# Structural Type Inference: If It Quacks Like a Duck...

Jay R Bolton

June 5, 2012

## Introduction

Python is a wildly dynamic language whose program behavior depends almost entirely on the execution path. Despite being very multi-paradigm, it is also not a very complicated language, with an abstract grammar of only around 100 lines. The enforcement of type correctness in Python is performed at run-time using Duck Typing, where we are only concerned with the comparative structure of two objects. We'll explore this further in sections below.

All languages can be statically typed, no matter how dynamic their current implementation may be. Our goal here is to perform static type checking (albeit in a limited form) to a very dynamic language with very minimal inherent type enforcement. Our type checking system will be optional and will not affect the ability to run the program. The advantages of having such a system are:

- Retain 'agile development' without limiting possible correct programs.
- Still be able to analyze the type correctness of your programs, catching errors that you may not find in tests.
- Provide documentation and meaningful type annotations that can be helpful in understanding your program in a number of different ways, discussed in the last section.
- Potentially increase the performance of the language using the static type information.

This project is not concerned specifically with any one of those items. It serves more as a demonstration of pure attribute type inference.

### Terms and Definitions

When I use the term *static*, I am referring to something that **does not depend** upon the execution of the program. When I say *dynamic*, I mean something that depends upon or arises from the execution of the program.

A *type* is a labeled set of data bound by certain properties. In this project, those properties will be object attributes, which we'll discuss in more detail below.

## The Type System of Python

Python is a 100% dynamic language and as such uses no static type information. The interpreter parses and then executes your code, and any errors arise as they are encountered.

Everything in Python is an object, which provides a nice uniformity for our type system.

### Duck Typing

Duck Typing, or structural typing, is simply types represented as sets of attributes.

An attribute is an object-oriented term that refers to the data fields in a class or object. Thus a type is a set of all the data fields for some object. For example:

```
class A(object):
  x = 1
y = 'a'
```

The class A is given two fields, x and y, so it has the type: $\{x : int, y : str\}$, which is a two-element set of keys and values, where the keys are *attribute labels* and the values are *other types.*

Duck typing is usually defined as a "dynamic typing" system, but this does not necessarily have to be true. The language Scala allows programmers to write explicit, static duck types as parameters to functions:

```
def f(obj: { def x; def y }) = {
  obj.x
obj.y
}
```

This is a Scala function that takes an object with two fields: x and y. We are not concerned with any other properties of 'obj' except that it has those two fields. We do not even care what its superclass might be.

### Type Enforcement

In Python, type errors occur almost entirely when we reference attributes for some object. For example:

```
a = A()
a.x # int
a.q # undefined
```

We reference two attributes of the object 'a.' First, we reference x, which is defined and has type 'int.' We then reference the undefined attribute 'q.' This will return a type error when the Python interpreter tries to execute it.

Undefined references comprise the vast majority of type errors in python. Our goal will then be to infer these sets of attributes for all the objects in a program and do lookups on references to make sure they exist.

## Type Inference Using Only Attributes

Structural type inference is somewhat of a conceptual branching off from traditional Hindley-Milner type inference found in Haskell and ML. In describing the rules for structural type inference, I will try to use similar notation as found in the papers on Hindley-Milner, but will have to alter much of it to fit this different paradigm.

### The Grammar

Our type grammar is fairly simple; we use sets of pairs to represent types, and map those to labels.

The 'type environment,' or the set of all types inferred so far, is simply a type itself. It can also be said to be the type of the module, or the file, of the program.

$$\begin{aligned}
&Type = \alpha\{Name : Type, \ldots, Name : Type\}\\
&Name = [a - z]\\
&Environment = Type\\
&List = [Type, Type, \ldots, Type]\\
&Tuple = [Type, Type, \ldots, Type]\\
&Error = \text{“}message\text{”}
\end{aligned}$$

The $\alpha$ represents a type label and may be any lower-case Greek letter.

We must distinguish between *type labels* and *attribute names.* An attribute name is paired with both a type label and a type. We may have two attributes with two different names but with the same type. We may also have two attributes with the same name but with different types. It is therefore necessary to be able to distinguish types in and of themselves.

Certain built-in Python types—Lists, Tuples, and Dictionaries—are special cases. They are container objects whose items we would like to be able to see all at once in the type signature, so we store a list of the types of its contents.

Recall that in Python, everything is an object, including functions and classes. A function type is not a special case in this system; it is simply an object type with attributes that make it callable. A class object is also callable, and the return value is the type of its instances.

## Notation

We'll use the capital gamma, or Γ, to denote our 'environment,' which includes all the built-ins, such as types for addition and subtraction.

We'll use capital Greek letters to indicate whole types, lower case Greek letters for type labels, and Latin letters to indicate attribute names.

We use a crossbeam symbol, $\Gamma \vdash \ e \ : \ T$, to indicate that we are able to derive the type $T$ for attribute $e$ from the environment Γ.

A long right arrow indicates the application of a rule. Above the arrow, we have the assumptions and conditions; below it is the conclusion that we can draw using those assumptions.

For example:

$$\begin{array}{r} \Gamma \ \vdash \ a : T, T \vdash \ b : E \\ \Longrightarrow \\ c : E \vdash \Gamma \end{array}$$

States that if we assume we can derive the type of attribute $a$ from the environment Γ to be of type $T$, and the type of attribute $b$ from $T$ to be $E$, then we can say that attribute $c$ has type $E$, and that we can insert that mapping back into Γ.

The $\vdash$ operator is overloader for our purposes. If the left hand side is a type, then we are doing a lookup on the attributes in that type. If the left hand side is a pair of attribute mapped to type, then we are inserting that pair into the type on the right hand side.

Assumptions above the arrow are separated by commas or line breaks and often start with the actual syntax we are defining a rule for.

## Lookup and Assignment

We'll start with the most basic case: the 'identity', or simple look-up of a single object, which is a sort of axiom or first principle rather than a rule.

$$\begin{array}{l} \textit{Lookup} \\ \Gamma \vdash \ e \ : \ T \end{array}$$

The assignment rule is the next simplest. The type of the left hand side of an assignment is derived from the type of the right hand side.

$$\begin{array}{l} \textit{Assignment} \\ \texttt{x = e}, \ \Gamma \ \vdash \ e : T \\ \Longrightarrow \\ x \ : \ T \vdash \Gamma \end{array}$$

Attribute reference is a look-up of the type of the object. We first get the type of the object 'x' from the environment, and then get the type of the attribute from the type of x.

$$\begin{array}{l} \textit{Attribute} \\ \texttt{x.y},\ \Gamma\ \vdash\ x : T,\ T \vdash y : \Upsilon \\ \Longrightarrow \\ \texttt{x.y}\ :\ \Upsilon \end{array}$$

The next step is function definition, which is also a large step forward in complexity.

In order to define this rule, we need to have a new notation for *open types.* Open types are types that will absorb new attributes when they are referenced on them. In other words, when we reference an undefined attribute on an object, that object then acquires that attribute name mapped to an empty type inside the object's own type.

The notation we can use for an open type is $(T)$.

Here is the open type version of the attribute referencing rule:

$$\begin{array}{l} \textit{OpenAttribute} \\ \texttt{x.y},\ \Gamma\ \vdash\ x : (T),\ (T) \vdash y : \textit{None} \\ \Longrightarrow \\ \texttt{x.y}\ :\ \alpha\{\},\ y\ :\ \alpha\{\} \vdash (T) \end{array}$$

Now we can define our large function definition inference rule:

Capital delta represents our "scoped environment." It is a copy of the environment with the parameter types overwritten, or "shadowed."

$$\begin{array}{l} \textit{FunctionDefinition} \\ \texttt{def f(p):b},\ \Gamma = \Delta, \\ p = (p_1, \ldots, p_n, p) : \textit{Tuple}([(\alpha_1\{\}), \ldots, (\alpha_n\{\})]) \vdash \Delta \\ \Delta \vdash b : T,\ T \vdash \texttt{return} : E \\ T \vdash p : X \\ \Longrightarrow \\ f : \{*r : E, *p : X\} \vdash \Gamma \end{array}$$

First we take a duplicate environment $\Delta$, and insert a tuple type of empty open types into it, representing our shadowed parameters.

We want our parameter types to be open so that any attribute references in the body of the function modify their type.

'*r' and '*p' represent special meta-attributes for the return type and the parameter type which we can use for callable objects.

Next up is the actual function call, which is when we apply our function definition types. Luckily, it is slightly simpler.

$$\begin{array}{l} \textit{FunctionCall} \\ \texttt{f(a)},\ \Gamma \vdash f : T \\ \Gamma \vdash a : X,\ T \vdash *p : X' \\ X \approx X' = S \\ S \rightarrow T \vdash \texttt{return} : E \\ \Longrightarrow \\ \texttt{f(a)} : E \end{array}$$

This rule introduces a couple new symbols. $\approx$ is used to denote *unification*, which is explained in the next section. In the third line we assume that $X$ unifies with $X'$. The right arrow denotes that the unification produces a substitution $S$. In the forth line, we apply the substitution $S$ to the function type $T$, and then derive from that our return type.

Our final type inference rule is that of class definition.

$$\begin{array}{l} ClassDefinition \\ \texttt{class C: b},\ \Gamma = \Delta, \\ \Delta \vdash b : T,\ T \vdash \texttt{init} : E \\ T \vdash \texttt{self} : X \\ \Longrightarrow \\ C : T + \{*r : X + E\} \end{array}$$

$X + E$ represents the merging of types X and E, where duplicate types in E overwrite those in X.

In this rule, we derive the type of our body to be $T$, and then look up the type of the constructor or `init` function from within the type of the body. For the the attributes of the instance objects, we look up the type of 'self.' We combine the type from the constructor with the one from self to put into the return field for the class. We give the class itself all the remaining fields.

This last rule is quite rough and does not contain all the details in the implementation of class definition type inference. For example, in order for the open type 'self' to propogate up into the type of the class body, we would have to add a special case in function definition. Also in python, assignments in the body provide fields for both the class and the instances. Despite its lack of accuracy, hopefully the rule will convery the general idea.

### Unification and Substitution

Unification is used in the function call rule to check that the given function type and the applied type are compatible.

Unification produces a substitution, which is a mapping of labels to types, indicating that the types for each of the lables should be replaced with those listed.

$$\mathit{Substitution} = \{a_1 \mapsto s_1, \ldots, a_n \mapsto s_n\}$$

Unification itself is performed according to the following rules:

1. An attribute 'a' with an empty type $A$ unifies with any other type $T$ , producing $\{a \mapsto T\}$
2. When unifying non-empty types $a : \Upsilon$ and $T$, all attributes in $T$ must be present and unifiable in $\Upsilon$. This also produces the substitution $\{a \mapsto T\}$
3. List and tuple types must unify their contained types element-wise. For example: the list $l : \Lambda$ unified with $T$ produces $\{l \mapsto T\}$ where all types inside $\Lambda$ and $T$ are unifiable element-wise.

A unification of types $X$ and $Y$ produce a substitution $S$, denoted $X \approx Y = S$. We can apply that substitution to a type $Y$ with $S \rightarrow Y = Y'$, where $Y'$ is the type $Y$ but with all substitutions in $S$ applied.

## Examples and Cases

Our example will use the shortest code possible that demonstrates all the type inference rules. We'll walk through the heuristics required to then infer all the types in the environment (also known as the type of the module).

```
class C:
  x = 1
  __init__(self):
    self.y = "s"
    self.z = 2
  smeth(x,y): return x
  imeth(self): return self.x
c = C()
d = C.smeth(9,"q")
e = c.imeth()
f = C.none()
```

We'll briefly walk through the application of the type inference rules. In order to save space, we won't step through the actual rules, we'll just show their conclusions. The algorithm will go top-down, and upon reaching names and primitives, will then traverse bottom-up.

$$Assignment, FunctionDefinition, ClassDefinition \\ \Longrightarrow \\ \Gamma \vdash C : \alpha\{x : 1,\ *r : \{y : str, z : int, x : int\} * p : Tuple[]\}$$

Analysis of the body of the class produces a type for $C$ that is callable. This callable type has an empty parameter tuple and a return type that represents the type of the instances for the class.

$$FunctionCall, Assignment \Longrightarrow \Gamma \vdash c : \alpha\{y : str, z : int, x : int\}$$

Our constructor produces the instance type of class C.

$$FunctionCall, Assignment \Longrightarrow \Gamma \vdash d : int$$

Calling the static method `smeth` returns the type of the first parameter, an integer.

$$FunctionCall, Assignment \Longrightarrow \Gamma \vdash e : int$$

Calling the instance method `imeth` returns the type of the attribute 'x' inside 'c', with is an integer. Note that in the function call, 'c' is bound to 'self' in the first parameter.

$$FunctionCall, Assignment \Longrightarrow f : Error\ "undefined"$$

When we try to reference an undefined attribute "none" on 'c', then we return an Error type with a message.

## Improvement and Expansion

The two major features not included in this system are rules for module importing and rules for super classes.

## The Implementation

This system has been implemented in Jython (a Python interpreter that produces byte code for the Java Virtual machine), located at `https://github.com/aesacus/python-type-inferencer`. The design of that program is described in a presentation at:

`https://docs.google.com/presentation/d/19vKKkBxZPOV1TP4DecFaXtIoMewjXlAVXo2GhZ0t0j0`.

The above program works well on a pre-written demo of sample Python code. It could use expansion and further testing, as well as more readable and usable data output. It comes with a demo script (`demo.py`) that showcases its features.

## References and Further Reading

I derived most of the rules and functions myself. The basic structure of type inference, where you pass around an environment, apply unification and get substitutions, was learned from Cardelli.

- Luca Cardelli. “Basic Polymorphic Typechecking”
- Bryan Hackett and Shu-yu Guo. “Fast and Precise Hybrid Type Inference for Javascript”