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

Senxi Li
The University of Tokyo, Japan

Tetsuro Yamazaki
The University of Tokyo, Japan

Shigeru Chiba
The University of Tokyo, Japan

Abstract

Supporting automatic type inference is in demand in modern language development. It is a challenging task but without appropriate supporting toolkits. This paper presents InferType, a Java library that helps implement constraint-based type inference. A compiler writer uses InferType's classes and methods to describe type constraints and typing rules for type inference. InferType then performs constraint solving by translation to the Z3 SMT solver. InferType is equipped with our developed optimization technique. It reduces the search space for type variables by pre-computing the structures of those type variables for mitigating the performance bottleneck of constraint solving with deeply nested types. We use InferType to implement type inference for a subset of Python, and conduct experiments to evaluate how the developed optimization technique can affect the performance of type inference. Our results show that InferType's optimization can greatly mitigate the performance bottleneck for programs with deeply nested types, and can potentially improve the performance for large nested types.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Theory of computation → Type theory

Keywords and phrases Domain Specific Languages, Compilation, Static Analysis, Type Inference, Constraint Solving, SMT Solver

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.23

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.11`

Funding This work was supported by JSPS KAKENHI Grant Numbers JP20H00578 and JP24H00688.

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 [22, 41], and type checker generators by language workbenches [45, 18]. Since many 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 [4, 42, 44, 14]. 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.

In this paper, we propose InferType, a domain-specific language embedded in Java for implementing constraint-based type inference. A compiler writer implements 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 an InferType's method 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 [7] 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 compatible 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.

The rest of this paper is organized as follows. Section 2 motivates the reader by a scenario of developing a small demonstrative language. Section 3 presents our proposal. Section 4 gives our experiment results by the implemented type inference engine using InferType. Section 5 relates our work to preceding researches and a brief conclusion ends this paper.

2 Implementing Type Inference in Language Development

Many 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 supports. Classical supporting tools for language development include lex, yacc [22] and bison [41], which are lexer and parser generators. There are also tool suites commonly called "language workbench", which are development environments for making domain-specific languages [45, 11]. The Spoofax [18] language workbench, for instance, supports code generators that produce type checkers and editor plugins from high-level language definitions. However, existing tools do not support code generators for 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 a constraint-based type inference. A number of existing type inference systems are formulated into a constraint-based approach because of its extensibility and flexibility [31, 35, 39, 38, 19]. We below consider implementing a constraint-based type inference for Mini-λ.

The syntax of expressions and types of Mini-λ is given in Listing 1.

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

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

Listing 1 Syntax of Mini-λ expressions and types

n, r and s range over integer, real numbers and string literals. x ranges over variables. `fun` x`.` e represents a function with parameter x and a body e. Mini-λ 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 Mini-λ, the plus operator can either add numbers or concatenate strings. The type syntax of Mini-λ consists of primitive types `int`, `float` and `str`, and the composite type `arrow` for describing function types.

Some of the typing rules in Mini-λ's type system are given in Figure 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.

An example program in Mini-λ is given in Listing 2. We will use this example program for demonstration in the rest of the paper because of its simplicity although this program is simple enough to be able to perform type inference in a bottom-up manner.

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

Listing 2 An example program in Mini-λ

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"`.

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

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

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

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

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

Figure 1 Part of the typing rules of Mini-λ.

To implement a constraint-based type inference for Mini-λ, 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 paper, we consider those type constraints in Listing 3.

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

Listing 3 Type constraints

`<:` represents the subtype relation between two types. ∧ and ∨ represent logical conjunction and disjunction; → and ¬ represent logical implication and negation.

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 Mini-λ. The compiler writer will generate the type constraints in Listing 4 for the program in Listing 2 by traversing its AST.

```
(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)
```

Listing 4 Type constraints for the example program

$t_?$ represents the unique type variables assigned to the variables and expressions in the program. $\texttt{t}_\texttt{f}$, $\texttt{t}_\texttt{x}$ and $\texttt{t}_\texttt{y}$ are assigned to parameter `f`, `x` and `y` of the defined functions. $\texttt{t}_\texttt{fx}$ and $\texttt{t}_\texttt{plus}$ are assigned to the resulting types of `f(x)` and `y + "a"`. $\texttt{t}_\texttt{e}$ is assigned to the resulting type of the whole expression. Constraint (1) and (2) arise from `T-App`. For example, in (1), $\texttt{t}_\texttt{f}$ is constrained to be a subtype of $\texttt{arrow<t}_\texttt{x}\texttt{, t}_\texttt{fx}\texttt{>}$ 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, $\texttt{t}_\texttt{y}$ and `str` (type of the string literal `"a"`), and the resulting type $\texttt{t}_\texttt{plus}$ must be subtype and supertype of either `float` or `str`, represented using logical connectives.

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 Mini-λ must consider `ST-Int-Float` and `ST-Arrow` in Figure 1 to solve the type constraints in Listing 4. Although well-known algorithms such as unification [37] and closure computation [33] 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 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 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 involved 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. InferType does not support type systems expressed by more than one type relation. For example, InferType could not support a gradual type system [40] which contains a subtype relation and also a type consistency relation.

Below in this section, we first demonstrate how compiler writers can use InferType to implement their type inference engines in Section 3.1. Then we show how InferType performs type inference by translation to Z3 in Section 3.2. In Section 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 paper.

3.1 Programming Interface

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

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 Mini-λ 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

```
inf.typevar("t_k")
```

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
```

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:

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

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. For example, constraint (3) in Listing 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 4 can be described and given to InferType in a similar manner.

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 3.1.1 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

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

A premise or conclusion is one single subtype relation created by `inf.subtype`.

```
premise, conclusion := inf.subtype(t1, t2)
```

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 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`.

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.

3.1.3 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();
```

`Typevar` is a subclass of `Type` representing only type variables. 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 yes, InferType outputs that binding as the type inference results. For Listing 2, it will output a `Map` object representing the following binding.

```
{tf ↦ arrow<str, str>, tx ↦ str, tfx ↦ str,
 ty ↦ str, tplus ↦ str, te ↦ 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.

```
if (!inf.untypableTypevars.isEmpty()){ // type error occurred
  Set<Typevar> untvars = inf.untypableTypevars();
  // 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.

3.1.4 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 phrase before calling `inf.solve` for constraint sovling.

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();
```

where `vehicleTy` is the type for class `Vehicle` described as `inf.type("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.

3.2 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 to 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.

3.2.1 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 Mini-λ, the data type declaration for `ztype` is generated as (represented using the SMT-LIBv2 [3] syntax)

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

It corresponds to t in Listing 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 are straightforwardly translated to Z3 boolean propositional formulae by the following translation function.

```
trans{inf.subtype(t1, t2)} = (zsubtype z1 z2)
trans{inf.and(c1, c2)} = (and trans{c1} trans{c2})
trans{inf.or(c1, c2)} = (or trans{c1} trans{c2})
trans{inf.implies(c1, c2)} = (=> trans{c1} trans{c2})
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 3.1.1 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.

3.2.2 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 [14].

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:

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

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<`$\texttt{t}_1$`, `$\texttt{t}_2$`>`, where $\texttt{t}_1$ and $\texttt{t}_2$ are type variables; but it does not compute subtypes or super types for an individual type such as `arrow<int, int>` or `arrow<int, `$\texttt{t}_1$`>`. Besides, InferType does not consider the reflexivity rule when enumerating subtypes and super types for composite types.

By this enumeration, in the case of Mini-λ, InferType enumerates `float` (by reflexivity) and `int` (by `ST-Int-Float`) as subtypes of `float`. By `ST-Arrow`, InferType enumerates `arrow<`$\texttt{t}_{11}$`, `$\texttt{t}_{12}$`>` as a subtype of `arrow<`$\texttt{t}_1$`, `$\texttt{t}_2$`>` under premises $\texttt{t}_1$ `<:` $\texttt{t}_{11}$ and $\texttt{t}_{12}$ `<:` $\texttt{t}_2$.

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

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

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

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

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 [20], which is a complete and minimal substitution. Existential quantifiers are added[1] 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.

[1] 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 paper.

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:

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

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:

$$\frac{t_1 \texttt{ <: } t \qquad t \texttt{ <: int}}{\texttt{list<}t_1\texttt{> <: intArray}}\ (\texttt{a and b})$$

Furthermore, InferType implicitly applies the transitivity rule and obtains:

$$\frac{t_1 \texttt{ <: int}}{\texttt{list<}t_1\texttt{> <: intArray}}\ (\texttt{a and b})$$

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:

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

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:

$$\frac{t_1 \texttt{ <: } t_2}{\texttt{list<}t_1\texttt{> <: array<array<}t_2\texttt{>>}}\ (\texttt{a and r})$$

InferType will further combine `a and 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) (z_0 ztype) (z_1 ztype) ...)
  (=> (zsubtype z type_T)
    (or (and (= z subtype_1) premise_1)
        (and (= z subtype_2) premise_2)
        ...)))
```

`(z ztype)` expresses that a Z3 variable `z` represents a type in the target language such as Mini-λ. `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 Mini-λ, 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 type variables in those unsat translated type constraints by calling `inf.untypableTypevars` in Section 3.1.3.

3.3 Optimizing Constraint Solving for Deeply Nested Types

Using Z3 in a straightforward way as in Section 3.2 works with respect to performance in common cases such as Listing 2. However, the constraint 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 Mini-λ given in Listing 5.

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

Listing 5 Another program with nested types in Mini-λ

This program extends Listing 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 paper, 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 3.2 performed constraint solving for the programs with `f3` in 0.18s, `f4` in 14.07s and `f5` in 179m15.35s!

We expect that this slow down 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 Mini-λ, 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 `?`.

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

`*` represents only primitive types; it can never be any composite type. `?` represents composite-type names such as "`arrow`" in Mini-λ, which constructs a nested structure for shapes by taking other shapes as arguments. For example, suppose that the shape of a type variable in Mini-λ 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>`, ...

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.

```
shape-eq := shape == shape
          | shape-eq ∧ shape-eq
          | shape-eq ∨ shape-eq
```

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. Shape equations are derived by the following function:

```
derive{t_1 <: t_2} = shapeof{t_1} == shapeof{t_2}
derive{¬ t_1 <: t_2} = TRUE
derive{c_1 ∧ c_2} = derive{c_1} ∧ derive{c_2}
derive{c_1 ∨ c_2} = derive{c_1} ∨ derive{c_2}
```

The input is the Negation Normal Form (NNF) converted from the given type constraints in the form of Listing 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. 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 ¬ s_h `==` `*` cannot be derived even if a type constraint ¬ t_h `<: int` is given. s_h can be arbitrary because t_h can be any type except a subtype of `int`.

For instance, the shape equations in Listing 6 are derived from the given type constraints in Listing 4.

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

Listing 6 Shape equations for the type constraints

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 substituions 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 6:

```
{sf ↦ ?<*, *>, sx ↦ *, sfx ↦ *,
 sy ↦ *, splus ↦ *, se ↦ ?<*, *>}
```

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 3.3.2. 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. Then, 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.

3.3.2 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 $\texttt{t}_\texttt{f}$ in Listing 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 $\texttt{z}_\texttt{f}$ 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 Mini-λ. The arguments `sh_1` and `sh_2` are generated Z3 constants.

Suppose that Mini-λ defined list types `list<`t`>` and array types `array<`t`>`, and also that the shape of a type variable $\texttt{t}_\texttt{h}$ 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))
```

$\texttt{z\_h}$ is the translated Z3 constant for $\texttt{t}_\texttt{h}$. `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.

For example, consider the following type constraints in the Mini-λ extended with type `object` where an arrow type is a subtype of the `object` type:

```
(21) arrow<int, int> <: tc
(22) object <: tc ∨ td <: tc
(23) object <: td
```

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') ?<*, *> == sc
(22') * == sc ∨ sd == sc
(23') * == sd
```

By applying the approximate shape computation, (22') will be excluded because 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 because 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 2 times when our approximate shape computation encounters this limitation.

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 because 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 Mini-λ 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 $\mathtt{t_a}$ can be only inferred as `object` for satisfaction. The shape computation will compute the shape for $\mathtt{t_a}$ as `?<*, *>` from the derived shape equations:

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

However, InferType then could not find a binding for $\mathtt{t_a}$ with the asserted extra Z3 formulae because $\mathtt{t_a}$ 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 $\mathtt{t_a}$ as type `object`, a type constraint $\mathtt{t_a}$ `<: 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 3.3.1, $\mathtt{s_a}$ will lose a binding in the shape computation because the derived shape equations including $\mathtt{s_a}$ cannot be solved by unification. InferType would then correctly perform type inference although the constraint solving for $\mathtt{t_a}$ 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 3.3.2. Although the constraint solving is entirely unoptimized in such cases, InferType would be able to correctly compute a binding as the type inference result.

4 Experiment

We implemented a type inference engine for Mini-λ using InferType. We include it in our artifact on Zenodo [2]. The source code contains 291 LOC in Java with (manually counted) 48 LOC using InferType. Owing to InferType's optimization, the implemented engine performed type inference within around 0.18s for each program with nested types `f3`, `f4` and `f5` as shown in Listing 5, 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. We choose Python as the target language because we

[2] `https://zenodo.org/records/10981733`

Table 1 Dataset overview. Each value is the average number per program/project in each set.

	set1-Typpete	set2-fs	set3-CodeNet	set4-large	set5-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

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 concern of choosing Python is being aware of an existing work Typpete [14]. Typpete is one of the state-of-the-art type inference engines for Python using the Z3 SMT solver. Comparing a type inference engine by InferType with generated Z3 code 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 (manually counted) 226 LOC using InferType. It uses the Python `ast` module 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 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 principle types by manually encoding soft and hard constraints in syntax-related typing rules, while principle typing is generally not considered by InferType. Nevertheless, Subpet could be 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 (set1-Typpete) contains 44 Python benchmark programs obtained from Typpete's artifact [3]. 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 (set2-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)
```

[3] https://zenodo.org/records/3996670

`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 5 in Mini-λ. The program can be further extended with more function definitions `f2`, `f3`, ... for larger nested types similar to what we showed in Mini-λ. We created the cases up to `f7`.

The third set (set3-CodeNet) contains 30 programs obtained from *Project CodeNet* [36]. Project CodeNet is a 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 at Sep. 6, 2023. We used the `grep` command to search programs containing keywords `graph` and `dijkstra`. Then we obtained 12 problem sets that had more than 10 hits of the two keywords. We used 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 [16]. Then 5 programs were randomly picked up from each problem set (each problem set contains $100 \sim 300$ solution programs), resulting in 60 programs. 30 of the 60 programs were filtered by the implementation limitation of Subpet (mostly because of unknown library usage with respect to types). The remaining 30 programs were finally included in the third set.

The fourth set (set4-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 were collected from the MOPSA project [4]. A few code modifications were made to overcome the implementation limitations of Subpet. The other two projects were 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 (set5-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 set1-Typpete. 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 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 line 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.

4.1 Comparing Subpet with Typpete

Firstly, we show the performance of type inference by Subpet and Typpete. Table 2 shows the time results by running the two engines over set1-Typpete and set2-fs in our dataset. We selected and included some of the larger programs by LOC in set1-Typpete in Table 2a. The experiments were conducted on a Ubuntu 19.10 machine with 2.8 GHz Intel(R) Core(TM) i7-6700T CPU and 32GB RAM, equipped with OpenJDK 14, Python 3.9.6, Z3 4.12.2 and Typpete 0.1. The results from Subpet were 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 modified Typpete to loop 20 times for each input Python program and then took the average of the later 10 runs.

[4] `https://mopsa.lip6.fr`

Table 2 Time for Type Inference by Typpete and Subpet. †timeout (after 1,000s)

(a) set1-Typpete.

Name(.py)	LOC	Typpete	Subpet
bellman_ford	61	2.54s	0.23s
crafting_challenge	132	3.32s	0.27s
deceitful	36	1.41s	0.12s
disjoint_sets	45	2.20s	0.20s
lattice	81	2.50s	0.16s
rockpaperscissor	79	2.87s	0.13s
vehicle	92	2.65s	0.14s

(b) set2-fs.

Name(.py)	LOC	Typpete	Subpet
prog_f1	8	0.85s	0.09s
prog_f2	11	1.17s	0.11s
prog_f3	15	1.51s	0.11s
prog_f4	17	6.52s	0.17s
prog_f5	20	101.4s	0.28s
prog_f6	23	†	1.04s
prog_f7	27	†	3.08s

Both Subpet and Typpete could infer the types for normal programs without nested types or with smaller nested types (programs in set1-Typpete and prog_f1 to prog_f3 in set2-fs) within a reasonable time period. Subpet was faster than Typpete mainly because of the language advantage: Subpet is implemented in Java and Typpete is implemented in Python. However, Typpete exposed its performance downside for prog_f4 to prog_f7 in set2-fs with deeply nested types. Subpet could finish their type inference much faster. Typpete ran prog_f5 within 101.4s; it ran into timeout (†) for prog_f6 and prog_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.

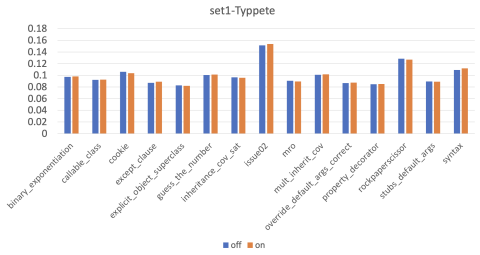
We also demonstrate the behaviour of handling type errors by Subpet. It can detect the type errors for all the ill-typed programs in set5-ill. Subpet tracks the types and type variables to program locations (including the file name, line number and column number) in a `Map` object during constraint generation 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 resource. For example, one of the ill-typed programs in set5-ill is written as

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

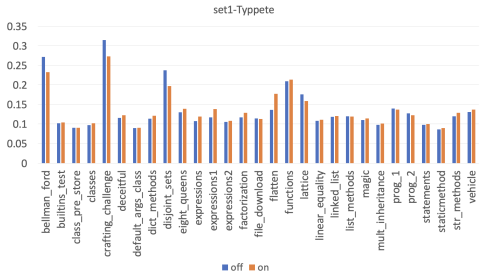
This program is ill-typed because function `f` is called without argument but it is defined as taking one argument. Subpet reports the type error message as

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

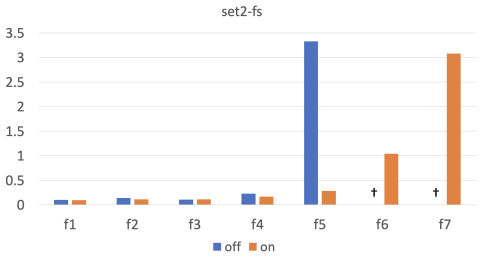
Typpete would report a similar type error message to the above by Subpet. Note that InferType can detect and help report type errors whenever the optimization is disabled or enabled. InferType would not wrongly infer types with our shape computation if the program is ill-typed.



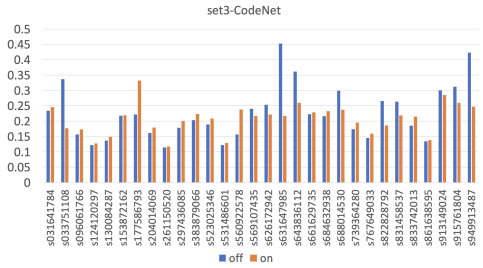
(a) set1-Typpete (no extra formula).



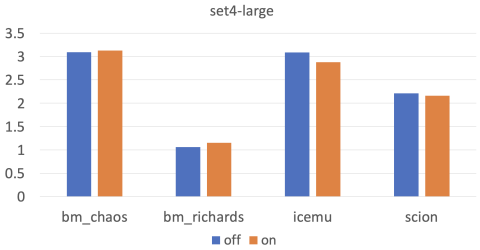
(b) set1-Typpete.



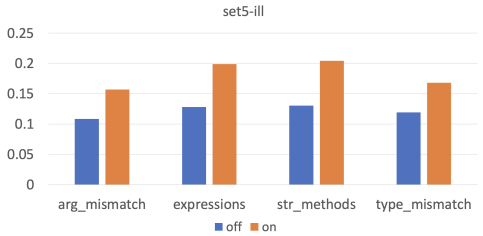
(c) set2-fs.



(d) set3-CodeNet.



(e) set4-large.



(f) set5-ill.

Figure 2 Time for Type Inference by disabling and enabling InferType's shape computation. *off* columns are results by disabling shape computation. *on* columns are results by enabling shape computation. y-axes are the elapsed time in second. x-axes are program names. Figure 2a gives the results where the shape computation is enabled in *on* but extra Z3 formulae are not added in constraint solving. The other subfigures gives the results where extra Z3 formulae are added in *on*. †timeout (after 1,000s)

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 to validate how our shape computation can affect the performance of type inference. Figure 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 shown in Section 3.2.

The results from Figure 2a show that InferType gives a small performance overhead by the shape computation. Figure 2a gives the results for part of the programs in set1-Typpete. Particularly, extra Z3 formulae were not asserted in the constraint solving when the shape computation was enabled in the programs in Figure 2a for evaluating the overhead of the shape computation. Extra Z3 formulae were asserted for all the other programs when the shape computation was enabled in the other subfigures in Figure 2. In average, Subpet degraded the performance of type inference by 0.23% with the shape computation than

without the shape computation among all the programs in Figure 2a. Here and below, the performance rate is calculated as $(t_{off} - t_{on}) / t_{off}$ for each program, where t_{off} and t_{on} are the elapsed time when the shape computation is disabled and enabled. This 0.23% was the overhead of InferType's shape computation for computing the shapes.

The results from Figure 2c show that InferType's optimization can greatly mitigate the performance bottleneck for constraint solving containing deeply nested types. When the shape computation was enabled, Subpet performed type inference for all the programs in set2-fs within a reasonable time period. When the shape computation was disabled, Subpet performed much slower for prog_f5; it could not finish prog_f6 or prog_f7 within a time limit. These results were similar to those obtained by Typpete given in Table 2b. It indicates that the better performance by Subpet against Typpete was not simply because of 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 outperformed that without the shape computation in 8/29 (27.6%) programs among Figure 2b (set1-Typpete). Among these 8 programs that Subpet with the shape computation gave a better performance, it outperformed by an average of 8.87%. For example, program bellman_ford (containing a nested list of list type) and crafting_challenge (containing a nested dict of dict type) in Figure 2b had a better performance by 16.7% and 15.1%. Subpet with the shape computation outperformed that without shape computation in 11/30 (36.7%) programs among Figure 2d (set3-CodeNet). Among these 11 programs, it outperformed by an average of 40.9%. In the best case (s631647985), it accelerated the performance of type inference by 108.2%. The programs in set3-CodeNet have more nested types than programs in set1-Typpte because set3-CodeNet was collected on purpose by specific keyword search. Since set3-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 a 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 degraded the performance of the two programs s177586793 and s560922578 by 49.6% and 51.6% in Figure 2d.

Our experiments empirically show the scalability of the shape computation to larger programs. Figure 2e presents the time results of type inference for the collected programs/-projects in set4-large. Subpet took around 2 seconds for type inference in average, which is 10 times longer than the average elapsed time for the programs in set1-Typpte and set3-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 2f gives the elapsed time of type inference for the ill-typed programs in our dataset. Subpet took a longer time to find the type error in type inference when the shape computation was 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 was disabled or enabled.

Table 3 Average number of computed shapes and average shape computation time. Each data cell shows the results by Approximate / Precise shape computation.

	set1-Typpete	set2-fs	set3-CodeNet	set4-large	set5-ill
shape	23 / 35	17 / 28	41 / 69	370 / 443	17 / 25
time	0.91ms / 74.3ms	0.71ms / 30.4ms	1.75ms / 434ms	68.7ms / 2.58s	0.69ms / 38.9ms

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 6 (which are derived from Listing 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 3.3.2. Our approximate shape computation is implemented in Java.

The approximate shape computation computes more than half of the shapes that the precise shape computation computes in average among all the programs in the dataset. The first row in Table 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 computed 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 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 was the average of the later 10 runs by looping 20 times. The elapsed time of the approximate and precise shape computation differed in scale.

In our dataset, we did not find a program that had a faster elapsed time for type inference with the precise shape computation than approximate by Subpet, though the precise shape computation could compute more shapes in 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 did not find a case such that Subpet performed a type inference failure caused by the limitation of the approximate shape computation as we discussed in Section 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 Section 3.3.2, by excluding whose derived shape equation the approximate shape computation would 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.

5 Related Work

Language development is enjoying significant growth in number and diversity both by academia and industry. Language workbenches such as MPS [45] and Xtext [11] are development environments that provide high-level mechanisms for implementing domain-specific languages [12]. Spoofax [18] allows language developers to write declarative specifications of language definitions to produce parsers, interpreters and editor plugins. Efftinge [10] presented a Java framework, Xbase, for implementing DSLs based on Xtext. Recent studies proposed a standardization of name resolution and type checking with a constraint-based approach integrated in Spoofax [1, 43]. Unlike existing language development toolkits such as Spoofax, our proposed tool produces constraint-based type inference.

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 [31, 35, 39, 38, 19, 27]. 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 [15, 23, 6]. 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 [32, 26], which is regarded as local type inference, developed by carefully controlling the introduction and elimination of type variables for inferring parameter types. Turnstile [5] 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 introduced wobbly types [30] to distribute over type constructors for handling GADT type inference using type annotations. Later, Pottier [34] 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 are similar to those in [34, 46]. 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 [4, 3]. State-of-the-art SMT solvers include Yices [9], CVC5 [2], and Z3 [7]. InferType currently relies on Z3 for performing constraint solving. Translating InferType expressions to a higher-level language such as JavaSMT [17] 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 [8]. Later, Niemetz [25] presented an approach for accelerating quantified constraint solving. The developed optimization in our proposal is rather 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 [42] presented the language F* with a dependent type and effect system using a combination of SMT solving and manual proofs. Vazou and Jhala [44] 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 [28] 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 [14] proposed a type inference engine, Typpete, that generates Python 3 type annotations by encoding type constraints as a MaxSMT problem using Z3. In this paper, we implemented a subset of Typpete by using the proposed tool for the experiments. There are also emerging studies of statically typing Python programs based on other techniques [21, 13, 24, 29].

6 Conclusion

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 developed an optimization technique for InferType to relieve the performance bottleneck for better practical usage. 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 demonstrated the usage of InferType and experimented the effectiveness of its optimization by implementing a type inference engine for a Python subset using InferType. We found that, the implemented engine had compatible performance of type inference compared with a state-of-the-art type inference engine for Python using Z3 with a manual encoding of Z3 formulae. InferType's optimization could greatly improve the performance for programs with deeply nested types. We also observed that InferType could potentially improve the performance of type inference for programs containing nested types. We believe that InferType is practically useful to help implement constraint-based type inference for language development.

References

1 Hendrik Antwerpen, Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60, January 2016. doi:10.1145/2847538.2847543.

2 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

3 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)*, volume 13, page 14, 2010.

4 Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

5 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, pages 694–705, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009886.

6 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, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/582153.582176.

7 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

8 David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting Symmetry in SMT Problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 222–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

9 Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.

10 Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-Specific Languages for Java. *SIGPLAN Not.*, 48(3):112–121, September 2012. doi:10.1145/2480361.2371419.

11 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, pages 307–309, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869542.1869625.

12 Martin Fowler and R Parsons. Addison-Wesley signature, Domain specific languages . Massachussets, 2010.

13 google. pytype, 2021. URL: https://google.github.io/pytype/.

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

15 R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. URL: http://www.jstor.org/stable/1995158.

16 Donald B. Johnson. A Note on Dijkstra's Shortest Path Algorithm. *J. ACM*, 20(3):385–388, July 1973. doi:10.1145/321765.321768.

17 Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. JavaSMT: A Unified Interface for SMT Solvers in Java. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 139–148, Cham, 2016. Springer International Publishing.

18 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, pages 444–463, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869459.1869497.

19 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, pages 112–125, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426422.3426985.

20 Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Comput. Surv.*, 21(1):93–124, March 1989. doi:10.1145/62029.62030.

21 Jukka Lehtosalo, Guido van Rossum, and Ivan Levkivskyi. mypy, June 2012. URL: http://mypy-lang.org/.

22 John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc (2nd Ed.)*. O'Reilly & Associates, Inc., USA, 1992.

23 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

24 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.17.

25 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Syntax-Guided Quantifier Instantiation. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163, Cham, 2021. Springer International Publishing.

26 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, pages 41–53, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360207.

27 Lionel Parreaux. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3409006.

28 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, pages 525–542, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660230.

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

30 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, September 2006. doi:10.1145/1160074.1159811.

31 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

32 Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. doi:10.1145/345099.345100.

33 François Pottier. A Framework for Type Inference with Subtyping. *SIGPLAN Not.*, 34(1):228–238, September 1998. doi:10.1145/291251.289448.

34 François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. *SIGPLAN Not.*, 41(1):232–244, January 2006. doi:10.1145/1111320.1111058.

35 Francois Pottier and Didier Remy. *The Essence of ML Type Inference*, pages 389–489. MIT press, January 2005. doi:10.7551/mitpress/1104.003.0016.

36 Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks, 2021. arXiv:2105.12655.

37 J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.

38 Michael I. Schwartzbach. Type inference with inequalities. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91*, pages 441–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

39 Tatsurou Sekiguchi and Akinori Yonezawa. A complete type inference system for subtyped recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 667–686, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

40 Jeremy Siek and Walid Taha. Gradual Typing for Objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

41 Richard M. Stallman. Bison: The Yacc-Compatible Parser Generator, 2015. URL: `https://api.semanticscholar.org/CorpusID:60543798`.

42 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. *SIGPLAN Not.*, 51(1):256–270, January 2016. `doi:10.1145/2914770.2837655`.

43 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as Types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. `doi:10.1145/3276484`.

44 Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with Refinement Types in the Real World. *SIGPLAN Not.*, 49(12):39–51, September 2014. `doi:10.1145/2775050.2633366`.

45 Markus Voelter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1449–1450. IEEE Press, 2012.

46 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OUTSIDEIN(X): modular type inference with local assumptions. *J. Funct. Program.*, 21:333–412, September 2011. `doi:10.1017/S0956796811000098`.