Stockholm University

# How do Python Programmers Use Python?

## Python Dynamicity & Other Ideas

Beatrice Åkerblom

Department of Computer and Systems Sciences
Stockholm University
beatrice@dsv.su.se

| | Statically typed languages | Dynamically typed languages |
|---|---|---|
| Dynamic proponent’s view | <br>Lawful evil | <br>Chaotic good |
| Static proponent’s view | <br>Lawful good | <br>Chaotic evil |

# “Historically” in Language Research

- Type inference (Smalltalk, Various Python projects, Diamondback Ruby)
- Gradual typing (e.g. Siek, Taha)
- Soft typing (e.g. Fagan)
- Pluggable types (e.g. Bracha)

- Generally tries to make dynamic languages more “controllable” and predictable, that is static
- Assumptions are made about how programs are developed

# Approaches used before

Selected examples:

- "Usually, no further properties are defined after the initialization and the type of the properties rarely changes."
  -- Peter Thiemann
- "Giving people a dynamically-typed language does not mean that they write dynamically-typed programs"
  -- John Aycock
- "Yet while the presence of such abundant dynamism makes traditional static optimization impossible, in most programs, there is surprisingly little dynamism present."
  -- Michael Salib

# Approaches used before

Selected examples:

- “Usually, no further properties are defined after the initialization and the type of the properties rarely changes.”
-- Peter Thiemann
- “Giving people a dynamically-ty language does not mean that th dynamically-typed programs”
-- John Aycock
- “Yet while the presence of such dynamism makes traditional static optimization impossible, in most programs, there is surprisingly little dynamism present.”
-- Michael Salib

True?

We don’t know

# When/Where, How & Why (if at all) is the dynamic power of dynamic languages used in real applications?

# What’s Dynamic?

- Dynamic features - use of introspection, reflection, dynamic code evaluation

- Duck typing  - polymorphism without need for inheritance or declared interfaces

- Dynamic objects - how dynamic are class and object structures

# What's Dynamic?

- Dynamic features - use of introspection, reflection, dynamic code evaluation
- Duck typing - polymorphism without need for inheritance or declared interfaces
- Dynamic objects - how dynamic are class and object structures

what is the program?
do our objects reflect the class definitions?
how dynamic are variable accesses, etc?
how common is dynamic code generation?

# What’s Dynamic?

- Dynamic features - use of introspection, reflection, dynamic code evaluation
- Duck typing  - polymorphism without need for inheritance or declared interfaces
- Dynamic objects - how dynamic are class and object structures

do variables change type?
will different paths lead to different types?
how polymorphic are method calls?
can common supertypes be found?

# What’s Dynamic?

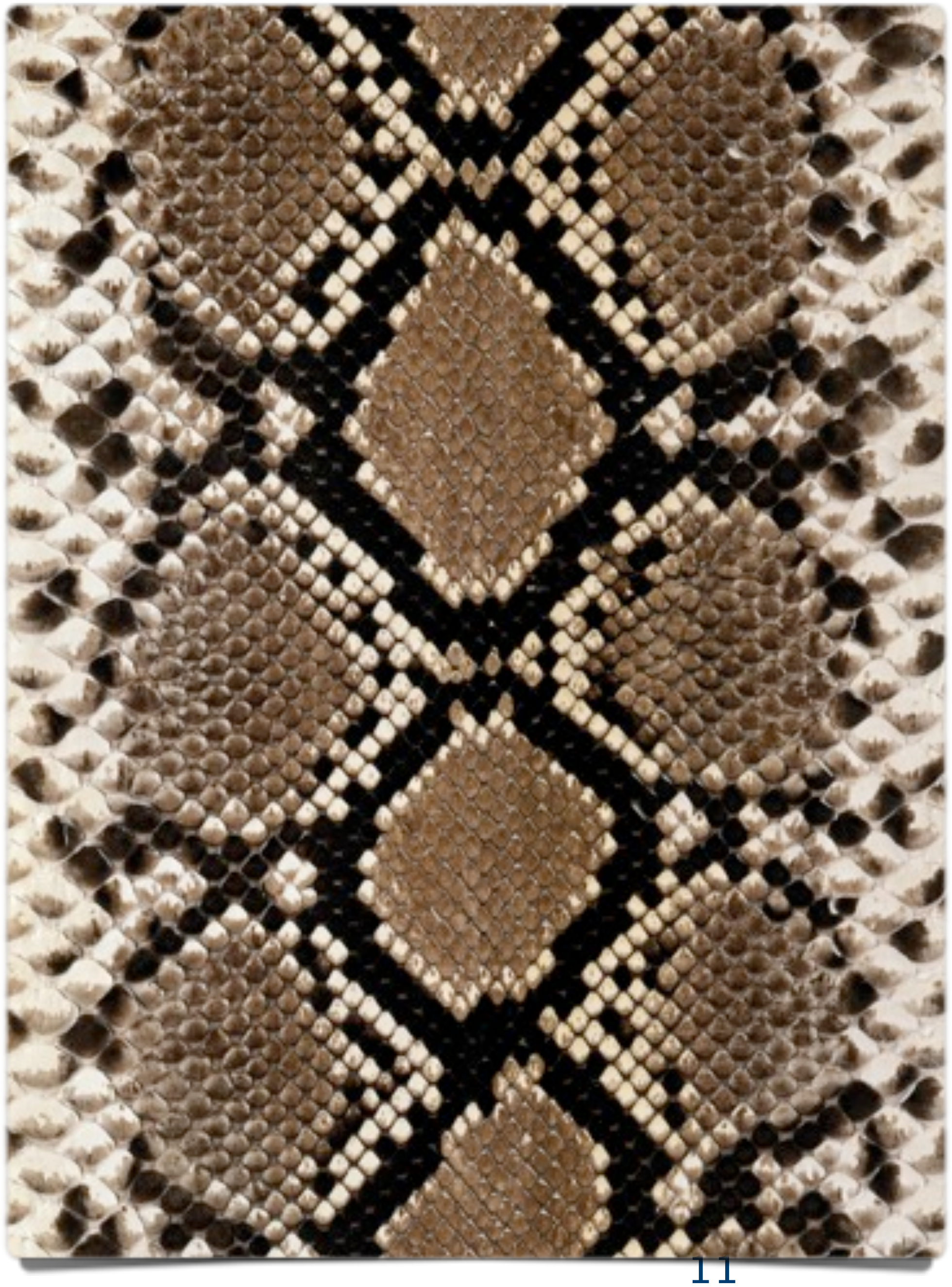
- Dynamic features - use of introspection, reflection, dynamic code evaluation
- Duck typing  - polymorphism without need for inheritance or declared interfaces
- Dynamic objects - how dynamic are class and object structures

how stable is the OO (objects, classes, inheritance structures) of Python programs?
do we find interface-like structures in Python programs?

# Why is this important?

We’ll be able to:

- know how much of a “typical” Python program could (or could not) be annotated with types
- know how well Python source code does represent the running program
- know to what extent we need to support dynamic behaviour e.g. when building tools or new language constructs for Python
- emphasize the focus on how Python is used when designing new constructs



# Different Sources, different methods

- Programs (Quantitative)
  - Static analysis (what is the program?)
  - Dynamic analysis: Measure behaviour at runtime, e.g. use of language constructs, inheritance hierarchies, polymorphic call sites, etc.
- Code snippets (Qualitative)
  - Search for language constructs usage patterns
  - Read to understand how/why
- Programmers (Sociological)
  - Interview
  - Observe

# What Have We Done?

- Modified the Python 2.6 interpreter to log information about running programs
  - class creation
  - method and function calls
  - instance member access
  - use of dynamic features
- Python programs selected from Source Forge
- Programs run on a Debian machine
  - interactive
  - tests
  - examples
- Program runs documented
  - tests
  - recordings
  - use cases

# Dynamic Features in Python Programs

- Anomos, Bleachbit, Comix, ConvertAll, Exaile, Kodos, Mcomix, Pysolfc, Rednotebook, Retext, Sbackup, Solfege, Task coach, Torrent Search, Wikidpad, Zmail

- hasattr, eval, reload, getattr, __delattr__, __getattr__, execfile, __getattribute__, del attribute, __import__, exec, setattr, vars, __setattr__, delattr

0: Id-nummer

1: the path, filename and row number from which the call was made,

2: Caller id.

3: Caller type.

4: Target Id

5: Target type

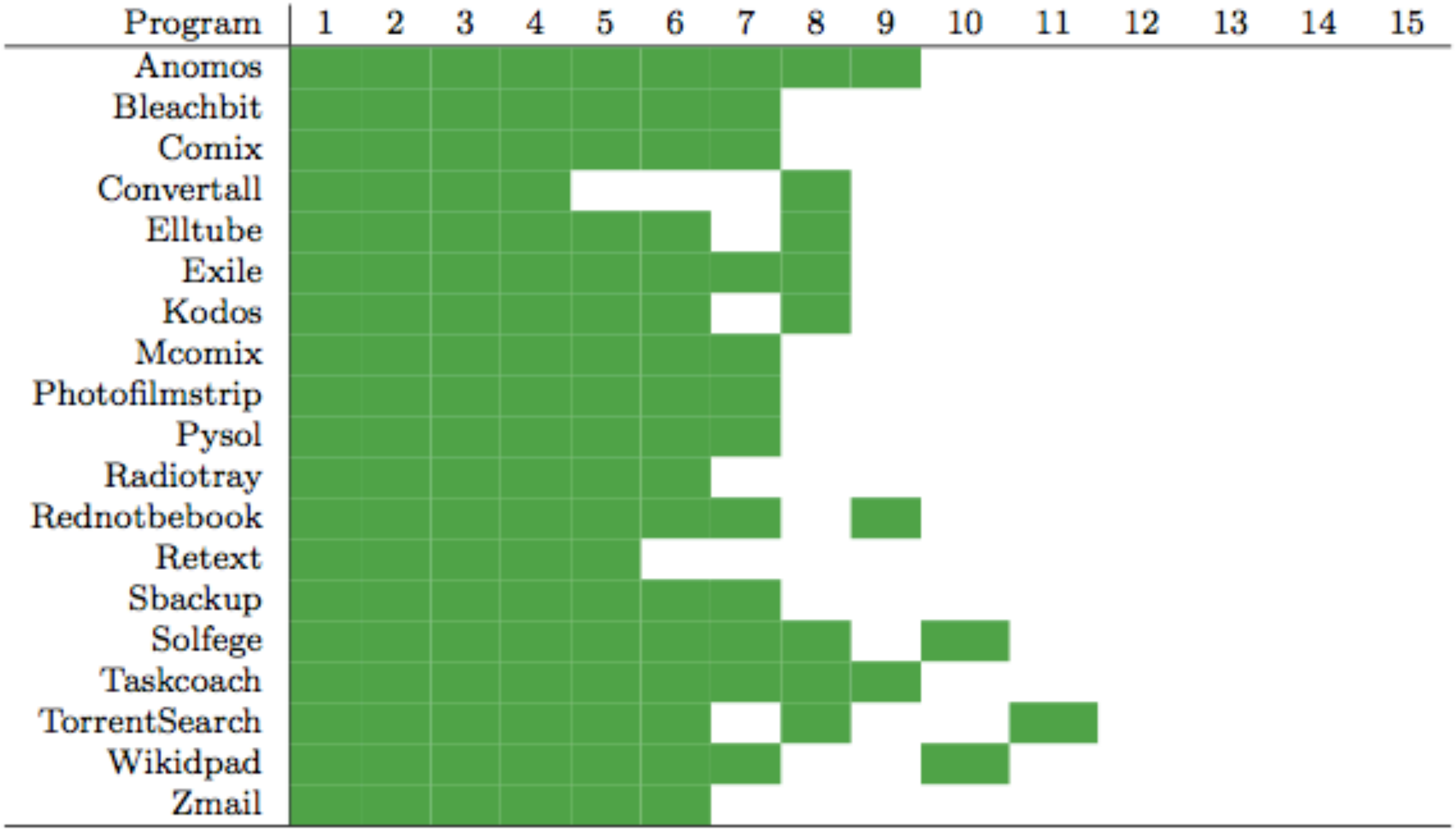
6: Feature name

7: Argument types

8: Results

# What about Holkner & Harland’s “Evaluating the dynamic behaviour of Python applications”?

# Number of Features Used by Programs



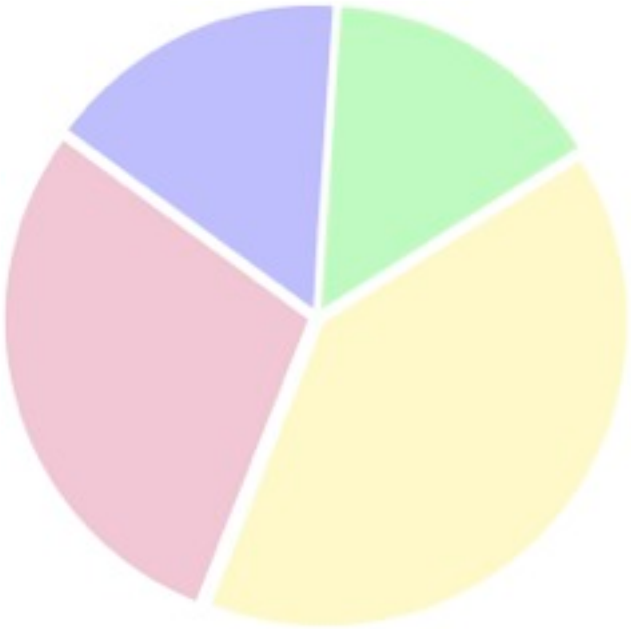
| Program | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Anomos | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |
| Bleachbit | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Comix | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Convertall | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  | 🟩 |  |  |  |  |  |  |  |
| Elltube | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  | 🟩 |  |  |  |  |  |  |  |
| Exile | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |
| Kodos | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  | 🟩 |  |  |  |  |  |  |  |
| Mcomix | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Photofilmstrip | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Pysol | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Radiotray | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |  |
| Rednotbebook | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  | 🟩 |  |  |  |  |  |  |
| Retext | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |  |  |
| Sbackup | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |
| Solfege | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  | 🟩 |  |  |  |  |  |
| Taskcoach | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |
| TorrentSearch | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  | 🟩 |  |  | 🟩 |  |  |  |  |
| Wikidpad | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  | 🟩 |  |  |  |  |  |
| Zmail | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |  |  |  |  |  |  |  |  |  |

1 = hasattr, 2 = getattr, 3 = _import_, 4 = setattr, 5 = exec, 6= _getattr_, 7 = del attribute, 8 = eval, 9 = vars, 10 = _setattr_, 11 = delattr, 12 = _delattr_, 13 = _getattribute_, 14 = execfile, 15 = reload

Stockholm University

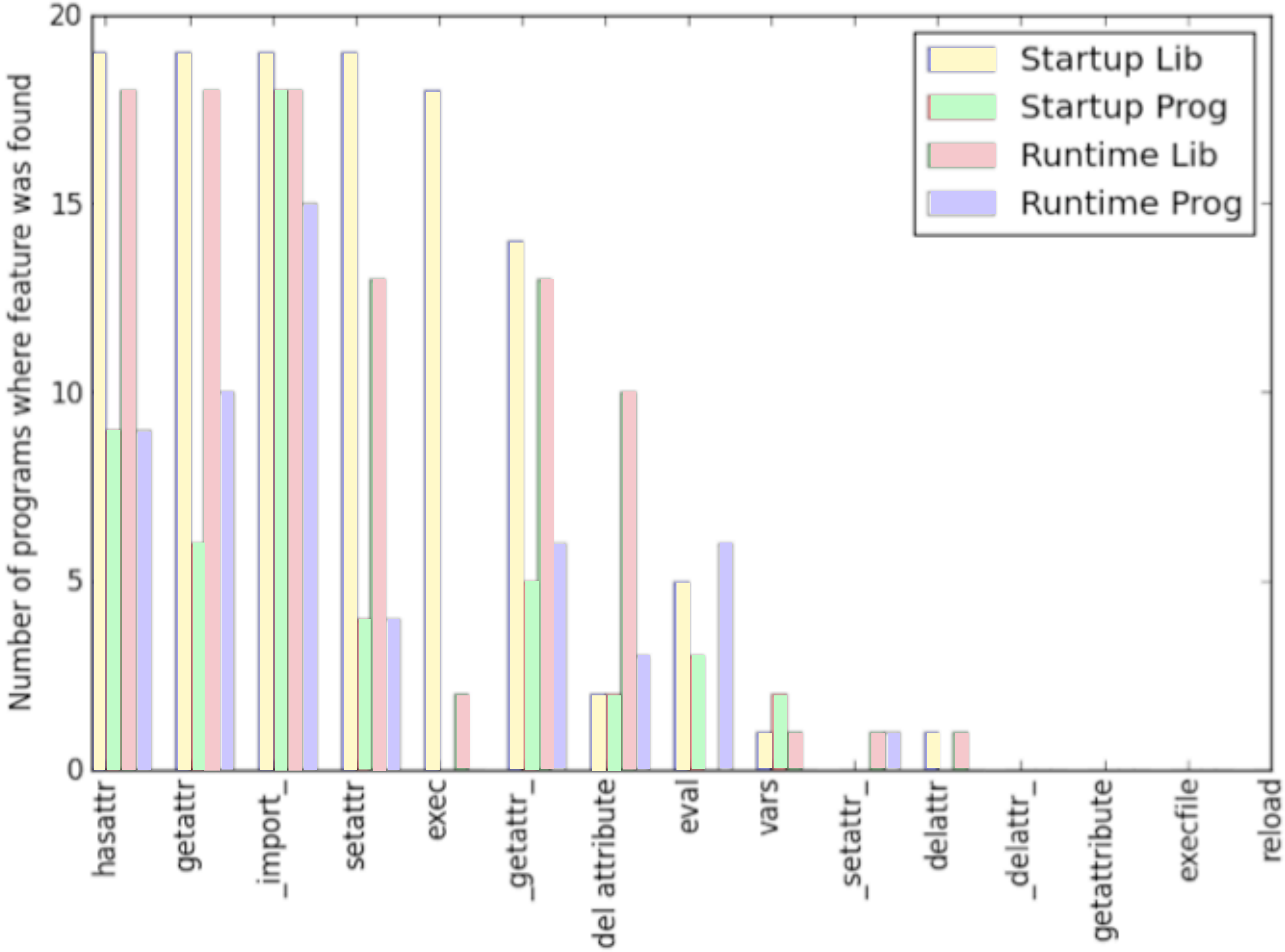
# Distribution of Dynamism for All Traces

| Run-time | Startup |
|---|---|
| Program 16% | Program 15% |
| Library 29% | Library 40% |

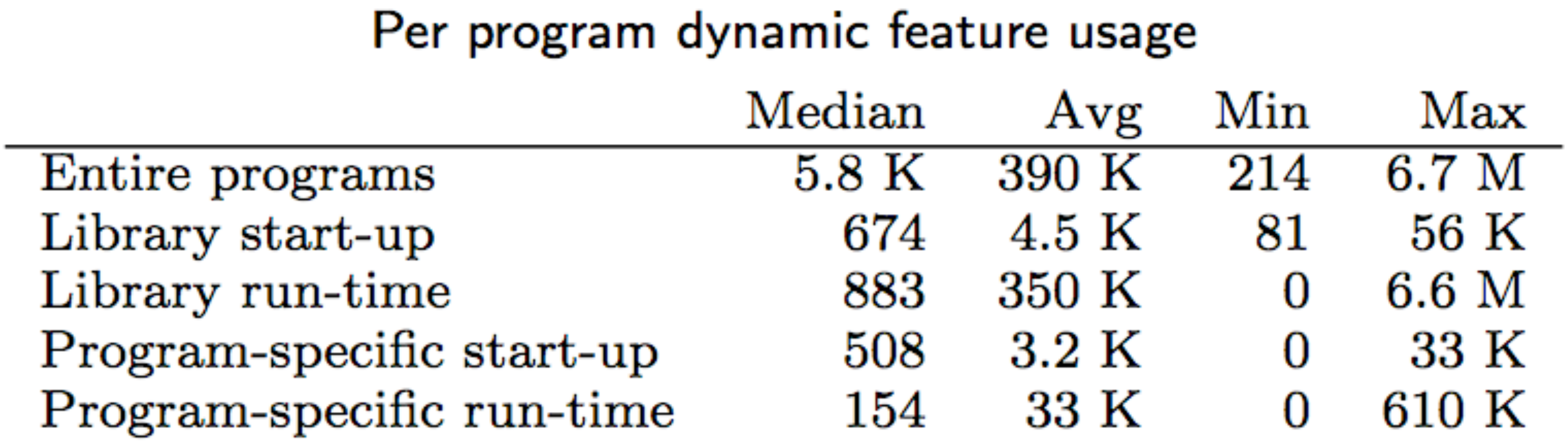


- Distribution of dynamic features over libraries and program-specific code during start-up and run-time

# Number of Programs Where Features Were Traced



# Median, Average, Minimum and Maximum for All Features



Per program dynamic feature usage

| | Median | Avg | Min | Max |
|---|---|---|---|---|
| Entire programs | 5.8 K | 390 K | 214 | 6.7 M |
| Library start-up | 674 | 4.5 K | 81 | 56 K |
| Library run-time | 883 | 350 K | 0 | 6.6 M |
| Program-specific start-up | 508 | 3.2 K | 0 | 33 K |
| Program-specific run-time | 154 | 33 K | 0 | 610 K |

# Polymorphism in Python Programs

- Task Coach, SciPy, Pootle, Virtaal & the Translate Toolkit, PhotoFilmStrip, Brain Workshop, Eric4, PyMol, Childsplay, GNU Solfege, WikidPad, BleachBit, Mnemosyne, RedNotebook, DispcalGUI, Scikit Learn, Python parsing module, PDF-Shuffler, Link checker, Mcomix, Python megawidgets, Autocomplete for Notepad++, PyTruss, Idle, Radiotray, PyX, TorrentSearch, Diffuse, Timeline, GImageReader, PySolFC, PyPe, Requests, Youtube-dl, Docutils, Pychecker

0. Event ID
1. Source file path
2. Caller ID (current this at the call-site)
3. Caller type
4. Target ID (the receiver of the method call)
5. Class name of target + : + class id
6. Name of called function/method
7. Argument types
8. Call line
9. A list of all super classes of the target type

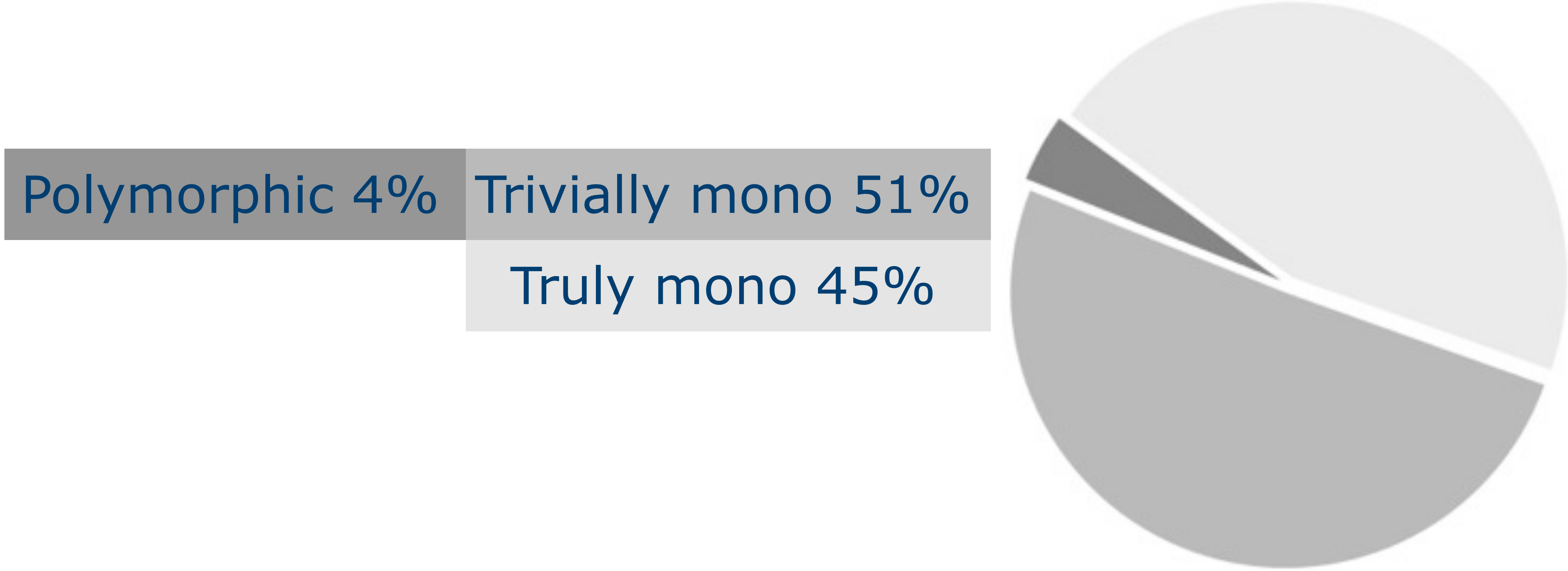
# Questions Asked

- How many unique call-sites?
- How many call-sites are monomorphic?
  - Trivially monomorphic vs. monomorphic
- How many polymorphic call-sites?
- Distribution of the degree of polymorphism seen

- For call-sites that saw several different types as receiver, what were the types and do they share a common supertype containing the method called?

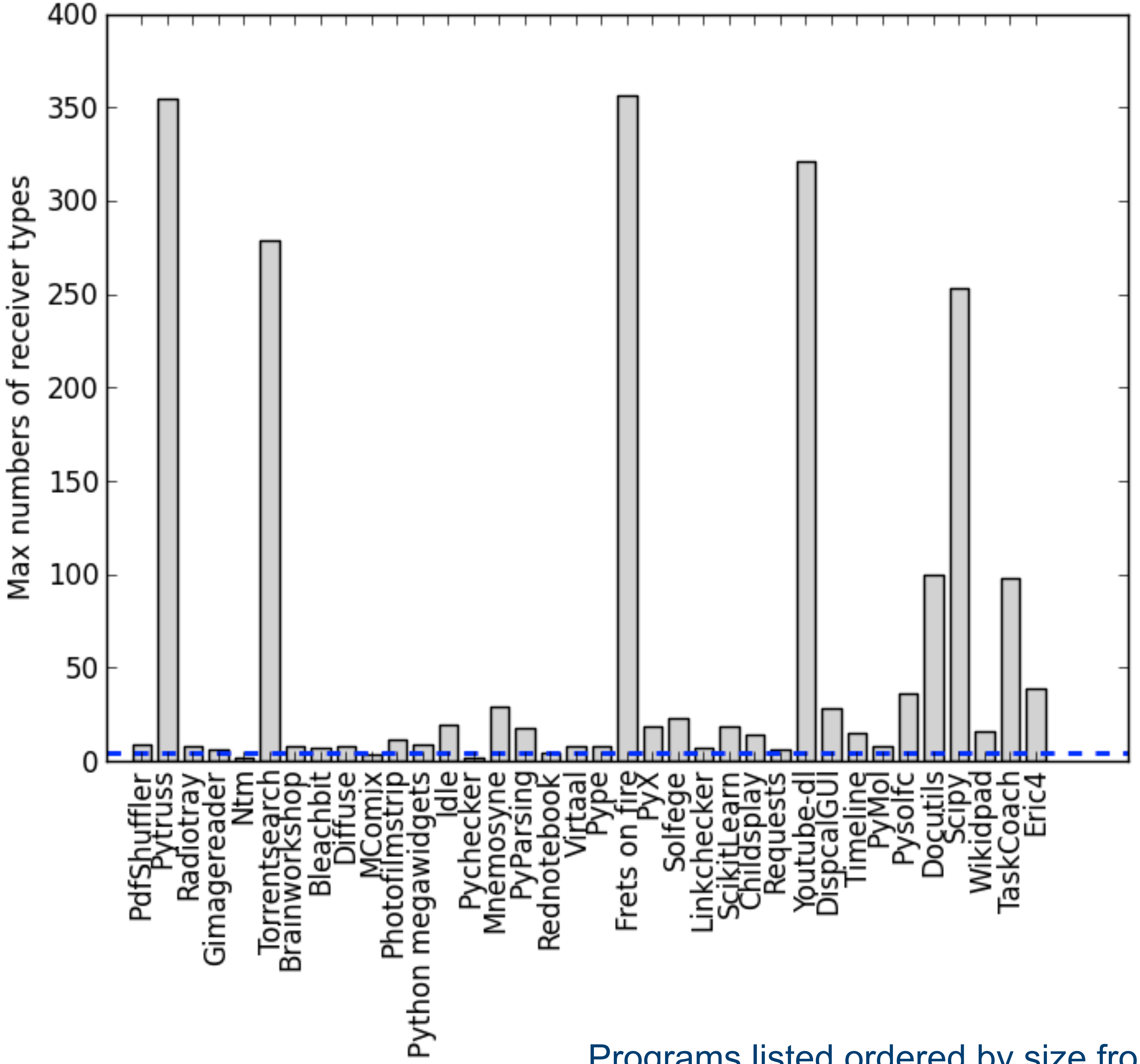
# Monomorphic Call Sites

- Trivially monomorphic: We have only recorded one single execution of this call site
- Monomorphic: We have recorded more than one execution of this call site, and the types seen were always the same



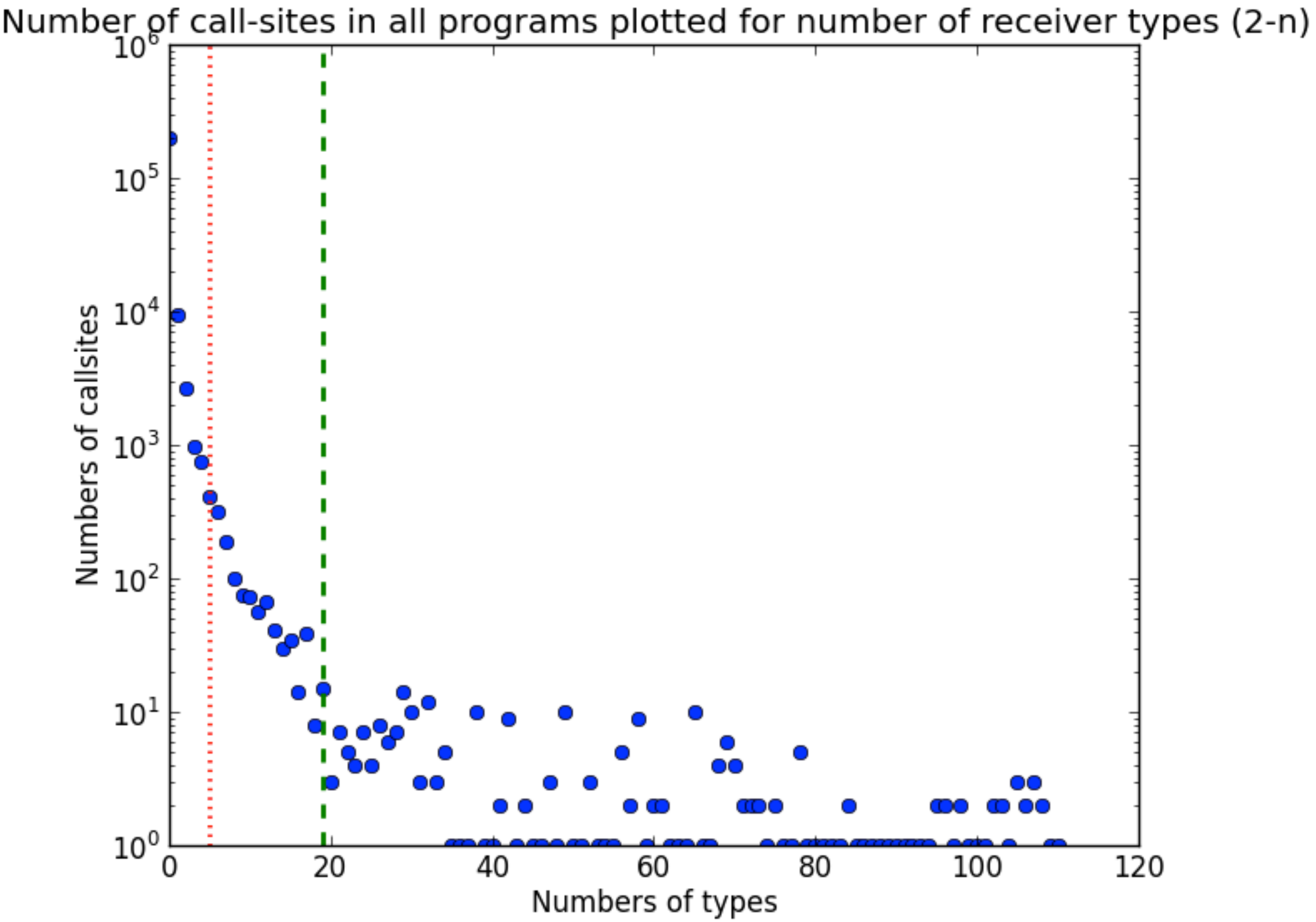
# How Polymorphic are Python Call Sites?

Stockholm University



Programs listed ordered by size from smallest to largest

# How Polymorphic are Python Call Sites?



# References

- Phillip Heidegger and Peter Thiemann, “Recency types for analyzing scripting languages”, ECOOP 2010.
- John Aycock, “Aggressive Type Inference”, In Proceedings of the 8th International Python Conference, 2000.
- Michael Salib, “Faster than C: Static type inference with Starkiller”, In PyCon Proceedings, 2004.
- A. Holkner and J. Harland, “Evaluating the Dynamic Behaviour of Python Applications”, Proceedings of ACSC ’09, 2009.
- Beatrice Åkerblom and Tobias Wrigstad, “Tracing Dynamic Features in Python Programs”, Proceedings the 11th Working Conference on Mining Software Repositories, 2014.