An Embedding of the Python Type System in ACL2s

Samuel Xifaras, Panagiotis Manolios, Andrew T. Walter, William Robertson

May 13, 2025

What is Python?

- First released in 1991
- Second most popular language in the world (behind JavaScript)
- **Powers the modern world:** data science, machine learning, SaaS companies
- Thriving community and rich ecosystem of open source packages
- Not statically typed
- **Verifiably correct Python code is becoming increasingly important**

What are Type Annotations?

- Introduced to Python in 2014, in **PEP-484**
- **Static typing has many benefits:**
  - Improved modularity and earlier catching of simple bugs (Siek, 2014)
  - Reduced development time for certain categories of tasks (Hanenberg et al., 2014)
- Type annotations are growing in popularity in recent years (Di Grazia and Pradel, 2022)
- Effectively erased at runtime; not enforced

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Static Type Checkers

- Type annotations enforced by *type checkers*: e.g.: mypy, pytype
- Often integrated into CI/CD workflows
- Empirical study shows that mypy and pytype emit many false alarms (false positives)
  - Rak-amnouykit et al., 2020
- Rak-amnouykit et al. find that many repositories contain statically detectable errors
- Roth finds that Python's type hints are Turing complete (Roth, 2022)

Problem: Type checking is undecidable, hence cannot be sound & complete, and emits false positives and negatives

Motivating Example

```
def average(xs: List[float]) -> float:
    return sum(xs) / len(xs)
```

Motivating Example

- Type checking passes!

```
def average(xs: List[float]) -> float:
    return sum(xs) / len(xs)
```

Motivating Example

- Type checking passes!
- What if xs is empty?

```
def average(xs: List[float]) -> float:
    return sum(xs) / len(xs)
```

Motivating Example

- Type checking passes!
- What if xs is empty?
- What if xs[0] is NaN?

```
def average(xs: List[float]) -> float:
    return sum(xs) / len(xs)
```

Motivating Example

- Type checking passes!
- What if xs is empty?
- What if xs[0] is NaN?

- Type system is not expressive enough to express desired inputs/outputs

```
def average(xs: List[float]) -> float:
    return sum(xs) / len(xs)
```

```
average([]) # raises ZeroDivisionError

average([float('nan'), 3.0])  # nan
```

Motivating Example

- Type checking passes!
- What if xs is empty?
- What if xs[0] is NaN?

```
def average(xs: List[float]) -> float:
    assert len(xs) > 0
    return sum(xs) / len(xs)
```

- **Devs often try to make up for this with `assert`**

```
average([]) # raises AssertionError

average([float('nan'), 3.0]) # nan
```

Motivating Example

- Type checking passes!
- What if xs is empty?
- What if xs[0] is NaN?

```
def average(xs: List[float]) -> float:
    assert len(xs) > 0
    return sum(xs) / len(xs)
```

- **Devs often try to make up for this with `assert`**
- But this doesn't solve the problem

```
average([]) # raises AssertionError

average([float('nan'), 3.0]) # nan
```

Motivating Example

- Type checking passes!
- What if xs is empty?
- What if xs[0] is NaN?

```
def average(xs: List[float]) -> float:
    assert len(xs) > 0
    return sum(xs) / len(xs)
```

- **Devs often try to make up for this with `assert`**
- But this doesn't solve the problem

```
average([]) # raises AssertionError

average([float('nan'), 3.0]) # nan
```

Type checking cannot catch such issues

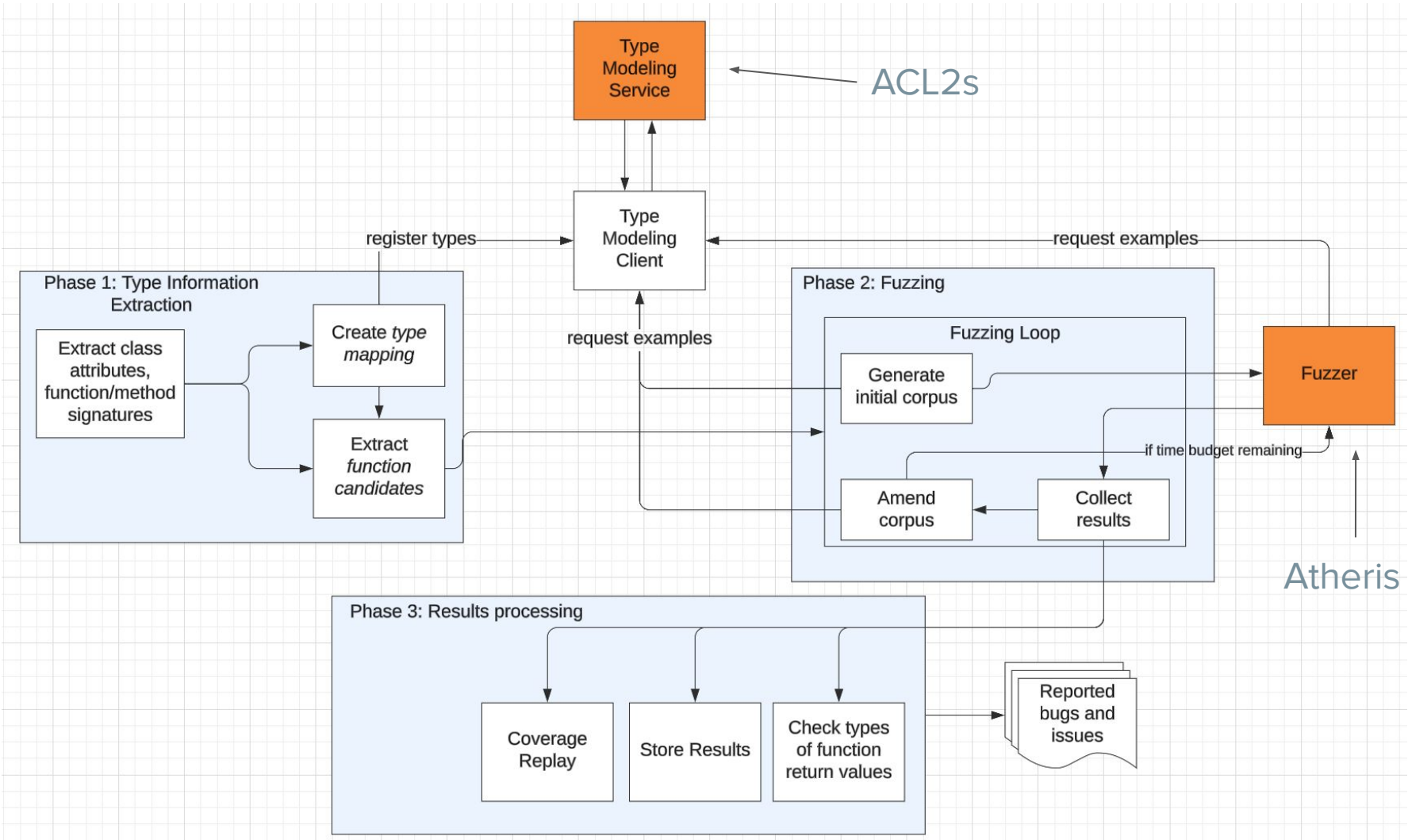
Easy to miss edge cases in unit testing

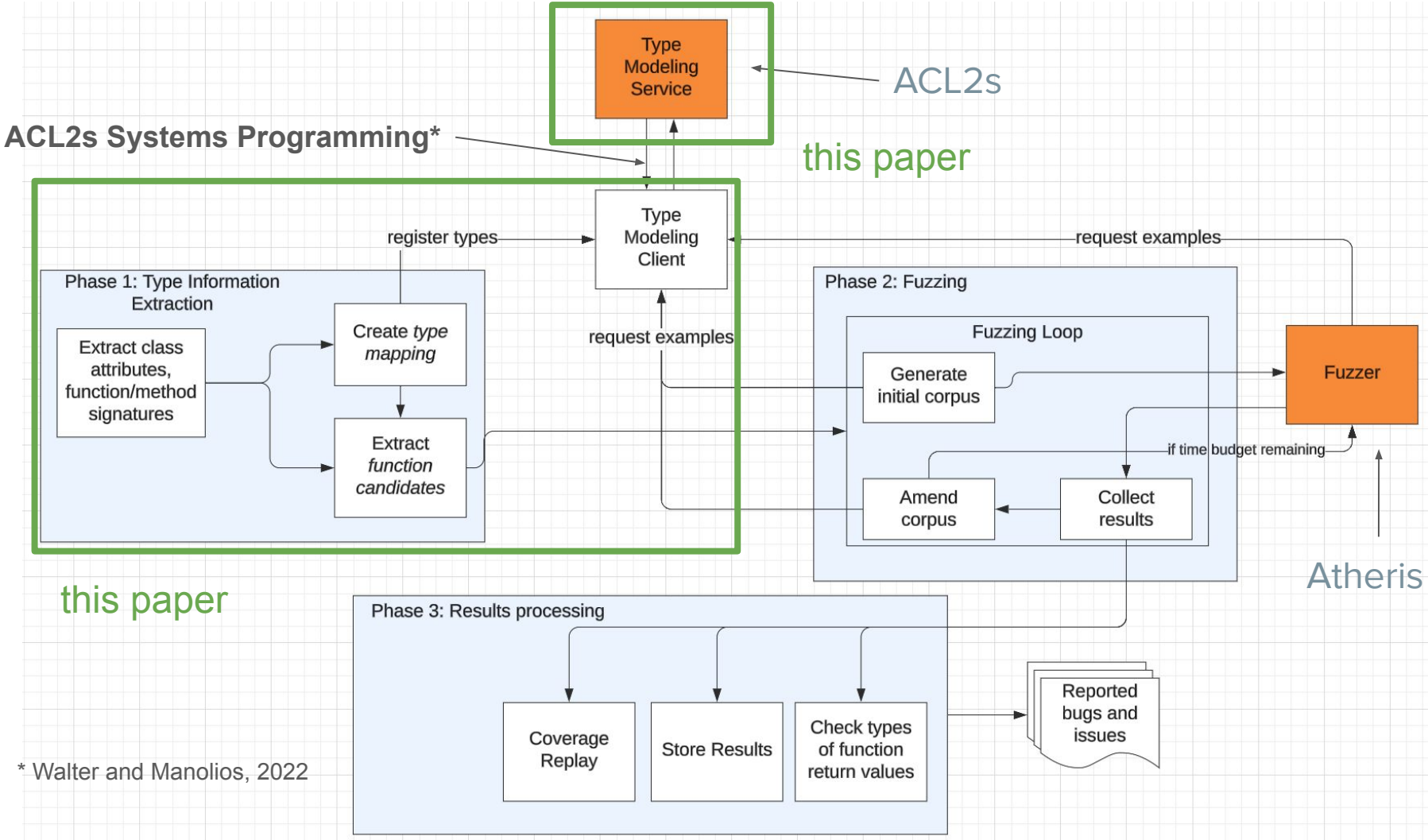
A third, complementary approach is needed

Our Approach: *Type Hint Fuzzing*

- Fuzz annotated functions in code, automatically supplying correct inputs according to parameter type annotations
- Catch crashes thrown by the code-under-test which indicate problematic behavior and type mismatches in the return values of functions.
- An autonomous "sanity check" that code works as expected given information about the types

Every issue is accompanied by a counterexample that can reproduce it, solving the false alarm problem.

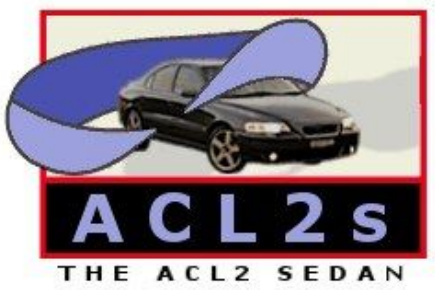

Type Modeling Service
ACL2s
Type Modeling Client
register types
request examples
Phase 1: Type Information Extraction
Extract class attributes, function/method signatures
Create type mapping
Extract function candidates
request examples
Phase 2: Fuzzing
Fuzzing Loop
Generate initial corpus
Fuzzer
if time budget remaining
Amend corpus
Collect results
Atheris
Phase 3: Results processing
Coverage Replay
Store Results
Check types of function return values
Reported bugs and issues



* Walter and Manolios, 2022

Why ACL2s?

- Theorem prover with a type system and data definition framework
  - Dillinger, Manolios, Moore, and Vroon, 2007
  - Chamarthi, Dillinger, and Manolios, 2014
- We leverage ACL2s for its data definition capabilities, and ***enumerative data types***
  - We also leverage the ability to define ***custom enumerators***
- Integration with ACL2s may allow the fuzzing system to leverage theorem proving in the future



Custom Enumerators for Primitive Types

- Integers
  - Arbitrary precision in both Python and ACL2s
  - **Enumerator**: covers small, large values, powers of 2 and values close to powers of 2
- Floats
  - Several special values that might be easy to neglect in manual test cases: **-0, nan, -inf, inf**
  - Encoded as rational numbers to leverage the Integer enumerator
  - **Enumerator**: p/q where p, q are integers + very large, very small, special cases
- Strings
  - The string type is embedded as a sequence of 32-bit integers (which we decode into Unicode chars)
  - **Enumerator**: length up to 10^4, mixture of various code point ranges (Greek letters, math symbols, emojis, diacritics, etc.)

Embedding Complex Types

- User-defined types can be admitted if they are *recursively representable*

```python
from typing import List, Tuple

class TestClassA:
    def __init__(self, a: float, b: List[int]) -> None:
        self.a = a
        self.b = b

class TestClassB:
    def __init__(self, a: int, b: TestClassA) -> None:
        self.a = a
        self.b = b

def use_a(a: TestClassA) -> Tuple[float, List[int]]:
    return (a.a, a.b)

def use_b(b: TestClassB) -> Tuple[int, TestClassA]:
    return (b.a, b.b)

a_inst = TestClassA(3.5, [1, 2, 3])
b_inst = TestClassB(4, a_inst)

use_a(a_inst)
use_b(b_inst)
```

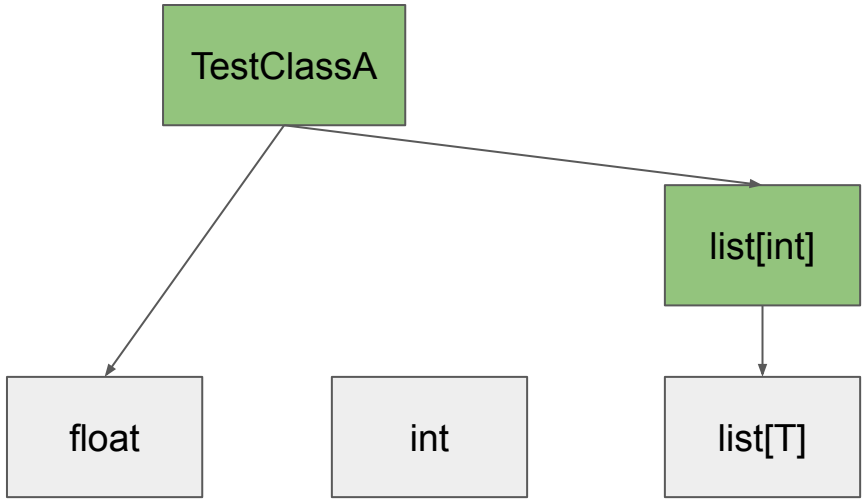
float

int

list[T]

Embedding Complex Types

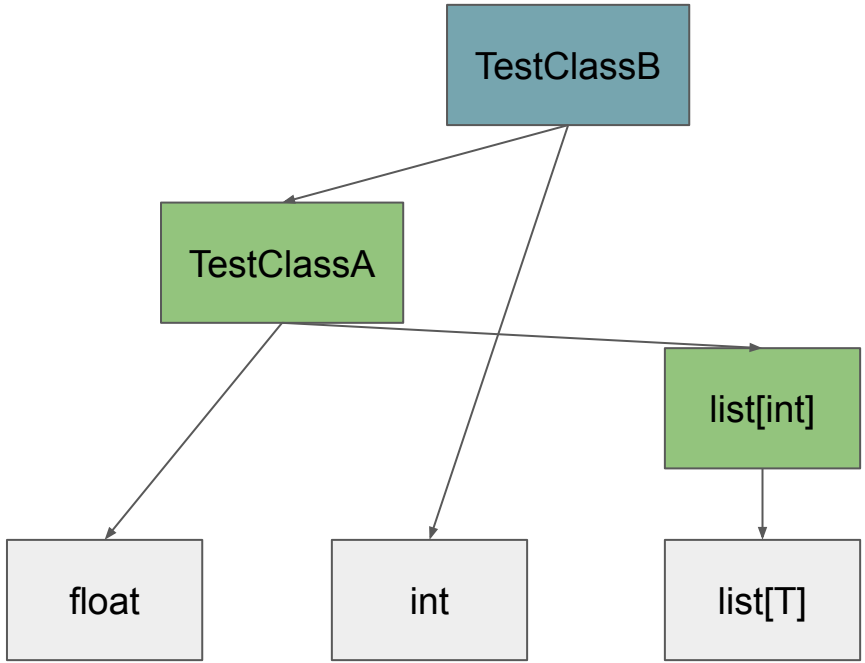
- User-defined types can be admitted if they are *recursively representable*




```python
from typing import List, Tuple

class TestClassA:
    def __init__(self, a: float, b: List[int]) -> None:
        self.a = a
        self.b = b

class TestClassB:
    def __init__(self, a: int, b: TestClassA) -> None:
        self.a = a
        self.b = b

def use_a(a: TestClassA) -> Tuple[float, List[int]]:
    return (a.a, a.b)

def use_b(b: TestClassB) -> Tuple[int, TestClassA]:
    return (b.a, b.b)

a_inst = TestClassA(3.5, [1, 2, 3])
b_inst = TestClassB(4, a_inst)

use_a(a_inst)
use_b(b_inst)
```

Embedding Complex Types

- User-defined types can be admitted if they are *recursively representable*




```python
from typing import List, Tuple

class TestClassA:
    def __init__(self, a: float, b: List[int]) -> None:
        self.a = a
        self.b = b

class TestClassB:
    def __init__(self, a: int, b: TestClassA) -> None:
        self.a = a
        self.b = b

def use_a(a: TestClassA) -> Tuple[float, List[int]]:
    return (a.a, a.b)

def use_b(b: TestClassB) -> Tuple[int, TestClassA]:
    return (b.a, b.b)

a_inst = TestClassA(3.5, [1, 2, 3])
b_inst = TestClassB(4, a_inst)

use_a(a_inst)
use_b(b_inst)
```

TestClass{A, B} Defdata Calls

- Defdata calls emitted when analyzing entire file
- Note that the complex return annotation types on `use_a` and `use_b` are also admitted
- **TY1100** is the ID of the TestClassB type

```
;; Layer 0 (primitive definitions)
(DEFDATA TY1039 PY-INTEGER)       ;; int
(DEFDATA TY1041 PY-FLOAT)         ;; float

;; Layer 1: List[int] & TestClassA
(DEFDATA TY1043 (LISTOF TY1039))  ;; List[int]
(DEFDATA TESTCLASSA
         (RECORD (A . TY1041) (B . TY1043))

;; Layer 2: TestClassB
(DEFDATA TESTCLASSB
         (RECORD (A . TY1039) (B . TESTCLASSA))

;; Return types

;; Tuple[float, List[int]]
(DEFDATA TY1096 (LIST TY1041 TY1043))

;; Tuple[int, TestClassA]
(DEFDATA TY1100 (LIST TY1039 TESTCLASSA))
```

TestClassB JSON Payload

```
[
  [
    { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "0TAG" },
    { "TYPE": "sym", "PACKAGE": "ACL2S", "NAME": "TESTCLASSB" }
  ],
  [
    { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "A" },
    { "TYPE": "int", "VAL": "-281474976743424" }
  ],
  [
    { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "B" },
    [
      { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "0TAG" },
      { "TYPE": "sym", "PACKAGE": "ACL2S", "NAME": "TESTCLASSA" }
    ],
    [
      { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "A" },
      { "TYPE": "rat", "N": "14", "D": "134217729" }
    ],
    [
      { "TYPE": "sym", "PACKAGE": "KEYWORD", "NAME": "B" },
      { "TYPE": "int", "VAL": "5764607523034890 24" },
...
```

name of class

field a = -281474976743424

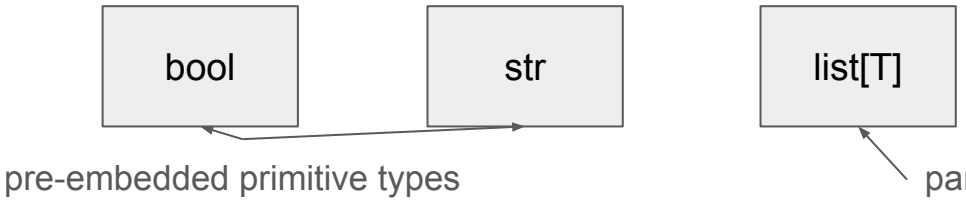
field b = instance of TestClassA

Real-world example: _Arguments

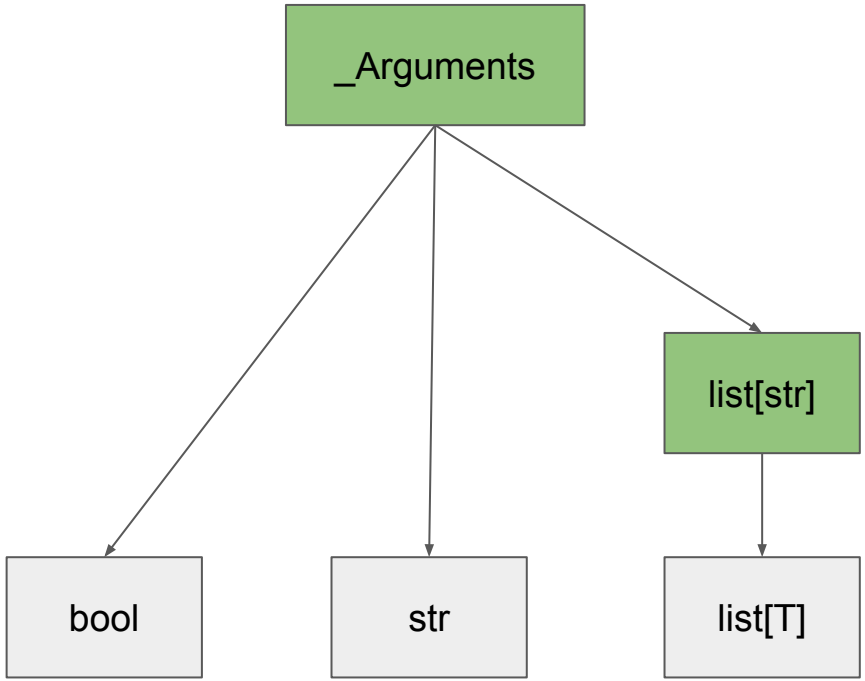
```
class _Arguments:
    modules: list[str]
    concise: bool
    ignore_missing_stub: bool
    ignore_positional_only: bool
    allowlist: list[str]
    generate_allowlist: bool
    ignore_unused_allowlist: bool
    mypy_config_file: str
    custom_typeshed_dir: str
    check_typeshed: bool
    version: str
```

Embedding of Complex Type: _Arguments

```
class _Arguments:
    modules: list[str]
    concise: bool
    ignore_missing_stub: bool
    ignore_positional_only: bool
    allowlist: list[str]
    generate_allowlist: bool
    ignore_unused_allowlist: bool
    mypy_config_file: str
    custom_typeshed_dir: str
    check_typeshed: bool
    version: str
```



Embedding of Complex Type: _Arguments




```
class _Arguments:
    modules: list[str]
    concise: bool
    ignore_missing_stub: bool
    ignore_positional_only: bool
    allowlist: list[str]
    generate_allowlist: bool
    ignore_unused_allowlist: bool
    mypy_config_file: str
    custom_typeshed_dir: str
    check_typeshed: bool
    version: str
```

Real-world example: Detected Issue

- Embedding of this class type allowed us to detect an *error in its definition*
- It is given as the return type for `mypy.stubtest.parse_options`
- Fuzzing experiments in Xifaras's MS Thesis revealed imprecise annotations for two of the class's fields (Xifaras, 2024)
- **This change was proposed and accepted in the mypy repository**

```
class _Arguments:
    modules: list[str]
    concise: bool
    ignore_missing_stub: bool
    ignore_positional_only: bool
    allowlist: list[str]
    generate_allowlist: bool
    ignore_unused_allowlist: bool
-   mypy_config_file: str
-   custom_typeshed_dir: str
+   mypy_config_file: str | None
+   custom_typeshed_dir: str | None
    check_typeshed: bool
    version: str
```

Evaluation

- Goal: "sanity check" that enumerators generate examples that cover code we would expect to cover
- Extracted "appropriate functions" from four open source repositories
- Performed five independent trials of 440-second fuzzing campaigns
- **All examples generated by ACL2s enumerators**
- Measured code coverage with `coverage.py`

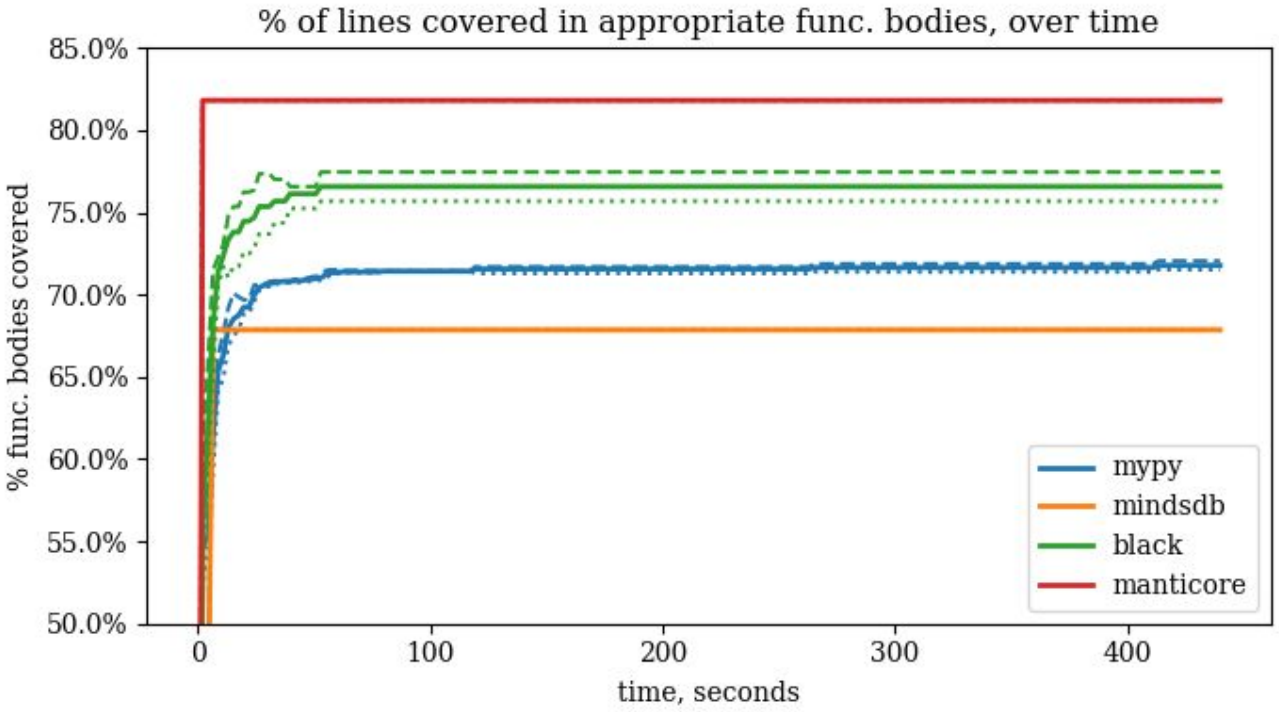
Appropriate Functions

- A function is **appropriate** iff its signature is 1) *fully annotated* and 2) every annotation in the signature is registered in the embedding
- A signature is *fully annotated* iff all arguments and return type are annotated

Repository	Total Functions	Annotated Functions	Appropriate Functions
mypy	1028	1028	132
mindsdb	400	55	5
black	248	248	35
manticore	211	26	5

Table 4: Function breakdown by repository.

Evaluation Results



Example of good coverage

- Basic blocks
- Easy conditionals
- Makes sense that we should cover this based on knowledge of types

```
def format_key_list(keys: list[str], *, short: bool = False) -> str:
    formatted_keys = [f'"{key}"' for key in keys]
    td = "" if short else "TypedDict "
    if len(keys) == 0:
        return f"no {td}keys"
    elif len(keys) == 1:
        return f"{td}key {formatted_keys[0]}"
    else:
        return f"{td}keys ({', '.join(formatted_keys)})"
```

Example 1 of bad coverage

- Complex conditional: unlikely that a randomly generated string would start and end with "__"

```
def infer_method_ret_type(name: str) -> str | None:
    """Infer return types for known special methods"""
    if name.startswith("__") and name.endswith("__"):
        name = name[2:-2]
        if name in ("float", "bool", "bytes", "int", "complex", "str"):
            return name
        # Note: __eq__ and co may return arbitrary types, but bool is good enough for stubgen.
        elif name in ("eq", "ne", "lt", "le", "gt", "ge", "contains"):
            return "bool"
        elif name in ("len", "length_hint", "index", "hash", "sizeof", "trunc", "floor", "ceil"):
            return "int"
        elif name in ("format", "repr"):
            return "str"
        elif name in ("init", "setitem", "del", "delitem"):
            return "None"
    return None
```

Example 2 of bad coverage

- Filesystem dependency

```
@mypyc_attr(patchable=True)
def parse_pyproject_toml(path_config: str) -> Dict[str, Any]:
    """Parse a pyproject toml file, pulling out relevant parts for Black.

    If parsing fails, will raise a tomllib.TOMLDecodeError.
    """
    pyproject_toml = _load_toml(path_config)
    config: Dict[str, Any] = pyproject_toml.get("tool", {}).get("black", {})
    config = {k.replace("--", "").replace("-", "_"): v for k, v in config.items()}

    if "target_version" not in config:
        inferred_target_version = infer_target_version(pyproject_toml)
        if inferred_target_version is not None:
            config["target_version"] = [v.name.lower() for v in inferred_target_version]

    return config
```

Example 3 of bad coverage

- Vague type annotation

```
def _get_current_node(profiling: dict) -> dict:
    """ return the node that the pointer points to

        Args:
            profiling (dict): whole profiling data

        Returns:
            dict: current node
    """
    current_node = profiling['tree']
    for child_index in profiling['pointer']:
        current_node = current_node['children'][child_index]
    return current_node
```

Discussion

- Our code coverage is good, given black-box function bodies
- We fail where we expect to fail:
  - Complex conditionals
  - File system dependencies
  - Vague type annotations
- Xifaras discusses the larger system this embedding is a part of (Xifaras, 2024)

Future Work

Embedding

- Expand set of supported types
  - Generics, co/contravariance, Protocol, Iterator, Sequence, etc.
- Embed semantics of commonly used conditional expressions and operators in ACL2s & use them to generate examples
  - "Enumerative data types with constraints" (Walter, Greve, and Manolios, 2022)
- Long term: embed semantics of Python (?)

Fuzzing System

- Leverage control flow structure of the code to improve code coverage
  - E.g. when fuzzing gets stuck, extract constraints and generate examples that satisfy/unsatisfy
- Property-based testing
- Type repair

Related Work

- Type checking in Python: not sound/complete; emits false positives
  - Rak-amnouykit et al., 2020; Di Grazia and Pradel, 2022; Roth, 2022
- Unit testing in Python is lackluster
  - Zhai et al., 2019
- Several attempts to formally specify Python, but they are of a particular version or particular subset of Python
  - Ranson, 2008; Politz et al., 2013; Smeding, 2009; Köhl, 2021
- Fuzzing in Python has been receiving more attention
  - Li et al., 2023 ("PyRTFuzz"); Zac Hatfield-Dodds, 2022 ("HypoFuzz"); Xifaras, 2024

Thank you!
Questions?

Link to source code: https://github.com/acl2/acl2/tree/master/books/projects/python/embedding
Get in touch! s.xifaras999@gmail.com