The University of Tokyo
Graduate School of Information Science and Technology
Department of Creative Informatics

博士論文

A Toolkit for Efficient Type Inference and its Usage for Gradual Typing

(効率的な型推論のためのツールキットとその漸進的型付けへの応用)

Doctoral Dissertation of:
Senxi Li
李 森曦

Academic Advisor:
Shigeru Chiba
千葉　滋

Abstract

Despite the increasing demand for type inference in modern language development, existing language workbenches offer limited support for its practical implementation. This is because achieving computational efficiency in type inference remains a significant challenge even when leveraging powerful off-the-shelf solvers. A naively designed toolkit would generate less performant code for type inference compared with manually tuned implementations, which discourages the development and adoption of a generative toolkit. This dissertation presents the studies on supporting type inference in language development. It proposes a practical toolkit named InferType to address the absence of comprehensive tool support for type inference in existing language workbenches. InferType is a Java library for implementing constraint-based type inference. Compiler writers use InferType as an embedded DSL to specify typing rules and type constraints. InferType then performs constraint solving for type inference by translation to the Z3 SMT solver. To alleviate the performance bottleneck of constraint solving in type inference, InferType is equipped with a novel optimization technique. It reduces the search space by pre-computing a structure of each type variable. The dissertation then demonstrates the usage of InferType within the context of gradual typing. To complement the efficient generation of inferred type annotations, a novel method is designed for mitigating the execution overhead of runtime casts caused by excessive type annotations. Our method selectively adds additional type annotations derived from type inference for preventing values from repeatedly crossing the boundaries between untyped and typed code. The selection of type annotations is quickly performed in a lightweight, amortized manner based on the data flows. Experimental results indicate that InferType with the proposed optimization techniques can help address the performance bottlenecks associated with type inference in modern language development. The results reveal that InferType can substantially mitigate the performance bottleneck for programs with deeply nested types, and can potentially improve the performance for large

nested types. A tool named TypePycker is implemented by using InferType for improving the execution speed of programs within a sound gradually typed language Reticulated Python. Compared with existing type inference engines, InferType can produce efficient type inference and TypePycker can maintain more stable time for selecting type annotations. TypePycker also achieves performance improvement compared with naively using all type annotations inferred by type inference.

Acknowledgements

I consider myself incredibly fortunate to have received unwavering support in my doctoral studies. I am grateful to numerous individuals who have walked alongside me during this journey.

First and foremost, I would like to express my sincere and heartfelt gratitude to my supervisor, Prof. Shigeru Chiba, for his invaluable instruction, guidance, and encouragement throughout my doctoral journey. His mentorship has shaped not only my research but also the way of thinking. His critiques on my logical reasoning would always remind me to carefully examine ideas and reconsider everything from a starting point. I fondly recall the discussions when we held paper and pens to sketch ideas during some nightly walk from the laboratory to the subway station. Things I have learned from him would be a profound source of inspiration for the rest of my life.

I would also like to extend the appreciation to the committee members of my defense: Prof. Hiroshi Esaki, Prof. Naoki Kobayashi, Prof. Nobuyuki Umetani, Prof. Tomoharu Ugawa, and Prof. Ryota Shioya. They gave me expert critiques and insightful feedback from their respective specialities, which helped refine the structure and elevate the overall quality of this dissertation.

I would like to express my appreciation to my colleagues in our laboratory, with whom I have had the privilege of collaborating. I am particularly grateful to the co-authors of my papers, Dr. Tetsuro Yamazaki and Dr. Feng Dai. Dr. Yamazaki shared his professional insights regarding formal foundations and implementation techniques for programming languages. I would never forget his passion during our discussions even when they continued until midnight. I carry memories of the discussions with Dr. Dai on how to derive comprehensive evaluations from raw experimental data. Those discussions were often held during our walk to the convenience store for drinks and fried chicken, which were as enlightening as they were enjoyable. I would also like to thank Prof. Soramichi Akiyama and Prof. Tomoki Nakamaru. Despite the limited time we shared in the laboratory, I am truly

grateful for your valuable suggestions regarding career decisions.

I would also like to extend my thanks to all other members in our laboratory, including Shouya Ikezaki, Yilin Zhang, Fumika Mochizuki, Yuefeng Hu, Yuze Fu, and Hanhaotian Liu. It has been a pleasure getting to know each of you. I have greatly valued the meaningful conversations and the lasting memories we made together.

Finally, I would like to express my deepest gratitude to my family for their unconditional love. My father reminded me to stay calm and resilient when making mistakes. My mother patiently listened to me during my most difficult moments and helped me reconstruct the attitude to face challenges in life. My brother supported me by generously purchasing a new laptop for me using his first-year salary. My achievement would not have been possible without my family.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Assisting Development of Programming Languages

Programming languages serve as the fundamental instruments through which computer scientists and software engineers conceptualize, construct, and maintain the modern digital world. By providing powerful abstractions and rigorous guarantees, programming languages enable programmers to improve their productivity and ensure the reliability of complex software systems. The design and evolution of programming languages have a long history, characterized by extensive research dedicated to refining the discipline of software engineering. The development of programming languages has become a cornerstone of modern software engineering in response to the accelerating demand for rapid software evolution requested by both industry and academia.

There is a persistent and growing demand for developing programming languages. Today, the landscape of programming languages is more diverse than ever before. This diversity exists because no single, universal language can satisfy the various disciplines of modern computing. Each programming language satisfies a unique set of trade-offs among disciplines ranging from low-level efficiency to high-level expressiveness. For instance, a programmer would choose C for its low-level efficiency and portability in system programming, while one would prefer scripting languages such as Perl and Python due to their flexibility during rapid prototyping. There are languages known as domain-specific languages (DSLs). Compared with general-purpose languages, a DSL is specifically tailored and it can offer better programming experience for one particular application domain. Examples include SQL

for relational database management, the DOT language in Graphviz [29] for graph drawing, Halide [85] for image processing, and Exo [44] for hardware accelerators. Accordingly, the pursuit of language innovation remains a perennial and increasingly vital endeavor within computer science.

One long-standing and also challenging task is how we can assist language designers to develop and implement new languages. One common approach is to reduce the cost of the implementation by code reuse. Code reuse is typically achieved by adopting a series of program pieces organized for cross-project utility, which is called a library or a supporting toolkit. The implementation of a practical programming language requires a sophisticated pipeline of interconnected components, including lexers, parsers, type checkers, intermediate representation generators, optimizers, and code generators. Manually implementing every component from scratch is not only labor-intensive but also increasingly impractical. As is standard in modern software engineering, the adoption of reusable toolkits is a preferred choice in developing languages. Language designers can then concentrate their effort on defining unique semantics and high-level abstractions of their specific languages instead of repeatedly reinventing the wheel.

Fortunately, several well-developed toolkits already exist to support the development and implementation of programming languages. Historically, the most successful toolkits are parser generators. These toolkits allow language designers or compiler writers to specify high-level, declarative specifications of grammar rules instead of implementing complex, detailed parsing algorithms by hand. The toolkits will then automatically generate runnable code that can parse input source code into abstract syntax trees (ASTs) from the specifications written by the compiler writer. Examples are yacc [58] and bison [97].

Beyond traditional generators, language workbenches have emerged to further enhance the productivity of implementing languages. A *language workbench* is an integrated development environment for implementing domain-specific languages [32]. It is designed to facilitate the language implementation by unifying various stages of the development cycle. Compiler writers can write high-level specifications of their languages to automate a number of tasks in the development. For example, language workbenches can assist in generating code for parsers, namespace resolution, type checkers, and interpreters. Some of the widely noted language workbenches are JetBrains MPS [106] and Spoofax [49].

1.2 Missing Support for Type Inference

Our research studies aim to contribute to the assistance of *type inference* in implementing programming languages. Type inference is the process of automatically reconstructing types for programs without explicit type annotations. For example,

```
x = 42;
if (y)
    z = x;
```

although the type annotations for the variables are not given, a compiler equipped with type inference can be reasonably able to statically infer the types of variable `x`, `z` as type integer and variable `y` as type boolean based on the context of this program. Type inference can benefit programmers by alleviating the burden of writing a massive amount of type annotations in their source code. Type inference is also instrumental in enabling compiler optimization. For instance, compilers can generate highly specialized code for execution instead of slow execution paths with massive runtime type checking since the static types are already inferred and known by type inference.

The adoption of type inference has become widespread across modern programming languages. There are a number of practical languages that adopt type inference such as Scala, Haskell in functional languages, and C#, Java, TypeScript in object-oriented languages. There are also evolving tools that enable type inference for dynamically typed languages such as Flow [17] in JavaScript, Sorbet [98], InferDL [50] in Ruby, and pytype [35], Pyright [65], Pyrefly [64] in Python.

Conventionally, language designers manually implement type inference when developing their own languages. However, implementing type inference by hand involves a great amount of engineering effort, which is labor-intensive and error-prone. One common approach is via a straightforward implementation by AST traversal. In this approach, the type inference mechanism is often highly integrated with other compiler components such as name resolution and type checking. Although the implemented engine is straightforward and can be efficient with eager computation, its monolithic integration makes the engine highly language-dependent and thus with low extensibility. A classic example is Algorithm W [66], which is an efficient implementation of the Hindley-Milner type inference [42].

Another well-established approach is the constraint-based approach [80]. In this approach, type inference is performed in a two-pass manner. In

the first pass, it traverses the AST to collect type constraints that specify the conditions types must satisfy for correct typing. In the second pass, it invokes a solver to solve these collected type constraints. Depending on the type system, language designers may implement the solver using unification [91], graph-based algorithms, or fixed-point iterations. Due to this separation, the type inference engine is of high extensibility and can be portable to different languages. However, sophisticated solvers are required and they may incur slow inference time without careful consideration.

While compiler components like parsers have long benefited from mature supporting toolkits, comparable support for implementing type inference engines remains notably limited in developing languages. One primary reason for the limited tool support is that achieving computational efficiency in generated code for type inference is difficult. Generated code is often less performant than manually tuned implementations in software applications including that for type inference. It is challenging due to the inherent complexity of the underlying algorithms for type inference. Without domain-specific optimizations or specialized tuning techniques, an engine may perform slow inference since it must find a valid type for every variable to satisfy all the invariants specified in a program.

For instance, a standard algorithm for type inference in the presence of subtyping based on closure computation would exhibit cubic time complexity related to the sizes of types involved in a program [82]. There are also existing approaches [102, 77, 39, 109] that translate type inference into a constraint satisfaction or a search problem using modern powerful solvers such as Satisfiability Modulo Theories (SMT) [8, 7]. However, a straightforward translation may lead to impractical efficiency since a solver must search for a solution within large search space of all possible types. As a result, if a toolkit is naively designed following a standard algorithm, or implemented as a search engine in a straightforward manner, the generated code may easily incur a performance penalty for larger input programs. The performance pitfall has discouraged the development of automatic, generative toolkits for type inference. A compiler writer would be reluctant to adopt a generative toolkit since an easy solution for the compiler writer is just to implement the type inference engine by hand considering practical usage.

Achieving efficiency in type inference is critical for enabling new algorithms and techniques in modern language design. Specifically, a type inference engine must be performant enough to leave sufficient computational room for subsequent downstream optimizations within a compilation pipeline. For instance, leveraging static type inference for improving the exe-

cution speed in gradual typing [95, 94] has become a well-accepted approach. Gradual typing has emerged as a compelling paradigm for bridging the gap between static and dynamic typing within one single language. However, its flexibility often introduces a significant challenge of unexpected performance degradation for partially typed programs. Several research studies have explored approaches to improve the execution speed based on the idea of deriving more static type annotations by type inference [87, 12, 104, 51, 13]. Despite the diversity of the proposed techniques, their research shares the same dependency of a type inference engine for deriving additional type annotations. The type inference process is integrated with their proposed techniques for further improving the execution speed of gradually typed programs. If the type inference engine performs slow inference that consumes too much time budget in a compilation pipeline, the researcher might not be able to argue her proposed technique as practical due to excessively long compilation.

Another example of the need for efficiency in type inference is automatic migration from dynamic codebases to high-performance languages. In this scenario, developers maintain productivity and rapid prototyping by writing code in a dynamic, interpreted language, while an underlying pipeline automatically translates the code to a high-performance, compiled language for optimized execution [54]. Type inference is needed in that pipeline to derive additional type annotations required for the automatic translation and further compilation, as a number of existing programs in dynamically typed languages do not contain optional type annotations such as those in Python [86, 25]. For instance, one study [109] uses type inference for translating kernel functions of Python programs in data science [63] and scientific computing [103, 111] to C++ code. The generated C++ code is then compiled to highly performant code and further loaded as a module for use in Python applications. If the type inference engine is built by using a naive, generative toolkit without considering performance bottlenecks, each translation would lead to non-negligible time for codebases with larger and more complex kernel functions. Such slow translation would degrade the efficiency in the rapid iteration cycles of development.

Existing language workbenches and a few research studies have explored approaches for supporting type inference in a generative way. However, they are limited in supporting global type inference in the presence of subtyping. To our best knowledge, as one state-of-the-art existing language workbench, Spoofax [49] supports type inference only for unification-based systems. Namely, Spoofax does not support type inference for languages with type relations in partial orders such as subtyping or for languages

supporting ad-hoc polymorphism such as operator overloading. Its type inference does not prioritize the performance of the generated code, which might incur non-negligible overhead for larger programs. Another notable toolkit in the literature is Turnstile [16], which is a meta-language in Racket for implementing languages with bidirectional type checking. While Turnstile is expressive enough to support a wide range of languages with typing features including subtyping, overloading, and generics, Turnstile is fundamentally constrained to bidirectional type checking of local type inference. It cannot support global type inference required for untyped programs that must reconstruct types across broader program contexts such as from the call sites of a function. Consequently, it remains challenging to create a generative toolkit for supporting efficient type inference in practical language development.

1.3 Contributions

This dissertation proposes a practical toolkit for implementing constraint-based type inference. By our developed techniques incorporated with the proposed toolkit, we address some of the critical performance penalties that hinder the support and adoption of generative toolkits for type inference in modern language development. We demonstrate that our toolkit is suited to generate comparably efficient code compared with existing, manually implemented engines within a research study. We empirically show evidence that the efficient code generated by our toolkit is capable of enabling other techniques for further downstream optimizations.

We propose *InferType*, a pragmatic compiler toolkit for implementing constraint-based type inference. InferType is designed to prioritize efficient inference, though its expressiveness is restricted in its range of supported type systems. Instead of serving as an enabler for formalizing fundamental verifications or building proof assistants, it stays practical and well-suited for implementing type inference engines that provide type hints in a conservative way for enabling other downstream techniques in a broader compilation pipeline. The contributions are listed below:

- We design InferType as a Java library. It allows compiler writers to specify typing rules and type constraints as an embedded domain-specific language in Java. InferType then automatically performs constraint solving by translation to SMT.

- We develop a novel optimization technique that can help generate code

for efficient type inference. The technique reduces the search space in constraint solving by pre-computing a structure of each type variable. Thereby, the constraint solving for type inference is expected to run faster compared with a straightforward translation to SMT. As a preprocess, we develop it in an approximate and efficient manner to ensure that it does not consume too much analysis time.

To demonstrate the practical utility of InferType, we conduct a research study in the context of gradual typing. We provide empirical evidence to show that InferType is capable of generating sufficiently efficient code for type inference to enable other techniques in research studies. The technique we develop is a novel lightweight method for mitigating the execution degradation of gradually typed programs. It selectively adds additional type annotations derived by type inference. We implement our method as a practical tool named *TypePycker*, which leverages InferType for performing its type inference. The contributions in this study are:

- We demonstrate the integration of InferType with existing language systems by extending a sound gradually typed language Reticulated Python with automatic type inference. Our empirical results show that InferType can produce efficient type inference compared with existing type inference engines, which better enables our proposed technique for optimizing the execution speed of gradually typed programs.

- We propose a novel technique for improving the execution speed of gradually typed programs. Our technique identifies and annotates program fragments that are expected to prevent values from repeatedly crossing the boundaries between untyped and typed code. It selects type annotations in a lightweight, amortized manner based on data-flow reachability. Our method allows an efficient implementation such as TypePycker enabled by InferType since it only needs the inferred results of an external type inference engine instead of relying on the internal process of type inference.

1.4 Organization

The remainder of this dissertation is organized as follows. Specifically, the primary contributions of the dissertation are established in our academic publications [59, 60]. One publication [59] focuses on the design of a practical toolkit for implementing efficient constraint-based type inference. The other

publication [60] covers the use of the toolkit InferType for optimizing the execution performance in gradual typing.

Chapter 2 presents the background of supporting type inference in language development. Particularly, we review the current state and common supporting features in language workbenches, and also existing methodologies for implementing type inference. Our review serves to delineate the scope of this dissertation and identify the specific technical limitations that our studies aim to address.

Chapter 3 presents our practical toolkit InferType for implementing constraint-based type inference. As we discussed above, existing language workbenches do not contain such a supporting toolkit. Our toolkit addresses this gap and serves as a candidate tool support for implementing type inference in practical language development. The corresponding academic paper [59] of this chapter is published at the 38th European Conference on Object-Oriented Programming (ECOOP) 2024.

Chapter 4 demonstrates the usage of InferType in a study of mitigating the unexpected execution degradation of gradually typed programs. We show evidence to support our claim that as a generative toolkit, InferType can generate code efficient enough to enable other techniques for subsequent downstream optimizations. The corresponding academic paper [60] of this chapter is published at Volume 11, Issue 1 of The Art, Science, and Engineering of Programming. That paper also received Best Paper Award.

Chapter 5 briefly concludes this dissertation. It highlights our proposal, the key scientific insights, and our empirical findings from the experiments. We additionally discuss possible directions of future research.

The studies presented in the dissertation are accompanied by publicly available artifacts. The artifacts include the used materials of source code and benchmark programs mentioned in the following chapters. They also allow the reproduction of the experiments. The artifacts are openly available on Zenodo. The detailed references for data availability are given in the corresponding chapters.

Chapter 2

Background

Implementing programming languages without automated tool support is impractical. It is crucial to adopt powerful, tailored toolkits, and also involved techniques to meet the demand for rapid evolution in modern language development. In this chapter, we introduce the essential concepts and technical foundations necessary to position our studies within the broader field of programming languages.

2.1 Language Workbench

A language workbench is a development environment integrated with tailored toolkits for the definition, reuse, and composition of programming languages. It is a term coined since 2005 [32]. A language workbench assists language designers or compiler writers to implement their languages by providing reusable code fragments. It can generate code from the high-level specifications by the compiler writers. With language workbenches, compiler writers are expected to implement their own languages with comparably less engineering effort compared to implementing languages by hand.

One classic kind of supporting tools in a language workbench is to generate parsers by parser generators. A parser generator is a program that takes a description for the grammar of a language and automatically generates a parser. A parser is a program that takes a sequence of tokens as input and builds a structured representation such as abstract syntax trees (ASTs) of that input. For example, for a simple language that describes numbers and booleans, a parser generator takes a description of its grammar as input, which is usually a formal notation called Backus-Naur Form (BNF) as follows:

Listing 2.1: Syntax of a simple language.

```
exp ::= let x: type = exp in exp
      | x
      | if exp then exp else exp
      | succ exp
      | 0
      | iszero exp
      | true
      | false

type ::= int
       | bool
```

For instance, a parser generator yacc [58] would accept a grammar description similar to the above and generate source code in C that is able to perform parsing. Its generation is based on the LALR(1) algorithm [24], which is a bottom-up parsing algorithm by deterministic pushdown automaton. In other words, it eases the implementation of parsers in a generative way by allowing compiler writers to declare a specification of the grammar syntax such as Listing 2.1 instead of manually implementing a parser algorithm. Other widely-used parser generators include bison [97], Jison [14], and ANTLR [75].

Another well-established support in language workbenches is for generating type checkers. A type checker is a software program that computes a relation among a typing environment, a code fragment, and a type by a type system. A type system, by definition, is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [80]. A type checker can confirm whether or not a given program is consistent with its usage of data and reject programs with wrong usage before execution. A type checker is usually part of a compiler and it is automatically run during compilation.

For instance, the Spoofax language workbench [49] can accept declarations similar to Listing 2.2 as input to generate code of performing type checking for the language specified in Listing 2.1.

Listing 2.2: Pseudocode for implementing a type checker in a language workbench.

```
typeOfExp(env, Let(x, e1, e2)) = T2 :-
        typeOfExp(env, e1) = T1,
        new letEnv -> env,
```

```
        declareVar(letEnv, x, T1),
        typeOfExp(env, e2) = T2.

typeOfExp(env, Var(x)) = typeOfVar(env, x).

typeOfExp(env, If(e1, e2, e3)) = T2 :-
        typeOfExp(env, e1) = Bool(),
        typeOfExp(env, e2) = T2,
        typeOfExp(env, e3) = T3,
        T2 = T3.

typeOfExp(env, Succ(e)) = Int() :-
        typeOfExp(env, e) = Int().

typeOfExp(env, Zero()) = Int().

typeOfExp(env, IsZero(e)) = Bool() :-
        typeOfExp(env, e) = Int().

typeOfExp(env, True()) = Bool().

typeOfExp(env, False()) = Bool().
```

Each declaration specifies a typing rule for type-checking each kind of expression or AST node in Listing 2.1. The language workbench generates detailed code that performs type-checking for those declarations. For instance, generated code that type-checks if expressions can be in correspondence to the code presented in Listing 2.3.

Listing 2.3: Code for type-checking.

```
Type typeOfExpIf(Env env, AstNodeIf node) {
    Type t1 = typeOfExp(env, node.ifSubExp1());
    Type t2 = typeOfExp(env, node.ifSubExp2());
    Type t3 = typeOfExp(env, node.ifSubExp3());
    equalType(t1, TypeBool());
    equalType(t2, t3);
    return t2;
}
```

Language workbenches ease the implementation of type checkers in a generative way by allowing compiler writers to instead of manually defining

and implementing traversal methods. It also automatically manages typing environments and scopes for name resolution.

Language workbenches have been widely adopted for supporting language creation but also in providing better editing experience for language users [32, 30, 49]. A key benefit is the integration of a defined language into an Integrated Development Environment (IDE). It can automatically provide editor services such as syntax highlighting, error marking, performing semantic analysis in the background, code navigation of jumping to definition for variable declarations, and code completion. Having IDE support is crucial for the widespread adoption of a programming language in practice [107, 5].

2.2 Type Inference

Type inference is the process of automatically reconstructing types for programs without explicit type annotations. It is considered as a more powerful algorithm that can perform type checking. While algorithms for type checking are all dependent on explicit type annotations such as annotated variable and parameter types for let bindings and function definitions, type inference allows that some or all of the type annotations are left unspecified in a program [80]. Those unspecified type annotations for variables or parameters are reconstructed by type inference.

The most prevalent approach is constraint-based type inference. It performs type inference in a two-pass manner of translating the input programs into a set of type constraints and solving the type constraints. The core idea is, instead of immediately checking what type an expression must have during AST traversal, it simply records type constraints based on the way of how that expression is used in the program for later consideration. It introduces a type variable when the type of an expression is not known yet. After recording all the type constraints, a solver is invoked to solve the recorded type constraints for finding types for the introduced type variables. If the type constraints can be resolved, then the program can be type-checked and the results from the resolved type constraints are the inferred types. Otherwise, the program is ill-typed and corresponding type errors are reported.

To perform type inference for the language in Listing 2.1 using a constraint-based approach, it first traverses the AST to collect type constraints as given in Listing 2.4.

Listing 2.4: Code for collecting type constraints by AST traversal.

```
Type typeOfExpIf(Env env, AstNodeLet node) {
```

```
    String x = node.identifier();
    TypeVar tyVar = freshTypeVar();
    Type ty1 = typeOfExp(letEnv, node.subExp1());
    Env letEnv = bind(env, x, tyVar);
    Type ty2 = typeOfExp(letEnv, node.subExp2());
    return ty2;
}

Type typeOfExpIf(Env env, AstNodeIf node) {
    Type ty1 = typeOfExp(env, node.ifSubExp1());
    Type ty2 = typeOfExp(env, node.ifSubExp2());
    Type ty3 = typeOfExp(env, node.ifSubExp3());
    constrainEqualType(ty1, TypeBool());
    constrainEqualType(ty2, ty3);
    return ty2;
}
```

Type annotations for let expressions are not given in the source code and a fresh type variable is generated for representing the type of the declared variable in a let expression. Compared to a type checker given in Listing 2.3, it does not check the type of the sub expressions of the AST node immediately after it retrieves the type. Instead, the constraints are collected by a method `constrainEqualType` which specifies that the two argument types must be lexically identical.

After collecting the type constraints by AST traversal, a solver is invoked to solve the type constraints for type inference. To solve the collected type constraints in equality for the language in Listing 2.1, a solver based on an algorithm called unification is invoked, which is a fundamental algorithm to a number of areas in computer science [91, 52]. It finds a substitution of types that makes all the type constraints are satisfied. A substitution is a finite binding from type variables to types. A type constraint is satisfied by a substitution if the two types involved in type equality can be lexically identical after the substitution is applied to the two types.

The constraint-based type inference can scale to support diverse type systems by accepting and solving different kinds of type constraints [108]. Suppose that we implement a type inference engine that infers types for a language supporting subtyping. Collecting type constraints by AST traversal is implemented in a similar way to that presented in Listing 2.4. The difference is to collect type constraints is replaced with a method for recording type constraints in subtype relations. After that, a solver for resolving

those type constraints in subtyping will be called for performing type inference. Following a standard algorithm [82], that solver typically treats type constraints in subtype relations as edges in a directed graph, where nodes represent types, type variables and edges represent type constraints in subtype relations. It then propagates lower and upper bounds for each type variable by computing the transitive closure of the graph. The accumulated lower and upper bounds are further used to compute an inferred type of each type variable.

Another practical approach to implementing solvers of type constraints for performing type inference is to leverage general-purpose solvers for logical reasoning. The idea is to model relations between types such as type equality and subtyping into logical formulae and let an external solver resolve the logical formulae. The resolved logical formulae are then translated back into the language type domain as the result of type inference. A major benefit of utilizing an external solver in logical reasoning is the reduction in engineering complexity. Compiler writers can avoid detailed implementation of constraint solving in constraint-based type inference. Compiler writers can also leverage theories in logical reasoning for modelling complex typing features such as using logical disjunctions to model operator overloading and using logical predicates to model refinement types [102]. Automated reasoning solvers that have either been deployed or shown potential for the usage of type inference. Examples include SAT solvers (such as MiniSat [27]), SMT solvers (such as Z3 [22] and cvc5 [6]), declarative logic programming languages (such as Prolog and Datalog), and Interactive Theorem Provers (such as Coq).

One widely-accepted class of logical solvers for performing type inference is Satisfiability Modulo Theories (SMT). An SMT solver is a software tool for determining whether a given logical formula is satisfiable with respect to background theories [8]. Since SMT solvers are designed as low-level tools, they are able to be flexibly adapted to a diversity of verification and constraint-satisfaction tasks. Formally speaking, an SMT solver checks whether a formula F is satisfiable by finding if there exists an interpretation that assigns values to variables such that F evaluates to true. In the context of type inference, type constraints are mapped into logical formulae F in SMT. An SMT solver is then used to perform constraint solving as finding an interpretation to make F satisfiable. The found interpretation for F is the result of the inferred types, which is a concrete type assignment that can make the program be type-checked.

Straightforwardly delegating constraint solving to an SMT solver for type inference is not practical considering performance. Performance bottlenecks

would occur since a type inference engine based on SMT essentially behaves as a search engine and it attempts to discover a valid binding of types within an underlying search space. For instance, suppose that type inference is performed using an SMT solver for a language supporting array types. The SMT solver must search for a valid binding for all type variables that satisfy all type constraints. Every type variable ranges over infinite number of possible types from `int, array<array<int>>, bool, array<int>, array<array<array<bool>>>, ...`. SMT solvers are designed to be efficient for handling infinite search domains in several contexts such as arithmetic and bit-vectors since they utilize builtin, specialized theories. However, they are not specifically designed for handling user-defined domains such as types and type contraints in type inference. As a result, without careful consideration or specialized optimization techniques, the implemented type inference engine directly using SMT sovlers may not be practical. It may perform unacceptably slow inference for larger or complex input programs.

2.3 Leveraging Type Inference in Gradual Typing

One of the common usage of type inference is to enable other techniques for subsequent downstream optimizations. Besides as fundamental verification of type checking, the inferred types by type inference often serve as additional type hints that are further used for other optimizations and code generation. In this section, we introduce basic concepts of gradual typing and describe a scenario of how type inference is leveraged in the context of gradual typing.

Gradual typing describes a type system that allows parts of a program to be statically typed and the other parts to be dynamically typed [95]. It is a typing paradigm where programmers can use dynamic and static typing within one single language. A programmer takes control of which parts to be which typed by either specifying type annotations or not in a gradually typed program.

Sound gradual typing is the idea that typed code is able to execute as if there were no untyped code in gradual typing. In a sound gradually typed language, typed code can rely on the type soundness and invariants specified by the static type annotations so that a compiler is capable of enabling type-driven optimizations. In gradual typing, untyped values can flow into typed code and vice versa. If a value with an unexpected type flows into a typed code fragment, it will violate type invariants by user type annotations. If

gradual typing is implemented by naively erasing the type annotations at runtime, it cannot prevent such violations and the static type annotations cannot be trusted. To achieve sound gradual typing, a gradually typed language must protect the guarantees obtained through static type checking by inserting runtime casts at the locations in the code where values flow from untyped code regions to typed code regions. That runtime cast is to check whether a value has an expected type at runtime. If a value from untyped code fails to have an expected type at runtime, an exception is thrown. Thus, statically type-checked code of the program is guaranteed to be passed with only well-typed values. The execution speed of partially typed programs may incur degradation due to the inserted runtime casts for sound gradual typing.

Type inference is a widely-accepted technique that is used to improve the execution speed of partially typed programs in the context of gradual typing. The intuition is to use type inference to reconstruct static type annotations for dynamically typed code fragments so that more code can be optimized driven by static types under compilation, which is expected to yield faster execution. Ideally, if a partially typed program can be fully enhanced with static type annotations by type inference, the program can reach peak performance since it would be equivalent to static typing. If a program is still partially typed after type inference, careful attention must be paid when adding additional type annotations by type inference because adding more type annotations does not necessarily yield fewer runtime casts. The compiled code may have slower execution due to massive or heavy runtime casts by added type annotations even if more code can be optimized by static types. Under this scenario, that compilation pipeline would involve a sequence of components including type inference and also several other downstream techniques such as for handling partially typed code fragments and inserting runtime casts. The compilation would be practical with respect to performance if every involved component is sufficiently fast enough to not consume too much time.

2.4 Our Motivation

Our goal is to provide a generative toolkit that automates the implementation to a certain degree and also delivers efficient code of type inference in language development. We aim to develop such a toolkit that can be one candidate component of language workbenches. The generation of efficient code for type inference has not been well-studied as far as we know and a

supporting toolkit targeting global type inference with subtyping for practical usage is not present yet. If a supporting toolkit is available, a compiler writer can use it to realize a type inference engine without manually implementing detailed algorithms. She can further leverage the implemented engine to enable other downstream optimizations since the generated code for type inference by that toolkit is sufficiently fast that will not disturb other techniques within a compilation pipeline. Without appropriate tool support, the implementation of type inference engines remains either labor-intensive by manual engineering or impractically inefficient due to naive automation.

In this chapter, we reviewed language workbenches and their common supporting features for those including parser generators and generation for type checkers. We also reviewed foundational concepts of type inference, and conventional methodologies of how compiler writers would implement type inference engines manually. While common compiler components like parsers and type checkers have benefited from mature, automated tool support, comparable support for type inference remains limited even though its manual implementation is highly laborious and prone to engineering defects. We additionally introduced gradual typing, which is a typing paradigm that combines static and dynamic typing. Type inference is often leveraged to add more type information for improving the execution speed of gradually typed programs.

Note that we do not aim at automating every computational step of type inference in a supporting toolkit. We aim at automating those steps that impose heavy burden on human engineering and consume more computational cost in performing type inference. Code for AST traversal is relatively straightforward to implement within a compiler. We believe that manually assigning types and type variables to AST nodes by AST traversal is a reasonably good implementation to maintain better control over various syntactic structures and also internal data representations in a target language. On the other hand, solving type constraints is often much more labor-intensive and demands a higher-level abstraction. The most complex and also time-consuming part within type inference is constraint solving compared to AST traversal. More attention should be paid to automating the process of constraint solving.

To bridge the gap of the limited support, we claim that techniques are needed for generating performant code of type inference considering practical usage. Generating code for performing type inference is relatively straightforward since well-known algorithms and off-the-shelf solvers are present. Constructing a type inference engine by some automation is fairly feasible

although the range of the supported type systems would be restricted based on the adapted algorithm. However, generating workable and also efficient code remains a significant challenge. The performance of a type inference engine is crucial in its practical adoption since its efficiency conditions the viability of downstream compilation techniques. If a type inference engine generated by a toolkit performs slow inference, the slow inference would consume an unacceptable amount of time within a compilation pipeline. Then subsequent compiler components are disturbed or forced to operate within limited time. As a result, a generative toolkit without tailored techniques is not practical since the compiler writer would be reluctant to use its generated code as a component of the whole compiler considering performance.

Chapter 3

InferType: A Compiler Toolkit for Implementing Efficient Constraint-Based Type Inference

Supporting automatic type inference is in demand in modern language development. It is a challenging task but without appropriate supporting toolkits. In this chapter, we tackle this problem by proposing InferType, a Java library that helps implement constraint-based type inference. We introduce its programming interfaces, and describe how InferType performs type inference by translation to the Z3 SMT solver. We then present our developed optimization technique in InferType for mitigating the performance bottleneck of constraint solving with deeply nested types. It is a pre-process that reduces the search space for type variables by pre-computing the structures of those type variables. After that, we evaluate the effectiveness of InferType's optimization by implementing a type inference engine for a Python subset.

The remainder of this chapter is organized as follows. Section 3.2 motivates the reader with a scenario of developing a type inference engine for a small demonstrative language. Section 3.3 presents the proposed toolkit InferType from its programming interfaces to the underlying optimization techniques for improving the performance of type inference. Section 3.4 gives our experiment results by an implemented type inference engine for a Python subset using InferType. We compare the time of type inference to validate whether the proposed optimization effectively mitigates the perfor-

mance bottleneck of constraint solving. Section 3.5 discusses the limitations of InferType. Section 3.6 positions our work within prior research of existing supporting tools in language workbenches and techniques used in type inference. Finally, a concise summary of our contributions ends this chapter.

3.1 Introduction

The goal of this work is to provide an assisting tool for implementing compilers supporting type inference, which is an in-demand task in modern language development but not well supported by existing approaches. Several supporting toolkits for language development have been proposed such as lexer and parser generators [58, 97], and type checker generators by language workbenches [106, 49]. Since a number of modern languages support type inference that automatically reconstruct types for programs without explicit type annotations, compiler writers have to implement type inference when developing their own languages. Unfortunately, there are no applicable toolkits for supporting this labor-intensive and also challenging task.

Major static languages support type inference that can be performed in a simple bottom-up manner. It does not need a complex inference engine. However, in recent languages like TypeScript, more aggressive type inference is supported. For example, they may allow programmers to omit a type annotation for a function's return type. A program in such languages may require complicated type inference when it includes mutually recursive functions. Its compiler may contain an intricate type inference engine that solves type constraints retrieved from a program.

One of the approaches to implementing such intricate type inference is utilizing modern Satisfiability Modulo Theories (SMT) solvers. Several static analyses including automatic type inference have been established based on SMT solving [8, 99, 102, 39]. However, it is not simple to implement an efficient type inference engine for practical usage even when using a powerful SMT solver. For example, we observe that the performance of type inference using a SMT solver may drastically degrade when the source program contains nested types such as function types that take other function types as arguments and list types that take other list types as arguments. Such types are frequently used in real-world programs for describing higher-order functions and data structures in hierarchy.

We propose InferType, a domain-specific language embedded in Java for implementing constraint-based type inference. A compiler writer imple-

ments a constraint-based type inference engine for a target language by describing type constraints and encoding typing rules using InferType's classes and methods. She then calls a method in InferType to get a type inference result, which is a binding of type variables to concrete types that satisfies the described type constraints and encoded typing rules. InferType uses the Z3 [22] SMT solver to perform constraint solving for computing that binding by translating the described type constraints and typing rules to Z3 formulae.

We develop an optimization technique for InferType to mitigate the performance bottleneck for handling programs containing deeply nested types. InferType's optimization is based on the idea of reducing the search space of type variables in SMT solving. It first pre-computes a structure of each type variable in the given type constraints described by the compiler writer. It then generates extra Z3 formulae that explicitly list the possible types each type variable ranges over based on the pre-computed structure for reducing the search space.

We use InferType to implement type inference engines for a small demonstrative language and a subset of Python. This Python subset does not allow explicit type annotations as type annotations are ignored by the ordinary Python without an external type checker. We choose this Python subset for evaluation since type inference for this subset requires complex typing rules to reconstruct static types in dynamically typed programs. We conduct experiments over a collected dataset to evaluate the performance of type inference by our implemented type inference engine for the Python subset, and also to validate the effectiveness of our developed optimization technique. Compared with an existing, manual type inference engine for Python using Z3, the implemented type inference engine using InferType is able to infer types for real-world programs with comparable performance. Furthermore, the experiment results show that the implemented type inference engine can handle programs with deeply nested types much more efficiently. We also observe that InferType's optimization can potentially improve the performance of type inference for programs containing nested types. Our findings support the claim that InferType is pragmatically applicable of helping implement constraint-based type inference in language development.

3.2 Implementing Type Inference Engines

A number of modern languages have support for automatic type inference. Language developers, or compiler writers have to implement type inference when designing a new language. However, it is a challenging task without appropriate tool support. Classical supporting tools for language development include lex, yacc [58], and bison [97], which are lexer and parser generators. There are also tool suites commonly called language workbenches, which are development environments for making domain-specific languages [106, 31]. The Spoofax [49] language workbench, for instance, supports code generators that produce type checkers and editor plugins from high-level language definitions. However, existing tools have limited support for code generators targeting type inference.

For instance, let us consider implementing a type inference engine for a small language called *Mini*λ. It automatically reconstructs types for programs without explicit type annotations. We use this language for demonstrative purposes because of its simplicity. One of the common approaches to implementing such a type inference engine is to use constraint-based type inference. A number of existing type inference systems are formulated into a constraint-based approach because of its extensibility and flexibility [80, 93, 92, 50]. We below consider implementing a constraint-based type inference for Miniλ.

Listing 3.1: Syntax of Miniλ.

```
e ::= n
    | r
    | s
    | x
    | fun x. e
    | e(e)
    | e + e

t ::= int
    | float
    | str
    | arrow<t, t>
```

The syntax of expressions and types in Miniλ is given in Listing 3.1. n, r, and s range over integers, real numbers, and string literals. x ranges over variables. *fun x. e* represents a function with parameter x and a body e. Miniλ does not have explicit type annotations. Literals can be regarded as

type-annotated expressions. Functions or function parameters do not have type annotations. $e(e)$ represents a function application. $e + e$ represents an addition of two sub-expressions. In Miniλ, the plus operator can either add numbers or concatenate strings. The type syntax of Miniλ consists of primitive types `int`, `float`, and `str`. It also consists of the composite type `arrow` for describing function types.

A subset of the typing rules in Miniλ's type system are given in Figure 3.1. T-App specifies how a function application is typed so that the applied expression must be an arrow type consistent with the argument type. The type constraint in the premises specifies the subtype relation that the type variables must satisfy. T-Plus-Float and T-Plus-Str specify how a plus expression is typed by overloading. The operand and resulting types should all be consistent with either `float` or `str`. ST-Int-Float specifies that `int` is a subtype of `float`. ST-Arrow specifies that two arrow types are subtype of one another if the parameter type is contravariant and the return type is covariant.

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad t_1 <: \texttt{arrow<}t_2\texttt{, }t_3\texttt{>}}{\Gamma \vdash e_1(e_2) : t_3}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : \texttt{float} \qquad \Gamma \vdash e_2 : \texttt{float}}{\Gamma \vdash e_1 + e_2 : \texttt{float}}\ \text{T-Plus-Float}$$

$$\frac{\Gamma \vdash e_1 : \texttt{str} \qquad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash e_1 + e_2 : \texttt{str}}\ \text{T-Plus-Str}$$

$$\frac{}{\texttt{int} <: \texttt{float}}\ \text{ST-Int-Float}$$

$$\frac{t_{21} <: t_{11} \qquad t_{12} <: t_{22}}{\texttt{arrow<}t_{11}\texttt{, }t_{12}\texttt{>} <: \texttt{arrow<}t_{21}\texttt{, }t_{22}\texttt{>}}\ \text{ST-Arrow}$$

Figure 3.1: Selected typing rules for Miniλ.

An example program in Miniλ is given in Listing 3.2. We will use this example program for demonstration in the rest of the chapter due to its simplicity although this program is simple enough to be able to perform type inference in a bottom-up manner. This program first defines a function with parameter `f`; its body is another function with parameter `x`. The body of the function with parameter `x` is a function application `f(x)`. After that, the function with parameter `f` is applied to a function with parameter `y`. The

resulting value of the whole expression is a function taking one argument, and its body concatenates the passed argument to a string literal `"a"`.

Listing 3.2: An example program in Miniλ.

```
fun f. fun x. f(x)
    (fun y. y + "a")
```

To implement a constraint-based type inference for Miniλ, a compiler writer first assigns a type variable to every variable and expression in a program. Then the compiler writer generates the relations between type variables by applying the typing rules while traversing the abstract syntax tree (AST) of the program. A relation between type variables, which we call a *type constraint*, is a logical assertion that describes how the type variables should be constrained. In this study, we consider those type constraints in Listing 3.3. $<:$ represents the subtype relation between two types. $\wedge$ and $\vee$ represent logical conjunction and disjunction; $\rightarrow$ and $\neg$ represent logical implication and negation.

Listing 3.3: Type constraints.

```
constraint ::= t <: t
             | constraint ∧ constraint
             | constraint ∨ constraint
             | constraint → constraint
             | ¬ constraint
```

Type constraints are generated during AST traversal by applying part of the typing rules in the target language, which we call *syntax-related typing rules*. A syntax-related typing rule is a typing rule that includes program syntax symbols. For example, T-App, T-Plus-Float, and T-Plus-Str are syntax-related typing rules in Miniλ. The compiler writer will generate the type constraints in Listing 3.4 for the program in Listing 3.2 by traversing its AST. $t_?$ represents the unique type variables assigned to the variables and expressions in the program. t_f, t_x, and t_y are assigned to parameter `f`, `x`, and `y` of the defined functions. t_{fx} and t_{plus} are assigned to the resulting types of `f(x)` and `y + "a"`. t_e is assigned to the resulting type of the whole expression. Constraint (1) and (2) arise from T-App. For example, in (1), t_f is constrained to be a subtype of `arrow<`t_x`,` t_{fx}`>` generated from the application `f(x)`. Constraint (3) arises from T-Plus-Float and T-Plus-Str. This constraint must be properly included in the generated set of type constraints to represent the overloading of the plus operator. It describes that the operand types, t_y and `str` (type of the string literal `"a"`), and the

resulting type t_{plus} must be subtype and supertype of either `float` or `str`, represented using logical connectives.

Listing 3.4: Type constraints for the example program.

(1) t_f <: arrow<t_x, t_{fx}>
(2) arrow<t_f, arrow<t_x, t_{fx}>> <: arrow<arrow<t_y, t_{plus}>, t_e>
(3) (t_y <: float ∧ str <: float ∧ float <: t_{plus})
∨ (t_y <: str ∧ str <: str ∧ str <: t_{plus})

After constraint generation, the compiler writer must then implement a constraint solver considering the rest of the typing rules to solve the generated type constraints. The solver must find a consistent binding of concrete types assigned to the type variables that satisfies all the type constraints. The solver must consider the other typing rules from the applied typing rules in AST traversal, which we call *subtype-related typing rules*, to find such a binding. Unlike syntax-related typing rules applied during AST traversal, a subtype-related typing rule derives a conclusion in the form of a subtype relation and a subtype-related typing rule does not include syntax symbols of program expressions. For example, the implementing solver for Miniλ must consider ST-Int-Float and ST-Arrow in Figure 3.1 to solve the type constraints in Listing 3.4. Although well-known algorithms such as unification [91] and closure computation [82] have been developed, implementing these algorithms is technically labor-intensive and error-prone. Furthermore, implementing an efficient solver becomes more sophisticated especially for handling complex programs when developing real-world languages. As we will show in Section 3.3.3, it would be time-consuming for inferring types of programs with nested types by a straightforward implementation of such a solver. Technical approaches are needed to overcome those challenges considering practical usage.

3.3 InferType

InferType is an embedded domain-specific language that helps compiler writers implement efficient constraint-based type inference. A compiler writer gives type constraints derived by syntax-related typing rules and subtype-related typing rules to InferType by using InferType's classes and methods. InferType then solves the given type constraints based on the subtype-related typing rules by invoking the Z3 SMT solver. To perform constraint solving efficiently, InferType pre-computes structures for type variables in-

volved in the given type constraints, and then reduces the search space of the involved type variables in SMT solving.

InferType supports constraint-based type inference whose type system is expressed by one single binary type relation, which is often called *subtype*. InferType assumes that the supported binary type relations hold reflexivity and transitivity.

Below in this section, we first demonstrate how compiler writers can use InferType to implement their type inference engines in Section 3.3.1. Then we show how InferType performs type inference by translation to Z3 in Section 3.3.2. In Section 3.3.3, we present the pre-process of InferType for mitigating the performance bottleneck in constraint solving, which is our main scientific contribution in this study.

3.3.1 Programming Interface

InferType is a Java library for implementing constraint-based type inference engines, which are software programs that automatically infer types for programs without explicit type annotations in a target language. A compiler writer uses classes and methods of InferType to describe type constraints derived by syntax-related typing rules and encode subtype-related typing rules. Constraint solving is then performed by calling a method of InferType to get a type inference result.

3.3.1.1 Describing types and type constraints

First, types in the target language must be described in InferType. In InferType, a type is an object taking one string argument as the name of that type, and zero or more type arguments.

```
type(typename, type, type, ...)
```

For instance, a primitive type `int` in Miniλ is described as

```
InferType inf = new InferType();
Type intTy = inf.type("int");
```

`inf` is an instance of InferType's main class. The compiler writer declares this instance to describe types, type constraints, and encode subtype-related typing rules. `Type` is the class representing types. Primitive types are described as types taking zero type argument in InferType. A composite type `arrow<int, str>` is described as

```
inf.type("arrow", inf.type("int"), inf.type("str"))
```

InferType also deals with type variables. A type variable is an object taking one string identifier.

```
typevar(identifier)
```

A type variable t_k is described as

```
TypeVar tk = inf.typevar("t_k");
```

`TypeVar` is a subclass of `Type` representing only type variables. Type variables can also be used as type arguments to make a composite type. An arrow type `arrow<`t_k`, int>` is described as

```
inf.type("arrow", inf.typevar("t_k"), inf.type("int"
   ))
```

InferType can also help produce a type variable with a generated, unique string identifier by calling `inf.typevar` without an argument.

```
inf.typevar() // a fresh generated type variable
```

Listing 3.5: Describe type constraints.

```
inf.subtype(t1, t2)  // <:
inf.and(c1, c2)      // ∧
inf.or(c1, c2)       // ∨
inf.implies(c1, c2)  // →
inf.not(c)           // ¬
```

Then, the compiler writer describes type constraints generated by applying syntax-related typing rules during AST traversal, and gives the type constraints to InferType for type inference. A compiler writer describes the type constraints using InferType's methods given in Listing 3.5. t_1, t_2 represent types and c, c_1, c_2 represent type constraints. Each method corresponds to each kind (comments on the right) of the type constraints in Listing 3.3. For example, constraint (3) in Listing 3.4 is described as

```
Constraint cst3 = inf.or(
  inf.and(inf.subtype(t_y, floatTy),
          inf.subtype(strTy, floatTy),
          inf.subtype(floatTy, t_plus)),
  inf.and(inf.subtype(t_y, strTy),
          inf.subtype(strTy, strTy),
          inf.subtype(strTy, t_plus)));
```

`Constraint` is the class representing type constraints. `t_y` and `t_plus` refer to type variables t_y and t_{plus} created by `inf.typevar`. `floatTy` and `strTy` are primitive types created by `inf.type`.

Finally, the described type constraint is given to InferType for type inference by

```
inf.add(cst3);
```

The other type constraints in Listing 3.4 can be described and given to InferType in a similar manner.

3.3.1.2 Encoding subtype-related typing rules for constraint solving

The compiler writer encodes the rest of the typing rules, which are not used in Section 3.3.1.1 and are called subtype-related typing rules, and give them to InferType for type inference. A subtype-related typing rule in InferType is an implication from zero or more premises to one conclusion encoded as in Listing 3.6. A premise or conclusion is one single subtype relation created by `inf.subtype`.

`premise, conclusion ::= inf.subtype(`t_1`,` t_2`)`

Method `declare` specifies the conclusion; the following method `when` specifies the premises. In InferType, all type variables involved in the premises must be included in the conclusion of an encoded subtype-related typing rule. For example, ST-Arrow in Figure 3.1 is encoded as

```
inf.ruleBuilder
    .declare(inf.subtype(
              inf.type("arrow", t11, t12),
              inf.type("arrow", t21, t22)))
    .when(inf.subtype(t21, t11),
          inf.subtype(t12, t22));
```

`t11`, `t12`, ... are type variables created by `inf.typevar`.

Listing 3.6: Encode subtype-related typing rules.

```
inf.ruleBuilder
    .declare(conclusion)
    .when(premise, premise, ...)
```

A subtype-related typing rule without premises can also be encoded as

```
inf.ruleBuilder
    .declare(conclusion)
    .always()
```

Method `always` specifies that the conclusion holds without a condition, which is a syntax sugar for method `when` without argument. For example, ST-Int-Float is encoded as

```
inf.ruleBuilder
    .declare(inf.subtype(intTy, floatTy))
    .always();
```

Encoded subtype-related typing rules by `inf.ruleBuilder` are implicitly given to InferType for type inference.

3.3.1.3 Solving type constraints based on the encoded subtype-related typing rules

After describing type constraints and encoding subtype-related typing rules, the compiler writer calls method `solve` to solve the given type constraints based on the given encoded subtype-related typing rules.

```
Map<TypeVar, Type> solution = inf.solve();
```

By calling this method, InferType computes if there is a binding of the involved type variables in the given type constraints to concrete types so that all type constraints are evaluated to be true under the given encoded subtype-related typing rules by replacing the type variables with their concrete types. If so, InferType outputs that binding as the type inference results. For Listing 3.2, it will output a `Map` object representing the following binding.

$\{t_f \mapsto$ `arrow<str, str>`$, t_x \mapsto$ `str`$, t_{fx} \mapsto$ `str`$,$
$t_y \mapsto$ `str`$, t_{plus} \mapsto$ `str`$, t_e \mapsto$ `arrow<str, str>`$\}$

Otherwise, it indicates that the constraint solving fails such that InferType could not find a binding that makes all the given type constraints hold. InferType then can provide a set of type variables appearing in the ill-typed constraints.

```
Set<TypeVar> untvars = inf.untypableTypeVars();
if (!untvars.isEmpty()) { // type error occurred
    // further locate type errors
}
```

The compiler writer can manually track the type variables to program expressions such as recording them into a `Map` object during AST traversal:

```
locTrack.put(tv, String.format(
    "at file %s: line %s, column %s",
    fileName, lineNum, columnNum));
```

where `locTrack` is the `Map` object that associates a type variable `tv` to its program location. Here, a value in the `Map` object is a string representation of a program location, where `fileName`, `lineNum`, and `columnNum` are managed by the compiler writer. Then she is expected to use the untypable type variables returned by InferType to retrieve the expressions that causes the type error, and further generate a type error message.

3.3.1.4 Declaring User-Defined Types

InferType also supports user-defined types such as `struct` in the C language and classes in object-oriented languages. During AST traversal, a compiler write can declare subtype-related typing rules for those user-defined types, and give those declarations to InferType. A compiler writer can describe new types, encode new subtype-related typing rules, and give them to InferType at any phase before calling `inf.solve` for constraint solving.

For example, suppose that a compiler writer is implementing a type inference engine for a language supporting class definitions. When traversing an AST node for a class definition such as

```
class Car(Vehicle):
    ...
```

which defines a class `Car` that inherits another class `Vehicle`, the compiler writer describes a new type for that class as

```
Type carTy = inf.type("car");
```

She then encodes a new subtype-related typing rule specifying the inheritance as

```
inf.ruleBuilder
    .declare(inf.subtype(carTy, vehicleTy))
    .always();
```

`vehicleTy` is described as `inf.type("vehicle")` and it is the type for class `Vehicle`. Like other subtype-related typing rules, the encoded subtype-related typing rule here is also implicitly given to InferType for type inference. InferType will perform constraint solving considering this encoded subtype-related typing rule when method `inf.solve` is called.

3.3.2 Translation to Z3

InferType performs constraint solving using the Z3 SMT solver. When method `inf.solve` is called, it translates the type constraints and subtype-related typing rules given by the compiler writer into Z3 formulae. The translated Z3 formulae are all asserted to check satisfaction by invoking the Z3 SMT solver to find if there is an interpretation of variables that makes all the asserted formulae true.

3.3.2.1 Translating types and type constraints

Types are translated to Z3 constants over a generated Z3 data type declaration. The data type declaration is generated by accumulating all the `Type` objects created by method `inf.type` included in the type constraints and encoded subtype-related typing rules given to InferType. This generated data type declaration represents the type definition of the target language. For Miniλ, the data type declaration for `ztype` is generated as [1]

```
(declare-datatypes () ((ztype
    (Int)
    (Float)
    (Str)
    (Arrow (ArrowP1 ztype) (ArrowP2 ztype)))))
```

It corresponds to t in Listing 3.1. `Int`, `Float`, and `Str` are translated Z3 constructors representing the primitive types. `Arrow` represents the composite type `arrow`, where `ArrowP1` and `ArrowP2` are generated accessors for `Arrow`, which indicates that arrow types take two arguments. A type variable t_k is translated to a Z3 constant z_k ranging over `ztype`.

`(declare-constant` z_k `ztype)`

A composite type `arrow<`t_k`, int>` is translated to an application of `ztype` constructors.

`(Arrow` z_k `Int)`

The given type constraints described by Listing 3.5 are straightforwardly translated to Z3 boolean propositional formulae by the following translation function.

`trans{inf.subtype(`t_1`,` t_2`)} = (zsubtype` z_1 z_2`)`
`trans{inf.and(`c_1`,` c_2`)} = (and trans{`c_1`} trans{`c_2`})`

[1] Z3 code is represented using the SMT-LIBv2 [7] syntax in the following.

`trans{inf.or(`c_1`, `c_2`)} = (or trans{`c_1`} trans{`c_2`})`
`trans{inf.implies(`c_1`, `c_2`)} = (=> trans{`c_1`} trans{`c_2`})`
`trans{inf.not(`c`)} = (not trans{`c`})`

c, c_1, and c_2 represent type constraints. The subtype relation $<:$ is declared as `zsubtype` in Z3.

```
(declare-fun zsubtype (ztype ztype) Bool)
```

It specifies that `zsubtype` is a Z3 function that takes two arguments of the data type `ztype` and returns a boolean value. Logical connectives in type constraints are directly translated to Z3 logical operators. `=>` is the logical implication operator in Z3. For instance, `cst3` in Section 3.3.1.1 is translated to

```
(or (and (zsubtype z_y Float)
         (zsubtype Str Float)
         (zsubtype Float z_plus))
    (and (zsubtype z_y Str)
         (zsubtype Str Str)
         (zsubtype Str z_plus)))
```

`z_y` and `z_plus` in Z3 refer to `t_y` and `t_plus` in Java.

3.3.2.2 Translating subtype-related typing rules

The encoded subtype-related typing rules are processed by InferType to compute subtype relations, which are then translated into Z3 formulae. Since InferType implicitly assumes the reflexivity and transitivity rules, it also considers these rules when computing subtype relations. We borrow the idea of this process from Typette [39].

For each primitive type and composite type, InferType enumerates all its subtypes and super types in accordance with given subtype-related typing rules. It may also generate type constraints that those subtypes and super types must hold. We below mention how InferType enumerates subtypes. InferType does the same for enumerating super types.

Suppose that InferType enumerates subtypes for a target type $type_T$. Since InferType assumes the reflexivity rule, it first obtains this subtype relation: $type_T$ is a subtype of $type_T$. Next, suppose that the following subtype-related typing rule is given:

$$\frac{premise_1}{type_1 <: type_2}$$

Here, $type_k$ is either a primitive type, a composite type, or a type variable. If $type_2$ and $type_T$ can be *unified*, that is, they are lexically equivalent by replacing the type variables in $type_2$ and $type_T$ with other type variables, primitive types, or composite types, then InferType obtains the following subtype relation: $type_1$ is a subtype of $type_T$ when $premise_1$ holds, where the occurrences of some type variables in $type_1$ and $premise_1$ are replaced as they are for the unification between $type_2$ and $type_T$. When enumerating subtypes and super types for composite types, InferType only enumerates for the most generic forms of composite types, where all type arguments are type variables. For example, InferType computes subtypes and super types for `arrow<`t_1`,` t_2`>`, where t_1 and t_2 are type variables; but it does not compute subtypes or super types for an individual type such as `arrow<int, int>` or `arrow<int,` t_1`>`. Besides, InferType does not consider the reflexivity rule when enumerating subtypes and super types for composite types.

By this enumeration, in the case of Miniλ, InferType enumerates `float` (by reflexivity) and `int` (by ST-Int-Float) as subtypes of `float`. By ST-Arrow, InferType enumerates `arrow<`t_{11}`,` t_{12}`>` as a subtype of `arrow<`t_1`,` t_2`>` under premises $t_1 <: t_{11}$ and $t_{12} <: t_2$.

Furthermore, InferType considers the transitivity rule. Given the following rules, if $type_2$ and $type_3$ are unified,

$$\frac{premise_1}{type_1 <: type_2} \quad \frac{premise_2}{type_3 <: type_4}$$

InferType combines the two rules to derive the following new rule:

$$\frac{premise_1 \qquad premise_2}{type_1 <: type_4}$$

where the occurrences of some type variables in $type_1$, $type_4$ and $premise_1$, $premise_2$ are replaced as they are for the unification between $type_2$ and $type_3$. They are replaced according to the substitution for a most general unifier [52], which is a complete and minimal substitution. Existential quantifiers are added[2] to the type variables that are included in $premise_1$ and $premise_2$ but not in $type_1$ or $type_4$. When $premise_1$ and $premise_2$ include $type_i <: type_j$ and $type_j <: type_k$, and all the type variables in $type_j$ are not included except $type_i$, $type_j$, and $type_k$, InferType combines $type_i <: type_j$ and $type_j <: type_k$ and transforms them into $type_i <: type_k$ by the transitivity rule.

[2]When an existential quantifier is included in the resulting Z3 formulae, Z3 might fail to correctly derive types. This is a limitation of InferType but it is out of the scope of this study.

The derived new subtype-related typing rules are used to enumerate a subtype of the target type $type_T$. Furthermore, it may be combined with another rule to derive another new rule by the transitivity rule. InferType iterates this to enumerate all subtypes of $type_T$. It tries all possible combinations between subtype-related typing rules including the derived ones.

For example, suppose that InferType is computing a subtype of `intArray` (which is described as a primitive type `inf.type("intArray")`) and given two subtype-related typing rules:

$$\frac{t_1 <: t_2}{\texttt{list<}t_1\texttt{>} <: \texttt{array<}t_2\texttt{>}}\ \text{a} \qquad \frac{t <: \texttt{int}}{\texttt{array<}t\texttt{>} <: \texttt{intArray}}\ \text{b}$$

InferType first unifies `array<`t_2`>` and `array<`t`>`, and finds a substitution $\{t_2 \mapsto t\}$ for that unification. Then it combines the rules and derives:

$$\frac{t_1 <: t \qquad t <: \texttt{int}}{\texttt{list<}t_1\texttt{>} <: \texttt{intArray}}\ \text{a-b}$$

Furthermore, InferType implicitly applies the transitivity rule and obtains:

$$\frac{t_1 <: \texttt{int}}{\texttt{list<}t_1\texttt{>} <: \texttt{intArray}}\ \text{a-b}$$

This new rule is considered for the subtype enumeration, and InferType obtains `list<`t_1`>` as a subtype of `intArray` under the premise $t_1 <:$ `int`.

The derived new rule may be combined with existing other rules, and another new rule may be derived from that combination by the transitivity rule. The iteration of this combining may not terminate within finite steps. For example, this iteration never terminates if the given subtype-related typing rules include a self-recursive subtype relation such as:

$$\frac{t_1 <: t_2}{\texttt{list<}t_1\texttt{>} <: \texttt{array<}t_2\texttt{>}}\ \text{a} \qquad \frac{}{t <: \texttt{array<}t\texttt{>}}\ \text{r}$$

Here, the conclusion of r includes a self-recursive subtype relation such that a type is a subtype of an array type of itself. InferType combines a and r, and derives:

$$\frac{t_1 <: t_2}{\texttt{list<}t_1\texttt{>} <: \texttt{array<array<}t_2\texttt{>>}}\ \text{a-r}$$

InferType will further combine a-r and r. It will then infinitely combine and derive new rules. It is the InferType users' responsibility to ensure that the given subtype-related typing rules do not cause infinite iteration.

After enumerating all subtypes and super types for each primitive and composite type, InferType generates Z3 formulae representing type constraints for these subtype relations. Suppose that, for a target type $type_T$, its subtypes are $subtype_1$ when $premise_1$ holds, $subtype_2$ when $premise_2$ holds, ..., InferType then generates the following Z3 formula:

```
(forall ((z ztype) (z0 ztype) (z1 ztype) ...)
  (=> (zsubtype z typeT)
    (or (and (= z subtype1) premise1)
        (and (= z subtype2) premise2)
        ...)))
```

`(z ztype)` expresses that a Z3 variable `z` represents a type in the target language such as Miniλ. `zsubtype` is the Z3 function returning true if the first argument is a subtype of the second argument. In the above formula, z_0, z_1, ... are Z3 variables representing type variables appearing in $type_T$, $subtype_1$, $subtype_2$, ... and $premise_1$, $premise_2$, ... `=>` is an implication operator in Z3. For example, for the subtypes of `arrow` in Miniλ, InferType generates the following formula:

```
(forall ((z ztype) (z21 ztype) (z22 ztype))
  (=> (zsubtype z (Arrow z21 z22))
      (and (= z (Arrow (ArrowP1 z) (ArrowP2 z))
           (zsubtype z21 (ArrowP1 z))
           (zsubtype (ArrowP2 z) z22)))))
```

This reads as, for all `z`, `z21`, and `z22`, if `z` is a subtype of `arrow<z21, z22>`, then `z` is an arrow type, `z21` is a subtype of the first argument of `z`, and the second argument of `z` is a subtype of `z22`. InferType also generates a Z3 formula for the super types of `arrow`.

Finally, all translated Z3 formulae are asserted to find if there is an interpretation of variables that makes all the translated Z3 formulae true. If Z3 can find such an interpretation, InferType retrieves and translates that interpretation back to a binding of type variables to concrete types. This binding is returned to the compiler writer. Otherwise, InferType uses the unsat core extraction feature of Z3 to obtain a list of unsat translated type constraints. It then returns the type variables in those unsat translated type constraints by calling `inf.untypableTypeVars` in Section 3.3.1.3.

3.3.3 Optimizing Constraint Solving for Deeply Nested Types

Using Z3 in a straightforward way as in Section 3.3.2 works with respect to performance in common cases such as Listing 3.2. However, the con-

straint solving sometimes becomes extremely slow: we observe that the constraint solving time increases exponentially when the programs contain deeply nested types.

Consider another program in Miniλ given in Listing 3.7. This program extends Listing 3.2 by adding an outer function with parameter `f2`. The body of the function with parameter `x` is a function application, whose argument is `x` and the called function is another function application `f2(f)`. The function with parameter `f2` is then applied to the function with parameter `g`; its resulting value is applied to the function with parameter `y`. The type of parameter `f2` is supposed to be inferred as `arrow<arrow<str, str>, arrow<str, str>>`, which we call a nested type (arrow of arrow). In this study, a nested type is a composite type that takes composite types as arguments. This program can be further modified by defining another outer function `f3` to create a larger nested type, where the type of `f3` is supposed to be inferred as a nested arrow of arrow of arrow type. We create the programs containing `f4` and `f5` by adding more outer functions for larger nested types. By our testing, a straightforward translation to Z3 in Section 3.3.2 performed constraint solving for the programs with `f3` in 0.18s, `f4` in 14.07s, and `f5` in 179m15.35s!

Listing 3.7: Another example program with nested types in Miniλ.

```
fun f2. fun f. fun x. f2(f)(x)
     (fun g. g)
     (fun y. y + "a")
```

We expect that this slowdown arises from the increasing search space for SMT solving. For example, when a type variable is constrained to be a subtype (or super type) of a type taking arrow types as arguments in a given type constraint, that type variable ranges over increasingly many possible types by the size of arrow types in the arguments. In Miniλ, such a type variable ranges over possible types `int`, `float`, `str`, `arrow<int, int>`, `arrow<arrow<int, ...>, int>`, ... by the increasing size of the arrow types in the arguments. When Z3 solves the given type constraints, it tries to search for a possible solution by instantiating the type variables over their possible types.

To mitigate the performance bottleneck of constraint solving containing deeply nested types in Z3, we develop a pre-process for InferType. It first computes structures for the type variables involved in the given type constraints. We call these structures *shapes*. Then it generates and asserts extra Z3 formulae based on the computed shapes. They reduce the search space

of the involved type variables since type variables range over only limited depth of nested types (or primitive types) specified by the computed shapes.

A shape is either a shape variable, a symbol `*`, or a nested structure starting with a symbol `?` presented in Listing 3.8. `*` represents only primitive types; it can never be any composite type. `?` represents composite-type names such as "`arrow`" in Miniλ, which constructs a nested structure for shapes by taking other shapes as arguments. For example, suppose that the shape of a type variable in Miniλ is pre-computed as `?<*, *>`, that type variable would range over only arrow types with primitive types as arguments, which are `arrow<int, int>`, `arrow<int, str>`, ...

Listing 3.8: Shape definition.

```
shape ::= shapevar
        | *
        | ?<shape, shape, ...>
```

3.3.3.1 Pre-Computing Shapes

InferType's pre-process first computes shapes for the type variables involved in the given type constraints generated from syntax-related typing rules during AST traversal. Subtype-related typing rules are not considered when computing shapes.

Given the type constraints from syntax-related typing rules, our pre-process first converts the involved types to shapes by the following function:

```
shapeof{t_k} = s_k
shapeof{primType} = *
shapeof{compTypeName<t_1, t_2, ...>} =
        ?<shapeof{t_1}, shapeof{t_2}, ...>
```

A type variable is converted to a shape variable. A primitive type is converted to `*`. A composite type is converted to a nested structure with `?`. For example, a type `arrow<`t_k`, int>` is converted to a shape `?<`s_k`, *>`.

Then, *shape equations* are derived from the given type constraints. A shape equation is a logical assertion with conjunctions and disjunctions over a binary equality between two shapes given in Listing 3.9. A binary equality $shape_1 == shape_2$ specifies that $shape_1$ and $shape_2$ are lexically identical. Shape equations do not contain logical implication or negation.

Listing 3.9: Shape equations.

```
equation ::= shape == shape
```

```
          | equation ∧ equation
          | equation ∨ equation
```

Shape equations are derived by the following function:

```
derive{t1 <: t2} = shapeof{t1} == shapeof{t2}
derive{¬ t1 <: t2} = TRUE
derive{c1 ∧ c2} = derive{c1} ∧ derive{c2}
derive{c1 ∨ c2} = derive{c1} ∨ derive{c2}
```

The input is the Negation Normal Form (NNF) converted from the given type constraints in the form of Listing 3.3. The NNF is converted to handle type constraints in logical implication and negation. A subtype relation $t_1 <: t_2$ is derived to a binary equality $s_1 == s_2$ between the converted two shapes. For instance, the shape equations in Listing 3.10 are derived from the given type constraints in Listing 3.4.

Listing 3.10: Shape equations for the type constraints.

```
(1') sf == ?<sx, sfx>
(2') ?<sf, ?<sx, sfx>> == ?<?<sy, splus>, se>
(3') (sy == * ∧ * == * ∧ * == splus)
   ∨ (sy == * ∧ * == * ∧ * == splus)
```

Note that no shape equations are derived from subtype relations in logical negation (represented as returning a logical `TRUE` literal). For example, a shape equation $\neg\ s_h$ `==` `*` cannot be derived even if a type constraint $\neg\ t_h$ `<:` `int` is given. s_h can be arbitrary since t_h can be any type except a subtype of `int`.

Next, the derived shape equations are solved to find a binding of shape variables to shapes so that all those shape equations are satisfied. A shape equation consists of conjunction, disjunction, and equality. Disjunctions may make it time-consuming to solve shape equations. An equality such as `?<`s_1`>` `==` `?<*>` may need to run an expensive unification algorithm when solving shape equations.

To make the shape computation fast, InferType adopts an approximate approach. Each shape equation is first transformed into a disjunctive normal form (DNF). Then, each clause in the DNF is separately solved; a set of substitutions for shape variables is found so that the left and right operands of every == operator included in that clause will be lexically identical after the substitutions are applied to the operands. These substitutions are found by unification, which we implement by the disjoint-set (union-find) algorithm. For example, when a clause of DNF is `?<`s_1`>` `==` `?<*>` $\wedge$ s_2 `==` s_1, a set of

substitutions $\{s_1 \mapsto$ `*`, $s_2 \mapsto s_1\}$ is found. Note that one arbitrary set of substitutions is found when there exists multiple valid sets of substitutions.

After the unification, the resulting set of substitutions are compared with the sets of substitutions for the other clauses of the shape equation. If all the sets of substitutions are identical, the first clause of the shape equation, which is a conjunction of equalities, is used in the next step. Otherwise, the whole shape equation is excluded in the next step. This makes our shape computation approximate. When valid sets of substitutions are not found for some clauses, the shape equation is also excluded.

Finally, the remaining clauses, which are not excluded in the previous step, are combined by conjunction, and they are solved. The substitutions for shape variables are found by unification as in the previous step. A shape variable is bound to a shape obtained by applying those substitutions to that shape variable. If a shape is not obtained, the shape variable does not have a binding. For example, our approximate shape computation computes the following binding for Listing 3.10:

$\{s_f \mapsto$ `?<*, *>`, $s_x \mapsto$ `*`, $s_{fx} \mapsto$ `*`,
$s_y \mapsto$ `*`, $s_{plus} \mapsto$ `*`, $s_e \mapsto$ `?<*, *>`$\}$

Note that some shape variables do not have a binding. For the type variables corresponding to those shape variables, InferType does not generate extra Z3 formulae in the next step in Section 3.3.3.2. Those type variables are not optimized for constraint solving. For example, in a target language, suppose that a composite type `arrow` is a subtype of a primitive type `object`, and an operator `!=` takes operands of `object` type. Also suppose that a variable v_k is initialized with a value of an arrow type and v_k appears as an operand for `!=`. This will generate a shape equation s_k `== ?<*, *>` $\wedge$ s_k `==` `*`, where s_k corresponds to a type variable t_k for the variable v_k. The initialization of v_k generates s_k `== ?<*, *>`, but the `!=` operator generates s_k `==` `*`. s_k does not have a binding since there is no concrete shape for s_k to satisfy that shape equation. Note that not all shape variables s_k lose a binding when an arrow type is a subtype of a primitive type `object`. They lose a binding only when the type variables t_k corresponding to s_k appears in an expression where a syntax-related typing rule specifies that t_k is a subtype of `object`. Recall that shape computation does not consider subtype-related typing rules.

3.3.3.2 Reducing Search Space

After solving shape equations, InferType generates extra Z3 formulae that explicitly represent what types the involved type variables range over regarding the computed shapes for reducing the search space. If a shape is `*`, its type variable ranges over only all primitive types. If a shape is `?` with n arguments, its type variable ranges over all composite types with n arguments.

Given a type variable t_k and the computed shape `?<`$shape_1$`,` $shape_2$`, ...,` $shape_n$`>` for its corresponding shape variable s_k, where $shape_i$ represents some computed shapes, InferType generates extra Z3 formulae for t_k as

```
(or (= z_k (c_1^n shape_1 shape_2 ... shape_n))
    (= z_k (c_2^n shape_1 shape_2 ... shape_n))
    ...)
(or (= shape_1 (c_1^m shape_11 shape_12 ... shape_1m))
    (= shape_1 (c_2^m shape_11 shape_12 ... shape_1m))
    ...)
(or (= shape_2 p_1) (= shape_2 p_2) ...)
...
(or (= shape_n ...) ...)
...
```

z_k is the translated Z3 constant for t_k. c_i^n represents all composite types that take n arguments in the target language. The possible types for z_k are explicitly listed by logical disjunction. If $shape_j$ is a nested structure with `?` that takes m arguments, InferType will then generate extra Z3 formulae for $shape_j$ similar to the extra Z3 formulae for z_k. An example is $shape_1$ in the above code. $shape_{1i}$ represents its arguments. If $shape_j$ is `*`, InferType will then generate extra Z3 formulae specifying that $shape_j$ ranges over only primitive types. An example is $shape_2$ in the above code. p_i represents all primitive types in the target language.

For example, by the computed shape `?<*, *>` for t_f in Listing 3.4, the extra Z3 formulae are generated as

```
(or (= z_f (Arrow sh_1, sh_2)))
(or
    (= sh_1 Int)
    (= sh_1 Float)
    (= sh_1 Str))
(or
    (= sh_2 Int)
```

```
    (= sh_2 Float)
    (= sh_2 Str))
```

They specify that `z_f` ranges over only arrow types with arguments of primitive types. `?` here corresponds to composite type names that take two arguments, which only "`arrow`" matches in Miniλ. The arguments `sh_1` and `sh_2` are generated Z3 constants.

Suppose that Miniλ defined list types `list<`t`>` and array types `array<`t`>`, and also suppose that the shape of a type variable t_h were computed as `?<?<*>>`. Then InferType would generate the extra Z3 formulae as:

```
(or
    (= z_h (List sh_4))
    (= z_h (Array sh_4)))
(or
    (= sh_4 (List sh_5))
    (= sh_4 (Array sh_5)))
(or
    (= sh_5 Int)
    (= sh_5 Float)
    (= sh_5 Str))
```

`z_h` is the translated Z3 constant for t_h. `sh_4` and `sh_5` are generated Z3 constants for the arguments. `?` here corresponds to composite type names that take one argument, which "`list`" or "`array`" matches.

Note that asserting extra Z3 formulae generated from some approximately computed shapes in our shape computation might cause type inference failure, that is, the constraint solving would result in unsat even though the types in the given type constraints could be inferred. When our shape computation approximately computes some shapes by excluding some shape equations, InferType would encounter this limitation if that type variable in the given type constraints does not have an inferred type with that approximately computed shape.

Listing 3.11: Type constraints involving object types.

(21) `arrow<int, int>` <: t_c
(22) `object` <: $t_c \vee t_d$ <: t_c
(23) `object` <: t_d

For example, consider the type constraints given in Listing 3.11 in the Miniλ extended with type `object` where an arrow type is a subtype of the `object` type. The (only) valid binding that satisfies the type constraints is $\{t_c \mapsto$ `object`, $t_d \mapsto$ `object`$\}$. The shape equations are derived as:

(21') ?<*, *> == s_c
(22') * == s_c $\vee$ s_d == s_c
(23') * == s_d

By applying the approximate shape computation, (22') will be excluded since it finds two different substitutions $\{s_c \mapsto$ *$\}$ and $\{s_d \mapsto s_c\}$ by unification over the clauses of DNF. The approximate shape computation then only computes (21') and (23'), and outputs $\{s_c \mapsto$?<*, *>, $s_d \mapsto$ *$\}$. The constraint solving will then fail since t_c will be restricted to range over only arrow types by the extra Z3 formulae generated from the approximately computed shape ?<*, *> for s_c. On the other hand, a precise shape computation by considering (22') would discard the shapes for s_c and s_d, and then the type constraints can be correctly solved.

To handle this limitation for InferType producing correct type inference, in our implementation, when InferType cannot infer the types with the pre-process, it will recover the constraint solving once again without the pre-process. In the second run of the recovery, there is no extra Z3 formula from the computed shapes and all the constraint solving is not optimized. Note that if the given type constraints contain type errors, both the first and second run will result in unsat. InferType can produce correct type inference instead of producing some wrongly inferred types, though it will take extra time by running two times when our approximate shape computation encounters this limitation.

3.3.3.3 Discussion

InferType's shape computation is sound, that is: If InferType can compute a binding that satisfies the given type constraints with the asserted extra Z3 formulae from the shape computation, then that binding must be one of the bindings that satisfies the original type constraints. Since InferType asserts the extra Z3 formulae by conjunction with the given original type constraints, if InferType computes a binding for satisfaction, then that binding must also satisfy the original type constraints. Although InferType might not be able to compute some of the bindings that satisfy the original type constraints since the possible inferred types for the type variables are restricted by the computed shapes, a computed binding by InferType must be a valid solution. InferType is guaranteed to never produce a wrongly inferred type if a program is ill-typed.

InferType's shape computation is not complete in general, where the completeness of the shape computation is defined as: InferType can compute a binding that satisfies the type constraints with the asserted extra

Z3 formulae if the original type constraints can be satisfied. InferType's shape computation would be complete under the following assumption: all subtype relations hold only between types with the same structure. In such a target language, if a type variable is constrained to be a subtype or super type of a type, then the structure of the inferred type for that type variable must be lexically identical to the structure of that type. For instance, the shape computation for Miniλ is complete. InferType's shape computation is not complete when the target language supports subtype relations between types with different structures. In practical cases, such languages often support a type called `object` which is a super type of any type. For example, suppose that the following type constraints are given in a target language where the only common super type of arrow and map types is a primitive type `object`:

`arrow<int, int>` `<:` t_a
`map<int, int>` `<:` t_a

The type variable t_a can be only inferred as `object` for satisfaction. The shape computation will compute the shape for t_a as `?<*, *>` from the derived shape equations:

`?<*, *>` `==` s_a
`?<*, *>` `==` s_a

However, InferType then could not find a binding for t_a with the asserted extra Z3 formulae since t_a cannot be inferred as a type with that computed shape.

InferType incorporates a workaround to avoid its failure of computing shapes when the target language supports subtype relations between types with different structures. Some shape variables lose their bindings in our shape computation if part of the shape equations cannot be solved by unification, while the other shape variables can still be computed and the constraint solving for those corresponding type variables can still be optimized. For the previous example containing arrow and map types, a common practice for InferType correctly computing shapes and further computing types is by programmers' type annotations. If a programmer annotates the expression for t_a as type `object`, a type constraint t_a `<:` `object` will be collected by AST traversal and given to InferType together with the other type constraints. As we discussed in the last paragraph in Section 3.3.3.1, s_a will lose a binding in the shape computation because the derived shape equations including s_a cannot be solved by unification. InferType would then correctly perform type inference although the constraint solving for t_a would not be

optimized.

When InferType wrongly computes some shapes and cannot infer types because of the incompleteness of the shape computation, InferType would encounter a fallback to recover the constraint solving as we mentioned in Section 3.3.3.2. Although the constraint solving is entirely unoptimized in such cases, InferType would be able to correctly compute a binding as the type inference result.

3.4 Experiment

We implement a type inference engine for Miniλ using InferType. We include it in our artifact on Zenodo [3]. The source code contains 291 LOC in Java with 48 LOC (manually counted) using InferType. Owing to InferType's optimization, the implemented engine performs type inference within around 0.18s for each program with nested types `f3`, `f4`, and `f5` as shown in Listing 3.7, which is way faster than a straightforward translation to Z3.

To further demonstrate the usage of InferType and the effectiveness of InferType's optimization, we conduct several experiments by implementing a type inference engine using InferType for a subset of Python, named *Subpet*. We choose Python as the target language because we suppose that it is a good testbed for evaluating the usage capability of InferType when implementing a type inference engine with complex typing rules for a dynamic language like Python. Another motivation of choosing Python is being aware of an existing work Typpete [39]. Typpete is one of the state-of-the-art type inference engines for Python using the Z3 SMT solver. Comparing a type inference engine of generated Z3 code by InferType with a manually written one would be a valuable evaluation for the usage of InferType.

Subpet is a re-implementation of a subset of Typpete using InferType. Typpete performs type inference by a manual encoding of type constraints and subtype-related typing rules to Z3 formulae while Subpet does it by using InferType. Subpet adopts a similar type system to Typpete. The main differences between Subpet's and Typpete's type systems are: The type system of Subpet is flow-insensitive; Subpet does not support some container types such as sequence types `sequence<`*t*`>`. Subpet contains 2,892 LOC in Java with 226 LOC (manually counted) using InferType. It uses the `ast` module in Python to parse the source programs. The other code mainly consists of AST traversal and typing environment, class table definitions. Typpete encodes the type inference of Python programs into a MaxSMT

[3] `https://zenodo.org/records/10981733` (visited on 2023-09-06).

problem by a manual encoding of type constraints and typing rules to Z3 formulae. It outputs type annotations for an input Python program. Typpete contains 6,189 LOC implemented in Python. Subpet is not so powerful as Typpete because Subpet is limited by the amount of our implementation resource. Besides, InferType encodes the type inference for a target language as a SMT problem while Typpete encodes it for Python into a MaxSMT problem. Typpete could possibly infer some principal types by manually encoding soft and hard constraints in syntax-related typing rules, while principal typing is generally not considered by InferType. Nevertheless, Subpet is able to infer types for the programs in our dataset including real-world programs.

For the experiments, we build a dataset. It is publicly available in our artifact. The dataset contains five sets of Python programs including no type annotations:

The first set (*Bench*) contains 44 Python benchmark programs obtained from Typpete's artifact [4]. A few innocuous code modifications are made to overcome the implementation limitations of Subpet such as using explicit arguments instead of implicit ones. These modifications do not impact the functionality of the code.

The second set (*Fs*) is a series of 7 artificial Python programs containing larger nested types. One of these programs, for example, is manually created as:

```
def f0(x):
    return x + x
def f1(f0, x):
    return f0(x)

f1(f0, 42)
```

`f1` is a function taking functions as parameter. Its type is assumed to be inferred as a nested type. This Python program can be considered as a similar correspondence to Listing 3.7 in Miniλ. The program can be further extended with more function definitions `f2`, `f3`, ... for larger nested types similar to what we showed in Miniλ. We create the cases up to `f7`.

The third set (*Codenet*) contains 30 programs obtained from *Project CodeNet* [84]. Project CodeNet is an open-source, large-scale dataset containing solution programs to coding problems from online judge websites. It is expected to be diverse and representative for real-world programs. We downloaded its Python benchmarks, Version 1 on Sep. 6, 2023. We use the `grep`

[4] `https://zenodo.org/records/3996670` (visited on 2023-09-06).

command to search programs containing keywords `graph` and `dijkstra`. Then we obtain 12 problem sets that have more than 10 hits of the two keywords. We use such keyword searching because we expect that there is a high potential of programmers defining data of nested types to solve graph coding tasks using the Dijkstra's algorithm [47]. Then 5 programs are randomly picked up from each problem set (each problem set contains around 100 to 300 solution programs), resulting in 60 programs. 30 of the 60 programs are filtered by the implementation limitations of Subpet (mostly because of unknown library usage with respect to types). The remaining 30 programs are finally included in the third set.

The fourth set (*Large*) contains 4 larger programs/projects. The programs in this set are aimed to estimate the scalability of the shape computation in InferType. Two of the larger programs with 358 and 445 LOC are collected from the MOPSA project [5]. A few code modifications are made to overcome the implementation limitations of Subpet. The other two projects are collected from Typpete's artifact. One project includes 5 Python files with 312 LOC. The other project includes 9 Python files with 846 LOC. Although these programs and projects are not exceedingly large, each of them is more than 10 times larger than the average of the programs in the other sets by LOC.

The fifth set (*Ill*) contains 4 ill-typed programs. These programs are used to demonstrate the behaviour of handling type errors in InferType. Two of the ill-typed programs are benchmark programs collected from MOPSA. The other two ill-typed programs are from Bench. We manually modified the programs to make them become ill-typed. A complete list of all the code changes in the dataset is included in our artifact.

Table 3.1 gives an overview of our dataset. Each value represents the average number per program/project in each set. *LOC* shows the average number of lines of code. *def* shows the average number of function and method definitions. *typevar* and *constraint* show the average number of generated type variables and type constraints derived by syntax-related typing rules during Subpet's AST traversal.

The experiments are conducted on a Ubuntu 19.10 desktop machine with 2.8 GHz Intel(R) Core(TM) i7-6700T CPU and 32GB RAM. The machine is equipped with OpenJDK 14, Python 3.9.6, Z3 4.12.2, and Typpete 0.1.

[5] `https://mopsa.lip6.fr` (visited on 2023-09-06).

Table 3.1: Dataset overview. Each value is the average number per program/project in each set.

	Bench	Fs	Codenet	Large	Ill
LOC	37	17	29	490	18
def	3	5	0.4	45	0.3
typevar	61	29	175	745	35
constraint	45	12	79	613	32

3.4.1 Comparing Subpet with Typpete

First, we show the performance of type inference by Subpet and Typpete. Table 3.2 shows the time results by running the two engines over Bench and Fs in our dataset. We select and include some of the larger programs by LOC in Bench in Table 3.2a. The results from Subpet are the average elapsed time of the later 10 runs by looping Subpet 20 times within the same JVM execution for each input Python program. This is in the interest of the slow JVM startup time. For fair comparison, we also modify Typpete to loop 20 times for each input Python program and then take the average of the later 10 runs.

Both Subpet and Typpete can infer the types for normal programs without nested types or with smaller nested types (programs in Bench and program `f1` to `f3` in Fs) within a reasonable time period. Subpet is faster than Typpete mainly because of the language advantage: Subpet is implemented in Java and Typpete is implemented in Python. However, Typpete exposes its performance downside for program `f4` to `f7` in Fs with deeply nested types. Subpet could finish their type inference much faster. Typpete runs `f5` within 101.4s; it times out (†) for `f6` and `f7` after 1,000s. The performance advantage is not simply because Subpet is implemented in Java rather than Python. It mainly comes from InferType's optimization. We will show our findings in the later subsection.

We also demonstrate the behaviour of handling type errors by Subpet. It can detect the type errors for all the ill-typed programs in Ill. Subpet tracks the types and type variables to program locations (including the file name, line number, and column number) in a `Map` object during constraint generation in the AST traversal methods. By invoking InferType for type inference, Subpet uses the untypable type variables returned by InferType to retrieve the program locations and report them to the programmer, though a better readable text message is not given due to our limited implementation

Table 3.2: Time for type inference by Typpete and Subpet. †timeout (after 1,000s)

(a) Bench

Program	LOC	Typpete (s)	Subpet (s)
`bellman`	61	2.54	0.23
`crafting`	132	3.32	0.27
`deceitful`	36	1.41	0.12
`disjoint`	45	2.20	0.20
`lattice`	81	2.50	0.16
`rock`	79	2.87	0.13
`vehicle`	92	2.65	0.14

(b) Fs

Program	LOC	Typpete (s)	Subpet (s)
`f1`	8	0.85	0.09
`f2`	11	1.17	0.11
`f3`	15	1.51	0.11
`f4`	17	6.52	0.17
`f5`	20	101.4	0.28
`f6`	23	†	1.04
`f7`	27	†	3.08

resource. For example, one of the ill-typed programs in Ill is written as

```
0 # arg_mismatch.py
1 def f(x):
2     return x
3
4 f()
```

This program is ill-typed since function `f` is called without an argument but it is defined as taking one argument. Subpet reports the type error message as

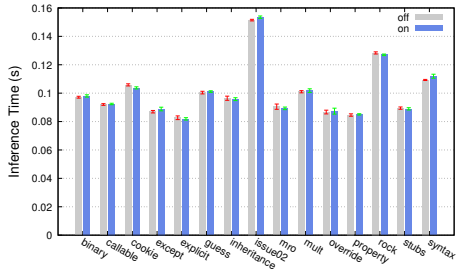
```
type inference failed.
type error location:
arg_mismatch.py: line 4, column 0
```

Typpete would report a similar type error message to the above by Subpet. Note that InferType can detect and help report type errors whenever the optimization is disabled or enabled. InferType would not wrongly infer types with our shape computation if the program is ill-typed.

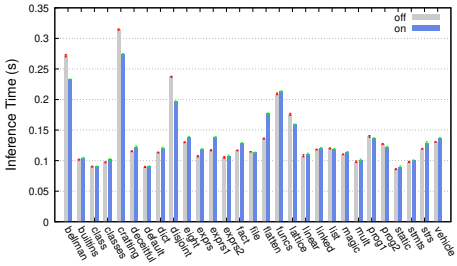
3.4.2 Validating the Effectiveness of InferType's Optimization

We measure and compare the elapsed time of Subpet by disabling and enabling InferType's shape computation in Section 3.3.3 to validate how our shape computation can affect the performance of type inference. Figure 3.2 shows the elapsed time for the programs in our dataset by disabling and enabling InferType's shape computation. Each data column is the average elapsed time of the later 10 runs by looping Subpet 20 times. Column *off* and *on* show the elapsed time when InferType disables and enables the shape computation. When the shape computation is disabled, InferType performs type inference by a straightforward translation to Z3 as shown in Section 3.3.2.

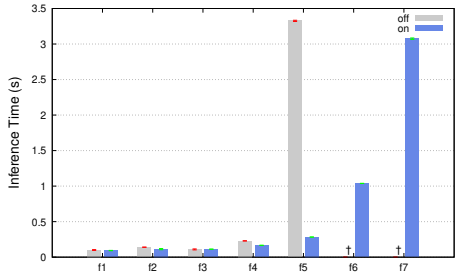
The results from Figure 3.2a show that InferType gives small performance overhead by the shape computation. Figure 3.2a gives the results for part of the programs in Bench. Particularly, extra Z3 formulae are not asserted in the constraint solving when the shape computation is enabled in the programs in Figure 3.2a for evaluating the overhead of the shape computation. Extra Z3 formulae are asserted for all the other programs when the shape computation is enabled in the other sub-figures in Figure 3.2. On average, Subpet degrades the performance of type inference by 0.23% with



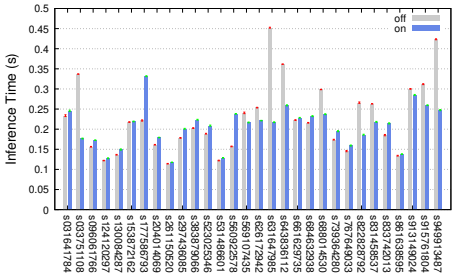
(a) Bench (no extra formulae when shape computation is enabled in *on*)



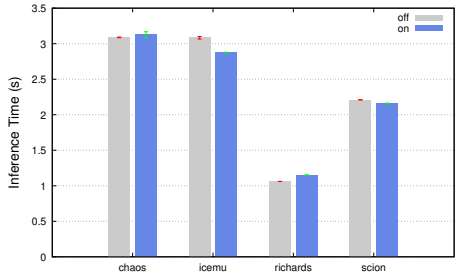
(b) Bench



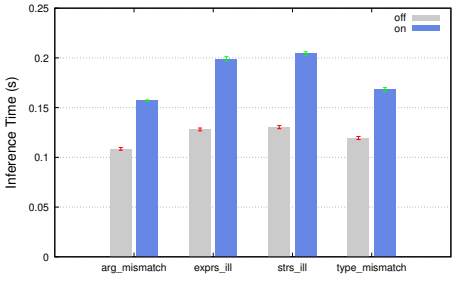
(c) Fs



(d) Codenet



(e) Large



(f) Ill

Figure 3.2: Type inference time by disabling and enabling InferType's shape computation. *off* bars are results by disabling shape computation. *on* bars are results by enabling shape computation. X-axis shows program names. Y-axis shows the elapsed time in second. Each bar is mean of 10 runs with error bars denoting standard errors. †timeout (after 1,000s).

the shape computation than without the shape computation among all the programs in Figure 3.2a. Here and below, the performance rate is calculated as $(t_{\text{off}} - t_{\text{on}})/t_{\text{off}}$ for each program, where t_{off} and t_{on} are the elapsed time when the shape computation is disabled and enabled. This 0.23% is the overhead of InferType's shape computation for computing the shapes.

The results from Figure 3.2c show that InferType's optimization can greatly mitigate the performance bottleneck for constraint solving containing deeply nested types. When the shape computation is enabled, Subpet performs type inference for all the programs in Fs within a reasonable time period. When the shape computation is disabled, Subpet performs much slower for program `f5`; it could not finish program `f6` or `f7` within a time limit. These results are similar to those obtained by Typpete given in Table 3.2b. It indicates that the better performance by Subpet against Typpete is not simply due to the language advantage, but because of InferType's optimization.

Our results demonstrate a tendency that InferType's optimization has a higher potential to improve the performance of type inference for programs containing larger nested types. Subpet with the shape computation outperforms that without the shape computation in 8/29 (27.6%) programs among Figure 3.2b (Bench). Among these 8 programs that Subpet with the shape computation give better performance, it outperforms by an average of 8.87%. For example, program `bellman` (containing a nested list of list type) and `crafting` (containing a nested dict of dict type) in Figure 3.2b have better performance by 16.7% and 15.1%. Subpet with the shape computation outperforms that without the shape computation in 11/30 (36.7%) programs among Figure 3.2d (Codenet). Among these 11 programs, it outperforms by an average of 40.9%. In the best case, it accelerates the performance of type inference by 108.2%. The programs in Codenet have more nested types than programs in Bench since Codenet is collected on purpose by specific keyword search. Since Codenet has bias collected by specific keyword searching, we could not clearly show how frequently nested types are used in real-world Python programs. However, our results provide evidence that at least there are programs using nested types in wild Python programs. InferType is practically useful to help implement type inference engines such as Subpet to potentially handle such programs more efficiently. It is also observable that InferType's optimization would not always give better performance for the constraint solving with nested types. It sometimes degrades the performance even if the programs contain nested types. For example, the optimization degrades the performance of two programs by 49.6% and 51.6% in Figure 3.2d.

Our experiments empirically show the scalability of the shape computation to larger programs. Figure 3.2e presents the time results of type inference for the collected programs/projects in Large. Subpet takes around 2 seconds for type inference on average, which is 10 times longer than the average elapsed time for the programs in Bench and Codenet. InferType's shape computation does not significantly boost or degrade the performance of type inference. These results show that it is practically feasible to apply our shape computation to larger real-world programs.

Figure 3.2f gives the elapsed time of type inference for the ill-typed programs in our dataset. Subpet takes longer time to find type errors in type inference when the shape computation is enabled. This is because InferType launches a second run of constraint solving if it cannot find a binding to satisfy the given type constraints in the first run for handling the limitation of our shape computation. Yet, Subpet could detect the type errors in all the ill-typed programs whenever the shape computation is disabled or enabled.

3.4.3 Assessing the Precision of the Approximate Shape Computation

We assess the precision of our approximate shape computation by counting the number of computed shapes for type variables, compared with an implemented, precise shape computation. Instead of computing only part of the shape equations in the approximate shape computation, the precise shape computation processes all the derived shape equations including disjunctions. It is implemented using Z3 by a straightforward translation from shape equations to Z3 formulae. For example, the shape equations in Listing 3.10 (which are derived from Listing 3.4) are translated to the following Z3 formulae to compute the shapes in the precise shape computation.

```
(= s_f (Q2 s_x s_fx))
(= (Q2 s_f (Q2 s_x s_fx))
   (Q2 (Q2 s_y s_plus) s_e))
(or (and (= s_y Star)
         (= Star Star)
         (= Star s_plus))
    (and (= s_y Star)
         (= Star Star)
         (= Star s_plus)))
```

`s_?` represents translated Z3 constants for shape variables, which ranges

over a Z3 datatype declaration `zshape` for shapes generated as

```
(declare-datatypes () ((zshape
    (Star)
    (Q2 (Q2P1 zshape) (Q2P2 zshape)))))
```

`Star` represents the symbol `*`. `Q2` represents the symbol `?` that takes two arguments, where `Q2P1` and `Q2P2` are generated accessors. The precise shape computation outputs a binding of shape variables to shapes by invoking Z3. InferType then generates and asserts extra Z3 formulae for type variables regarding the computed shapes to optimize constraint solving same as in Section 3.3.3.2. Our approximate shape computation is implemented in Java.

Table 3.3: Average number of computed shapes and average shape computation time. Each data cell shows the results by approximate/precise shape computation.

	Bench	Fs	Codenet	Large	Ill
shape	23/35	17/28	41/69	370/443	17/25
time(ms)	0.91/74.3	0.71/30.4	1.75/434	68.7/2580	0.69/38.9

The approximate shape computation computes more than half of the shapes that the precise shape computation computes on average among all the programs in the dataset. The first row in Table 3.3 shows the average number of computed shapes of all programs in each dataset by the approximate and precise shape computation. The approximate shape computation computes 70.3%, 55.5%, 60.4%, 83.6%, and 68.0% shapes of those by the precise one in each set of programs.

The approximate shape computation is much faster than the precise shape computation. The second row in Table 3.3 lists the average elapsed time of the two shape computations for all programs in each dataset. Same as the measuring before, the elapsed time of the shape computation for each program is the average of the later 10 runs by looping 20 times. The elapsed time of the approximate and precise shape computation differ in scale.

In our dataset, we do not find a program that has shorter elapsed time for type inference with the precise shape computation than the approximate one by Subpet, though the precise shape computation could compute more shapes on average and reduce the search space of more type variables for expected, faster constraint solving. This might suggest that our approximate shape computation is practically useful enough with respect to the

performance, or the programs in our dataset are relatively small so that the precise shape computation could not show its advantage. We could not show clear evidence to support these claims.

Besides, we do not find a case such that Subpet performs a failure of type inference caused by the limitation of the approximate shape computation as we discussed in Section 3.3.3.2. This is because Subpet's syntax-related typing rules during AST traversal do not generate a type constraint like (22) given in Listing 3.11. Excluding a derived shape equation of such a type constraint would let the approximate shape computation imprecisely compute a shape. However, it is possible that our approximate shape computation encounters this limitation when developing other languages whose syntax-related typing rules generate such type constraints.

3.5 Limitation

InferType is designed as a practical toolkit though it is limited in several ways. One of its limitations lies in the scope of its supported type systems. InferType supports type systems that can be described by one partial order [82, 88], with first-order type constraints in propositional logic [8]. Consequently, InferType is fundamentally limited to natively support only a restricted class of type systems.

InferType represents types as finite tree structures consisting of type variables and type constructors. InferType supports common first-order type constructors including array, tuple, and record types. Accordingly, InferType can support the implementation of languages with simple type systems similar to the C language. For instance, suppose that a target language supports array and integer types. By using InferType, a type `array<int>` representing an array of integers is specified as:

```
Type intArrayTy = inf.type("array", intTy);
```

Here, `inf` is an instance of InferType's main class as we introduced in Section 3.3.1. `"array"` is a user-defined name for describing array types. `intTy` is a type object specified as `inf.type("int")` in InferType representing the integer type. A type `array<array<int>>` representing an array of integer arrays is specified as

```
inf.type("array", intArrayTy);
```

A type `array<`t_k`>` is specified as

```
inf.type("array", inf.typevar("t_k"));
```

where t_k is a type variable created by `inf.typevar`. A subtype-related typing rule for array types can be encoded in InferType similar to those we introduced in Section 3.3.1.2.

```
inf.ruleBuilder
    .declare(inf.subtype(
              inf.type("array", t1),
              inf.type("array", t2)))
    .when(inf.subtype(t1, t2));
```

`t1` and `t2` are type variables created by `inf.typevar`. It corresponds to a subtype-related typing rule

$$\frac{t_1 <: t_2}{\texttt{array<}t_1\texttt{>} <: \texttt{array<}t_2\texttt{>}}$$

Assume that arrays in that target language are covariant like arrays in Java.

InferType also supports tuple types for representing products of other types with fixed arities. The compiler writer must assign a distinct constructor name for specifying tuple types with each arity. For example, the type for a 3-tuple `(1, 3, true)` containing two integers and a boolean value is specified as

```
inf.type("tuple3", intTy, intTy, boolTy)
```

`"tuple3"` is a user-defined constructor name for specifying tuple types with 3-arity. `boolTy` is a type object specified as `inf.type("bool")` representing the boolean type. The type for a 2-tuple value is specified as

```
inf.type("tuple2", inf.typevar("t_k"), intArrayTy)
```

where `"tuple2"` is the constructor name for specifying tuple types with 2-arity. The compiler writer must also encode distinct subtype-related typing rules for tuple types with different arities. For instance, the compiler writer must encode two separate subtype-related typing rules

$$\frac{s_1 <: t_1 \qquad s_2 <: t_2}{\texttt{tuple2<}s_1\texttt{, }s_2\texttt{>} <: \texttt{tuple2<}t_1\texttt{, }t_2\texttt{>}}$$

for supporting subtype relations for tuple types with 2-arity and

$$\frac{s_1 <: t_1 \qquad s_2 <: t_2 \qquad s_3 <: t_3}{\texttt{tuple3<}s_1\texttt{, }s_2\texttt{, }s_3\texttt{>} <: \texttt{tuple3<}t_1\texttt{, }t_2\texttt{, }t_3\texttt{>}}$$

for tuple types with 3-arity.

InferType can also be used to specify record types. For nominal record types such as

```
struct point {
    int x;
    int y;
};
```

it can be specified as

```
inf.type("point")
```

InferType treats `point` as an ordinary type constructor. A compiler writer is responsible for maintaining the association between names (`x` and `y`) to corresponding field types for reference by name. For structural record types such as {`x: int, y: int`}, it can be specified as a type object with a user-defined type constructor name

```
inf.type("recordXY", intTy, intTy)
```

InferType does not automatically derive structural subtype relations since `recordXY` is also treated as an ordinary type constructor. A compiler writer is expected to enumerate and encode subtype-related typing rules for record types with different structures. For instance, to support subtype relations such as {`x: int, y: int`} `<:` {`x: float`}, the compiler writer encodes the subtype-related typing rule as

$$\frac{s_1 <: t_1}{\texttt{recordXY<}s_1\texttt{, }s_2\texttt{>} <: \texttt{recordX<}t_1\texttt{>}}$$

where `recordX` is the type constructor specified for record types such as {`x: float`}. The compiler writer must encode another subtype-related typing rule as

$$\frac{s_2 <: t_2}{\texttt{recordXY<}s_1\texttt{, }s_2\texttt{>} <: \texttt{recordY<}t_2\texttt{>}}$$

for supporting subtype relations such as {`x: int, y: int`} `<:` {`y: int`}.

InferType does not natively support more expressive type features such as intersection, union, or recursive types. Supporting those types is desirable for InferType to enhance its practical applicability. However, extending InferType with those types while preserving the benefits of its optimization technique remains an open challenge. It is unclear whether the shape computation could remain effective and efficient under such an extension. The idea of the optimization in InferType is to pre-compute one single finite structure for each type variable to reduce the search space in constraint solving for faster inference. Extending InferType with more expressive types fundamentally complicates this process.

Our optimization by shape computation might be less effective when introducing intersection types. For example, intersection types are used to describe values that satisfy multiple interfaces simultaneously. They are often used to represent overloaded functions by assigning one function with multiple callable types. Suppose that an overloaded function is able to accept both integers and integer arrays and to return a value of the same type in a target language. The type of that overloaded function can be described as an intersection type

`arrow<int, int>` $\cap$ `arrow<array<int>, array<int>>`

where $\cap$ represents intersection types. Suppose that a type constraint is given as

(40) `arrow<int, int>` $\cap$ `arrow<array<int>, array<int>>` `<:` t_p

t_p is a type variable. Since our shape computation is based on the assumption that most subtype relations hold between types with the same structure, a corresponding shape computation is derived as

(40') `?<*, *>` $\cap$ `?<?<*>, ?<*>>` `==` s_p

s_p is the shape variable for t_p. However, in a type system with intersection types, subtype relations are typically defined as

$$\frac{}{t_1 \cap t_2 <: t_1}$$

The left operand type (which is an intersection type) has a different structure from that of the right operand type, which violates our shape assumption. For example, t_p would be inferred as `arrow<int, int>` to make (40) hold. However, s_p would be computed as some shape with intersection from (40'), which has a different structure from type `arrow<int, int>`. Assume that shapes are extended with intersections. As a result, the shape computation might easily encounter the fallback by computing some wrong shapes. As we mentioned in Section 3.3.3.3, the fallback would discard all computed shapes and rerun type inference without any optimization, which makes the overall inference approximately twice as slow as the baseline.

The effect of our optimization might be reduced and the shape computation might consume long time as a pre-process when introducing union types. Union types are used to represent values that may belong to one of several alternative types. They are commonly used to express APIs that accept heterogeneous values without requiring a common supertype. For example, suppose that a function accepts either an integer or an integer array as its argument. Its parameter type can be described as a union type

`int` $\cup$ `array<int>`

where $\cup$ represents union types. Suppose that an argument typed as t_k is passed to that function, and a type constraint specifying that the argument type should be a subtype of the parameter type is collected as

(50) t_k `<:` `int` $\cup$ `array<int>`

The corresponding shape equation is derived as

(50') s_k `==` `*` $\cup$ `?<*>`

s_k is the shape variable for type variable t_k. The shape computation indicates that s_k is either `*` or `?<*>`. Thus, t_k would range over all possible alternative shapes by union instead of one single shape. In this case, t_k would range over all primitive types and all types that take one primitive type argument. The possible number of types that type variable ranges over may remain large. Such a computed shape might not be useful since the search space for that type variable may not be sufficiently reduced. The constraint solving may not become faster by our optimization in real-world cases. Besides, the shape computation itself might no longer be fast as a pre-process since union types introduce choices during decomposition. The number of possible cases would be exponential in general. In the above example, to make the shape equation (50') hold, it must consider both two possible shapes for s_k to check if either of them can satisfy other shape equations. A possible workaround might be to rely on SMT [6] for resolving such complex shape computation. However, we have no direct evidence to show whether that approach would be practical to preserve the efficiency.

The effect of our optimization is unclear and the shape computation becomes more challenging when introducing recursive types. Recursive types allow types to be defined in terms of themselves. They are often used to describe recursively defined data structures such as linked lists, trees, and streams. For example, the type of integer lists can be described as a recursive type $\mu X.unit \oplus (int \otimes X)$, where μ introduces a recursive type. It represents that a list is either empty or consists of an integer together with another list. Extending shapes with recursive representations would lead to redesigning the shape computation to reason about cyclic structures. InferType's optimization is based on the idea of pre-computing a finite structure to reduce the search space before constraint solving. Although recursive types can be represented by finite cyclic graphs, it is unclear how shapes

[6] We demonstrated such a translation in Section 3.4.3 though it is for the current shape computation.

extended with recursion could be exploited to benefit the constraint solving for faster inference.

Besides, InferType only supports type systems in which type variables abstract over concrete types rather than type constructors. For instance, InferType does not support languages like Haskell or Scala. Their type systems support specifying type constraints such as `f<int> <: g<int>`, where `f` and `g` are type variables that range over type constructors such as `maybe` [71, 46]. Such type constraints sit strictly outside a first-order type relation that InferType supports. In addition, InferType does not support a type system that is expressed by more than one type relation. An example is Choral [34] for choreographic programming, whose type system would be expressed by a subtype relation and also a type relation for describing distributed locations.

The restricted expressiveness of types is present since automation itself poses profound challenges in performance. Inferring simple types may be time-consuming since the performance of a standard algorithm is bound to cubic time complexity [82] even with manual implementation. One main focus of InferType is to deliver efficient code for type inference by an SMT solver for practical usage. Although the design of InferType is simple and the scope of its supported type systems is restricted, we believe that the scientific contribution of InferType is clear in automating type inference while maintaining acceptable performance.

Despite the restrictions in its expressiveness, InferType remains practical for applications that provide type hints for other downstream tasks. As a supporting toolkit, InferType is not good at formalizing fundamental verifications or implementing proof assistants of performing type checking. Instead, it is well-suited for implementing engines that provide type hints in a conservative way within a subset of a language. When encountering complex or dynamic features outside its supported range, that engine just returns a default, uninformative value and the code will fall back as unoptimized in the downstream pipeline. An example is to infer an additional type annotation for a code fragment while leaving it as dynamically typed if that code fragment is not statically typable in a gradually typed language. We will present our study on leveraging InferType in Chapter 4.

InferType can also be practically adapted to implement languages with richer types though these types may not be natively supported. For instance, a common workaround for a compiler writer is to use InferType with some manual effort to implement a language with parametric polymorphism such as Generic Java [43] and C++ templates. It is also viable to adapt similar manual effort to describe flexible overloading in languages like JavaScript and Python. A compiler writer can explicitly instantiate a fresh type vari-

able for each function call of a parameterized function. InferType is then capable of finding valid bindings for those instantiated type variables to perform type inference.

Another primary limitation of InferType is its lack of support for principal typing. In standard type theory, a type inference engine seeks the most general type that subsumes all possible valid types for a program fragment [80]. InferType prioritizes performance by leveraging an SMT solver and also our developed technique to produce only one valid binding of inferred types as the result of type inference. For instance, for a polymorphic function that simply returns whatever argument it receives, a type inference engine supporting principal typing would infer its type as `arrow<a, a>`, where `a` is a universally quantified type variable that ranges over all types. A type inference engine that straightforwardly uses InferType would infer one concrete type such as `arrow<int, int>` for that function.

One single valid binding is sufficient for several downstream tasks such as generating specialized machine code using concrete type annotations. However, it is not ideal for use cases such as generating type signatures for library functions or modular type inference [45]. A compiler with modular type inference is able to perform separate compilation. When a large program is divided into several modules, each of the modules can be compiled with type inference in isolation. A type inference engine implemented by InferType is used to infer types for the whole program in each compilation. Nevertheless, InferType is aimed at supporting the practical implementation of type inference engines. Our experiments are conducted to provide evidence that the proposed optimization technique is able to improve the performance of type inference in practical cases. We do not have direct evidence to show whether the proposed optimization technique can benefit a type inference engine supporting principal typing.

3.6 Related Work

Language development is enjoying significant growth in number and diversity by both academia and industry. Language workbenches, such as MPS [106] and Xtext [31], are development environments that provide high-level mechanisms for implementing domain-specific languages [33]. The Spoofax [49] language workbench allows language developers to write declarative specifications of language definitions to produce parsers, interpreters, and editor plugins. Efftinge et al. [28] presented a Java framework Xbase for implementing DSLs based on Xtext. Recent studies proposed a stan-

dardization of name resolution and type checking with a constraint-based approach integrated in Spoofax [4, 3]. Unlike existing language development toolkits such as Spoofax, our proposed tool produces constraint-based type inference supporting subtyping.

Specifying and implementing type inference using constraints is an established approach. A number of existing type inference systems adopt constraint-based approaches because of the modularity of separating constraint generation and solving for providing high flexibility and extensibility [80, 93, 92, 50, 76]. Our proposed tool adopts the constraint-based type inference for similar reasons to support various type systems of handling type inequalities such as subtyping. Besides the constraint-based approaches, one of the most traditional and influential type inference approaches is the Hindley-Milner type inference [42, 66, 20]. The Hindley-Milner type inference targets one particular type system and infers types by unification as the heart of its algorithm. Another approach is the bidirectional type checking [81, 74], which is regarded as local type inference, developed by carefully controlling the introduction and elimination of type variables for inferring parameter types. Turnstile [16] is a meta-language for creating typed languages supporting bidirectional type checking. Compared with Turnstile, InferType is designed for languages supporting global type inference by the constraint-based approach. Jones et al. introduced wobbly types [79] to distribute over type constructors for handling GADT type inference using type annotations. Later, Pottier et al. [83] proposed a stratified type inference with a pre-process of inferring shapes for propagating type annotations by local type inference. The idea of a two-strata type inference and the computation for shapes in this research is similar to those research works [83, 108]. However, our shapes are automatically computed from the given type constraints without extra information like type annotations, and the computed shapes are used for optimizing constraint solving.

Satisfiability Modulo Theories (SMT) is an area of automated deduction for checking the satisfiability of first-order formulae with respect to logical theories [8, 7]. State-of-the-art SMT solvers include Yices [26], CVC5 [6], and Z3 [22]. InferType currently relies on Z3 to perform constraint solving for type inference. Translating InferType expressions to a higher-level language such as JavaSMT [48] can be a possible extension for enabling different SMT solvers. There have been works on improving the performance of SMT solvers in general such as pruning the search space by detecting symmetries in the input formulae [23]. Later, Niemetz et al. [73] presented an approach for accelerating quantified constraint solving. The developed optimization in our proposal is a domain-specific approach to reducing the

search space of type variables in SMT solving based on InferType's shape computation.

SMT solving has been a critical part of several static analyses including automatic type inference. Swamy et al. [99] presented the language F* with a dependent type and effect system using a combination of SMT solving and manual proofs. Vazou et al. [102] introduced LiquidHaskell, which is a refinement type-based verifier for Haskell using SMT solvers. InferType does not directly support such type systems. However, a compiler writer can handle complex types such as generics by manually resolving them (instantiating fresh type variables) before encoding them into InferType, though InferType would compute a concrete type instead of a generic type as the inferred type. Pavlinovic et al. [77] presented an encoding of the OCaml type system to a weighted MaxSMT problem for localizing type errors when the type inference fails. InferType considers type errors by providing the type variables in a minimal set of unsat type constraints for helping locate ill-typed program expressions, while the generation of a text message is left to the compiler writer. Generating and customizing type error messages by InferType is treated as our future work. Hassan et al. [39] proposed a type inference engine Typpete that generates type annotations in Python by encoding type constraints as a MaxSMT problem using Z3. In this study, we implemented a subset of Typpete by using the proposed toolkit for the experiments. There are also emerging studies of statically typing Python programs based on other techniques [57, 35, 69, 78].

3.7 Summary

We presented InferType, a Java library for implementing constraint-based type inference. InferType performs constraint solving by translation to the Z3 SMT solver. Because the constraint solving in SMT may be exponentially slow by the increasing search space for large nested types, we develop an optimization technique in InferType to alleviate the performance bottleneck for improving its practical usability. InferType pre-computes a structure of a type variable and reduces the search space of that type variable based on the pre-computed structure.

We experimented the effectiveness of InferType's optimization by implementing a type inference engine for a Python subset using InferType. We find that InferType introduces acceptable overhead by our developed preprocess for normal and small programs. InferType can significantly improve the performance of type inference for programs with deeply nested types,

and potentially improve the performance for programs containing nested types.

Chapter 4

Efficient Selection of Type Annotations for Performance Improvement in Gradual Typing

In the previous chapter, we have established InferType as a toolkit for implementing constraint-based type inference. However, a fundamental question remains: can InferType, as a generative toolkit, compete with manually implemented engines in developing compilers and related techniques? Without specialized optimization techniques, generated code including that for type inference is inherently less performant compared with manually tuned implementations in general. This performance gap often discourages the use of generative tools in both developing compilers and implementing prototypes. Straightforwardly generating code for type inference or using an existing type inference engine might be ill-suited since it might consume too much inference time to enable other techniques in ahead-of-time compilation or translation.

In this chapter, we demonstrate the usage of InferType in a study related to gradual typing. Our goal is to present evidence to indicate that InferType can produce efficient type inference by its code generation and leave sufficient computational room for enabling further techniques within a compilation pipeline.

Gradual typing has gained popularity as a design choice for integrating static and dynamic typing within a single language. Several practical languages have adopted gradual typing to offer programmers the flexibility

to annotate their programs as needed. Meanwhile, there is a key challenge of unexpected performance degradation in partially typed programs. The execution speed may significantly decrease when simply adding more type annotations. Prior studies have investigated strategies of selectively adding type annotations for better performance. However, they are restricted in substantial compilation time, which impedes the practical usage. This chapter presents a new technique to select a subset of type annotations derived by type inference for improving the execution performance of gradually typed programs. The advantage is its shorter analysis time by employing a lightweight, amortized approach. It selects type annotations along the data flows, which is expected to avoid expensive runtime casts caused by a value repeatedly crossing the boundaries between untyped and typed code. We implement our technique as a tool named TypePycker by extending Reticulated Python, and conduct experiments to validate its effectiveness in improving the execution time on Reticulated Python. Our extension includes the usage of InferType for inferring additional type annotations. Experiment results show that TypePycker outperforms a naive strategy of using all type annotations derived by type inference among the benchmark programs. In comparison with existing approaches and engines, our tool enabled by InferType shows an advantage in maintaining more stable analysis time of deriving and selecting type annotations. Our results empirically indicate that the proposed technique is practical within Reticulated Python for mitigating the performance bottleneck of gradually typed programs.

The rest of the chapter is structured as follows. Section 4.2 motivates the reader by introducing the unexpected performance degradation when simply adding more type annotations to gradually typed programs. Section 4.3 presents our proposed technique for improving the execution performance of gradually typed programs, which selectively adds additional type annotations by type inference. Section 4.4 describes our experiments, which compare our implementation to existing engines and also evaluate the effectiveness of the proposed optimization. Section 4.5 discusses the potential factors that would threaten the validity of our study. Section 4.6 reviews related literature, ranging from static type inference to other techniques that improve the execution performance of gradually typed programs. A brief summary ends this chapter.

4.1 Introduction

Gradual typing offers a flexible bridge between dynamically and statically typed languages. It also introduces a critical challenge of execution degradation. In a gradually typed language, programmers can add type annotations in their programs to use static typing as needed. To ensure the soundness so that a variable annotated with a type annotation T must only be assigned values of type T, runtime casts must be performed at the boundaries between dynamically (untyped) and statically typed code. The execution degradation is often caused by these runtime casts, which check and raise an error unless values have expected types. Runtime casts are enforced in sound gradually typed languages with non-erasure semantics [36, 19]. Sound gradual typing has appealed to several practical languages including Racket [101, 53], Python [105], and ActionScript [87].

A commonly used approach to improving the execution performance is to automatically add additional type annotations by type inference so that more variables will be statically typed. However, naively adding all type annotations suggested by type inference without careful consideration may unexpectedly cause execution degradation. This is because adding some type annotations may cause values repeatedly crossing the boundaries between untyped and typed code, which would introduce extra runtime casts. For example, when an untyped value is assigned to a variable v and v is an argument to an untyped function parameter, adding an additional type annotation to v would let a value cross the boundaries from untyped to typed, and then back to untyped. Another commonly used approach is to leverage Just-In-Time (JIT) compilation to avoid redundant runtime casts at runtime. While JIT has been an effective approach for performance improvement in gradual typing [9, 89, 90], its applicability is limited for systems with sufficient resources since JIT requires additional memory by dynamic compilation. For example, microcontrollers such as MicroPython [21] and other embedded systems often lack JIT support due to their constrained memory.

Research studies have revealed that it does not reliably lead to performance improvement by naively adding more type annotations. In particular, Takikawa et al. report non-negligible performance degradation when a program is partially typed [100]. Their work shows that unless a program is entirely statically typed, there would be significant execution slowdown compared with the untyped program such as more than 100 times even if the program is mostly typed. The performance pitfall discourages the usage of gradual typing since an easy workaround is to just remove all type

annotations.

A few existing research studies have explored approaches to select added type annotations from the ones derived by type inference for improving the performance of gradually typed programs. However, they may incur non-negligible compilation time for deriving and selecting type annotations. For instance, we observe that one existing tool Herder [12] would incur a significant amount of compilation time such as more than 10 minutes for several benchmark programs, though they are relatively small in lines of code compared with real-world programs. It remains challenging to select effective type annotations for improving the performance considering practical usage.

We propose a lightweight method, named *TypePycker*, to selectively add type annotations for improving the execution time of gradually typed programs. The novelty of TypePycker compared with existing approaches is its lightweight method that achieves comparable execution performance but does not require long compilation time. Given a program, it performs type inference and selects only a subset of the type annotations derived by the type inference. To avoid significant costs of runtime casts caused by values repeatedly crossing the boundaries between untyped and typed code, TypePycker uses a quick, amortized approach to select type annotations placed along the data flows.

To evaluate the effectiveness of the proposed method, we conduct experiments on a set of collected programs in a gradual typing language Reticulated Python [105]. Our experiment results show that the proposed method achieves performance improvement compared with naively using all inferred types derived by type inference. Across 41 collected benchmark programs, our method outperforms using all inferred type annotations in 32 programs. It achieves a speedup up to more than 5x compared with a program appended with all inferred type annotations. Our method gives comparable execution time against using all inferred type annotations in 6 programs. For assessing the compilation time, we compare TypePycker with the existing tool Herder [12]. We observe that TypePycker maintains more stable and acceptable compilation time, and also achieves comparable execution time across the benchmark programs. We also find that TypePycker offers clear advantages in several programs such that it maintains much faster compilation time than Herder while at the same time our selection gives better performance than using all inferred types. Our findings show that the proposal is a practical method to mitigate the performance degradation of gradually typed programs in Reticulated Python.

4.2 Execution Slowdown in Gradual Typing

Predicting the execution performance of gradually typed programs is challenging. The reason is that a gradually typed language would perform runtime casts when a value crosses the boundary between untyped and typed code. This often causes serious degradation of execution performance. A runtime cast is an operation executed at runtime to check if a value has an expected type. Runtime casts are executed for ensuring type invariants specified by the program [87, 96, 13]. A number of gradually typed languages perform runtime casts for sound static types [70, 105, 53]. Several research studies have observed significant runtime overhead caused by runtime casts where a partially typed program is more than 100 times slower than its untyped version [100, 38].

Consider an example program in a gradually typed language given in Listing 4.1. A local variable `z` is annotated with a type annotation by the programmer. A local variable, function parameter, or return type that without a type annotation is implicitly typed as the unknown type `*`. We assume that `succ` is a library function typed as `Function([*], *)`, which denotes a function type with a parameter type `*` and return type `*`.

Listing 4.1: An example program. `succ` is assumed to be typed as `Function([*], *)`. `*` denotes the unknown type.

```
1 def f(x, y):
2     u = x
3     z: Bool = y
4     return if z then succ(x) else succ(u)
5
6 f(succ(1), true)
```

The runtime casts for the example program are presented in Listing 4.2. Note that the language implicitly gives static types to literals such as `1` and `true`. This makes the program partially typed and involves runtime casts even when the program does not include any explicit type annotations. For illustration purposes, we use the notation $\{T_2 \Leftarrow T_1\}e$ to explicitly show the occurrence of a runtime cast. It reads as expression e has type T_1 at compile time while e is expected to have type T_2 at runtime. For example, a runtime cast $\{\texttt{Bool} \Leftarrow \texttt{*}\}\texttt{y}$ occurs at line 12 since `y` is implicitly typed as `*` but it is assigned to a local variable typed as `Bool`. This runtime cast is performed at runtime to check whether `y` is a boolean value, which causes runtime penalty.

Listing 4.2: Example program with runtime casts. `succ` is assumed to be typed as `Function([*], *)`.

```
10 def f(x, y):
11     u = x
12     z: Bool = {Bool ⇐ *}y
13     return if z then succ(x) else succ(u)
14
15 f(succ({* ⇐ Int}1), {* ⇐ Bool}true)
```

A widely used approach to mitigating the execution slowdown due to runtime casts in gradual typing is to automatically add additional type annotations derived by type inference so that a number of runtime casts will be eliminated. Suppose that a type inference engine is run to derive additional type annotations for local variables, function parameter, and return types for the example program in Listing 4.1. It infers the type of parameter `y` as type `Bool`. This additional type annotation is expected to improve the execution speed since it can eliminate a runtime cast `{* ⇐ Bool}y` at line 15 and a runtime cast `{Bool ⇐ *}y` at line 12 shown in Listing 4.2.

However, adding type annotations as recommended by the type inference engine, without careful consideration, may lead to performance degradation that contradicts programmers' intuitive expectations. Suppose that an additional type annotation `Int` is added to parameter `x` in Listing 4.1. The runtime casts would occur as presented in Listing 4.3. Compared with the runtime casts presented in Listing 4.2, three extra runtime casts occur since a value repeatedly crosses the boundaries from untyped code `succ(1)`, to typed code `x`, then back to untyped code `u`, and the argument in `succ(x)`. Recall that `succ` is assumed to be typed as `Function([*], *)`.

Listing 4.3: Typed example program with runtime casts.

```
def f(x: Int, y):
    u = {* ⇐ Int}x
    z: Bool = {Bool ⇐ *}y
    return if z then succ({* ⇐ Int}x) else succ(u)

f({Int ⇐ *}succ({* ⇐ Int}1), {* ⇐ Bool}true)
```

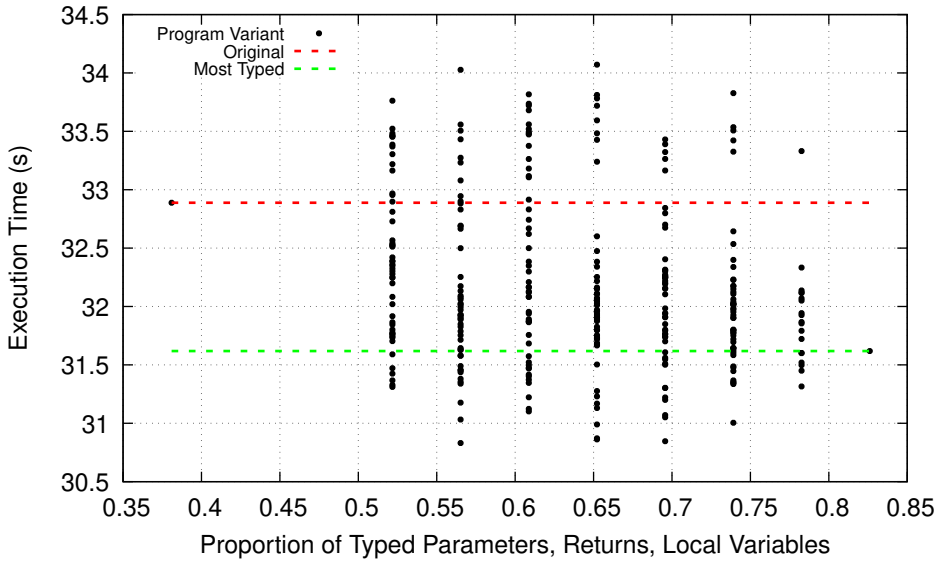
The unexpected performance degradation caused by excessive type annotations is well-known by existing studies. Takikawa et al.'s seminal paper [100], for instance, shows that unless a program is entirely statically typed, the execution time may be slow even if the program is mostly typed. We also observe that simply adding more additional type annotations de-

rived by type inference does not necessarily lead to better performance. We conduct similar experiments to those by Takikawa et al. as our preliminary study. In our preliminary study, we measure the execution time of several public Python programs annotated with different numbers of type annotations on Reticulated Python [105]. Reticulated Python is a gradually typed dialect of Python that allows programmers to annotate programs with static types. It performs runtime casts for sound static types. Given a Python program, we give a set of type annotations for its local variables, function parameters, and return types. Those type annotations are derived by an external type inference engine. Note that we modify Reticulated Python to support static types for local variables. The original Reticulated Python does not support explicit type annotations for local variables. Then we generate variants of the program annotated with random subsets of the type annotations. We measure the execution time of the program with zero number of type annotations, the program including all the type annotations derived by type inference, and the generated program variants with different subsets of those type annotations.

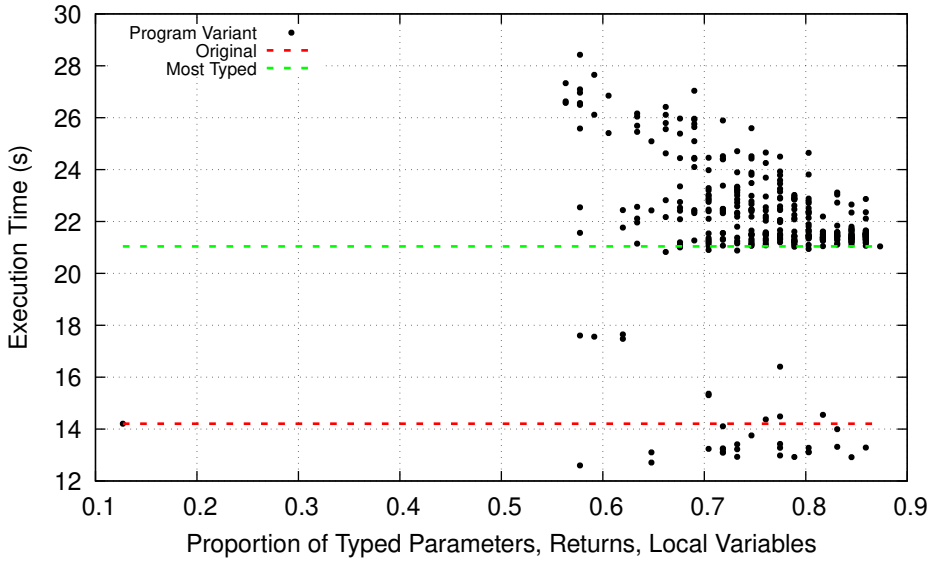
Figure 4.1 demonstrates the results for two of the tested programs [1]. For the program in Figure 4.1a, the program including all the type annotations derived by type inference gives better performance than the program with zero number of type annotations. This showcases that adding type annotations can improve the performance. However, it is also observed that the performance can be further improved with fewer type annotations. For the program in Figure 4.1b, although type inference generates type annotations for more than 80% of variables and others, the program that includes all of these annotations performs worse than the program without any annotations. The execution time can be improved by discarding some of the type annotations.

An interesting research direction is to develop techniques for selecting appropriate type annotations, from those derived by type inference, that can improve execution performance. However, existing approaches have limitations. One study [12] suggests appropriate type annotations by analyzing the overhead of runtime casts with different subsets of possible type annotations. A drawback of this approach is that it consumes extremely long compilation time for several benchmark programs including real-world ones. In this study, the compilation time means the time for deriving additional type annotations by type inference and inserting selected type annotations into the original program. Another recent study [51] tries to tackle this

[1] All the other results are included in our artifact.



(a) storage



(b) nbody

Figure 4.1: Execution time for Python program variants. Each dot is a generated program variant with a different subset of type annotations. X-axis shows the proportion of typed function parameters, returns, and local variables. Y-axis shows execution time in seconds (lower is better). The red dashed line represents the program with zero number of type annotations. The green dashed line represents the program including all the type annotations derived by type inference.

problem by predicting the optimal execution performance with a machine-learning based method. However, it requires the execution of a number of programs with different subsets of type annotations for training. As a result, it remains challenging to select candidates of type annotations considering practical usage.

4.3 Efficiently Selecting Added Type Annotations

We propose a new lightweight method for improving the execution time of gradually typed programs by selectively adding type annotations. Our method first performs type inference and selects only a subset of the type annotations derived by the type inference so that the program including the selected type annotations will run faster than the program before the type annotations are added. Unlike previous methods, our method is lightweight and hence does not require long selection time. Despite being lightweight, our method shows comparable performance improvements, and in some cases, even outperforms an existing tool, according to our experiments.

Our key idea is to select type annotations based on their positions on the data flows. The costs of runtime casts tend to increase when a value is passed from a typed variable to an untyped (or unknown-type) variable, and then back to a typed variable, repeatedly. Thus, our method selects the inferred type annotation for a variable v when type inference infers the types for all the other untyped variables (and function calls) that can reach v along the data flows. Our method quickly checks this condition using a lightweight, amortized approach. It does not repeatedly perform expensive traversal through data flows, nor enumerate all possible occurrences of runtime casts in a brute-force manner.

4.3.1 SimpliPy

To present the proposed method, we begin by defining a small gradually typed language called *SimpliPy*. SimpliPy serves as a simplified, illustrative version of Python. We will use SimpliPy to present the proposal for simplicity.

The syntax of SimpliPy is given in Listing 4.4. A program is a sequence of statements, and an expression supports addition, arrays, conditional, and function application. `[expr, ...]` and `(expr, ...)` represent a sequence consisting of zero or more instances of `expr`, separated by commas and surrounded with brackets `[]` and `()`, respectively. `[expr, ...]` is an array literal. A type is either boolean, integer, function, array, or unknown. The

unknown type is denoted by `*`. An expression of the unknown type is often called *an untyped expression*. In SimpliPy, programmers may not omit a type annotation. Instead, they explicitly specify the unknown type using the type annotation `: *`. When a concrete type annotation `: Int` is substituted for `: *`, however, we below refer to this as adding or appending a type annotation to an untyped variable or parameter for the sake of readability.

Listing 4.4: Syntax of SimpliPy.

```
prog ::= stmt
stmt ::= expr
       | returnStmt
       | defStmt
       | var = expr
       | expr[expr] = expr
       | stmt; stmt
returnStmt ::= return expr
defStmt ::= def id(param, ...) -> T: stmt
var, param ::= id: T
expr ::= expr + expr
       | expr(expr, ...)
       | if expr then expr else expr
       | [expr, ...]
       | expr[expr]
       | id
       | number
       | true | false

T ::= *
    | Bool
    | Int
    | Array(T)
    | Function([T, ...], T)
```

4.3.2 Graph Construction

The proposed method first constructs a directed graph for a given SimpliPy program. This graph represents data flows potentially affecting runtime casts for gradual typing. Our graph is statically constructed based on the syntax of a given program. It also assumes that a type inference engine generates inferred types for the given program.

A vertex represents a variable, a function parameter, a function name, a literal, or an expression. When an expression contains sub-expressions, all the expressions are represented by vertices. For example, an expression `x + y` generates three vertices for `x`, `y`, and `x + y`. An expression `a[i]` generates three vertices for `a`, `i`, and `a[i]`.

Each distinct variable, parameter, or function name, corresponds to one single vertex, regardless of the number of occurrences within the program. Their different occurrences are represented by one single vertex if they are bound to the same variable, parameter, or function name in the program's name scope determined by static name resolution. Furthermore, every function definition generates a vertex for representing a `return` statement. A function definition generates a single vertex even if it contains multiple `return` statements.

Each vertex has two properties. One property is a given type. We represent the given type for a vertex v as *given(v)*. If a vertex represents a variable or a function parameter, its given type is the type given by a type annotation written by programmers. For example, for `x: Int = 3`, the given type for the vertex representing `x` is `Int`. For `x: * = 3`, the given type for the vertex representing `x` is `*`. If a vertex represents a return statement, its given type is the return type given by the type annotation for its function definition. If a vertex represents a function name, its given type is the function type given by its type annotations. If a vertex represents a number or boolean literal, then its given type is `Int` and `Bool`, respectively. If a vertex represents an empty array literal `[]`, its given type is `Array(*)`. If a vertex represents a non-empty array literal such as `[x, y]`, its given type is `Array(*)`. The given types of the other vertices are `*`. Note that some vertices, such as ones for number literals, are not associated with type annotations written by programmers, yet their given types are not `*`.

The other property of a vertex is an inferred type. We represent the inferred type for a vertex v as *infer(v)*. We assume that an external type inference engine is run and the resulting inferred types are reflected on the graph. For example, when the type of an expression `f(x)` is inferred as type `Int`, then `Int` is the inferred type for the vertex representing that function application. The type inference engine may infer the types of other kinds of expressions. These types are used as the inferred types of the vertices corresponding to those expressions. If the type inference engine does not infer any type, then unknown type `*` is considered as the inferred type. The inferred type for the vertex representing a `return` statement is the type inferred as the return type of its function definition. The inferred type for the vertex representing a function name is the function type inferred for

that function. The inferred types for number and boolean literals are their value types, `Int` and `Bool`. We assume that *infer(v)* is more concrete than *given(v)*, meaning that at least one occurrence of `*` in *given(v)*, if existing, is replaced with a concrete type in *infer(v)*, as a type inference engine will consider *given(v)* when computing *infer(v)*. For example, if a type inference engine could not derive additional type information for v, then *infer(v)* would be equivalent to *given(v)*.

A directed edge is created to represent a data flow in a program. Figure 4.2 presents how directed edges are created. The edge creation largely follows standard practices in data flow analysis. For a function application $c(a_1, a_2, ...)$, edges are created based on the potential callee functions that c may refer to. To discover a callee function, we run an arbitrary points-to analysis. When the analysis discovers multiple potential callee functions, edges are created between the function application and every potential callee function. When the analysis does not discover any callee function, no edge is created. The points-to analysis does not need to be highly precise. Imprecise analysis would simply result in excessive type annotations, which could potentially degrade runtime performance.

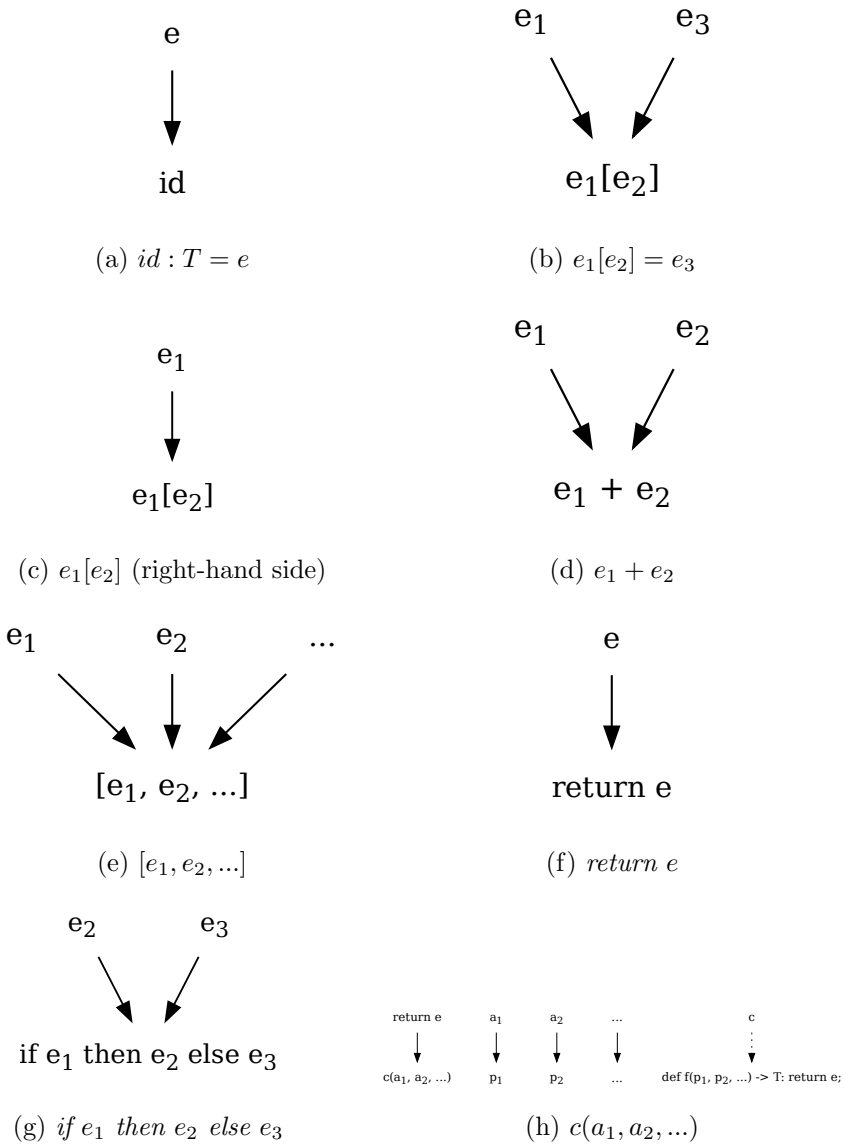
4.3.3 Selecting Inferred Types

After the graph construction, the proposed method determines whether an inferred type is appended to the original source code as a type annotation.

First, our proposed method selects the vertices v that represent either a variable, a parameter, or a `return` statement for which *given(v)* is a type containing `*`. A type contains `*` if it is `*`, an array type containing elements of such a type, or a function type including such a type as a parameter type or a return type. `Array(*)` and `Function([*],Int)` are examples. These selected vertices v are candidates that new type annotations are appended to, based on *infer(v)*. Recall that we assume that *infer(v)* is always more concrete than *given(v)*.

Then, for each of the selected vertices v, it enumerates all the *closest source vertices* reachable to v. If no closest source vertex exists or all the closest source vertices w hold that *given(w)* is not `*`, then *infer(v)* is appended as a type annotation for v. Here, a source vertex is a vertex without an incoming edge or a vertex w holding that *given(w)* does not contain `*`. A source vertex w is a closest source vertex reachable to v if there exists a path from w to v and that path does not include a source vertex except for v and w ($w \neq v$).

Consider the example program in Listing 4.1. First, vertices representing



(a) $id : T = e$ (b) $e_1[e_2] = e_3$

(c) $e_1[e_2]$ (right-hand side) (d) $e_1 + e_2$

(e) $[e_1, e_2, ...]$ (f) *return* e

(g) *if* e_1 *then* e_2 *else* e_3 (h) $c(a_1, a_2, ...)$

Figure 4.2: Edge creation. Solid arrows denote graph edges. Dashed arrows indicate callee functions discovered via a points-to analysis.

variable `u`, parameters `x` and `y`, and the return statement of function `f` are selected as candidates. The vertex representing variable `z` is not selected as a candidate since its given type is `Bool`, which does not contain `*`. For the vertex representing `y`, its closest source vertex is the vertex representing `true`. The inferred type for `y` is appended as a type annotation since the given type for `true` is not `*`. For the vertices representing `x` and `u`, their closest source vertex is the vertex representing `succ(1)`. For the vertex representing the return statement of function `f`, its closest source vertices are vertices representing `succ(x)` and `succ(u)`. The inferred types for `x`, `u` and the return type of function `f` are not appended as additional type annotations since the given types for the function applications of `succ` are `*`. Recall that function `succ` is typed as `Function([*], *)`.

4.3.4 Correctness

The correctness of the proposed selection method is guaranteed by a correct type inference engine and the underlying gradually typed language. The correctness is defined as: an optimized program with additional type annotations must either produce the same value or fail at the same location as the original program [87]. This correctness follows the *gradual guarantee* [96] of a gradually typed language, that is, a gradually typed program is still well-typed or correctly executed when discarding some of its type annotations. Since our method only appends a type annotation that is either *infer(v)* or *given(v)*, a selection of the additional type annotations is equivalent to discarding some type annotations of a gradually typed program whose type annotations are *infer(v)*. If the type inference engine only derives correct additional type annotations, then a selection of those type annotations is also guaranteed to be correct.

4.4 Experiment

We conduct experiments on a set of collected benchmark programs using a gradually typed language Reticulated Python. The goal of our experiments is to verify whether our method can improve the execution time of gradually typed programs while maintaining stable and acceptable compilation time of deriving and selecting additional type annotations enabled by the fast inference of InferType.

In the following experiments, we first present that our method improves the execution time in Section 4.4.2. We compare the execution time of each benchmark program appended with all additional type annotations by type

inference, against that appended with only selected type annotations by our method. We then provide evidence to show whether InferType serves as an enabler for the proposed method in generating performant code of type inference. In Section 4.4.3, we compare the compilation time of deriving and selecting additional type annotations that are inferred by InferType against by other state-of-the-art type inference engines. We also show that our method enabled by InferType achieves more stable compilation time than an existing work Herder [12]. In the later subsections, we assess whether our method is effective under different environments. In Section 4.4.4, we demonstrate that our method provides performance gains across different implementations of Reticulated Python. In Section 4.4.5, we test several benchmark programs on Reticulated Python with PyPy to provide hints whether our method is possibly beneficial to a gradually typed language with JIT compilation.

4.4.1 Implementation and Dataset

Our configuration for the experiments consists of a tool that appends type annotations to a program, and an extension over Reticulated Python for running the program with appended type annotations. We name the tool as *TypePycker*. To derive additional type annotations, we implement a constraint-based type inference engine [80] for a Python subset using InferType as presented in Chapter 3. Our engine supports inferring types for functions, local variables, class methods, and class fields. Given a Python program, our type inference engine assigns a unique type variable to each expression, and collects type constraints over types and type variables by traversing the AST of the program. It then computes a binding that associates type variables to their inferred types by solving the collected type constraints. InferType invokes an SMT solver for computing that binding. We omit a detailed description of the adapted type system for brevity. Comprehensive details on the typing rules can be found in existing studies [39, 50].

We implement the proposed method on a Python subset. We extend the presented techniques in SimpliPy to Reticulated Python by following common data flow analysis on a graph-based representation. For example, the graph construction for other binary operations in Reticulated Python is similar to that presented in SimpliPy. To identify potential callee functions of function applications in the graph construction, we adopt a context-insensitive, flow-insensitive points-to analysis. The analysis approximates a set of callee functions for each function application without considering

conditional branches or the order of execution. For example, consider the following code fragment:

```
1 e = f
2 e(true)
3 e = if cond then h else g
4 e(9)
```

where `f`, `g`, and `h` are identifiers of three function definitions. The analysis would compute that `e` at both line 2 and line 4 may refer to either function `f`, `g`, or `h` though `true` is never passed to `h` or `g`, and `9` is never passed to `f` during execution. It would create edges from arguments `true` and `9` to the parameters of `f`, `g`, and `h` in the graph construction as we described in Section 4.3. Despite its simplicity and imprecision, the adopted points-to analysis is practical for improving the performance among the benchmark programs in our experiments.

Our implementation extends Reticulated Python with two main modifications. First, we adapt a technique from a recent study [13]. The goal of adapting this technique is to preserve program behaviors while maximizing optimization opportunities. Preserving program behaviors here means that optimized programs must either produce the same value or fail at the same location as the original program [87]. The main idea is to generate an optimized version of each function appended with additional type annotations while also keeping the original function. It then dispatches function applications based on the static types of arguments and parameters. For example, the example program in Listing 4.1 with additional type annotations will be generated as shown in Listing 4.5. An optimized function called `f_fast` is generated. The original function is also kept in the source code. In the main part of program, the optimized function is called because the static types of the arguments are subtype of the additional parameter types. Below we refer to this technique as *fast-slow*.

Listing 4.5: Example program with a generated fast function.

```
def f_fast(x: *, y: Bool):
    u = x
    z: Bool = y
    return if z then succ(x) else succ(u)

def f(x, y):
    u = x
    z: Bool = y
```

```
    return if z then succ(x) else succ(u)

f_fast(succ(1), true)
```

Second, we extend Reticulated Python to support type annotations for local variables. This enables us to select additional type annotations for local variables as we described in SimpliPy. The original Reticulated Python does not support explicit type annotations for local variables.

We build a dataset for the experiments. We make it publicly available in our artifact. The programs in the dataset are not annotated with explicit type annotations for function parameters, function returns, or local variables by programmers. Note that although we use those un-annotated programs as baseline in the experiments, the programs are not entirely untyped. The programs are partially typed since the underlying language Reticulated Python always gives static types to several expressions such as number and array literals. This partial typing introduces non-trivial scenarios with runtime casts where there are opportunities to append or discard additional type annotations for improving the execution performance. Thus, although the programs are not ideally annotated with explicit type annotations by programmers, our dataset is qualified as a test bed for experimenting if the proposal can improve the execution performance.

The dataset contains 50 Python programs collected from several public sources. 31 programs are collected from several academic studies [12, 62, 59]. Those programs include a subset of the Python performance benchmark suite [2], and programs that are used to evaluate Reticulated Python used by existing research studies [13, 11, 51, 105]. Below we refer to these programs as *MicroBench*. 6 programs are collected from a Python textbook for educating undergraduate students used at The University of Tokyo [3]. Although they were not designed or used for gradual typing, we assume that they exhibit realistic Python coding patterns in education and serve as representative real-world programs.

9 programs originate from the book Structures and Interpretation of Computer Programs (SICP) [1]. Those programs are expected to be challenging in terms of compilation time for type inference and also statically constructing, analyzing data flow graphs, as they contain a larger number of function definitions and function applications (presented in Table 4.1) including the use of function pointers. Since the original programs are written

[2] `https://github.com/python/pyperformance` (visited on 2025-09-29).

[3] `https://www.graco.c.u-tokyo.ac.jp/labs/morihata/textbook/python_textbook.htm` (visited on 2025-09-29).

in Scheme [4], we utilize a large language model (Gemini 2.5 Flash) to help translate them to Python. We used the following prompt appended with an original Scheme program for the translation:

```
Translate this program to Python. This program is adapted
from the book Structures and Interpretation of Computer
Programs. Include a main function for execution. Do not
use f-string, lambda expression, starred expression,
variable argument or class.
```

Due to the limitations in Reticulated Python and Herder, several Python features such as classes are excluded during the translation, as they are not used in the resulting programs.

The other 4 programs are modified variants with nested function calls of 4 programs from MicroBench. These programs are designed to test the compilation time and performance impact of our method for programs with nested function calls. Here, a nested function call refers to a function call within another function. Given a program, we manually refactor complex or long statements to use helper functions. We create 3 to 6 helper functions for each program. For example, a code fragment from one benchmark program is given as

```
def SOR_execute(omega, G, cycles):
    w,h,d = G
    for p in range(cycles):
        for y in range(1, h - 1):
            ...
```

We refactor it into

```
def SOR_execute(omega, G, cycles):
    w,h,d = G
    for p in range(cycles):
        loop_heights(p, w, h, d, omega, G)

def loop_heights(p, w, h, d, omega, G):
    for y in range(1, h - 1):
        ...
```

`loop_heights` is a manually created helper function and it is called within function `SOR_execute`, which introduces a nested function call. Below we refer to these manually synthesized programs as *Syn*. All code changes do

[4] `https://github.com/ivanjovanovic/sicp/tree/master` (visited on 2025-09-29).

not change the functionality of the original programs. A complete list of all code changes is included in our artifact.

Table 4.1 outlines some detailed characteristics of the benchmark programs. Each row in the table categorizes the dataset by its data source. Although our dataset is not exceedingly large and may introduce some collection bias, it includes more programs from multiple public sources, compared to other datasets used in existing studies [104, 12, 51, 13].

Table 4.1: Dataset overview. Programs are grouped by data source (number of included programs). Each cell gives min/mean/max values in the programs. Def and Call are numbers of lexical function (and method) definitions and function applications. Edge is number of edges in the constructed graph by TypePycker.

Group (Program)	LOC	Def	Call	Edge
MicroBench (31)	22/64/145	2/7/15	2/11/25	51/319/823
PyTextbook (6)	55/99/215	7/12/25	13/25/48	259/489/1011
SICP (9)	181/317/504	19/42/75	18/56/105	509/1052/1760
Syn (4)	55/85/139	9/11/12	22/30/38	390/500/804

The experiments are performed on a Ubuntu 24.04.2 LTS desktop machine equipped with an AMD Ryzen 9 9950X 16-Core Processor. The desktop machine is installed with OpenJDK 17.0.5 and Python 3.9.7 for our implementation. The programs are executed using the guarded semantics of Reticulated Python [105, 104].

4.4.2 Validating Selection Effectiveness

We compare the execution time of the benchmark programs to examine if the proposed selection of additional type annotations can improve the performance of gradually typed programs. Given a benchmark program, we compare the execution times of its three variants: a *Given* variant that is equivalent to the original benchmark program; an *Infer* variant that includes all additional type annotations derived by type inference; a *Chosen* variant that includes only selected additional type annotations by TypePycker. We execute each of the program variants 10 times and measured the arithmetic mean of the elapsed time. Below we use the terms Given, Infer, and Chosen to refer to these three kinds of programs.

Our method selects all the additional type annotations derived by type inference in 9 of 50 benchmark programs. This is an expected behavior of

our method. Intuitively, if a gradually typed program can be enhanced by a type inference engine to be fully typed, or if all additional type annotations are expected to eliminate runtime casts, one should use those additional type annotations for optimal performance instead of discarding them. For example, one benchmark `Mandelbrot` from MicroBench calculates the classic fractal, which mainly involves binary operations over floating-point numbers. It does not use any unknown library functions with respect to types. Our type inference engine infers most of the types within the program. Below we exclude the results of these 9 programs from further comparisons in this subsection. Infer and Chosen for these programs also give close performance in the experiments.

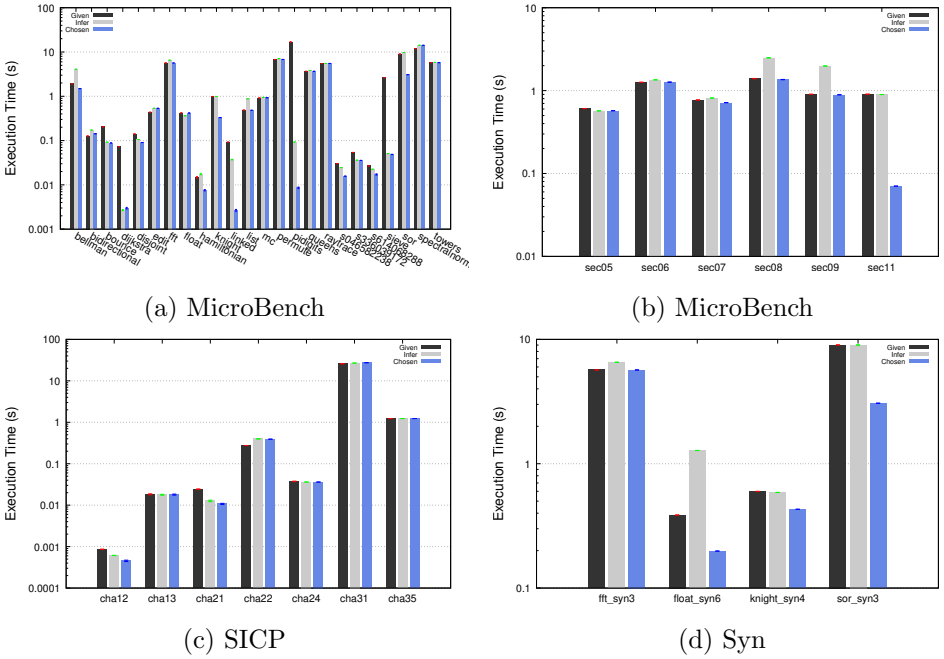
Experiment results show that the proposed method gives positive performance impact in the benchmark programs. Figure 4.3 presents the execution times of Given, Infer, and Chosen for each benchmark program. To qualitatively assess the performance impact, we categorize the relative execution time of Infer and Chosen for each benchmark program into three cases. The categorization is based on the arithmetic mean ($Mean$) and standard error (SE) of the measured execution times:

$$\text{case} = \begin{cases} \text{win} & Mean_{Chosen} + SE_{Chosen} < Mean_{Infer} - SE_{Infer} \\ \text{loss} & Mean_{Chosen} - SE_{Chosen} > Mean_{Infer} + SE_{Infer} \\ \text{tie} & \text{otherwise} \end{cases}$$

A program is considered as either a *win* or *loss* case if the error bars of Infer and Chosen do not overlap. Otherwise, it is considered as a *tie* case. This indicates that the performance impact by our method is not significant.

Our method gives win cases in 32 of 41 programs, and gives tie cases in 6 of 41 programs. Figure 4.4 presents the speedup ratio comparing Infer and Chosen, which is calculated as $Mean_{Infer}/Mean_{Chosen}$. Chosen outperforms Infer with a more than 1.1x speedup ratio in 22 programs, and with a more than 5x speedup ratio in 4 programs. Infer is slower than Given, while Chosen is faster than both Given and Infer in 10 programs. This demonstrates scenarios when simply appending all the type annotations suggested by type inference degrades the performance of gradually typed programs, but the selection of them can lead to performance improvement.

It is also observed that our method does not always improve the performance. It gives loss cases in 3 programs. We manually inspect the slower programs and find that the performance degradation is mainly due to the limitation of static analysis. In `dijkstra` from Figure 4.3a, the code fragment is illustrated below:



(a) MicroBench (b) MicroBench

(c) SICP (d) Syn

Figure 4.3: Execution time results. Each sub-figure groups programs by data source. X-axis shows program names. Y-axis shows execution time in seconds. Each bar is mean of 10 runs with error bars denoting standard errors. Each program features Given, Infer, and Chosen from left to right. Lower is better.

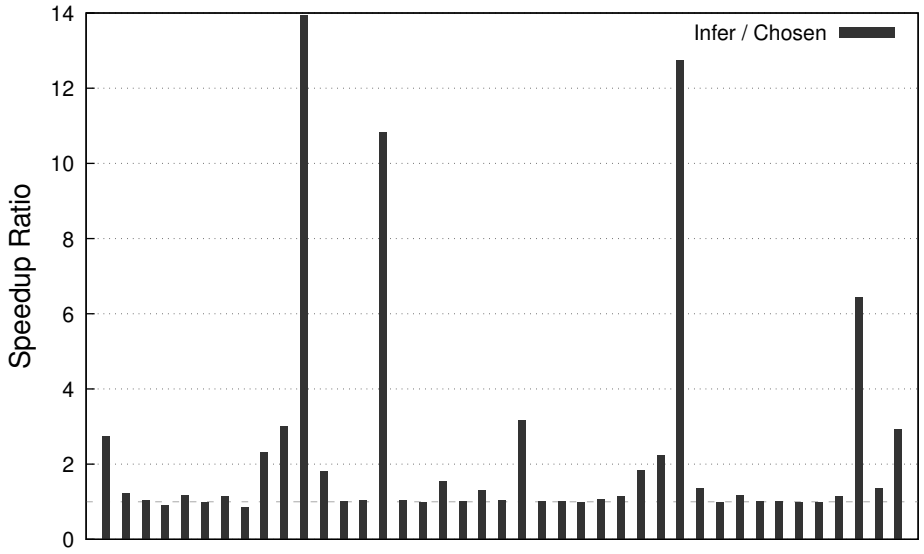


Figure 4.4: Speedup ratio of Infer over Chosen. X-axis shows programs. > 1 is better.

```
u = expr
for v in vertices:
    distances[v] = distances[u] + graph[u][v]
```

Variable `u` is assigned to `expr` that is typed as the unknown type `*`, and `u` is used as an index for array accesses later, which is expected to be type `Int`. The type for `u` is inferred as `Int`, which is appended in Infer but not appended in Chosen. This changes the occurrence of a runtime cast `Int` $\Leftarrow$ `*` from the assignment to the later array accesses. However, the later runtime casts occur in the for loop, which causes more overhead during execution in Chosen. Another slower case in program `float` has a similar scenario. For the other slower case `cha31`, Chosen appends different type annotations and produces multiple different occurrences of runtime casts from those by Infer. It is difficult to conclude the slowdown into a straightforward reason.

We also inspect and find that our method may not give better performance when runtime casts would occur in different control-flow branches. For example, one code fragment in program `sec08` is shown below:

```
1 def mergesort(xs):
2     if len(xs) <= 1:
3         return xs
```

```
4       else:
5           return merge(...)
6
7   mergesort([randint(0, 100) for _ in range(70)])
```

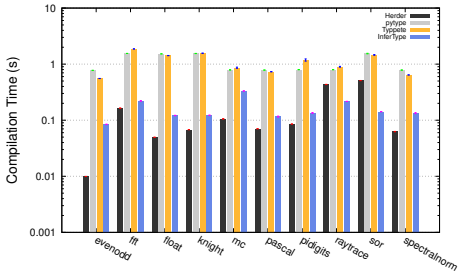
The argument at line 7 is an array literal typed as `Array(*)` since `randint` is a library function whose type is not available. The return type of function `merge` is `Array(Int)` in both Infer and Chosen. The return type of `mergesort` is inferred as `Array(Int)`. In Infer, this return type annotation would incur an occurrence of a runtime cast `Array(*)` $\Leftarrow$ `Array(Int)` in the if branch, but no runtime cast in the else branch. In Chosen, a runtime cast `*` $\Leftarrow$ `Array(*)` occurs in the else branch. TypePycker does not select the return type annotation for `mergesort` since one of its closest source vertices (representing the array literal at line 7) contains `*`. However, the program executes the else branch more frequently, which results in less runtime overhead in the else branch in Infer. The selection for this function return type degrades the execution performance.

4.4.3 Demonstrating Stable Compilation Time Enabled by InferType

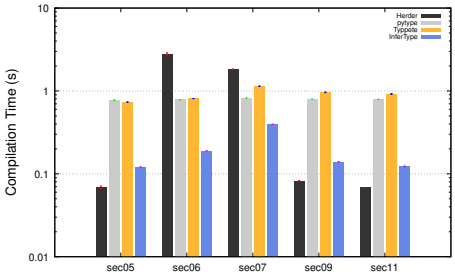
We compare TypePycker enabled by InferType with that enabled by two existing type inference engines pytype [35] and Typpete [39] for Python. The goal of the following experiments is to show that with the generated efficient code by InferType, it can practically enable more stable compilation time and comparable execution time by TypePycker compared to existing engines. InferType does not disturb the benefit of the fast selection in our proposed method in Section 4.3. A toolkit with straightforward code generation unlike InferType might perform slow type inference, which might consume too much budget time of compilation for enabling TypePycker's selection. Using such a straightforward toolkit on TypePycker might not be able to derive comparable compilation time compared to using existing type inference engines.

We also compare TypePycker with an existing work Herder. We show that our method of selectively adding type annotations is more stable and practical with respect to compilation time independent of a used external type inference engine. We treat Herder as a baseline compared to TypePycker enabled by different type inference engines. Herder [12] is a tool that statically analyzes the cost of runtime casts of a gradually typed program with different subsets of additional type annotations. Those additional type annotations are derived by their type inference. It outputs a program with

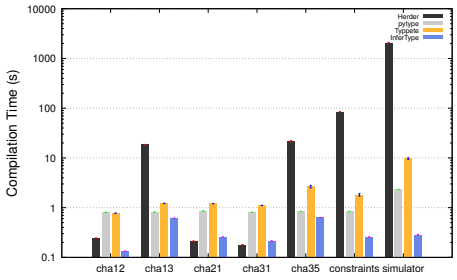
optimal performance suggested by its analysis, where a subset of additional type annotations are appended to that program. The main difference from our method is the approach to select additional type annotations. Herder performs comprehensive analysis of all possible subsets of additional type annotations with a carefully designed cost analysis, while TypePycker employs a lightweight, amortized approach. Another difference is that Herder integrates type inference with its selection, while our selection is independent of an external type inference engine.



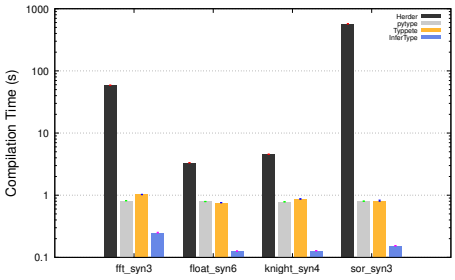
(a) MicroBench



(b) PyTextbook



(c) SICP



(d) Syn

Figure 4.5: Compilation time results by Herder and TypePycker enabled by pytype, Typpete, and InferType. Each sub-figure groups programs by data source. X-axis shows program names. Y-axis shows compilation time in seconds. Each bar is mean of 10 runs with error bars denoting standard errors.

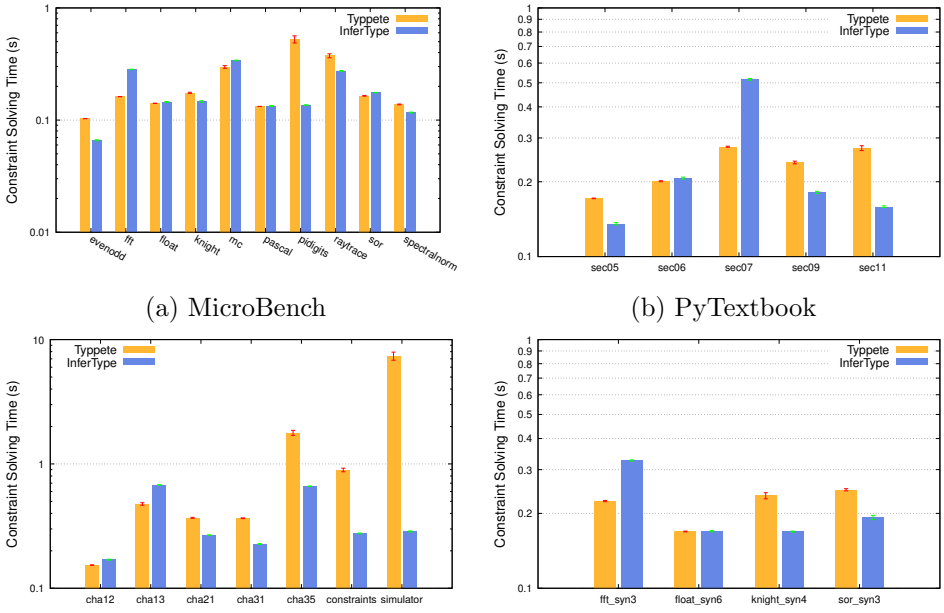
The compilation time by Herder and TypePycker enabled by pytype, Typpete, InferType across the benchmark programs is given in Figure 4.5. Note that some benchmarks are excluded from the comparison since Herder does not support several Python features such as classes. We also manually disable some I/O operations in Typpete that are irrelevant to the type inference process for enabling TypePycker in our experiments.

Experiment results show that InferType can enable stable compilation time by its generated code of type inference compared to other existing type inference engines and Herder. Our configuration of TypePycker enabled by InferType shows stable compilation time across all benchmark programs. It is less than 1 second in most of the cases including those cases where the other configurations consume a significant amount of time. TypePycker enabled by InferType clearly shows faster compilation time for the programs from SICP and Syn.

Although Herder, pytype, and Typpete show acceptable compilation time for small programs, it scales poorly to other programs. For example, all four configurations give stable and acceptable compilation time for the programs from MicroBench in Figure 4.5a. Herder is faster in most of these programs. The higher cost by TypePycker is because it invokes an external type inference engine such as invoking an SMT solver by InferType. This relatively heavy cost is a reason that TypePycker is slower than Herder for MicroBench. However, the compilation time by Herder exceeds 1 second in 12 of 28 programs, and exceeds 10 seconds in 7 programs. It ranges from 3 to 627 seconds for the programs from Syn in Figure 4.5d. It takes more than 2000 seconds in the worst case, which is a non-artificial, real-world program from SICP in Figure 4.5c. The compilation time by pytype and Typpete also exceeds 1 second for the programs in SICP and they are up to 3.4 and 9.7 seconds, respectively. The slow compilation time arises from the large search space that Herder must explore. Although Herder adopts variational typing [18] for efficient computation compared to a brute-force enumeration, the possible number of different occurrences of runtime casts in a program is exponential considering general cases. For example, a program would contain numerous different occurrences of runtime casts under different sets of type annotations for function parameters if the program contains deeply nested function calls. Herder must analyze the costs for all different combinations to suggest optimal performance.

We also measured the elapsed time for constraint solving within type inference by our implemented engine using InferType and for that within Typpete, which results are given in Figure 4.6. Here, the measured constraint-solving time is the duration of performing type inference by the Z3 SMT solver, called by InferType or Typpete. Both of them use the same underlying Z3 installed in our tested machine. The measured time excludes that for collecting type constraints by AST traversal. This measurement is for fair comparison since InferType is implemented in Java while Typpete is implemented in Python. The results indicate that the fast and stable inference time is not naively due to our implementation language, but due to

the generated code by InferType. Across the benchmark programs, InferType delivers more stable time of constraint solving and its time is less than 0.5 second in all the programs. On the other hand, the constraint-solving time by Typpete varies across programs and it is up to 7.4 seconds. We could not conduct a direct comparison of the constraint-solving time between that by InferType against Herder or pytype since their tools integrate type inference with other processes.



(a) MicroBench

(b) PyTextbook

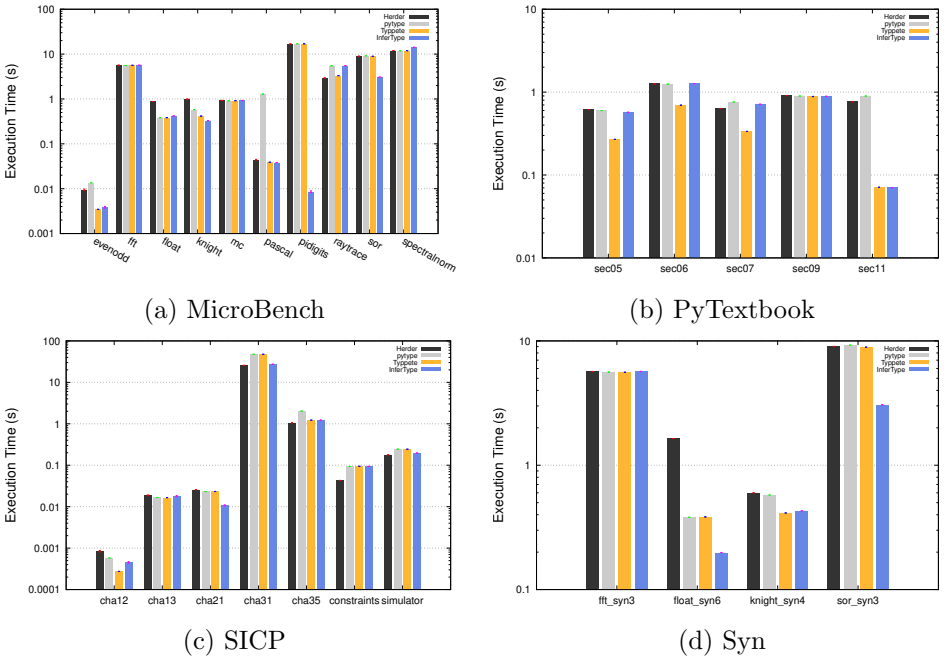
(c) SICP

(d) Syn

Figure 4.6: Constraint solving time results by Typpete and InferType. Each sub-figure groups programs by data source. X-axis shows program names. Y-axis shows constraint solving time in seconds. Each bar is mean of 10 runs with error bars denoting standard errors.

We demonstrate that our configuration produces comparable execution performance compared to using other existing type inference engines or Herder. Figure 4.7 shows the execution time of the benchmark programs appended with additional type annotations selected by Herder or TypePycker enabled by pytype, Typpete, or InferType. The programs are executed using our modified Reticulated Python. The only difference in the comparison here is the appended type annotations. Compared to Herder, TypePycker enabled by InferType gives faster execution speed in 17 of 28 programs, while

it is slower in the remaining 11 programs. The execution time ratio (Herder over TypePycker) ranges from 0.5x to 10x. A main reason of the faster execution speed by our configuration is because our implemented type inference engine by InferType could infer more function parameter types, which enables more optimization opportunities.



(a) MicroBench (b) PyTextbook

(c) SICP (d) Syn

Figure 4.7: Execution time results by Herder and TypePycker enabled by pytype, Typpete, InferType. Each sub-figure groups programs by data source. X-axis shows program names. Y-axis shows execution time in seconds. Each bar is mean of 10 runs with error bars denoting standard errors.

4.4.4 Assessing Performance Impact across Language Platforms

We demonstrate how the proposed method affects the execution performance by running the benchmark programs on four different language platforms based on Reticulated Python. Here, a language platform refers to a concrete implemented variant of the underlying language Reticulated Python. Each implemented language variant would produce different occurrences of runtime casts given a same program since expressions may be given different

static types. We implement the four language platforms by applying different combinations of our two main modifications over the original Reticulated Python: the fast-slow technique and type annotation support for local variables. Note that omitting the fast-slow technique will not preserve program behaviors because additional type annotations will be directly appended to the original functions. The goal of this subsection is to demonstrate that the performance gains are not simply due to our modifications over the original Reticulated Python, but the selection of additional type annotations by our proposal contributes to the observed performance gains.

We compare the execution time of appending only selected type annotations by our method with the execution time of appending all additional type annotations derived from type inference. Each program was executed 10 times on each language platform, and the elapsed time was measured. Same as the comparison in Section 4.4.2, we categorize the relative execution time of Infer and Chosen into win, tie, loss cases by considering the mean and standard error. The results for 9 benchmark programs are excluded from the comparison below because Chosen appends the same additional type annotations as Infer.

The results indicate that the proposed method can give positive performance impact on four different language platforms. We denote a language platform with support for the fast-slow technique as F, support for local variable type annotations as L, and no support as n. For example, a language platform supporting local variable type annotations but no fast-slow technique is denoted as nL. Chosen either outperforms (win), or achieves comparable performance (tie) compared to Infer in 29, 27, 40, and 38 programs across 41 programs on the four language platforms nn, nL, Fn, and FL, respectively. The results from the four language platforms indicate that the performance gains are not only from our modifications over Reticulated Python. The selection of type annotations by our method contributes to the performance improvement.

Table 4.2 demonstrates both categorization results and performance ratios for several selected benchmark programs across language platforms. Those programs are selected to showcase three scenarios: Given a program, our method presents a) either win, tie, or loss cases on all language platforms; b) a mix of win/loss and tie cases on all language platforms; c) a win case on one platform but a loss case on another platform.

Table 4.2: Results of selected programs on four language platforms. Horizontal lines group programs by categorization scenarios. The second column shows the categorization results on four platforms. W, T, L represent win, tie, loss. The later columns show speedup ratios (Infer over Chosen) on each platform. > 1 is better.

Program	Categorizations	*nn*	*nL*	*Fn*	*FL*
`disjoint`	WWWW	1.19	1.18	1.20	1.17
`list`	WWWW	2.45	2.44	1.80	1.81
`sec07`	WWWW	1.20	1.14	1.06	1.14
`sec09`	WWWW	1.34	1.35	2.25	2.22
`fft`	WWTW	1.19	1.15	1.00	1.16
`sieve`	TWTW	0.99	1.03	0.99	1.04
`cha13`	TWWT	1.02	1.11	1.13	0.99
`cha24`	LLTT	0.98	0.92	0.99	1.00
`dijkstra`	LLWL	0.87	0.91	1.08	0.91
`knight`	LLWW	0.24	0.56	1.20	3.00
`cha31`	WLTL	1.02	0.94	1.00	0.98
`sor_syn3`	LLWW	0.06	0.06	2.94	2.94

4.4.5 Testing Behaviors with JIT

We test several benchmark programs on Reticulated Python with PyPy [5], a Python variant with JIT compilation, to showcase whether TypePycker can improve the execution performance on a gradually typed language with JIT support. Figure 4.8 presents the execution time of the tested programs appended with all additional type annotations by type inference and only selected type annotations by TypePycker. We find that TypePycker can achieve performance improvement in some benchmark programs. It is also observed that TypePycker gives worse or different performance impact on PyPy compared to CPython. For instance, TypePycker outperforms using all additional type annotations by type inference on CPython on `fft` and `s046582238` given in Figure 4.3a. However, TypePycker slows down the execution speed of the two programs with PyPy. Since the comparison is limited and also the number of tested benchmarks is relatively small, a more comprehensive evaluation is needed to draw a reliable conclusion whether TypePycker can benefit a gradually typed language based on JIT or other dynamic-analysis techniques. We regard this as one direction of our future work.

4.5 Threats to Validity

This research work is threatened by several potential factors that could affect the validity of its findings. Below we list the main factors.

This work does not formally discuss the correctness of its optimization with respect to preserving program behaviors. The correctness relies on the underlying type inference engine. To maintain the correctness, one must use a type inference engine only deriving sound types that do not change program behaviors. Exploring such an engine falls outside the scope of this study. We mitigate this threat in two limited ways. First, we manually validate the correctness of the inferred types for benchmark programs in the experiments. We confirm that they do not change the behaviors of the benchmark programs. Second, we test our implementation with several challenging programs written in a recent research paper [13]. Those programs are specifically designed to test the correctness. Our engine successfully preserves the behaviors of those programs. We include those programs in our artifact.

[5] Most of the other benchmarks encounter errors on Reticulated Python with PyPy, which is also reported by an existing work [13].

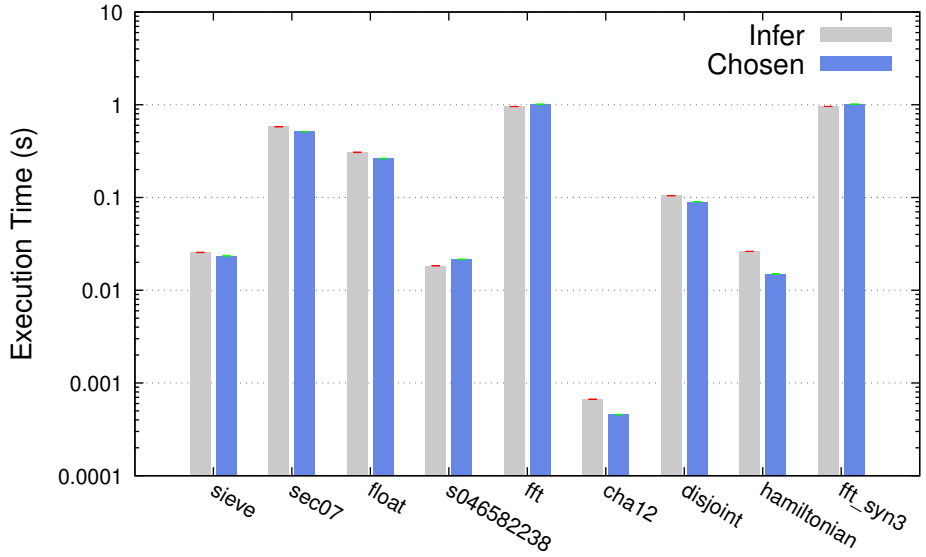


Figure 4.8: Execution time results on Reticulated Python with PyPy. X-axis shows program names. Y-axis shows execution time in seconds. Each bar is mean of 10 runs with error bars denoting standard errors.

The performance gains observed in the experiments might be more pronounced than what would be achieved in practical scenarios. This limitation arises primarily from using the collected un-annotated programs as our baseline. Although those benchmark programs are treated partially typed when executed by Reticulated Python in the experiments, real-world programs may already be annotated with some degree of concrete type annotations for function parameters, function returns, or local variables by programmers in a gradually typed language. We were not able to collect a suitable dataset containing real annotated programs.

4.6 Related Work

Understanding the performance changes in gradual typing has become a significant research area following the emergence of a wide range of gradually typed languages. Existing gradually typed languages are mainly classified into two broad categories: erasure and non-erasure semantics [36, 19]. Languages with erasure semantics allow values of unexpected types to flow from untyped code into typed code by erasing types at runtime. Examples

are TypeScript, Flow [17], and mypy [57]. Languages with non-erasure semantics, in contrast, enforce static types at runtime by performing runtime casts at the boundary between untyped and typed code. They are further distinguished between deep semantics such as Gradualtalk [2], Typed Racket [101] and first-order semantics such as Reticulated Python [105], Nom [72]. Several studies have collectively discovered significant runtime overhead in gradually typed languages with non-erasure semantics in such as Typed Racket [100, 37] and Reticulated Python [38, 12]. In this research, we do not consider languages such as TypeScript since programs written under erasure semantics would not impose significant runtime overhead from static type annotations. The goal of this work is to develop a compile-time technique for mitigating the performance penalty found in gradually typed languages with non-erasure semantics. The investigated language Reticulated Python in our experiments is one instance. Exploring the effectiveness of the proposal on other gradually typed languages [101, 53, 68] with non-erasure semantics is our future work.

A number of research efforts have emerged following the report of performance degradation in gradual typing. One major direction is to enhance static information for improving the performance. Rastogi et al. [87] first used type inference to optimize gradually typed programs on ActionScript [70] by considering a correctness goal of preserving program behaviors. Vitousek et al. [104] optimized the transient semantics of Reticulated Python by removing runtime casts verified by type inference. Following the principle of preserving program behaviors, Campora et al. [13] proposed discriminative typing to safely improve the performance by keeping both an optimized version of a function and also the unoptimized original function based on type inference. Our implementation adopts Campora et al.'s idea of maintaining both optimized and unoptimized functions. Unlike their approach, however, our method does not formally guarantee that a runtime error is blamed at the same location because we use a different type system. Our method differs in selectively adding type annotations derived by type inference for better performance.

There are research studies for addressing the performance pitfalls in gradual typing by statically reasoning how types affect the performance. Herder [12] is a static cost analysis tool to estimate the execution time of different combinations of typed parameters within a given program. These combinations are often termed *configurations* in the context of gradual typing. The tool outputs a configuration expected to yield optimal execution time based on its cost analysis. Our novelty compared to Herder is an efficient method for selecting additional type annotations. TypePycker does not

need to analyze all possible configurations. Experiment results showed that TypePycker can achieve comparable execution performance while having a more stable compilation time. Khan et al. [51] developed a machine-learning based approach *LearnPerf* to predict the performance of possible configurations in a given program. Similar to TypePycker, LearnPerf uses an external type inference engine to derive additional type annotations. Besides the difference presented in Section 4.2, LearnPerf differs from TypePycker in two ways. First, LearnPerf does not consider preserving program behaviors. Second, LearnPerf trains an individual model for each input program, which limits its portability for practical usage.

Another direction for understanding and optimizing the performance of gradual typing is the use of runtime techniques. Bauman et al. [9] showed practical performance improvement by using a tracing JIT after the report of significant runtime overhead by Takikawa et al. Richards et al. [89] bridged the gap between a virtual machine leveraging JIT compilation and gradually typed languages to eliminate redundant runtime casts. Roberts et al. [90] employed a standard JIT compilation to reduce the overhead of transient runtime casts. This work primarily examines the proposed method on Reticulated Python without additional runtime techniques such as JIT since we aim to study the genuine effect of the proposal as a compile-time technique. Although the experiments testing with PyPy in Section 4.4.5 are admittedly limited, we believe that the proposed method can be integrated and potentially give performance improvement to other offline compilation and runtime techniques. Greenman et al. [37] and Hejduk et al. [40] explored how profiling data can be leveraged to navigate decisions for type migration by a large-scale empirical study based on the rational-programmer [55, 56] method. Our work targets a different point in providing a compile-time technique to automatically improve the performance of a gradually typed program.

Static type inference has been a well-established technique across a variety of programming paradigms. Type inference benefits fundamental applications such as early error detection [42, 66] and performance improvement [15]. In Python, there are numerous existing type inference engines leveraging diverse techniques such as rule-based systems [35], abstract interpretation [69], SMT solvers [39, 109], and machine-learning based approaches [78, 41, 67]. In this work, we implement a custom type inference engine using a compiler toolkit InferType [59]. We show that InferType can enable TypePycker and perform stable type inference time in our experiments.

Several data flow analyses have been applied in the area of type in-

ference. Maggi et al. [61] presented an approach to infer types related to Java bytecode using a data flow analysis. Zhou et al. [110] proposed a method of type inference based on a data flow analysis for decompilation. Bhanuka et al. [10] captured the data flow by using subtype constraints for generating readable type error messages in constraint-based type inference systems. Our proposal shares similarities with those data flow techniques in the construction of a graph for modeling value propagation. Our contribution is distinct in selectively adding type annotations for improving the performance of gradually typed programs.

4.7 Summary

We demonstrated the usage of InferType in enabling our novel approach that efficiently selects type annotations derived by type inference for improving the execution performance of gradually typed programs. As significant costs of runtime casts often occur when a value repeatedly crosses the boundaries between untyped and typed code, our approach selects type annotations placed along the data flows. It quickly checks whether all untyped variables along the data flows are expected to obtain type annotations derived by type inference using a lightweight, amortized approach.

We implemented our proposed approach as TypePycker using InferType, and experimented its effectiveness on a gradually typed language Reticulated Python in improving the execution performance. We find that TypePycker can achieve better execution speed than simply using all the inferred types derived by type inference in the benchmark programs with a speedup ratio up to more than 5x. In comparison with existing approaches, TypePycker enabled by InferType delivers comparable execution performance while the compilation time of deriving and selecting type annotations is much more stable. We also observe that TypePycker achieves much faster compilation time in real-world programs where an existing approach incurs significantly long compilation time such as more than 10 minutes. We believe that our method is a candidate for helping mitigate the performance bottleneck of gradual typing in practical scenarios.

Chapter 5

Conclusion

This dissertation presents our studies on addressing the limited support for type inference in language development. Despite the increasing demand for type inference in modern language development, existing language workbenches offer limited support for its practical implementation. We highlight that a primary barrier to its support is the difficulty of achieving computational efficiency in generating code for performant type inference, even when leveraging powerful modern solvers. Generated code from a naively designed toolkit would perform inefficient type inference compared with manually tuned implementations. This discourages the use of generative toolkits for type inference since slow type inference may consume too much time in a compilation pipeline for enabling other techniques in downstream optimizations. Throughout the studies in the dissertation, we have sought to tackle some of the performance bottlenecks by developing practical techniques without harming the expressiveness or guarantees of the original systems.

We propose a toolkit named InferType for implementing constraint-based type inference. By using the interfaces provided by InferType, compiler writers describe type constraints via AST traversal, specify typing rules, and give them to InferType. InferType then performs type inference to compute one valid binding of solution types by translation to the Z3 SMT solver. InferType incorporates a novel optimization technique for mitigating the performance bottleneck of type inference involving nested types. It reduces the search space in constraint solving by pre-computing the structures of the type variables. Our experiment results indicate that InferType is able to produce efficient type inference compared with existing engines. We also show that the developed optimization is effective in alleviating the

performance bottleneck of inferring nested types. It can significantly speed up type inference for programs with deeply nested types, and potentially improve the performance of type inference for programs with nested types.

We demonstrate the utility of our developed toolkit InferType within the context of gradual typing. We tackle a prevalent challenge of the execution degradation in gradual typing by using InferType and also a proposed new method. Our method is a novel lightweight method for improving the execution speed of gradually typed programs by selectively adding type annotations. As the execution overhead tends to increase when values repeatedly cross the boundaries between untyped and typed code in gradual typing, our method only selects the additional type annotations by type inference for a variable when type inference can infer types for all the other untyped expressions that can reach that variable along the data flows. Our method quickly checks this condition using a lightweight, amortized approach. It does not perform expensive traversal through data flow paths or any brute-force enumeration. The experiment results empirically confirm that InferType can generate efficient type inference that is practically useful for research purposes. We implement our method in a tool named TypePycker as an extended version of an existing language Reticulated Python with automatic type inference using InferType. Compared with existing type inference engines, InferType offers a clear advantage of producing efficient type inference and also sufficient inferred type annotations for improving the execution speed. Our findings also indicate that TypePycker is practically useful to mitigate the performance degradation of gradually typed programs in Reticulated Python. We show that the proposed method achieves improvement in execution performance compared with a naive strategy of using all inferred types. We observe that TypePycker outperforms an existing tool in both faster type inference and better execution time on several real-world programs.

Future Work

This dissertation addresses existing gaps in tool support for implementing type inference engines in limited ways. Here we discuss several avenues for future research.

One significant direction for future studies is to consider the possibility of supporting principal typing in InferType. While the current InferType is able to efficiently compute a valid binding of inferred types for a given program, it does not directly support generating code that can produce

a principal type [20, 76], which is the most general type that subsumes all possible types that can be assigned to each program fragment in the given program. Supporting principal typing is essential for enabling modular type inference [45]. Achieving modularity in type inference is crucial for large-scale software development to facilitate separate compilation and faster incremental builds. It allows the independent analysis of separate modules without repeatedly compiling the whole project in each compilation. A challenging issue is to develop techniques to simplify type constraints without compromising the computational efficiency of InferType.

Assessing the practical significance of InferType is another direction for future research. In the dissertation, the use of InferType is only empirically studied in one particular research task related to gradual typing. It is interesting to explore whether the generated code by InferType is efficient enough to enable techniques in other downstream tasks such as automatic migration from dynamic codebases to compiled languages as we mentioned in Chapter 1. Furthermore, it is not clear whether InferType can be extended to support a richer range of type systems while maintaining efficiency in the generated code for type inference. Currently InferType supports only a limited range of type systems as we discussed in Section 3.5. It would be promising to show a broader impact of InferType in assisting the development of various languages with more expressive types while maintaining its benefit of generating code for efficient inference.

We studied a novel method for improving the execution of gradually typed programs in Chapter 4. It selectively adds type annotations during Ahead-of-Time compilation. The extent to which the proposed method benefits gradually typed languages with Just-in-Time (JIT) compilation support remains an open question, though we tested our method on a Python variant with JIT support for providing limited evidence. It is expected that future practical languages with gradual typing would be implemented with JIT support since research studies have revealed that JIT can effectively reduce the execution overhead by runtime casts [9, 89, 90]. Considering the benefit of combining a selection policy and generating fast paths in JIT compilation is left for future work.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge, MA, USA: MIT Press, 1996. ISBN: 978-0-262-01153-2.

[2] Esteban Allende et al. "Gradual typing for Smalltalk". In: *Science of Computer Programming* 96 (2014). Special issue on Advances in Smalltalk based Systems, pp. 52–69. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.06.006.

[3] Hendrik van Antwerpen et al. "Scopes as Types". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276484.

[4] Hendrik Antwerpen et al. "A constraint language for static semantic analysis based on scope graphs". In: Jan. 2016, pp. 49–60. DOI: 10.1145/2847538.2847543.

[5] Mikhail Barash. "Vision: the next 700 language workbenches". In: *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 16–21. ISBN: 978-1-4503-9111-5. DOI: 10.1145/3486608.3486907.

[6] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.

[7] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[8] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11.

[9] Spenser Bauman et al. “Sound gradual typing: only mostly dead”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133878.

[10] Ishan Bhanuka et al. “Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622812.

[11] John Peter Campora and Sheng Chen. “Taming type annotations in gradual typing”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428259.

[12] John Peter Campora, Sheng Chen, and Eric Walkingshaw. “Casts and costs: harmonizing safety and performance in gradual typing”. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236793.

[13] John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. “Type-Based Gradual Typing Performance Optimization”. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632931.

[14] Zachary Carter. *Jison*. URL: https://gerhobbelt.github.io/jison/docs/#installation (visited on 04/01/2026).

[15] Robert Cartwright and Mike Fagan. “Soft typing”. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI ’91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pp. 278–292. ISBN: 978-0-89791-428-4. DOI: 10.1145/113445.113469.

[16] Stephen Chang, Alex Knauth, and Ben Greenman. “Type Systems as Macros”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’17. Paris, France: Association for Computing Machinery, 2017, pp. 694–705. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009886.

[17] Avik Chaudhuri et al. “Fast and precise type checking for JavaScript”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133872.

[18] Sheng Chen, Martin Erwig, and Eric Walkingshaw. “Extending Type Inference to Variational Programs”. In: *ACM Trans. Program. Lang. Syst.* 36.1 (Mar. 2014). ISSN: 0164-0925. DOI: 10.1145/2518190.

[19] Benjamin Chung et al. "KafKa: Gradual Typing for Objects". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Ed. by Todd Millstein. Vol. 109. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 12:1–12:25. ISBN: 978-3-95977-079-8. DOI: 10.4230/LIPIcs.ECOOP.2018.12.

[20] Luis Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. ISBN: 978-0-89791-065-1. DOI: 10.1145/582153.582176.

[21] Damien P. George. *MicroPython*. https://micropython.org. 2014. (Visited on 09/29/2025).

[22] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 978-3-540-78799-0.

[23] David Déharbe et al. "Exploiting Symmetry in SMT Problems". In: *Automated Deduction – CADE-23*. Ed. by Nikolaj Bjørner and Viorica Sofronie-Stokkermans. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 222–236. ISBN: 978-3-642-22438-6.

[24] Frank DeRemer and Thomas J. Pennello. "Efficient computation of LALR(1) look-ahead sets". In: *Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '79. Denver, Colorado, USA: Association for Computing Machinery, 1979, pp. 176–187. ISBN: 0897910028. DOI: 10.1145/800229.806968.

[25] Luca Di Grazia and Michael Pradel. "The evolution of type annotations in python: an empirical study". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549114.

[26] Bruno Dutertre. "Yices 2.2". In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 737–744. ISBN: 978-3-319-08867-9.

[27] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 502–518. ISBN: 978-3-540-24605-3. DOI: 10.1007/978-3-540-24605-3_37.

[28] Sven Efftinge et al. "Xbase: implementing domain-specific languages for Java". In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE '12. Dresden, Germany: Association for Computing Machinery, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371419.

[29] John Ellson et al. "Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools". In: *Graph Drawing Software*. Ed. by Michael Jünger and Petra Mutzel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 127–148. ISBN: 978-3-642-18638-7. DOI: 10.1007/978-3-642-18638-7_6.

[30] Sebastian Erdweg et al. "The State of the Art in Language Workbenches". In: *Software Language Engineering*. Ed. by Martin Erwig, Richard F. Paige, and Eric Van Wyk. Cham: Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02654-1. DOI: 10.1007/978-3-319-02654-1_11.

[31] Moritz Eysholdt and Heiko Behrens. "Xtext: Implement Your Language Faster than the Quick and Dirty Way". In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 307–309. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869625.

[32] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: https://martinfowler.com/articles/languageWorkbench.html (visited on 04/01/2026).

[33] Martin Fowler and R Parsons. *Addison-Wesley signature, Domain specific languages . Massachussets*. 2010.

[34] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. "Choral: Object-oriented Choreographic Programming". In: *ACM Trans. Program. Lang. Syst.* 46.1 (Jan. 2024). ISSN: 0164-0925. DOI: 10.1145/3632398.

[35] Google. *pytype*. 2021. URL: https://google.github.io/pytype/ (visited on 09/29/2025).

[36] Ben Greenman and Matthias Felleisen. "A spectrum of type soundness and performance". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236766.

[37] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. "How Profilers Can Help Navigate Type Migration". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622817.

[38] Ben Greenman and Zeina Migeed. "On the cost of type-tag soundness". In: *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '18. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 30–39. ISBN: 978-1-4503-5587-2. DOI: 10.1145/3162066.

[39] Mostafa Hassan et al. "MaxSMT-Based Type Inference for Python 3". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 12–19. ISBN: 978-3-319-96142-2. DOI: 10.1007/978-3-319-96142-2_2.

[40] Nathaniel Hejduk et al. "Navigating Mixed-Typed Migration with Profilers". In: *ACM Trans. Program. Lang. Syst.* 48.1 (Mar. 2026). ISSN: 0164-0925. DOI: 10.1145/3785004.

[41] Vincent J. Hellendoorn et al. "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*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 152–162. ISBN: 978-1-4503-5573-5. DOI: 10.1145/3236024.3236051.

[42] J. Roger Hindley. "The Principal Type-Scheme of an Object in Combinatory Logic". In: *Transactions of the American Mathematical Society* 146 (1969), pp. 29–60. ISSN: 0002-9947. DOI: 10.2307/1995158.

[43] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Trans. Program. Lang. Syst.* 23.3 (May 2001), pp. 396–450. ISSN: 0164-0925. DOI: 10.1145/503502.503505.

[44] Yuka Ikarashi et al. "Exocompilation for productive programming of hardware accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523446.

[45] Trevor Jim. "What are principal typings and what are they good for?" In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, 1996, pp. 42–53. ISBN: 978-0-89791-769-8. DOI: 10.1145/237721.237728.

[46] Patricia Johann and Andrew Polonsky. "Higher-Kinded Data Types: Syntax and Semantics". In: *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2019, pp. 1–13. DOI: 10.1109/LICS.2019.8785657.

[47] Donald B. Johnson. "A Note on Dijkstra's Shortest Path Algorithm". In: *J. ACM* 20.3 (July 1973), pp. 385–388. ISSN: 0004-5411. DOI: 10.1145/321765.321768.

[48] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. "JavaSMT: A Unified Interface for SMT Solvers in Java". In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Sandrine Blazy and Marsha Chechik. Cham: Springer International Publishing, 2016, pp. 139–148. ISBN: 978-3-319-48869-1.

[49] Lennart C.L. Kats and Eelco Visser. "The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[50] Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. "Sound, heuristic type annotation inference for Ruby". In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 112–125. ISBN: 978-1-4503-8175-8. DOI: 10.1145/3426422.3426985.

[51] Mohammad Wahiduzzaman Khan, Sheng Chen, and Yi He. "Learning Gradual Typing Performance". In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 21:1–21:27. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.21.

[52] Kevin Knight. "Unification: A Multidisciplinary Survey". In: *ACM Comput. Surv.* 21.1 (Mar. 1989), pp. 93–124. ISSN: 0360-0300. DOI: 10.1145/62029.62030.

[53] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. "Toward efficient gradual typing for structural types via coercions". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 517–532. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314627.

[54] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: a LLVM-based Python JIT compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162.

[55] Lukas Lazarek et al. "How to evaluate blame for gradual types". In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473573.

[56] Lukas Lazarek et al. "How to Evaluate Blame for Gradual Types, Part 2". In: *Proc. ACM Program. Lang.* 7.ICFP (Aug. 2023). DOI: 10.1145/3607836.

[57] Jukka Lehtosalo, Guido van Rossum, and Ivan Levkivskyi. *mypy*. June 2012. URL: http://mypy-lang.org/ (visited on 06/08/2021).

[58] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc (2nd Ed.)* USA: O'Reilly & Associates, Inc., 1992. ISBN: 978-1-56592-000-2.

[59] Senxi Li, Tetsuro Yamazaki, and Shigeru Chiba. "InferType: A Compiler Toolkit for Implementing Efficient Constraint-Based Type Inference". In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 23:1–23:28. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.23.

[60] Senxi Li et al. "Efficient Selection of Type Annotations for Performance Improvement in Gradual Typing". In: *The Art, Science, and Engineering of Programming* 11.1 (Feb. 2026). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2026/11/3.

[61] Paolo Maggi and Riccardo Sisto. "Using data flow analysis to infer type information in Java bytecode". In: *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation* (2001), pp. 211–222. DOI: 10.1109/SCAM.2001.972683.

[62] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. "Cross-language compiler benchmarking: are we fast yet?" In: *Proceedings of the 12th Symposium on Dynamic Languages*. DLS 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 120–131. ISBN: 978-1-4503-4445-6. DOI: 10.1145/2989225.2989232.

[63] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *SciPy*. 2010. DOI: 10.25080/Majora-92bf1922-00a.

[64] Meta. *Pyrefly*. URL: https://pyrefly.org (visited on 04/01/2026).

[65] Microsoft. *Pyright*. URL: https://microsoft.github.io/pyright/#/ (visited on 04/01/2026).

[66] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: 10.1016/0022-0000(78)90014-4.

[67] Amir M. Mir et al. "Type4Py: practical deep similarity learning-based type inference for python". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2241–2252. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510124.

[68] Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. "Interactive Programming for Microcontrollers by Offloading Dynamic Incremental Compilation". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. MPLR 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 28–40. ISBN: 979-8-4007-1118-3. DOI: 10.1145/3679007.3685062.

[69] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. "Static Type Analysis by Abstract Interpretation of Python Programs". In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 17:1–17:29. ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.17.

[70] Colin Moock. *Essential ActionScript 2.0*. O'Reilly Media, Inc., 2004. ISBN: 0-596-00652-7.

[71] Adriaan Moors, Frank Piessens, and Martin Odersky. "Generics of a higher kind". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA '08. Nashville, TN, USA: Association for Computing Machinery, 2008, pp. 423–438. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449798.

[72] Fabian Muehlboeck and Ross Tate. "Sound gradual typing is nominally alive and well". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133880.

[73] Aina Niemetz et al. "Syntax-Guided Quantifier Instantiation". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 145–163. ISBN: 978-3-030-72013-1.

[74] Martin Odersky, Christoph Zenger, and Matthias Zenger. "Colored Local Type Inference". In: *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '01. London, United Kingdom: Association for Computing Machinery, 2001, pp. 41–53. ISBN: 978-1-58113-336-3. DOI: 10.1145/360204.360207.

[75] T. J. Parr and R. W. Quong. "ANTLR: A predicated-LL(k) parser generator". In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810. DOI: 10.1002/spe.4380250705. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705.

[76] Lionel Parreaux. "The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3409006.

[77] Zvonimir Pavlinovic, Tim King, and Thomas Wies. "Finding Minimum Type Error Sources". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 525–542. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660230.

[78] Yun Peng et al. "Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 2019–2030. ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510038.

[79] Simon Peyton Jones et al. "Simple unification-based type inference for GADTs". In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*. ICFP '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 50–61. ISBN: 1595933093. DOI: 10.1145/1159803.1159811.

[80] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 978-0-262-16209-8.

[81] Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100.

[82] François Pottier. "A framework for type inference with subtyping". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ICFP '98. Baltimore, Maryland, USA: Association for Computing Machinery, 1998, pp. 228–238. ISBN: 978-1-58113-024-9. DOI: 10.1145/289423.289448.

[83] François Pottier and Yann Régis-Gianas. "Stratified type inference for generalized algebraic data types". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. Charleston, South Carolina, USA: Association for Computing Machinery, 2006, pp. 232–244. ISBN: 1595930272. DOI: 10.1145/1111037.1111058.

[84] Ruchir Puri et al. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. 2021. arXiv: 2105.12655 [cs.SE].

[85] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462176.

[86] Ingkarat Rak-amnouykit et al. "Python 3 types in the wild: a tale of two type systems". In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 57–70. ISBN: 978-1-4503-8175-8. DOI: 10.1145/3426422.3426981.

[87] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. "The ins and outs of gradual type inference". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 481–494. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103714.

[88] Didier Rémy. "Simple, partial type-inference for System F based on type-containment". In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*. ICFP '05. Tallinn, Estonia: Association for Computing Machinery, 2005, pp. 130–143. ISBN: 1595930647. DOI: 10.1145/1086365.1086383.

[89] Gregor Richards, Ellen Arteca, and Alexi Turcotte. "The VM already knew that: leveraging compile-time knowledge to optimize gradual typing". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133879.

[90] Richard Roberts et al. “Transient Typechecks Are (Almost) Free”. In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, 5:1–5:28. ISBN: 978-3-95977-111-5. DOI: 10.4230/LIPIcs.ECOOP.2019.5.

[91] J Alan Robinson. “Computational logic: The unification computation”. In: *Machine intelligence* 6 (1971), pp. 63–72.

[92] Michael I. Schwartzbach. “Type inference with inequalities”. In: *TAPSOFT '91*. Ed. by S. Abramsky and T. S. E. Maibaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 441–455. ISBN: 978-3-540-46563-8.

[93] Tatsurou Sekiguchi and Akinori Yonezawa. “A complete type inference system for subtyped recursive types”. In: *Theoretical Aspects of Computer Software*. Ed. by Masami Hagiya and John C. Mitchell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 667–686. ISBN: 978-3-540-48383-0.

[94] Jeremy Siek and Walid Taha. “Gradual Typing for Objects”. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 2–27. ISBN: 978-3-540-73589-2.

[95] Jeremy G Siek and Walid Taha. “Gradual typing for functional languages”. In: *Scheme and functional programming workshop*. Vol. 6. 2006, pp. 81–92.

[96] Jeremy G. Siek et al. “Refined Criteria for Gradual Typing”. In: *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Ed. by Thomas Ball et al. Vol. 32. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 274–293. ISBN: 978-3-939897-80-4. DOI: 10.4230/LIPIcs.SNAPL.2015.274.

[97] Richard M. Stallman. “Bison: The Yacc-Compatible Parser Generator”. In: 2015. URL: https://www.gnu.org/software/bison/manual/ (visited on 04/01/2026).

[98] Stripe. *Sorbet*. URL: https://sorbet.org (visited on 04/01/2026).

[99] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F*". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 256–270. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837655.

[100] Asumu Takikawa et al. "Is sound gradual typing dead?" In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 456–468. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837630.

[101] Sam Tobin-Hochstadt and Matthias Felleisen. "The design and implementation of typed scheme". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 395–406. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328486.

[102] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. "LiquidHaskell: experience with refinement types in the real world". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 39–51. ISBN: 978-1-4503-3041-1. DOI: 10.1145/2633357.2633366.

[103] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[104] Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. "Optimizing and evaluating transient gradual typing". In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2019. Athens, Greece: Association for Computing Machinery, 2019, pp. 28–41. ISBN: 978-1-4503-6996-1. DOI: 10.1145/3359619.3359742.

[105] Michael M. Vitousek et al. "Design and evaluation of gradual typing for python". In: *Proceedings of the 10th ACM Symposium on Dynamic Languages*. DLS '14. Portland, Oregon, USA: Association for Computing Machinery, 2014, pp. 45–56. ISBN: 978-1-4503-3211-8. DOI: 10.1145/2661088.2661101.

[106] Markus Voelter and Vaclav Pech. "Language Modularity with the MPS Language Workbench". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 1449–1450. ISBN: 978-1-4673-1067-3.

[107] Markus Voelter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. ISBN: 978-1-4812-1858-0. URL: http://www.dslbook.org (visited on 04/01/2026).

[108] Dimitrios Vytiniotis et al. "OUTSIDEIN(X): modular type inference with local assumptions". In: *J. Funct. Program.* 21 (Sept. 2011), pp. 333–412. DOI: 10.1017/S0956796811000098.

[109] Fangke Ye et al. "Concrete Type Inference for Code Optimization using Machine Learning with SMT Solving". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622825.

[110] Lina Zhou et al. "A Method of Type Inference Based on Dataflow Analysis for Decompilation". In: *2009 International Conference on Computational Intelligence and Software Engineering*. 2009, pp. 1–4. DOI: 10.1109/CISE.2009.5362985.

[111] Tong Zhou et al. "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*. Onward! 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 65–83. ISBN: 978-1-4503-8178-9. DOI: 10.1145/3426428.3426915.