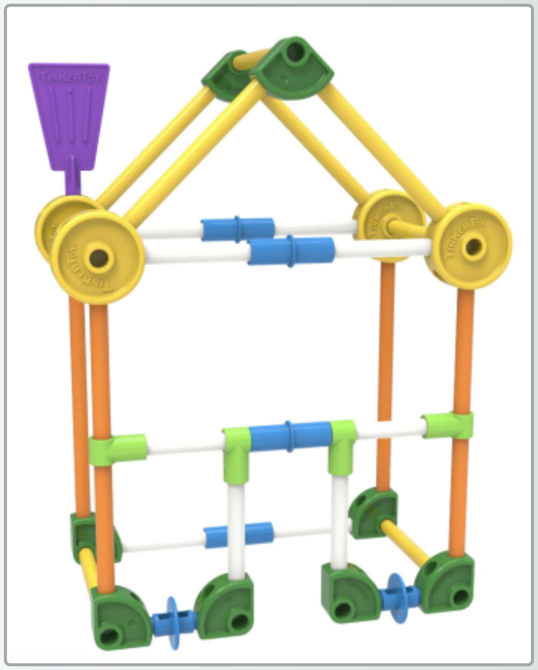
# Kicking the Ladder Away:

# From Gradual Types to Plain Types

Ben Greenman
2025-06-05
THE UNIVERSITY OF UTAH

Software is complicated.
Types help, but ...

... but Types are limiting.

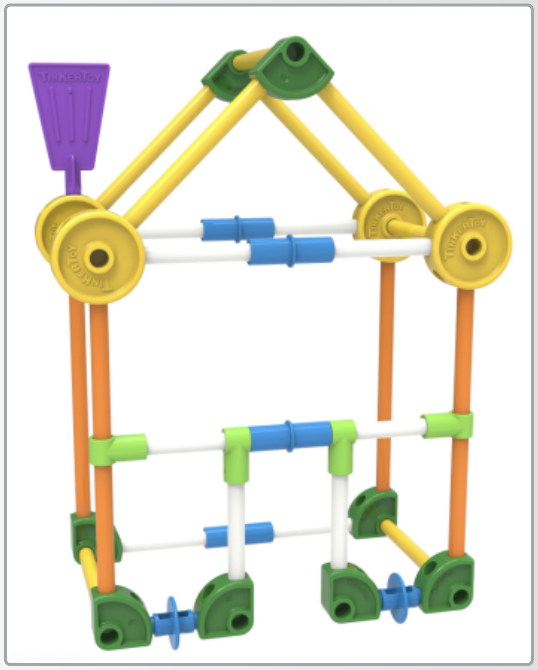


... but Types are limiting.

```
def div(a, b):
  return a / b if b != 0 else "div0"
```

vs.

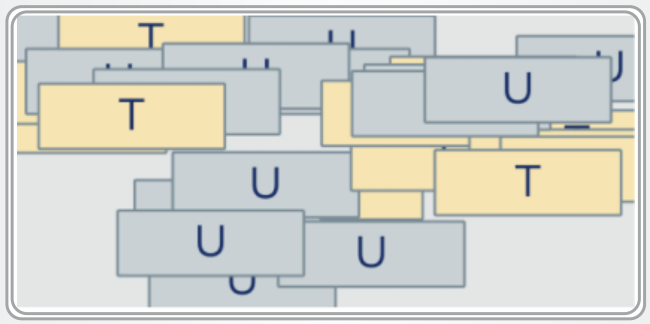
```
enum DivRes = Ok(int) | Err(str)

def div(a, b) -> DivRes:
  return Ok(a / b) if b != 0 else Err("div0")
```

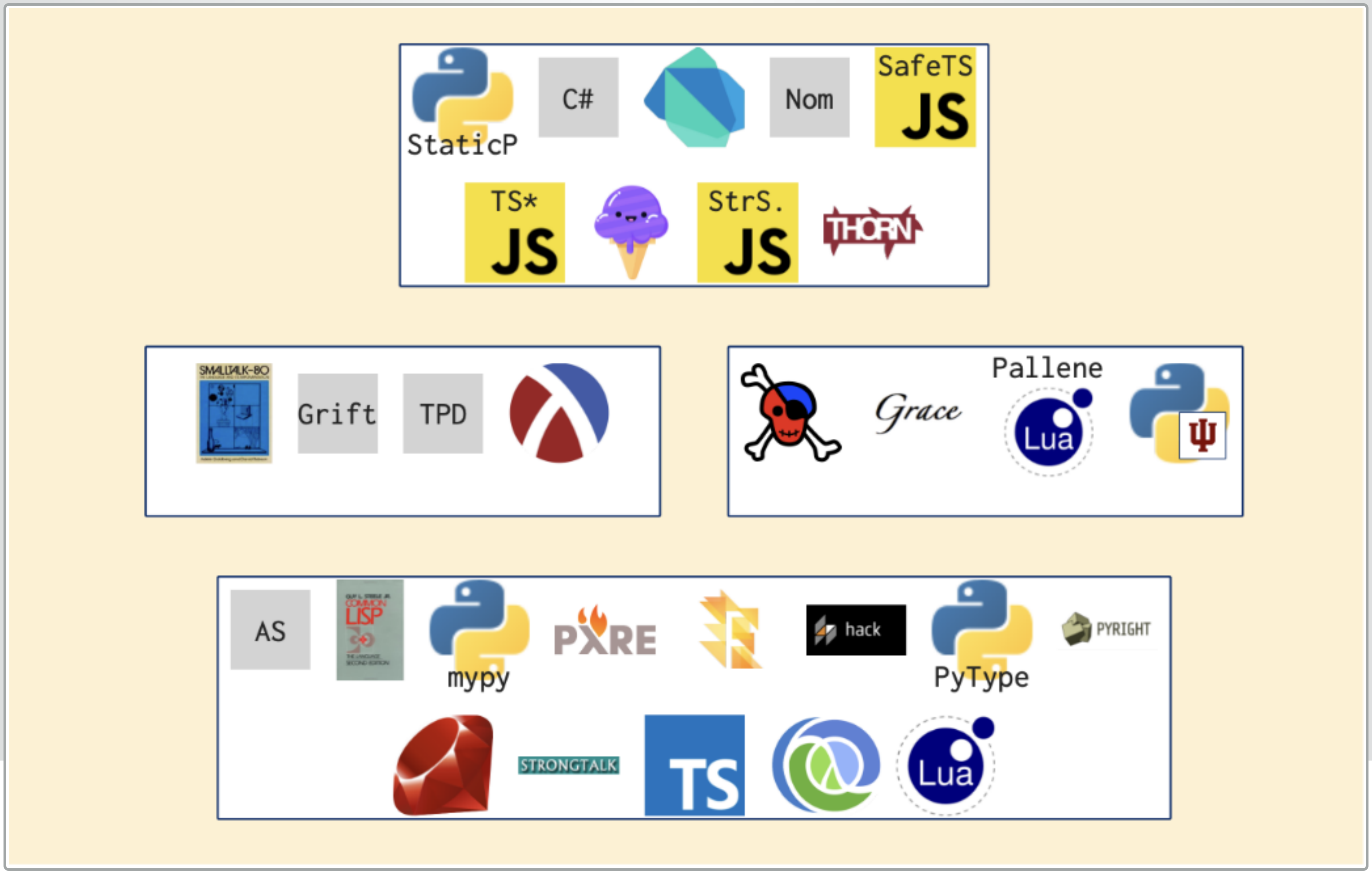


# Gradual Typing

Types where you need them, that's all!

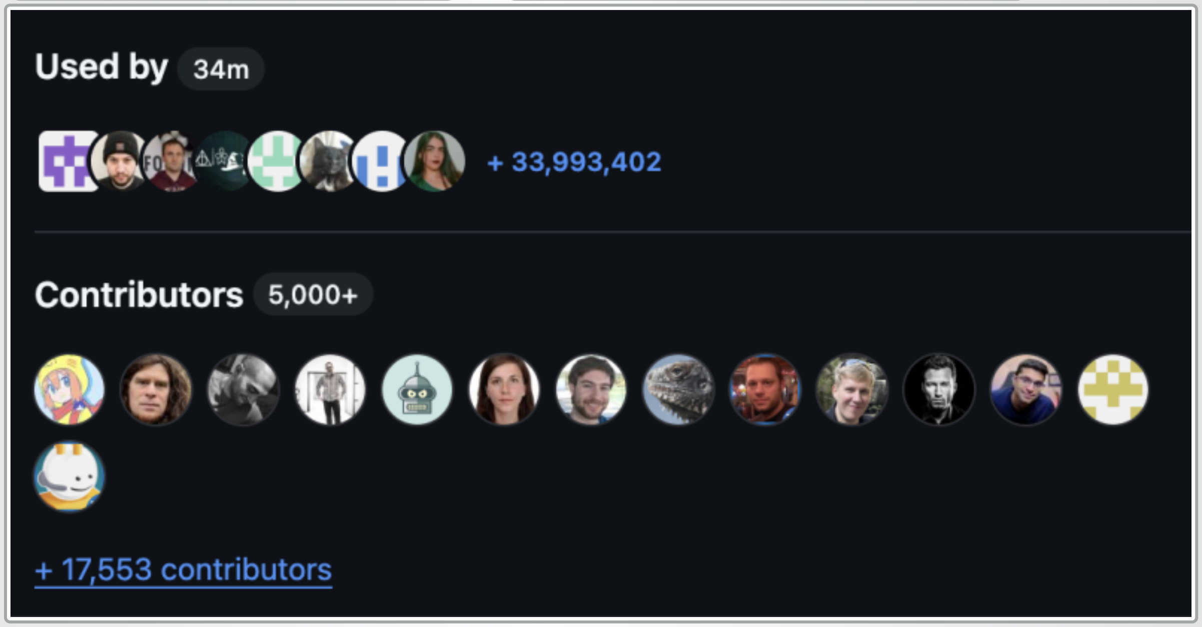
```python
def div(a: int, b: int) -> Any:
  return a / b if b != 0 else "div0"
```


U
T
U
T
U
U


StaticP
C#
Nom
SafeTS
JS
TS*
JS
StrS.
JS
THORN
SMALLTALK-80
Grift
TPD
Grace
Pallene
Lua
AS
COMMON LISP
mypy
PYRE
hack
PyType
PYRIGHT
STRONGTALK
TS
Lua

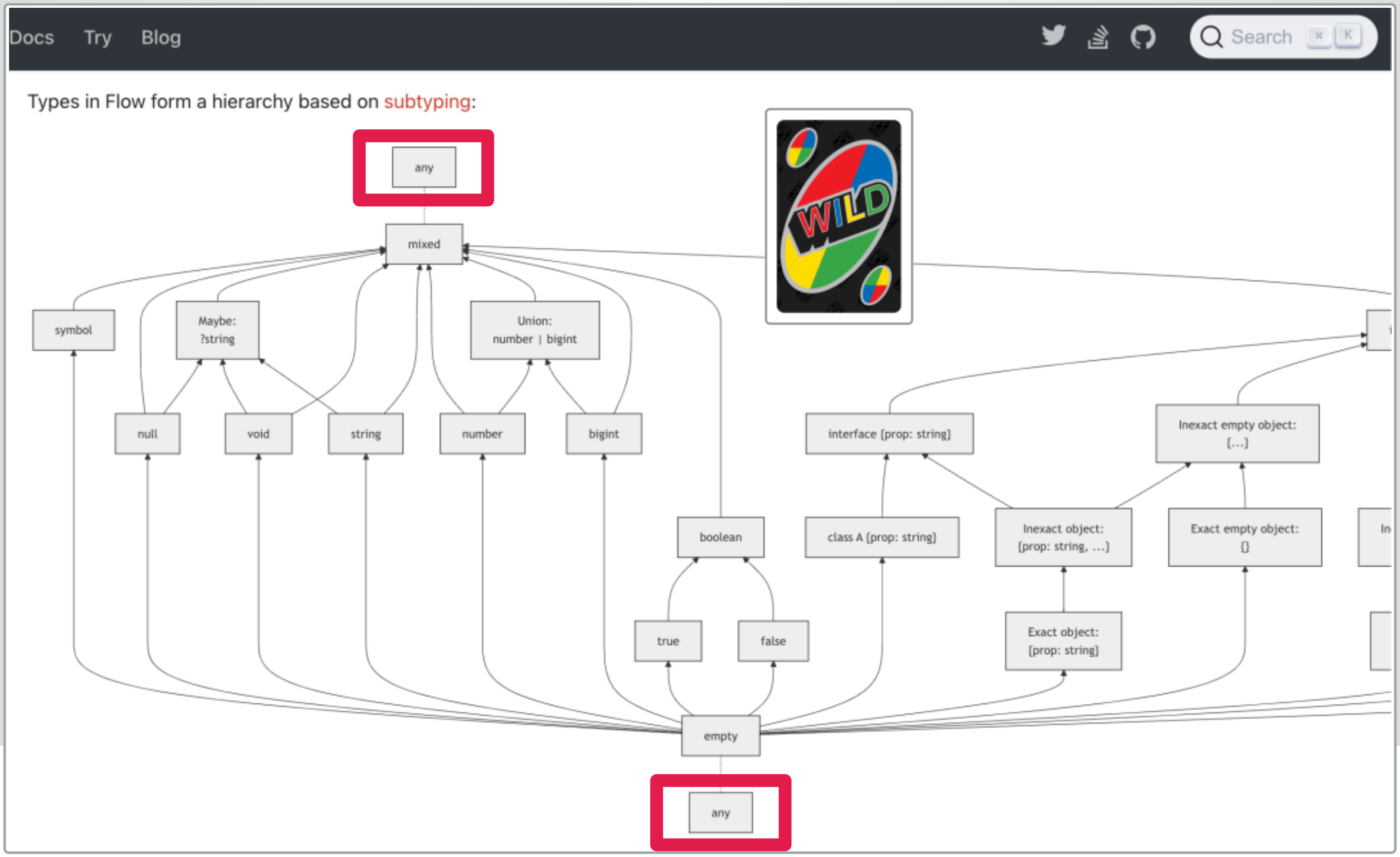
Definitely Typed

34 million clients!



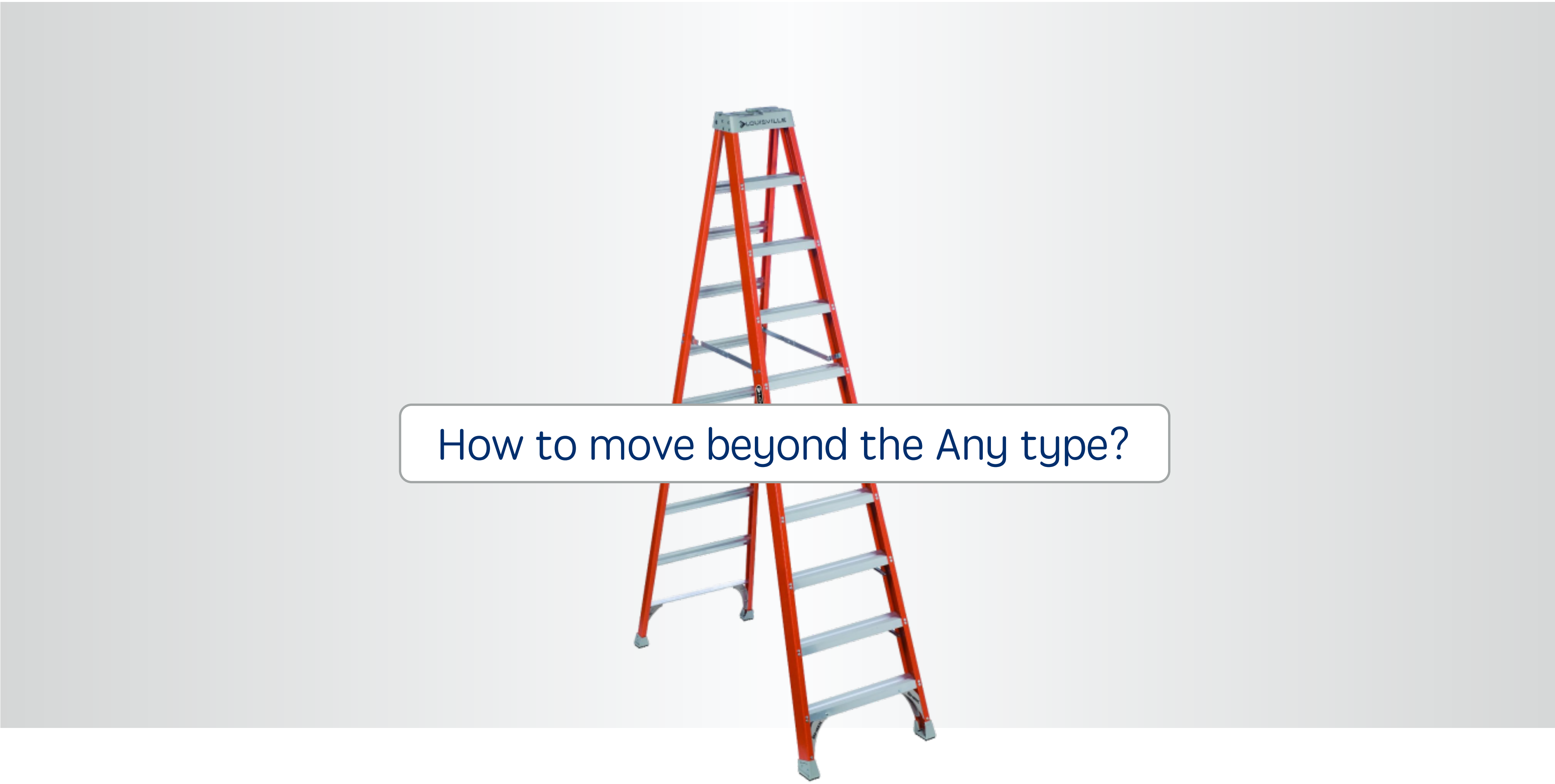
Problem:

Gradual Types << Types


Docs
Try
Blog
Search
Types in Flow form a hierarchy based on subtyping:
any
mixed
symbol
Maybe:
?string
Union:
number | bigint
null
void
string
number
bigint
boolean
true
false
empty
any
interface {prop: string}
class A {prop: string}
Inexact empty object:
{...}
Inexact object:
{prop: string, ...}
Exact empty object:
{}
Exact object:
{prop: string}
WILD

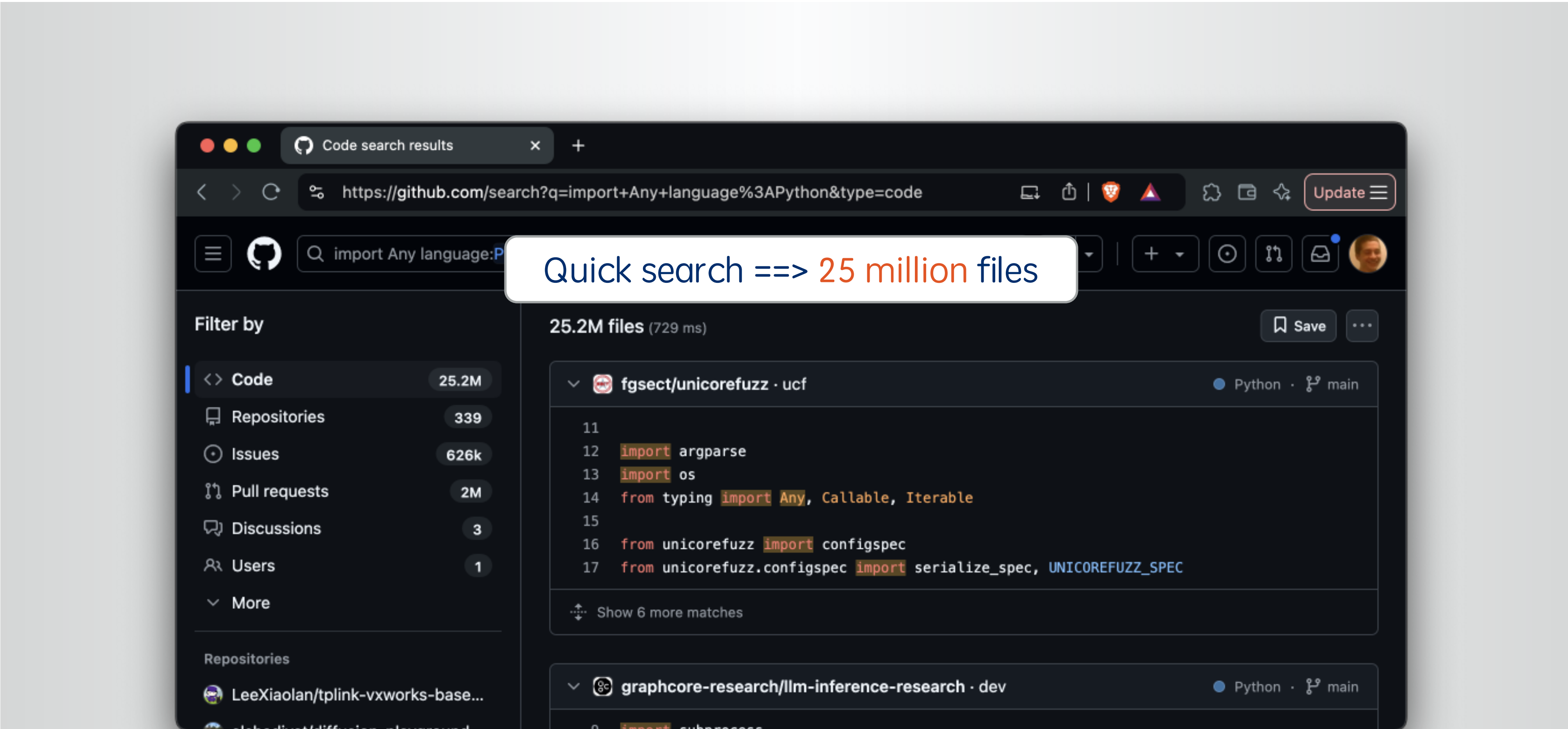
```python
def div(a: int, b: int) -> Any:
  return a / b if b != 0 else "div0"

div(x, y).helloworld()
# No type error!!!
```

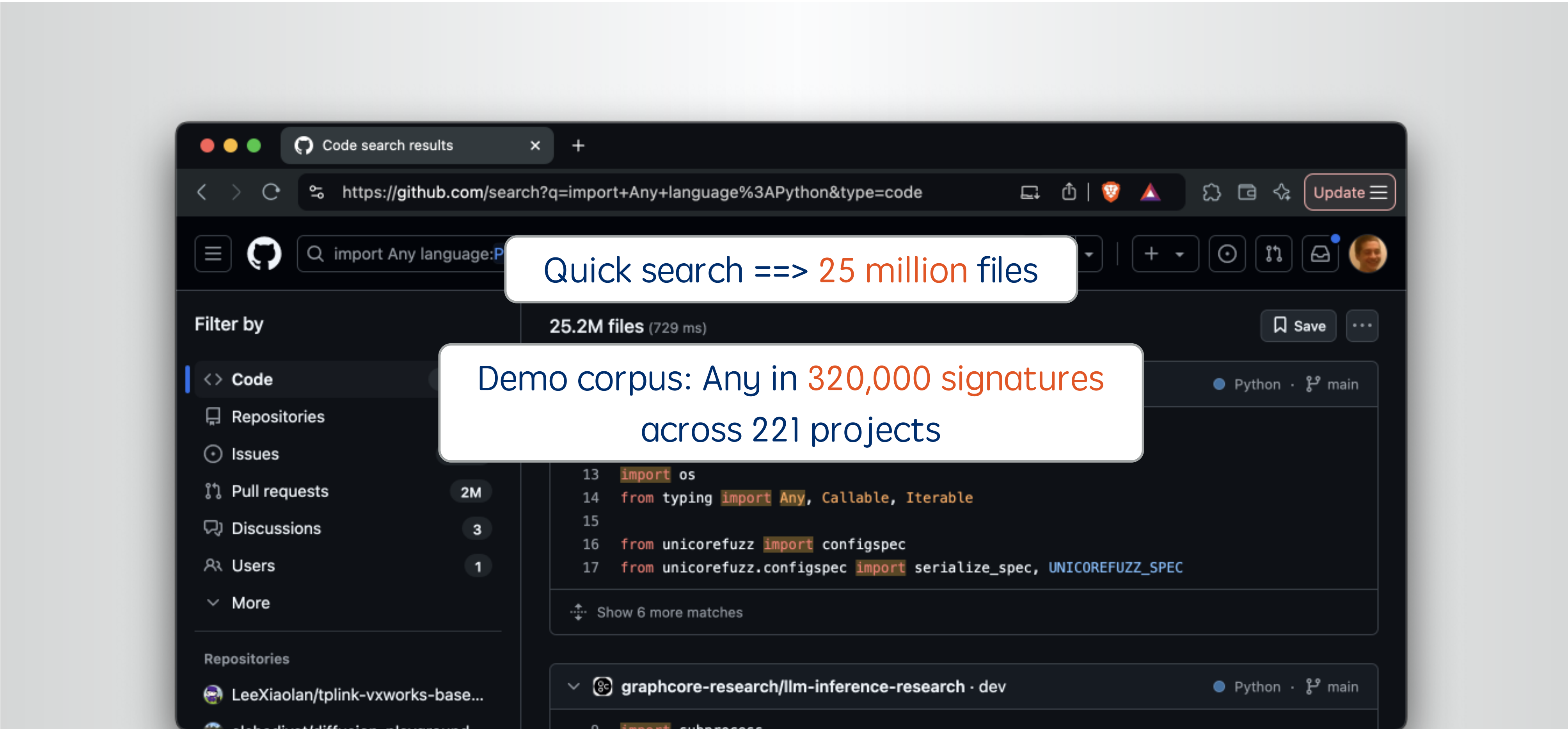
```
def div            y:
  retur            "div0"

div(x,
# No ty
```



How to move beyond the Any type?

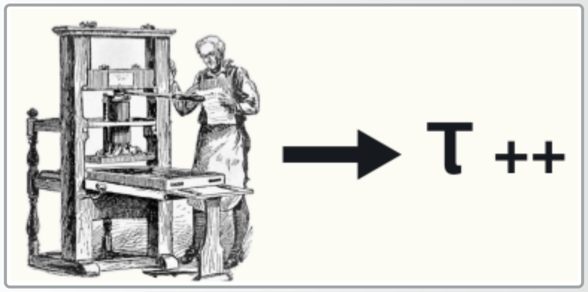

Any
is an opportunity


Code search results
https://github.com/search?q=import+Any+language%3APython&type=code
import Any language:P
Quick search ==> 25 million files
Filter by
Code 25.2M
Repositories 339
Issues 626k
Pull requests 2M
Discussions 3
Users 1
More
Repositories
LeeXiaolan/tplink-vxworks-base...
25.2M files (729 ms)
Save
fgsect/unicorefuzz · ucf
Python · main
11
12 import argparse
13 import os
14 from typing import Any, Callable, Iterable
15
16 from unicorefuzz import configspec
17 from unicorefuzz.configspec import serialize_spec, UNICOREFUZZ_SPEC
Show 6 more matches
graphcore-research/llm-inference-research · dev
Python · main

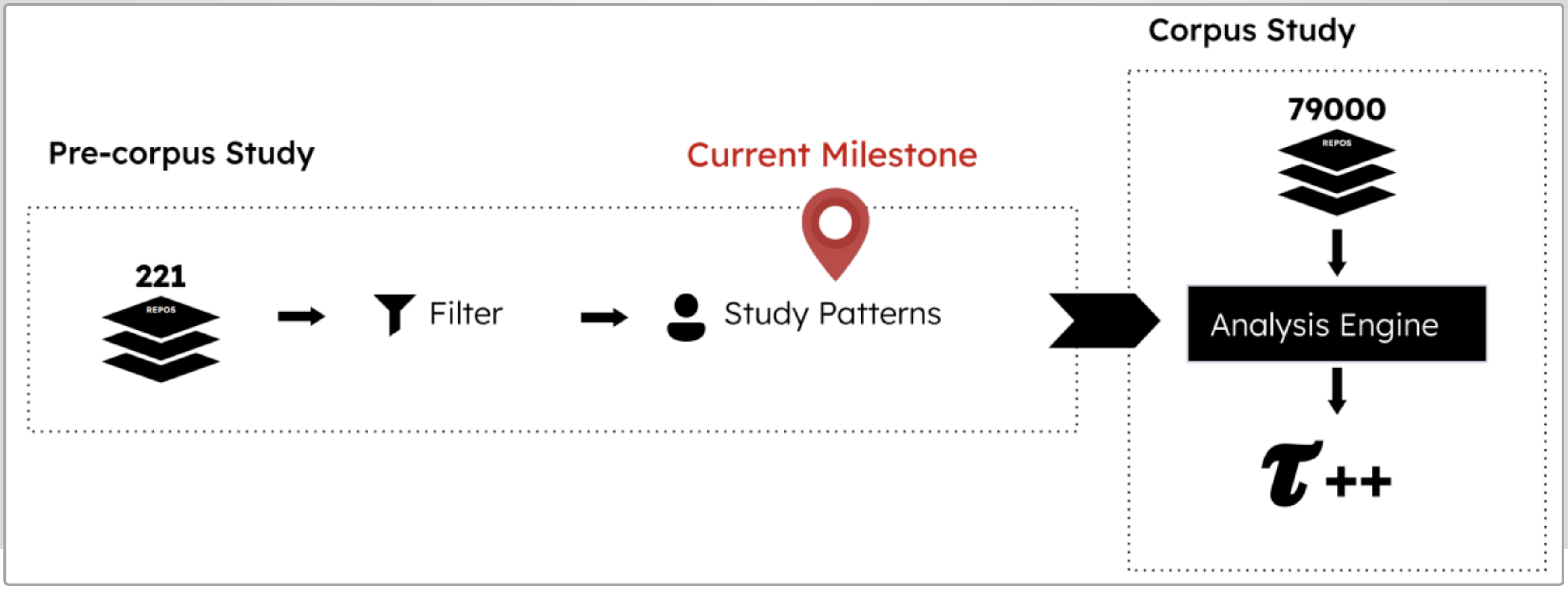

Code search results
https://github.com/search?q=import+Any+language%3APython&type=code
Update
import Any language:P
Quick search ==> 25 million files
Filter by
Code
Repositories
Issues
Pull requests 2M
Discussions 3
Users 1
More
Repositories
LeeXiaolan/tplink-vxworks-base...
25.2M files (729 ms)
Save
Demo corpus: Any in 320,000 signatures across 221 projects
Python · main
13 import os
14 from typing import Any, Callable, Iterable
15
16 from unicorefuzz import configspec
17 from unicorefuzz.configspec import serialize_spec, UNICOREFUZZ_SPEC
Show 6 more matches
graphcore-research/llm-inference-research · dev
Python · main

Goal: Data Science for the Any Type

+ recognize patterns
+ set priorities
+ improve type precision



# Roadmap


Corpus Study
Pre-corpus Study
Current Milestone
221
REPOS
Filter
Study Patterns
79000
REPOS
Analysis Engine
τ++

6 Major Patterns

T-Var

U-Var

Self

Dep-Dict

Wrap

Override

T-Var

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

U-Var

```python
Car = TypeVar('Car') # unconstrained var

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
```

Self

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

class Circle(Shape):
  pass

Circle().move(4) # ideally, a Circle
```

Dep-Dict

```python
def add_tax(item: Dict[str, Any]) -> float:
  base = item.get("price", 0) # Any
  return base + (base * 0.10)
```

Wrap

```python
def outer(fn: Callable) -> Callable: # imprecise
  def inner(self, s1: str, *args, **kwargs) -> Callable:
    s2 = string_to_stat(s1)
    return fn(self, s2, *args, **kwargs)
  return string_fn
```

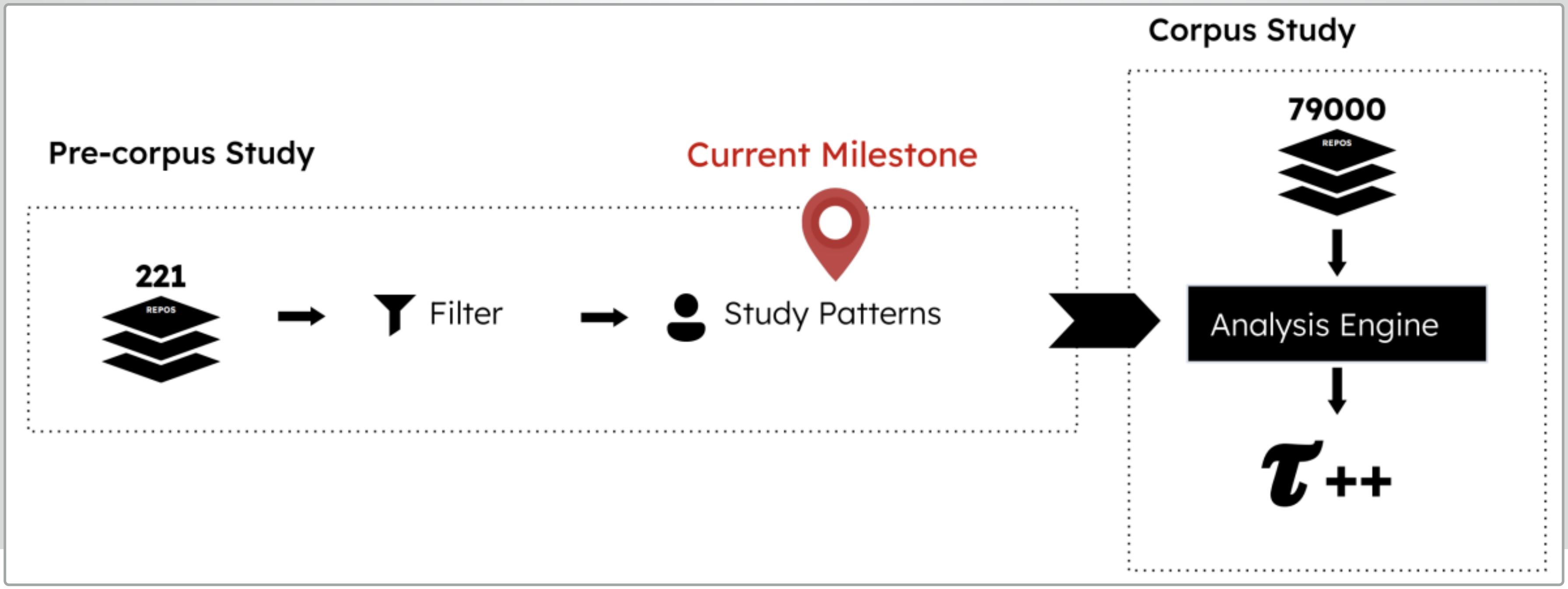
## Override

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

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

## Automation, Roughly

Clean AST → `pip install` → Run 1 → Edit → Run 2 → Line #s

# Roadmap


Corpus Study
Pre-corpus Study
Current Milestone
221
REPOS
Filter
Study Patterns
79000
REPOS
Analysis Engine
τ++

Challenge:

Typing Control Flow

```
if x is an Int:
  x + 1
```

## Rainfall Problem

Compute average rainfall from a list of possibly-faulty weather reports.

Ignore data that is non-numeric, too high (>999), or too low (<0).

## Rainfall Problem

Compute average rainfall from a list of possibly-faulty weather reports.

Ignore data that is non-numeric, too high (>999), or too low (<0).

```
def rainfall(data : List[JSON]) -> Number:
  let total = 0, count = 0
  for report in data:
    if report is Object:
      let val = report["rainfall"]
      if val is Number and 0 <= val <= 999:
        total += val
        count += 1
  return total / count  # assuming count >0
```

```
def rainfall(data : List[JSON]) -> Number:
  let total = 0, count = 0
  for report in data:
    if report is Object:
      let val = report["rainfall"]
      if val is Number and 0 <= val <= 999:
        total += val
        count += 1
  return total / count
```

Type Tests

Aliasing!

Data Structure

## Type Narrowing

```
def fst(a : tuple[object, object]) -> int:
  if type(a[0]) is int:
    return a[0] + 1
```

```
def filter_nums(bs: list[object]) -> list[int]:
  return [b for b in bs if type(b) is int]
```

```
class Node:
  parent : Node | None
  ...

def score(n: Node) -> int:
  if node.parent is not None:
    return node.parent.wins + node.parent.losses
  ...
```

## Type Narrowing

```
def fst(a : tuple[object, object]) -> int:
  if type(a[0]) is int:
    return a[0] + 1
```

```
def filter_nums(bs: list[object]) -> list[int]:
  return [b for b in bs if type(b) is int]
```

```
class Node:
  parent : Node | None
  ...

def score(n: Node) -> int:
  if node.parent is not None:
    return node.parent.wins + node.parent.losses
  ...
```

# Type Narrowing

```python
def fst(a : tuple[object, object]) -> int:
  if type(a[0]) is int:
    return a[0] + 1
```

```python
def filter_nums(bs: list[object]) -> list[int]:
  return [b for b in bs if type(b) is int]
```

```python
class Node:
  parent : Node | None
  ...

def score(n: Node) -> int:
  if node.parent is not None:
    return node.parent.wins + node.parent.losses
  ...
```

## Type Narrowing

```
def fst(a : tuple[object, object]) -> int:
  if type(a[0]) is int:
    return a[0] + 1
```

```
def filter_nums(bs: list[object]) -> list[int]:
  return [b for b in bs if type(b) is int]
```

```
class Node:
  parent : Node | None
  ...

def score(n: Node) -> int:
  if node.parent is not None:
    return node.parent.wins + node.parent.losses
  ...
```

# Design Space of Type Narrowing

**Set-Theoretic Types**

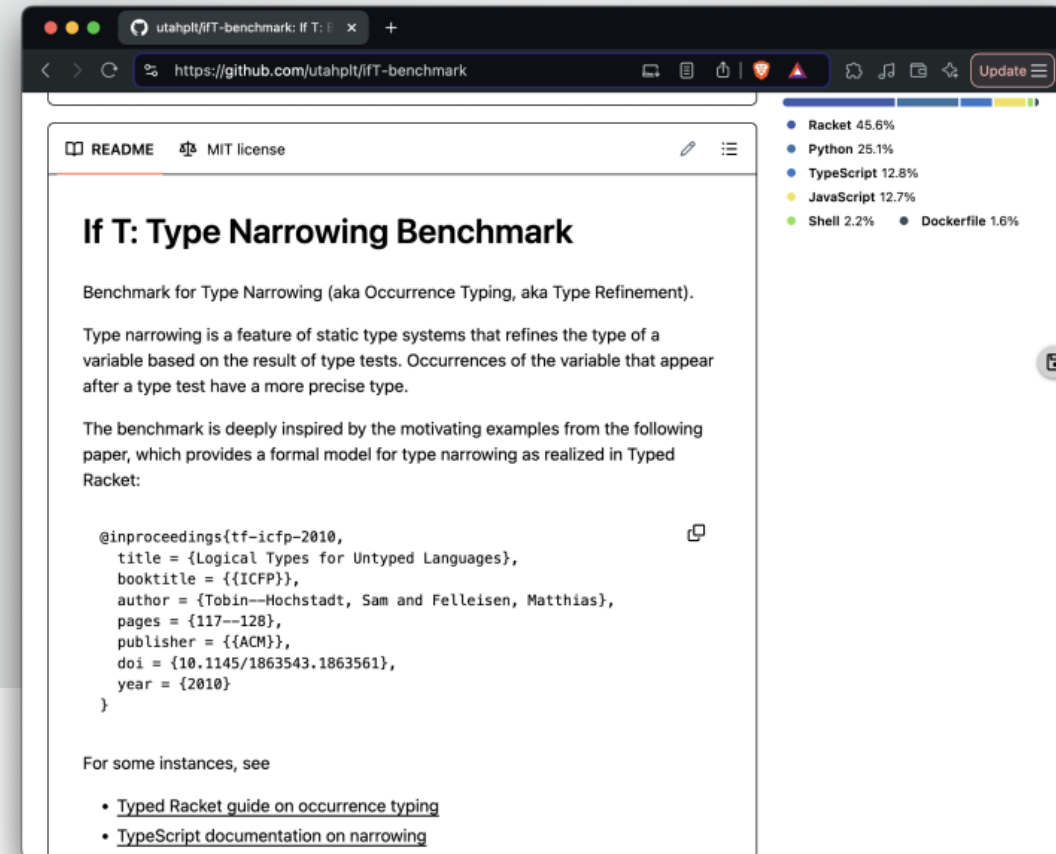
$T := T \cup T \mid T \cap T \mid \neg T$

**Occurrence Typing**

$\Gamma \vdash e : \tau ; \varphi+ \mid \varphi- ; o$

**??**

github.com/utahplt/ift-benchmark



`github.com/utahplt/ift-benchmark`

+ 4 core dimensions
    + 13 benchmark items
    + pass & fail tests

+ practical examples
+ datasheet

| | | |
|---|---|---|
| | Basic Narrowing: | |
| 1. | positive | Refine when condition is true |
| 2. | negative | Refine when condition is false |
| 3. | connectives | Handle logic connectives: not, or, and |
| 4. | nesting_body | Conditionals nested within branches |
| | Compound Structures: | |
| 5. | struct_fields | Refine (immutable) structure fields |
| 6. | tuple_elements | Refine tuple elements |
| 7. | tuple_length | Refine based on tuple size |
| | Advanced Control Flow: | |
| 8. | alias | Track logical implication through variables |
| 9. | nesting_condition | Conditionals nested within conditions |
| 10. | merge_with_union | Correctly merge control-flow branches |
| | Custom Predicates: | |
| 11. | predicate_2way | Custom predicates that narrow positively and negatively |
| 12. | predicate_1way | Custom predicates that narrow only positively |
| 13. | predicate_checked | Typecheck the body of custom predicates |

| | | |
|---|---|---|
| | Basic Narrowing: | |
| 1. | positive | Refine when condition is true |
| 2. | negative | Refine when condition is false |
| 3. | connectives | Handle logic connectives: not, or, and |
| 4. | nesting_body | Conditionals nested within branches |
| | Compound Struc | |
| 5. | struct_fields | ls |
| 6. | tuple_elements | |
| 7. | tuple_length | |
| | Advanced Contr | |
| 8. | alias | variables |
| 9. | nesting_condition | Conditionals nested within conditions |
| 10. | merge_with_union | Correctly merge control-flow branches |
| | Custom Predicates: | |
| 11. | predicate_2way | Custom predicates that narrow positively and negatively |
| 12. | predicate_1way | Custom predicates that narrow only positively |
| 13. | predicate_checked | Typecheck the body of custom predicates |

Intentionally left out:

- Subtyping
- Mutation
- Concurrency

# If-T Pseudocode

**nesting_condition: Success**

```
define f(x: Top, y: Top) -> Number:
  if (if x is Number: y is String else: false):
    return x + String.length(y)
  else:
    return 0
```

**nesting_condition: Failure**

```
define f(x: Top, y: Top) -> Number:
  if (if x is Number: y is String else: y is String):
    return x + String.length(y)
  else:
    return 0
```

**Table 2** Benchmark Results: ● = passed, × = failed imprecisely, [×] = failed unsoundly

| | | Typed Racket | TypeScript | Flow | Mypy | Pyright |
|---|---|---|---|---|---|---|
| | Basic Narrowing: | | | | | |
| 1. | positive | ● | ● | ● | ● | ● |
| 2. | negative | ● | ● | ● | ● | ● |
| 3. | connectives | ● | ● | ● | ● | ● |
| 4. | nesting_body | ● | ● | ● | ● | ● |
| | Compound Structures: | | | | | |
| 5. | struct_fields | ● | ● | ● | ● | ● |
| 6. | tuple_elements | ● | ● | ● | ● | ● |
| 7. | tuple_length | × | ● | ● | ● | ● |
| | Advanced Control Flow: | | | | | |
| 8. | alias | ● | ● | × | × | ● |
| 9. | nesting_condition | ● | × | × | × | × |
| 10. | merge_with_union | ● | ● | ● | × | ● |
| | Custom Predicates: | | | | | |
| 11. | predicate_2way | ● | ● | ● | ● | ● |
| 12. | predicate_1way | ● | × | ● | ● | ● |
| 13. | predicate_checked | ● | [×] | ● | [×] | [×] |

# Custom Predicates

```
function is_num(x: any)
 : x is number
```

```
def is_num(x) -> TypeIs(int)
```

```
function is_even(x: any)
 : implies x is number
 // Flow, not TypeScript
```

```
def is_even(x) -> TypeGuard(int)
```

# Custom Predicates

```
opt-proposition =
                | : type
                | : pos-proposition
                    neg-proposition
                  object

pos-proposition =
                | #:+ proposition ...

neg-proposition =
                | #:- proposition ...

         object =
                | #:object index
```

```
proposition = Top
            | Bot
            | type
            | (! type)
            | (type @ path-elem ... index)
            | (! type @ path-elem ... index)
            | (and proposition ...)
            | (or proposition ...)
            | (implies proposition ...)

  path-elem = car
            | cdr

      index = positive-integer
            | (positive-integer positive-integer)
            | identifier
```

https://github.com/utahplt/ifT-benchmark/blo...

main · ifT-benchmark / TypeScript / examples.ts · Top

Code · Blame · Raw

```typescript
interface TreeNodeSuccess {
  value: number;
  children?: TreeNodeSuccess[];
  // Recursive reference to the same type
}

function isTreeNodeSuccess(node: any): node is TreeNodeSuccess {
  if (typeof node !== 'object' || node === null) {
    return false;
  }

  if (typeof node.value !== 'number') {
    return false;
  }

  if (node.children) {
    if (!Array.isArray(node.children)) {
      return false;
    }

    // Recursively check each child
    for (const child of node.children) {
      if (!isTreeNodeSuccess(child)) {
        return false;
      }
    }
  }
  return true;
}
```

**Table 2** Benchmark Results: ● = passed, × = failed imprecisely, × = failed unsoundly

| | | Typed Racket | TypeScript | Flow | Mypy | Pyright |
|---|---|---|---|---|---|---|
| | Basic Narrowing: | | | | | |
| 1. | positive | ● | ● | ● | ● | ● |
| 2. | negative | ● | ● | ● | ● | ● |
| 3. | connectives | ● | ● | ● | ● | ● |
| 4. | nesting_body | ● | ● | ● | ● | ● |
| | Compound Structures: | | | | | |
| 5. | struct_fields | ● | ● | ● | ● | ● |
| 6. | tuple_elements | ● | ● | ● | ● | ● |
| 7. | tuple_length | × | ● | ● | ● | ● |
| | Advanced Control Flow: | | | | | |
| 8. | alias | ● | ● | × | × | ● |
| 9. | nesting_condition | ● | × | × | × | × |
| 10. | merge_with_union | ● | ● | ● | × | ● |
| | Custom Predicates: | | | | | |
| 11. | predicate_2way | ● | ● | ● | ● | ● |
| 12. | predicate_1way | ● | × | ● | ● | ● |
| 13. | predicate_checked | ● | × | ● | × | × |

What about soundness?

```
function query(d : DataFrame, row_pos: Int) : Row {

    // typed function, can we optimize the body?
    // NO.

}
```

```
function query(d : DataFrame, row_pos: Int) : Row {

    // typed function, can we optimize the body?
    // NO.

}
```

```
query("hello world", 99)  ==>  ??!
```

TypeScript does not protect types.

Typed Racket does protect types ... and the cost is high

Typed Racket does protect types ... and the cost is high

OOPSLA '18

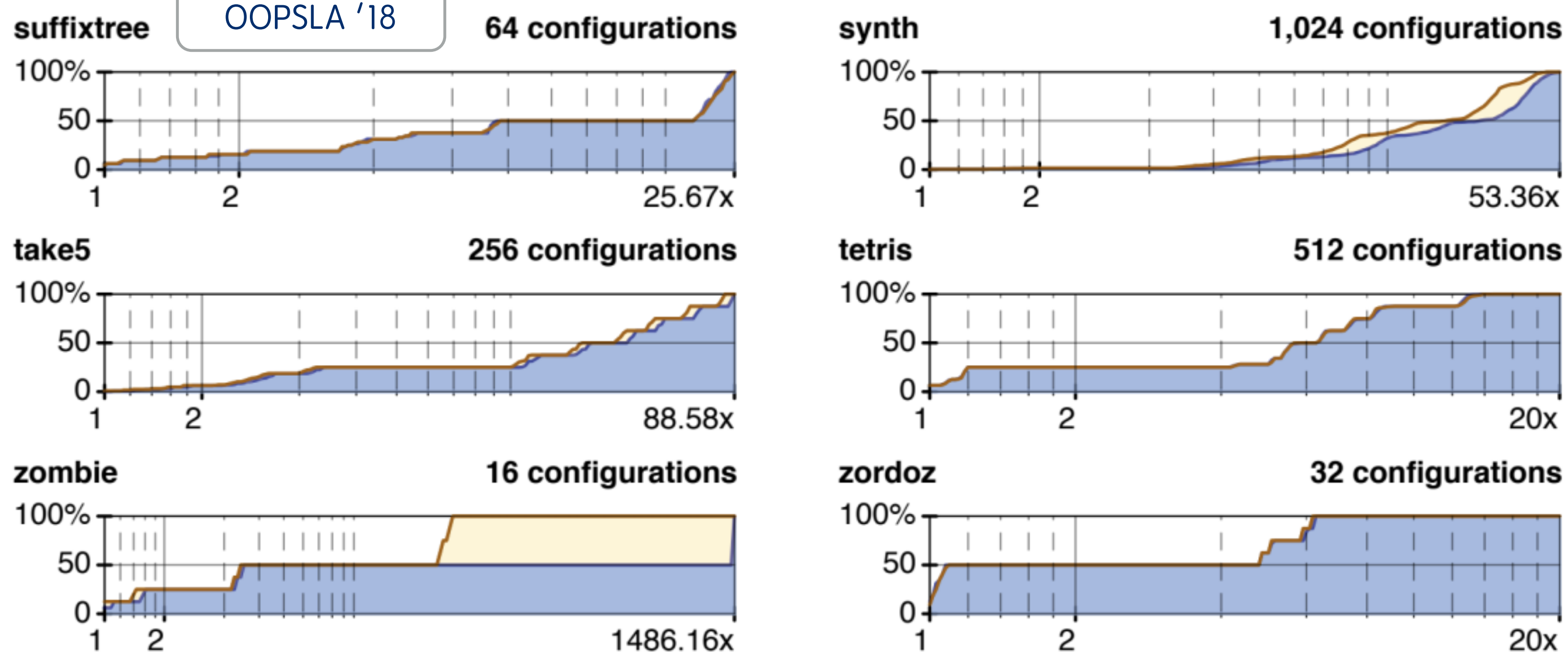
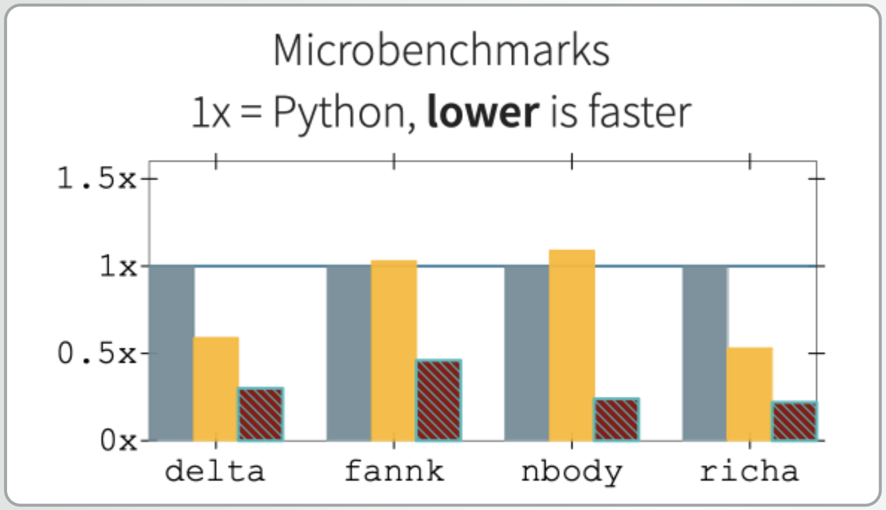



Fig. 14. Overhead of gradual typing in the benchmark programs. The blue (■) curve is for Racket 6.12 and the orange (■) curve is for collapsible contracts. A point (x,y%) on the line means y% of the configurations incur a slowdown of at most x.

Static Python is sound* and fast*



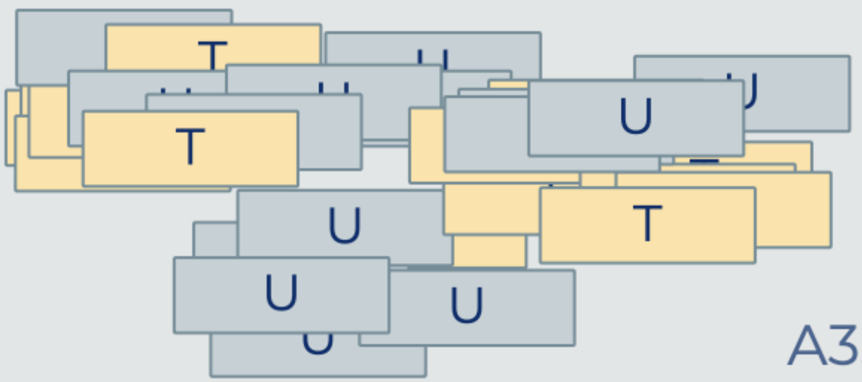

3 bars =
untyped,
shallow-typed,
concrete-typed

Types ==> faster!

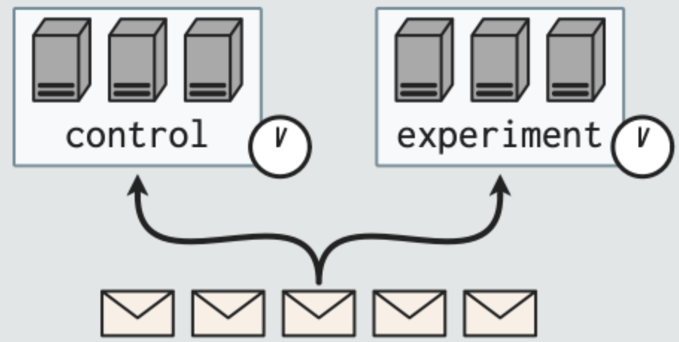
# Experience @ Instagram Web Server

+500 modules with **sound types** (upgraded from Pyre)

+30k T — U interactions




**3.9% increase** in CPU efficiency




A3. Progressive static types + tags

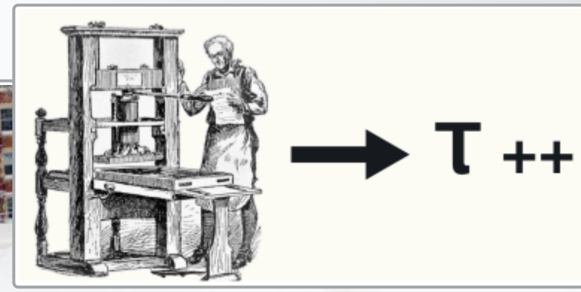
Whence gradual types?

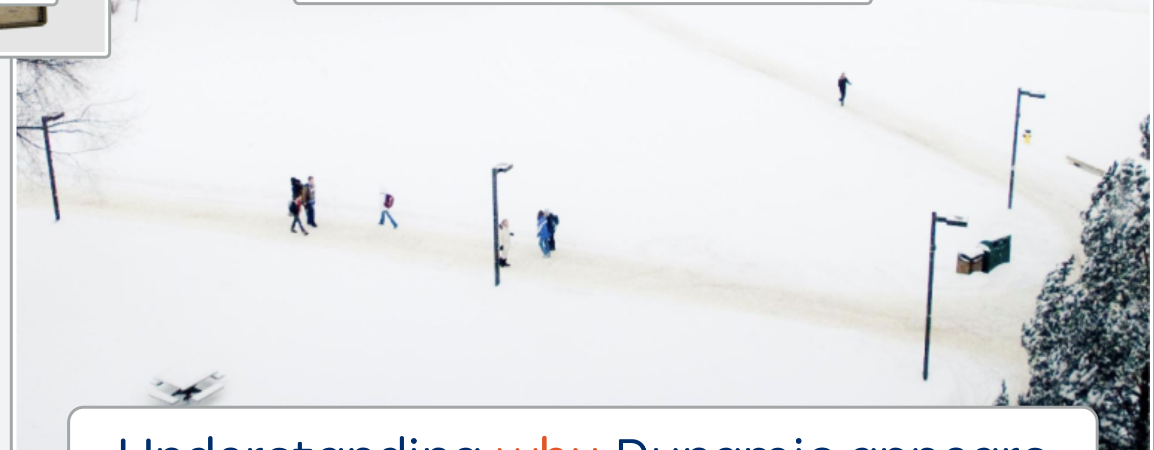
Whence gradual types?

Two leading strategies:


1. +Dynamic

2. Wizardry

Understanding why Dynamic appears


Metrics for fair cross-language comparison

→ $\tau$ ++

UTAH PLT ifT-benchmark