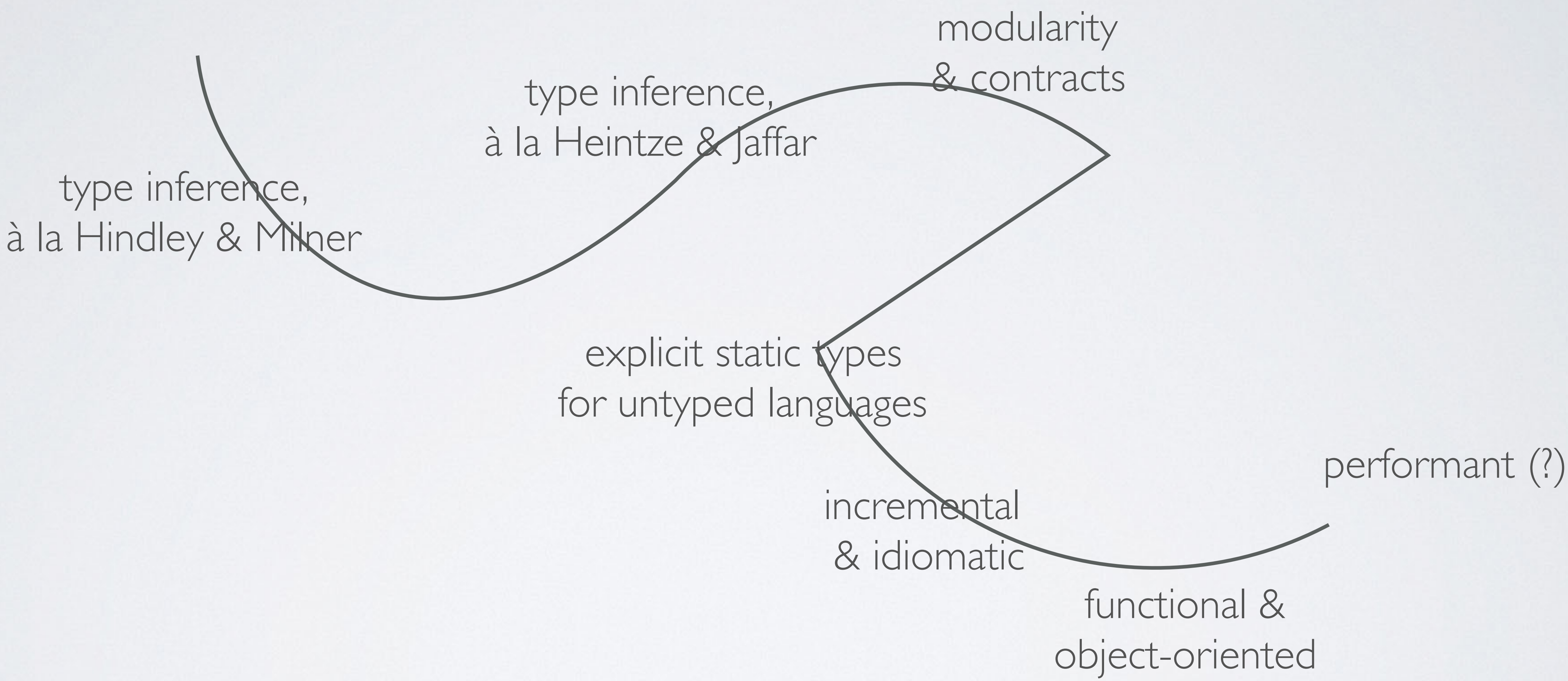
28 YEARS OF TYPES FOR UNTYPED LANGUAGES

Matthias Felleisen, PLT & NUPRL

A Personal Walk through Type Land

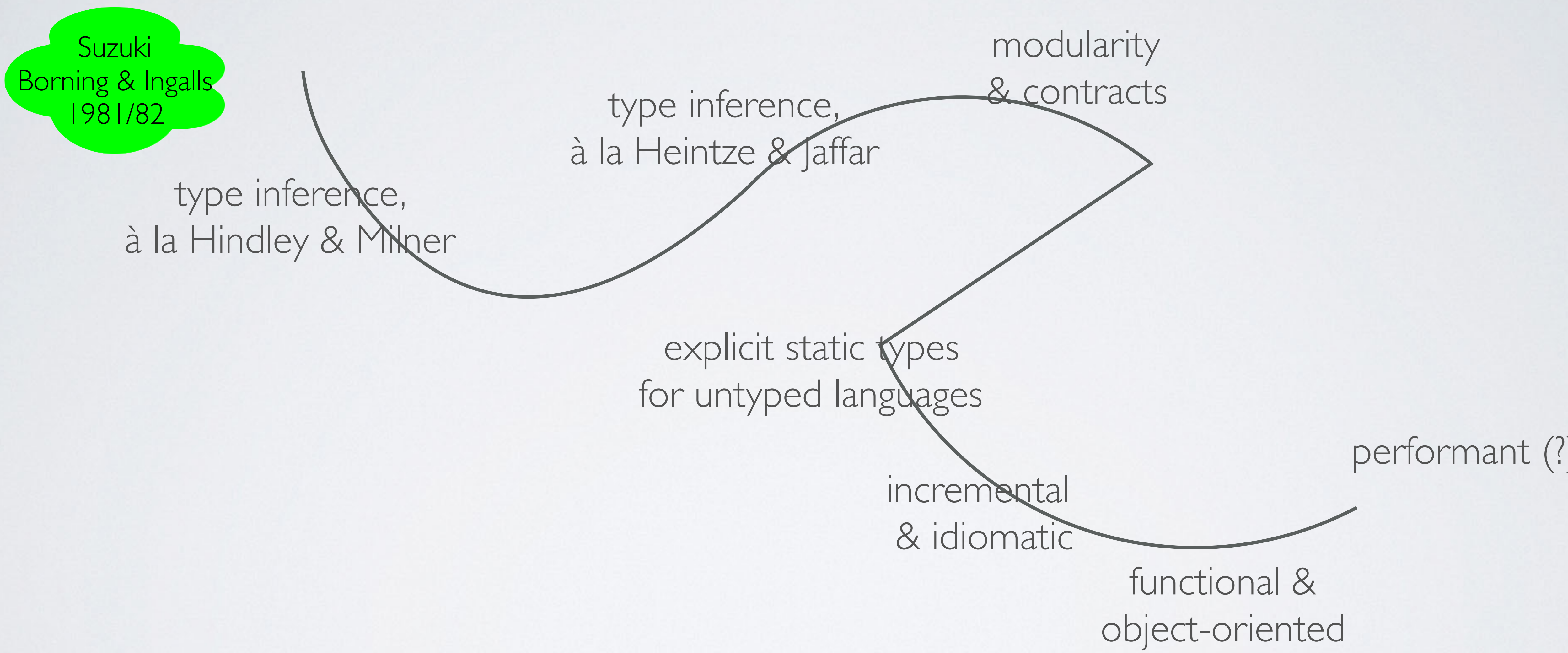
I am an untyped
academic (1987)



I *still* am an untyped
academic (2016).

A Personal Walk through Type Land

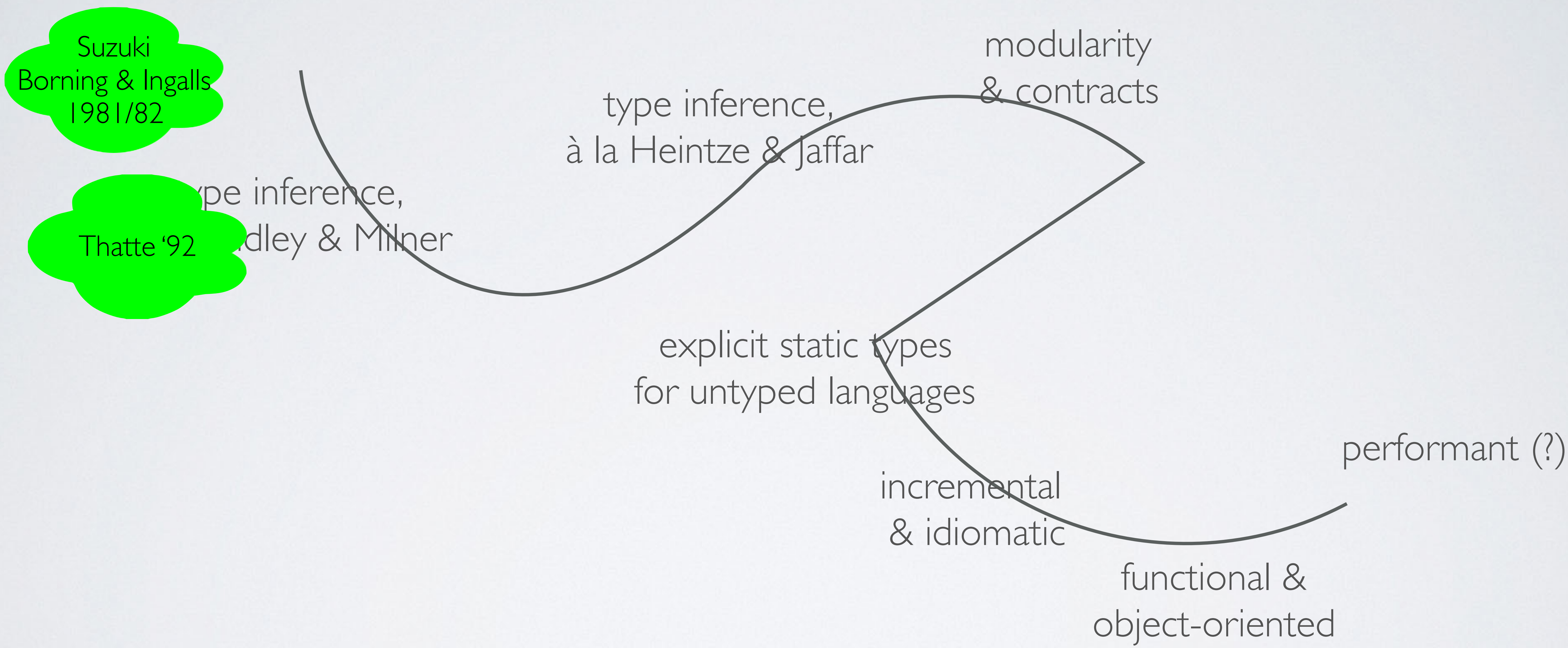
I am an untyped
academic (1987)



I *still* am an untyped
academic (2016).

A Personal Walk through Type Land

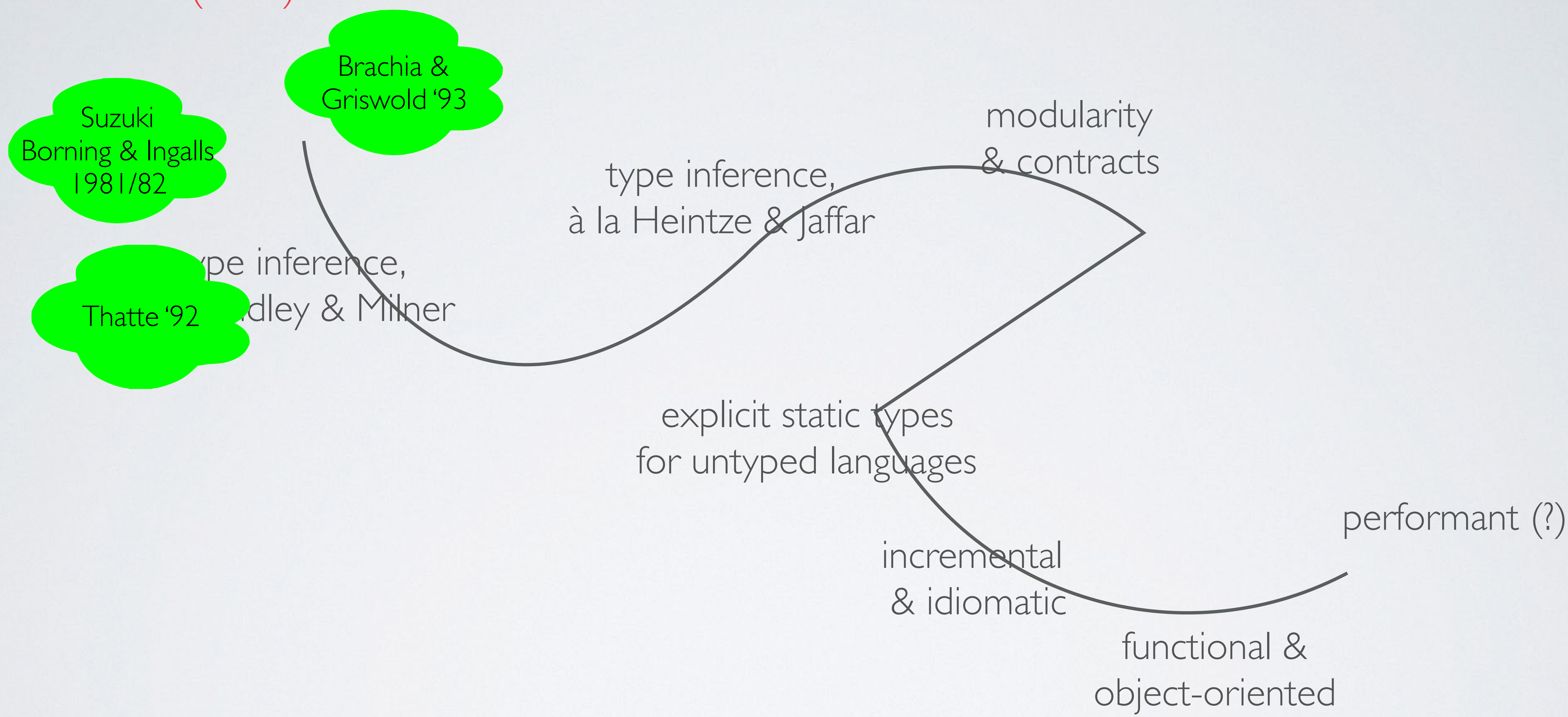
I am an untyped
academic (1987)



I *still* am an untyped
academic (2016).

A Personal Walk through Type Land

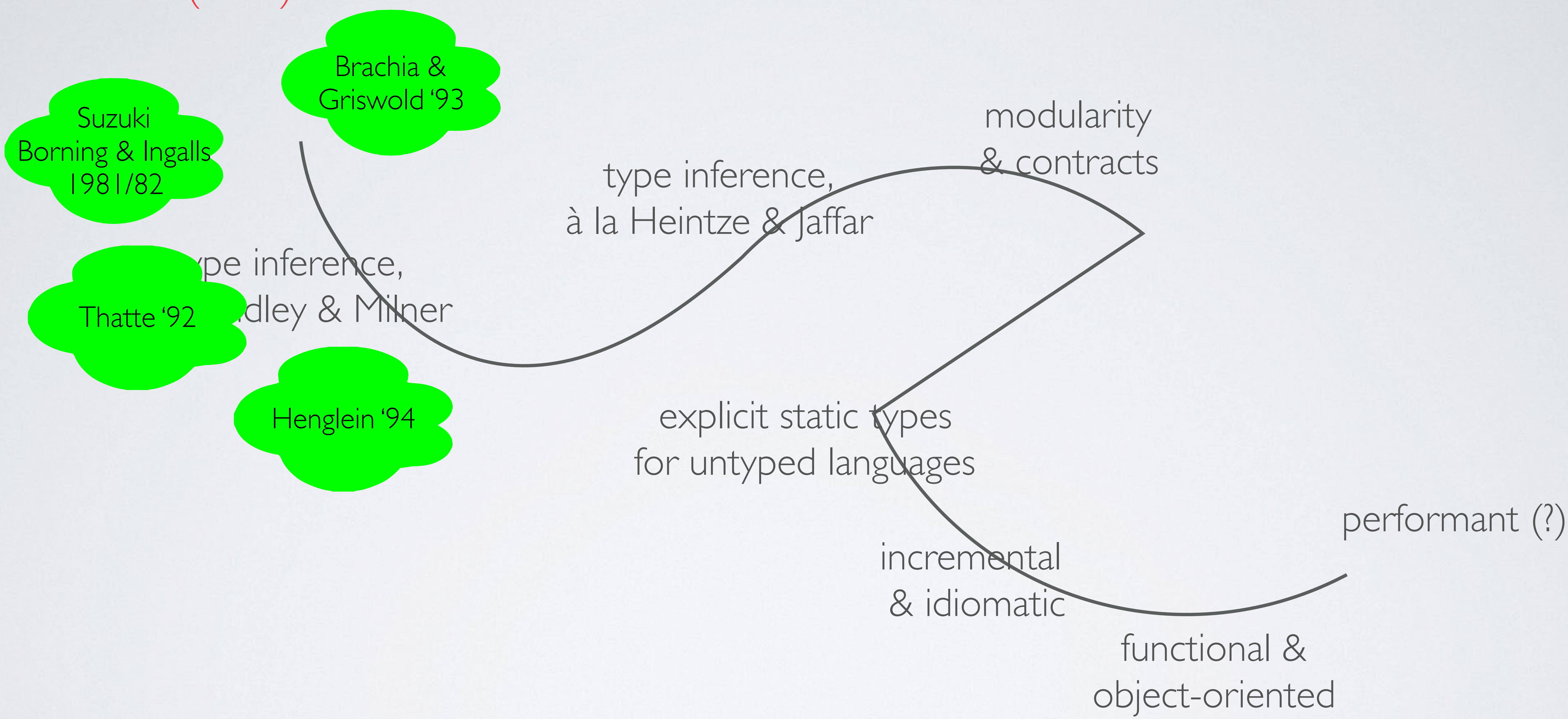
I am an untyped academic (1987)



I *still* am an untyped academic (2016).

A Personal Walk through Type Land

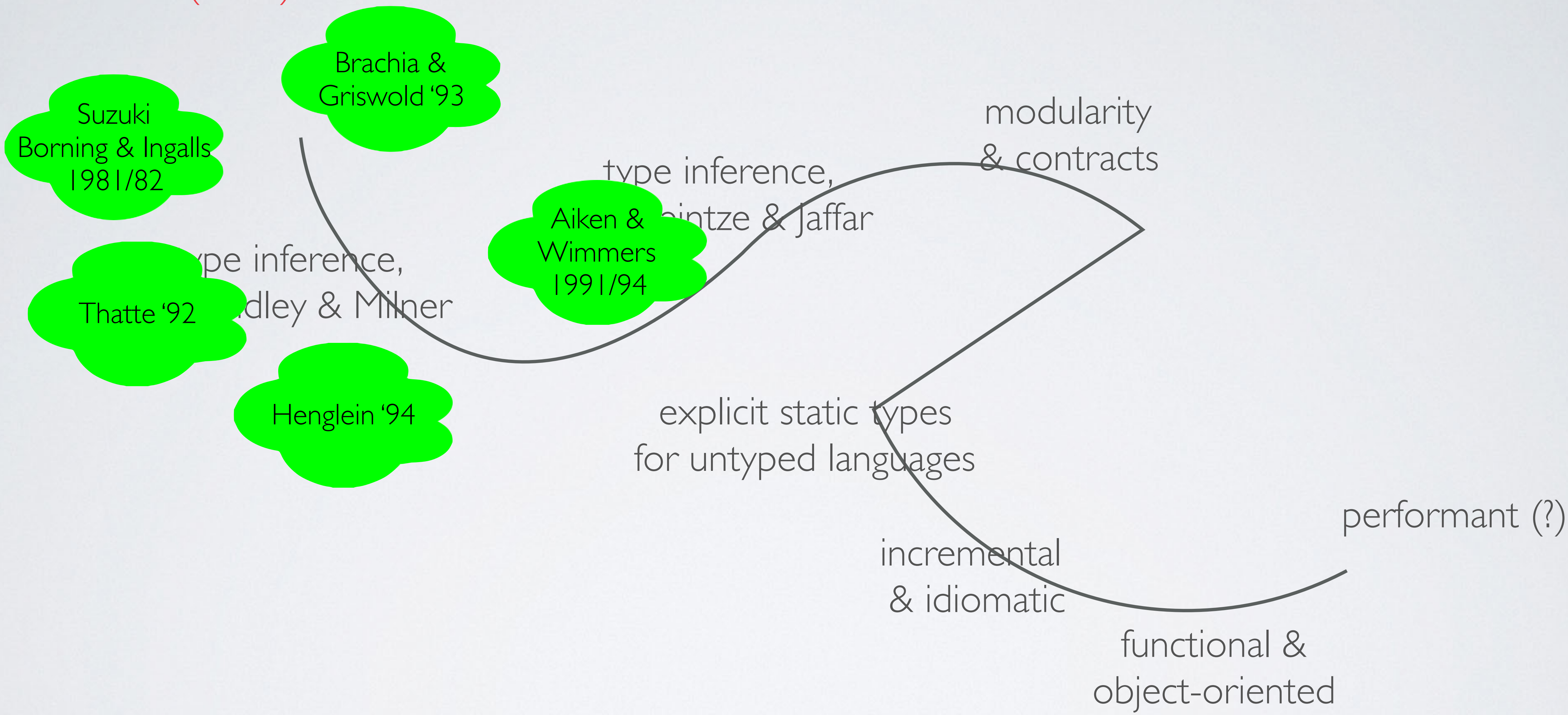
I am an untyped academic (1987)



I *still* am an untyped academic (2016).

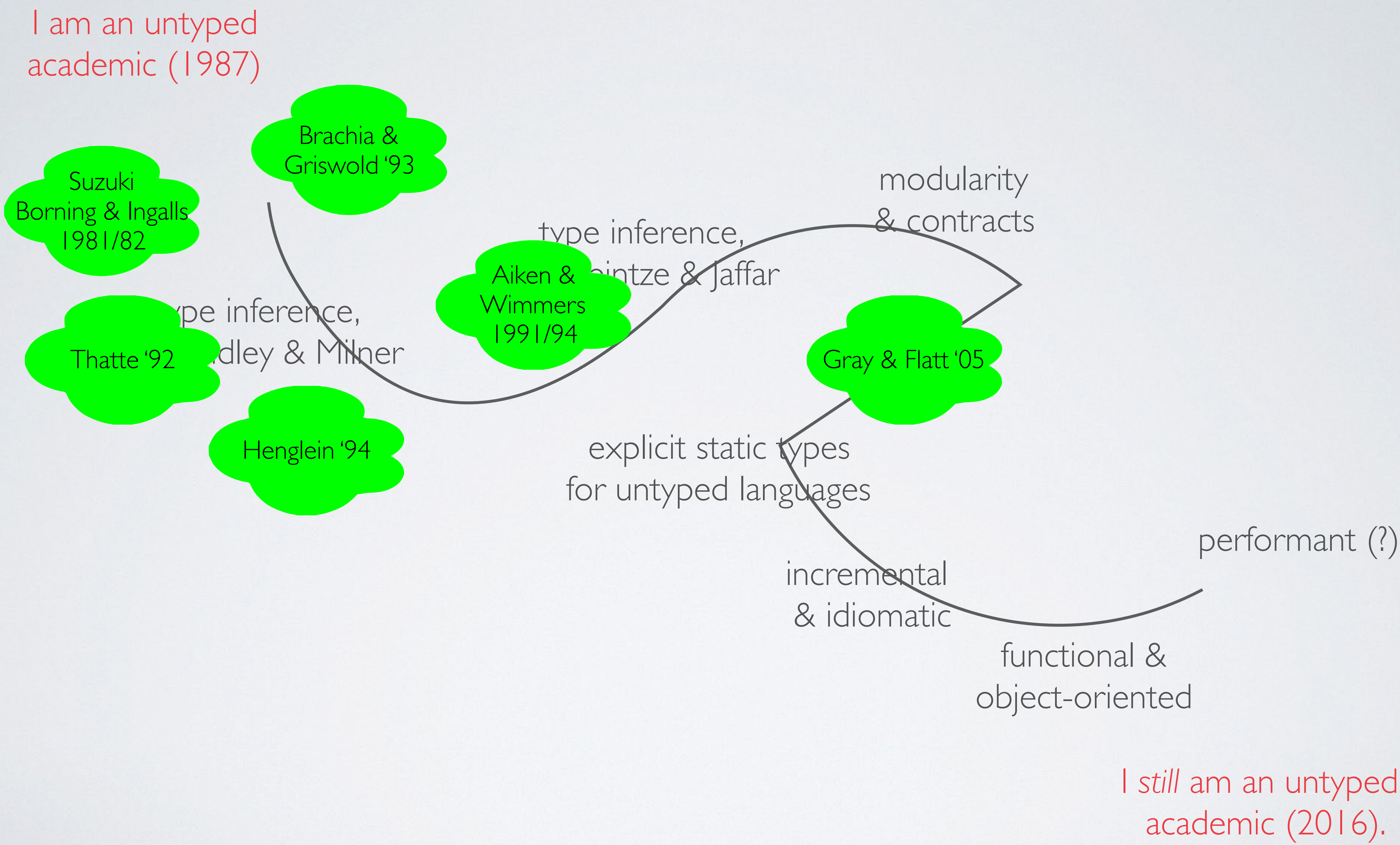
A Personal Walk through Type Land

I am an untyped academic (1987)



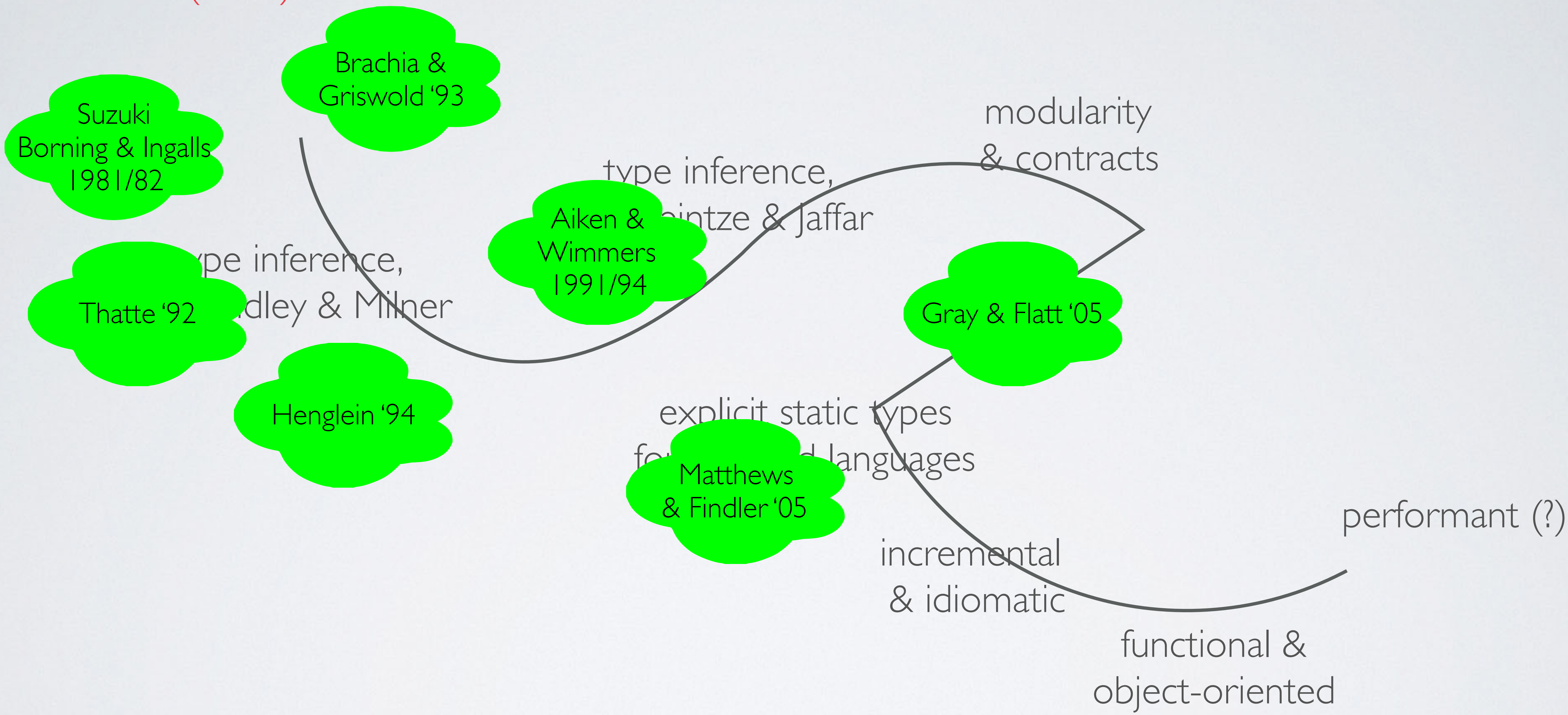
I *still* am an untyped academic (2016).

A Personal Walk through Type Land



A Personal Walk through Type Land

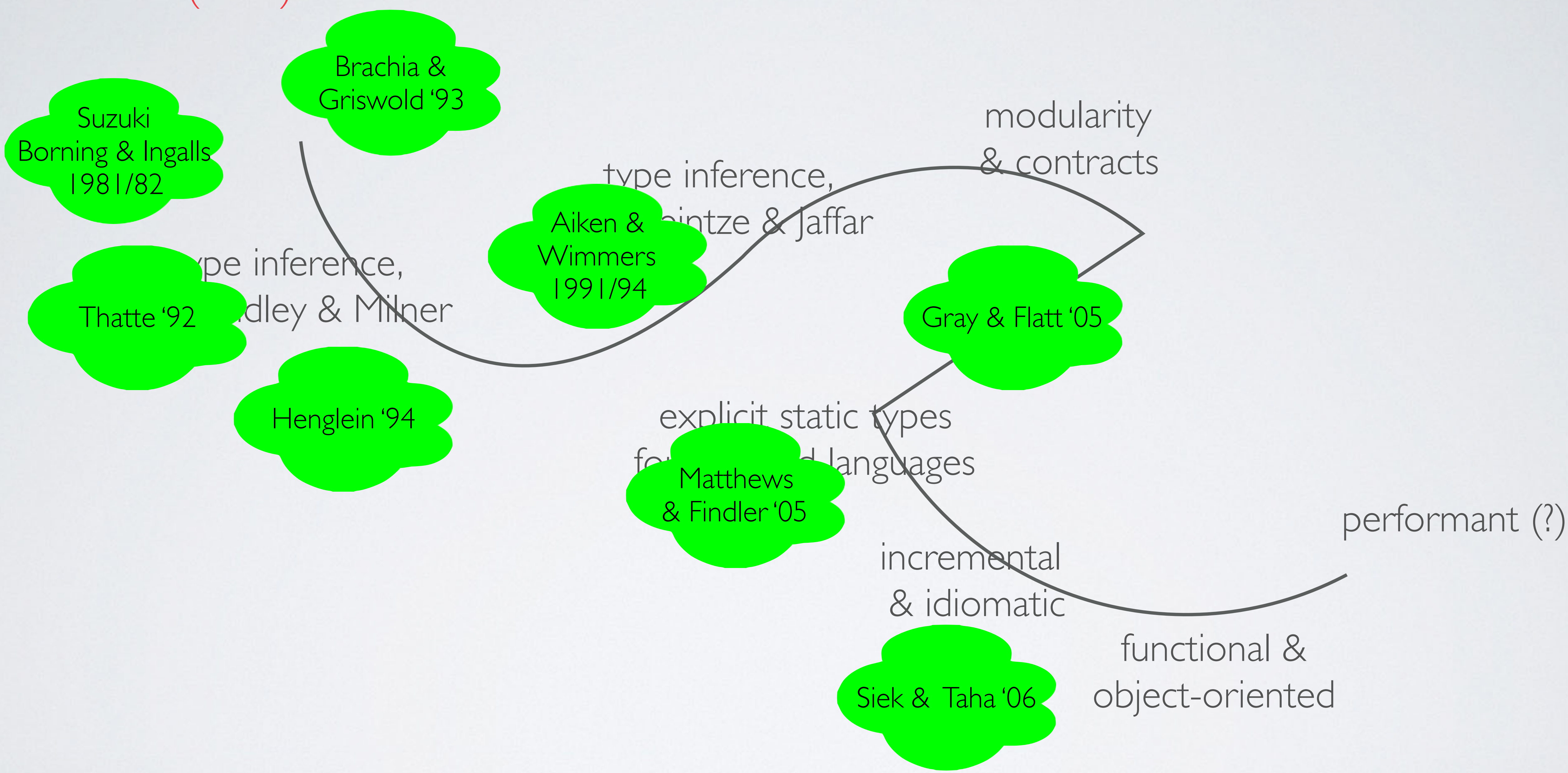
I am an untyped
academic (1987)



I *still* am an untyped
academic (2016).

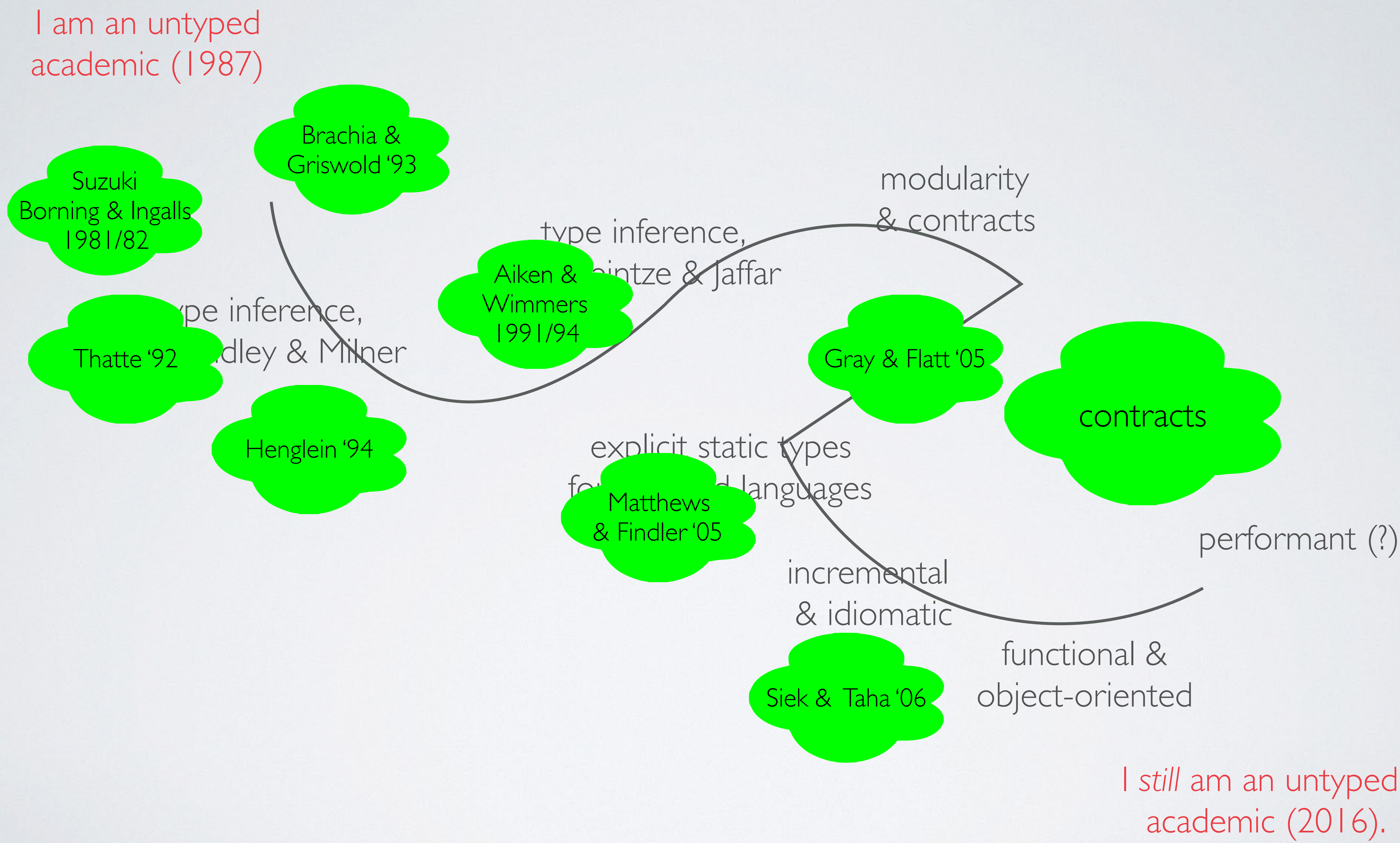
A Personal Walk through Type Land

I am an untyped
academic (1987)



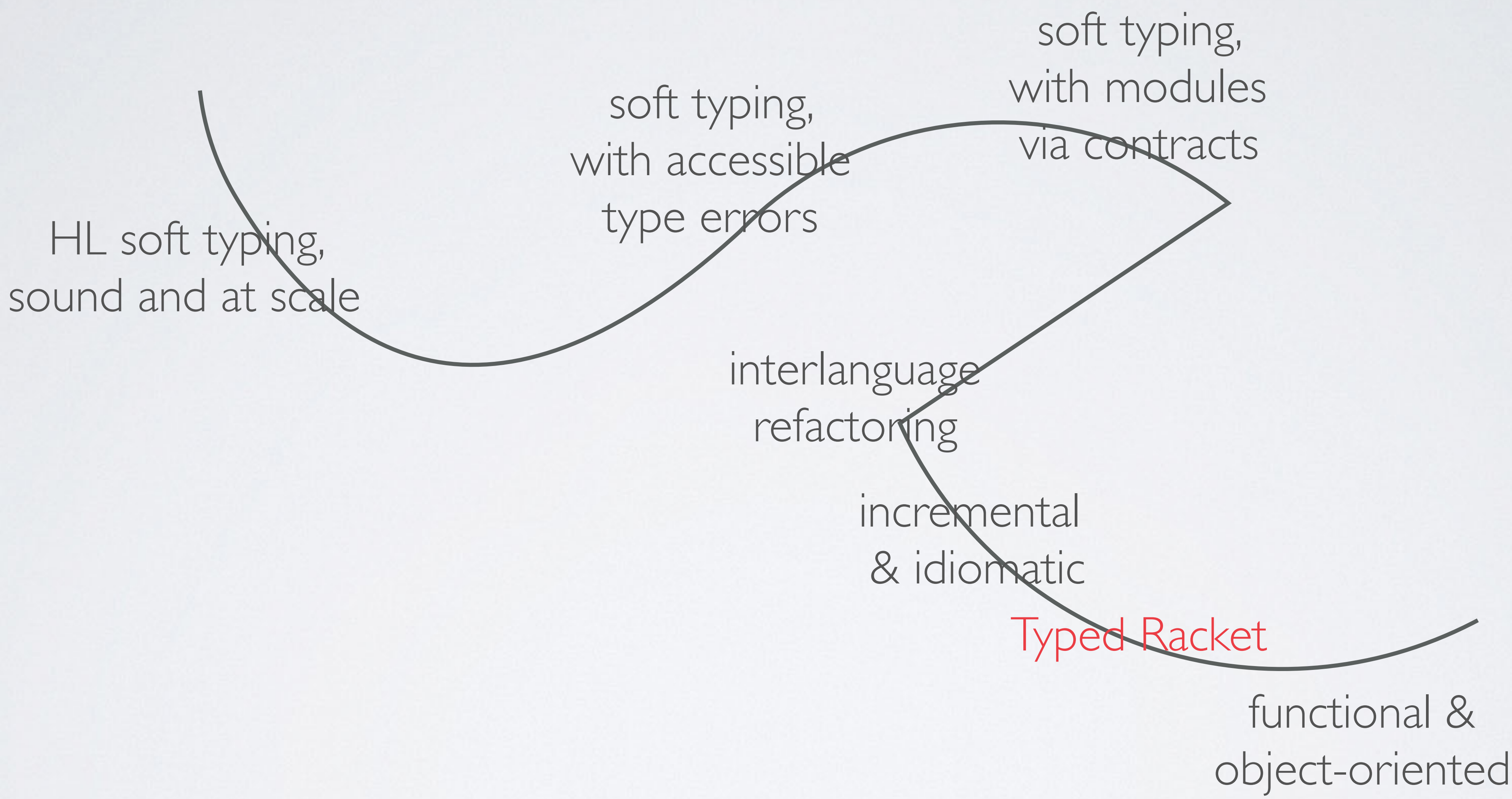
I *still* am an untyped
academic (2016).

A Personal Walk through Type Land



A Personal Walk through Type Land

I am an untyped academic (1987)



I *still* am an untyped academic (2016).

A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Andrew Wright

soft typing, with accessible type errors

soft typing, with modules via contracts

interlanguage refactoring

incremental & idiomatic

Typed Racket

functional & object-oriented

I *still* am an untyped academic (2016).

A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

Cormac Flanagan

Andrew Wright

HL soft typing, sound and at scale

soft typing, with accessible type errors

soft typing, with modules via contracts

interlanguage refactoring

incremental & idiomatic

Typed Racket

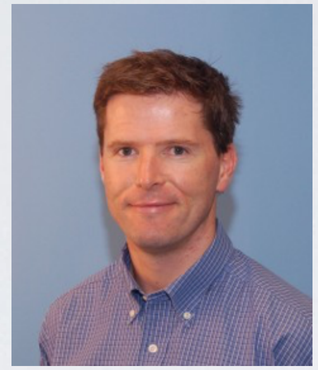
functional & object-oriented

I *still* am an untyped academic (2016).

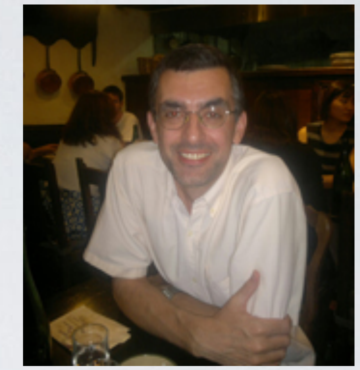
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan



Cormac Flanagan



Philippe Meunier

HL soft typing, sound and at scale

soft typing, with accessible type errors

soft typing, with modules via contracts

Andrew Wright

interlanguage refactoring

incremental & idiomatic

Typed Racket

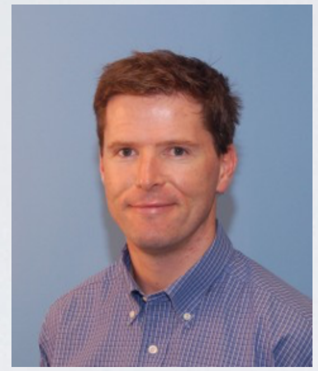
functional & object-oriented

I *still* am an untyped academic (2016).

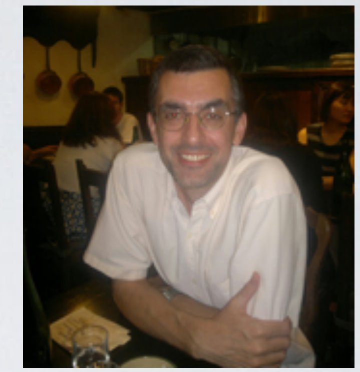
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan



Cormac Flanagan



Philippe Meunier

HL soft typing, sound and at scale

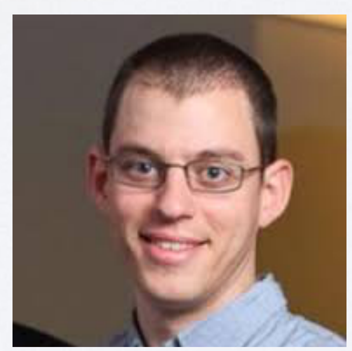
soft typing, with accessible type errors

soft typing, with modules via contracts

interlanguage refactoring

incremental & idiomatic

Andrew Wright



Sam Tobin-Hochstadt

Typed Racket

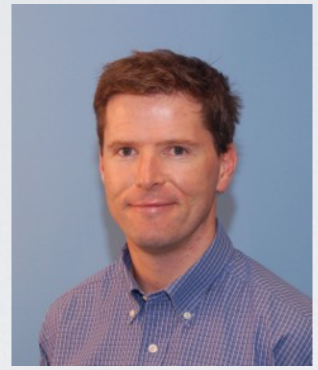
functional & object-oriented

I *still* am an untyped academic (2016).

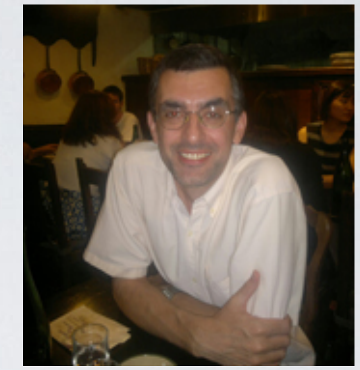
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan



Cormac Flanagan



Philippe Meunier

HL soft typing, sound and at scale

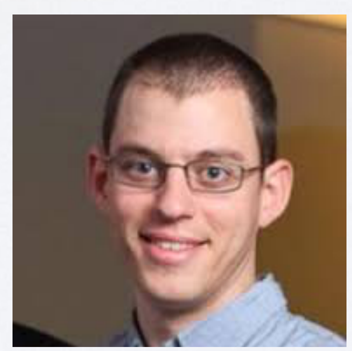
soft typing, with accessible type errors

soft typing, with modules via contracts

interlanguage refactoring

Andrew Wright

incremental & idiomatic



Sam Tobin-Hochstadt

Typed Racket

functional & t-oriented

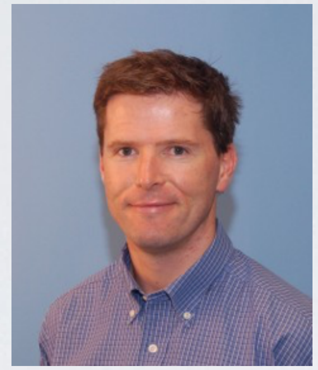
Asumu Takikawa

I *still* am an untyped academic (2016).

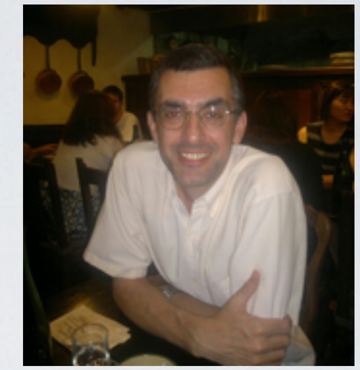
A Personal Walk through Type Land

I am an untyped
academic (1987)

Mike Fagan



Cormac Flanagan



Philippe Meunier

HL soft typing,
sound and at scale

soft typing,
with accessible
type errors

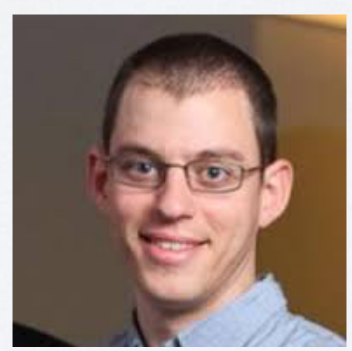
soft typing,
with modules
via contracts

interlanguage
refactoring

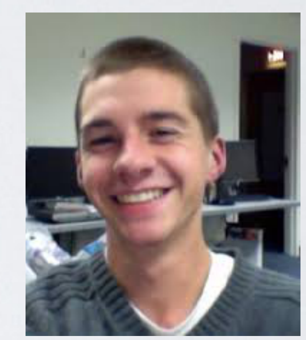
incremental
& idiomatic

Typed Racket

Andrew Wright



Sam Tobin-Hochstadt



Ben Greenman

functional &
t-oriented

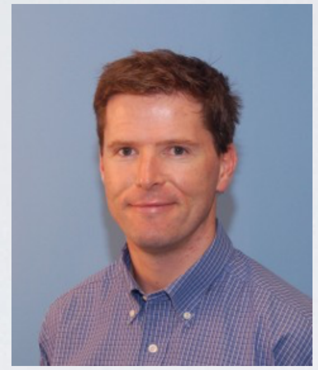
Asumu Takikawa

I *still* am an untyped
academic (2016).

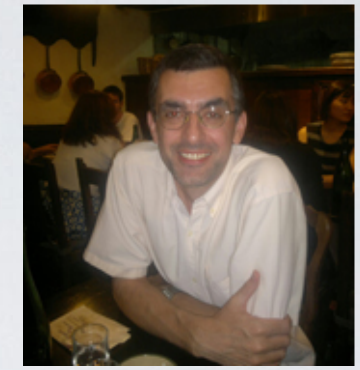
A Personal Walk through Type Land

I am an untyped
academic (1987)

Mike Fagan



Cormac Flanagan

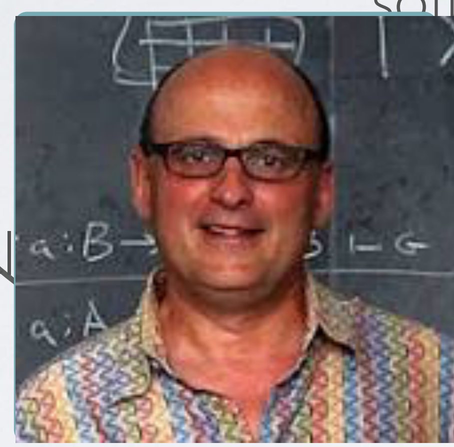


Philippe Meunier

soft typing,
accessible
errors

soft typing,
with modules
via contracts

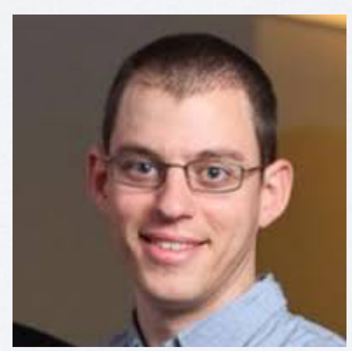
HL soft typing,
sound and at scal



interlanguage
refactoring

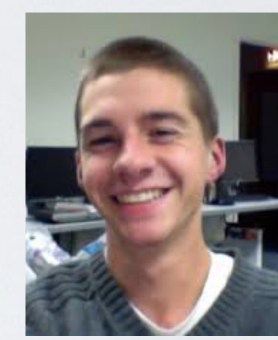
Andrew Wright

incremental
& idiomatic



Sam Tobin-Hochstadt

Typed Racket



Ben Greenman

functional &
t-oriented

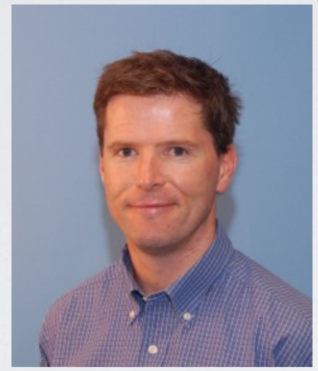
Asumu Takikawa

I *still* am an untyped
academic (2016).

A Personal Walk through Type Land

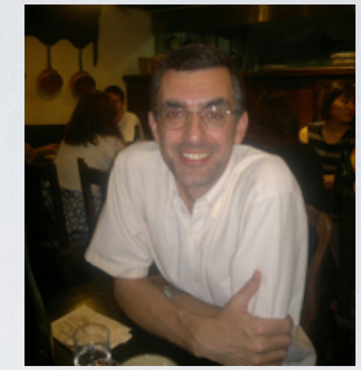
I am an untyped academic (1987)

Mike Fagan



Cormac Flanagan

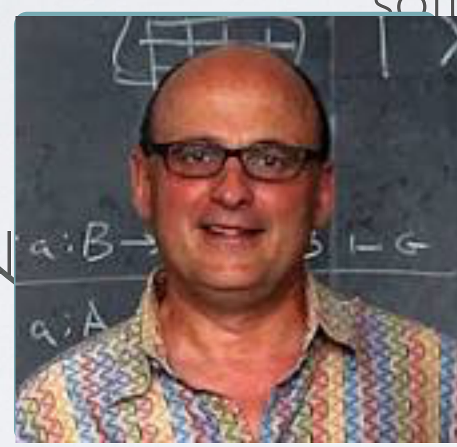
soft typing,
accessible
errors



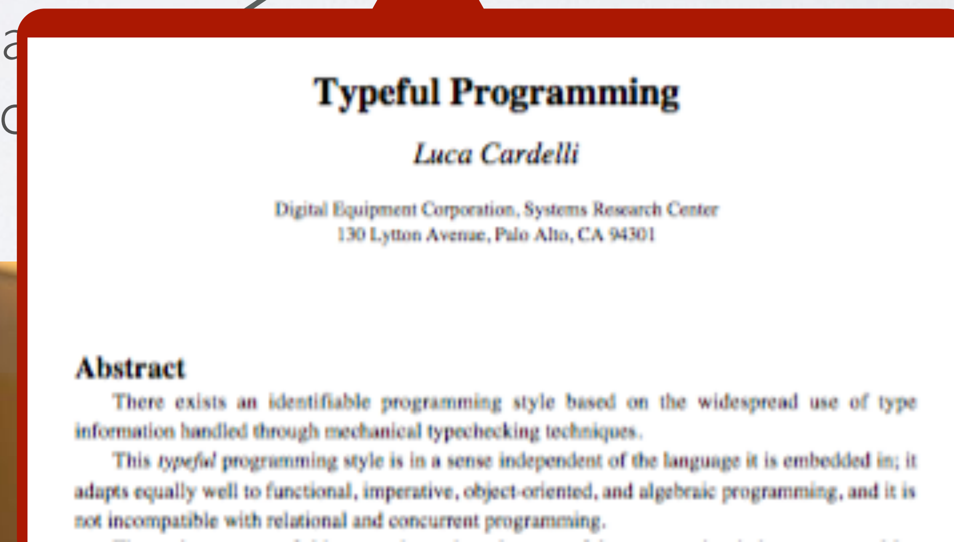
soft typing,
with modules
via contracts

Philippe Meunier

HL soft typing,
sound and at scal



interla
refac



Typeful Programming

Luca Cardelli

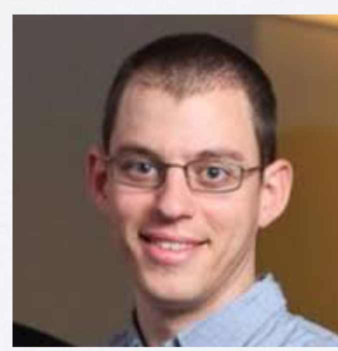
Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Abstract

There exists an identifiable programming style based on the widespread use of type information handled through mechanical typechecking techniques.

This *typeful* programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming.

Andrew Wright



Sam Tobin-Hochstadt

n Greenman

t-oriented

Asumu Takikawa

I *still* am an untyped academic (2016).

Robert “Corky” Cartwright

User-Defined Data Types as an
Aid to Verifying LISP Programs
ICALP 1976, pp. 228–256.

Robert "Corky" Cartwright

User-Defined Data Types as an Aid to Verifying LISP Programs
ICALP 1976, pp. 228–256.

Write functional LISP, instead of imperative Algol:

- write functional programs
- describe them with user-defined types
- use these types to prove theorems

Functional programs are theories of first-order logic.

Robert "Corky" Cartwright

User-Defined Data Types as an Aid to Verifying LISP Programs
ICALP 1976, pp. 228–256.

Write functional LISP, instead of imperative Algol:
- write functional programs
- describe them with user-defined types
- use these types to prove theorems

Functional programs are theories of first-order logic.

When I arrived at Rice in 1987:

"let's add types to Scheme."

What does "adding types to Scheme" mean? Why is it hard?

```
;; Representing Russian dolls and computing their depth
;; RussianDoll = ‘doll u (cons RussianDoll ‘())


;; RussianDoll -> Natural
(define (depth r)
  (cond
    [(symbol? r) 0]
    [else (+ 1 (depth (first r)))]))


(depth ‘doll) ;; —> 0
(depth ‘(((doll))) ;; —> 3
```

What does "adding types to Scheme" mean? Why is it hard?

```
;; Representing propositions and checking tautology

;; Proposition = Boolean u [Boolean -> Proposition]


;; Proposition -> Boolean

(define (tautology? p)

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

```
type proposition = InL of bool | InR of (bool -> proposition)

let rec is_tautology p =

 match p with

  | InL b -> b

  | InR p -> is_tautology(p true) && is_tautology(p false)


is_tautology (InR(fun x -> InL true))

is_tautology (InR(fun x -> InR(fun y -> or (InL x) (InL y))))
```

My idea: add a universal type to the program and add injections and projections where needed. That's a practical version of Scott's view that *untyped languages are unityped.*

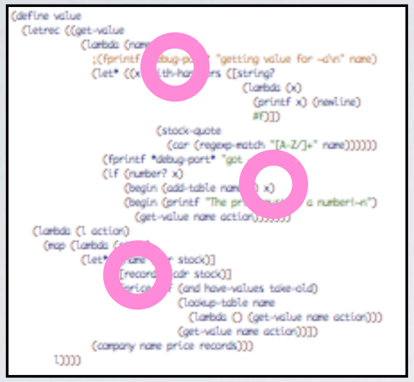

```
type proposition = InL                  -> proposition)

let rec is_tautology p

 match p with

  | InL b -> b

  | InR p -> is_tautol                  ogy(p false)


is_tautology (InR(fun

is_tautology (InR(fun x -> InR(fun y -> or (InL x) (InL y))))
```

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



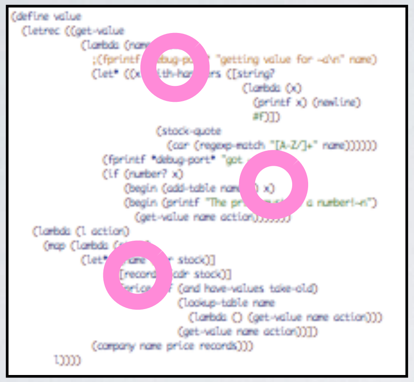
$s = (t \rightarrow \underline{int})$
$t = (v \rightarrow \underline{char})$
$v = \underline{double}$

→

$s = ((double \rightarrow char) \rightarrow \underline{int})$
$t = (double \rightarrow \underline{char})$
$v = \underline{double}$

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



→

think of all missing type declarations as variables, derive system of equations

$s = (t \rightarrow \underline{int})$
$t = (v \rightarrow \underline{char})$
$v = \underline{double}$

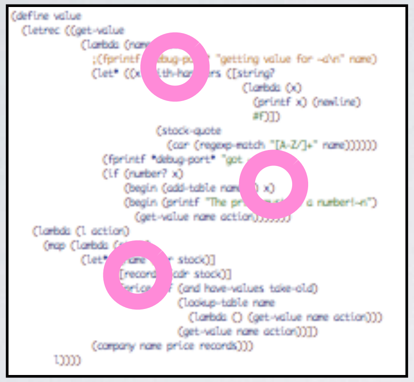
→

unification ~ Gaussian elimination

$s = ((double \rightarrow char) \rightarrow \underline{int})$
$t = (double \rightarrow \underline{char})$
$v = \underline{double}$

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



→

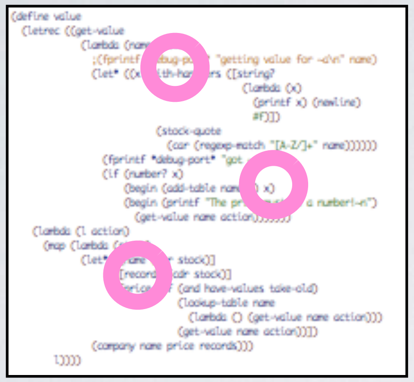
s = (t → int)
t = (v → char)
v = double

→

s = ((double→char)→int)
t = (double→char)
v = double

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



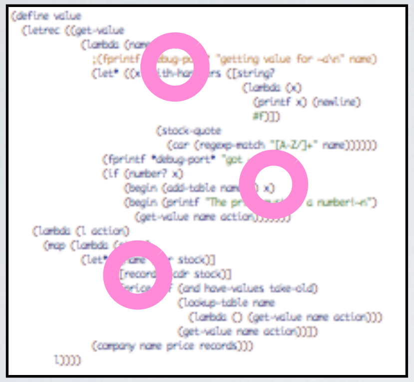
s = (t → <u>int</u>)
t = (v → <u>char</u>)
v = <u>double</u>

s = ((<u>double</u>→<u>char</u>)→<u>int</u>)
t = (<u>double</u>→<u>char</u>)
v = <u>double</u>

s ⊆ { dom : t, rng : int } u { num : 0 }
t ⊆ { dom : v, rng : char, num : 0 }
v ⊆ <u>double</u>

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



$s = (t \rightarrow \underline{int})$
$t = (v \rightarrow \underline{char})$
$v = \underline{double}$

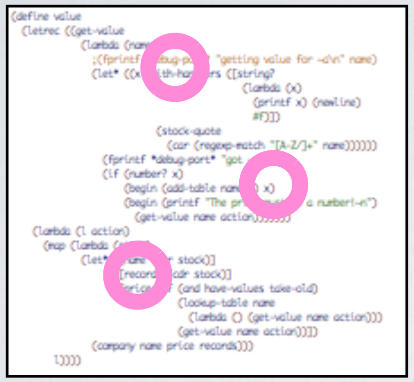
$s = ((double \rightarrow char) \rightarrow \underline{int})$
$t = (double \rightarrow \underline{char})$
$v = \underline{double}$

$s \subseteq \{\ dom : t,\ rng : int\ \}\ u\ \{\ num : 0\ \}$
$t \subseteq \{\ dom : v,\ rng : char,\ num : 0\ \}$
$v \subseteq \underline{double}$

$s = \{\ dom : t,\ rng : int\ \}\ u\ \{\ num : 0\ \}\ u\ \gamma$
$t = \{\ dom : v,\ rng : char,\ num : 0\ \}\ u\ \delta$
$v = \underline{double}\ u\ \varepsilon$

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



s = (t → <u>int</u>)
t = (v → <u>char</u>)
v = <u>double</u>

s = ((double→char)→<u>int</u>)
t = (double→<u>char</u>)
v = <u>double</u>

s ⊆ { dom : t, rng : int } u { num : 0 }
t ⊆ { dom : v, rng : char, num : 0 }
v ⊆ <u>double</u>

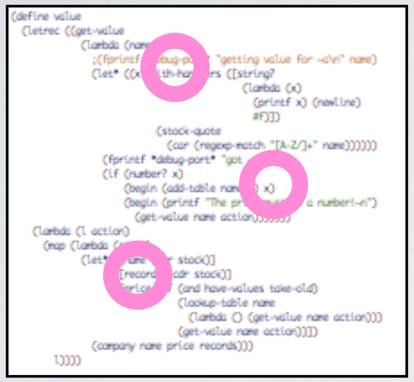
s = { dom : t, rng : int } u { num : 0 } u γ
t = { dom : v, rng : char, num : 0 } u δ
v = <u>double</u> u ε

unification
~
Gaussian elimination

s = … γ = ∅
t = … δ = ∅
v = <u>double</u> ε = ∅

Fagan uses a record type algebra à la Remy [POPL '88] instead of the ->, +, * algebra and then run Hindley Milner.

Mike Fagan



$s = (t \to \underline{\text{int}})$
$t = (v \to \underline{\text{char}})$
$v = \underline{\text{double}}$

$s = ((\text{double} \to \text{char}) \to \underline{\text{int}})$
$t = (\text{double} \to \underline{\text{char}})$
$v = \underline{\text{double}}$

$s \subseteq \{\text{ dom : t, rng : int }\} \cup \{\text{ num : 0 }\}$
$t \subseteq \{\text{ dom : v, rng : char, num : 0 }\}$
$v \subseteq \underline{\text{double}}$

$s = \{\text{ dom : t, rng : int }\} \cup \{\text{ num : 0 }\} \cup \gamma$
$t = \{\text{ dom : v, rng : char, num : 0 }\} \cup \delta$
$v = \underline{\text{double}} \cup \varepsilon$

unification ~ Gaussian elimination

$s = \ldots \quad \gamma = \varnothing$
$t = \ldots \quad \delta = \varnothing$
$v = \underline{\text{double}} \quad \varepsilon = \varnothing$

if they are not ∅ we found a type error

Fagan's "soft typer" works on all of our "hard" examples

```
;; Representing Russian dolls and computing their depth

;; RussianDoll = 'doll u (cons RussianDoll '())


;; RussianDoll -> Natural

(define (depth r)

  (cond

    [(symbol? r) 0]

    [else (+ 1 (depth (first r)))]))


(depth 'doll) ;; —> 0

(depth '(((doll))) ;; —> 3
```

```
[[μ (rd)(U 'doll (cons RussianDoll '()))]

 —>

 Natural]
```

Fagan's "soft typer" works on all of our "hard" examples

```
;; Representing propositions and checking tautology
;; Proposition = Boolean u [Boolean -> Proposition]


;; Proposition -> Boolean
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true) (tautology? (p false))))]))


(tautology? true)
(tautology? (lambda (x) (lambda (y) (
```

```
[[μ (p)(U Boolean (—> Boolean p))]
 —>
 Boolean]
```

Fagans' soft typer *cannot*

- present types in an accessible manner
- deal with more than small toy programs
- cope with anything but the core functional language

Fagans' soft typer *cannot*

- present types in an accessible manner
- deal with more than small toy programs
- cope with anything but the core functional language

Can we deal with

???

- 1,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an "actionable" manner

Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

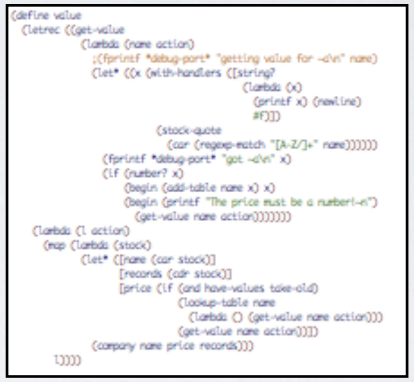

Andrew Wright

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

Andrew Wright

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization



Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

Andrew Wright

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

```
(define value
  (letrec ((get-value
             (lambda (name action)
               ;(fprintf *debug-port* "getting value for ~a\n" name)
               (let* ((x (with-handlers ([string?
                                            (lambda (x)
                                              (printf x) (newline)
                                              #f)])
                           (stock-quote
                             (car (regexp-match "[A-Z/]+" name))))))
                 (fprintf *debug-port* "got ~a\n" x)
                 (if (number? x)
                     (begin (add-table name x) x)
                     (begin (printf "The price must be a number!~n")
                       (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
        l))))
```

$s \subseteq \{$ dom : t, rng : int $\}$ u $\{$ num : 0 $\}$
$t \subseteq \{$ dom : v, rng : char, num : 0 $\}$
$v \subseteq$ <u>double</u>

Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference


Andrew Wright

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

```
(define value
  (letrec ((get-value
            (lambda (name action)
              ;(fprintf *debug-port* "getting value for ~a\n" name)
              (let* ((x (with-handlers ([string?
                                          (lambda (x)
                                            (printf x) (newline)
                                            #f)])
                          (stock-quote
                            (car (regexp-match "[A-Z/]+" name))))))
                (fprintf *debug-port* "got ~a\n" x)
                (if (number? x)
                    (begin (add-table name x) x)
                    (begin (printf "The price must be a number!~n")
                      (get-value name action)))))))
    (lambda (l action)
      (map (lambda (stock)
             (let* ([name (car stock)]
                    [records (cdr stock)]
                    [price (if (and have-values take-old)
                               (lookup-table name
                                 (lambda () (get-value name action)))
                               (get-value name action))])
               (company name price records)))
        l))))
```

s ⊆ { dom : t, rng : int } ∪ { num : 0 }
t ⊆ { dom : v, rng : char, num : 0 }
v ⊆ <u>double</u>

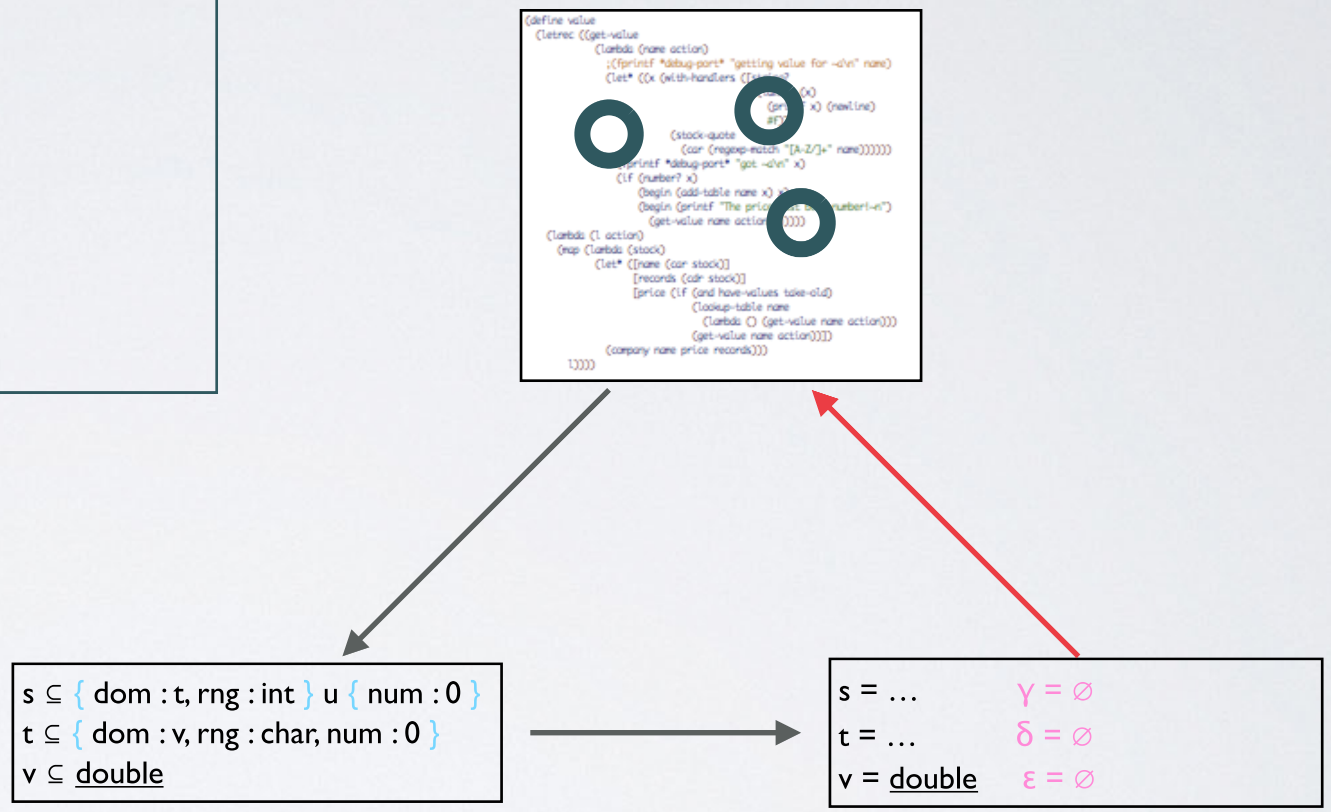
→

s = … γ = ∅
t = … δ = ∅
v = <u>double</u> ε = ∅

Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

Andrew Wright

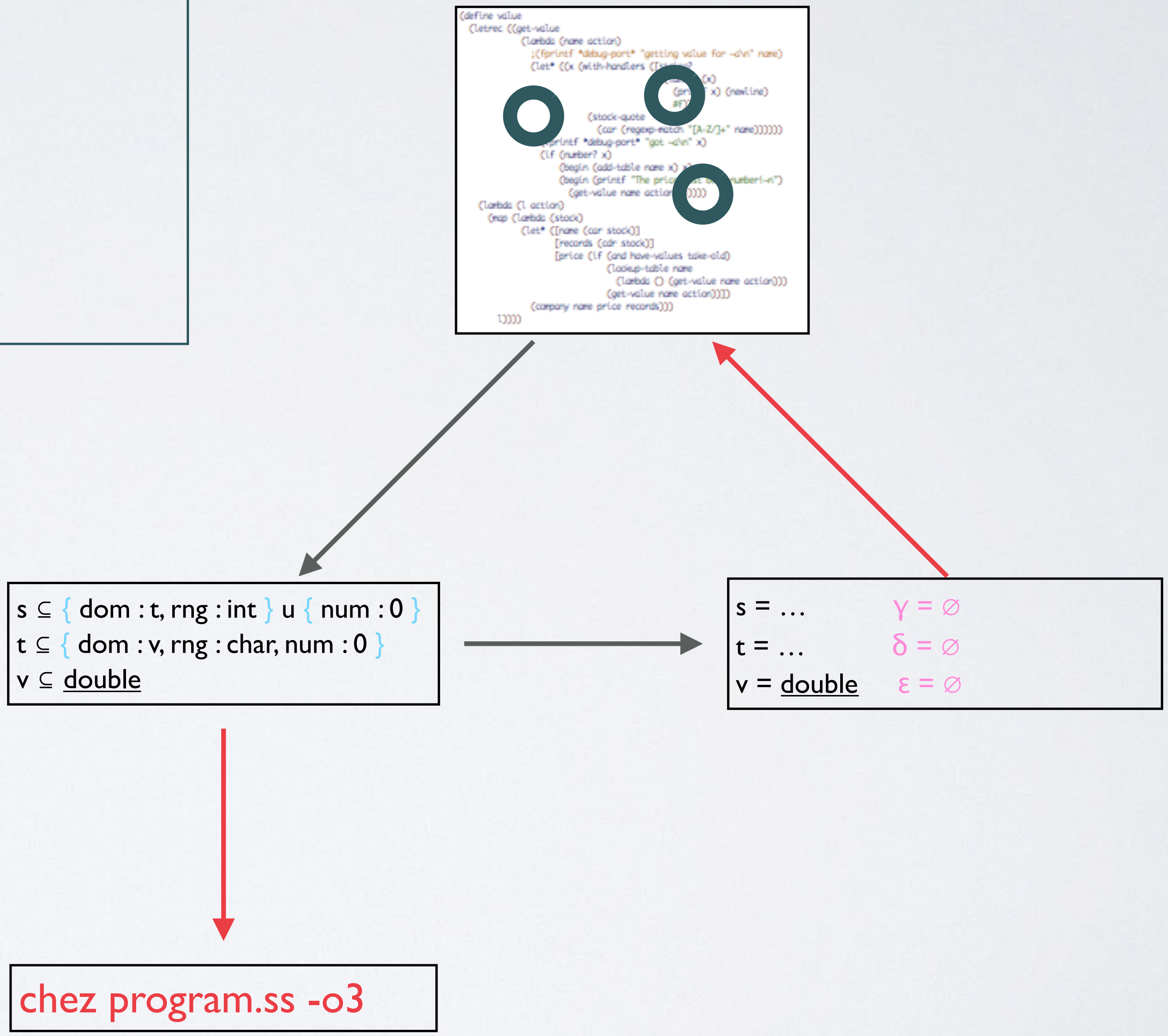
- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization



Wright's 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference


Andrew Wright

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization



My first sabbatical (1993-94)

Shriram Krishnamurthi's starter project

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

Shriram Krishnamurthi's starter project

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

RESULT: It works.

My first sabbatical (1993-94)

Shriram Krishnamurthi's starter project

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

RESULT: It works.

My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

RESULT: It works.

Shriram Krishnamurthi's starter project

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

Soft Scheme's were violations of the Geneva convention.

My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

RESULT: It works.

Soft Scheme supports my module's but is *not* modular.

Shriram Krishnamurthi's starter project

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

Soft Scheme's were violations of the Geneva convention.

My first sabbatical (1993-94)

Shriram Krishnamurthi's starter project

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

Analyze Sitaram's SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

RESULT: It works.

Soft Scheme's were violations of the Geneva convention.

Soft Scheme supports my module's but is *not* modular.

Undergraduates cannot use Soft Scheme in PL course.

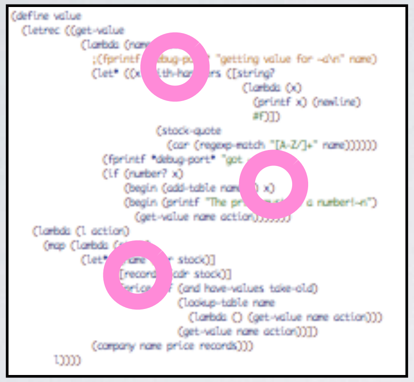
Errors matter.

Errors matter.

Developers matter.

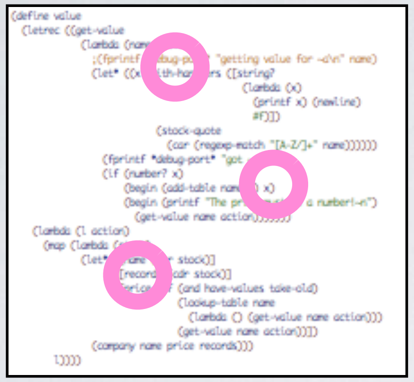
Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar's SBA

Cormac Flanagan



Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar's SBA

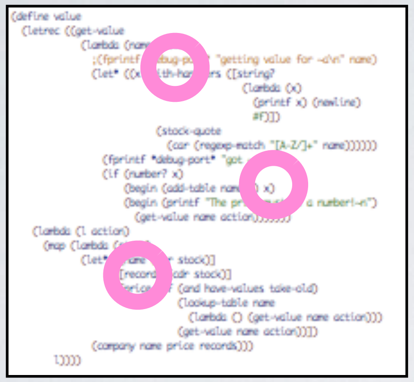
Cormac Flanagan



$$\underline{\text{double}} \subseteq \text{dom}(t)$$
$$t \subseteq \text{rng}(v)$$
$$\text{dom}(v) \subseteq \underline{\text{double}}$$

Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar's SBA

Cormac Flanagan



$\underline{\text{double}} \subseteq \text{dom}(t)$

$t \subseteq \text{rng}(v)$

$\text{dom}(v) \subseteq \underline{\text{double}}$

s = …

t = …

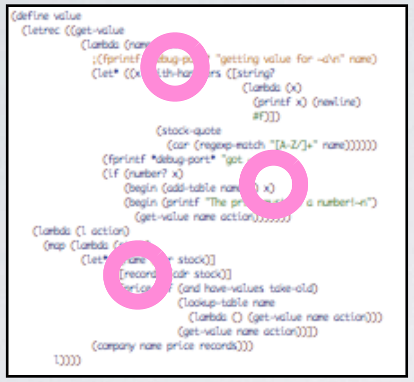
v = (-> d<u>ouble</u> (-> double …))

transitive closure through constructors

the solution is a least-fix point in a lattice

Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar's SBA

Cormac Flanagan



[Listof X] ⊆ dom(first @1) ✗

[Pairof Y Z] ⊆ dom(first @2) ✓

compare with specifications for primitive operations

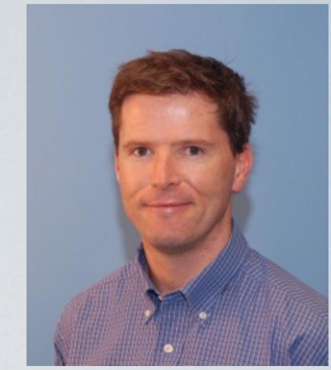
double ⊆ dom(t)
t ⊆ rng(v)
dom(v) ⊆ double

s = …
t = …
v = (-> double (-> double …))

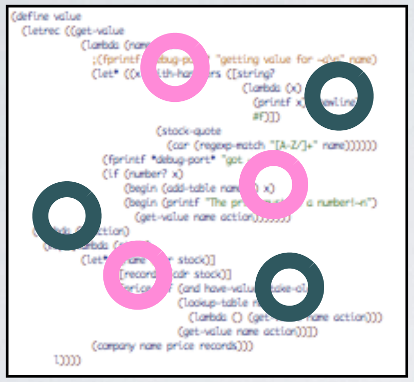
transitive closure through constructors

the solution is a least-fix point in a lattice

Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar's SBA



Cormac Flanagan



[Listof X] $\subseteq$ dom(first @1) ✗
because (first '()) raises an exn
[Pairof Y Z] $\subseteq$ dom(first @2) ✓

compare with specifications for primitive operations

double $\subseteq$ dom(t)
t $\subseteq$ rng(v)
dom(v) $\subseteq$ double

transitive closure through constructors

s = …
t = …
v = (-> double (-> double …))

the solution is a least-fix point in a lattice

- HM performs in near-linear time in practice
- HM is easy to understand *in principle*
- HM "smears" origin information across solution due to bi-directional flow

- SBA performs in linear time up to 2,500 loc
- SBA is also easy to explain to programmers
- SBA pushes information *only* along actual edges in the flow graph

- HM performs in near-linear time in practice
- HM is easy to understand *in principle*
- HM "smears" origin information across solution due to bi-directional flow

- SBA performs in linear time up to 2,500 loc
- SBA is also easy to explain to programmers
- SBA pushes information *only* along actual edges in the flow graph

And we can visualize those!

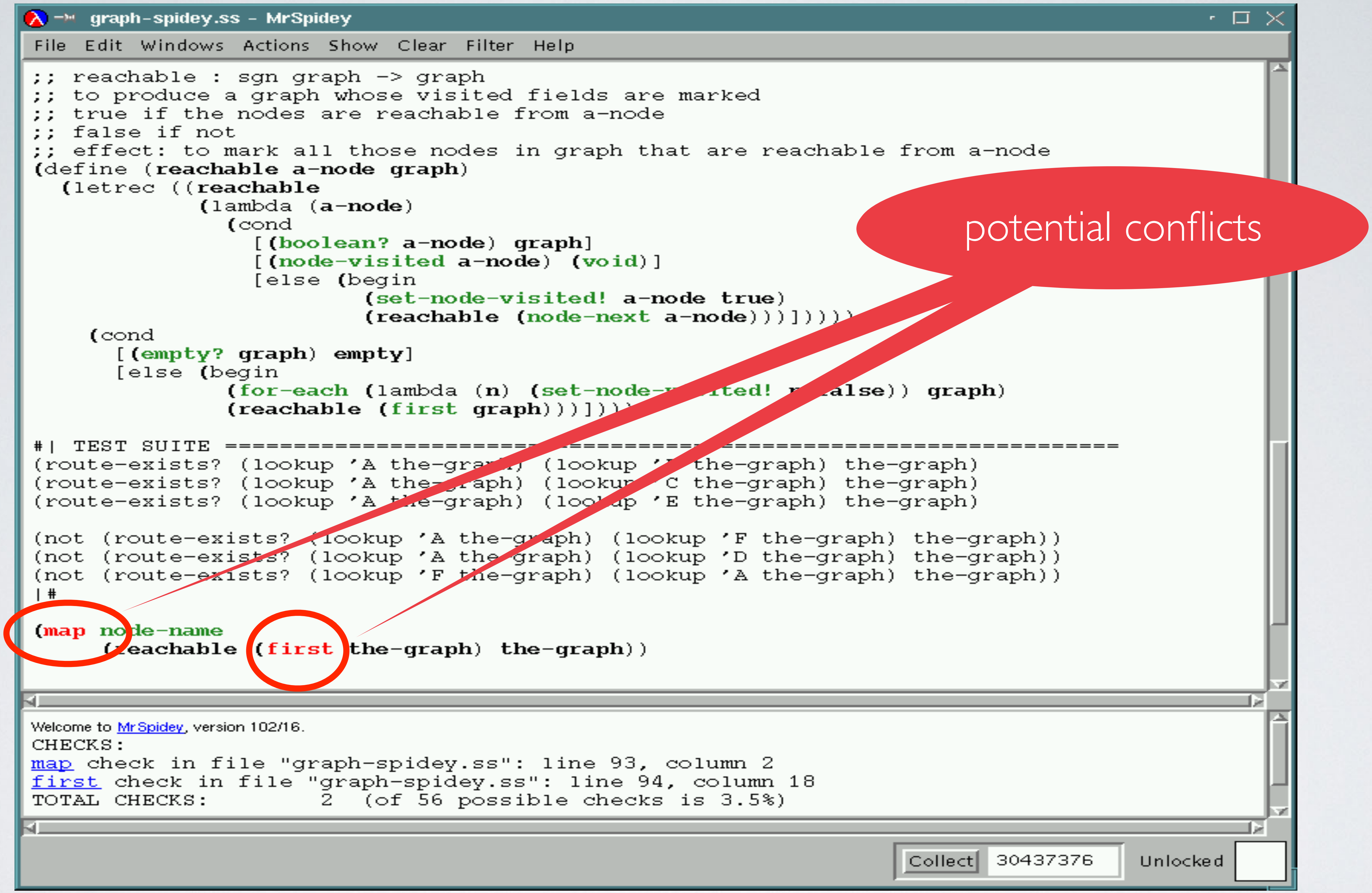
graph-spidey.ss - MrSpidey

File Edit Windows Actions Show Clear Filter Help

```scheme
;; reachable : sgn graph -> graph
;; to produce a graph whose visited fields are marked
;; true if the nodes are reachable from a-node
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
  (letrec ((reachable
            (lambda (a-node)
              (cond
                [(boolean? a-node) graph]
                [(node-visited a-node) (void)]
                [else (begin
                        (set-node-visited! a-node true)
                        (reachable (node-next a-node)))])))))
    (cond
      [(empty? graph) empty]
      [else (begin
              (for-each (lambda (n) (set-node-visited! n false)) graph)
              (reachable (first graph)))])))

#| TEST SUITE ===========================================================
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)

(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#

(map node-name
     (reachable (first the-graph) the-graph))
```

```
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS:        2  (of 56 possible checks is 3.5%)
```

Collect 30437376 Unlocked


graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; reachable : sgn graph -> graph
;; to produce a graph whose visited fields are marked
;; true if the nodes are reachable from a-node
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))]))))
(cond
[(empty? graph) empty]
[else (begin
(reachable (first graph)))])))
#| TEST SUITE
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph))
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376
Unlocked
potential conflicts

graph-spidey.ss - MrSpidey

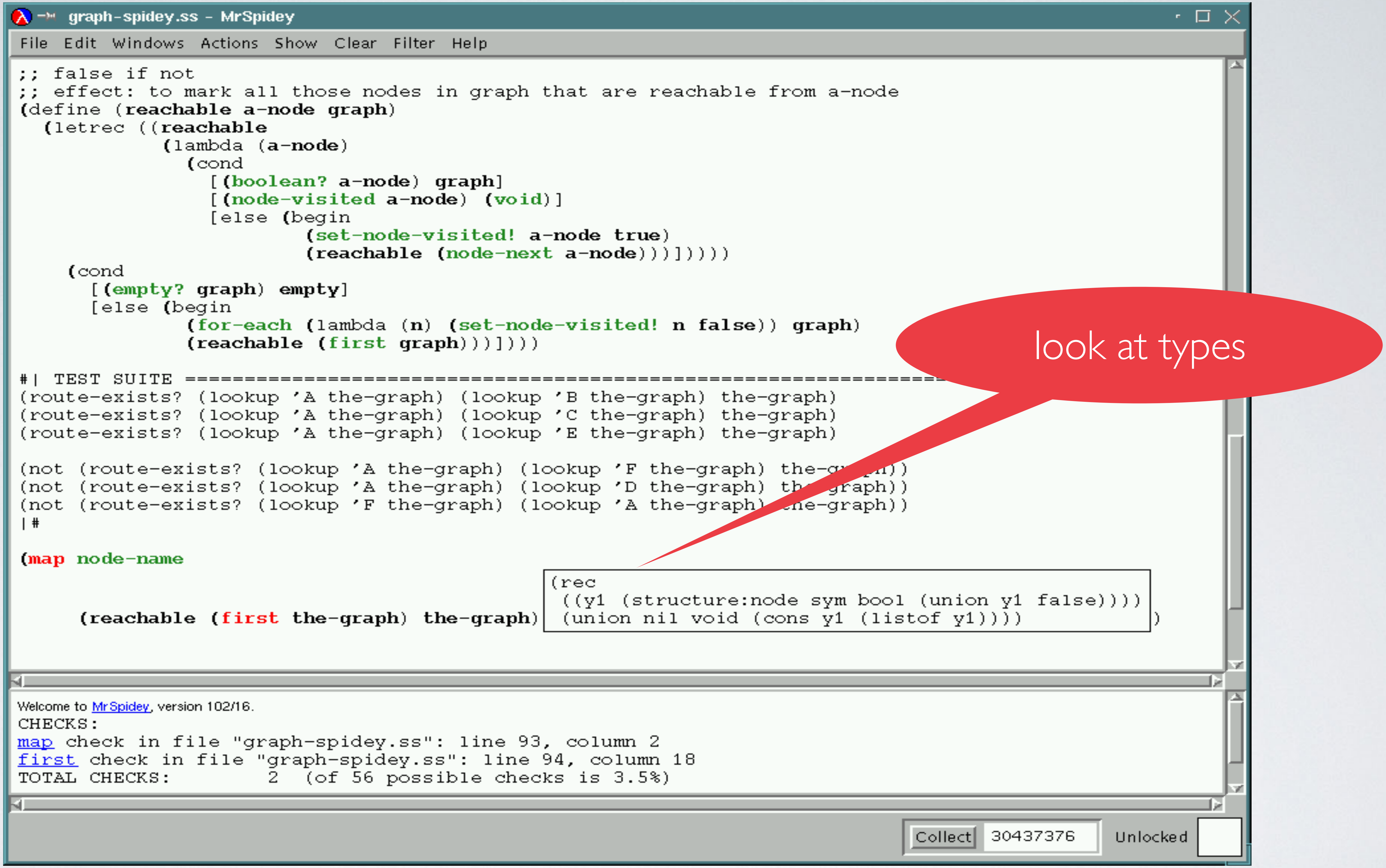
File Edit Windows Actions Show Clear Filter Help

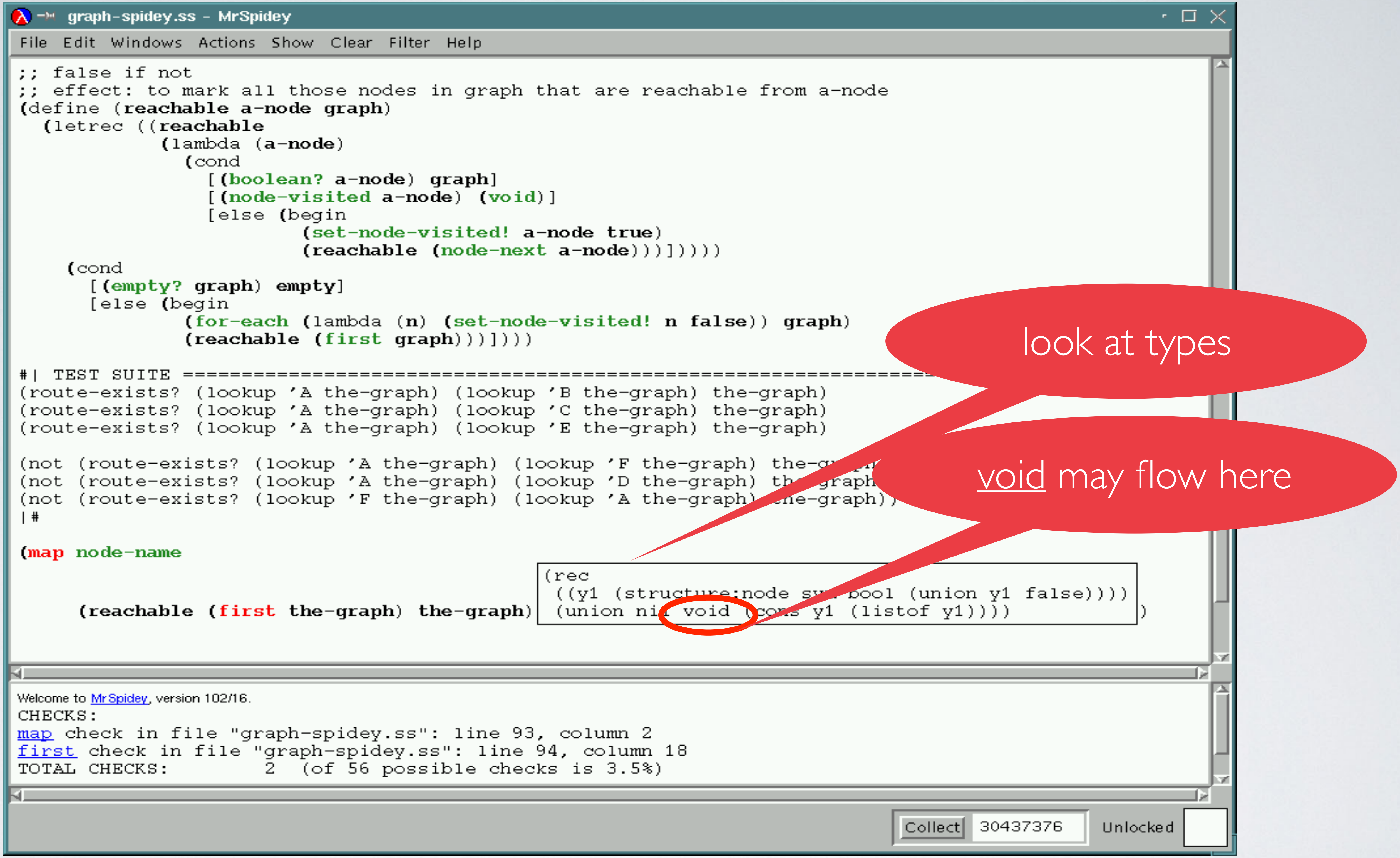
```scheme
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
  (letrec ((reachable
            (lambda (a-node)
              (cond
                [(boolean? a-node) graph]
                [(node-visited a-node) (void)]
                [else (begin
                        (set-node-visited! a-node true)
                        (reachable (node-next a-node)))]))))
    (cond
      [(empty? graph) empty]
      [else (begin
              (for-each (lambda (n) (set-node-visited! n false)) graph)
              (reachable (first graph)))])))

#| TEST SUITE ==============================================================
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)

(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#

(map node-name
     (reachable (first the-graph) the-graph)
     (rec
      ((y1 (structure:node sym bool (union y1 false))))
      (union nil void (cons y1 (listof y1)))))
```

Welcome to MrSpidey, version 102/16.

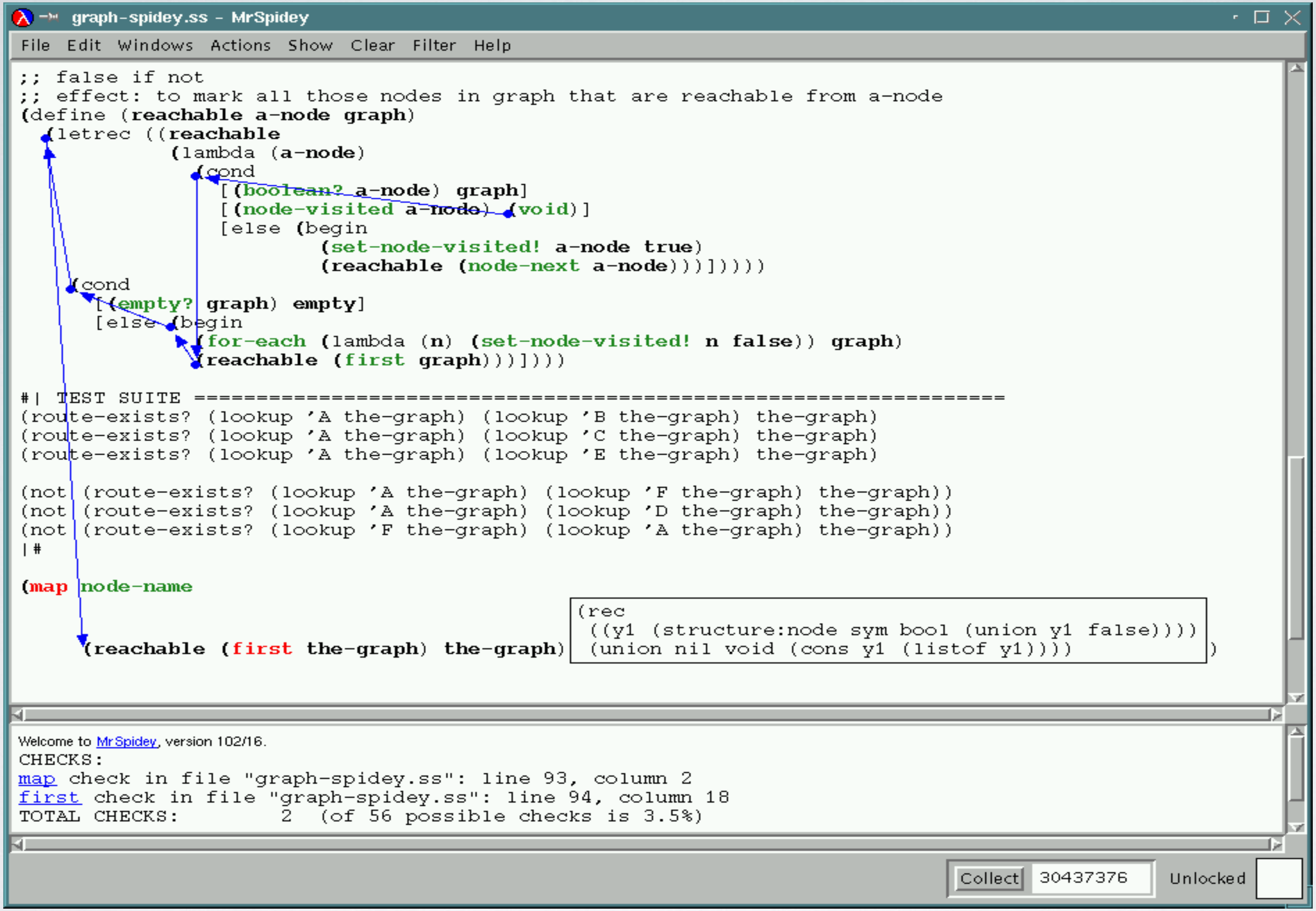
```
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS:        2  (of 56 possible checks is 3.5%)
```

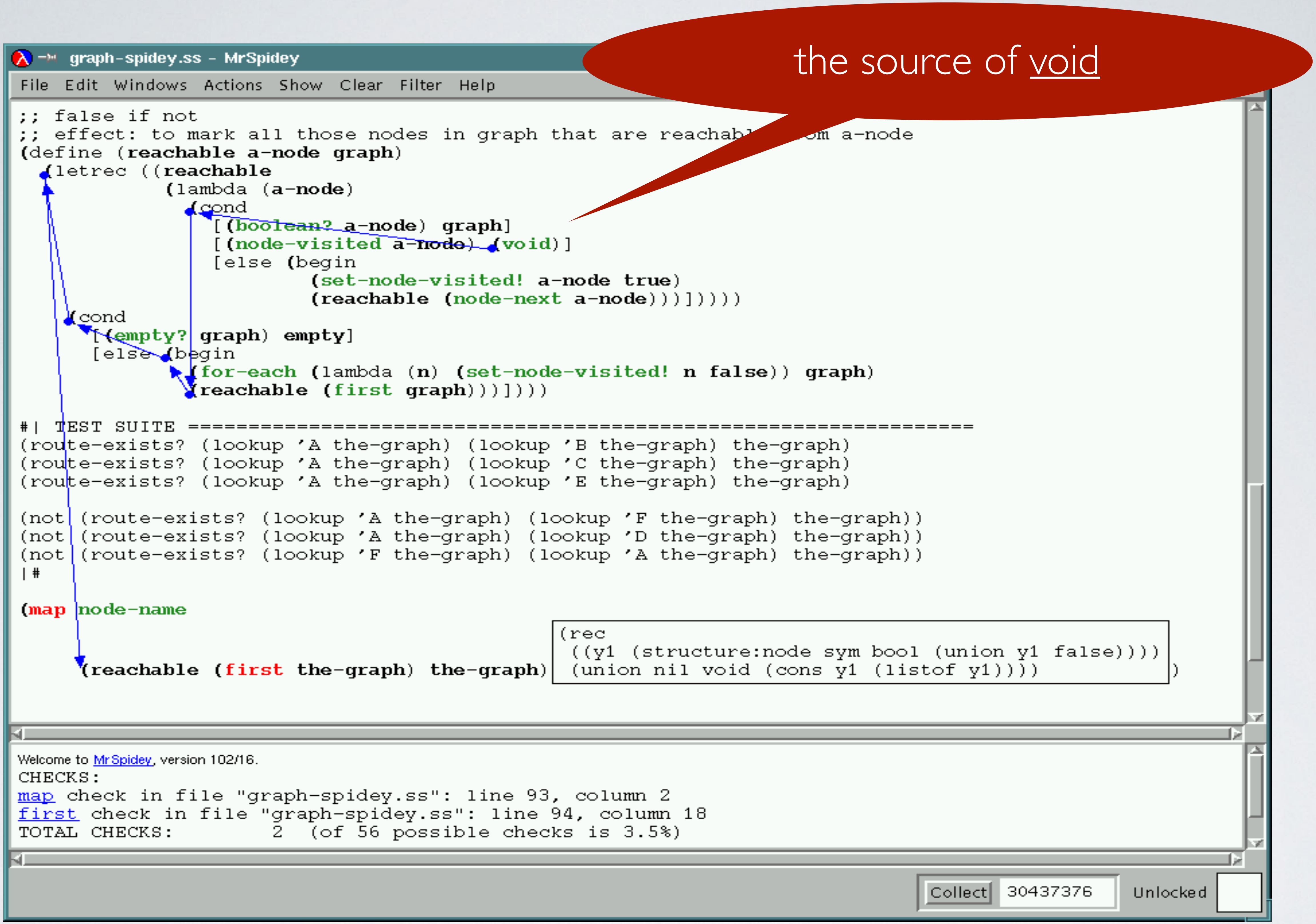
Collect 30437376 Unlocked

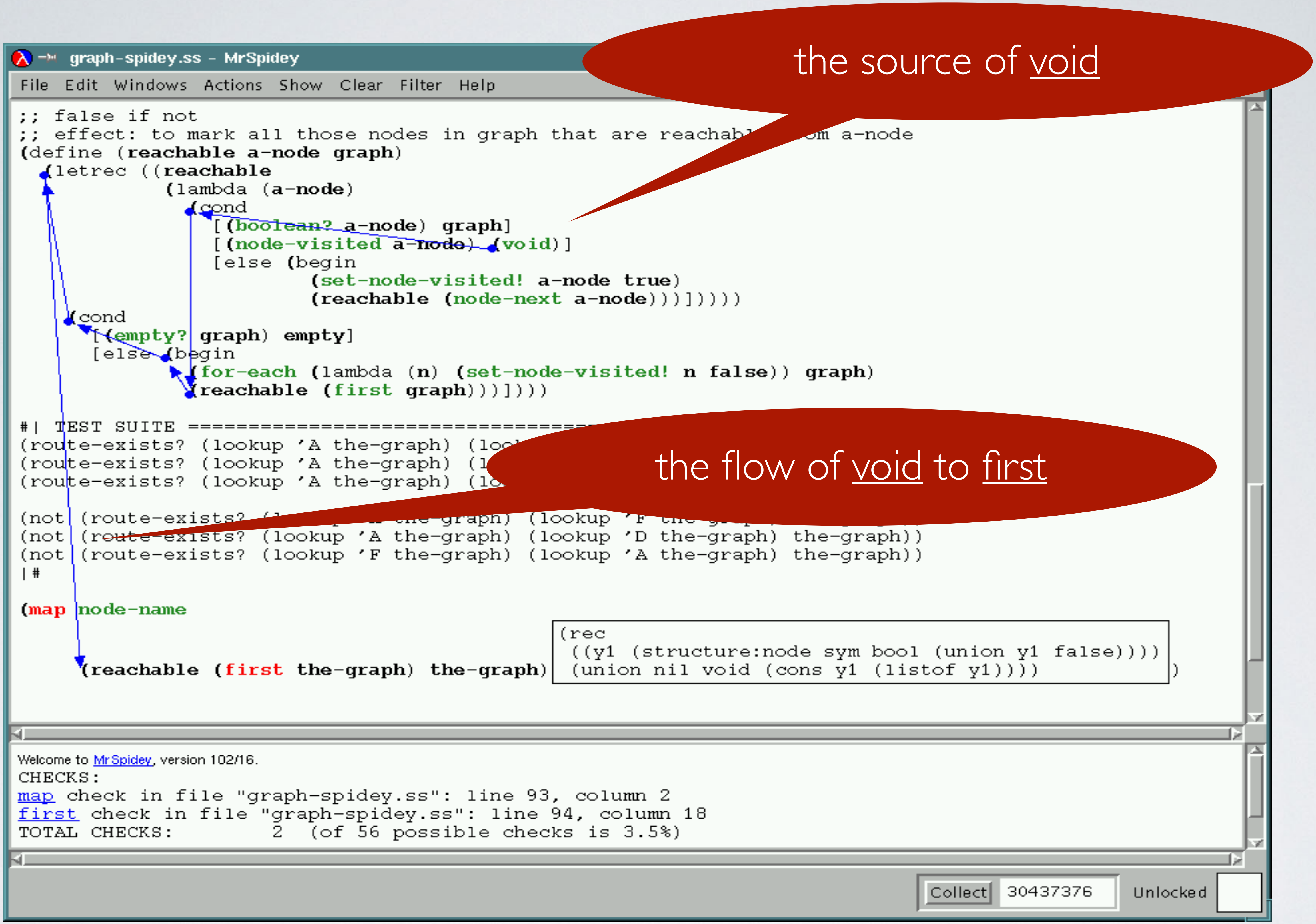

graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))]))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ====================
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376
Unlocked
look at types



graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))])))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ====
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph)
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph)
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
(union
void
y1 (listof y1))))
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376 Unlocked
look at types
void may flow here


graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))])))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ==
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
)
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376
Unlocked


the source of void
graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachab om a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))]))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))])))
#| TEST SUITE ==
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
)
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376 Unlocked


the source of void
the flow of void to first
graph-spidey.ss - MrSpidey
File Edit Windows Actions Show Clear Filter Help
;; false if not
;; effect: to mark all those nodes in graph that are reachab
om a-node
(define (reachable a-node graph)
(letrec ((reachable
(lambda (a-node)
(cond
[(boolean? a-node) graph]
[(node-visited a-node) (void)]
[else (begin
(set-node-visited! a-node true)
(reachable (node-next a-node)))])))))
(cond
[(empty? graph) empty]
[else (begin
(for-each (lambda (n) (set-node-visited! n false)) graph)
(reachable (first graph)))]))))
#| TEST SUITE ====================
(route-exists? (lookup 'A the-graph) (lo
(route-exists? (lookup 'A the-graph) (l
(route-exists? (lookup 'A the-graph) (l
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))
|#
(map node-name
(reachable (first the-graph) the-graph)
(rec
((y1 (structure:node sym bool (union y1 false))))
(union nil void (cons y1 (listof y1))))
Welcome to MrSpidey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
Collect 30437376
Unlocked

Flanagan can deal with

- 2,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an "actionable" manner

Flanagan can deal with

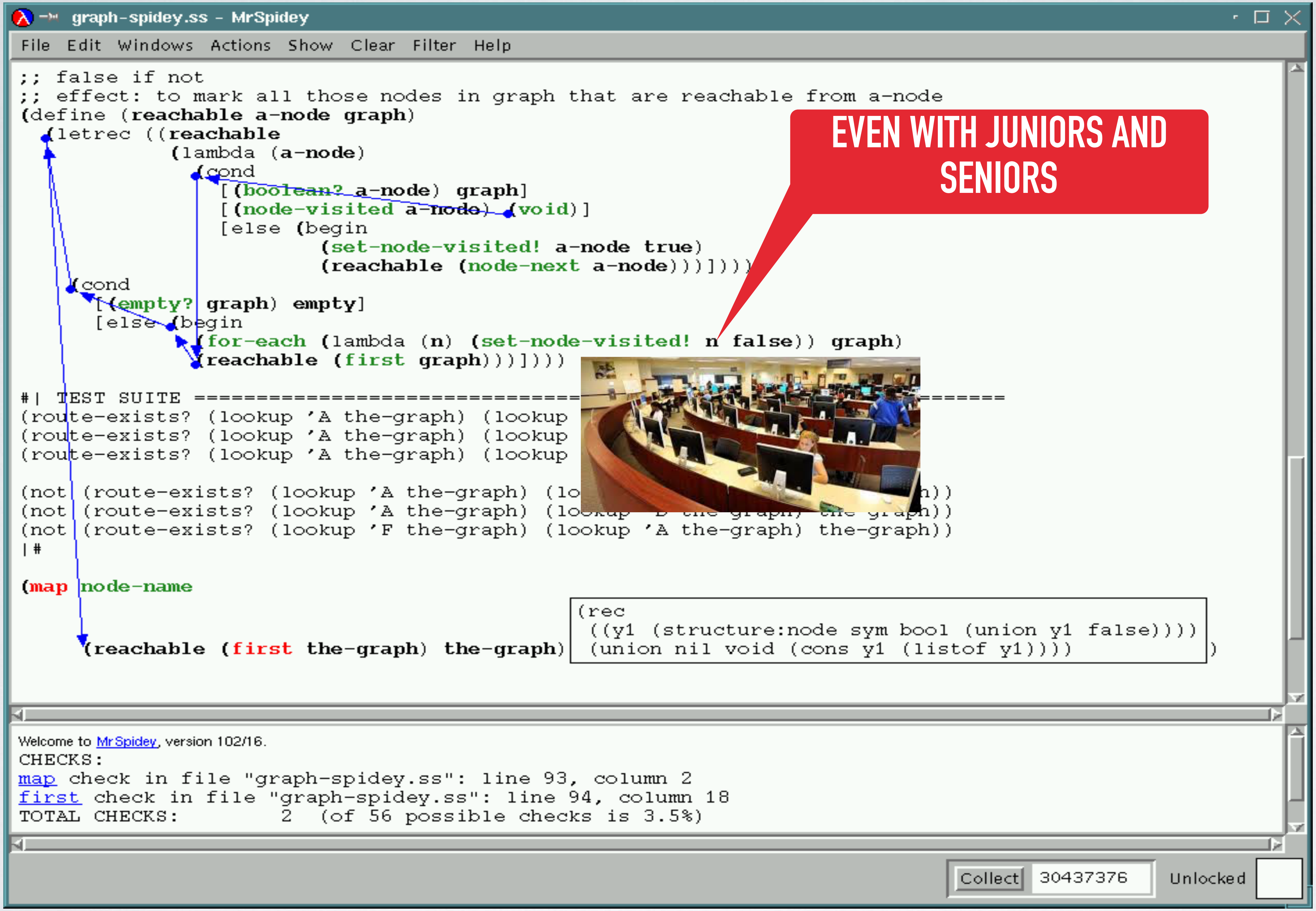
- 2,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an "actionable" manner

???

Can we deal with

- get juniors and seniors to use it (future devs)
- improve precision (e.g., arity of functions)
- "modules" (independently developed pieces)?

The good news



The not so good news

```
;; Natural Symbol -> S-expression

(define (wrap depth stuff)

  (cond

    [(zero? depth) stuff]

    [else (list (wrap (- depth 1) stuff)]]))


(wrap 3 ‘pizza) ;; —> ‘(((pizza)))

(wrap 2 ‘doll)  ;; -> ‘((doll))
```

The not so good news

~ (list depth stuff) = arg

```
;; Natural Symbol -> S-expression
(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff)]]))


(wrap 3 ‘pizza) ;; –> ‘(((pizza)))
(wrap 2 ‘doll)  ;; -> ‘((doll))
```

The not so good news

```
;; Natural Symbol -> S-expression
(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff))]))

(wrap 3 ‘pizza) ;; —> ‘(((pizza)))
(wrap 2 ‘doll)  ;; -> ‘((doll))
```

~ (list depth stuff) = arg

~ (second arg)

~ (first arg)

The not so good news

```
;; Natural Symbol -> S-expression
(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff))]))

(wrap 3 ‘pizza) ;; —> ‘(((pizza)))
(wrap 2 ‘doll)  ;; -> ‘((doll)
```

~ (list depth stuff) = arg

~ (second arg)

~ (first arg)

O(n^8)

Selectors Make Set-Based Analysis Too Hard

Philippe Meunier, Robert Bruce Findler[†], Paul Steckler and Mitchell Wand

College of Computer and Information Science
Northeastern University
Boston, MA 02115
{meunier,steck,wand}@ccs.neu.edu

[†] Department of Computer Science
University of Chicago
Chicago, IL 60637
robby@cs.uchicago.edu

The bad news

```
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    … )
…
… (extract (dispatch-table k) m)…
```

The bad news

```
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    … )
…
… (extract (dispatch-table k) m)…
```

(U False Window)

The bad news

(U False Window)

```
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    … )
…
… (extract (dispatch-table k) m)…
```

```
;; Natural -> Table
(define (dispatch-table n)
  (define v (make-vector n))
  (for ((i v)) (vector-set! v i …))
  ;; --- client code
  … )
…
… (extract (dispatch-table k) m)…
```

The bad news

```
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    … )
…
… (extract (dispatch-table k) m)…
```

(U False Window)

```
;; Natural -> Table
(define (dispatch-table n)
  (define v (make-vector n))
  (for ((i v)) (vector-set! v i …))
  ;; --- client code
  … )
…
… (extract (dispatch-table k) m)…
```

(U False
…
20 more lines
…
Window)

The bad news

```
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    … )
…
… (extract (dispatch-table k) m)…
```

(U False Window)

Small syntactic changes without semantic meaning imply large changes to inferred types

```
;; Natural -> Table
(define (dispatch-tab
  (define v (make-vec
  (for ((i v)) (vector-set! v i …))
  ;; --- client code
  … )
…
… (extract (dispatch-table k) m)…
```

(U False
. . .
20 more lines
. . .
Window)

The worse news

It's not only n^8, it's also whole-program only.

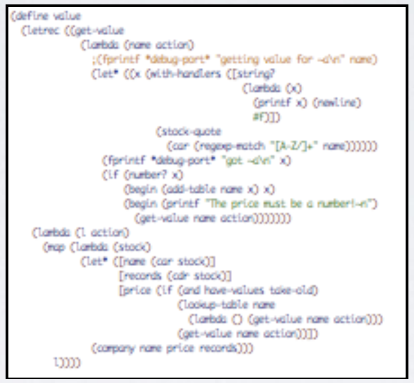
```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

The worse news

It's not only n^8, it's also whole-program only.

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

Components

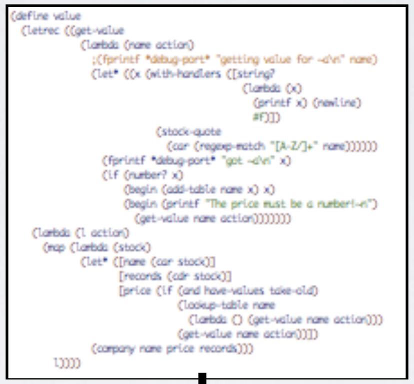


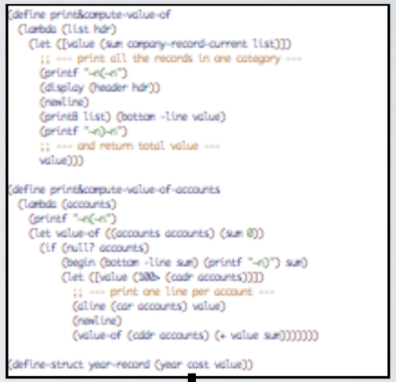
The worse news

It's not only n^8, it's also whole-program only.

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

Components





Constraints

s ⊆ {dom : t, rng : int}
t ⊆ {dom : v} ∪ {num : 0}
v ⊆ double

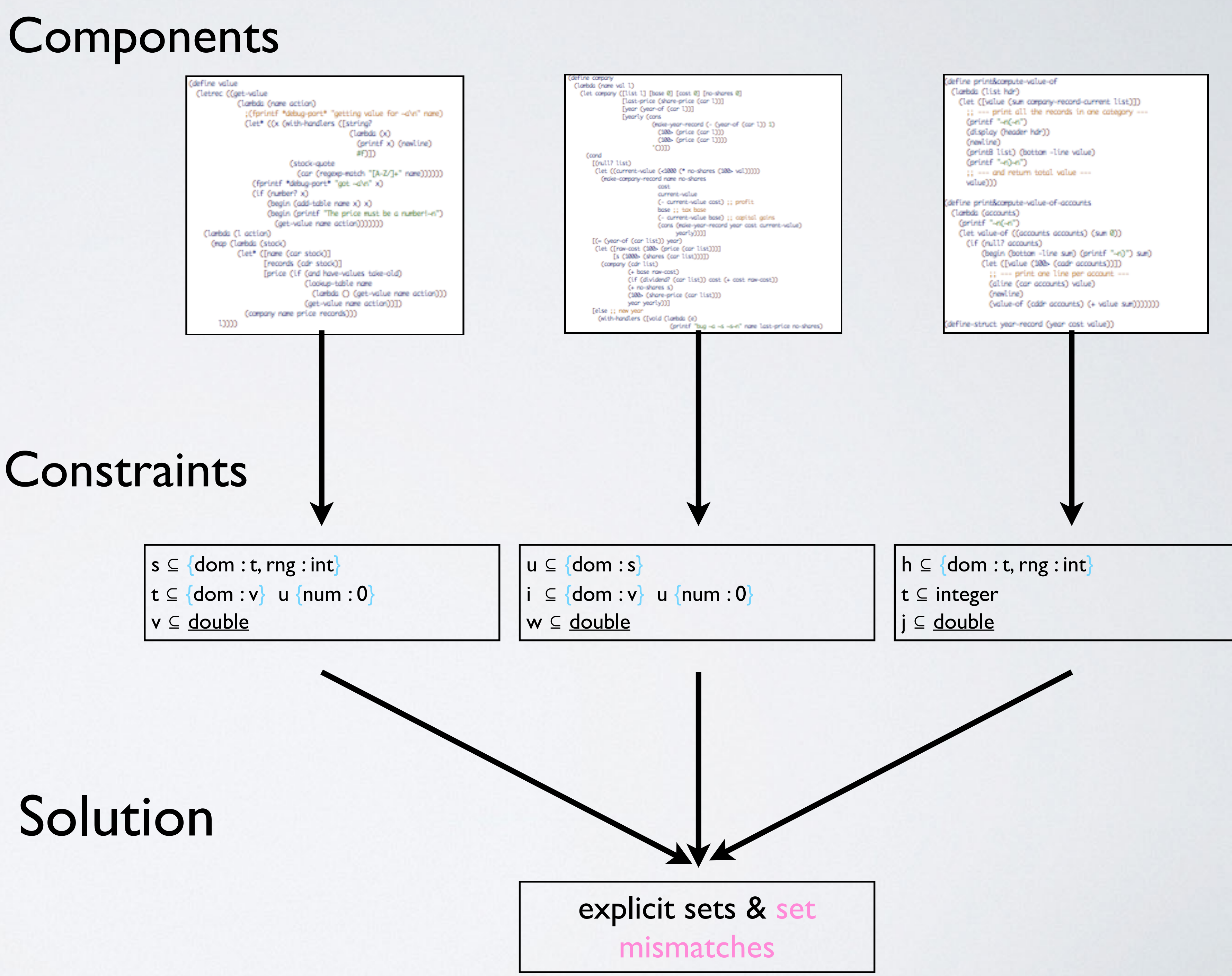
u ⊆ {dom : s}
i ⊆ {dom : v} ∪ {num : 0}
w ⊆ double

h ⊆ {dom : t, rng : int}
t ⊆ integer
j ⊆ double

The worse news

It's not only n^8, it's also whole-program only.

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

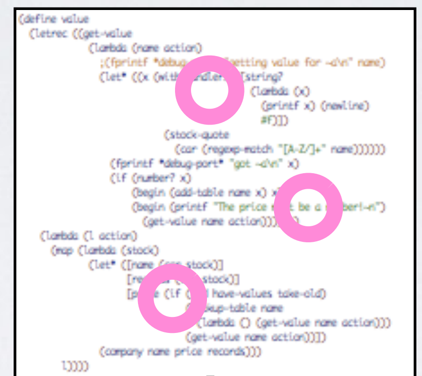


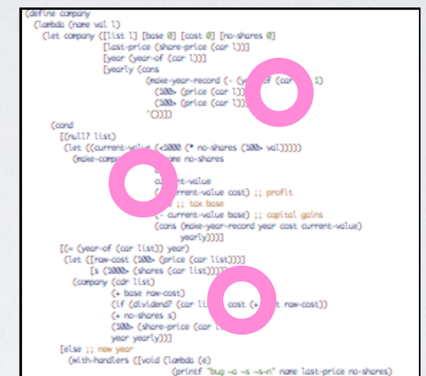
The worse news

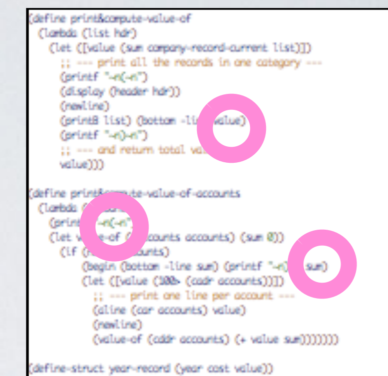
It's not only n^8, it's also whole-program only.

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

Components







Constraints

s ⊆ {dom : t, rng : int}
t ⊆ {dom : v} u {num : 0}
v ⊆ double

u ⊆ {dom : s}
i ⊆ {dom : v} u {num : 0}
w ⊆ double

h ⊆ {dom : t, rng : int}
t ⊆ integer
j ⊆ double

Solution

explicit sets & set mismatches

The worse news

It's not only n^8, it's also whole-program only.

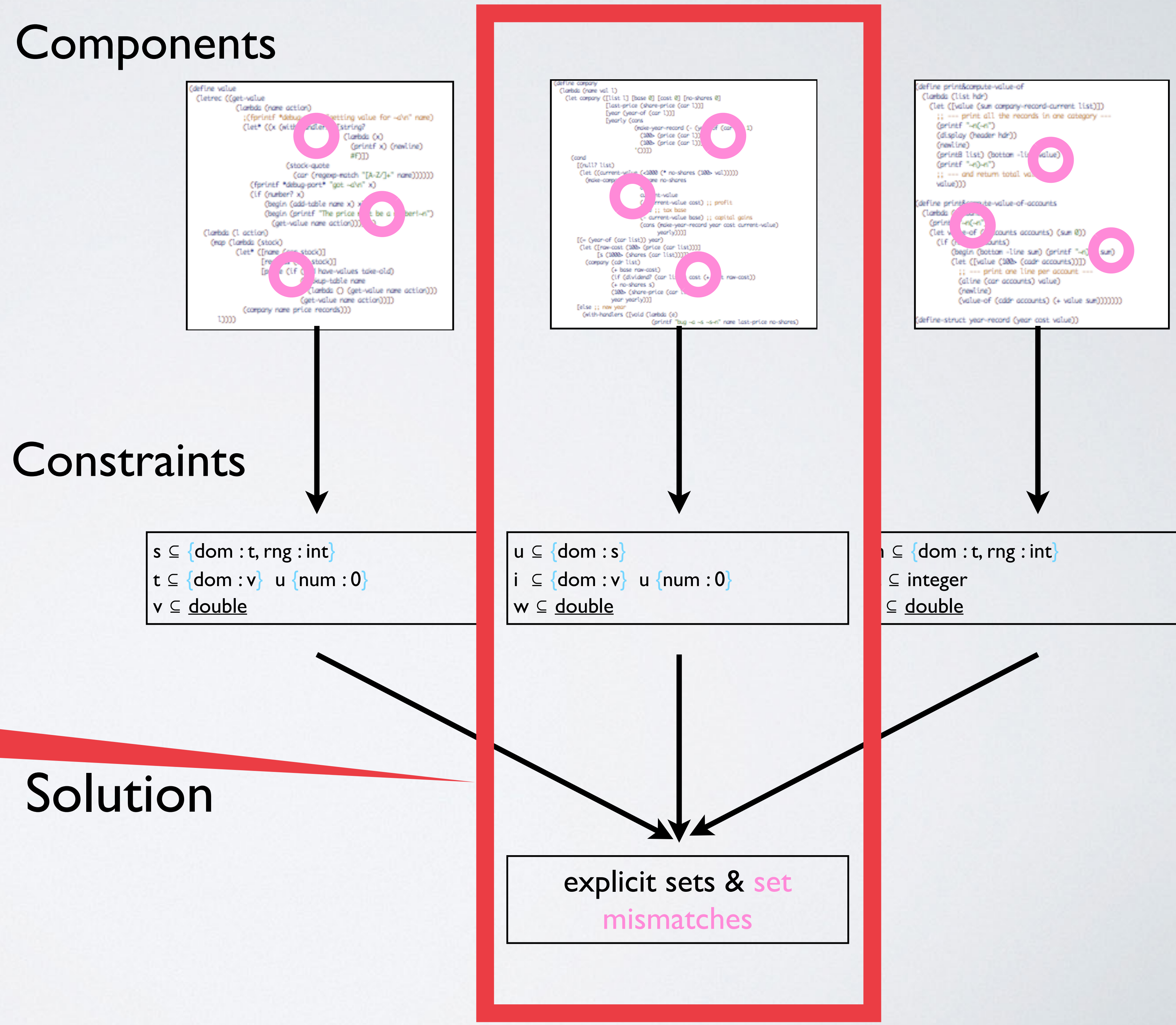
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs

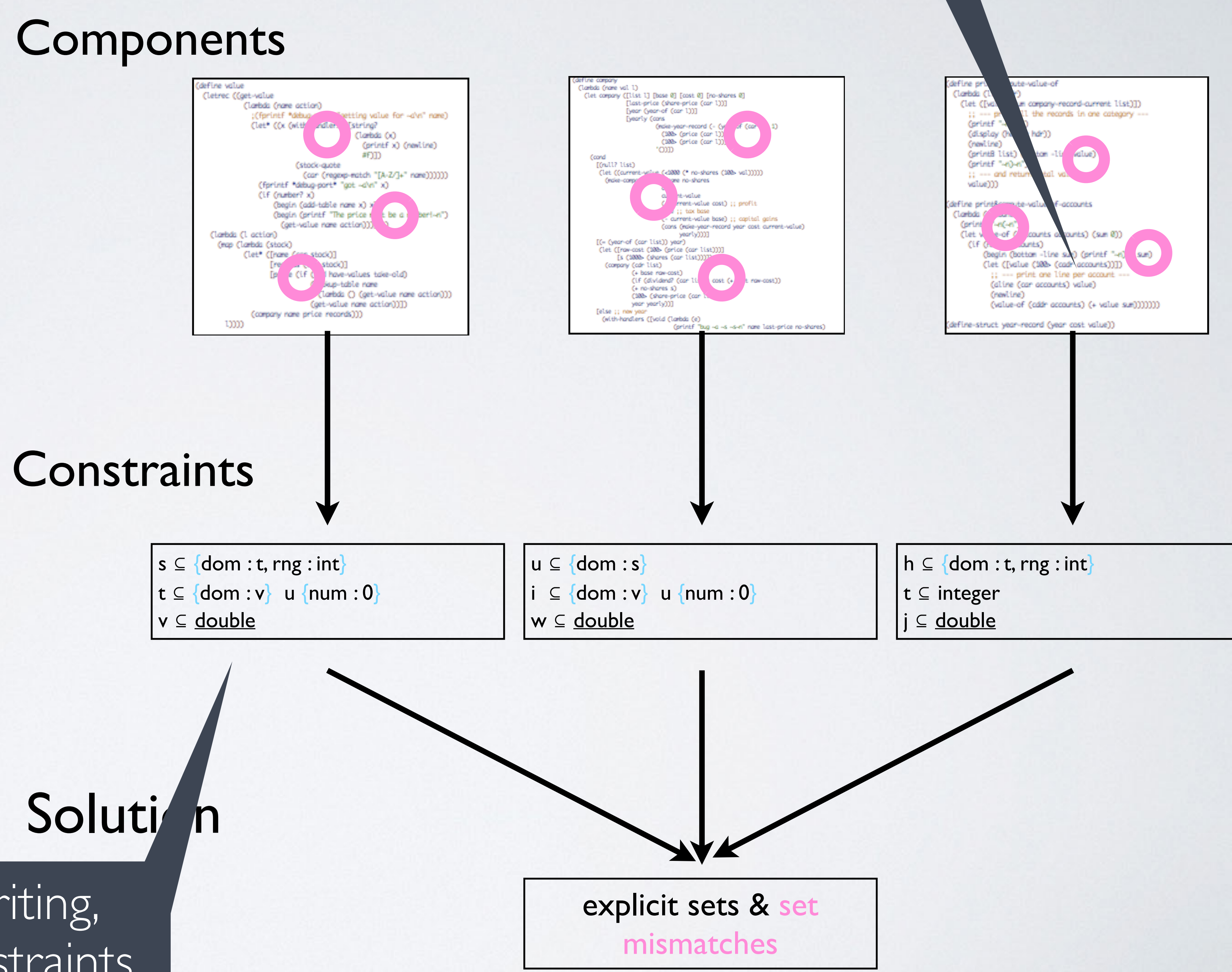


Now we can work with 1 module and get on-line analysis

The worse news

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

What's worse, we can't just add another module.

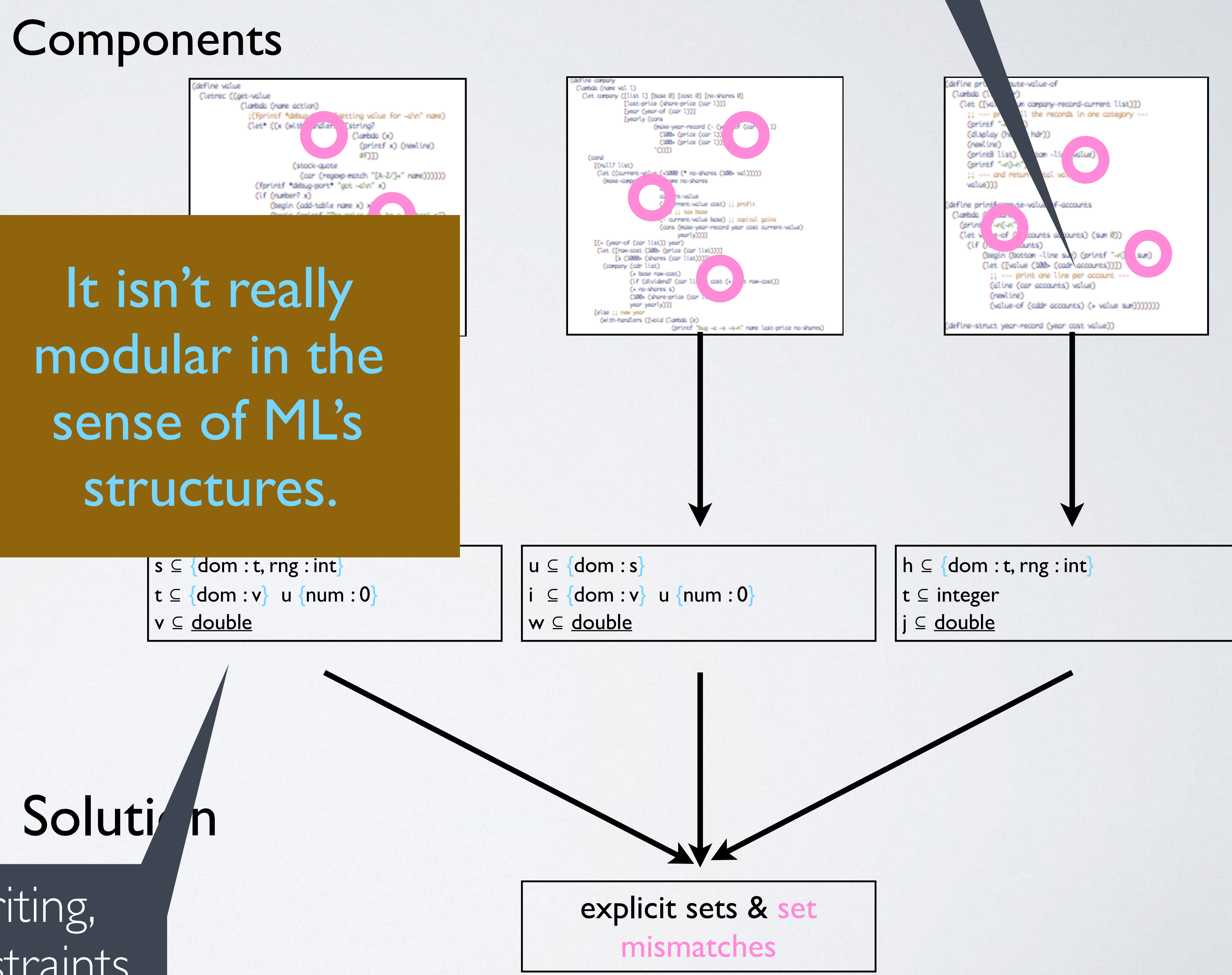


It costs O(n^2) space (writing, reading) to store graph constraints.

The worse news

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

What's worse, we can't just add another module.



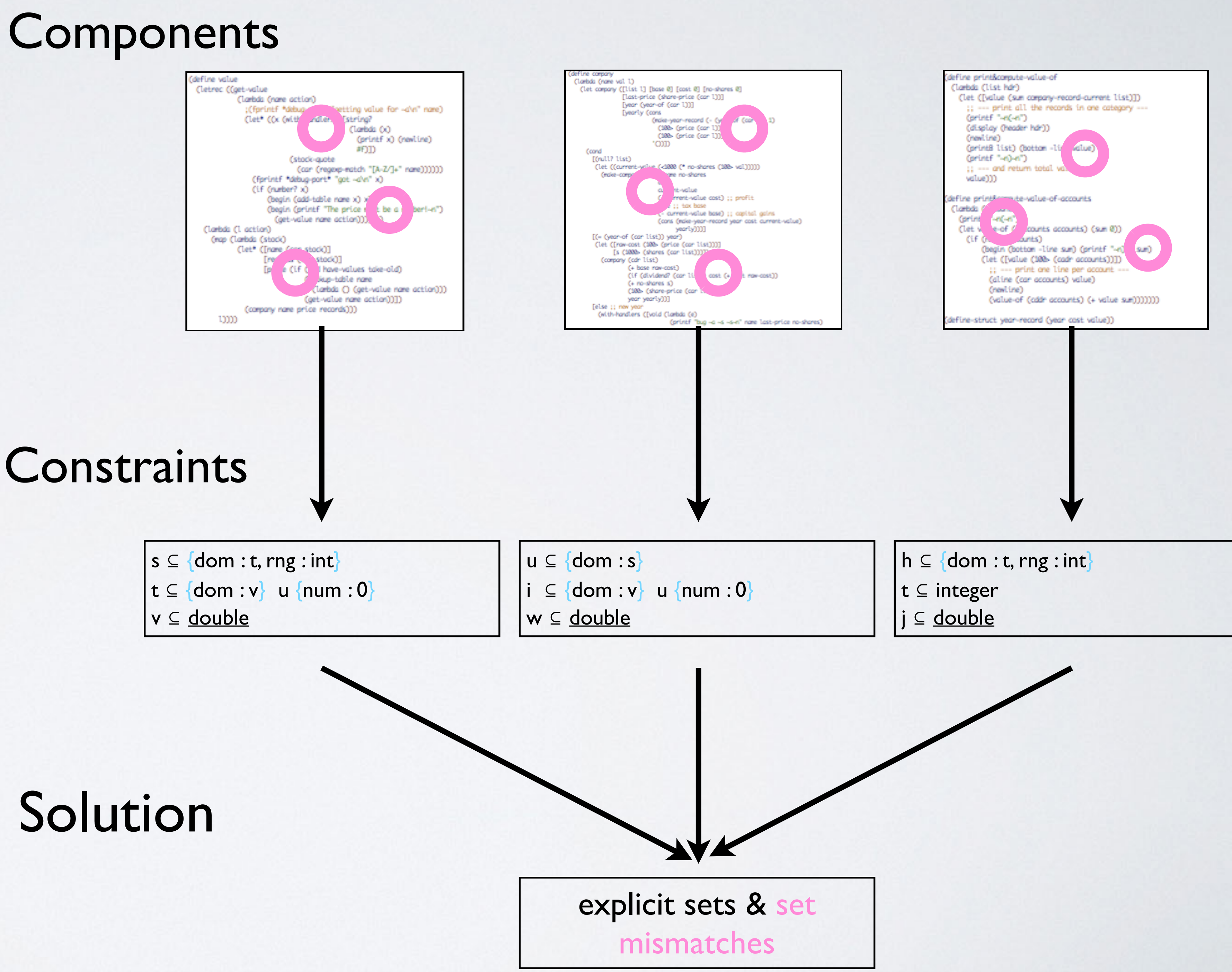
It isn't really modular in the sense of ML's structures.

It costs O(n^2) space (writing, reading) to store graph constraints.

Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints

Philippe Meunier

```
1,000 lines ~ 1 min

2,000 lines ~ 2 min

3,000 lines ~ 3 min

3,500 lines ~ 20 min

40,000 lines ~ 10 hrs
```

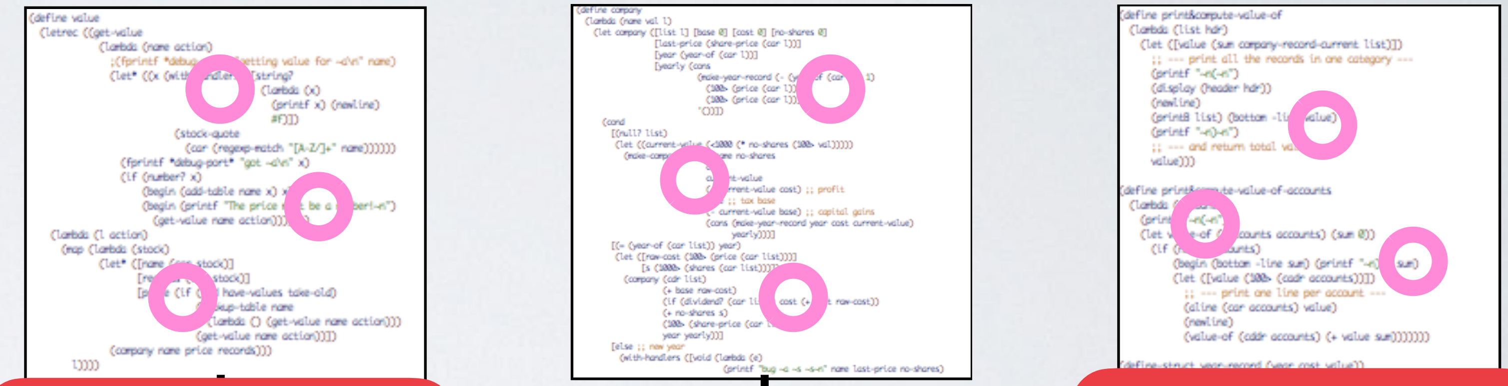


Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints

Philippe Meunier

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

Components



contract

contract

Use contracts in lieu of signatures.

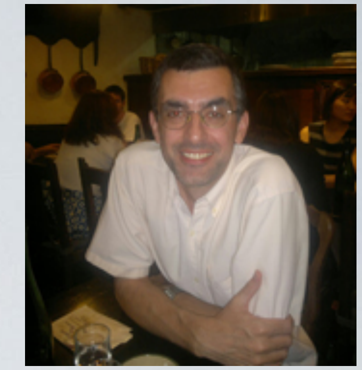
Constraints

s ⊆ {dom : t, rng : int} t ⊆ {dom : v} u {num : 0} v ⊆ double	u ⊆ {dom : s} i ⊆ {dom : v} u {num : 0} w ⊆ double	h ⊆ {dom : t, rng : int} t ⊆ integer j ⊆ double

Solution

explicit sets & set mismatches

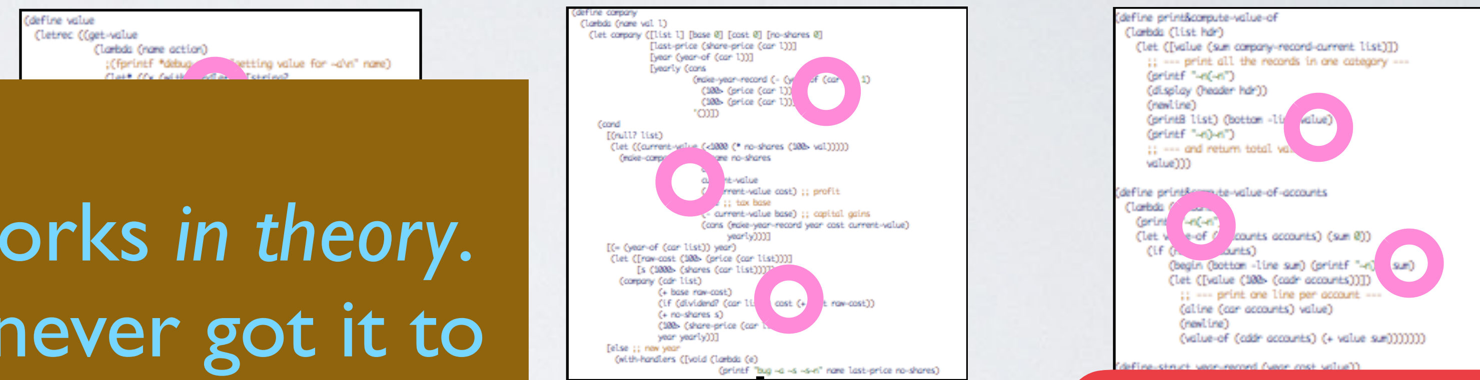
Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints



Philippe Meunier

```
1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs
```

Components



contract

It works *in theory*. We never got it to work well *in practice*.

Constraints

Use contracts in lieu of signatures.

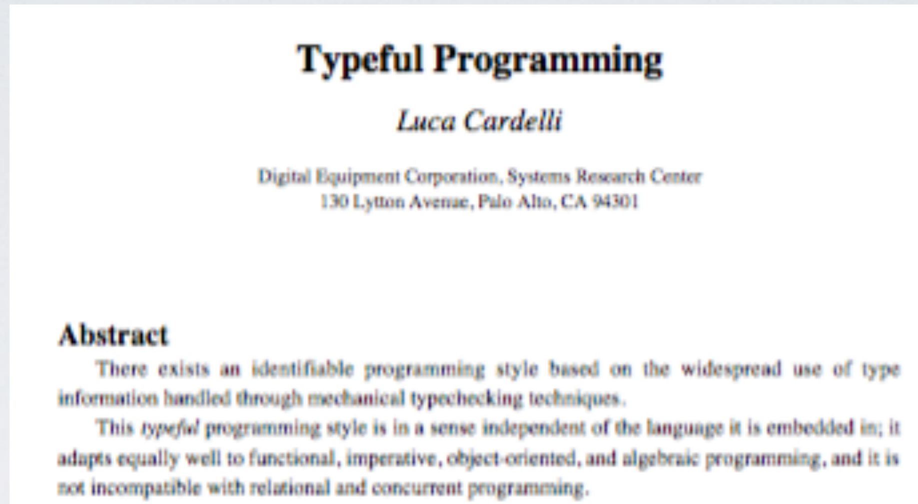
$s \subseteq \{dom : t, rng : int\}$	$u \subseteq \{dom : s\}$	$h \subseteq \{dom : t, rng : int\}$
$t \subseteq \{dom : v\}$ u $\{num : 0\}$	$i \subseteq \{dom : v\}$ u $\{num : 0\}$	$t \subseteq$ integer
$v \subseteq$ <u>double</u>	$w \subseteq$ <u>double</u>	$j \subseteq$ <u>double</u>

Solution

explicit sets & set mismatches

Modularity matters.

- Our code base has grown to 500,000 loc.
- A language renaissance has spread Untyped Languages beyond their niche uses.



Typeful Programming

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

Abstract

There exists an identifiable programming style based on the widespread use of type information handled through mechanical typechecking techniques.

This *typeful* programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming.

Modularity matters.

- Our code base has grown to 500,000 loc.
- A language renaissance has spread Untyped Languages beyond their niche uses.

Signatures matter.

- Nobody ought to read an entire module to understand its services.
- Racket programmers use contracts as signatures.

Can we add types to this code **without** the ML-style projections/injections?

```
;; Representing Russian dolls and computing their depth

;; RussianDoll = ‘doll u (cons RussianDoll ‘())



;; RussianDoll -> Natural

(define (depth r)

  (cond

    [(symbol? r) 0]

    [else (+ 1 (depth (first r)))]))



(depth ‘doll) ;; —> 0

(depth ‘(((doll))) ;; —> 3
```

Can we add types to this code **without** the ML-style projections/injections?

```
;; Representing Russian dolls and computing their depth

TYPE RussianDoll = ‘doll u (cons RussianDoll ‘())




(define (depth r : RussianDoll) : Natural
  (cond
    [(symbol? r) 0]
    [else (+ 1 (depth (first r)))]))

(depth ‘doll) ;; —> 0
(depth ‘(((doll))) ;; —> 3
```

Can we add types to this code **without** the ML-style projections/injections?

```
;; Representing propositions and checking tautology
TYPE Proposition = Boolean u [Boolean -> Proposition]




(define (tautology? p : Proposition) : Boolean
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Sam Tobin-Hochstadt

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Sam Tobin-Hochstadt

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Sam Tobin-Hochstadt

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable "components" (files, packages, classes, modules) with clear boundaries.

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable "components" (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a "by need" basis plus the addition of typed components — incrementality

Sam Tobin-Hochstadt

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Sam Tobin-Hochstadt

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable "components" (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a "by need" basis plus the addition of typed components — incrementality

WANTED annotations should *go* on variables and other names and should **not** disturb existing code — idiomaticity

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level


Sam Tobin-Hochstadt

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable "components" (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a "by need" basis plus the addition of typed components — incrementality

WANTED annotations should *go* on variables and other names and should **not** disturb existing code — idiomaticity

WANTED the type annotations ought to be ***useful*** and ***meaningful*** — type soundness

Tobin-Hochstadt *incrementally* and *idiomatically* adds types to existing large systems at appropriate granularity level

Sam Tobin-Hochstadt

Gray, Findler, Flatt

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable "components" (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a "by need" basis plus the addition of typed components — incrementality

WANTED annotations should *go* on variables and other names and should **not** disturb existing code — idiomaticity

WANTED the type annotations ought to be ***useful*** and ***meaningful*** — type soundness

And all of this works for (almost) the full language — coverage

Typed Racket satisfies "idiomaticy"

```
;; Representing Russian dolls and computing their depth
(define-type RussianDoll (U ‘doll [cons RussianDoll ‘()]))
(define (depth {r : RussianDoll}) : Natural
  (cond
    [(symbol? r) 0]
    [else (+ 1 (depth (first r)))]))


(depth ‘doll) ;; —> 0
(depth ‘(((doll))) ;; —> 3
```

Typed Racket satisfies "idiomaticy"

```
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) : Boolean

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

Typed Racket satisfies “idiomaticy”

```
;; Representing propositions and checking tautology
(define-type Proposition (U Boolean [Boolean -> Proposition]))
(define (tautology? {p : Proposition}) : Boolean
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

no projection needed

no injection needed

Typed Racket satisfies “idiomaticy” **via ``flow propositions”**

```
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) : Boolean

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

```
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) :

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

boolean? : Any -> Boolean: ``and if it is true, the given value belongs to Boolean"

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

```
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) :

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

boolean? : Any -> Boolean:
``and if it is true, the given
value belongs to Boolean"

p is **not** a Boolean,
ergo it must be in
[Boolean -> Proposition]

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

```
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) :

  (cond

    [(boolean? p) p]

    [else (and (tautology? (p true)) (tautology? (p false)))]))


(tautology? true)

(tautology? (lambda (x) (lambda (y) (or x y))))
```

boolean? : Any -> Boolean:
``and if it is true, the given
value belongs to Boolean"

p applied to true
is OK

p is **not** a Boolean,
ergo it must be in
[Boolean -> Proposition]

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

$\Gamma \vdash e : \tau \mid (p+, p-)$

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

IN type environment
(the type of variables in e)

the expression e HAS

type **τ**

$\Gamma \vdash e : \tau \mid (p+, p-)$

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

IN type environment
(the type of variables in e)

the expression e HAS

type **τ**

$\Gamma \vdash e : \tau \mid (p+, p-)$

and if e evaluates
to a Truish value,
we KNOW p+

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

IN type environment (the type of variables in e)

the expression e HAS

type **τ**

$\Gamma \vdash e : \tau \mid (p+, p-)$

and if e evaluates to a Truish value, we KNOW p+

and if e evaluates to a False value, we KNOW p-

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

IN type environment
(the type of variables in e)

the expression e HAS

type **τ**

$\Gamma \vdash e : \tau \mid (p+, p-)$

and if e evaluates
to a Truish value,
we KNOW p+

and if e evaluates
to a False value, we
KNOW p-

The **knowledge** deals with plain values and paths into values:

- (odd? n) ~> if this yields False, n is even
- (prime? (second l)) ~> if this yields True, we know l has the shape [one, two, ?] *and* two is a prime number.

Typed Racket satisfies "idiomaticy" **via ``flow propositions"**

IN type environment
(the type of variables in e)

the expression e HAS

type **τ**

$$\Gamma \vdash e : \tau \mid (p_+, p_-)$$

and if e evaluates to a Truish value, we KNOW p+

and if e evaluates to a False value, we KNOW p-

The **logic** can cope with the usual Boolean primitives in a programming language: and, or, not, if (conditionals), etc.

The **knowledge** deals with plain values and paths into values:

- (odd? n) ~> if this yields False, n is even
- (prime? (second l)) ~> if this yields True, we know l has the shape [one, two, ?] *and* two is a prime number.

Typed Racket satisfies "incrementality"

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Typed Racket satisfies "incrementality"

Racket has always been a family of languages

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Typed Racket satisfies "incrementality"

Racket has always been a family of languages

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this
(define (      area s)
  (   (rect-width s) (rect-height s)))
```

Racket modules already specify their implementation language

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Typed Racket satisfies "incrementality"

Racket has always been a family of languages

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this
(define (   area s)
  (  (rect-width s) (rect-height s)))
```

Racket modules already specify their implementation language

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

`#lang racket`

Typed Racket satisfies "incrementality"

Racket has always been a family of languages

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this
(define (   area s)
  (   (rect-width s) (rect-height s)))
```

Racket modules already specify their implementation language

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Adding `#lang typed/racket` is easy

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

`#lang racket`

Typed Racket satisfies "incrementality" **at the module level**

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

Typed Racket satisfies "incrementality" **at the module level**

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

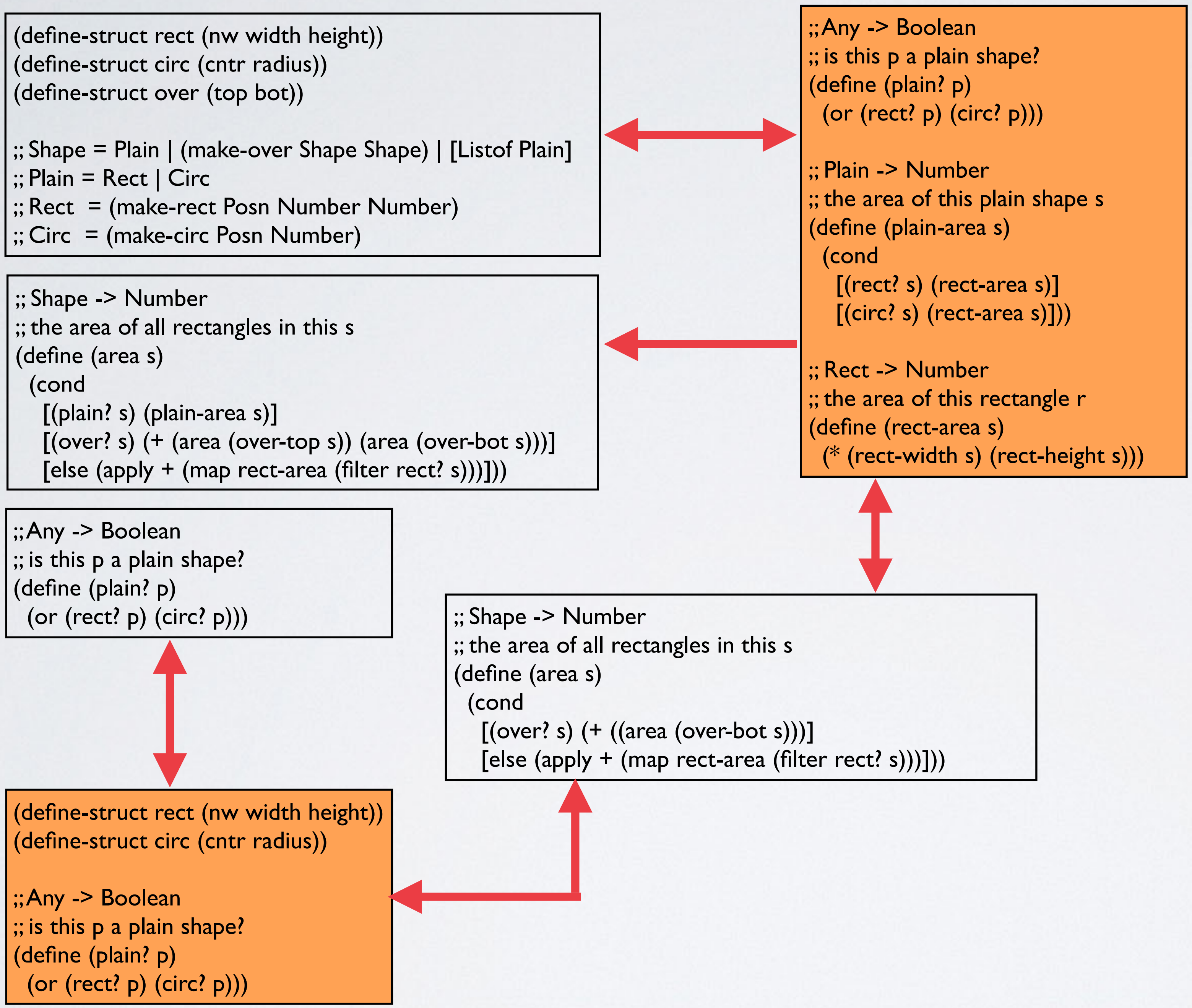
```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

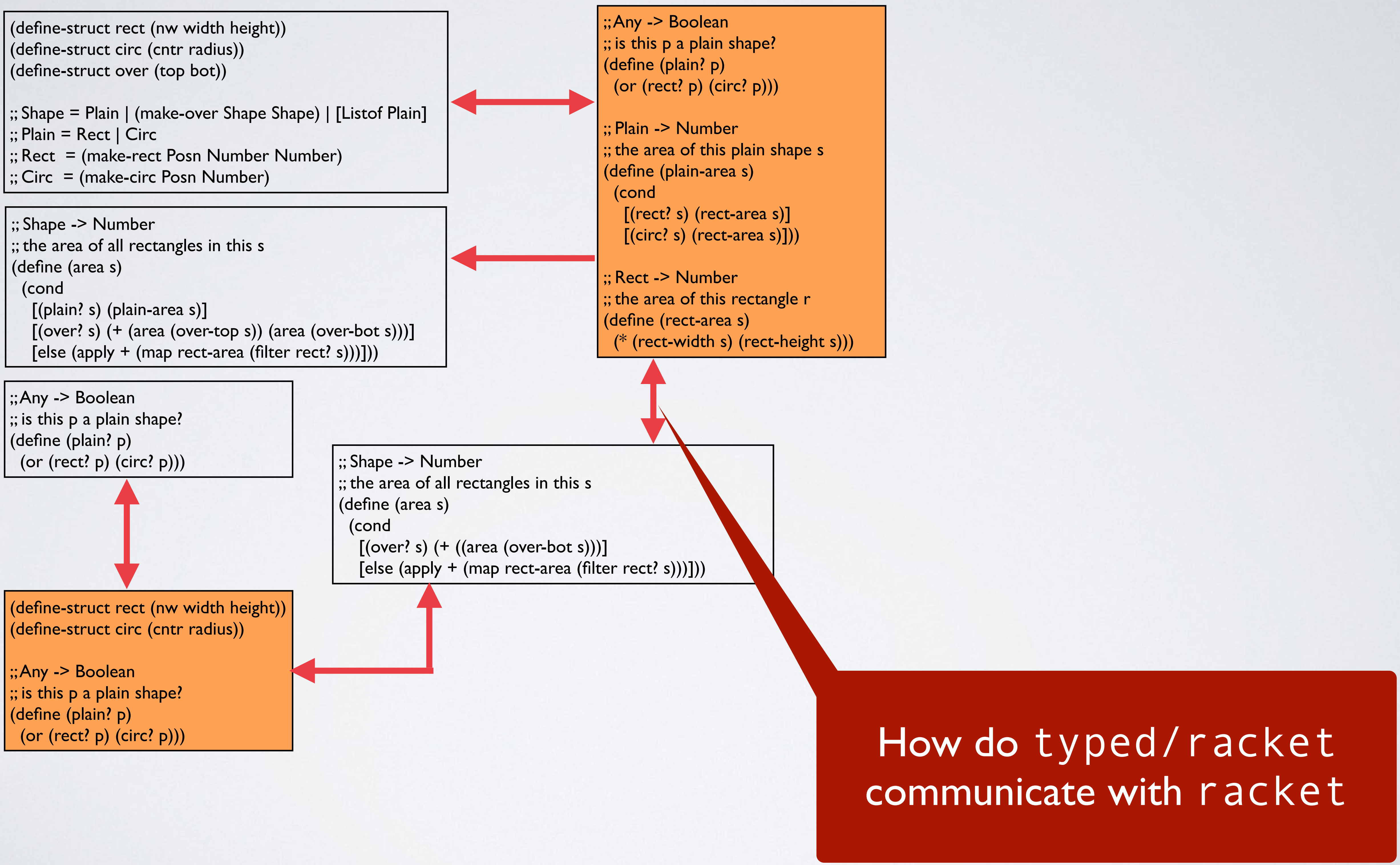
```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

`#lang typed/racket`

Typed Racket satisfies "incrementality" at the module level



Typed Racket satisfies "incrementality" at the module level



Typed Racket satisfies "incrementality" at the module level

```
[Integer -> Integer]
->
Integer
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ?

;; Plain -> N    ber
;; the ar      of this plain shape s
(def   e (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(plain? s) (plain-area s)]
    [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

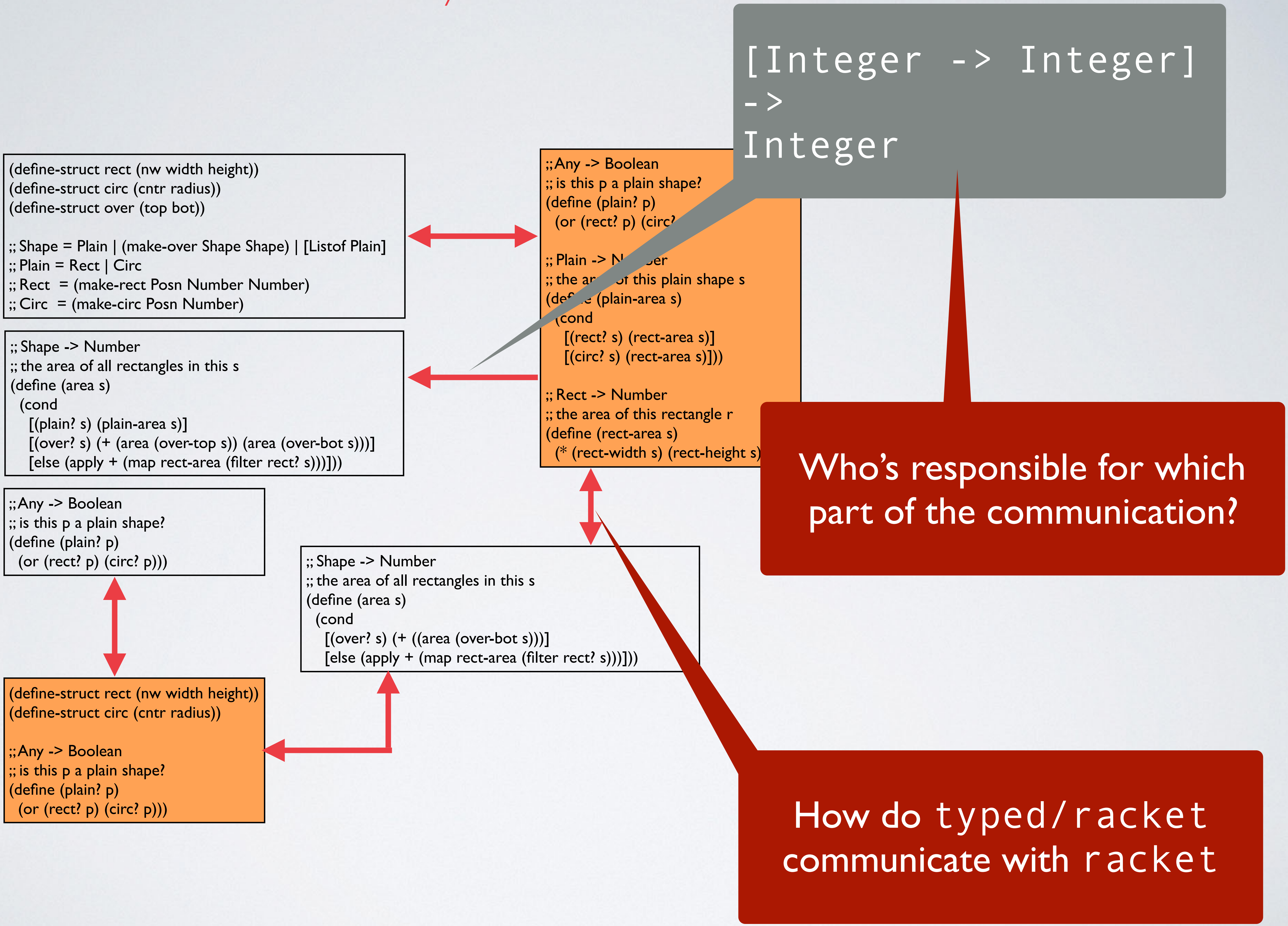
```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ ((area (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

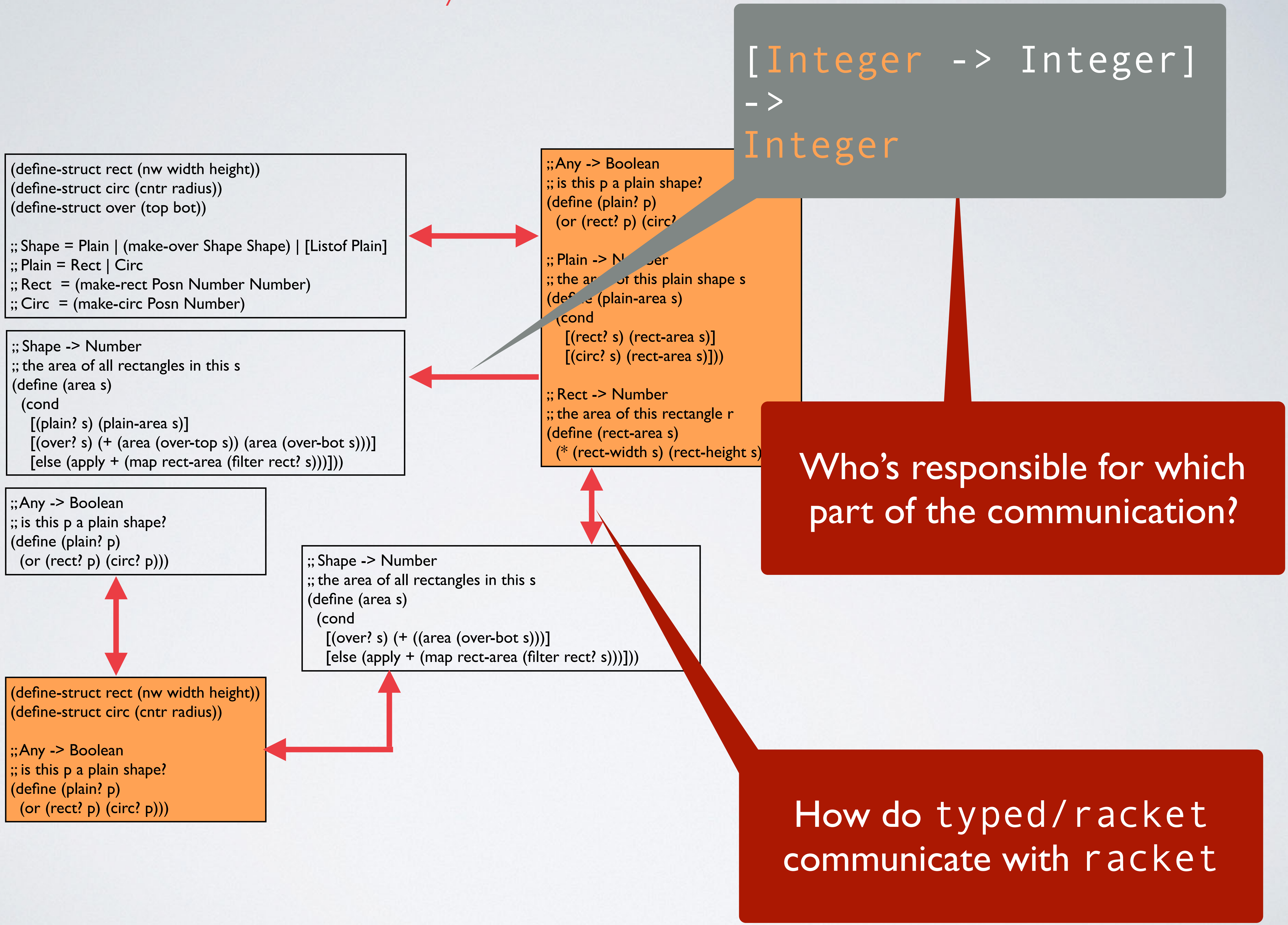
```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

How do typed/racket communicate with racket

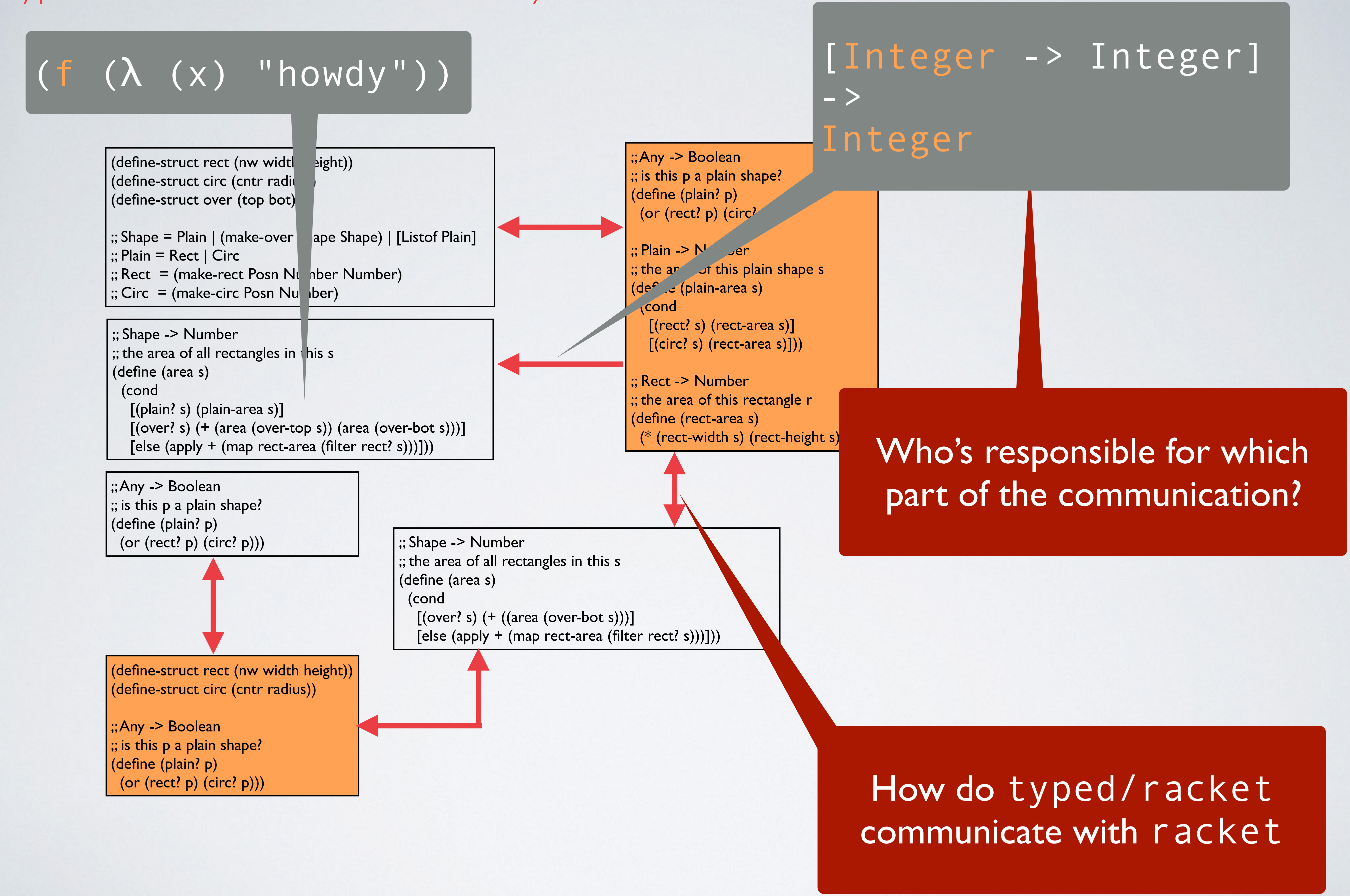
Typed Racket satisfies "incrementality" at the module level



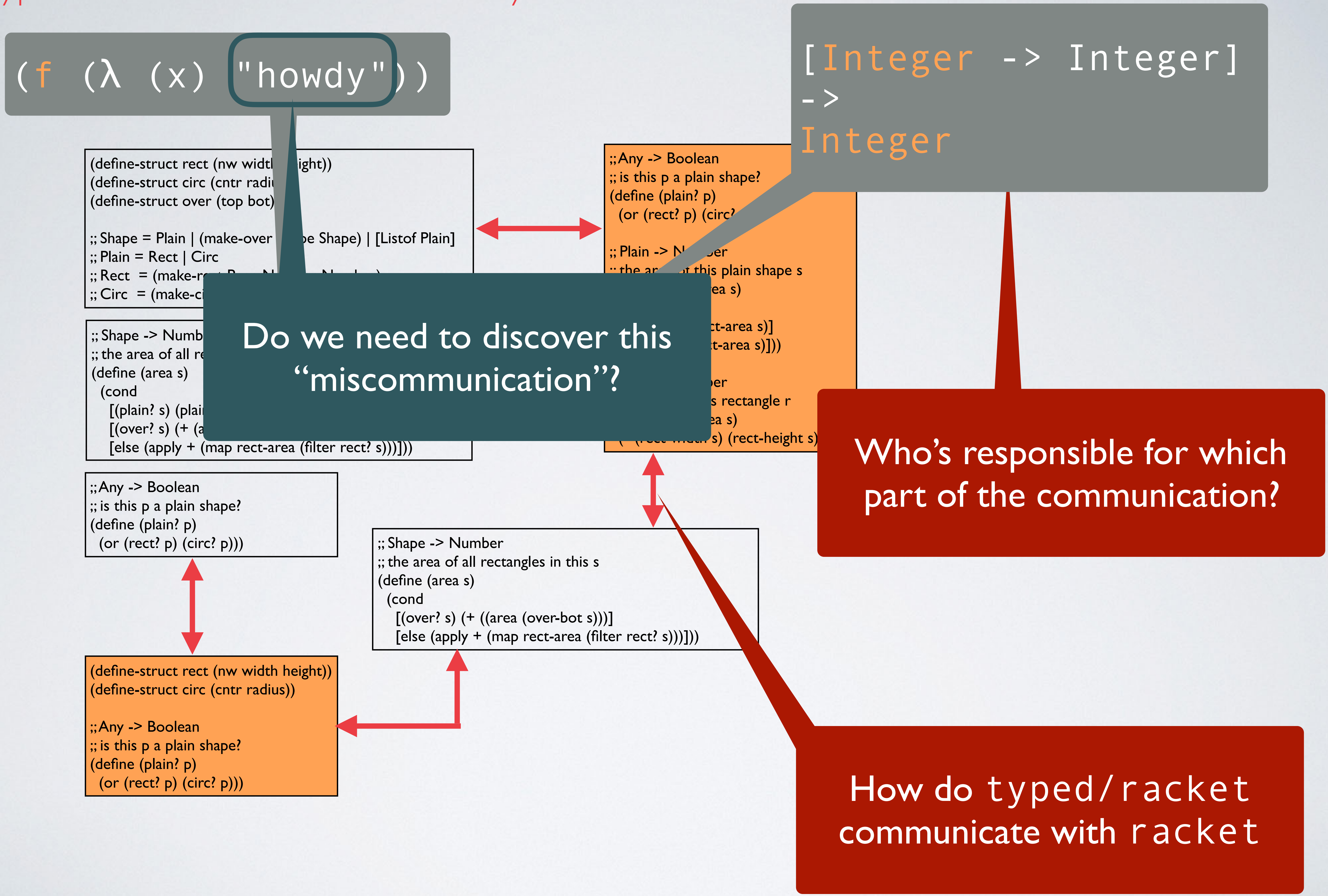
Typed Racket satisfies "incrementality" at the module level



Typed Racket satisfies "incrementality" at the module level



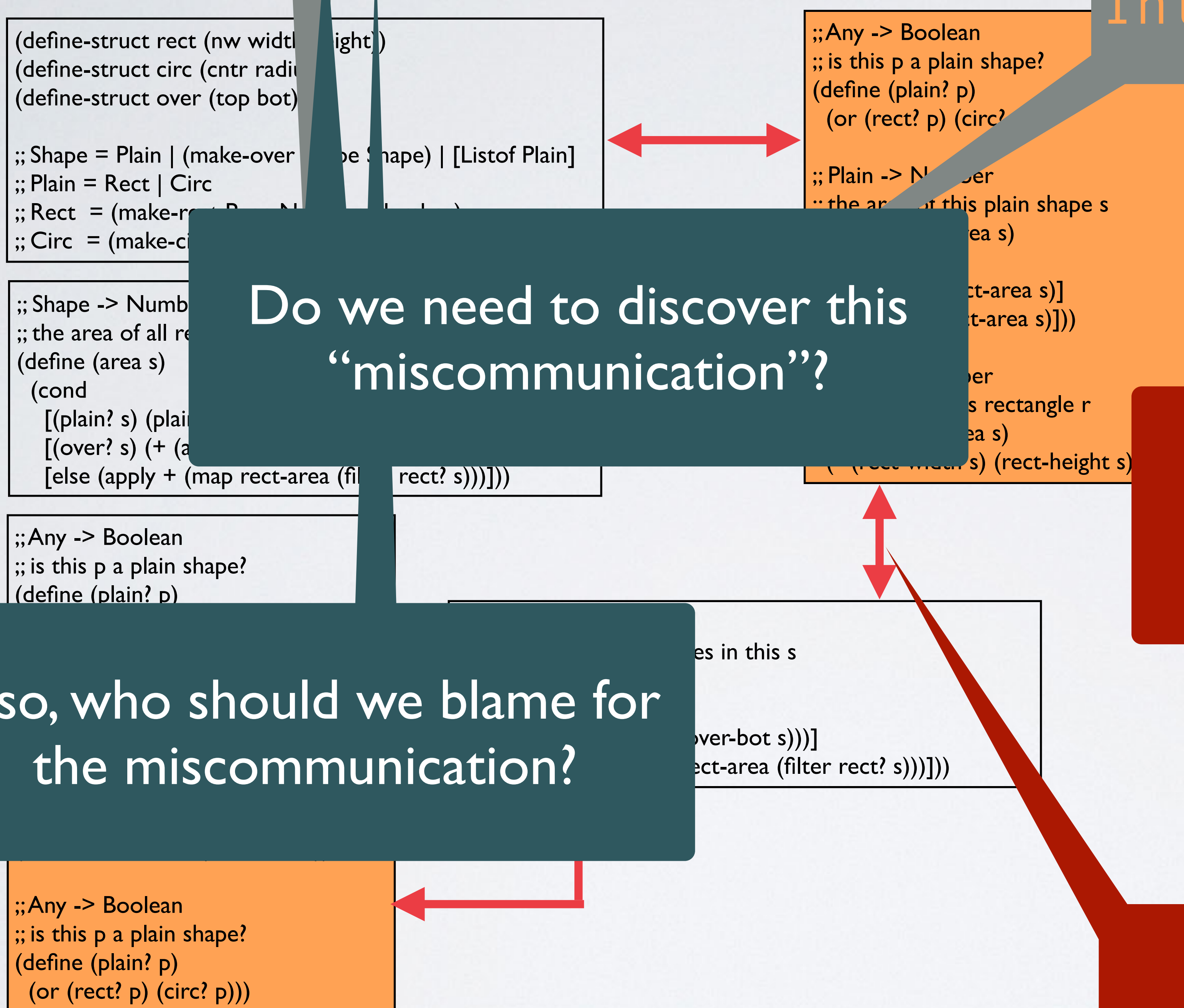
Typed Racket satisfies "incrementality" at the module level



Typed Racket satisfies "incrementality" at the module level

```
(f (λ (x) "howdy"))
```

```
[Integer -> Integer]
->
Integer
```



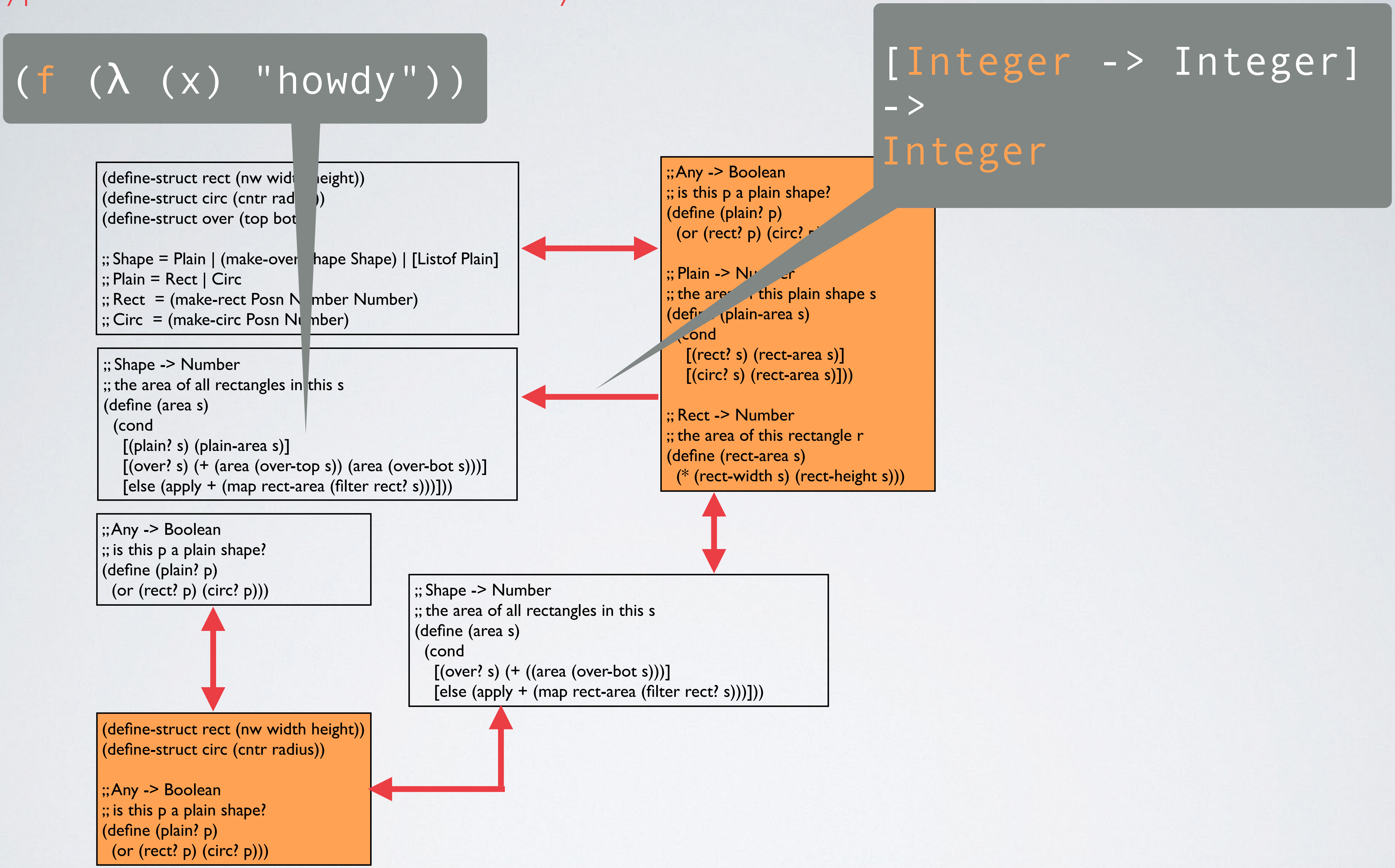
Do we need to discover this "miscommunication"?

If so, who should we blame for the miscommunication?

Who's responsible for which part of the communication?

How do `typed/racket` communicate with `racket`

Typed Racket satisfies "incrementality" at the module level



Typed Racket satisfies "incrementality" at the module level

```
(f (λ (x) "howdy"))
```

```
[Integer -> Integer]
->
Integer
```

```
(define-struct rect (nw wid   eigh  ))
(define-struct circ (cntr rad   ))
(define-struct over (top bot

;; Shape = Plain | (make-over  hape   hape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn N  mber  Number)
;; Circ  = (make-circ Posn N  mber
```

```
;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? 
 
;; Plain -> Nu    er
;; the are     this plain shape s
(defi     (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

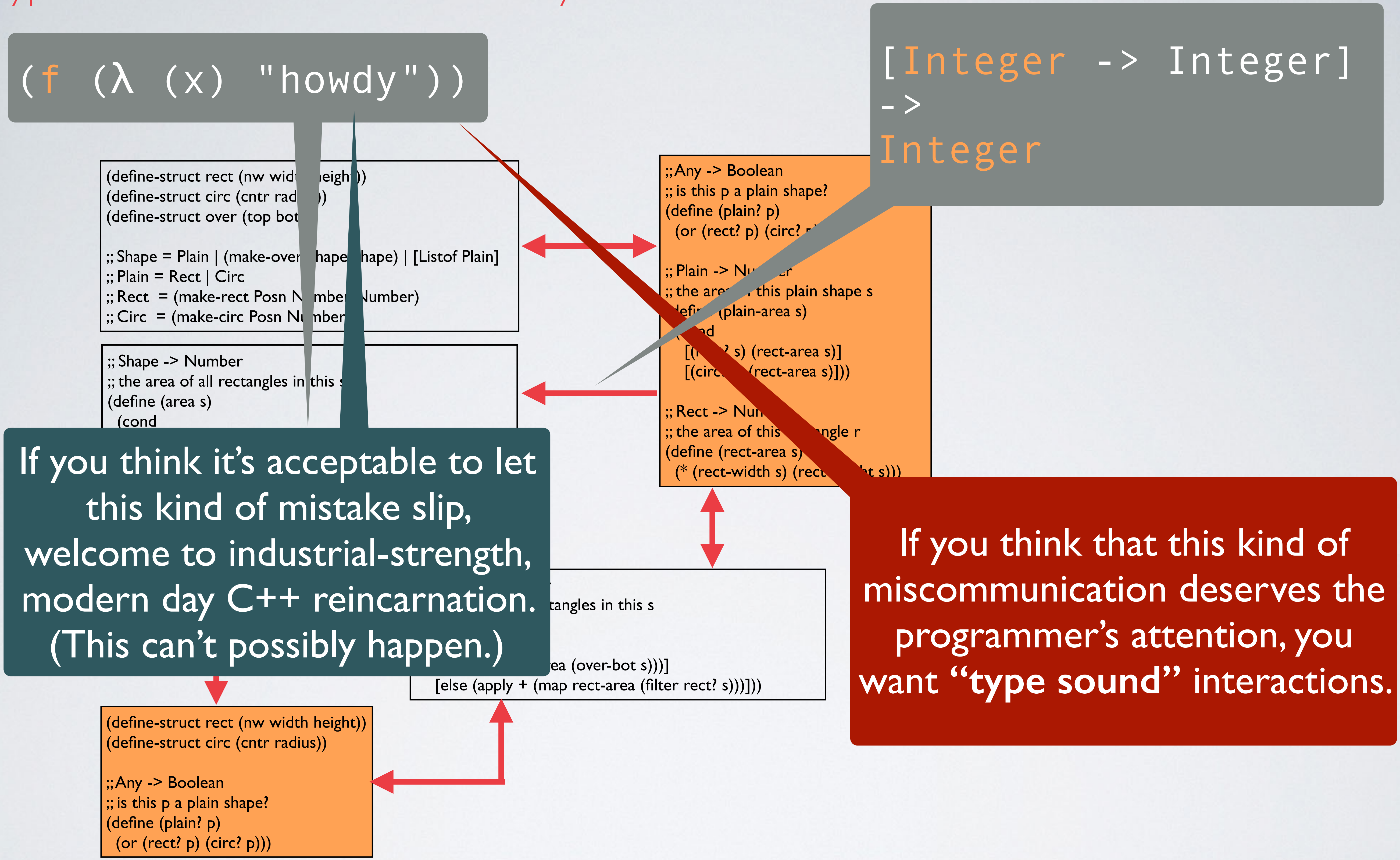
```
;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
```

If you think it's acceptable to let this kind of mistake slip, welcome to industrial-strength, modern day C++ reincarnation. (This can't possibly happen.)

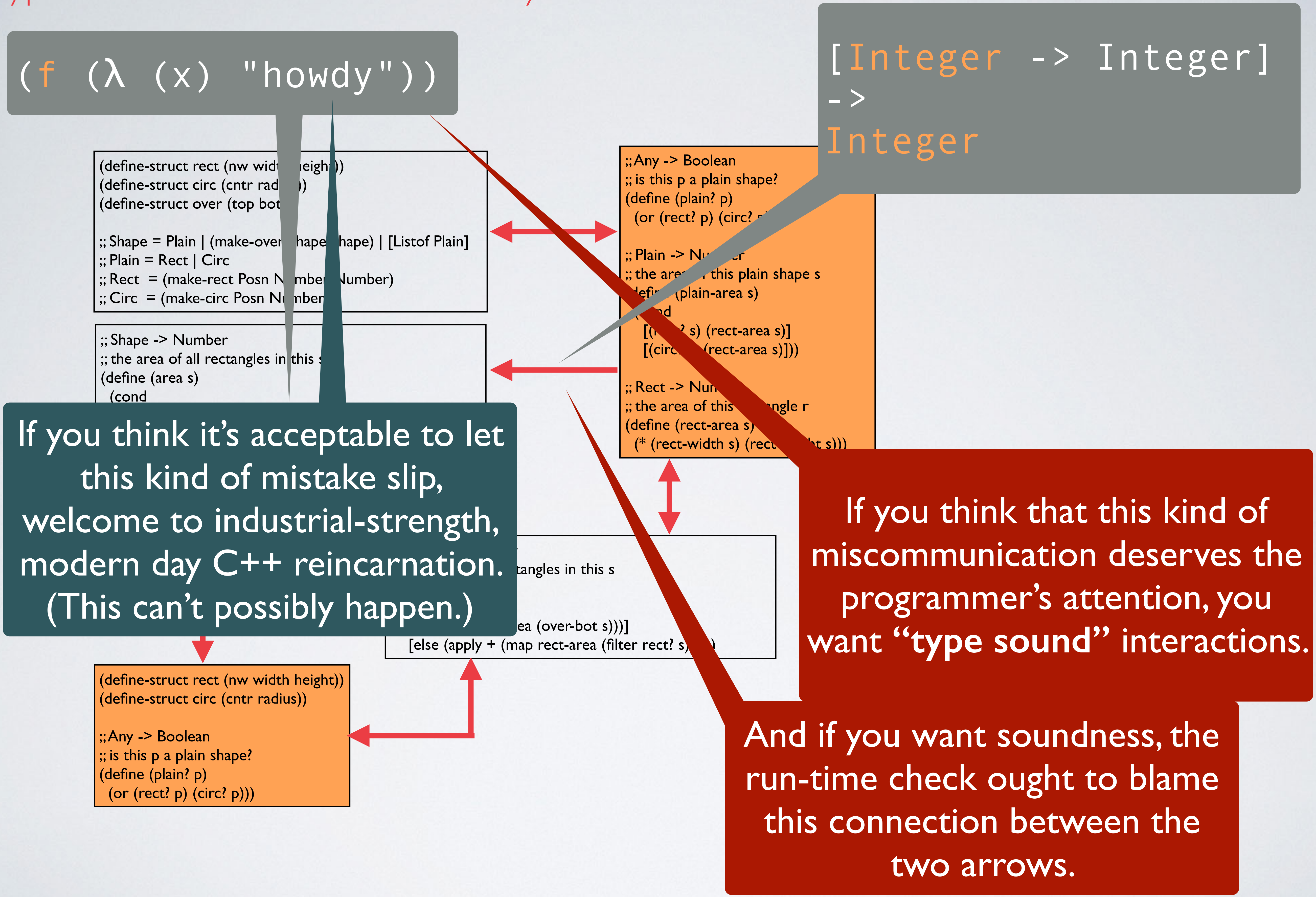
```
tangles in this s

ea (over-bot s)))]
    [else (apply + (map rect-area (filter rect? s)))]))
```

```
(define-struct rect (nw width height))
(define-struct circ (cntr radius))

;;Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

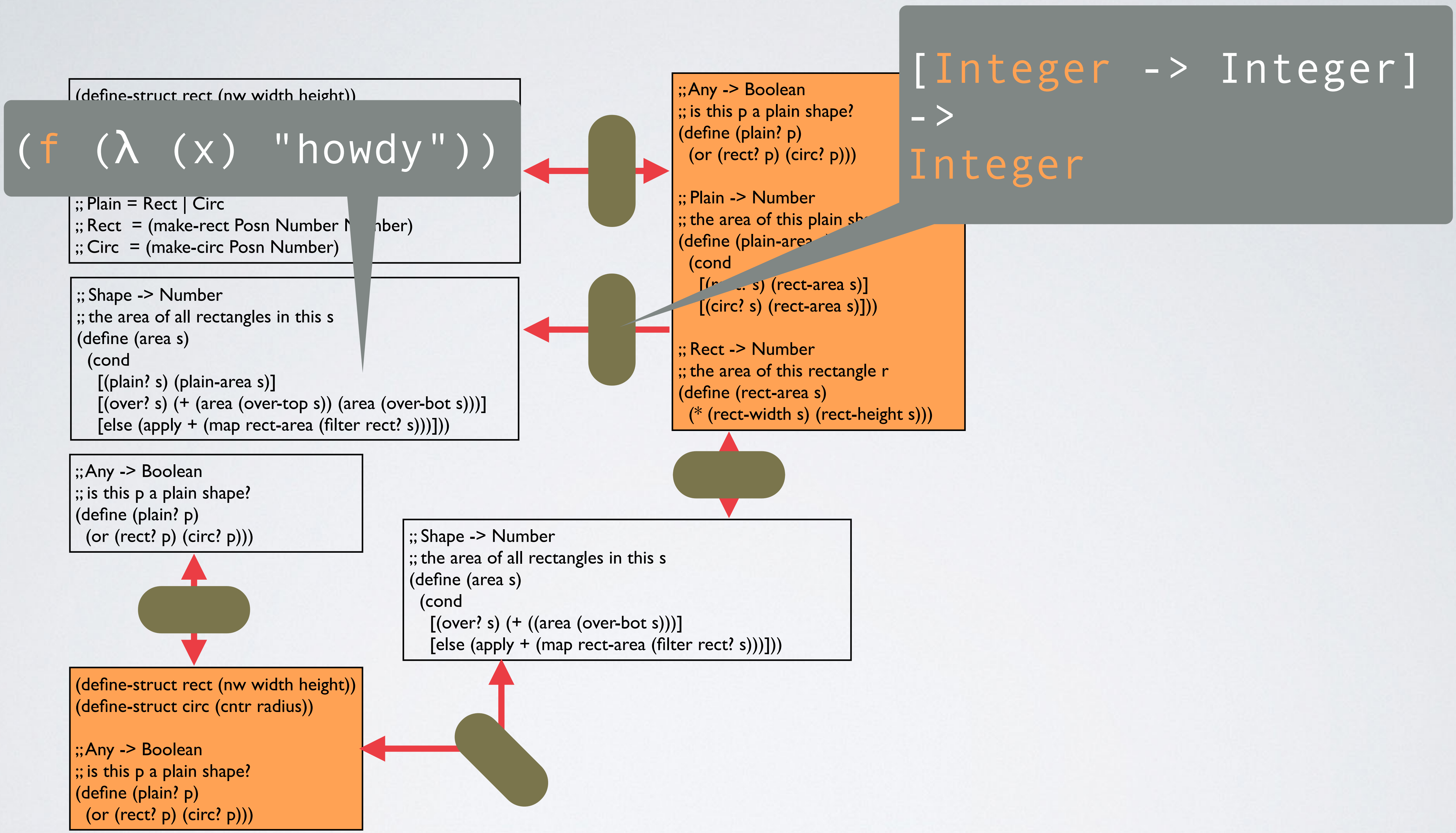
Typed Racket satisfies "incrementality" at the module level



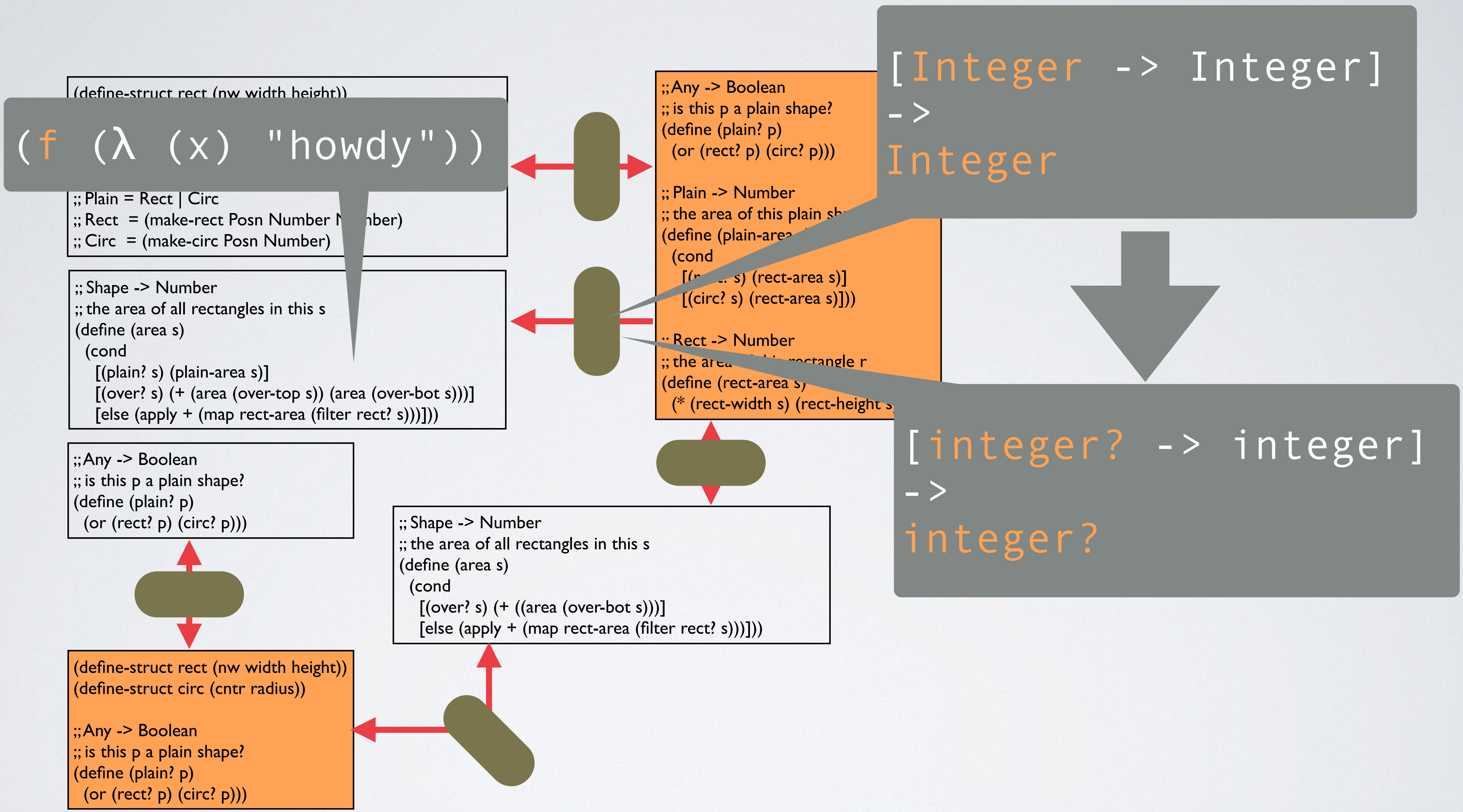
Typed Racket satisfies "incrementality" at the module level



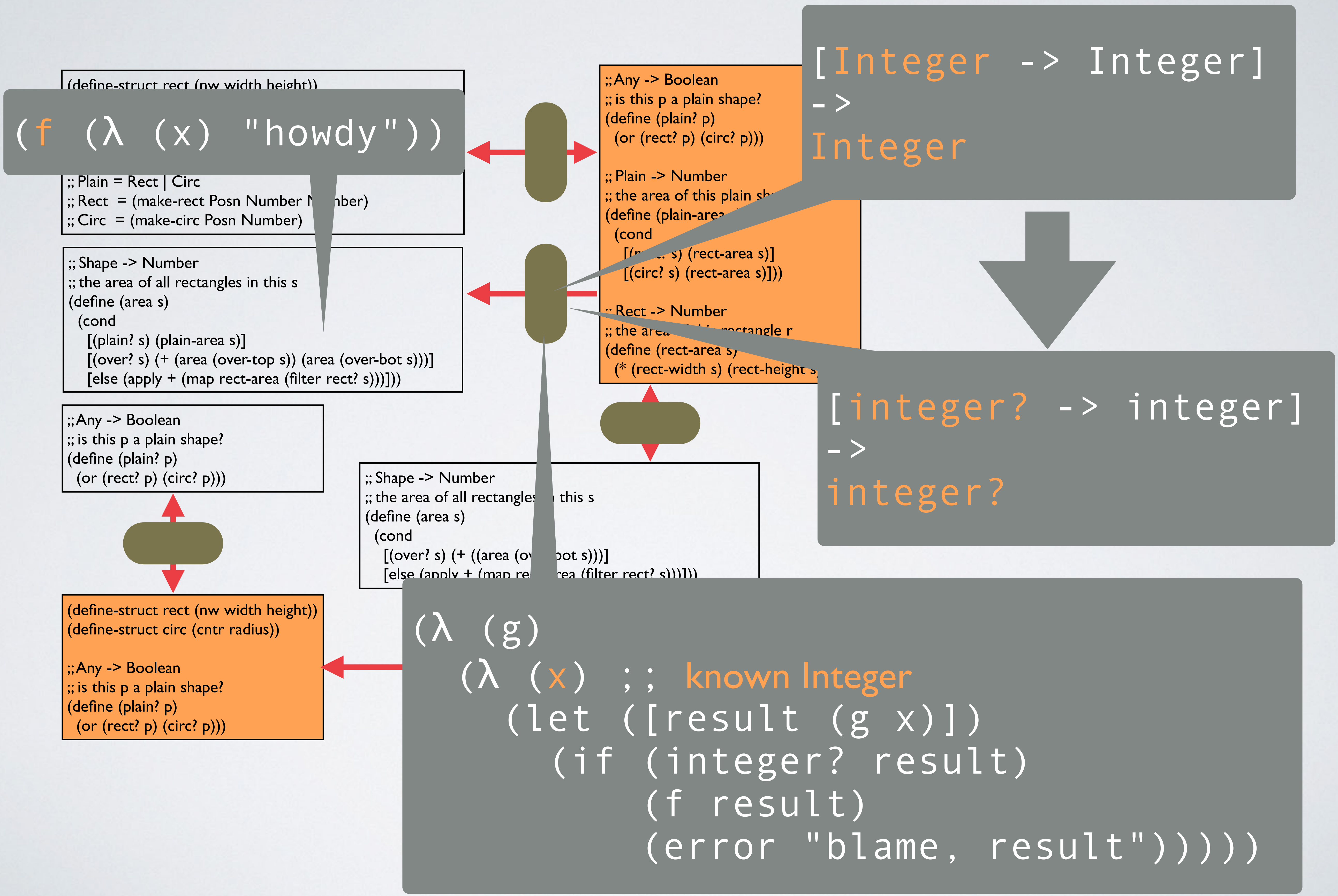
Typed Racket satisfies "**soundness**" at the module levels **via the compilation of types to higher-order contracts**



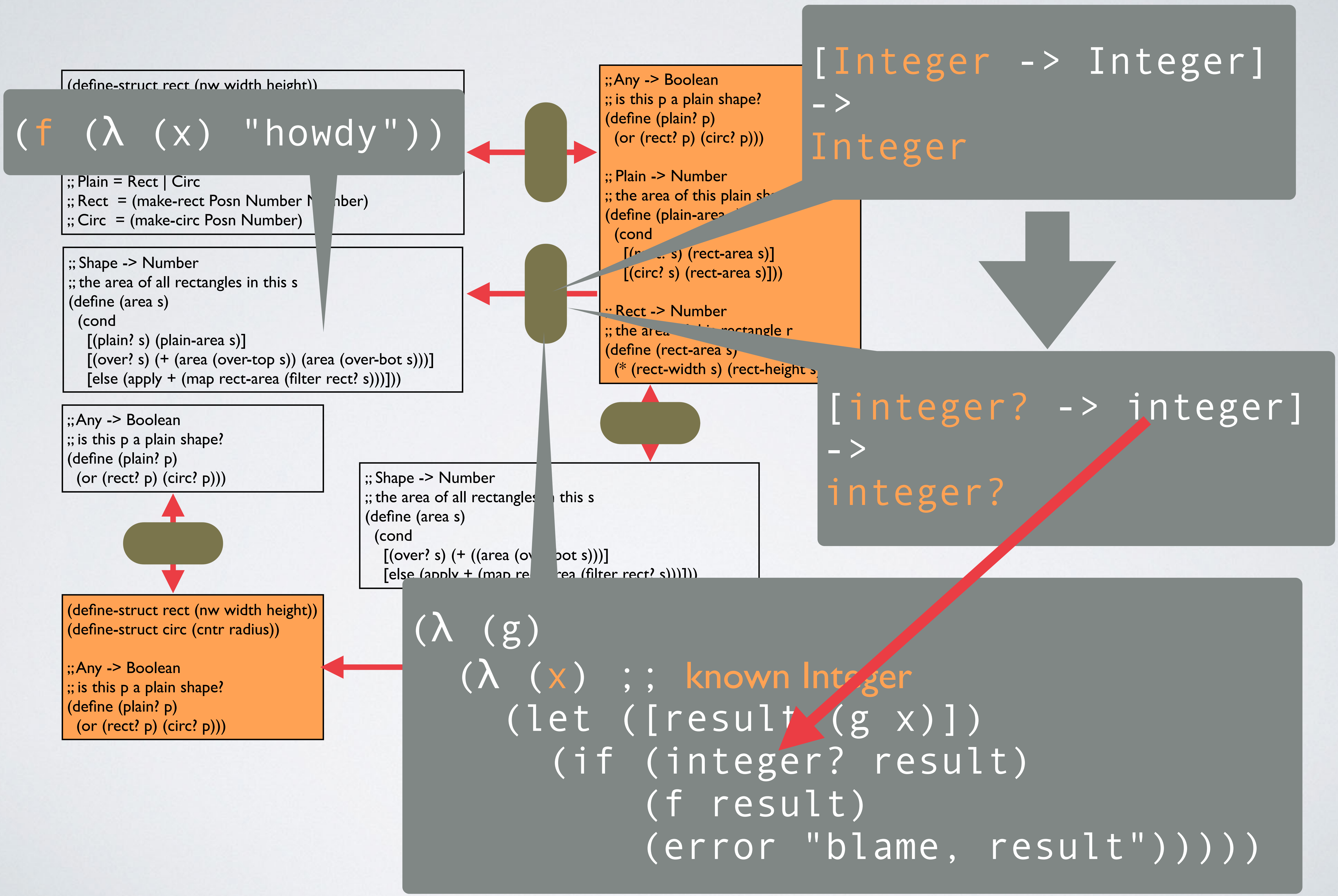
Typed Racket satisfies “**soundness**” at the module levels **via the compilation of types to higher-order contracts**



Typed Racket satisfies "**soundness**" at the module levels **via the compilation of types to higher-order contracts**



Typed Racket satisfies "**soundness**" at the module levels **via the compilation of types to higher-order contracts**



Findler introduced higher-order contracts [ICFP 2002]

Robby Findler

Dimoulas developed *elegant, flexible* technique for proving the soundness of mixed systems [ESOP 2012]

Christos Dimoulas

Theorem

For all mixed programs e ∈ Racket ⊕ Type Racket, one of these statements holds:

- eval(e) is a value
- eval(e) is a known exception from TR
- eval(e) is a contract error blaming a specific boundary between a typed and an untyped module
- eval(e) diverges.

Typed Racket can cope with (almost) all linguistic constructs from Racket

Asumu Takikawa

```
#lang racket
;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

add-search%

```
#lang racket
… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket


Asumu Takikawa

```
#lang racket

;; a mixing tha  adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

a function from class to class

add-search%

```
#lang racket

… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket


Asumu Takikawa

a function from class to class

```
#lang racket
;; a mixing tha  adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

exported …

add-search%

```
#lang racket
… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

Asumu Takikawa

```
#lang racket

;; a mixing tha  adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

a function from class to class

exported …

add-search%

… and used in a separate module

```
#lang racket

… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket


Asumu Takikawa

```
#lang racket
;; a mixing tha  adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

a function from class to class

Yes, this is real-world code.

exported …

add-search%

… and used in a separate module

```
#lang racket
… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

Asumu Takikawa

```
#lang racket
;; a mixing tha  adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

a function from class to class

Yes, this is real-world code.

Yes, you can do this is Python, too.

add-search%

… and used in a separate module

```
#lang racket
… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

```
#lang typed/racket

;; a mixing that adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

add-search%

```
#lang racket

… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

What kind of types do classes have?

```
#lang typed/racket

;; a mixing that adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

add-search%

```
#lang racket

… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

What kind of types do classes have?

```
#lang typed/racket

;; a mixing that adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

What contracts do these types compile to?

add-search%

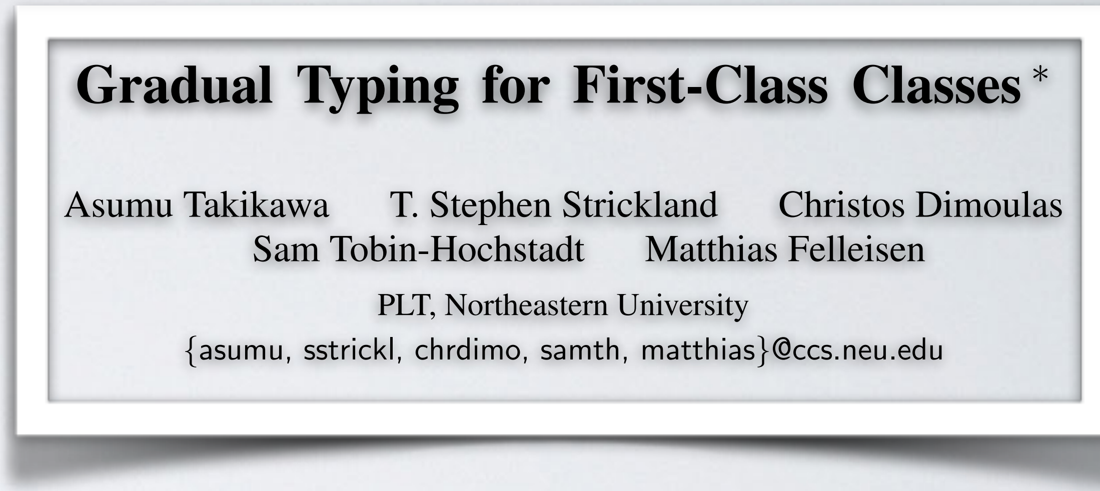
```
#lang racket

… (add-search analysis-presentation%)…
```

Typed Racket can cope with (almost) all linguistic constructs from Racket

```
#lang typed/racket

;; a mixing that adds search capabilities

(define (add-search %)

  (class %

    (inherit text)

    (field [state #f])

    (define/public (search str)

      ...)))
```

add-search%

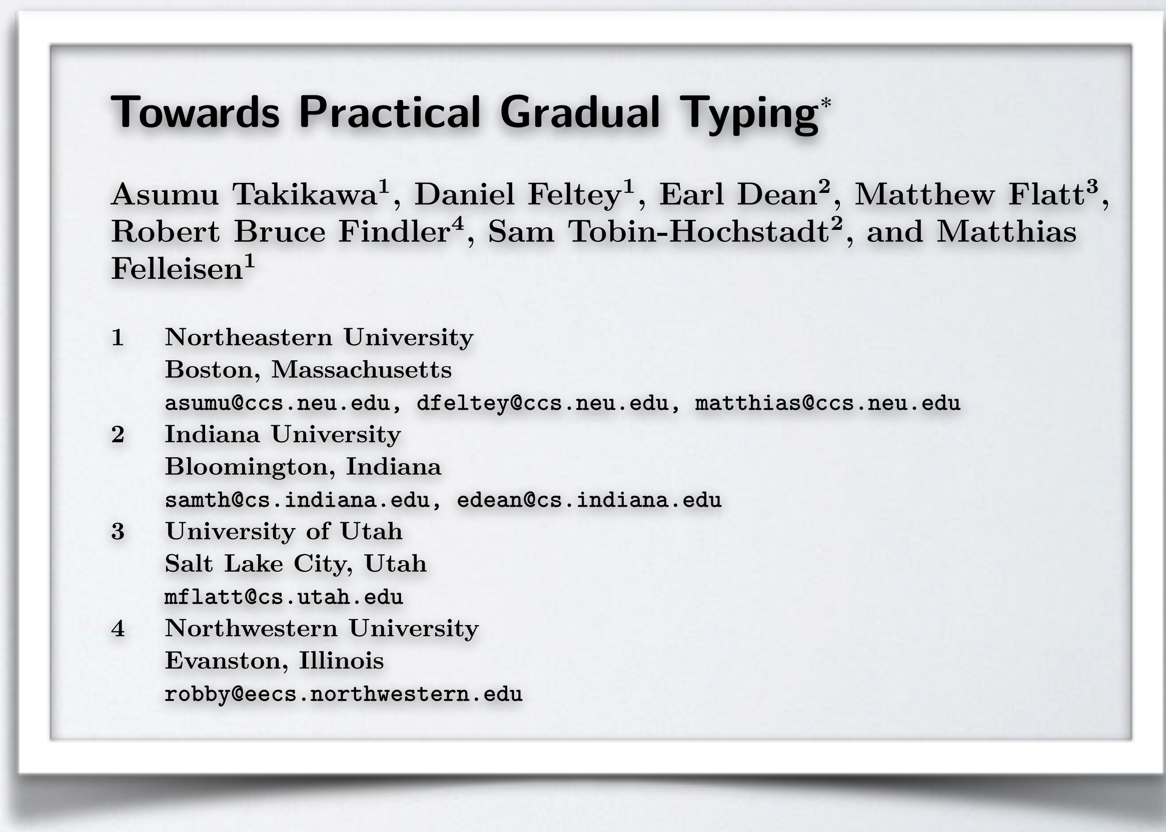
```
#lang racket

… (add-search analysis-presentation%)…
```



Gradual Typing for First-Class Classes *

Asumu Takikawa T. Stephen Strickland Christos Dimoulas
Sam Tobin-Hochstadt Matthias Felleisen

PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu



Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias@ccs.neu.edu
2 Indiana University
Bloomington, Indiana
samth@cs.indiana.edu, edean@cs.indiana.edu
3 University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu
4 Northwestern University
Evanston, Illinois
robby@eecs.northwestern.edu

Typed Racket can cope with (almost) all linguistic constructs from Racket

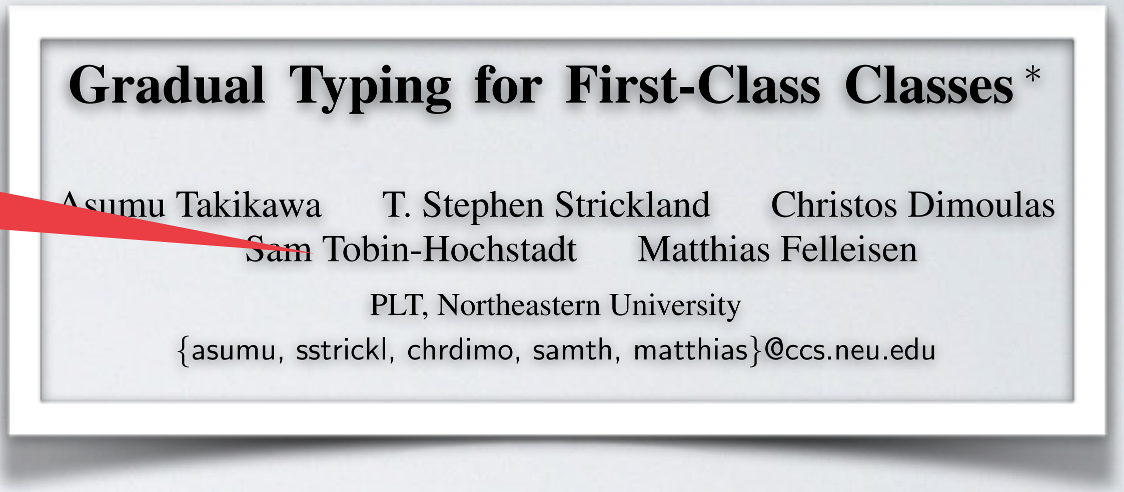
Innovations needed:

- class types, with row polymorphism
- sealing contracts for enforce polymorphism
- innovative soundness proof

```
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

add-search%

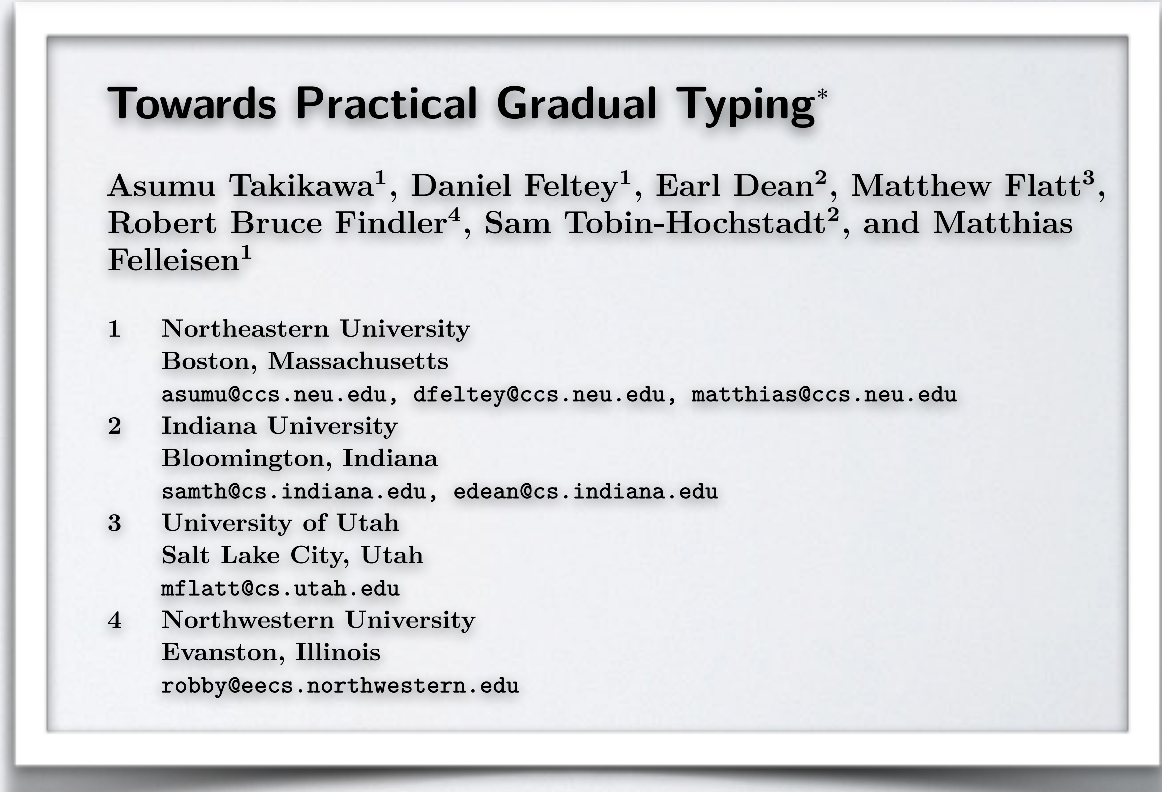
```
#lang racket
… (add-search analysis-presentation%)…
```



Gradual Typing for First-Class Classes *

Asumu Takikawa T. Stephen Strickland Christos Dimoulas
Sam Tobin-Hochstadt Matthias Felleisen

PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu



Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias@ccs.neu.edu

2 Indiana University
Bloomington, Indiana
samth@cs.indiana.edu, edean@cs.indiana.edu

3 University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu

4 Northwestern University
Evanston, Illinois
robby@eecs.northwestern.edu

Typed Racket can cope with (almost) all linguistic constructs from Racket

Innovations needed:

- class types, with row polymorphism
- sealing contracts for enforce polymorphism
- innovative soundness proof

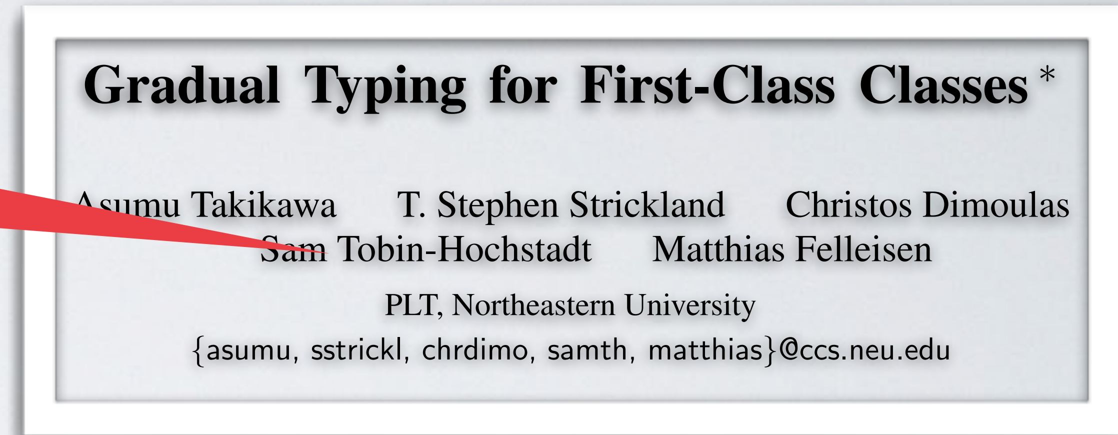
```
(define (add-search %)

  (class %

      ...)))
```

Translating theory into practice:

- design for usability
- implementation engineering
- performance evaluation

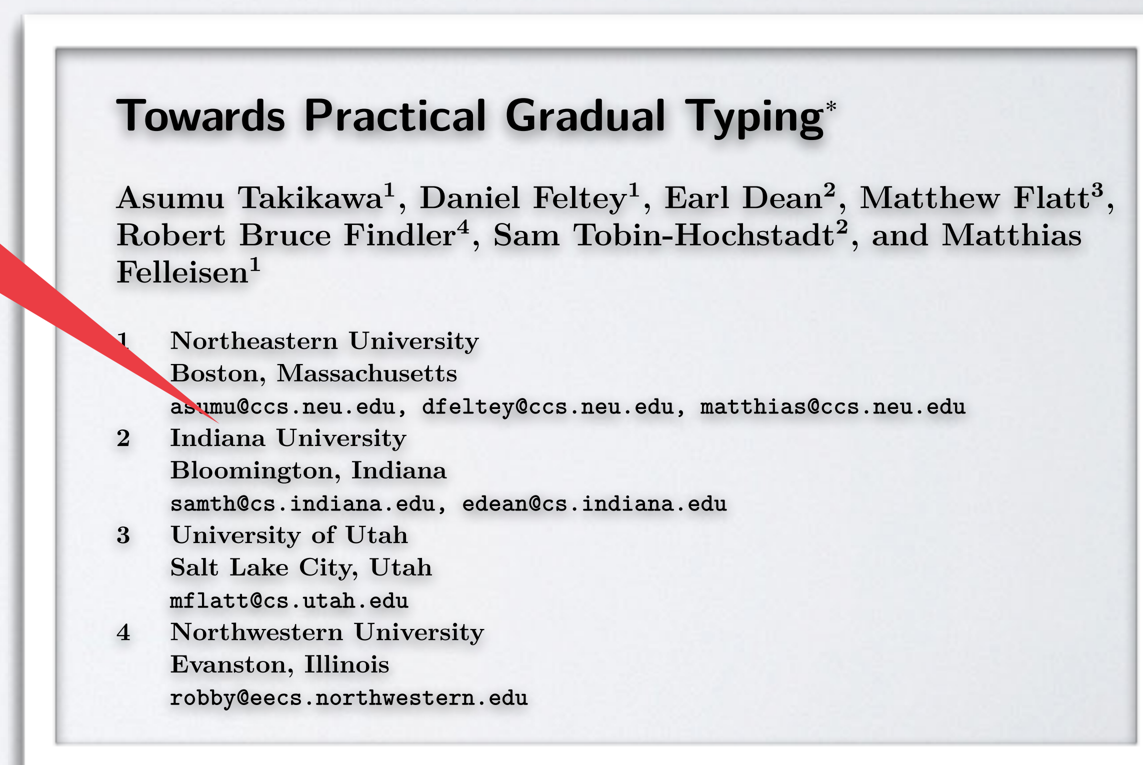
add-search%

```
#lang racket

… (add-search analysis-presentation%)…
```



Gradual Typing for First-Class Classes *

Asumu Takikawa   T. Stephen Strickland   Christos Dimoulas
Sam Tobin-Hochstadt   Matthias Felleisen
PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu



Towards Practical Gradual Typing*

Asumu Takikawa[1], Daniel Feltey[1], Earl Dean[2], Matthew Flatt[3], Robert Bruce Findler[4], Sam Tobin-Hochstadt[2], and Matthias Felleisen[1]

1 Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias@ccs.neu.edu
2 Indiana University
Bloomington, Indiana
samth@cs.indiana.edu, edean@cs.indiana.edu
3 University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu
4 Northwestern University
Evanston, Illinois
robby@eecs.northwestern.edu

Design matters.

- Typed Racket is *incremental*. ✓
- Typed Racket is *idiomatic*. ✓
- Typed Racket is *sound*. ✓
- Typed Racket *covers it all*. ✓

- Does it work?
- Does it really work?
- Truthfully?
- No cheating?

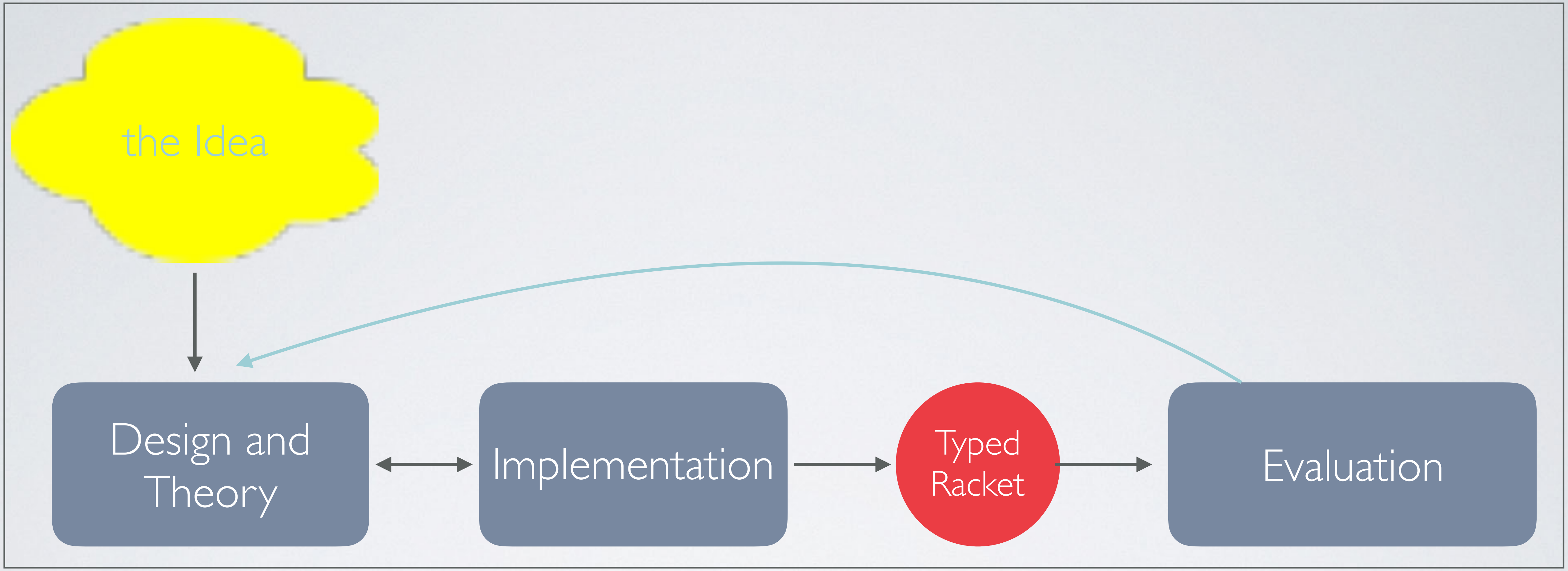
Design matters.

- Typed Racket is *incremental*. ✓
- Typed Racket is *idiomatic*. ✓
- Typed Racket is *sound*. ✓
- Typed Racket *covers it all*. ✓

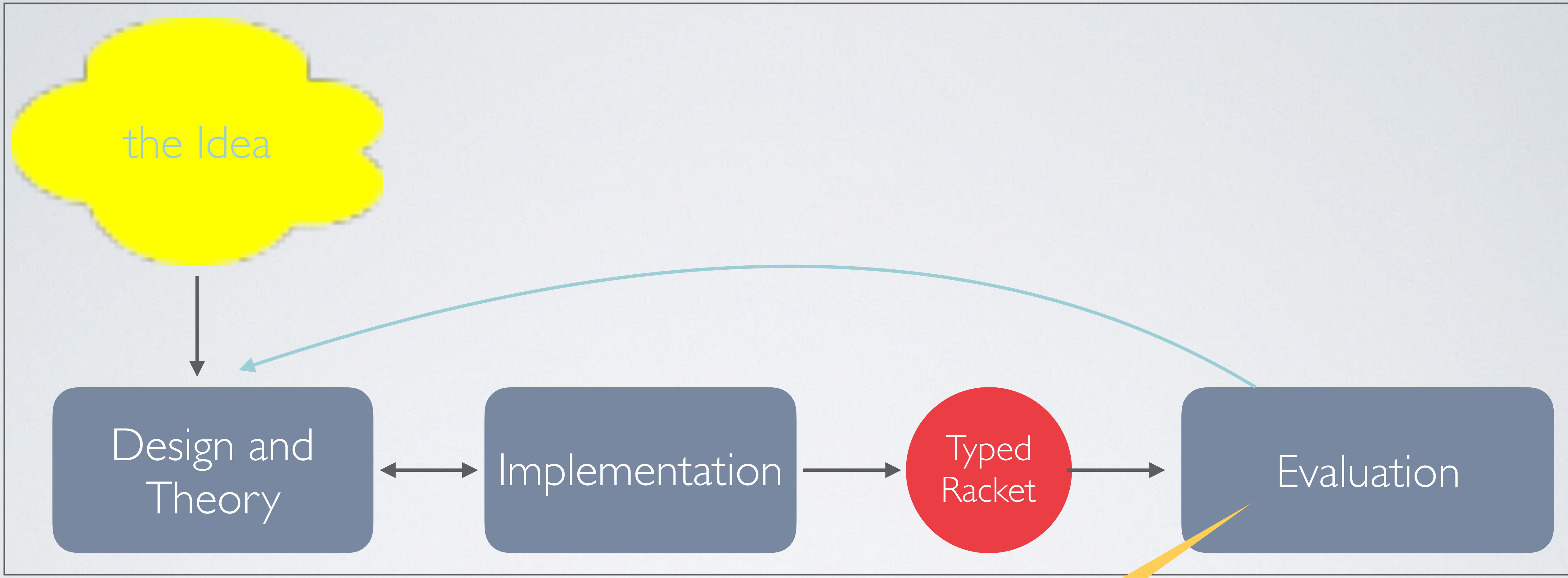
Evaluation matters even more.

- Does it work?
- Does it really work?
- Truthfully?
- No cheating?

Design needs feedback loop.



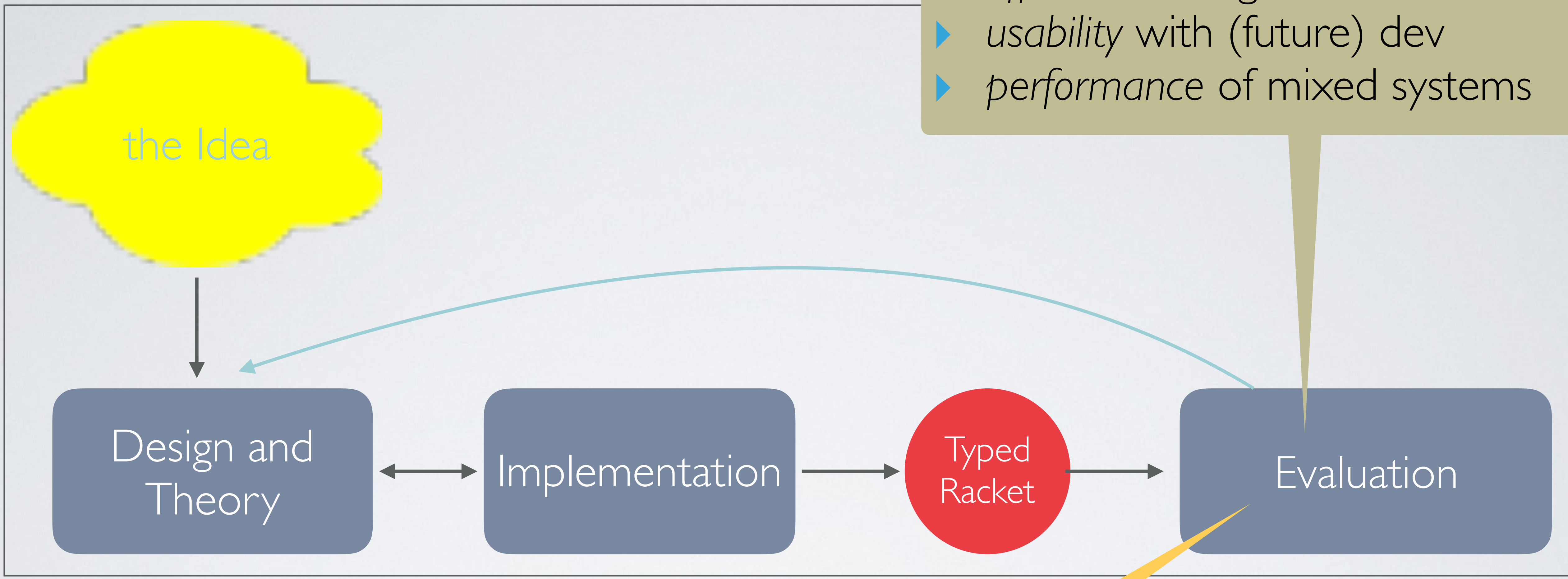
Design needs feedback loop.



Two kinds of evaluation:

- *formative*
- *summative*

Design needs feedback loop.



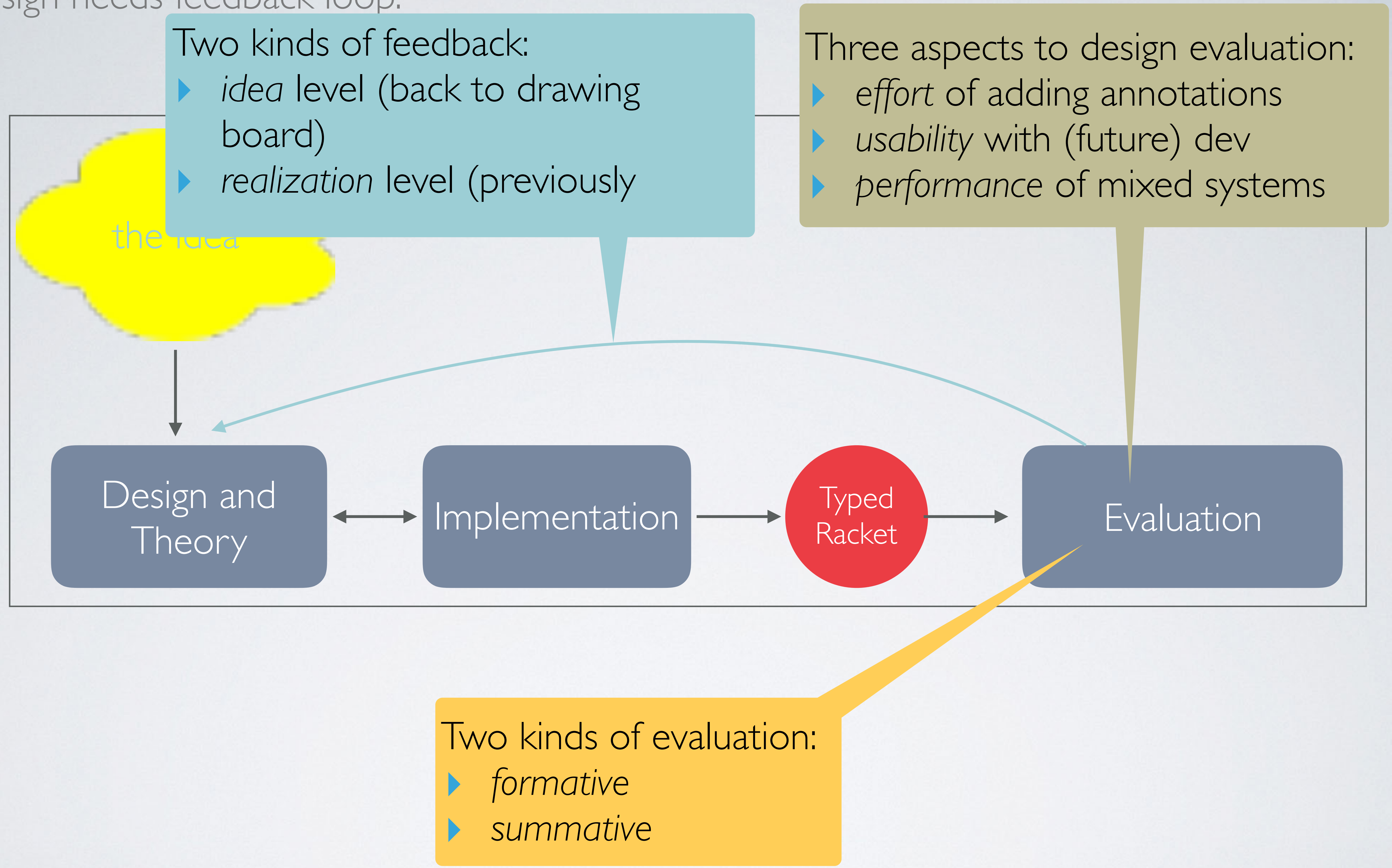
Three aspects to design evaluation:

- *effort* of adding annotations
- *usability* with (future) dev
- *performance* of mixed systems

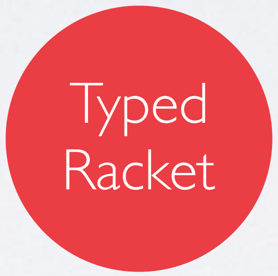
Two kinds of evaluation:

- *formative*
- *summative*

Design needs feedback loop.



Design needs feedback loop.



Design needs feedback loop.

Typed Racket

Effort of adding type annotations:

- FP style calls for **3-5%** changes
- OOP style needs **10-15%** changes
- mostly annotations, some changes to code to get around the type checker

Design needs feedback loop.

Usability of Typed Racket:

- TR devs are easily proficient
- seniors in a PL course
- real-world users

Typed Racket

Effort of adding type annotations:

- FP style calls for **3-5%** changes
- OOP style needs **10-15%** changes
- mostly annotations, some changes to code to get around the type checker

Design needs feedback loop.

Usability of Typed Racket:

- TR devs are easily proficient
- seniors in a PL course
- real-world users

Typed Racket

Performance!

Effort of adding type annotations:

- FP style calls for **3-5%** changes
- OOP style needs **10-15%** changes
- mostly annotations, some changes to code to get around the type checker

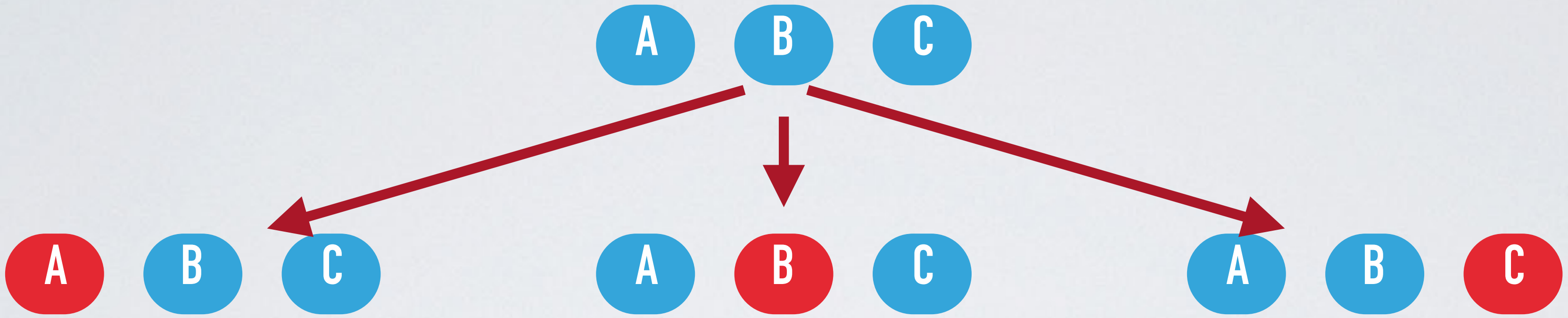
Greenman create and evaluate all possible mixed configurations of existing multi-module systems

A B C

Ben Greenman

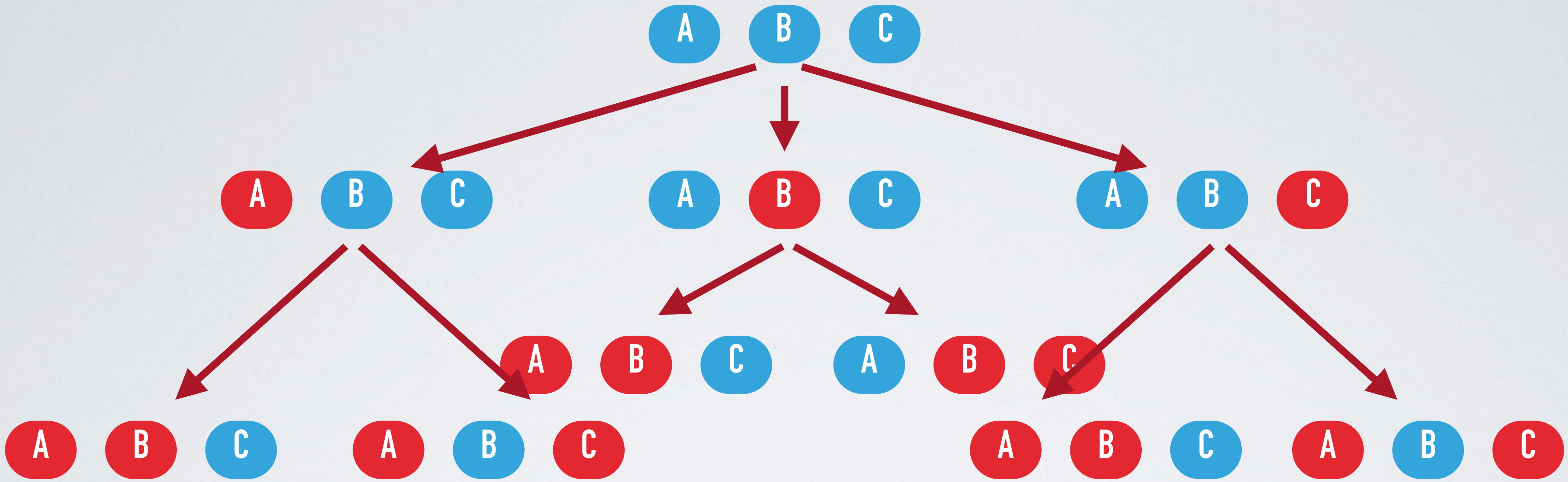
WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Greenman create and evaluate all possible mixed configurations of existing multi-module systems



WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Greenman create and evaluate all possible mixed configurations of existing multi-module systems

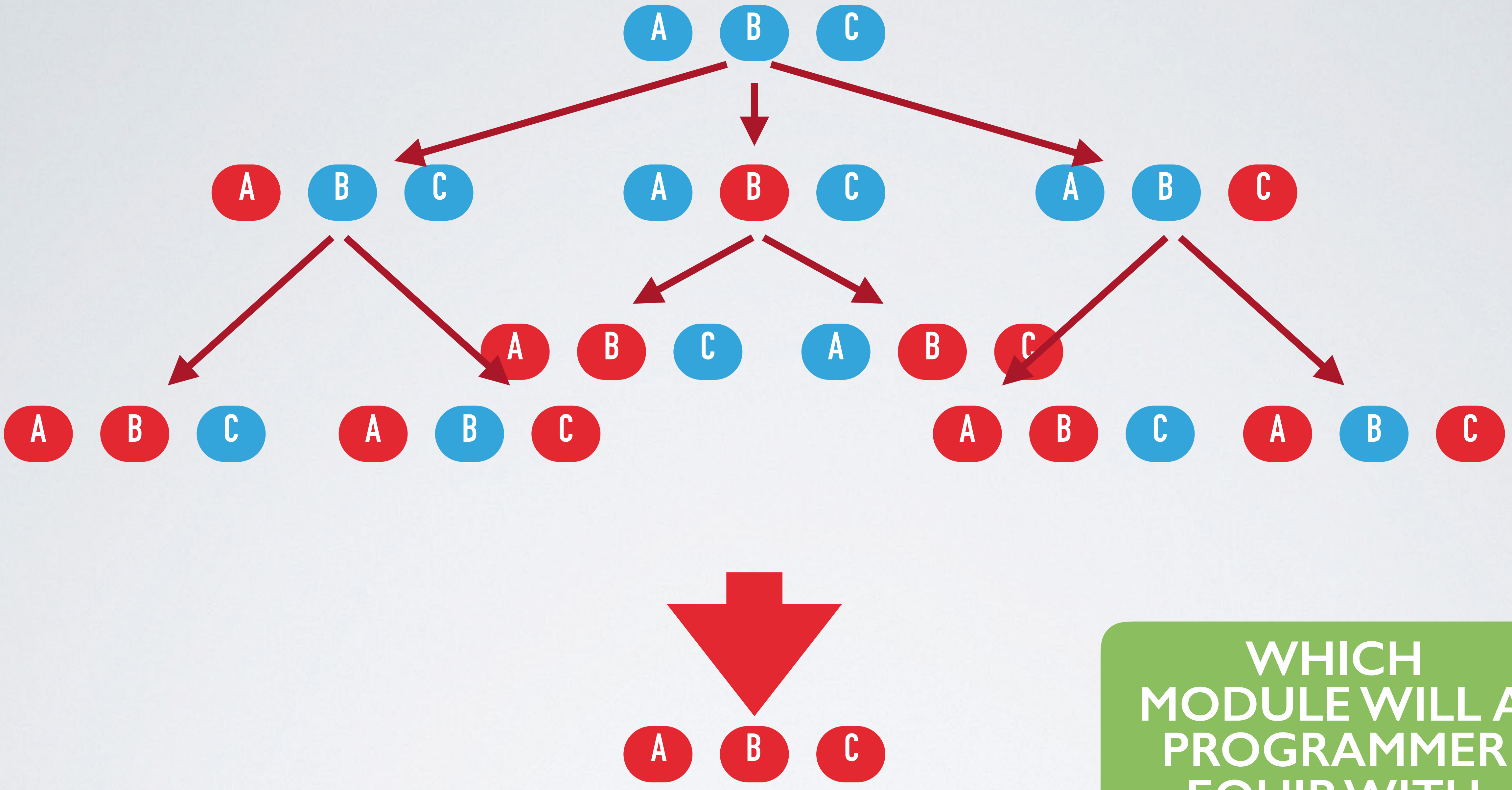


Ben Greenman

WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Greenman create and evaluate all possible mixed configurations of existing multi-module systems

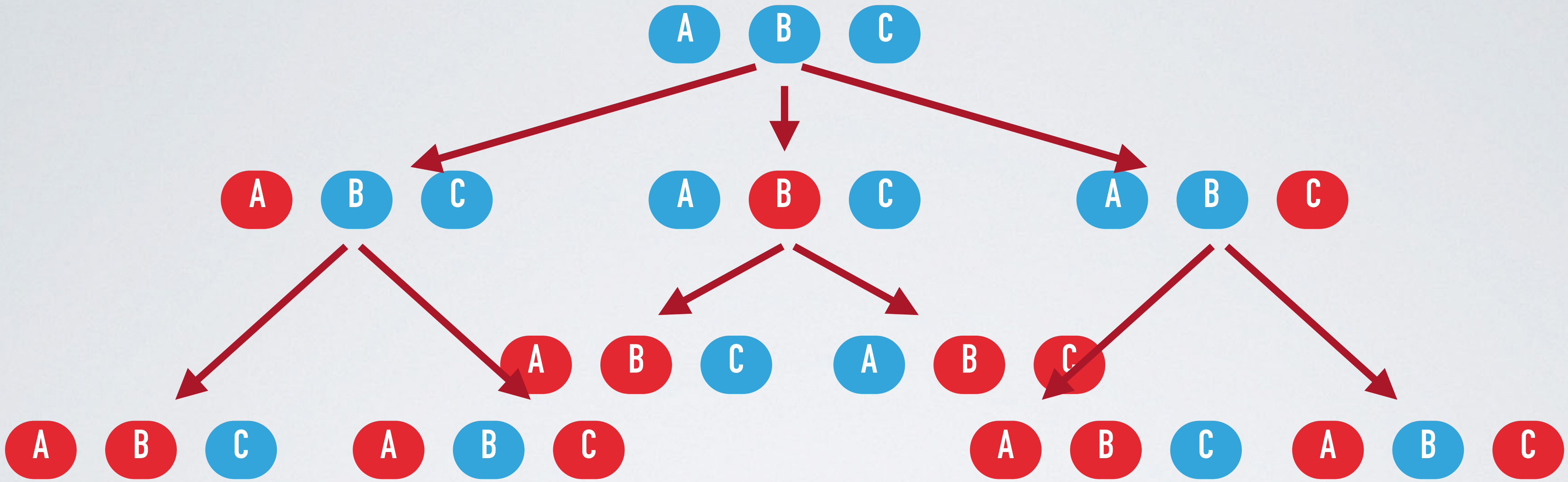
Ben Greenman



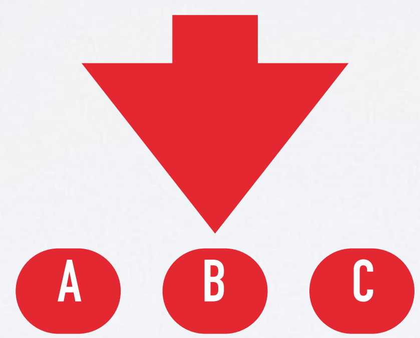
WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Greenman create and evaluate all possible mixed configurations of existing multi-module systems

Ben Greenman



WE DON'T KNOW. ALL 2^N OF THESE CONFIGURATIONS ARE FEASIBLE.



WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Typed Racket's contract impose a high run-time cost on mixed system performance.

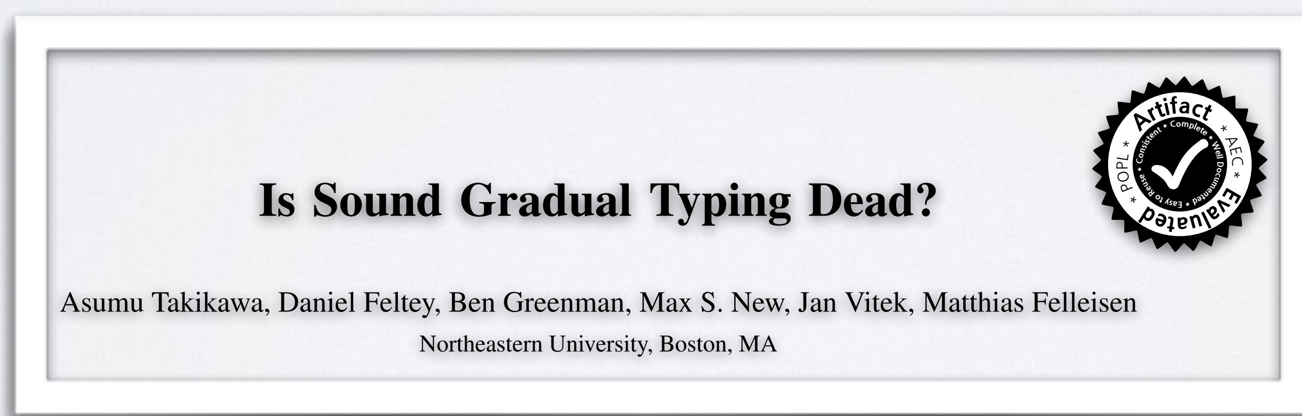
POPL 2016 and Journal of Functional Programming [in preparation]

- ~20 modular programs with ~100,000 configurations.
- 90% of those impose a penalty of 3x or more.
- many configurations impose a 10x penalty
- some configurations cost as much as 100x of the baseline

Typed Racket's contract impose a high run-time cost on mixed system performance.

POPL 2016 and Journal of Functional Programming [in preparation]

- ~20 modular programs with ~100,000 configurations.
- 90% of those impose a penalty of 3x or more.
- many configurations impose a 10x penalty
- some configurations cost as much as 100x of the baseline



Is Sound Gradual Typing Dead?

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen
Northeastern University, Boston, MA

Premature Death?

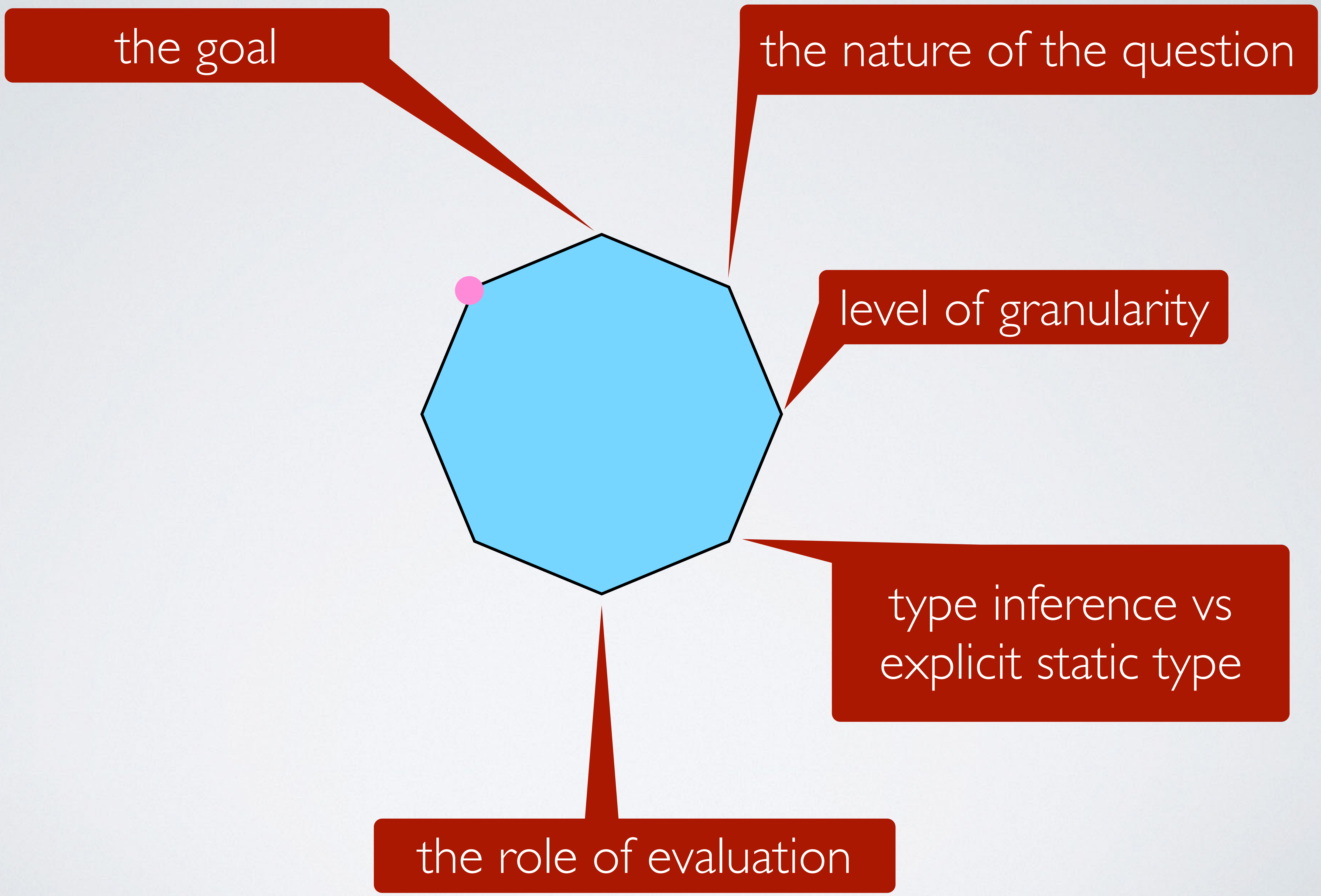
- Practical evaluations are critical for the design feedback loop.
- They focus our mind and our research efforts.

Premature Death? **Research is when it can fail.**

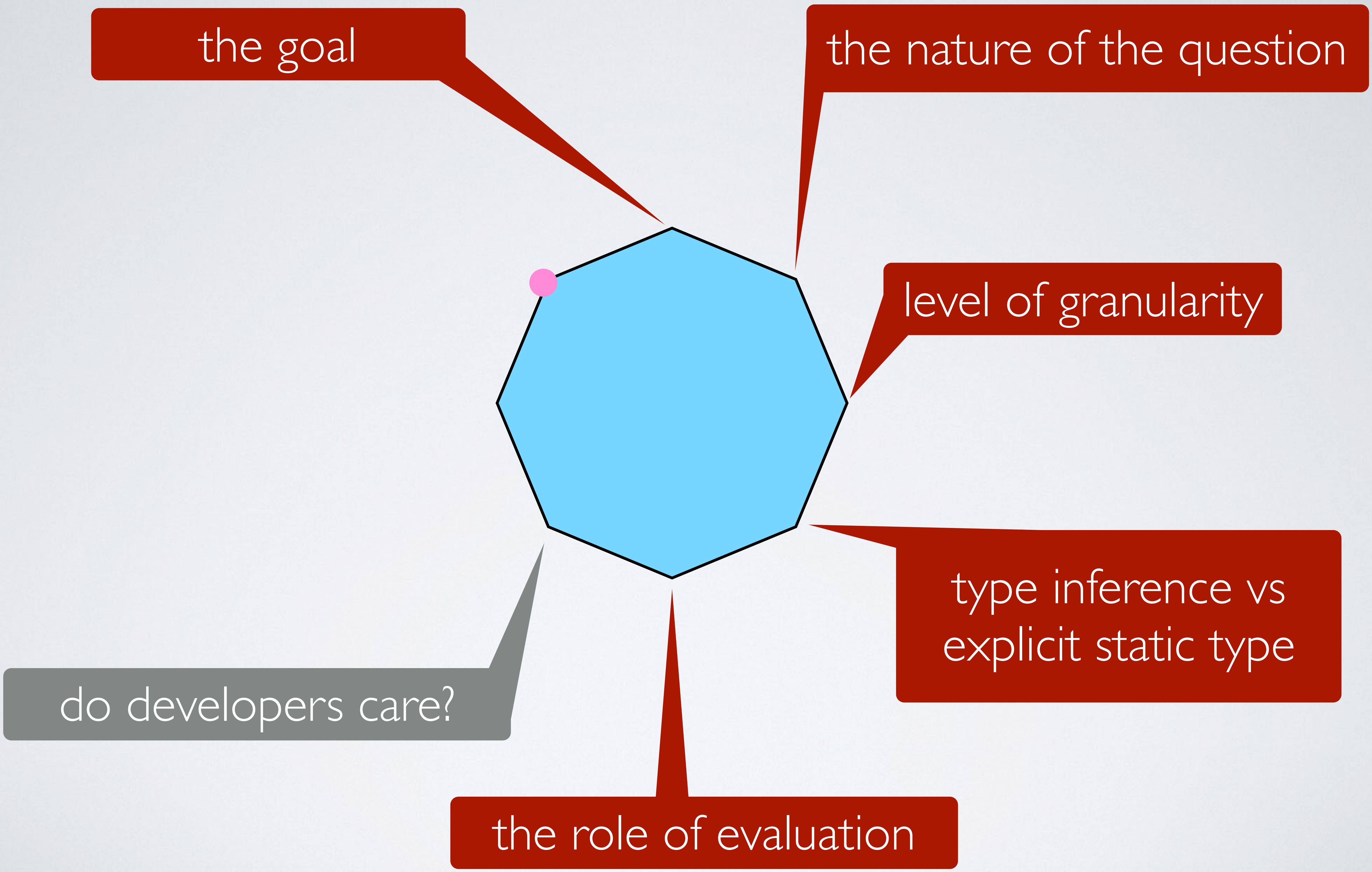
- Practical evaluations are critical for the design feedback loop.
- They focus our mind and our research efforts.

Lessons Learned

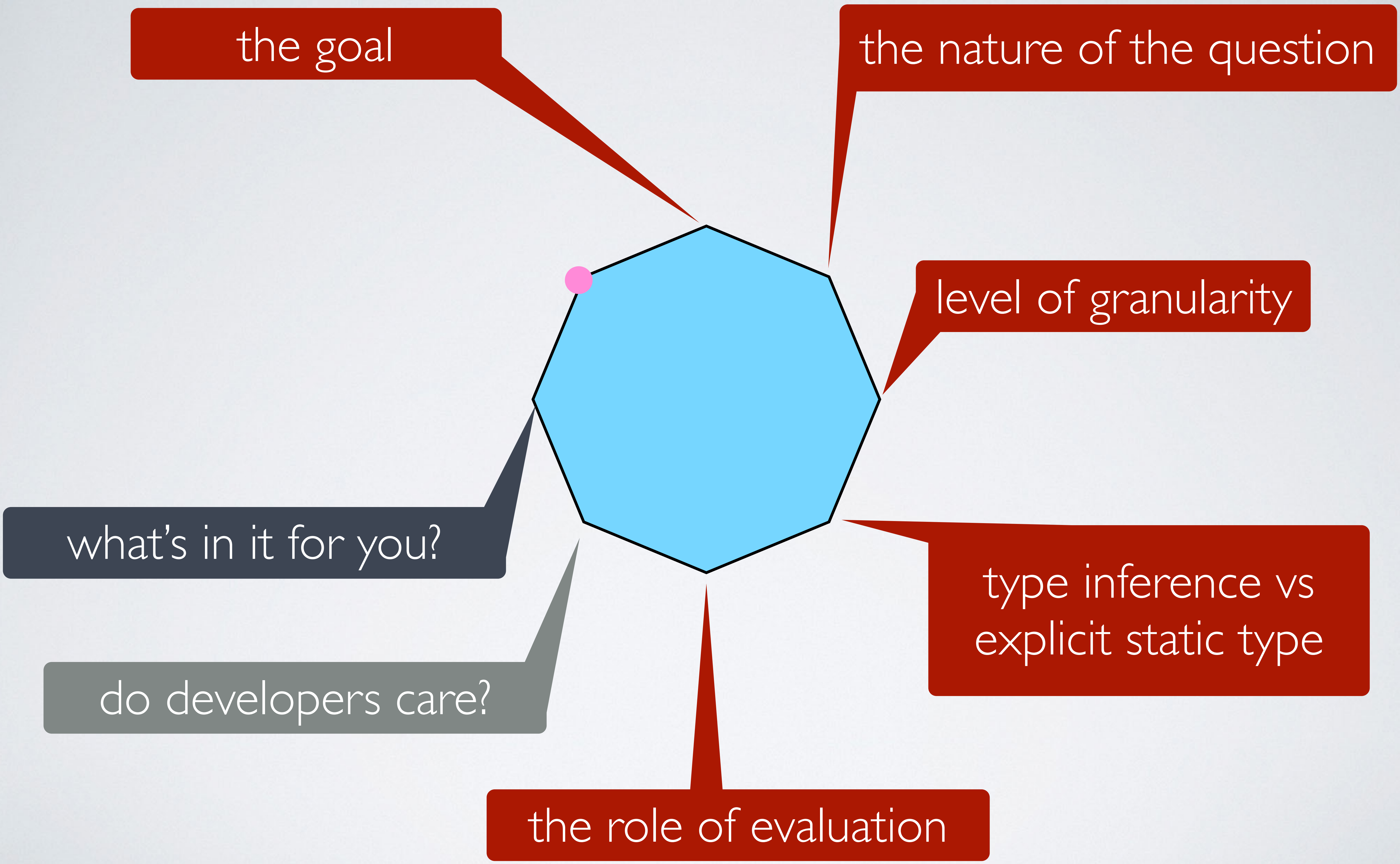
Lessons Learned



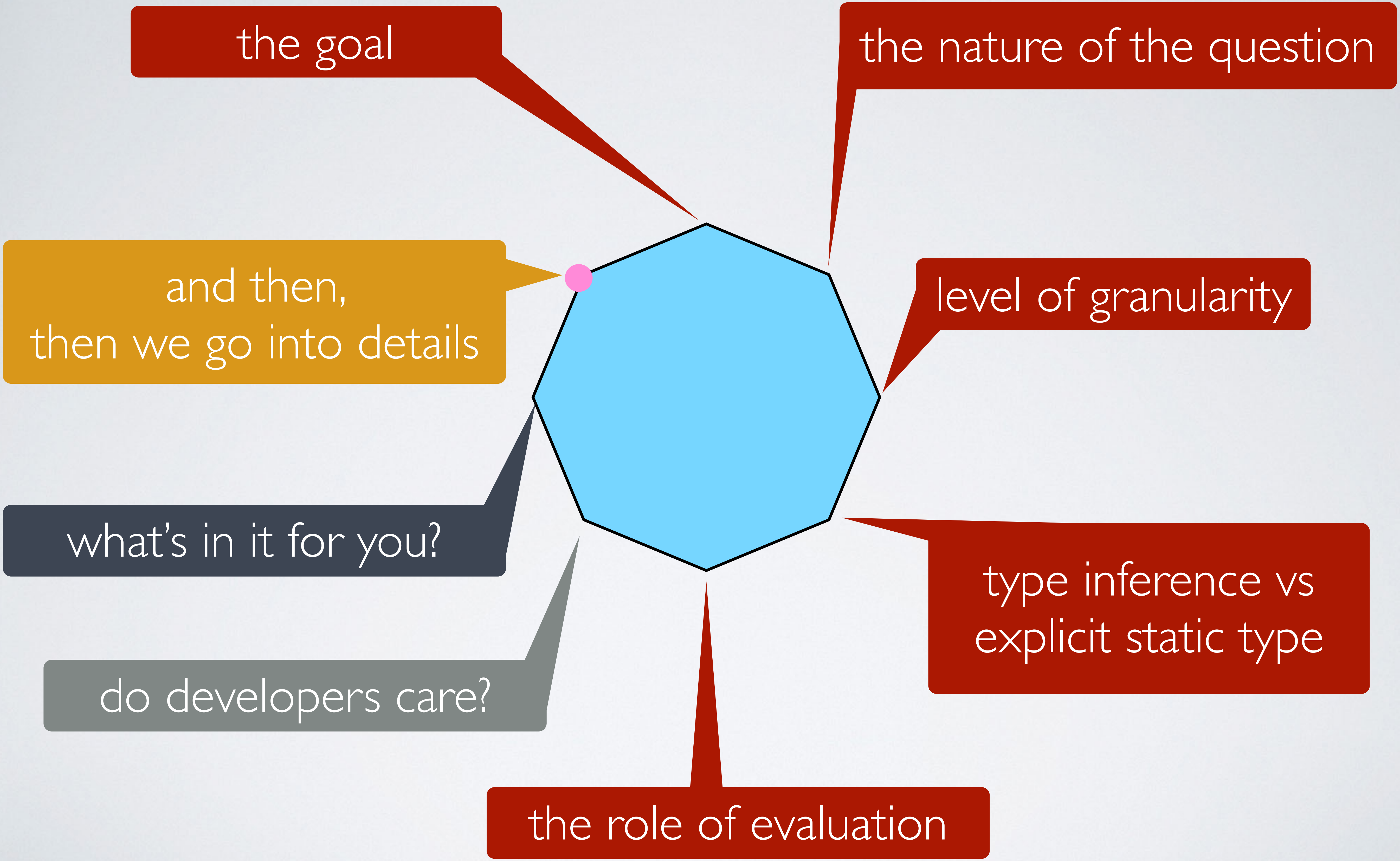
Lessons Learned



Lessons Learned



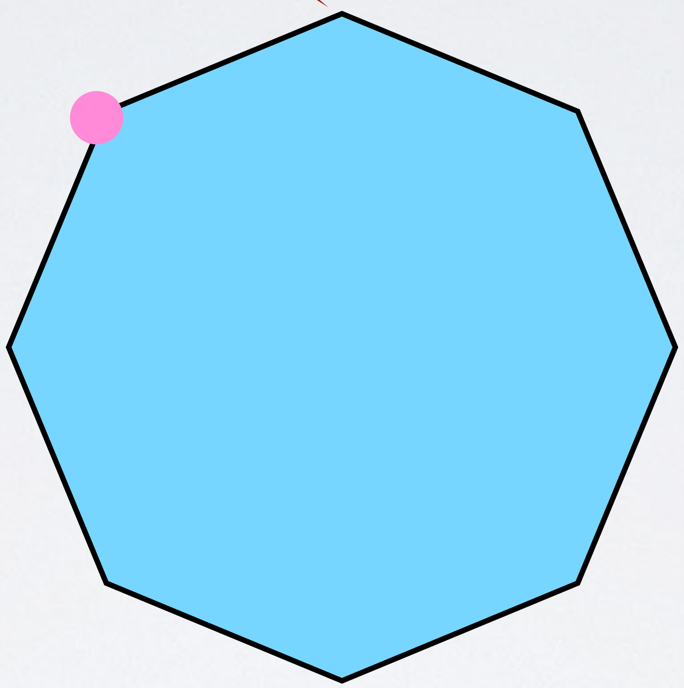
Lessons Learned



Lessons Learned

the goal

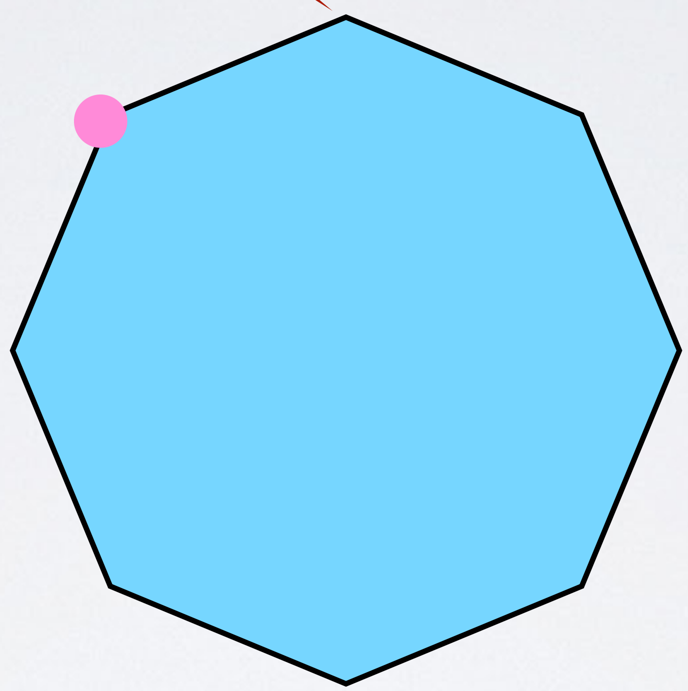
Why do we add types to untyped languages?



Lessons Learned

the goal

Why do we add types to untyped languages?



Is it about bug finding?

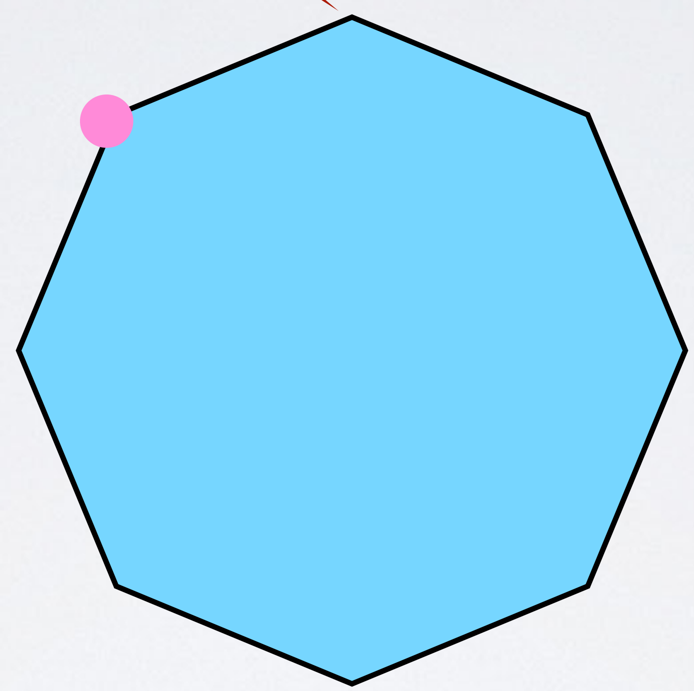
Is it about IDE mechanics?

Is it about execution speed?

Lessons Learned

the goal

Why do we add types to untyped languages?



Is it about bug finding?

Is it about IDE mechanics?

Is it about execution speed?

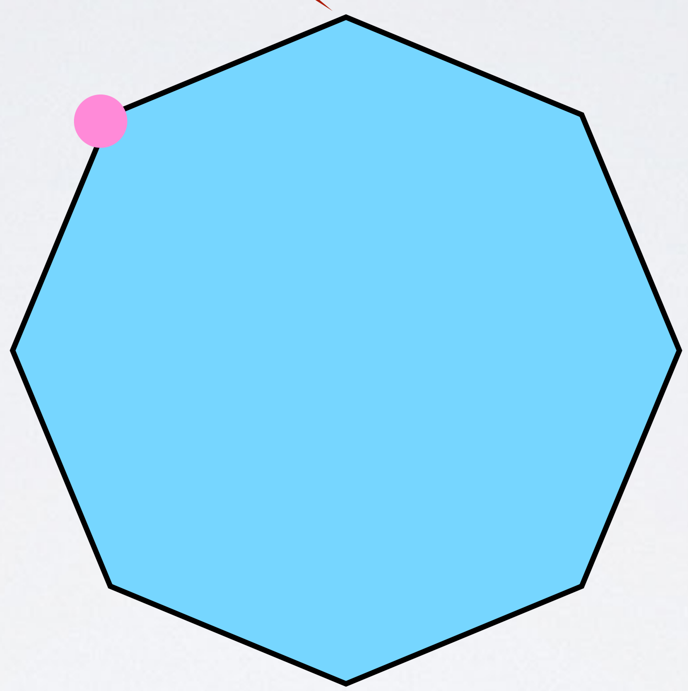
It is about communicating yourself and others developers in the future.

Lessons Learned

the goal

Is it about bug finding?

Why do we add types to untyped languages?



Is it about IDE mechanics?

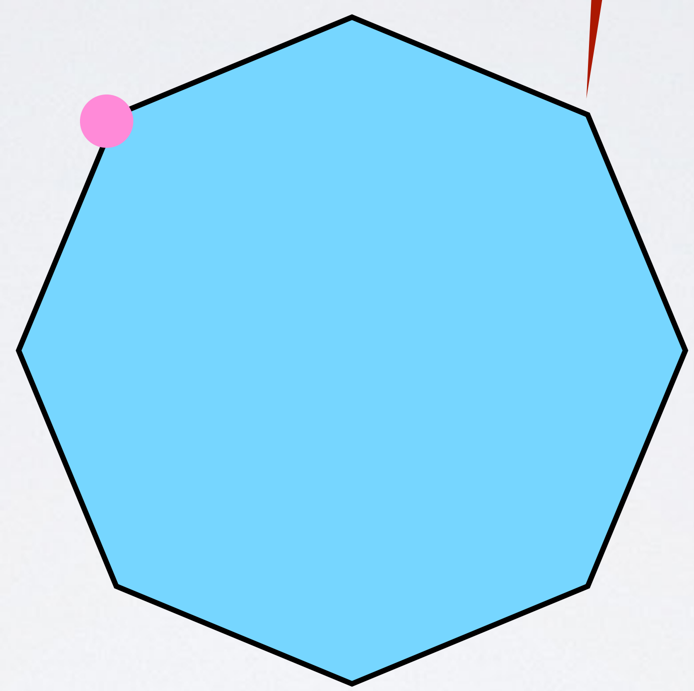
Is it about execution speed?

It is about communicating yourself and others developers in the future.

Challenge ~ how to gather evidence for that?

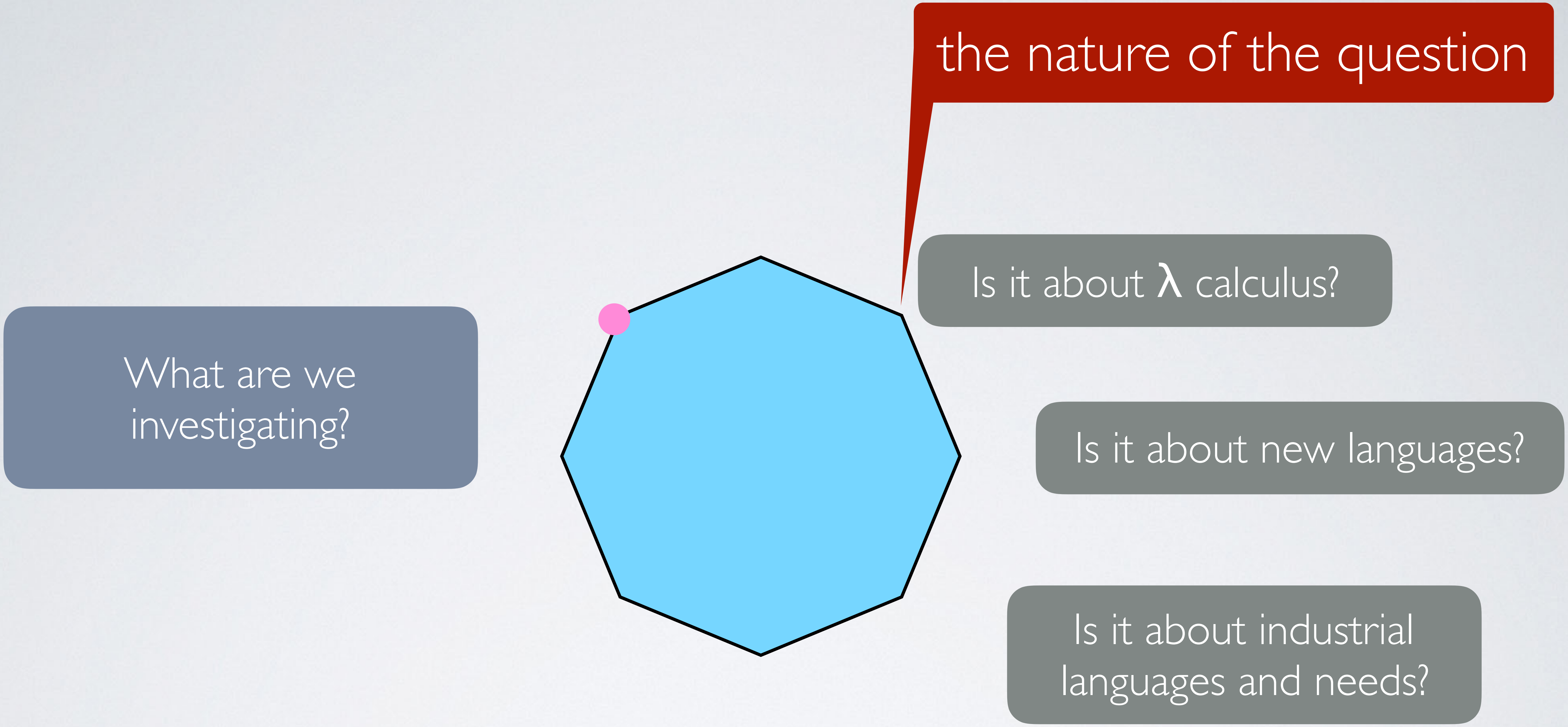
Lessons Learned

What are we investigating?



the nature of the question

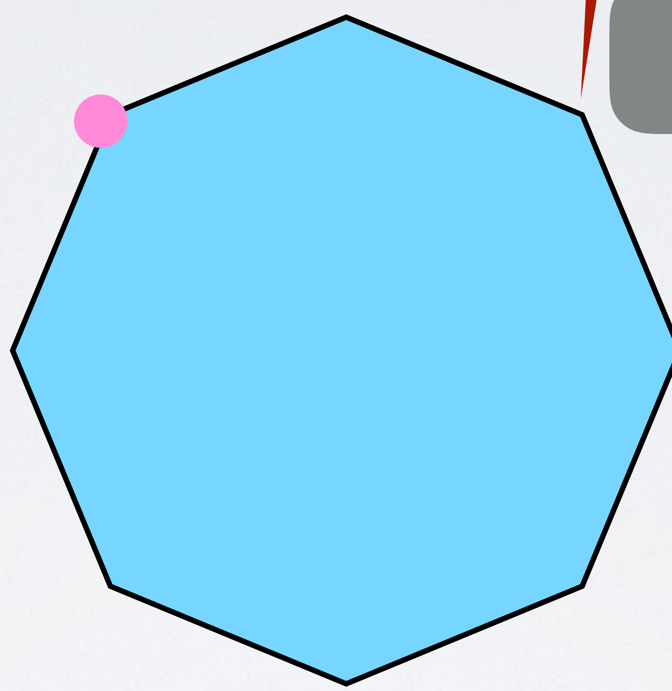
Lessons Learned



Lessons Learned

the nature of the question

What are we investigating?



Is it about λ calculus?

Is it about new languages?

Is it about industrial languages and needs?

We use **Racket** for two reasons:

- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must

Lessons Learned

the nature of the question

Is it about λ calculus?

What are we investigating?

Is it about new languages?

Should we aim for soundness?

Is it about industrial languages and needs?

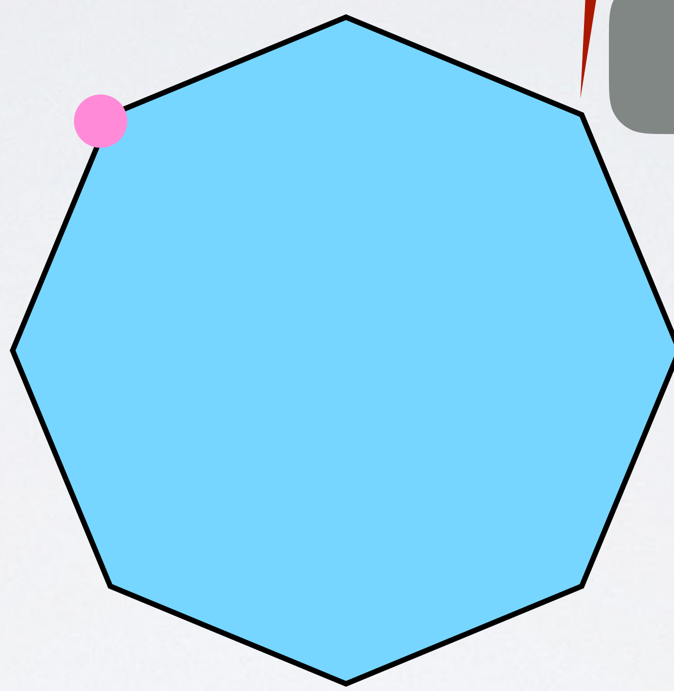
We use **Racket** for two reasons:

- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must

Lessons Learned

the nature of the question

What are we investigating?



Is it about λ calculus?

Is it about new languages?

Is it about industrial languages and needs?

Should we aim for soundness?

Absolutely! If academics don't, nobody will as the numerous designs of hybrid languages in industry show (exception: C#).

We use **Racket** for two reasons:

- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must

Lessons Learned

the nature of the question

Challenge ~ can we make it work? What does a compromise look like?

Is it about λ calculus?

What are we investigating?

Is it about new languages?

Should we aim for soundness?

Is it about industrial languages and needs?

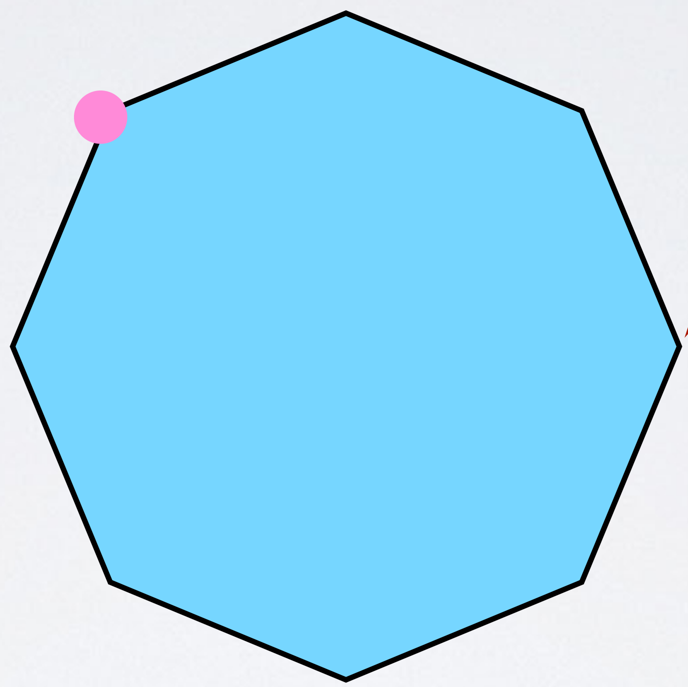
Absolutely! If academics don't, nobody will as the numerous designs of hybrid languages in industry show (exception: C#).

We use **Racket** for two reasons:

- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must

Lessons Learned

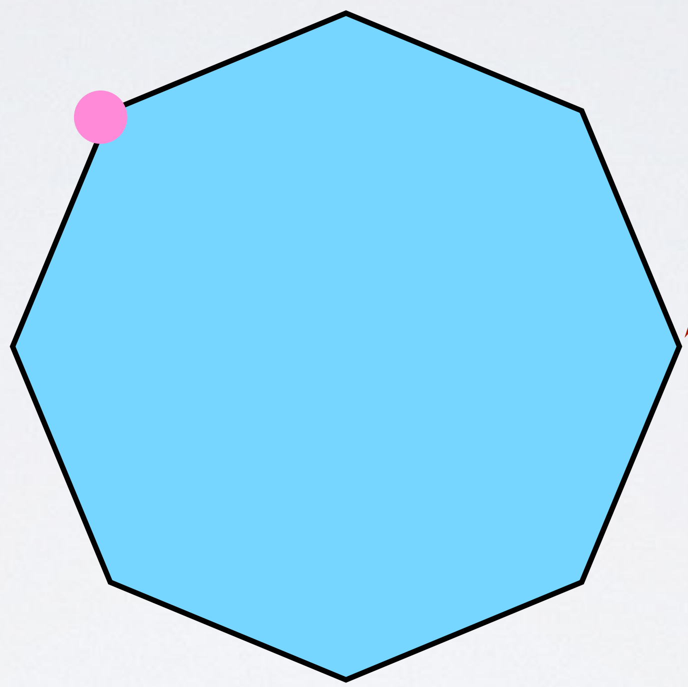
What do programmers want when they add types?



level of granularity

Lessons Learned

What do programmers want when they add types?



level of granularity

Expressions?

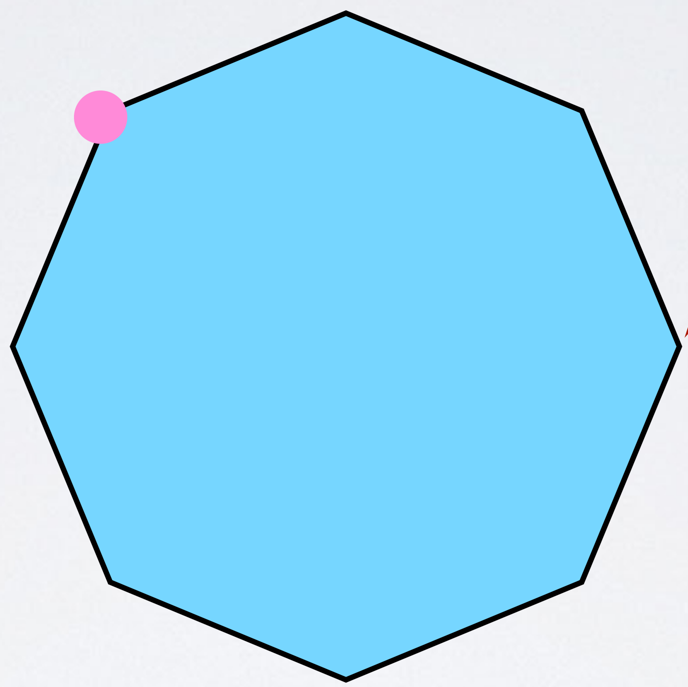
Classes?

Functions?

Modules?

Lessons Learned

What do programmers want when they add types?



level of granularity

Expressions?

Classes?

Functions?

Modules?

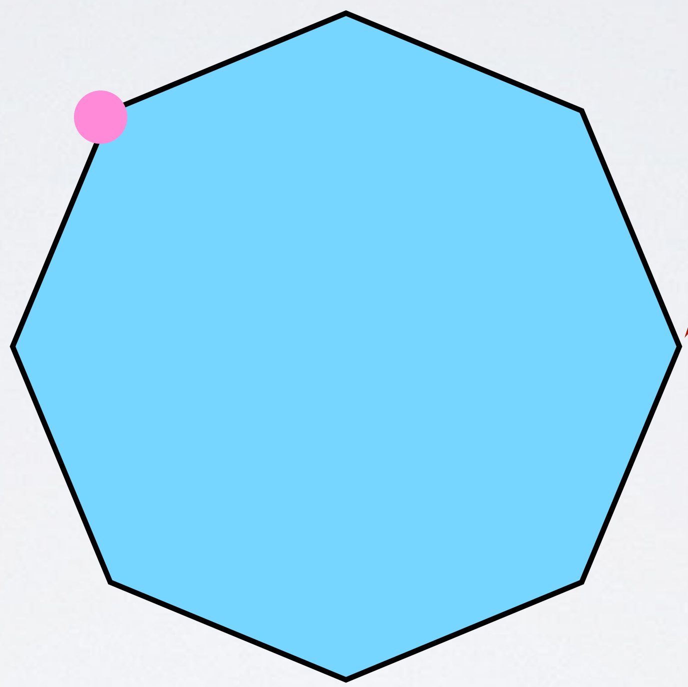
Typed Racket bets on modules, for two reasons:

- typically small enough for conversion
- large enough to keep cost of contracts low

Lessons Learned

I was **wrong**.

What do programmers want when they add types?



level of granularity

Expressions?

Classes?

Functions?

Modules?

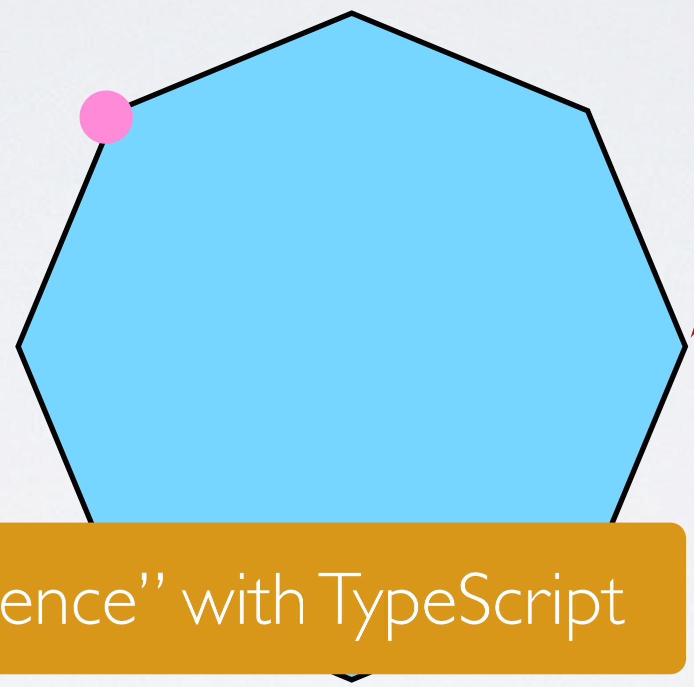
Typed Racket bets on modules, for two reasons:

- typically small enough for conversion
- large enough to keep cost of contracts low

Lessons Learned

I was **wrong**.

What do programmers want when they add types?



level of granularity

Expressions?

Classes?

Functions?

Modules?

the "Eli experience" with TypeScript

Typed Racket bets on modules, for two reasons:

- typically small enough for conversion
- large enough to keep cost of contracts low

Lessons Learned

I was **wrong**.

the performance evaluation is disastrous (until proven otherwise)

What do programmers want when they add types?

level of granularity

Expressions?

Classes?

Functions?

Modules?

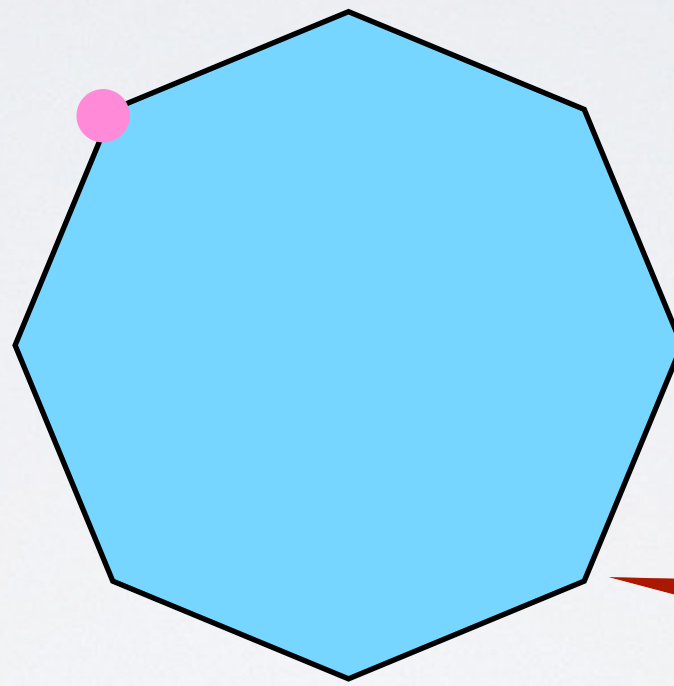
the "Eli experience" with TypeScript

Typed Racket bets on modules, for two reasons:

- typically small enough for conversion
- large enough to keep cost of contracts low

Lessons Learned

Does type inference work for Untyped Languages?



type inference vs explicit static type

Lessons Learned

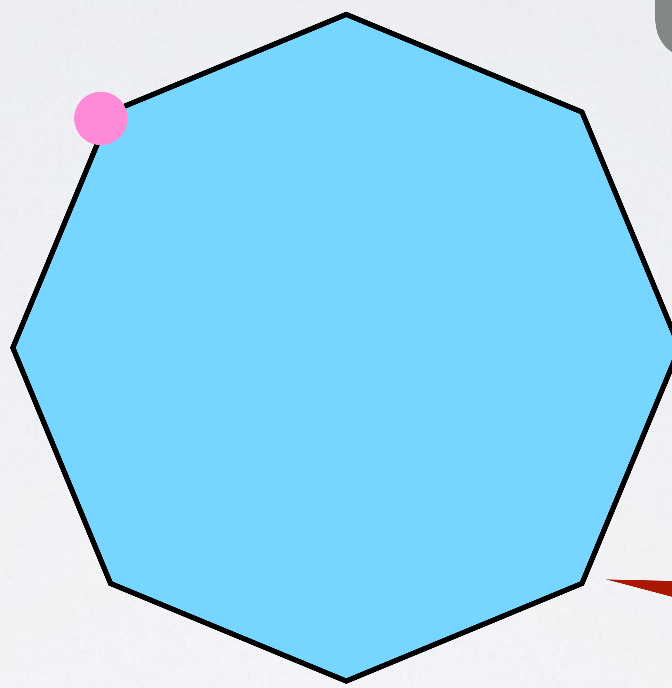
Hindley-Milner?

Local?

Set-based?

Modules?

Does type inference work for Untyped Languages?



type inference vs explicit static type

Lessons Learned

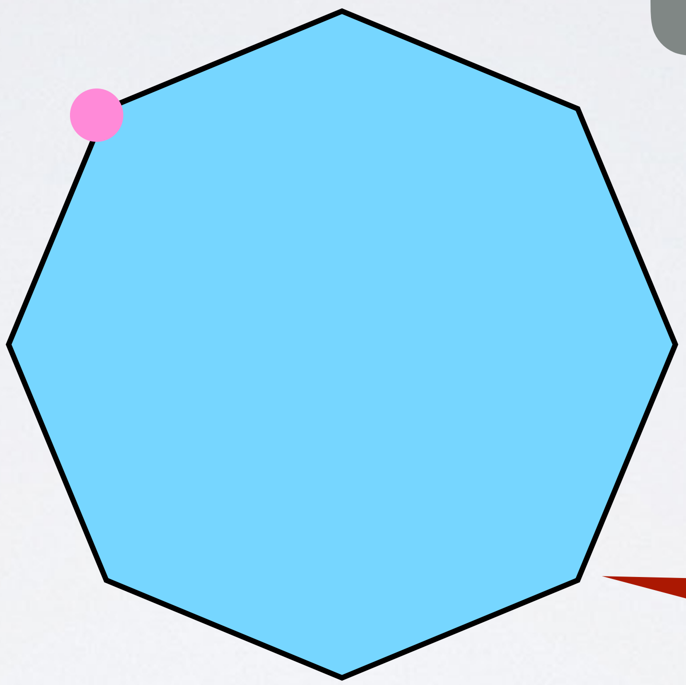
Hindley-Milner?

Local?

Set-based?

Modules?

Does type inference work for Untyped Languages?



type inference vs explicit static type

Probably not:

- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn't compositional

Lessons Learned

Hindley-Milner?

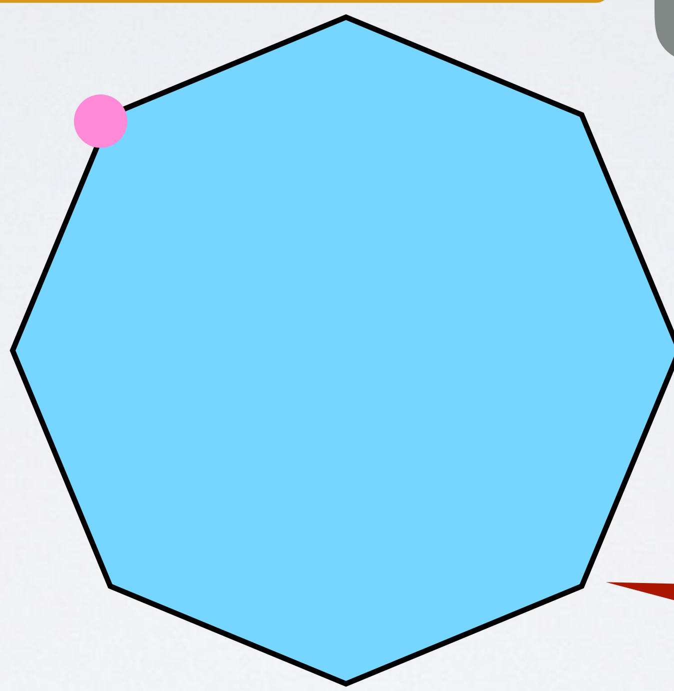
Local?

Set-based?

Modules?

But I: "run time" inference (see work by Shriram Krishnamurthi and Jeff Foster)

Does type inference work for Untyped Languages?



type inference vs explicit static type

Probably not:

- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn't compositional

Lessons Learned

But 2: IDE tools that assist "conversion"

But 1: "run time" inference (see work by Shriram Krishnamurthi and Jeff Foster)

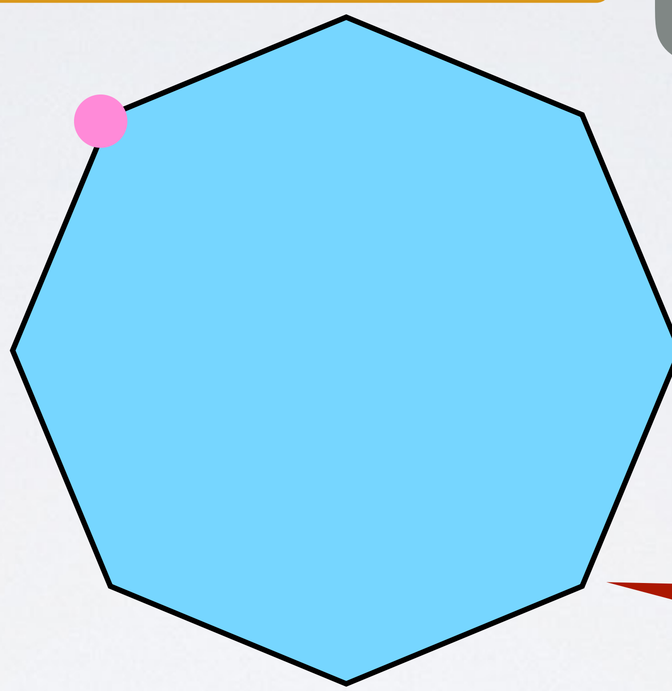
Hindley-Milner?

Local?

Set-based?

Modules?

Does type inference work for Untyped Languages?



type inference vs explicit static type

Probably not:

- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn't compositional

Lessons Learned

But 3: the syntax system necessitates more than plain local inference

But 2: IDE tools that assist “conversion”

But 1: “run time” inference (see work by Shriram Krishnamurthi and Jeff Foster)

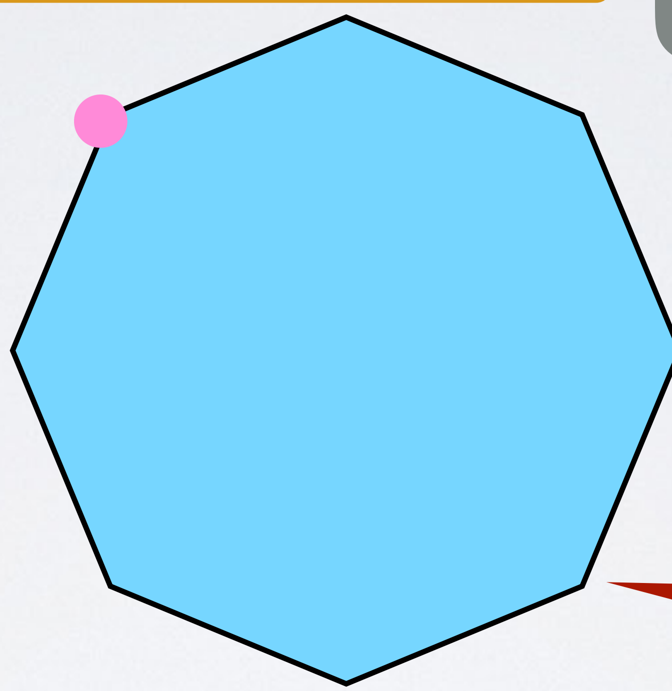
Hindley-Milner?

Local?

Set-based?

Modules?

Does type inference work for Untyped Languages?

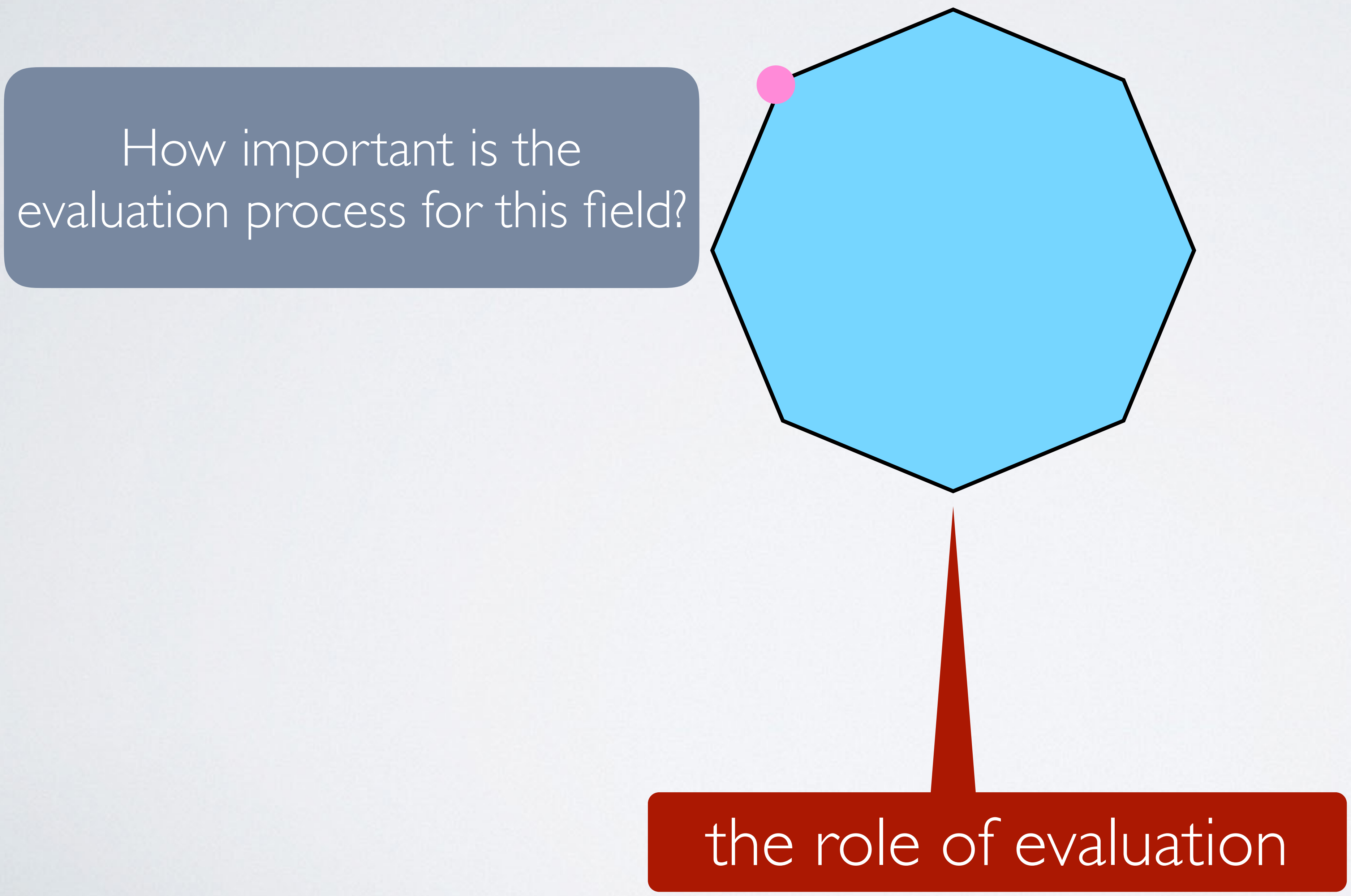


Probably not:

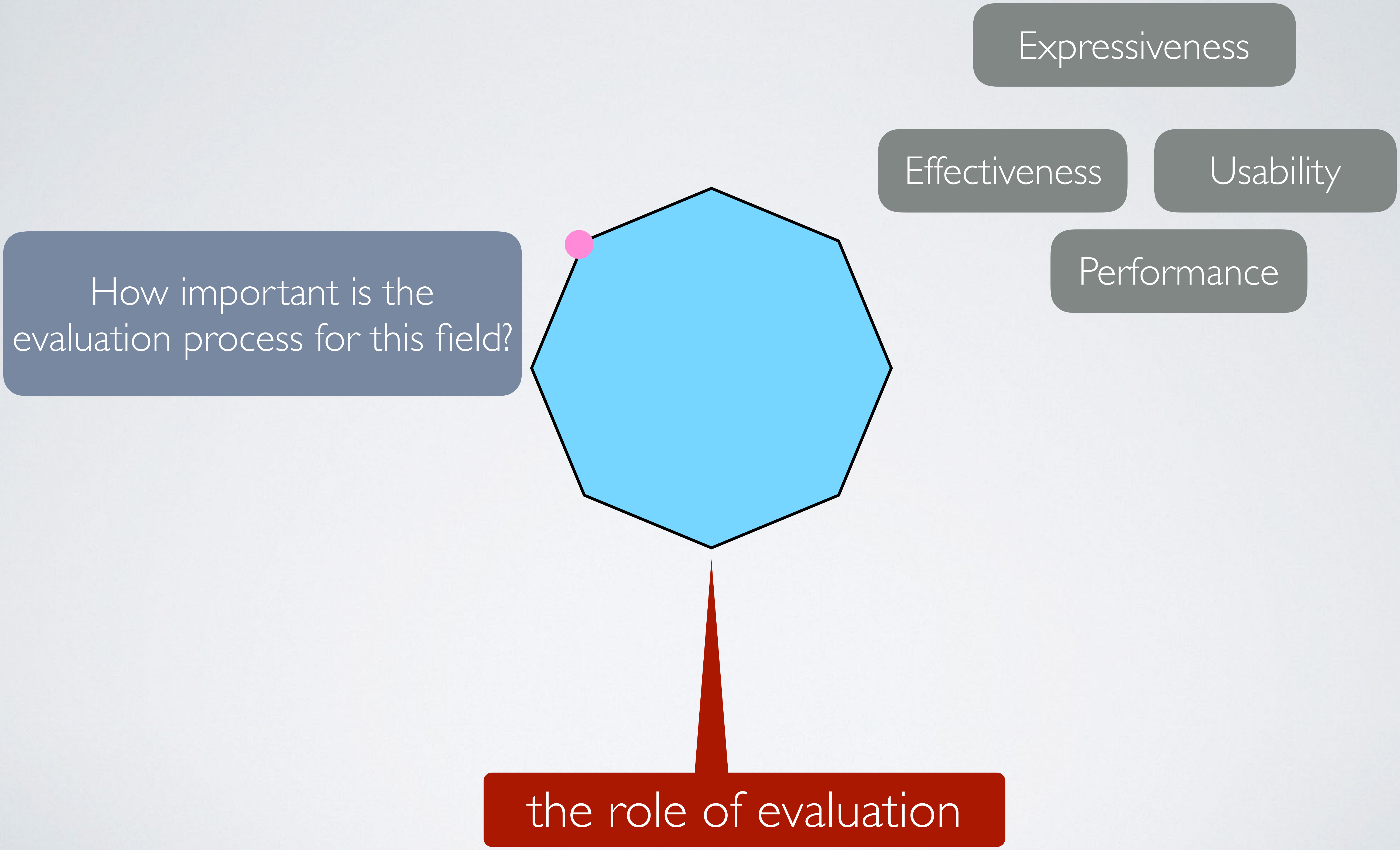
- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn’t compositional

type inference vs explicit static type

Lessons Learned



Lessons Learned



Lessons Learned

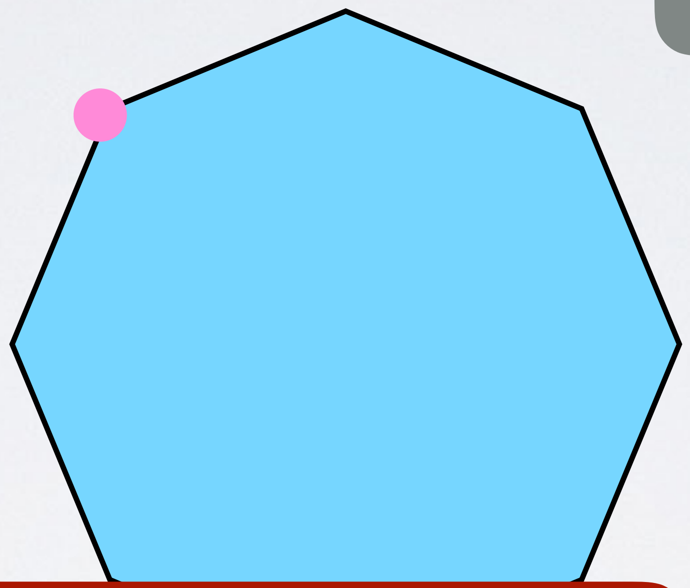
Expressiveness

Effectiveness

Usability

Performance

How important is the evaluation process for this field?



Our "business" is **design,** evaluation is imperative:

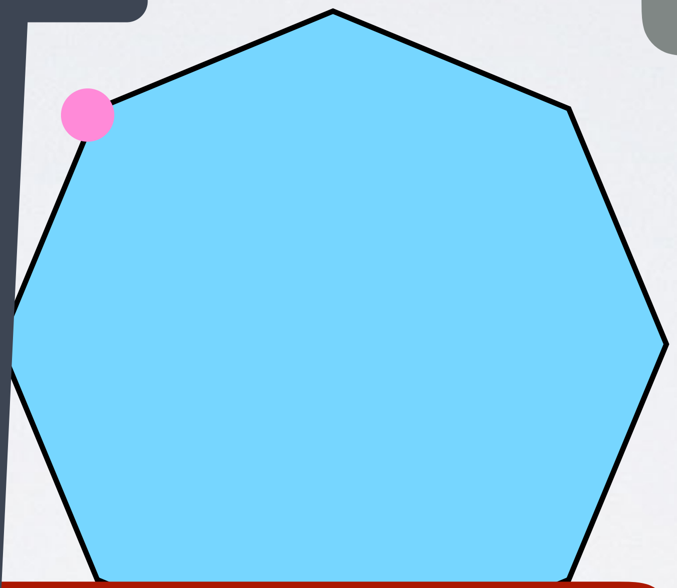
- calculi help with soundness
- existing body of code is **critical**
- but we are academic so preserve flexibility

the role of evaluation

Lessons Learned

Challenge ~ how can academic teams create and maintain a PL?

How important is the evaluation process for this field?



Expressiveness

Effectiveness

Usability

Performance

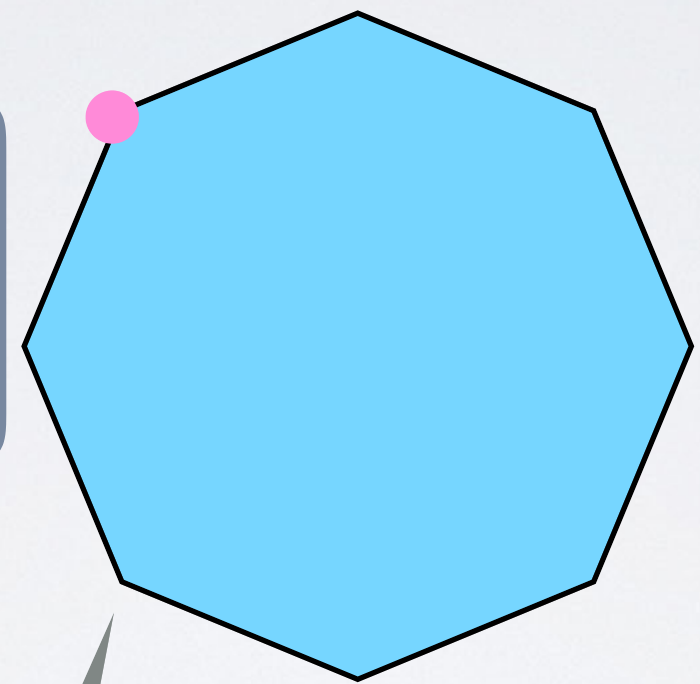
Our "business" is **design,** evaluation is imperative:

- calculi help with soundness
- existing body of code is **critical**
- but we are academic so preserve flexibility

the role of evaluation

Lessons Learned

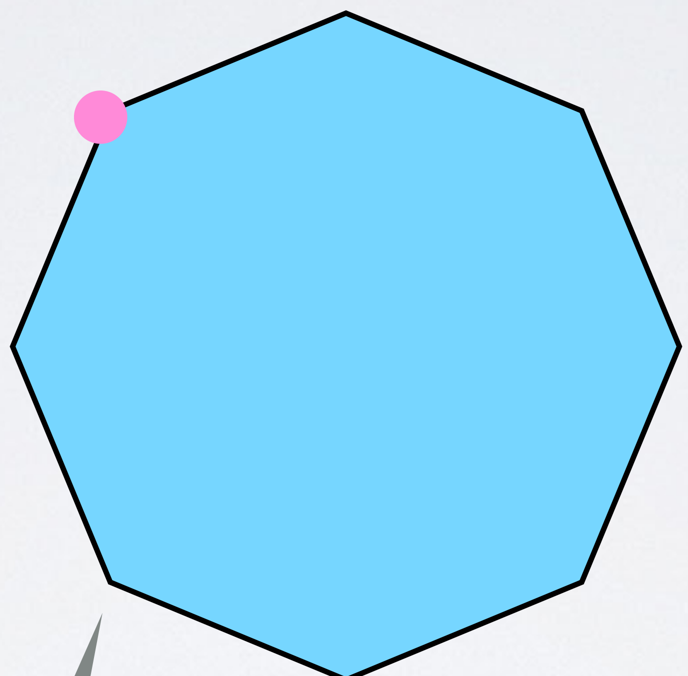
Even academics care in PL ought to care whether the "developer on the street" will eventually care.



do developers care?

Lessons Learned

Even academics care in PL ought to care whether the "developer on the street" will eventually care.

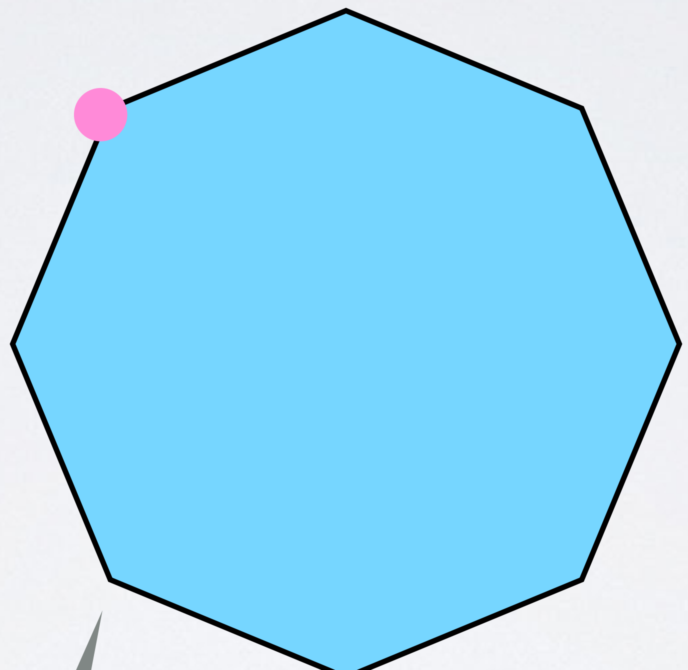


do developers care?

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.

Lessons Learned

Even academics care in PL ought to care whether the "developer on the street" will eventually care.



PL has failed to gather data that support soundness and sound design.

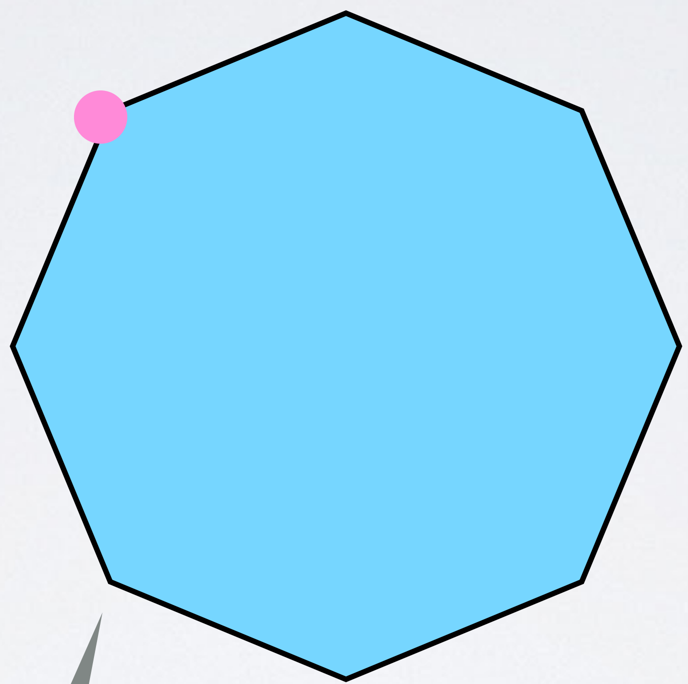
do developers care?

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.

Lessons Learned

PL fails to make the argument (even) at the "theoretical" level of courses.

Even academics care in PL ought to care whether the "developer on the street" will eventually care.



PL has failed to gather data that support soundness and sound design.

do developers care?

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/ gradual approach to typing.

Lessons Learned

Challenge ~ how can academic PL improve its teaching?

PL fails to make the argument (even) at the "theoretical" level of courses.

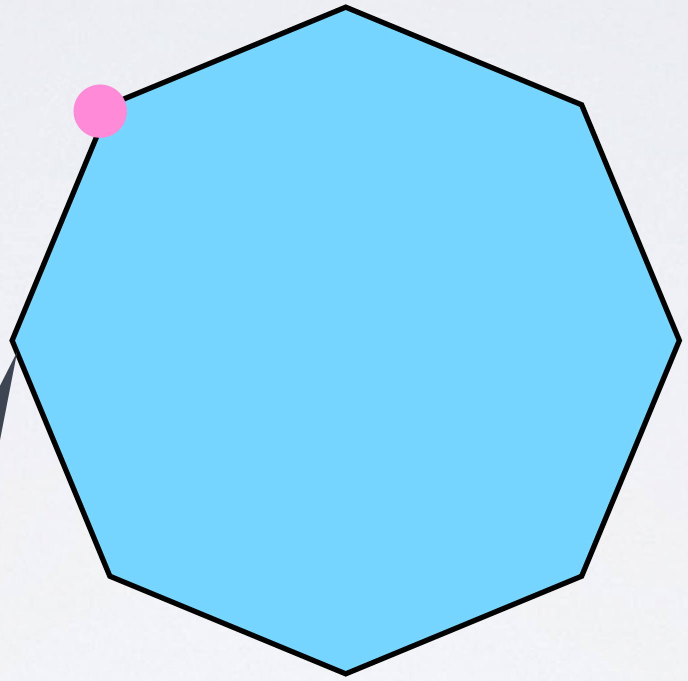
Even academics care in PL ought to care whether the "developer on the street" will eventually care.

PL has failed to gather data that support soundness and sound design.

do developers care?

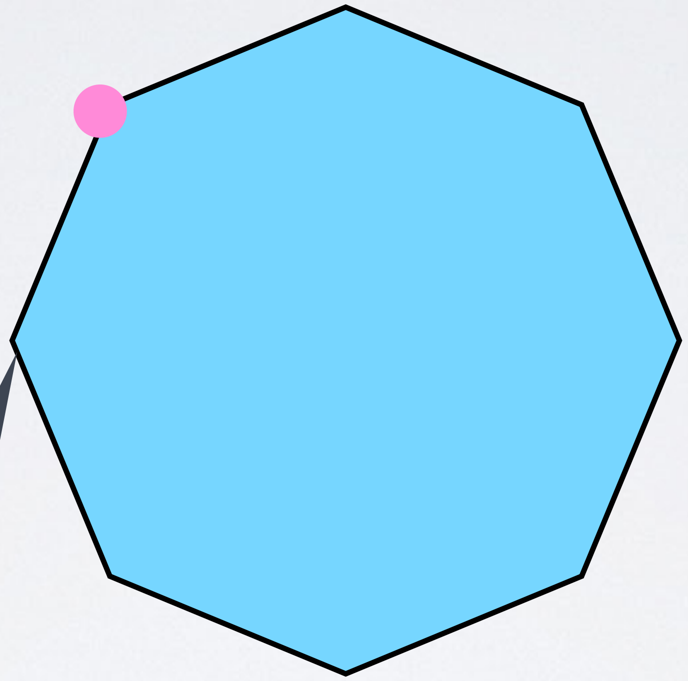
Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/ gradual approach to typing.

Lessons Learned



what's in it for you?

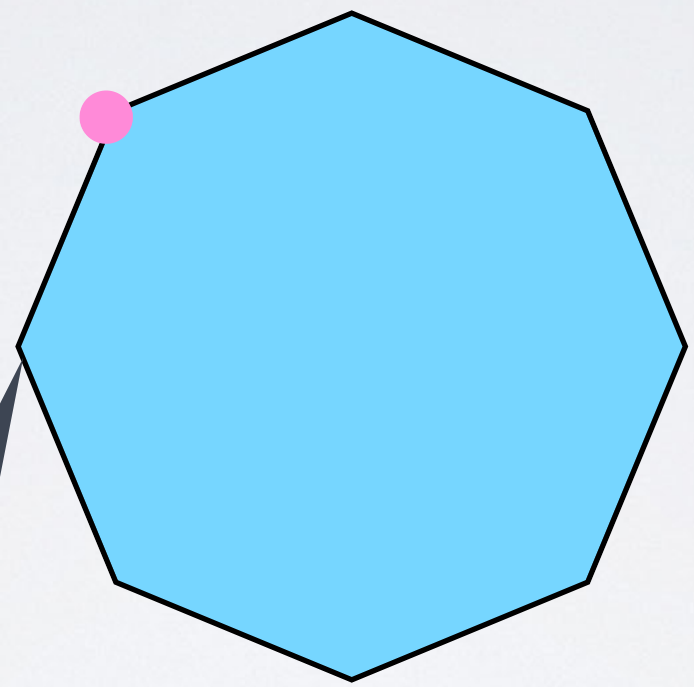
Lessons Learned



what's in it for you?

The area provides a **rich field of challenging problems,** ranging from the incredibly theoretical to the highly practical.

Lessons Learned



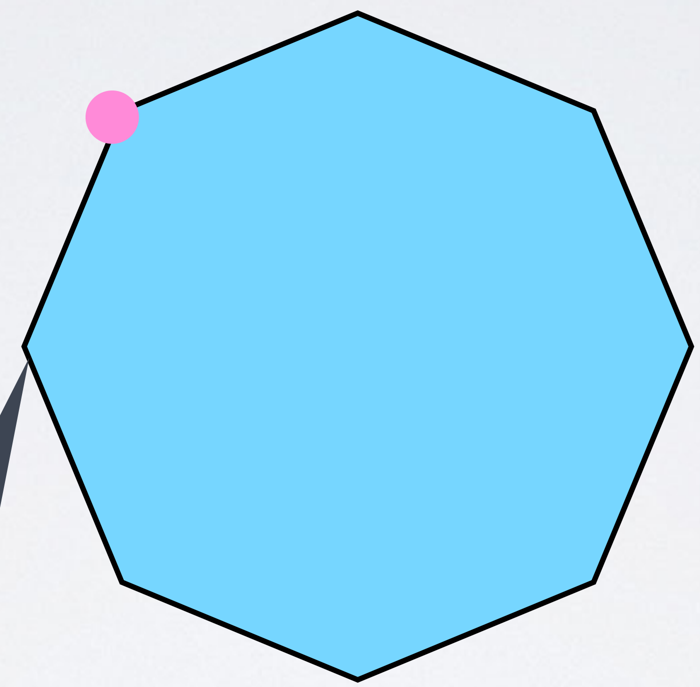
Practical grounding matters.

what's in it for you?

The area provides a **rich field of challenging problems,** ranging from the incredibly theoretical to the highly practical.

Lessons Learned

Take a the long-term view (Wright, Flanagan, Krishnamurthi, Tobin-Hochstadt).



Practical grounding matters.

what's in it for you?

The area provides a **rich field of challenging problems,** ranging from the incredibly theoretical to the highly practical.

The End

Soft Typists

Robert “Corky” Cartwright, Mike Fagan, Andrew Wright

The MrSpidey Crew

Cormac Flanagan, Shriram Krishnamurthi, Matthew Flatt

Contractors

Robby Findler, Christos Dimoulas Philippe Meunier, Stevie Strickland

Typed Racketeers

Sam Tobin-Hochstadt, Vincent, St-Amour, Asumu Takikawa

Evaluators

Ben Greenman, Max New, Jan Vitek