Measuring Polymorphism in Python Programs

Beatrice Åkerblom
Stockholm University, Sweden
beatrice@dsv.su.se

Tobias Wrigstad
Uppsala University, Sweden
tobias.wrigstad@it.uu.se

Abstract

Following the increased popularity of dynamic languages and their increased use in critical software, there have been many proposals to retrofit static type system to these languages to improve possibilities to catch bugs and improve performance.

A key question for any type system is whether the types should be structural, for more expressiveness, or nominal, to carry more meaning for the programmer. For retrofitted type systems, it seems the current trend is using structural types. This paper attempts to answer the question to what extent this extra expressiveness is needed, and how the possible polymorphism in dynamic code is used in practise.

We study polymorphism in 36 real-world open source Python programs and approximate to what extent nominal and structural types could be used to type these programs. The study is based on collecting traces from multiple runs of the programs and analysing the polymorphic degrees of targets at more than 7 million call-sites.

Our results show that while polymorphism is used in all programs, the programs are to a great extent monomorphic. The polymorphism found is evenly distributed across libraries and program-specific code and occur both during program start-up and normal execution. Most programs contain a few "megamorphic" call-sites where receiver types vary widely. The non-monomorphic parts of the programs can to some extent be typed with nominal or structural types, but none of the approaches can type entire programs.

Categories and Subject Descriptors D.3 Programming Languages [*D.3.3 Language Constructs and Features*]: Polymorphism

Keywords Python, dynamic languages, polymorphism, trace-based analysis

1. Introduction

The increasing use of dynamic languages in critical application domains [21, 25, 30] has prompted academic research on "retrofitting" dynamic languages with static typing. Examples include using type inference or programmer declarations for Self [1], Scheme [36], Python [4, 5, 29], Ruby [3, 15], JavaScript [17, 35], and PHP [14].

Most mainstream programming languages use static typing from day zero, and thus naturally imposed constraints on the run-time flexibility of programs. For example, strong static typing usually guarantees that a well-typed $x.m()$ at compile-time will not fail at run-time due to a "message not understood". This constraint restricts developers to updates that grow types monotonically.

Retrofitting a static type system on a dynamic language where the definitions of classes, and even individual objects, may be arbitrarily redefined during runtime poses a significant challenge. In previous work for Ruby and Python, for example, restrictions have been imposed om the languages to simplify the design of type systems. The simplifications concern language features like dynamic code evaluation [15, 29], the possibility to make dynamic changes to definitions of classes and methods [4], and possibility to remove methods [15]. Recent research [2, 24, 27] has shown that the use of such dynamic language features is rare—but non-negligible.

Apart from the inherent plasticity of dynamic languages described above, a type system designer must also consider the fact that dynamic typing gives a language unconstrained polymorphism. In Python, and other dynamic languages, there is no static type information that can be used to control polymorphism *e.g.,* for method calls or return values.

Previous retrofitted type systems use different approaches to handle ad-hoc polymorphic variables. Some state prerequisites disallowing polymorphic variables [5, 35], assuming that polymorphic variables are rare [29]. Others use a flow-sensitive analysis to track how variables change types [11, 15, 18]. Disallowing polymorphic variables is too restrictive as it rules out polymorphic method calls [19, 24, 27].

There are not many published results on the degree of polymorphism or dynamism in dynamic languages [2, 8, 19, 24, 27]. This makes it difficult to determine whether or not relying on the absence of, or restricting, some dynamic behaviour is possible in practise, and whether certain techniques

for handling difficulties arising due to dynamicity is preferable over others.

This article presents the results of a study of the runtime behaviour of 36 open source Python programs. We inspect traces of runs of these programs to determine the extent to which method calls are polymorphic in nature, and the nature of that polymorphism, ultimately to find out if programs' polymorphic behaviour can be fitted into a static type.

1.1 Contributions

This paper presents the results of a trace-based study of a corpus of 36 open-source Python programs, totalling oven 1 million LOC. Extracting and analysing over 7 million call-sites in over 800 million events from trace-logs, we report several findings – in particular:

- A study of the run-time types of receiver variables that shows the extent to which the inherently polymorphic nature of dynamic typing is used in practise.

  We find that variables are predominantly monomorphic, *i.e.,* only holds values of a single type during a program. However, most programs have a few places which are megamorphic, *i.e.,* variables containing values of many different types at different times or in different contexts. Hence, a retrofitted type system should consider both these circumstances.

- An approximation of the extent to which a program can be typed using nominal or structural types using three typeability metrics for nominal types, nominal types with parametric polymorphism, and structural types. We consider both individual call-sites and clusters of call-sites inside a single source file.

  We find that, because of monomorphism, most programs can be typed to a large extent using simple type systems. Most polymorphic and megamorphic parts of programs are not typeable by nominal or structural systems, for example due to use of value-based overloading. Structural typing is only slightly better than nominal typing at handling non-monomorphic program parts.

Our trace data and a version of this article with larger figures is available from `dsv.su.se/~beatrice/python`.

Outline The paper is organised as follows. § 2 gives a background on polymorphism and types. § 3 describe the motivations and goals of the work. § 4 accounts for how the work was conducted. § 5 presents the results. § 7 discusses related research and finally in § 8 we present our conclusions and present ideas for future work.

2. Background

We start with a background and overview of polymorphism and types (§ 2.1) followed by a quick overview of the Python programming language (§ 2.2). A reader with a good understanding of these areas may skip over either or both part(s).

2.1 Polymorphism and Types

Most definitions of object-oriented programming lists polymorphism—the ability of an object of type T to appear as of another type T'—as one of its cornerstones.

In dynamically typed languages, like Python, polymorphism is not constrained by static checking and error-checking is deferred to the latest possible time for maximal flexibility. This means that T and T' from above need not be explicitly related (through inheritance or other language mechanisms). It also means that fields can hold values of any type and still function normally (without errors) as long as all uses conform to the run-time type of the current object they store. This kind of typing/polymorphic behaviour is commonly referred to as "duck typing" [23].

Subtype polymorphism in statically typed languages is bounded by the requirements needed for static checking (*e.g.,* that all well-typed method calls can be bound to suitable methods at run-time). This leads to restrictions for how T and T' may be related. In a nominal system this may mean that the classes used to define T and T' must have an inheritance relation. A nominal type is a type that is based on names, that is that type equality for two objects requires that the name of the types of the objects is the same. In a structural type system, type equivalence and subtyping is decided by the definition of values' structures. For example, in OCaml and Strongtalk, type equivalence is determined by comparing the fields and methods of two objects and also comparing their signatures (method arguments and return values).

Strachey [33] separates the polymorphism of functions into two different categories: *ad hoc* and *parametric*. The main difference between the categories is that *ad-hoc* polymorphism lacks the structure brought by parameterisation and that there is no unified method that makes it possible to predict the return type from an *ad-hoc polymorphic* function based on the arguments passed in as would be the case for the *parametric polymorphic* function [33]. As an example of ad hoc polymorphism, consider overloading of `/` for combinations of integers and reals always yielding a real.

Cardelli and Wegner [10] further divide polymorphism into two categories at the top level: *universal* and *ad-hoc*. Universal polymorphism corresponds to Strachey's parametric polymorphism together with *call inclusion polymorphism*, which includes object-oriented polymorphism (subtypes and inheritance). The common factor for universal polymorphism is that it is based on a common structure (type) [10]. Ad-hoc polymorphism, on the other hand, is divided into overloading and coercion, where overloading allows using the same name for several functions and coercion allowing polymorphism in situations when a type can automatically be translated to another type [10].

Using the terms from above, "duck typing" can be described as a lazy structural typing [23] (late type checking) and is a subcategory of ad-hoc polymorphism [10].

2.2 Python

Python is a class based language, but Python's classes are far less static than classes normally found in statically typed systems. Class definitions are executed during runtime much like any other code, which means that a class is not available until its definition has been executed. Class definition may appear anywhere, *e.g.,* in a subroutine or within one branch of a conditional statement. If two class definitions with the same name are executed within the same name-space, the last definition will replace the first (although already created objects will keep the old class definition). If a class is reloaded, it might have been reloaded with a different set of methods than the original one. Given this possibility to reload classes, the same code creating objects from the class `C` may end up creating objects of different classes at different times during execution, objects that may have a different set of methods.

Python allows multiple inheritance, *i.e.,* a class may have many superclasses [28]. Subclasses may override methods in its superclass(es) and may call methods in its superclass(es). Python's built-in classes can be used as superclasses.

Python classes are represented as objects at runtime. Class objects can contain attributes and methods. All members in a Python class (attributes and methods) are public. Methods always take the receiver of the method call as the first argument. It must be explicitly included in the method's parameter list but is passed in implicitly in the method call.

There are two different types of classes available in Python up to version 3.0: old-style/classic classes and new-style classes. The latter were introduced in Python 2.2 (released in 2001) to unify class and type hierarchies of the language and they also (among other things) brought a new method resolution order for multiple inheritance. From Python 3.0, all classes are new-style classes.

Python objects are essentially hash tables in that attributes and methods and their names may be regarded as key-value pairs. Both attributes and methods may be added, replaced and entirely removed also after initialisation. For an object `foo`, we can add an attribute `bar` by simply assigning to that name, i.e. `foo.bar = 'Baz'`. The same attribute may then be removed, *e.g.,* by the statement `del foo.bar`, which removes both key and value.

Classes in Python are thus less templates for object creation than what we may be used to from statically typed languages, but more like factories creating objects–objects that may later change independent of their class and the other objects created from the same class. This more dynamic approach to classes has implications on and may increase program polymorphism.

In nominally typed language, a type `Sub` is a subtype of another type `Sup` only if the class `Sup` is explicitly declared to be its supertype. In some languages, Python for example, this declaration may be updated and changed during runtime.

2.3 Measuring Polymorphism

When the code below is run, a class `Foo` is first defined containing two methods; `__init__` and `bar`, both expecting one argument. The `__init__` method creates the instance variable `a` and assigns the expected argument to it. In the `bar` method a call is made to the method `foo` on the instance variable `a` and then a call is made to the method `baz` on the argument variable `b`.

```
01                              10 f = Foo(...)
02 class Foo:                   11
03    def __init__(self, a):   12 for e in range(0,100):
04       self.a = a            13    class Bar:
05                              14       def baz(self):
06    def bar(self, b):         15          pass
07       self.a.foo()           16
08       b.baz()                17    f.bar(Bar())
09                              18
```

After the class definition is finished, a variable f is created and it is assigned with a new object of the class `Foo`.

On line 12–17, follows a `for` loop that will iterate 100 times and for every iteration the class `Bar` is created with a method `baz` that has no body. On the last line in the `for` loop a call is made to the method `bar` for the `Foo` object in `f` (from line 10) passing a new object of the current `Bar` class as an argument.

Several lines in the code above (7, 8 and 17), contain method calls. These lines are *call-sites*.

DEFINITION 1 (Call-site). *A call-site in a program is a point (on a line in a Python source file) where a method call is made.*

Every call-site has two points, the receiver and the argument(s), where types may vary depending on the path taken through the program up to the call-site. In the analyses made for this paper, the focus has been on the receiver types. Arguments will generally become receivers at a later point in the program execution, which means that also that polymorphism will get captured by the logging.

On line 17, a call is made to the method `bar`, where the receiver will always be an object of the class `Foo`, since the assignment to `f` is made from a call to the constructor of `Foo` on line 10. This means that the call-site `f.bar(...)` on line 10 is *monomorphic* and will always resolve to the same method at run-time.

DEFINITION 2 (Monomorphic). *A call-site that has the same receiver type in all observations is monomorphic.*

The call-site on line 7 may be monomorphic, but that cannot be concluded from the static information in the available code. The type of the receiver on line 7 depends on the type of the argument to the constructor when the object was created. If objects are created storing objects of different types in the instance variable `a`, the line 7 will potentially be executed with more than one receiver type, that is, it is *polymorphic*. If the number of receiver types is very high, the call-site is

instead *megamorphic*. Following Agesen [1] we count a call-site as megamorphic if it has been observed with six or more receiver types.

DEFINITION 3 (Polymorphic). *A call-site that has 2–5 different receiver types in all observations is polymorphic.*

DEFINITION 4 (Megamorphic). *A call-site that has six or more receiver types in all observations is megamorphic.*

Line 8 in the code above shows an example of a megamorphic call-site with a call to the method `baz` for the object in the variable `b`. The value of `b` depends on what is passed as the argument with the method call to `bar`, made on line 17. The loop on line 12–17 runs the class definition of `Bar` in every iteration, which means that every call to the method `baz` will be made to an object of a new class. Nevertheless, since the class always has the same name and contains the same fields and methods, the classes created here should be regarded as the same class. This megamorphism is false and will not be considered as such by our analysis.

3. Motivation and Research Questions

A plethora of proposals for static type systems for dynamic languages exist [1, 3–5, 15, 17, 29, 35, 36]. The inherent plasticity of the dynamic languages (for example, the possibility to add and remove fields and methods and change an object's class at run-time) is a major obstacle for designers of type systems but the use of these possibilities have been shown to be infrequent [2, 19, 24, 27]. Additionally, a type system designer must also take duck typing into consideration, where objects of statically unrelated classes may be used interchangeably in places where common subsets of their methods are used.

We examine several aspects of Python programs of interest to designers of type systems for dynamic languages in general and for Python specifically. These aspects of program dynamicity may also be used to enable comparisons of different proposed type system solutions.

We study Python's unlimited polymorphism—duck typing—in particular the degree of polymorphism in receivers of method calls in typical programs: How many different types are used and how related the receivers' types are *e.g.,* in terms of inheritance. We study how the underlying dynamic nature of Python affects the polymorphism of programs due to classes being dynamically created and possibly modified at run-time.

Analysis Questions Our questions belong to three categories: *program structure*, *extent and degree* and *typeability*:

1. Program structure
   (a) How many classes do Python programs use/create at run-time? How often are classes redefined?
   (b) How many methods do Python classes have and how many methods are overridden in subclasses?
2. Extent and degree
   (a) What is the proportion between monomorphic and polymorphic call-sites?
   (b) What is the average, median and maximum degrees of polymorphism and megamorphism (that is, number of receiver types) of non-monomorphic call-sites?
   (c) To what extent are non-monomorphic call-sites "megamorphic"?
   (d) Does the degree of polymorphism and megamorphism differ between library and program or between start-up and normal runtime?
   (e) What types are seen at extremely megamorphic call-sites (*e.g.,* with 350 different receiver types)?
3. Typeability
   (a) How do types at polymorphic and megamorphic call-sites and clusters relate to each other in terms of inheritance and overridden methods?
   (b) To what extent is it possible to find a common super type for all the observed receiver types that makes it possible to fit the polymorphism into a nominal static type?
   (c) To what extent is it possible to find a common super type for all the observed receiver types if the nominal types are extended with parametric polymorphism?
   (d) To what extent do receiver types in clusters contain all the methods that are called at the call-sites of the cluster? That is, to what extent can we find a common structural type for all the receiver types found in clusters?

Following [2, 19, 24, 27] we also examine the applicability of the phenomenon of Folklore, put forward by Richards et al [27] which states that there is an initialisation phase that is more dynamic than other phases of the runtime. We compare if there are differences in the use of polymorphism depending on where we find the method calls; during start-up vs. during normal execution and also if there are differences between libraries and program-specific code.

4. Methodology

Studying how polymorphism is used in Python programs necessitates studying real programs. We discarded static approaches such as program analysis and abstract interpretation because of their over-approximate nature. Instead, we base our study on traces of running programs obtained by an instrumented version of the standard CPython interpreter that saves data about all method calls made throughout a program run. Our instrumented interpreter is based on CPython 2.6.6 because of Debian packaging constraints, which was important to study certain proprietary code which in the end did not end up in this study.

The results are obtained from in total 522 runs of 36 open source Python programs (see Table 1) collected from Source-

Forge [32]. Selection was based on programs' popularity (>1,000 downloads), that the program was still maintained (updated during the last 12 months) and was classified as stable, *i.e.,* had been under development for some time. For pragmatic reasons, we excluded programs that used C extensions, and programs that for various reasons would not run under Debian. For equally pragmatic reasons, we excluded plugins (*e.g.,* to web browsers), programs that required specific hardware (*e.g.,* microscopes, network equipment or servers) and software that required subscriptions (*e.g.,* poker site accounts).

To separate events in the start-up phase from "normal program run-time" in our analyses, we followed the example of Holkner and Harland [19] and placed markers in the source of all programs at the point where the start-up phase finished. This would typically be at the point where the graphical user interface had finished loading and just before entering the main loop of the program.

We have chosen to include libraries in our study to make it possible to compare the library code to program specific code to see if we find any difference in polymorphic behaviour. To separate the events originating in library code from those originating in program specific code in our analyses, a fully qualified file name was saved for all events.

Command line programs were run using commands given in official tutorials and manuals to capture the execution of all standard expected use cases. Libraries were used in a similar way with examples from official tutorials. Depending on the availability of examples, command line programs and some libraries shared between multiple programs were run over 100 times.

For applications with a GUI the official tutorials and examples were followed by hand and care was taken to ensure that each menu alternative and button was used. The interactive GUI applications were run for 10–15 minutes between 2 and 12 times depending on the number of functions available.

The Python interpreter we used was instrumented to trace all method calls (including calls caused by internal Python constructs, like the use of operators, etc.) and all loaded class definitions. For all method calls made, we logged the call-site's location in the source files, the receiver type and identity, the method name, the identity of the calling context (`self` when the method call was made), the arguments' types and return types. Every time a class definition was executed, we logged the class name, names of superclasses and the names of the methods.

Program Structure To answer our questions on program structure from § 3, we collect data about classes loaded at run-time. We count recurrences of class definitions and compare their sets of methods.

Extent and Degree (of Polymorphism) To answer our questions in § 3 § 2a – § 2e, we collect receiver type information found at each call-site, and categorise the call-sites

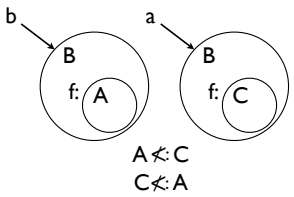

Figure 1. Parametric polymorphism. Different instances of `B` hold objects of different types in the `f` fields.

based on how many receiver types were found according to the following categories:

Single-call The call-site was only executed once. It is therefore *trivially monomorphic*, but we conservatively refrain from classifying it any further.

Monomorphic The call-site was monomorphic and executed more than once, so it is *observably monomorphic*. "Observably" refers to the nature of our trace-based method, which does not exclude the possibility that a different run of the same program might observe polymorphic behaviour for the same call-site.

Polymorphic The call-site was observed with between two and five different receiver types.

Megamorphic The call-site was observed with more than five different receiver types.

Typeability The questions in § 3 § 3a – § 3d are all concerned with to what extent the polymorphism found in real Python programs could be retrofitted with a type system.

All monomorphic call-sites are always typeable with a nominal or a structural type. Receivers at a specific call-site in isolation will always have the same structural type (see § 2.1). For a polymorphic call-site to be nominally typeable, all receivers must share a common supertype that defines the method in question.

We define a metric, *N-typeable* to approximate static typeability with a hypothetical simple nominal type system:

DEFINITION 5 (N-typeable). *A polymorphic call-site is N-typeable if there is, for all its receiver types, a common superclass that contains the method called at the call-site.*

Nominal typing could be extended with parametric polymorphism (see § 2.1 to increase the flexibility to account for different types being used in the same source locations across different run-time contexts. In that case, a call-site can be typed for unrelated receiver types given that it is *N-typeable* for each sender identity (that is the value of `self` when the call was made).

This would mean that the receiver was typeable for all calls that were executed inside some specific object, as is illustrated in Figure 1 with objects `a` and `b`, both instances of the class `B`. The field `f` in `a` holds an instance of the class `C`, while the field `f` in `b` holds an instance of the class `A`. A call-site in the code of the class `B`, that has the field `f` as a

receiver would in this case always have the same type for all calls made in the same caller context.

For all polymorphic and megamorphic call-sites we also examine if they are *NPP-typeable*:

DEFINITION 6 (NPP-typeable). *A polymorphic call-site is NPP-typeable if it is N-typeable or, if the receiver types were grouped by the identity of the sender (*`self` *when the call was made), we find a common supertype for each group that contains the method called at the call-site.*

The typeability considered so far has been based on individual call-sites (*i.e.,* individual source locations). This might lead to an over-estimate of the typeability of programs. For example, in the code example below, calls are made to the method `example(a, b)` with a first argument of either the type `T` or `T'`, where `T` has the methods `foo()` and `bar()` but not the method `baz()` and where `T'` has the method `foo()` and `baz()` but not the method `bar()`. The second argument for the method calls is always a boolean; a boolean that is always `True` when `a` is of the type `T` and `False` when `a` is of the type `T'` (so-called value-based overloading).

```
02 def example(a, b):
03   a.foo()
04   if b:
05     a.bar()
06   else:
07     a.baz()
```

Considering each call-site in isolation, the call-sites on line 5 and 7 are typeable since they will always have the same receiver type. However, giving a static type to the program without significant rewrite would assign a single type to `a` which means typing line 3, 5 and 7 with a single static type.

To assign types to co-dependent source locations, we cluster call-sites connected by the same receiver values (*i.e.,* 3 & 5 and 3 & 7) plus transitivity (*i.e.,* 5 & 7, indirectly via 3). We then attempt to type the cluster as a whole.

DEFINITION 7 (Cluster). *A cluster is a set of call-sites, from the same source file, connected by the receivers they have seen. For all pairs of call-sites* `A` *and* `B` *in a cluster, they have either seen the same receiver or there exists a third call-site* `C` *that has seen the same receiver as both* `A` *and* `B`*.*

Typing the cluster in the code example above, we search for a common supertype of `T` and `T'` that contains all of `foo()`, `bar()` and `baz()`, *i.e.,* the union of the call-sites' methods in the cluster. If such a type does not exist, the cluster can not be typed. It can be argued that rejecting the cluster in its entirety is a better approximation than claiming 66% of the method's call-sites typeable.

DEFINITION 8 (N-typeable Cluster). *A cluster is N-typeable iff* `T'`*, the most specific common supertype of the types of all receivers in all call-sites in the cluster, contains all the methods called at all call-sites in the cluster.*

For the cluster to be typeable with a structural type, all the types (`T` and `T'`) seen at all call-sites (on line 3, 5 and 7) must contain all the methods that were called at all call-sites in the cluster (`foo()`, `bar()` and `baz()`.

DEFINITION 9 (S-typeable Cluster). *A cluster is S-typeable iff the intersection of all types of all its receivers contains all the methods called at all call-sites in the cluster.*

Whereas considering individual call-sites may be overly optimistic, considering clusters of call-sites may be overly pessimistic. For the code example above, for example, we would conclude that the cluster was neither N-typeable nor S-typeable, since there exists no type `T''` that contain all the three methods called at the cluster's call-sites. A more powerful type system might be able to capture this value-based overloading, such as a system with refinement types. Whether such a system used nominal or structural types is insignificant in this case.

5. Results

This section presents the results from analysing 528 program traces of the 36 Python programs in our corpus. The results are grouped into the same categories that were presented in § 3; Program structure, Extent and degree and Typeability.

5.1 Program Structure

Classes in Python Programs The underlying dynamic nature of Python affects the polymorphism of programs in that classes are dynamically created and possibly modified at runtime. The possibility to reload a class with a different definition during runtime and the possibility that the path taken through the program affects the numbers and/or versions of classes that are loaded all contribute to the polymorphism of Python programs. This polymorphism makes it more difficult to predict statically what types will be needed to type the the program the next time it runs.

Our traces contained 31,941 unique classes. The source code of the 36 programs contained the definition of 11,091 classes (libraries uncounted). The source of the individual programs contain between 4 and 1,839 class definitions with an average of 308 classes and a median of 129 classes and (see Table 2).

With only three exceptions (Pychecker, Docutils and Eric4), the number of classes loaded by the program was larger than the number of classes defined in its source code. The number of declared classes found in the source code can be found as the first figure in the column titled "Class defs. top/nested" in Table 2. That the number of classes used in a program is larger than the number defined in the program's code is what should be expected since Python comes with a large ecosystem of libraries containing important utilities. The loading of these library modules leads to loading and creation of classes; classes that can not be found in the current program's source code. The exceptions (Pychecker, Docutils and Eric3)

Table 1. A list of the programs included in the study, sorted on size (see Table 2). The third column contains the share of the call-sites that were polymorphic + megamorphic (P+M), and the fourth one the share of these P+M that were N-typeable (P+M N-t). The fifth column contains the share of all call-sites that were N-typeable (N-t). Column 3-5 all contain figures for whole programs. Column 6-7 contain P+M and N-t for program startup, column 8-9 P+M and N-t for runtime, column 10 P+M for library code and finally column 11 P+M for program specific code. All figures denote the share of call-sites compared with the total numbers of call-sites in the program traces, except column 4 (Typeable Poly (%)). Program version numbers can be found in Table 4.

		Whole			Startup		Runtime		Lib.	Prog.
		P+M	Typeable	N-t	P+M	N-t	P+M	N-t	P+M	P+M
No.	Name	(%)	Poly (%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
1.	Pdfshuffler	2.5	32.7	0.8	0.6	0.0	4.2	0.5	2.0	4.6
2.	PyTruss	1.6	3.9	0.1	-	-	-	-	-	-
3.	Radiotray	1.6	18.8	0.3	1.5	0.0	1.7	0.4	1.8	0.0
4.	Gimagereader	3.0	4.4	0.1	0.9	0.1	7.6	0.1	3.2	2.0
5.	Ntm	1.1	3.8	0.0	1.2	0.0	1.0	0.0	1.3	0.2
6.	Torrentsearch	12.4	4.6	0.6	4.9	0.5	15.1	0.6	-	-
7.	Brainworkshop	1.0	20.9	0.2	0.6	0.1	2.7	0.4	0.6	3.6
8.	Bleachbit	4.2	6.8	0.3	3.4	0.3	7.5	0.2	2.5	9.7
9.	Diffuse	1.5	0.6	0.0	0.8	0.0	2.1	0.0	12.4	2.5
10.	Photofilmstrip	3.6	37.5	1.3	0.6	0.0	5.3	1.0	4.0	1.8
11.	Comix	3.5	4.1	0.1	0.7	0.0	4.9	0.1	4.3	1.4
12.	Pmw	3.1	49.0	1.7	-	-	-	-	-	-
13.	Requests	2.8	24.9	0.6	-	-	-	-	3.3	2.9
14.	Virtaal	2.5	18.1	0.5	1.4	0.0	3.0	0.4	2.5	2.5
15.	Pychecker	1.5	8.7	0.3	-	-	-	-	1.8	0.7
16.	Idle	5.6	56.0	3.2	1.1	0.4	7.7	4.2	3.7	8.1
17.	Fretsonfire	2.2	18.3	0.4	1.2	0.0	3.7	1.0	1.7	3.3
18.	PyPe	2.5	17.8	0.4	1.3	0.7	4.5	1.1	2.1	4.8
19.	PyX	3.5	33.9	1.2	-	-	-	-	1.3	4.5
20.	Pyparsing	5.7	72.0	4.1	-	-	-	-	1.6	11.9
21.	Rednotebook	1.4	3.7	0.1	1.2	0.0	1.8	0.0	1.5	1.3
22.	Linkchecker	6.6	2.6	0.2	1.2	0.0	13.7	0.1	5.3	8.8
23.	Solfege	2.8	41.4	1.2	1.2	0.0	3.6	1.5	1.3	3.9
24.	Chilsdplay	4.1	33.9	1.4	0.9	0.0	6.3	3.3	1.7	8.5
25.	Scikitlearn	3.1	60.9	2.1	-	-	-	-	-	-
26.	Mnemosyne	3.0	57.2	1.8	1.2	0.2	3.1	2.0	2.8	3.6
27.	Youtube-dl	1.2	11.6	0.1	-	-	-	-	-	-
28.	Docutils	6.2	31.7	2.0	-	-	-	-	2.2	8.7
29.	Pymol	8.6	0.6	0.1	-	-	-	-	10.7	4.4
30.	Timeline	2.0	21.1	0.4	0.5	0.0	2.8	0.7	-	-
31.	DispcalGUI	2.9	15.5	0.4	0.8	0.0	4.1	0.6	2.1	4.2
32.	Pysolfc	4.3	40.7	1.8	1.0	0.4	9.4	3.9	3.1	4.9
33.	Wikidpad	3.9	23.5	0.9	2.6	1.1	5.3	0.5	3.8	6.7
34.	Task Coach	6.4	37.1	2.4	-	-	-	-	3.6	8.4
35.	SciPy	6.8	42.4	2.8	-	-	-	-	3.8	7.8
36.	Eric4	2.2	37.0	0.8	1.7	0.6	3.2	0.8	1.6	2.5
	Average	3.9	25.0	0.96	1.35	0.18	5.18	0.98	3.12	4.61

Table 2. A list of the programs included in the study sorted on size (LOC from the second column) with the smallest one at the top. The third column shows the range (min-max number) of unique classes loaded when the programs were run. The fourth column contains the number of class definitions found in the source code of the programs and the number of class definitions that were found in a nested environment (e.g. inside a method) and the fifth the average number of method definitions loaded during the program runs. The sixth column contains the average number of method definitions loaded during a program run that were redefinitions of inherited methods. The seventh column contains the number of classes that were found defined with more than one interface (set of methods). The eighth and last column contains the the percent of all classes that use multiple inheritance.

	Program	LOC	#Classes (range)	Class defs. top/nested	Avg.# meth.	Avg.# overr.	Int. diff.	Mult. inh.(%)
1.	PDF-Shuffler	1.0K	181-181	4/0	1.6K	0.2K	0	11.0
2.	PyTruss	1.5K	731-745	19/0	11.4K	1.7K	0	3.1
3.	Radiotray	1.5K	353-353	25/0	3.1K	0.4K	0	5.8
4.	GImageReader	2.2K	361-361	15/0	2.8K	0.4K	0	3.5
5.	Ntm	2.8K	239-239	10/0	2.0K	0.3K	0	5.4
6.	TorrentSearch	3.0K	471-479	63/0	5.1K	0.5K	0	1.7
7.	BrainWorkshop	3.6K	673-677	43/0	6.0K	0.7K	0	2.4
8.	BleachBit	4.1K	249-250	39/2	2.0K	0.2K	0	5.9
9.	Diffuse	5.6K	154-154	47/24	1.4K	0.1K	1	3.9
10.	PhotoFilmStrip	6.1K	791-795	66/0	12.3K	1.9K	0	3.0
11.	Comix	7.7K	287-308	45/0	2.3K	0.3K	0	3.3
12.	Pmw	10.3K	97-113	41/0	1.2K	0.1K	0	8.5
13.	Requests	11.2K	366-423	109/6	2.9K	0.5K	0	10.2
14.	Virtaal	11.4K	644-654	133/18	13.7K	2.4K	0	3.2
15.	Pychecker	12.7K	82-2180	311/35	2.0K	0.7K	0	5.6
16.	Idle	13.0K	285-311	146/10	3.0K	0.4K	0	6.1
17.	FretsOnFire	14.0K	772-797	365/8	2.8K	0.8K	1	7.2
18.	PyPe	15.3K	891-891	320/30	7.2K	1.5K	1	5.8
19.	PyX	15.8K	409-453	303/15	3.5K	0.5K	0	7.9
20.	Pyparsing	16.6K	111-160	109/4	3.8K	0.6K	0	5.9
21.	RedNotebook	17.4K	485-513	123/7	5.9K	1.2K	0	6.1
22.	LinkChecker	20.6K	891-891	235/9	3.9K	0.7K	1	6.3
23.	Solfege	20.7K	489-502	248/7	11.4K	2.4K	1	
24.	Childsplay	22.0K	929-957	233/16	8.2K	1.3K	0	8.4
25.	ScikitLearn	22.5K	403-1208	184/11	8.9K	1.9K	4	3.0
26.	Mnemosyne	26.8K	1237-1237	125/1	0.8K	0.2K	4	5.3
27.	Youtube-dl	28.5K	672-702	416/8	4.9K	1.3K	0	3.2
28.	Docutils	32.1K	45-1239	541/14	4.3K	1.7K	0	14.1
29.	PyMol	35.2K	276-281	46/10	3.3K	0.4K	0	3.2
30.	Timeline	42.3K	819-944	769/12	13.0K	2.1K	0	4.7
31.	DispcalGUI	44.1K	1030-1030	180/11	14.9K	2.2K	0	3.9
32.	PySolFC	61.9K	2143-2156	1839/7	14.1K	5.6K	0	3.0
33.	WikidPad	84.9K	1185-1292	845/34	17.0K	2.7K	0	4.3
34.	TaskCoach	101.5K	1848-2301	1230/69	22.7K	4.1K	2	9.3
35.	SciPy	130.6K	1030-1777	1074/91	15.7K	2.9K	10	2.4
36.	Eric4	177.3K	804-980	989/12	9.3K	1.1K	0	13.2
	Averages	28.6K	623-793	314/13	6.9K	1.3K	0.7	5.8

may be explained by the fact that each example that was run for Pychecker and Docutils was small and focused on explaining some specific part of the program functionality and thus did not run all of the the programs. Eric4, in turn, is an interactive program with large functionality and all functions were not executed in each run of the program.

In most programs, one or a few of the classes were loaded several times, but only in 9 of them, at least one reloaded class had more than one set of defined methods (shown in Col. "Int. diff. in Table 2). Out of these, only 4 had more than 1 redefined class with more than one set of methods. Scipy had 10 classes with multiple interfaces, SciKitLearn and Mnemosyne had 4 each and TaskCoach had 2.

The dynamism of Python classes usually does not change the interfaces of classes, but sometimes classes change during runtime. This make types difficult to predict statically and complicates typing of Python programs.

Old Style vs. New Style Classes As a result of the introduction of new-style classes, a Python class hierarchy has two possible root classes. If old style classes can be found in current programs, it would mean that the development of a type system for Python needs to account for both of these root classes. Python 3 abolishes old-style classes but has failed to achieve the popularity of Python 2.6/7, possibly because of its several backwards incompatibilities.

In our program traces, 22% of all classes were old style classes. The programs were all but five initiated after 2001, the year of the release of Python 2.2 which introduced the new-style classes as a parallel hierarchy. As shown in Figure 5, there seems to be no correlation between the program's age and the percentage of old-style classes in the program.

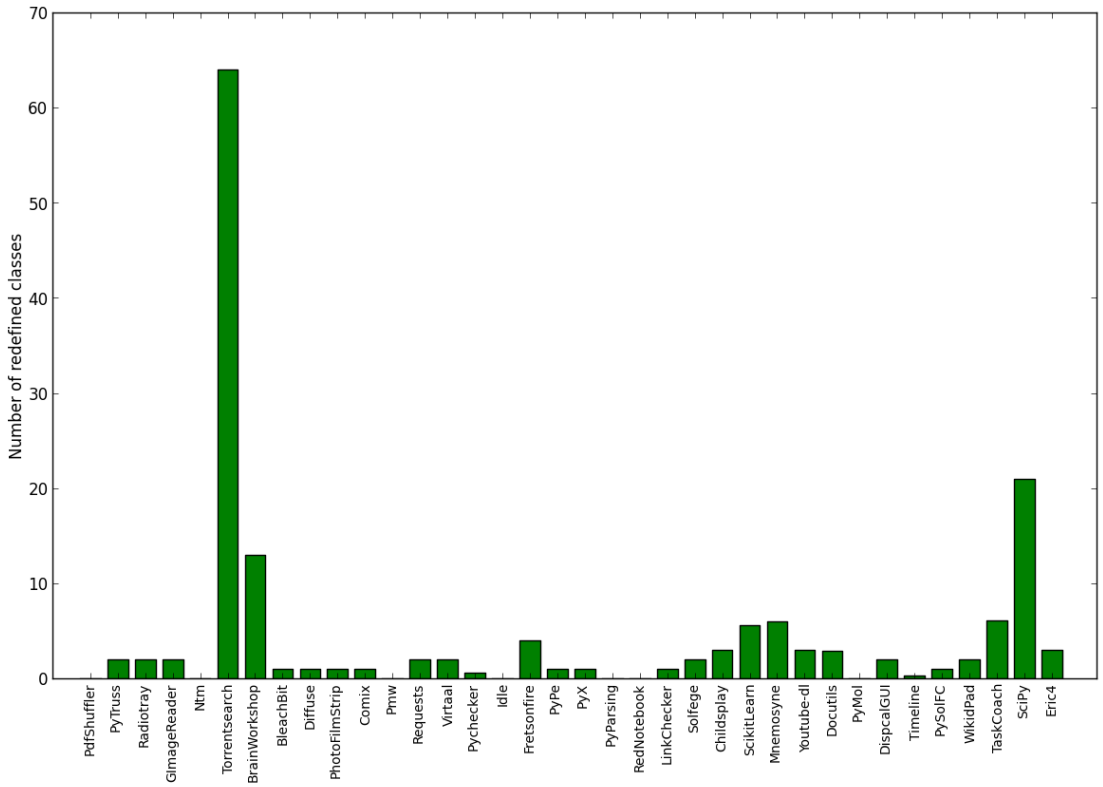

Figure 2. For all programs the number of classes for which the class definition has been loaded more than once. Programs sorted on size in LOC.

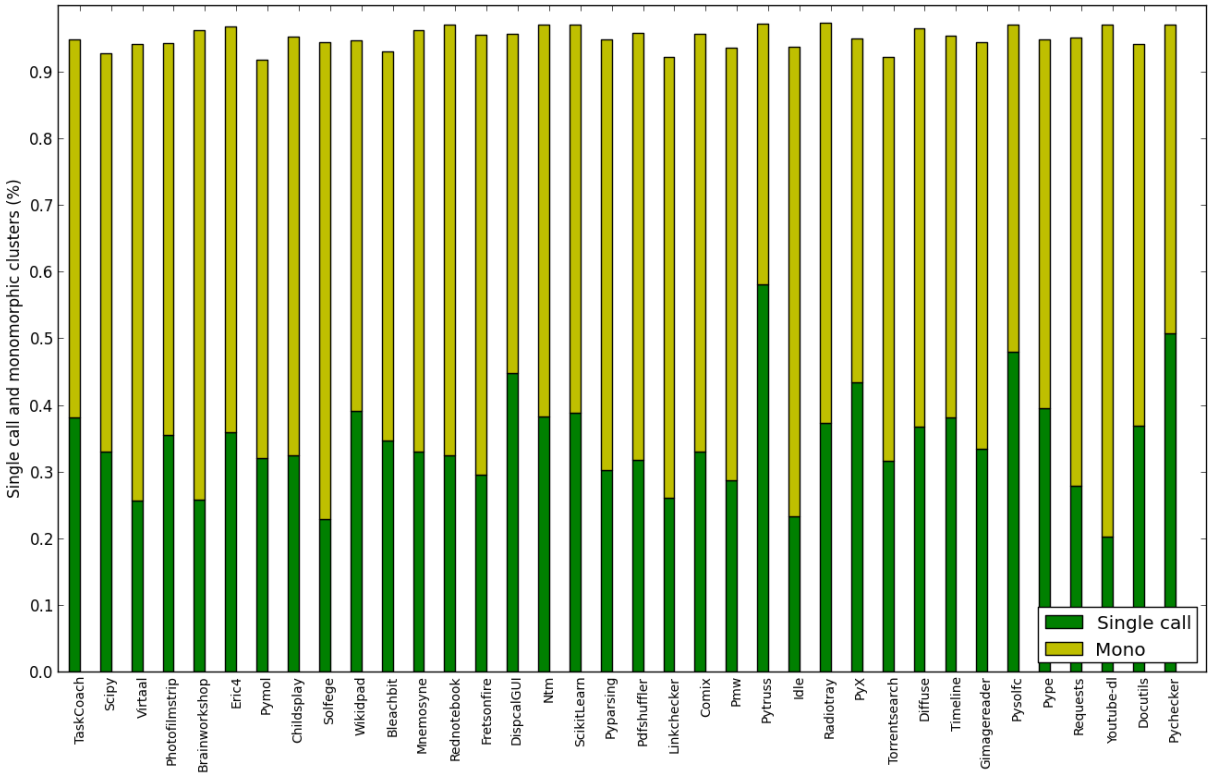

Figure 3. For all programs the average shares (in %) of the clusters that were single call and monomorphic.

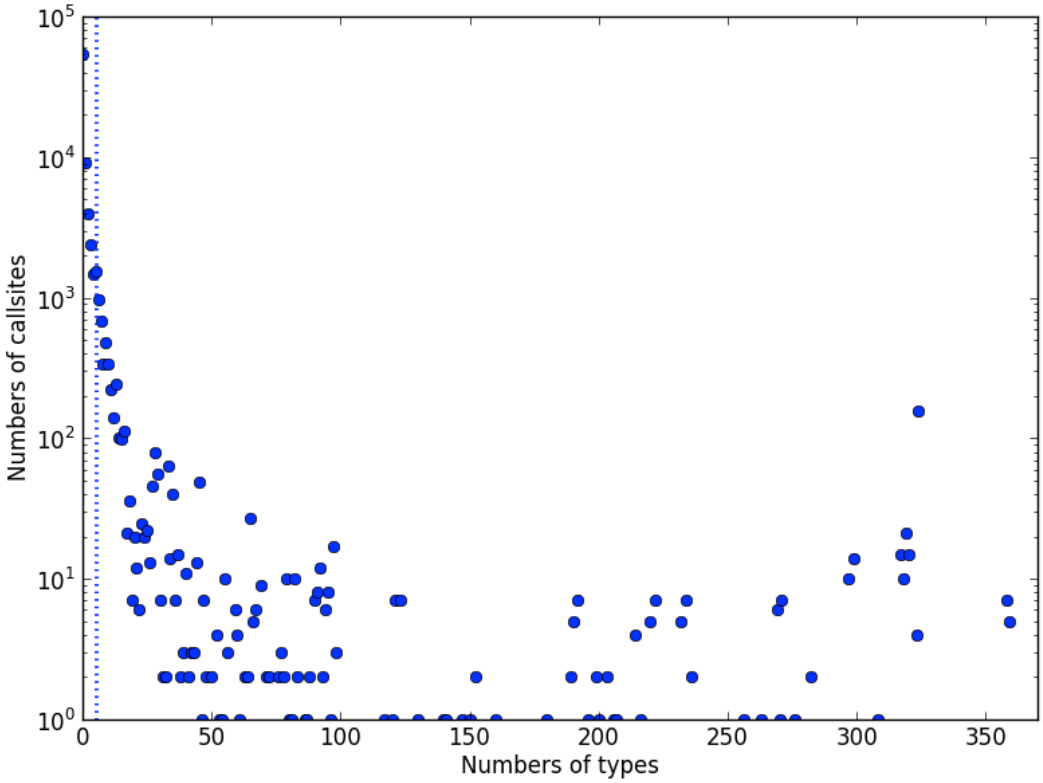

Figure 4. Call-sites/receiver types.

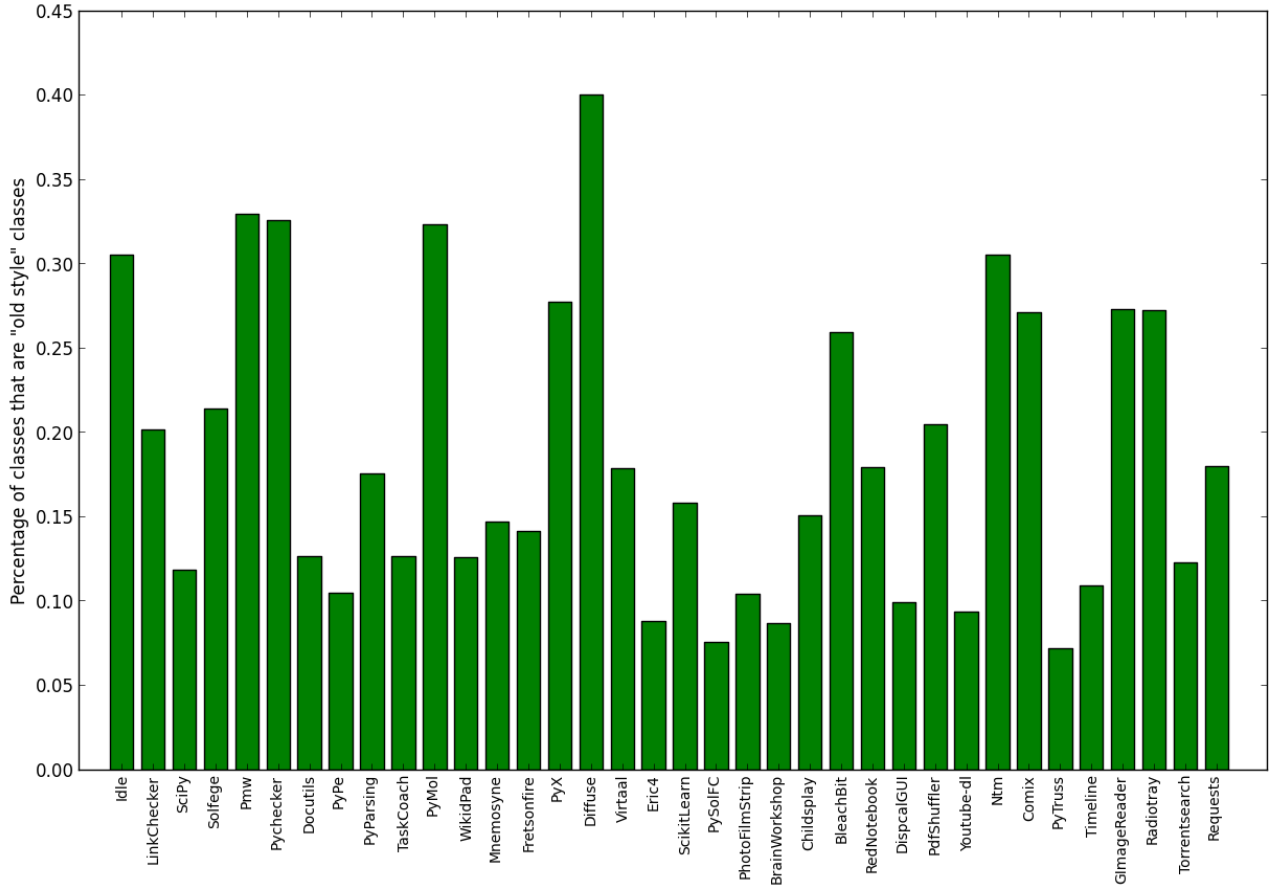

Figure 5. The percentage of traced classes that were “old style”. Programs sorted on age with the oldest to the left and the youngest to the right.

For the programs started before 2001, this likely means that many old style classes have been changed into new style equivalents (the use of old style classes has been strongly discouraged). Many of the old-style classes were imported from libraries, both standard libraries and third party libraries.

A pattern to reduce the amount of old-style classes found in several programs in our corpus is the insertion of an explicit derivation from `object` in addition to its old style superclasses, which increases the use of multiple inheritance.

We conclude that the use of old-style and new-style classes in parallel means that a type system for Python has two choices: it either must account for two root classes, or it must exclude (support for) old libraries and require changes to commonly more than a fifth of all classes.

Use of Multiple Inheritance All programs use multiple inheritance, ranging from 2.4% to 17.5% of all classes with an average of 5.9% (see Col. "Multiple Inheritance" in Table 2). These are the figures after removing any multiple inheritance due to the pattern for making old-style classes into new-style classes mentioned above in Section § 5.1. Multiple inheritance is found both in library classes and program-specific classes. Classes used as superclasses in multiple inheritance are also both library classes and program-specific classes.

5.2 Extent and Degree of Polymorphism

Overridding In our analysis to decide if a call-site is *N-typeable* or *NPP-typeable*, (see Def. 5, Def. 6) we first look for a common super type for all receiver types. If such a type is found, the second step is to check if the method called at the call-site can be found in that type. Thus, to be be *N-typeable* or *NPP-typeable*, the program needs method overriding. Such overriding is at times required in statically typed code leading to the insertion of abstract methods to be allowed to call methods on a polymorphic type[1]. Since there is no such need in dynamically typed programs, this analysis is in this respect a conservative approximation.

If the method has been overridden in all subclasses, execution of the call-site will lead to execution of different methods with potentially different behaviour for every receiver type. A program designed in this way is arguably more polymorphic than if all executions of the call-site leads to a call to the same method in the superclass. On the downside, method overriding makes code harder to read, understand and debug due to the increased complexity of the control flow.

Column 6 ("Avg. # overr.") in Table 2 shows the average number of overridden methods per program, that is the number of methods that are redefinition of inherited methods. Comparing with column 5 in the same table ("Avg. # meth.) we can see that 19% of all methods are re-definitions of methods inherited from some superclass. This suggests that our Python programs are quite object-oriented, and use its object-oriented concepts similar to statically typed languages like Java.

[1] In a statically typed language, classes `B` and a class `C` both with a method `m()` with a common supertype `A`, the supertype could be used as a static type for objects of `B` and `C`, but we could not make calls to `m()` through a variable declared as `A` unless `A` also contains a definition of `m()`. This way, overriding is necessary in statically typed languages in a way that it is not in a dynamic language.

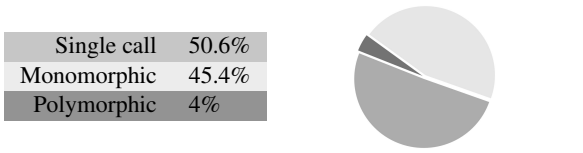

Figure 6. Distribution of call- sites between polymorphic, single call and monomorphic in whole programs.

Individual Call-Site Polymorphism To give a high-level overview of the polymorphism of a program, we classify call-sites depending its measured degree of polymorphism. A call-site is either monomorphic, polymorphic or megamorphic. A fourth category, single call, was added to avoid classifying call-sites observed only once as monomorphic.

For all program runs, the share of monomorphic call-sites (including single call) ranged between 88–99% with an average of 96% (see Figure 6). This means that in most programs only a very small share of the call-sites exhibits any receiver-polymorphic behaviour at all. To avoid wrongful classifications due to bad input or non-representative runs, all programs were run multiple times. The amount of monomorphic and single call call-sites did not vary significantly between different runs of the same program, including uses of the same library by different programs, as shown by the error bars in Figure 8.

Single call call-sites accounted for 27–81% of the total number of call-sites for all runs of all programs with an average of 51% and a median at 49%.

Monomorphic call-sites are always typeable since all receivers have the same run-time type. Single call call-sites are typeable for the same reason, at least for that run of the program. Many call-sites would still be single call even if input was increased/made more complex, etc.

The table below shows the degree of monomorphism, polymorphism and megamorphism for all the programs sorted by increasing size (in terms of lines of code). There seems to be no correlation between program size and the ration of monomorphism, polymorphism and megamorphism. The polymorphism for the smaller programs (numbers 1–18) is similar to the polymorphism in the larger programs (numbers 19–36). We perform a t-test (two-tailed, independent, equal sample sizes, unequal variance) with null hypothesis that the average degree of polymorphism is the same in the small programs and in the large programs. Column 5 contains the result, confirming the hypothesis for all degrees of polymorphism. All values are lower than (α=0.05,d.f.=17) = 2.110.

Figure 7 shows the maximal polymorphic degree for all runs of all programs, ranging from 2 to 356 receiver types. The average maximal polymorphic degree in the programs in

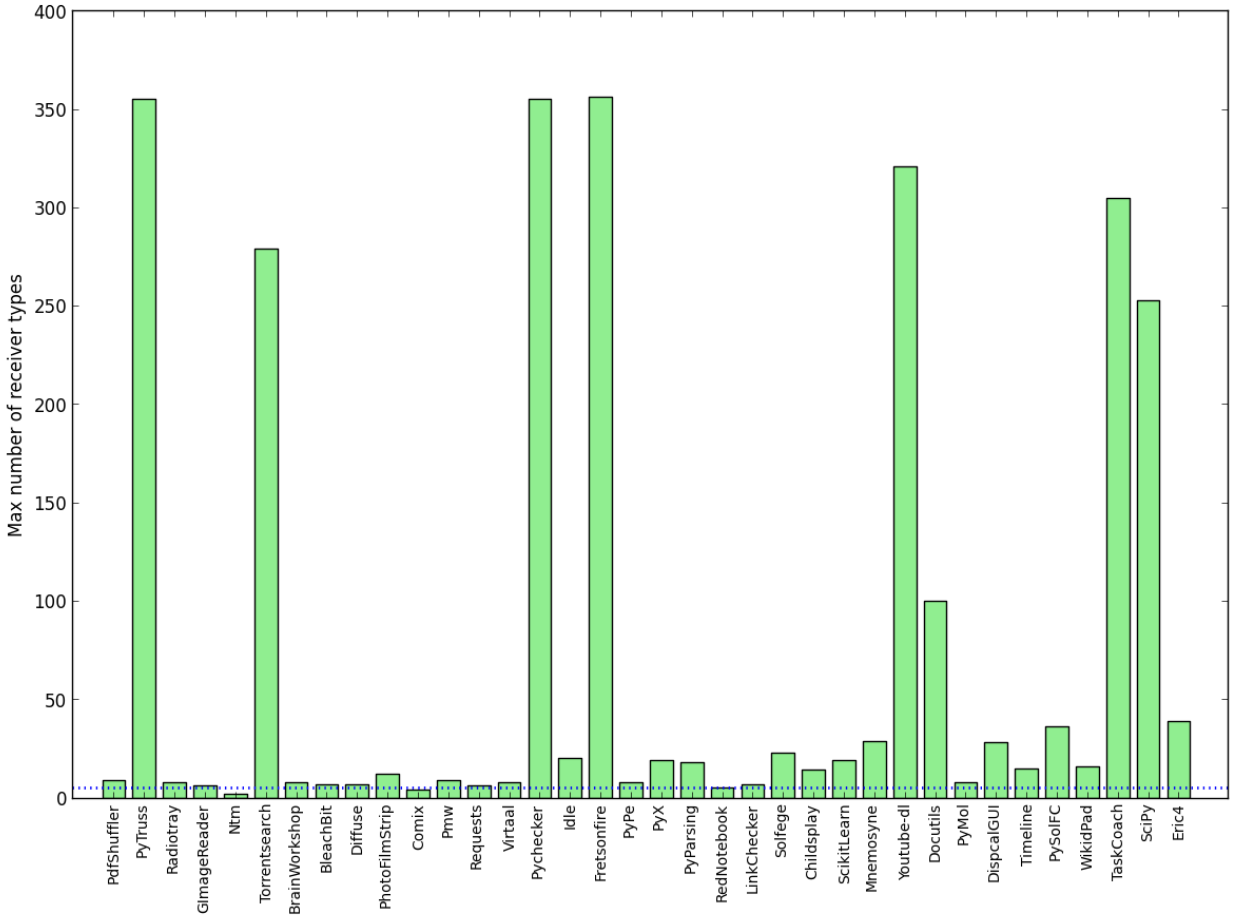

Figure 7. For all programs, the polymorphism max values.

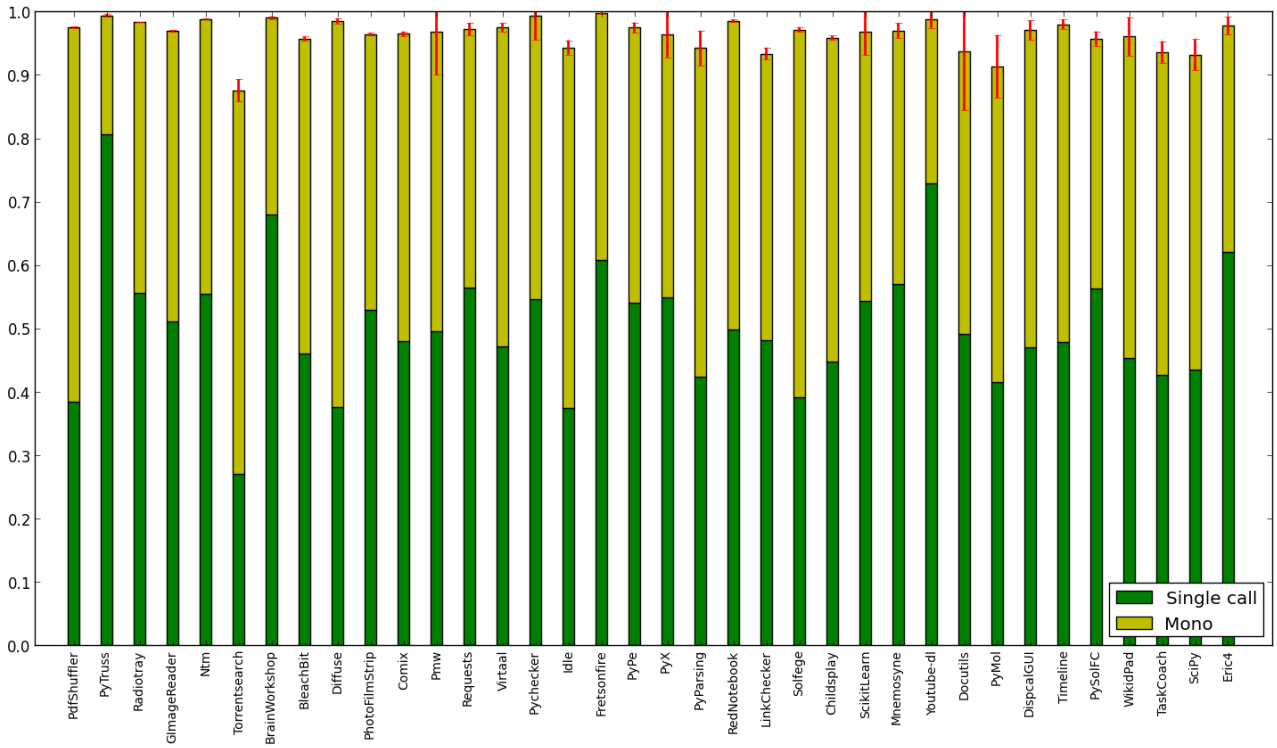

Figure 8. For all programs the average shares (in %) of the call-sites that were single call and monomorphic. Error bars shows the distance between max and min values. Sorted on size in LOC.

our corpus was 75 and the median 27. The blue dotted line marks the border between polymorphism and megamorphism at 5 receiver types. Only 3 programs contain no megamorphic call-sites at all (Ntm 2, Comix 4 and RedNotebook 5).

7 of 36 programs had at least one call-site with a very high number of receiver types—close to or above 10 times the average maximum. The maximal degree of polymorphism in these programs (PyTruss 355, Torrentsearch 279, Pychecker 355, Fretsonfire 356, Youtube-dl 321, TaskCoach 305 and SciPy 253 respectively) was much higher than in the other programs. There seems to be no correlation between program size and the programs with high degrees of polymorphism. The programs that contained the highest polymorphism are distributed evenly over Table 1 which is sorted on program size, although the concentration is somewhat higher at the bottom of the table (larger programs). The programs with highest maximum polymorphism are number 17, 15, 2, 27, 34, 6 and 35 (descending).

Column 2 of Table 1, "Whole – P+M %", shows the proportions of the call-sites that were polymorphic and megamorphic for each program (program averages). There is no strong correlation between the size of the program and the degree of polymorphism. The average of the upper half of the table is 3.1%, the average for the lower part of the table is 4.0% and the average for the whole is 3.5%. Which means that the larger programs contain more polymorphism but the difference is only 25.4%. Both the programs with the highest share of polymorphic and megamorphic call-sites (Torrentsearch, 12%) and the program with the lowest share (Brainworkshop, 1%) are small programs. They both have less than 5K lines of code, which is well below both the average and the median sizes.

Cluster Polymorphism We apply the same classification for individual call-sites to clusters. This reduces the size of the single call category, as call-sites involving the same receiver will be placed in a single cluster. The size of the category is still large, which could suggest that it is common to create objects and operate on them only once.

On average, 35% of all clusters are single call, ranging from 20% in Youtube-dl to 58% in Pytruss as shown in Figure 3. The monomorphic clusters, shown in the same figure, were on average 61% of all clusters for the programs, ranging from 46% in Pychecker to 77% in Youtube-dl. The single call is lower for the cluster analysis and the monomorphic higher, compared to the call-site analysis. The overall result is that the monomorphic share of is slightly lower for clusters than for call-sites, on average 95.2% (0.8%).

Most clusters are small, 59% contained only 1 call-site[2] (which may have been observed multiple times with possibly different receiver types). The largest cluster had 2.720 call-sites. The average cluster size was 5.

Degree of Polymorphism at Individual Call-sites The degree of polymorphism at a call-site is the number of different receiver types we observed at that call-site.

Figure 4 shows the degree of polymorphism for all polymorphic and megamorphic call-sites. In Figure 7 and in Figure 4, the border between polymorphism and megamorphism is represented by the dotted line. The vast majority, 88%, of all polymorphic and megamorphic call-sites are not megamorphic (69,367 polymorphic against 7,870 megamorphic).

Table 3. Polymorphism of small/large programs in Table 2

	Prog. 1–18	Prog. 19–36	All	Student's t-test (α=0.05,d.f.=17) = 2.110
Single call	49.7	50.4	50.1	-0.06
Monom.	46.9	45.3	46.1	0.02
Polym. (2)	2.3	2.9	2.6	0.02
Polym. (3)	0.52	0.45	0.47	0.001
Polym. (4)	0.17	0.27	0.23	0.005
Polym. (5)	0.10	0.10	0.14	0.003
Megam.	0.34	0.30	0.38	0.01

Table 4. A list of the programs, sorted on size (see Table 2), followed by 7 columns showing the percent of the total amount of call-sites that were single-call (S-C), monomorphic (Mono), or polymorphic to different degrees up to megamorphic (types >5). Finally, in column 8, also the percent of the megamorphic call-sites for every program that was found in library code.

		Call-sites with N receiver types							
	Program name	S-C	Mono	2	3	4	5	>5	%Lib.
1.	PDF-Shuffler 0.6.0	38%	59%	2%	0	<1%	<1%	2%	100
2.	PyTruss	80%	19%	1%	<1%	<1%	<1%	1%	100
3.	Radiotray 0.6	56%	43%	1%	<1%	<1%	0	<1%	100
4.	GImageReader 0.9	51%	46%	2%	1%	<1%	<1%	<1%	100
5.	Ntm 1.3.1	56%	43%	1%	0	0	0	0	-
6.	Torrent Search 0.11-2	27%	61%	7%	4%	1%	<1%	1%	43
7.	Brain Workshop 4.8.1	68%	31%	1%	<1%	<1%	0	<1%	100
8.	BleachBit 0.8.0	46%	50%	3%	<1%	<1%	<1%	<1%	0
9.	Diffuse 0.4.3	38%	61%	1%	<1%	0	<1%	<1%	0
10.	PhotoFilmStrip 1.5.0	53%	44%	3%	<1%	<1%	<1%	<1%	100
11.	Comix 4.0.4	48%	49%	3%	<1%	<1%	0	0	-
12.	Python megawidgets	49%	48%	2%	<1%	<1%	<1%	<1%	99/-
13.	Requests 2.2.1	58%	39%	3%	<1%	<1%	<1%	<1%	1
14.	Virtaal 0.6.1	47%	50%	2%	<1%	<1%	<1%	<1%	95
15.	Pychecker 0.8.18-7	56%	42%	2%	<1%	<1%	<1%	<1%	100
16.	Idle 2.6.6-8	37%	57%	5%	1%	<1%	<1%	<1%	100
17.	Frets on fire 1.3.110	59%	39%	1%	<1%	<1%	<1%	1%	89
18.	PyPe 2.9.4	54%	43%	2%	<1%	<1%	<1%	<1%	58
19.	PyX 0.10-2	53%	43%	3%	<1%	<1%	<1%	<1%	0
20.	Python parsing 1.5.2-2	42%	52%	2%	1%	<1%	<1%	2%	0
21.	RedNotebook 1.0.0	50%	49%	1%	<1%	<1%	<1%	0	-
22.	Link checker 5.2	48%	45%	4%	1%	<1%	<1%	<1%	28
23.	Solfege 3.16.4-2	39%	58%	2%	<1%	<1%	<1%	<1%	6
24.	Childsplay 1.3	45%	51%	3%	1%	<1%	<1%	<1%	19
25.	Scikit Learn 0.8.1	54%	43%	2%	<1%	<1%	<1%	<1%	1
26.	Mnemosyne 2.1	57%	40%	2%	<1%	<1%	<1%	<1%	84
27.	Youtube-dl 2013.01.02	76%	22%	1%	<1%	<1%	0	<1%	69
28.	Docutils 0.7-2	43%	49%	5%	1%	<1%	<1%	1%	1
29.	PyMol 1.2r2-1.1+b1	40%	50%	7%	<1%	2%	<1%	<1%	100
30.	Timeline 1.1.0	47%	51%	1%	<1%	<1%	<1%	<1%	100
31.	DispcalGUI 1.2.7.0	47%	50%	2%	<1%	<1%	<1%	<1%	50
32.	PySolFC 2.0	56%	40%	2%	1%	<1%	1%	1%	26
33.	WikidPad 2.1-01	45%	51%	3%	<1%	<1%	<1%	<1%	10
34.	Task Coach 1.3.22	42%	51%	4%	1%	<1%	<1%	1%	10
35.	SciPy 0.7.2+dfsg1-1	44%	49%	5%	1%	<1%	<1%	1%	54
36.	Eric4 4.5.12	62%	36%	2%	<1%	<1%	<1%	<1%	0
	Averages	50%	46%	2.6%	<1%	<1%	<1%	<1%	52

[2] This means that in a source file there was just one single place that manipulated (a) certain value(s).

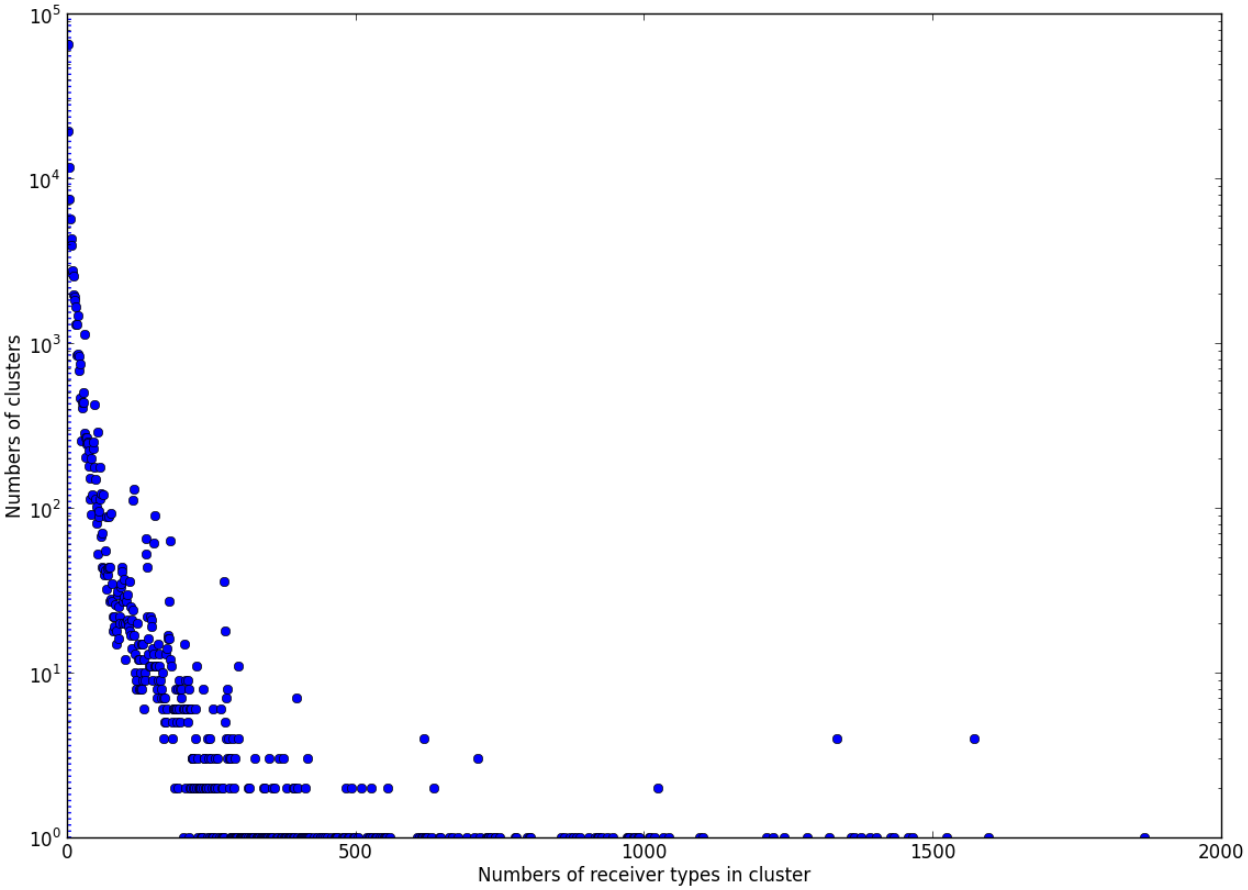

Figure 9. Polymorphic degree of the clusters of polymorphic and megamorphic clusters.

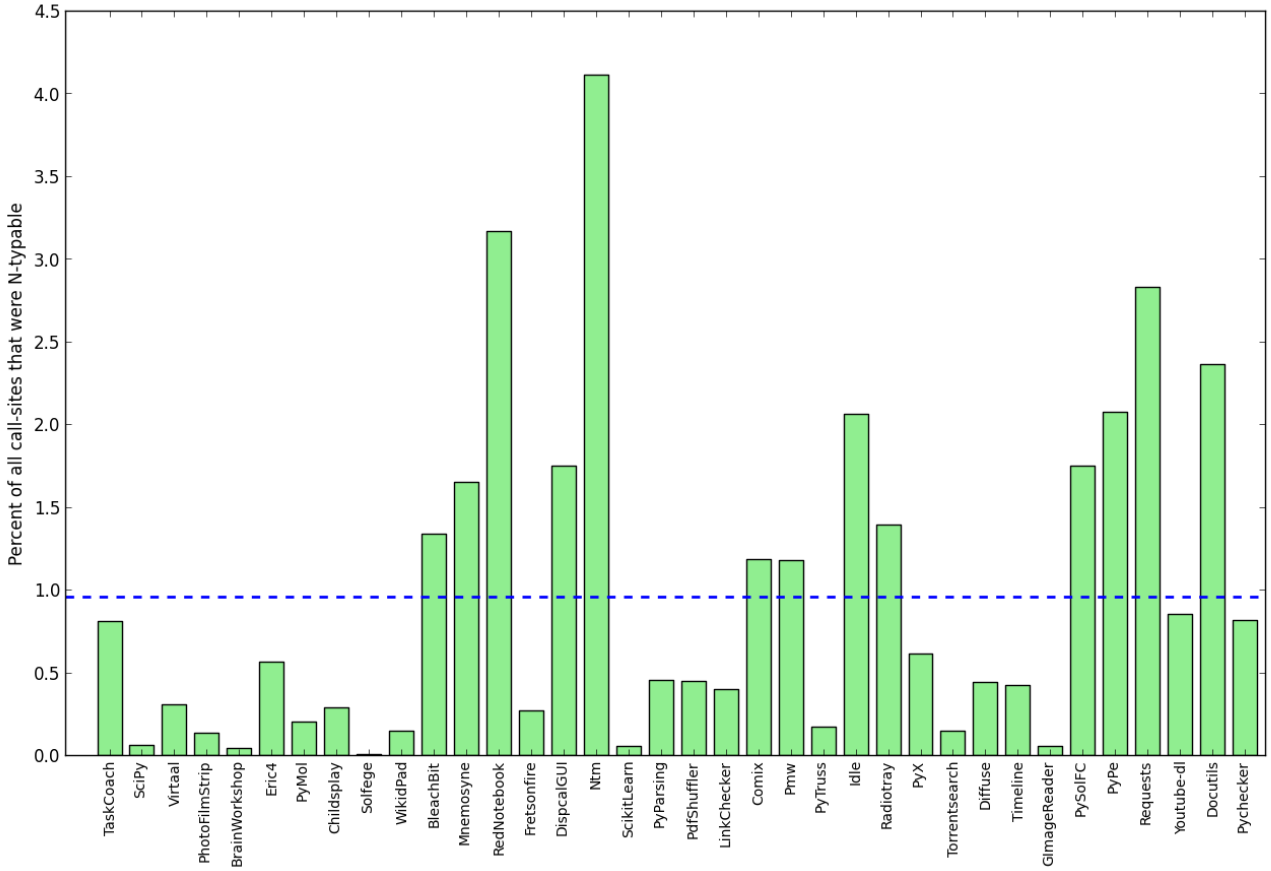

Figure 10. N-Typeable call-sites.

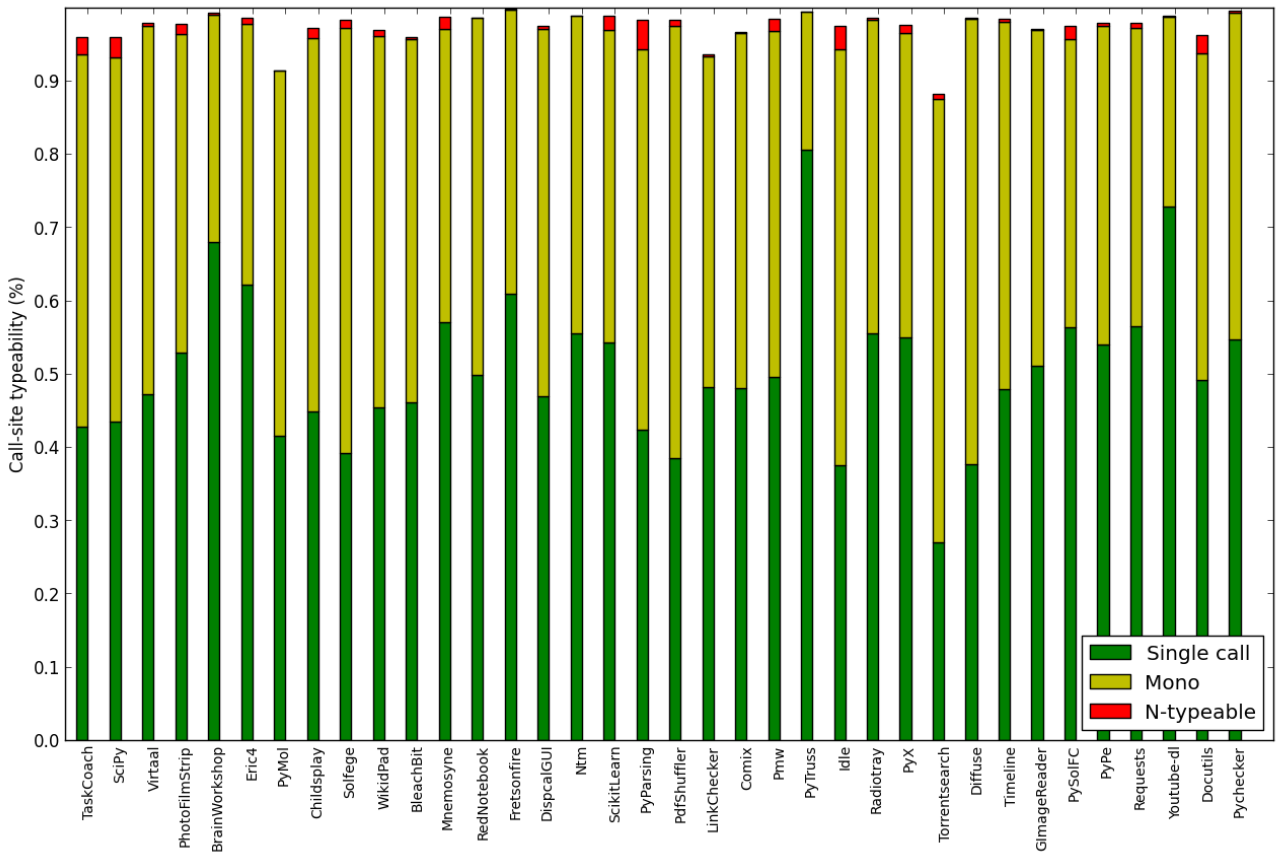

Figure 11. N-Typeable, single call and monomorphic call-sites.

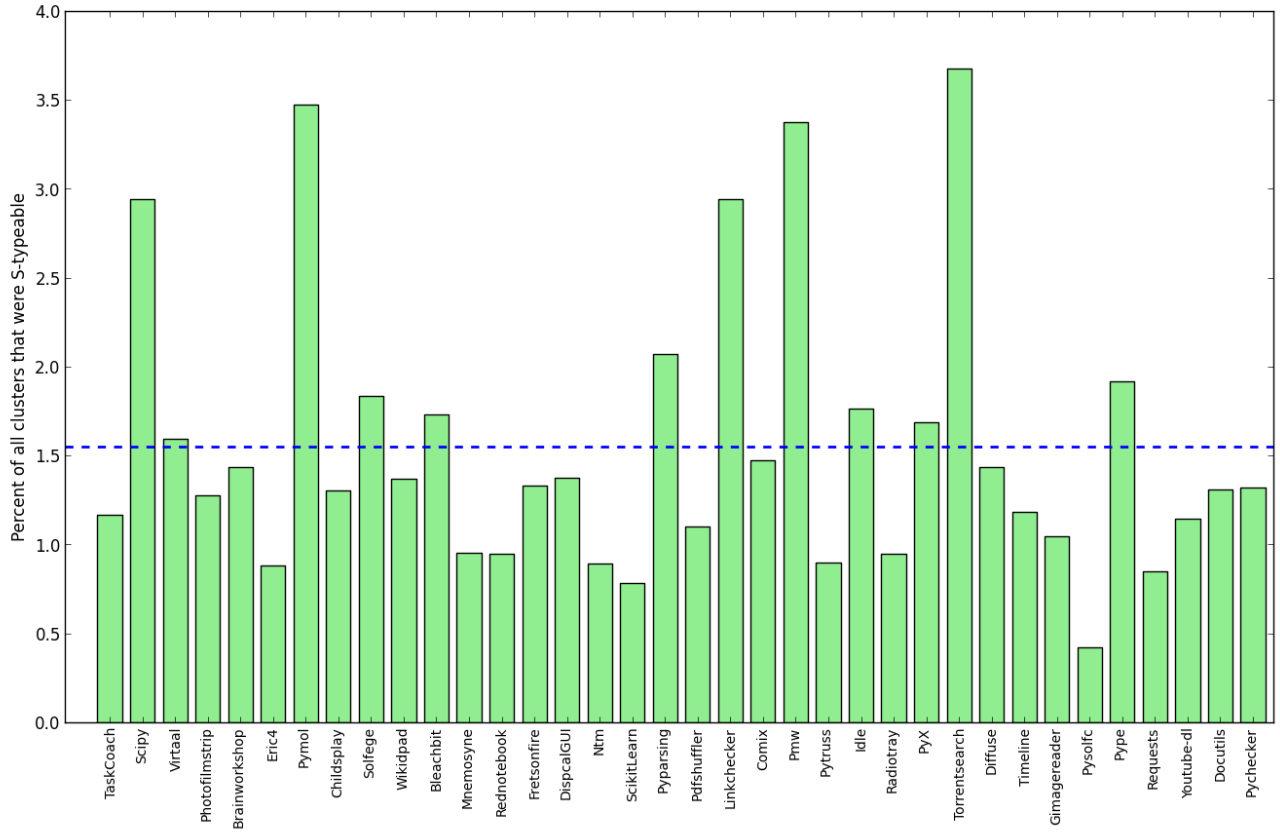

Figure 12. The % of all clusters that were S-Typeable.

78% of the polymorphic call-sites had a polymorphic degree of 2, that is two different receiver types.

While these numbers show that megamorphic call-sites are relatively rare, they are not concentrated to specific programs. Almost all programs (33 of 36) exhibited some form of megamorphic behaviour, see Table 4, Column 9, "Call-sites with N receiver types >5". The programs without megamorphic call-sites were Ntm, Mcomix and Rednotebook. In 83% of all programs (30 out of 36), 1% or less of all call-sites were megamorphic. The largest share of megamorphic call-sites, 2%, were seen in PdfShuffler and Python parsing.

Manual Inspection To better understand their nature, we investigated the receiver types of the extremely megamorphic call-sites for the five programs with the highest megamorphic maximum value (Pytruss, Torrentsearch, Frets on fire, Youtube-dl and Scipy)

In two of these programs (Pytruss, Frets on fire), the same OpenGL library was the main cause of megamorphism and all call-sites of degree >10 (Pytruss) or >50 (Frets on fire) originated from calls on OpenGL objects. Frets on fire also had some very program-specific receivers in megamorphic call-sites such as songs, menus, etc., related to the game.

For Torrentsearch and Youtube-dl, the megamorphism stemmed from the singleton class implementation of the representation of different torrent sites or sites from which content could be downloaded. For Youtube-dl, the megamorphism also varied a lot between runs.

The very high megamorphic values in SciPy with degrees >100 were all caused by testing frameworks (the built-in `unittest` or the `nose` unit test extension framework). All call-sites with a megamorphic degree <100 and >22 were either part of the testing frameworks or used to create distributions using generator classes. The call-sites with a megamorphic degree <23 and >5 often used arguments of different classes to handle different shapes or components used to create plots.

From the manual inspection, it seemed that to a large extent, megamorphism is due to patterns emerging from convenience (the simplicity to create specific, singleton classes at run-time in Python), and not from an actual need to create widely different unrelated classes. Thus, in many cases, it is possible to reduce megamorphism by redesigning how classes are used. Nevertheless, the proliferation of megamorphism (by convenience or not) must be considered by retrofitted type systems for Python.

Degree of Polymorphism in Clusters The degree of polymorphism in a cluster is the number of receiver types we observed at that call-site.

Figure 9, shows the degree of polymorphism for all polymorphic and megamorphic clusters. The border between polymorphism and megamorphism is represented by the dotted line. Similar to the call-sites, the majority, 67%, of all polymorphic and megamorphic clusters are non-megamorphic. 42% of the polymorphic call-sites had a polymorphic degree of 2 (i.e., two different receiver types).

Polymorphism in Library Code vs. Program-specific The columns under "Lib." and "Prog." in Table 1 shows the share of all call-sites from library code and program specific code that were polymorphic or megamorphic. Assuming that polymorphism on average does not differ between library code and program-specific code we ran a statistical test (a Student's t-test, two-tailed, independent, equal sample sizes, unequal variance) comparing all data all 28 programs where the separation of libraries and program-specific code was made. The result was that the hypothesis holds for all of the programs. For α=0.05, and a degree of freedom that ranges from 1 to 56, and a p-value ranging from 1.98 to 12.706, the t-values ranged from -0.44 to 0.12.

To uncover differences between megamorphism in libraries and program code, we manually inspected all megamorphic call-sites of all programs to see if they were found in libraries or in program-specific code. No clear pattern emerged and, on average, 59% of the megamorphic call-sites originated from library code. As shown in the last column of table Table 1, the share varied from 0 to 100%. For 10 of the programs (28%) all megamorphism stemmed from library code. Only 5 programs had none of their megamorphic call-sites in the library code (14%).

Polymorphism at Start-up vs. Runtime Using the marker we inserted in all programs to separate the programs' start-up time from the actual runtime, we separated the trace data gathered during start-up from that gathered during "normal program execution". This was only done for interactive programs (24/36) as it was relatively easy to identify the end of the start-up for those programs as the time control is handed over to the main event loop waiting for user input. Remaining programs are marked with a "–" in the columns under "Startup" and "Runtime" in Table 1. Assuming first that polymorphism does not differ between start-up and runtime we ran a statistical test comparing all runtime data to all start-up data for all 24 programs where the separation of runtime and start-up data was done (a Student's t-test, two-tailed, independent, equal sample sizes, unequal variance). This test, and the assumption, fails for all but one of the tested programs. Since the t-values in all cases except one (23/24) are negative we conclude that the average polymorphism during startup is lower than the average polymorphism during runtime.

Notably, all program traces contain a *lower* degree of polymorphic and megamorphic call-sites during start-up compared the whole program run. This is an interesting find given that RPython [4] is based on the idea that programs are *more* dynamic during start-up, limiting the use of the more dynamic features of Python to an initial, bootstrapping phase. About 1% of the call-sites seen at start-up were polymorphic or megamorphic. During normal program execution, on average, 5% of all call-sites were polymorphic or megamorphic.

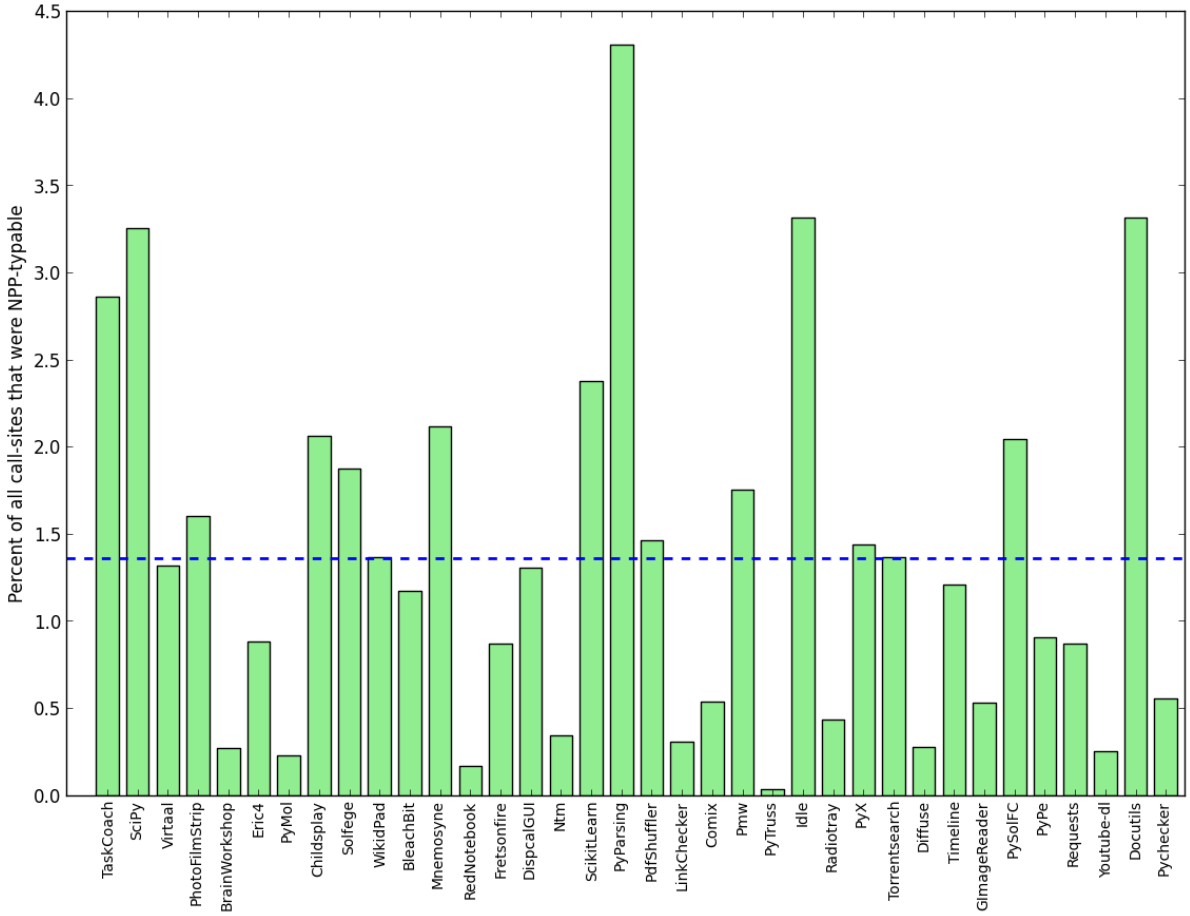

Figure 13. NPP-Typeable call-sites.

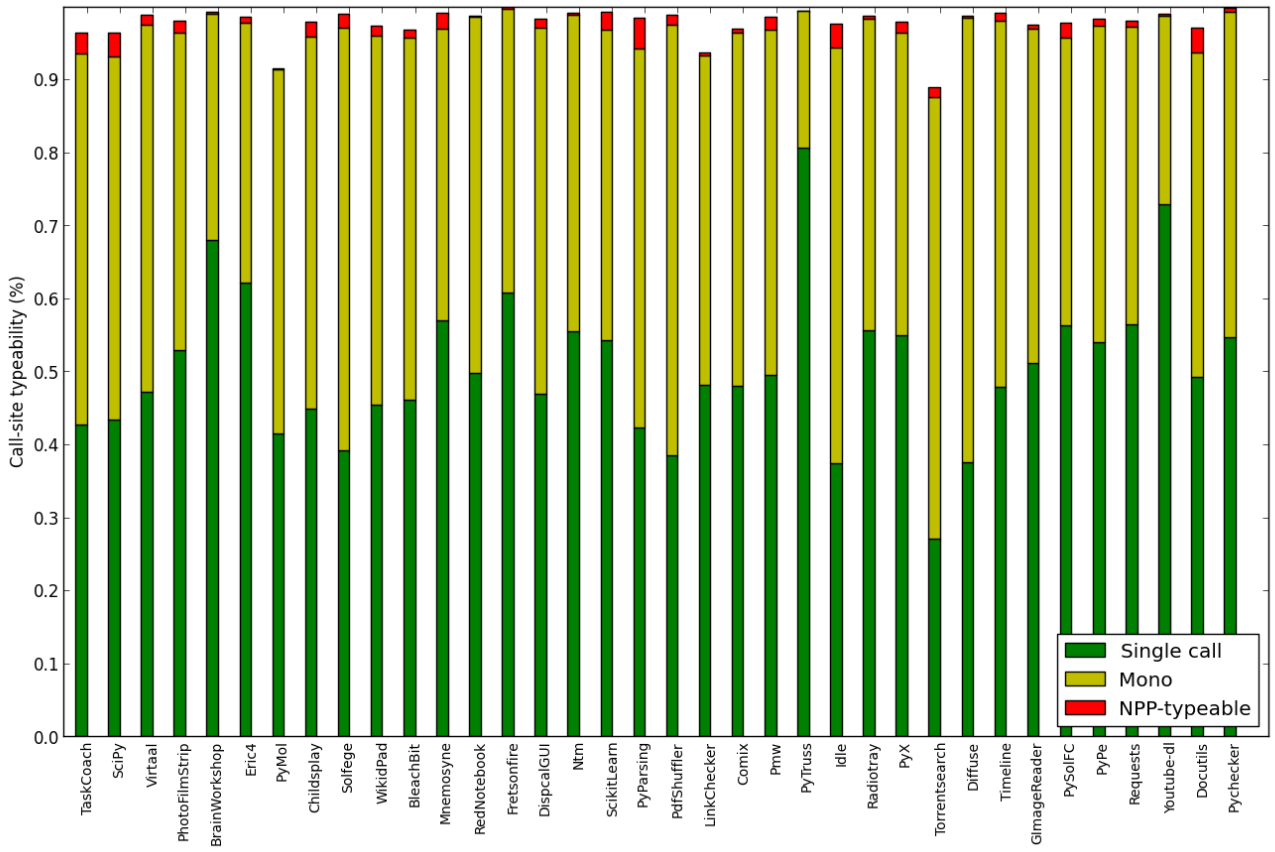

Figure 14. NPP-Typeable, single call and monomorphic call-sites.

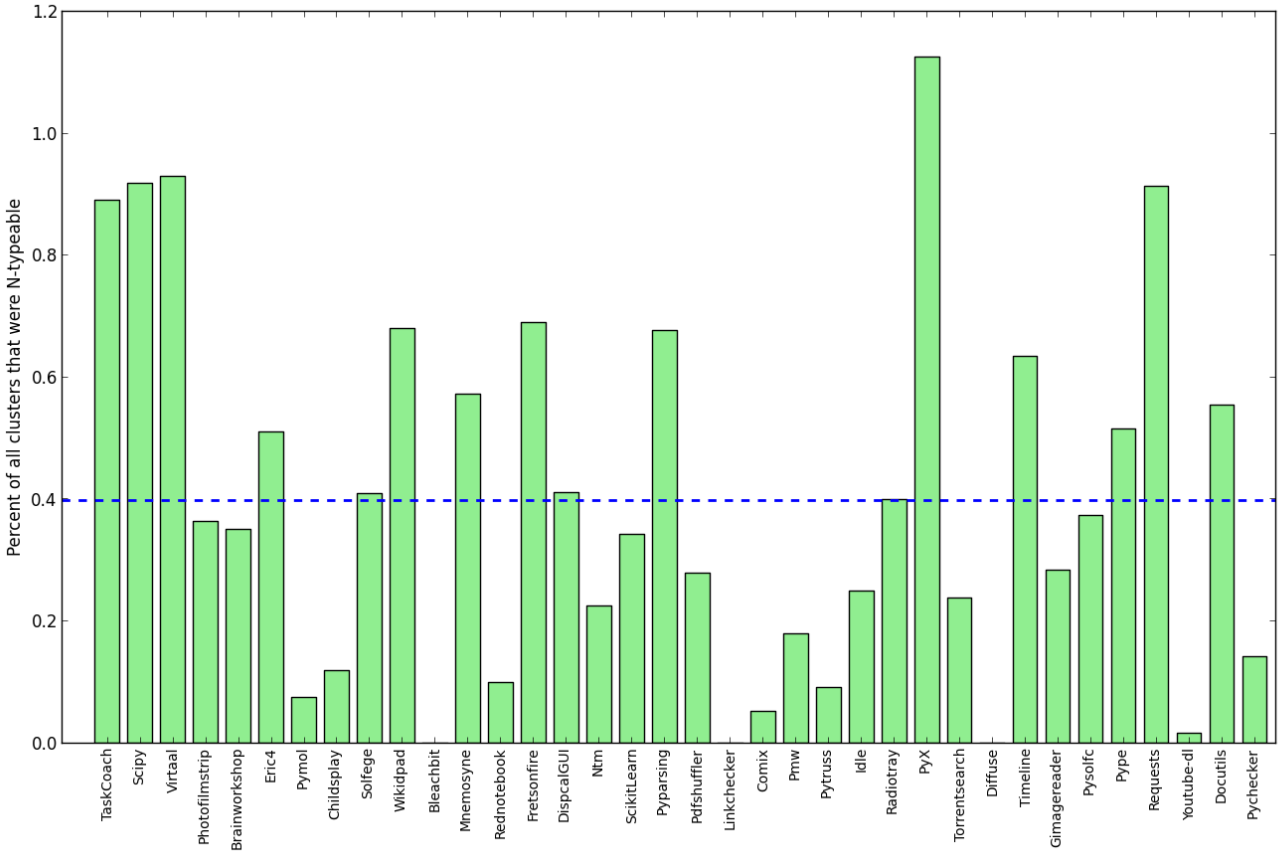

Figure 15. The % of all clusters that were N-Typeable.

5.3 Typeability

We applied our three metrics for approximating typeability using nominal, nominal and parametrically polymorphic, and structural typing to the call-sites and clusters in our trace logs.

N-Typeable Call-sites Figure 10 shows the percentage of all call-sites that were polymorphic or megamorphic and *N-typeable*. In Figure 10 as well as in column 4 ("Whole program"/"N-t") of Table 1, all programs contain call-sites that are *N-typeable*, although the *N-typeable* share of the call-sites is always low. In column 3 ("Whole program"/"Typeable Poly") of Table 1 we see the *N-typeable* shares of the non-monomorphic parts of the programs. The dashed line in Figure 10 marks the average value at 0.96%. The program with the highest amount of *N-typeable* call-sites was Pyparsing with 4.11%, which also had the highest *N-typeability* share, 72.0%, when considering only non-monomorphic call-sites.

The average of the upper half of Table 1 (the smaller programs), was 0.9%. The programs in the lower half of the same table (the larger programs) had a slightly higher average for the *N-typeability* (1.1%).

Figure 11 shows the amount of *N-typeable* call-sites on top of the shares of monomorphic (which are always typeable) and the single call (which are typeable for this run of the program). In combination, they can be used to type between 88.1% and 99.9% of the call-sites, with an average at 97.4%.

In conclusion, most call-sites in Python programs are not polymorphic or megamorphic, but when they are, our simple and conservative nominal types cannot in general be used to type them.

NPP-typeable Call-sites All call-sites that are *N-typeable* (see Def. 5) are also *NPP-typeable* (see Def. 6), but the *NPP-typeability* analysis increases our possibilities to find typeable call-sites.

All call-sites in the programs, that were not *N-typeable*, were sorted and separated on the identity of the caller, that is the identity of `self` at the time when the call was made. After this separation, we again search for a common supertype for all receiver types and if found check if that supertype contains the method called at the call-site.

Figure 13, shows for each program the percentage of all call-sites that were polymorphic or megamorphic and *NPP-typeable*. All programs contain call-sites that are *N-typeable*, and we for all programs find that they are *NPP-typeable* to a higher extent than they are *N-typeable*. The dashed line in Figure 13 marks the average value at 1.34%. As was the case for our *N-typeable* analysis, the program with the highest amount call-sites was Pyparsing also for the *NPP-typeability* analysis, with 4.31%.

Figure 14 shows the amount of *NPP-typeable* call-sites on top of the shares of monomorphic (which are always typeable) and the single call (which are typeable for this run of the program). In combination, they can be used to type between 88.9% and 100% of the call-sites, with an average at 97.8%.

By extending the nominal type system with parametric polymorphism, we can type more call-sites for all programs. For one program, Frets on Fire, we could even type all call-sites. But for the rest of the programs, our simple and conservative nominal types are not powerful enough even when extended with parametric polymorphism.

N-Typeable Clusters The figures reported in this section to this point are optimistic as they only consider individual call-sites. We apply the same analysis to clusters of call-sites as discussed in § 4.

For all the polymorphic and megamorphic clusters, we search for a common supertype among the receiver types that contains all methods called in the call-sites of the cluster. If the methods were found, the cluster is *N-typeable* (see Def. 8).

Figure 15, shows the results of applying our *N-typeability* analysis on all polymorphic or megamorphic clusters. The staples represent the % of all clusters that were *N-typeable* and the dashed blue line marks the average at 0.4%. The highest typeability, 1.1% was found in Torrentsearch, and the lowest, 0% in both Mnemosyne and Comix. This result is, as expected, lower compared to the *N-typeability* analysis we made for call-sites.

When we combine the *N-typeability* with the single call and monomorphic clusters, as shown in Figure 16, between 91.9% and 97.8% of the call-sites are typeable in the 36 programs, with an average at 95.6%. This result is lower than the typeability we reached for call-sites, as expected.

In conclusion, clusters in the programs of our corpus are predominately monomorphic or single call. When a cluster is polymorphic or megamorphic, nominal types cannot in general be used to type them.

S-Typeable Clusters The shares of program clusters that were *S-typeable* (Def. 9) are shown in Figure 12. On average, 1.6% of all the clusters are *S-typeable*, with a minimum at 0.4% in Pype and a maximum at 3.7% in Diffuse.

As with nominal typing, we can combine the *S-typeable* clusters with single call and monomorphic clusters to find out how large parts of the programs we could type in total. Figure 17 show the results. Combined, these three typeable shares give a typeability of 96.7%, on average. Lowest in BleachBit with 94.8% and highest in PyChecker with 98.4%.

Unsurprisingly *S-typeability* analysis for clusters gives a higher overall typeability (12.0% higher) than achieved with *N-typeability*, but no program can be typed to more than 98.4%. Again, monomorphism dominates the programs. The small parts that are polymorphic and megamorphic cannot be typed entirely using a structural approach.

6. Threats to Validity

Validity of our findings is affected by several decisions and choices. The program selection was not made entirely at ran-

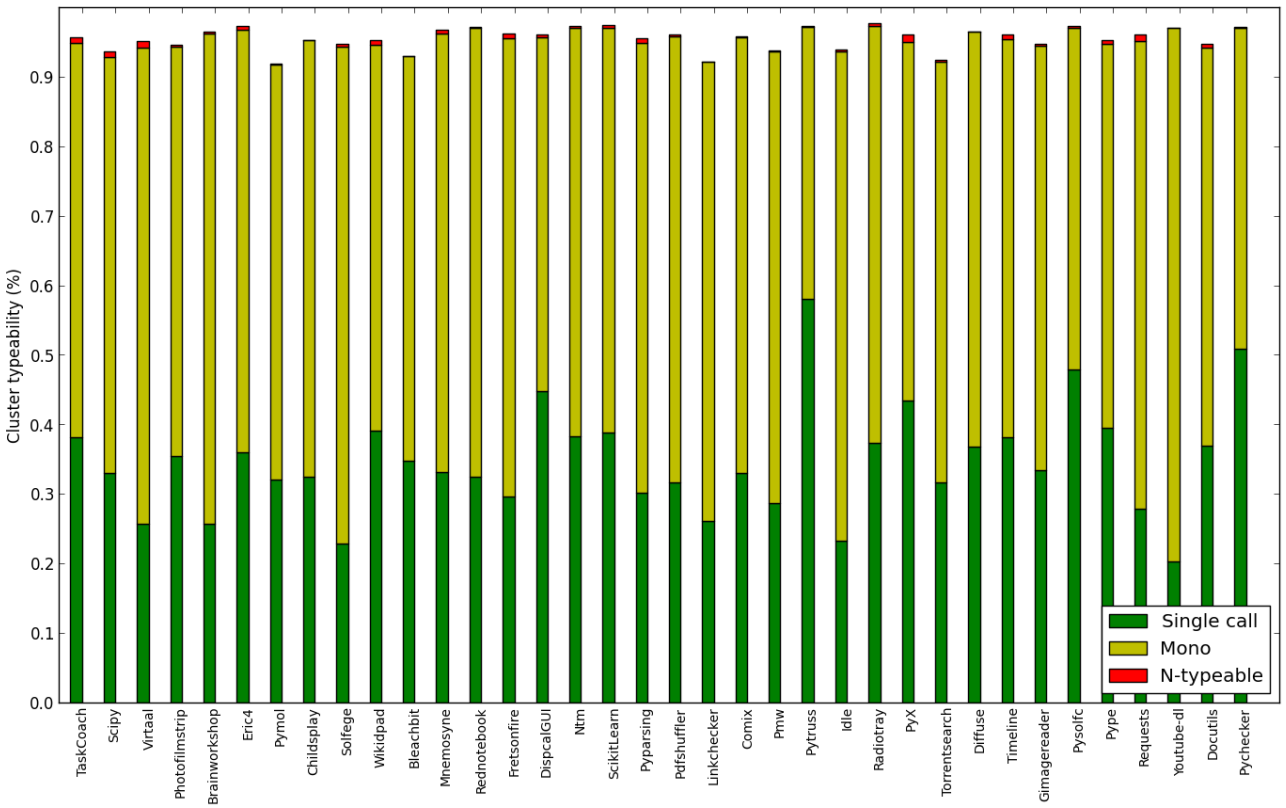

Figure 16. N-Typeable, single call and monomorphic clusters.

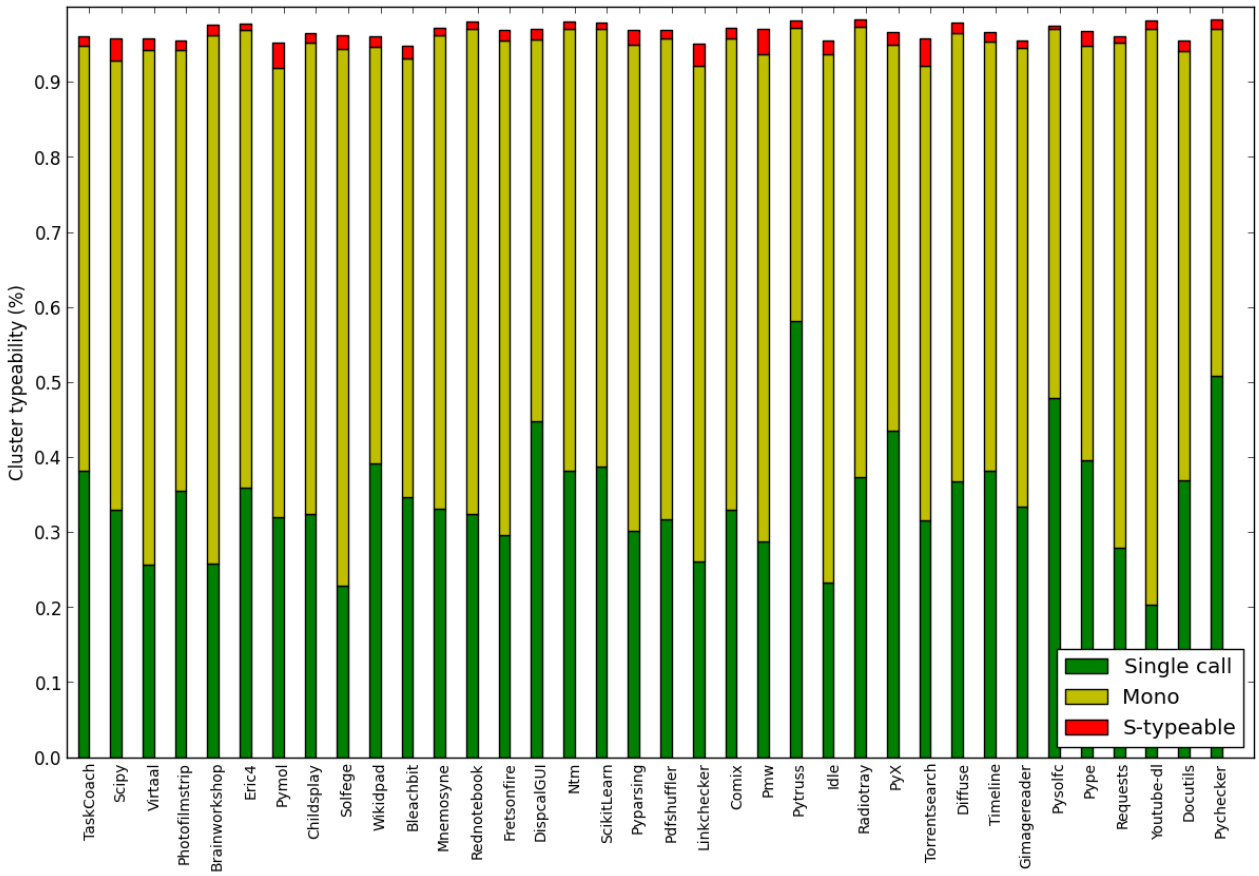

Figure 17. S-Typeable, single call and monomorphic clusters.

dom, which might lead to that the programs are not representative for Python programs in general. Generating representative runs for a particular program requires representative input. Many programs are very large, and since we do not have coverage measurements, we do not know to what extent the source code of the programs was executed. Especially program parts that are used less frequently *e.g.,* parts of programs checking for updates and installing updates have probably not been run. An approximate coverage measurement would be to compare the number of call-sites in the code with the number of call-sites visited by a trace, but it would be very rough so we have not included it.

Our typeability analyses are based on receiver types at call-sites and clusters. Individual call-sites is very fine-grained and clusters may be too coarse-grained. The individual call-site analysis will likely over-estimate typeability since it does not consider connected call-sites. Similarly, cluster analysis will under-estimate typeability by connecting call-sites too greedily (*e.g.,* due to value-based overloading) forcing them to be typed together. We believe that the two approached function as upper and lower bound for (our definitions of) typeability.

Static typing is difficult in the presence of powerful reflective support for non-monotonic changes to classes at run-time. Although we found no use of this in our corpus, Python, for example, allows assigning the `__class__` variable of an object, thereby changing its class. Prior work by ourselves [2] and others [19] investigate the actual usage of such mechanisms in Python and conclude that, although not very common, programs actually contain code of this kind. Our typeability analyses do not consider this, leaving this for future work.

In this paper we set out to understand how structural and nominal types can be used to type untyped Python code with respect to polymorphism. Clearly, an actual implementation of static typing must consider use of reflection. We leave the decision of what strategy to employ (run-time detection, constraining of Python's reflective protocol, etc.) to the designers of such systems.

7. Related Work

The idea to study real programs to understand how languages are used and then use the knowledge for designing better languages and tools is not a new one. In 1971, Knuth studied how Fortran programs were written to make new guidelines for compiler designers [22].

Many efforts have later been presented, for several different dynamic languages, to increase our understanding for how dynamic languages are used in practice. We have seen the study of use of dynamic features by Holkner and Harland [19], where they study the use of *e.g.,* reflection, changes made to objects at runtime (adding/removing attributes, etc.), variables that are used to point out objects of different types, and dynamic code evaluation in 24 Python programs. This work is based on two assumptions; first that Python programs do not generally contain much use of dynamic features and if use of dynamic features can be found it will be easy to rewrite in a more static style, and second that if use of dynamic features is found it will be found during start-up. The first assumption was found to be false and the second to be true. Their study was trace-based and operated at byte-code level, whereas our approach has been to modify the interpreter to produce log files in plain text format. In comparison with the work done by Holkner and Harland our approach has no noticeable impact noticeable impact on the programs' performance and the logs produced are manageable in size. Both performance and log file sizes were problematic for Holkner's & Harland's study. Their goal is similar to what we are aiming to achieve with this study, but do they not consider method polymorphism.

Another study with a similar goal has been made for Smalltalk where Callaú et al [8] first made a static analysis of 1.000 Smalltalk programs to see which dynamic features were used, how frequent the use is and where in the programs the use could be found. They then studied code to understand why the features were used. Their results were that dynamic features are used sparsely although not seldom enough to be disregarded, that the use of dynamic features is more common in some application areas, that the dynamic Smalltalk features that have been included also in statically typed languages like Java are the most popular features, and that use of dynamic features to some extent can be replaced with more statically checkable code. The studies of code revealed that the majority of the use of dynamic features was benefiting from their dynamic nature and would be impossible to replace with more static code, but some use of dynamic features were really a sign of limitations in the language design that programmers solved by using unnecessarily dynamic solutions. These cases could be rewritten without dynamic features but the code would get more complex. Yet other uses of dynamic features could be replaced by less dynamic code. In this study, most of the programs were only studied statically and polymorphism was never considered.

Lebresne et al. [24] and Richards et al. [27] have done a similar study for JavaScript where the interaction with 103 web sites was traced and three common benchmark suites analysed. Common assumptions about JavaScript programs are listed and the goal of the paper is to find support for or invalidate these assumptions. Results from the analysis show that the programs use dynamic features and that the division into an initialisation phase and a division of program runs into different phases is less applicable for JavaScript, since *e.g.,* objects are observed to have properties added or removed long after initialisation. One of the assumptions used as a starting point for the study was that call-site polymorphism would be low. Their result was that 81% of all call-sites were monomorphic, which is less than the 96% we have observed for Python programs (see Table 1). Our study was also

similar to theirs in that they also examined the receivers of the call-sites.

Method polymorphism in particular has been studied in the context of inline caching and polymorphic inline caching in Smalltalk [12] and Self [20]. Polymorphic inline caching in Smalltalk has a reported 95% hit frequency with a size one cache [12], suggesting that Smalltalk call-sites are either relatively monomorphic, or that call-sites are executed often between changes to receiver types. (However unlikely, a Smalltalk program may have a 95% hit frequency while still having 100% megamorphic call-sites.) Our Python-specific result is similar, and has a stronger bearing on the typeability of whole programs as we consider polymorphism in clusters of call-sites. In Hölzle's et al. work on Self [20], the number of receiver types in a call-site are usually lower than 10. In our study, 88% of all call-sites have 5 or fewer receiver types.

Other related work include initiatives to build type or type inference systems for dynamic languages starting in functional languages based on the work of Hindley, Milner and Damas [13]. The use of inferred types has been successfully implemented in languages like ML and Haskell.

Type inference for object-oriented languages, on the other hand, turned out to be more complex and computationally demanding as in Suzuki's case where the inference algorithm [34] failed because of the Smalltalk environment's restriction on the number of live objects. His work was followed up more successfully by others both in Smalltalk [6, 16, 26] and other object-oriented languages. Type inference for Smalltalk was mainly motivated by increased reliability although readability and performance often also are mentioned as other expected improvements.

Following Smalltalk, Diamondback Ruby [15] focuses on finding and isolating errors by combining inferred static types with type annotations made by the programmer where annotated code is excluded from type inference and its types will be checked at runtime. The type system was tested on a set of benchmark programs of 29–1,030 LOC and proved to be useful for finding errors.

Type inference systems implemented for Python have often focused on improving performance rather than program quality aspects as reliability or readability.

Aycock's aggressive type inference [5] was designed as a first step towards translating Python code to Perl. The aggressiveness is expressed in that the program has to adhere to the restriction rules for how Python programs may be written for the type inference to work. *e.g.,* runtime code generation is not allowed, and types for local variables must not change during execution.

Following this, also targeting performance but without restrictions for the language, Cannon first [9] thoroughly discusses difficulties met when implementing type inference for Python and then presents a system for inferring types in a local name-space. Tests show that performance improvements were around 1%.

Recently, the work on type inference for Python has been dominated by the PyPy initiative, originally aiming to implement Python in Python. PyPy uses type inference in RPython, the restricted version of the language that is used to implement the interpreter [4].

Types inferred for object-oriented languages are often nominal [4, 9, 26] but there are other solutions, like Strongtalk [6] that infers structural types and DRuby where class (nominal) types are combined with object (structural) types.

8. Conclusions

Our results show that while Python's dynamic typing allows unbounded polymorphism, Python programs are predominantly monomorphic, that is, variables only hold values of a single type. This is true for program start-up and normal runtime, in library code and in program-specific code.

Nevertheless, most programs have a few places which are megamorphic, meaning that variables in those places contain values of many different types at different times or in different contexts. Smaller programs do not generally differ from larger programs in this.

Because of the high degree of monomorphism, most programs can be typed to a large extent using a very simple type systems. Our findings show that the receiver in 97.4% of all call-sites in the average program can be described by a single static type using a conservative nominal type system using single inheritance. If we add parametric polymorphism to the type system, we increase the typeability to 97.9% of all call-sites for the average program.

For clusters, the receiver objects are typeable using a conservative nominal type system using single inheritance to 95.6% (on average). If we instead use a structural type, the typeability increases somewhat to 96.7% (on average).

Most polymorphic and megamorphic parts of programs are *not typeable by nominal or structural systems*, for example due to use of value-based overloading. Structural typing is only slightly better than nominal typing at handling non-monomorphic program parts. This suggests that nominal and structural typing is not a deciding factor in type system design if typing polymorphic code is desirable. More powerful constructs are needed in these cases, such as refinement types. We will investigate this in future research.

5. REFERENCES

[1] O. Agesen, "The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism", Proc. ECOOP'95, pp 2–26, 1995.

[2] B. Åkerblom, J. Stendahl, M. Tumlin and T. Wrigstad, "Tracing Dynamic Features in Python Programs", Proc. MSR'14, 2014.

[3] J.D. An, A. Chaudhuri, J.S. Foster, and M. Hicks, "Dynamic inference of static types for Ruby", In POPL'11, pp 459–472, 2011.

[4] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. “RPython: Reconciling Dynamically and Statically Typed OO Languages”, In DLS’07, 2007.

[5] J. Aycock, “Aggressive Type Inference”, pp 11–20, Proc. of the 8th International Python Conference, 2000.

[6] G. Bracha and D. Griswold, “Strongtalk: Typechecking Smalltalk in a Production Environment”, In Proc. OOPSLA’93, pp. 215–230, 1993.

[7] F. Brito e Abreu, “The MOOD Metrics Set,” Proc. ECOOP’95 Workshop on Metrics, 1995.

[8] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, “How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk”, In MSR’11, 2011.

[9] B. Cannon, “Localized Type Inference of Atomic Types in Python”, Master Thesis, California Polytechnic State University, San Luis Obispo, 2005.

[10] L. Cardelli, and P. Wegner, “On understanding types, data abstraction, and polymorphism”, ACM Computing Surveys Volume 17, pp 471-222, 1985.

[11] R. Chugh, D. Herman, and R. Jhala, “Dependent Types for JavaScript”, In SIGPLAN Not., Vol 47, No. 10 Oct 2012.

[12] L. P. Deutsch and A. M. Schiffman, “Efficient Implementation of the Smalltalk-80 System”, In POPL 1984.

[13] L. Damas and R. Milner, “Principal Type-schemes for Functional Programs”, In POPL’82, pp. 207–212, 1982.

[14] Facebook, Inc., “Specification for Hack”, Facebook, Inc., `https://github.com/hhvm/hack-langspec`, 2015.

[15] M. Furr, J.D. An, J.S. Foster, and M. Hicks, “Static type inference for Ruby”, In the 2009 ACM Symposium on Applied Computing, 2009.

[16] J. O. Graver and R. E. Johnson, “A Type System for Smalltalk”, In Proc. POPL’90, pp. 136–150, 1990.

[17] B. Hackett and S. Guo, “Fast and Precise Hybrid Type Inference for JavaScript”, In PLDI’12, pp. 239–250, 2012.

[18] P. Heidegger and P. Thiemann, “Recency types for analyzing scripting languages”, In ECOOP’10, pp. 200–224, 2010.

[19] A. Holkner and J. Harland, “Evaluating the Dynamic Behaviour of Python Applications”, In ACSC’09, pp. 19-28, 2009.

[20] U. Hölzle and C. Chambers and D. Ungar, “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches”, In ECOOP ’91, LNCS 512, July, 1991.

[21] S. Karabuk and F.H. Grant, “A common medium for programming operations-research models”. IEEE Software, 24(5):39-47, 2007.

[22] D. E. Knuth: “An Empirical Study of FORTRAN Programs. Software”, Practice and Experience, 1(2): 105-133, 1971.

[23] A. Lamaison, “Inferring Useful Static Types for Duck Typed Languages”, Ph.D. Thesis, Imperial College, London, U.K., 2012.

[24] S.Lebresne, G.Richards, J.Östlund, T.Wrigstad, J.Vitek, “Understanding the dynamics of JavaScript”, In Script to Program Evolution, 2009.

[25] J. McCauley. “About POX”. 2013. `http://www.noxrepo.org/pox/about-pox/`

[26] J. Palsberg and M. I. Schwartzbach, “Object-Oriented Type Inference”, In OOPSLA’91, pp. 146–161, 1991.

[27] G. Richards, S. Lebresne, B. Burg and J. Vitek, “An Analysis of the Dynamic Behavior of JavaScript Programs” , In PLDI’10, 2010.

[28] G. van Rossum and F.L. Drake, “PYTHON 2.6 Reference Manual”, CreateSpace, Paramount, CA, 2009.

[29] M. Salib, “Faster than C: Static type inference with Starkiller”, In PyCon Proceedings, Washington DC, 2004.

[30] Securities and Exchange Commission. Release Nos. 33-9117; 34-61858; File No. S7-08-10. 2010. http://www.sec.gov/rules/proposed/2010/33-9117.pdf

[31] J. Siek, and W. Taha, “Gradual Typing for Objects”, in Proc. ECOOP’07, Berlin, Germany, July 2007.

[32] SourceForge, `http://sourceforge.net/`.

[33] C. Strachey, “Fundamental Concepts in Programming Languages”, in Higher Order Symbol. Comput., pp. 11–49, 2000.

[34] N. Suzuki, “Inferring types in smalltalk”, in Proceedings POPL’81, pp. 187–199, New York, USA, 1981.

[35] P. Thiemann, “Towards a type system for analyzing JavaScript programs”, in Proceedings of ESOP’05, pp. 408–422, 2005.

[36] S. Tobin-Hochstadt and M. Felleisen, “Interlanguage migration: from scripts to programs”, in DLS’06, 2006.

[37] L. Tratt, “Dynamically Typed Languages”, Advances in Computers 77. Vol, pp. 149-184, ed. Marvin V. Zelkowitz, 2009.