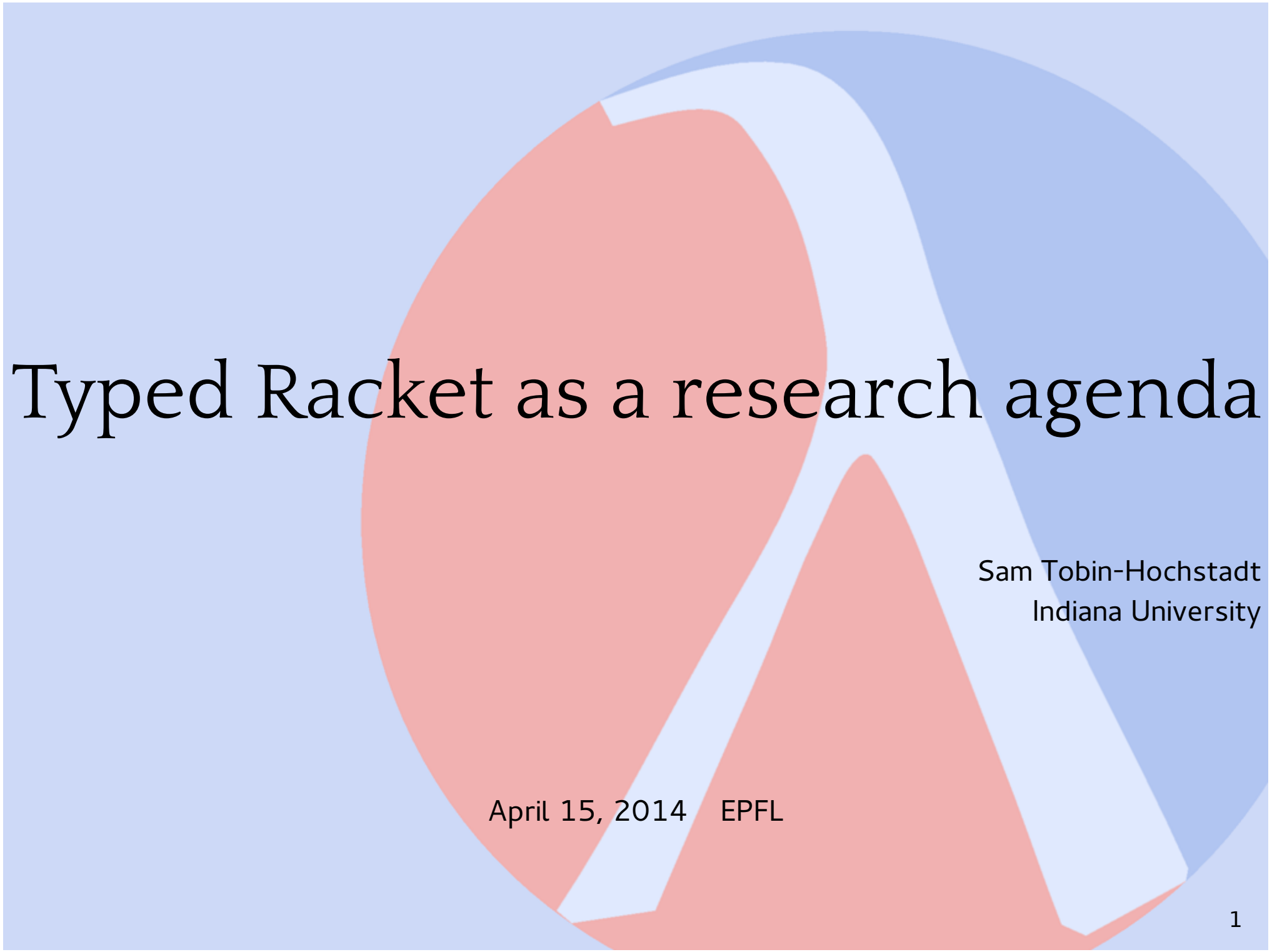
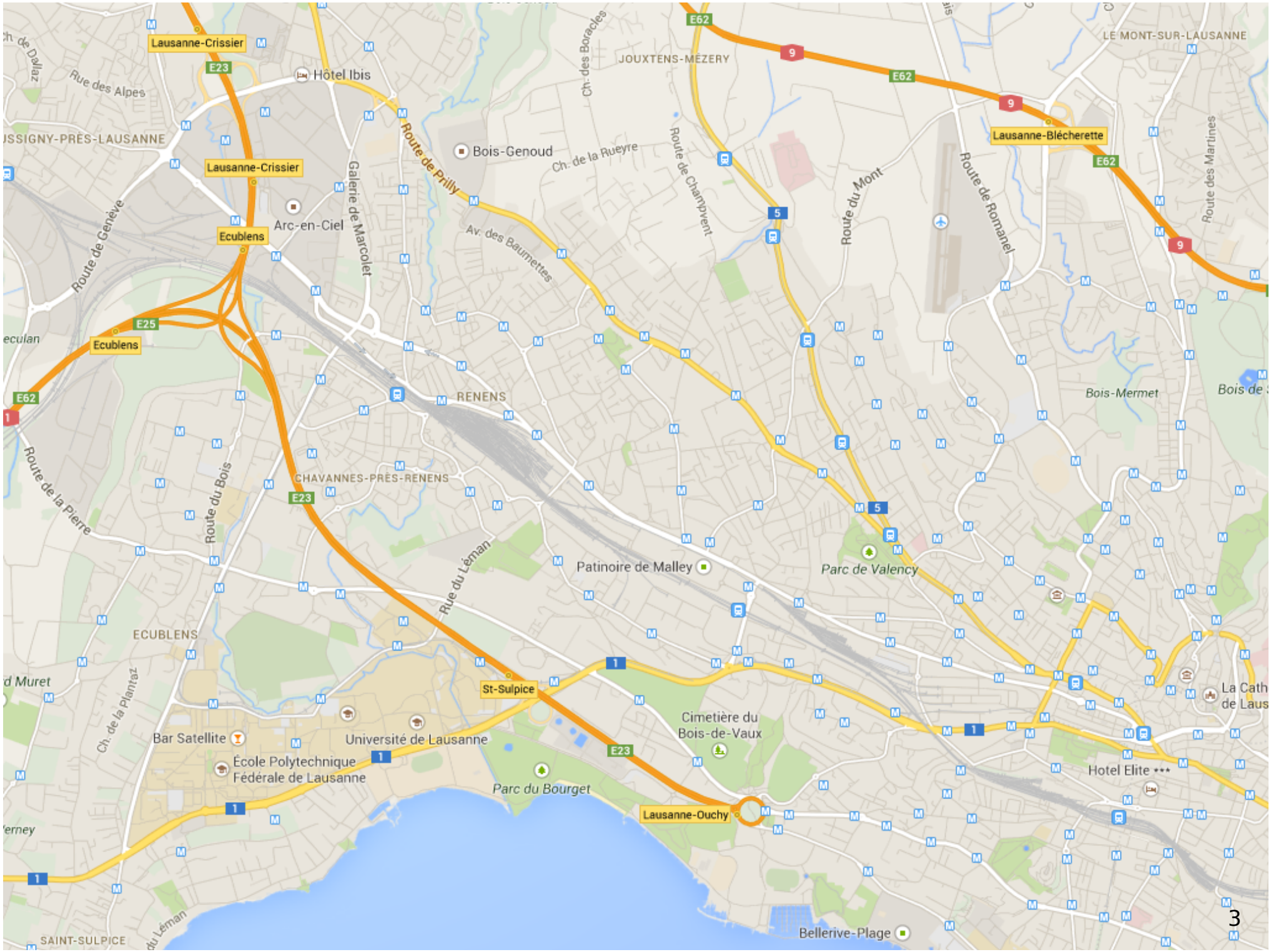


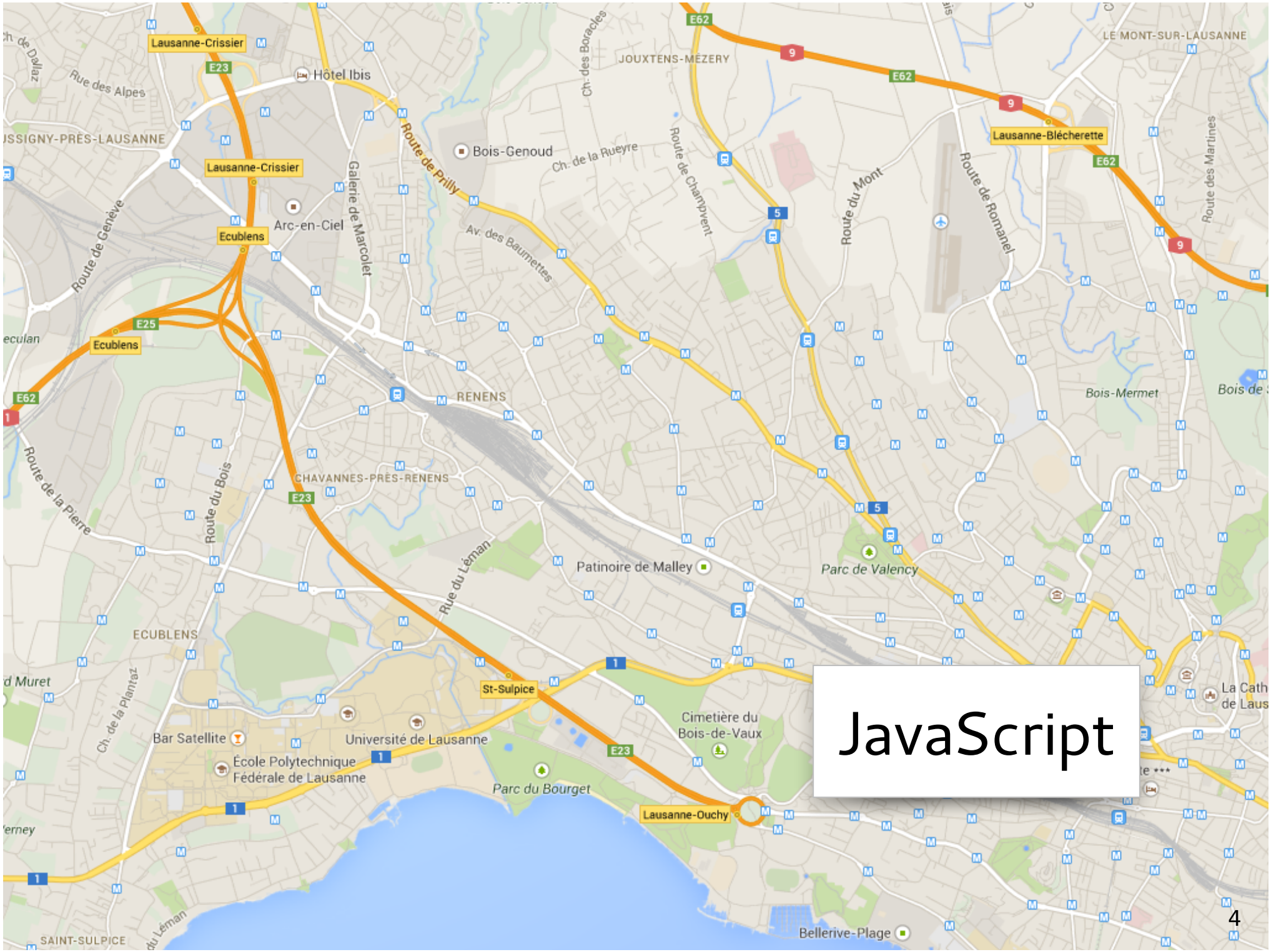
Typed Racket as a research agenda

Sam Tobin-Hochstadt
Indiana University

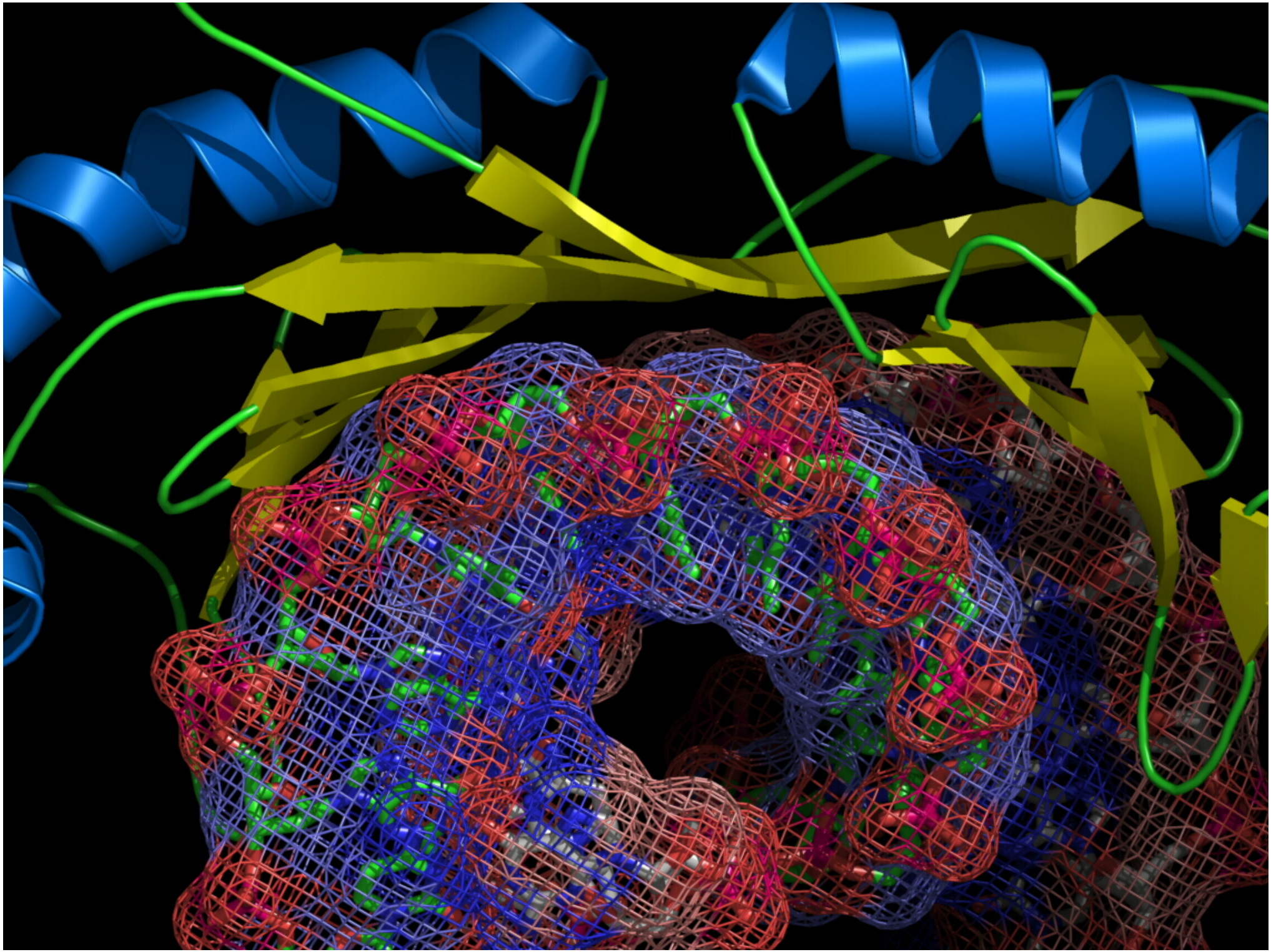
April 15, 2014 EPFL

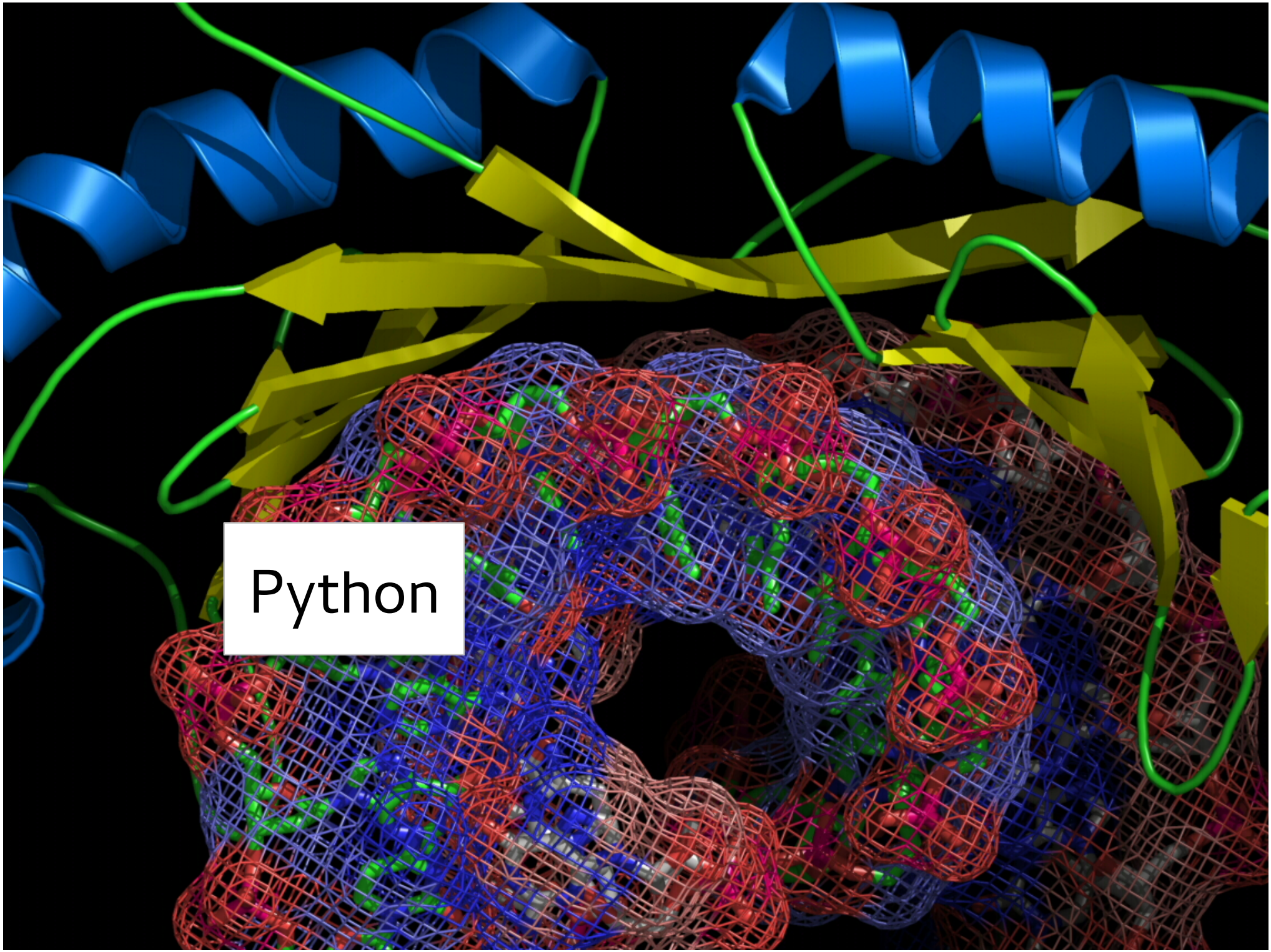
The Rise of Dynamic Languages


Lausanne-Crissier
E23
Hôtel Ibis
Route de Prilly
Bois-Genoud
Ch. de la Rueyre
Route de Champvent
JOUXTENS-MÉZERY
Ch. des Boracles
E62
9
LE MONT-SUR-LAUSANNE
Lausanne-Blécherette
Route du Mont
Route de Romanel
Route des Martines
Galerie de Marcolet
Arc-en-Ciel
Ecublens
Av. des Baumettes
Route de Genève
E25
RENENS
Bois-Mermet
5
CHAVANNES-PRÈS-RENENS
Route du Bois
Route de la Pierre
Rue du Léman
Patinoire de Malley
Parc de Valency
ECUBLENS
St-Sulpice
Cimetière du
Bois-de-Vaux
Bar Satellite
Université de Lausanne
École Polytechnique
Fédérale de Lausanne
Parc du Bourget
Lausanne-Ouchy
Hotel Elite
Bellerive-Plage
SAINT-SULPICE
1


JavaScript
Lausanne-Crissier
Hôtel Ibis
Rue des Alpes
Route de Prilly
Bois-Genoud
Ch. de la Rueyre
Route de Champvent
JOUXTENS-MÉZERY
Ch. des Boracles
LE MONT-SUR-LAUSANNE
Lausanne-Blécherette
Route du Mont
Route de Romanel
Route des Martines
Galerie de Marcolet
Arc-en-Ciel
Ecublens
Route de Genève
Av. des Baumettes
E62
E25
E23
RENENS
Bois-Mermet
CHAVANNES-PRÈS-RENENS
Route du Bois
Route de la Pierre
Rue du Léman
Patinoire de Malley
Parc de Valency
ECUBLENS
St-Sulpice
Cimetière du
Bois-de-Vaux
Bar Satellite
Université de Lausanne
École Polytechnique
Fédérale de Lausanne
Parc du Bourget
Lausanne-Ouchy
Bellerive-Plage
SAINT-SULPICE
Ch. de la Plantaz


Lua




Python

Ecole Polytechnique Fédérale de Lausanne (EPFL)

Like | Follow | Messag

★★★★½ (542 ratings)

30,134 likes · 1,552 talking about this · 19,091 were here

Federal Institutes of
ssions: education, research
he highest international

Photos

Youtube

Twitter

Instagram

Ecole Polytechnique Fédérale de Lausanne (EPFL)

Like | Follow | Messag

★★★★½ (542 ratings)

30,134 likes · 1,552 talking about this · 19,091 were here

Federal Institutes of
ssions: education, research
he highest international

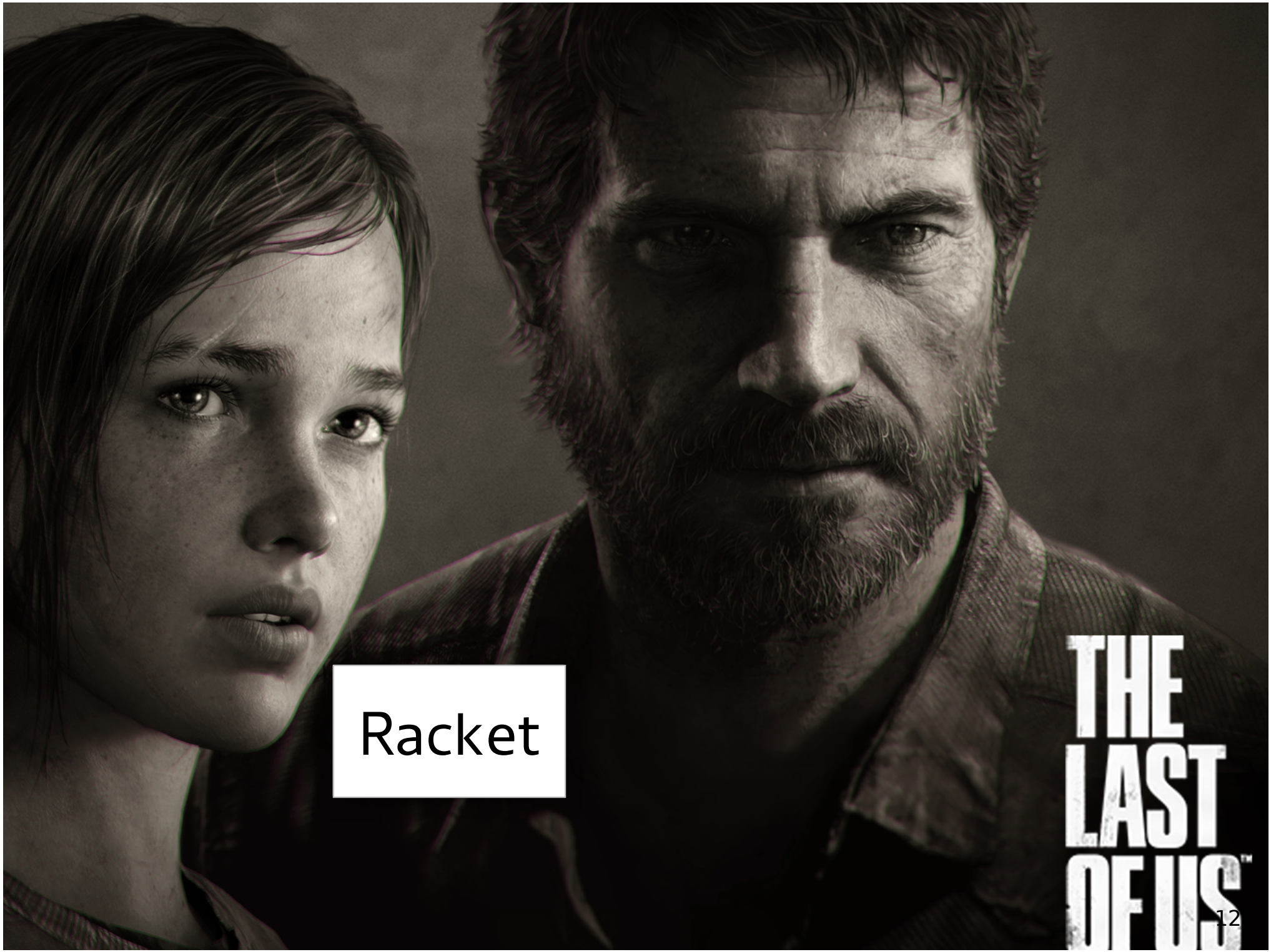
Photos

Youtube

Twitter

Instagram


THE
LAST
OF US™


Racket
THE LAST OF US™

PENSIONS MYNDIGHETEN

Sök på Pensionsmyndigheten

Ange sökord | Sök

A–Ö | Webbkarta | Fondsök

Startsida | Gå i pension | Så fungerar pensionen | Dina pensionssidor | Blanketter och broschyrer | Aktuellt och press

För pensionärer

- Pensionsutbetalning
- Ändra skatteavdrag
- Beställ pensionärsintyg
- Ansök om bostadstillägg
- Räkna ut ditt bostadstillägg
- Pensionär utomlands

För pensionssparare

- Gör en pensionsprognos
- Ansök om pension
- Var i livet är du?
- Pensionslabbet
- Anmälan till pensionsträff
- Om efterlevandepension

Fondsparande

- Sök fonder
- Jämför fonder
- Byt fonder (logga in)
- Fondhändelser
- Ta hand om ditt sparande
- Fondvalsguiden

Dina p

Som inlo
annat gö
ansöka o
fonder.

- Logga
- Hjälp r

PENSIONS MYNDIGHETEN

Sök på Pensionsmyndigheten

Ange sökord | Sök

A–Ö | Webbkarta | Fondsök

Startsida | Gå i pension | Så fungerar pensionen | Dina pensionssidor | Blanketter och broschyrer | Aktuellt och press

För pensionärer

- Pensionsutbetalning
- Ändra skatteavdrag
- Beställ pensionärsintyg
- Ansök om bostadstillägg
- Räkna ut ditt bostadstillägg
- Pensionär utomlands

För pensionssparare

- Gör en pensionsprognos
- Ansök om pension
- Var i livet är du?
- Pensionslabbet
- Anmälan till pensionsträff
- Om efterlevandepension

Fond

- Sök
- Jäm
- Byt fonder (logga in)
- Fondhändelser
- Ta hand om ditt sparande
- Fondvalsguiden

Perl

Dina p

Som inlog
annat gö
ansöka o
fonder.

- Logga
- Hjälp r

"whipitupitude" — Larry Wall

So what's the problem?

"whipitupitude" — Larry Wall

What's not so good

```
(define (main stx trace-flag super-expr
              deserialize-id-expr name-id
              interface-exprs defn-and-exprs)
```

```
(let-values ([(this-id) #'this-id]
              [(the-obj) (datum->syntax (quote-syntax here) (gensym 'self))]
              [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))])
  (let* ([def-ctx (syntax-local-make-definition-context)]
         [localized-map (make-bound-identifier-mapping)]
         [any-localized? #f]
         [localize/set-flag (lambda (id)
                              (let ([id2 (localize id)])
                                (unless (eq? id id2)
                                  (set! any-localized? #t))
                                id2))]
         [bind-local-id (lambda (id)
                          (let ([l (localize/set-flag id)])
                            (syntax-local-bind-syntaxes (list id) #f def-ctx)
                            (bound-identifier-mapping-put!
                             localized-map
                             id
                             l)))]
         [lookup-localize (lambda (id)
                            (bound-identifier-mapping-get
                             localized-map
                             id
                             (lambda ()
                               ; If internal & external names are distinguished,
                               ; we need to fall back to localize:
                               (localize id))))])
    ; ----- Expand definitions -----
    (let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)]
          [bad (lambda (msg expr)
                 (raise-syntax-error #f msg stx expr))]
          [class-name (if name-id
                          (syntax-e name-id)
                          (let ([s (syntax-local-infer-name stx)])
                            (if (syntax? s)
                                (syntax-e s)
                                s)))])
      ; ------ Basic syntax checks -----
      (for-each (lambda (stx)
                  (syntax-case stx (-init init-rest -field -init-field inherit-field
                                          private public override augride
                                          public-final override-final augment-final
                                          pubment overment augment
                                          rename-super inherit inherit/super inherit/inner rename-inner
                                          inspect)
                    [(form orig idp ...)
                     (and (identifier? #'form)
                          (or (free-identifier=? #'form (quote-syntax -init))
                              (free-identifier=? #'form (quote-syntax -init-field))))]))))))
```

+ 900 lines

What's not so good

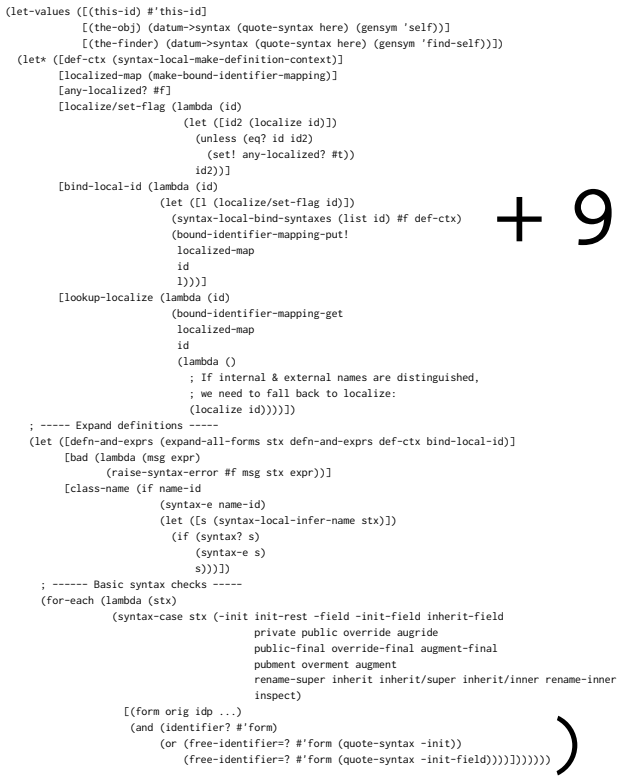
```
; Start here:
(define (main stx trace-flag super-expr
              deserialize-id-expr name-id
              interface-exprs defn-and-exprs)
```

```
(let-values ([(this-id) #'this-id]
              [(the-obj) (datum->syntax (quote-syntax here) (gensym 'self))]
              [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))])
  (let* ([def-ctx (syntax-local-make-definition-context)]
         [localized-map (make-bound-identifier-mapping)]
         [any-localized? #f]
         [localize/set-flag (lambda (id)
                              (let ([id2 (localize id)])
                                (unless (eq? id id2)
                                  (set! any-localized? #t))
                                id2))]
         [bind-local-id (lambda (id)
                          (let ([l (localize/set-flag id)])
                            (syntax-local-bind-syntaxes (list id) #f def-ctx)
                            (bound-identifier-mapping-put!
                             localized-map
                             id
                             l)))]
         [lookup-localize (lambda (id)
                            (bound-identifier-mapping-get
                             localized-map
                             id
                             (lambda ()
                               ; If internal & external names are distinguished,
                               ; we need to fall back to localize:
                               (localize id))))])
    ; ----- Expand definitions -----
    (let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)]
          [bad (lambda (msg expr)
                 (raise-syntax-error #f msg stx expr))]
          [class-name (if name-id
                          (syntax-e name-id)
                          (let ([s (syntax-local-infer-name stx)])
                            (if (syntax? s)
                                (syntax-e s)
                                s)))])
      ; ------ Basic syntax checks -----
      (for-each (lambda (stx)
                  (syntax-case stx (-init init-rest -field -init-field inherit-field
                                          private public override augride
                                          public-final override-final augment-final
                                          pubment overment augment
                                          rename-super inherit inherit/super inherit/inner rename-inner
                                          inspect)
                    [(form orig idp ...)
                     (and (identifier? #'form)
                          (or (free-identifier=? #'form (quote-syntax -init))
                              (free-identifier=? #'form (quote-syntax -init-field))))]))))))
```

+ 900 lines

What's not so good

```
; main : stx bool stx          id  id stxs stxs -> stx
(define (main stx trace-flag super-expr
              deserialize-id-expr name-id
              interface-exprs defn-and-exprs)
```



```
(let-values ([(this-id) #'this-id]
              [(the-obj) (datum->syntax (quote-syntax here) (gensym 'self))]
              [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))])
  (let* ([def-ctx (syntax-local-make-definition-context)]
         [localized-map (make-bound-identifier-mapping)]
         [any-localized? #f]
         [localize/set-flag (lambda (id)
                              (let ([id2 (localize id)])
                                (unless (eq? id id2)
                                  (set! any-localized? #t))
                                id2))]
         [bind-local-id (lambda (id)
                          (let ([l (localize/set-flag id)])
                            (syntax-local-bind-syntaxes (list id) #f def-ctx)
                            (bound-identifier-mapping-put!
                             localized-map
                             id
                             l)))]
         [lookup-localize (lambda (id)
                            (bound-identifier-mapping-get
                             localized-map
                             id
                             (lambda ()
                               ; If internal & external names are distinguished,
                               ; we need to fall back to localize:
                               (localize id))))])
    ; ----- Expand definitions -----
    (let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)]
          [bad (lambda (msg expr)
                 (raise-syntax-error #f msg stx expr))]
          [class-name (if name-id
                          (syntax-e name-id)
                          (let ([s (syntax-local-infer-name stx)])
                            (if (syntax? s)
                                (syntax-e s)
                                s)))])
      ; ------ Basic syntax checks -----
      (for-each (lambda (stx)
                  (syntax-case stx (-init init-rest -field -init-field inherit-field
                                          private public override augride
                                          public-final override-final augment-final
                                          pubment overment augment
                                          rename-super inherit inherit/super inherit/inner rename-inner
                                          inspect)
                    [(form orig idp ...)
                     (and (identifier? #'form)
                          (or (free-identifier=? #'form (quote-syntax -init))
                              (free-identifier=? #'form (quote-syntax -init-field))))]))))))))
```

+ 900 lines

What's not so good

```
; main : stx bool stx (or #f id) id stxs stxs -> stx
(define (main stx trace-flag super-expr
              deserialize-id-expr name-id
              interface-exprs defn-and-exprs)
```

```
(let-values ([(this-id) #'this-id]
              [(the-obj) (datum->syntax (quote-syntax here) (gensym 'self))]
              [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))])
  (let* ([def-ctx (syntax-local-make-definition-context)]
         [localized-map (make-bound-identifier-mapping)]
         [any-localized? #f]
         [localize/set-flag (lambda (id)
                              (let ([id2 (localize id)])
                                (unless (eq? id id2)
                                  (set! any-localized? #t))
                                id2))]
         [bind-local-id (lambda (id)
                          (let ([l (localize/set-flag id)])
                            (syntax-local-bind-syntaxes (list id) #f def-ctx)
                            (bound-identifier-mapping-put!
                             localized-map
                             id
                             l)))]
         [lookup-localize (lambda (id)
                            (bound-identifier-mapping-get
                             localized-map
                             id
                             (lambda ()
                               ; If internal & external names are distinguished,
                               ; we need to fall back to localize:
                               (localize id))))])
    ; ----- Expand definitions -----
    (let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)]
          [bad (lambda (msg expr)
                 (raise-syntax-error #f msg stx expr))]
          [class-name (if name-id
                          (syntax-e name-id)
                          (let ([s (syntax-local-infer-name stx)])
                            (if (syntax? s)
                                (syntax-e s)
                                s)))])
      ; ------ Basic syntax checks -----
      (for-each (lambda (stx)
                  (syntax-case stx (-init init-rest -field -init-field inherit-field
                                          private public override augride
                                          public-final override-final augment-final
                                          pubment overment augment
                                          rename-super inherit inherit/super inherit/inner rename-inner
                                          inspect)
                    [(form orig idp ...)
                     (and (identifier? #'form)
                          (or (free-identifier=? #'form (quote-syntax -init))
                              (free-identifier=? #'form (quote-syntax -init-field))))]))))))
```

+ 900 lines

Enter Typed Racket

What's not so good

```
(: main (Stx Bool Stx (U  #f Id) Id Stxs Stxs -> Stx))
(define (main stx trace-flag super-expr
              deserialize-id-expr name-id
              interface-exprs defn-and-exprs)
```

```
(let-values ([(this-id) #'this-id]
              [(the-obj) (datum->syntax (quote-syntax here) (gensym 'self))]
              [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))])
  (let* ([def-ctx (syntax-local-make-definition-context)]
         [localized-map (make-bound-identifier-mapping)]
         [any-localized? #f]
         [localize/set-flag (lambda (id)
                              (let ([id2 (localize id)])
                                (unless (eq? id id2)
                                  (set! any-localized? #t))
                                id2))]
         [bind-local-id (lambda (id)
                          (let ([l (localize/set-flag id)])
                            (syntax-local-bind-syntaxes (list id) #f def-ctx)
                            (bound-identifier-mapping-put!
                             localized-map
                             id
                             l)))]
         [lookup-localize (lambda (id)
                            (bound-identifier-mapping-get
                             localized-map
                             id
                             (lambda ()
                               ; If internal & external names are distinguished,
                               ; we need to fall back to localize:
                               (localize id))))])
    ; ----- Expand definitions -----
    (let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)]
          [bad (lambda (msg expr)
                 (raise-syntax-error #f msg stx expr))]
          [class-name (if name-id
                          (syntax-e name-id)
                          (let ([s (syntax-local-infer-name stx)])
                            (if (syntax? s)
                                (syntax-e s)
                                s)))])
      ; ------ Basic syntax checks -----
      (for-each (lambda (stx)
                  (syntax-case stx (-init init-rest -field -init-field inherit-field
                                          private public override augride
                                          public-final override-final augment-final
                                          pubment overment augment
                                          rename-super inherit inherit/super inherit/inner rename-inner
                                          inspect)
                    [(form orig idp ...)
                     (and (identifier? #'form)
                          (or (free-identifier=? #'form (quote-syntax -init))
                              (free-identifier=? #'form (quote-syntax -init-field))))]))))))
```

+ 900 lines

Typed Racket Goals

Typed sister language to Racket

Sound interoperability with untyped code

Easy porting of existing programs and idioms

Typed Racket in 3 Slides

Hello World

```
#lang        racket                                    hello

(printf "Hello World\n")
```

Hello World

```
#lang typed/racket                                    hello

(printf "Hello World\n")
```

Functions

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Functions

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Modules

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang        racket

(require ack)

(ack 2 3)
```

Modules

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang       racket

(require ack)

(ack 2 3)
```

Modules

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang typed/racket

(require [ack
          (Integer Integer -> Integer)])
(ack 2 3)
```

Modules

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

compute

```
#lang typed/racket

(require ack)

(ack 2 3)
```

Idiomatic Types

With Takikawa, Strickland, Felleisen
[POPL 08, ESOP 09, ICFP 10, OOPSLA 12, ESOP 13]

How do Racket programmers think?

Racket programs are not secretly Scala programs

How do Racket programmers think?

Ruby
Python
JavaScript
Lua

Racket programs are not secretly

Java
ML
Scala programs
Haskell
C++

How do Racket programmers think?

Ruby
Python
JavaScript
Lua

Java
ML
Scala
Haskell
C++

Racket programs are not secretly programs

Consider the native idioms of a language

Types for Racket Idioms

occur

```
#lang typed/racket

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
```

Types for Racket Idioms

union

```
#lang typed/racket

(define-type Peano (U 'Zero (List 'S Peano)))
(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

Types for Racket Idioms

varar

```
#lang typed/racket

(: wrap (∀ (B A ...)
           ((A ... -> B) -> (A ... -> B))))
(define (wrap f)
  (lambda args
    (printf "args are: ~a\n" args)
    (apply f args)))
```

Mixins in Racket

racket-esq

```
#lang       racket
```

```
; add REPL functions to `base-class`
(define (esq-mixin base-class)
  (class base-class
    (super-new)
    (inherit insert last-position get-text erase)

    (define/public (new-prompt) ...)
    (define/public (output s) ...)
    (define/public (reset) ...)))
```

Mixins in Typed Racket

racket-esq

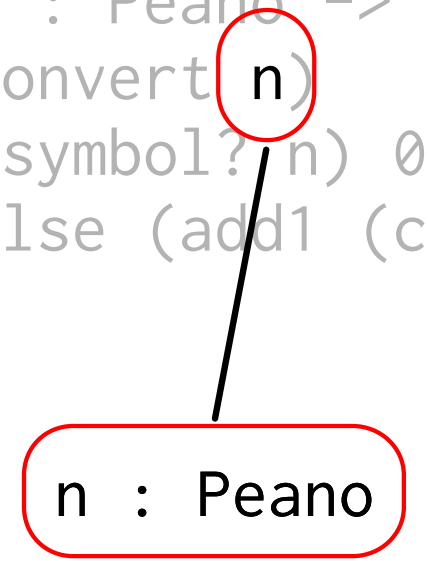
```
#lang typed/racket

(define-type Esq-Text%
  (Class #:implements Text%
         [new-prompt (-> Void)]
         [output (String -> Void)]
         [reset (-> Void)]))
(: esq-mixin
   (All (r #:row)
        (-> (Class #:row-var r #:implements Text%)
            (Class #:row-var r #:implements Esq-Text%))))
; add REPL functions to `base-class`
(define (esq-mixin base-class)
  (class base-class
    (super-new)
    (inherit insert last-position get-text erase)

    (define/public (new-prompt) ...)
    (define/public (output s) ...)
    (define/public (reset) ...)))
```

Occurrence Typing, Informally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

Occurrence Typing, Informally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```



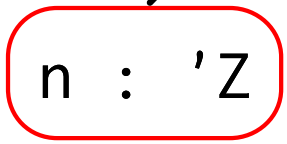
Occurrence Typing, Informally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

n : Peano

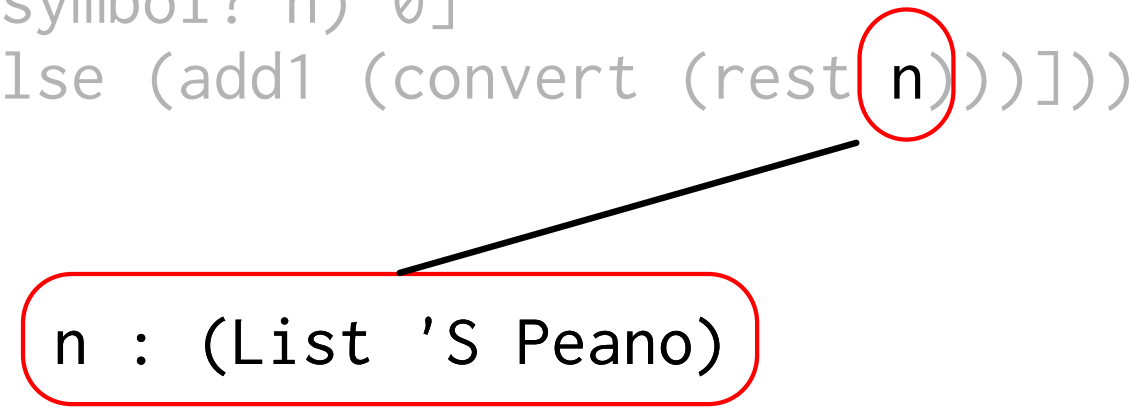
Occurrence Typing, Informally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```



Occurrence Typing, Informally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

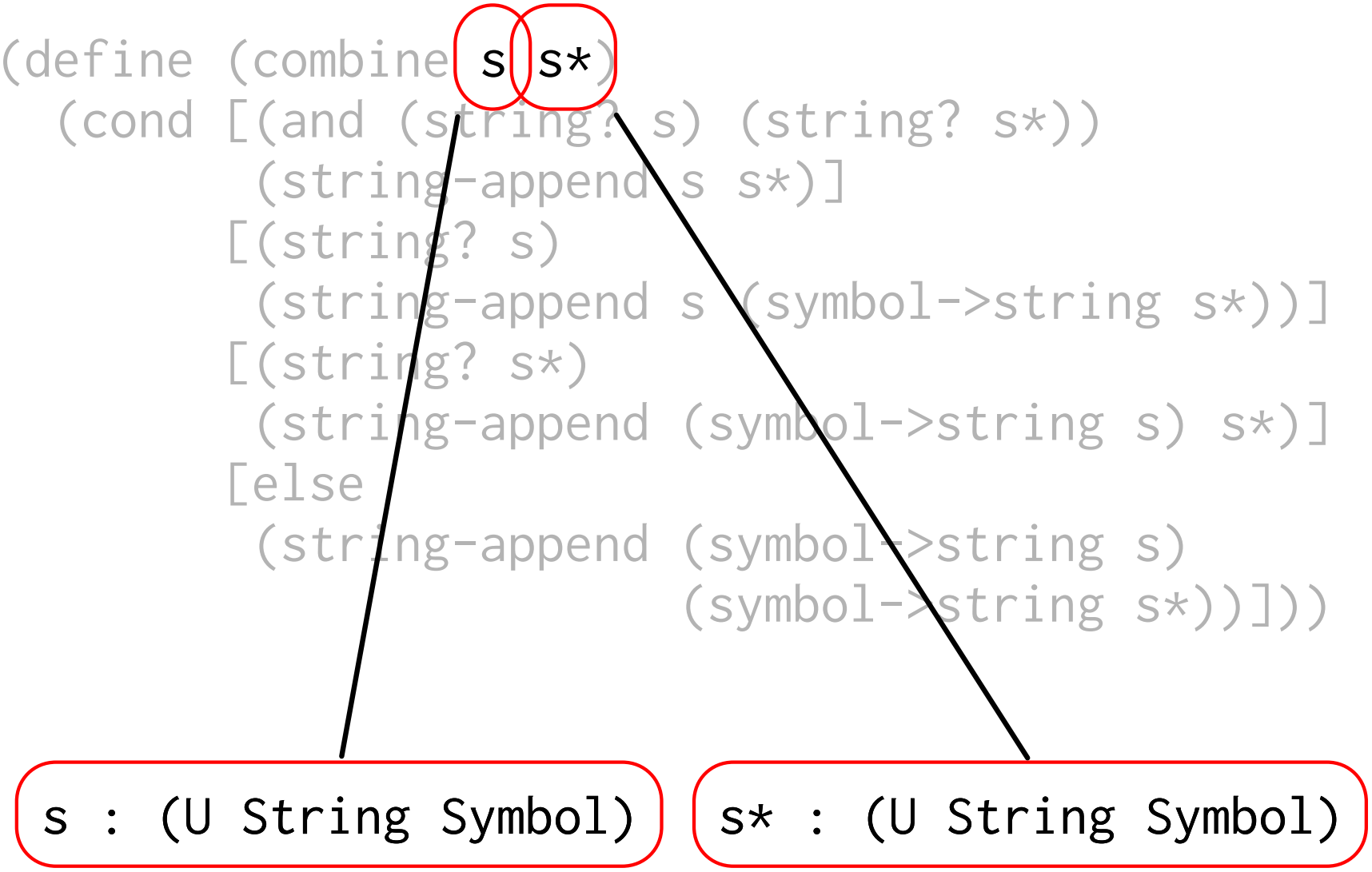


Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```

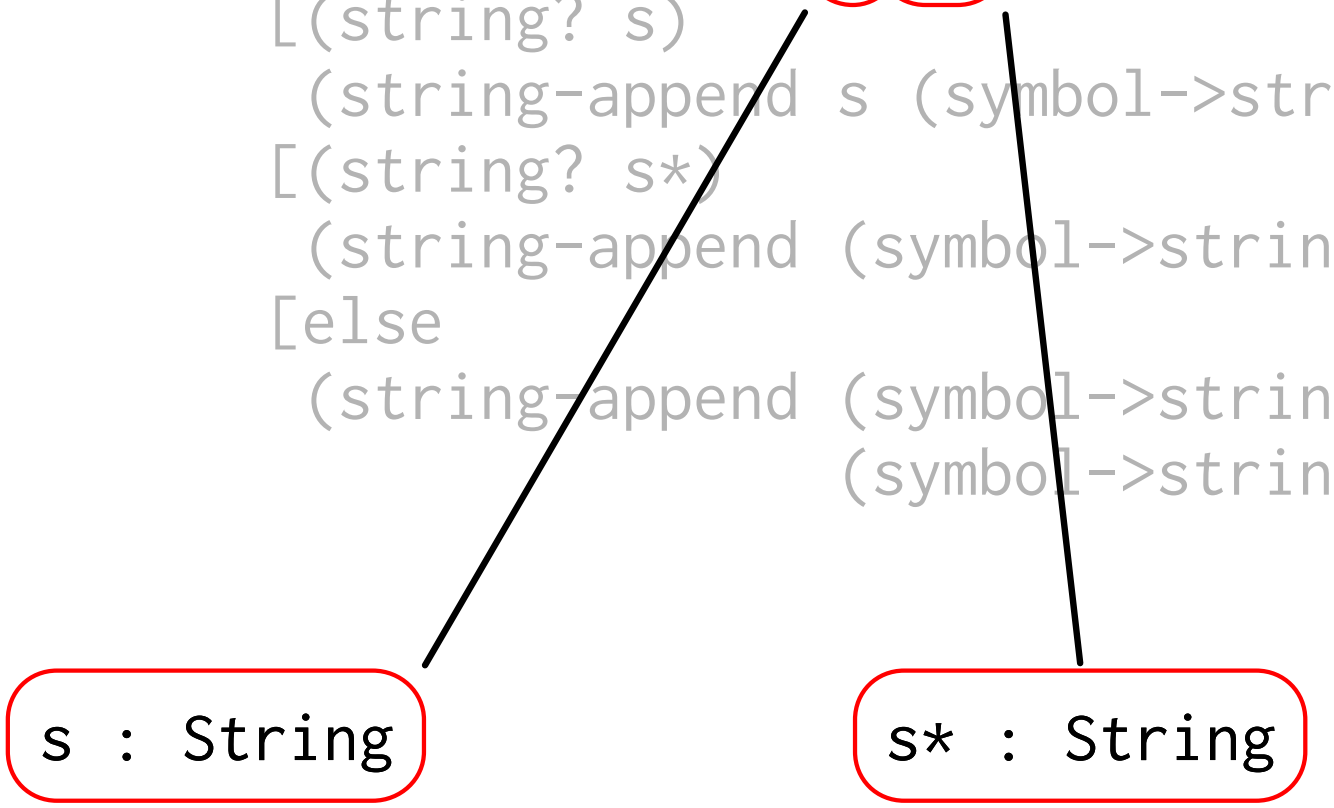
Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```



Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```



Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```

s : String

s* : Symbol

Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```

s : Symbol

s* : String

Occurrence Typing, Informally

```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(define (combine s s*)
  (cond [(and (string? s) (string? s*))
         (string-append s s*)]
        [(string? s)
         (string-append s (symbol->string s*))]
        [(string? s*)
         (string-append (symbol->string s) s*)]
        [else
         (string-append (symbol->string s)
                        (symbol->string s*))]))
```

s : Symbol

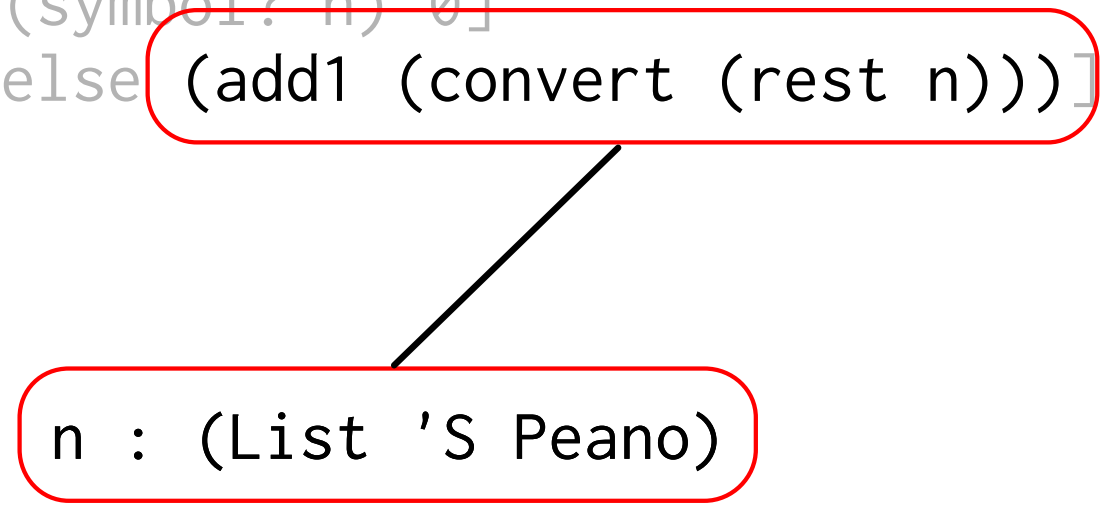
s* : Symbol

Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

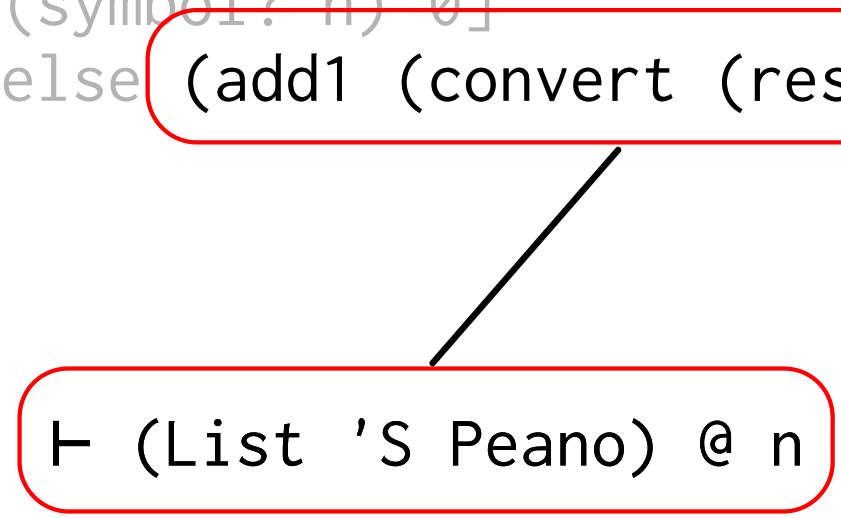
Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```



Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```



Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

$$\frac{\vdash \overline{\texttt{Symbol @ n}}}{\vdash \texttt{(List 'S Peano) @ n}}$$

Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

$$\frac{\vdash \texttt{Peano @ n} \qquad \vdash \overline{\texttt{Symbol @ n}}}{\vdash \texttt{(List 'S Peano) @ n}}$$

Occurrence Typing, Formally

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

$$\frac{\vdash \texttt{(U 'Z (List 'S Peano))} \ @\ n \qquad \vdash \overline{\texttt{Symbol} \ @\ n}}{\vdash \texttt{(List 'S Peano)} \ @\ n}$$

Lessons

Existing idioms are a source of type system ideas

Repeated in TypeScript, Typed Clojure, Hack, ...

Effective Contracts

With Takikawa, Strickland, Flatt, Findler, Felleisen
[DLS 06, ESOP 13, OOPSLA 12]

Typed & Untyped

server

```
#lang typed/racket

(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

client

```
#lang       racket

(require server)
(add5 7)
```

Typed & Untyped

Untyped code can make mistakes

server

```
#lang typed/racket

(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

client

```
#lang        racket

(require server)
(add5 "seven")
```

Typed & Untyped

Untyped code can make mistakes

server

```
#lang typed/racket

(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

client

```
#lang        racket

(require server)
(add5 "seven")
```

+: expects type <number> as 1st argument

Typed & Untyped

Catch errors dynamically at the boundary

server

```
#lang typed/racket

(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

client

```
#lang       racket

(require server)
(add5 "seven")
```

client broke the contract on add5

Typed & Untyped

Catch errors dynamically at the boundary

server

```
#lang        racket

(define (add5 x) "x plus 5")
```

client

```
#lang typed/racket

(require server
         [add5 (Number -> Number)])
(add5 7)
```

server interface broke the contract on add5

Typed & Untyped

Catch errors dynamically at the boundary

server

```
#lang typed/racket

(: addx (Number -> (Number -> Number)))
(define (addx x) (lambda (y) (+ x y)))
```

client

```
#lang       racket

(require server)
((addx 7) 'bad)
```

client broke the contract on add5

Contracts for functions

server

```
#lang typed/racket

(: addx (Number -> (Number -> Number)))

(define (addx x) (lambda (y) (+ x y)))
```

Contracts for functions

server

```
#lang        racket

(provide/contract
 [addx (-> number? (-> number? number?))])
(define (addx x) (lambda (y) (+ x y)))
```

Contracts for functions

server

```
#lang        racket

(provide addx-c)
(define (addx-c x)
  (if (number? x)
      (contract (addx x) (-> number? number?))
      (error "blame the client")))
(define (addx x) (lambda (y) (+ x y)))
```

Contracts for vectors

server

```
#lang typed/racket

(provide primes)
(: primes : (Vectorof Integer))
(define primes (vector 2 3 5 7 11))
```

Contracts for vectors

```
#lang          racket                    server

(provide/contract
 [primes (vector/c integer?)])
(define primes (vector 2 3 5 7 11))
```

But how does `vector/c` work?

Chaperones

```
#lang        racket                                  vector/c

(chaperone-vector
 primes
 (lambda (v i res)
   (unless (number? res) (error "blame"))
   res)
 ...)
```

Chaperones

```
#lang        racket                                  vector/c

(chaperone-vector
 primes
 (lambda (v i res)
   (unless (number? res) (error "blame"))
   17)
 ...)
```

Is this ok?

The Chaperone Invariant

A chaperoned value behaves like the original value, but with extra errors.

Chaperones vs Impersonators

Chaperones

- Less expressive
- Apply to more values

Impersonators

- No invariants
- Only apply to mutable values

Further Extension

Classes, Mixins, Objects

Delimited Continuations

Abstract Data Types

Channels and Events

Lessons

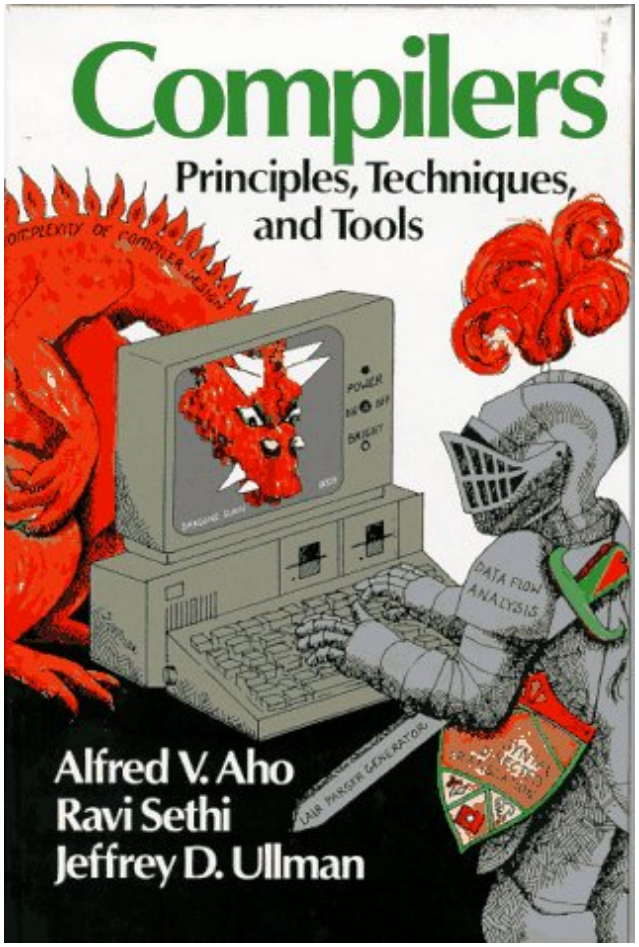
Proxy mechanisms must be expressive while respecting invariants

Now applied in JavaScript proxies

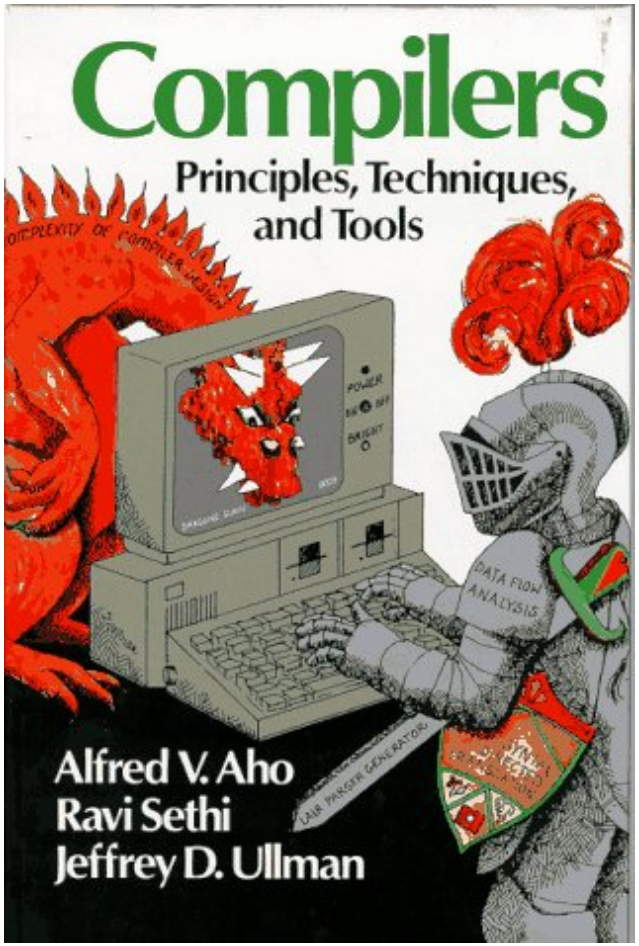
Extensible Languages, Extensible Compilers

With Culpepper, St-Amour, Flatt, Felleisen
[SFP 08, PLDI 11]

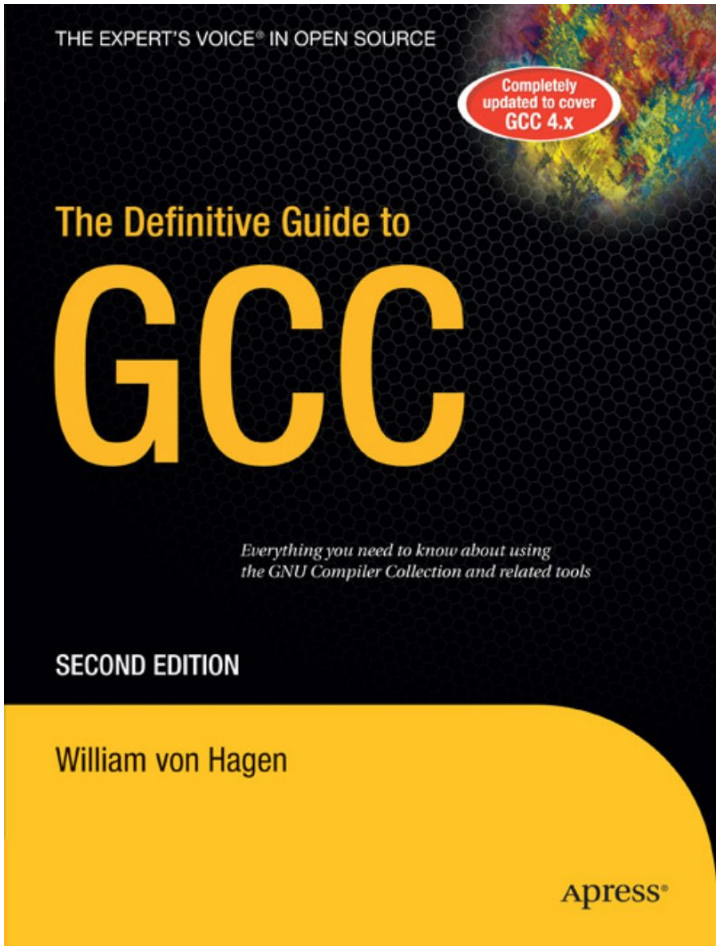
The Traditional Approach to Compilers



The Traditional Approach to Compilers



Produces impressive results



The Macro Approach to Compilers

```
(define-syntax and
  (syntax-parser
   [(_ e1 e2)
    #'(if e1 e2 #f)]))
```

The Macro Approach to Compilers

```
(define-syntax and
  (syntax-parser
   [(_ e1 e2)
    #'(if e1 e2 #f)]))
```

Supports linguistic reuse

Scoping

`if`

…

Functions

Classes

Modules

The Typed Racket approach:

Linguistic reuse of the macro approach

- Re-uses existing language extensions

Capabilities of the traditional approach

- Including typechecker and optimizer

The Typed Racket approach:

Linguistic reuse of the macro approach

- Re-uses existing language extensions

Capabilities of the traditional approach

- Including typechecker and optimizer

By exposing compiler tools to library authors

Static Checking

ack

```
#lang        racket


 ;  ack : Integer Integer -> Integer
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3)
```

Static Checking

ack

```
#lang typed/racket


(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Static Checking

ack

```
#lang typed/racket

(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Type checking is a *global* process

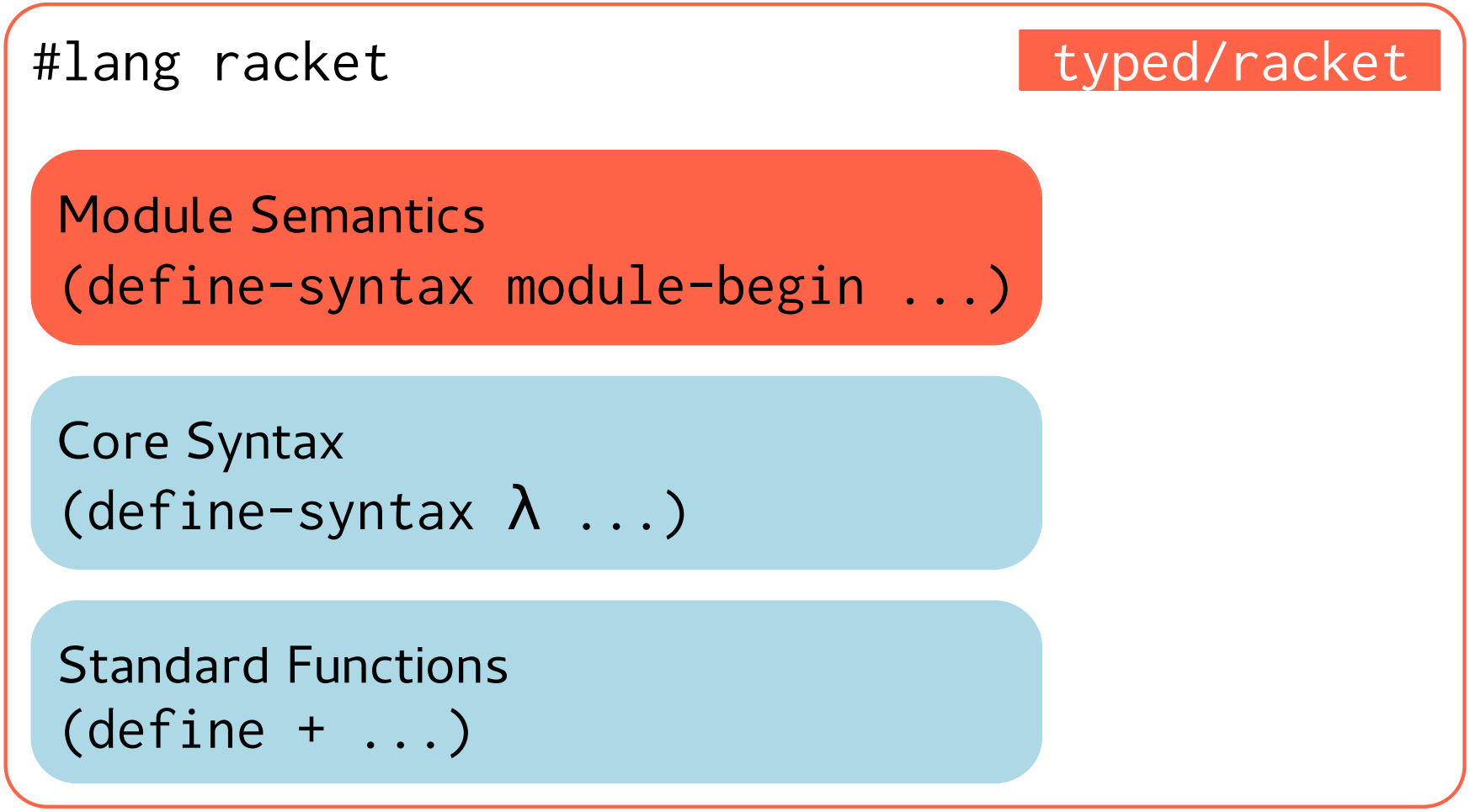
module-begin

ack

```
#lang typed/racket

(module-begin
 (: ack : Integer Integer -> Integer)
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3))
```

Languages control the whole module

Implementing a language



Implementing a language

typed/racket

```
#lang racket

(define-syntax module-begin
  (syntax-parser
   [(_ forms ...)
    (for ([form #'(forms ...)])
      (typecheck form))

    #'(forms ...)]))
```

The Typechecker

typechecker

```
#lang racket

(define (typecheck form)
  (syntax-parse form
    [v:identifier
     ...]
    [(λ args body)
     ...]
    [(define v body)
     ...]

    ... other syntactic forms ...))
```

Why Intermediate Languages?

"The compiler serves a broader set of programmers than it would if it only supported one source language"

— Chris Lattner

Why Intermediate Languages?

Most forms come from libraries

```
(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

Why Intermediate Languages?

Most forms come from libraries

```
(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

Also: pattern matching, keyword arguments,

classes, loops, comprehensions, any many more

- Can’t know static semantics ahead of time

Core Racket

Racket defines a common subset that expansion targets

```
expr ::= identifier
         (plain-lambda args expr)
         (app expr ...+)
         …
         a dozen core expressions
```

```
def ::= expr
        (define-values ids expr)
        (require spec)
        …
```

local-expand

typed/racket

```
#lang racket

(define-syntax module-begin
  (syntax-parser
   [(_ forms ...)
    (define expanded-forms
          (local-expand #'(forms ...)))
    (for ([form expanded-forms])
      (typecheck form))

    expanded-forms]))
```

The Revised Typechecker

typechecker

```
#lang racket

(define (typecheck form)
  (syntax-parse form
    [v:identifier
     ...]
    [(plain-lambda args body)
     ...]
    [(define-values vs body)
     ...]

    ... two dozen core forms ...))
```

Adding Optimization

typed/racket

```
#lang racket

(define-syntax module-begin
  (syntax-parser
   [(_ forms ...)
    (define expanded-forms
          (local-expand #'(forms ...)))
    (for ([form expanded-forms])
      (typecheck form))
    (define opt-forms (map optimize expanded-forms))
    opt-forms]))
```

Adding Optimization

Problem: generic arithmetic is slow

```
(: norm : Float Float -> Float)
(define (norm x y)
  (sqrt (+ (sqr x) (sqr y))))
```

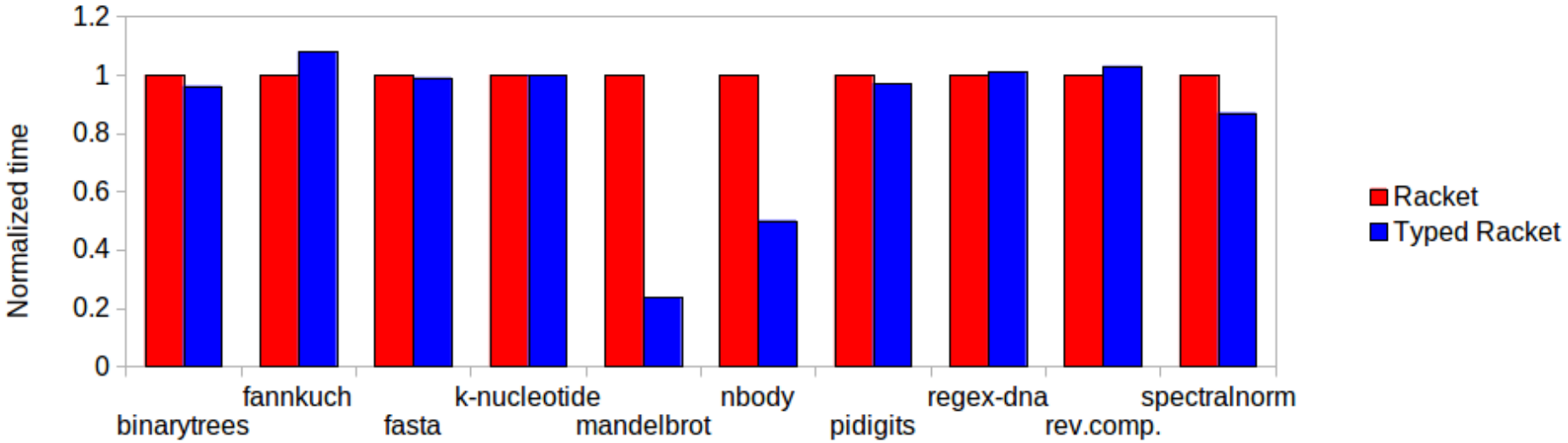
Adding Optimization

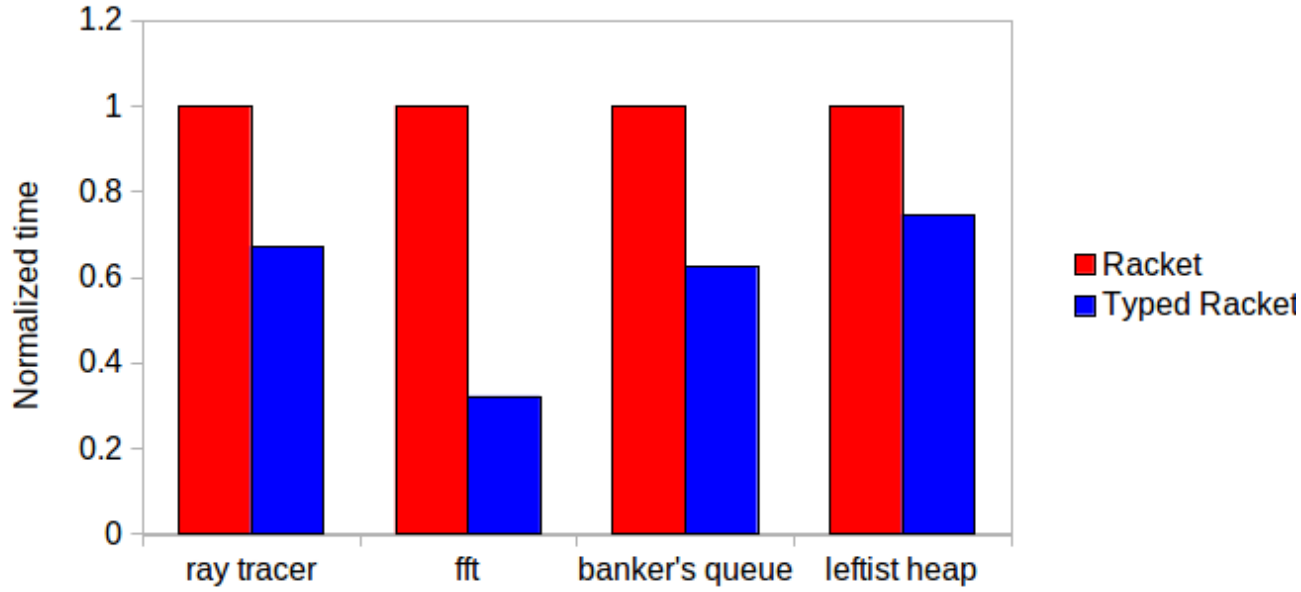
Express guarantees as rewritings

```
(: norm : Float Float -> Float)
(define (norm x y)
  (unsafe-flsqrt
   (unsafe-fl+ (unsafe-fl* x x)
               (unsafe-fl* y y))))
```

Low-level operations expose code generation to libraries

Performance results





Lessons

Language extensibility makes compilers into libraries

See LMS, Scala macros, ...

And more ...

With St-Amour, Dimoulas, Felleisen
[DLS 06, ESOP 12, OOPSLA 12]

Proofs and Techniques

If the program raises a contract error, the blame is not assigned to a typed module.

Proofs and Techniques

Well-typed modules can’t get blamed.

Proofs and Techniques

Allows local reasoning about typed modules, without changing untyped modules.

Choose how much static checking you want.

Proofs and Techniques

Closely connected to contract semantics

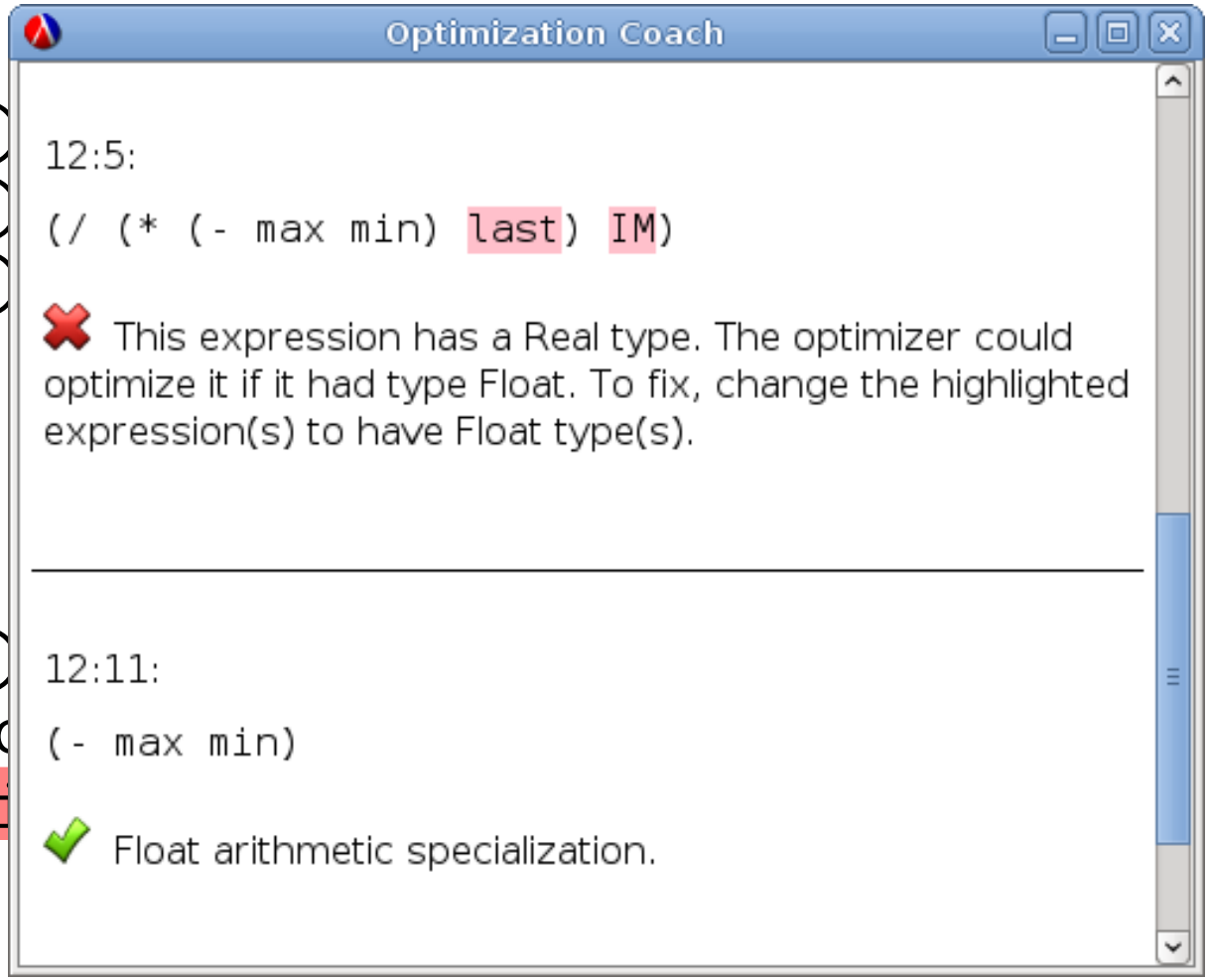
Proved by showing that all communication is monitored

Developer Tools

```
(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define min  35.3)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (+ (/ (* (- max min) last) IM) min))
```

Developer Tools

```
(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define min  35.3)
(define max  156.8)
(define (gen-random)
  (set! last (modulo
  (+ (/ (* (- max mi
```



Developer Tools

```
(define IM   139968)
(define IA     3877)
(define IC    29573)

(define last 42)
(define min  35.3)
(define max  156.8)
(define (gen-random)
  (set! last (modulo (+ (* last IA) IC) IM))
  (+ (/ (* (- max min) (fx->fl last)) (fx->fl IM)) min))
```

3x speedup

Future Challenges

Polymorphic Contracts

poly

```
#lang        racket

(define (id x)
  (cond [(number? x)
         (+ x 1)]
        [else x]))
```

checked

```
#lang typed/racket

(require/typed poly [id (All (a) a -> a)])
(id 5)
```

A clear error

Polymorphic Contracts

poly

```
#lang        racket

(define (id x)
  (cond [(number? x)
         (display x) x]
        [else x]))
```

checked

```
#lang typed/racket

(require/typed poly [id (All (a) a -> a)])
(id 5)
```

What should this do?

New Compiler Techniques

Meta-tracing makes a fast Racket

Carl Friedrich Bolz
Software Development Group
King's College London
cfbolz@gmx.de

Tobias Pape
Hasso-Plattner-Institut
Potsdam
Tobias.Pape@hpi.uni-potsdam.de

Jeremy Siek
Indiana University
jsiek@indiana.edu

Sam Tobin-Hochstadt
Indiana University
samth@cs.indiana.edu

ABSTRACT

Tracing just-in-time (JIT) compilers record and optimize the instruction sequences they observe at runtime. With some modifications, a tracing JIT can perform well even when the executed program is itself an interpreter, an approach called meta-tracing. The advantage of meta-tracing is that it separates the concern of JIT compilation from language implementation, enabling the same JIT compiler to be used with many different languages. The RPython meta-tracing JIT compiler has enabled the efficient interpretation of several dynamic languages including Python (PyPy), Prolog, and Smalltalk. In this paper we present initial findings in applying the RPython JIT to Racket. Racket comes from the Scheme family of programming languages for which there are mature static optimizing compilers. We present the result of spending just a couple person-months implementing and tuning an implementation of Racket written in RPython. The results are promising, with a geometric mean equal to Racket's performance and within a factor of 2 slower than Gambit and Larceny on a collection of standard Scheme benchmarks. The results on individual benchmarks vary widely. On the positive side, our interpreter is sometimes up to two to five times faster than Gambit, three times faster than Larceny, and two or-

1. INTRODUCTION

Just-in-time (JIT) compilation has been applied to a wide variety languages, with early examples including Lisp, APL, Basic, Fortran, Smalltalk, and Self [Aycock, 2003]. These days, JIT compilation is mainstream; it is responsible for running both server-side Java applications [Paleczny et al., 2001] and client-side JavaScript applications in web browsers [Hölttä, 2013].

Mitchell [1970] observed that one could obtain an instruction sequence from an interpreter by recording its actions. The interpreter can then jump to this instruction sequence, this *trace*, when it returns to interpret the same part of the program. For if-then-else statements, there is a trace for only one branch. For the other branch, Mitchell suggests reinvoking the interpreter. Bala et al. [2000] applied tracing JIT compilation to optimize native assembly code and Gal et al. [2006] showed that tracing JIT compilers can efficiently execute object-oriented programs, which feature control flow that is highly dynamic and data-dependent.

Developing a just-in-time compiler is traditionally a complex undertaking. However, Bolz et al. [2009] developed *meta-tracing*, an approach that can significantly reduce the development cost for tracing JIT compilers. With meta-tracing, the language implementer

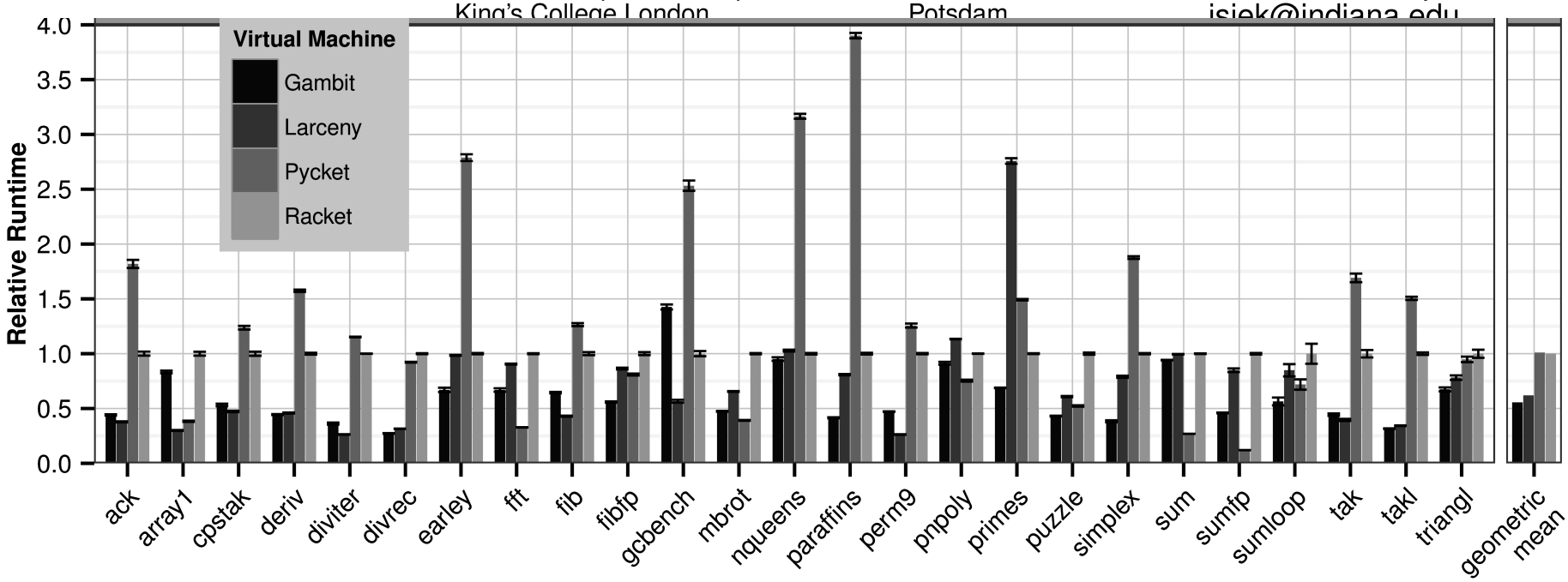
New Compiler Techniques

Meta-tracing makes a fast Racket

Carl Friedrich Bolz
Software Development Group
King's College London

Tobias Pape
Hasso-Plattner-Institut
Potsdam

Jeremy Siek
Indiana University
jsiek@indiana.edu



optimizing compilers. We present the result of spending just a couple person-months implementing and tuning an implementation of Racket written in RPython. The results are promising, with a geometric mean equal to Racket's performance and within a factor of 2 slower than Gambit and Larceny on a collection of standard Scheme benchmarks. The results on individual benchmarks vary widely. On the positive side, our interpreter is sometimes up to two to five times

ing JIT compilation to optimize native assembly code and Gal et al. [2006] showed that tracing JIT compilers can efficiently execute object-oriented programs, which feature control flow that is highly dynamic and data-dependent.

Developing a just-in-time compiler is traditionally a complex undertaking. However, Bolz et al. [2009] developed *meta-tracing*, an approach that can significantly reduce the development cost for tracing JIT compilers. With meta-tracing, the language implementer

Typed Racket is not just a nice language
... it's also informing PL research at every level

Runtimes, proofs, compilers, metaprogramming, contracts

Typed Racket is not just a nice language
... it's also informing PL research at every level

Runtimes, proofs, compilers, metaprogramming, contracts

Thank you

`samth.github.io` `racket-lang.org`