Towards a Static Type Checker for Python

Francisco Ortin
University of Oviedo
ortin@uniovi.es

J. Baltasar G. Perez-Schofield
University of Vigo
jbgarcia@uvigo.es

Jose M. Redondo
University of Oviedo
redondojose@uniovi.es

Abstract

We present the preliminary stage of *stypy*, a static type checker for Python. *stypy* translates each Python program into Python code that type-checks the original program. The generated code replaces each variable with a type variable, evaluating expression types instead of their values. The generated type checker detects type errors in different tricky Python idioms, and ensures termination.

1. Introduction

There have been different efforts to bring static type-checking to object-oriented dynamic languages such as Python, Ruby or JavaScript [5]. These languages provide no type annotations, and type checking is postponed until runtime. However, a compiler may infer type information, and use it to detect some type errors statically and to improve the runtime performance of the target code [4].

Constraint-based type inference is an approach to statically infer types of dynamically typed code. DRuby [3] and *StaDyn* [7] are two example languages that represent types as constraints, checked by a constraint resolution algorithm. Rubydust also follows this approach, and introduces the idea of generating constraints with dynamic program executions [5]. Runtime variables are wrapped to associate them to type variables, and the wrapper generates constraints when the wrapped value is used. At the end of the execution, constraints are solved to type-check the program. This constraint-based dynamic type inference approach is also followed by Flow [1], which additionally provides gradual typing [11] for JavaScript.

Refinement types have also been used to type-check dynamic languages. For example, Dependent JavaScript (DJS) is a statically typed dialect of JavaScript that provides a dependent type system, expressive enough to reason about a variety of tricky JavaScript idioms [2]. DJS requires the programmer to annotate types, but some type annotations might be inferred by analyzing the source program.

Our proposal is based on type-checking Python programs with Python itself. Static type checking is obtained with the dynamic execution of a convergent Python type checker generated for the source program. Instead of generating constraints or predicates in a refinement logic, types are represented in a subset of Python that avoids non-termination. The generated type checkers take advantage of the rich meta-programming features of the Python programming language: introspection is used to inspect the structures of objects, classes and modules, and the inheritance graph; the AST of the original program is easily obtained, transformed, compiled into the target type checker, and executed; recursion can be detected dynamically with the use of decorators; and the types of variables, functions, classes and modules can evolve throughout the execution of the type checker.

```
class Counter:
  count = 0
  def inc(self, value):
    self.count += value
    return self.count
obj = Counter()
sum = obj.inc(1) + obj.inc(0.2)
```

Figure 1. An example Python program.

```
class Counter:
  count = int
  def inc(self, *args)
    if len(args) != 1:
      return TypeError("Wrong number of args")
    self.count = op('+', get_member(self, "count"), args[0])
    return get_member(self, "count")
ts.set("Counter", Counter)
ts.set("obj", ts.get("Counter")())
ts.set("sum", op('+', get_member(ts.get("obj"),"inc")(int),
                      get_member(ts.get("obj"),"inc")(float))
```

Figure 2. A simplification of the type checker generated for the program in Figure 1.

2. Type checker generation

Figures 1 and 2 show a simple example program and (a simplification of) the corresponding type checker generated, respectively. Each variable in the original program in Figure 1 (`Counter`, `count`, `inc`, `obj` and `sum`) represents a type variable in the generated type checker (Figure 2). Python facilitates this association, since most of the language entities (modules, classes, functions, types. . .) are first-class objects.

The code in Figure 2 evaluates types of expressions instead of their values. Python code is used to infer types instead of representing them as constraints or predicates [6]. When the inferred type of an expression is `TypeError`, it means that it is not well typed. After executing the type checker, the collection of instantiated `TypeErrors` is consulted to see if the program is well typed.

Types are saved in a type store (`ts`), instead of using the Python `globals` and `locals` functions. Thereby, when the original program uses an undefined variable, the `get` method of `ts` returns a `TypeError`. Similarly, the `get_member` function makes use of Python meta-programing to check if an object, class or module provide a member, inspecting the inheritance tree. The generated type checker must never produce a runtime error, since it is aimed at type-checking the source program.

The `inc` method in Figure 2 checks the original `inc` method in Figure 1. If the number of parameters is not the expected one, a `TypeError` is returned. Otherwise, the type of an addition[1] is com-

[1] The op function is part of the type system that the generated type checkers use (Section 5). That function returns the type of any binary Python operator, passing the types of the operands as parameters.

puted and assigned to the `count` field of the particular `Counter` object pointed by `self`. In our example, the second invocation to `inc` changes the type of `obj.count` from `int` to `float` (in other languages, this side effect is represented with a special kind of constraint [7]).

Our type system considers a different type for each single object (e.g., `obj`). The type of an object is represented with another object holding the types of its members. This consideration is especially useful for type-checking structural intercession operations, since the structure of Python objects, classes and modules can be modified at runtime [10].

3. Flow Sensitiveness

In dynamic languages, programmers often give variables flow-sensitive types. In the left-hand side of Figure 3, the dynamic type of `obj.count` depends on the dynamic value of `condition`. We represent this flow-sensitiveness with union types [9].

The generated code (right-hand side of Figure 3) runs the conditional execution paths sequentially. Before each branch, the type store is cloned. The `else` body is executed with the initial type store, before the `if` body. After the `if-else` statement, the modified types in each execution path are "joined" with union types, following the algorithm described in [8]. After type-checking the code in Figure 3, the type of `obj.count` is `int` ∨ `float`.

```
obj = Counter()           ts.set("obj", ts.get("Counter")())
if condition:             _ts_1 = ts.clone()
  obj.inc(1)              get_member(ts.get("obj"), "inc")(int)
else:                     _ts_2 = ts.clone(); ts.set(_ts_1)
  obj.inc(0.5)            get_member(ts.get("obj"), "inc")(float)
                          ts = join(_ts_1, _ts_2, ts)
```

Figure 3. Original Python program (left), and the generated type checker (right) that infers a flow-sensitive type for `obj.count`.

4. Termination

Since type checking is performed via dynamic program execution, termination of the generated type checkers must be ensured. The following criteria were applied to avoid divergent type checkers:

- Python loops are analyzed the same way as conditionals, executing the code sequentially and representing flow-sensitiveness with union types.
- All the generated functions are annotated with the `norecursion` decorator we developed. This decorator dynamically detects and avoids recursive calls when types are being inferred.
- Native functions (e.g., `str`, `len` and `range`) are replaced with convergent Python functions that infer the return type depending on the parameter types passed.
- All the algorithms used in the type system (provided as an API to the generated type checkers) are designed to be convergent.

5. Project status

As shown in Figure 4, *stypy* consists of a compiler and an API. The compiler takes the original program and generates a specific type checker for that program. Then, the compiler executes the type checker, which uses the type system implemented as a Python API. The list of type errors, if any, is collected by the compiler and shown to the programmer.

We implemented the type system (API) and tested it with most Python features. We support tricky idioms such as conditional code depending on the type of arguments [2], special method names (e.g., `__add__`, `__repr__` and `__float__`), adaptation of class and

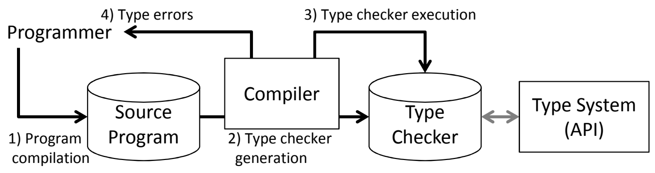

Figure 4. Architecture of *stypy*.

object structures, and flow-sensitive definition of functions and classes. The types of native built-in functions are included in the type system (API), expressing their types with Python code.

We do not support dynamic code evaluation with the `exec` and `eval` functions. For structural intercession, we type-check programs that add/remove class and object members using identifiers, but not with dynamically evaluated strings (i.e., we do not support direct access to `__dict__`). Currently, our type system detects different type errors that existing tools (such as Pylint, PyChecker, PyCharm, Pyntch, PyStarch and pyflakes) do not detect.

We are currently developing the compiler. We plan to validate its implementation with mutations of real programs, comparing its accuracy with the existing alternatives.

Acknowledgments

This work is funded by the European Union, through the European Regional Development Funds (ERDF); and by the Principality of Asturias, through its Science, Technology and Innovation Plan (grant GRUPIN14-100).

References

[1] A. Chaudhuri. Flow, a static type checker for JavaScript. http://flowtype.org, 2015.

[2] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA'12, pages 587–606, New York, NY, USA, 2012.

[3] M. Furr, A. D. Jong-Hoon, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *ACM symposium on Applied Computing (SAC)*, pages 1859–1866, Honolulu, Hawaii, March 2009. ACM.

[4] M. Garcia, F. Ortin, and J. Quiroga. Design and implementation of an efficient hybrid dynamic and static typing language. *Software: Practice and Experience*, to be published, 2015.

[5] D. A. Jong-Hoon, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic Inference of Static Types for Ruby. In *Symposium on Principles of Programming Languages (POPL)*, pages 459–472, 2011.

[6] G. Kuan, D. MacQueen, and R. B. Findler. A rewriting semantics for type inference. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 426–440. Springer-Verlag, 2007.

[7] F. Ortin. Type inference to optimize a hybrid statically and dynamically typed language. *Computer Journal*, 54(11):1901–1924, 2011.

[8] F. Ortin and M. Garcia. Supporting dynamic and static typing by means of union and intersection types. In *Progress in Informatics and Computing (PIC)*, pages 993–999, Shanghai (China), Dec. 2010.

[9] F. Ortin and M. Garcia. Union and intersection types to support both dynamic and static typing. *Information Processing Letters*, 111(6): 278–286, 2011.

[10] F. Ortin, M. A. Labrador, and J. M. Redondo. A hybrid class- and prototype-based object model to support language-neutral structural intercession. *Information and Software Technology*, 44(1):199–219, Feb. 2014.

[11] J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, Berlin, Germany, 2007. Springer-Verlag.