The PL Renaissance
The PL Renaissance
The PL Renaissance
What’s good

These languages are

- interactive
- designed for rapid development
- supported by an active community
- modular
- higher-order

And they’re exciting!
(define (main stx trace-flag super-expr deserialize-id-expr name-id interface-exprs defn-and-exprs)

(let-values ([(this-id) #©]
    (define (stx (datum-syntax (quote-syntax here) (gensym 'self)))
      (define (trace-flag (datum-syntax (quote-syntax here) (gensym 'find-self)))
        (let* ([localized-map (make-bound-identifier-mapping)]
             [any-localized? #f]
             [localize/set-flag (lambda (id)
                                 (let ([id (localize id)])
                                   (set! any-localized? #t)
                                   id))]
             [bind-local-id (lambda (id)
                             (let ([id (localize/set-flag id)])
                               (bound-identifier-mapping-get!
                                 localized-map id)])]
             [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 msg stx expr))]
    [class-name (if name-id
                   (syntax-e name-id)
                   (let ([(syntax-local-infer-name stx)])
                     [if (syntax? a)
                        (syntax-e a)
                        (lambda ())])])

; ----- Basic syntax checks ----- (for-each (lambda (ets)
    (syntax-case stx (-init init-rest -field -init-field inherit-field private public override super override/override augment/override augment super inherit/super inherit/inherit super inherit/inner inherit/inner augment/inner inherit/inner augment/inner augment super inherit/super inherit/inherit inner rename/inner rename/inner inspect)
      (if (form orig idp ...)
          [and (identifier? #form)
               (or (free-identifier? #form)
                   (free-identifier? #form (quote-syntax -init)))
               (free-identifier? #form (quote-syntax -init-field)))])))

+ 900 lines

What's not so good
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)
               (stx-obj) (datum->syntax (quote-syntax here) (gensym 'self))
               (the-obj) (datum->syntax (quote-syntax here) (gensym 'find-self))
               (def-cts (syntax-local-make-definition-context))
               (localized-map (make-bound-identifier-mapping))
               (any-localized? #f)
               (localize/set-flag (lambda (id)
                                   (let* ([def-ctx (syntax-local-make-definition-context)]
                                        [localized-map (make-bound-identifier-mapping)]
                                        [any-localized? #f]
                                        [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))))))))
               (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)
                                    (quote-syntax nil))))))

; ----- Expand definitions -----
(let ((defn-and-exprs (expand-all-forms stx defn-and-exprs def-cts bind-local-id)
       (raise-syntax-error #f meg stx expr)))
  (class-name (if name-id
                 (syntax-e name-id)
                 (let ([s (syntax-local-infer-name stx)])
                   (if (syntax? s)
                       (syntax-e s)
                       (quote-syntax nil))))))

; ----- Basic syntax checks -----
(for-each (lambda (ets)
            (syntax-case stx (-init rest -field -init-field inherit-field
                              private public override suprise
                              public-final override-final augment-final
                              public monotone augment
                              rename-super inherit/super inherit/inner rename/inner
                              inspect)
              (form orig lid ...)
              (and (identifier? #form)
                   (or (free-identifier? #form (quote-syntax -init))
                       (free-identifier? #form (quote-syntax -init-field))))))))

+ 900 lines
What's not so good

+ 900 lines

; 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)
  (this-obj) (datum->syntax (quote-syntax here) (gensym 'self)))
(let* ((def-ctx (syntax-local-make-definition-context))
  [localized-map (make-bound-identifier-mapping)]
  [any-localized? #f]
  [localize/set-flag (lambda (id)
    (let ([id (localize id)])
      (unless (eq? (syntactic? id) #f)
        (any-localized? id)))]
  [bind-local-id (lambda (id)
    (let ([id (syntax-local-set-flag id)])
      (syntax-local-bind-syntaxes (list id) #f def-ctx)
      (bound-identifier-mapping-get!
        localized-map id)])]
  [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)
            #f)]))]

; ----- Basic syntax checks -----
(for-each (lambda (ets)
  (syntax-case stx (init rest field -field init-field inherit-field
    private public override suprise
    public-final override-final augment-final
    public-maint inherit/maint super/maint inherit/inner
    rename/maint super/maint super/innerrename/maint super/inner
    inspect))
    [form orig idp ...]
    [and (identifier?) #form]
    [or (identifier?) #form (quote-syntax -init)]
    [free-identifier? #form (quote-syntax -init)]))

})
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 ([stxs (stx)])
  (stx->syntax (quote-syntax here) (gensym 'self))
  (stx->syntax (quote-syntax here) (gensym 'find-self))
  (let* ([def-cts (syntax-local-make-definition-context])
    [localized-map (make-bound-identifier-mapping)]
    [any-localized? #f]
    [localize/set-flag (lambda (id)
      (let ([(id (localize id))]
        (set! any-localized? #t))
      id))]
    [bind-local-id (lambda (id)
      (let ([(id (localize/set-flag id))]
        (syntax-local-bind-syntaxes (list id) def-cts)
        (bound-identifier-mapping-put! localized-map id)]))]
    [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-cts 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)
        (syntax-e s)))]))

; ------ Basic syntax checks -------
(for-each (lambda (ets)
  (syntax-case stx (-init init-rest -field -init-field inherit-field
data private-public override-super override-final augment-final
public-final override-final augment-final
public-override augment
rename-super inherit/super
rename-super inherit/super 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 (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) #f)
               [(trace-id) (datum->syntax (quote-syntax here) (gensym 'self))]
               [(the-finder) (datum->syntax (quote-syntax here) (gensym 'find-self))]
               [localized-map (make-bound-identifier-mapping)]
               [(any-localized? #f)]
               [(localize/set-flag (lambda (id)
                                     (let ([id (localize id)])
                                         (set! any-localized? #t)))
                          #f)]
               [(bind-local-id (lambda (id)
                                (let ([l (localize/set-flag id)])
                                   (syntax-local-bind-syntaxes (list id) #f
                                     (bound-identifier-mapping-get!
                                      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-cts 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 (ets)
              (syntax-case stx (-init init-rest -field -init-field inherit-field private public override super)
                private public override final augment-final public override augment super
                rename-super inherit/super inherit/inner rename/inner inspect)
              [form orig idp ...]
              [(and identifier? #form)
               (or (free-identifier? #form)
                   (free-identifier? #form (quote-syntax -init)))]
              )) )

+ 900 lines
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.
Why PLT Scheme?

- Modules
- Contracts
- Abstractions
Typed Scheme in 3 Slides
#lang scheme

(printf "Hello World\n")
#lang typed-scheme

(printf "Hello World\n")
#lang scheme

; 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)
#lang typed-scheme

(: 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)
#lang scheme

; 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))))]))

#lang scheme

(require ack)

(ack 2 3)
#lang typed-scheme

(: 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)))]))

#lang scheme

(require ack)

(ack 2 3)
#lang    scheme

; 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)))])
)

#lang typed-scheme

(require [ack
  (Integer Integer -> Integer)])

(ack 2 3)
#lang typed-scheme

(: 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))))]))

#lang typed-scheme

(require ack)

(ack 2 3)
Sound Interoperation
Typed & Untyped

### Server Code

```scheme
#lang typed-scheme  
(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

### Client Code

```scheme
#lang scheme  
(require server)
(add5 7)
```
Untyped code can make mistakes

```scheme
#lang typed-scheme
(: add5 (Number -> Number))
(define (add5 x) (+ x 5))
```

```scheme
#lang scheme
(require server)
(add5 "seven")
```
Untyped code can make mistakes

```scheme
#lang typed-scheme

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

#lang scheme

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

`+:` expects type `<number>` as 1st argument
Catch errors dynamically at the boundary

```scheme
#lang typed-scheme

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

```scheme
#lang scheme

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

client broke the contract on add5
Catch errors dynamically at the boundary

```
#lang scheme

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

```
#lang typed-scheme

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

(add5 7)
```

*server interface broke the contract on add5*
Catch errors dynamically at the boundary

```
#lang typed-scheme

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

```
#lang scheme

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

*client broke the contract on add5*
The Blame Theorem

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

Well-typed modules can’t get blamed.
The Blame Theorem

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

Choose how much static checking you want.
Types for Scheme

Occurrence Typing

Variable-Arity

Refinement Types

Ad-Hoc Data
#lang typed-scheme

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

(: check (String -> (Refinement sql-safe?)))
(define (check s)
  (if (sql-safe? s)
      s
      (error "unsafe string!")))
#lang typed-scheme

(define-type-alias BT
  (U Number (Pair BT BT)))

(: sizeof (BT -> Number))

(define (sizeof b)
  (if (number? b)
      1
      (+ 1 (sizeof (car b)) (sizeof (cdr b))))))
#lang typed-scheme

( : wrap ( ∀ (B A ...)  
       ((A ... → B) → (A ... → B)) ) )

(define (wrap f)
  (lambda args
    (printf "args are: ~a\n" args)
    (apply f args)))
#lang scheme

(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

(: f (Any -> Number))
(define (f x)  type: Any
  (if (number? x)  number?
    (add1 x)
    0))
#lang typed-scheme

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

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

type: Number
#lang typed-scheme

(: f (Any -> Number))

(define (f x)
  (if (number? x)
    (add1 x)
    0))

type: (Any -> Boolean : Number)
#lang typed-scheme

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

(: f (Any -> Number))
(define (f x)
  (if (number? x)
      (add1 x)
      0))
#lang typed-scheme

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

filter: Number

type: Boolean
#lang typed-scheme

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

env: x:Any + Number_x
#lang typed-scheme

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

env: x:Number

env: x:Number

env: x:Number

env: x:Number

env: x:Number
;; s is a symbol, number or string
(define (->string s)
  (cond
   [(symbol? s) (symbol->string s)]
   [(number? s) (number->string s)]
   [else s])))
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))

(define (->string s)
  (cond [[(symbol? s) (symbol->string s)]
         [(number? s) (number->string s)]
         [else s]]))
Then & Else

```scheme
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String)
(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
       [(number? s) (number->string s)]
       [else s]))
```

53
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String) ->string
  (define (->string s)
    (cond [(symbol? s) (symbol->string s)]
          [(number? s) (number->string s)]
          [else s])))
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))

(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s]))

type: (U Number String)
Then & Else

#lang typed-scheme

(:  ->string ((U Symbol Number String)  ->  String))
(define (->string s)
  (cond 
    [(symbol? s) (symbol->string s)]
    [(number? s) (number->string s)]
    [else s]))

type: Number
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))
(define (->string s)
  (cond 
      [(symbol? s) (symbol->string s)]
      [(number? s) (number->string s)]
      [else s]))

%type: String
#lang typed-scheme

(: —>string ( (U Symbol Number String) —> String ))

(define (—>string s)
  (cond [(symbol? s) (symbol—>string s)]
        [(number? s) (number—>string s)]
        [else s]))
#lang typed-scheme

(: →string ((U Symbol Number String) → String))

(define (→string s)
  (cond [(symbol? s) (symbol→string s)]
        [(number? s) (number→string s)]
        [else s])))
#lang typed-scheme

(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s])))
#lang typed-scheme

(: →string ((U Symbol Number String) → String))

(define (→string s)
  (cond [(symbol? s) (symbol→string s)]
        [(number? s) (number→string s)]
        [else s])))
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))

(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s])))

inv: s:(U Symbol Number String) + Symbols
#lang typed-scheme

(: →string ((U Symbol Number String) → String))
(define →string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s]))

env: s:(U Number String)
#lang typed-scheme

(: ➞ string ((U Symbol Number String) ➞ String))

(define ➞ string s)
  (cond
    [(symbol? s) (symbol->string s)]
    [(number? s) (number->string s)]
    [else s])

**type:** (Any ➞ Boolean : Number | Number)
#lang typed-scheme

#:->string ((U Symbol Number String) -> String))
(define #:->string s)
  (cond "["(symbol? s) (symbol->string s)]
    [(number? s) (number->string s)]
    [else s]])
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))

(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s]))

env: s:(U Number String) + Numbers
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))

(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s]))

env: s:Number
#lang typed-scheme

(: ->string ((U Symbol Number String) -> String))
(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s])))

eval: s:(U Number String) + Number_s
#lang typed-scheme

( : → string ( ( U Symbol Number String ) → String ) )

(define ( → string s )
  ( cond [ ( symbol? s ) ( symbol→ string s ) ]
      [ ( number? s ) ( number→ string s ) ]
      [ else s ] ))

env: s: String
#lang scheme

; g : Any (U String Number) -> Number
(define (g x y)
  (cond [(and (number? x) (string? y))
        (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y]))
#lang typed-scheme

(: g (Any (U Number String) -> Number))

(define (g x y)
  (cond [(and (number? x) (string? y))
        (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y])))
Logical Reasoning

```scheme
#lang typed-scheme
(: g (Any (U Number String) -> Number))
(define (g x y)
  (cond [(and
           (number? x)
           (filter: Number x)
           (string? y)
           (string-length y))
        (+ x (string-length y))]
        [(number? x)
         (+ x y)]
        [else y]))
```
#lang typed-scheme

(: g (Any (U Number String) -> Number))

(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y])))
#lang typed-scheme

(filter: Number \_ \ Function \_ String \_ \ String)

(define (g x y)
  (cond 
    [(and (number? x) (string? y))
      (+ x (string-length y))]
    [(number? x) (+ x y)]
    [else y])))
Logical Reasoning

```scheme
#lang typed-scheme
(filter: Number_x String_y | Number_x ⊇ String_y)
(define (g x y)
  (cond [(and (number? x) (string? y))
        (+ x (string-length y))]
      [(number? x) (+ x y)]
      [else y]))
```
#lang typed-scheme
(: g (Any (U Number String) -> Number))
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y]))

env: Number_x ⊇ String_y
filter: Number_x
#lang typed-scheme
((: g (Any (U Number String) -> Number)))
(define (g x y)
  (cond [(and (number? x) (string? y))
          (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y])))

env: Number_{x} \supset String_{y}
filter: Number_{x} String_{y}
Logical Reasoning

```scheme
#lang typed-scheme
(: g (Any (U Number String) -> Number))
(define (g x y)
  (cond [(and (number? x) (string? y))
        (+ x (string-length y))]
        [(number? x) (+ x y)]
        [else y]))
```

env: x:Any    y:(U Number String) + Numberx Stringy
Logical Reasoning

```scheme
#lang typed-scheme
(: g (Any (U Number String) -> Number))
(define (g x y)
  (cond [(and (number? x) (string? y))
    (+ x (string-length y))]
    [(number? x) (+ x y)]
    [else y]))
env: x:Number y:Number
```
#lang typed-scheme

( : f ((U Number String) (Pair Any Any) -> Number))

(define (f input extra)
  (cond
   [(and (number? input) (number? (car extra)))
    (+ input (car extra))]
   [(number? (car extra))
    (+ (string-length input) (car extra))]
   [else 0])))
Easy Integration
Implementation Validation
#lang typed-scheme

(: subtitle-pict : (String -> Pict))
(define (subtitle-pict s)
  (text s (current-title-font) large-text-size))
<table>
<thead>
<tr>
<th>Lines</th>
<th>Squad Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>System</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2369</td>
<td>511</td>
<td>407</td>
<td>315</td>
<td>1290</td>
<td>618</td>
</tr>
<tr>
<td>Increase</td>
<td>7%</td>
<td>25%</td>
<td>7%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Problems (Bad)</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 5510

%: 7%
Sample Fixes

```
#lang scheme

(+ 10 (string->number str))
```
#lang typed-scheme

(+ 10 (assert (string->number str)))
(define (divs . args)
  (* -1 (apply / args)))
#lang typed-scheme

(define (divs arg . args)
  (* -1 (apply / arg args)))
Sample Problems

```scheme
#lang scheme

(cond [(< x 0) 'negative]
      [(= x 0) 'zero]
      [(> x 0) 'positive])
```
Sample Problems

#lang typed-scheme

(cond [(< x 0) 'negative]
      [(= x 0) 'zero]
      [else 'positive])
(define pr (make-pair x y))
(when (string? (pair-left pr))
  (set-pair-left! pr (string->symbol (pair-left pr))))
#lang typed-scheme

(define pr (make-pair
    (if (string? x) (string->number x) x)
    y))
Related Work
Interlanguage Integration

Professor J

Gray et al. (2005)

Multilanguage Systems

Matthews and Findler (2007)
Types for Untyped Languages

John Reynolds (1968)

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

Soft Typing

Types for Scheme

Strongtalk
Types for Untyped Languages

John Reynolds (1968)
Soft Typing
  Wright (1997), Flanagan (1999)
Types for Scheme
Strongtalk
Types for Untyped Languages

John Reynolds (1968)
Soft Typing
Types for Scheme

  SPS (Wand 1984), Leavens (2005)

  Infer (Haynes 1995)

Strongtalk
Types for Untyped Languages

John Reynolds (1968)
Soft Typing
Types for Scheme
Strongtalk

Bracha and Griswold (1993)
Contracts & Modules

Contracts

Findler & Fellesien (2002)

Modules with Macros

Flatt (2002)
Recent Work

Gradual Typing


DRuby

Furr et al (2009)
Conclusion
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.
Sound Typed-Untyped Interoperation
Type System for Scheme
Full-scale Implementation
Empirical Validation
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Olin Shivers
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Aaron Turon
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Felix Klock
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to James Hamblin
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to James Jungbauer
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Dan Brown
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Lazlo Babai
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Matthew Flatt
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Stevie Strickland
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Guy Steele
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Robby Findler
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Sukyoung Ryu
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Carl Eastlund
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Eric Allen
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Ivan Gazeau
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Katie Edmonds
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Mitch Wand
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Dave Herman
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Jan-Willem Maessen
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Matthias Felleisen
Try Typed Scheme

Installer and Documentation

http://www.plt-scheme.org

Thanks to Ryan Culpepper
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Vincent St-Amour
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Jesse Tov
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Christine Flood
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Steve Hochstadt
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to David Chase
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Elizabeth Tobin
Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Thanks to Victor Luchangco
Try Typed Scheme

Installer and Documentation

http://www.plt-scheme.org

Thanks to Ryan Culpepper
Proposal

Occurrence Typing
  - Done, and more

Variable-arity Polymorphism
  - Done

Keyword and Optional Arguments
  - Done: Use & Import/Export
  - Not Done: Definition

Validation
  - Some Done