fault since it has a higher probability of being faulty for the two choices, giving rise to the type graph depicted in Figure 1(f).

Although the ground truth for the return type of `readdir` is a pointer to `dirent`, which is a structure containing five members (Figure 1(c)), Stir concludes that the type of `dirent` is a structure containing only `d_type` and `d_name`, the only two used in the code snippet (Figure 1(b)). Given this usage context, the terms involved with `dirent` are still well-typed, as the inferred type is a supertype of the ground truth type. In general, an inferred type does not have to be the same as its corresponding ground-truth type.

3 STIR: STATISTICAL TYPE INFERENCE

We describe how Stir uses two statistical type models in its two stages to infer types in incomplete programs. Section 3.1 gives an overview. Sections 3.2 and 3.3 focus on its two stages, respectively.

3.1 Overview

As shown in Figure 2a, Stir proceeds in two stages. The prediction stage, as illustrated in Figure 2b, is responsible for predicting a type tag for each token in the source code by using neural networks. The tokens with χ or primitive type tags are simple type tokens, and will not be refined. The refinement stage, as illustrated in Figure 2c, uses first a generative model with multi-head attention to generate type expressions for the input tokens and then a fault-tolerant probabilistic model to recover complex types from these type expressions. Simple type tokens are used only as the context in this stage. These models are trained with complete programs in order to acquire ground-truth types. When performing type inference for incomplete programs, Stir only makes use of a scanner to extract their tokens without the need of performing any syntactic analysis.

3.2 Predicting Simple Types

In this prediction stage, Stir tags each token in an incomplete program with a type tag by using a BiLSTM-CRF neural network. Given a set of simple type tags $\Gamma = \{\chi, int, char, ptr, struct, func, \cdots\}$ and a token alphabet N, Stir aims to learn a type-prediction function $\Delta : N^* \longrightarrow \Gamma^*$, such that $|\omega| = |\Delta(\omega)|$ for each $\omega \in N^*$.

Feature Selection. Given an incomplete program, its source code is converted into a sequence of tokens $\sigma = \omega_1\omega_2\cdots$. Stir uses word embedding [30] to capture local type hints in such a type tag prediction task. As a result, the initial token sequence is transformed into a sequence of vectors $R_\sigma = R_{\omega 1}R_{\omega 2}\cdots$, where $R_{\omega i}$ is the vector representation of ω_i. By constructing a lookup table, $R_{\omega i}$ can be retrieved using the index of ω_i in N.

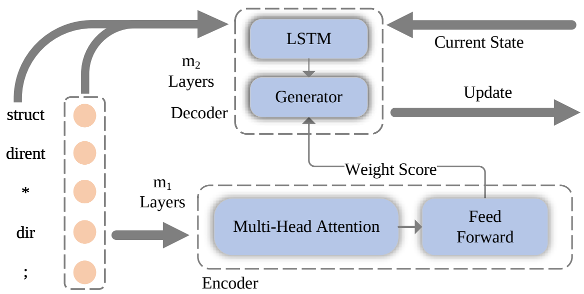
Neural Networks. Stir uses neural networks to capture contextual and global type hints. Unlike natural languages, the token sequences in programs are typically very long and difficult to split without destroying their contexts. We choose BiLSTM [16], which performs better than the traditional BiRNN [41] in capturing long-term dependencies to model contextual type hints. In addition, we use CRF (Conditional Random Field) [21], which aims at maximizing the probability of an entire sequence of tokens, to capture global type dependencies such as the declaration and uses of an identifier. By combining BiLSTM and CRF, Stir aims to predict type tags accurately for a sequence of input tokens. Compared to transformer models, our neural network model is more efficient in terms of running performance. In addition, adding attention mechanism, which is an important component in transformers, does not provide extra benefit in terms of accuracy, as illustrated in section 4.1.

As depicted in Figure 2b, a token sequence is fed into a bi-directional LSTM. The context of each token is modeled from the forward and backward LSTMs and sent to the CRF layer. At each step, each token x_i in the sequence is fed into a BiLSTM model:

$$\begin{aligned} z_i &= concat(\overrightarrow{z_i}, \overleftarrow{z_i}) \\ \overrightarrow{z_i} &= LSTM(x_i, \overrightarrow{z}_{i-1}; \overrightarrow{\theta}) \\ \overleftarrow{z_i} &= LSTM(x_i, \overleftarrow{z}_{i-1}; \overleftarrow{\theta}) \end{aligned} \tag{1}$$

where z_i, together with the forward $\overrightarrow{z_i}$ and backward $\overleftarrow{z_i}$, is the information obtained for x_i. Note that $\overrightarrow{\theta}$ and $\overleftarrow{\theta}$ are the training parameters of the forward and backward LSTMs, respectively.

A sequential CRF is used at the next layer of the bidirectional LSTM. This layer is responsible for decoding the type labels for the



iteration	0	1	2	3	4	5	6	7	8
struct	χ	χ	χ	χ	χ	χ	χ	χ	χ
dirent	struct	char	ptr	char	eot	eot	χ	χ	χ
*	χ	χ	χ	χ	χ	χ	χ	χ	χ
dir	ptr	struct	char	ptr	char	eot	eot	eot	χ
;	χ	χ	χ	χ	χ	χ	χ	χ	χ

Type Expressions

Figure 3: Generating type expressions based on the encoder-decoder architecture. The encoder leverages multi-head attention to gather the context information from input tokens. The decoder considers each token at each iteration and generates a new type symbol (if needed), based on the weight scores from the encoder and the portions of type expressions already generated.

entire token sequence and obtaining a type label for each token. For each token, there are a total of $|\Gamma|$ potential states (i.e., type tags). If $Z = z_1 z_2 \cdots z_n$ is an input sequence and Ω is the set for all possible label sequences, then $|\Omega| = |\Gamma|^n$. Let $Y = y_1 y_2 \cdots y_n$ be the ground-truth label sequence of Z and $Y' = y'_1 y'_2 \cdots y'_n$ be one possible label sequence in Ω. The conditional probability $p\,(Y|Z; W, b)$ represents the probability of Y among all possible sequences. Let $\psi_i\,(Y', Y, Z) = exp\left(W^T_{Y',Y} z_i + b_{Y',Y}\right)$. Then $p\,(Y|Z; W, b)$ is calculated as follows:

$$p\,(Y|Z; W, b) = \frac{\prod_{i=1}^{n} \psi_i\,(y_{i-1}, y_i, Z)}{\sum_{Y' \in \Omega} \prod_{i=1}^{n} \psi_i\,(y'_{i-1}, y'_i, Z)} \tag{2}$$

Here, $W^T_{Y',Y} z_i$ represents the weight vector, and $b_{Y',Y}$ the offset vector. CRF applies the maximum conditional relief estimation during training. Given a training set (σ, Y), the training process aims to maximize the conditional probability $p\,(Y|Z; W, b)$ for the ground truth Y and reduce the probability of other sequences in Ω. Given a sequence of tokens, we will then obtain a corresponding sequence of type tags with the highest probability in Ω.

3.3 Refining Complex Types

In the refinement stage, Stir infers complex types for tokens with complex type tags (e.g., "struct", "func" and "ptr"). As shown in Figure 2c, Stir takes as input a sequence of tokens with their type tags and produces as output a type expression for each token by using a generative model with multi-head attention. We propose a probabilistic model, FT-PCFG (Fault-Tolerant Probabilistic Context Free Grammar), to convert a type expression into a type graph.

Type Expression. Given a set of type symbols $\overline{\Gamma} = \Gamma \cup \{eot\}$, a type expression $\sigma \in \overline{\Gamma}^*$ for a type (including χ) is a string of type symbols. Stir uses type expressions, which are specified by the attribute grammar given in Table 1, as an intermediate representation for specifying types. E (the start symbol), T, and P are non-terminals and ptr, $array$, $struct$, $union$, $func$, $enum$, $typeName$, eot and χ are terminals. A type expression E can be a pointer (Prod 1), an array (Prod 2), a structure (Prod 3), a union (Prod 4), a function (Prod 5), an enumeration (Prod 6), or a primitive or simple type (Prod 7). T

Table 1: Attribute grammar for type expressions.

No.	Production	Semantic Rule
1	$E \rightarrow \boldsymbol{ptr}\ E_1\ eot$	$E.type := mk\_ptr(E_1.type)$
2	$E \rightarrow \boldsymbol{array}\ E_1\ eot$	$E.type := mk\_array(E_1.type)$
3	$E \rightarrow \boldsymbol{struct}\ T\ eot$	$E.type := mk\_struct(T.types)$
4	$E \rightarrow \boldsymbol{union}\ T\ eot$	$E.type := mk\_union(T.types)$
5	$E \rightarrow \boldsymbol{func}\ E_1\ T\ eot$	$E.type := mk\_func(E_1.type, T.types)$
6	$E \rightarrow \boldsymbol{enum}\ T\ eot$	$E.type := mk\_enum(T.types)$
7	$E \rightarrow P$	$E.type := P.type$
8	$T \rightarrow T_1\ E$	$T.types := T_1.types \mid\mid E.type$
9	$T \rightarrow \epsilon$	$T.types := []$
10	$P \rightarrow \boldsymbol{typeName}$	$P.type := mk\_primitive(typeName.literal)$[2]
11	$P \rightarrow \chi$	$P.type := mk\_primitive(\chi)$

may yield a sequence of E's including the empty string ϵ (Prods 8 and 9), where "||" is the list concatenation operator. P yields all possible primitive types (Prod 10) and a single χ (Prod 11).

The semantic rules describe how to construct type graphs for their associated productions. E has a synthesized attribute, $type$, representing a graph node. T has a synthesized attribute, $types$, which maintains a list of graph nodes. P has a synthesized attribute, $type$, which represents a graph node constructed by $mk\_primitive()$ for a primitive type stored in $typeName$ or χ. The other graph node constructors are: mk_ptr builds a pointer node with an outgoing edge to $E_1.type$, mk_array builds an array node with an outgoing edge to $E_1.type$, mk_struct builds a structure node with an outgoing edge to each node in $T.types$, mk_union builds a union node with an outgoing edge to each node in $T.types$, mk_enum builds a enumeration node with an outgoing edge to each node in $T.types$, and finally, mk_func builds a function node with the first outgoing edge to $E_1.type$ and an outgoing edge to each node in $T.types$.

Consider `malloc()` in Figure 1, with its type being "$int \rightarrow (void*)$". We can obtain the following type expression:

$$\begin{aligned} E \Rightarrow\ & func\ E\ T\ eot \Rightarrow func\ ptr\ E\ eot\ T\ eot \\ \Rightarrow\ & func\ ptr\ P\ eot\ T\ eot \Rightarrow func\ ptr\ void\ eot\ T\ eot \\ \Rightarrow\ & func\ ptr\ void\ eot\ T\ E\ eot \Rightarrow func\ ptr\ void\ eot\ E\ eot \\ \Rightarrow\ & func\ ptr\ void\ eot\ P\ eot \Rightarrow func\ ptr\ void\ eot\ int\ eot \end{aligned} \tag{3}$$

[2] $typeName$ represents a primitive type, e.g., char, int, or float defined in the C99 standard. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

The corresponding type graph in Figure 1(f) can be obtained by applying the given semantics rules straightforwardly.

According to this attribute grammar, all simple (or primitive) types, which are built-in types, cannot be zero-shot types. This means that zero-shot types must be complex types.

Learning Type Expressions. We propose a sequential decision model, which adopts the encoder-decoder architecture, to generate type expressions for all tokens. As shown in Figure 2c, the encoder calculates the weight scores each token for the other tokens and the decoder determines how to generate type expressions.

The encoder tries to discover type dependencies among the tokens by using a multi-head attention mechanism and a feed-forward network, inspired by the Transformer [48]:

$$MultiHead(Q, K, V) = concat(head_1, ..., head_h) \times weights \quad (4)$$

where Q, K, V denote queries, keys, and values that are calculated from the embedding of a token sequence. The correlation among the tokens is stored in *weights*, a learnable parameter matrix.

The scaled dot-product attention [48] $head_i$ is computed by:

$$head_i = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (5)$$

where d_k denotes the dimension of queries Q and keys K.

Different kinds of type dependencies exist. For example, the type expression of a variable with a structure type may be affected by one member variable, whose declaration can be anywhere in the program, yielding relatively a long-term dependency. On the other hand, the dependency between `dir` and `readdir` in Figure 1(a) is relatively short. In our type inference setting, the intuition for using multi-head attention is that by employing $head_1, ..., head_h$ to attend to different parts of a program, these dependencies can be discovered by multiple heads. The feed-forward network is made up of two linear transformations with a ReLU activation. The objective is to fuse the type information of the other tokens into each token based on the attention weights. By stacking several such blocks (with each consisting of multi-head attention and the feed-forward network), the encoder can capture hierarchical dependencies.

The decoder requires multiple rounds of generation. At each round, the state is built based on generated expressions. As a result, we redesign the decoder instead of using an off-the-shelf Transformer model directly. The decoder takes as input a sequence of tokens with their type tags inferred in the prediction stage, together with their weights acquired by the encoder, and produces as output the inferred type information for each token. Note that the type expressions generated for simple type tokens are ignored. The decoder handles all tokens simultaneously in parallel. For each type expression, its type symbols are generated iteratively. At each iteration, the decoder tries to add a new type symbol based on the current state l_i:

$$\begin{aligned} l_i &= LSTM(x_i, l_{i-1}; \theta) \\ x_i &= concat(w, t) \end{aligned} \quad (6)$$

Here, w represents the entire input token sequence, t represents the latest generated type symbols for the tokens in w, x_i is the concatenation of w and t, and θ is a trainable parameter. At the first iteration, the type expression for each token is empty and the type tag obtained for each token in the prediction stage is used in defining t. Based on the current state l_i, the decisions $d \in \Gamma^{|w|}$ for all the input tokens made by the decoder are given by:

$$d = argmax(softmax(f(multiply(weights, l_i)))) \quad (7)$$

where f is a dense layer and *weights* is the output of the encoder. Specifically, if d_j is a decision made for token w_j in w, the decoder appends d_j as a new type symbol to its type expression.

Figure 3 illustrates how the decoder generates the type expressions for w = (“`struct`”, “`dirent`”, “`*`”, “`dir`”, “`;`”) simultaneously in nine iterations. At iteration 0, t = (“χ”, “`struct`”, “χ”, “`ptr`”, “χ”), which represents the sequence of type tags inferred for w in the prediction stage. Given w and the updated t as shown at each of the next eight iterations, the decoder generates and appends a new type symbol to the type expression for each token. At the end of the last iteration, all redundant trailing χ's (according to the attribute grammar in Table 1) are removed. Note that the type expressions for tokens with non-complex types are ignored.

Type Graph Construction. Given a well-formed type expression, Stir can construct its type graph straightforwardly by applying the semantic rules in Table 1. To handle a type expression that is incomplete or ill-formed, we propose a probabilistic model with a fault-tolerant mechanism, FT-PCFG (Fault-Tolerant Probabilistic Context Free Grammar), to construct its type graph.

Let $G = (V_N, V_T, P, S)$ be a context-free grammar, where V_N, V_T and P are sets of non-terminals, terminals, and productions, respectively, and S is the start symbol. The *fault-tolerant grammar* of G is a context-free grammar $FT(G) = (V_N, V_T \cup \{err\}, P_{FT}, S)$, where $P_{FT} = P \cup \{A \rightarrow \alpha\ err\ \beta \mid A \rightarrow \alpha\ \beta \in P, \alpha \in (V_N \cup V_T)^+, \beta \in (V_N \cup V_T)^*\}$. The new terminal err represents an unexpected string in a type expression. If $(E \rightarrow \boldsymbol{ptr}\ E\ eot) \in P$, for example, then productions $(E \rightarrow \boldsymbol{ptr}\ err\ E\ eot)$, $(E \rightarrow \boldsymbol{ptr}\ E\ err\ eot)$, and $(E \rightarrow \boldsymbol{ptr}\ E\ eot\ err)$ are all in P_{FT}. By introducing such new productions, $FT(G)$ may become ambiguous. To resolve ambiguity, the probability of each production in $FT(G)$ should be trained before it can be used to parse ambiguous type expressions.

After converting $FT(G)$ into Chomsky Normal Form (CNF) [8], we train FT-PCFG via the inside-outside algorithm [9] and the expectation-maximization algorithm [32]. The training data of FT-PCFG are the type expressions generated from all the training programs. Given a CNF production r, let $\phi(r) \in \mathbb{R}^d$ be its one-hot vector representation, where d is the size of the production set. Let $v \in \mathbb{R}^d$ be a vector storing the probabilities $\psi(r)$ of all the productions r, which are initially identical, in $FT(G)$. Given a parse tree t, its probability $\psi(t)$ is defined as follows:

$$\psi(t) = \prod_{r \in t} \psi(r) = \prod_{r \in t} exp\{v \cdot \phi(r)\} = exp\{\sum_{r \in t} v \cdot \phi(r)\} \quad (8)$$

There may be several different parse trees for a given type expression, $x_1 \dots x_n$, that contains err. Let τ be the the set of all its possible parse trees. The probability of one possible parse tree is:

$$p(t|x_1 \dots x_n) = \frac{\psi(t)}{\sum_{t \in \tau} \psi(t)} \quad (9)$$

When training PCFG, if a parse tree t yields $x_1 \dots x_n$, the parameter v will be adjusted by the expectation-maximization algorithm to maximize the likelihood estimation of its probability $p(t|x_1 \dots x_n)$.

With multiple training type expressions generated from the training programs, the objective of training PCFG is to maximize the joint probability of parse trees for all type expressions. Given a type expression, the trained FT-PCFG model is used to find the parse tree with the maximum probability. The type graphs are then constructed by applying the semantic rules in Table 1.

4 EVALUATION

We demonstrate that STIR advances the state of the art in performing type inference in incomplete C programs. We have considered C, since compared with other programming languages, C (as a strongly typed language) requires strict type checking, which makes type inference in a program challenging when the program is incomplete or ill-formed. If STIR can handle effectively type inference for incomplete programs written in C, STIR is also expected to work well for other programming languages.

We address the following three research questions:

- **RQ1.** Is STIR effective in predicting type tags?
- **RQ2.** How effectively can STIR infer complex types?
- **RQ3.** Is STIR still effective on zero-shot types?

As all simple (or primitive) types must have been seen during the training stage (Table 1), zero-shot types are all complex types.

To the best of our knowledge, STIR is the first to infer complex types (including zero-shot types) in incomplete or ill-formed programs with an infinite type vocabulary.

Dataset. We collect source programs from GNU[3]. The dataset contains 6637 source files, with 4348 of these source files containing no more than 1000 tokens. We have modified Clang[4], a frontend of LLVM[5], to parse the source files to acquire the ground-truth types for the identifies appearing in these programs.

Table 2: Dataset characteristics.

Category	#Projects	#Files	#Tokens	#Types	#Zero-shot Types
Training Set	98	3506	1042196	6246	-
Test Set	77	842	253108	2551	834

Table 2 gives more details of the dataset. All the programs are randomly divided into a training set and a test set. The ratio of the training programs over the test programs is 4:1. In both cases, the programs with more than 1,000 tokens are dropped, as STIR is designed to handle (small) incomplete programs or code snippets. The entire dataset contains a total of 7080 distinct types: the training set contains 6246 distinct types and the test set contains 2551 distinct types (with some types appearing in both sets). There are 834 zero-shot types, implying that over one-third of the types that appear in the test set do not also appear in the training set.

Table 3 shows that the source-code files used in both the training and test sets have been selected to simulate the code snippets typically found in programming forums in terms of code sizes.

Table 4 gives top-10 types in the training and test sets.

[3]http://www.gnu.org/
[4]http://clang.llvm.org/
[5]http://llvm.org/

Table 3: File size distribution (as revealed by the number of files containing the number of tokens in a given interval).

Category	≤ 250	251 ∼ 500	501 ∼ 750	751 ∼ 1000	Total
Training Set	1979	737	466	324	3506
Test Set	474	173	113	82	842

Table 4: Top-10 types in the dataset.

Training Set	Total	Test Set	Total
int	2342241	int	383128
char*	358127	char*	79492
long	147771	long	43503
struct*	89672	struct*	27707
double	68695	double	17996
void*	45975	void*	13000
int*	45041	int*	11602
char	28434	char	7994
char**	21854	char[]	6232
char[]	21341	struct {int, char*, struct*}*	5296

Baseline. In practice, most incomplete programs are ill-formed, as they contain unresolved macros and undeclared identifiers. Therefore, existing type inference approaches that rely on whole-program syntactic analysis cannot be used as baselines. PsycheC [29] can handle ambiguous yet well-formed incomplete C programs, but it cannot infer types in ill-formed programs (caused by, e.g., unresolved macros). DeepTyper [14], on the other hand, is the only one that requires no syntactic analysis but applies NLP to infer types in incomplete programs, which are interpreted as plain text. However, as a classifier, DeepTyper is limited to inferring only type tags. Nevertheless, for incomplete programs, DeepTyper represents a state-of-the-art baseline for comparison purposes.

To evaluate the effectiveness of STIR in inferring simple types in its prediction stage, we compare STIR with DeepTyper. To evaluate the effectiveness of STIR in inferring complex types (including zero-short ones) in its refinement stage, we resort to graph similarity.

Training. The neural networks in STIR are implemented in PyTorch [35] as the back-end. As the training parameters of DeepTyper are not provided [14], we have consulted TypeWriter [38], which is also a deep-learning-based approach, to set up these parameters, as listed in Table 5. Since the input sequences are relatively long, the batch size is set to 16 to avoid out of GPU memory. The learning rate is 2×10^{-3}, as is done in TypeWriter. While TypeWriter uses a dropout rate to 0.25, we use 0.5 in order to prevent our model from overfitting due to imbalanced data (Table 4). We set the L2 rate to 10^{-4} for the same reason. Like many other deep learning tasks, we choose cross entropy loss and Adam optimizer [20].

In the prediction stage, as the process of training CRF is relatively slow, the vector size used in token embedding and the hidden size of LSTMs are set to 128 and 256, respectively. After each training

Table 5: Main training parameters.

Parameter	Value
Batch size	16
Learning rate	2×10^{-3}
Dropout rate	0.5
Loss function	CE loss
L2 rate	10^{-4}
Optimizer	Adam [20]

epoch, we will check to see if the training model has achieved the desired optimality. The training process will be stopped after five epochs if the loss fluctuation is less than 1%.

Table 6: Top-10 classes of tokens classified according to the lengths of their type expressions.

Length of a Type Expression	#Tokens	Percentage
1	2110696	85.74%
3	131544	5.34%
4	393902	1.60%
6	19379	0.79%
5	13237	0.54%
9	7335	0.30%
7	7294	0.30%
8	6667	0.27%
226	5320	0.22%
12	4924	0.20%

In the refinement stage, the vector size used in token embedding is increased to 200 in order to better represent the token information. As for multi-head attention settings (Figure 2), the number of heads is 2. In the encoder (Figure 2(c)), the number of layers is $m_1 = 3$. As revealed in Table 6, for over 95% tokens in the dataset, their type expressions contain no more than 10 type symbols each. Note that type expressions with a length of 1 represent simple types. In another word, 85.74% of the tokens in the dataset are associated with simple types. In the decoder (Figure 2(c)), the number of layers is set to be $m_2 = 10$, which means that each type expression contains no more than 10 symbols. This is a trade-off between accuracy and performance.

Metrics. Stir runs its prediction stage as a classifier, similarly as DeepTyper [14]. Therefore, we also use *accuracy*, the same metric used in DeepTyper, to compare both in predicting simple types.

For its refinement stage, we evaluate Stir's effectiveness for inferring complex types. For incomplete programs, as explained in Section 2, an inferred complex type does not have to be identical to its corresponding ground-truth type. Since Stir represents complex types as graphs, we use a graph similarity metric to measure the degree of similarity between an inferred type and its ground truth type. Graph Edit Distance (GED) [4] and Maximum Common Subgraph (MCS) [5] are classical methods to calculate graph similarity. However, computing either is known to be NP-complete [5, 52]. In this case, we have opted to use a more computationally efficient approach, Graph Kernel [27, 34], to measure graph similarity. Specifically, we use Weisfeiler-Lehman graph kernel [43], a state-of-the-art graph kernel, to evaluate our refinement stage.

Table 7: The accuracy for predicting type tags.

Model	Simple Type Tags	Complex Type Tags	All Type Tags
Stir	**97.37%**	**92.29%**	**96.40%**
Stir-A	90.91%	72.47%	87.36%
DeepTyper	78.50%	65.89%	76.95%

Computing Platform. We have conducted all our experiments on a Windows 10 desktop equipped with an 8-core Intel i7-7500 CPU of 3.40 GHz with 32GB memory, accelerated by a 12GB NVIDIA GeForce RTX 2080Ti GPU.

4.1 RQ1: Predicting Type Tags

We evaluate the effectiveness of Stir in predicting type tags, and consequently, inferring simple types in incomplete programs. We conduct three experiments, with each running an independent neural network model to predict the type tags for the test programs evaluated. Our results are reported in Table 7. The Stir model is the BiLSTM-CRF model adopted by Stir in its prediction stage. The DeepTyper model is built by following [14]. Since DeepTyper uses an attention layer to capture the relations among the tokens, we have also designed a so-called Stir-A model to investigate the impact of attention on predicting simple types, by adding one attention layer to our BiLSTM-CRF model.

For each model, we translate each source file in the test set into a sequence of tokens and then predict their type tags. In Table 7, we can see the accuracy achieved by each model in predicting simple type tags, complex type tags, and all the type tags altogether.

Based on these results, three observations are in order:

- Stir predicts type tags well, achieving an accuracy of 97.37%, 92.29%, and 96.40% for simple type tags, complex type tags and all the type tags, respectively. To put these results in perspective, JSNice [40] achieves an overall accuracy of 63.4% and Typilus [1] achieves an overal accuracy of 89%. However, as described in Section 1, these existing approaches rely on whole-program syntactic analysis and are thus inapplicable to incomplete programs. Therefore, we conclude that Stir advances the state of the art in predicting type tags for incomplete programs.
- Stir outperforms the baseline, DeepTyper, by increasing its accuracy by about 20% in absolute terms in all the three cases. Note that DeepTyper performs better here than in its original paper [14], where DeepTyper is reported to achieve an accuracy of 71.1% on its top-10 common types and of 29.6% on the other types. This is due to the fact that the number of distinct type tags in this paper is relatively smaller. As for Stir-A, adding one attention layer as in DeepTyper actually lowers the accuracy of Stir. This is because CRF in Stir employs a joint probability mechanism, which captures global type hints more effectively.

As a result, adding an attention layer will confuse CRF with its learned weights rather than provide extra information. That is why STIR does not choose transformer models to predict type tags, as the attention mechanism plays an important role in transformer models. However, STIR-A still outperforms DeepTyper for all the three cases.

- For all the three classifier-based models evaluated, predicting complex type tags is harder than predicting simple type tags. This decrease in accuracy is expected, as some sophisticated clues about complex types cannot be easily found by relatively simple neural networks adopted in classifiers. Nevertheless, STIR outperforms DeepTyper by achieving an increase of 26.4% in accuracy in absolute terms on predicting complex type tags. This advantage shows again that CRF (adopted by STIR) is more effective than a single attention layer in finding global type hints.

4.2 RQ2: Inferring Complex Types

We evaluate the effectiveness of STIR in refining the actual complex types for the tokens with complex type tags. To verify the impact of the first stage on inferring complex types, we compare STIR with its three variants, STIR-OT, STIR-DT, and STIR-GT. STIR-OT differs from STIR in that STIR-OT assumes that the type tags for all the tokens are "χ". STIR-DT uses the type tags generated by DeepTyper [14]. STIR-GT is the oracle that uses the ground-truth type tags.

We compare these four methods by using graph (or type) similarity. Given a method, type similarity measures the degree of similarity between a predicted complex type and its corresponding ground-truth type as described earlier. Our results are reported in Table 8. There are three categories of complex types. The "Pointer" category includes pointer and array types since arrays are treated as pointers in the C programming language. The "Structure" category contains structure types, union types and enumeration types. The "Function" category contains function types only. Note that all the zero-shot types in the test programs are also included.

STIR is highly effective when compared to STIR-GT, the oracle method that uses the ground-truth types. However, STIR-OT and STIR-DT are much less effective than STIR.

STIR achieves a macro average of 77.36%. An important factor that prevents STIR from improving its graph similarity further is that STIR can often infer only a strictly subset of the members of a ground-truth complex type based on how its variables are used in a small code snippet, as explained in Section 2. To put our results in perspective, graph generation tools in molecular prediction such as GraphAF [44], GraphDF [25] and GraphEBM [24] report their graph similarity scores as 66% , 65% , and 67%, respectively. Therefore, STIR is effective in inferring complex types measured in terms of graph similarity, especially since zero-short types are also included.

By examining the results for STIR and its three variants STIR-OT, STIR-DT and STIR-GT, we see that the quality of their input type tags affects their ability in inferring complex types. Without any type information in the input type tags used, STIR-OT achieves a macro average of 43.41%, as some tokens (e.g., char and int) carry type information themselves. STIR-DT achieves a higher macro average of 59.71% by using the type tags predicted by DeepTyper. STIR-GT performs better than STIR as expected, achieving a macro average of 78.85%, because it uses the ground-truth type tags.

Table 8: Graph similarity for complex types.

Model	Pointer	Structure	Function	Macro Avg
STIR	76.13%	79.34%	80.34%	77.36%
STIR-OT	43.89%	40.15%	43.48%	43.41%
STIR-DT	63.39%	49.82%	59.82%	59.71%
STIR-GT	**77.68%**	**82.59%**	**80.75%**	**78.85%**

Based on our experimental results, we see that STIR performs effectively in inferring complex types for code snippets.

4.3 RQ3: Inferring Zero-Shot Types

STIR applies a generative model to generate type expressions that represent type graphs. Thus, STIR is expected to infer zero-shot types. In Table 9, we report our results from applying STIR and its two variants STIR-DT and STIR-GT (introduced in Section 4.2) to infer zero-short types. Note that we do not consider STIR-OT, as STIR ignores the tokens tagged with χ in STIR's refinement stage.

Table 9: Graph similarity for zero-shot types.

Model	Pointer	Structure	Function	Macro Avg
STIR	60.65%	61.64%	56.21%	59.61%
STIR-DT	48.17%	37.47%	29.80%	41.10%
STIR-GT	**61.72%**	**62.46%**	**56.38%**	**60.39%**

STIR is highly effective when compared with STIR-GT, which uses the ground-truth types. However, STIR-DT is much less effective.

Let us analyze the performance of STIR in detail. STIR achieves a macro average of 59.61%, which is relatively low compared to its performance reported in Table 8. In general, the type graphs for zero-shot types are usually larger than those for non-zero-shot types, as small type graphs have a higher possibility of being included in the training set. The decoder generates a type expression by making a sequence of decisions. At each iteration, the probability of generating and appending a wrong type symbol to the type expression increases without the help of learned type knowledge. In addition, the possibility of generating a wrong type symbol becomes amplified for a long type expression. Therefore, the rear part of a type expression is mostly unmatched with the corresponding rear part in its ground-truth type expression. This explains why the graph similarity for zero-shot types is lower than that for non-zero-shot types. On the other hand, the front part of a type expression may have a good chance to match with the front part of its ground-truth type expression (according to the attribute grammar given in Table 1). Therefore, STIR can still succeed in generating similar type graphs for zero-shot types as demonstrated in this paper. Note that the similarity for function types is the lowest among the three type categories. This is because many GNU functions return a pointer to a structure (e.g., `opendir` in Figure 1), making the type expressions for their function types relatively long.

4.4 Discussions

We describe the benefits and limitations of STIR and how we have mitigated threats to validity in our design and implementation.

4.4.1 Applications. We envisage that Stir can be incorporated into some existing software engineering frameworks [12, 13, 15] as an independent tool to improve their effectiveness in handling their tasks. Many machine-learning-based software engineering tasks [2, 40] retrieve code snippets from programming forums such as Stack Overflow as training data. For example, code search [6, 45, 46, 50] uses such incomplete programs (as plain text) to learn the association between source code and a query. Given its high accuracy in type tag prediction, Stir can be used to provide precise type information in the training code to improve its quality.

Table 10: Average inference time of Stir (ms).

Prediction	Refinement	Overall
80	1829	1909

Stir can infer the types in under two seconds per file on average (Table 10). So Stir can be used in real-time scenarios where whole-program analysis is not applicable. For example, a lightweight editor can perform code completion on partly written code based on the complex types predicted by Stir instead of analyzing all related source/header files as done traditionally in many IDEs.

4.4.2 Limitations. Stir can be improved along two directions:

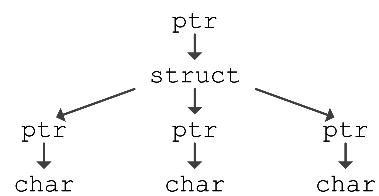
Recursive Types. Currently, Stir does not support recursive types. Stir uses type expressions, which are finite strings, to represent non-recursive complex types. In order to express recursive types, which are infinite structures, some additional machineries such as the μ operator in type theory may be introduced into the current attribute grammar used for specifying type expressions. However, the idea of working with type expressions is to reduce the search space for complex types. The grammar is expected to be as simple as possible. Combining Stir and a rule-based type inference approach may be a solution for handling recursive types.

Error Recovering. Due to the nature of machine learning, Stir may assign tokens with wrong type tags. When generating the type expression for a user-defined identifier token, the decoder starts with the type tag predicted for the token as its leftmost type symbol. Since the type expression is generated from left to right, the first type symbol (i.e., the predicted type tag) cannot be modified. Thus, the type tags that are predicted incorrectly in the prediction stage cannot be fixed in the refinement stage. Let us consider an example in Figure 4. The code snippet in Figure 4(a) is from `rpc_clntout.c` of project `acm-5.1` in our dataset. The ground-truth type of `procs` is depicted in Figure 4(b) and its ground-truth type expression is given in Figure 4(c). Stir predicts a wrong type tag, i.e., χ for `procs` in its prediction stage. However, its refinement stage still generates a type expression based on the context of `procs`. The type expression generated (Figure 4(d)) is nearly identical to the ground-truth type expression (Figure 4(c)) except for the first type symbol χ. Currently, however, the erroneous χ cannot be repaired. In future work, this kind of errors may be fixed by using a probabilistic model trained with a set of error type expressions after the refinement stage.

4.4.3 Threats to Validity. We have mitigated these as follows:

```
static
write_program(def)
        definition *def;
{
        ...
        for (proc = vp->procs; ...; ...) {
                ...
        }
        return 0;
}
```

(a) Code Snippet



(b) Ground-Truth Type Graph

ptr struct ptr char eot ptr char eot ptr char eot eot

(c) Ground-Truth Type Expression

χ struct ptr char eot ptr char eot ptr char eot eot

(d) Generated Type Expression

Figure 4: A case study for incorrect type prediction.

Preprocessor. The programs in our dataset contain many macros, which may cause syntactic errors in their incomplete code snippets. Given a macro definition "`#define INT int`", the ground-truth type of `INT` can be intuitively set to int. On the other hand, an identifier may be expanded into an arbitrary string by its macro definition. As a result, this particular identifier may have no type information. Due to the complexity of the C preprocessor, we choose not to handle all possible macro expansions. Instead, if an identifier `x` is expanded into another identifier `y`, we assign the ground-truth type of `y` to `x`. Otherwise, the ground-truth type is set to χ. Therefore, some identifiers with macro definitions may be assigned with χ by mistake. In addition, an identifier may have different declarations under different configurations due to conditional compilation. Our experiments are conducted under only one configuration.

Neural Networks. We train neural networks by following commonly used settings. Their parameters are not fine-tuned. For DeepTyper [14], its parameters are not provided in its paper. Its implementation here may not be tuned identically to the original one.

5 RELATED WORK

We review only prior work closely related to our work.

Probabilistic Type Inference. JSNice [40] formulates the type inference problem as CRF-based structured prediction and predicts JavaScript type tags based on a dependency network among program variables. Xu et al. [51] conduct probabilistic type inference for Python programs by using multiple type hints derived from their data-flow, attribute access, type checking predicates, and variable names. Nate [42] utilizes logistic regression, decision tree, random forest, etc. to locate type errors, so as to improve type inference. These probabilistic methods, which require whole-program syntactic analysis to construct data structures (e.g., dependency network in JSNice) for type inference, cannot handle incomplete programs.

Deep-Learning-based Type Inference. Deep learning has been widely adopted in inferring types for dynamic languages (e.g., Python), where types are determined at run time. SnR [11] involves repairing the program before performing type inference, while Huang et al. [17] utilize prompt-based language models for type inference. Some recent efforts focus on some specific types like functions. NL2Type [26] and DLTPy [3] use information like function names, comments, parameter names and return expressions to infer function signatures. TypeWriter [38] extends a probabilistic model with recurrent neural networks to infer the return and argument types for functions from partially annotated Python programs. Other approaches use neural networks to predict primitive types and user-defined types, where types are treated as tags. DeepTyper [14] uses a sequence-to-sequence model to predict type tags. LambdaNet [49] makes use of GNNs (Graph Neural Networks) to predicts type tags in TypeScript. Typilus [1] implements a GNN to map variables to their type embeddings, which are later used to find the nearest types in a type space. HiTyper [36] is a rule-based type inference framework where neural networks are used to recommend type tags. Type4Py [31] employs a hierarchical neural network (HNN) to infer types, which are then translated into vectors by a deep similarity learning.

Among these earlier efforts, DeepTyper is the only one that requires no syntactic analysis. The common idea behind the others is to use syntactic analysis to construct dependency information (e.g., AST in Type4Py and type dependency graph in HiTyper) from the source program and then apply deep learning to learn the association between types and the dependency information. These approaches are inapplicable to incomplete programs.

Table 11: Type vocabularies in inferring zero-shot types.

Approach	Type Representation	Type Vocabulary Size
DeepTyper [14]	Type Tag	11830
DLTPy [3]	Type Tag	1000
NL2Type [26]	Type Tag	1000
LambdaNet [49]	Type Tag	100
TypeWriter [38]	Type Tag	1000
Typilus [1]	Type Tag	Unlimited
Type4Py [31]	Vector	Unlimited
HiTyper [36]	Structural Type	Fixed
SnowWhite [22]	Linear Representation	Unlimited
Stir	Type Graph	Unlimited

As revealed in Table 2, nearly one-third types in the test set are zero-shot types. Therefore, how to infer zero-shot types represents an important problem faced by machine-learning-based type inference, as zero-shot types are actually out of the vocabulary in the training data. We survey the type vocabularies used recently in Table 11. DeepTyper [14], DLTPy [3], NL2Type [26], LambdaNet [49] and TypeWriter [38] regard types as discrete tags by using finite type vocabularies. Therefore, these methods cannot handle zero-shot types, which are out of their vocabularies. Typilus [1] maintains an open vocabulary, which is a map from variables to their type tags. Once a variable is manually confirmed (e.g., by developers) with a type that is outside its vocabulary, this association is updated. Typilus use this strategy to avoid retraining for unseen types. Type4Py [31] uses known types to train a similarity neural network so as to map types to vectors. Although Type4Py can find the vector representations for zero-shot types, it does not reveal their structural details. HiTyper [36], on the other hand, uses structural type representations, as it combines rule-based inference and neural networks. Although HiTyper employs a rule-based framework, the size of its type vocabulary is fixed. If HiTyper's neural network recommends a zero-shot type, it will find a similar known type from the vocabulary as an alternative. SnowWhite [22] adopts a sequence-to-sequence model to recover complex types for function parameters and return values in WebAssembly binaries. Although binary programs follow a simple syntax, they are still well-formed. However, SnowWhite does not predict individual fields of aggregated types (i.e., struct, union, and enum). Therefore, SnowWhite uses a linear representation of types, which can be handled by classical sequence-to-sequence nerual networks. Stir trains neural networks to generate type graphs, which are capable of expressing all types, from incomplete programs. For zero-shot types, Stir is still able to generate similar type graphs without syntactic knowledge.

6 CONCLUSION

We have introduced Stir, a novel technique for predicting both simple and complex types in incomplete programs. Our approach offers the potential to learning information from random code files and provides type information to programmers. Stir is expected to provide significant benefits to many software engineering tasks, including code search, code recommendation, code completion, program summarization, defect prediction, and fault localization, where type inference for arbitrary code snippets is required.

7 DATA AVAILABILITY

We have submitted an artifact, which is also available online[6], to allow Tables 7 −10 to be reproduced.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable comments. Thanks also to Weijun Hong for his help on PCFG experiments. Jing Xie is currently working at Baidu. This work is supported by National Natural Science Foundation of China (61872272) and State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCH A202112.

[6]https://github.com/StirArtifact/stir/tree/fse2023

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 91–105. https://doi.org/10.1145/3385412.3385997

[2] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[3] Casper Boone, Niels de Bruin, Arjan Langerak, and Fabian Stelmach. 2019. DLTPy: Deep Learning Type Inference of Python Function Signatures using Natural Language Context. *CoRR* abs/1912.00680 (2019). arXiv:1912.00680 http://arxiv.org/abs/1912.00680

[4] Horst Bunke. 1982. Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation. *IEEE Trans. Pattern Anal. Mach. Intell.* 4, 6 (1982), 574–582. https://doi.org/10.1109/TPAMI.1982.4767310

[5] Horst Bunke and Kim Shearer. 1998. A graph distance metric based on the maximal common subgraph. *Pattern Recognit. Lett.* 19, 3-4 (1998), 255–259. https://doi.org/10.1016/S0167-8655(97)00179-7

[6] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-Domain Deep Code Search with Meta Learning. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 487–498. https://doi.org/10.1145/3510003.3510125

[7] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *PACMPL* 1, OOPSLA (2017), 48:1–48:30. https://doi.org/10.1145/3133872

[8] Noam Chomsky. 1959. On Certain Formal Properties of Grammars. *Inf. Control.* 2, 2 (1959), 137–167. https://doi.org/10.1016/S0019-9958(59)90362-6

[9] Michael Collins. 2013. The Inside-Outside Algorithm. *Lecture Notes* (2013).

[10] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 60–72. http://dl.acm.org/citation.cfm?id=3009882

[11] Yiwen Dong, Tianxiao Gu, Yongqiang Tian, and Chengnian Sun. 2022. SnR: Constraint-Based Type Inference for Incomplete Java Code Snippets. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1982–1993. https://doi.org/10.1145/3510003.3510061

[12] Angelo Furfaro, Teresa Gallo, Alfredo Garro, Domenico Saccà, and Andrea Tundis. 2016. ResDevOps: A Software Engineering Framework for Achieving Long-Lasting Complex Systems. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016.* IEEE Computer Society, 246–255. https://doi.org/10.1109/RE.2016.15

[13] Félix García, Oscar Pedreira, Mario Piattini, Ana Cerdeira-Pena, and Miguel R. Penabad. 2017. A framework for gamification in software engineering. *J. Syst. Softw.* 132 (2017), 21–40. https://doi.org/10.1016/j.jss.2017.06.021

[14] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 152–162. https://doi.org/10.1145/3236024.3236051

[15] Karen Henricksen and Jadwiga Indulska. 2004. A Software Engineering Framework for Context-Aware Pervasive Computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 2004), 14-17 March 2004, Orlando, FL, USA.* IEEE Computer Society, 77–86. https://doi.org/10.1109/PERCOM.2004.1276847

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9, 8 (1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[17] Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Prompt-tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022.* ACM, 79:1–79:13. https://doi.org/10.1145/3551349.3556912

[18] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers.* https://www.aclweb.org/anthology/P16-1195/

[19] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. CodeFill: Multi-token Code Completion by Jointly learning from Structure and Naming Sequences. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 401–412. https://doi.org/10.1145/3510003.3510172

[20] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* http://arxiv.org/abs/1412.6980

[21] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001.* 282–289.

[22] Daniel Lehmann and Michael Pradel. 2022. Finding the dwarf: recovering precise types from WebAssembly binaries. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 410–425. https://doi.org/10.1145/3519939.3523449

[23] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, William L. Hamilton, David Duvenaud, Raquel Urtasun, and Richard S. Zemel. 2019. Efficient Graph Generation with Graph Recurrent Attention Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 4257–4267. https://proceedings.neurips.cc/paper/2019/hash/d0921d442ee91b896ad95059d13df618-Abstract.html

[24] Meng Liu, Keqiang Yan, Bora Oztekin, and Shuiwang Ji. 2021. GraphEBM: Molecular Graph Generation with Energy-Based Models. *CoRR* abs/2102.00546 (2021). arXiv:2102.00546 https://arxiv.org/abs/2102.00546

[25] Youzhi Luo, Keqiang Yan, and Shuiwang Ji. 2021. GraphDF: A Discrete Flow Model for Molecular Graph Generation. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7192–7203. http://proceedings.mlr.press/v139/luo21a.html

[26] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 304–315. https://doi.org/10.1109/ICSE.2019.00045

[27] Hafez Eslami Manoochehri and Mehrdad Nourani. 2020. Drug-target interaction prediction using semi-bipartite graph model and deep learning. *BMC Bioinform.* 21-S, 4 (2020), 248. https://doi.org/10.1186/s12859-020-3518-6

[28] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Trans. Software Eng.* 44, 5 (2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[29] Leandro T. C. Melo, Rodrigo Geraldo Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. 2018. Inference of static semantics for incomplete C programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 29:1–29:28. https://doi.org/10.1145/3158117

[30] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 3111–3119. https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html

[31] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 2241–2252. https://doi.org/10.1145/3510003.3510124

[32] Todd K. Moon. 1996. The expectation-maximization algorithm. *IEEE Signal Process. Mag.* 13, 6 (1996), 47–60. https://doi.org/10.1109/79.543975

[33] Giang Nguyen, Md Johirul Islam, Rangeet Pan, and Hridesh Rajan. 2022. Manas: Mining Software Repositories to Assist AutoML. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1368–1380. https://doi.org/10.1145/3510003.3510052

[34] Hemant Palivela, C R Nirmala, and Divesh Ramesh Kubal. 2017. Application of various graph kernels for finding molecular similarity in ligand based drug discovery. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS).* 1–8. https://doi.org/10.1109/ICACCS.2017.8014688

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo

Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[36] Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael R. Lyu. 2022. Static Inference Meets Deep learning: A Hybrid Type Inference Approach for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2019–2030. https://doi.org/10.1145/3510003.3510038

[37] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

[38] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: neural type prediction with search-based validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 209–220. https://doi.org/10.1145/3368089.3409715

[39] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *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*. 731–747. https://doi.org/10.1145/2983990.2984041

[40] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2019. Predicting program properties from 'big code'. *Commun. ACM* 62, 3 (2019), 99–107. https://doi.org/10.1145/3306204

[41] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Trans. Signal Processing* 45, 11 (1997), 2673–2681. https://doi.org/10.1109/78.650093

[42] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 60:1–60:27. https://doi.org/10.1145/3138818

[43] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.* 12 (2011), 2539–2561. http://dl.acm.org/citation.cfm?id=2078187

[44] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, and Jian Tang. 2020. GraphAF: a Flow-based Autoregressive Model for Molecular Graph Generation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=S1esMkHYPr

[45] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 388–400. https://doi.org/10.1145/3510003.3510140

[46] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1609–1620. https://doi.org/10.1145/3510003.3510160

[47] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2291–2302. https://doi.org/10.1145/3510003.3510621

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[49] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Hkx6hANtwH

[50] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-Stage Attention-Based Model for Code Search with Textual and Structural Features. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 342–353. https://doi.org/10.1109/SANER50967.2021.00039

[51] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 607–618. https://doi.org/10.1145/2950290.2950343

[52] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing Stars: On Approximating Graph Edit Distance. *Proc. VLDB Endow.* 2, 1 (2009), 25–36. https://doi.org/10.14778/1687627.1687631

Received 2023-02-02; accepted 2023-07-27