Toward a Corpus Study of the Dynamic Gradual Type

DIBRI NSOFOR, University of Utah, USA

BEN GREENMAN, University of Utah, USA

Gradually-typed languages feature a dynamic type that supports implicit coercions, greatly weakening the type system but making types easier to adopt. Understanding how developers use this dynamic type is a critical question for the design of useful and usable type systems. This paper reports on an in-progress corpus study of the dynamic type in Python, targeting 221 GitHub projects that use the mypy type checker. The study reveals eight patterns-of-use for the dynamic type, which have implications for future refinements of the mypy type system and for tool support to encourage precise type annotations.

1 INTRODUCTION

Programming languages have traditionally forced developers to choose between the "flexibility and simplicity" of untyped code [32] and the maintenance benefits of static types (e.g. [31]). Gradual typing offers a compromise through a flexible *dynamic type* that can be used in any context without a type error [60, 61]. For example, a function that expects an argument of the dynamic type (named Any in Python) can treat the argument as though it has any precise type; the code below assumes its dynamic argument is a class, adds a method to it, and returns the updated class:

```python
import types; from typing import Any # the dynamic type in Python

def addPrice(cls: Any) -> Any: # mixin, add method to class
    cls.price = types.MethodType(lambda self: 99, cls)
    return cls
```

A caller can send any value to the addPrice function without raising a compile time type error. Incorrect calls, such as addPrice(11) result in a runtime error when executed.

The dynamic type adds a degree of optimism to the typechecker, fundamentally changing the nature of type analysis. Without the dynamic type, a well-typed program is certifiably made of parts that fit together (it cannot "go wrong" [46]). With the dynamic type, a well-typed program is code that *may* fit together provided that every occurrence of the dynamic type receives well-behaved values at runtime—which, as noted in work on static blame [63], may be impossible. Moreover, the dynamic type renders type analysis unsound (unless coercions are backed by runtime checks, see section 2): any value can inhabit any type after coercion. The dynamic type also

Authors' addresses: Dibri Nsofor, Kahlert School of Computing, University of Utah, Salt Lake City, Utah, 84112, USA, dibrinsofor@gmail.com; Ben Greenman, Kahlert School of Computing, University of Utah, Salt Lake City, Utah, 84112, USA, benjamin.l.greenman@gmail.com.

pokes loopholes into other traditional properties. For example, the simply-typed lambda calculus is strongly normalizing, but loses this property with the addition of a dynamic type [45].

Given the perils of the dynamic type, it ought to be used sparingly: either as a temporary measure in prototype software, or as a last resort when precise types do not exist. And yet, its use is widespread. In our collection of 221 projects that use the mypy typechecker (section 3), there are 28,478 occurrences of the dynamic Any type. Understanding why these Anys appear in widely-used code is a critical question for the adoption of types. If more-precise alternatives exist, that motivates tool support for writing types. If the Any types cover up limitations of the typechecker, that motivates research on type system design. If the rationale for certain Any types is unclear, that motivates interviews and developer surveys.

This paper is a progress report of our journey to discover the precise reason behind Any types across a large corpus of Python projects. Based on a combination of manual and script-driven analysis on a small number of sample projects, it presents:

- the design of a corpus study to study use of the dynamic type (section 3),
- eight usage patterns (section 4) and discussions of how to identify the patterns automatically.

A major goal of this paper is to solicit early-stage feedback on the design of our corpus study experiment. Automated software analysis is a powerful but dangerous tool, as flaws in the protocol can lead to questionable conclusions [6, 7, 54, 55]. In particular, our methods for detecting usage patterns must guard against false positives before we apply them at scale. The paper concludes with future work (section 5), related work (section 6), and a discussion that positions this study in the broader context of gradual languages (section 7).

2 BACKGROUND: OPTIONAL TYPING AND MYPY

Gradually-typed languages come in many varieties because the question of how to enforce static types in the midst of untyped code exposes a complex design space. Full enforcement via behavioral contracts is a compelling vision [21, 64], but leads to high run-time overhead [29] without custom runtime support [3, 20, 39]. One alternative is to enforce only the top-level shape of types, but this can still lead to overheads of 2x or more [25, 28, 72]. A second alternative is to forbid untyped data structures from entering typed code [43, 48, 75]. A third alternative is to forgo enforcement (and type soundness) altogether. This third option is the most popular choice [26], and is known as optional typing [10].

Mypy is an optional typing system for Python [51]. It equips the PEP 484 syntax for types [69] with static checks that detect inconsistencies between types and code. Mypy can efficiently analyze millions of lines of code [40] and has been widely adopted. Mypy supports annotations on variable, class, and function declarations. Developers can annotate as many or as few of these positions as they choose. By default, unannotated positions receive the dynamic Any type, though developers can change this default through configuration options that fine-tune mypy or by writing a comment of the form #type:ignore[ERR] with ERR replaced by a specific mypy error code [50]. For programs that depend on untyped libraries, developers can write interfaces, called *type stubs*, that declare types for library exports. The (untyped) definitions of these exports are not typechecked, but their uses (in typed code) are typechecked.

In summary, there are several ways that the Any type can enter a mypy codebase: explicitly through developer-provided Any annotations, implicitly through configuration options, and explicitly through lines with #type:ignore comments. All three sources are important for a corpus study. But whereas configuration options and comments come with metadata to describe how they systematically incorporate an Any type, developer-provided annotations require manual analysis. The main focus of this paper is on the manual analysis.

Fig. 1. Project overview with a marker at the current milestone: Study Patterns

3 CORPUS STUDY DESIGN

Our goal is to learn why developers resort to the dynamic type by analyzing thousands of open-source projects. Currently, we are in the midst of an initial, formative stage of the project (herein called the pre-corpus study), in which we study instances of the dynamic type by hand. As patterns emerge, the next step is to build automated tools and measure how common each pattern is across our corpus of approximately 79,000 projects. Figure 1 sketches this overall pipeline and highlights the current focus. This section describes our protocols for building a corpus (section 3.1), filtering uninteresting instances of the Any type (section 3.2), and manually finding patterns (section 3.3).

3.1 Building the Corpus

We built a corpus by querying GitHub via Mozilla's `agithub` utility [47] and filtering the results. The query asked for Python projects with a modest number of GitHub stars (80 stars) and at least one of the following configuration files, which may contain mypy options: `mypy.ini`, `pyproject.toml`, `setup.cfg`, and `config`. This query resulted in over 79K projects. We chose a sample of 221 projects that had at least 250 GitHub stars and that actually used mypy (confirmed manually) as a target for the pre-corpus study.

3.2 Filtering Any Types

Mypy comes with a utility called `stubgen` that extracts the type annotations from a Python file into a type stub interface (section 2). We use these stubs as the first step toward manual analysis. The stubs from our 221 sample projects contain 318,564 lines of annotations that include the Any type.

Many of the annotations are of low priority for building formative hypotheses. First, method annotations whose only Any is in the first argument position are common and often uninteresting; these come about because mypy infers Any when a method argument lacks an annotation. Second, a file may contain several duplicate signatures. After filtering both sorts of annotations, we arrived at 41,447 distinct lines in which to search for patterns.

The majority of annotations in project files are for functions or methods. We further classified these arrow types into three categories using a script: methods with Any for their first parameter (i.e., `self` parameter), arrows with Any inside a callback argument (`Callable[T1,T2]`), and arrows with Any inside a dictionary argument (`Dict[T1,T2]` or `Mapping[T1,T2]`). These categories helped to guide our manual analysis.

3.3 Manually Studying Patterns

Both authors of this paper manually examined type annotations and code to identify patterns of use for the Any type. One author studied every annotation in 10 large projects for a total of over

5K annotations. The other author took an ad-hoc sample of 30 projects (spread throughout the alphabet, from BentoML to wandb) and sampled types in those projects that used Any inside of a Callable or Dict type, inspecting a total of 92 annotations. Whenever the authors found an Any type that seemed likely to reveal typing challenges, they searched for its origin in the source code to learn more.

4 PATTERNS OF DYNAMIC TYPE USE

Our manual study of types and code in a few dozen sample projects has revealed eight ways in which developers appear to use the Any type in mypy (implicitly and explicitly). These eight categories range from specific anti-patterns to vague cases where precise types exist but the code uses Any nevertheless. Below, we present each pattern with a code example and discuss how source-code analysis might identify the pattern at scale. For some patterns, we have implemented a related analysis; where applicable, we report data based on our 221 sample projects.

4.1 Dynamic Instead of a Type Variable

Every occurrence of the Any type is a wildcard. By contrast, type variables can introduce a degree of uniformity. Consider this equality function from our corpus:

```python
def eq(a: Any, b: Any) -> bool:
  return a == b
```

Every call of the form eq(x, y) is type correct, even though the function cannot return True when its arguments have different types. If the type signature used a variable in place of the two Anys, developers would know more about the function's behavior (even though mypy cannot easily report a type error, see section 4.2). If the intent is indeed to allow type-mismatched inputs, the type could say so with distinct variables for its two parameters.

We do not have an automatic analysis for this pattern. It is unclear what to look for in a type signature: although type variables would help this eq function, other functions that take two Anys as input may not benefit. As another example, we have found reduce functions that return an Any and therefore fail to describe type constraints between their inputs and outputs. But, not every function with Any as its return type is parametrically polymorphic in the same way as a reducer.

4.2 Implicit Dynamic Due to Unconstrained Type Variables

Two occurrences of the same type variable must be instantiated with matching types. Mypy's default notion of matching is rather loose because of subtyping, and thus may lead to surprising outcomes. In the following example, the call to count_cars instantiates the variable Car as both a string and a number; this is not a type error because strings and numbers have a common supertype in mypy:

```python
Car = TypeVar('Car') # car is unconstrained / unbounded
Traffic = Union[Car, List['Traffic']]

def count_cars(x: Traffic, car: Car) -> int:
  if isinstance(x, List):
    x.append(car)
  return len(x)

count_cars(["FJ40", "Baja Buggy"], 5) # NOT a type error
```

To prevent such weakening, type variables must be declared with either a set of constraints or an upper bound (e.g., `TypeVar('Car', str, bytes)` or `TypeVar('Car', bound=AnyStr)`).

Unconstrained type variables are straightforward to detect: there are 471 of them across our 221 sample projects. Discovering where these variables lead to weak types is another matter, and this calls for instrumentation within mypy to report variables that get instantiated with a top type.

4.3 Dynamic Instead of a Self Type

Some methods that return their receiver object use Any as the return type to allow for subclass polymorphism. In the example below, the return type Shape would be too conservative (it would prevent the use of subclass methods after `Circle().move(...)`) and the return type Circle would be unsound. Using Any gets rid of spurious errors by skipping type analysis altogether:

```python
class Shape:
  def move(self, dist: int) -> Any: # imprecise return type, Self is better
    self.position += dist
    return self

class Circle(Shape):
  pass

Circle().move(4)
```

A precise alternative is Self as a return type, which propagates the type of the receiver to the result. Mypy added support for Self in version 1.0 [49]; code in our corpus evidently needs an upgrade, either because of a knowledge gap or because it was written before the mypy 1.0 release. To flag this pattern, an analysis must find methods that return their first parameter and use Any as the return type.

4.4 Dynamic for Dependent Dictionaries

The mypy type `Dict[T1, T2]` describes simple dictionaries with keys of type T1 and values of type T2. This type cannot express dictionaries in which specific keys point to values of different types. In the example below, for instance, the key `"price"` assumes float values:

```python
def get_discount(item: Dict[str, Any]) -> int:
  if "price" in item:
    discount = item["price"] * 0.15
    return item["price"] - discount
```

This is a modest form of dependent types that appears throughout our corpus. In the 221 sample projects, there are 6,831 signatures that use Any for dictionary values (i.e., Dict, Dict[T1,Any], or Mapping[T1,Any]). Row polymorphism using scoped labels [41] or singleton types (as in Haskell [19], Typed Racket [65], or Elixir [14]) may suffice to improve mypy. Alternatives are to adapt TypeScript's indexed access types [68], introduce a metaprogramming layer for type tailoring [74], or integrate an SMT solver [37, 70].

4.5 Dynamic for Method Overrides

TypeScript infamously allows covariant method overrides—even though they are unsound—because developers apparently want to write subclass methods that assume a narrow set of inputs [8]. Mypy

prevents covariant overrides by default, but developers can opt for unsound overrides by writing a special comment after the subclass method signature. Such comments do not use Any directly, but are nevertheless relevant to our study of how developers choose to weaken a type checker. The following example is from the ivre [35] project:

```python
class BinaryIO(IO[bytes]):
  def write(self, s: Union[bytes, bytearray]) -> int:
    pass


class FileOpener(BinaryIO):
  def write(self, s: bytes) -> int: # type: ignore[override]
    return self.fdesc.write(s)
```

We have written an analysis for this pattern that finds these special comments, looks for the parent class (which may be in a different file, or inaccessible to us in another project), and examines the relationship between the parent and child method types. There were 652 occurrences of this special comment across our sample corpus. We found the parent class for 146 such comments, and 80.8 % (118 / 146) of these cases use different argument types for parent and child methods. These may be covariant overrides, but further analysis is needed to confirm.

4.6 Dynamic for Wrapper Functions

A wrapper takes a function as input and returns a modified or enhanced version of the function. The wrapper below, for instance, takes a function fn that expects a statistics object and returns a function that expects a string; the wrapped function converts input strings to objects and calls the original fn function:

```python
def validate_stat(fn: Callable) -> Callable:
  def string_fn(self, stat: str, *args, **kwargs) -> Callable:
    stat = string_to_stat(stat)
    return fn(self, stat, *args, **kwargs)
  return string_fn
```

The type for this wrapper is merely Callable -> Callable, which says nothing about the close relationship between the input and output functions. It may be possible to adapt types for variable-arity polymorphism [62] to directly express the concept of wrappers.

We have written an analysis for this pattern. It examines functions that have a parameter with the plain type Callable and reports a match if this parameter is called always with catch-all positional and keyword arguments. There are 906 such functions across the sample projects.

4.7 Dynamic to Hide Unnecessary Details

When types require details that are unimportant to the code at hand, developers can use the Any type to fill in the holes. For example, this function from the nanonote [52] project simply returns the last value from a dictionary. Any dictionary is a valid input, so the signature asks for a value of type Dict[Any, Any] (a type variable would be reasonable here):

```python
def _get_dict_last_added_item(dct: Dict[Any, Any]) -> Any:
  return list(dct.values())[-1]
```

Similar examples come from parameterized types such as `PathLike[T]` or `Callable[T1, T2]`. Replacing the parameter with `Any` is a simple, valid choice. It might be more precise to ask for a top parameterized type, but that requires thought about whether each parameter should use the top type (`object`) or bottom type (`Never`) depending on whether it is co- or contra-variant.

We do not have an automatic analysis for this pattern. One idea is to collect all uses of a parameter such as `dct` and see what constraints the uses impose on its type parameters. If there are no constraints and the parameter type is `Any`, we have a match.

4.8 Uncategorized Dynamic Types

Many `Any` occurrences do not fit a clear pattern. Some could easily be replaced with precise alternatives. For example, the wandb [73] project contains a function that always raises an exception but gives `Any` rather than `Never` (or `NoReturn`) as its return type:

```python
def __getattr__(self, key: str) -> Any:
  if not key.startswith("_"):
    raise wandb.Error(f"...")
  else:
    raise AttributeError
```

Refining such types may not be a priority for developers. Perhaps they would accept a patch generated through the use of a typing tool, such as Scotty [33] or MonkeyType [34].

5 FUTURE WORK

Our work is the first step toward a large-scale corpus study to discover how the dynamic type is used in practice. We have identified eight patterns and we have plans to implement analyses for five of them (§§4.2, 4.3, 4.5, 4.6, 4.7). Next, we need to implement these analyses and develop methods for the remaining patterns: type variables (§4.1), dependent dictionaries (§4.4), and some uncategorized cases (§4.8). Along the way, we may of course discover additional patterns or refine the current patterns into more-specific categories.

In parallel, we can work to improve mypy. Dependent dictionaries (§4.4) and wrapper functions (§4.6) call for a more expressive typechecker. The patterns related to type variables (§§4.1, 4.2), self types (§4.3), and omitted details (§§4.7,4.8) would benefit from tools that suggest type improvements. In fact, implementing a tool may be the most effective way to gain insight about uncategorized Anys.

6 RELATED WORK

There have been several corpus studies related to gradual types. Rak-amnouykit et al. [53] study types in 2,678 Python projects from GitHub. Those projects contained thousands of instances of the `Any` type, which reflects our experience. Their study focuses on which types get used and whether the types actually typecheck, whereas our work investigates the reasoning behind the `Any` type to suggest type system improvements. Jin et al. [36] study the use of non-`Any` types in Python projects to uncover design patterns for complex types, providing insights to grow the type system in a way that is complementary to our work. They find that `Any` is the second most common type, behind `Optional`, with over 12K instances across the revision histories of 19 projects. Lin et al. [42] apply multiple typecheckers to 13 Python projects to compare the defect reports; they do not study the adoption of types. Corpus studies of types in R [23, 67] and Ruby [18] provided further inspiration for our work.

344   Other works have studied types and code quality. Bogner and Merkel [9] study the effect of
345   TypeScript types on quality metrics, such as the number of code smells and the bugfix-commit
346   ratio, with an eye toward the Any type. They find that Any correlates positively with code smells,
347   which suggests that removing Any is valuable, but they also find no correlation to bugfix commits,
348   which may mean that Any is not an impediment to maintenance. Khan et al. [38] examine 400 bugs
349   that had been committed to 210 Python repositories and attempt to catch these bugs by adding
350   mypy annotations. Gao et al. [22] perform a similar experiment with 400 bugs in 398 JavaScript
351   projects, adding TypeScript and Flow annotations. Both studies concluded that at least 15 % of the
352   bugs could have been caught with types. Xu et al. [76] conduct a similar study using 40 bugs across
353   10 Python projects and find that the use of three typecheckers can detect 35 % of bugs with no
354   additional annotations. Adding types improves the detection rate to 72 %.

355   We remark that corpus studies alone cannot tell the whole story about the use of the dynamic
356   type. Code in public repositories says nothing about the trial-and-error development work that led
357   to the latest checkpoint. Our study is even more limited because it considers only one version of
358   each codebase, unlike some related works [9, 22, 38, 76]. Other types of studies are needed to capture
359   the bigger picture. Interviews and observations of developers at work (such as [2, 24]) would give
360   highly-detailed data on sample developers. Surveys are a lightweight alternative that can reach a
361   broader audience [1, 4, 5, 13, 44, 59, 66]. Developers might submit examples of how and why they
362   use the dynamic type to an online or IDE-integrated form. Telemetry that automatically detects and
363   reports usage of the dynamic type is a third option. Privacy is a concern with telemetry; by way of
364   mitigation, prior works have relied on summary statistics [17, 27] and differential privacy [30, 77].

365   A final point to consider is whether lab studies can effectively assess use of the dynamic type. Lab
366   studies are typically confined to small, self-contained tasks (as in [15, 16, 56–58]), but the benefits
367   and challenges of the dynamic type shine most in complex projects. With the dynamic type, code
368   has more flexibility in the face of evolution and offers less guidance to developers trying to learn
369   the codebase. How to effectively study these tradeoffs in a lab setting is an open challenge.

7 DISCUSSION

372   The dynamic type is widely used, and therefore appears crucial for gradual type adoption. It also
373   weakens type analysis, and is thus best as a temporary stop along the road to precise types. We
374   have identified eight patterns of dynamic type use in mypy through a combination of manual
375   (in dozens of projects) and automated analysis (in 221 projects). While it remains to be seen how
376   often the patterns arise in a wider sample and thus how critical each pattern is at large, the
377   results so far suggest improvements to mypy, such as modest dependent types for configuration
378   dictionaries (section 4.4), and to the mypy ecosystem, such as a tool to insert Self types (section 4.3).

379   Looking beyond mypy, the long-term goal of this work is to develop a method for gradual type
380   system design: start by extending a conventional type system with the dynamic type, observe
381   how developers use the dynamic type, and introduce precise types that meet developers where
382   they are. We expect that several patterns we have found, such as dynamic instead of a type
383   variable (section 4.1), will lead to useful observations in related languages including Luau [11, 12]
384   and TypeScript [8]. Such patterns belong in a toolkit that delivers insights from *big code* [71] to
385   find common friction points and thereby balance type system expressiveness with usability.

REFERENCES

[1] Amjad AlTadmri and Neil C. C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *SIGCSE*. ACM, 522–527. https://doi.org/10.1145/2676723.2677258

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *PACMPL* 7, OOPSLA (2023), 85–111. https://doi.org/10.1145/3586030

[3] Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: only Mostly Dead. *PACMPL* 1, OOPSLA (2017), 54:1–54:24. https://doi.org/10.1145/3133878

[4] Brett A. Becker, Graham Glanville, Ricardo Iwashima, Claire McDonnell, Kyle Goslin, and Catherine Mooney. 2016. Effective Compiler Error Message Enhancement for Novice Programming Students. *Computer Science Education* 26, 2-3 (2016), 148–175. https://doi.org/10.1080/08993408.2016.1225464

[5] Moritz Beller, Igor Levaja, Annibale Panichella, Georgios Gousios, and Andy Zaidman. 2016. How to Catch 'Em All: WatchDog, a Family of IDE Plug-Ins to Assess Testing. In *ICSE*. ACM, 53–56. https://doi.org/10.1145/2897022.2897027

[6] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. 2019. On the Impact of Programming Languages on Code Quality: A Reproduction Study. *Transactions on Programming Languages and Systems* 41, 4, Article 21 (2019), 24 pages. https://doi.org/10.1145/3340571

[7] Emery D. Berger, Petr Maj, Olga Vitek, and Jan Vitek. 2019. FSE/CACM Rebuttal[2]: Correcting A Large-Scale Study of Programming Languages and Code Quality in GitHub. *CoRR* abs/1911.11894 (2019), 18 pages. arXiv:1911.11894

[8] Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*. ACM, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[9] Justus Bogner and Manuel Merkel. 2022. To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub. In *MSR*. ACM, 658–669. https://doi.org/10.1145/3524842.3528454

[10] Gilad Bracha. 2004. Pluggable Type Systems. https://www.bracha.org/pluggableTypesPosition.pdf Accessed 2024-07-01.

[11] Lily Brown, Andy Friesen, and Alan Jeffrey. 2021. Position Paper: Goals of the Luau Type System. In *HATRA*. 7 pages. arXiv:2109.11397

[12] Lily Brown, Andy Friesen, and Alan Jeffrey. 2023. Goals of the Luau Type System, Two Years On. In *HATRA*. 2 pages. https://asaj.org/papers/hatra23.pdf Accessed 2024-08-27.

[13] Neil C. C. Brown, Amjad AlTadmri, Sue Sentance, and Michael Kölling. 2018. Blackbox, Five Years On: An Evaluation of a Large-scale Programming Data Collection Project. In *ICER*. ACM, 196–204. https://doi.org/10.1145/3230977.3230991

[14] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2024. The Design Principles of the Elixir Type System. *Programming* 8, 2 (2024), 4:1–4:39. https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/4

[15] Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. 2022. Garbage Collection Makes Rust Easier to Use: A Randomized Controlled Trial of the Bronze Garbage Collector. In *ICSE*. ACM, 1021–1032. https://doi.org/10.1145/3510003.3510107

[16] Michael J. Coblenz, Jonathan Aldrich, Brad A. Myers, and Joshua Sunshine. 2020. Can Advanced Type Systems be Usable? An Empirical Study of Ownership, Assets, and Typestate in Obsidian. *PACMPL* 4, OOPSLA (2020), 132:1–132:28. https://doi.org/10.1145/3428200

[17] Russ Cox. 2023. Transparent Telemetry, Part 1. https://research.swtch.com/telemetry-intro Accessed 2024-07-01.

[18] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. 2009. Work In Progress: an Empirical Study of Static Typing in Ruby. In *PLATEAU*. 6 pages. https://www.cs.tufts.edu/~jfoster/papers/plateau09-ruby.pdf Accessed 2024-07-07.

[19] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *Haskell*. ACM, 117–130. https://doi.org/10.1145/2364506.2364522

[20] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *PACMPL* 2, OOPSLA (2018), 133:1–133:27. https://doi.org/10.1145/3276503

[21] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59. https://doi.org/10.1145/581478.581484

[22] Zheng Gao, Christian Bird, and Earl T. Barr. 2017. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. In *ICSE*. IEEE, 758–769. https://doi.org/10.1109/ICSE.2017.75

[23] Aviral Goel and Jan Vitek. 2019. On the Design, Implementation, and Use of Laziness in R. *PACMPL* 3, OOPSLA (2019), 153:1–153:27.

[24] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *ICSE*. ACM, Article 187, 13 pages. https://doi.org/10.1145/3597503.3639581

[25] Ben Greenman. 2022. Deep and Shallow Types for Gradual Languages. In *PLDI*. ACM, 580–593. https://doi.org/10.1145/3519939.3523430

[26] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. Typed–Untyped Interactions: A Comparative Analysis. *TOPLAS* 45, 4 (2023), 1–54. Issue 1. https://doi.org/10.1145/3579833

[27] Ben Greenman, Alan Jeffrey, Shriram Krishnamurthi, and Mitesh Shah. 2024. Privacy-Respecting Type Error Telemetry at Scale. *Programming* 8, 3 (2024), 12:1–12:30. https://doi.org/10.22152/programming-journal.org/2024/8/12

[28] Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *PEPM*. ACM, 30–39. https://doi.org/10.1145/3162066

[29] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019. How to Evaluate the Performance of Gradual Type Systems. *Journal of Functional Programming* 29, e4 (2019), 1–45. https://doi.org/10.1145/3473573

[30] Yu Hao, Sufian Latif, Hailong Zhang, Raef Bassily, and Atanas Rountev. 2021. Differential Privacy for Coverage Analysis of Software Traces. In *ECOOP*, Vol. 194. Schloss Dagstuhl, 8:1–8:25. https://doi.org/10.4230/LIPIcs.ECOOP.2021.8

[31] Anders Hejlsberg. 2023. TypeScript: Static Types for JavaScript. https://icfp23.sigplan.org/details/icfp-2023-icfp-keynotes/52/TypeScript-Static-types-for-JavaScript

[32] Charles A. Hoare. 1989. *Hints on Programming-Language Design.* Prentice-Hall, 193–216.

[33] Joshua Hoeflich, Robert Bruce Findler, and Manuel Serrano. 2022. Highly illogical, Kirk: spotting type mismatches in the large despite broken contracts, unsound types, and too many linters. *PACMPL* 6, OOPSLA2 (2022), 479–504. https://doi.org/10.1145/3563305

[34] Instagram. 2024. MonkeyType. https://github.com/Instagram/MonkeyType Accessed 2024-07-01.

[35] IVRE Team. 2024. ivre. https://github.com/ivre/ivre Accessed 2024-08-09.

[36] Wuxia Jin, Dinghong Zhong, Zifan Ding, Ming Fan, and Ting Liu. 2021. Where to Start: Studying Type Annotation Practices in Python. In *ASE*. IEEE, 529–541. https://doi.org/10.1109/ASE51524.2021.9678947

[37] Andrew M. Kent, David Kempe, and Sam Tobin-Hochstadt. 2016. Occurrence Typing Modulo Theories. In *PLDI*. ACM, 296–309. https://doi.org/10.1145/2908080.2908091

[38] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. 2022. An Empirical Study of Type-Related Defects in Python Projects. *IEEE Transactions on Software Engineering* 48, 8 (2022), 3145–3158. https://doi.org/10.1109/TSE.2021.3082068

[39] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *PLDI*. ACM, 517–532. https://doi.org/10.1145/3314221.3314627

[40] Jukka Lehtosalo. 2019. Our Journey to Checking 4 Million Lines of Python. https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python

[41] Daan Leijen. 2005. Extensible Records with Scoped Labels. In *TFP*. 179–194. https://www.microsoft.com/en-us/research/publication/extensible-records-with-scoped-labels/

[42] Xinrong Lin, Baojian Hua, Yang Wang, and Zhizhong Pan. 2023. Towards a Large-Scale Empirical Study of Python Static Type Annotations. In *SANER*. IEEE, 414–425. https://doi.org/10.1109/SANER56733.2023.00046

[43] Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2023. Gradual Soundness: Lessons from Static Python. *Programming* 7, 1 (2023), 2:1–2:40.

[44] Kuang-Chen Lu, Shriram Krishnamurthi, Kathi Fisler, and Ethel Tshukudu. 2023. What Happens When Students Switch (Functional) Languages (Experience Report). *PACMPL* 7, ICFP, Article 215 (2023), 17 pages. https://doi.org/10.1145/3607857

[45] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *Transactions on Programming Languages and Systems* 31, 3 (2009), 1–44. https://doi.org/10.1145/1498926.1498930

[46] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375.

[47] Mozilla. 2022. The Agnostic GitHub API. https://github.com/mozilla/agithub

[48] Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA (2017), 56:1–56:30. https://doi.org/10.1145/3133880

[49] Mypy Team. 2023. Mypy 1.0 Released. https://mypy-lang.blogspot.com/2023/02/mypy-10-released.html

[50] Mypy Team. 2024. Error Codes. https://mypy.readthedocs.io/en/stable/error_codes.html

[51] Mypy Team. 2024. Mypy Language. http://www.mypy-lang.org

[52] Nanonote Team. 2024. nanonote. https://github.com/agateau/nanonote Accessed 2024-08-09.

[53] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana L. Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 types in the wild: a tale of two type systems. In *DLS*. ACM, 57–70. https://doi.org/10.1145/3426422.3426981

[54] Baishakhi Ray, Prem Devanbu, and Vladimir Filkov. 2019. Rebuttal to Berger et al., TOPLAS 2019. *CoRR* abs/1911.07393 (2019), 12 pages. arXiv:1911.07393

[55] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in GitHub. In *FSE*. ACM, 155–165. https://doi.org/10.1145/2635868.2635922

[56] Jan Reichl, Stefan Hanenberg, and Volker Gruhn. 2023. Does the Stream API Benefit from Special Debugging Facilities? A Controlled Experiment on Loops and Streams with Specific Debuggers. In *ICSE*. IEEE, 576–588. https://doi.org/10.1109/ICSE48619.2023.00058

[57] Phyllis Reisner. 1981. Human Factors Studies of Database Query Languages: A Survey and Assessment. *Comput. Surveys* 13, 1 (1981), 13–31. https://doi.org/10.1145/356835.356837

[58] Nico Ritschel, Anand Ashok Sawant, David Weintrop, Reid Holmes, Alberto Bacchelli, Ronald Garcia, Chandrika K. R., Avijit Mandal, Patrick Francis, and David C. Shepherd. 2023. Training Industrial End-User Programmers with Interactive Tutorials. *Software: Practice and Experience* 53, 3 (2023), 729–747. https://doi.org/10.1002/SPE.3167

[59] Eric L. Seidel, Ranjit Jhala, and Westley Weimer. 2018. Dynamic Witnesses for Static Type Errors (or, Ill-Typed Programs Usually Go Wrong). *Journal of Functional Programming* 28 (2018), e13. https://doi.org/10.1017/S0956796818000126

[60] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *SFP. University of Chicago, TR-2006-06.* 81–92. http://scheme2006.cs.uchicago.edu/scheme2006.pdf

[61] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *SNAPL.* Schloss Dagstuhl, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

[62] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. 2009. Practical Variable-Arity Polymorphism. In *ESOP.* Springer, 32–46. https://doi.org/10.1007/978-3-642-00590-9_3

[63] Chenghao Su, Lin Chen, Yanhui Li, and Yuming Zhou. 2024. Static Blame for Gradual Typing. *Journal of Functional Programming* 34, e4 (2024), 44 pages. https://doi.org/10.1017/S0956796824000029

[64] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS.* 964–974. https://doi.org/10.1145/1176617.1176755

[65] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *POPL.* ACM, 395–406. https://doi.org/10.1145/1328438.1328486

[66] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. 2018. The Behavior of Gradual Types: a User Study. In *DLS.* ACM, 1–12. https://doi.org/10.1145/3276945.3276947

[67] Alexi Turcotte, Aviral Goel, Filip Krikava, and Jan Vitek. 2020. Designing types for R, empirically. *PACMPL* 4, OOPSLA (2020), 181:1–181:25. https://doi.org/10.1145/3428249

[68] TypeScript Developers. 2024. Keyof Type Operator. https://www.typescriptlang.org/docs/handbook/2/keyof-types.html Accessed 2024-07-01.

[69] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2024. PEP 484 Type Hints. https://www.python.org/dev/peps/pep-0484

[70] Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover.* Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

[71] Martin Vechev and Eran Yahav. 2016. Programming with "Big Code". *Foundations and Trends in Programming Languages* 3, 4 (2016), 231–284. https://doi.org/10.1561/2500000028

[72] Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *POPL.* ACM, 762–774. https://doi.org/10.1145/3009837.3009849

[73] WANDB Team. 2024. wandb. https://github.com/wandb/wandb Accessed 2024-08-09.

[74] Ashton Wiersdorf, Stephen Chang, Matthias Felleisen, and Ben Greenman. 2024. Type Tailoring. In *ECOOP.* Schloss Dagstuhl, (to appear).

[75] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL.* ACM, 377–388. https://doi.org/10.1145/1706299.1706343

[76] Wenjie Xu, Lin Chen, Chenghao Su, Yimeng Guo, Yanhui Li, Yuming Zhou, and Baowen Xu. 2023. How Well Static Type Checkers Work with Gradual Typing? A Case Study on Python. In *ICPC.* IEEE, 242–253. https://doi.org/10.1109/ICPC58990.2023.00039

[77] Hailong Zhang, Yu Hao, Sufian Latif, Raef Bassily, and Atanas Rountev. 2020. Differentially-Private Software Frequency Profiling Under Linear Constraints. *PACMPL* 4, OOPSLA (2020), 203:1–203:24. https://doi.org/10.1145/3428271