MaxSMT-Based Type Inference for Python 3

Mostafa Hassan[1,2], Caterina Urban[2], Marco Eilers[2], and Peter Müller[2]

[1] German University in Cairo, Egypt
[2] Department of Computer Science, ETH Zurich, Switzerland

Abstract. We present Typpete, a sound type inferencer that automatically infers Python 3 type annotations. Typpete encodes type constraints as a MaxSMT problem and uses optional constraints and specific quantifier instantiation patterns to make the constraint solving process efficient. Our experimental evaluation shows that Typpete scales to real world Python programs and outperforms state-of-the-art tools.

1 Introduction

Dynamically-typed languages like Python have become increasingly popular in the past five years. Dynamic typing enables rapid development and adaptation to changing requirements. On the other hand, static typing offers early error detection, efficient execution, and machine-checked code documentation, and enables more advanced static analysis and verification approaches [15].

For these reasons, Python's PEP484 [25] has recently introduced optional type annotations in the spirit of gradual typing [23]. The annotations can be checked using MyPy [10]. In this paper, we present our tool Typpete, which automatically infers sound (non-gradual) type annotations and can therefore serve as a preprocessor for other analysis or verification tools.

Typpete performs whole-program type inference, as there are no principal typings in object-oriented languages like Python [1, example in Section 1]; the inferred types are correct in the given context but may not be as general as possible. The type inference is constraint-based and relies on the off-the-shelf SMT solver Z3 [7] for finding a valid type assignment for the input program. We show that two main ingredients allow Typpete to scale to real programs: (1) a careful encoding of subtyping that leverages efficient quantifier instantiation techniques [6], and (2) the use of optional type equality constraints, which considerably reduce the solution search space. Whenever a valid type assignment for the input program cannot be found, Typpete encodes type error localization as an optimization problem [19] and reports only a minimal set of unfulfilled constraints to help the user pinpoint the cause of the error.

Typpete accepts programs written in (a large subset of) Python 3. Having a static type system imposes a number of requirements on Python programs: (a) a variable can only have a single type through the whole program; (b) generic types have to be homogeneous (e.g., all elements of a set must have the same type); and (c) dynamic code generation, reflection and dynamic attribute additions and deletions are not allowed. The supported type system includes generic classes

```
class Item(metaclass=ABCMeta):
    @abstractmethod
    def compete(self, item):
        pass

    def evalEven(self, item):
        return ”WIN”

    def evalOdd(self, item):
        return ”LOSE”

class Even(Item):
    def compete(self, item):
        return item.evalEven(self)

class Odd(Item):
    def compete(self, item):
        return item.evalOdd(self)

def match(item1, item2):
    return item1.compete(item2)
```

Fig. 1. A Python implementation of the *Odds and Evens* hand game.

and functions. Users must supply a file and the *number* of type variables for any generic class or function. Typpete then outputs a program with type annotations, a type error, or an error indicating use of unsupported language features.

Our experimental evaluation demonstrates the practical applicability of our approach. We show that Typpete performs well on a variety of real-world open source Python programs and outperforms state-of-the-art tools.

2 Constraint Generation

Typpete encodes the type inference problem for a Python program into an SMT constraint resolution problem such that any solution of the SMT problem yields a valid type assignment for the program. The process of generating the SMT problem consists of three phases, which we describe below.

In a first pass over the input program, Typpete collects: (1) all globally defined names (to resolve forward references), (2) all classes and their respective subclass relations (to define subtyping), and (3) upper bounds on the size of certain types (e.g., tuples and function parameters). This pre-analysis encompasses both the input program — including all transitively imported modules — and *stub files*, which define the types of built-in classes and functions as well as libraries. Typpete already contains stubs for the most common built-ins; users can add custom stub files written in the format that is supported by MyPy.

In the second phase, Typpete declares an algebraic datatype Type, whose members correspond one-to-one to Python types. Typpete declares one datatype constructor for every class in the input program; non-generic classes are represented as constants, whereas a generic class with n type parameters is represented by a constructor taking n arguments of type Type. As an example, the class Odd in Figure 1 is represented by the constant $\mathsf{class}_{\mathsf{Odd}}$. Typpete also declares constructors for tuples and functions up to the maximum size determined in the pre-analysis, and for all type variables used in generic functions and classes.

The subtype relation $<:$ is represented by an uninterpreted function subtype which maps pairs of types to a boolean value. This function is delicate to define because of the possibility of *matching loops* (i.e., axioms being endlessly instantiated [7]) in the SMT solver. For each datatype constructor, Typpete generates axioms that explicitly enumerate the possible subtypes and supertypes. As an example, for the type $\mathsf{class}_{\mathsf{Odd}}$, Typpete generates the following axioms:

$$\forall t.\, \mathsf{subtype}(\mathsf{class}_{\mathsf{Odd}}, t) = (t = \mathsf{class}_{\mathsf{Odd}} \;\vee\; t = \mathsf{class}_{\mathsf{Item}} \;\vee\; t = \mathsf{class}_{\mathsf{object}})$$
$$\forall t.\, \mathsf{subtype}(t, \mathsf{class}_{\mathsf{Odd}}) = (t = \mathsf{class}_{\mathsf{none}} \;\vee\; t = \mathsf{class}_{\mathsf{Odd}})$$

Note that the second axiom allows None to be a subtype of any other type (as in Java). As we discuss in the next section, this definition of subtype allows us to avoid matching loops by specifying specific instantiation patterns for the SMT solver. A *substitution* function substitute, which substitutes type arguments for type variables when interacting with generic types, is defined in a similar way.

In the third step, Typpete traverses the program while creating an SMT variable for each node in its abstract syntax tree, and generating type constraints over these variables for the constructs in the program. During the traversal, a *context* maps all defined names (i.e., program variables, fields, etc.) to the corresponding SMT variables. The context is later used to retrieve the type assigned by the SMT solver to each name in the program. Constraints are generated for expressions (e.g., call arguments are subtypes of the corresponding parameter types), statements (e.g., the right-hand side of an assignment is a subtype of the left hand-side), and larger constructs such as methods (e.g., covariance and contravariance constraints for method overrides). For example, the (simplified) constraint generated for the call to `item1.compete(item2)` at line 21 in Figure 1 contains a disjunction of cases depending on the type of the receiver:

$$(\mathsf{v}_{\mathsf{item1}} = \mathsf{class}_{\mathsf{Odd}} \wedge \mathsf{compete}_{\mathsf{Odd}} = \mathsf{f\_2}(\mathsf{class}_{\mathsf{Odd}}, \mathsf{arg}, \mathsf{ret}) \wedge \mathsf{subtype}(\mathsf{v}_{\mathsf{item2}}, \mathsf{arg}))$$
$$\vee\; (\mathsf{v}_{\mathsf{item1}} = \mathsf{class}_{\mathsf{Even}} \wedge \mathsf{compete}_{\mathsf{Even}} = \mathsf{f\_2}(\mathsf{class}_{\mathsf{Even}}, \mathsf{arg}, \mathsf{ret}) \wedge \mathsf{subtype}(\mathsf{v}_{\mathsf{item2}}, \mathsf{arg}))$$

where f_2 is a datatype constructor for a function with two parameter types (and one return type ret), and $\mathsf{v}_{\mathsf{item1}}$ and $\mathsf{v}_{\mathsf{item2}}$ are the SMT variables corresponding to `item1` and `item2`, respectively.

The generated constraints guarantee that any solution yields a correct type assignment for the input program. However, there are often many different valid solutions, as the constraints only impose lower or upper bounds on the types represented by the SMT variables (e.g., $\mathsf{subtype}(\mathsf{v}_{\mathsf{item2}}, \mathsf{arg})$ shown above imposes only an upper bound on the type of $\mathsf{v}_{\mathsf{item2}}$). This has an impact on performance (cf. Section 4) as the search space for a solution remains large. Moreover, some type assignments could be more desirable than others for a user (e.g., a user would most likely prefer to assign type int rather than object to a variable initialized with value zero). To avoid these problems, Typpete additionally generates optional type *equality* constraints in places where the mandatory constraints only demand subtyping (i.e., local variable assignments, return statements, passed function arguments), thereby turning the SMT problem into a MaxSMT optimization problem. For instance, in addition to $\mathsf{subtype}(\mathsf{v}_{\mathsf{item2}}, \mathsf{arg})$ shown above, Typpete generates the optional equality constraint $\mathsf{v}_{\mathsf{item2}} = \mathsf{arg}$. The optional constraints guide the solver to try the specified exact type first, which is often a correct choice and therefore improves performance, and additionally leads to solutions with more precise variable and parameter types.

3 Constraint Solving

Typpete relies on Z3 [7] and the MaxRes [18] algorithm for solving the generated type constraints. We use *e-matching* [6] for instantiating the quantifiers used in the axiomatization of the subtype function (cf. Section 2), and carefully choose instantiation patterns that ensure that any choice made during the search immediately triggers the instantiation of the relevant quantifiers. For instance, for the axioms shown in Section 2, we use the instantiation patterns subtype($\text{class}_{\text{Odd}}$, t) and subtype(t, $\text{class}_{\text{Odd}}$), respectively. Our instantiation patterns ensure that as soon as one argument of an application of the subtype function is known, the quantifier that enumerates the possible values of the other argument is instantiated, thus ensuring that the consequences of any type choices propagate immediately. With a naïve encoding, the solver would have to *guess* both arguments before being able to *check* whether the subtype relation holds. The resulting constraint solving process is much faster than it would be when using different quantifier instantiation techniques such as *model-based quantifier instantiation* [12], but still avoids the potential unsoundness that can occur when using e-matching with insufficient trigger expressions.

When the MaxSMT problem is satisfiable, Typpete queries Z3 for a model satisfying all type constraints, retrieves the types assigned to each name in the program, and generates type annotated source code for the input program. For instance, for the program shown in Figure 1, Typpete automatically annotates the function `evalEven` with type Even for the parameter `item` and a str return type. Note that Item and object would also be correct type annotations for `item`; the choice of Even is guided by the optional type equality constraints.

When the MaxSMT problem is unsatisfiable, instead of reporting the unfulfilled constraints in the *unsatistiable core* returned by Z3 (which is not guaranteed to be minimal), Typpete creates a new *relaxed* MaxSMT problem where only the constraints defining the subtype function are enforced, while all other type constraints are optional. Z3 is then queried for a model satisfying as many type constraints as possible. The resulting type annotated source code for the input program is returned along with the remaining minimal set of unfulfilled type constraints. For instance, if we remove the abstract method `compete` of class `Item` in Figure 1, Typpete annotates the parameters of the function `match` at line 20 with type object and indicates the call `compete` at line 21 as problematic. By observing the mismatch between the type annotations and the method call, the user has sufficient context to quickly identify and correct the type error.

4 Experimental Evaluation

In order to demonstrate the practical applicability of our approach, we evaluated our tool Typpete on a number of real-world open-source Python programs that use inheritance, operator overloading, and other features that are challenging for type inference (but not features that make static typing impossible):

	T(SMT)	T(MaxSMT)	Unfulfilled	T(Relaxed)	Pytype
adventure	2.99s / 6.30s	3.27s / 6.76s	42 / 2	1.95s / 8.83s	0 [0]
icemu	9.45s / 6.79s	9.51s / 3.63s	4 / 2	0.08s / 21.76s	18 [2]
imp	16.88s / 59.95s	16.91s / 15.87s	67 / 2	0.82s / 82.56s	3 [2]
scion	4.65s / 3.35s	4.72s / 2.97s	28 / 2	0.16s / 3.39s	0 [0]
test suite	14.66s / 1.63s	14.66s / 2.17s	-	-	55 [34]

Fig. 2. Evaluation of Typpete on small programs and larger open source projects.

adventure [21]**:** An implementation of the *Colossal Cave Adventure* game (2 modules, 399 LOC). The evaluation (and reported LOC) excludes the modules `game.py` and `prompt.py`, which employ dynamic attribute additions.

icemu [8]**:** A library that emulates integrated circuits at the logic level (8 modules, 530 LOC). We conducted the evaluation on revision 484828f.

imp [4]**:** A minimal interpreter for the `imp` toy language (7 modules, 771 LOC). The evaluation excludes the modules used for testing the project.

scion [9]**:** A Python implementation of a new Internet architecture (2 modules, 725 LOC). For the evaluation, we used `path_store.py` and `scion_addr.py` from revision 6f60ccc, and provided stub files for all dependencies.

We additionally ran Typpete on our test suite of manually-written programs and small programs collected from the web (47 modules and 1998 LOC).

In order to make the projects statically typeable, we had to make a number of small changes that do not impact the functionality of the code, such as adding abstract superclasses and abstract methods, and (for the **imp** and **scion** projects) introducing explicit downcasts in few places. Additionally, we made a number of other innocuous changes to overcome the current limitations of our tool, such as replacing keyword arguments with positional arguments, replacing generator expressions with list comprehensions, and replacing super calls via inlining. The complete list of changes for each project is included in our artifact.

The experiments were conducted on an 2,9 GHz Intel Core i5 processor with 8GB of RAM running Mac OS High Sierra version 10.13.3 with Z3 version 4.5.1. Figure 2 summarizes the result of the evaluation. The first two columns show the *average running time* (over ten runs, split into constraint generation and constraint solving) for the type inference in which the use of optional type equality constraints (cf. Section 2) is disabled (SMT) and enabled (MaxSMT), respectively. We can observe that optional type equality constraints (considerably) reduce the search space for a solution as disabling them significantly increases the running time for larger projects. We can also note that the constraint solving time improves significantly when the type inference is run on the test suite, which consists of many independent modules. This suggests that splitting the type inference problem into independent sub-problems could further improve performance. We plan to investigate this direction as part of our future work.

The third column of Figure 2 shows the evaluation of the *error reporting* feature of Typpete (cf. Section 3). For each benchmark, we manually introduced two type errors that could organically happen during programming and compared the size of the unsatisfiable core (left of /) and the number of remaining unfulfilled constraints (right of /) for the original and relaxed MaxSMT problems given to Z3, respectively. We also list the times needed to prove the first problem unsatisfiable and solve the relaxed problem. As one would expect, the number of constraints that remain unfulfilled for the relaxed problems is considerably smaller, which demonstrates that the error reporting feature of Typpete greatly reduces the time that a user needs to identify the source of a type error.

Finally, the last column of Figure 2 shows the result of the *comparison* of Typpete with the state-of-the-art type inferencer Pytype [16]. Pytype infers PEP484 [25] gradual type annotations by abstract interpretation [5] of the bytecode-compiled version of the given Python file. In Figure 2, for the considered benchmarks, we report the number of variables and parameters that Pytype leaves untyped or annotated with Any. We excluded any module on which Pytype yields an error; in square brackets we indicate the number of modules that we could consider. Typpete is able to fully type all elements and thus outperforms Pytype *for static typing purposes*. On the other hand, we note that Pytype additionally supports gradual typing and a larger Python subset.

5 Related and Future Work

In addition to Pytype, a number of other type inference approaches and tools have been developed for Python. The approach of Maia et al. [17] has some fundamental limitations such as not allowing forward references or overloaded functions and operators. Fritz and Hage [11] as well as Starkiller [22] infer sets of *concrete types* that can inhabit each program variable to improve execution performance. The former sacrifices soundness to handle more dynamic features of Python. Additionally, deriving valid type assignments from sets of concrete types is non-trivial. MyPy and a project by Cannon [3] can perform (incomplete) type inference for local variables, but require type annotations for function parameters and return types. PyAnnotate [13] *dynamically* tracks variable types during execution and optionally annotates Python programs; the resulting annotations are not guaranteed to be sound. A similar spectrum of solutions exists for other dynamic programming languages like JavaScript [2,14] and ActionScript [20].

The idea of using SMT solvers for type inference is not new. Both F* [24] and LiquidHaskell [26] (partly) use SMT-solving in the inference for their dependent type systems. Pavlinovic et al. [19] present an SMT encoding of the OCaml type system. Typpete's approach to type error reporting can be seen as a simple instantiation of their approach.

As part of our future work, we want to explore whether our system can be adapted to infer gradual types. We also aim to develop heuristics for inferring which functions and classes should be annotated with generic types based on the reported unfulfilled constraints. Finally, we plan to explore the idea of splitting the type inference into multiple separate problems to improve performance.

Acknowledgments. We thank the anonymous reviewers for their feedback. This work was supported by an ETH Zurich Career Seed Grant (SEED-32 16-2).

References

1. D. Ancona and E. Zucca. Principal Typings for Java-like Languages. In *POPL*, pages 306–317, 2004.
2. C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.
3. B. Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, 2005.
4. J. Conrod. IMP Interpreter. `https://github.com/jayconrod/imp-interpreter`.
5. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
6. L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In *CADE*, pages 183–198, 2007.
7. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.
8. V. Dupras. Icemu. `https://github.com/hsoft/icemu`.
9. ETH Zurich. SCION. `https://github.com/scionproto/scion`.
10. D. Fisher, J. Lehtosalo, G. Price, and G. van Rossum. MyPy. `http://mypy-lang.org/`.
11. L. Fritz and J. Hage. Cost Versus Precision for Approximate Typing for Python. In *PEPM*, pages 89–98, 2017.
12. Y. Ge and L. M. de Moura. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In *CAV*, pages 306–320, 2009.
13. T. Grue, S. Vorobev, J. Lehtosalo, and G. van Rossum. PyAnnotate. `https://github.com/google/pytype`.
14. B. Hackett and S. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *PLDI*, pages 239–250, 2012.
15. S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *SAS*, pages 238–255, 2009.
16. M. Kramm, R. Chen, T. Sudol, M. Demello, A. Caceres, D. Baum, A. Peters, P. Ludemann, P. Swartz, N. Batchelder, A. Kaptur, and L. Lindzey. Pytype. `https://github.com/google/pytype`.
17. E. Maia, N. Moreira, and R. Reis. A Static Type Inference for Python. In *DYLA*, 2012.
18. N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *AAAI*, pages 2717–2723, 2014.
19. Z. Pavlinovic, T. King, and T. Wies. Finding Minimum Type Error Sources. In *OOPSLA*, pages 525–542, 2014.
20. A. Rastogi, A. Chaudhuri, and B. Hosmer. The Ins and Outs of Gradual Type Inference. In *POPL*, pages 481–494, 2012.
21. B. Rhodes. Adventure. `https://github.com/brandon-rhodes/python-adventure`.
22. M. Salib. Starkiller : a static type inferencer and compiler for Python. Master's thesis, Massachusetts Institute of Technology, 2004.
23. J. G. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP*, pages 2–27, 2007.
24. N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F. In *POPL*, pages 256–270, 2016.

25. G. van Rossum, J. Lehtosalo, and Ł. Langa. Type Hints. https://www.python.org/dev/peps/pep-0484/, 2014.
26. N. Vazou, E. L. Seidel, and R. Jhala. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51, 2014.