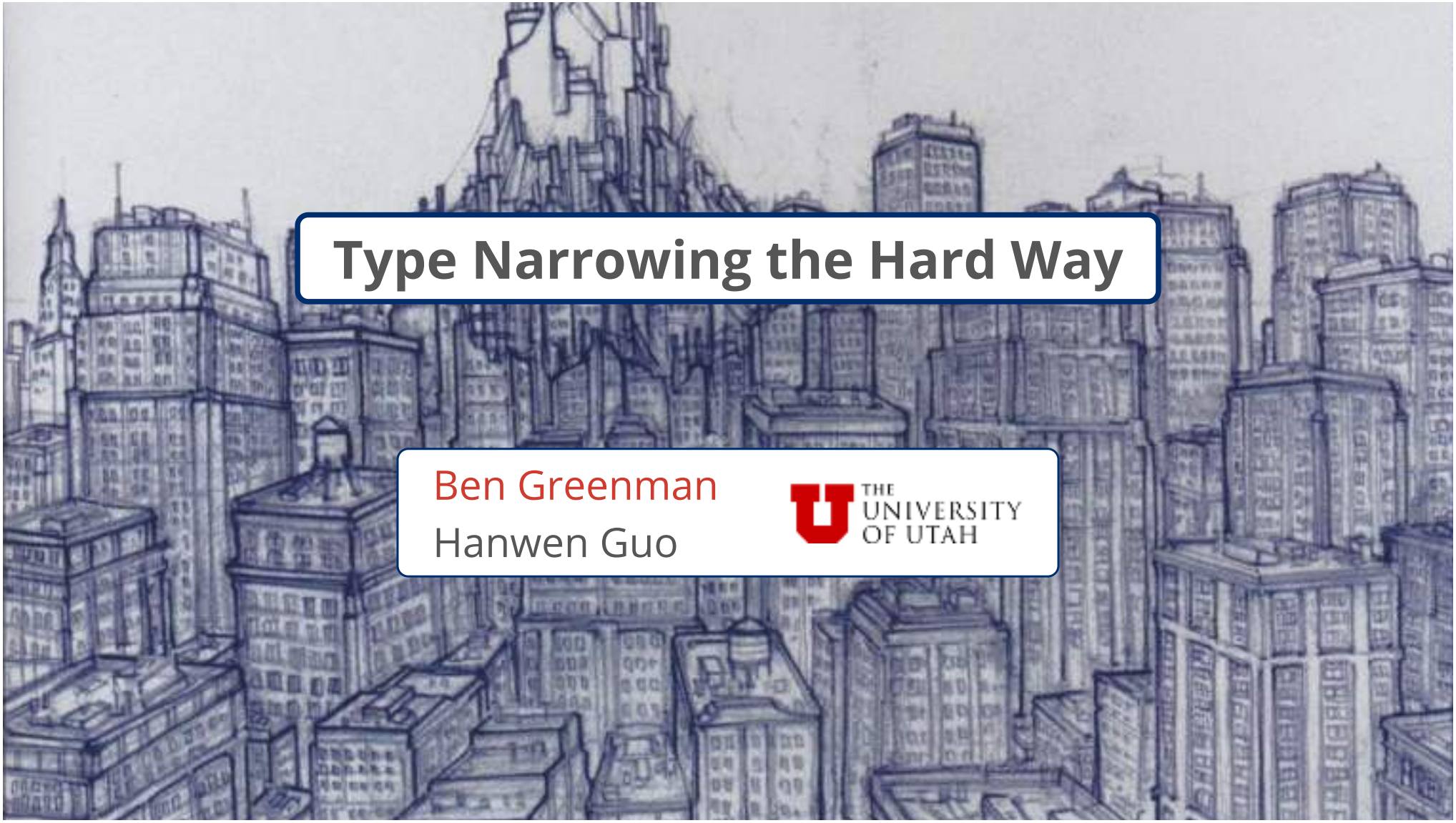
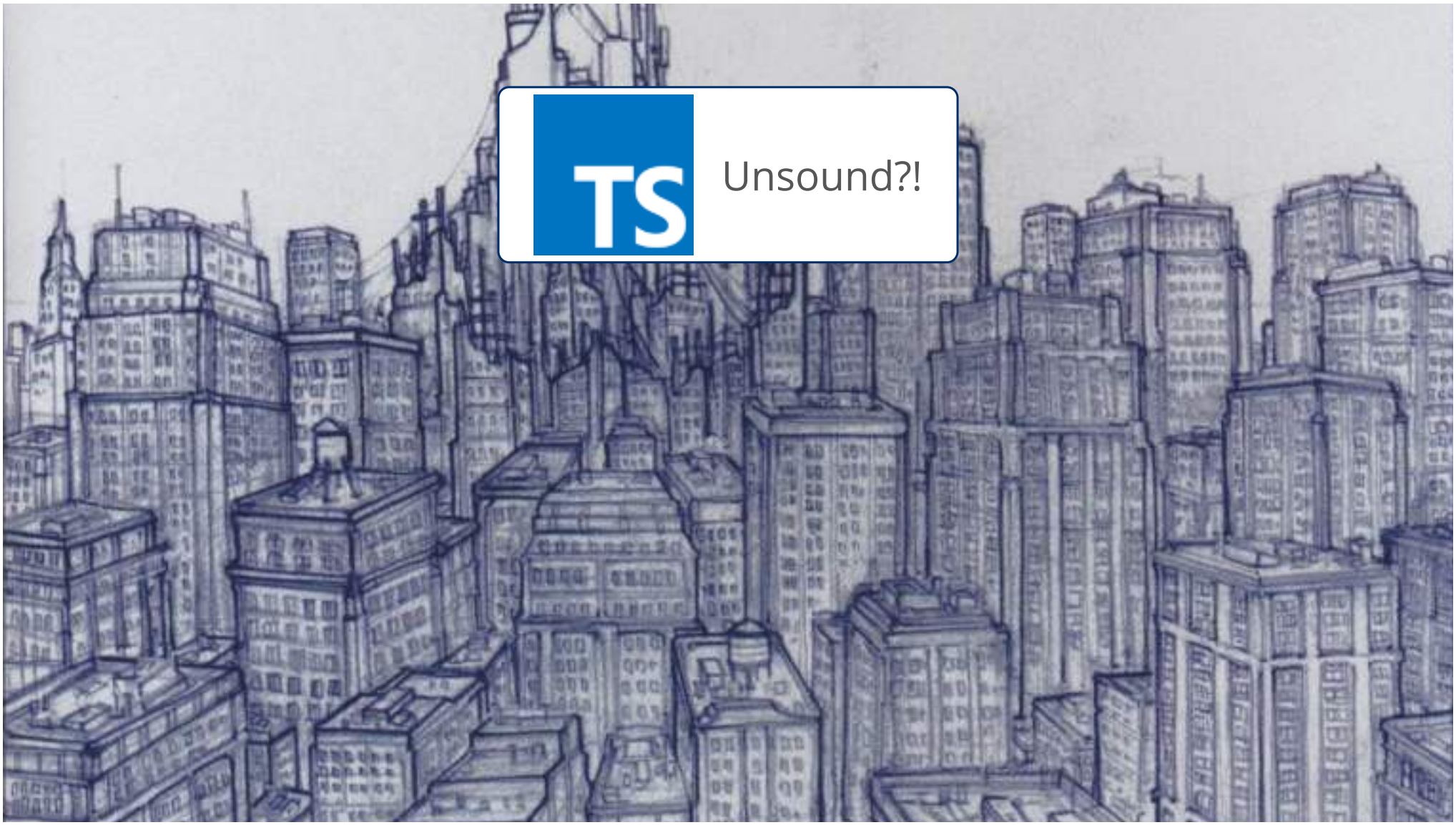
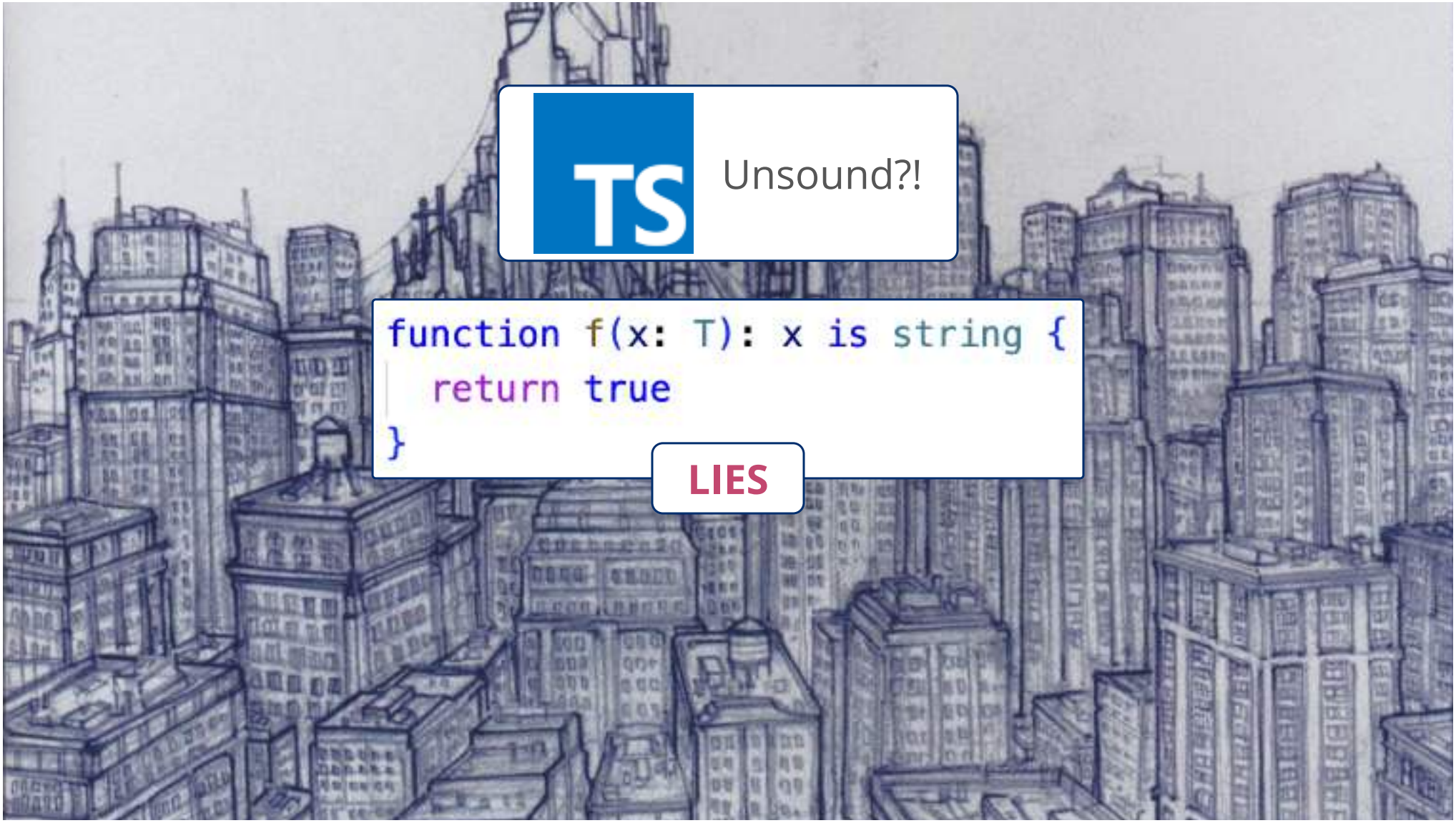


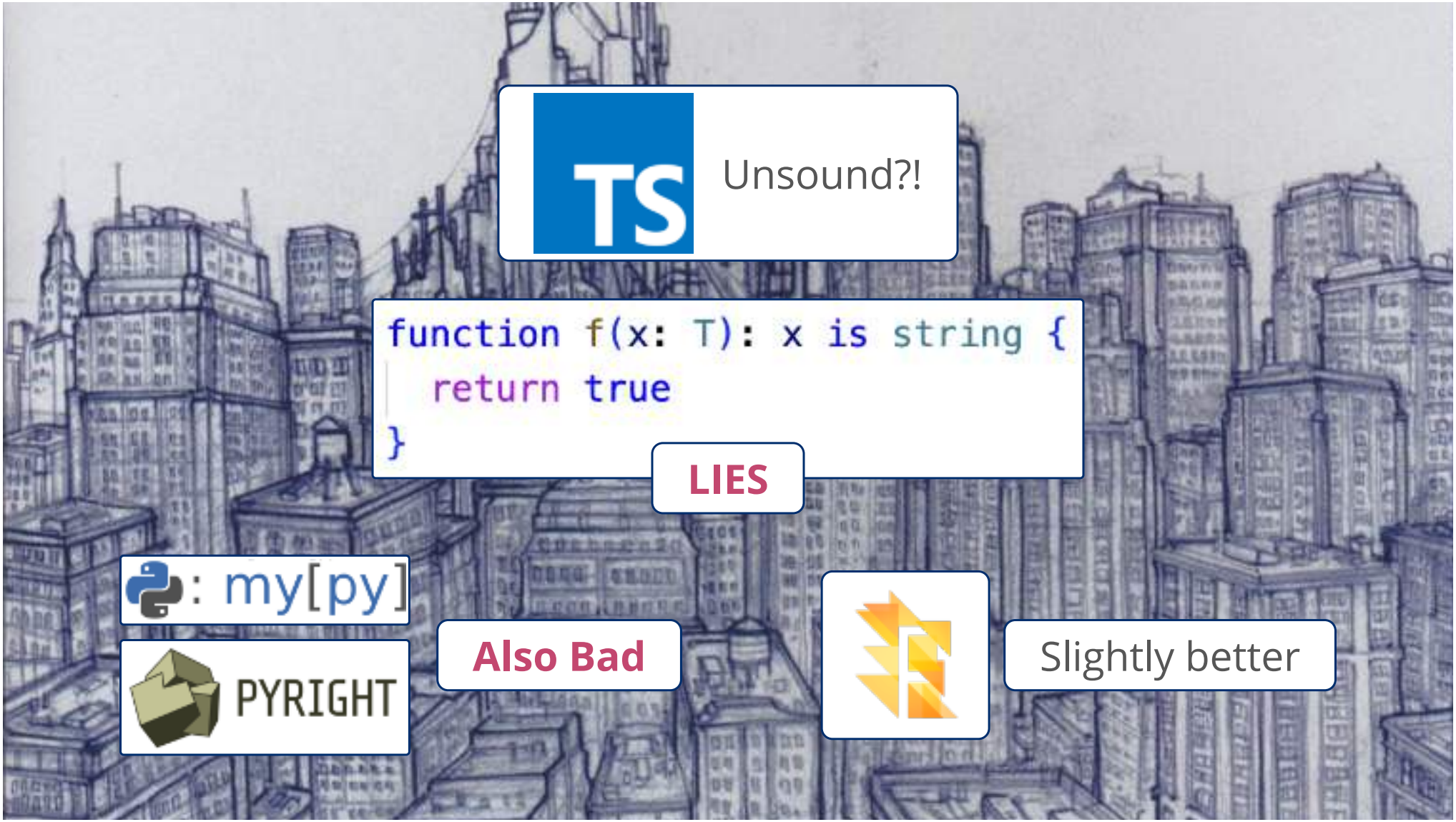
Type Narrowing the Hard Way

Ben Greenman
Hanwen Guo

THE UNIVERSITY OF UTAH


TS
Unsound?!

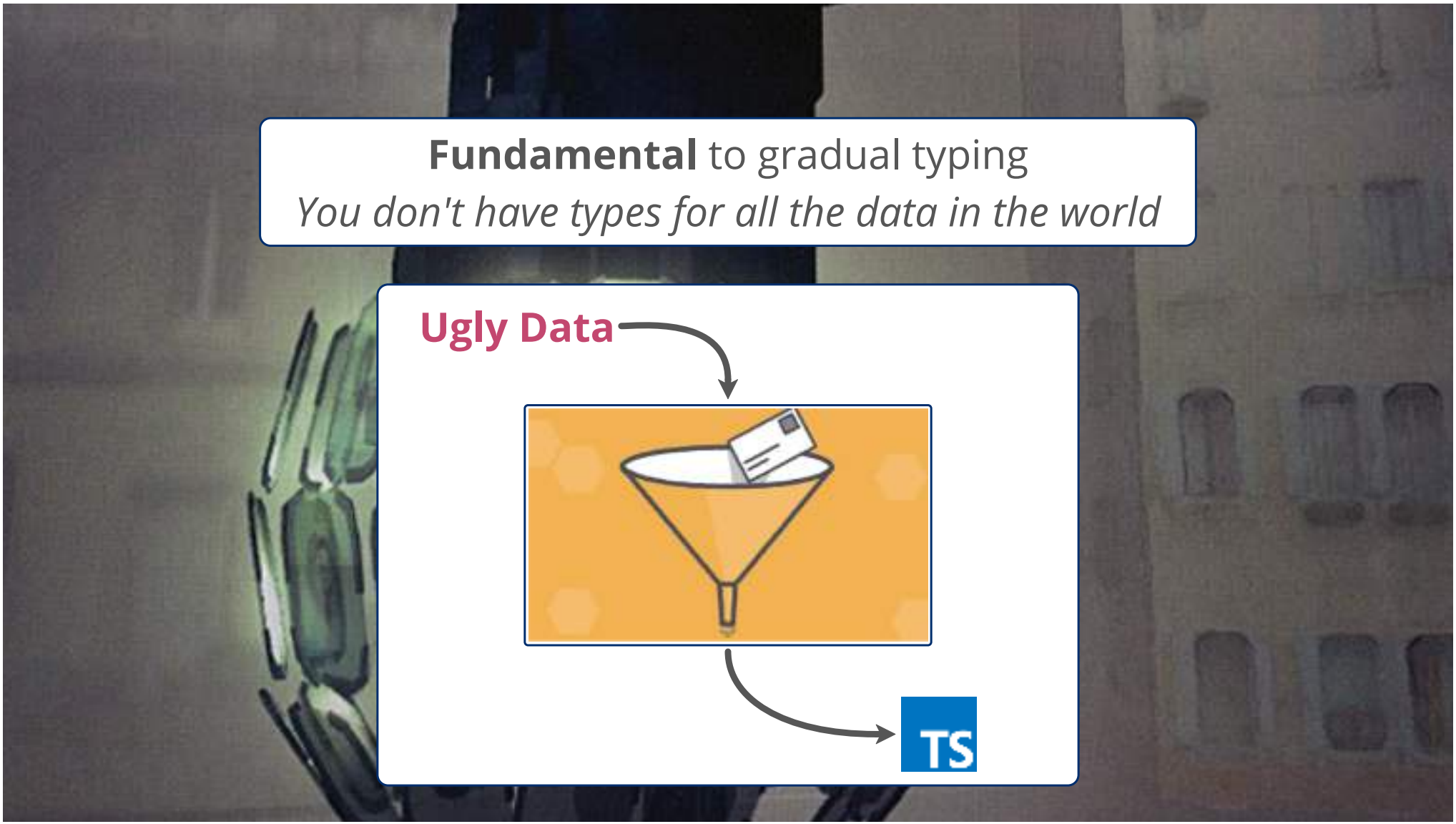

TS
Unsound?!
function f(x: T): x is string {
return true
}
LIES

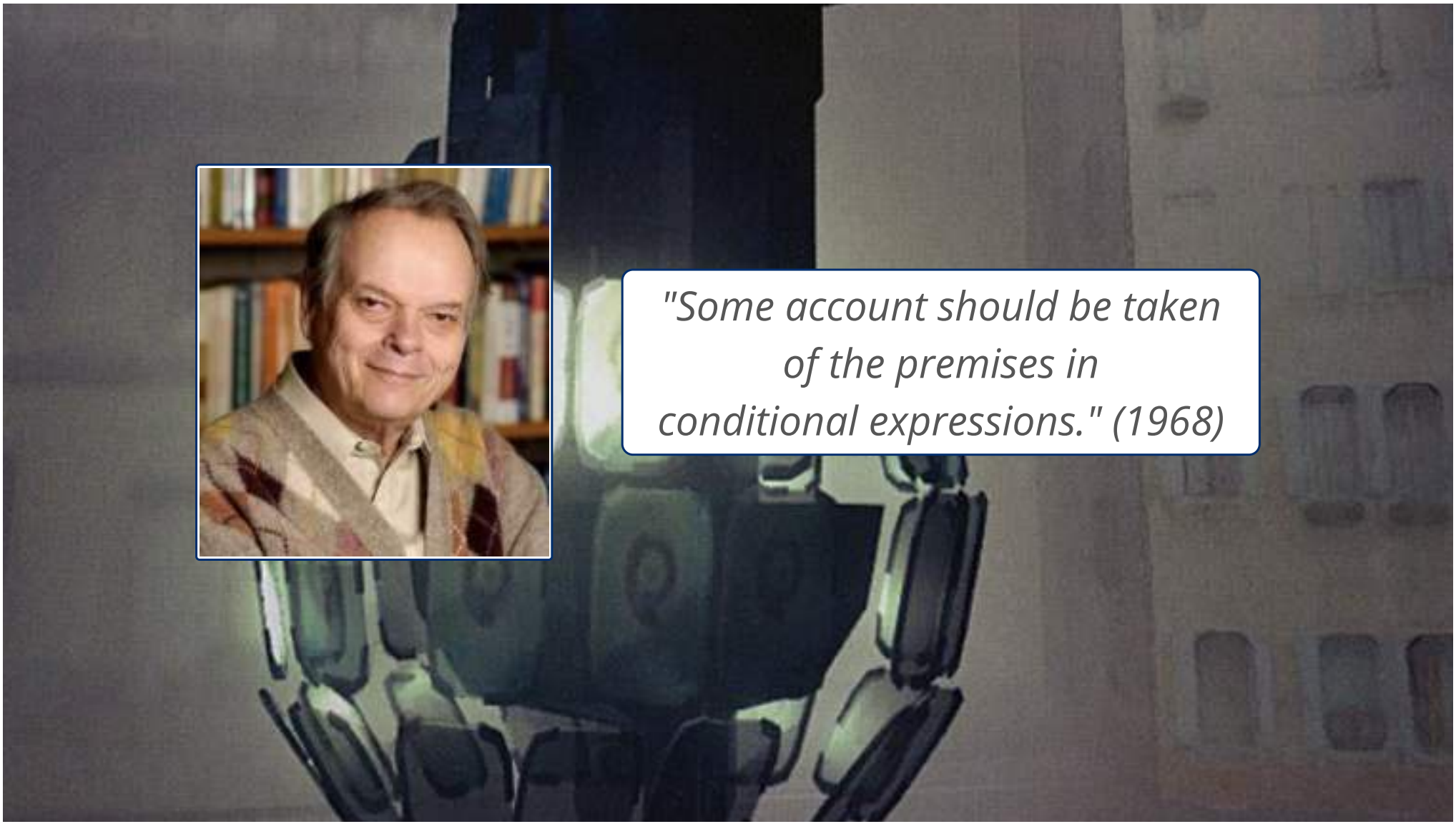

TS
Unsound?!
function f(x: T): x is string {
return true
}
LIES
: my[py]
PYRIGHT
Also Bad
Slightly better

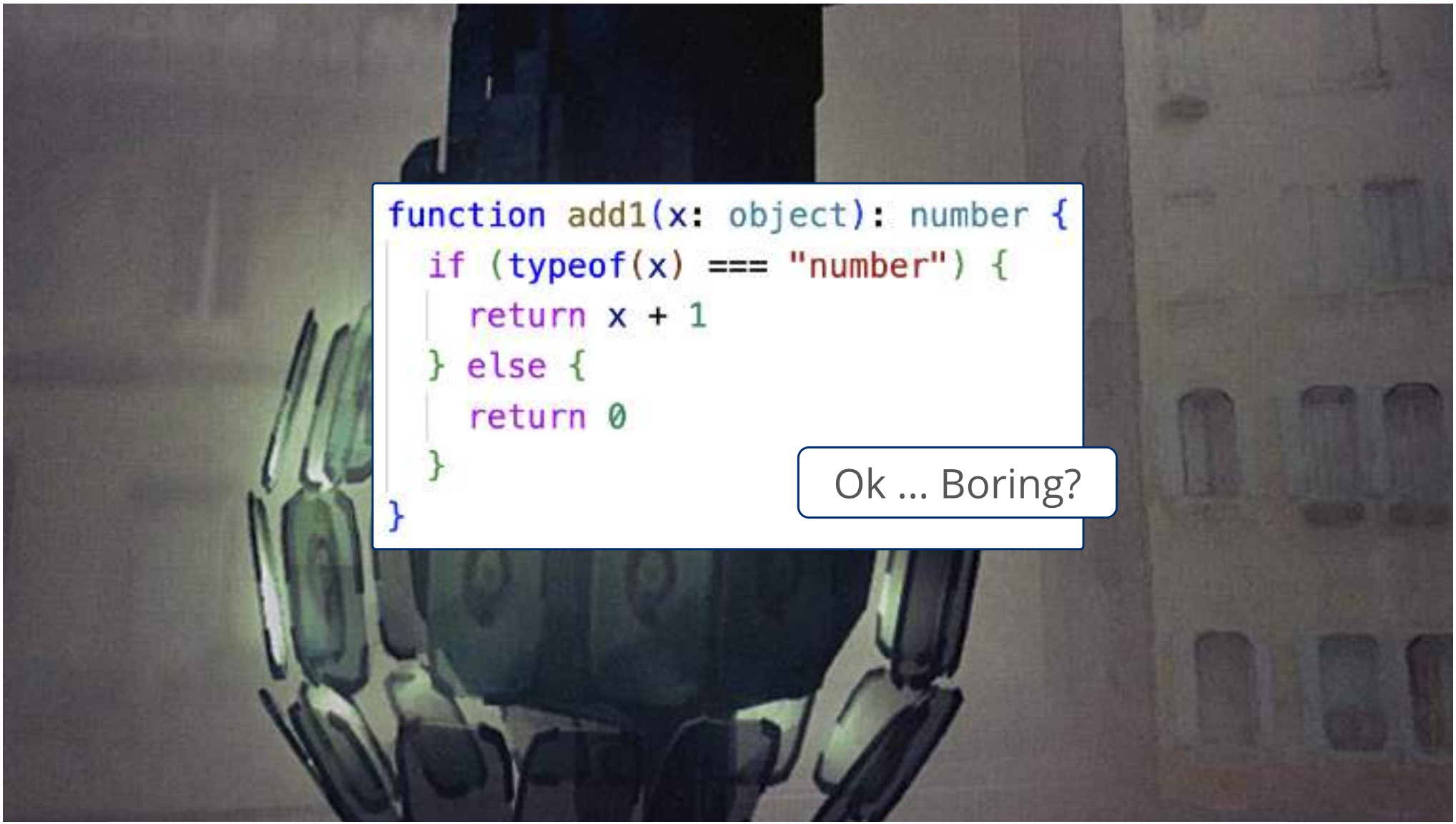
Type Narrowing

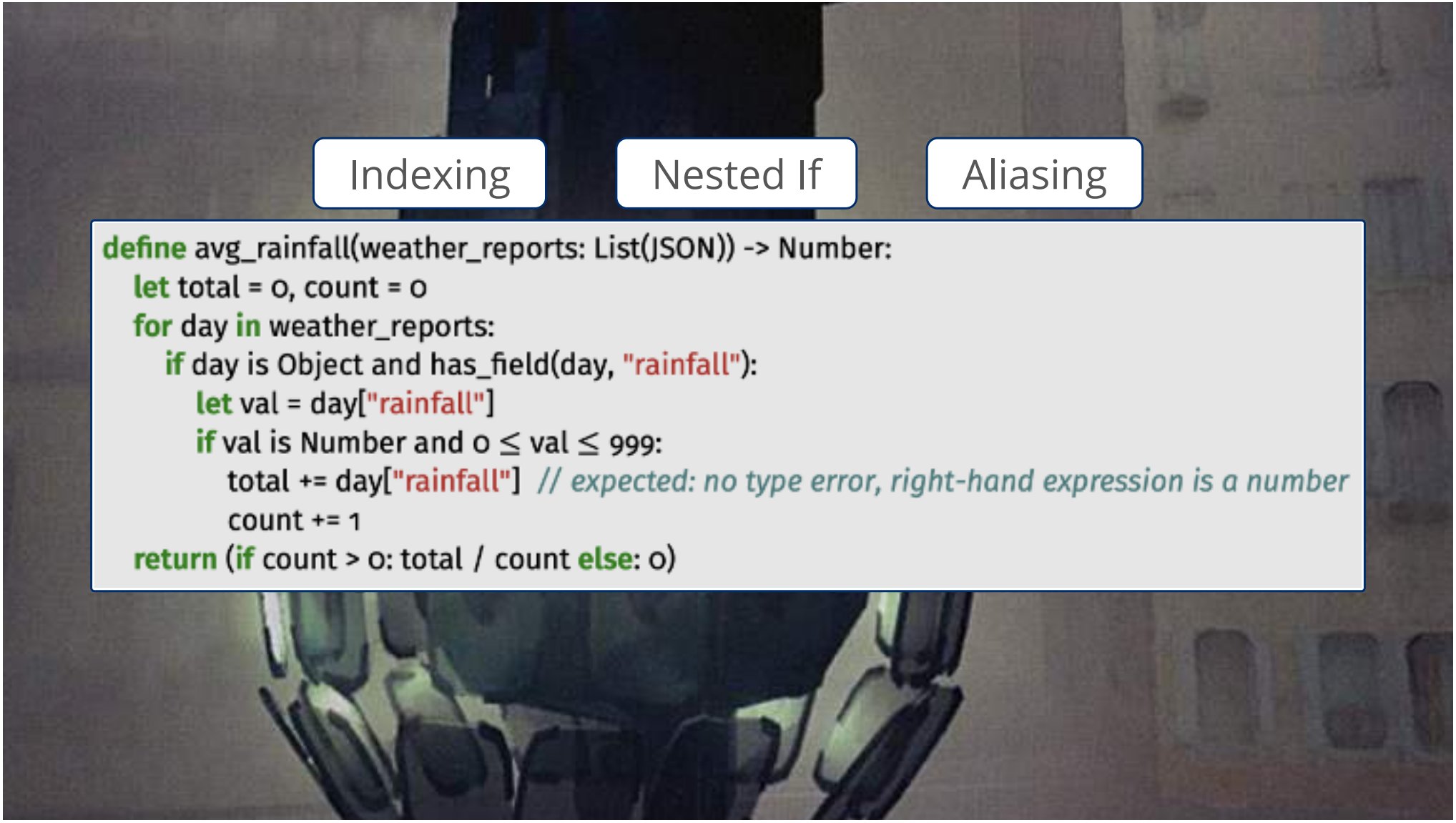
Refines types using tests

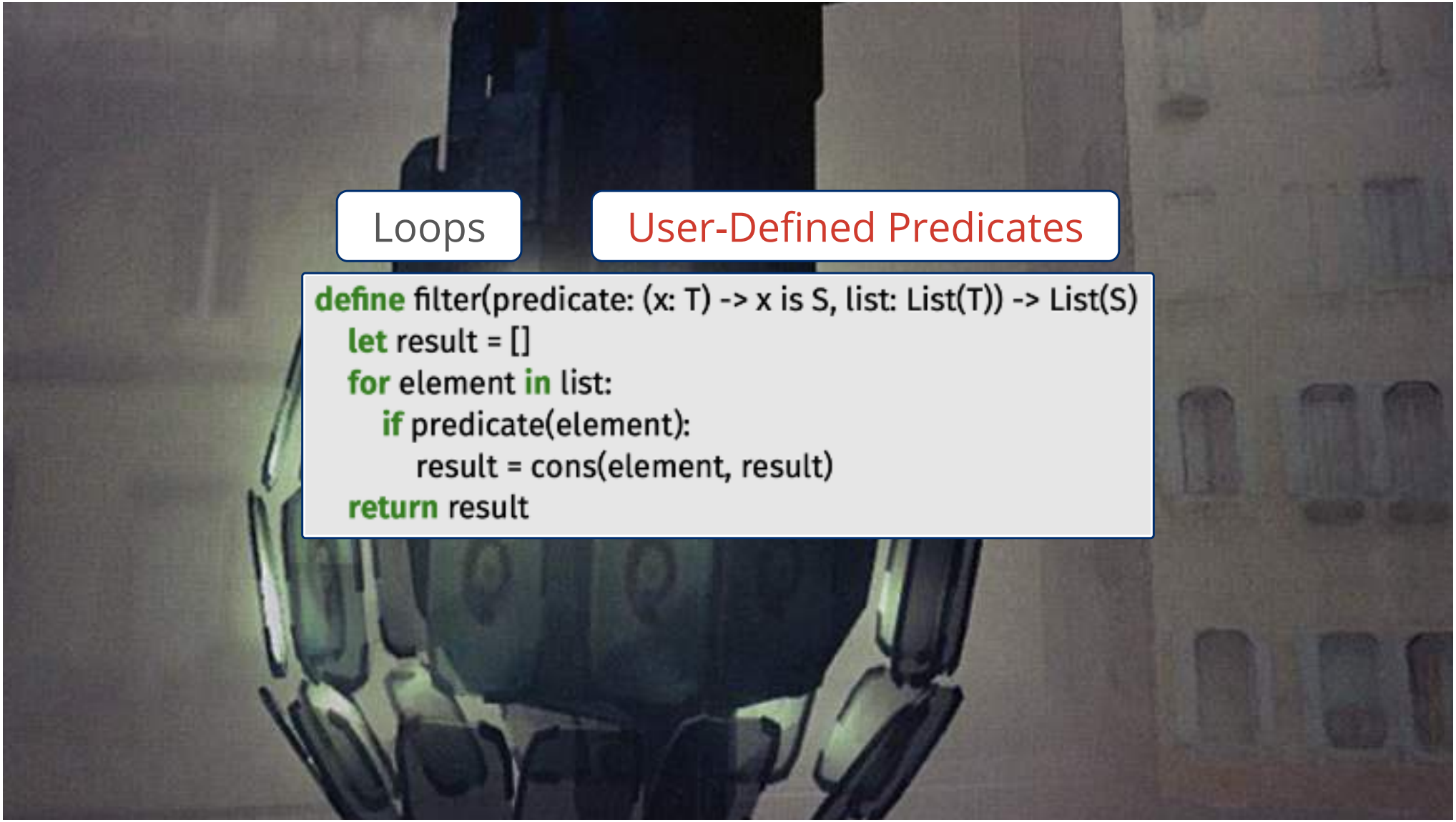
```
function add1(x: object): number {
  if (typeof(x) === "number") {
    return x + 1
  } else {
    return 0
  }
}
```

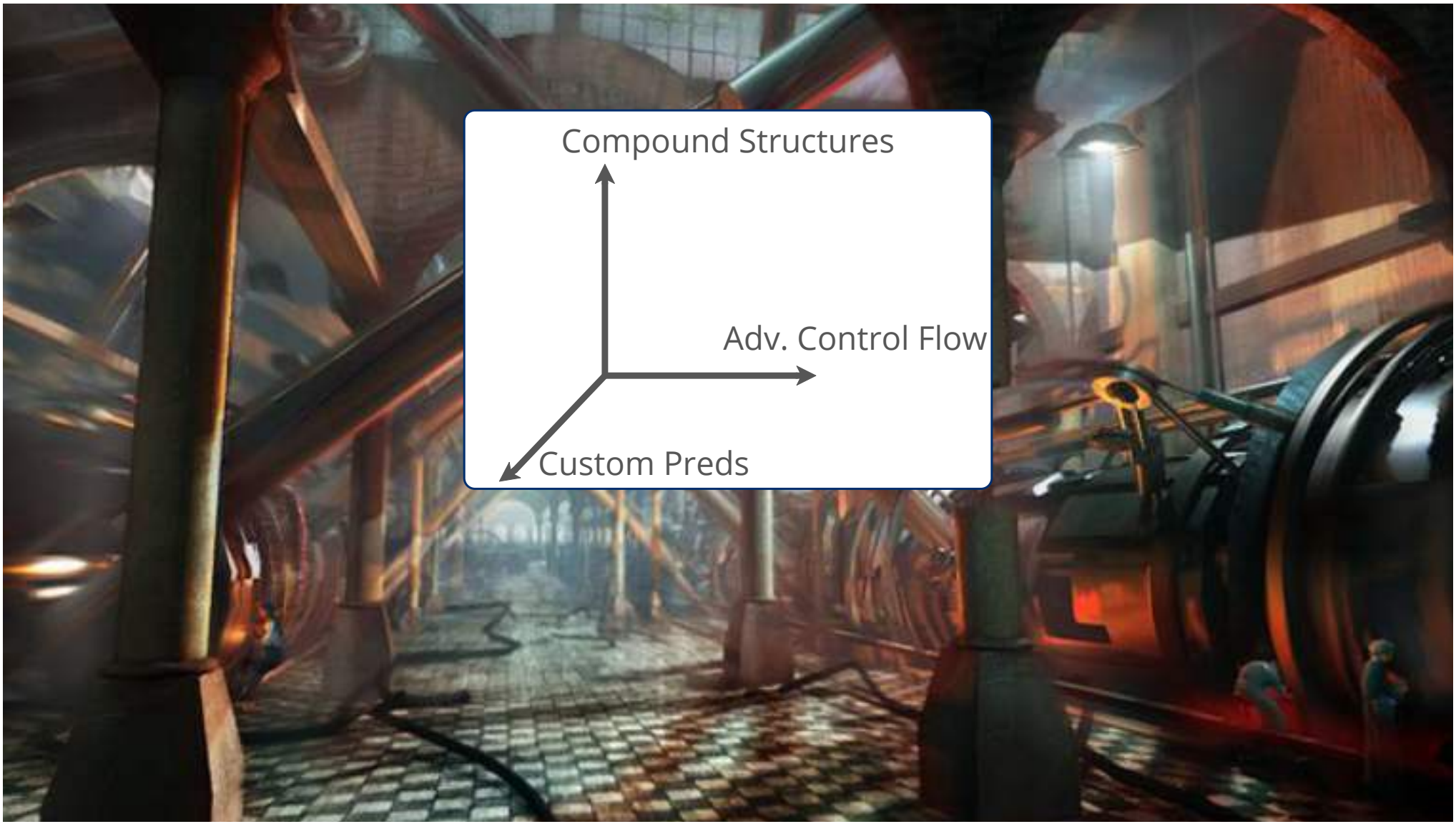

Fundamental to gradual typing
You don't have types for all the data in the world
Ugly Data
TS

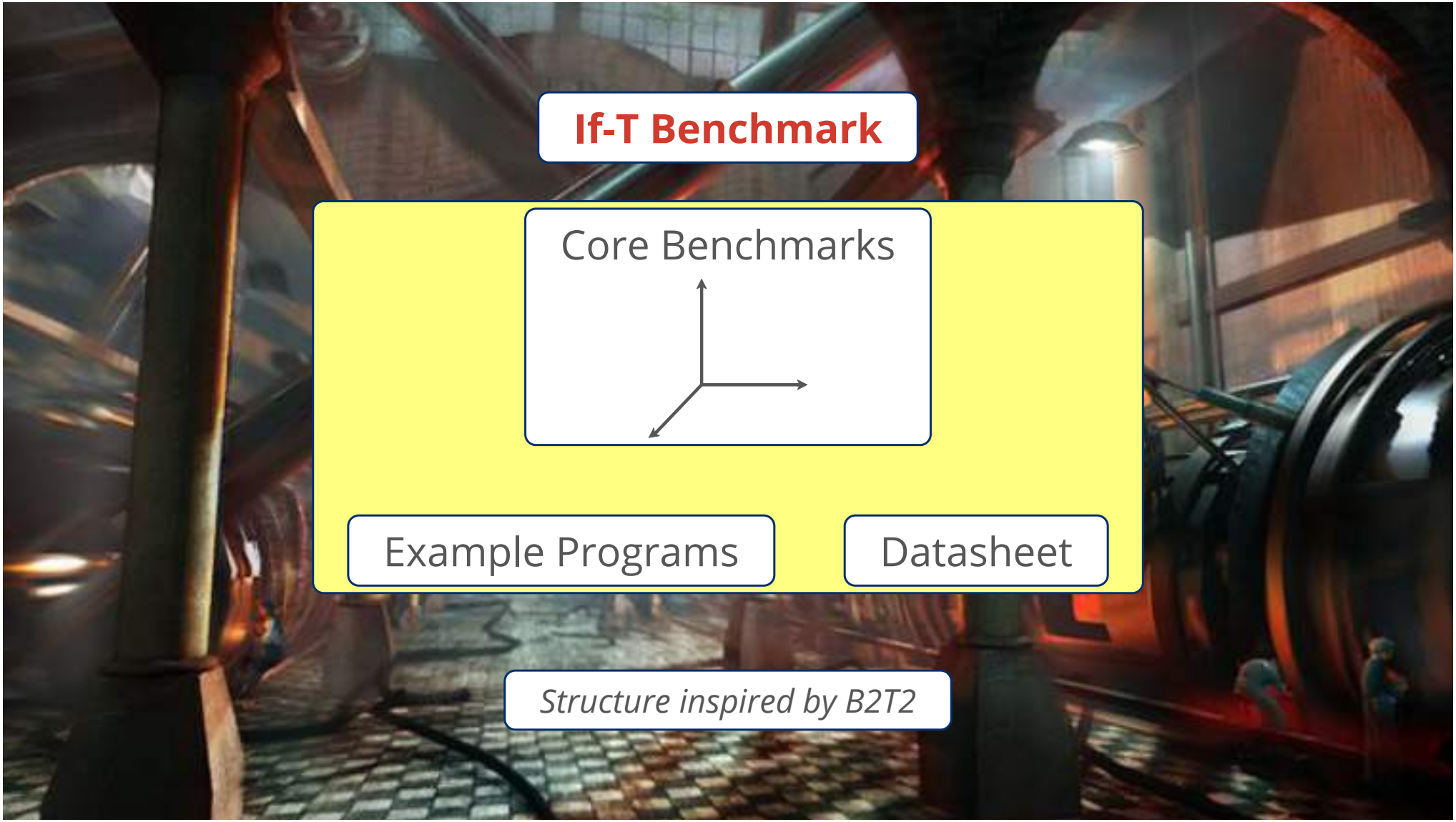

"Some account should be taken of the premises in conditional expressions." (1968)

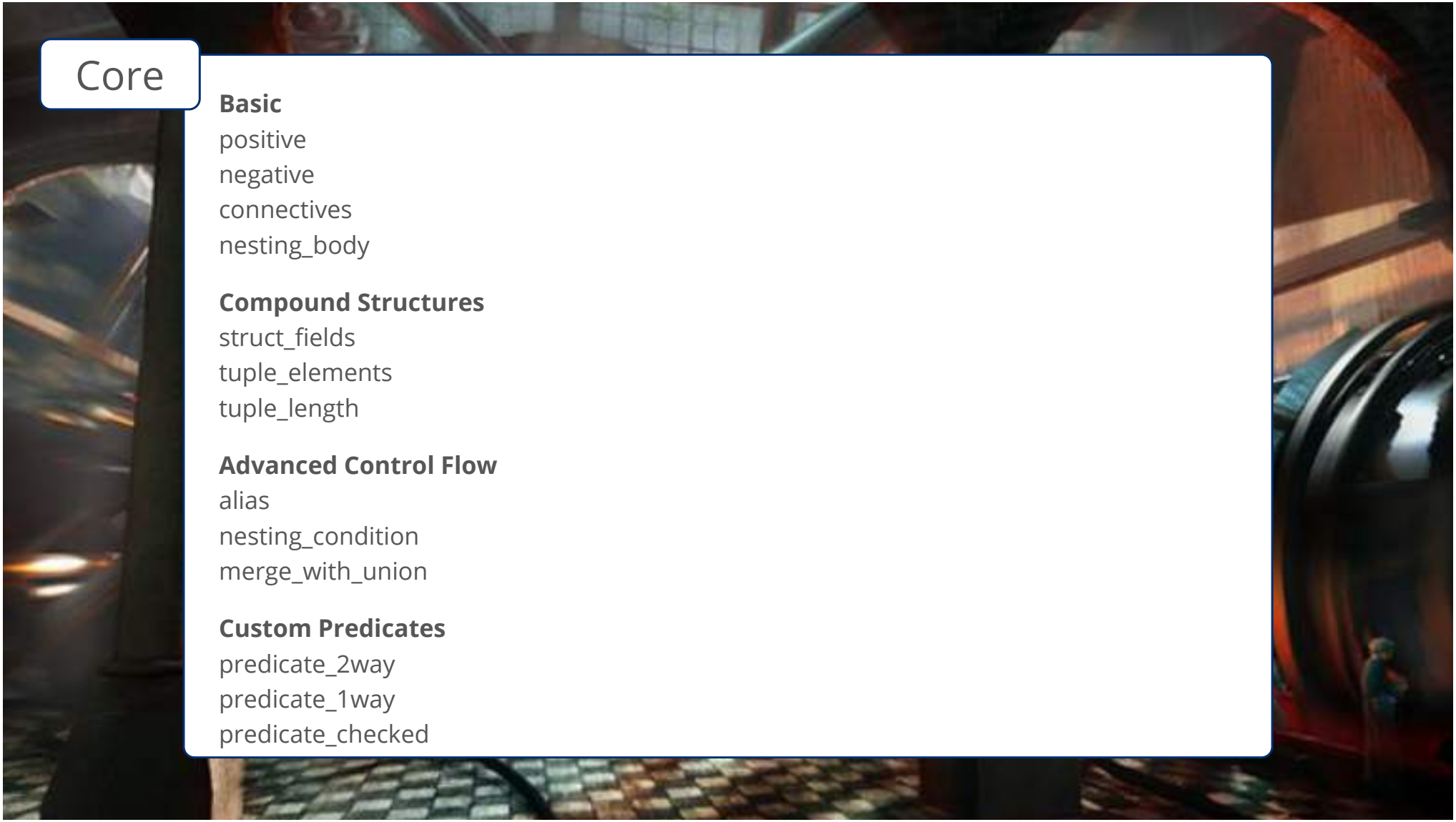

function add1(x: object): number {
if (typeof(x) === "number") {
return x + 1
} else {
return 0
}
}
Ok ... Boring?


Indexing
Nested If
Aliasing
define avg_rainfall(weather_reports: List(JSON)) -> Number:
let total = 0, count = 0
for day in weather_reports:
if day is Object and has_field(day, "rainfall"):
let val = day["rainfall"]
if val is Number and 0 ≤ val ≤ 999:
total += day["rainfall"] // expected: no type error, right-hand expression is a number
count += 1
return (if count > 0: total / count else: 0)


Loops
User-Defined Predicates
define filter(predicate: (x: T) -> x is S, list: List(T)) -> List(S)
let result = []
for element in list:
if predicate(element):
result = cons(element, result)
return result


Compound Structures
Adv. Control Flow
Custom Preds


If-T Benchmark
Core Benchmarks
Example Programs
Datasheet
Structure inspired by B2T2



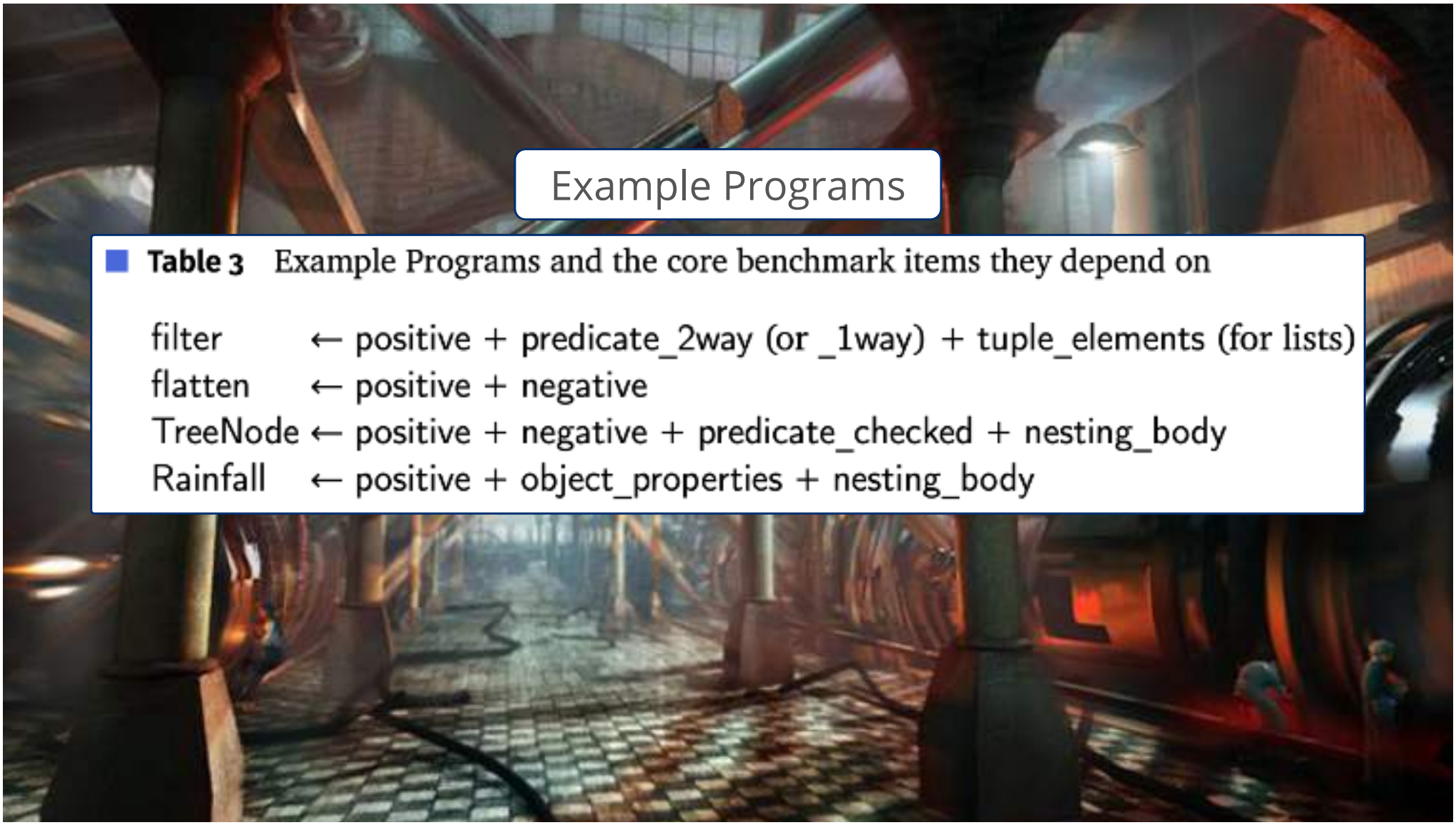
Core

Basic
positive
negative
connectives
nesting_body

Compound Structures
struct_fields
tuple_elements
tuple_length

Advanced Control Flow
alias
nesting_condition
merge_with_union

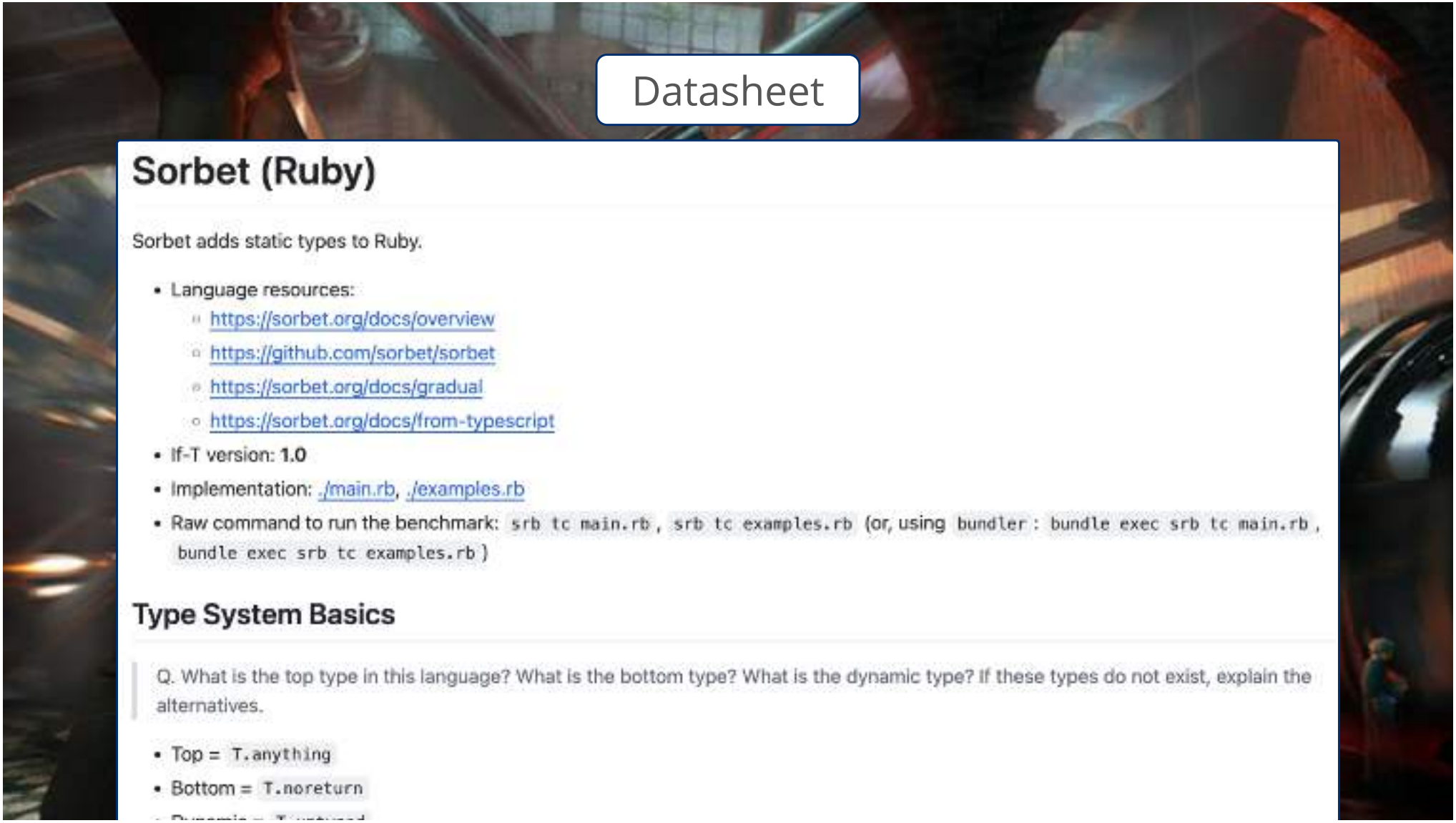
Custom Predicates
predicate_2way
predicate_1way
predicate_checked



Example Programs

Table 3 Example Programs and the core benchmark items they depend on

filter	← positive + predicate_2way (or _1way) + tuple_elements (for lists)
flatten	← positive + negative
TreeNode	← positive + negative + predicate_checked + nesting_body
Rainfall	← positive + object_properties + nesting_body



Datasheet

Sorbet (Ruby)

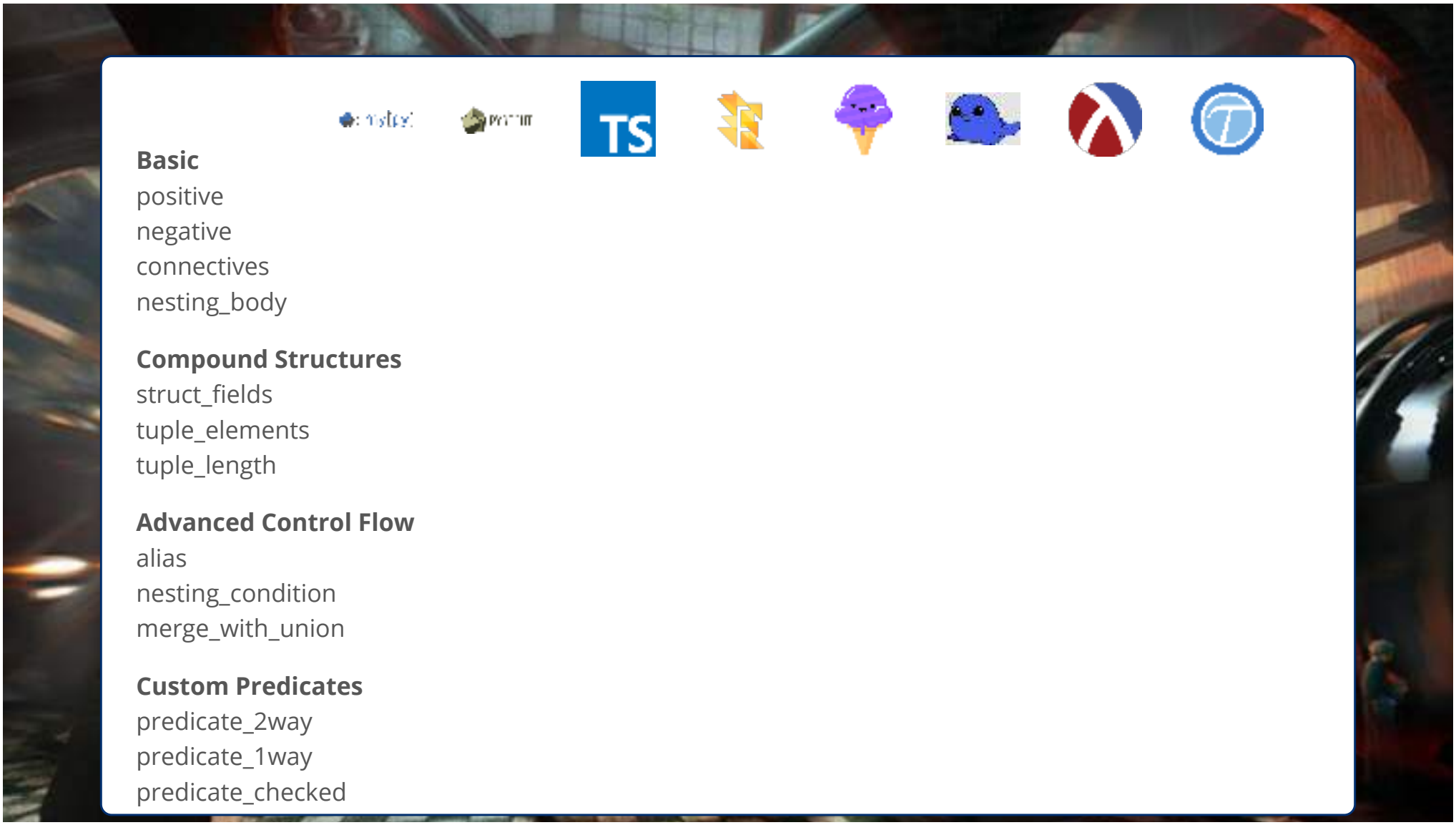
Sorbet adds static types to Ruby.

- 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`

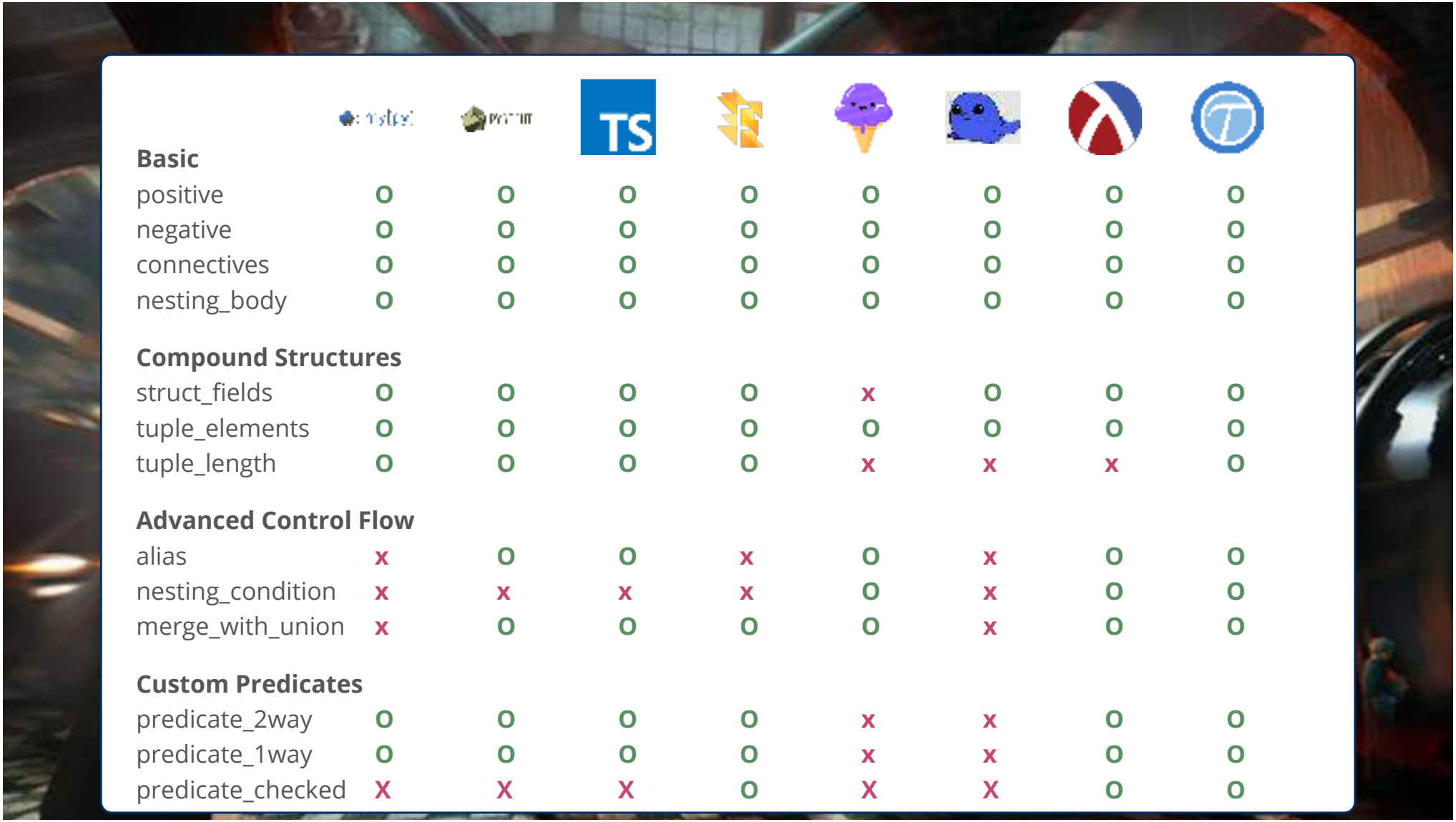


Basic
positive
negative
connectives
nesting_body

Compound Structures
struct_fields
tuple_elements
tuple_length

Advanced Control Flow
alias
nesting_condition
merge_with_union

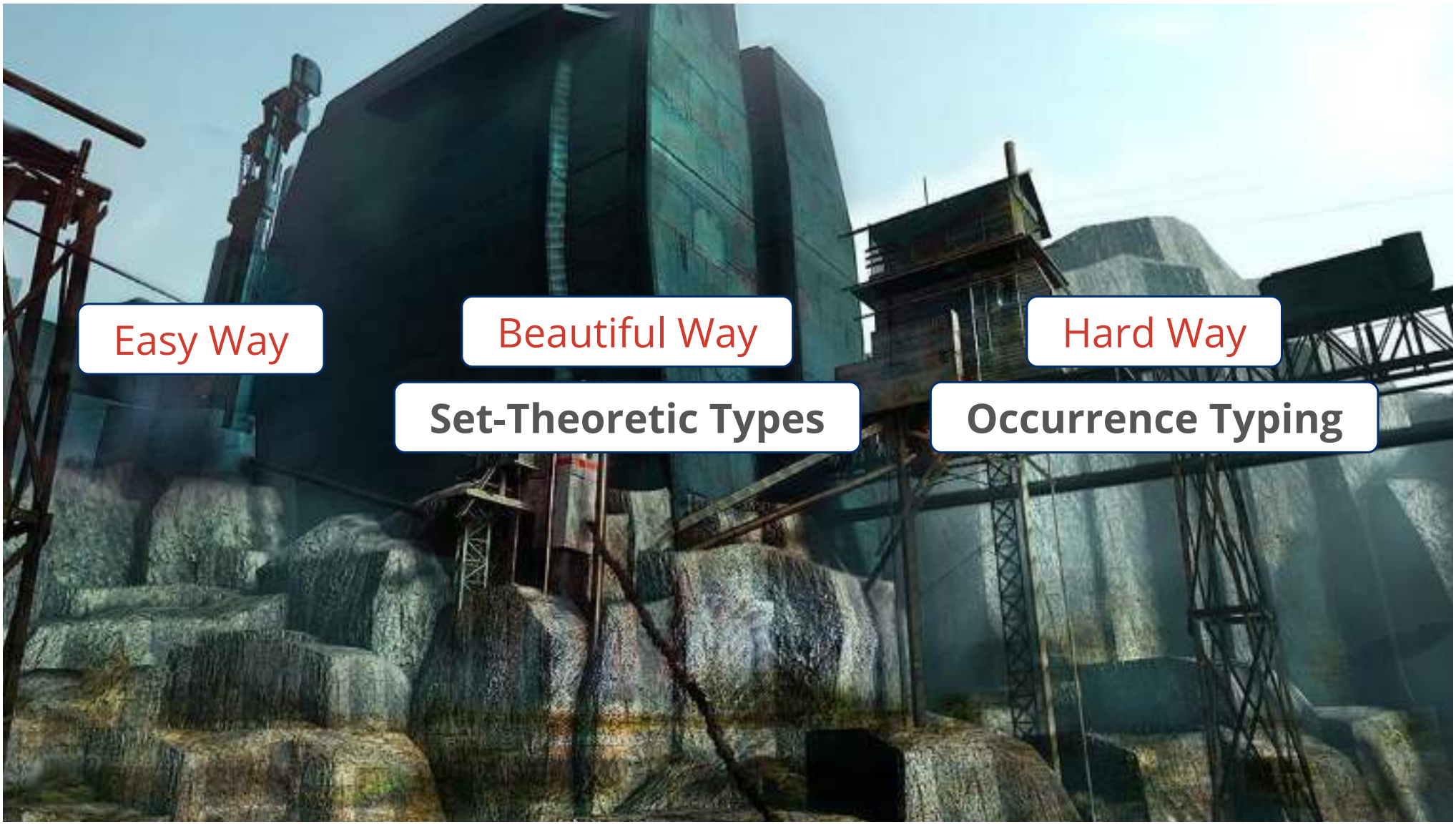
Custom Predicates
predicate_2way
predicate_1way
predicate_checked



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


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

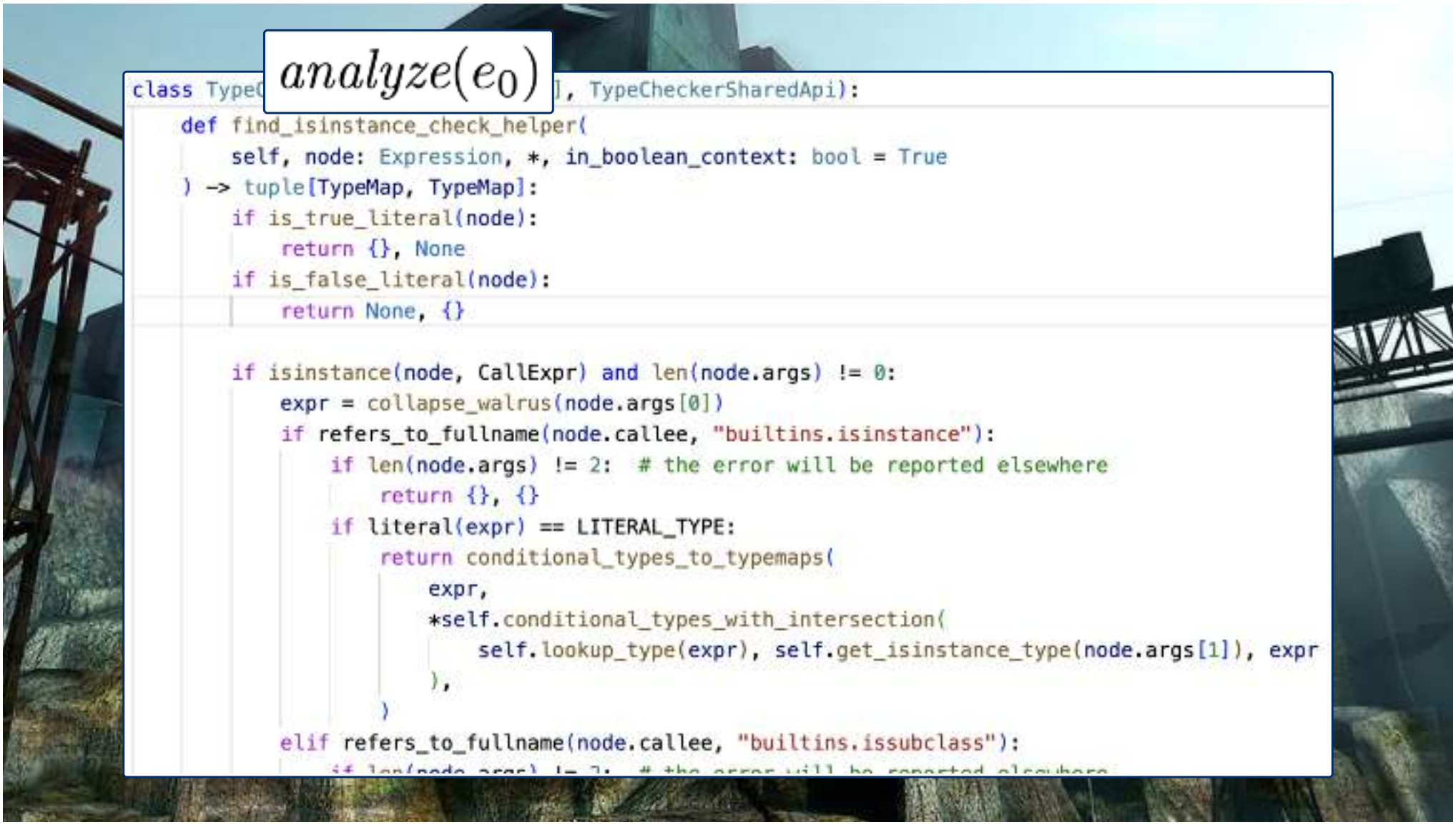

Easy Way
Beautiful Way
Hard Way

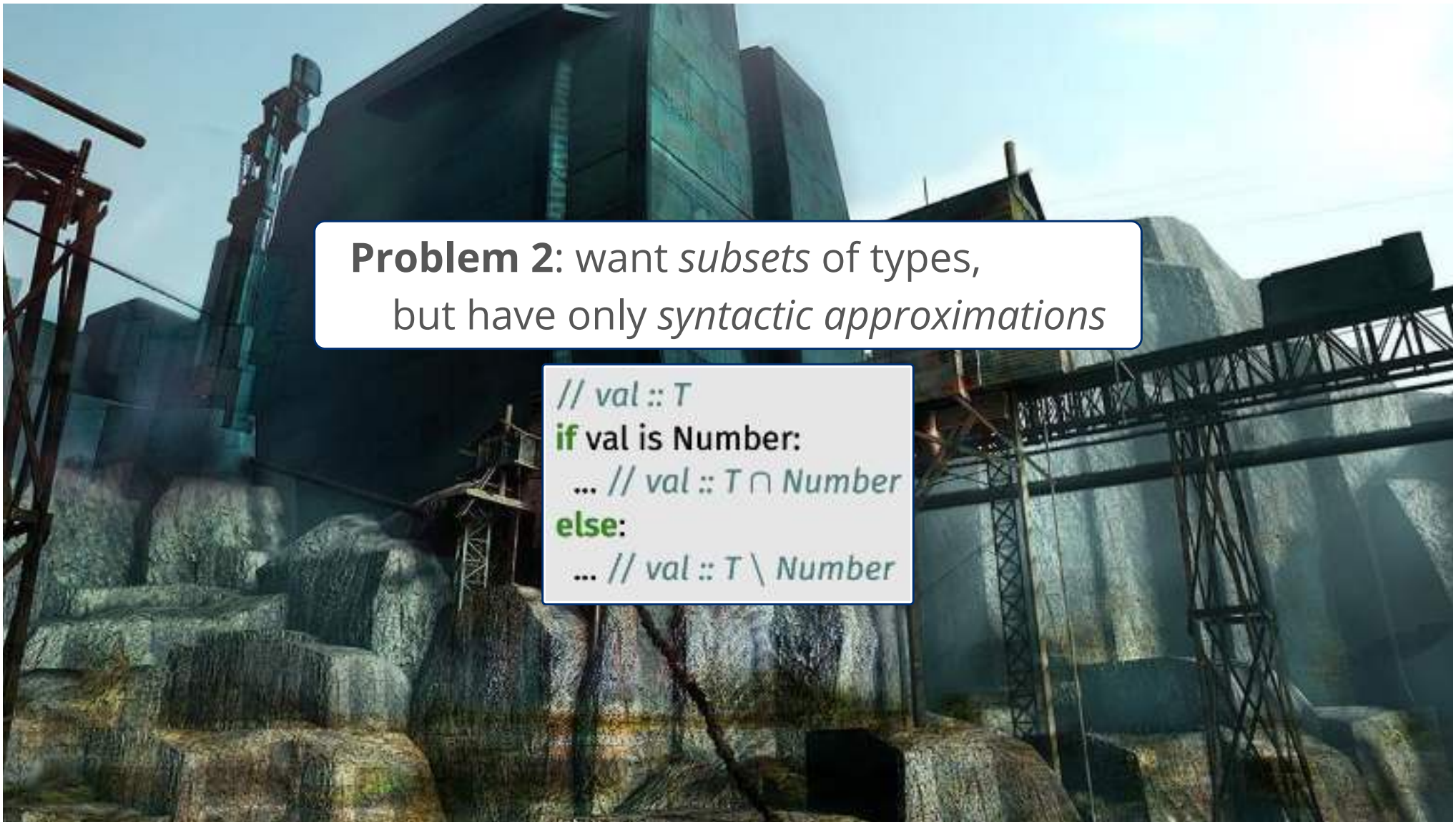

Easy Way
Beautiful Way
Hard Way
Set-Theoretic Types
Occurrence Typing


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

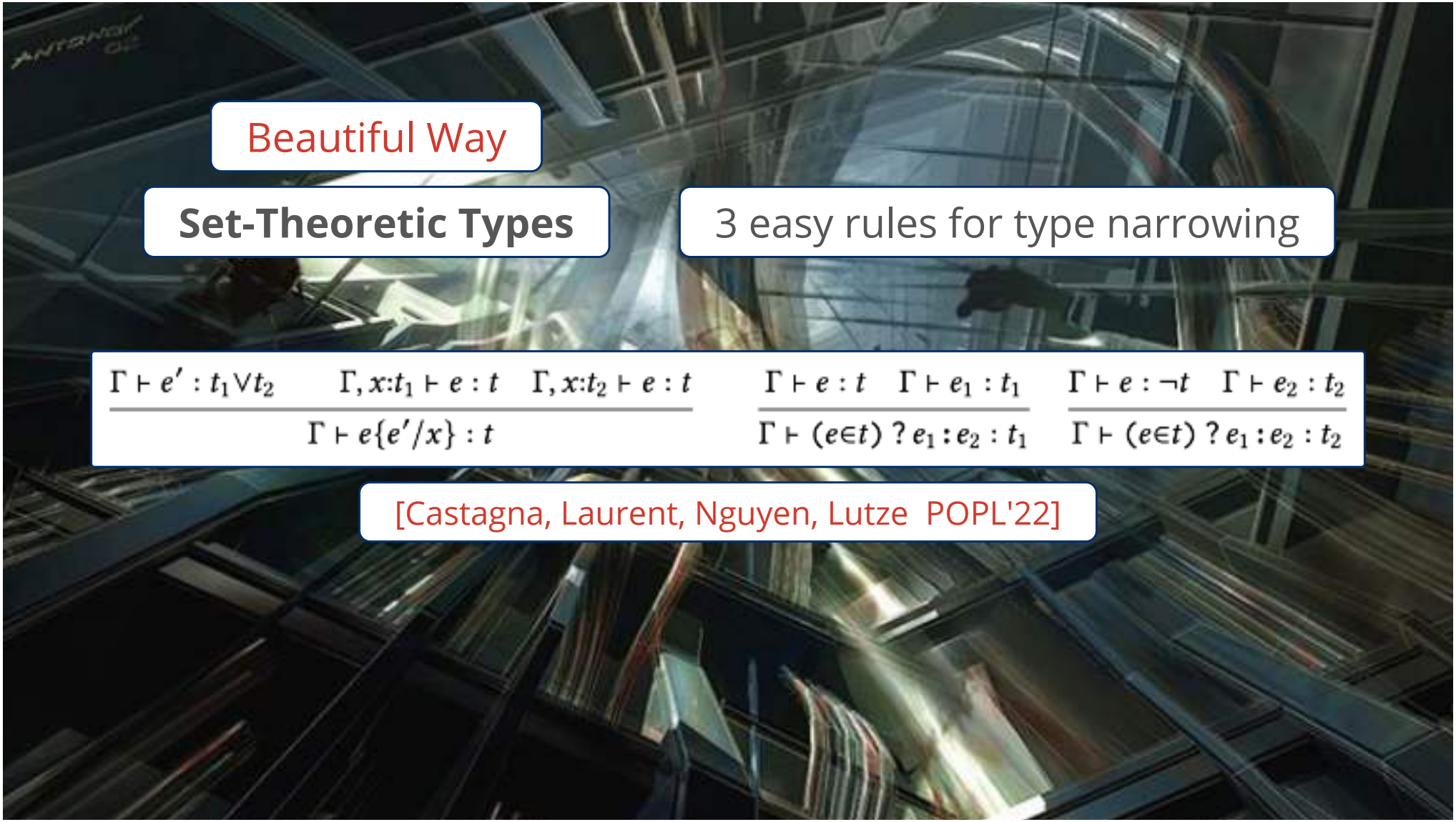

: my[py]
Easy Way
$$\frac{\Gamma \vdash e_0 : \text{Bool} \qquad \Gamma_+, \Gamma_- = analyze(e_0) \qquad \Gamma_+ \cup \Gamma \vdash e_1 : \tau \qquad \Gamma_- \cup \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_0\ e_1\ e_2 : \tau}$$

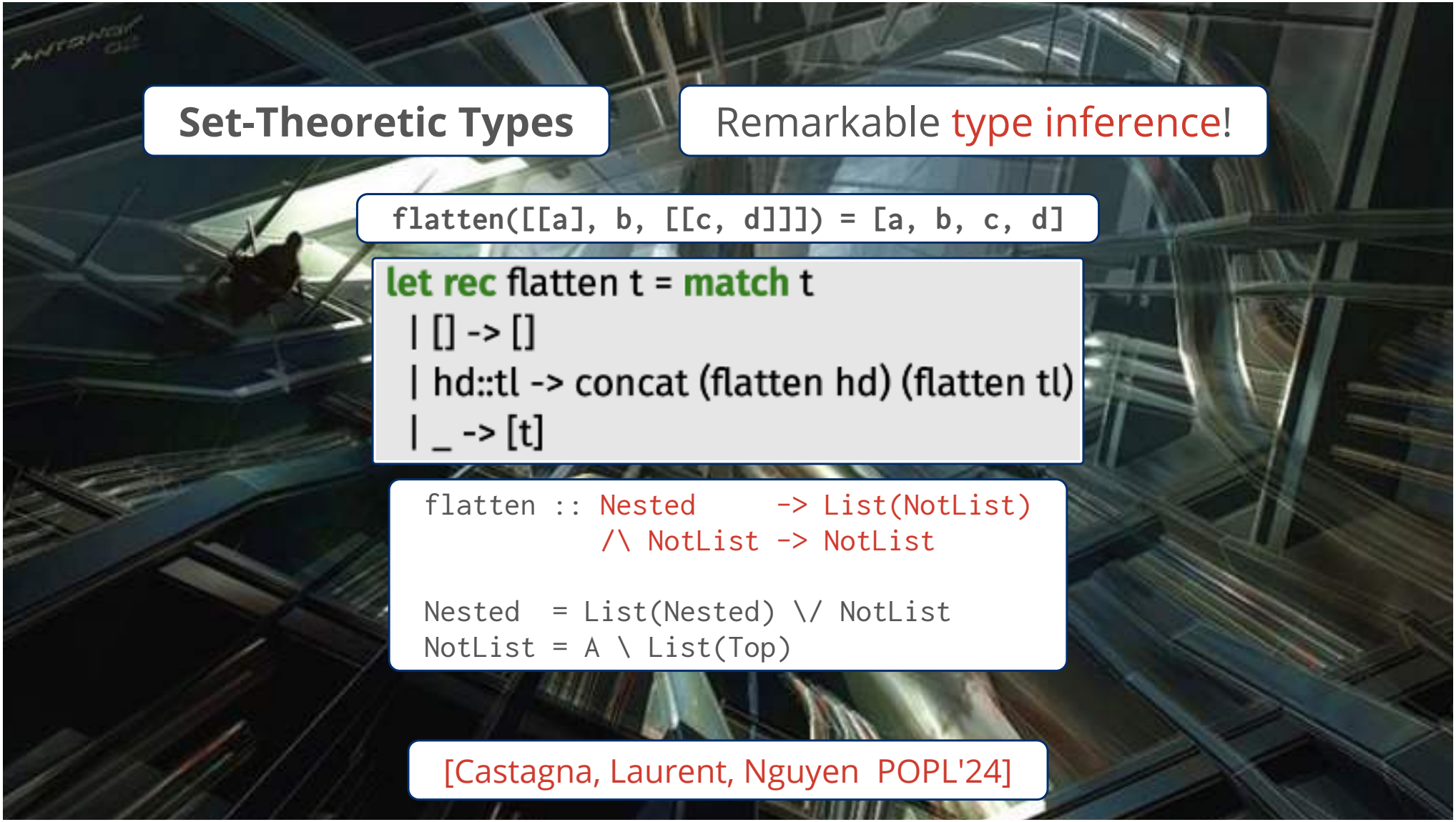

: my[py]
Easy Way
$$\frac{\Gamma_+, \Gamma_- = analyze(e_0) \quad \Gamma \vdash e_0 : \text{Bool} \quad \Gamma_+ \cup \Gamma \vdash e_1 : \tau \quad \Gamma_- \cup \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e_0\ e_1\ e_2 : \tau}$$
class TypeChecker(NodeVisitor[None], TypeCheckerSharedApi):
def 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

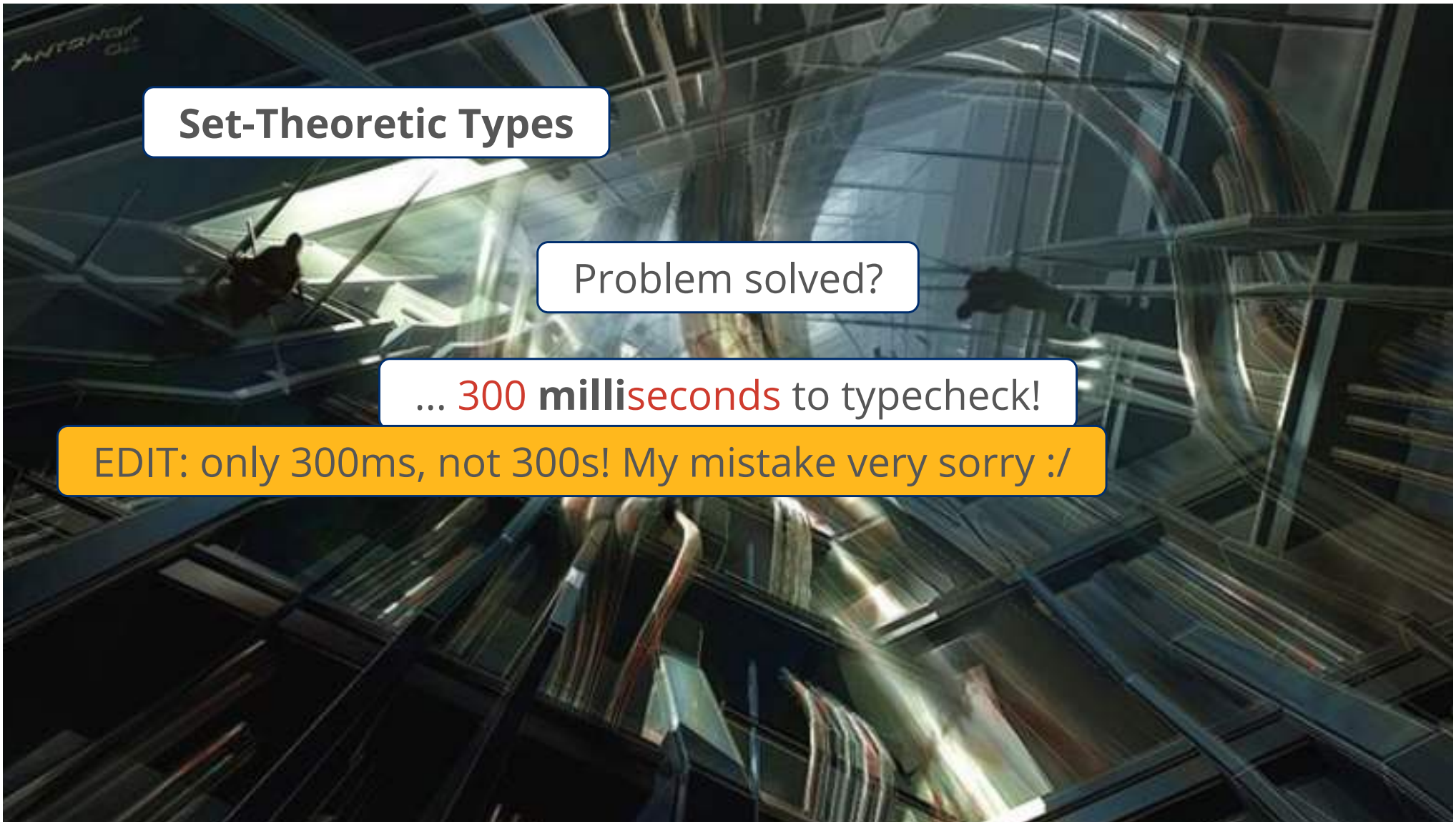

$analyze(e_0)$
class TypeC], TypeCheckerSharedApi):
def find_isinstance_check_helper(
self, node: Expression, *, in_boolean_context: bool = True
) -> tuple[TypeMap, TypeMap]:
if is_true_literal(node):
return {}, None
if is_false_literal(node):
return None, {}
if isinstance(node, CallExpr) and len(node.args) != 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 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
$\dfrac{\Gamma \vdash e' : t_1 \vee t_2 \qquad \Gamma, x{:}t_1 \vdash e : t \qquad \Gamma, x{:}t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$
$\dfrac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : t_1}$
$\dfrac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e \in t)\,?\,e_1 : e_2 : t_2}$
[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 :: Nested -> List(NotList)
/\ NotList -> NotList
Nested = List(Nested) \/ NotList
NotList = A \ List(Top)
[Castagna, Laurent, Nguyen POPL'24]

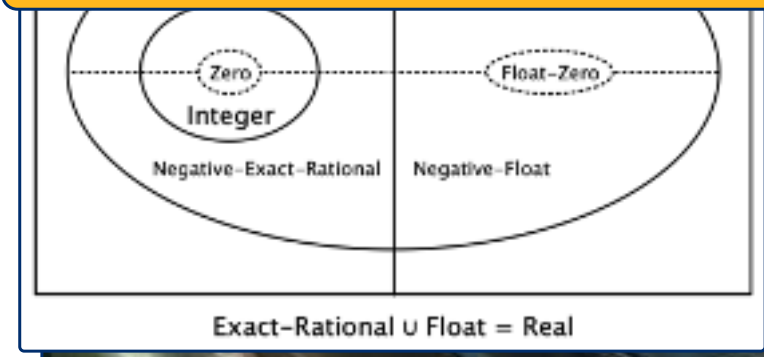

Set-Theoretic Types
Problem solved?
... 300 milliseconds to typecheck!
EDIT: only 300ms, not 300s! My mistake very sorry :/

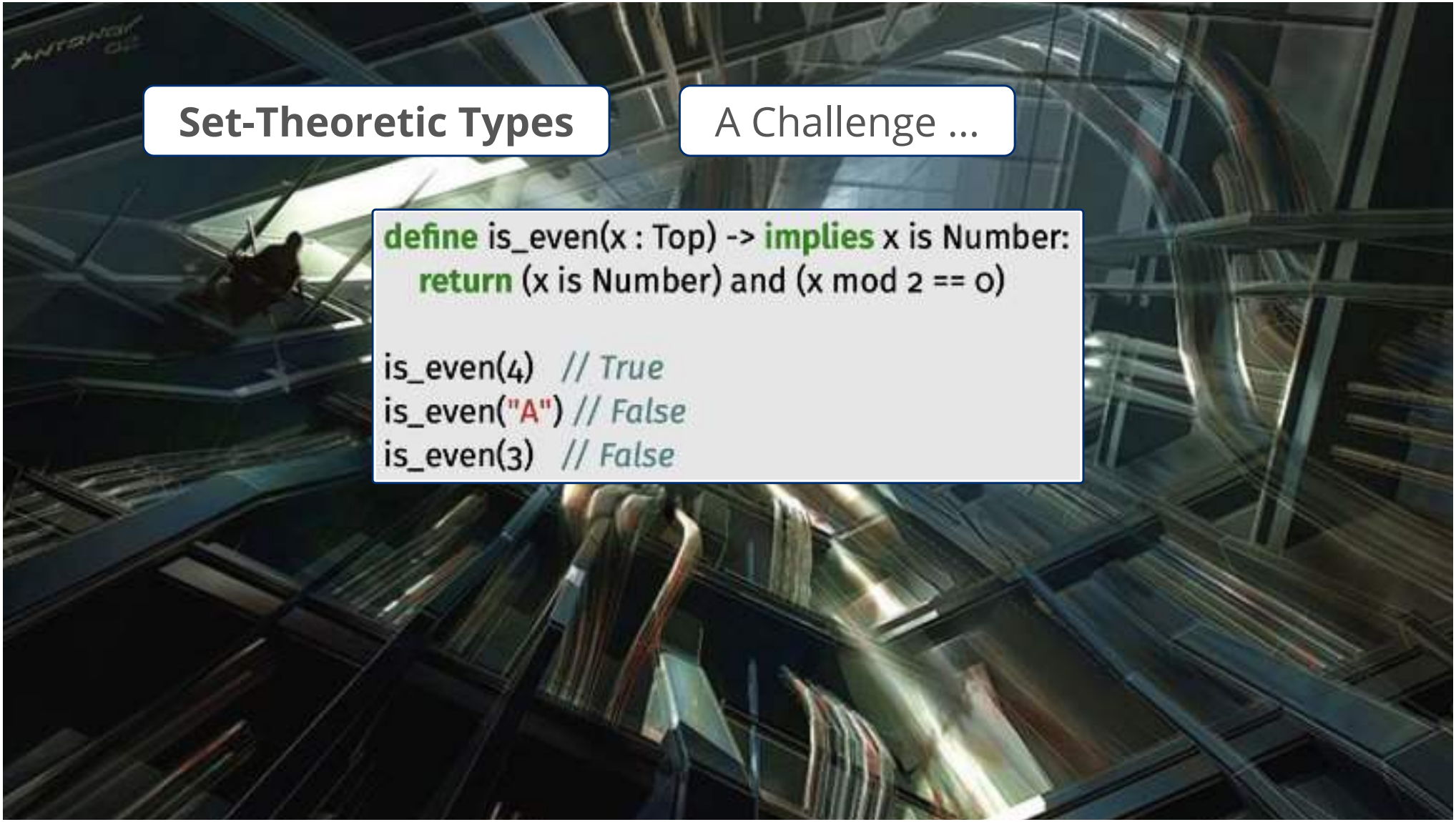
```
let plus : (byte, byte) -> index
         ; (index, index) -> nonnegFixnum
         ; (negFixnum, one) -> nonposFixnum
         ; (one, negFixnum) -> nonposFixnum
         ; (nonposFixnum, nonnegFixnum) -> fixnum
         ; (nonnegFixnum, nonposFixnum) -> fixnum
         ; (integer, integer) -> integer
         ; (float, real) -> float
         ; (real, float) -> float
         ; (singleFloat, rational | singleFloat) -> singleFloat
         ; (rational | singleFloat, singleFloat) -> singleFloat
         ; (posReal, nonnegReal) -> posReal
         ; (nonnegReal, posReal) -> posReal
         ; (negReal, nonposReal) -> negReal
         ; (nonposReal, negReal) -> negReal
         ; (nonnegReal, nonnegReal) -> nonnegReal
         ; (nonposReal, nonposReal) -> nonposReal
         ; (real, real) -> real
         ; (exactNumber, exactNumber) -> exactNumber
         ; (floatComplex, number) -> floatComplex
         ; (number, floatComplex) -> floatComplex
         ; (float, inexactComplex) -> floatComplex
         ; (inexactComplex, float) -> floatComplex
         ; (singleFloatComplex, rational | singleFloat | singleFloatComplex)
           -> singleFloatComplex
         ; (rational | singleFloat | singleFloatComplex, singleFloatComplex)
           -> singleFloatComplex
         ; (number, number) -> number
         )
  | (a, b) -> raise (a,b)
;;

let applyToPair (f : ('a , 'b) -> 'c) (p : ('a , 'b)) : 'c = f p;;

(* This line takes an extremely long time to type check (> 15 min): *)
let addPosBytes (b1 : posByte) (b2 : posByte) : posIndex =
    applyToPair plus (b1, b2);;
```

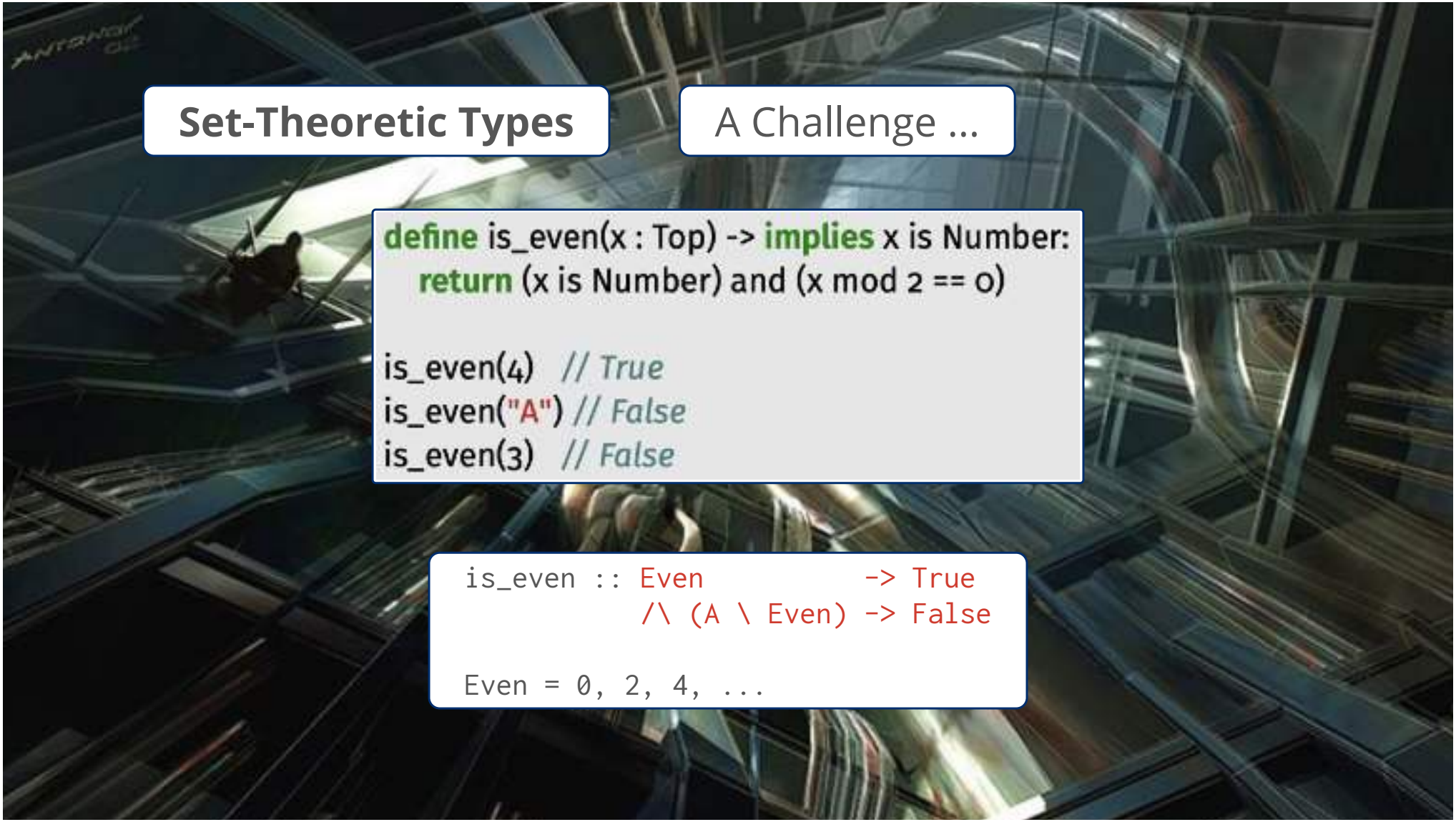
+15 minutes to typecheck!

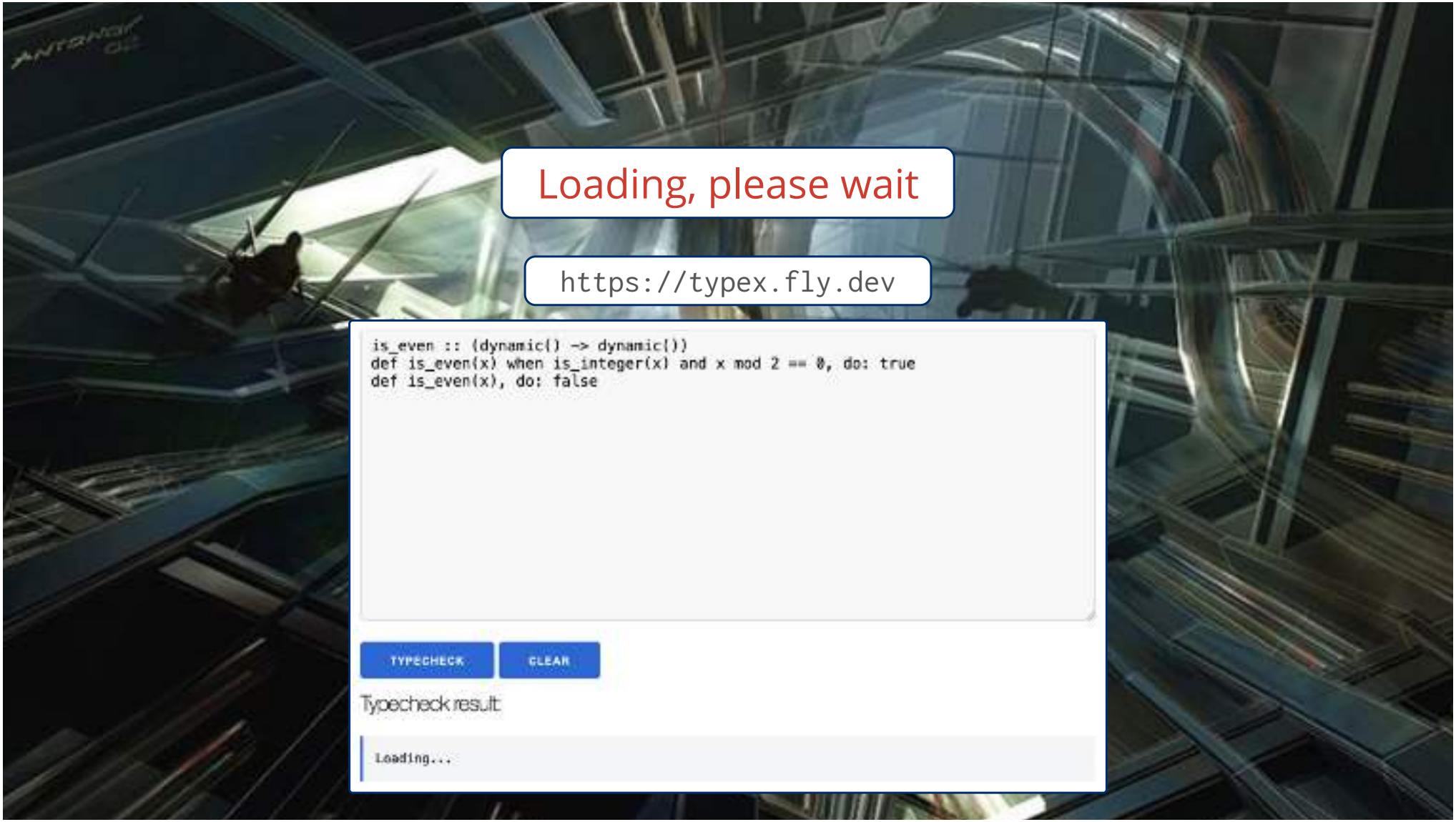
Andrew Kent, 2019

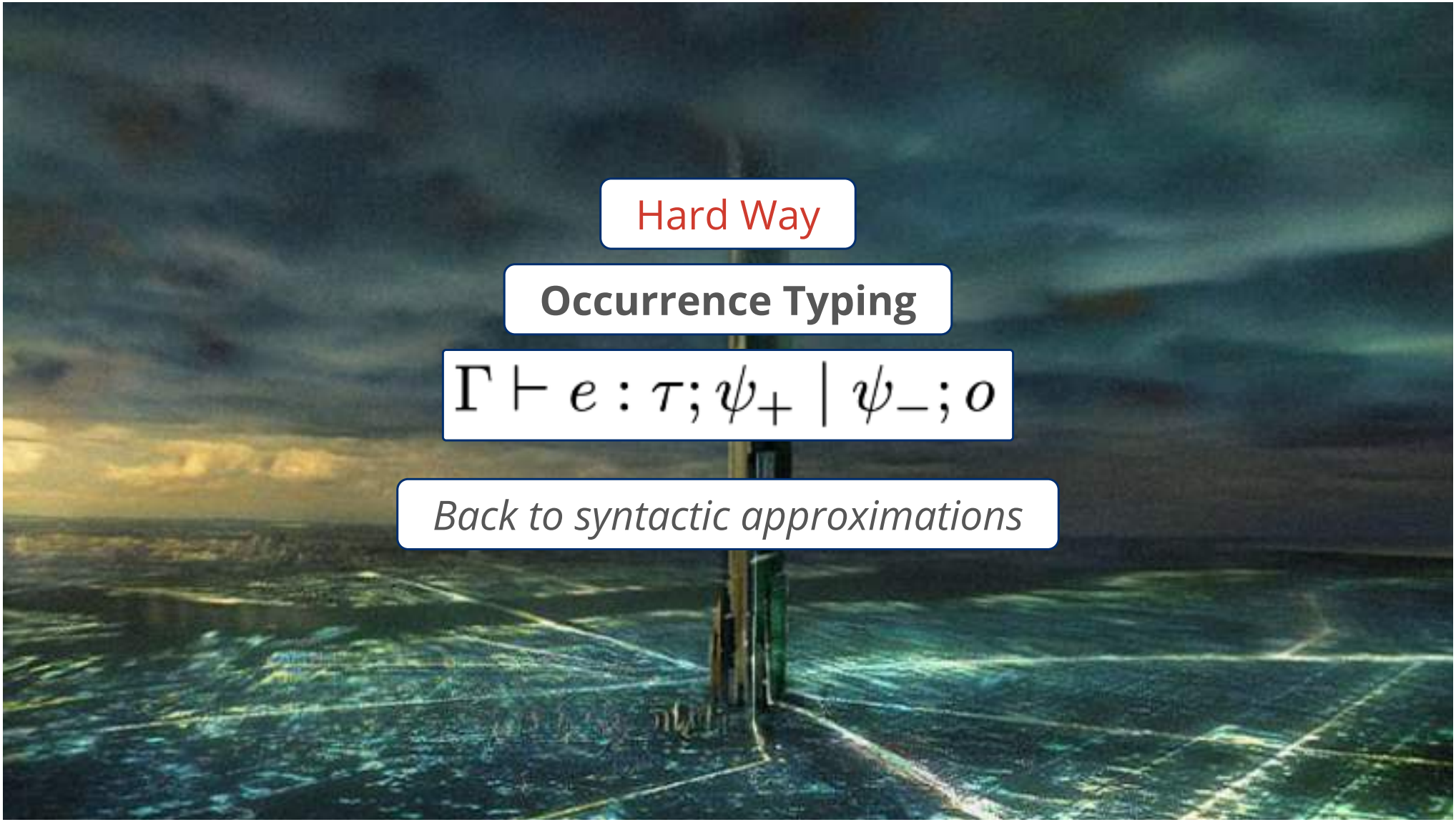
2026: fixed by M. Laurent, J. Valim, G. Duboc

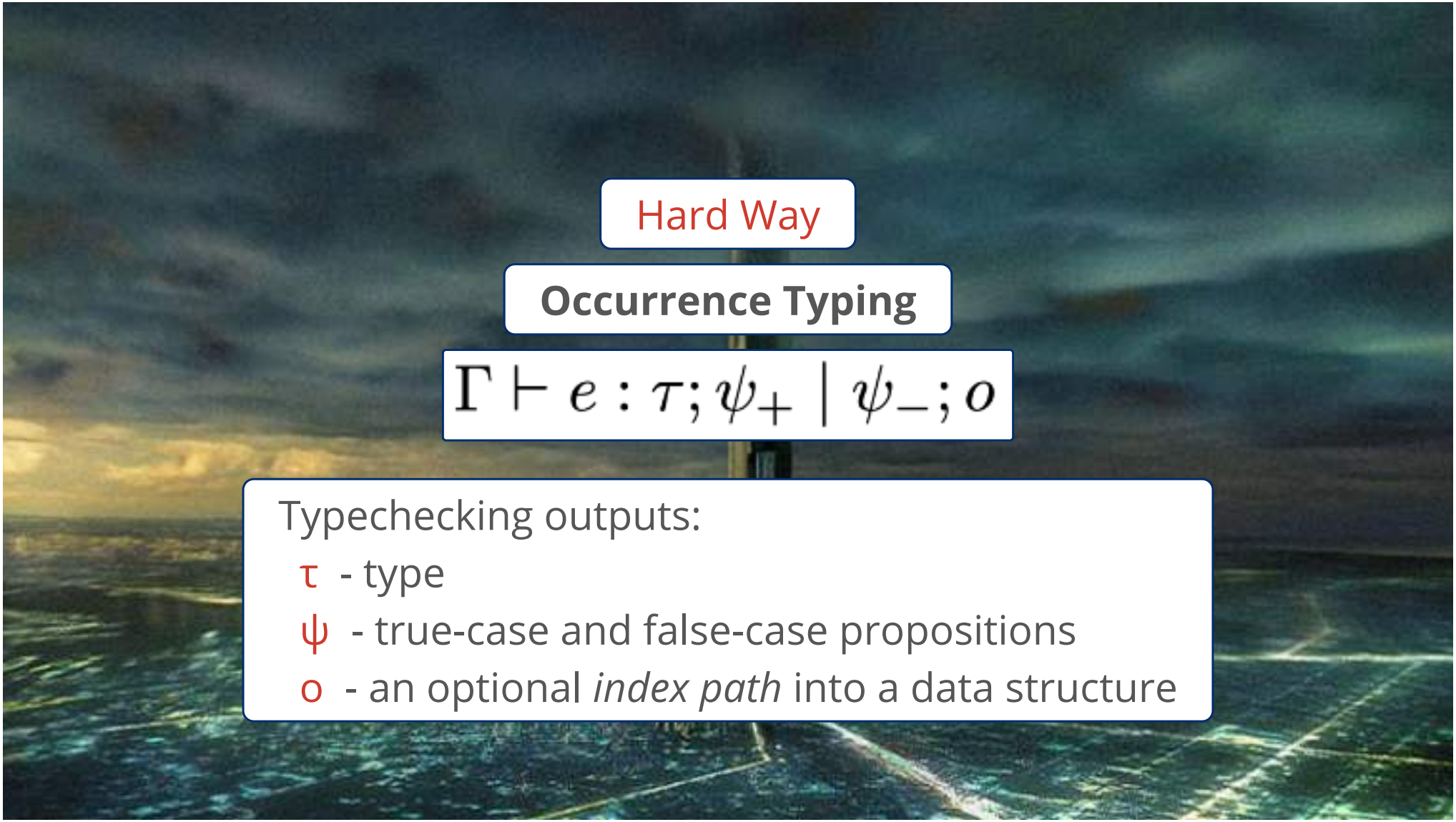


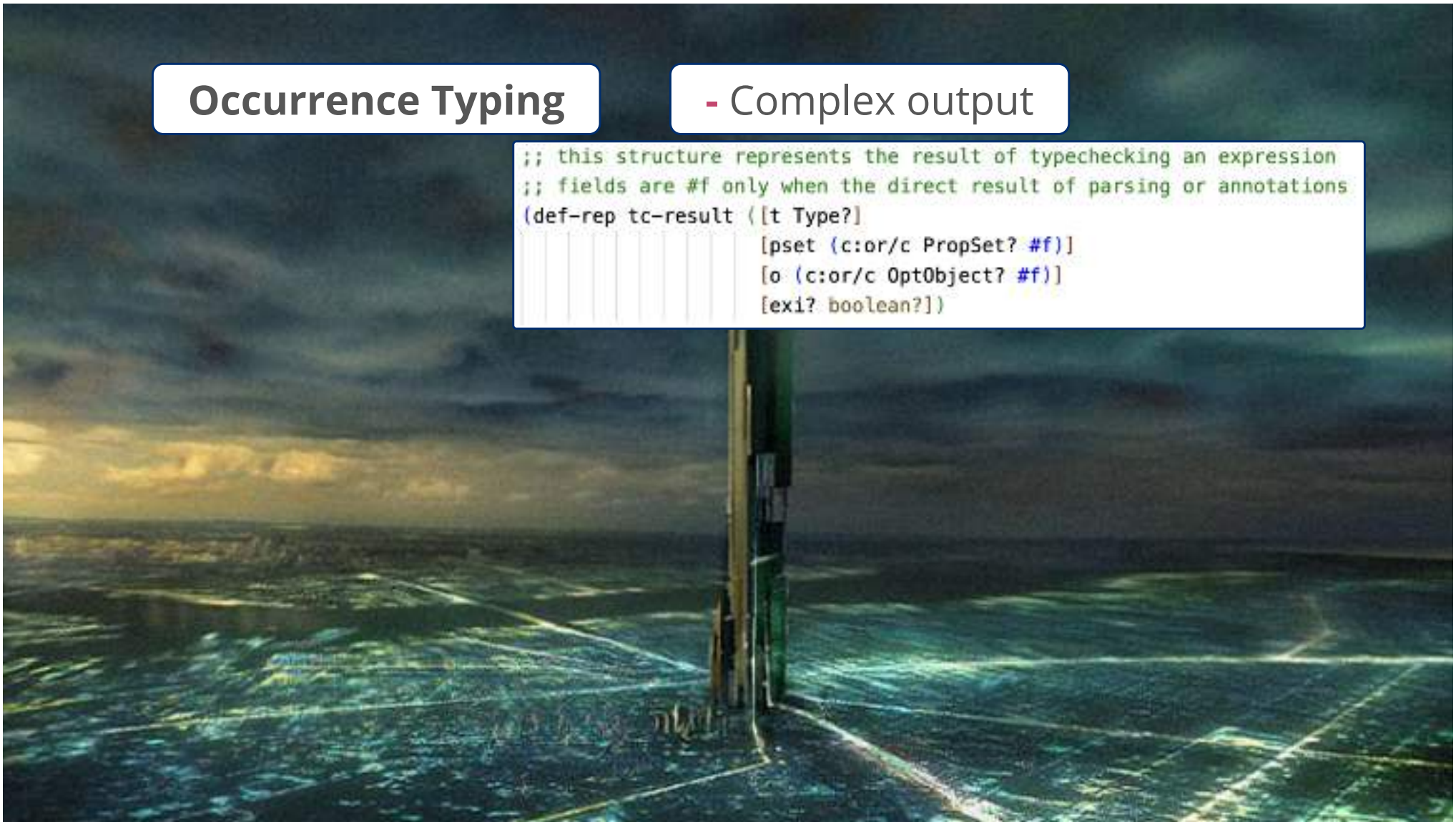

Set-Theoretic Types
A Challenge ...
define is_even(x : Top) -> implies x is Number:
return (x is Number) and (x mod 2 == 0)
is_even(4) // True
is_even("A") // False
is_even(3) // False


Set-Theoretic Types
A Challenge ...
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
is_even :: Even -> True
/\ (A \ Even) -> False
Even = 0, 2, 4, ...


Loading, please wait
https://typex.fly.dev
is_even :: (dynamic() -> dynamic())
def is_even(x) when is_integer(x) and x mod 2 == 0, do: true
def is_even(x), do: false
TYPECHECK
CLEAR
Typecheck result:
Loading...


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?])

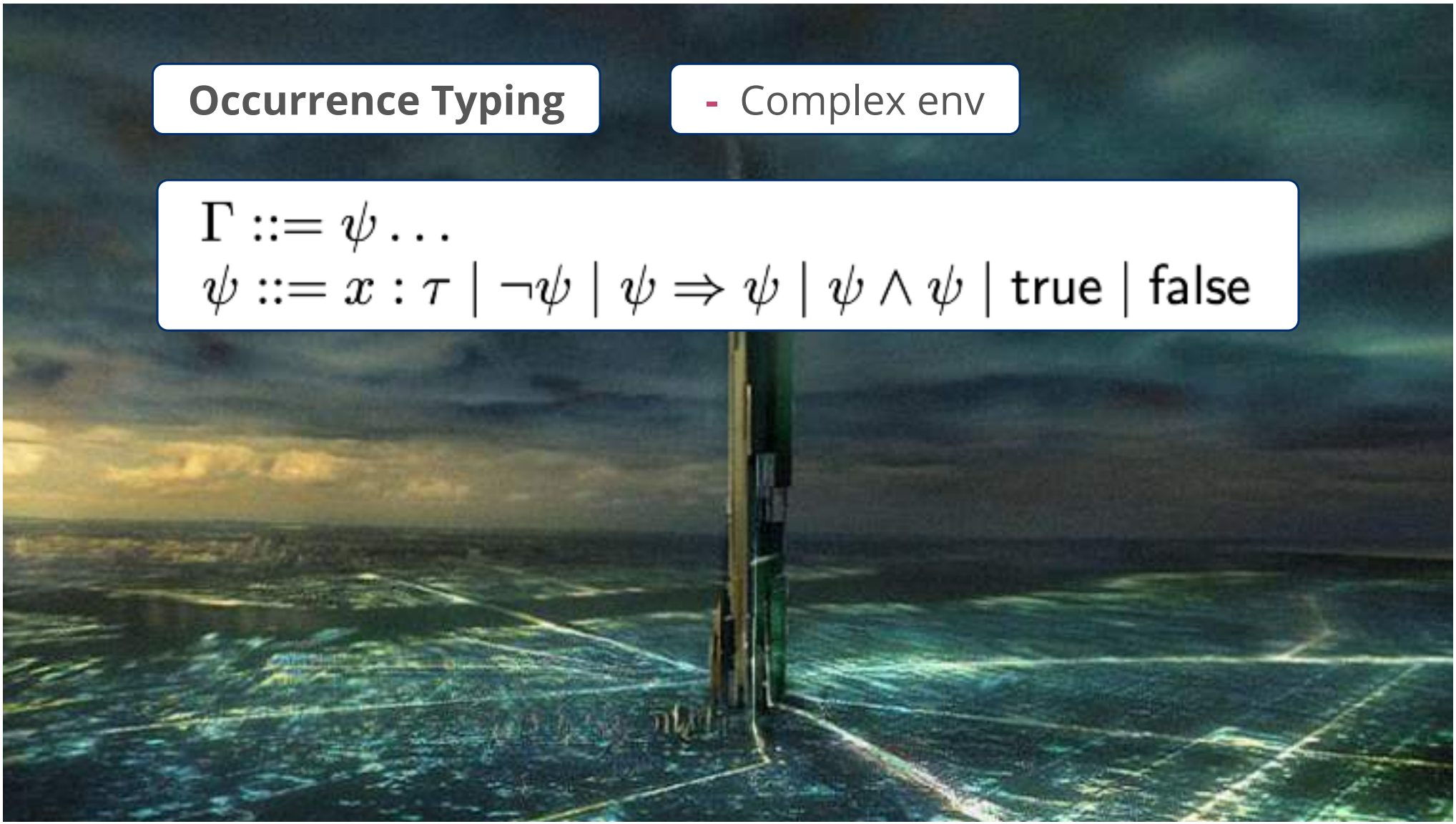
Occurrence Typing

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

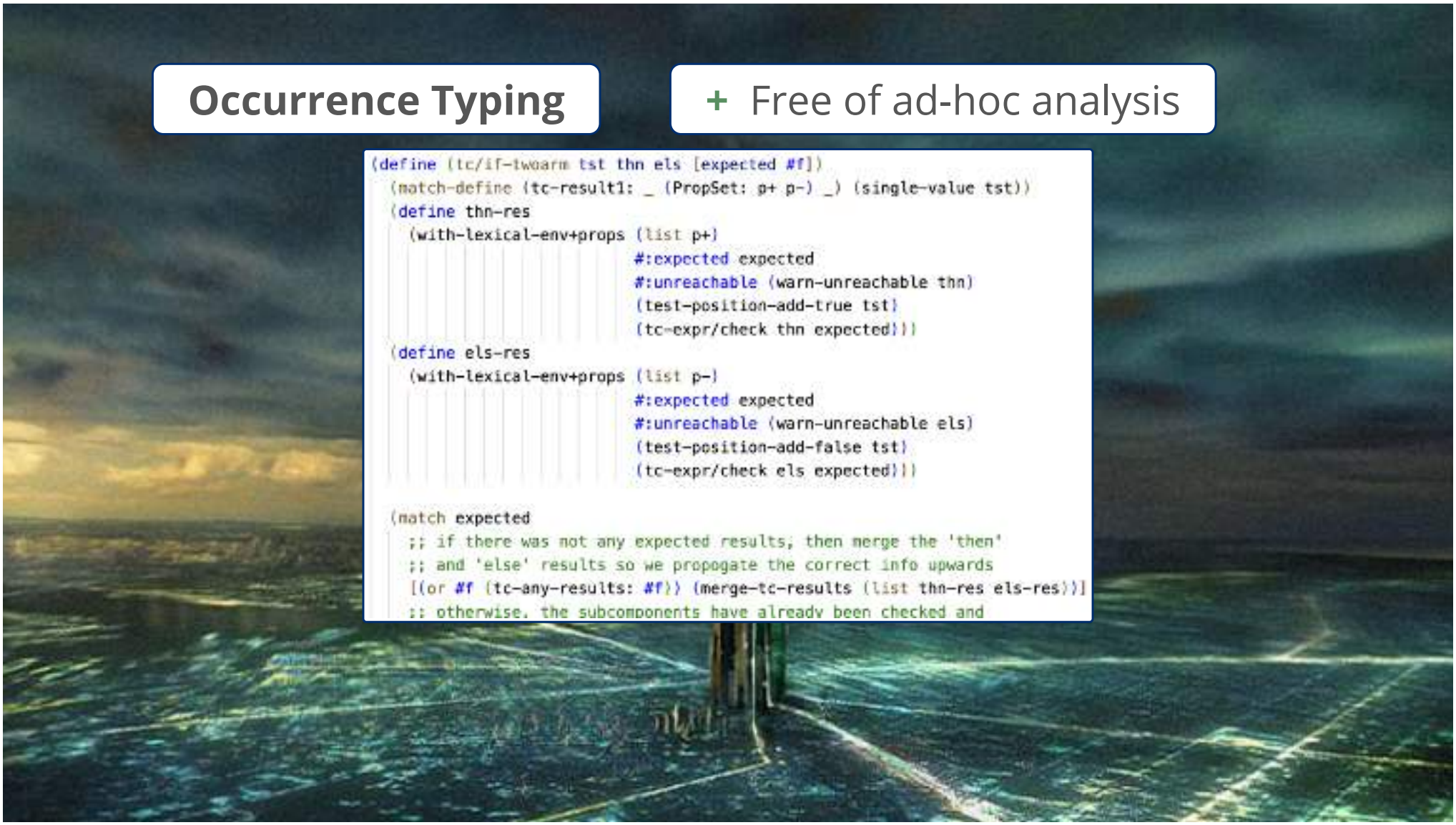
Occurrence Typing

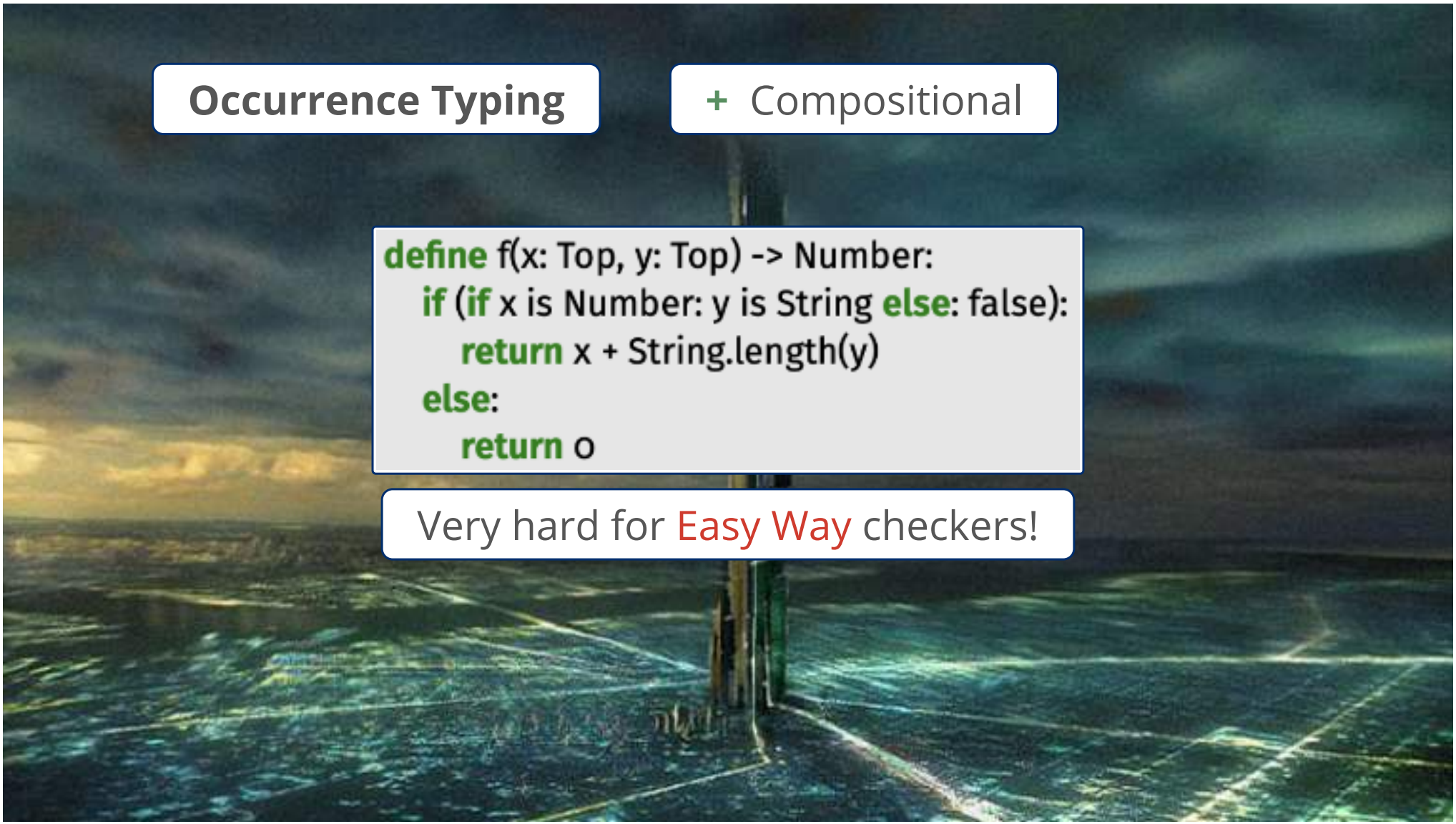
- Complex env

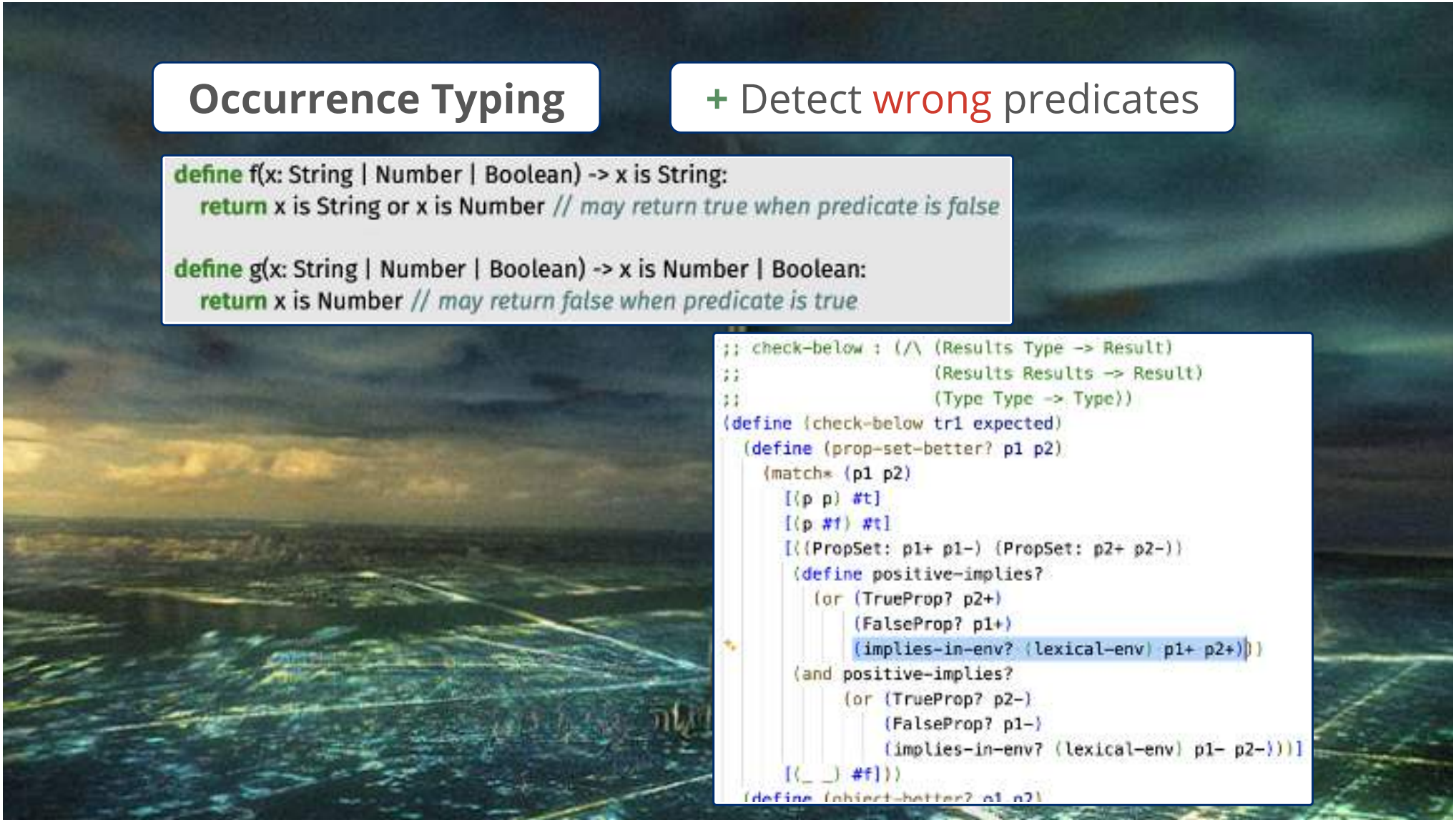
$$\Gamma ::= \psi \ldots$$
$$\psi ::= x : \tau \mid \neg\psi \mid \psi \Rightarrow \psi \mid \psi \wedge \psi \mid \mathsf{true} \mid \mathsf{false}$$

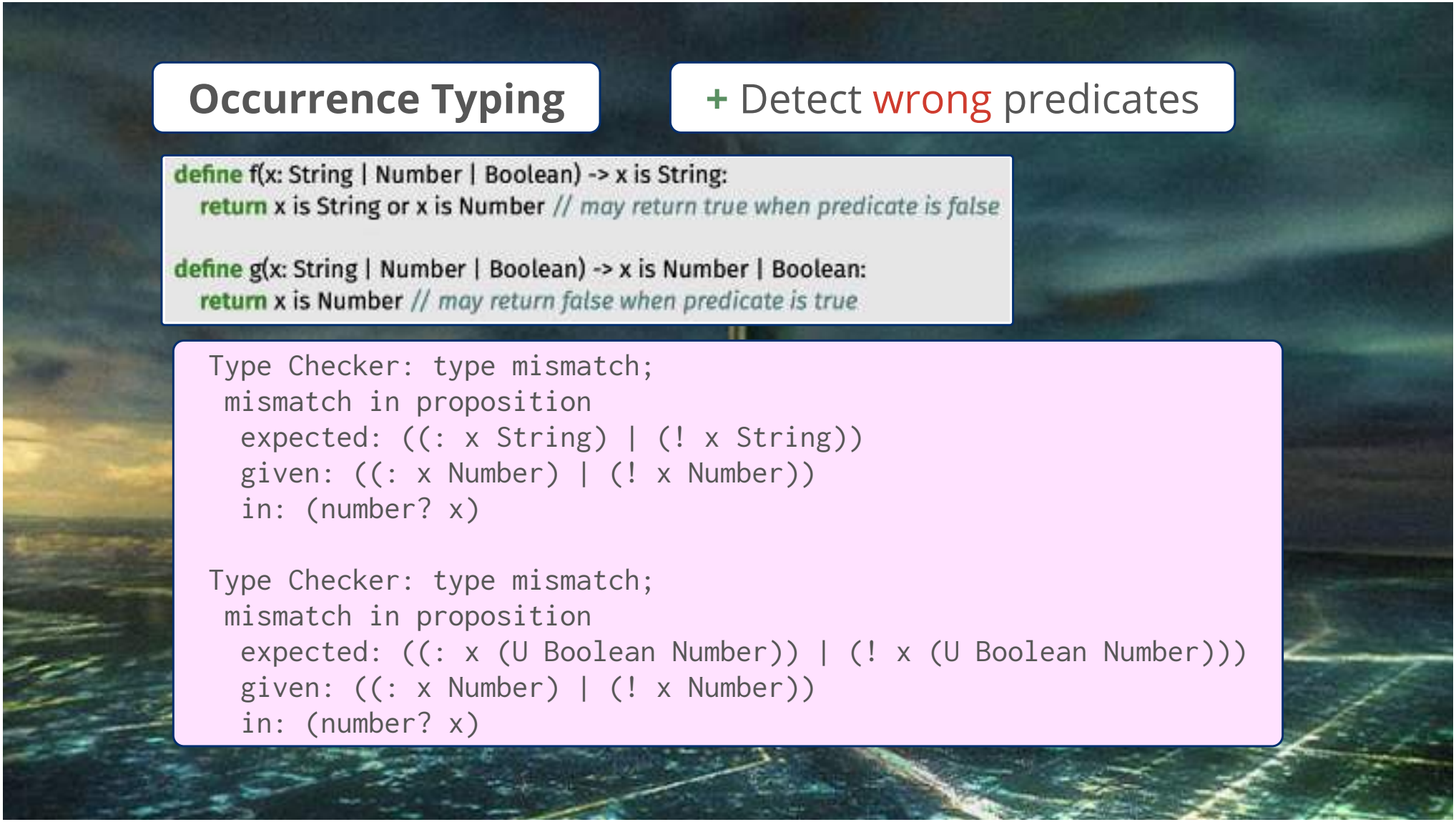
```
;; types is a free-id-table of identifiers to types
;; props is a list of known propositions
(define-struct env ([types immutable-free-id-table?]
                    [idx-types (hash/c Index? Type? #:immutable #t)]
                    [props (listof Prop?)]
                    [aliases immutable-free-id-table?]))
```

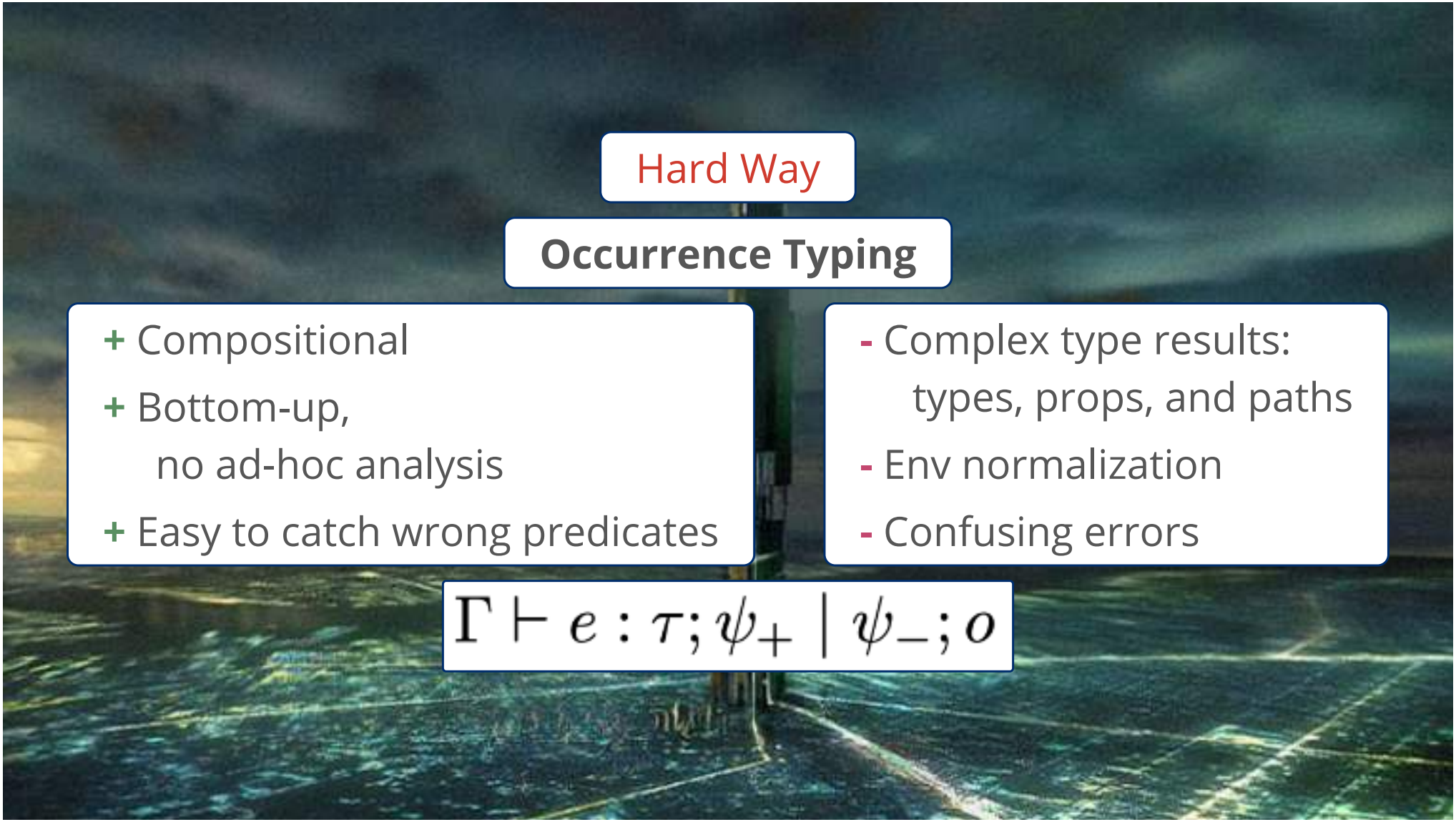
https://github.com/racket/typed-racket/blob/master/typed-racket-lib/typed-racket/env/type-env-structs.rkt

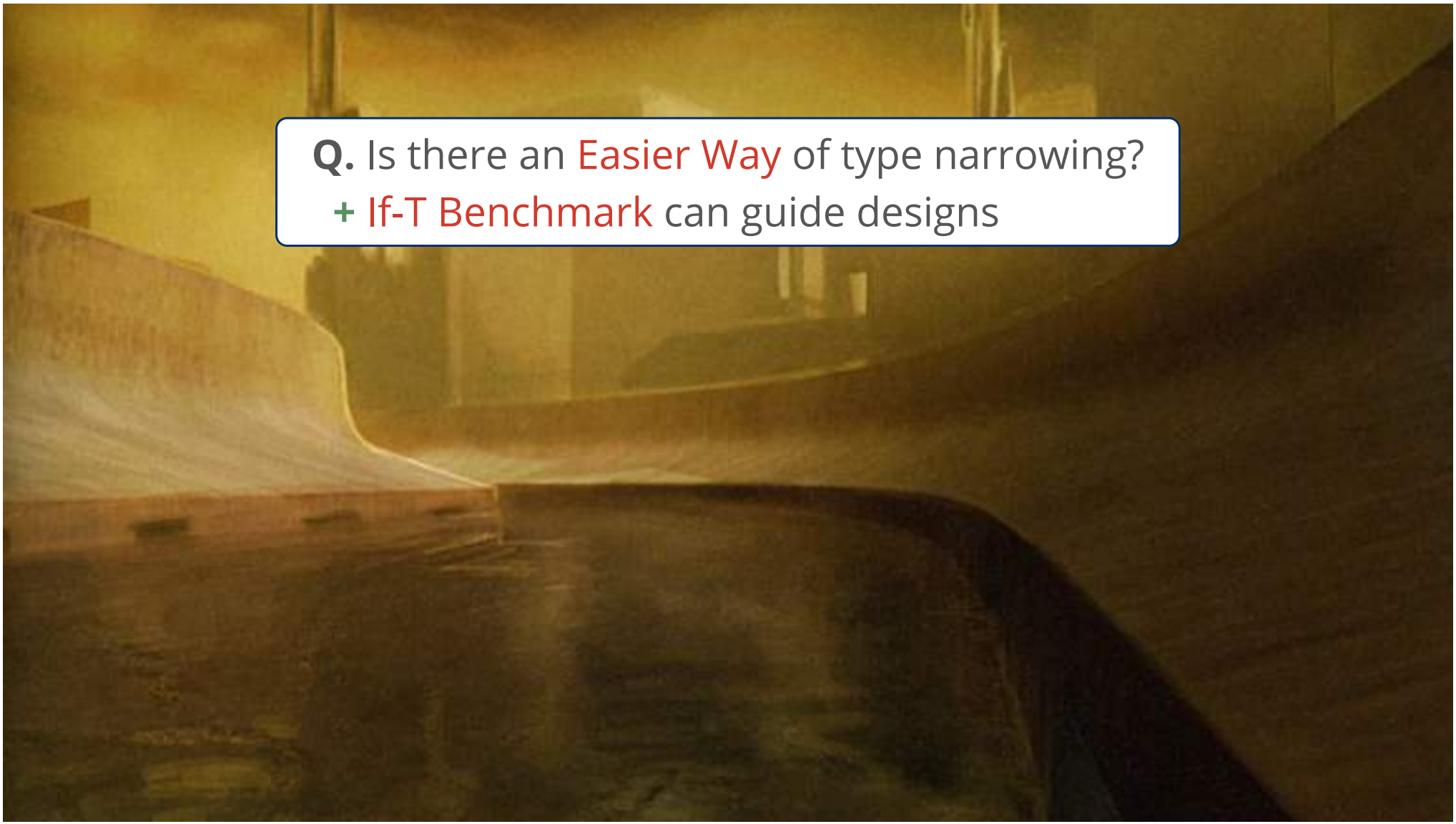

Occurrence Typing
+ Free of ad-hoc analysis
(define (tc/if-twoarm tst thn els [expected #f])
(match-define (tc-result1: _ (PropSet: p+ p-) _) (single-value tst))
(define thn-res
(with-lexical-env+props (list p+)
#:expected expected
#:unreachable (warn-unreachable thn)
(test-position-add-true tst)
(tc-expr/check 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

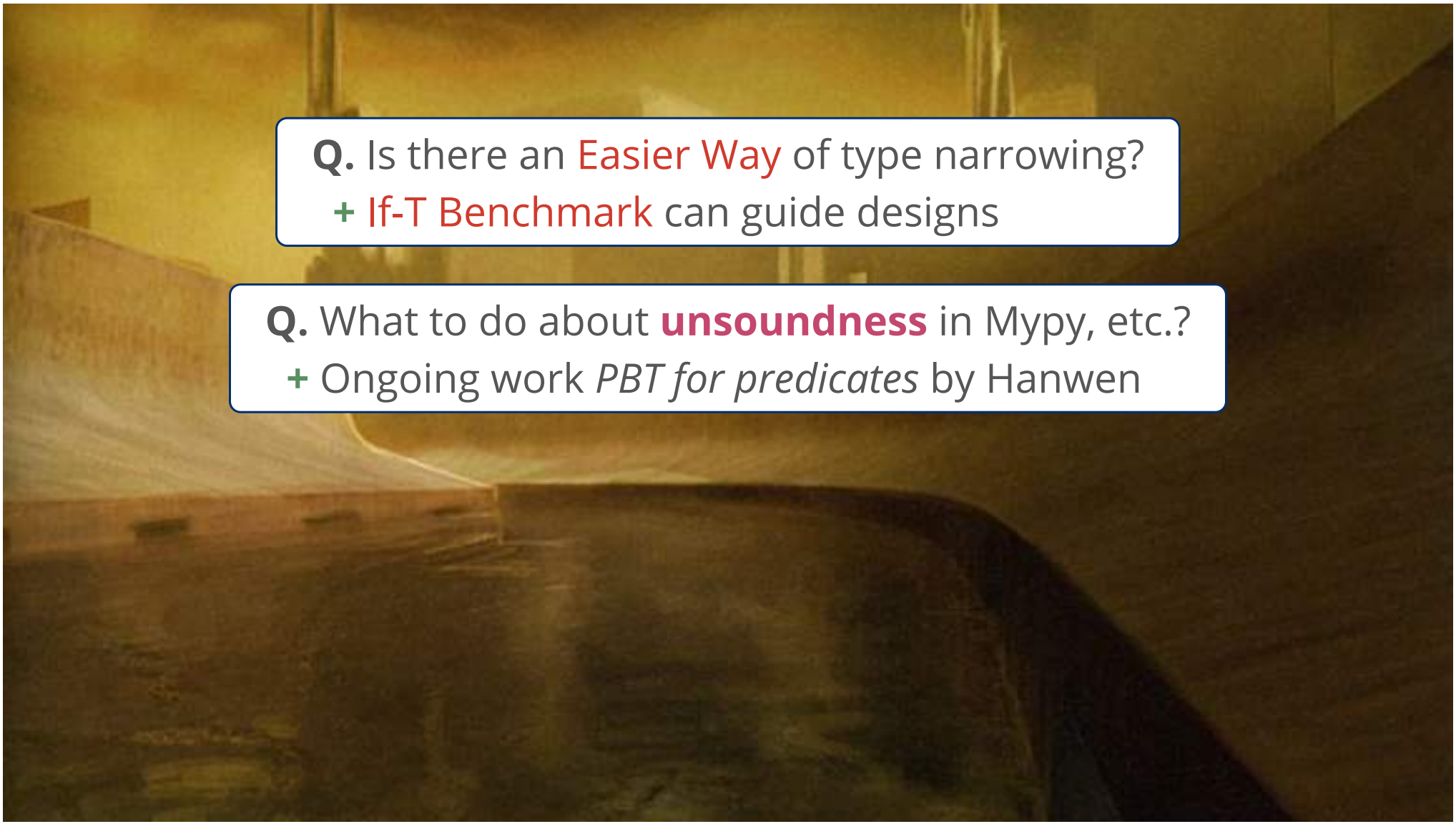

Occurrence Typing
+ Compositional
define f(x: Top, y: Top) -> Number:
if (if x is Number: y is String else: false):
return x + String.length(y)
else:
return 0
Very hard for Easy Way checkers!

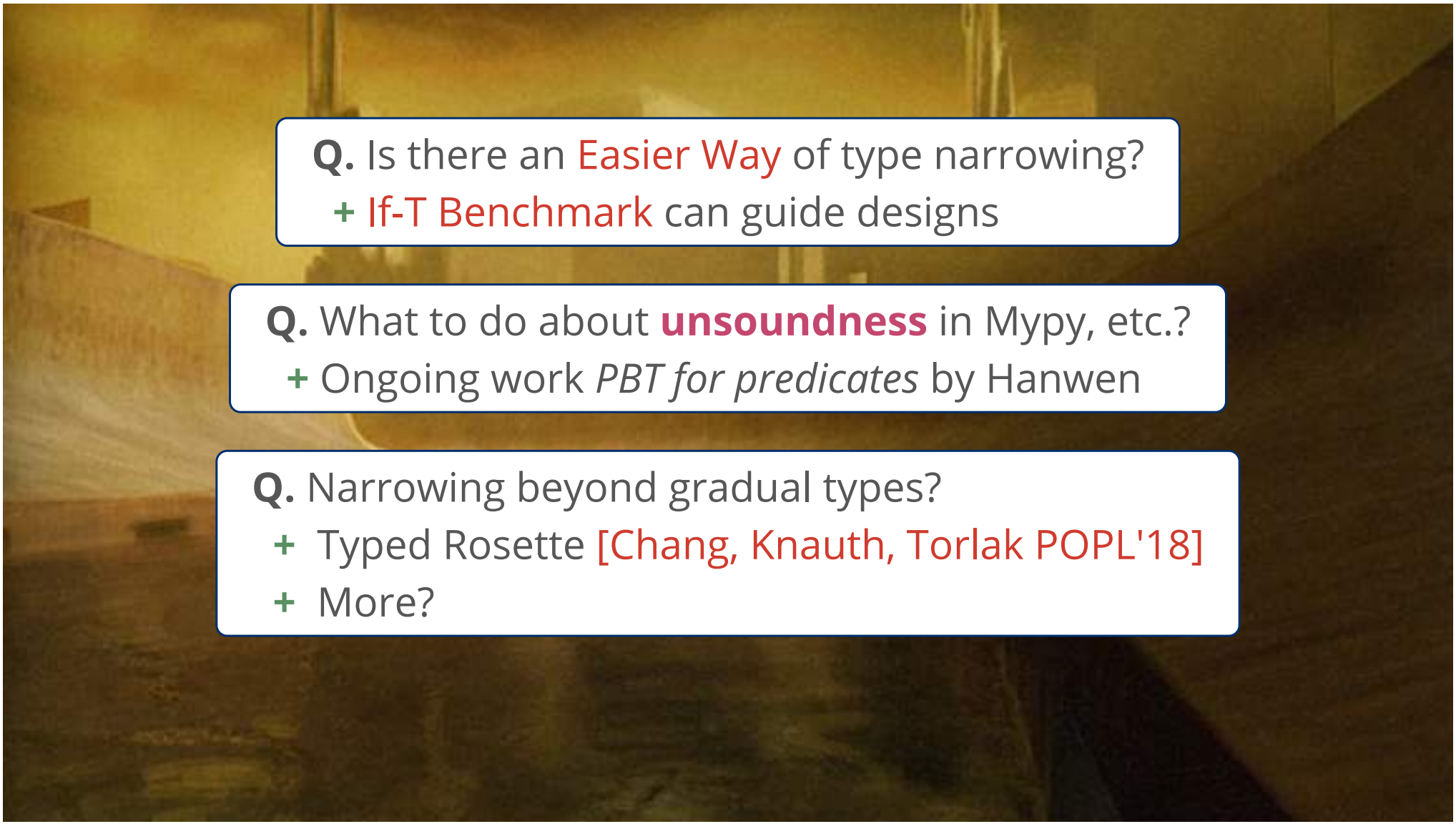

Occurrence Typing
+ Detect wrong predicates
define f(x: String | Number | Boolean) -> x is String:
return x is String or x is Number // may return true when predicate is false
define g(x: String | Number | Boolean) -> x is Number | Boolean:
return x is Number // may return false when predicate is true
;; 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. Is there an Easier Way of type narrowing?
+ If-T Benchmark can guide designs


Q. Is there an Easier Way of type narrowing?
+ If-T Benchmark can guide designs
Q. What to do about **unsoundness** in Mypy, etc.?
+ Ongoing work *PBT for predicates* by Hanwen


Q. Is there an Easier Way of type narrowing?
+ If-T Benchmark can guide designs
Q. What to do about **unsoundness** in Mypy, etc.?
+ Ongoing work *PBT for predicates* by Hanwen
Q. Narrowing beyond gradual types?
+ Typed Rosette [Chang, Knauth, Torlak POPL'18]
+ More?