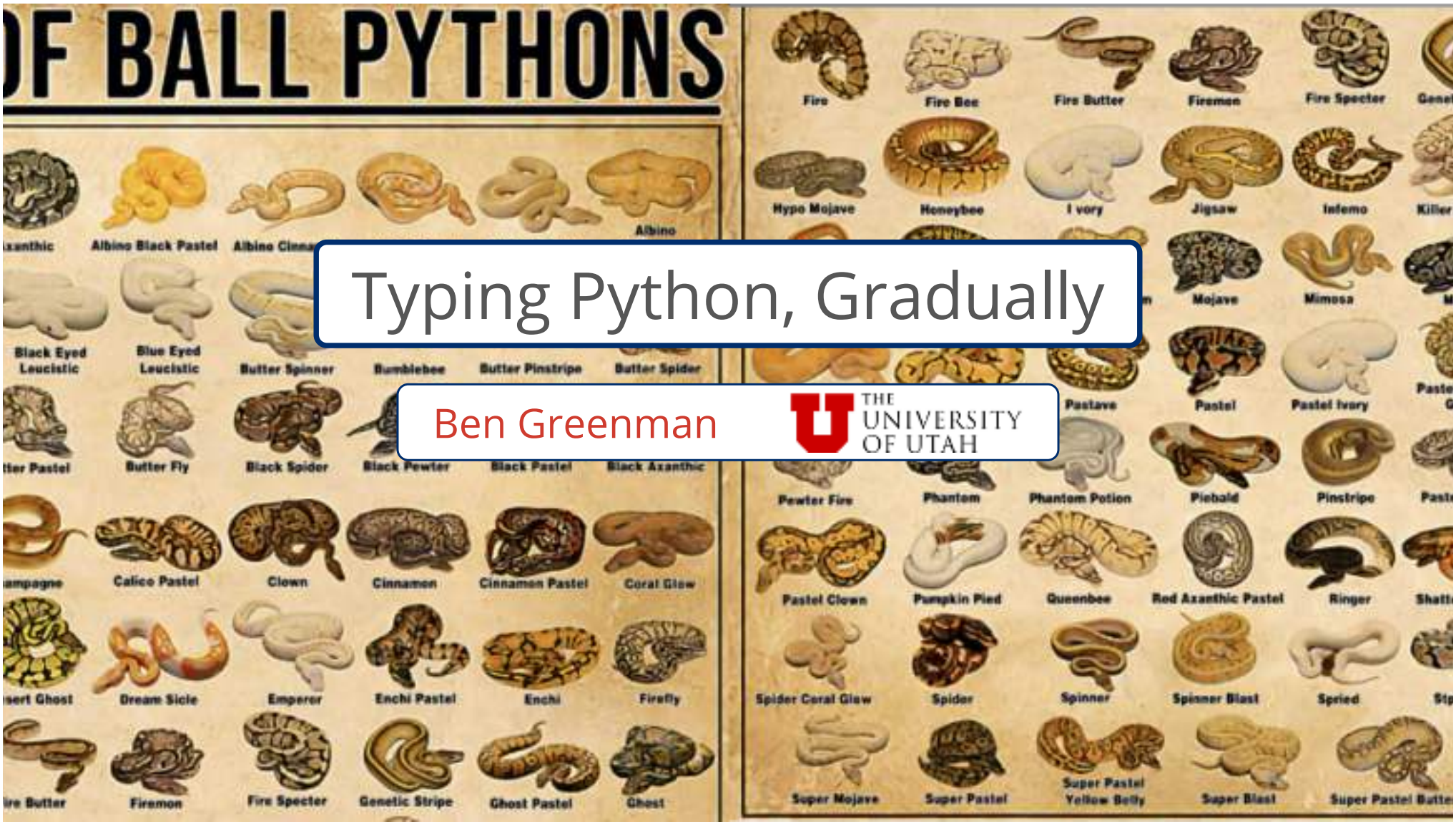
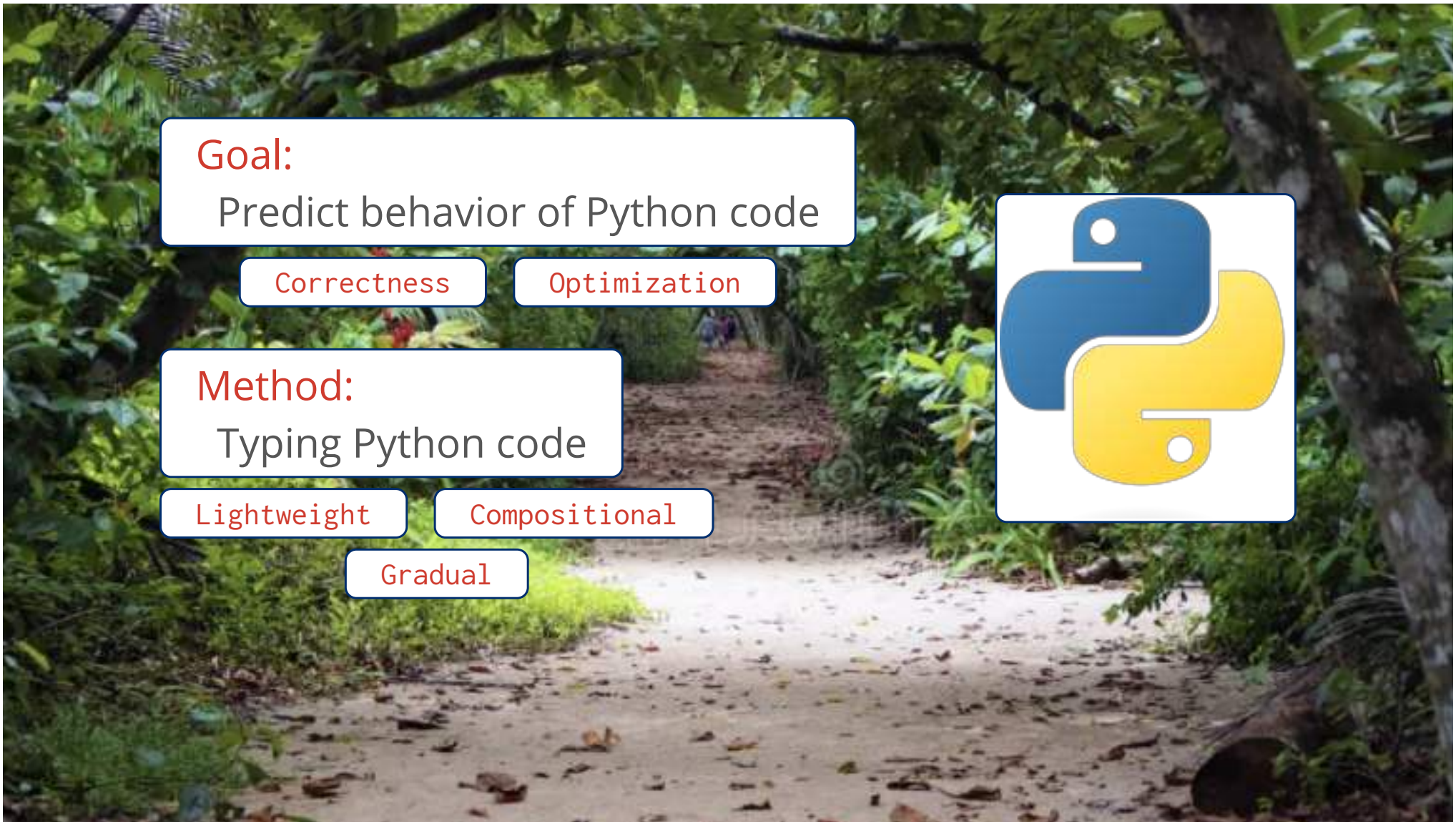
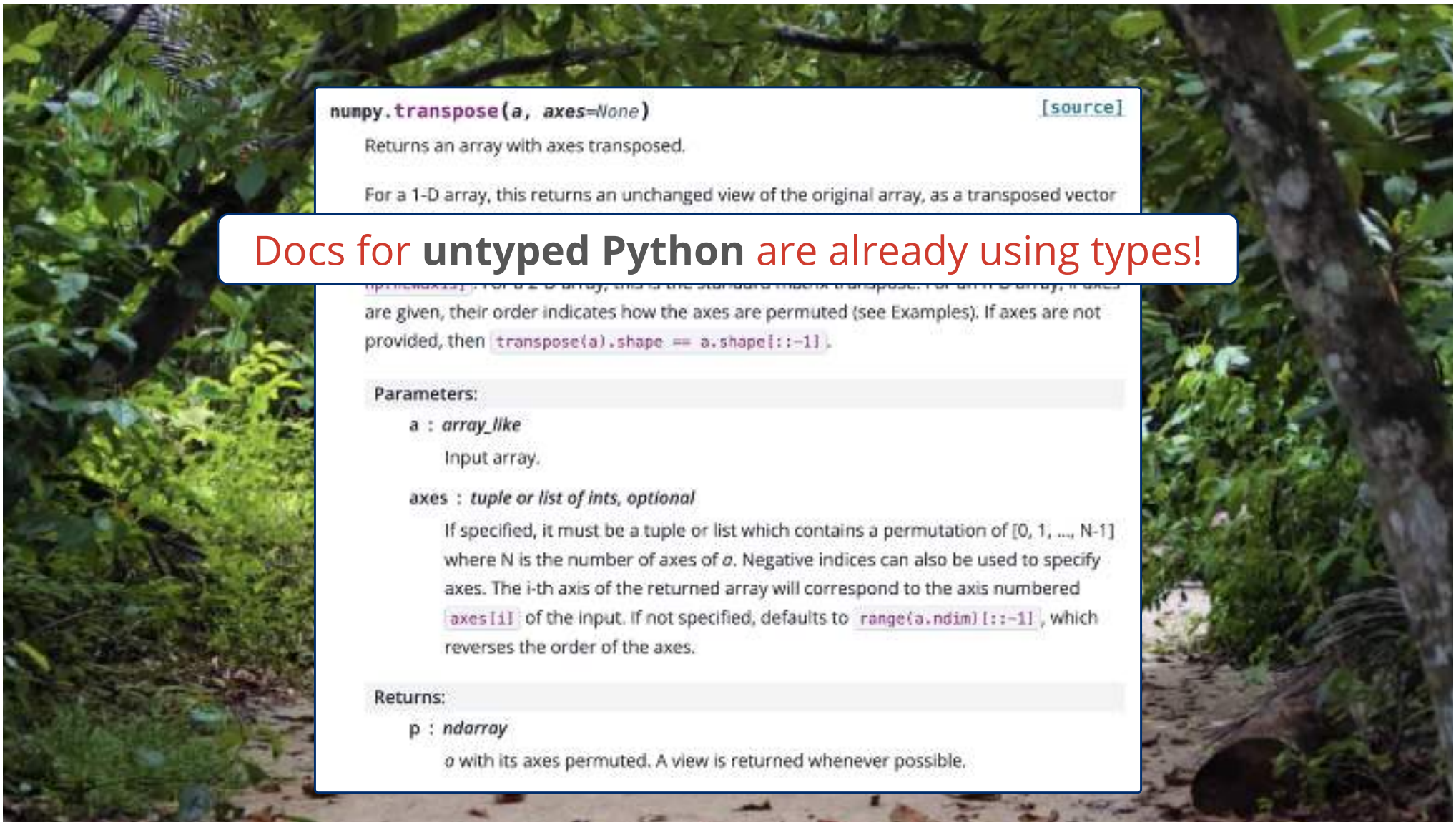


Typing Python, Gradually

Ben Greenman

THE UNIVERSITY OF UTAH


Goal:
Predict behavior of Python code
Correctness
Optimization
Method:
Typing Python code
Lightweight
Compositional
Gradual


numpy.transpose(a, axes=None) [source]
Returns an array with axes transposed.
For a 1-D array, this returns an unchanged view of the original array, as a transposed vector
Docs for untyped Python are already using types!
are given, their order indicates how the axes are permuted (see Examples). If axes are not provided, then transpose(a).shape == a.shape[::-1].
Parameters:
a : array_like
Input array.
axes : tuple or list of ints, optional
If specified, it must be a tuple or list which contains a permutation of [0, 1, ..., N-1] where N is the number of axes of a. Negative indices can also be used to specify axes. The i-th axis of the returned array will correspond to the axis numbered axes[i] of the input. If not specified, defaults to range(a.ndim)[::-1], which reverses the order of the axes.
Returns:
p : ndarray
a with its axes permuted. A view is returned whenever possible.

Even **Guido** agrees types are helpful!

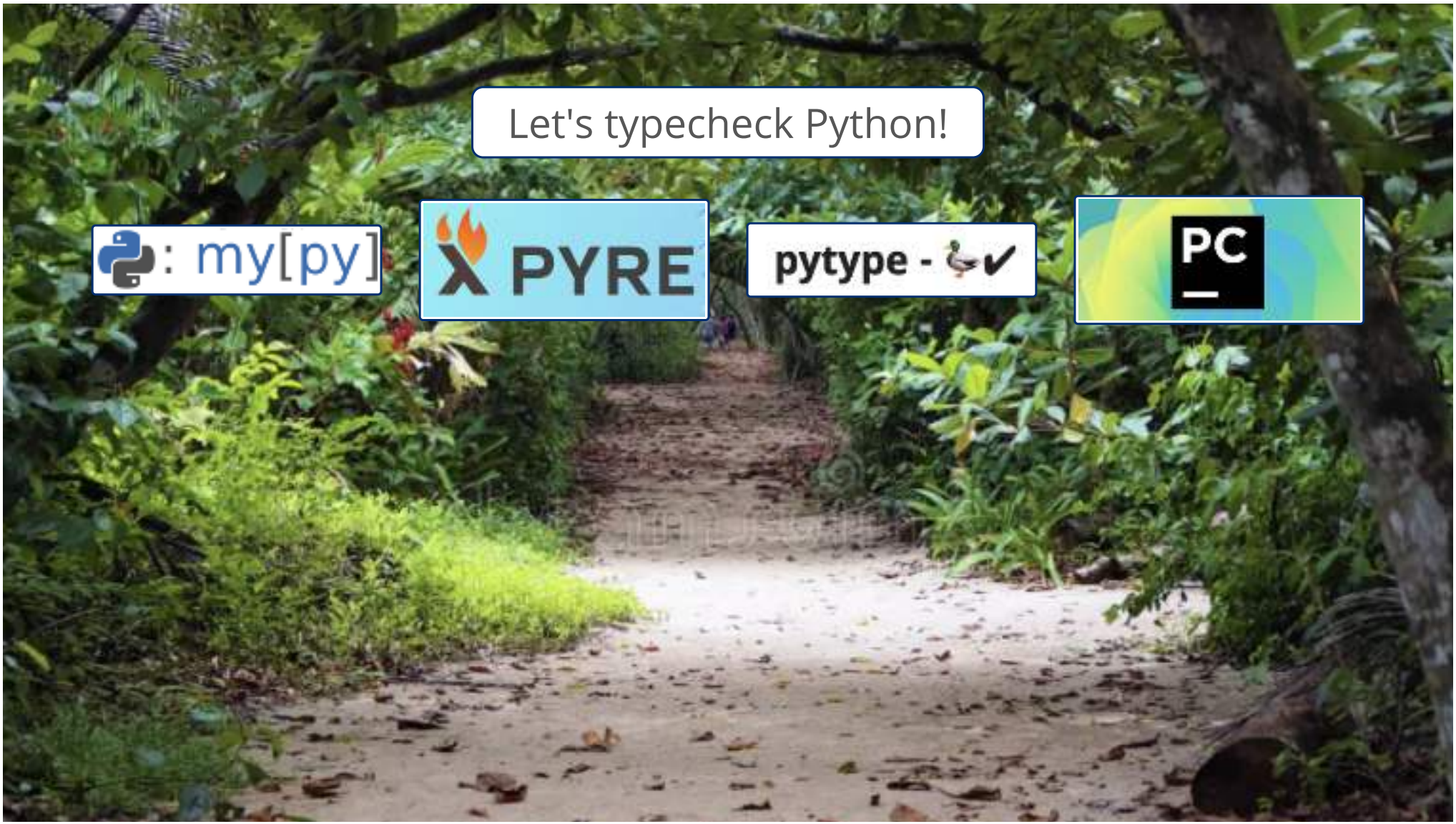
"For **really large codebases**, static languages* have their uses"

* (despite all their visual overhead and compilation cycles and build tools)

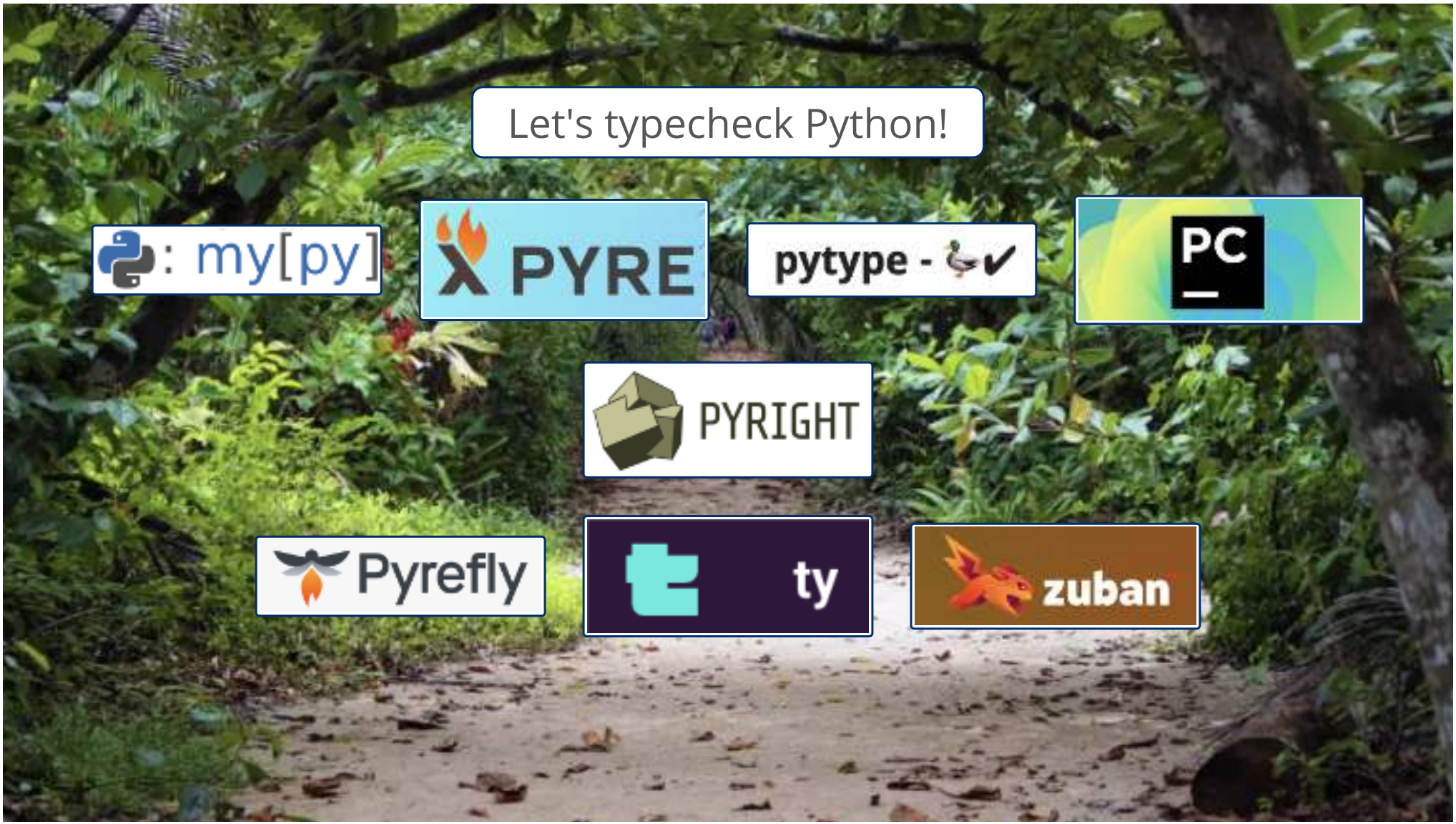
PEP 484 – Type Hints

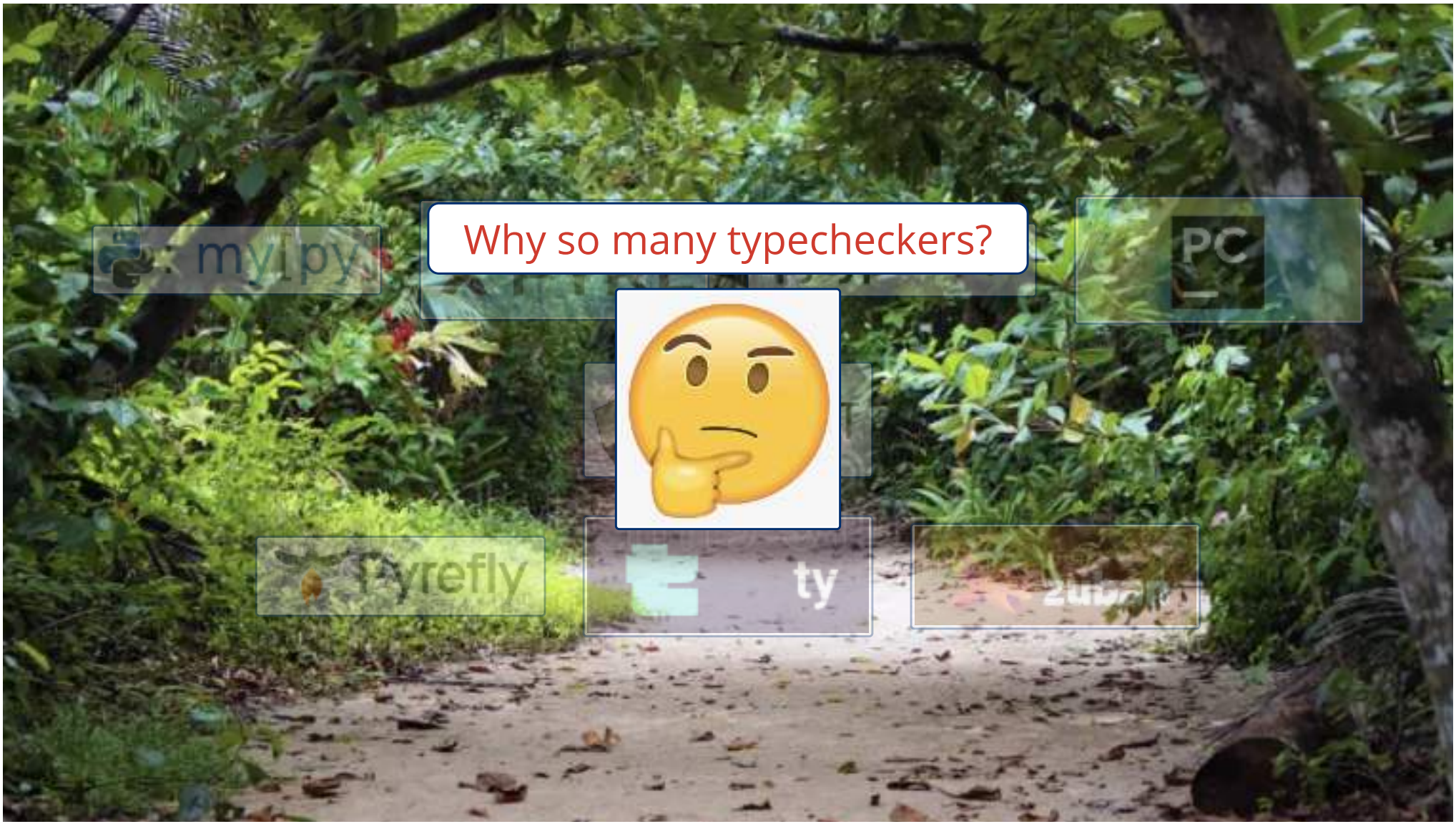
Author:	Guido van Rossum <guido at python.org>, Jukka Lehtosalo <jukka.lehtosalo at iki.fi>, Łukasz Langa <lukasz at python.org>
BDFL-Delegate:	Mark Shannon
Discussions-To:	Python-Dev list
Status:	Final
Type:	Standards Track
Topic:	Typing
Created:	29-Sep-2014
Python-Version:	3.5
Post-History:	16-Jan-2015, 20-Mar-2015, 17-Apr-2015, 20-May-2015, 22-May-2015
Resolution:	Python-Dev message

▸ Table of Contents


Let's typecheck Python!
: my[py]
PYRE
pytype -
PC


Let's typecheck Python!
: my[py]
PYRE
pytype -
PC
PYRIGHT


Let's typecheck Python!
: my[py]
PYRE
pytype -
PC
PYRIGHT
Pyrefly
ty
zuban


Why so many typecheckers?
mypy
PC
pyrefly
ty

DLS'20

Python 3 Types in the Wild: A Tale of Two Type Systems

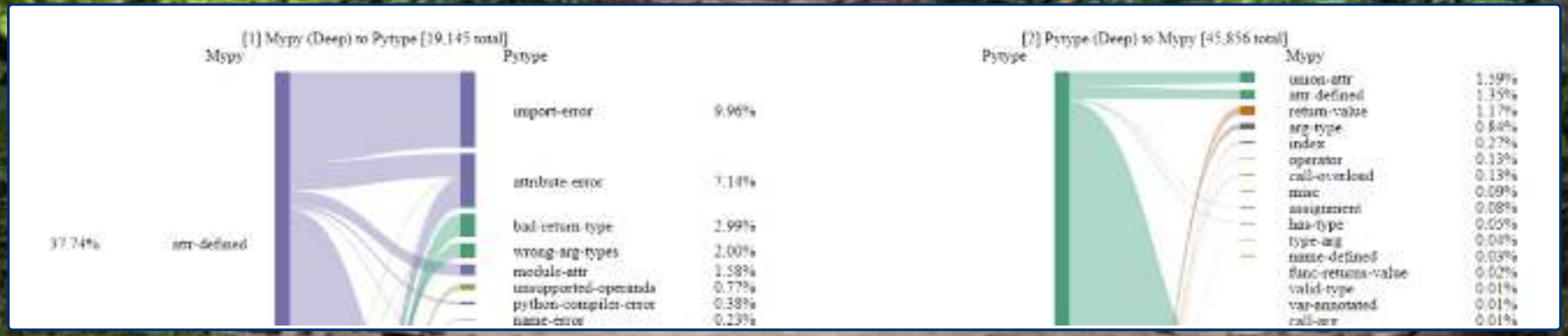
Ingkarat Rak-amnouykit
Rensselaer Polytechnic Institute
New York, USA
rakami@rpi.edu

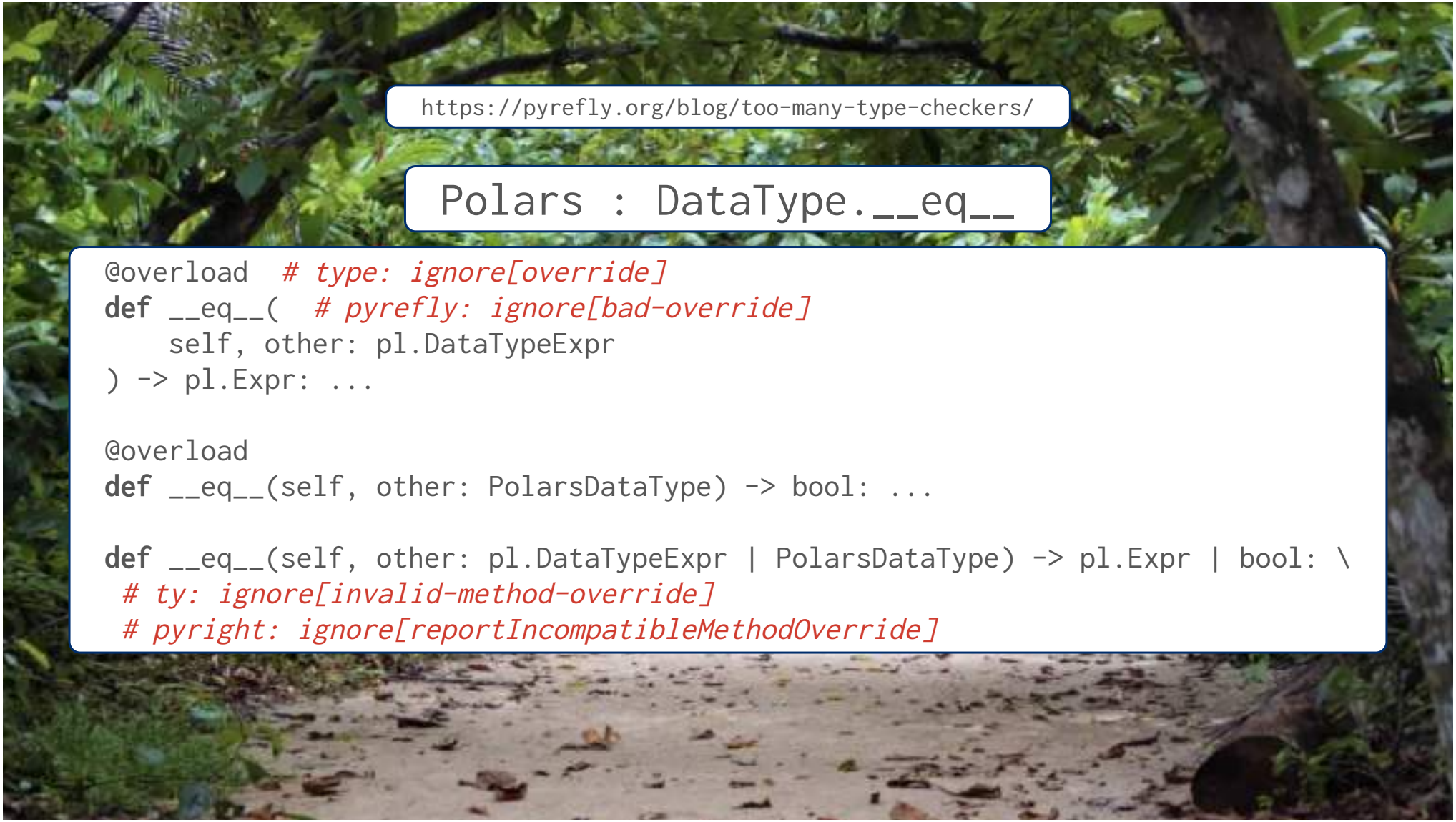
Daniel McCrevan
Rensselaer Polytechnic Institute
New York, USA
mccred@rpi.edu

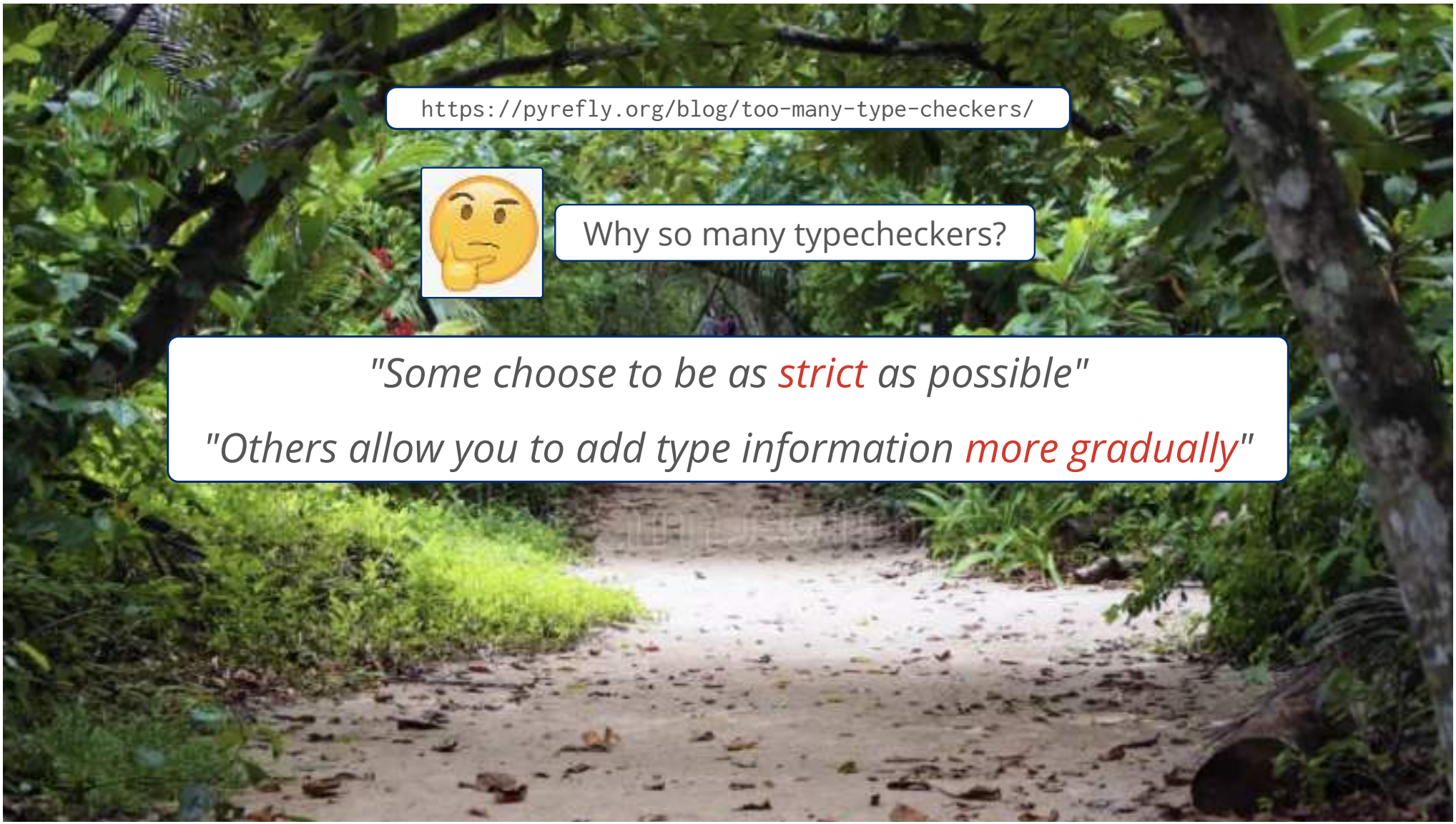
Ana Milanova
Rensselaer Polytechnic Institute
New York, USA
milanova@cs.rpi.edu

Martin Hirzel
IBM TJ Watson Research Center
New York, USA
hirzel@us.ibm.com

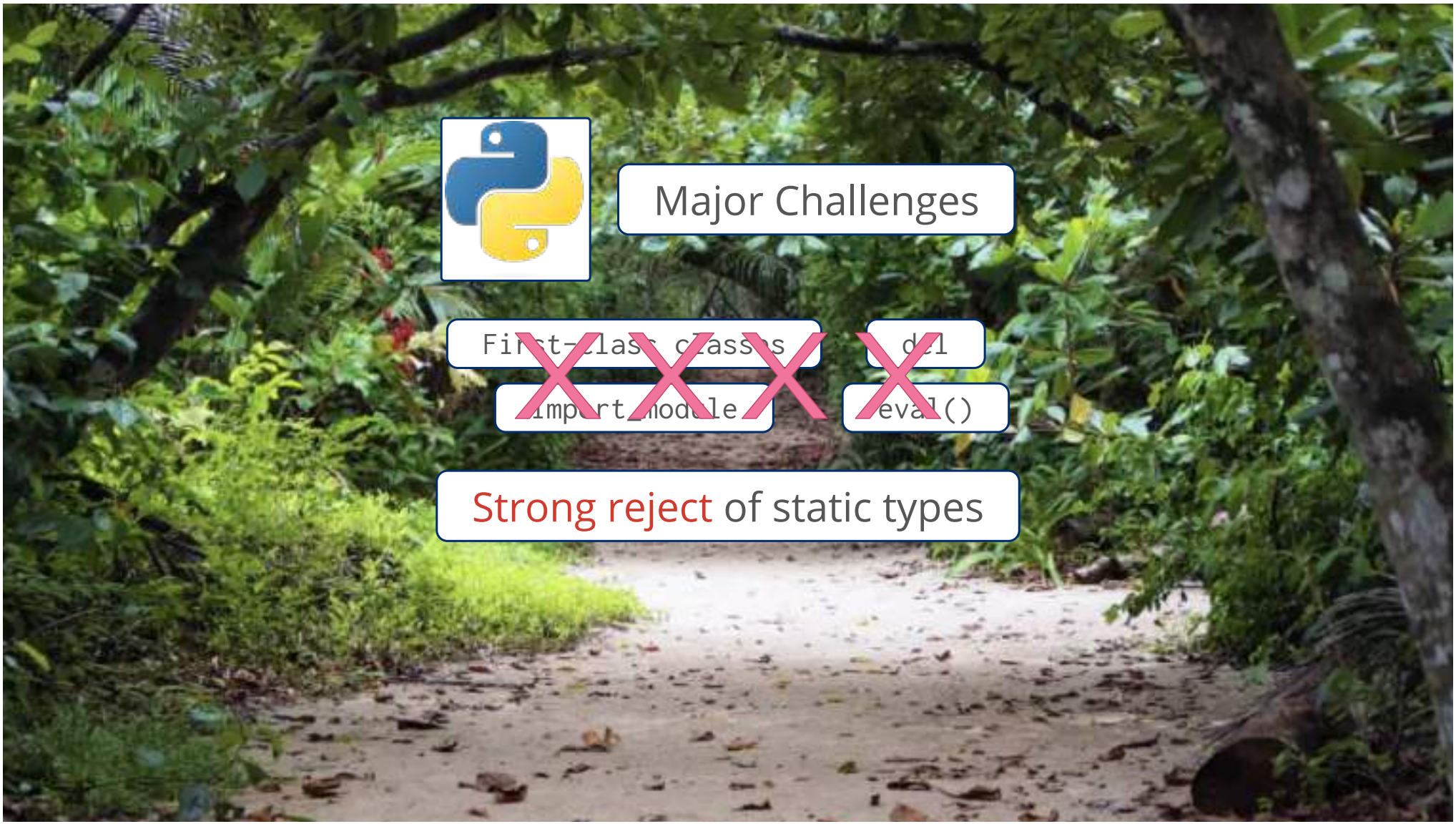
Julian Dolby
IBM TJ Watson Research Center
New York, USA
dolby@us.ibm.com

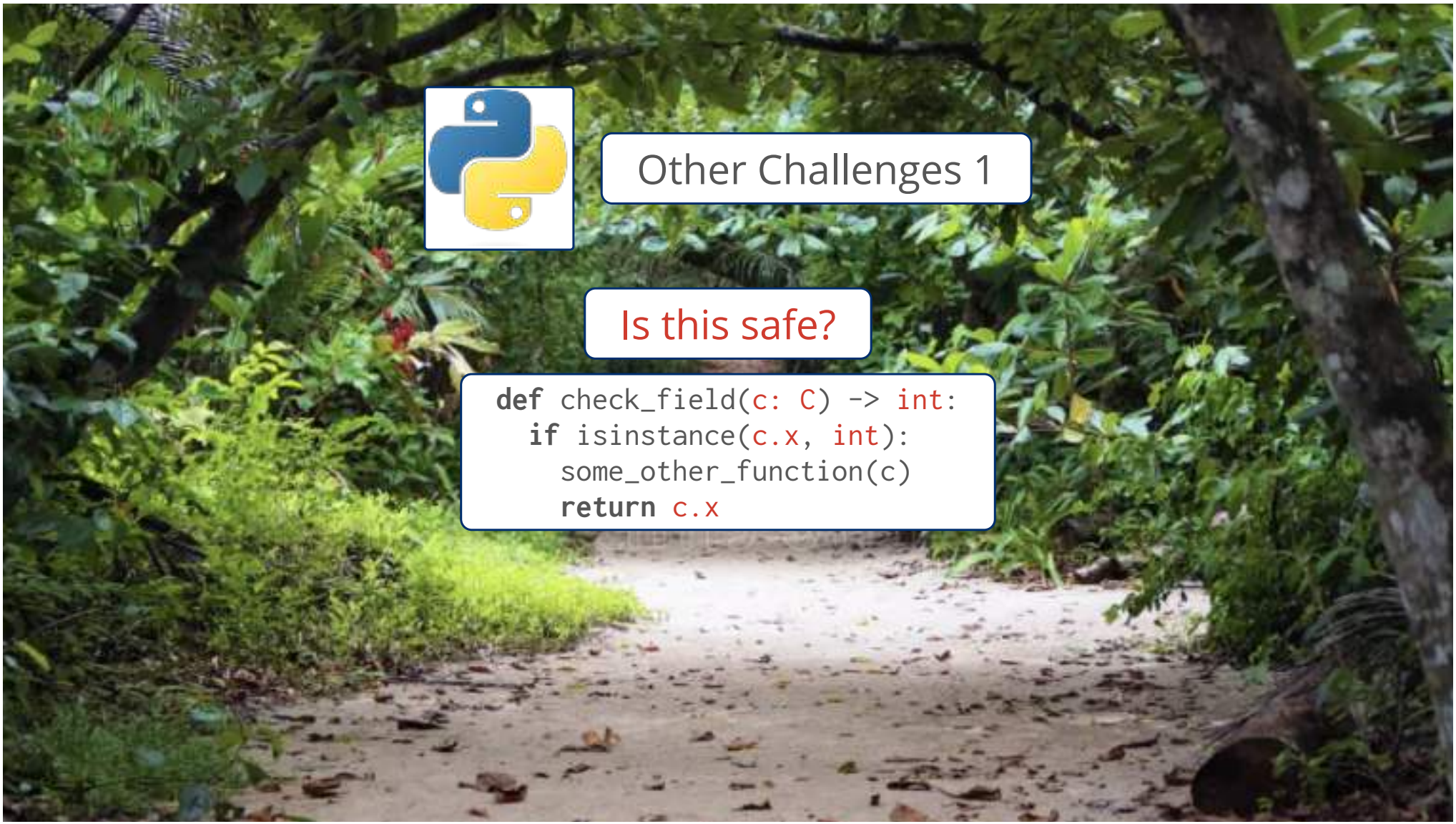


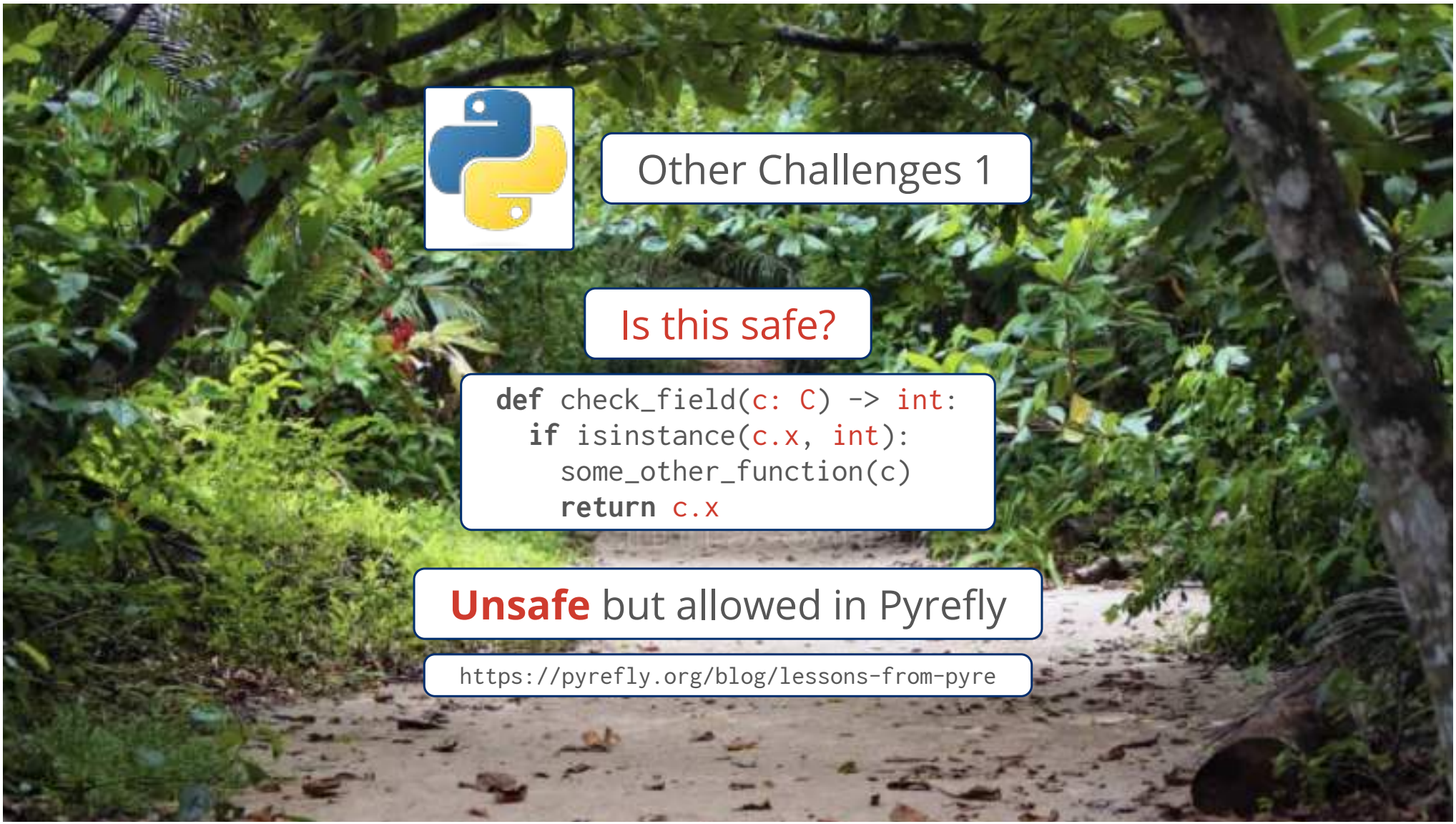

https://pyrefly.org/blog/too-many-type-checkers/
Polars : DataType.__eq__
@overload # type: ignore[override]
def __eq__(# pyrefly: ignore[bad-override]
self, other: pl.DataTypeExpr
) -> pl.Expr: ...
@overload
def __eq__(self, other: PolarsDataType) -> bool: ...
def __eq__(self, other: pl.DataTypeExpr | PolarsDataType) -> pl.Expr | bool: \
ty: ignore[invalid-method-override]
pyright: ignore[reportIncompatibleMethodOverride]

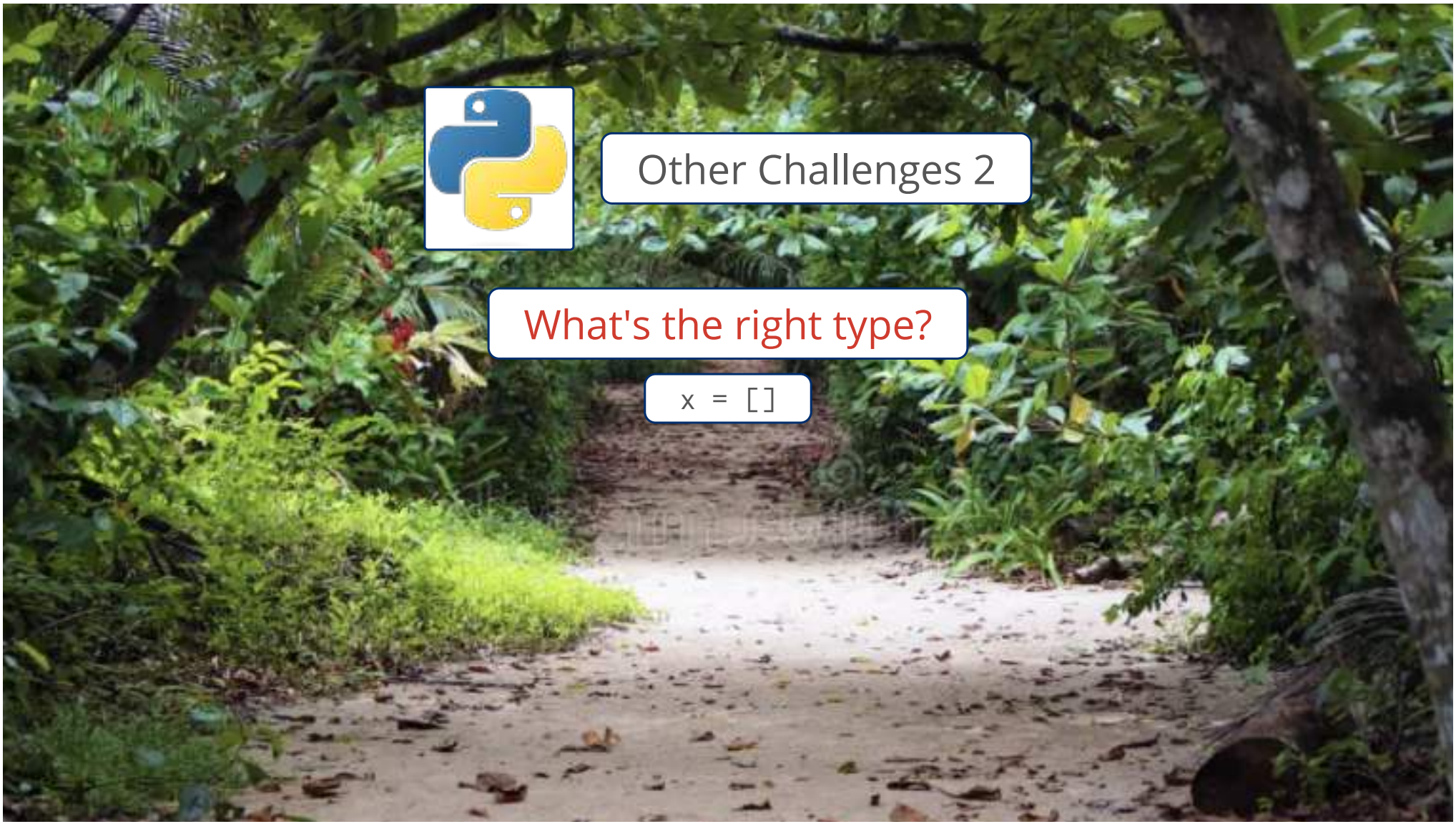

https://pyrefly.org/blog/too-many-type-checkers/
Why so many typecheckers?
"Some choose to be as strict as possible"
"Others allow you to add type information more gradually"

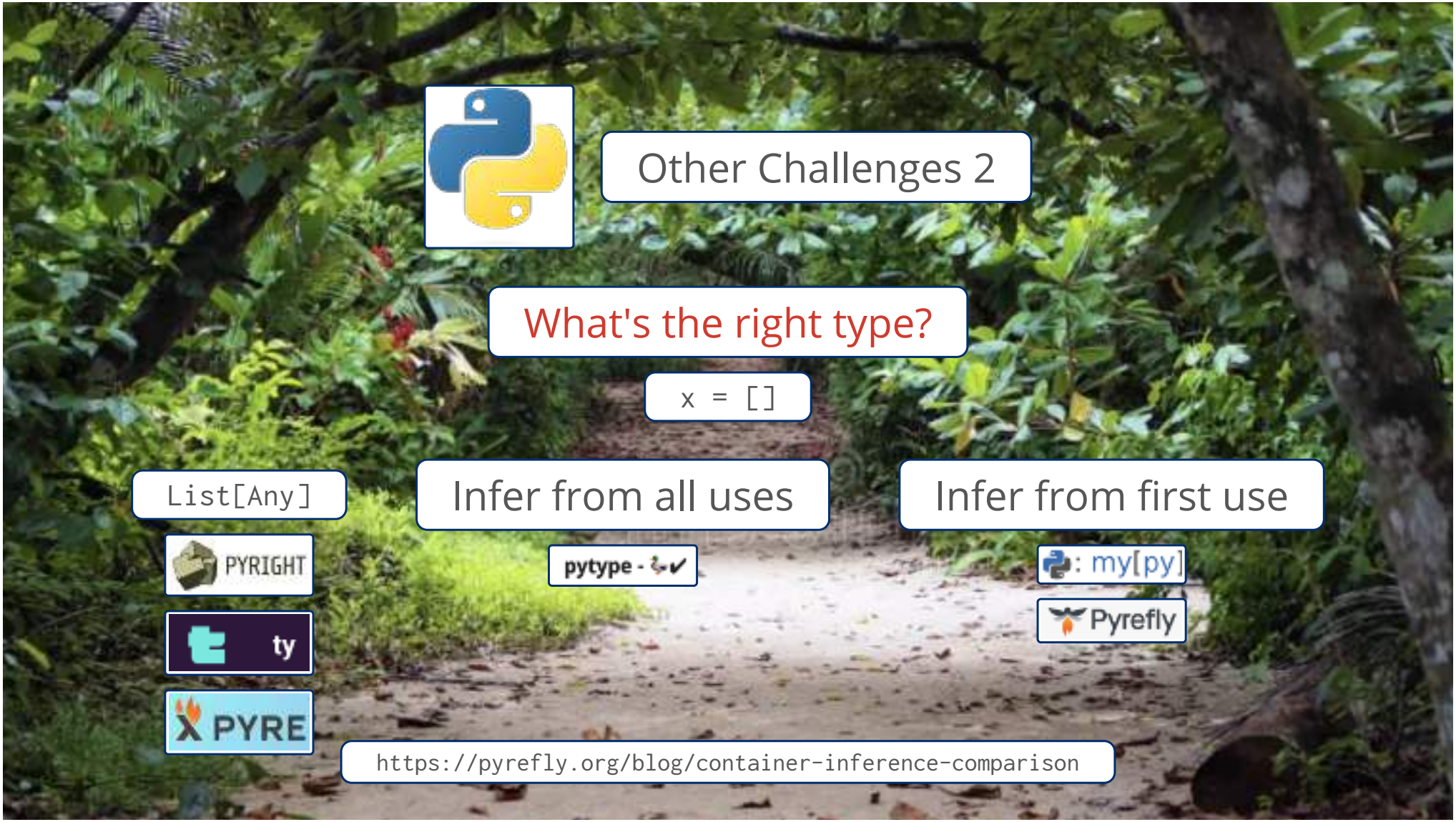

Major Challenges
First-class classes
del
import_module
eval()


Major Challenges
First-class classes
del
import_module
eval()
Strong reject of static types

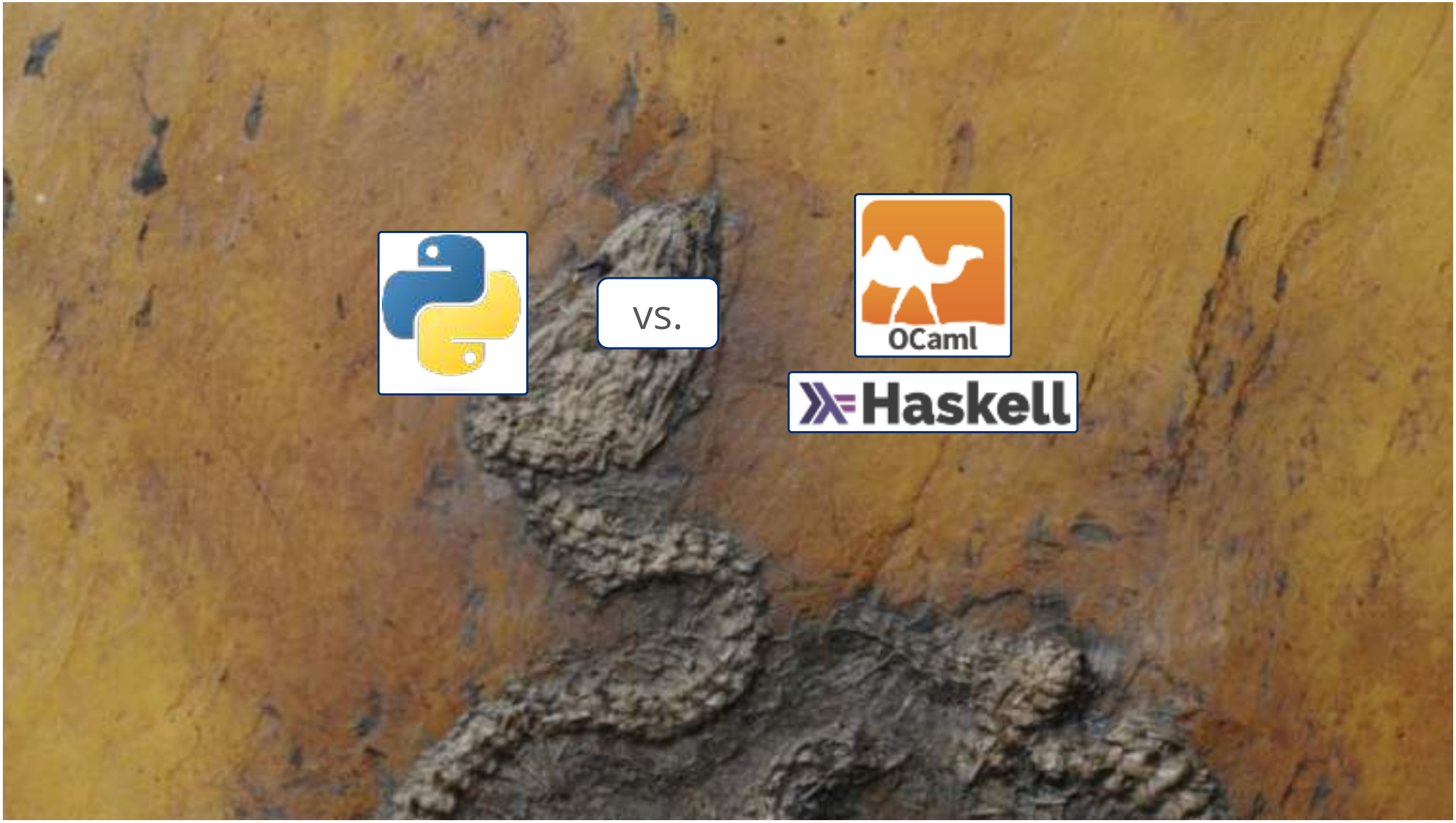

Other Challenges 1
Is this safe?
def check_field(c: C) -> int:
if isinstance(c.x, int):
some_other_function(c)
return c.x

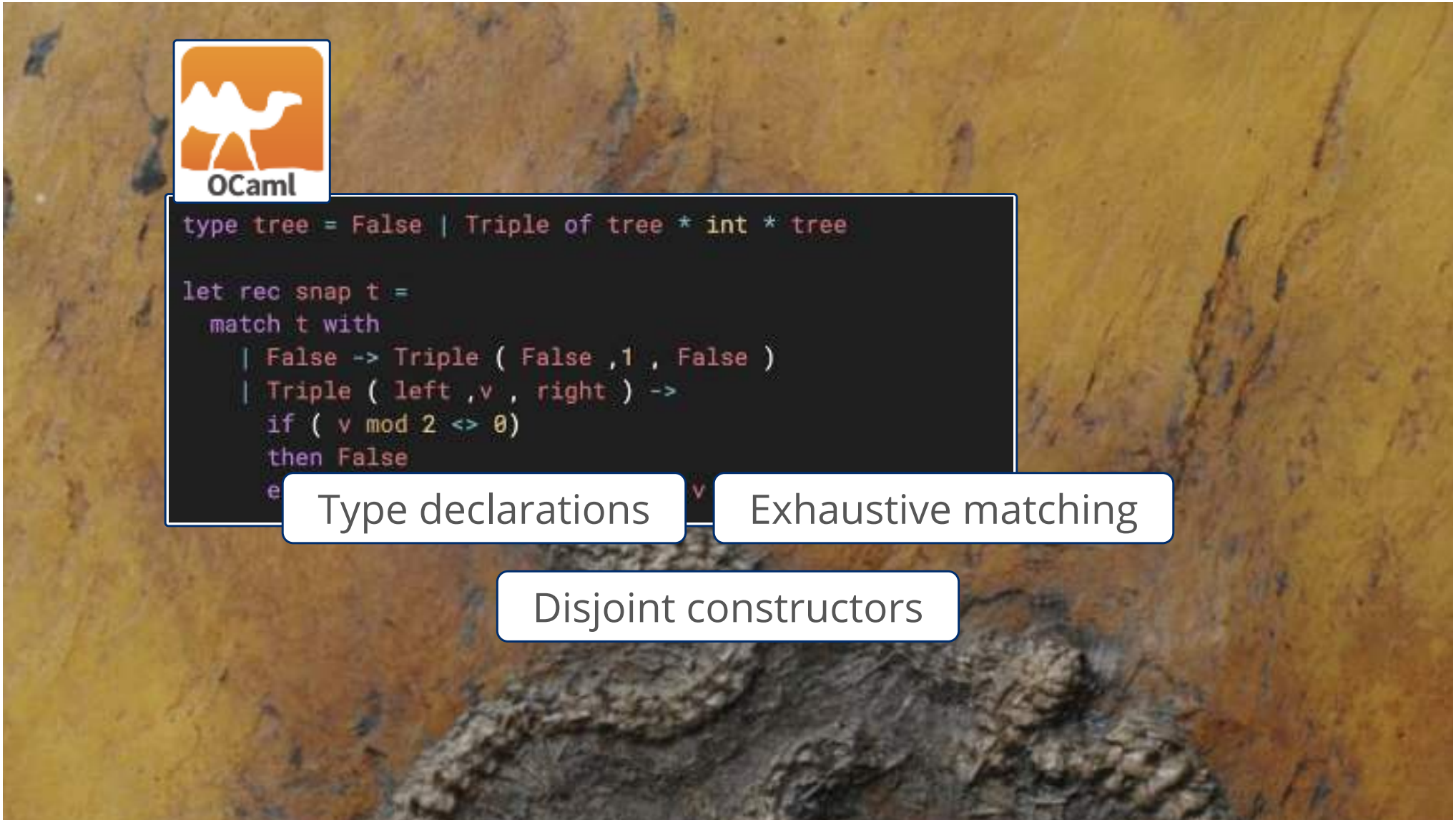

Other Challenges 1
Is this safe?
def check_field(c: C) -> int:
if isinstance(c.x, int):
some_other_function(c)
return c.x
Unsafe but allowed in Pyrefly
https://pyrefly.org/blog/lessons-from-pyre

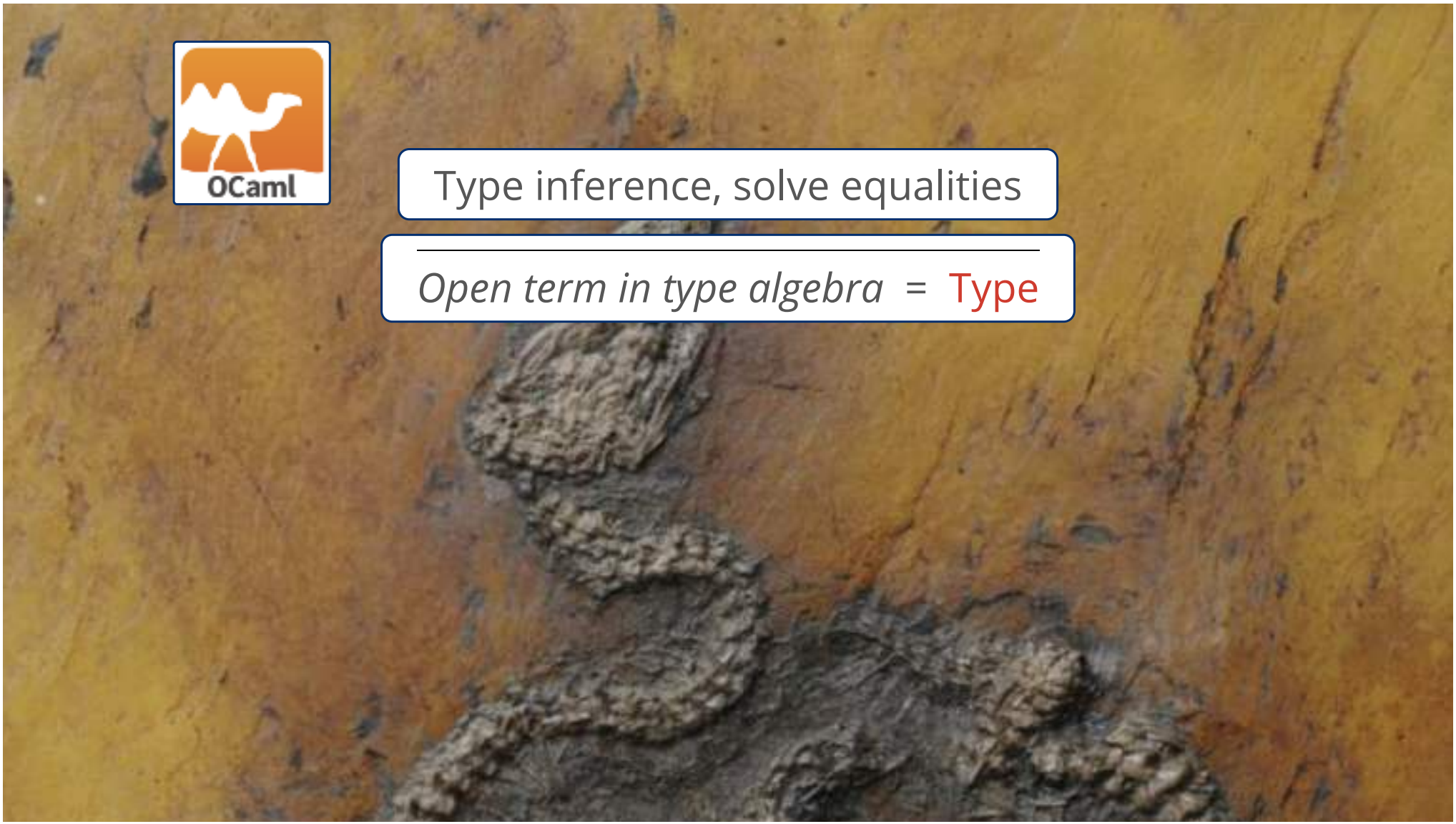

Other Challenges 2
What's the right type?
x = []

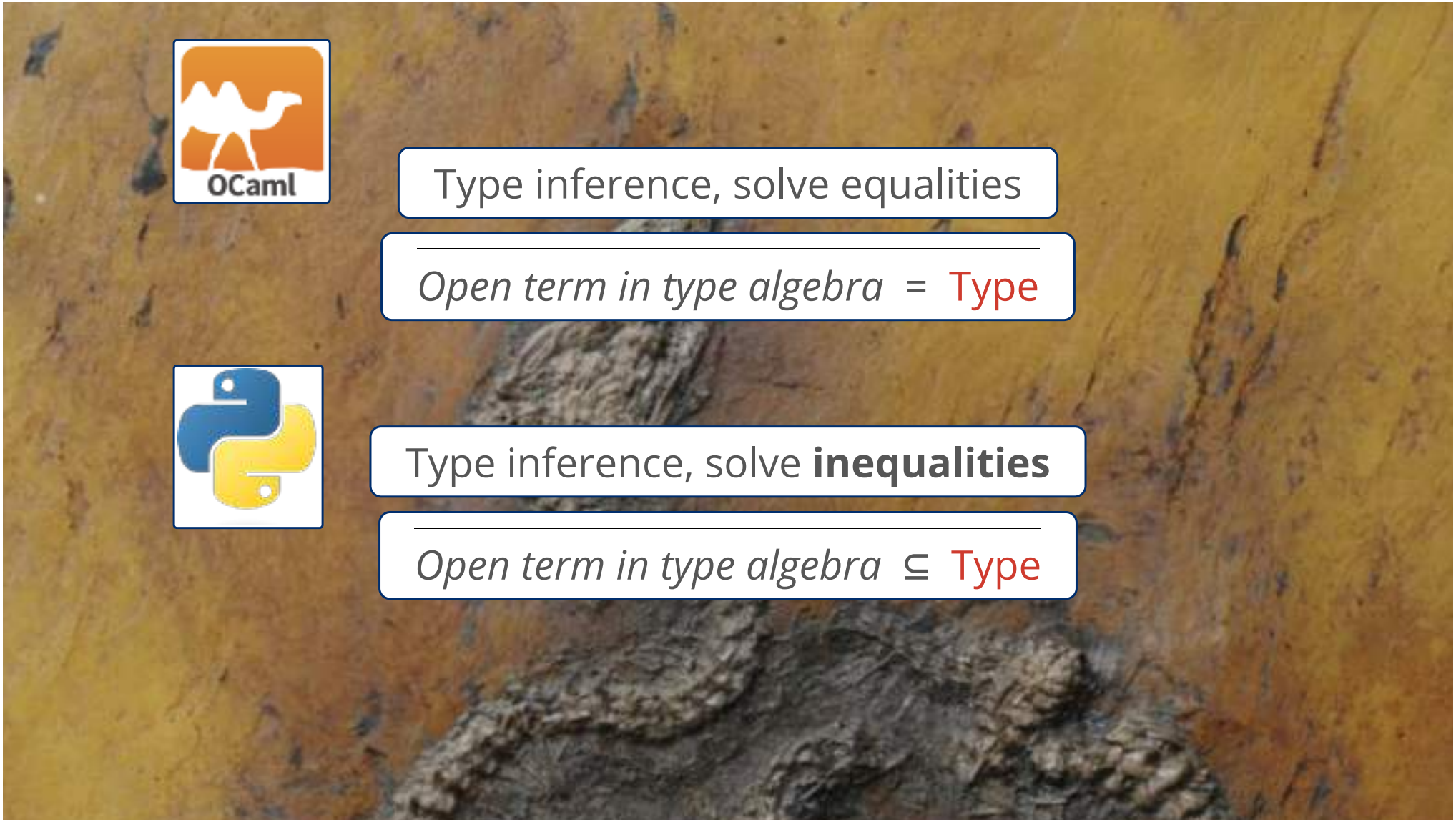

Other Challenges 2
What's the right type?
x = []
List[Any]
Infer from all uses
Infer from first use
PYRIGHT
pytype - 🐍✔
: my[py]
ty
Pyrefly
PYRE
https://pyrefly.org/blog/container-inference-comparison

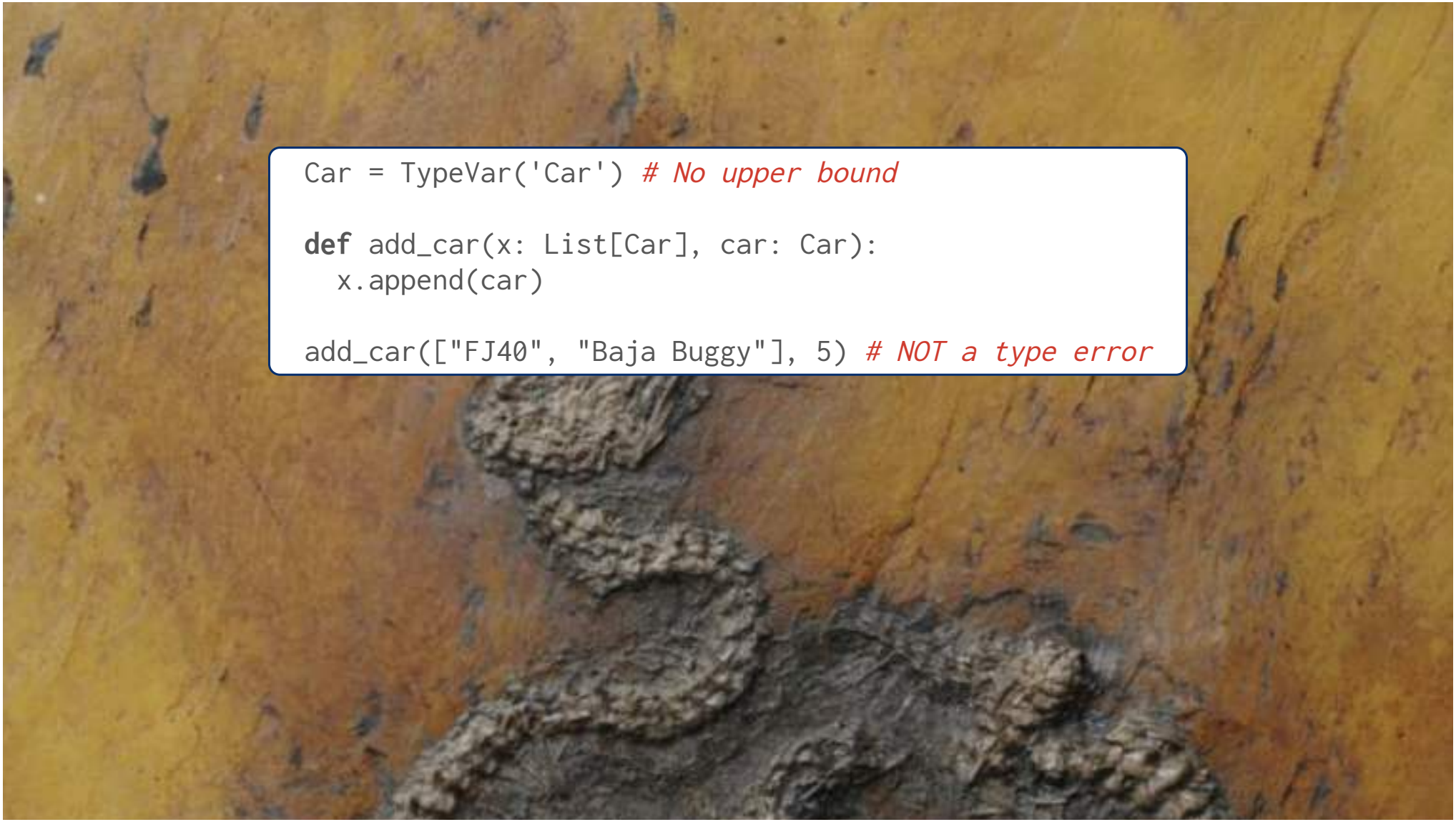

Even the **boring stuff** is hard!
Why?

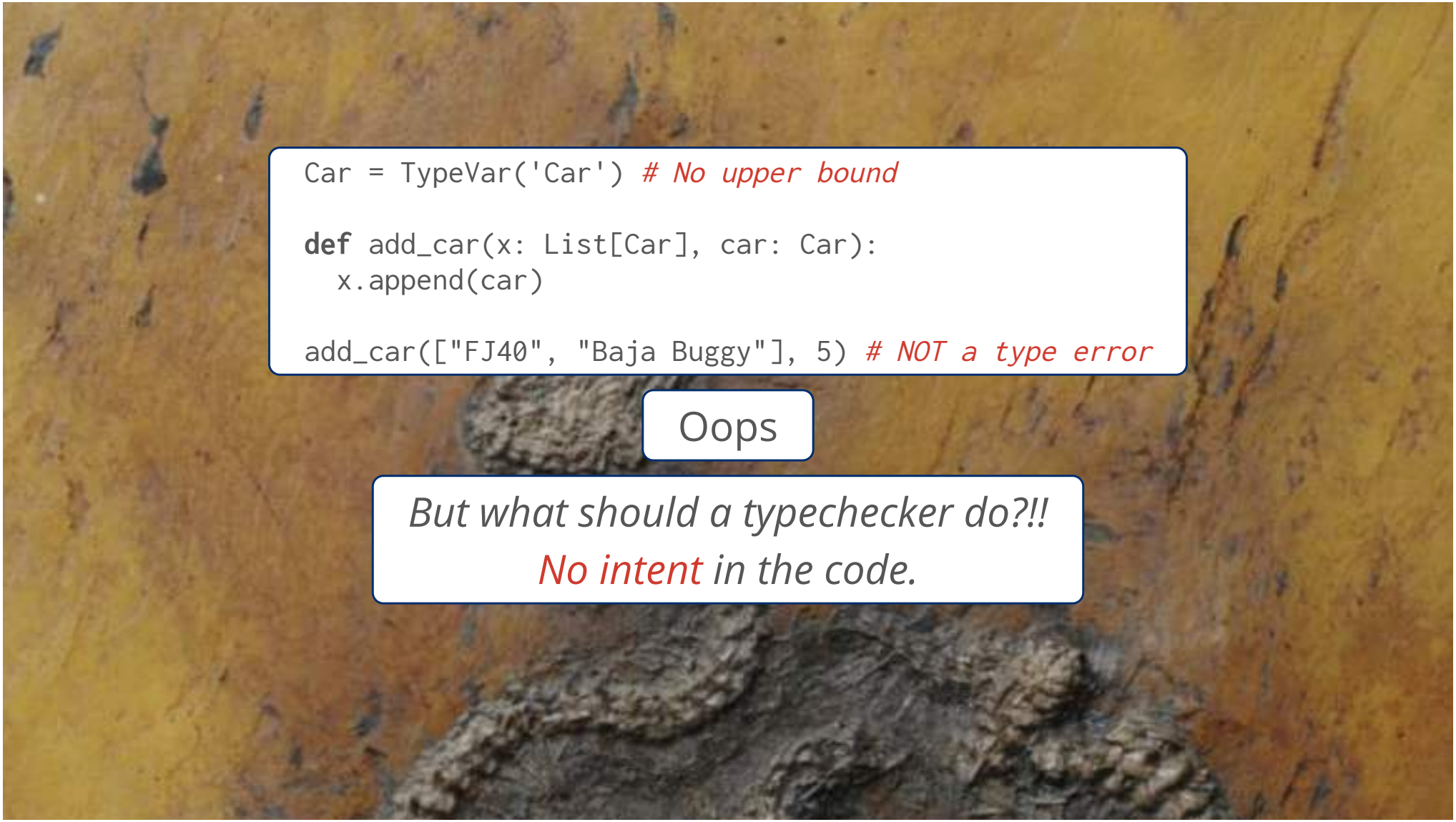

vs.
OCaml
Haskell


OCaml
type tree = False | Triple of tree * int * tree
let rec snap t =
match t with
| False -> Triple (False ,1 , False)
| Triple (left ,v , right) ->
if (v mod 2 <> 0)
then False
Type declarations
Exhaustive matching
Disjoint constructors


OCaml
Type inference, solve equalities
Open term in type algebra = Type


OCaml
Type inference, solve equalities
Open term in type algebra = Type
Type inference, solve **inequalities**
Open term in type algebra ⊆ Type


Car = TypeVar('Car') # No upper bound
def add_car(x: List[Car], car: Car):
x.append(car)
add_car(["FJ40", "Baja Buggy"], 5) # NOT a type error



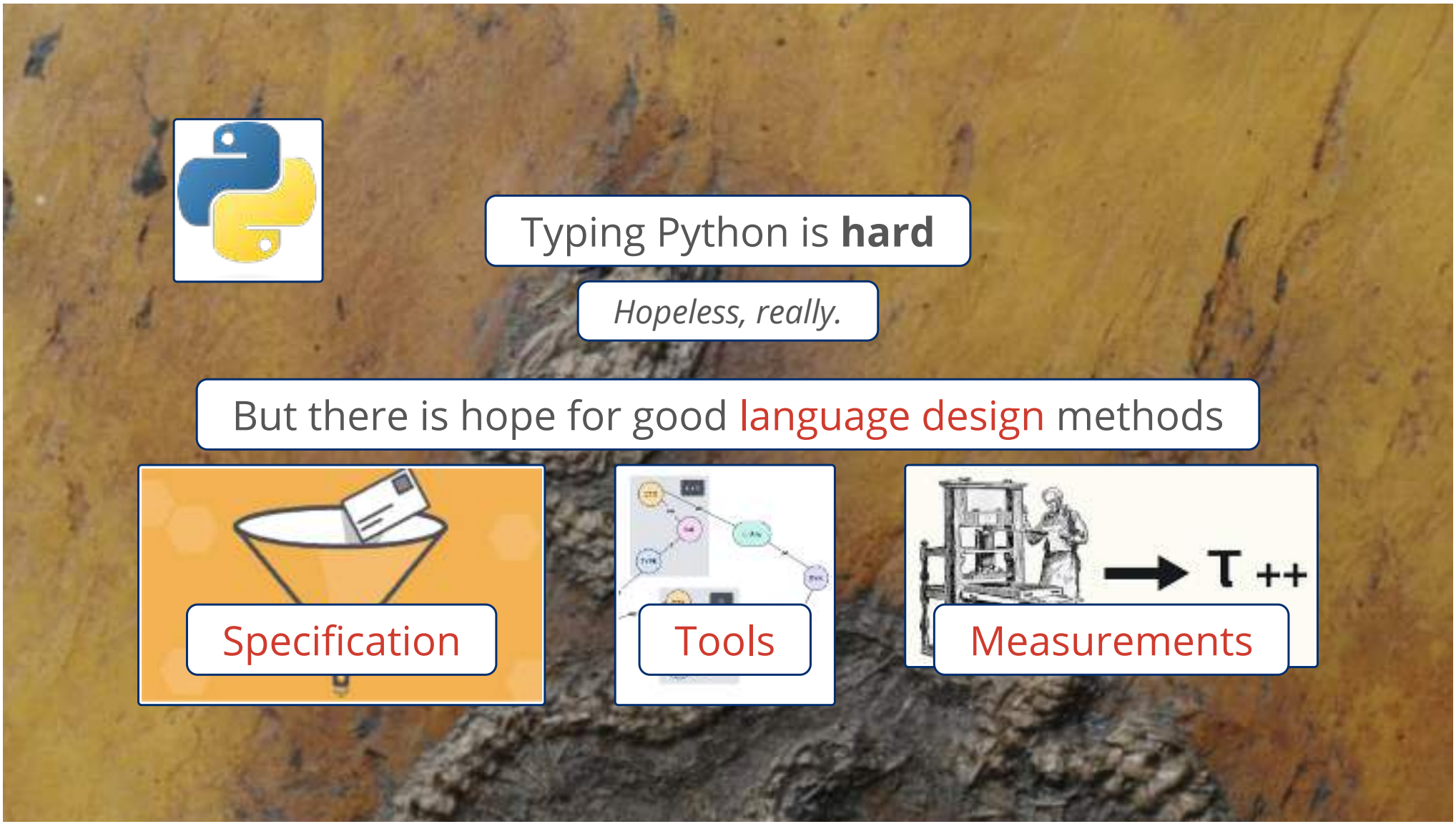
```
Car = TypeVar('Car') # No upper bound

def add_car(x: List[Car], car: Car):
  x.append(car)

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

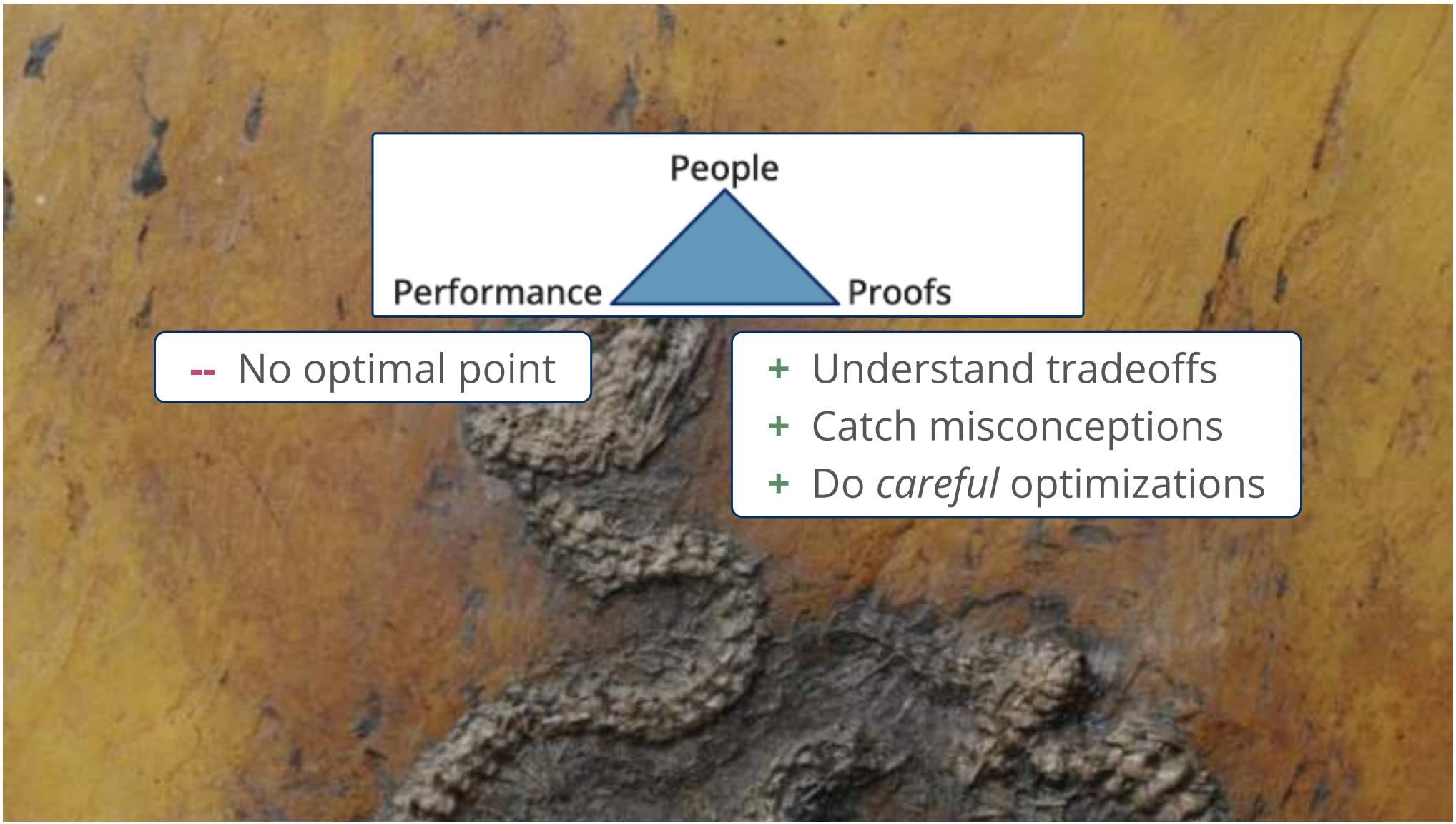
Oops

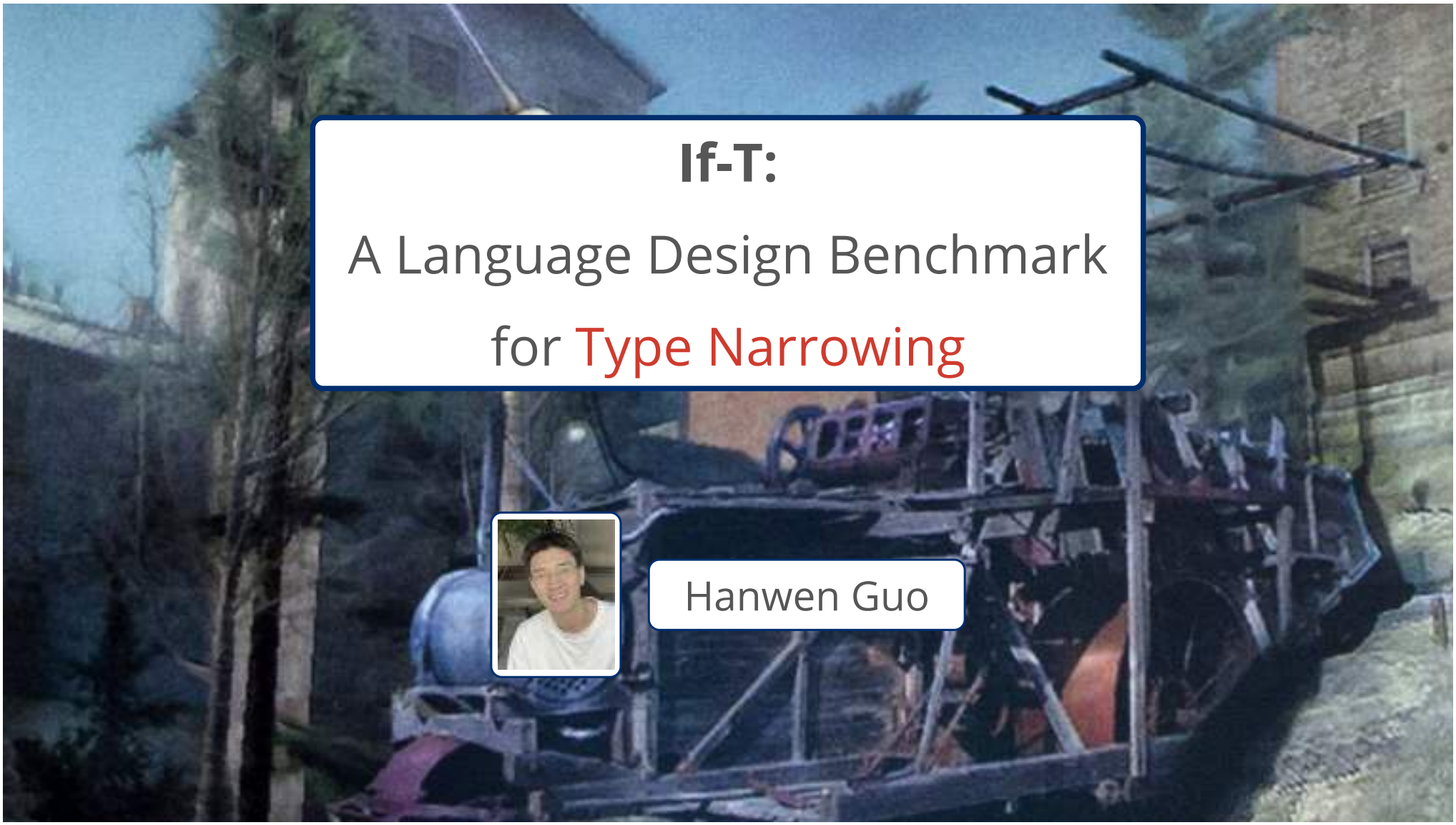
But what should a typechecker do?!!

No intent in the code.


Typing Python is **hard**
Hopeless, really.


Typing Python is **hard**
Hopeless, really.
But there is hope for good language design methods
Specification
Tools
Measurements
T ++


People
Performance
Proofs
-- No optimal point
+ Understand tradeoffs
+ Catch misconceptions
+ Do *careful* optimizations



If-T:

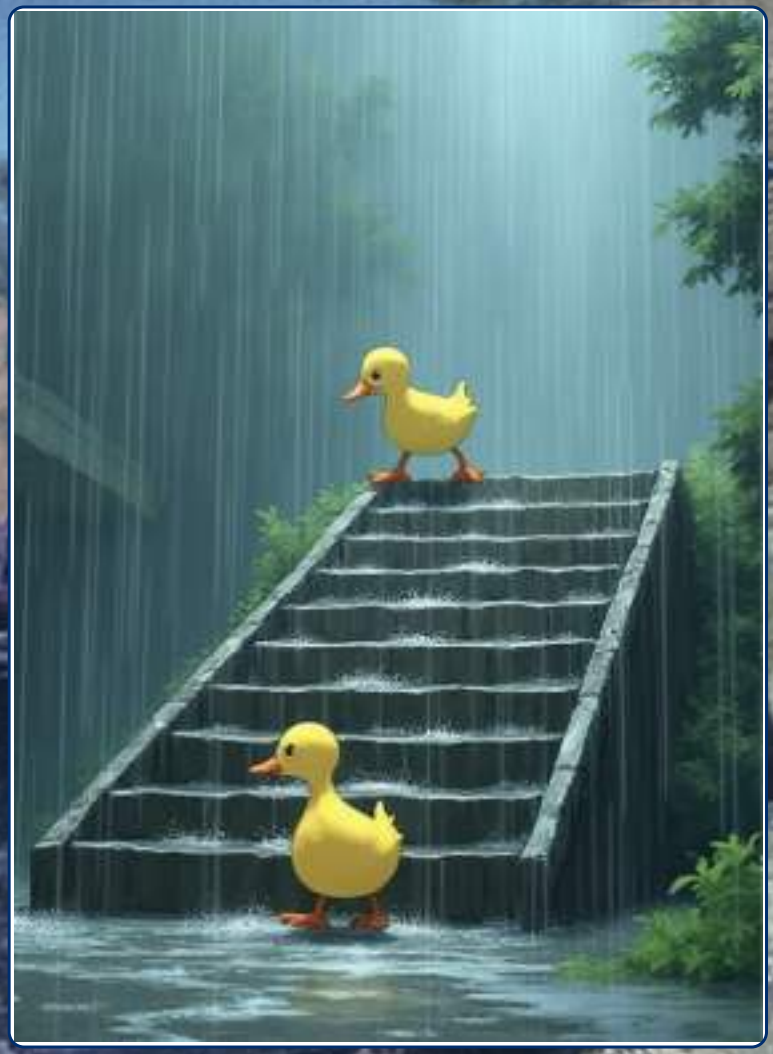
A Language Design Benchmark for Type Narrowing

Hanwen Guo

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).

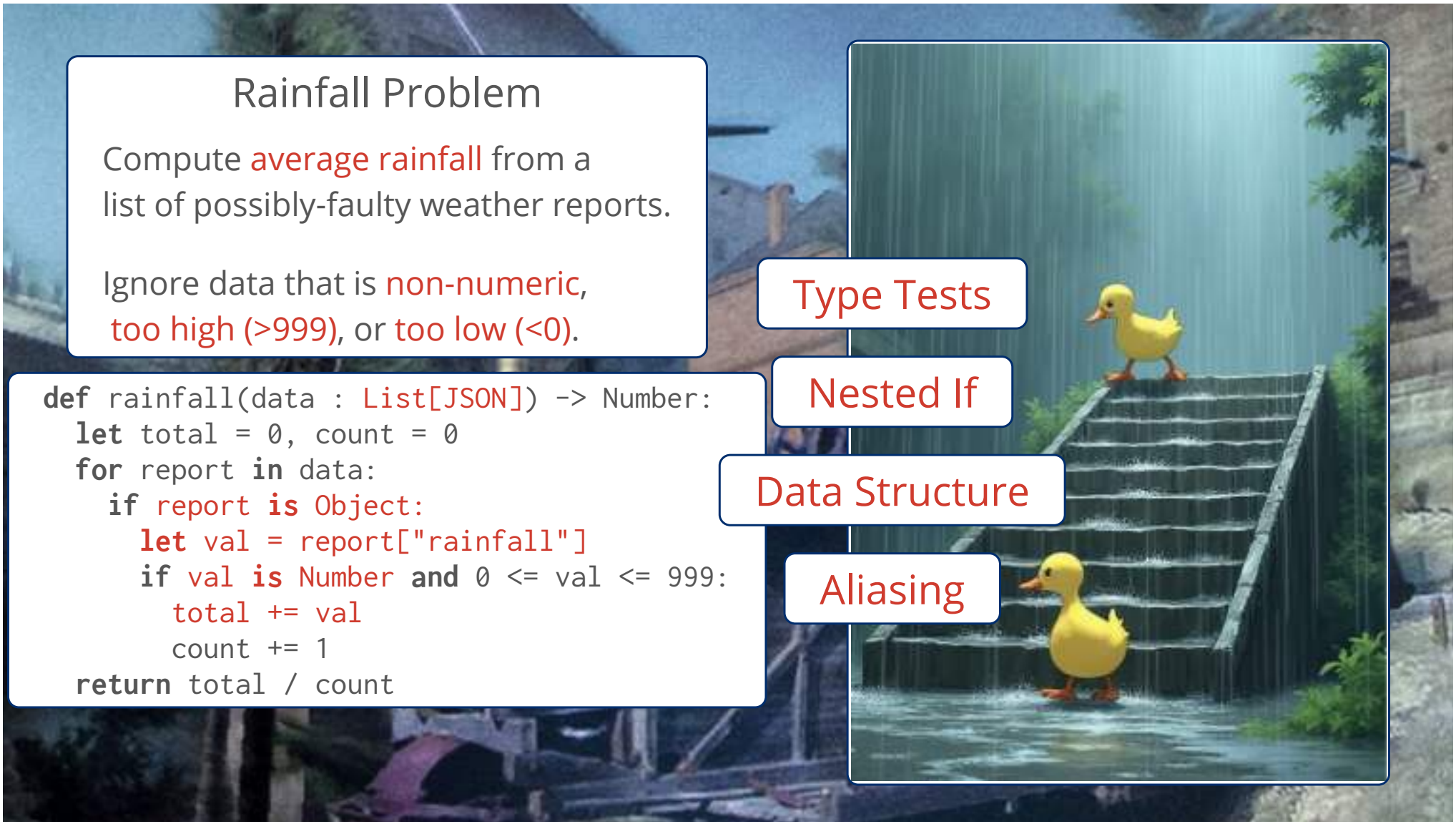


Rainfall Problem

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
```



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).

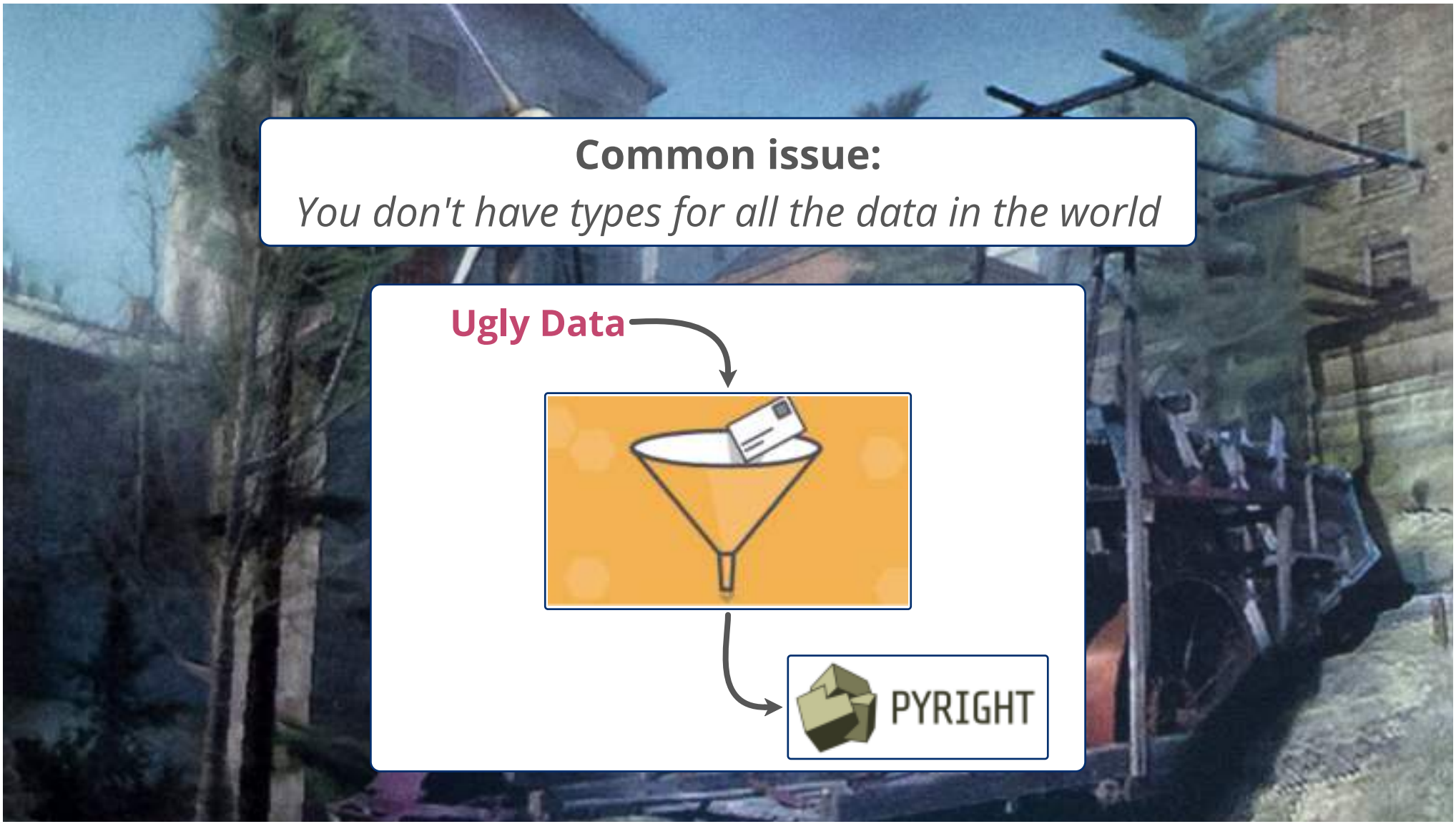
```
def rainfall(data : List[JSON]) -> Number:
  let total = 0, count = 0
  for report in data:
    if report is Object:
      let val = report["rainfall"]
      if val is Number and 0 <= val <= 999:
        total += val
        count += 1
  return total / count
```

Type Tests

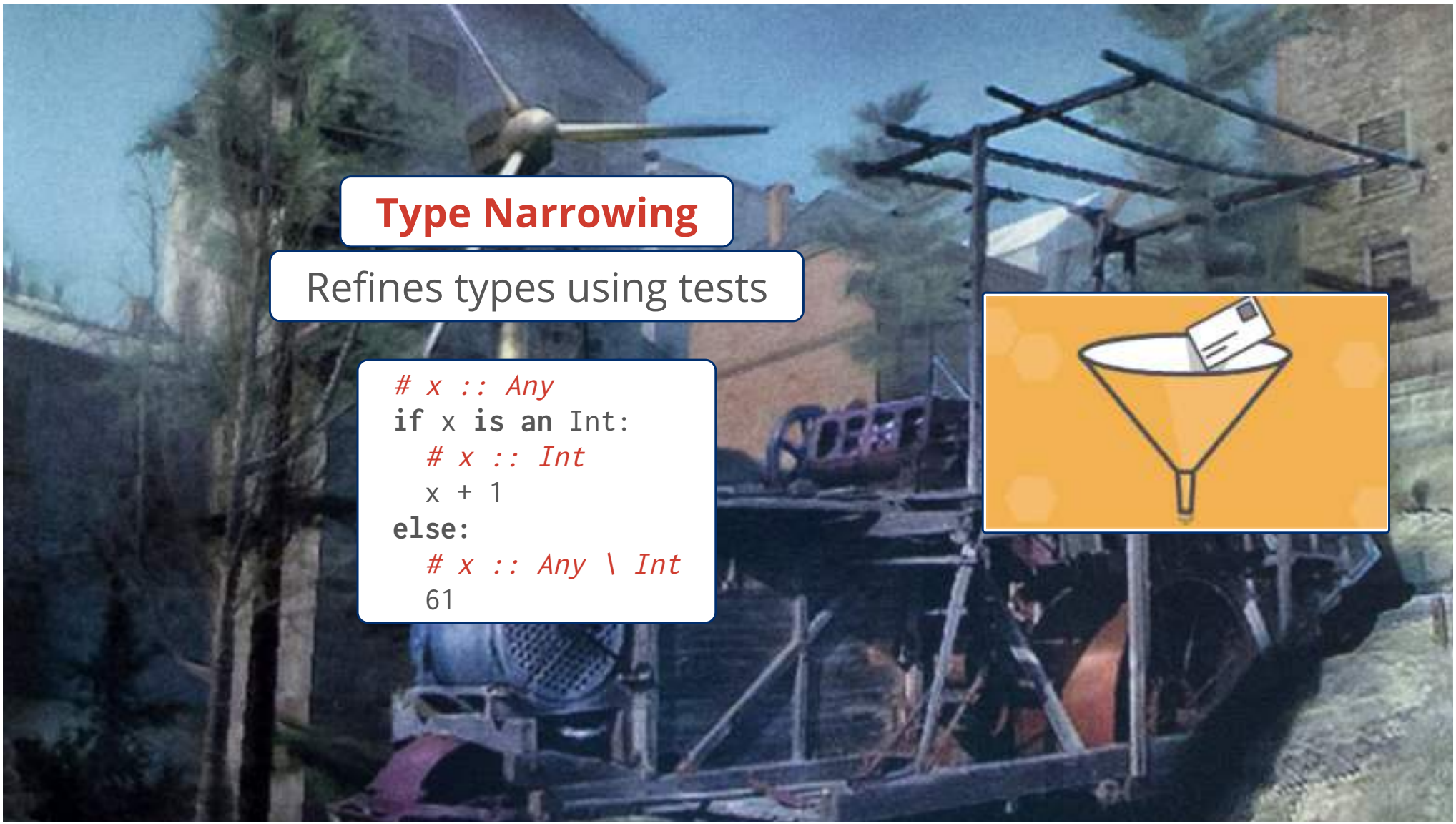
Nested If

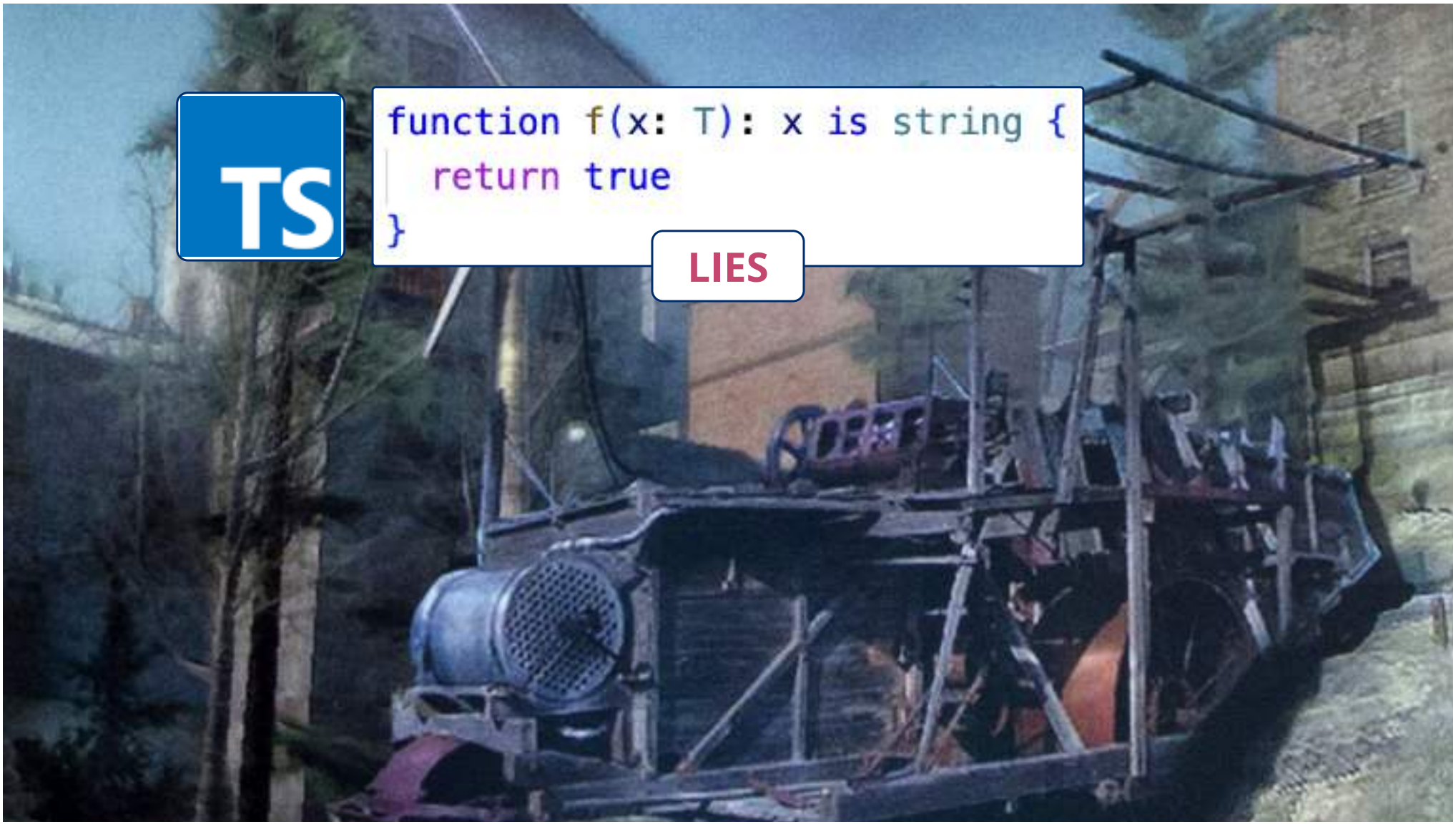
Data Structure

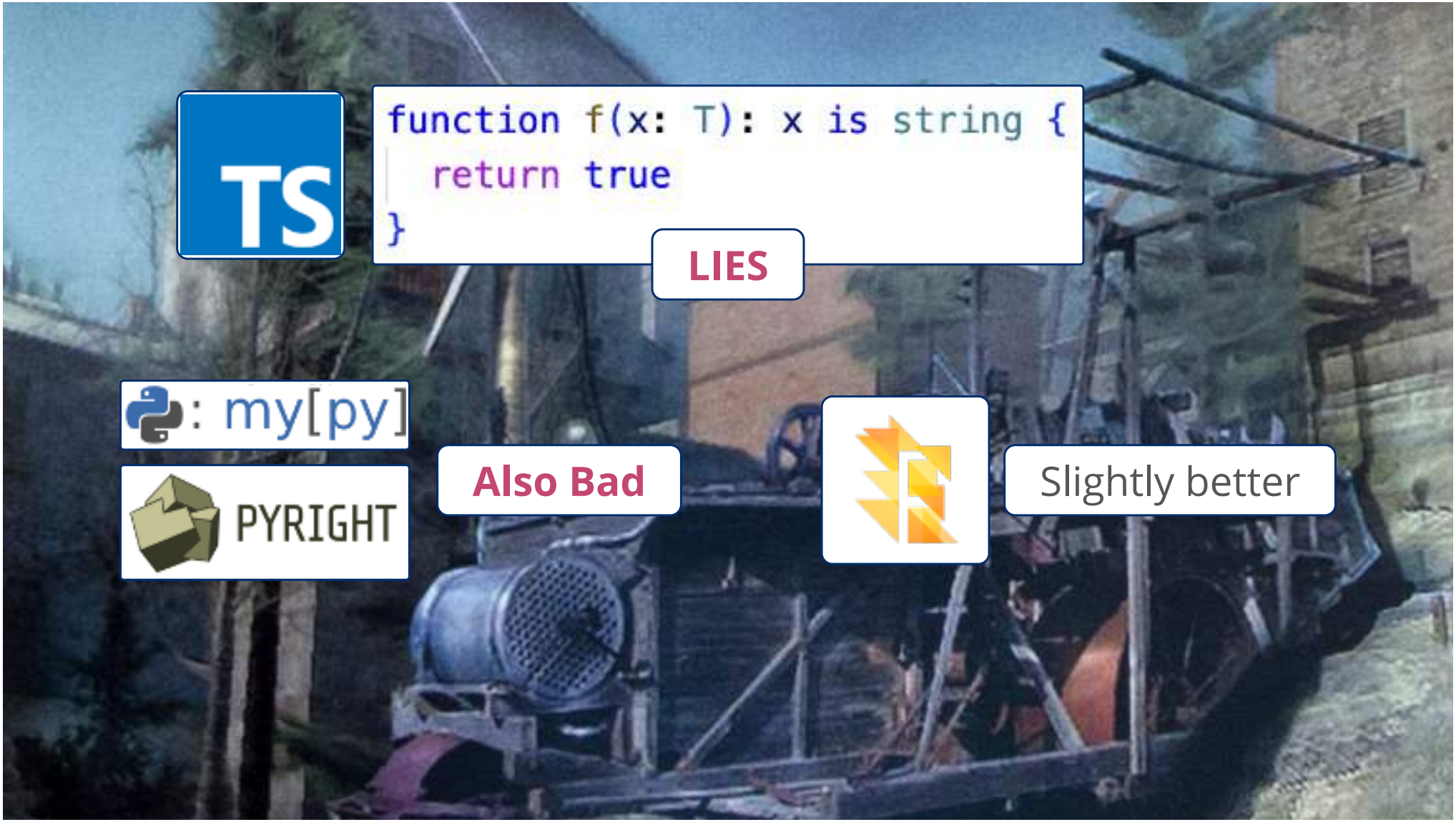
Aliasing

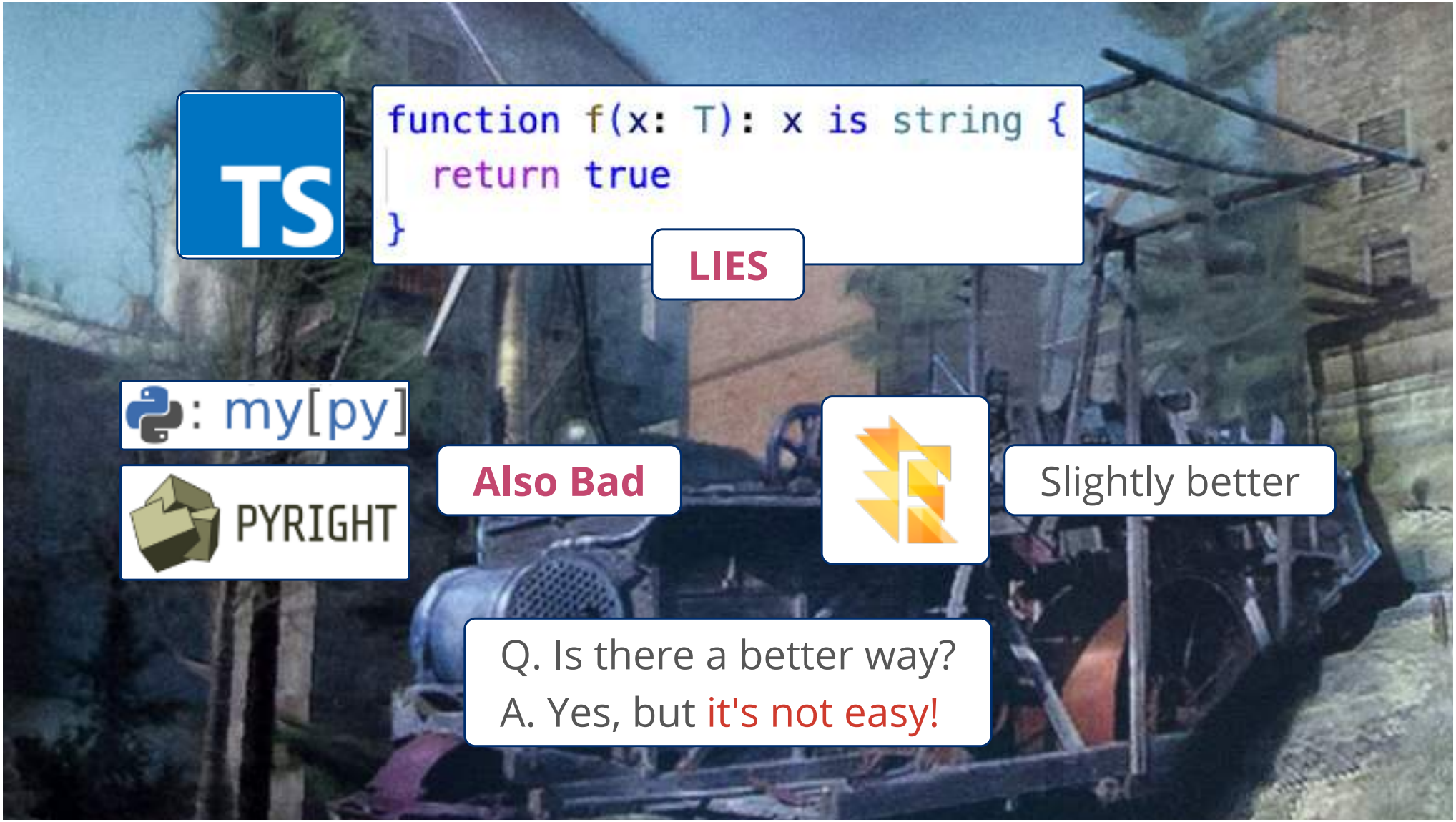

Common issue:
You don't have types for all the data in the world
Ugly Data
PYRIGHT


def int_filter(list: Listof(JSON)) -> Listof(Int):
let result = []
for element in list:
if element is Int:
result = cons(element, result)
return result
Should work for any user-defined predicate
def filter(f: Callable[Any, TypeGuard[int]], list ...)


Type Narrowing
Refines types using tests
# x :: Any
if x is an Int:
# x :: Int
x + 1
else:
# x :: Any \ Int
61


TS
function f(x: T): x is string {
return true
}
LIES

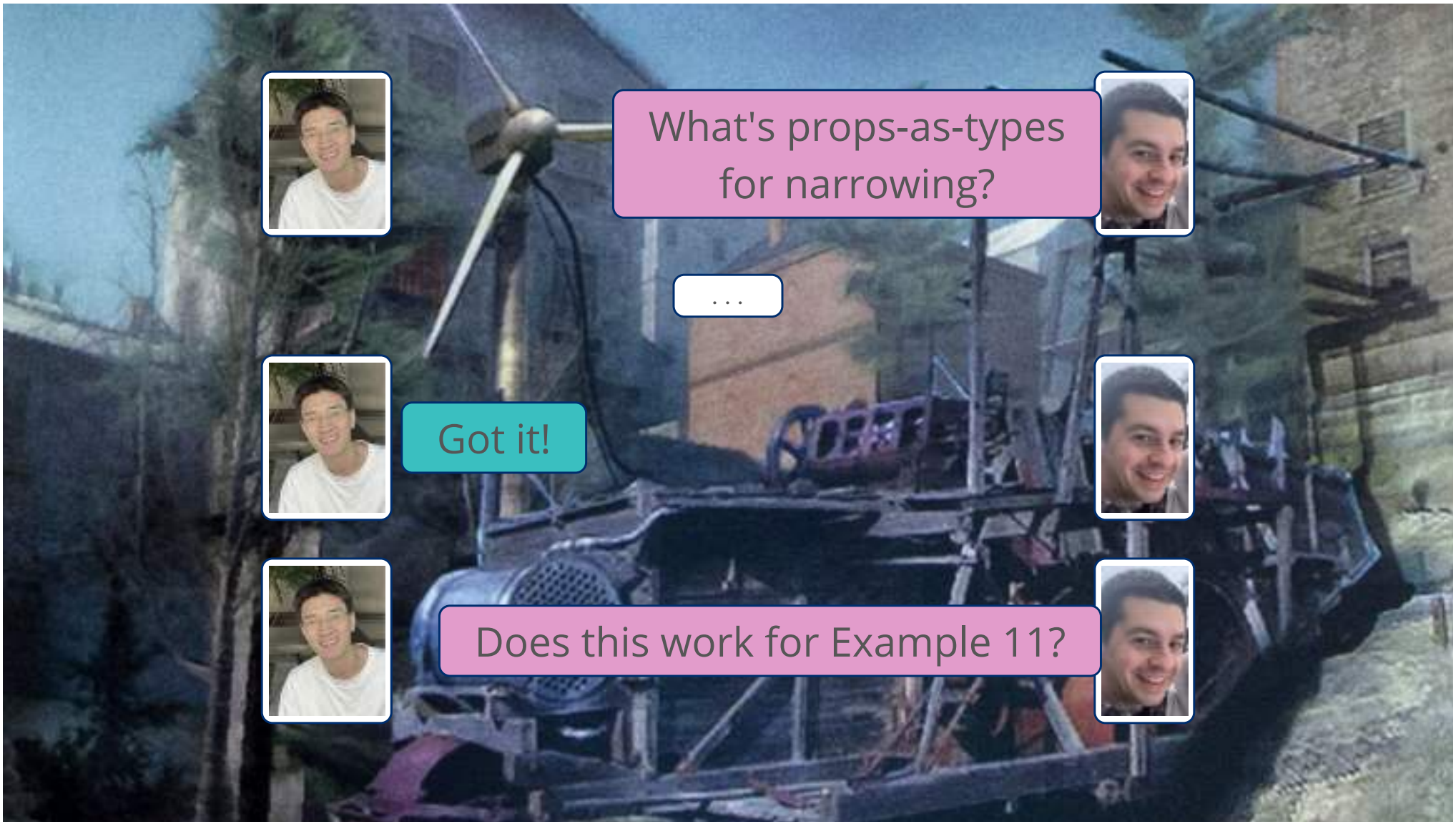

TS
function f(x: T): x is string {
return true
}
LIES
: my[py]
PYRIGHT
Also Bad
Slightly better

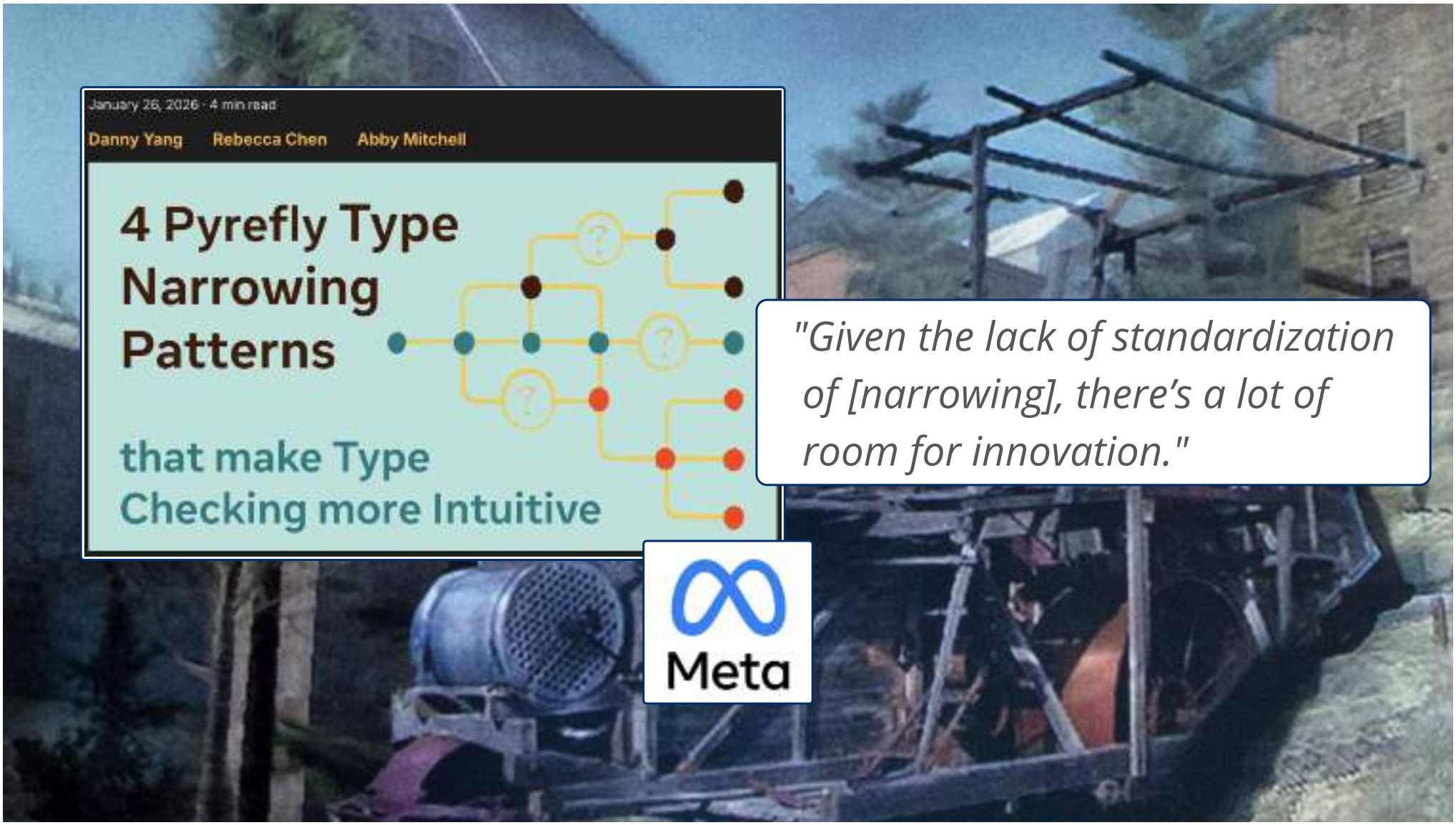

TS
function f(x: T): x is string {
return true
}
LIES
: my[py]
PYRIGHT
Also Bad
Slightly better
Q. Is there a better way?
A. Yes, but it's not easy!

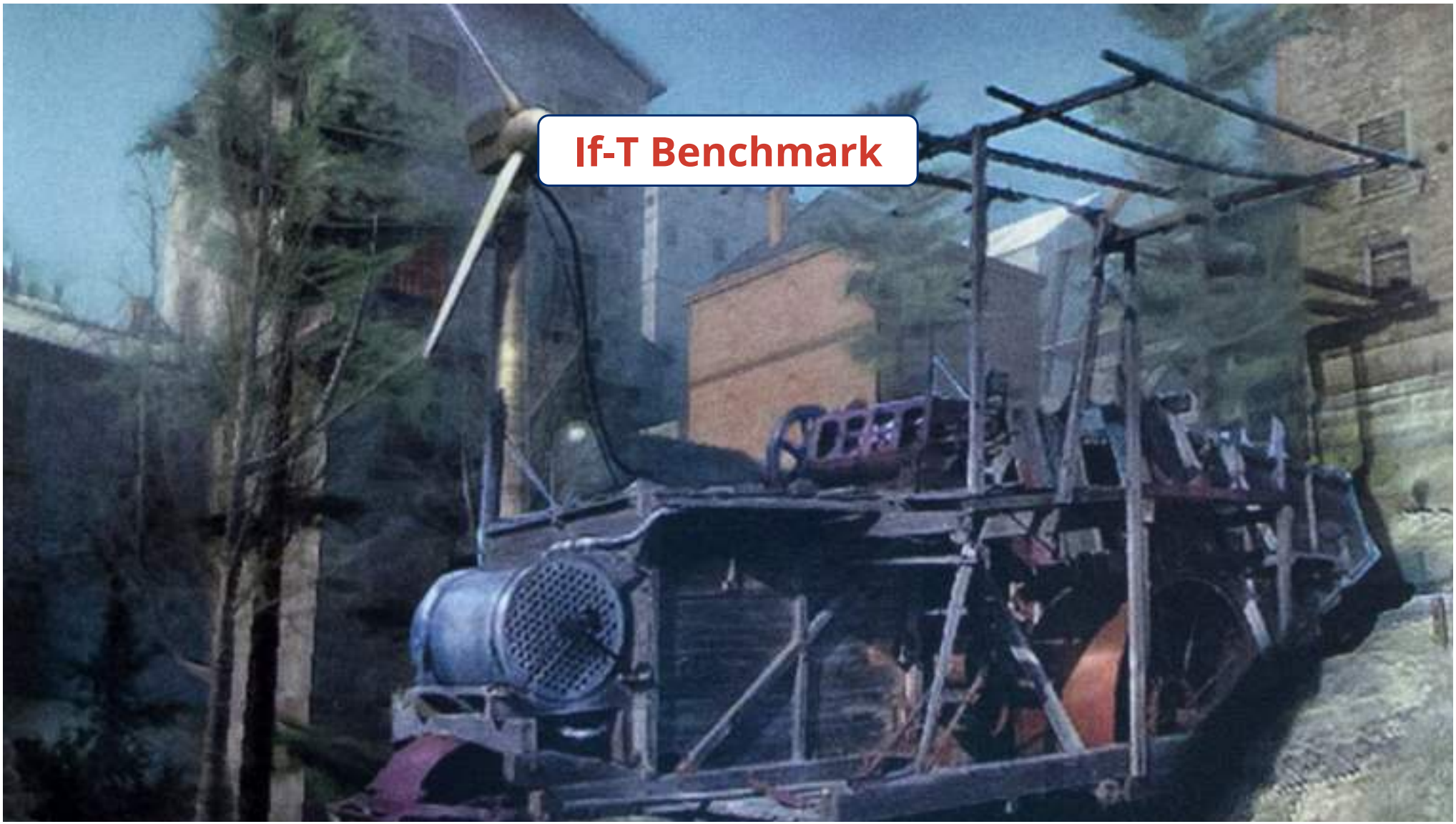

What's props-as-types for narrowing?


What's props-as-types for narrowing?
. . .


What's props-as-types for narrowing?
. . .
Got it!

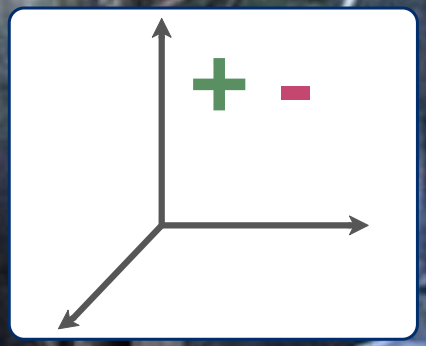

What's props-as-types for narrowing?
. . .
Got it!
Does this work for Example 11?


January 26, 2026 · 4 min read
Danny Yang
Rebecca Chen
Abby Mitchell
4 Pyrefly Type Narrowing Patterns
that make Type Checking more Intuitive
"Given the lack of standardization of [narrowing], there's a lot of room for innovation."
Meta


If-T Benchmark

If-T Benchmark

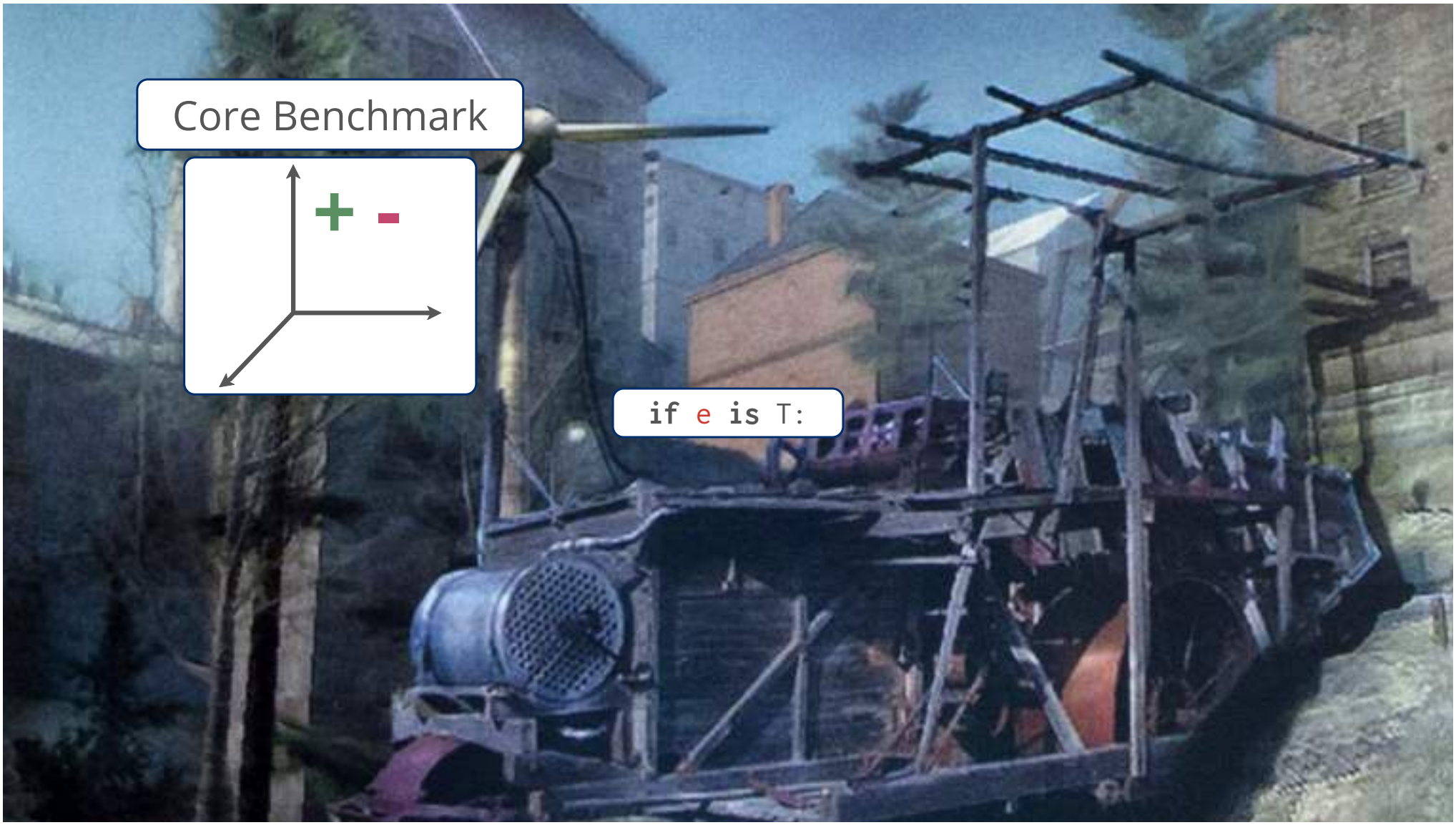
Core Benchmark

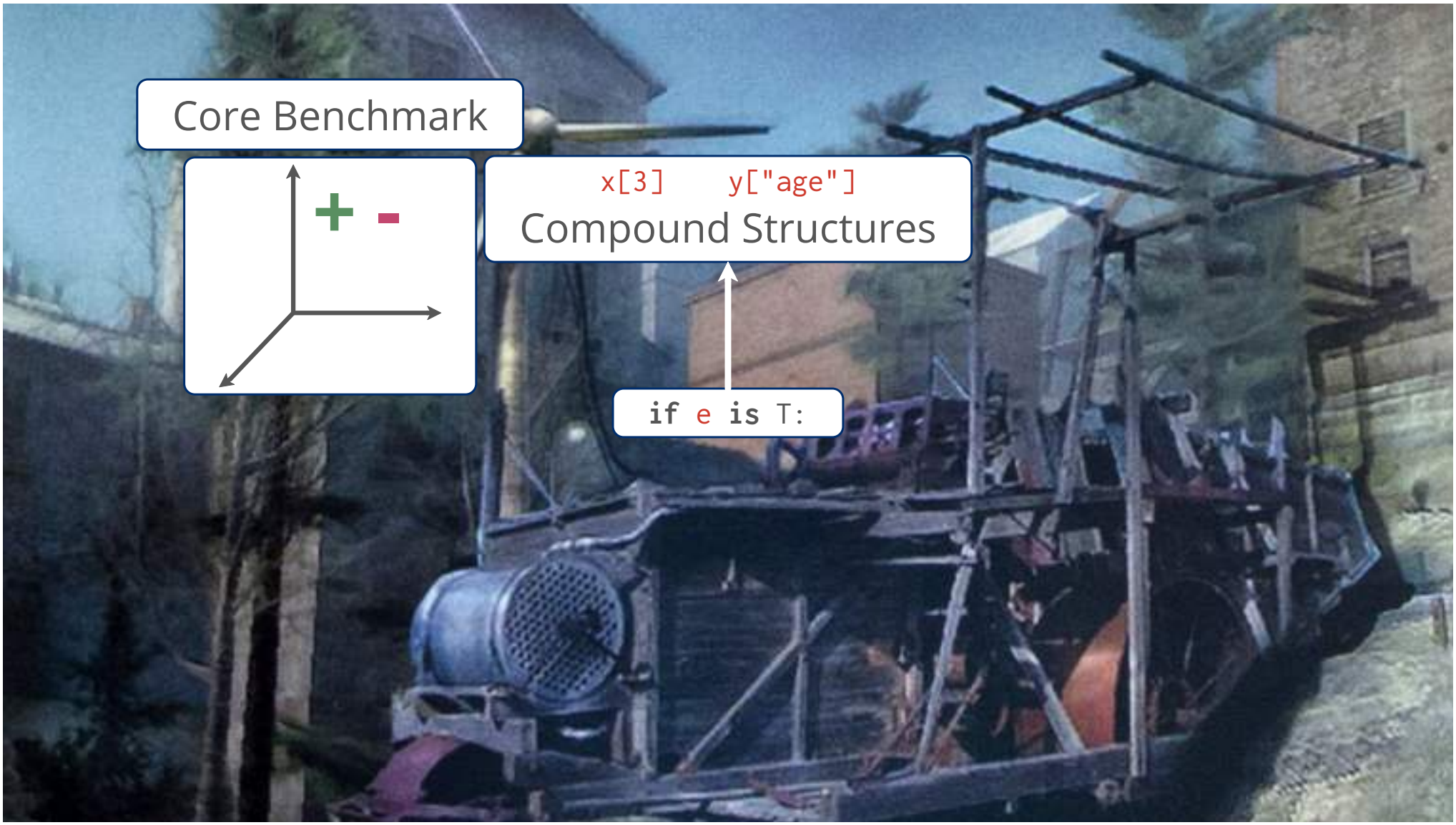


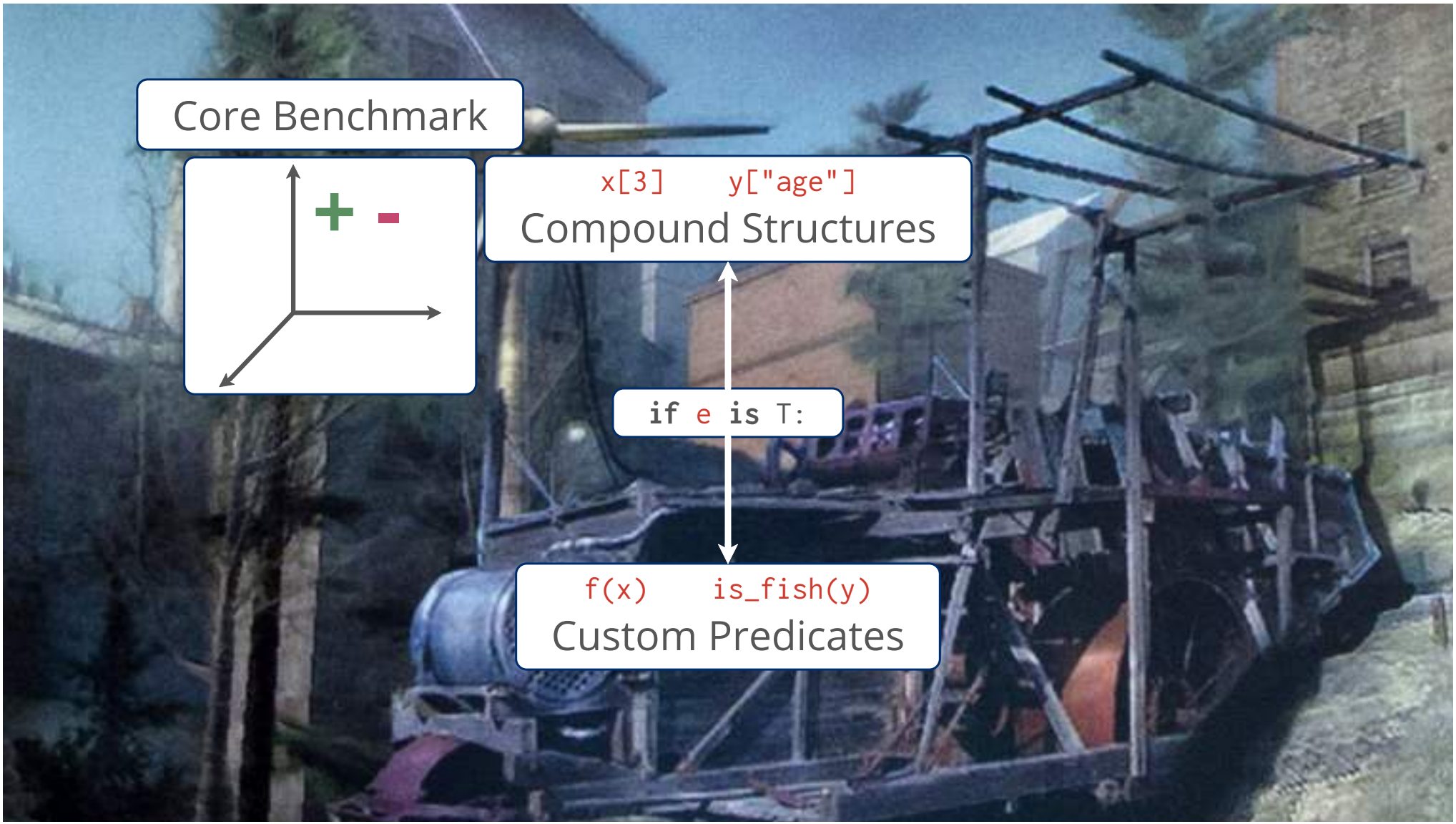
Example Programs

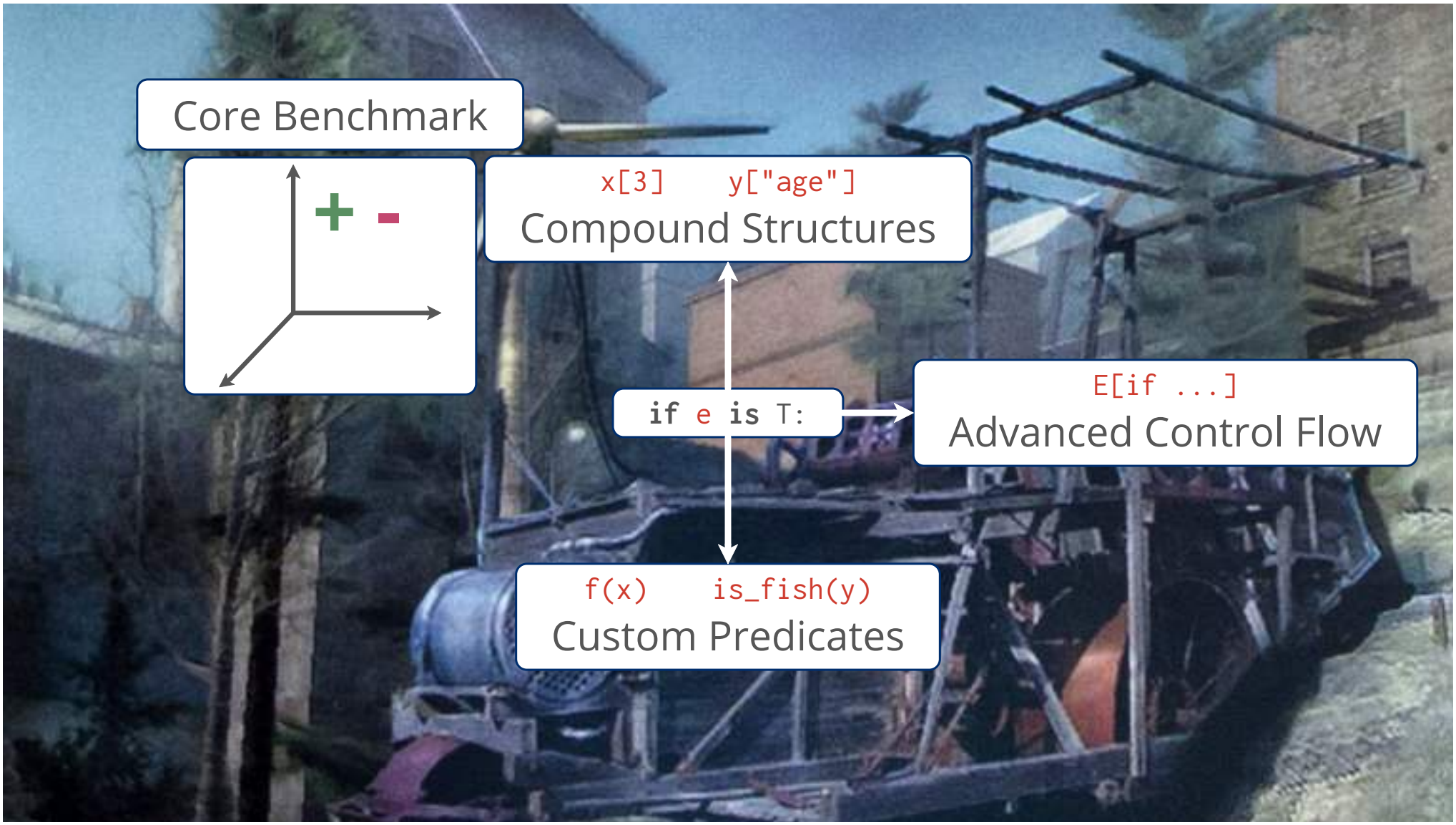
Datasheet

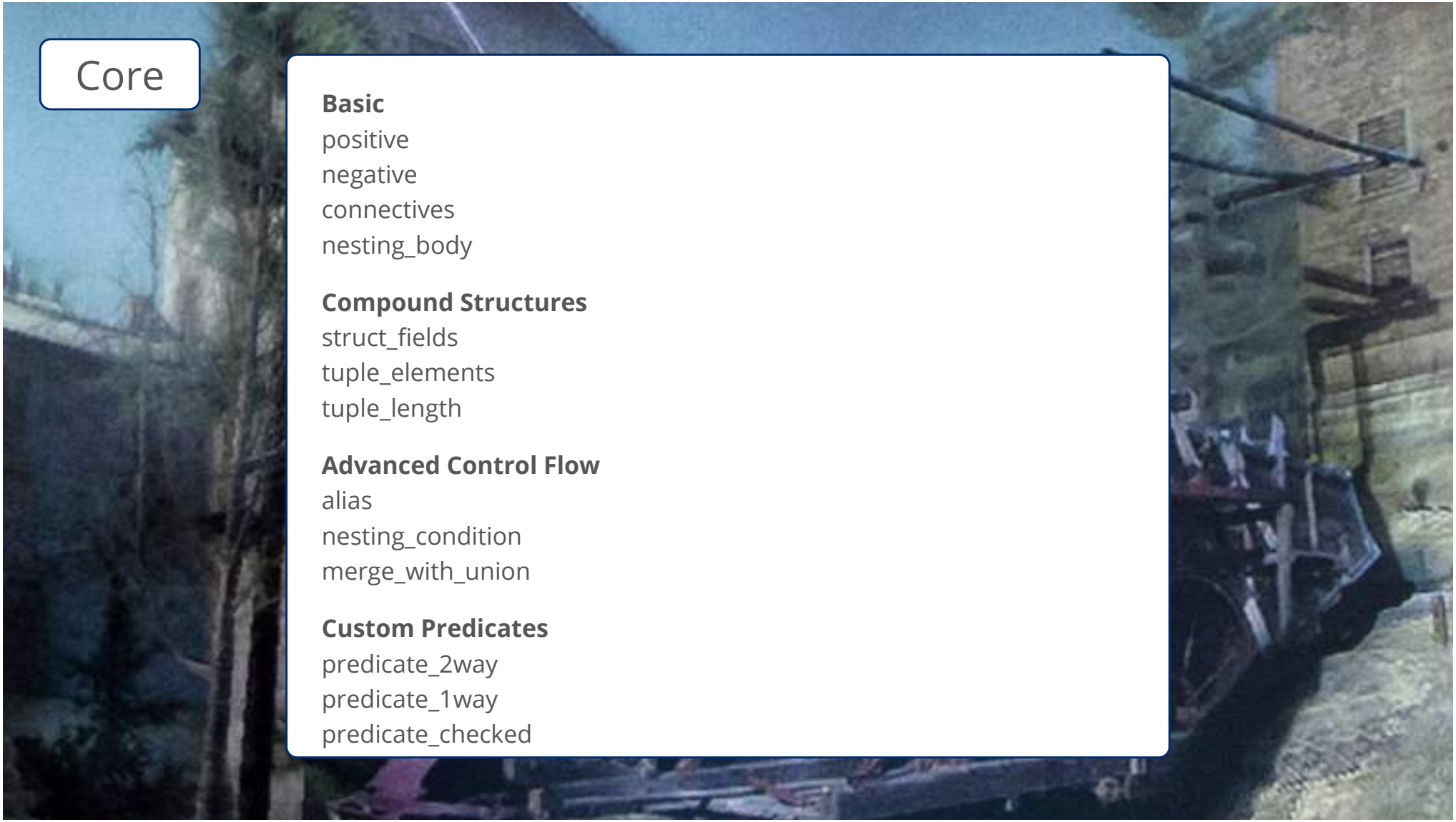
Structure inspired by **B2T2**


Core Benchmark
+ -
if e is T:


Core Benchmark
+ -
x[3] y["age"]
Compound Structures
if e is T:


Core Benchmark
+ -
x[3] y["age"]
Compound Structures
if e is T:
f(x) is_fish(y)
Custom Predicates


Core Benchmark
+ -
x[3] y["age"]
Compound Structures
if e is T:
E[if ...]
Advanced Control Flow
f(x) is_fish(y)
Custom Predicates



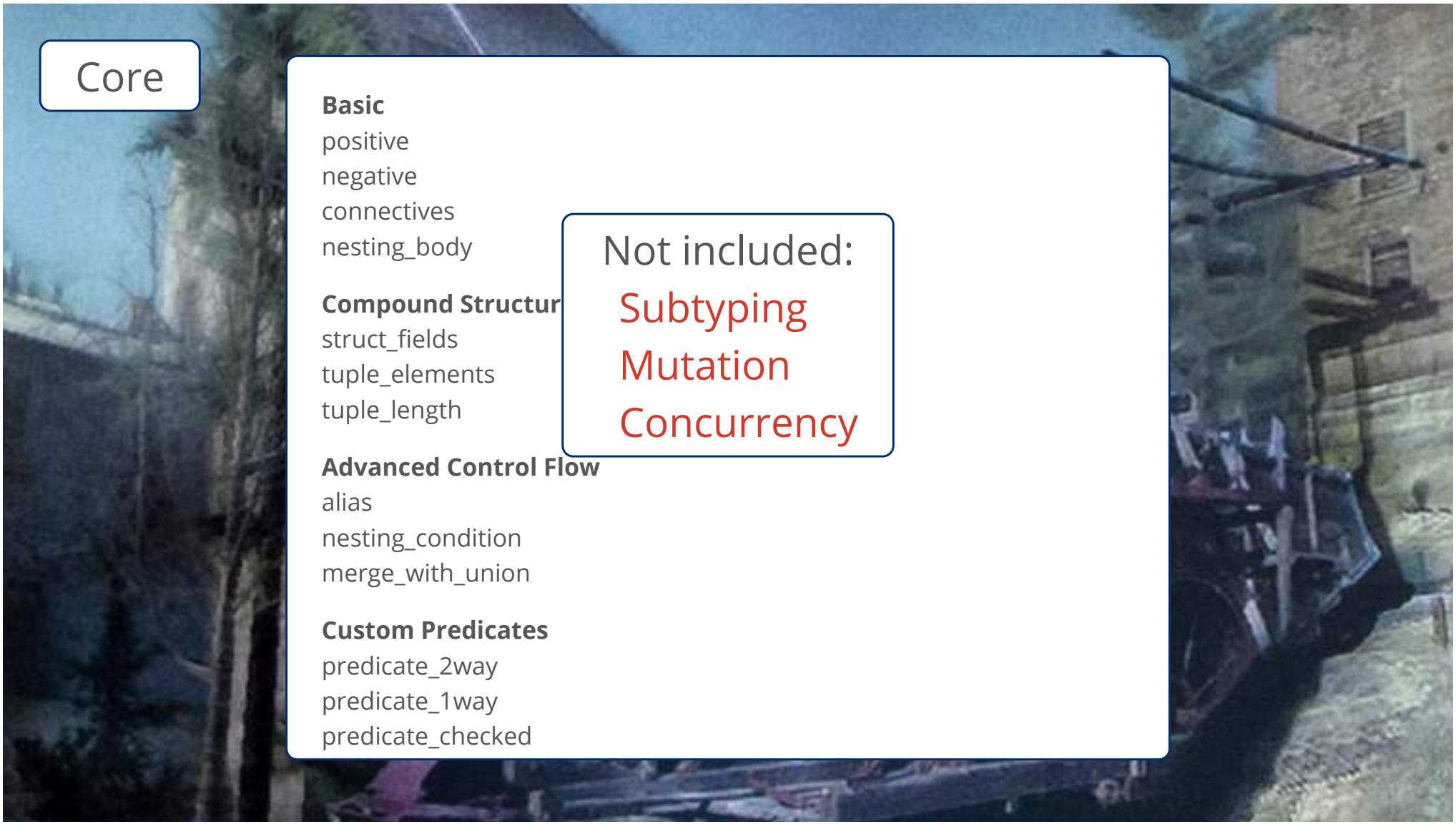
Core

Basic
positive
negative
connectives
nesting_body

Compound Structures
struct_fields
tuple_elements
tuple_length

Advanced Control Flow
alias
nesting_condition
merge_with_union

Custom Predicates
predicate_2way
predicate_1way
predicate_checked



Core

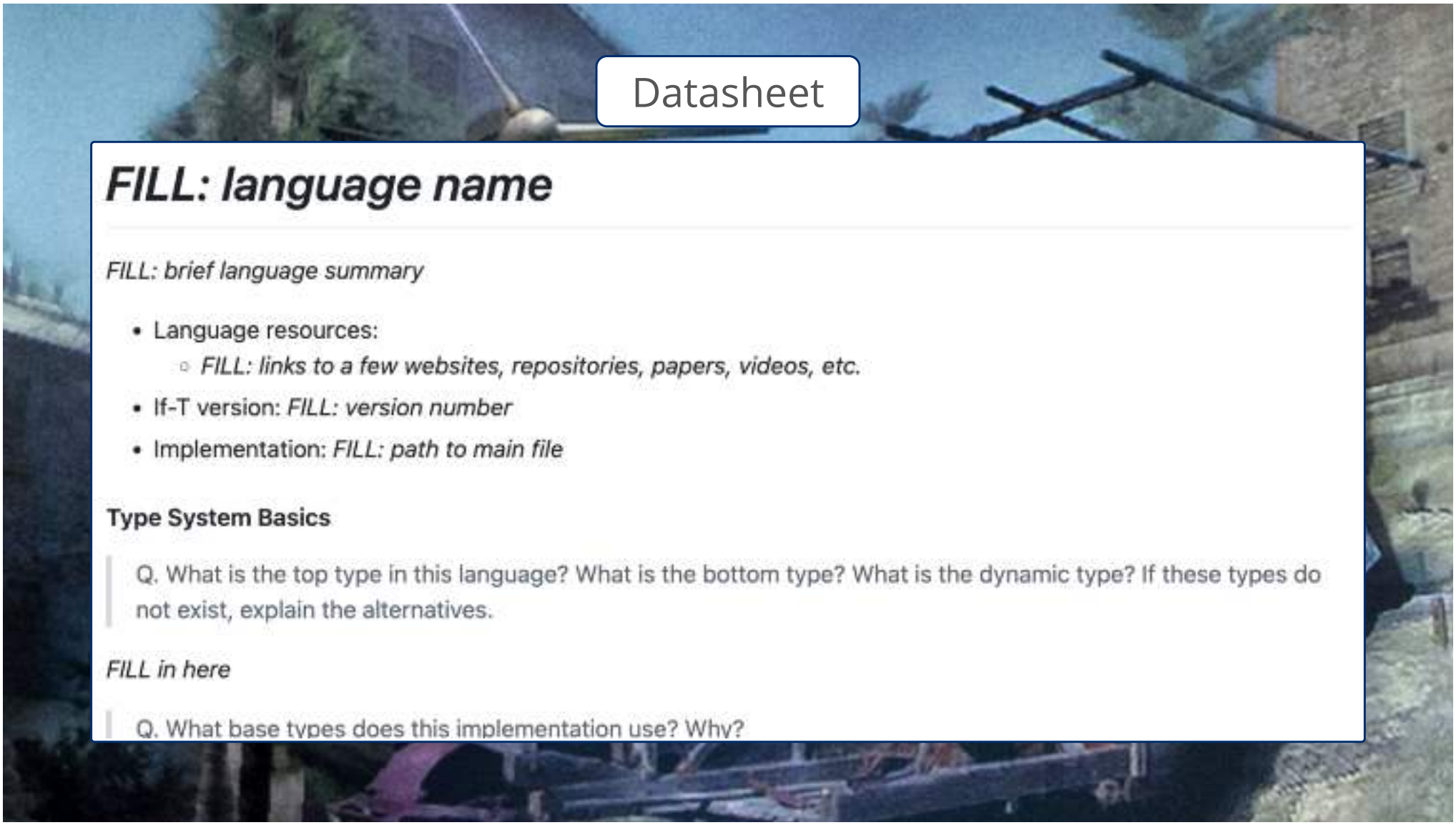
Basic
positive
negative
connectives
nesting_body

Compound Structur
struct_fields
tuple_elements
tuple_length

Advanced Control Flow
alias
nesting_condition
merge_with_union

Custom Predicates
predicate_2way
predicate_1way
predicate_checked

Not included:
Subtyping
Mutation
Concurrency



Datasheet

FILL: language name

FILL: brief language summary

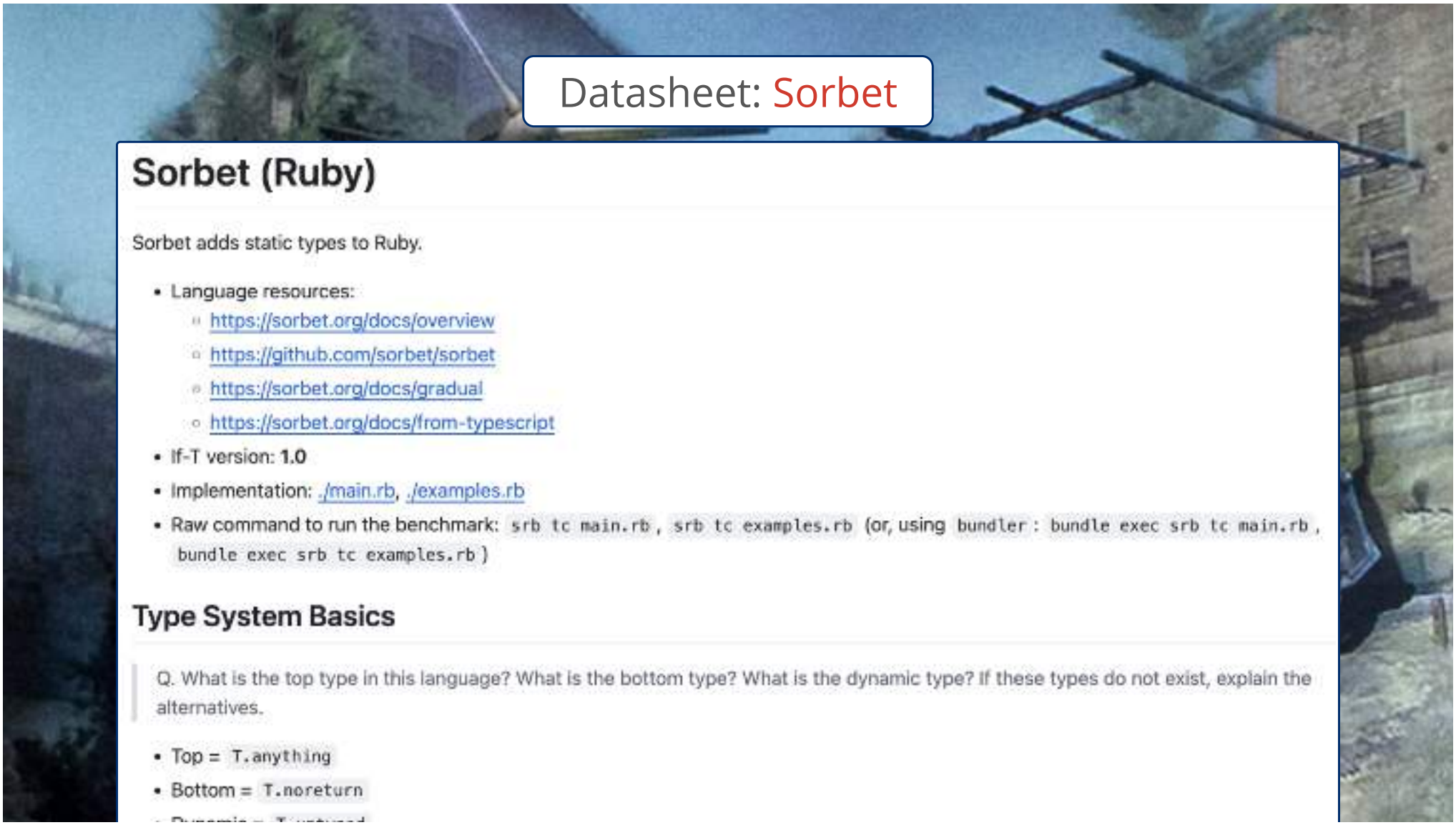
- Language resources:
  - *FILL: links to a few websites, repositories, papers, videos, etc.*
- If-T version: *FILL: version number*
- Implementation: *FILL: path to main file*

Type System Basics

> Q. What is the top type in this language? What is the bottom type? What is the dynamic type? If these types do not exist, explain the alternatives.

FILL in here

> Q. What base types does this implementation use? Why?



Datasheet: Sorbet

Sorbet (Ruby)

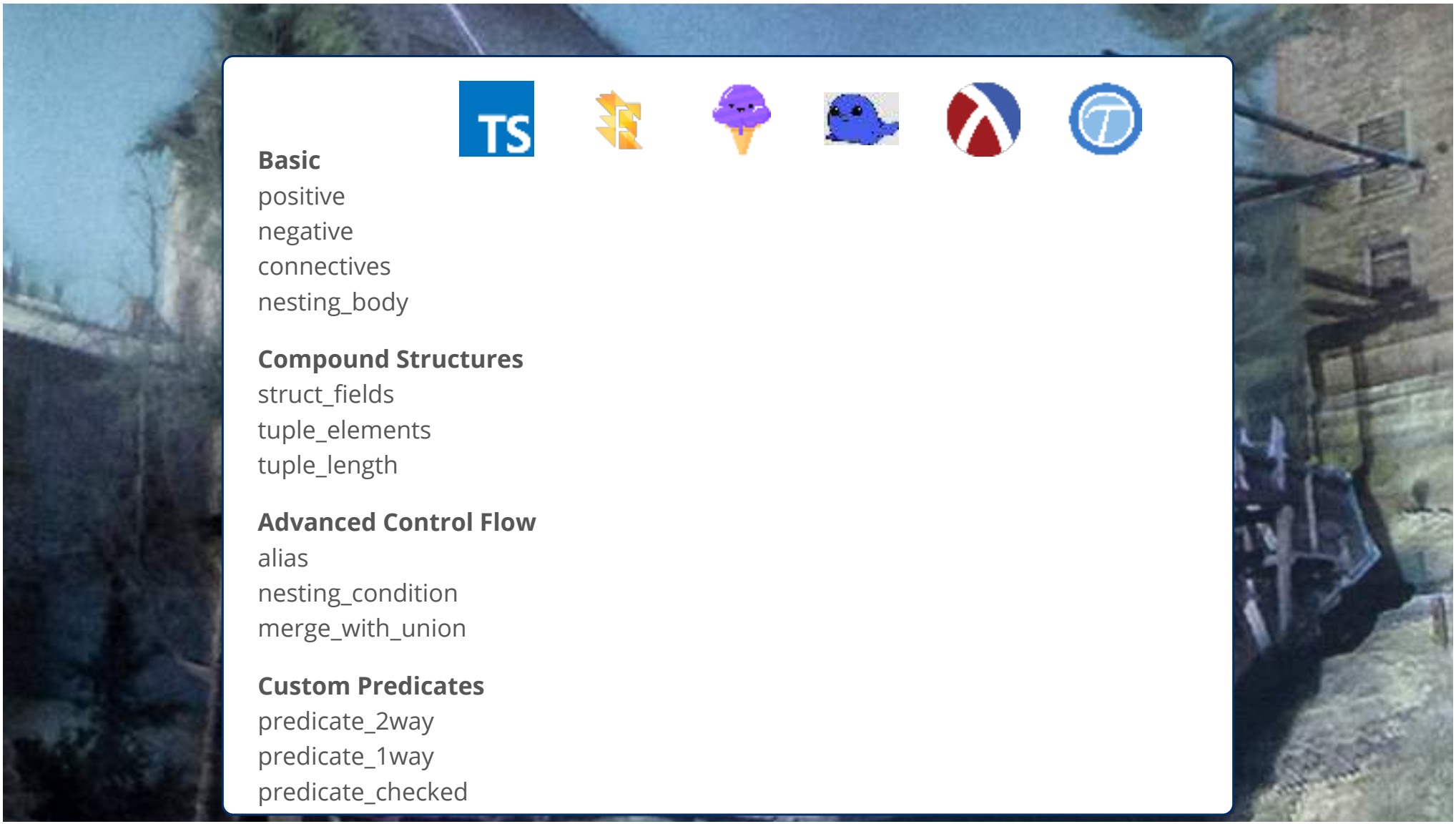
Sorbet adds static types to Ruby.

- Language resources:
  - https://sorbet.org/docs/overview
  - https://github.com/sorbet/sorbet
  - https://sorbet.org/docs/gradual
  - https://sorbet.org/docs/from-typescript
- If-T version: **1.0**
- Implementation: ./main.rb, ./examples.rb
- Raw command to run the benchmark: `srb tc main.rb`, `srb tc examples.rb` (or, using `bundler`: `bundle exec srb tc main.rb`, `bundle exec srb tc examples.rb`)

Type System Basics

> Q. What is the top type in this language? What is the bottom type? What is the dynamic type? If these types do not exist, explain the alternatives.

- Top = `T.anything`
- Bottom = `T.noreturn`

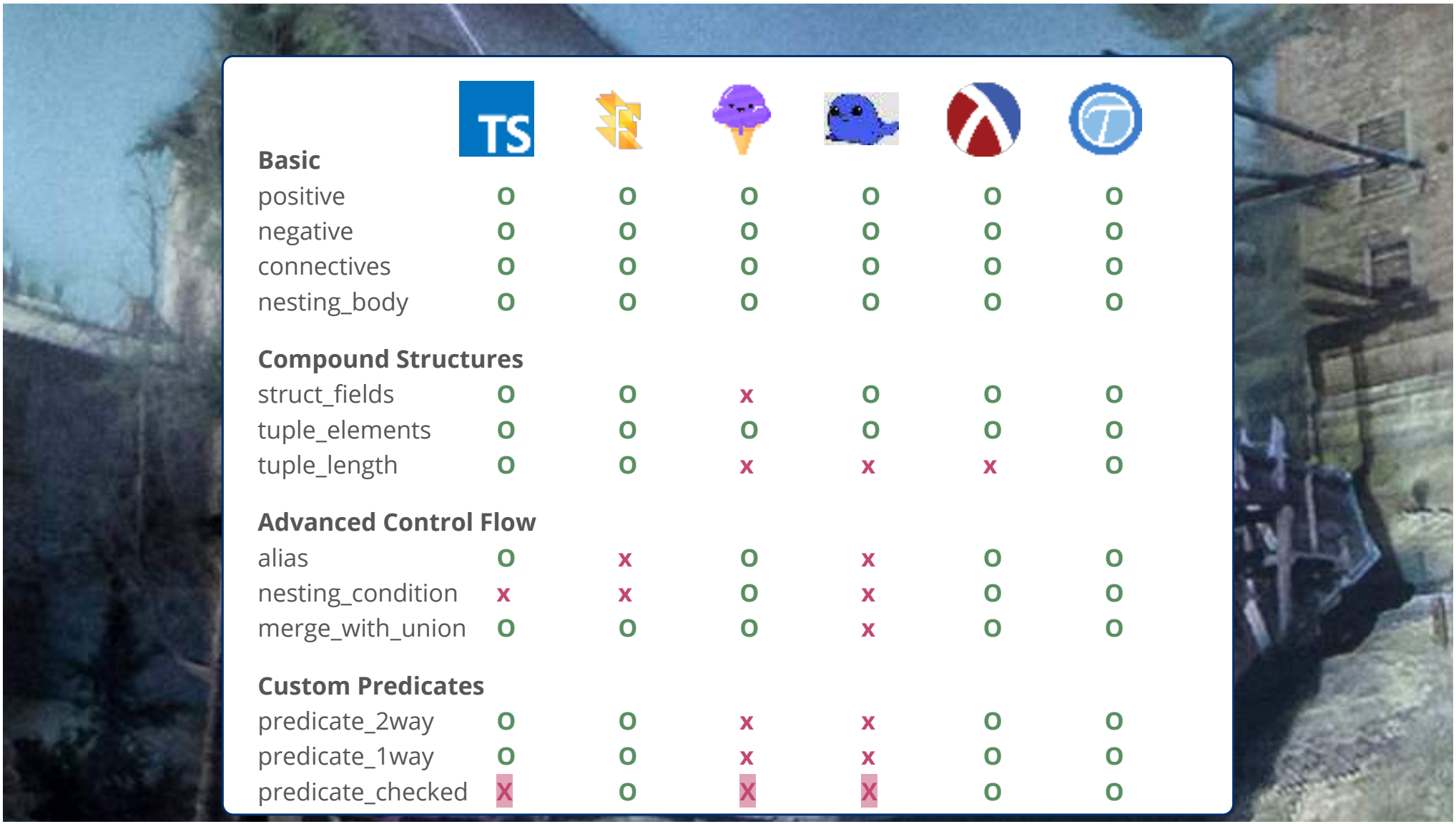


Basic
positive
negative
connectives
nesting_body

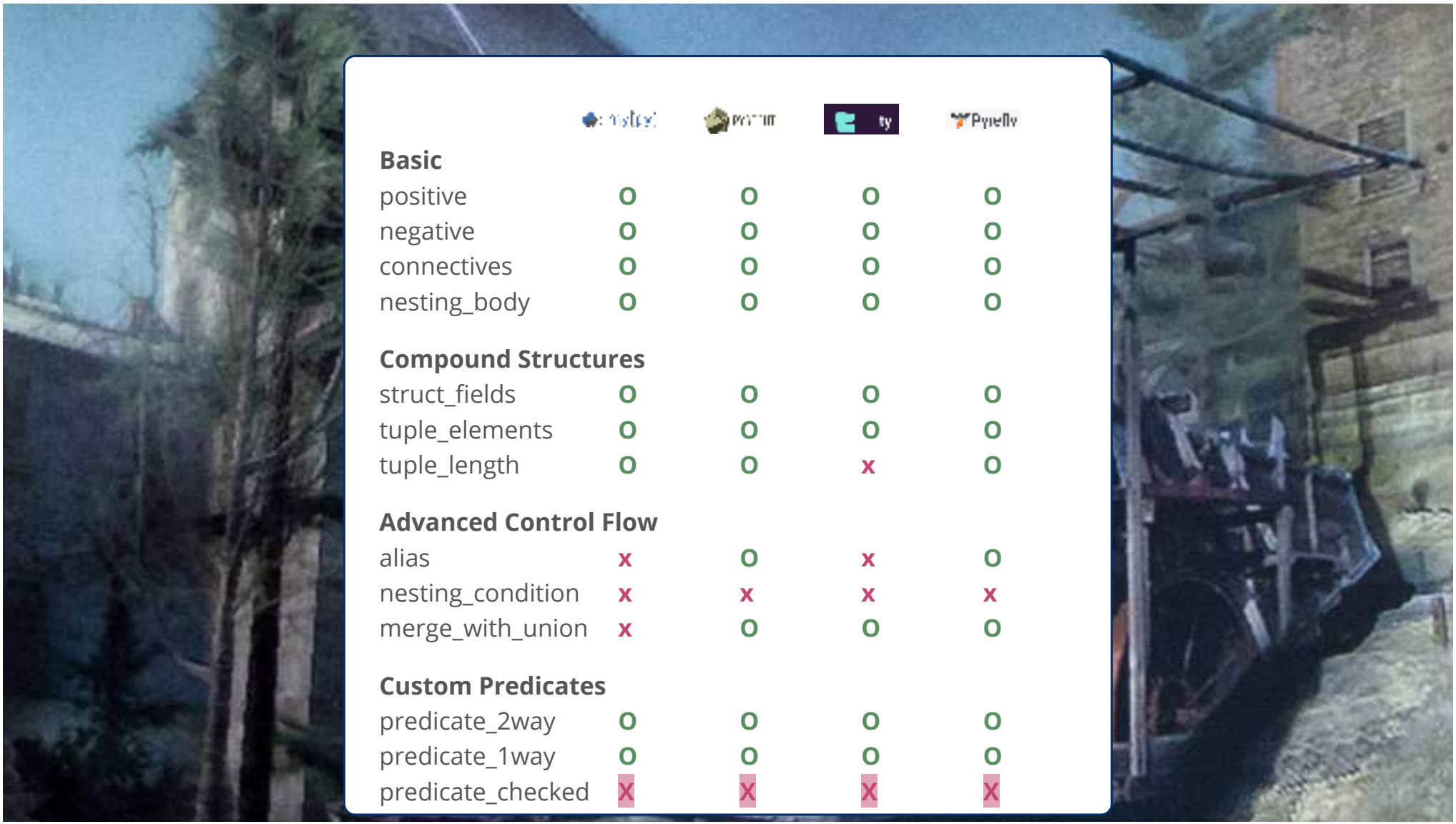
Compound Structures
struct_fields
tuple_elements
tuple_length

Advanced Control Flow
alias
nesting_condition
merge_with_union

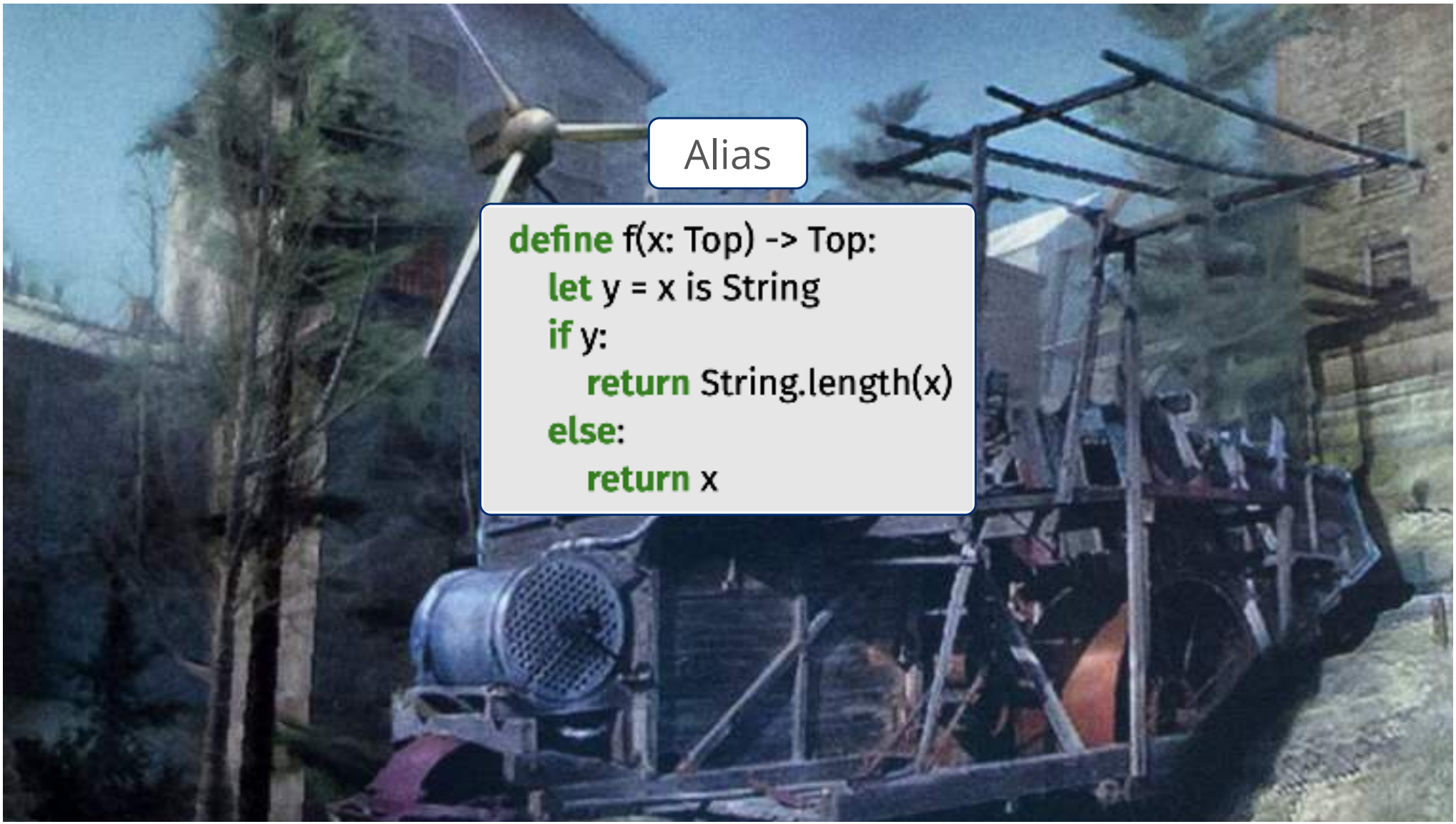
Custom Predicates
predicate_2way
predicate_1way
predicate_checked

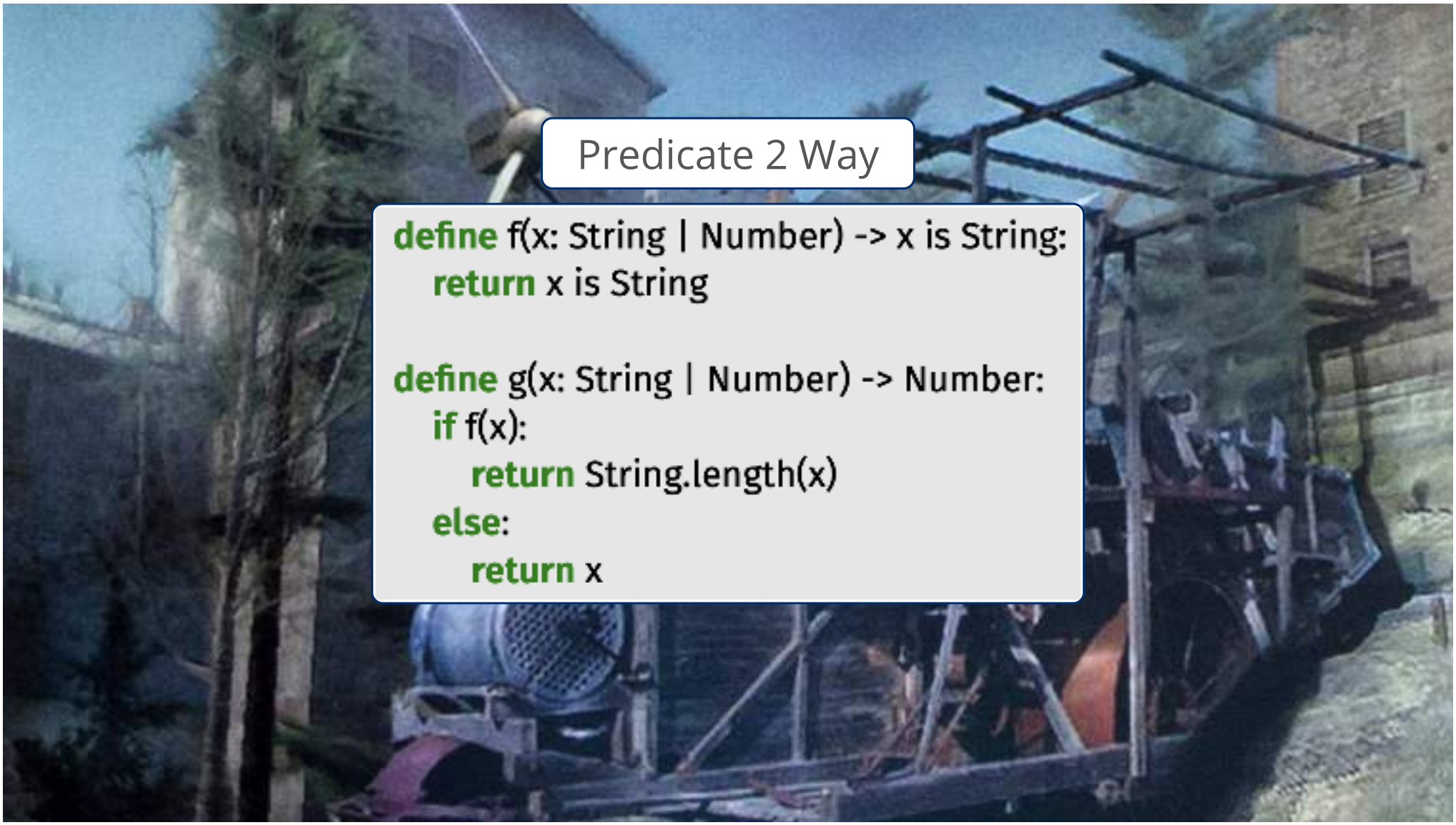


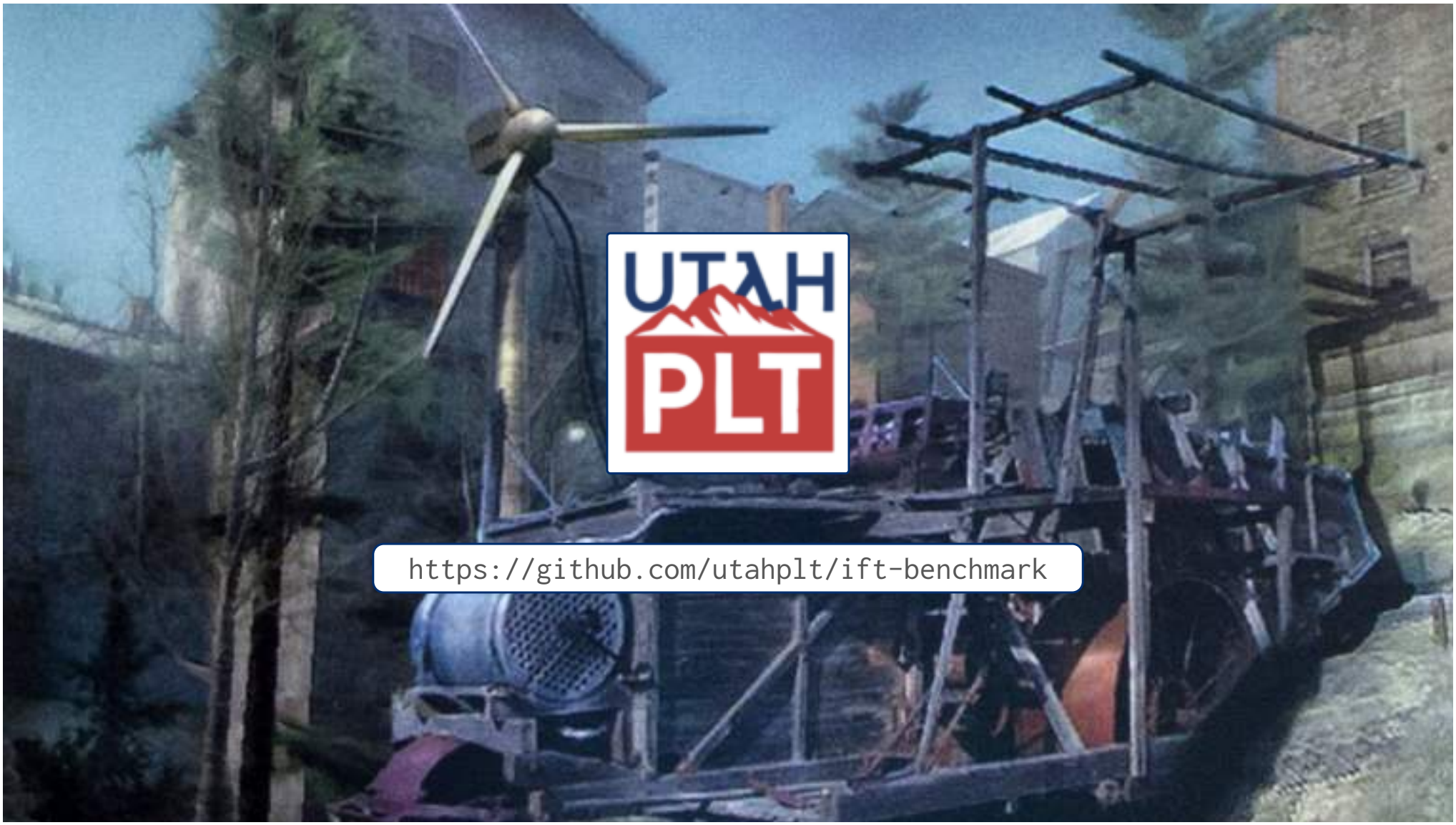
	TS					
Basic						
positive	O	O	O	O	O	O
negative	O	O	O	O	O	O
connectives	O	O	O	O	O	O
nesting_body	O	O	O	O	O	O
Compound Structures						
struct_fields	O	O	x	O	O	O
tuple_elements	O	O	O	O	O	O
tuple_length	O	O	x	x	x	O
Advanced Control Flow						
alias	O	x	O	x	O	O
nesting_condition	x	x	O	x	O	O
merge_with_union	O	O	O	x	O	O
Custom Predicates						
predicate_2way	O	O	x	x	O	O
predicate_1way	O	O	x	x	O	O
predicate_checked	X	O	X	X	O	O

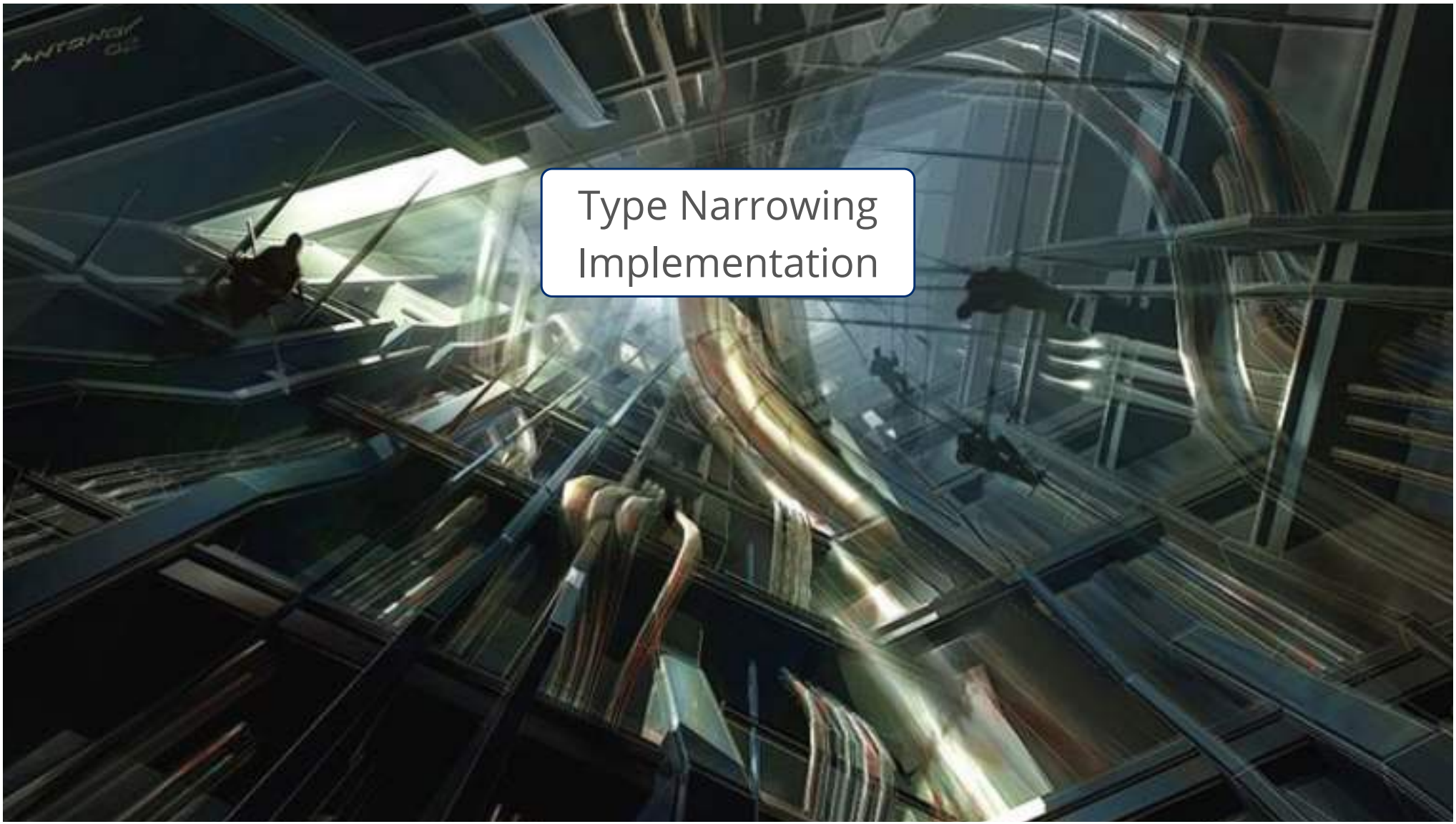


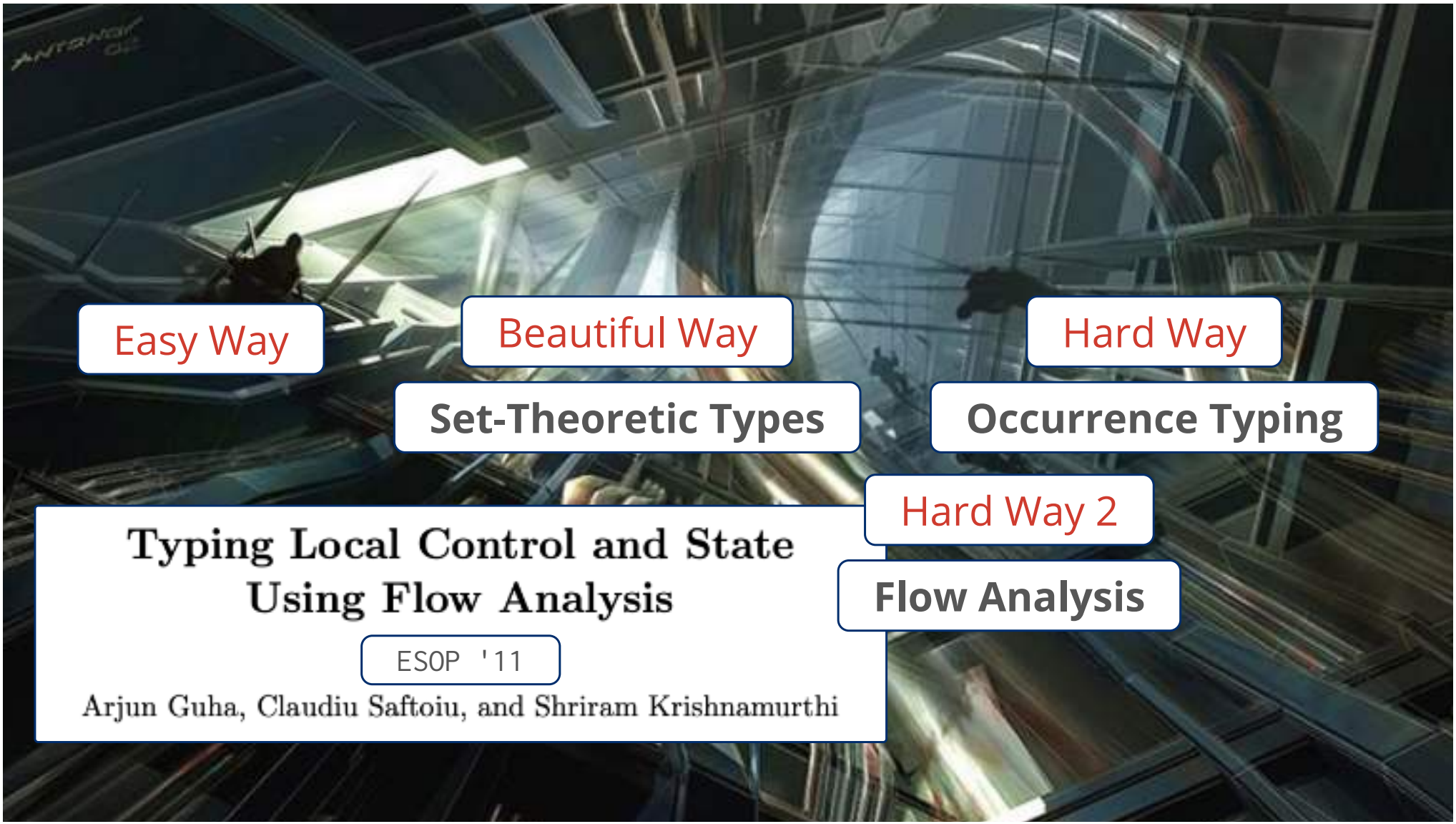
			ty	
Basic				
positive	O	O	O	O
negative	O	O	O	O
connectives	O	O	O	O
nesting_body	O	O	O	O
Compound Structures				
struct_fields	O	O	O	O
tuple_elements	O	O	O	O
tuple_length	O	O	x	O
Advanced Control Flow				
alias	x	O	x	O
nesting_condition	x	x	x	x
merge_with_union	x	O	O	O
Custom Predicates				
predicate_2way	O	O	O	O
predicate_1way	O	O	O	O
predicate_checked	X	X	X	X


Alias
define f(x: Top) -> Top:
let y = x is String
if y:
return String.length(x)
else:
return x


Predicate 2 Way
define f(x: String | Number) -> x is String:
return x is String
define g(x: String | Number) -> Number:
if f(x):
return String.length(x)
else:
return x

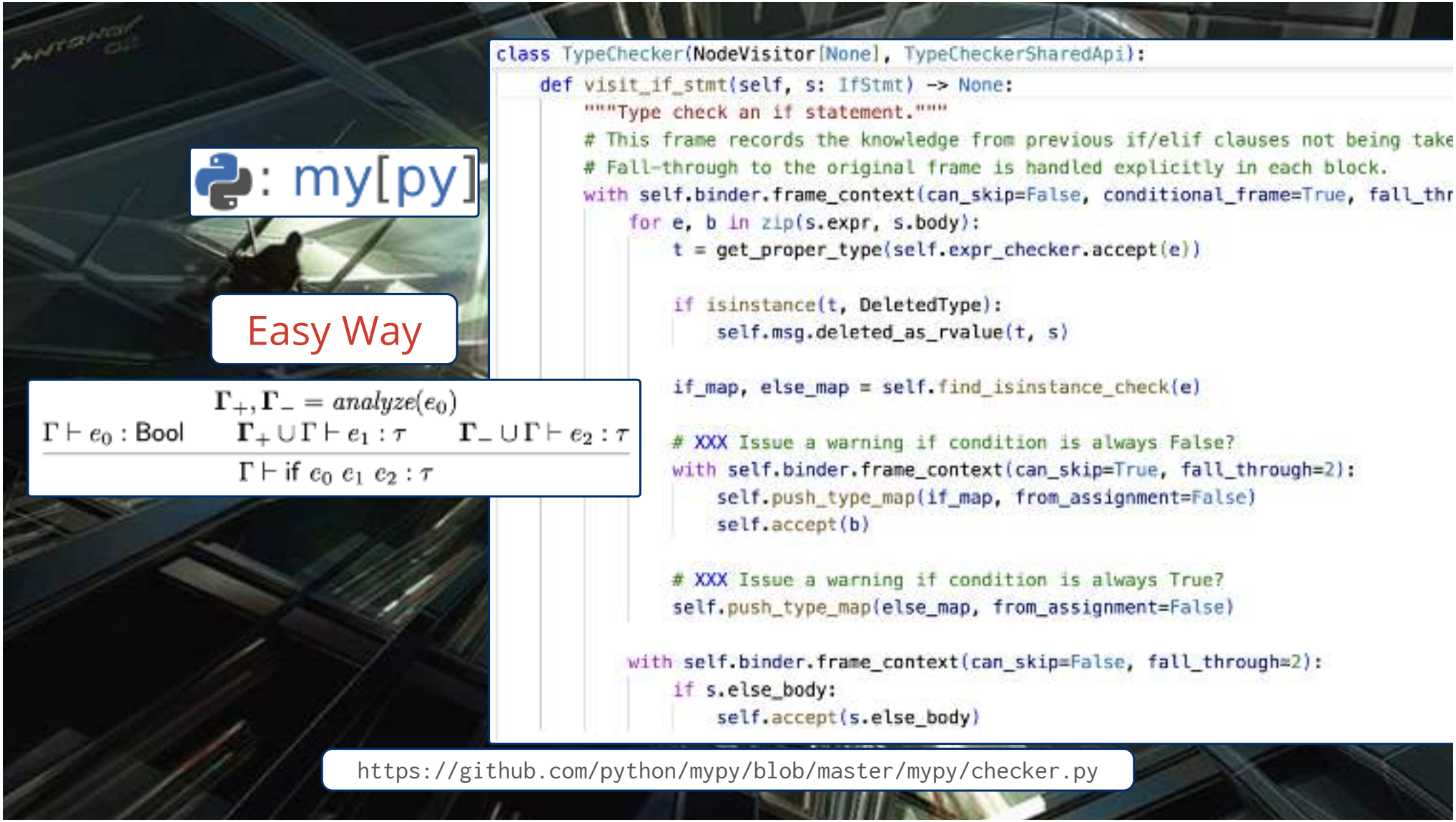

UTAH
PLT
https://github.com/utahplt/ift-benchmark

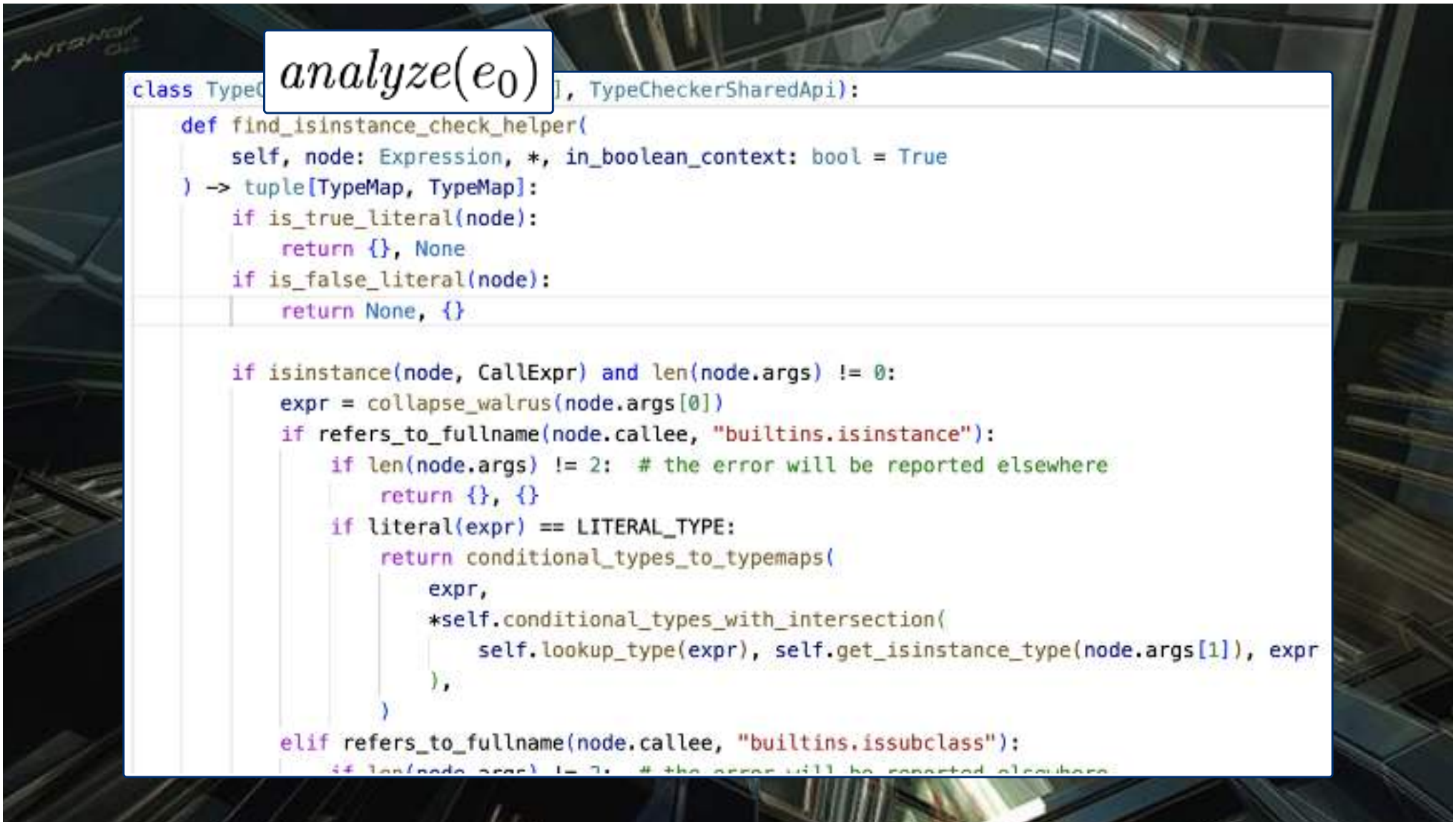

Type Narrowing
Implementation

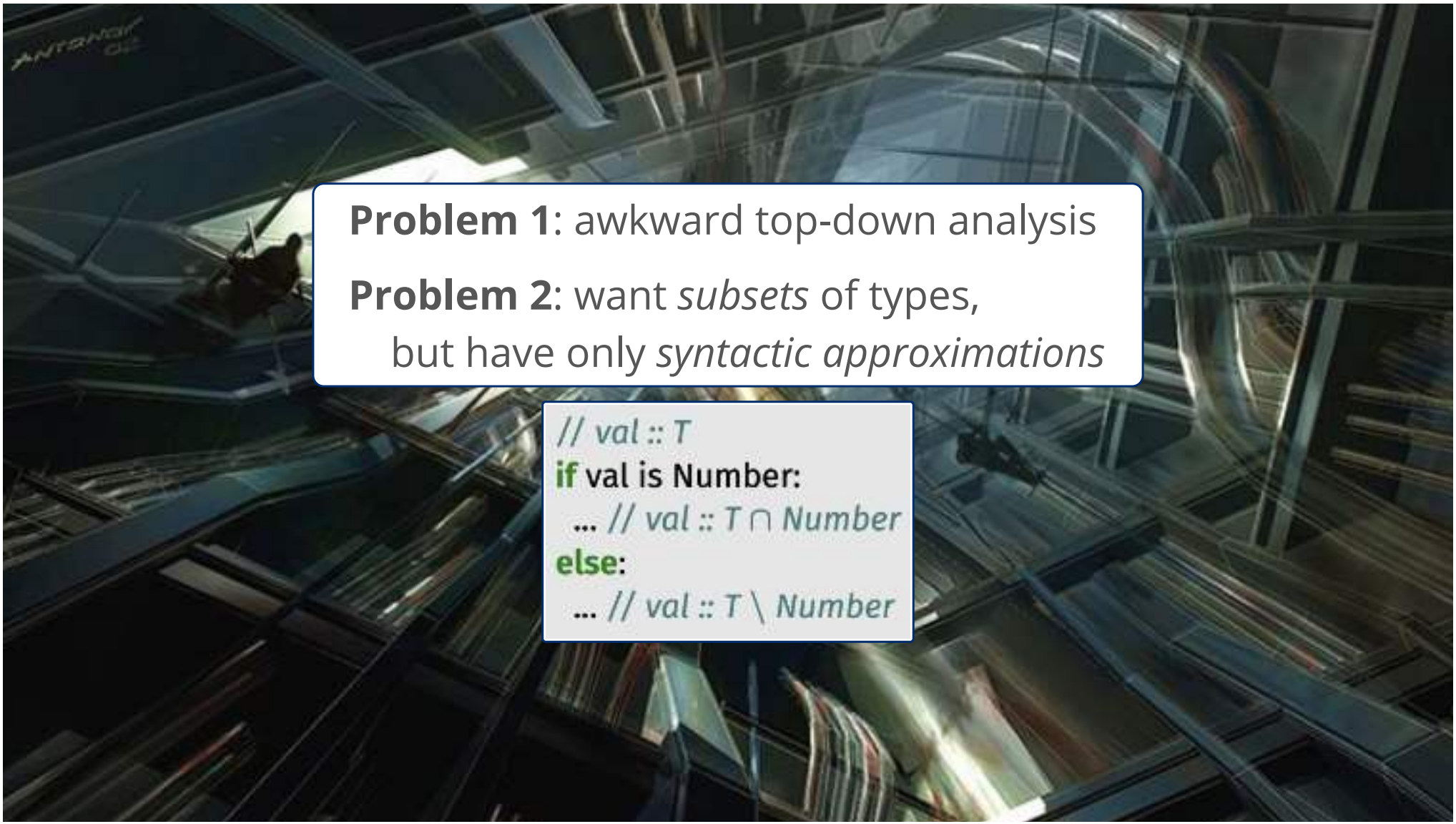

Easy Way
Beautiful Way
Hard Way
Set-Theoretic Types
Occurrence Typing
Hard Way 2
Flow Analysis
Typing Local Control and State Using Flow Analysis
ESOP '11
Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi


Easy Way
Beautiful Way
Hard Way
Set-Theoretic Types
Occurrence Typing

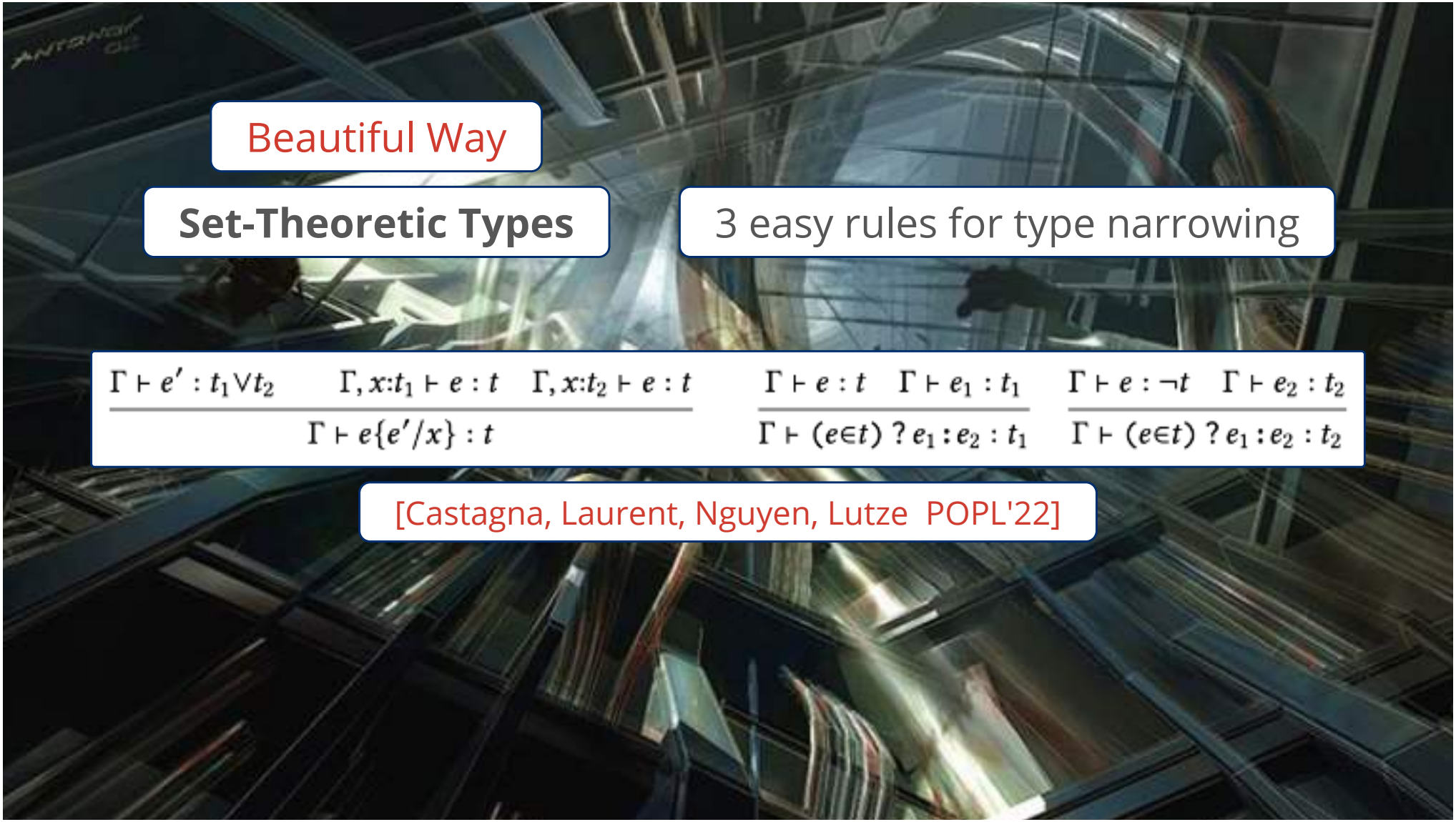

: my[py]
Easy Way
$$\frac{\Gamma \vdash e_0 : \text{Bool} \qquad \Gamma_+, \Gamma_- = analyze(e_0) \qquad \Gamma_+ \cup \Gamma \vdash e_1 : \tau \qquad \Gamma_- \cup \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_0\ e_1\ e_2 : \tau}$$

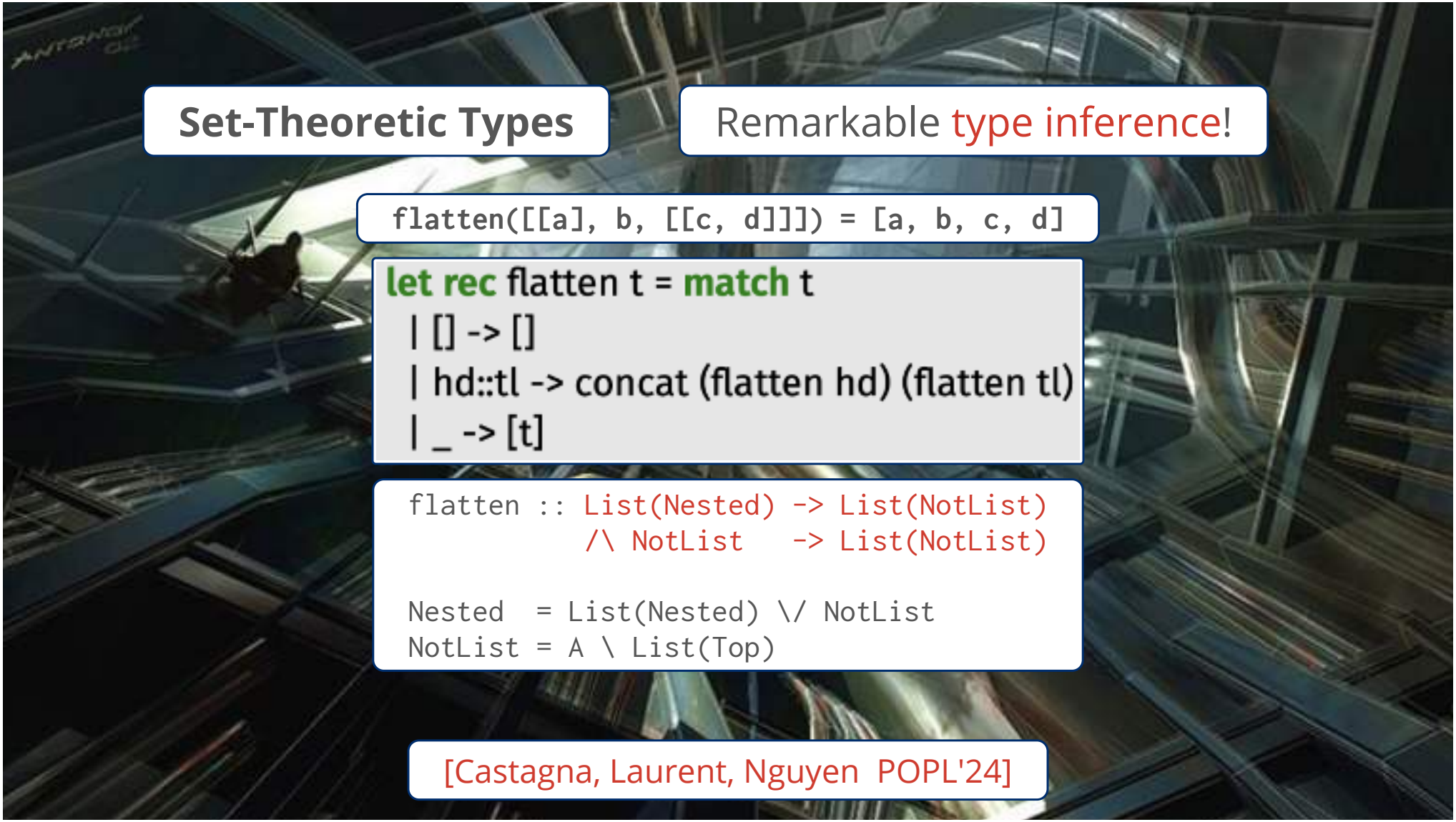

: my[py]
Easy Way
$$\frac{\Gamma \vdash e_0 : \text{Bool} \qquad \Gamma_+, \Gamma_- = analyze(e_0) \qquad \Gamma_+ \cup \Gamma \vdash e_1 : \tau \qquad \Gamma_- \cup \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_0\ e_1\ e_2 : \tau}$$
class TypeChecker(NodeVisitor[None], TypeCheckerSharedApi):
def visit_if_stmt(self, s: IfStmt) -> None:
"""Type check an if statement."""
This frame records the knowledge from previous if/elif clauses not being take
Fall-through to the original frame is handled explicitly in each block.
with self.binder.frame_context(can_skip=False, conditional_frame=True, fall_thr
for e, b in zip(s.expr, s.body):
t = get_proper_type(self.expr_checker.accept(e))
if isinstance(t, DeletedType):
self.msg.deleted_as_rvalue(t, s)
if_map, else_map = self.find_isinstance_check(e)
XXX Issue a warning if condition is always False?
with self.binder.frame_context(can_skip=True, fall_through=2):
self.push_type_map(if_map, from_assignment=False)
self.accept(b)
# XXX Issue a warning if condition is always True?
self.push_type_map(else_map, from_assignment=False)
with self.binder.frame_context(can_skip=False, fall_through=2):
if s.else_body:
self.accept(s.else_body)
https://github.com/python/mypy/blob/master/mypy/checker.py

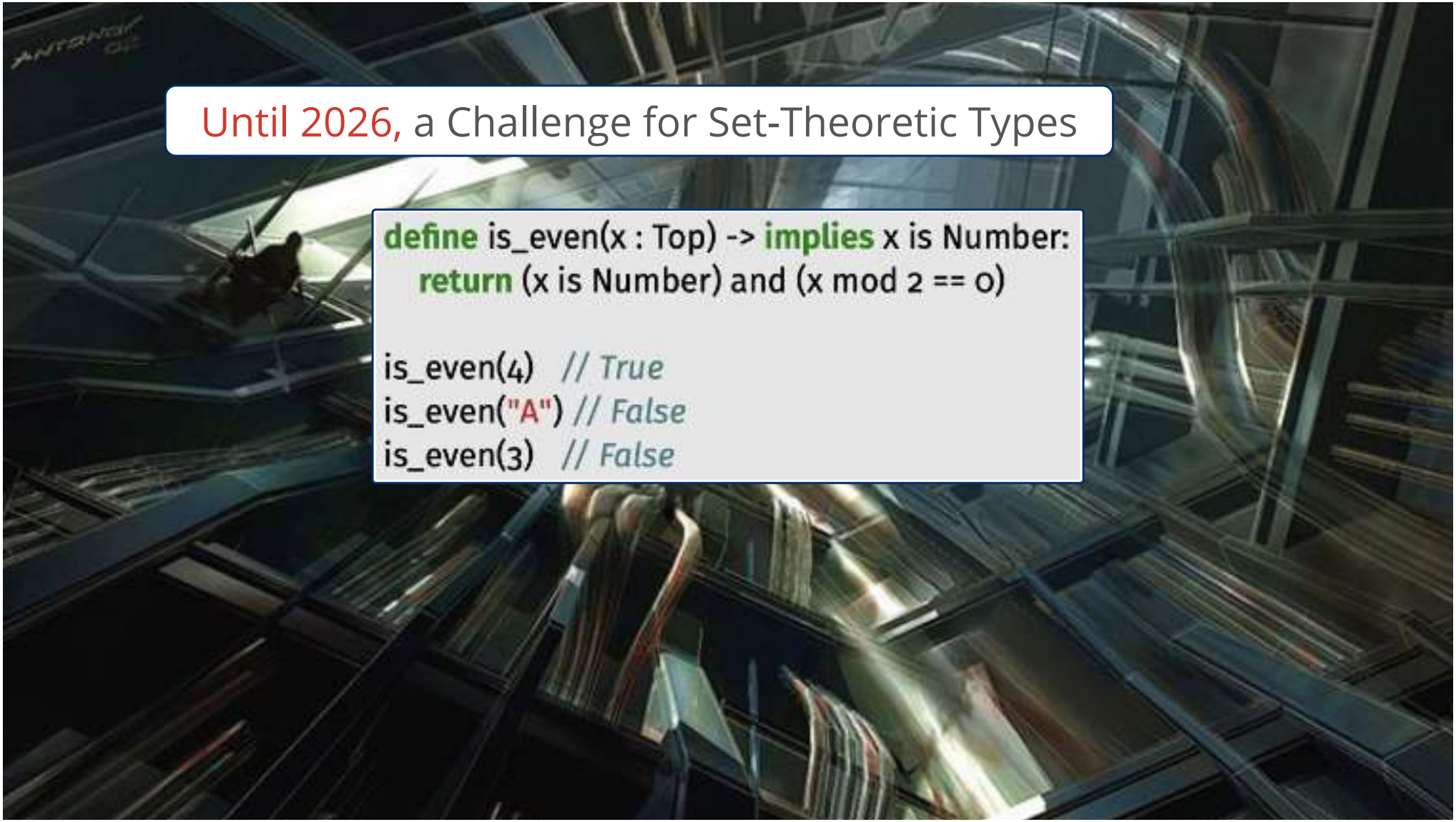

$analyze(e_0)$
class TypeC], TypeCheckerSharedApi):
def find_isinstance_check_helper(
self, node: Expression, *, in_boolean_context: bool = True
) -> tuple[TypeMap, TypeMap]:
if is_true_literal(node):
return {}, None
if is_false_literal(node):
return None, {}
if isinstance(node, CallExpr) and len(node.args) != 0:
expr = collapse_walrus(node.args[0])
if refers_to_fullname(node.callee, "builtins.isinstance"):
if len(node.args) != 2: # the error will be reported elsewhere
return {}, {}
if literal(expr) == LITERAL_TYPE:
return conditional_types_to_typemaps(
expr,
*self.conditional_types_with_intersection(
self.lookup_type(expr), self.get_isinstance_type(node.args[1]), expr
),
)
elif refers_to_fullname(node.callee, "builtins.issubclass"):

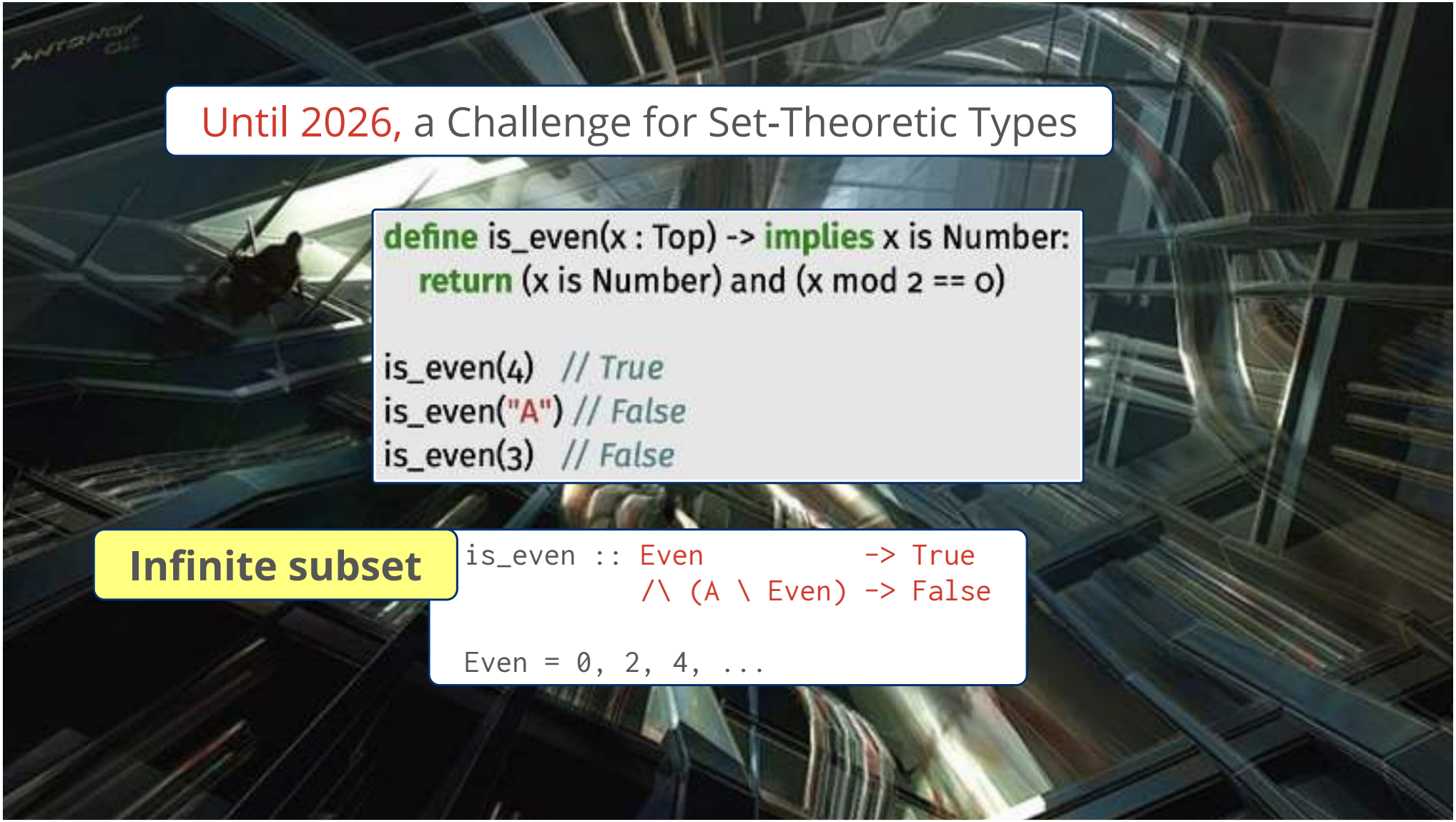

Problem 1: awkward top-down analysis
Problem 2: want *subsets* of types,
but have only *syntactic approximations*
// val :: T
if val is Number:
... // val :: T ∩ Number
else:
... // val :: T \ Number

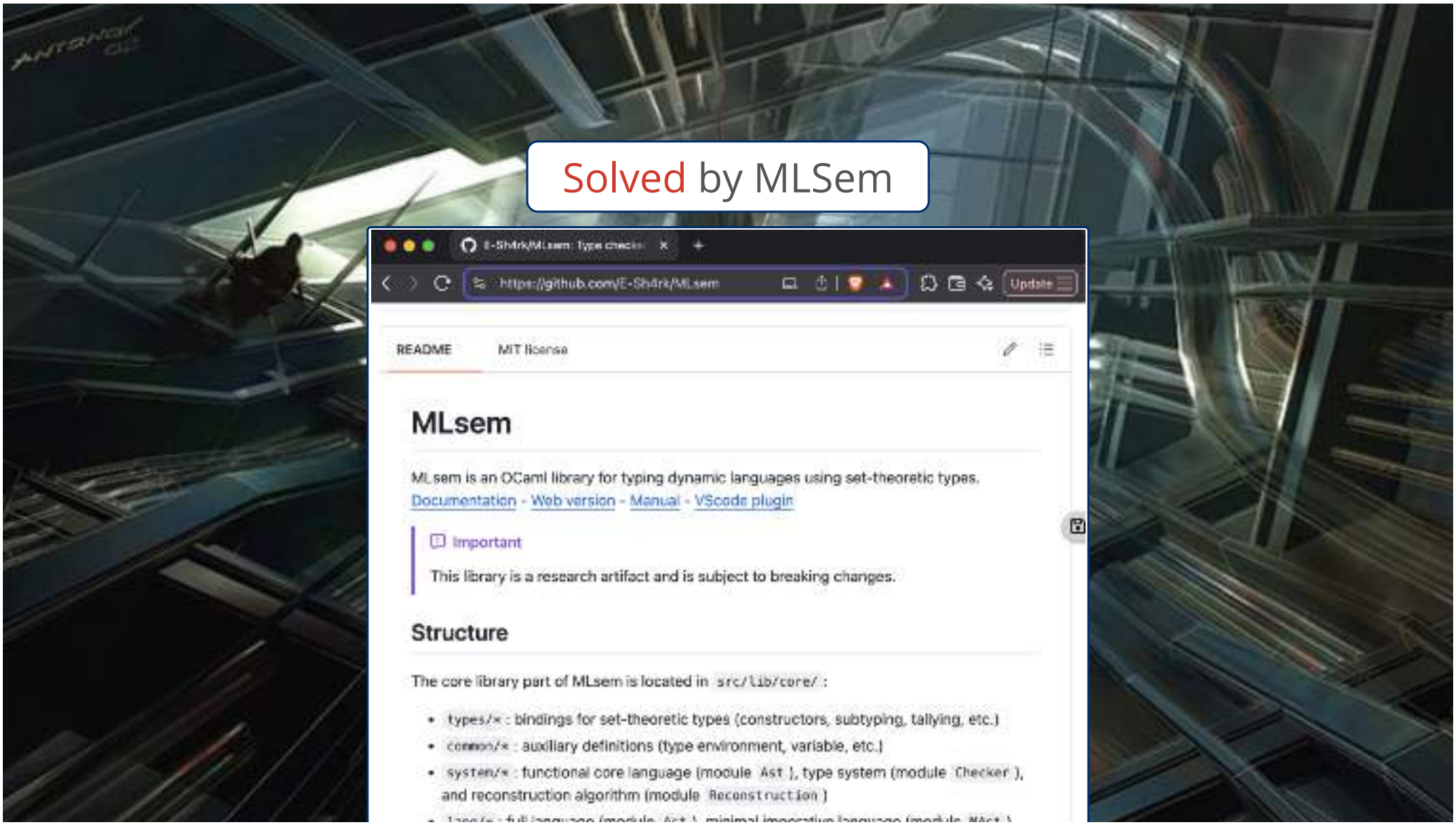

Beautiful Way
Set-Theoretic Types
τ ::= ... | τ ∪ τ | τ ∩ τ | ¬τ | τ \ τ


Beautiful Way
Set-Theoretic Types
3 easy rules for type narrowing
$$\frac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x{:}t_1 \vdash e : t \qquad \Gamma, x{:}t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$
$$\frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : t_1} \qquad \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : t_2}$$
[Castagna, Laurent, Nguyen, Lutze POPL'22]


Set-Theoretic Types
Remarkable type inference!
flatten([[a], b, [[c, d]]]) = [a, b, c, d]
let rec flatten t = match t
| [] -> []
| hd::tl -> concat (flatten hd) (flatten tl)
| _ -> [t]
flatten :: List(Nested) -> List(NotList)
/\ NotList -> List(NotList)
Nested = List(Nested) \/ NotList
NotList = A \ List(Top)
[Castagna, Laurent, Nguyen POPL'24]


Until 2026, a Challenge for Set-Theoretic Types
define is_even(x : Top) -> implies x is Number:
return (x is Number) and (x mod 2 == 0)
is_even(4) // True
is_even("A") // False
is_even(3) // False


Until 2026, a Challenge for Set-Theoretic Types
define is_even(x : Top) -> implies x is Number:
return (x is Number) and (x mod 2 == 0)
is_even(4) // True
is_even("A") // False
is_even(3) // False
Infinite subset
is_even :: Even -> True
/\ (A \ Even) -> False
Even = 0, 2, 4, ...


Solved by MLSem
https://github.com/E-Sh4rk/MLsem
Update
README
MIT license
MLsem
MLsem is an OCaml library for typing dynamic languages using set-theoretic types.
Documentation - Web version - Manual - VScode plugin
Important
This library is a research artifact and is subject to breaking changes.
Structure
The core library part of MLsem is located in src/lib/core/ :
types/* : bindings for set-theoretic types (constructors, subtyping, tallying, etc.)
common/* : auxiliary definitions (type environment, variable, etc.)
system/* : functional core language (module Ast), type system (module Checker), and reconstruction algorithm (module Reconstruction)

MLSem and more at OOPSLA next month!

Mickaël LAURENT

ABOUT

Hi! 👋 I am an associate professor at Sorbonne Université (France), within the team APR of the LIP6.

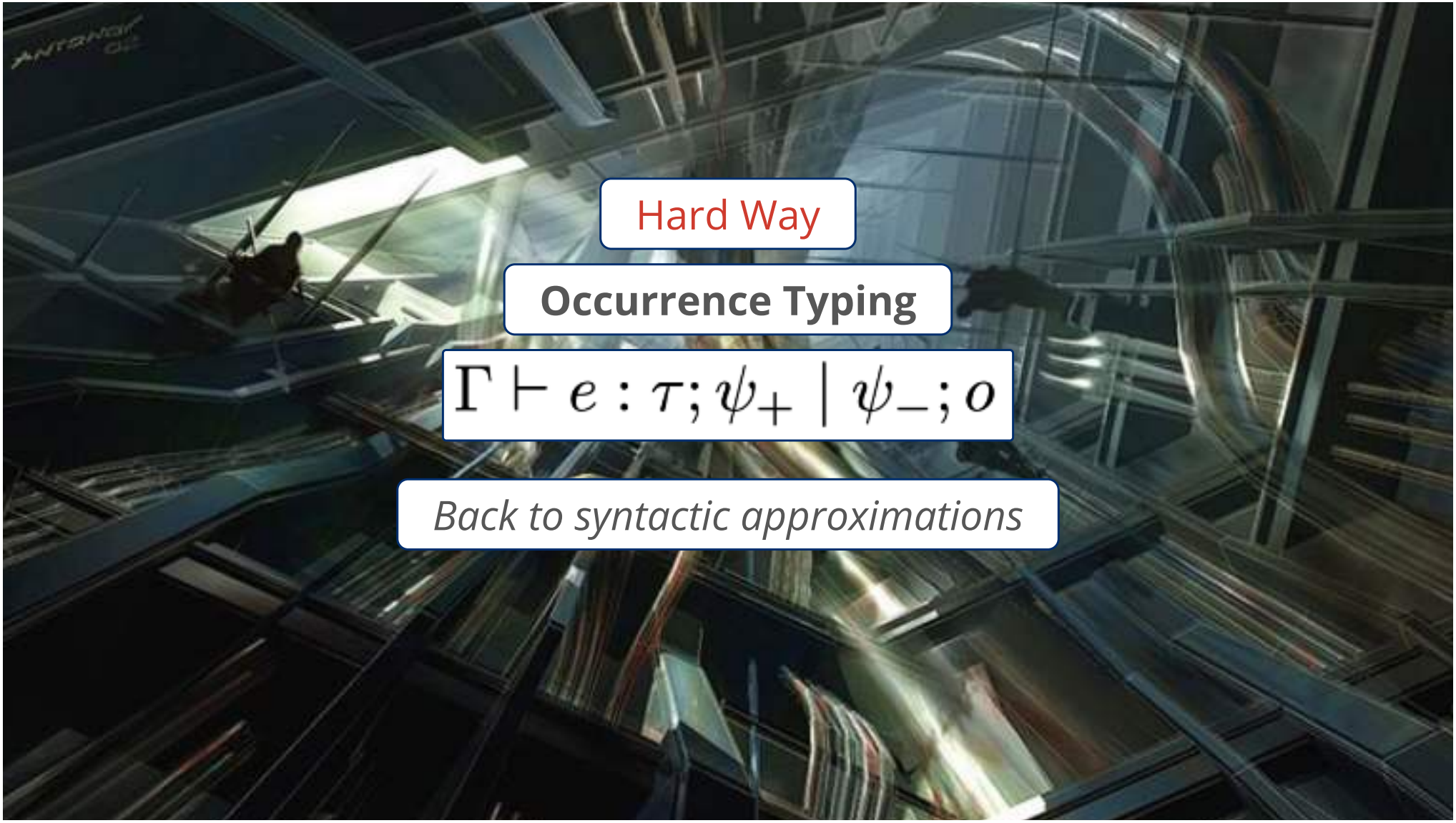
I'm interested in dynamic programming languages and type systems.

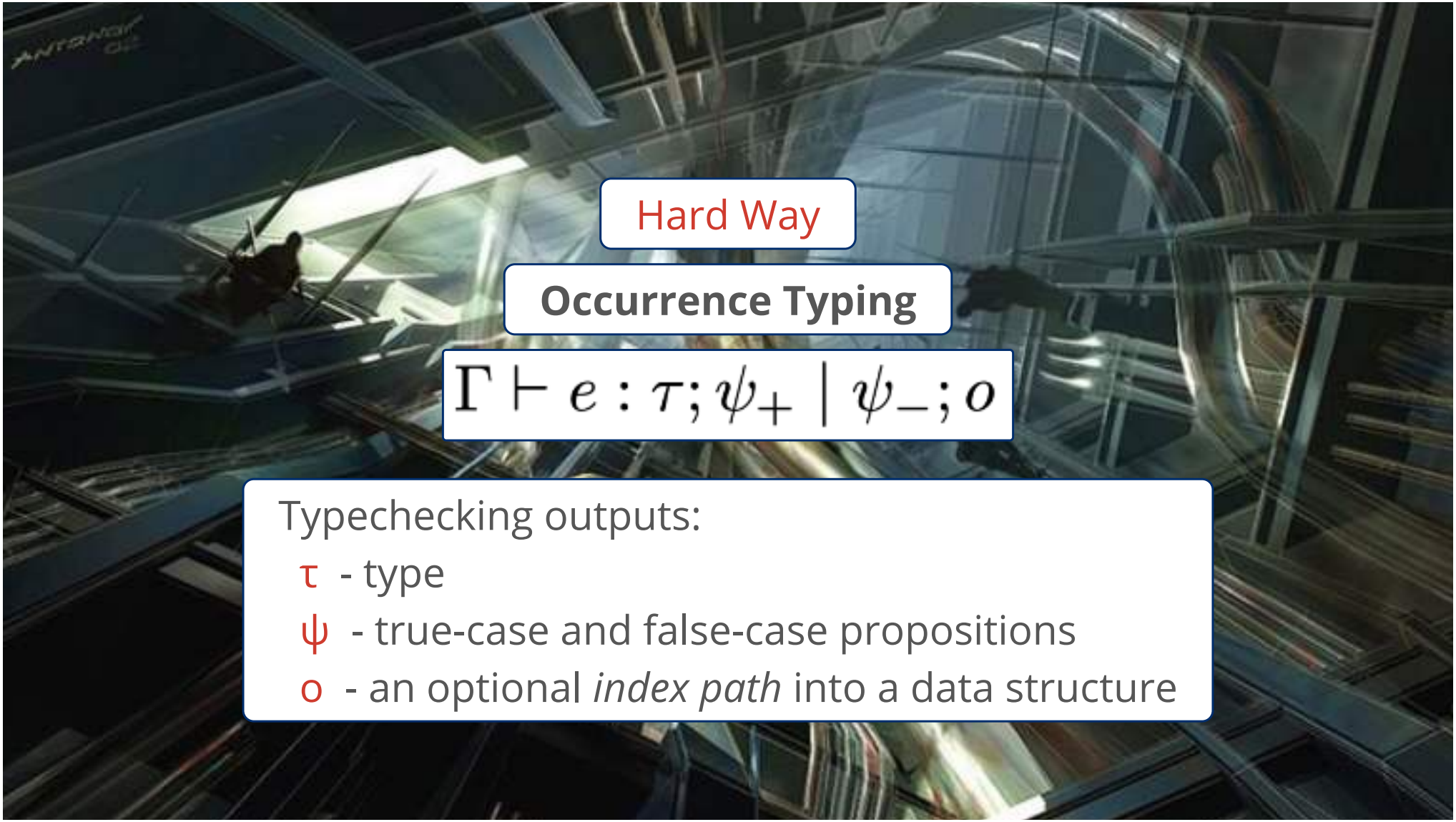
I defended my PhD at IRIF under the supervision of Giuseppe Castagna and Kim Nguyễn, and won first place in the 2025 Thesis Award

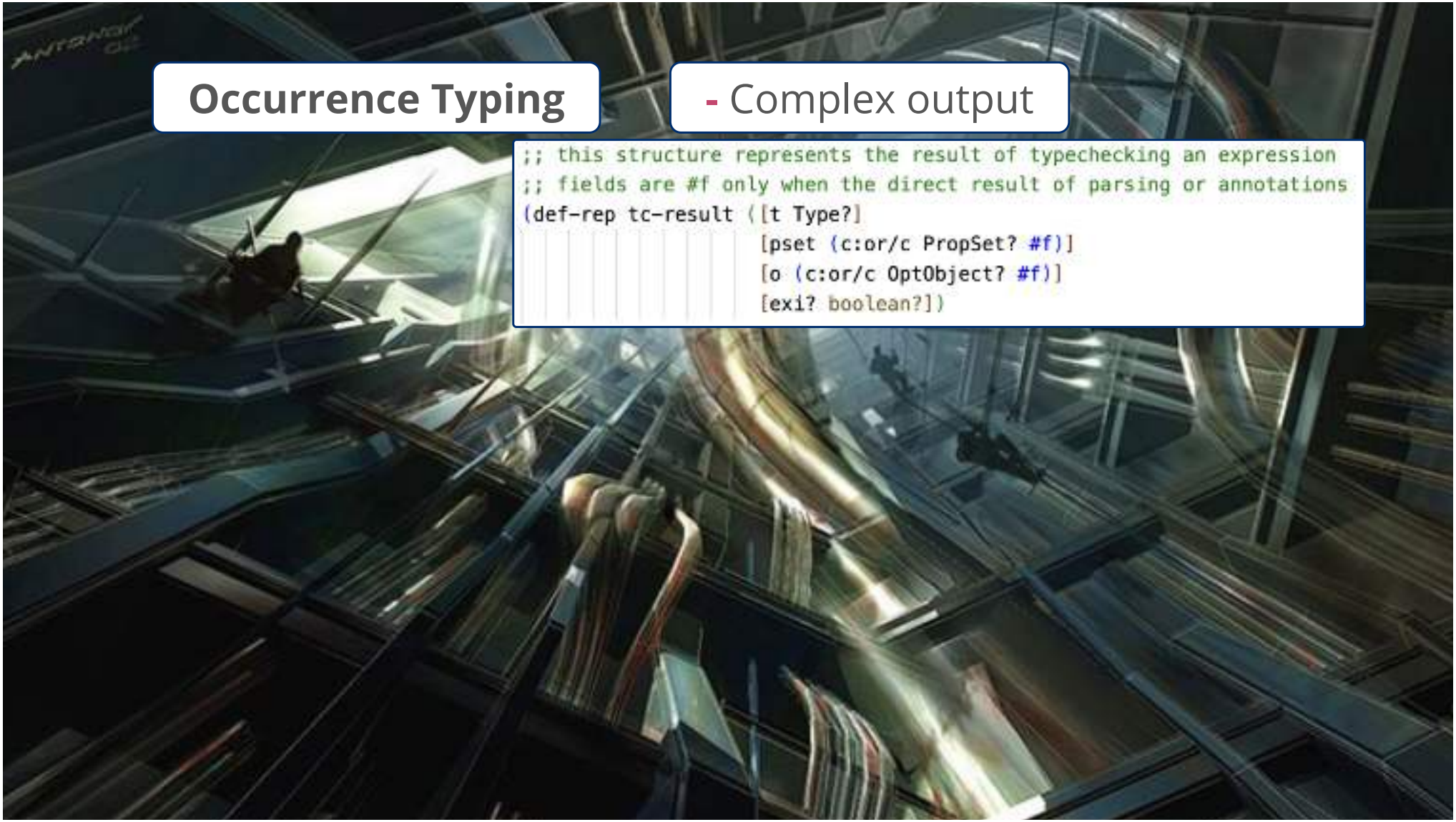
2026

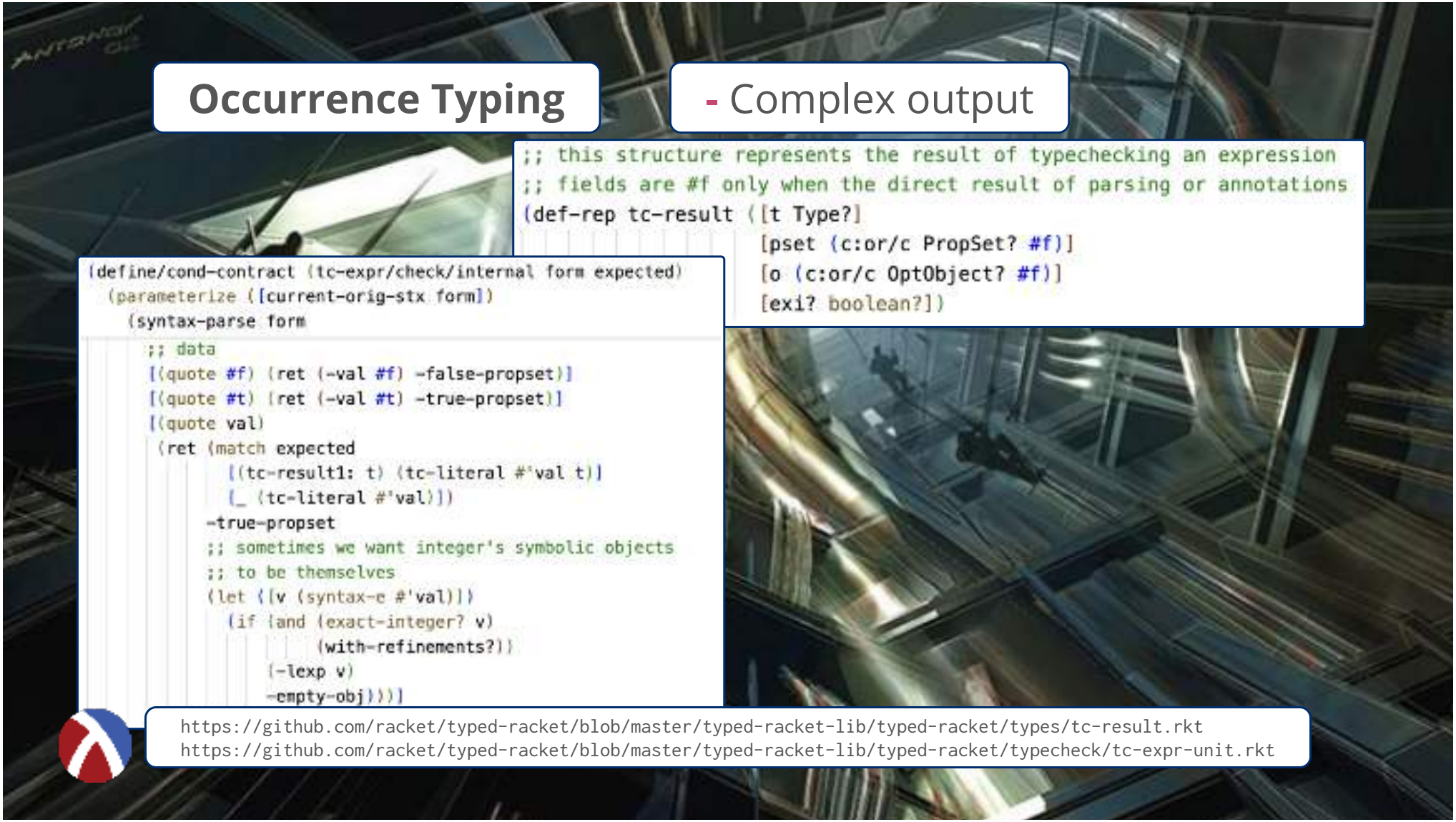
SPLASH

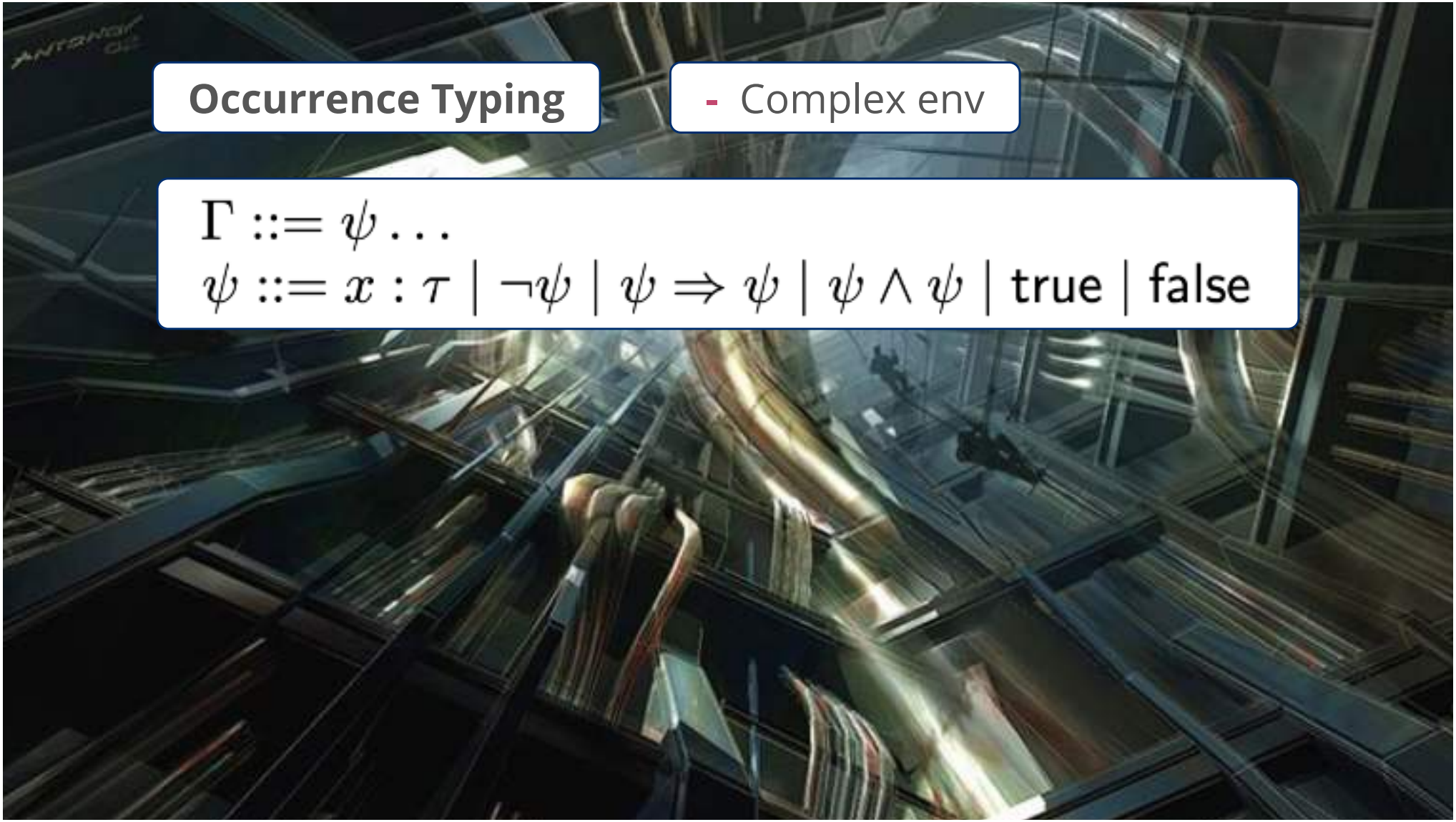
- Revisiting Row Polymorphism for Set-Theoretic Types
- Implementing Set-Theoretic Types
- Type inference for functional and imperative dynamic languages

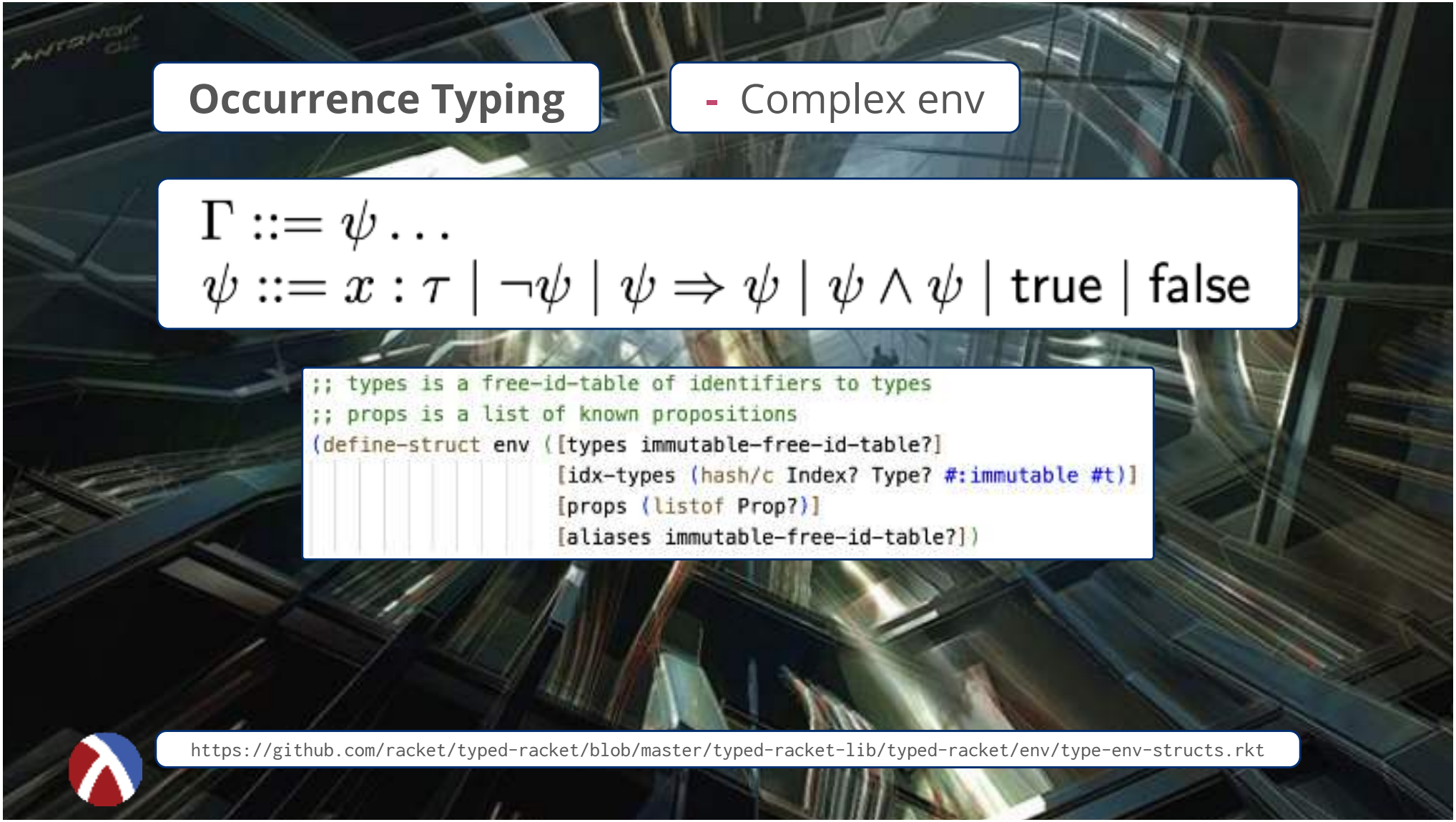

Hard Way
Occurrence Typing
$\Gamma \vdash e : \tau; \psi_+ \mid \psi_-; o$
Back to syntactic approximations

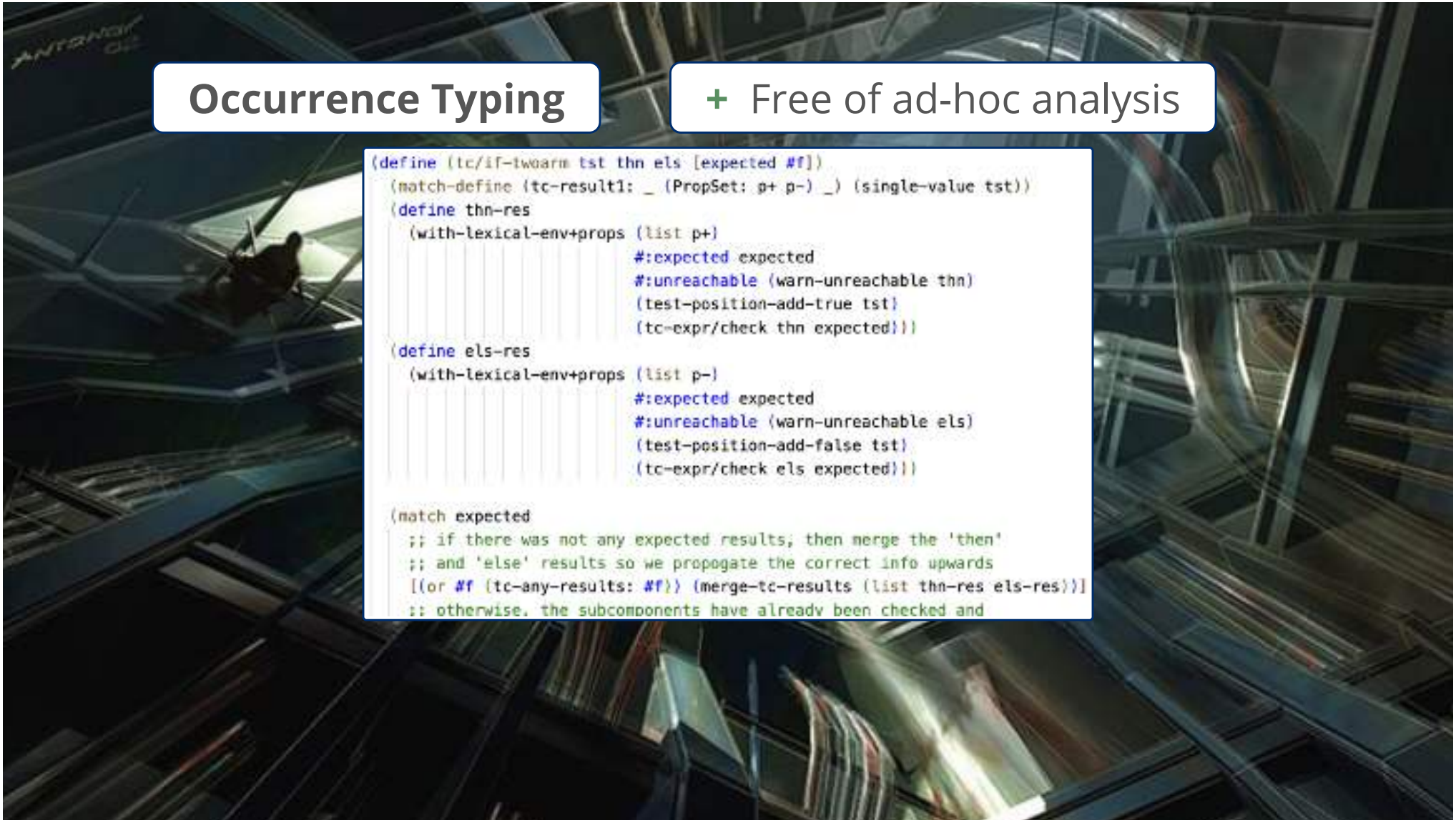

Hard Way
Occurrence Typing
$\Gamma \vdash e : \tau; \psi_+ \mid \psi_-; o$
Typechecking outputs:
τ - type
ψ - true-case and false-case propositions
o - an optional *index path* into a data structure

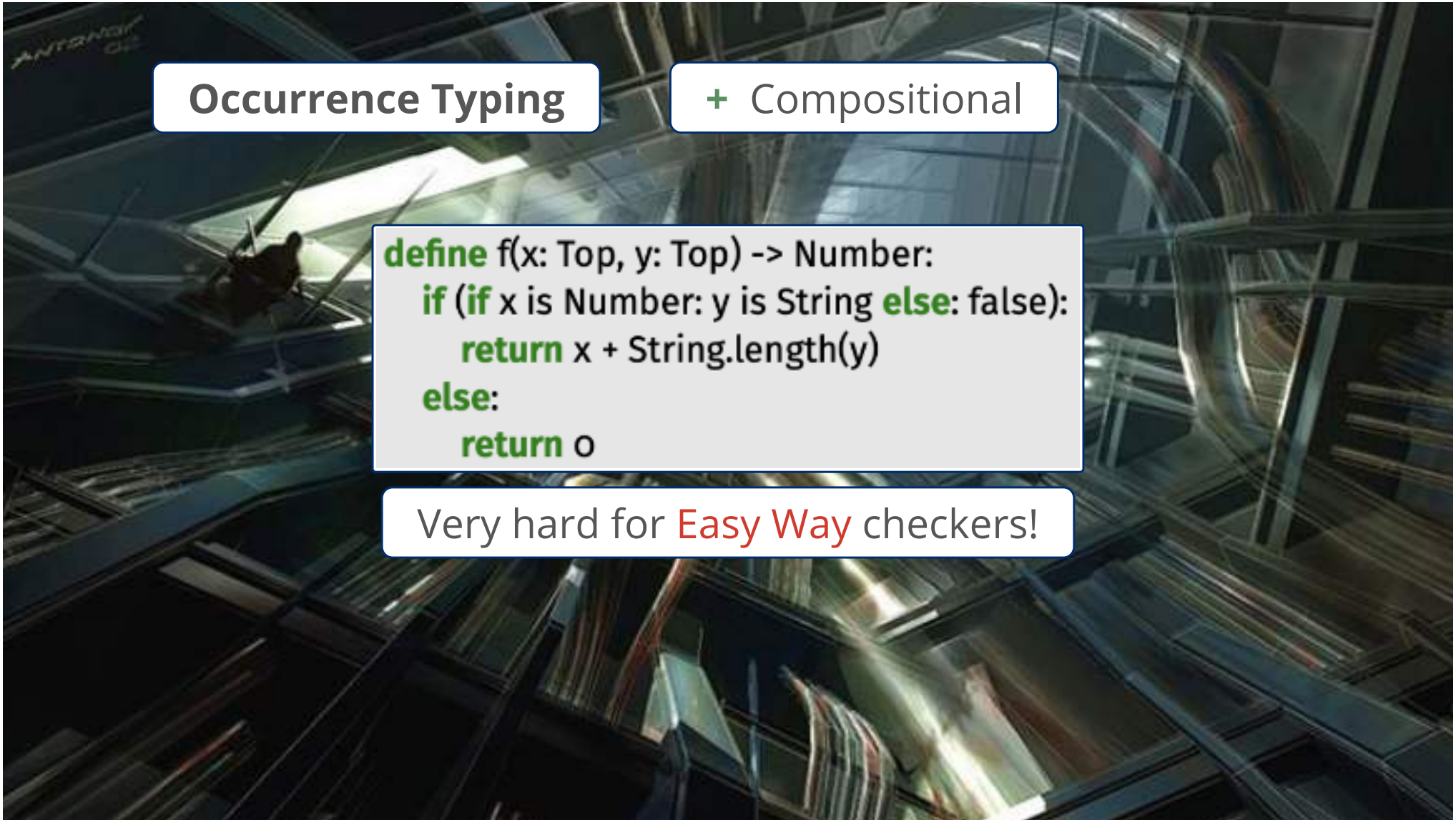

Occurrence Typing
- Complex output
;; this structure represents the result of typechecking an expression
;; fields are #f only when the direct result of parsing or annotations
(def-rep tc-result ([t Type?]
[pset (c:or/c PropSet? #f)]
[o (c:or/c OptObject? #f)]
[exi? boolean?])

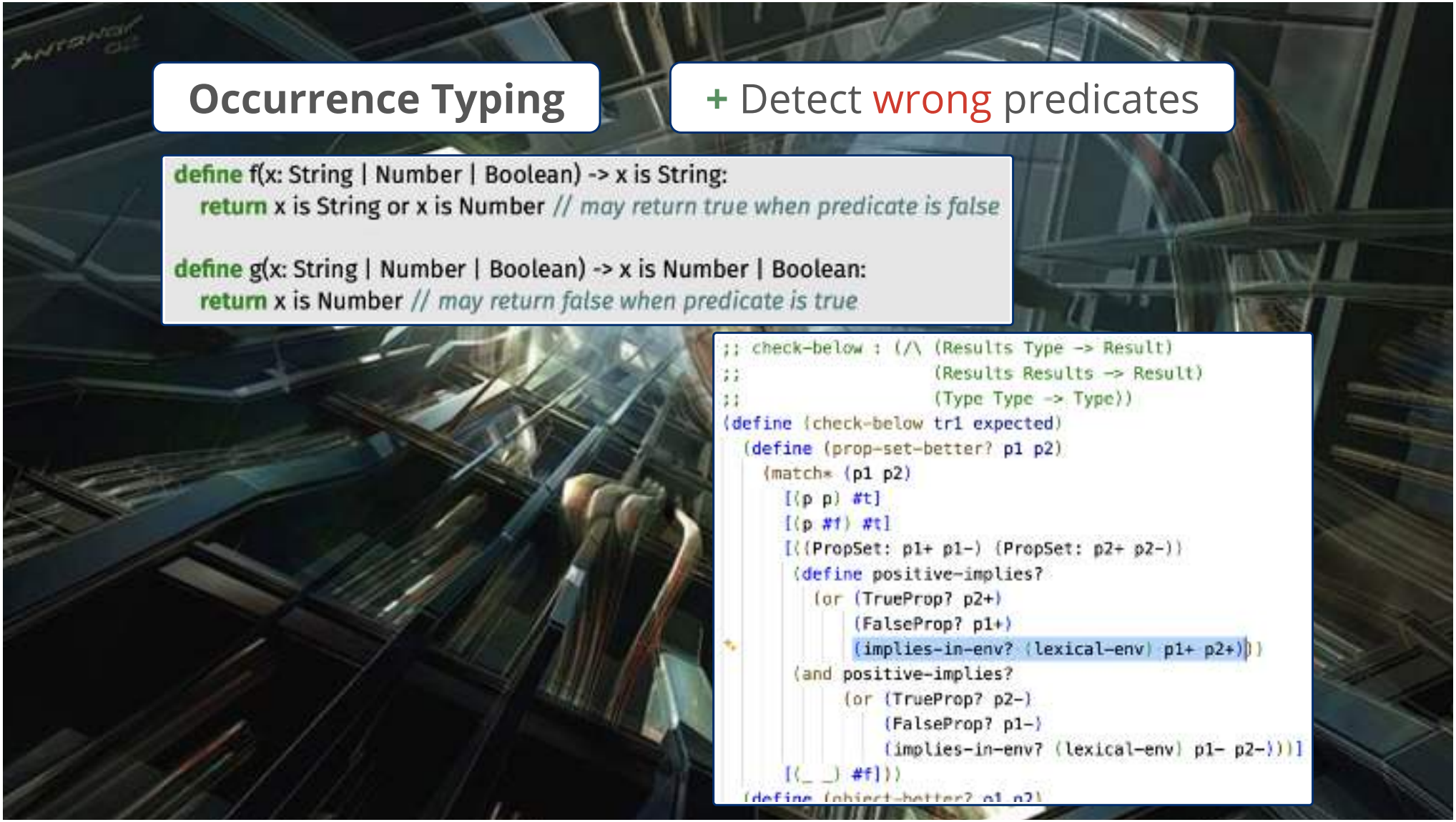

Occurrence Typing
- Complex output
;; this structure represents the result of typechecking an expression
;; fields are #f only when the direct result of parsing or annotations
(def-rep tc-result ([t Type?]
[pset (c:or/c PropSet? #f)]
[o (c:or/c OptObject? #f)]
[exi? boolean?])
(define/cond-contract (tc-expr/check/internal form expected)
(parameterize ([current-orig-stx form])
(syntax-parse form
;; data
[(quote #f) (ret (-val #f) -false-propset)]
[(quote #t) (ret (-val #t) -true-propset)]
[(quote val)
(ret (match expected
[(tc-result1: t) (tc-literal #'val t)]
[_ (tc-literal #'val)])
-true-propset
;; sometimes we want integer's symbolic objects
;; to be themselves
(let ([v (syntax-e #'val)])
(if (and (exact-integer? v)
(with-refinements?))
(-lexp v)
-empty-obj)))]
https://github.com/racket/typed-racket/blob/master/typed-racket-lib/typed-racket/types/tc-result.rkt
https://github.com/racket/typed-racket/blob/master/typed-racket-lib/typed-racket/typecheck/tc-expr-unit.rkt

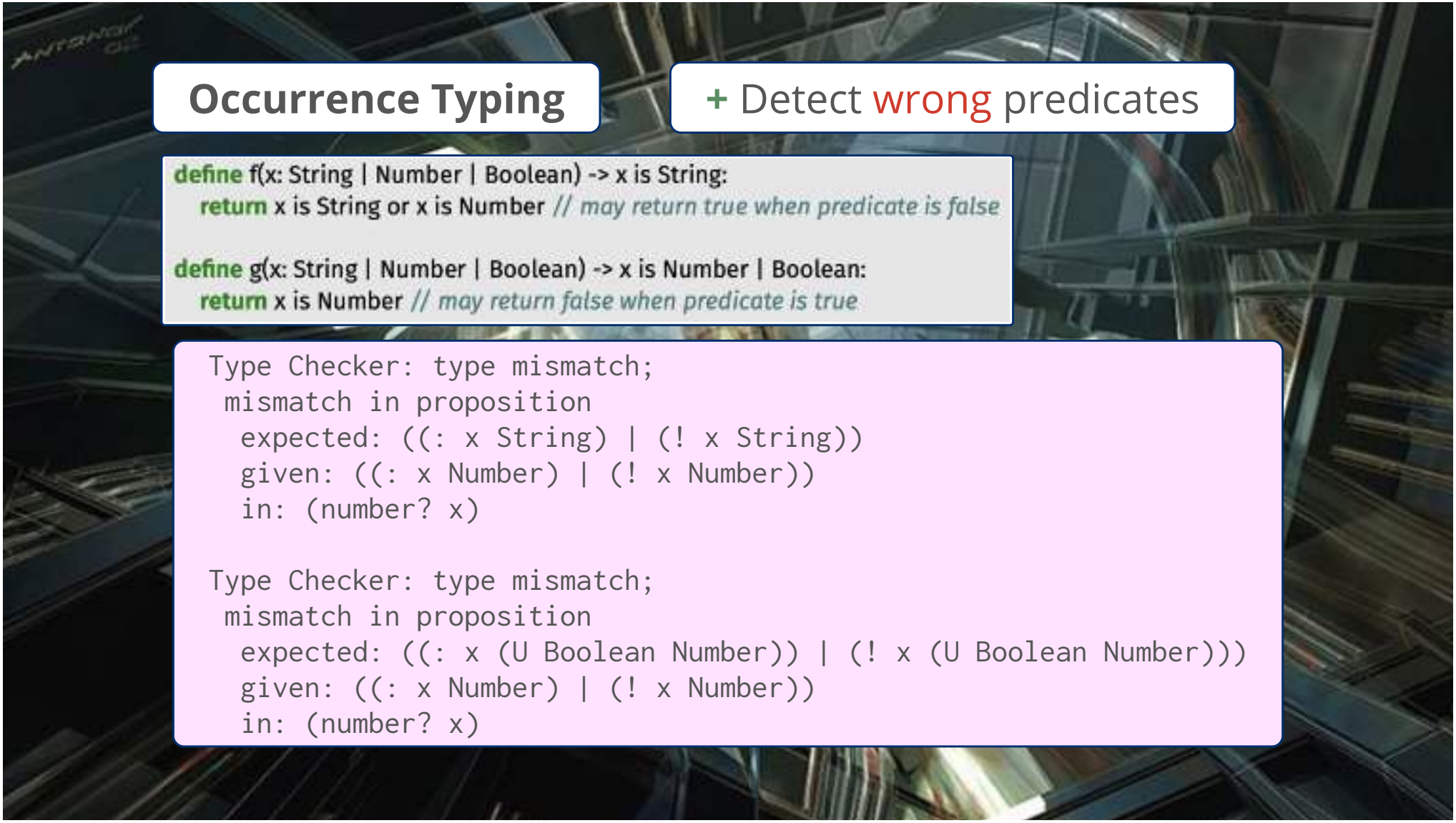

Occurrence Typing
- Complex env
$\Gamma ::= \psi \ldots$
$\psi ::= x : \tau \mid \neg\psi \mid \psi \Rightarrow \psi \mid \psi \wedge \psi \mid \mathsf{true} \mid \mathsf{false}$

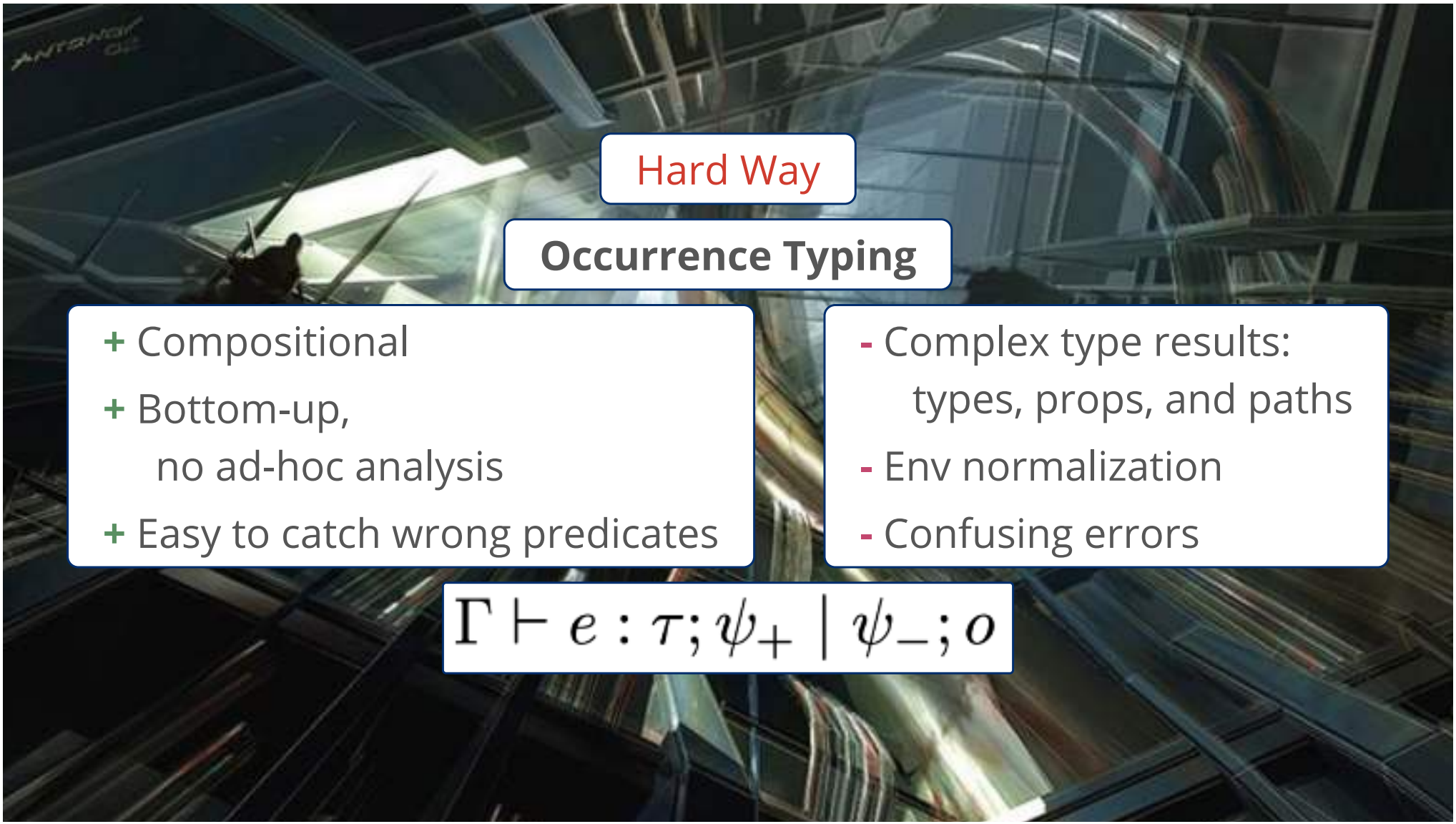

Occurrence Typing
- Complex env
$\Gamma ::= \psi \ldots$
$\psi ::= x : \tau \mid \neg\psi \mid \psi \Rightarrow \psi \mid \psi \wedge \psi \mid \mathsf{true} \mid \mathsf{false}$
;; types is a free-id-table of identifiers to types
;; props is a list of known propositions
(define-struct env ([types immutable-free-id-table?]
[idx-types (hash/c Index? Type? #:immutable #t)]
[props (listof Prop?)]
[aliases immutable-free-id-table?]))
https://github.com/racket/typed-racket/blob/master/typed-racket-lib/typed-racket/env/type-env-structs.rkt


Occurrence Typing
+ Free of ad-hoc analysis
(define (tc/if-twoarm tst thn els [expected #f])
(match-define (tc-result1: _ (PropSet: p+ p-) _) (single-value tst))
(define thn-res
(with-lexical-env+props (list p+)
#:expected expected
#:unreachable (warn-unreachable thn)
(test-position-add-true tst)
(tc-expr/check thn expected)))
(define els-res
(with-lexical-env+props (list p-)
#:expected expected
#:unreachable (warn-unreachable els)
(test-position-add-false tst)
(tc-expr/check els expected)))
(match expected
;; if there was not any expected results, then merge the 'then'
;; and 'else' results so we propogate the correct info upwards
[(or #f (tc-any-results: #f)) (merge-tc-results (list thn-res els-res))]
;; otherwise, the subcomponents have already been checked and


Occurrence Typing
+ Compositional
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
Very hard for Easy Way checkers!


Occurrence Typing
+ Detect wrong predicates
define f(x: String | Number | Boolean) -> x is String:
return x is String or x is Number // may return true when predicate is false
define g(x: String | Number | Boolean) -> x is Number | Boolean:
return x is Number // may return false when predicate is true
;; check-below : (/\ (Results Type -> Result)
;; (Results Results -> Result)
;; (Type Type -> Type))
(define (check-below tr1 expected)
(define (prop-set-better? p1 p2)
(match* (p1 p2)
[(p p) #t]
[(p #f) #t]
[((PropSet: p1+ p1-) (PropSet: p2+ p2-))
(define positive-implies?
(or (TrueProp? p2+)
(FalseProp? p1+)
(implies-in-env? (lexical-env) p1+ p2+)))
(and positive-implies?
(or (TrueProp? p2-)
(FalseProp? p1-)
(implies-in-env? (lexical-env) p1- p2-)))]
[(_ _) #f]))


Occurrence Typing
+ Detect wrong predicates
define f(x: String | Number | Boolean) -> x is String:
return x is String or x is Number // may return true when predicate is false
define g(x: String | Number | Boolean) -> x is Number | Boolean:
return x is Number // may return false when predicate is true
Type Checker: type mismatch;
mismatch in proposition
expected: ((: x String) | (! x String))
given: ((: x Number) | (! x Number))
in: (number? x)
Type Checker: type mismatch;
mismatch in proposition
expected: ((: x (U Boolean Number)) | (! x (U Boolean Number)))
given: ((: x Number) | (! x Number))
in: (number? x)



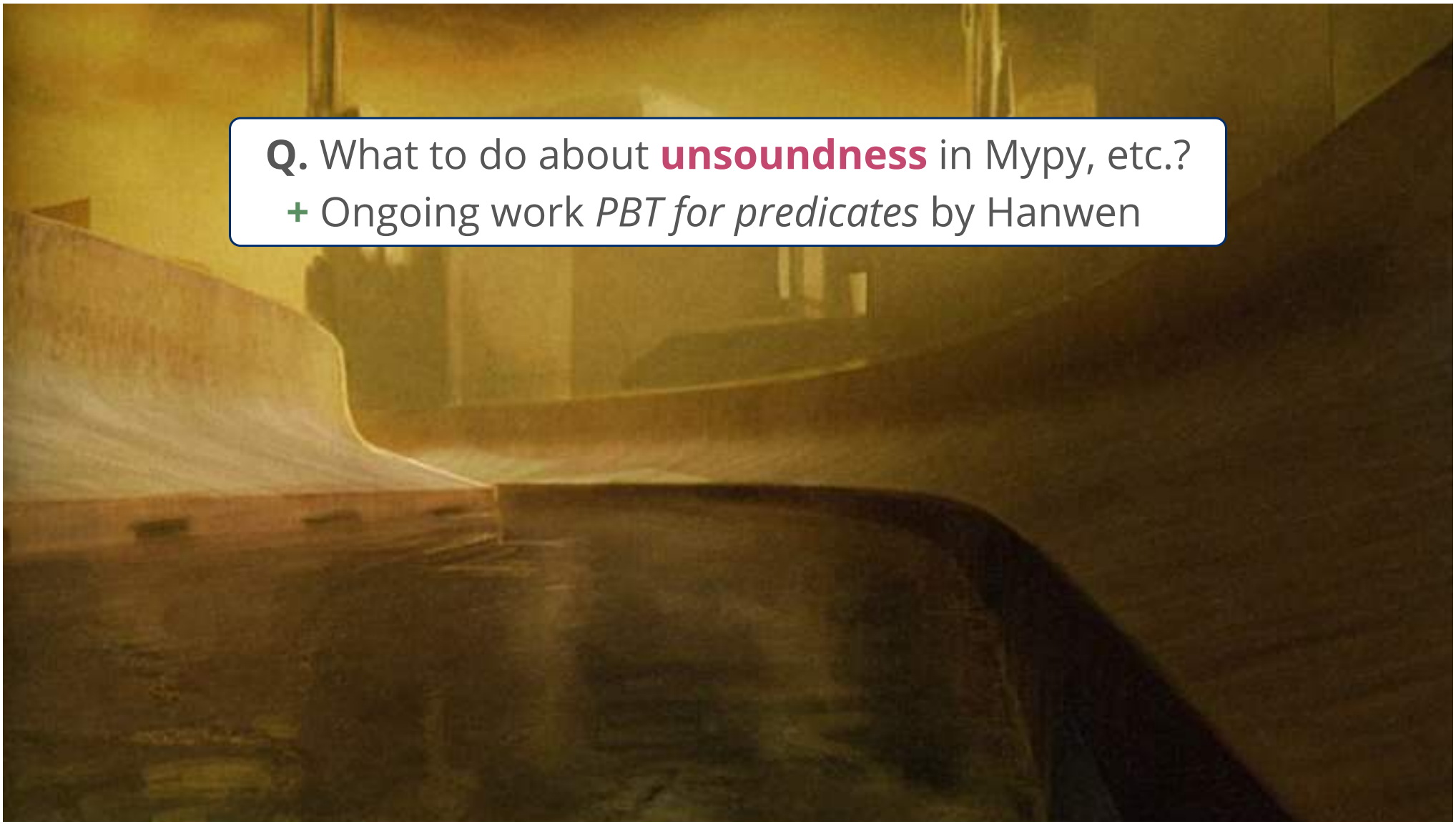
Hard Way

Occurrence Typing

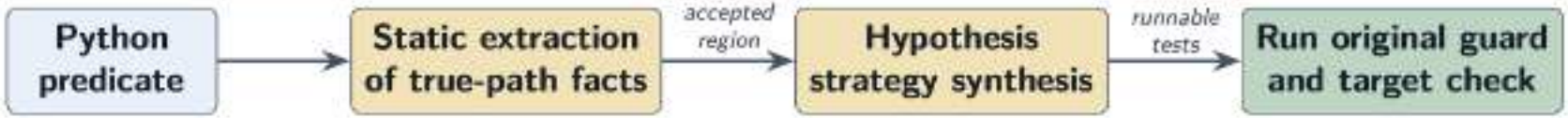
+ Compositional
+ Bottom-up,
  no ad-hoc analysis
+ Easy to catch wrong predicates

- Complex type results:
  types, props, and paths
- Env normalization
- Confusing errors

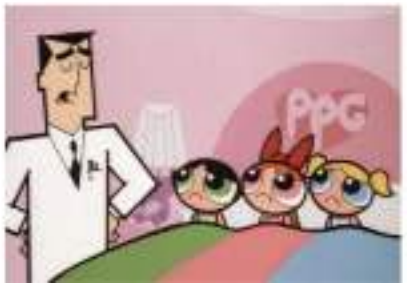
$$\Gamma \vdash e : \tau; \psi_+ \mid \psi_-; o$$


Q. What to do about **unsoundness** in Mypy, etc.?
+ Ongoing work *PBT for predicates* by Hanwen

PPG at a Glance



- Under-approximate only facts justified on `return True` paths.
- Generate values that the original predicate should accept.
- Report semantic evidence, not proof.

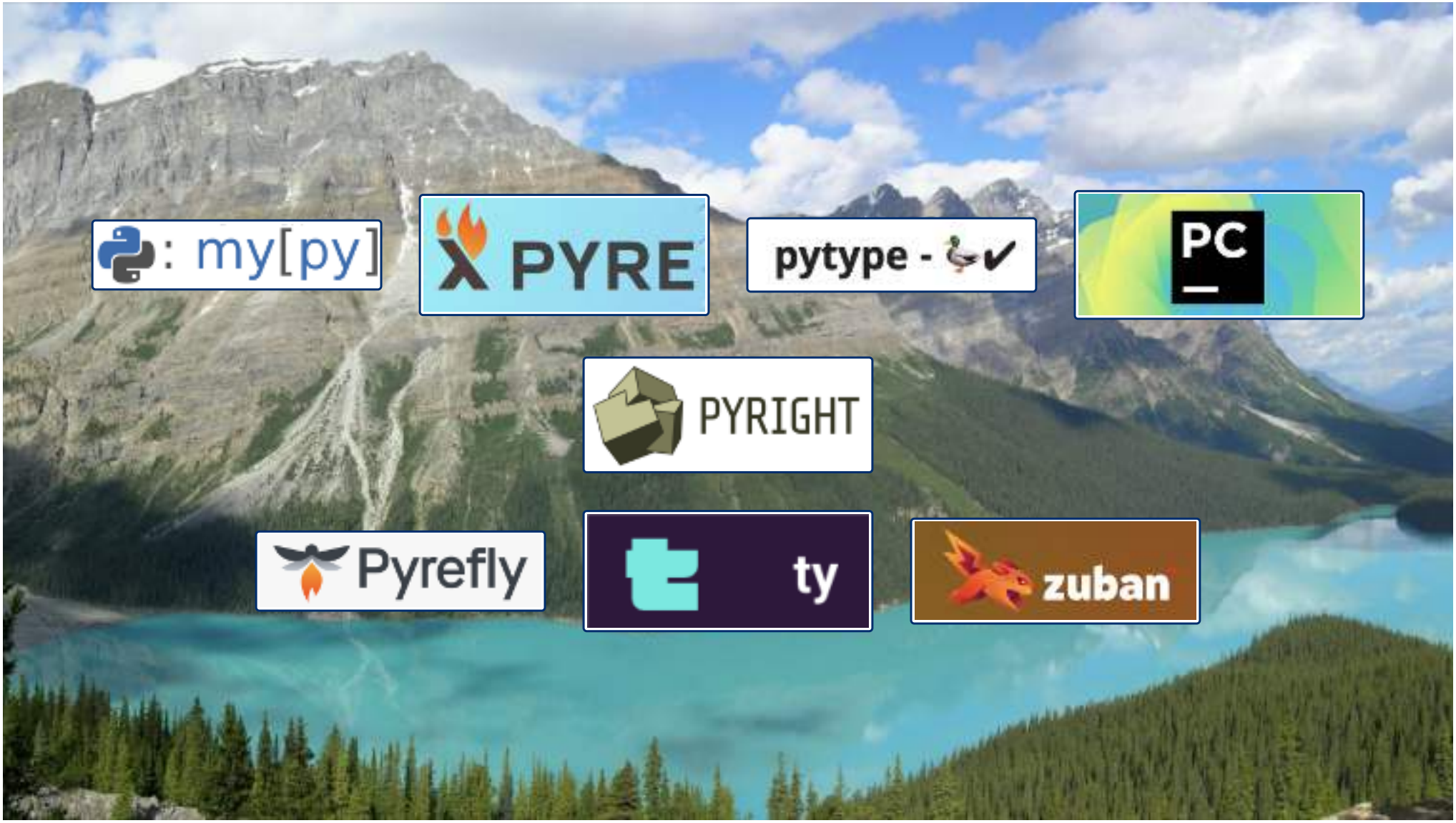


Outcome semantics

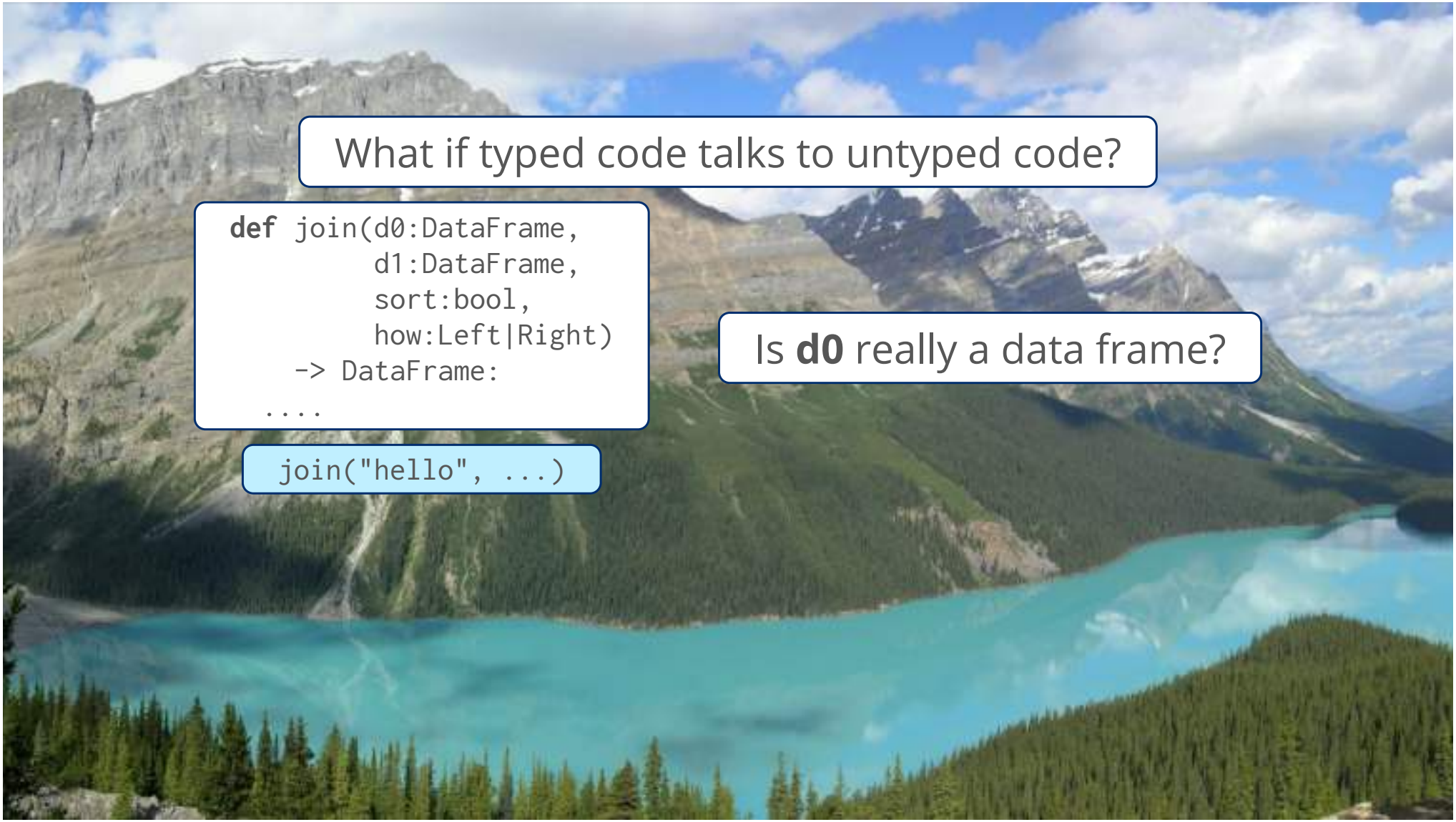
- **PASS**: no counterexample found
- **FAIL**: accepted value violates target type
- **MISS**: automated validation stalls

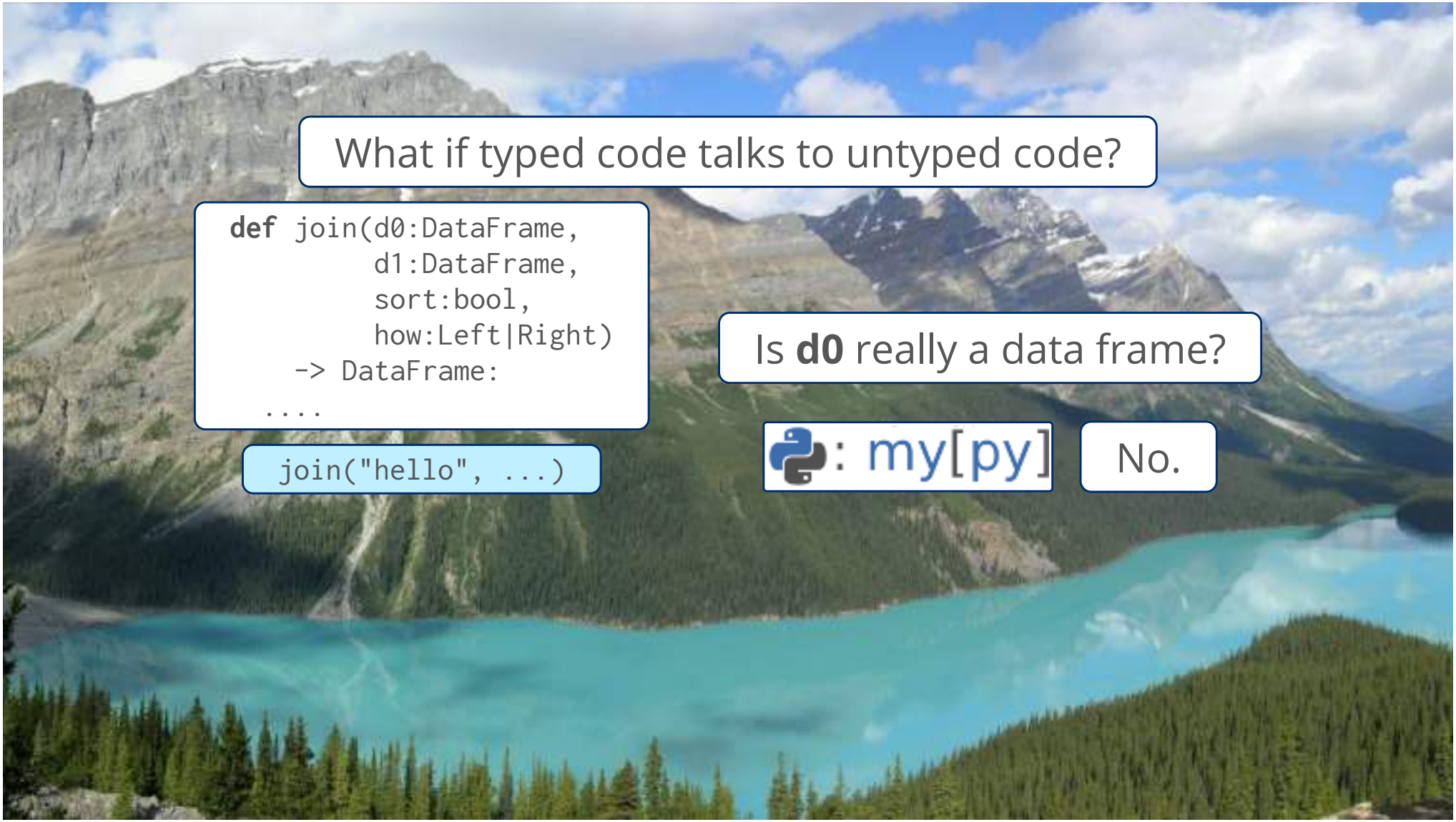
Practicality

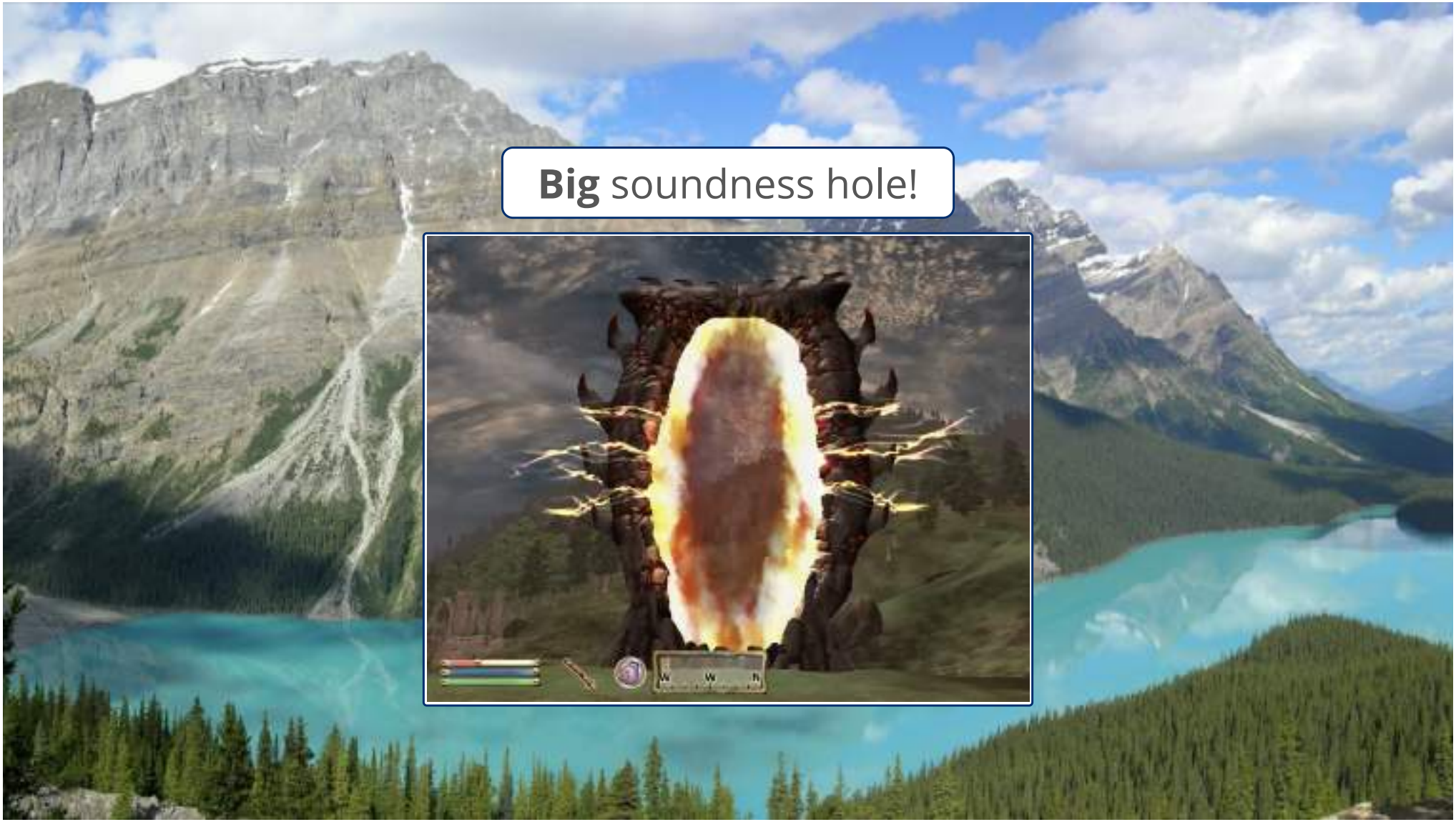
After repository setup, only 3 of 114 predicates needed manual generation support.


: my[py]
PYRE
pytype -
PC
PYRIGHT
Pyrefly
ty
zuban


What if typed code talks to untyped code?


What if typed code talks to untyped code?
def join(d0:DataFrame,
d1:DataFrame,
sort:bool,
how:Left|Right)
-> DataFrame:
....
join("hello", ...)
Is d0 really a data frame?


What if typed code talks to untyped code?
def join(d0:DataFrame,
d1:DataFrame,
sort:bool,
how:Left|Right)
-> DataFrame:
....
join("hello", ...)
Is d0 really a data frame?
: my[py]
No.


Big soundness hole!

TOPLAS '23

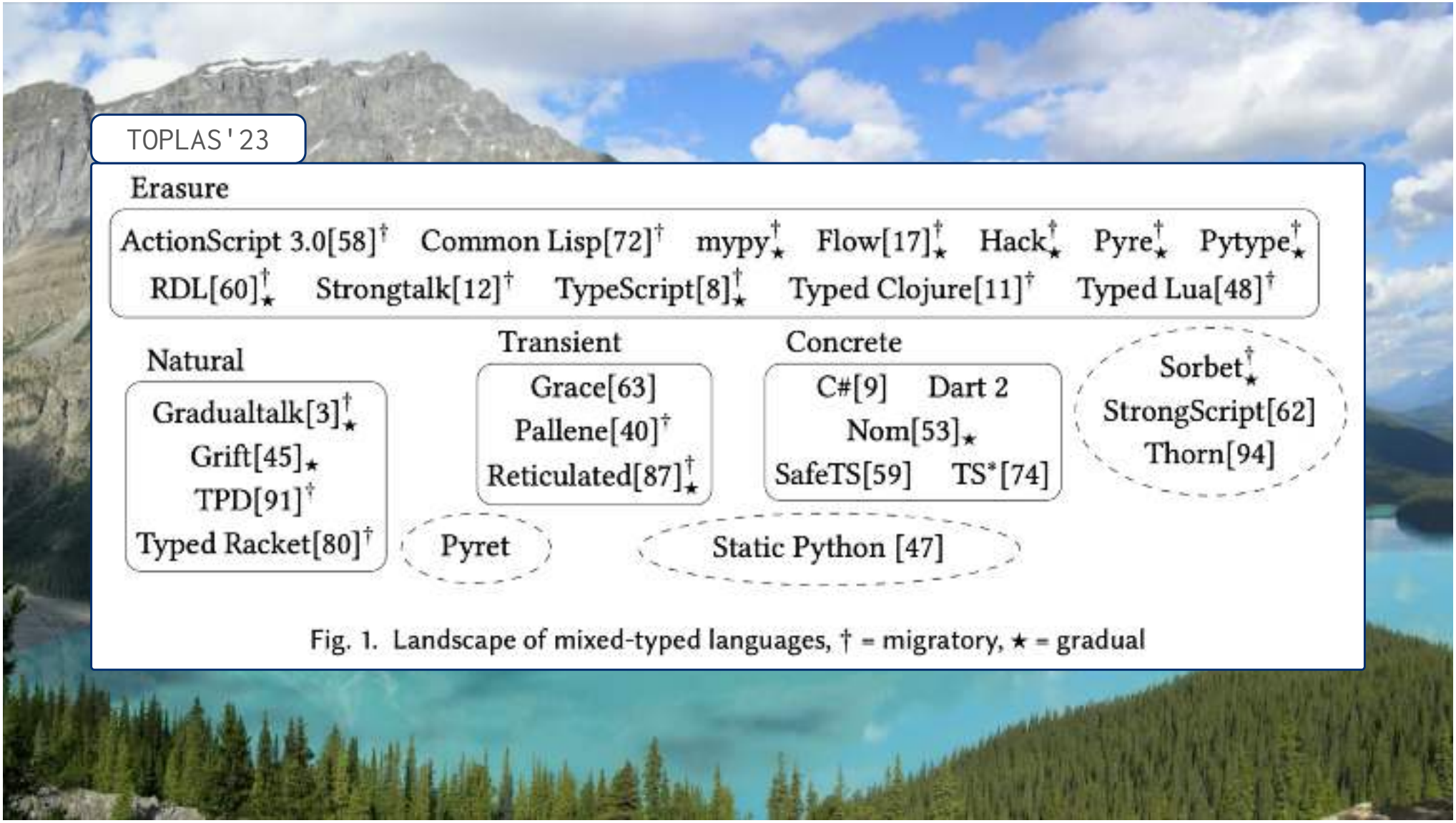
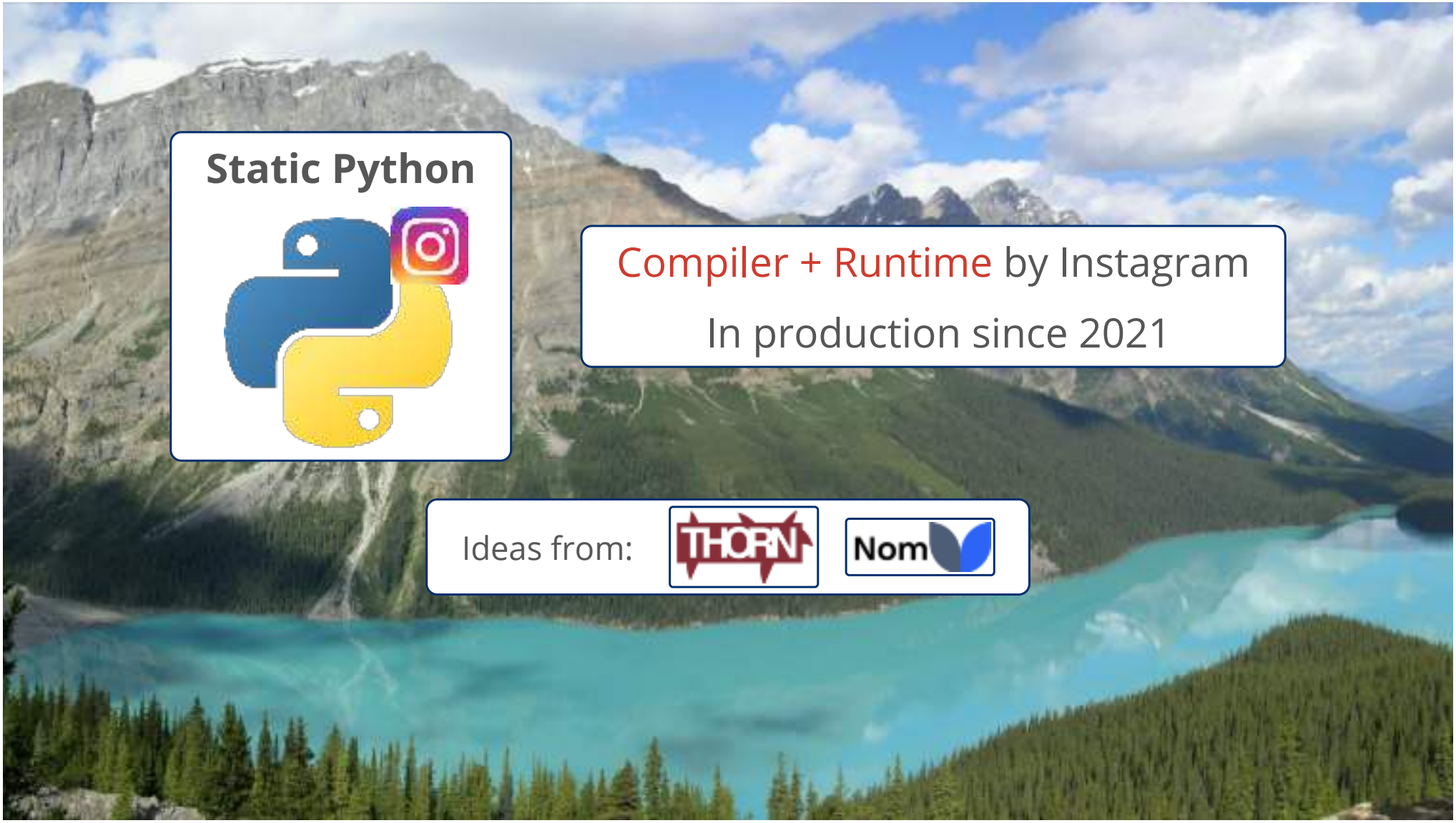
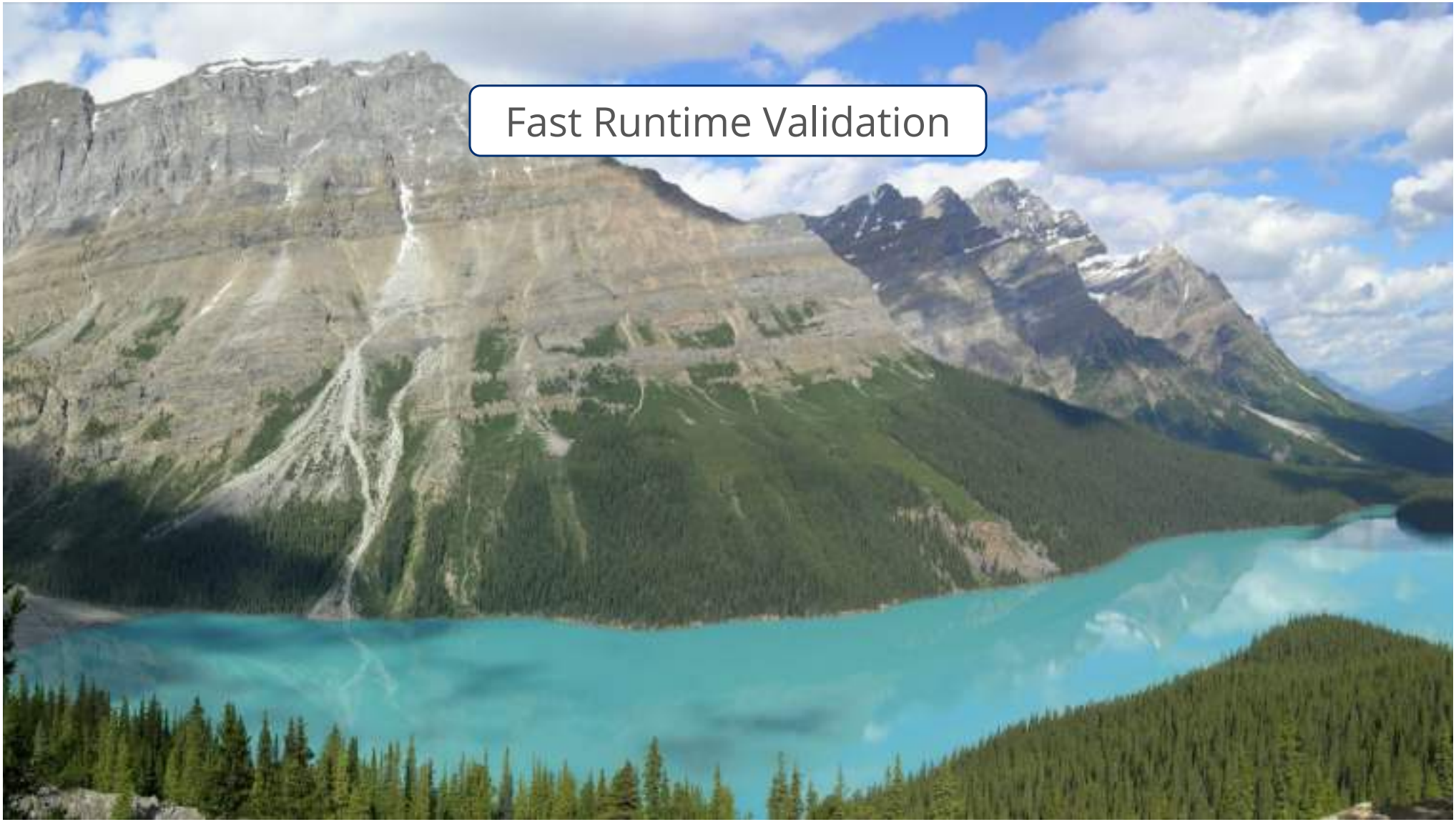
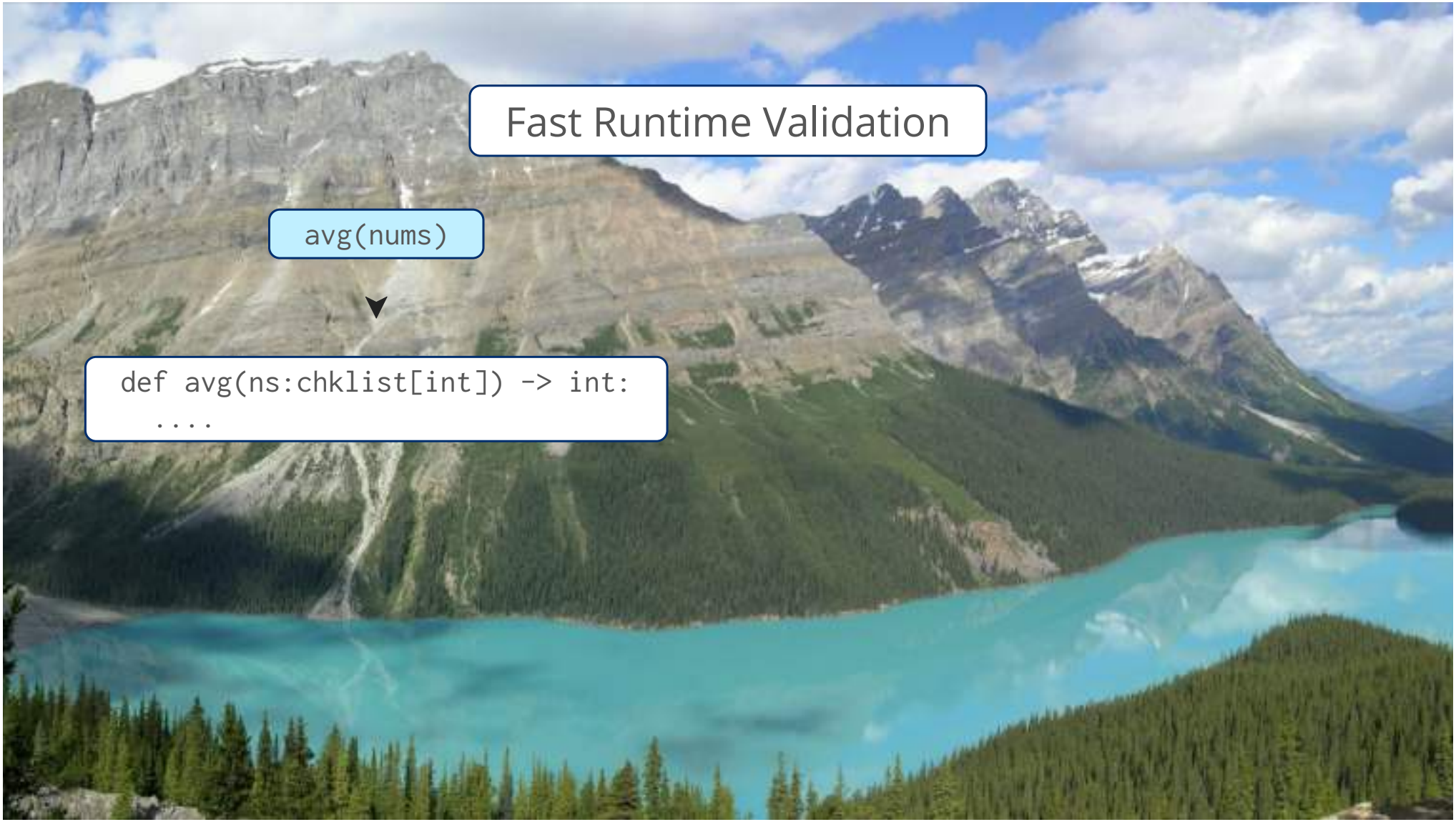
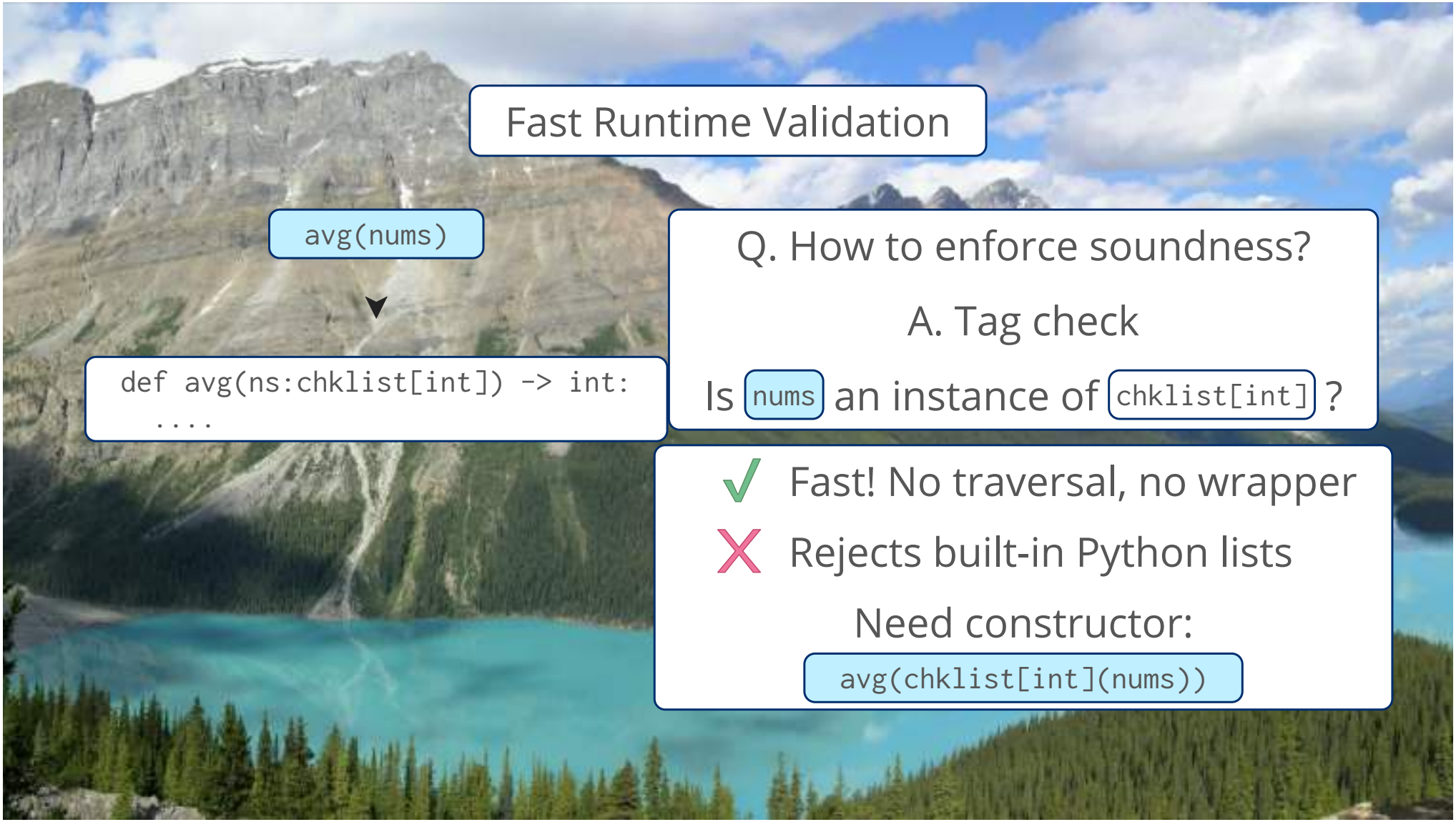


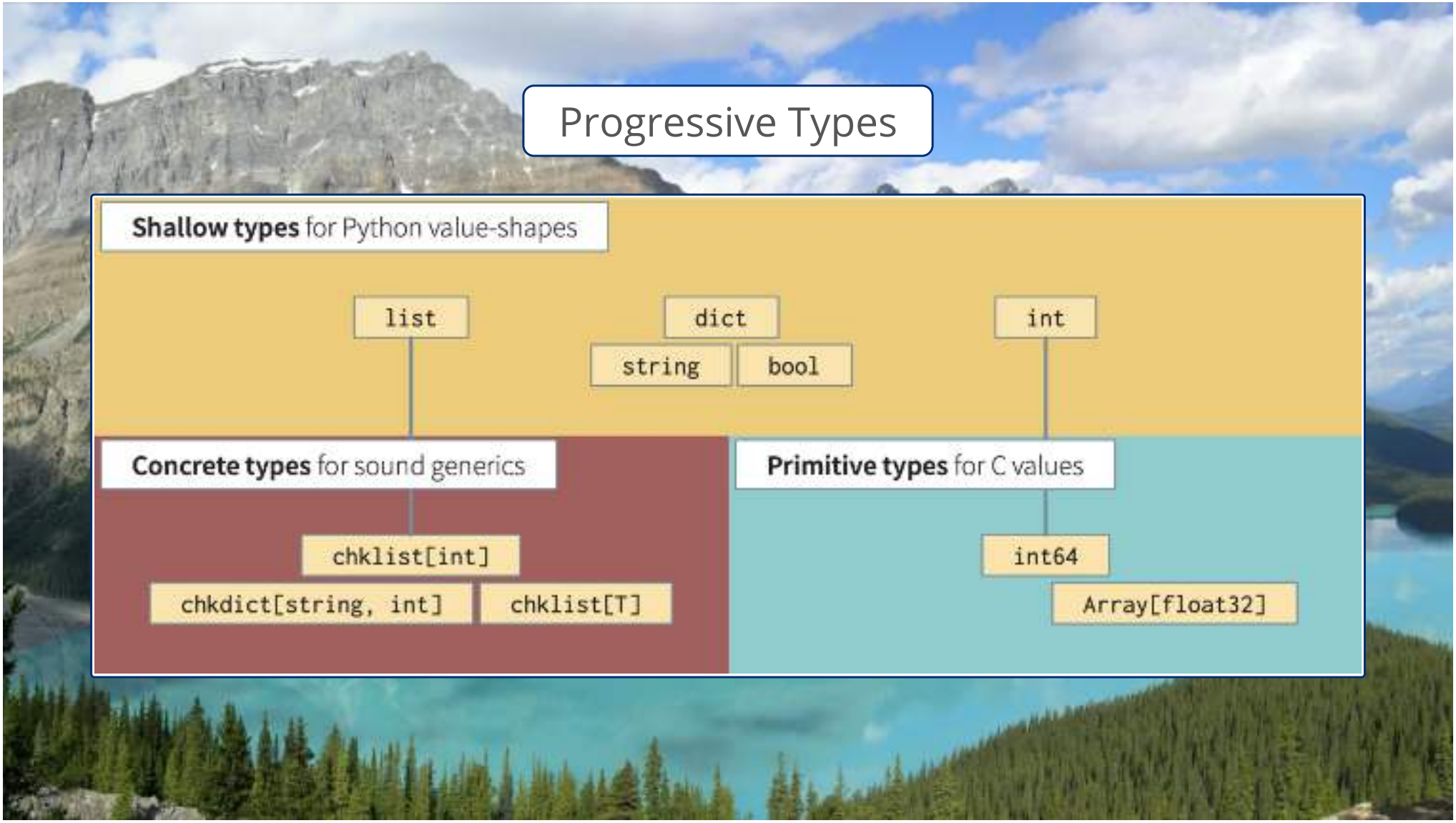
Fig. 1. Landscape of mixed-typed languages, † = migratory, ★ = gradual

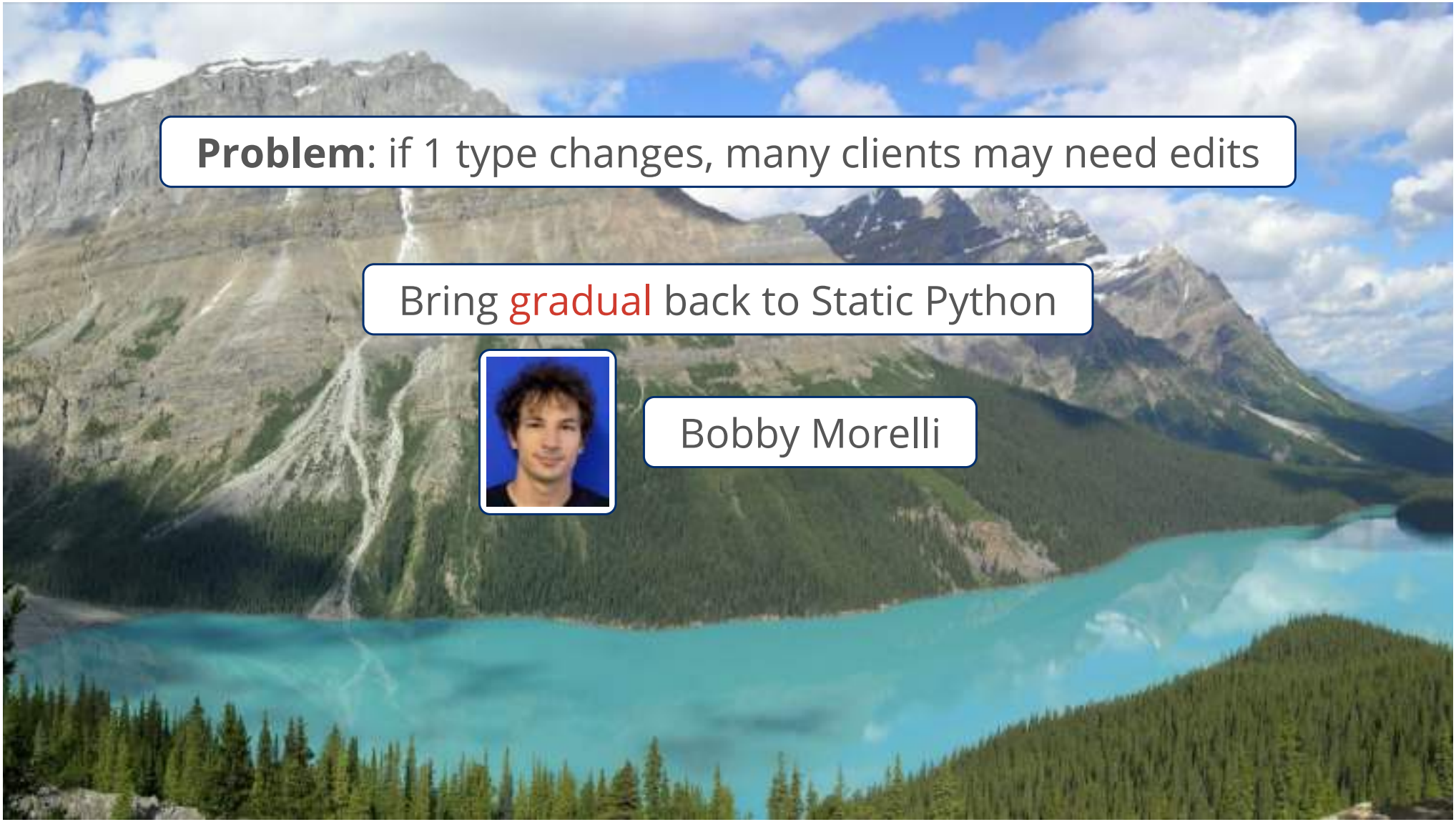

Static Python
Compiler + Runtime by Instagram
In production since 2021
Ideas from:
THORN
Nom


Fast Runtime Validation

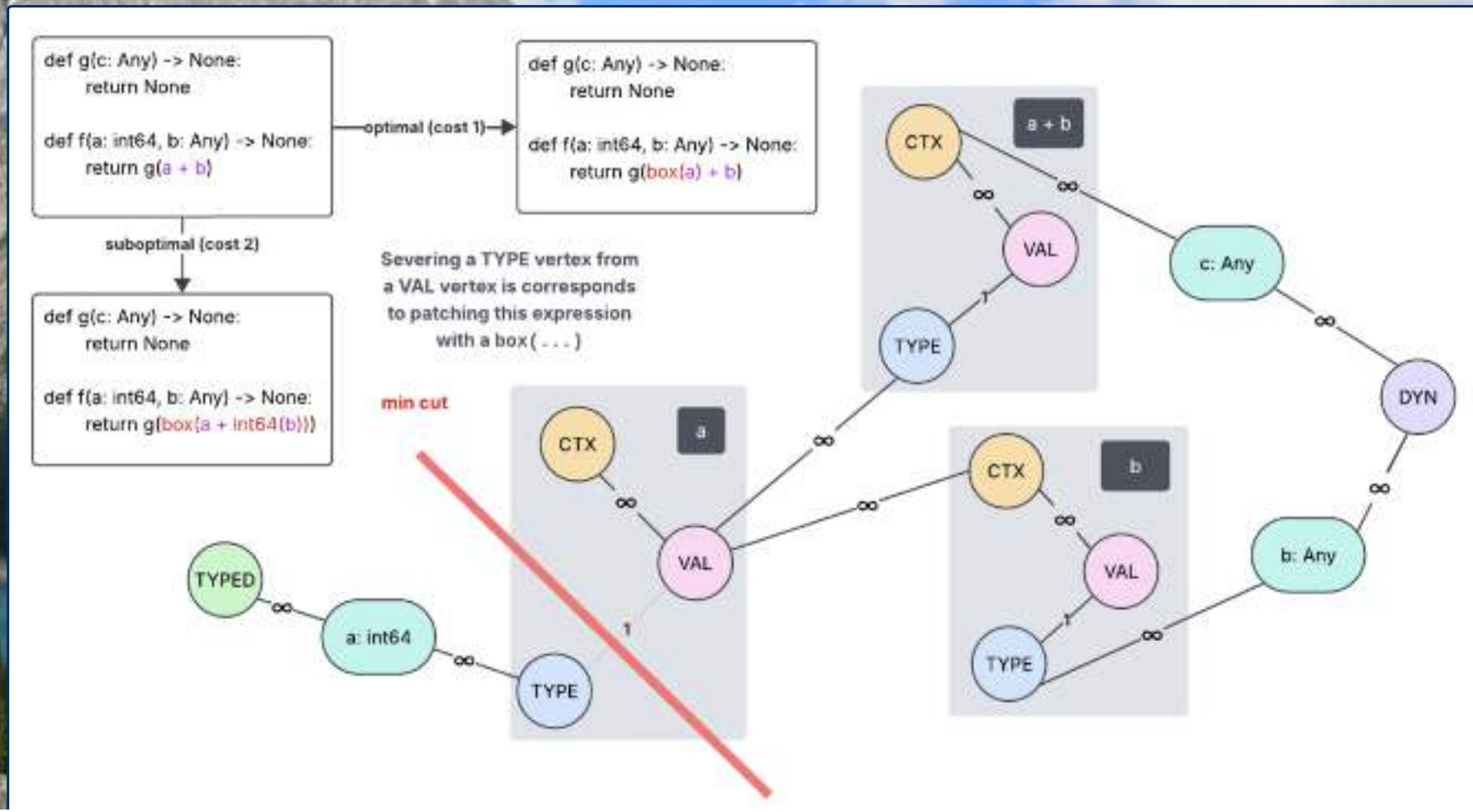

Fast Runtime Validation
avg(nums)
def avg(ns:chklist[int]) -> int:
....

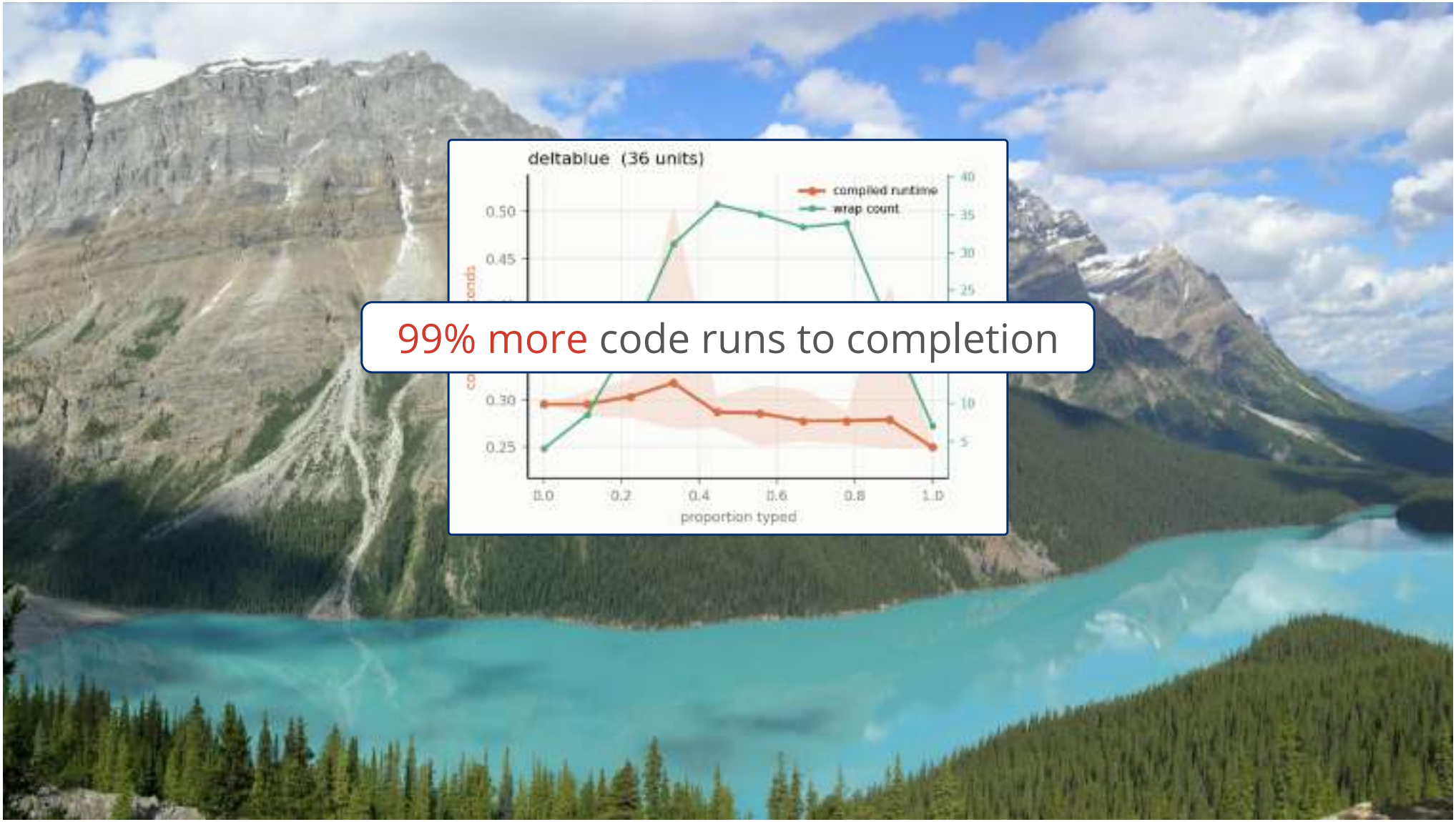

Fast Runtime Validation
avg(nums)
def avg(ns:chklist[int]) -> int:
....
Q. How to enforce soundness?
A. Tag check
Is nums an instance of chklist[int] ?
✓ Fast! No traversal, no wrapper
✗ Rejects built-in Python lists
Need constructor:
avg(chklist[int](nums))


Progressive Types
Shallow types for Python value-shapes
list
dict
string
bool
int
Concrete types for sound generics
Primitive types for C values
chklist[int]
chkdict[string, int]
chklist[T]
int64
Array[float32]


Problem: if 1 type changes, many clients may need edits
Bring gradual back to Static Python
Bobby Morelli

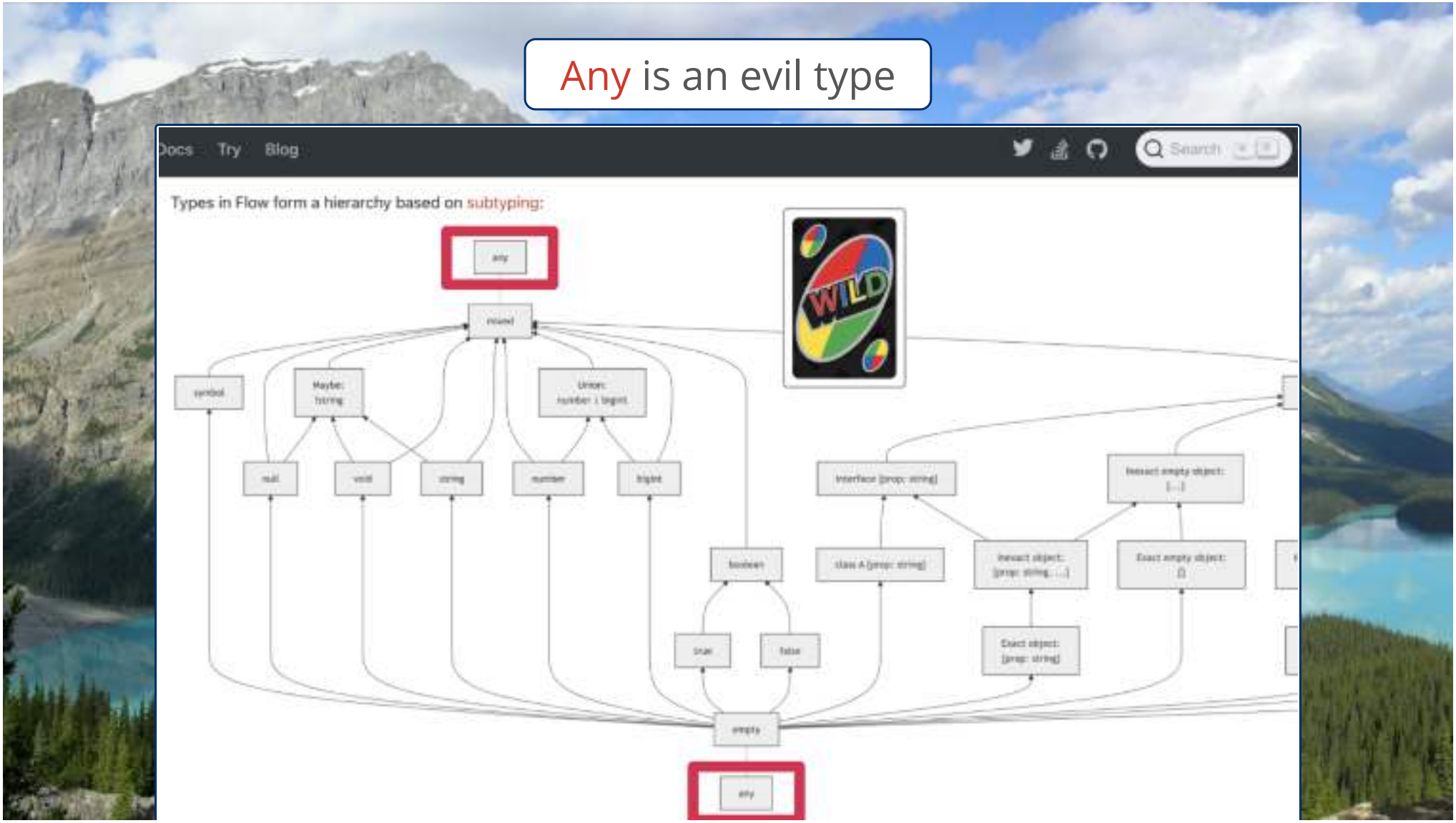
Detyping as min-cut

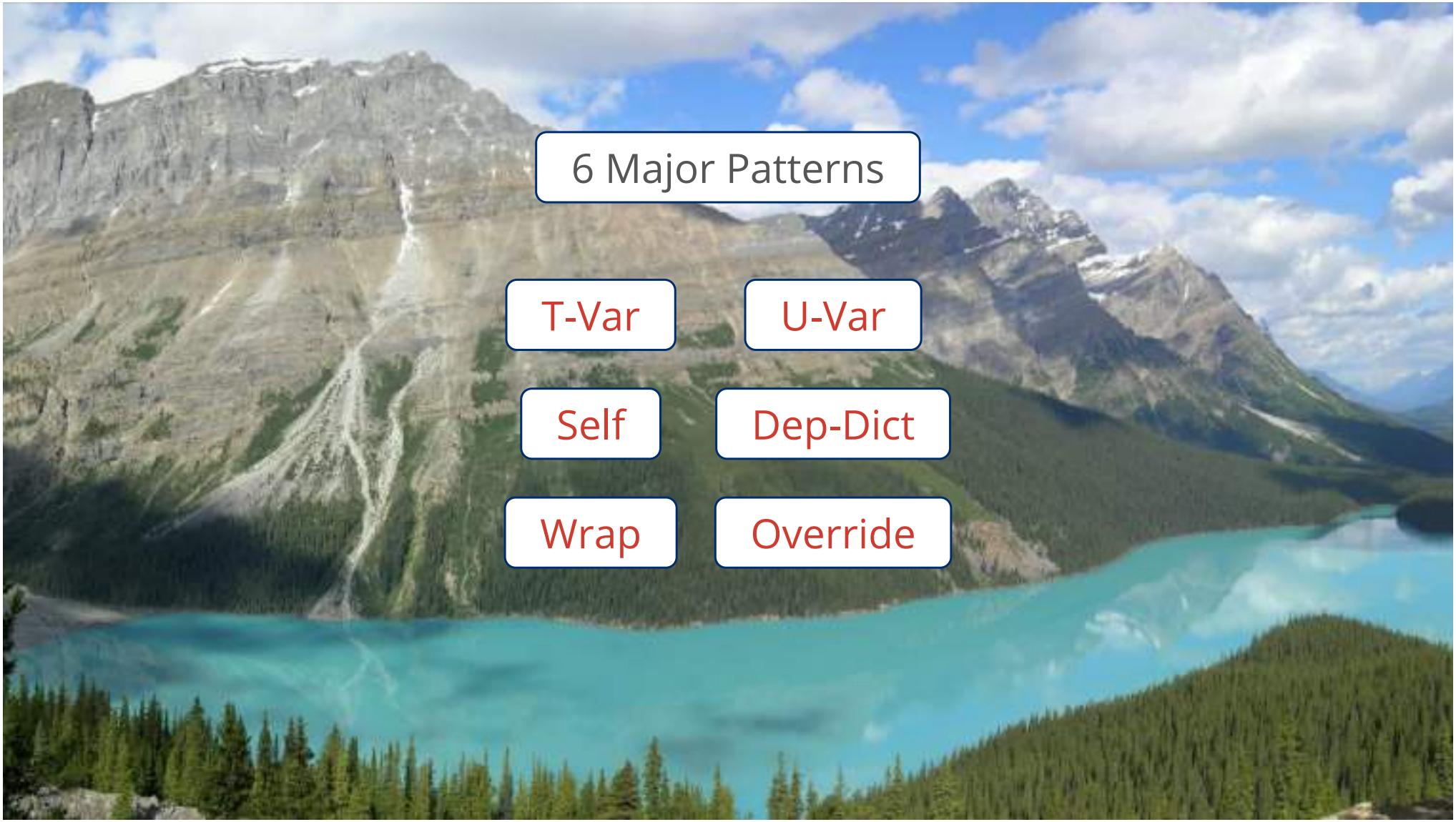


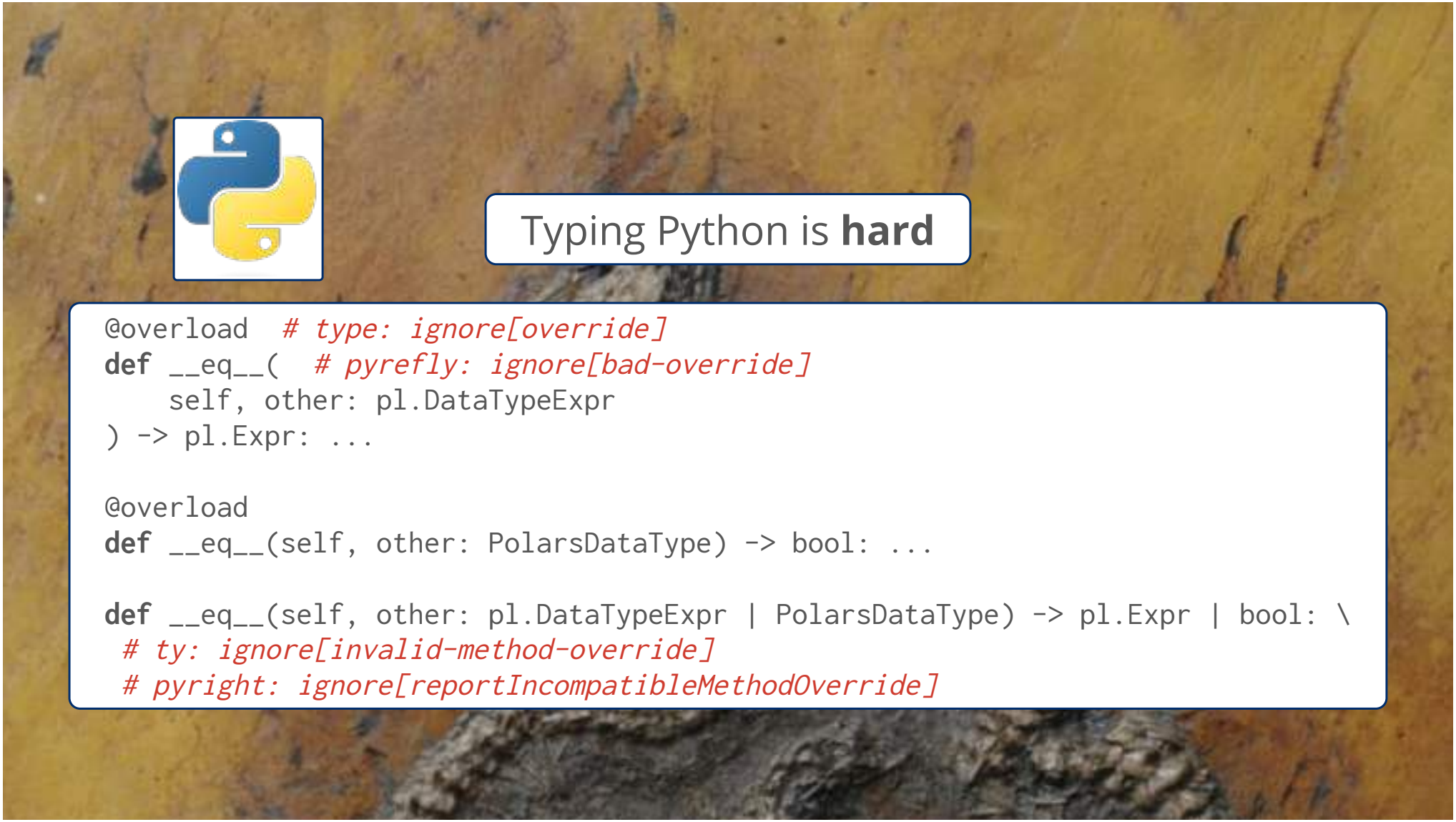


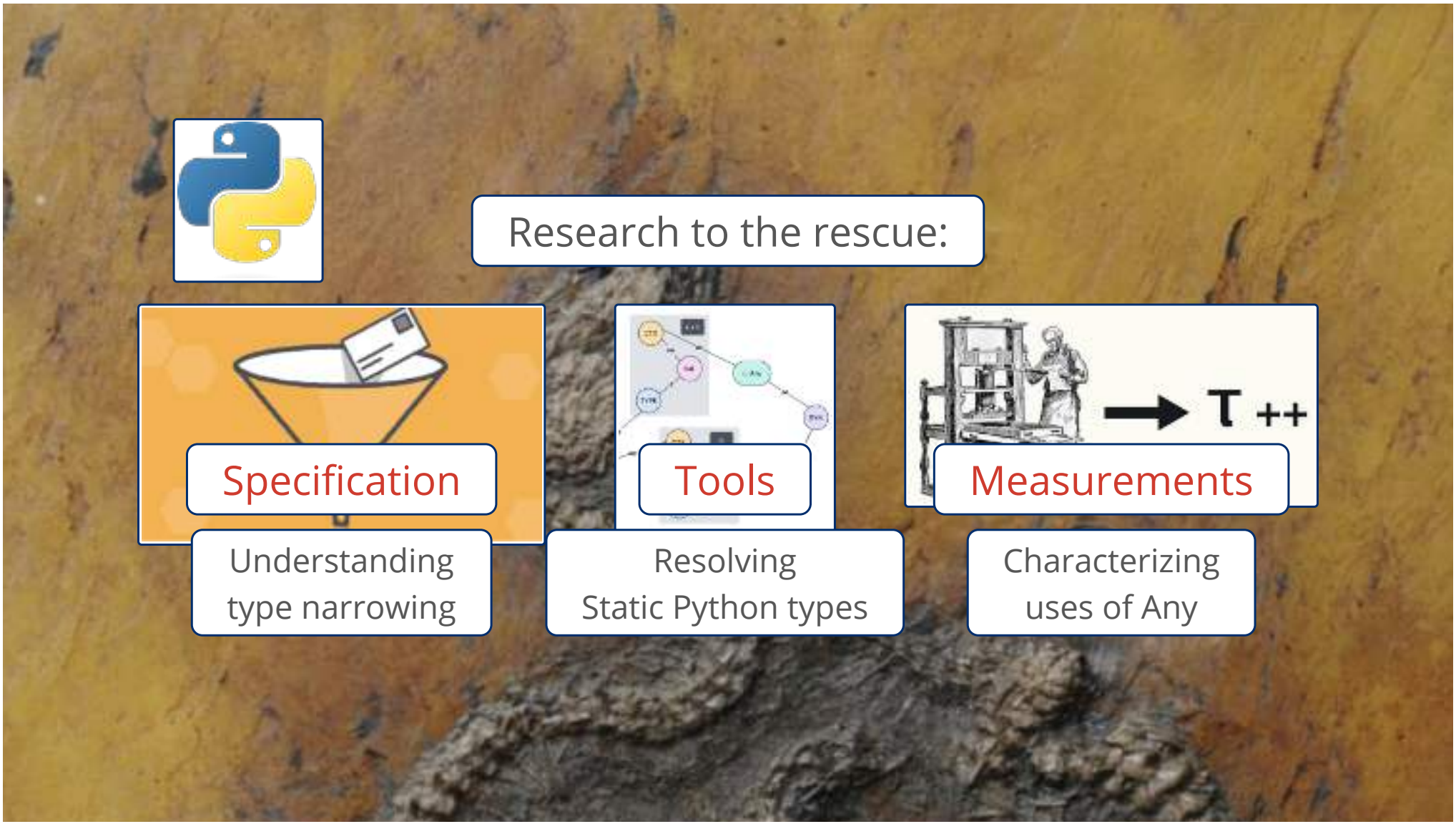
99% more code runs to completion

Any is an evil type




6 Major Patterns
T-Var
U-Var
Self
Dep-Dict
Wrap
Override


Typing Python is **hard**
@overload # type: ignore[override]
def __eq__(# pyrefly: ignore[bad-override]
    self, other: pl.DataTypeExpr
) -> pl.Expr: ...
@overload
def __eq__(self, other: PolarsDataType) -> bool: ...
def __eq__(self, other: pl.DataTypeExpr | PolarsDataType) -> pl.Expr | bool: \
 # ty: ignore[invalid-method-override]
 # pyright: ignore[reportIncompatibleMethodOverride]


Research to the rescue:
Specification
Understanding
type narrowing
Tools
Resolving
Static Python types
τ ++
Measurements
Characterizing
uses of Any