Concrete Type Inference for Code Optimization using Machine Learning with SMT Solving

FANGKE YE, Georgia Institute of Technology, USA
JISHENG ZHAO, Georgia Institute of Technology, USA
JUN SHIRAKO, Georgia Institute of Technology, USA
VIVEK SARKAR, Georgia Institute of Technology, USA

Despite the widespread popularity of dynamically typed languages such as Python, it is well known that they pose significant challenges to code optimization due to the lack of concrete type information. To overcome this limitation, many ahead-of-time optimizing compiler approaches for Python rely on programmers to provide optional type information as a prerequisite for extensive code optimization. Since few programmers provide this information, a large majority of Python applications are executed without the benefit of code optimization, thereby contributing collectively to a significant worldwide wastage of compute and energy resources.

In this paper, we introduce a new approach to concrete type inference that is shown to be effective in enabling code optimization for dynamically typed languages, without requiring the programmer to provide any type information. We explore three kinds of type inference algorithms in our approach based on: 1) machine learning models including GPT-4, 2) constraint-based inference based on SMT solving, and 3) a combination of 1) and 2). Our approach then uses the output from type inference to generate multi-version code for a bounded number of concrete type options, while also including a catch-all untyped version for the case when no match is found. The typed versions are then amenable to code optimization. Experimental results show that the combined algorithm in 3) delivers far superior precision and performance than the separate algorithms for 1) and 2). The performance improvement due to type inference, in terms of geometric mean speedup across all benchmarks compared to standard Python, when using 3) is 26.4× with Numba as an AOT optimizing back-end and 62.2× with the Intrepydd optimizing compiler as a back-end. These vast performance improvements can have a significant impact on programmers' productivity, while also reducing their applications' use of compute and energy resources.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Automated static analysis*; **Data types and structures**.

Additional Key Words and Phrases: Type Inference, Code Optimization, Python, Machine Learning

ACM Reference Format:
Fangke Ye, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2023. Concrete Type Inference for Code Optimization using Machine Learning with SMT Solving. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 249 (October 2023), 28 pages. https://doi.org/10.1145/3622825

1 INTRODUCTION

Python is one of the most widely used programming languages today. It has a rich ecosystem of libraries and tools. As a high-level programming language, Python's versatility and user-friendliness have made it a preferred choice for developers and researchers to build applications and research

Authors' addresses: Fangke Ye, yefangke@gatech.edu, Georgia Institute of Technology, USA; Jisheng Zhao, jisheng.zhao@cc.gatech.edu, Georgia Institute of Technology, USA; Jun Shirako, shirako@gatech.edu, Georgia Institute of Technology, USA; Vivek Sarkar, vsarkar@gatech.edu, Georgia Institute of Technology, USA.

2475-1421/2023/10-ART249
https://doi.org/10.1145/3622825

prototypes in domains including data science [McKinney et al. 2010], machine learning [Abadi et al. 2015; Paszke et al. 2019], and scientific computing [Virtanen et al. 2020].

Despite its advantages in programmability, Python poses significant challenges in terms of code optimization due to its dynamic typing. A well-established direction for optimizing programs with dynamic types is to use a just-in-time (JIT) optimizing compiler, with Numba [Lam et al. 2015] being a notable exemplar for Python. However, it is also widely recognized that the extent of code optimization that can be performed by JIT compilers is limited due to the fact that any time spent on code optimization contributes to dynamic execution time, as well as issues related to cold start and the overhead of cold paths. For this reason, even Numba offers an alternate ahead-of-time (AOT) optimization option, but that requires concrete types to be provided ahead of time by the programmer. Similarly, Intrepydd [Zhou et al. 2020] is a research AOT optimizing compiler that generates C++ code that can be statically compiled and loaded as a module for use in a Python application, but it too requires concrete types to be provided by the programmer (albeit only for function parameters). However, relying on programmers to provide type annotations can significantly hinder productivity as it can be a time-consuming and error-prone task. Consequently, a large portion of Python applications remains unoptimized, leading to wastage of compute and energy resources on a global scale.

In recent years, the considerable progress made in machine learning research has led to ongoing efforts to apply machine learning techniques to address the type inference challenge in dynamically-typed languages, with the goal of alleviating the need for manual type annotations [Allamanis et al. 2020; Hellendoorn et al. 2018; Pradel et al. 2020; Wei et al. 2020]. Typically, these approaches accept a partially typed program as input and generate probabilistic predictions for variable and expression types within the program. While these approaches can be helpful for software engineering and program understanding purposes, these predictions often lack the precision of concrete type inference needed for code optimization.

There are several recent studies that focus on enhancing the type correctness of machine learning based type inference by integrating its output with rule-based analysis. TypeWriter [Pradel et al. 2020] utilizes a gradual type checker to search for a type-correct assignment from the type predictions made by a machine learning model. OptTyper [Pandi et al. 2020] gathers logical constraints using rules and natural constraints from machine learning models, subsequently performing continuous relaxation to incorporate both types of constraints into a single continuous optimization problem. HiTyper [Peng et al. 2022] incorporates type recommendations from machine learning models into static type inference by iteratively alternating between model prediction and rule-based inference. These methods aim to determine one consistent set of type assignments, whether those include abstract types or concrete types. As a result, the inferred types may be too specific and only cover a limited number of cases, or too abstract to be useful for code optimization.

In this paper, we address the problem of concrete type inference for code optimization. The goal is to design a method to statically generate a manageable set of probable and consistent concrete type assignments for a given part of an input program (such as one or more functions in the program). By doing so, we can facilitate static optimization of the specified part of the program using the inferred type information. Note that our focus is not on performing whole-program static type inference, as enforcing static typing for the entire program might impede programming flexibility.

We explore 1) a variety of machine learning type inference models, as well as 2) constraint-based type inference using an SMT solver to help solve this problem. Recognizing the limitations inherent in both approaches, we introduce a novel method 3) that combines both approaches so as to fuse their strengths to overcome their individual drawbacks.[1] This is achieved by encoding machine

[1]Source code available at https://github.com/habanero-lab/cti-ml-smt.

learning predictions as constraints and applying progressive constraint relaxation to prioritize the discovery of the most likely solutions.

The key contributions presented in this paper can be summarized as follows:

(1) We develop a variety of machine learning-based type inference methods for concrete type inference for a NumPy-focused subset of Python, including a) a frequency-based prediction model, b) a neural network trained from scratch with an architecture inspired by prior work, c) a fine-tuned Transformer model pre-trained on a large multi-programming language corpus, and d) a prompt-based zero-shot type inference that leverages the cutting-edge GPT-4 large language model.
(2) We present a concrete type inference method based on SMT solving for the subset of Python considered above, incorporating array ranks into the type system to facilitate code optimization.
(3) We propose a novel approach for integrating machine learning and SMT solving for concrete type inference, aiming to generate type-correct parameter type combinations that effectively cover the intended types without generating an excessive number of code variants.
(4) We conduct a comprehensive evaluation and comparison of machine learning-based type inference models, SMT solving-based type inference methods, and their integration. The experimental results demonstrate that our combined approach, 3), achieves significantly higher precision and performance compared to the individual methods, 1) and 2).

2 BACKGROUND

2.1 Concrete Type Inference

Concrete type inference is the process of determining the concrete type (i.e., implementation type) of variables in a program at compile time. It differs from many conventional type inference methods that deduce abstract types, which describe the properties and interfaces of variables. By providing information on the implementation details of variables, concrete type inference can facilitate more precise code optimization. Previous research has explored several approaches for inferring concrete types, such as methods employing the Cartesian product algorithm [Agesen 1995] and constraint-based techniques [Plevyak and Chien 1994]. While past work has also explored the use of machine learning for type inference (e.g., [Allamanis et al. 2020; Hellendoorn et al. 2018; Pradel et al. 2020; Wei et al. 2020]), that pask work has typically not focused on concrete type inference for code optimization.

2.2 SMT Solver and SMT Based Type Inference

SMT (Satisfiability Modulo Theories) is an approach for determining the satisfiability of mathematical formulas. An SMT solver is a software tool for solving this kind of problem which involves a combination of theories, including Boolean and integer arithmetic. It can automatically determine whether a given formula is satisfiable by considering the constraints imposed by the underlying theories. SMT solvers are widely used in various areas of the computing industry, such as model checking, software/hardware verification, and automated theorem proving [Barrett and Tinelli 2018].

SMT solvers have been applied to the type inference problem, which can be formulated as a logical formula that deduces the type of a variable or expression in a given program. By translating typing rules into a set of constraints, an SMT solver can be utilized to determine whether these constraints are satisfiable and identify a set of type assignments that satisfy them [Hassan et al. 2018]. While there has been past work on combining rules-based inference with machine learning (e.g., [Pandi et al. 2020; Peng et al. 2022; Pradel et al. 2020]), to the best of our knowledge, we are

unaware of any past work that has combined SMT solvers with machine learning for concrete type inference.

2.3 Intrepydd Programming Language

The Intrepydd programming language [Zhou et al. 2020] introduced a subset of Python that is amenable to ahead-of-time (AOT) compilation of selected Python functions into C++. It is intended for writing kernel functions rather than complete or main programs. The C++ code generated from Intrepydd kernels can be imported into a Python application or a C++ application. A key constraint in the Intrepydd subset of Python is the requirement that Intrepydd function definitions include type annotations for parameters and return values. Given these type annotations, the Intrepydd compiler statically infers the types of local variables and expressions. The library knowledge base included in the Intrepydd tool chain specifies type rules for a wide range of library functions used by the Intrepydd programs, many of which are based on Python standard libraries such as NumPy. Combined with the library knowledge base that also provides per-function dataflow information, the statically inferred data types serve as the basis of: program analyses including def/use, dataflow, and dependence analyses; high-level program optimizations including loop invariant code motion, dense/sparse array operator fusion, and array allocation and slicing optimizations; and C++ code generation from the intermediate representation resulting from these code optimizations. In the final step, the generated C++ code is compiled into a binary module that can be invoked by the host program.

In this work, we aim to target the subset of Python utilized by Intrepydd, with types defined in Fig. 1. The type system incorporates array rank and tuple arity, requires homogeneous container types, and does not support first-class functions or nested function definitions. Additionally, scalar types are presented as zero-dimensional array types.

$$
\begin{array}{rcl}
\tau & ::= & \texttt{None} \mid \texttt{Array}(dtype, ndim) \mid \texttt{SparseMat}(dtype) \\
 & \mid & \texttt{List}(\tau) \mid \texttt{Dict}(\tau, \tau) \mid \texttt{Heap}(\tau, \tau) \mid \texttt{Tuple}(\tau, arity) \\
dtype & ::= & \texttt{int32} \mid \texttt{int64} \mid \texttt{float32} \mid \texttt{float64} \mid \texttt{bool} \\
ndim, arity & ::= & \text{non-negative integers} \\
\eta & ::= & \tau \mid \texttt{Function}(\overline{\tau}, \tau)
\end{array}
$$

Fig. 1. Types of Intrepydd.

3 OVERVIEW OF OUR APPROACH

In this paper, we address the problem of concrete type inference for enabling compiler optimizations with multi-version code generation. To be specific, the goal is to design a method to generate a likely and consistent[2] set of function parameter types based on a given input program. The rationale behind our focus on function parameter types is that their determination simplifies the inference for other types in the function, including return types and variable types. Once function parameter types are specified, the program can be fully typed and optimized by ahead-of-time optimizing compiler approaches for Python such as Numba AOT [Lam et al. 2015] and Intrepydd [Zhou et al. 2020]. We propose a novel approach to the concrete type inference problem by using a combination of machine learning and SMT solving.

An overview of our approach is shown in Fig. 2. To motivate our approach, we include an example function in the figure. By applying a machine learning type predictor to this function, we

[2]Throughout the paper, we use the terms "consistent", "type-check", "type-correct", and "valid" interchangeably to denote a function parameter type combination that does not lead to a type error.

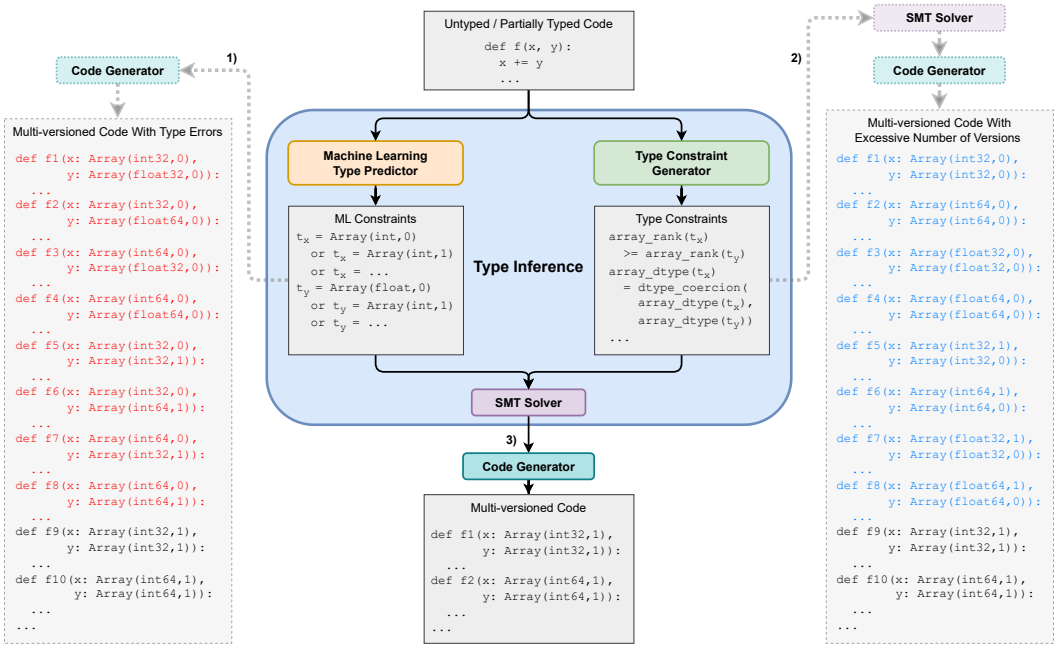


Fig. 2. Overview of our framework: 1) The left box illustrates the output obtained by only using machine learning models; 2) The right box illustrates the output obtained by only using type constraints; 3) The center box illustrates the output obtained by using the combination of machine learning and SMT solving introduced in this paper. The dashed lines and boxes illustrate the potential outputs of individual approaches. Red-colored code denotes the presence of type errors by using approach 1), while blue-colored code represents the excessive number of versions generated by approach 2) relative to the combined approach 3).

can obtain several potential types for each argument, as depicted in the left part of the figure as approach 1). When combined, these predictions may generate numerous combinations of parameter types, and many of them might not even type-check. As an illustration, consider the red-colored functions in the figure, whose parameter types are predicted by a machine learning model. In these functions, type-checking errors can occur for the code `x += y` since the type of `y` cannot be coerced or broadcasted to the type of `x`. On the other hand, a constraint-based approach can be utilized to generate type constraints from the code, as shown in the right part of the figure as approach 2). By solving these constraints, we can generate multiple parameter type combinations that are type-correct. However, the number of potential solutions can be extremely large, and impractical for use in static code optimization.

To overcome the issues present in these two approaches, we propose a hybrid approach 3) in the center of the figure. The predictions from the machine learning models are encoded as constraints. These constraints, along with the type constraints, are provided as input to an SMT solver, which subsequently generates multiple parameter type combinations that are both highly likely and type-correct. Approach 3) then uses the output from the SMT solver to generate multi-version code for a bounded number of concrete type options, while also including a catch-all untyped version for the case when no match is found. All the typed versions are then amenable to extensive code optimization.

4 CONCRETE TYPE INFERENCE BY COMBINING MACHINE LEARNING AND SMT SOLVING

In this section, we begin by presenting a number of machine learning type inference techniques tailored to our concrete parameter type inference problem, all of which represent different alternatives for approach 1). Next, we introduce our SMT solving type inference that represents approach 2). Lastly, we outline how we combine both methodologies to achieve their respective benefits while also addressing their limitations, thus creating a better-optimized approach 3).

4.1 Machine Learning Type Inference

Machine learning type inference models can often leverage the naturalness of human-written code by exploiting patterns in variable names, syntactic structures, and other features typically neglected by traditional type inference approaches [Allamanis et al. 2018]. These models are capable of producing multiple potential types along with their probabilities. This makes them well-suited for our type inference problem, as we lack sufficient type information to deduce a definitive type, but can still utilize machine learning to infer a few types of highest probability.

This section focuses on the machine learning approaches we have employed to solve our concrete type inference problem. Given the unique domain of our problem, it is not feasible to leverage existing machine learning type inference models directly. Thus, we have integrated various representative models to compare their relative efficacy.

In this section, we first define the scope of types that our machine learning models attempt to predict. Then, we introduce the methodology of constructing our dataset tailored to our particular problem. After that, we present the four distinct machine learning models that we have explored. Finally, we discuss the limitations of machine learning methods when applied to concrete type inference.

4.1.1 Type Prediction Space. Our machine learning models predict parameter types from a subset of the types defined in Fig. 1. We detail the design decisions for this type subspace, along with their rationales, in Table 1. The syntax defining this type prediction space is shown in Fig. 3.

$$
\begin{array}{rcl}
\tau & ::= & dtype \mid \texttt{Array}(dtype, ndim) \mid \texttt{List}(\tau) \mid \texttt{Dict}(\texttt{int}, \tau) \\
dtype & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{bool} \\
ndim & ::= & \texttt{1} \mid \texttt{2} \mid \texttt{3}
\end{array}
$$

Fig. 3. Syntax of types predicted by machine learning models.

Constraining the type space may lead to situations where the correct parameter types cannot be predicted because they are outside the prediction space. However, this drawback can be addressed by combining machine learning type inference with the SMT solving type inference (Section 4.2), which we discuss in more detail in Section 4.3.

4.1.2 Dataset Construction. In order to train our machine learning models for predicting function parameter types in Intrepydd, it would be ideal to use a dataset that includes Intrepydd source code and parameter type annotations. However, this is not feasible due to the fact that Intrepydd is relatively new and there are few programs currently written using it. Nonetheless, since Intrepydd is a subset of Python and many Intrepydd types can be mapped to built-in Python or NumPy types, we can instead use a dataset containing programs written in Python/NumPy, and rely on the models to transfer their knowledge learned from this dataset to Intrepydd programs.

Although there are public datasets available for training type inference models for Python [Allamanis et al. 2020; Mir et al. 2021; Raychev et al. 2016], they are not suitable for our case. Firstly,

Table 1. Design decisions for the type prediction space.

Type	Design decision and rationale
Scalar types	We represent a scalar type directly by its element type, rather than a 0-dimensional array. This simplifies the models' output without introducing any ambiguity.
Array element types with different bit widths	We do not differentiate between integer and floating-point types based on their bit width. This is because it is often challenging to infer the precise bit width solely from the source code, unless there is explicit type casting. Additionally, many programs are written in a way that allows data with different bit widths to function interchangeably. Including bit width in the dataset also reduces the number of data points for each type involving integer and floating-point types, making it more challenging for the machine learning models to learn.
`Array`	We cap the rank of arrays at 3, as most code in our dataset operates on arrays with ranks no greater than 3. This constraint enables the model to focus on the most common cases.
`Dict`	The `Dict` type only takes integers as keys due to an implementation limit imposed by Intrepydd's C++ back-end. By constraining the type space in this way, we prevent the models from predicting unsupported `Dict` types.
`None`	We do not predict the `None` type, as it is usually used as the return type of functions that do not return a value. Since we are only concerned about function parameter types, predicting a `None` type is rarely necessary.
`SparseMat`, `Heap`, and `Tuple`	We do not predict `SparseMat`, `Heap`, and `Tuple` types because it is challenging to construct a dataset containing these types. `SparseMat` and `Heap` are intrinsic types in Intrepydd and have no corresponding types in Python/NumPy. While tuples are often seen in Python, static-arity heterogeneous tuples are rarely used as parameter types.

these datasets contain a variety of Python projects, which may not be NumPy-intensive and thus not suitable for our specific case that focuses heavily on array operations. Secondly, their type labels are often not concrete enough, with annotations such as `List[Any]`. Additionally, NumPy arrays are usually annotated with `np.ndarray` without any information about element type and rank, which is not helpful for our purpose.

Therefore, we created our dataset from the source code of SciPy. We chose SciPy because it heavily utilizes NumPy and includes many programs for scientific computing, which is the domain Intrepydd is targeting. To obtain precise concrete type information, we executed SciPy's test suite and dynamically collected the runtime argument types for all functions using a library called MonkeyType,[3] which we modified to support extracting the element types and ranks of NumPy arrays. Subsequently, we extracted and deduplicated pairs of (function source code, argument types) to form our SciPy dataset. The source code was stripped of type annotations, comments, and docstrings, and we also filtered out types that could not be expressed using the syntax in Fig. 3.

4.1.3 Frequency-Based Model. To establish a baseline, we employed a simple model that predicts the most common types found in the training set. This model, which we refer to as *Freq* going forward, is effective in many cases due to the highly concentrated distribution of types in many programs, as noted by [Mir et al. 2021].

4.1.4 DeepTyper-FS: A Variant of DeepTyper Trained From Scratch. We developed a second model, an end-to-end deep learning type predictor, by utilizing the underlying neural network architecture of DeepTyper [Hellendoorn et al. 2018], and additionally incorporating an improved code tokenizer and an RNN-based type decoder. We trained this model from scratch (i.e., starting from randomly initialized weights) using our SciPy dataset.

[3] https://github.com/Instagram/MonkeyType

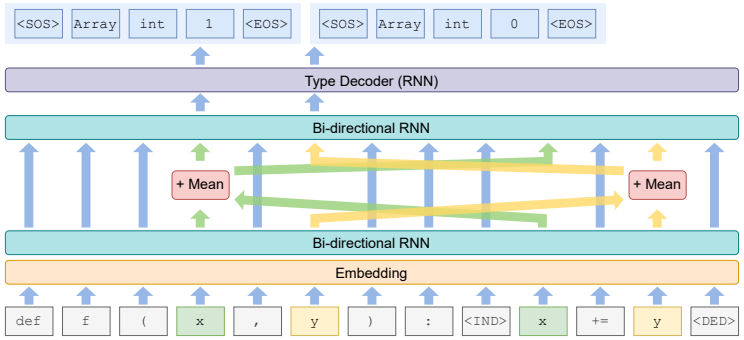
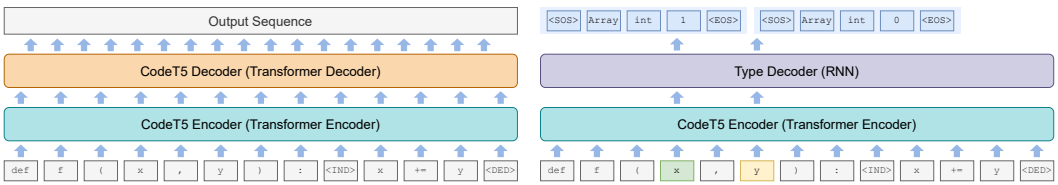


Fig. 4. Neural network architecture of DeepTyper-FS.

Neural Network Architecture. The architecture of DeepTyper-FS is shown in Fig. 4. It consists of two bi-directional RNNs with a consistency layer in between. The details of this architecture can be found in [Hellendoorn et al. 2018]. Different from the original method, we employ a subword tokenizer based on byte-pair encoding [Sennrich et al. 2016]. For type prediction, we take the second RNN's output vectors at positions of the function parameter identifiers and feed them into an RNN-based type decoder, which outputs a sequence of type tokens for each parameter.

Training Objective. The objective is to maximize the probability of inferring the correct parameter types, provided as labels in the training set, given the input source code. This is achieved by minimizing the average cross-entropy loss associated with predicting each type token across the entire training set.

4.1.5 CodeT5-FT: Fine-Tuned CodeT5. Pre-trained large Transformer models on source code have demonstrated exceptional performance in tasks like code completion, comment generation, and code synthesis [Xu et al. 2022]. In light of such capabilities, we have developed our third model for type inference by fine-tuning CodeT5, a state-of-the-art code-aware pre-trained Transformer model [Wang et al. 2021].



(a) CodeT5 (b) CodeT5-FT

Fig. 5. Neural network architectures of CodeT5 and CodeT5-FT.

Neural Network Architecture. In Fig. 5, we show the architectures of both CodeT5 and CodeT5-FT. CodeT5 employs an encoder-decoder architecture in which the tokenized input is converted into vector representations by a Transformer encoder. These encoded vectors are then decoded into an output sequence by a Transformer decoder. Our CodeT5-FT model utilizes only the CodeT5 encoder and appends to it an RNN-based type decoder (the same building block used in DeepTyper-FS in Section 4.1.4).

Fine-Tuning. We use the same dataset and loss function as in the training of DeepTyper-FS to fine-tune CodeT5-FT. The CodeT5 encoder's weights are initialized using the pre-trained weights, while the type decoder's weights are initialized randomly.

4.1.6 Zero-Shot Prompting With GPT-4. In recent years, there has been significant progress in natural language processing, which can be largely attributed to the development of large language models (LLMs) [Brown et al. 2020; Chowdhery et al. 2022; Radford et al. 2019]. GPT-4, one of the most advanced LLMs currently available, has shown outstanding performance in various professional and academic tasks [OpenAI 2023]. In this paper, we investigated the potential of GPT-4 to perform type inference through zero-shot prompting.

Please infer the types of function parameters in a program consisting of one or more functions. The syntax of types is defined as

<t> ::= <type>
| Array(<dtype>, <ndim>)
| List(<t>)
| Dict(int, <t>)
<dtype> ::= int | float | bool
<ndim> ::= 1 | 2 | 3

Please generate the top {num_preds} most probable types for each parameter using only types generated from the start symbol <t>. **Respond in the format "parameter_name: t1 | t2 | ...", with types sorted by their likelihood from highest to lowest. Please generate exactly {num_preds} types per parameter and do not include any additional information in your response.**

The program is given below:

{src}

(a) Prompt template.

Based on the code provided, I am unable to determine the exact types of the function parameters. However, I can provide the top 3 most probable types for each parameter using only types generated from the start symbol <t>:

Parameter x:
1. int
2. float
3. Array(int, 1)

Parameter y:
1. int
2. float
3. Array(int, 1)

These types are suggested based on the provided code, but they may not necessarily be the only valid types. The exact types would depend on the specific requirements and context of your program.

(b) Response without formatting instructions.

x: int | float | Array(int, 1)
y: int | float | Array(int, 1)

(c) Response with formatting instructions.

Fig. 6. Prompt template for GPT-4 and possible responses with/without formatting instructions (text in **bold**).

Zero-Shot Prompt-Based Type Inference. To use GPT-4 for type inference, we provide it with natural language instructions, or prompts. By providing a prompt as a prefix, the model generates a response message through text completion. This method does not require the model to be re-trained and we do not provide any examples in the prompt, making it a zero-shot approach. The prompt template we designed is shown in Fig. 6a.

We specify the syntax of the types to be predicted, the number of predictions required for each parameter, the program source code, and output formatting instructions in the prompt. Note that we do not provide the language's definition or the meanings of types. Since we perform type inference on a subset of Python, we expect GPT-4 to use its own knowledge of Python.

Response Formatting. Without explicit instruction on output formatting, GPT-4 responds in various formats and often provides a natural language explanation of its predictions. An example of

such responses is illustrated in Fig. 6b. This makes it challenging to integrate it into an automatic type inference and compilation pipeline. To tackle this issue, we incorporated formatting instructions into our prompts, as highlighted in bold in Fig. 6a. We have discovered that GPT-4 can comprehend these instructions and generate responses in the specified format most of the time (an example response is shown in Fig. 6c), with the occasional exception of producing fewer predictions than specified.

4.1.7 Limitations of Machine Learning Models for Concrete Type Inference. While machine learning type inference can provide useful predictions, it can have limitations in the context of concrete type inference for compilation, where the goal is to obtain consistent parameter types for compiling optimized versions of the input code. The precision of machine learning models may be inadequate, resulting in missed correct types and no valid parameter type sets. In addition, given multiple predictions per parameter, we are left with an exponential number of combinations, many of which may not be valid.

4.2 SMT Solving Type Inference

As discussed in Section 4.1.7, machine learning type inference may produce an enormous number of invalid parameter type combinations. Filtering out these invalid combinations using a type checker can be intractable when the number of combinations is large. On the other hand, constraint-based type inference has the ability to efficiently discard invalid types during constraint solving, which makes it a suitable option for addressing this issue.

In this section, we present our constraint-based concrete type inference method using an SMT solver. We begin by introducing the type encoding, followed by a detailed discussion of the generation of type constraints. We then explain how we extract valid parameter type combinations using SMT solving and discuss some limitations of this approach.

4.2.1 Type Encoding. We encode the types defined in Fig. 1 as algebraic data types, which are included in the latest SMT-LIB standard [Barrett et al. 2017] and are supported by several SMT solvers [Barbosa et al. 2022; De Moura and Bjørner 2008]. The definition of these algebraic data types, expressed in Haskell syntax for simplicity, is shown below:

```
data DType = bool | int32 | int64 | float32 | float64
data Type = None
          | Array { array_dtype :: DType, array_ndim :: Int }
          | Heap { heap_ktype :: Type, heap_vtype :: Type }
          | SparseMat { sparsemat_dtype :: DType }
          | Dict { dict_ktype :: Type, dict_vtype :: Type }
          | List { list_etype :: Type }
          | Tuple { tuple_etype :: Type, tuple_arity :: Int }
```

Accessor functions such as array_dtype and list_etype are defined to retrieve the constituting type from a given type. Tester predicates, represented by symbols like is_array and is_dict, are also defined for data types in order to determine their respective categories.

4.2.2 Type Constraints. We consider four kinds of type constraints: *type validity constraints*, *typing constraints*, *finite type space constraints*, and *uniform bit width constraints*. A detailed explanation of these constraints is presented below.

Type Validity Constraints. The type constructors defined in Section 4.2.1 can create invalid types, such as an array type with a negative rank or a dictionary type with a 2-d array key type, which is not supported. To avoid such cases, we create type validity constraints using the predicate defined

Table 2. Auxiliary definitions for constructing type constraints.

Term	Definition
libfuncnames	A set of Intrepydd library function names.
libfunctypes(id)	A mapping from a library function name id to its candidate function types.
dtype_coercion(d_1, d_2)	A function that takes in array element types, d_1 and d_2, and returns their coerced type.
is_scalar(t)	is_array(t) $\wedge$ array_ndim(t) $= 0$
is_bool(t)	$t =$ `Array(bool, 0)`
is_int(t)	is_scalar(t) $\wedge$ (array_dtype(t) $=$ `int32` $\vee$ array_dtype(t) $=$ `int64`)
is_float(t)	is_scalar(t) $\wedge$ (array_dtype(t) $=$ `float32` $\vee$ array_dtype(t) $=$ `float64`)

below (is_scalar and is_int are defined in Table 2):

$$\begin{aligned}\text{is\_valid\_type}(t) \equiv\ & (\text{is\_array}(t) \Rightarrow \text{array\_ndim}(t) \geq 0) \\ & \wedge (\text{is\_list}(t) \Rightarrow \text{is\_valid\_type}(\text{list\_etype}(t))) \\ & \wedge (\text{is\_dict}(t) \Rightarrow \text{is\_int}(\text{dict\_ktype}(t)) \wedge \text{is\_valid\_type}(\text{dict\_vtype}(t))) \\ & \wedge (\text{is\_heap}(t) \Rightarrow \text{is\_scalar}(\text{heap\_ktype}(t)) \wedge \text{is\_valid\_type}(\text{heap\_vtype}(t))) \\ & \wedge (\text{is\_tuple}(t) \Rightarrow \text{is\_valid\_type}(\text{tuple\_etype(t)}) \wedge \text{tuple\_arity}(t) \geq 0)\end{aligned}$$

We apply the is_valid_type predicate to the type of every expression in the input program and add the resulting terms as constraints.

Typing Constraint Generation. To generate typing constraints, we utilize syntax-guided constraint generation rules presented in Fig. 7, which leverage auxiliary definitions in Table 2. Expressions are typed using the judgment $\boxed{\Gamma \vdash_e e : t, C}$, where Γ is the typing context (a mapping from variable names to types), e is the expression, t is its type, and C is the set of constraints generated. For statements, including function definitions and blocks, we use the typing judgment $\boxed{\Gamma \vdash_s s \rightarrow t, C}$, where s is the statement, t is the return type of the enclosing function implied by s, and C is the set of constraints generated.

Note that most statements generate a return type of `None`, but this does not necessarily indicate that the enclosing function returns nothing. A non-`None` return type may be produced by a return statement within the same function body. Rules such as Block and If in Fig. 7 aggregate the return types implied by enclosed statements and ensure that they are consistent.

We have defined an additional set of rules for Intrepydd's library functions that constrain the types of the arguments and the returned value for each call site. Rule LibFuncId in Fig. 7 serves as the entry point for these rules. Due to space limitations, we do not list the rules for the library functions in this paper. We also support the use of existing (potentially incomplete) type annotations as additional constraints. The corresponding constraint generation rules are omitted from Fig. 7.

We refer to the typing constraints generated by the rules introduced above, along with the type validity constraints, as *type-correctness constraints*.

Finite Type Space Constraints. The type space defined in Fig. 1 is unbounded because types are defined recursively and there is no maximum value for array rank and tuple arity. This can lead to situations where a program has an unbounded number of valid parameter type combinations. This raises a problem because we need to obtain a limited number of parameter type combinations for the purpose of compilation. To mitigate this issue, we introduce additional constraints to restrict the type space of parameter types to a finite set.

Specifically, we set a maximum constant value for array rank and tuple arity, and limit the depth of the type syntax tree for parameter types. This is accomplished by defining a predicate similar

Typing judgment for expressions: $\boxed{\Gamma \vdash_e e : t, C}$ Typing judgment for statements: $\boxed{\Gamma \vdash_s s \to t, C}$

$$\frac{}{\Gamma \vdash_e boolconst : \texttt{Array}(d, \texttt{bool}), \emptyset}\ \textsc{Bool} \qquad \frac{t \text{ fresh}}{\Gamma \vdash_e intconst : t, \{\text{is\_int}(t)\}}\ \textsc{Int} \qquad \frac{t \text{ fresh}}{\Gamma \vdash_e floatconst : t, \{\text{is\_float}(t)\}}\ \textsc{Float}$$

$$\frac{id \in \text{libfuncnames} \qquad U \equiv \text{libfunctypes}(id) \qquad t \text{ fresh}}{\Gamma \vdash_e id : t, \left\{\bigvee_{u \in U} t = u\right\}}\ \textsc{LibFuncId} \qquad \frac{id \notin \text{libfuncnames} \qquad \Gamma(id) = t}{\Gamma \vdash_e id : t, \emptyset}\ \textsc{Id}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_1 : t_1, C_1 \qquad \Gamma \vdash_e e_2 : t_2, C_2 \\ d \equiv \text{dtype\_coercion}(\text{array\_dtype}(e_1), \text{array\_dtype}(e_2)) \\ n \equiv \max(\text{array\_ndim}(e_1), \text{array\_ndim}(e_2)) \\ C' \equiv \{\text{is\_array}(e_1), \text{is\_array}(e_2)\}\end{array}}{\Gamma \vdash_e e_1\ opa\ e_2 : \texttt{Array}(d, n), C' \cup C_1 \cup C_2}\ \textsc{ArithBin}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_1 : t_1, C_1 \qquad \Gamma \vdash_e e_2 : t_2, C_2 \\ n \equiv \max(\text{array\_ndim}(e_1), \text{array\_ndim}(e_2)) \\ C' \equiv \{\text{is\_array}(e_1), \text{is\_array}(e_2)\}\end{array}}{\Gamma \vdash_e e_1\ opb\ e_2 : \texttt{Array}(\texttt{bool}, n), C' \cup C_1 \cup C_2}\ \textsc{BoolBin}$$

$$\frac{\Gamma \vdash_e e : t, C \qquad d \equiv \begin{cases}\text{array\_dtype}(e) & opu \text{ is } + \text{ or } - \\ \texttt{bool} & opu \text{ is not}\end{cases}}{\Gamma \vdash_e opu\ e : \texttt{Array}(d, \text{array\_ndim}(e)), \{\text{is\_array}(e)\} \cup C}\ \textsc{Unary}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_1 : t_1, C_1 \quad \ldots \quad \Gamma \vdash_e e_n : t_n, C_n \\ t' \text{ fresh} \qquad C' \equiv \{t_1 = t', \ldots, t_n = t'\}\end{array}}{\Gamma \vdash_e (e_1, \ldots, e_n) : \texttt{Tuple}(t', n), C' \cup C_1 \cup \ldots \cup C_n}\ \textsc{Tuple}$$

$$\frac{\Gamma \vdash_e e_1 : t_1, C_1 \quad \ldots \quad \Gamma \vdash_e e_n : t_n, C_n \qquad t' \text{ fresh} \qquad C' \equiv \{t_1 = t', \ldots, t_n = t'\}}{\Gamma \vdash_e [e_1, \ldots, e_n] : \texttt{List}(t'), C' \cup C_1 \cup \ldots \cup C_n}\ \textsc{List}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_{1,1} : t_{1,1}, C_{1,1} \quad \Gamma \vdash_e e_{1,2} : t_{1,2}, C_{1,2} \quad \ldots \quad \Gamma \vdash_e e_{n,1} : t_{n,1}, C_{n,1} \quad \Gamma \vdash_e e_{n,2} : t_{n,2}, C_{n,2} \\ t'_1, t'_2 \text{ fresh} \qquad C' \equiv \{t_{1,1} = t'_1, t_{1,2} = t'_2, \ldots, t_{n,1} = t'_1, t_{n,2} = t'_2\}\end{array}}{\Gamma \vdash_e \{e_{1,1} : e_{1,2}, \ldots, e_{n,1} : e_{n,2}\} : \texttt{Dict}(t'_1, t'_2), C' \cup C_{1,1} \cup C_{1,2} \cup \ldots \cup C_{n,1} \cup C_{n,2}}\ \textsc{Dict}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e : t, C \qquad \Gamma \vdash_e e_1 : t_1, C_1 \qquad t' \text{ fresh} \\ C' \equiv \{(\text{is\_array}(t) \wedge \text{array\_ndim}(t) = 1 \wedge \text{is\_int}(t_1) \wedge t' = \texttt{Array}(\text{array\_dtype}(t), 0)) \vee \\ (t = \texttt{List}(t') \wedge \text{is\_int}(t_1)) \vee t = \texttt{Dict}(t_1, t')\}\end{array}}{\Gamma \vdash_e e[e_1] : t', C' \cup C \cup C_1}\ \textsc{SubscriptSingle}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e : t, C \quad \Gamma \vdash_e e_1 : t_1, C_1 \quad \ldots \quad \Gamma \vdash_e e_n : t_n, C_n \qquad n > 1 \qquad t' \text{ fresh} \\ C' \equiv \{\text{is\_array}(t) \wedge \text{array\_ndim}(t) = n \wedge \text{is\_int}(t_1) \wedge \ldots \wedge \text{is\_int}(t_n) \wedge t' = \texttt{Array}(\text{array\_dtype}(t), 0)\}\end{array}}{\Gamma \vdash_e e[e_1, \ldots, e_n] : t', C' \cup C \cup C_1 \cup \ldots \cup C_n}\ \textsc{SubscriptMulti}$$

$$\frac{\Gamma \vdash_e id : t, C \quad \Gamma \vdash_e e_1 : t_1, C_1 \quad \ldots \quad \Gamma \vdash_e e_n : t_n, C_n \qquad t' \text{ fresh}}{\Gamma \vdash_e id(e_1, \ldots, e_n) : t', \{t = \texttt{Function}((t_1, \ldots, t_n), t')\} \cup C \cup C_1 \cup \ldots \cup C_n}\ \textsc{Call}$$

$$\frac{\Gamma \vdash_e e : t, C}{\Gamma \vdash_s \texttt{return}\ e \to t, C}\ \textsc{Return} \qquad \frac{\Gamma \vdash_e e_1 : t_1, C_1 \qquad \Gamma \vdash_e e_2 : t_2, C_2}{\Gamma \vdash_s e_1 = e_2 \to \text{None}, \{t_1 = t_2\} \cup C_1 \cup C_2}\ \textsc{Assign}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_1 : t_1, C_1 \qquad \Gamma \vdash_e e_2 : t_2, C_2 \\ C' \equiv \{\text{is\_array}(e_1), \text{is\_array}(e_2), \text{array\_ndim}(e_1) \geq \text{array\_ndim}(e_2), \\ \text{array\_dtype}(e_1) = \text{dtype\_coercion}(\text{array\_dtype}(e_1), \text{array\_dtype}(e_2))\}\end{array}}{\Gamma \vdash_s e_1\ opa = e_2 \to \text{None}, C' \cup C_1 \cup C_2}\ \textsc{AugAssign}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e_1 : t_1, C_1 \qquad \Gamma \vdash_e e_2 : t_2, C_2 \qquad \Gamma \vdash_s b \to t_3, C_3 \\ C' \equiv \{(\text{is\_array}(t_2) \wedge \text{array\_ndim}(t_2) = 1 \wedge t_1 = \texttt{Array}(\text{array\_dtype}(t_2), 0)) \vee \\ t_2 = \texttt{List}(t_1) \vee (\text{is\_dict}(t_2) \wedge \text{dict\_ktype}(t_2) = t_1)\}\end{array}}{\Gamma \vdash_s \texttt{for}\ e_1\ \texttt{in}\ e_2 : b \to t_3, C' \cup C_1 \cup C_2 \cup C_3}\ \textsc{For}$$

$$\frac{\Gamma \vdash_e e : t_1, C_1 \qquad \Gamma \vdash_s b \to t_2, C_2}{\Gamma \vdash_s \texttt{while}\ e : b \to t_2, \{\text{is\_bool}(t_1)\} \cup C_1 \cup C_2}\ \textsc{While}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e e : t, C \qquad \Gamma \vdash_s b_1 \to t_1, C_1 \qquad \Gamma \vdash_s b_2 \to t_2, C_2 \qquad t' \text{ fresh} \\ C' \equiv \{t_1 = \text{None} \vee t_1 = t', t_2 = \text{None} \vee t_2 = t', \\ t_1 \neq \text{None} \vee t_2 \neq \text{None} \vee t' = \text{None}, \text{is\_bool}(t)\}\end{array}}{\Gamma \vdash_s \texttt{if}\ e : b_1\ \texttt{else} : b_2 \to t', C' \cup C_1 \cup C_2}\ \textsc{If}$$

$$\frac{\begin{array}{c}\Gamma \vdash_s s_1 \to t_1, C_1 \quad \ldots \quad \Gamma \vdash_s s_n \to t_n, C_n \qquad t' \text{ fresh} \\ C' \equiv \left\{t_1 = \text{None} \vee t_1 = t', \ldots, t_n = \text{None} \vee t_n = t', \right. \\ \left. \left(\bigvee_{i=1}^{n} t_i \neq \text{None}\right) \vee t' = \text{None}\right\}\end{array}}{\Gamma \vdash_s s_1 \ldots s_n \to t', C' \cup C_1 \cup \ldots \cup C_n}\ \textsc{Block}$$

$$\frac{\begin{array}{c}\Gamma \vdash_e id : t, C \qquad \Gamma, id_1 : t_1, \ldots, id_n : t_n \vdash_s b \to t_r, C_b \\ C' \equiv \{t = \texttt{Function}((t_1, \ldots, t_n), t_r)\}\end{array}}{\Gamma \vdash_s \texttt{def}\ id(id_1, \ldots, id_n) : b \to \text{None}, C' \cup C \cup C_b}\ \textsc{Func}$$

Fig. 7. Typing constraint generation rules.

to is_valid_type in Section 4.2.2, but with the recursion unfolded a constant number of times. We apply this predicate to parameter types, and incorporate the resulting terms as constraints.

Uniform Bit Width Constraints. Another major reason for an excessive number of valid parameter type combinations is that, in many cases, array element types with different bit widths can be used interchangeably. This results in an exponential increase in the number of type combinations when compared to cases where bit width distinctions are ignored. To mitigate this issue, we enforce uniform bit widths for integers and floating-point numbers in parameter types within a program. Specifically, we ensure that the integer element types of arrays in parameter types are either all `int32` or all `int64`, and the same applies to floating-point numbers (all `float32` or all `float64`). However, it is still possible for integer types and floating-point types within a program to have different bit widths. These constraints can be generated by modifying the predicate used for generating the finite type space constraints. Specifically, for an array type t, we add the following constraint:

$$\text{array\_dtype}(t) = dti \lor \text{array\_dtype}(t) = dtf \lor \text{array\_dtype}(t) = \texttt{bool}$$

where dti and dtf are shared within a program and subject to:

$$(dti = \texttt{int32} \lor dti = \texttt{int64}) \land (dtf = \texttt{float32} \lor dtf = \texttt{float64})$$

4.2.3 Valid Type Combination Extraction. The objective of our concrete type inference approach is to acquire valid combinations of function parameter types. Merely invoking the SMT solver to solve all constraints is insufficient, as it only generates a single set of type assignments or declares unsatisfiability in cases of type errors. To extract all valid type combinations adhering to the constraints gathered from an input program, we use a simple model enumeration algorithm described in Algorithm 1.

Algorithm 1: Valid parameter type combination extraction.

Function `extractTypes`$(C, t_1, \ldots, t_n)$
  Input: C: a set of constraints to satisfy;
    $t_1, \ldots, t_n$: all parameter type variables in the input program.
  Output: A set of parameter type combinations satisfying C.
1 $T \leftarrow \emptyset$
2 **while** *True*
3 $sat, assignments \leftarrow$ `solve`(C)
4 **if** $\neg sat$
5 break
6 $\tau_1, \ldots, \tau_n \leftarrow$ values of $t_1, \ldots, t_n$ obtained from *assignments*
7 $T \leftarrow T \cup \{(\tau_1, \ldots, \tau_n)\}$
8 $C \leftarrow C \cup \{\neg(t_1 = \tau_1 \land \ldots \land t_n = \tau_n)\}$
9 **return** T

When the number of valid parameter type combinations is unbounded, the algorithm cannot terminate. Even with the addition of the finite type space constraints, the enumeration of a large number of valid parameter type combinations can still be time-consuming. In practice, we set a limit on the number of valid combinations that can be discovered, as well as a time limit. Whenever these limits are exceeded, the algorithm immediately terminates and reports a failure.

4.2.4 Limitations of SMT Solving for Concrete Type Inference. While SMT Solving-based type inference is capable of generating only type-correct parameter type sets, it is susceptible to producing an overwhelming number of valid type combinations. It lacks an inherent method to prioritize the

discovery of type combinations that are more likely to be intended. As a result, when too many solutions are present, we may have to declare a failure.

4.3 Combining Machine Learning with SMT Solving

In this section, we present our approach to integrating machine learning and SMT solving in concrete type inference for code optimization, with the aim of overcoming their respective limitations. We will first present a straightforward approach that encodes the predictions of machine learning models as constraints. Then, we will move on to an effective approach that tackles the limitations of the first one with a progressive relaxation mechanism for machine learning constraints.

4.3.1 Machine Learning Predictions as Constraints. As discussed in Sections 4.1.7 and 4.2.4, machine learning type inference suffers from the issue of producing numerous inconsistent parameter type combinations, while type inference though SMT solving generates a vast number of valid type combinations, without a clear way to determine the most probable ones. To overcome these limitations, an intuitive approach is to combine the two methods by taking the intersection of their respective solution sets.

Consider a program with n parameters, where the type of each parameter t_i is unknown. Let the machine learning model predict the top-k types for each parameter, denoted by $\tau_{i,1}, \ldots, \tau_{i,k}$. To generate the intersection of the solution sets from two methods, we can add an additional constraint to the SMT solver:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} t_i = \tau_{i,j}$$

We view this constraint as a conjunction of machine learning constraints, where each constraint enforces a parameter to be assigned a type from the model's predictions. Once we include this constraint in the constraint set, we can use Algorithm 1 to obtain the new solution set. However, this simple approach contains some major issues:

First, as mentioned in Section 4.1.7, machine learning models may have trouble predicting consistent combinations of parameter types due to a reduced prediction space or insufficient precision. As a result, after intersecting with the solution set of SMT solving type inference, the outcome could be an empty set.

Second, determining the appropriate value of k can be challenging. A small value of k decreases the likelihood of generating excessive parameter type combinations, but increases the chances of excluding the desired type combination. Conversely, a large value of k may result in an exceedingly large number of type combinations, leading to higher computational costs or even failures.

Furthermore, the best value of k may not be universal for all programs or even all parameters in the same program. For example, consider a parameter named `num_rows`. It is apparent that the parameter type is more likely to have an `int` type than any other type. A machine learning model can easily capture this bias and identify `int` as its top-1 prediction. In this case, a small k would be beneficial. However, for a parameter named `x`, its type could be less apparent to the model, and it might be necessary to consider more predictions using a larger k.

4.3.2 Progressive Relaxation of Machine Learning Constraints. To address the issues associated with using machine learning predictions as constraints, we propose a method of relaxing these constraints progressively. To avoid situations where the machine learning model fails to predict consistent types and results in an empty solution set, our method allows dropping the machine learning constraint for certain parameters and relying on the SMT solver to determine the potential types. Our method also incorporates the ordering of the machine learning model's predictions for

each parameter, enabling the prioritization of more probable type candidates during the extraction of valid parameter type combinations. The details of this method are described below.

First, we choose a list $K = [k_1, \ldots, k_m]$ consisting of m values of k (used to represent top-k predictions), ordered in ascending order. An empirical selection for K is $[1, 5, 10, 20]$, which we employed in our experiments. Let n be the number of function parameters in the input program, and $t_1, \ldots, t_n$ be the type variables associated with the function parameters in the program. We also denote the set of top-k predictions for t_i obtained from the machine learning model as $P_{i,k}$. Then we can define a scoring function for parameter type combinations as follows:

$$\text{score}(t_1, \ldots, t_n) = \sum_{i=1}^{n} \sum_{j=1}^{m} n^{m-j} \cdot \mathbb{1}_{t_i \in P_{i,k_j}}$$

where $\mathbb{1}_c$ is the indicator function that evaluates to 1 when c is *true* and 0 otherwise.

This scoring function encodes a lexicographical order for parameter type combinations based on the likelihood of each parameter type predicted by the machine learning model. For example, suppose $K = [1, 5]$. When comparing two type combinations, we begin by examining how many top-1 predictions each combination includes. The one with more top-1 predictions will receive a higher score. If both combinations have the same number of top-1 predictions, we will compare how many top-5 predictions are included in each. The combination with more top-5 predictions will get a higher score. If both combinations have the same number of top-1 and top-5 predictions, they will have the same score.

It is important to note that the scoring function can be expressed as an SMT function using conditional expressions (**ite**) and thus we can enforce a minimum threshold score (s) for all parameter type combinations in the solution set by adding an SMT constraint as shown below:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} \textbf{ite}\left(\bigvee_{l=1}^{k_j} t_i = \tau_{i,l},\ n^{m-j},\ 0 \right) \geq s$$

where $\tau_{i,l}$ represents the top-l prediction made by the model for t_i.

By setting the threshold score to 0, we effectively disregard all machine learning predictions, while setting it to $\sum_{i=1}^{m} n^i$ requires all parameters to use the top-1 prediction.

The next step is to determine the threshold score. Initially, it is set as the maximum value that results in a non-empty solution set. Since the size of the solution set decreases monotonically with the increase of the threshold, binary search can be employed to quickly find the desired value. Specifically, we use Algorithm 2 with $s_{max} = \sum_{i=1}^{m} n^i$.

Once the threshold score has been determined, we enforce it by adding the SMT constraint discussed above. Then we can utilize Algorithm 1 to generate a consistent and highly probable set of parameter type combinations. However, it is possible that only a single type combination is generated. For instance, this situation may arise if the set of top-1 predictions is type-correct. Given the potential imprecision of machine learning models, it is advisable to generate multiple parameter type combinations to increase the likelihood of finding the intended combination. To accomplish this, we set a minimum number of type combinations to generate. This is achieved by contiguously decreasing the threshold score until the size of the solution set meets the requirement, or no more valid solutions can be found. A detailed description of this process is presented in Algorithm 3.

5 EVALUATION

In this section, we present the evaluation results for our concrete type inference methods. First, we provide our experimental setup. Next, we evaluate the performance of our machine learning methods on the test set and the benchmarks. Following this, we present the number of solutions

Algorithm 2: Finding the maximum value of the threshold score.

Function `findMaxThresholdScore`$(C, s_{max}, t_1, \ldots, t_n)$
- **Input:** C: a set of constraints to satisfy;
  s_{max}: the maximum threshold score to consider;
  $t_1, \ldots, t_n$: all parameter type variables in the input program.
- **Output:** The maximum value of the threshold score that leads to a non-empty set of parameter type combinations if C is satisfiable and -1 otherwise.

1 $left \leftarrow 0$
2 $right \leftarrow s_{max}$
3 $s \leftarrow -1$
4 **while** $left \leq right$
5 $\quad mid = (left + right)/2$
6 $\quad sat, assignments \leftarrow$ `solve`$(C \cup \{$`score`$(t_1, \ldots, t_n) \geq mid\})$
7 $\quad$ **if** sat
8 $\quad\quad left \leftarrow mid + 1$
9 $\quad\quad s \leftarrow mid$
10 $\quad$ **else**
11 $\quad\quad right \leftarrow mid - 1$
12 **return** s

Algorithm 3: Extracting the required number of parameter type combinations while progressively decreasing the threshold score.

Function `extractTypesWithThresholdScoreDesc`$(C, n_{sols}, t_1, \ldots, t_n)$
- **Input:** C: a set of constraints to satisfy;
  n_{sols}: the minimum number of solutions required;
  $t_1, \ldots, t_n$: all parameter type variables in the input program.
- **Output:** A set of parameter type combinations.

1 $s_{max} \leftarrow \sum_{i=1}^{m} n^i$
2 $T \leftarrow \emptyset$
3 **while** *True*
4 $\quad s \leftarrow$ `findMaxThresholdScore`$(C, s_{max}, t_1, \ldots, t_n)$
5 $\quad$ **if** $s < 0$
6 $\quad\quad$ break
7 $\quad U \leftarrow$ `extractTypes`$(C \cup \{$`score`$(t_1, \ldots, t_n) \geq s\}, t_1, \ldots, t_n)$
8 $\quad T \leftarrow T \cup U$
9 $\quad$ **if** $|T| \geq n_{sols} \vee s = 0$
10 $\quad\quad$ break
11 $\quad$ **for** $(\tau_1, \ldots, \tau_n)$ **in** U
12 $\quad\quad C \leftarrow C \cup \{\neg(t_1 = \tau_1 \wedge \ldots \wedge t_n = \tau_n)\}$
13 $\quad s_{max} \leftarrow s - 1$
14 **return** T

for SMT type inference problem instances from the benchmarks with different options. Then, we demonstrate the effectiveness of combining machine learning with SMT solving for type inference. Finally, we perform an end-to-end performance evaluation by integrating our combined type inference method into the Intrepydd compiler.

5.1 Experimental Setup

5.1.1 Benchmarks. The benchmarks used in our experiments consist of 10 Python programs. Six of them, namely bigCLAM [Romijnders 2017], changepoint [Johnson 2018], ipnsw [Johnson 2019a], ISTA [Johnson 2019b], PR-Nibble [Johnson 2019b], and sinkhorn_wmd [Johnson 2019c] were used in the Intrepydd paper [Zhou et al. 2020] and represent data analytics workloads. The remaining four programs, cholesky, jacobi_2d, lu, and ludcmp, were selected from the PolyBench/Python benchmarks [Abella-González et al. 2021] due to their compatibility with Intrepydd's optimizations. All of these benchmarks came with type annotations. To evaluate our type inference methods in a type-annotation-free setting, we removed all type annotations from these benchmarks in the input and used the type annotations as ground truth labels for evaluation.

Table 3. Benchmark statistics.

Benchmarks	#Funcs	#Params
bigCLAM	4	9
changepoint	2	3
ipnsw	3	42
ISTA	1	10
PR-Nibble	1	7
sinkhorn_wmd	1	9
cholesky	1	2
jacobi_2d	1	4
lu	1	2
ludcmp	1	5

Each benchmark program consists of two parts: the main program code in Python and the kernel code in Intrepydd, i.e., in a Python subset augmented with type annotations. The main program invokes kernel functions defined in the kernel code, which is the target for concrete type inference and compiler optimizations. The number of functions and the total number of function parameters in each benchmark kernel are shown in Table 3. Despite their modest size, these programs can present significant difficulties for type inference. Although the number of functions is small, the relatively larger number of parameters creates a notable challenge in generating a reasonable number of versions for code optimization. Smaller programs typically have fewer type constraints, which can result in a large number of valid solutions produced by SMT-based type inference. Additionally, the limited context (i.e., source code) available for ML models can negatively impact the accuracy of their predictions.

Ensuring that the benchmarks are not included in the training data of the machine learning models is crucial to the validity of the evaluation results. The source code for the type-annotated benchmarks used in our evaluation has only been stored in access-controlled repositories, and has never been made available publicly. Although some of the benchmark programs are derived from publicly accessible code, their original versions do not contain our type annotations. Therefore, it is highly unlikely that the machine learning models have been exposed to our type-annotated benchmarks.

5.1.2 The SciPy Dataset. As detailed in Section 4.1.2, we have created a dataset of run-time types from SciPy's test suite. The dataset includes 17,665 unique samples, where each sample is a pair consisting of the source code of a function and its corresponding argument types. However, there are instances of pairs with identical functions but different run-time argument types. Specifically, the dataset comprises 3,369 unique functions, with each function appearing in at least one sample and at most 461 samples. Of all the samples, 13,599 contain the `Array` type while 288 contain the `Dict` type. We randomly split the dataset into three parts: a training set containing 90% of the data, a validation set containing 5%, and a test set containing another 5%.

5.1.3 Machine Learning Models. A summary of the models is included below, including the training process and the inference methodologies.

Freq. The Freq model is trained by iterating through the training set and keeping track of the occurrence count for each observed type. During inference, it provides the most frequent k types as its top-k predictions.

DeepTyper-FS. As shown in Section 4.1.4, the non-decoder part of DeepTyper-FS follows the same specifications as DeepTyper, with an embedding size of 300 and RNNs implemented as gated recurrent units (GRUs) [Cho et al. 2014] with a hidden state size of 650. The RNN type decoder is a single-layer GRU with a hidden state size of 128. To train the model, we employed the AdamW optimizer [Loshchilov and Hutter 2019] with a learning rate of 10^{-3} and a minibatch size of 32 for a total of 50 epochs. We made a model checkpoint after each epoch and selected the one that resulted in the lowest validation loss for subsequent evaluations. To obtain its top-k predictions during inference, we utilized beam search with a beam size of 64.

CodeT5-FT. As detailed in Section 4.1.5, we based our CodeT5-FT model on the pre-trained CodeT5-large model with 770M parameters [Le et al. 2022]. We added a randomly initialized RNN type decoder that is identical to the one used in DeepTyper-FS to its Transformer encoder. The entire model was trained on our training set using the AdamW optimizer with a learning rate of 5×10^{-5} and a minibatch size of 16 for 10 epochs. We followed the same checkpoint selection and top-k inference procedure as in DeepTyper-FS.

GPT-4. There is no training involved in our zero-shot prompt-based method. We generated prompts using the template shown in Fig. 6a and fed them into GPT-4's API to obtain responses. The sampling temperature was set to 1. To ensure that our benchmarks were not part of the training data of the evaluated model, we consistently used the gpt-4-0314 model, which is the initial public version of GPT-4 released on March 14, 2023.

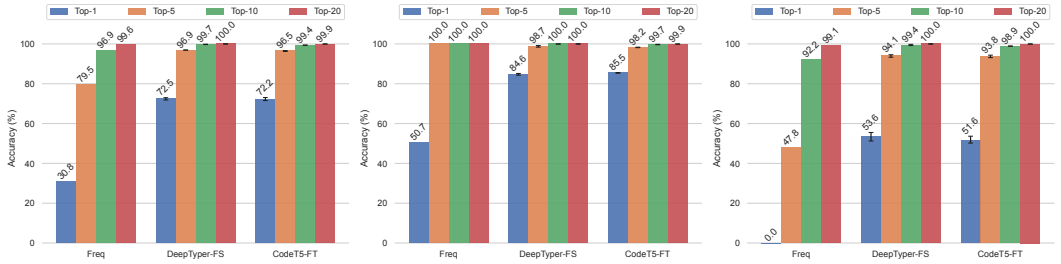
Addressing Randomness in Evaluation. Given the stochastic nature of deep learning model training, we trained DeepTyper-FS and CodeT5-FT three times with different random seeds. As for GPT-4, since its output is randomly sampled each time, we let it make three rounds of type inference for all benchmarks. We report the results from all runs when evaluating the accuracy of prediction. We then chose one trained model each for DeepTyper-FS and CodeT5-FT, as well as the predictions from one round for GPT-4, to be used in all subsequent evaluations. The selection criteria are discussed later in Section 5.2.2.

5.1.4 SMT Solver and Resource Budgets. In our experiments, we used cvc5 [Barbosa et al. 2022] as the SMT solver. The enumeration of solutions in our SMT-based methods can be time-consuming, particularly when the number of solutions is large. For practical purposes, we imposed a timeout of 5 minutes and a maximum threshold of 1,000 solutions to be discovered. Additionally, when evaluating the constraint relaxation method proposed in Section 4.3.2, we set the minimum solution count threshold to 10.

5.1.5 Hardware Platform. We conducted all our experiments, except for the training of machine learning models, on a desktop machine running Ubuntu 20.04.5. It is equipped with an Intel Core i5-7600 CPU (@3.5GHz) and 64GB of RAM. There is no discrete GPU on this platform and all local model inference was performed using the CPU. For training DeepTyper-FS and CodeT5-FT, we used one NVIDIA Quadro RTX 6000 GPU on a separate machine.

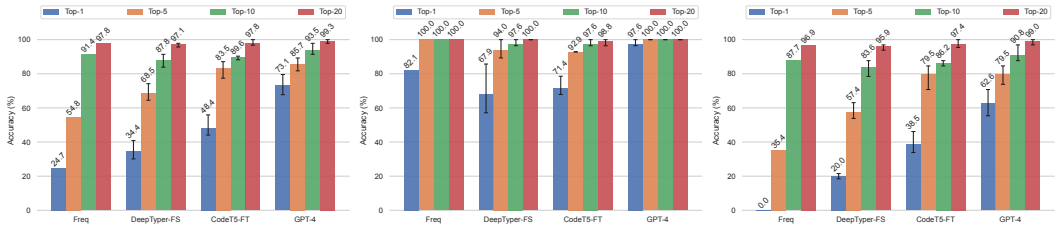
5.2 Evaluation of Machine Learning Type Inference

We assessed our machine learning based type inference models by evaluating them on both accuracy and consistency. Our evaluations were conducted on the test subset of the SciPy dataset and the benchmarks.



(a) All types. (b) Scalar types. (c) Non-scalar types.

Fig. 8. Accuracy on the test subset of the SciPy dataset (%). For DeepTyper-FS and CodeT5-FT, the bar height is the average accuracy of 3 models trained with different random seeds, with range bars showing min/max values.



(a) All types. (b) Scalar types. (c) Non-scalar types.

Fig. 9. Accuracy on the benchmarks (%). For DeepTyper-FS and CodeT5-FT, the bar height is the average accuracy of 3 models trained with different random seeds. For GPT-4, the bar height is the average accuracy of 3 rounds for predictions for all benchmarks. The range bars show min/max values from different models/rounds.

5.2.1 Prediction Accuracy. We first evaluated the machine learning models by their accuracy in predicting types for each function parameter. Fig. 8 shows the top-k accuracy of the models on the test subset of the SciPy dataset. We did not evaluate the performance of GPT-4 on the test set because it was not trained on the corresponding training set and we did not have access to its API for enabling large-scale evaluation. The figure indicates that the Freq model serves as a reasonably performing baseline, having a top-10 accuracy of over 90%. This suggests that the dataset's types are concentrated within the few most commonly occurring types. Both the DeepTyper-FS and CodeT5-FT models, which employ deep learning techniques, were able to surpass the baseline by a significant margin, particularly for non-scalar types, which are relatively infrequent in the dataset. However, no significant performance difference was observed between these two models on the test set.

The accuracy of all four machine learning models on the ten benchmarks is depicted in Fig. 9. Although DeepTyper-FS and CodeT5-FT performed similarly on the test set, CodeT5-FT exhibited better performance than DeepTyper-FS on the benchmarks. This observation suggests that CodeT5-FT may generalize better to the benchmark programs, which differ from the Python programs in the SciPy dataset. Remarkably, GPT-4 demonstrated outstanding zero-shot generalization ability on the benchmarks, surpassing all other models without any training on our dataset.

5.2.2 Prediction Consistency. Starting from this evaluation, we used one model checkpoint for DeepTyper-FS and CodeT5-FT, as well as the predictions from one round of type inference for

Table 4. Number of possible and type-correct parameter type combinations produced by machine learning models. The table displays the number of potential parameter type combinations generated by machine learning models (ML), alongside the number of type-correct combinations produced by these models (ML+TC).

Benchmark	Category	Freq				DeepTyper-FS			
		Top-1	Top-5	Top-10	Top-20	Top-1	Top-5	Top-10	Top-20
bigCLAM	ML	512	3.9×10^{8}	3.2×10^{11}	1.3×10^{14}	512	8.1×10^{8}	4.6×10^{11}	1.4×10^{14}
	ML + TC	0	1	3	3	0	0	8	8
changepoint	ML	8	729	6859	5.1×10^{4}	8	900	6498	5.5×10^{4}
	ML + TC	0	0	0	0	0	0	0	0
ipnsw	ML	4.4×10^{12}	1.2×10^{40}	5.1×10^{53}	7.3×10^{65}	2.2×10^{12}	1.2×10^{41}	3.2×10^{53}	3.1×10^{65}
	ML + TC	0	0	> 1000	> 1000	0	0	0	> 1000
ISTA	ML	1024	3.5×10^{9}	6.1×10^{12}	4.8×10^{15}	1024	6.6×10^{9}	6.1×10^{12}	4.6×10^{15}
	ML + TC	0	0	88	88	0	0	0	> 1000
PR-Nibble	ML	128	4.8×10^{6}	8.9×10^{8}	9.5×10^{10}	128	7.2×10^{6}	9.4×10^{8}	1.1×10^{11}
	ML + TC	0	0	980	> 1000	0	0	> 1000	> 1000
sinkhorn_wmd	ML	512	3.9×10^{8}	3.2×10^{11}	1.3×10^{14}	512	9.0×10^{8}	4.4×10^{11}	1.3×10^{14}
	ML + TC	0	0	556	556	0	0	> 1000	> 1000
cholesky	ML	4	81	361	1369	4	90	361	1443
	ML + TC	0	2	2	2	2	2	2	2
jacobi_2d	ML	16	6561	1.3×10^{5}	1.9×10^{6}	16	8100	1.4×10^{5}	2.0×10^{6}
	ML + TC	0	8	8	8	8	12	12	12
lu	ML	4	81	361	1369	4	90	342	1443
	ML + TC	0	2	2	2	2	2	2	2
ludcmp	ML	32	5.9×10^{4}	2.5×10^{6}	6.9×10^{7}	32	9.0×10^{4}	2.5×10^{6}	6.6×10^{7}
	ML + TC	0	1	13	31	2	4	8	30

Benchmark	Category	CodeT5-FT				GPT-4			
		Top-1	Top-5	Top-10	Top-20	Top-1	Top-5	Top-10	Top-20
bigCLAM	ML	512	9.0×10^{8}	4.6×10^{11}	1.9×10^{14}	512	8.1×10^{8}	3.2×10^{11}	1.9×10^{14}
	ML + TC	0	8	8	8	0	0	10	10
changepoint	ML	8	900	6859	5.2×10^{4}	8	1000	6859	5.5×10^{4}
	ML + TC	0	0	0	0	0	0	0	0
ipnsw	ML	4.4×10^{12}	4.8×10^{41}	1.5×10^{54}	6.4×10^{65}	4.4×10^{12}	1.2×10^{40}	4.8×10^{51}	2.1×10^{64}
	ML + TC	0	0	0	> 1000	0	> 1000	> 1000	> 1000
ISTA	ML	1024	4.8×10^{9}	5.2×10^{12}	4.9×10^{15}	1024	5.9×10^{9}	2.7×10^{12}	2.1×10^{15}
	ML + TC	0	0	> 1000	> 1000	0	> 1000	> 1000	> 1000
PR-Nibble	ML	128	7.3×10^{6}	9.9×10^{8}	1.2×10^{11}	128	4.8×10^{6}	4.9×10^{8}	5.3×10^{10}
	ML + TC	0	0	0	> 1000	64	> 1000	> 1000	> 1000
sinkhorn_wmd	ML	512	8.1×10^{8}	3.5×10^{11}	1.5×10^{14}	512	3.9×10^{8}	1.2×10^{11}	6.1×10^{13}
	ML + TC	0	1000	> 1000	> 1000	0	240	320	640
cholesky	ML	4	90	360	1480	4	90	340	1156
	ML + TC	2	2	2	2	2	2	2	2
jacobi_2d	ML	16	8100	1.4×10^{5}	2.0×10^{6}	16	6561	8.4×10^{4}	1.3×10^{6}
	ML + TC	8	12	12	12	8	12	12	12
lu	ML	4	90	380	1404	4	81	289	1156
	ML + TC	2	2	2	2	2	2	2	2
ludcmp	ML	32	9.0×10^{4}	3.0×10^{6}	7.5×10^{7}	32	5.9×10^{4}	1.4×10^{6}	4.5×10^{7}
	ML + TC	2	8	24	45	2	20	45	45

GPT-4. To avoid bias, we selected the model checkpoints based on the best top-1 accuracy on the test set instead of the benchmarks. For GPT-4, however, we didn't know its accuracy on the test set. We chose the round that produced the *lowest* top-1 accuracy on the benchmarks since GPT-4 was likely to be the top-performing model in all experiments.

Table 4 shows the number of function parameter type combinations implied by the machine learning models' predictions (ML rows), and how many of these combinations are consistent, i.e., type-correct (ML+TC rows). The former data is analytically calculated from the predictions, while the latter is obtained using Algorithm 1 with the type-correctness constraints defined in Section 4.2 and machine learning constraints generated as in Section 4.3.1.

The ML rows demonstrate that with an increase in the number of predictions taken into consideration, the quantity of potential type combinations expands exponentially. However, only a small proportion of these combinations are consistent in many cases. Using only a few predictions per parameter could result in the absence of valid type combinations. On the other hand, if more predictions are considered, there could be a much larger number of invalid combinations that need to be filtered out, which can be prohibitively expensive. Even if after filtering out all invalid combinations, the resulting solutions might still be excessively numerous.

By comparing the ML+TC rows across models, we can observe that more accurate models tend to yield more consistent predictions. For example, GPT-4 was able to produce valid combinations for PR-Nibble using only top-1 predictions, whereas the other three models were unable to do so even with top-5 predictions.

5.3 Evaluation of SMT Solving Type Inference

We evaluated the efficacy of SMT solving for type inference by executing Algorithm 1 with type-correctness constraints (TC), finite type space constraints (FTC), and uniform bit width constraints (UBW), as defined in Section 4.2. We counted the resulting number of parameter type combinations and versions of functions. A function version is a unique pair of a function and associated parameter types. The results are shown in Table 5.

Table 5. Number of parameter type combinations and versions of functions produced by SMT-based type inference. TC = Type-Correctness, FTS = Finite Type Space, UBW = Uniform Bit Width.

Benchmark	TC		TC + FTS		TC + FTS + UBW	
	#Combinations	#Functions	#Combinations	#Functions	#Combinations	#Functions
bigCLAM	10	21	10	21	3	10
changepoint	>1000	-	186	310	186	310
ipnsw	>1000	-	>1000	-	>1000	-
ISTA	>1000	-	>1000	-	54	54
PR-Nibble	>1000	-	>1000	-	>1000	-
sinkhorn_wmd	>1000	-	>1000	-	243	243
cholesky	>1000	-	2	2	2	2
jacobi_2d	>1000	-	12	12	8	8
lu	>1000	-	2	2	2	2
ludcmp	90	90	45	45	27	27

As shown in the table, using only type-correctness constraints led to an excessive number of solutions (i.e., function parameter combinations) for most of the benchmarks. The addition of finite type space constraints and uniform bit width constraints helped to significantly reduce the number of solutions for many benchmarks. Nonetheless, a few benchmarks still produced a large number of solutions even with these additional constraints.

For multi-function benchmarks bigCLAM and changepoint, the number of function versions is less than the product of solution count and function count. This is because some function versions are shared between solutions.

5.4 Evaluation of the Combination of Machine Learning and SMT Solving

Table 6. Number of parameter type combinations produced by adding machine learning predictions as constraints to SMT solving type inference and applying Algorithm 1. The numbers in **bold and green** indicate matches with the ground truth.

Benchmark	Freq				DeepTyper-FS				CodeT5-FT				GPT-4			
	Top-1	Top-5	Top-10	Top-20	Top-1	Top-5	Top-10	Top-20	Top-1	Top-5	Top-10	Top-20	Top-1	Top-5	Top-10	Top-20
bigCLAM	0	1	**2**	**2**	0	0	**2**	**2**	0	**2**	**2**	**2**	0	0	**3**	**3**
changepoint	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ipnsw	0	0	> 1000	**> 1000**	0	0	0	> 1000	0	0	0	> 1000	0	126	> 1000	**> 1000**
ISTA	0	0	**30**	**30**	0	0	0	**36**	0	0	**24**	**45**	0	**18**	**54**	**54**
PR-Nibble	0	0	**378**	**> 1000**	0	0	264	**> 1000**	0	0	0	**810**	**2**	**486**	**828**	**> 1000**
sinkhorn_wmd	0	0	**147**	**147**	0	0	**126**	**243**	0	75	**108**	**243**	0	**18**	**24**	**48**
cholesky	0	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
jacobi_2d	0	**8**	**8**	**8**	**4**	**8**	**8**	**8**	**4**	**8**	**8**	**8**	**4**	**8**	**8**	**8**
lu	0	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**
ludcmp	0	**1**	**8**	**18**	**1**	**2**	**4**	**18**	**1**	**4**	**12**	**27**	**1**	**12**	**27**	**27**

Table 6 displays the number of solutions acquired by intersecting solution sets obtained through machine learning and SMT solving, as described in Section 4.3.1. The results indicate that this method yields a smaller solution set compared to pure machine learning or SMT solving approaches, as evidenced by the comparison to numbers in Table 4 and Table 5. However, we have also observed its drawbacks discussed in Section 4.3.1. Notably, there were no valid solutions for changepoint, suggesting that the intersection may yield an empty solution set in certain cases. Moreover, for all four machine learning models, even though using the top-20 predictions yielded the desired parameter type combinations for more benchmarks than using the top-10, it also generated many more solutions for some other benchmarks. This suggests that there is no single optimal number of predictions to use for all benchmarks.

Table 7. Number of parameter type combinations and versions of functions produced by Algorithm 3. The numbers in **bold and green** indicate matches with the ground truth. "–" denotes timeout during compilation.

Benchmark	Freq		DeepTyper-FS		CodeT5-FT		GPT-4	
	#Combinations	#Functions	#Combinations	#Functions	#Combinations	#Functions	#Combinations	#Functions
bigCLAM	**3**	10	**3**	10	**3**	10	**3**	10
changepoint	**186**	310	**186**	310	**186**	310	**186**	310
ipnsw	131	263	-	-	-	-	12	28
ISTA	14	14	**10**	10	**10**	10	**10**	10
PR-Nibble	14	14	16	16	10	10	**24**	24
sinkhorn_wmd	11	11	11	11	11	11	**19**	19
cholesky	**2**	2	**2**	2	**2**	2	**2**	2
jacobi_2d	**8**	8	**8**	8	**8**	8	**8**	8
lu	**2**	2	**2**	2	**2**	2	**2**	2
ludcmp	**10**	10	**10**	10	**12**	12	**12**	12

Our improved algorithm, which incorporates progressive relaxation of machine learning constraints as described in Section 4.3.2, has successfully addressed these issues, as illustrated in Table 7. The machine learning models, when combined with SMT solving, have generated a limited number of parameter type combinations that cover most of the ground truth types. This implies that they have achieved high precision without inflating the output.

5.5 Performance Benchmarking with Compiler Integration

In order to showcase the effectiveness of our concrete type inference technique in enhancing the performance of untyped programs through ahead-of-time compilation, we incorporated the concrete types generated by our approach with a multi-versioning code generator based on the Intrepydd compiler [Zhou et al. 2020]. The code generator takes in code written in the supported subset of Python and emits type-annotated Python code and C++ code, which can be compiled by a Numba-AOT back-end, based on the ahead-of-time compilation module of Numba [Lam et al. 2015], and a C++ back-end, based on a C++ compiler, respectively. We assessed the performance gains achieved by the compiled versions relative to the original untyped versions. Note that the back-ends also generate unoptimized Python code as a fallback option in case the run-time types do not match with any of the provided types.

Table 8. Average kernel execution time (in seconds) and standard deviation of 6 runs (in parentheses) as a percentage of average execution time, compiled using different back-ends. The numbers in **bold and green** indicate that one of the inferred parameter type combinations matches with the ground truth.

Benchmark	Python	Numba-AOT Back-end		C++ Back-end	
		Ground Truth	GPT-4	Ground Truth	GPT-4
bigCLAM	4.487 (1.89%)	0.221 (0.70%)	**0.221 (0.72%)**	0.460 (1.35%)	**0.460 (1.95%)**
changepoint	18.270 (3.04%)	1.078 (0.22%)	**1.076 (0.31%)**	1.129 (0.12%)	**1.168 (0.25%)**
ipnsw	66.117 (3.35%)	0.634 (0.65%)	65.242 (0.30%)	1.784 (0.48%)	64.881 (1.17%)
ISTA	14.756 (0.23%)	42.751 (0.71%)	**42.452 (0.81%)**	11.945 (0.30%)	**11.768 (0.43%)**
PR-Nibble	15.128 (2.23%)	0.067 (0.33%)	**0.071 (0.24%)**	0.003 (1.03%)	**0.003 (0.52%)**
sinkhorn_wmd	29.079 (0.20%)	41.789 (0.45%)	**41.628 (0.32%)**	0.812 (0.41%)	**0.835 (1.40%)**
cholesky	84.296 (0.64%)	0.227 (0.22%)	**0.227 (0.16%)**	0.224 (0.22%)	**0.225 (0.25%)**
jacobi_2d	297.841 (0.40%)	0.817 (0.37%)	**0.816 (0.43%)**	0.322 (1.61%)	**0.324 (1.67%)**
lu	221.153 (0.97%)	0.696 (2.82%)	**0.689 (4.06%)**	0.676 (2.23%)	**0.631 (1.63%)**
ludcmp	142.048 (2.14%)	0.785 (1.44%)	**0.666 (3.01%)**	0.608 (1.74%)	**0.601 (2.69%)**

We present the execution times of the kernels (i.e., the part subject to AOT optimization) in the benchmarks compiled using the hybrid type inference technique discussed in Section 4.3.2 with the GPT-4 model, as shown in Table 8. Our type inference method successfully inferred the true types for nine out of the ten benchmarks, resulting in significant performance improvements after compilation using the two optimizing back-ends. However, it failed to identify the true types for the ipnsw benchmark, which includes a number of `Dict(int32,int32)` and `Array(int32,1)` types in its parameters. These two types can be assigned to function parameters in this benchmark interchangeably without breaking type correctness, leading to numerous valid type combinations. Moreover, the similar behavior exhibited by variables of these two types seemed to confuse the machine learning model, causing inaccurate predictions. As a result, our hybrid type inference method was unable to infer the intended types for this benchmark.

In Table 9, we demonstrate the performance improvements achieved by our hybrid type inference method using different machine learning models, in comparison to unoptimized Python code. It can be seen that the accuracy of the machine learning model significantly influences the ultimate code performance. A more accurate model led to a higher geometric mean of speedup across all benchmarks. Utilizing the most precise model, GPT-4, we attained a geometric mean speedup of 62.2× and an impressive $> 4,400\times$ speedup for PR-Nibble using the C++ back-end, whereas other models yielded no speedup for this benchmark due to their inability to deduce the intended types.

Table 9. Speedup of benchmarks achieved by compiling to Numba-AOT and C++ back-ends relative to the Python version. The numbers in **bold and green** indicate that one of the inferred parameter type combinations matches with the ground truth. The cells containing 1.0* indicate experiments that timed out during compilation. Although not implemented, the compiler can theoretically produce the Python version in such cases, leading to a speedup of 1.0.

Benchmarks	Freq		DeepTyper-FS		CodeT5-FT		GPT-4	
	Numba-AOT	C++	Numba-AOT	C++	Numba-AOT	C++	Numba-AOT	C++
bigCLAM	**20.4**	**9.8**	**20.2**	**9.7**	**20.2**	**9.8**	**20.3**	**9.7**
changepoint	**16.9**	**15.6**	**16.6**	**15.7**	**16.9**	**15.8**	**17.0**	**15.6**
ipnsw	1.0	1.0	1.0*	1.0*	1.0*	1.0*	1.0	1.0
ISTA	1.0	1.0	**0.3**	**1.2**	**0.3**	**1.2**	**0.3**	**1.3**
PR-Nibble	1.0	1.0	1.0	1.0	1.0	1.0	**212.4**	**4485.3**
sinkhorn_wmd	1.0	1.0	1.0	1.0	1.0	1.0	**0.7**	**34.8**
cholesky	**365.2**	**376.9**	**371.1**	**369.7**	**371.9**	**376.4**	**371.1**	**375.5**
jacobi_2d	**364.7**	**922.4**	**364.5**	**939.5**	**364.5**	**933.1**	**364.9**	**920.3**
lu	**298.2**	**345.4**	**299.9**	**335.3**	**319.1**	**345.8**	**321.2**	**350.6**
ludcmp	**210.8**	**233.6**	**215.3**	**234.3**	**212.6**	**235.3**	**213.3**	**236.5**
Geometric Mean	17.6	18.3	15.9	18.7	16	18.8	26.4	62.2

Furthermore, it becomes clear that type inference generally benefits both back-ends, as it enables their optimizations. Note that the slowdown for ISTA and sinkhorn_wmd when using Numba-AOT is attributed to the presence of sparse matrices in these benchmarks, which are not natively supported by Numba. Although we attempted to implement some support for these matrices by developing auxiliary Numba functions for use in the benchmarks, the resulting performance was still suboptimal.

6 RELATED WORK

Static type inference [Milner 1978] is a well-established technique that can enable compilers to determine types for a variety of program elements. However, due to the dynamic nature of programming languages like Python, static type inference often lacks precision. Various static type inference tools exist for Python, such as mypy,[4] Pyre,[5] PySonar2,[6] and pytype.[7] While these tools perform adequately in simple cases, they struggle with more intricate scenarios due to Python's dynamic characteristics, necessitating additional type annotations. To address some of the complex cases, Typpete [Hassan et al. 2018] employs an SMT solver to infer one consistent set of types for an entire program in a supported subset of Python. Different from Typpete, our SMT-based type inference operates on partial programs and generates multiple concrete type sets instead of a single type set potentially comprising abstract types.

In recent years, a large body of work has focused on utilizing deep learning for type inference. This includes studies that employ recurrent neural networks for predicting types from code [Hellendoorn et al. 2018; Malik et al. 2019; Mir et al. 2022], graph neural networks [Allamanis et al. 2020; Wei et al. 2020; Ye et al. 2021], and pre-trained transformers [Huang et al. 2023; Jesse et al. 2021; Wei et al. 2023]. Since machine learning models generate types without guaranteed correctness

[4] https://mypy-lang.org/
[5] https://pyre-check.org/
[6] https://github.com/yinwang0/pysonar2
[7] https://github.com/google/pytype

and can yield inconsistencies, several strategies have been proposed to integrate them with rule-based methods. These approaches encompass combining logical and natural constraints through continuous relaxation [Pandi et al. 2020], employing a type checker to validate predicted types [Pradel et al. 2020], and utilizing models to offer type suggestions during the rule-based static type inference process [Peng et al. 2022]. Nonetheless, no previous methods have combined machine learning with SMT solving, as presented in this paper.

Efforts have been made to compile programs written in Python and languages with similar syntax and semantics. Just-in-time compilers like Numba [Lam et al. 2015] and JAX [Bradbury et al. 2018] dynamically generate high-performance code from Python using type information collected during run-time. Intrepydd [Zhou et al. 2020] and Mojo[8] compile portions of type-annotated code ahead of time, which other parts of the program can invoke during run-time to enhance performance. Codon [Shajii et al. 2023] introduces a framework for creating statically-typed, Pythonic domain-specific languages, facilitating the writing of Python-like programs that can be compiled ahead of time. Unlike the past work, our method enables ahead-of-time compilation without requiring any type annotations or statically-typed code, and also without requiring a whole program.

7 CONCLUSION

In this paper, we introduced a new approach to concrete type inference and demonstrated its effectiveness in enabling code optimization for dynamically typed languages, without requiring the programmer to provide any type information. We explored three kinds of type inference algorithms in our approach based on: 1) machine learning models including GPT-4, 2) constraint-based inference based on SMT solving, and 3) a combination of 1) and 2). Our approach used the output from type inference to generate multi-version code for a bounded number of concrete type options, while also including a catch-all untyped version for the case when no match is found. Experimental results showed that the combined algorithm in 3) delivers far superior precision and performance than the separate algorithms for 1) and 2). The performance improvement due to type inference, in terms of geometric mean speedup across all benchmarks compared to standard Python, when using 3) is 26.4× with Numba as an AOT optimizing back-end and 62.2× with the Intrepydd optimizing compiler as a back-end. These vast performance improvements can have a significant impact on programmers' productivity, while also reducing their applications' use of compute and energy resources.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. 2021. PolyBench/Python: Benchmarking Python Environments with Polyhedral Optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/3446804.3446842

Ole Agesen. 1995. The Cartesian Product Algorithm. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, Mario Tokoro and Remo Pareschi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–26.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (jul 2018), 37 pages. https://doi.org/10.1145/3212695

[8] https://www.modular.com/mojo

Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3385412.3385997

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

Clark Barrett and Cesare Tinelli. 2018. *Satisfiability modulo theories*. Springer.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. http://github.com/google/jax

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. arXiv:1409.1259 [cs.CL]

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling Language Modeling with Pathways. arXiv:2204.02311 [cs.CL]

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 337–340.

Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. 2018. MaxSMT-Based Type Inference for Python 3. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 12–19.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

Qing Huang, Zhiqiang Yuan, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. Prompt-Tuned Code Language Model as a Neural Knowledge Base for Type Inference in Statically-Typed Partial Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 79, 13 pages. https://doi.org/10.1145/3551349.3556912

Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. https://doi.org/10.1145/3468264.3473135

Ben Johnson. 2018. graph-changepoint. https://github.com/bkj/graph-changepoint.

Ben Johnson. 2019a. IP-NSW. https://github.com/prog-eval/prog-eval/tree/master/ipnsw.

Ben Johnson. 2019b. lgc (local graph clustering). https://github.com/prog-eval/prog-eval/tree/master/lgc.

Ben Johnson. 2019c. Sinkhorn Word Movers Distance (sinkhorn_wmd). https://github.com/prog-eval/prog-eval/tree/master/sinkhorn_wmd.

Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG]

Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG]

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Amir M. Mir, Evaldas Latoskinas, and Georgios Gousios. 2021. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-based Type Inference. *CoRR* abs/2104.04706 (2021). arXiv:2104.04706 https://arxiv.org/abs/2104.04706

Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. https://doi.org/10.1145/3510003.3510124

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Irene Vlassi Pandi, Earl T. Barr, Andrew D. Gordon, and Charles Sutton. 2020. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. https://doi.org/10.48550/ARXIV.2004.00348

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 Yang, Zach 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. arXiv:1912.01703 [cs.LG]

Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. 2022. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2019–2030. https://doi.org/10.1145/3510003.3510038

John Plevyak and Andrew A. Chien. 1994. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications* (Portland, Oregon, USA) *(OOPSLA '94)*. Association for Computing Machinery, New York, NY, USA, 324–340. https://doi.org/10.1145/191080.191130

Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-Based Validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/3368089.3409715

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language Models are Unsupervised Multitask Learners. *OpenAI blog* 1, 8 (2019), 9.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic Model for Code with Decision Trees. *SIGPLAN Not.* 51, 10 (oct 2016), 731–747. https://doi.org/10.1145/3022671.2984041

Rob Romijnders. 2017. RobRomijnders/bigclam: Implements the bigCLAM algorithm. https://github.com/RobRomijnders/bigclam.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. 2023. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 191–202. https://doi.org/10.1145/3578360.3580275

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. arXiv:2303.09564 [cs.SE]

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Hkx6hANtwH

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

Fangke Ye, Jisheng Zhao, and Vivek Sarkar. 2021. Advanced Graph-Based Deep Learning for Probabilistic Type Inference. arXiv:2009.05949 [cs.PL]

Tong Zhou, Jun Shirako, Anirudh Jain, Sriseshan Srikanth, Thomas M. Conte, Richard Vuduc, and Vivek Sarkar. 2020. Intrepydd: Performance, Productivity, and Portability for Data Science Application Kernels. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 65–83. https://doi.org/10.1145/3426428.3426915

Received 2023-04-14; accepted 2023-08-27