Typed Scheme

From Scripts to Programs

Sam Tobin-Hochstadt
Northeastern University

The PL Renaissance

The PL Renaissance

python

The PL Renaissance

python

What’s good

These languages are

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

And they’re exciting!

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

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

Thesis

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.

Thesis

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.

Thesis

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.

Thesis

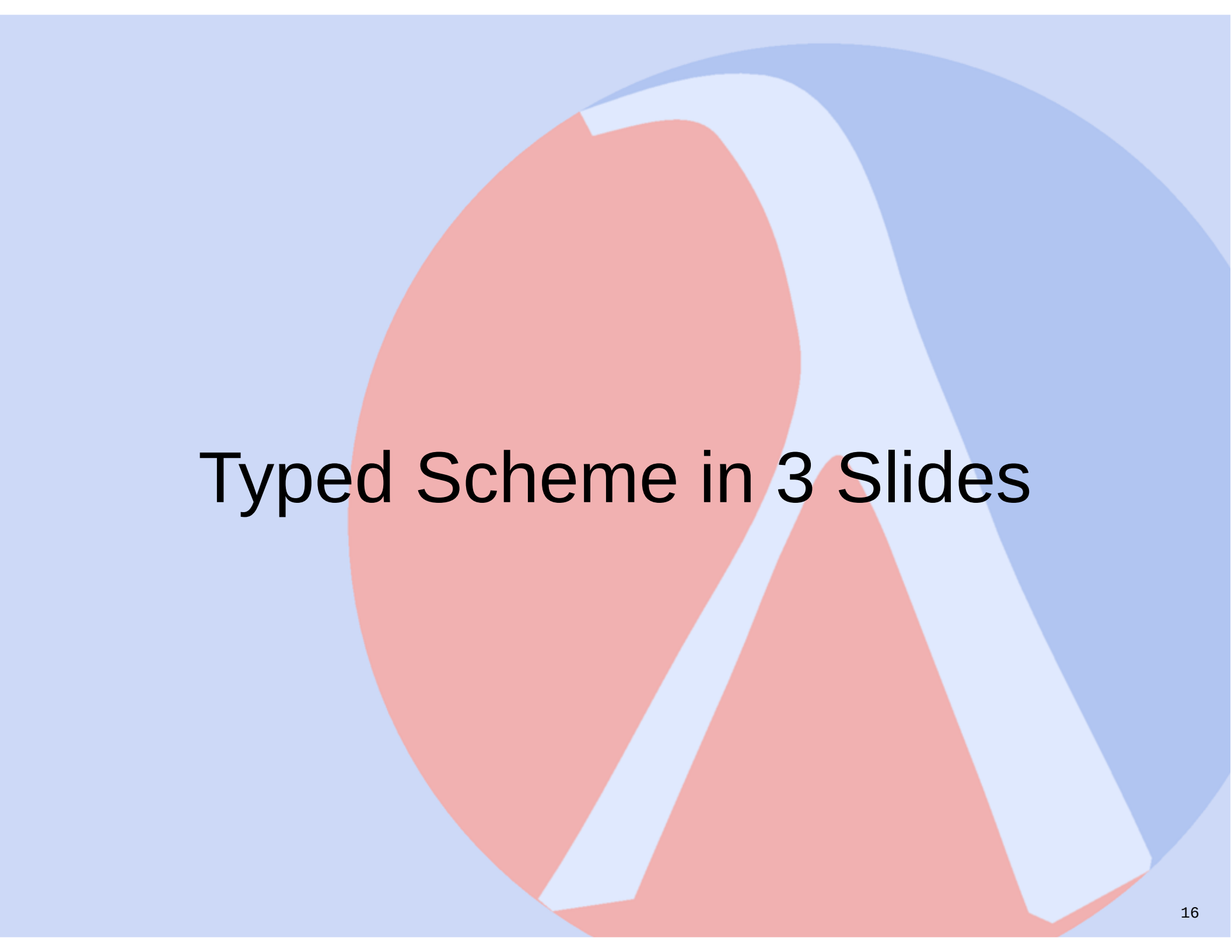
Module-by-module porting of code from an untyped language to a typed sister language allows for an easy transition from untyped scripts to typed programs.

Why PLT Scheme?

Modules

Contracts

Abstractions


Typed Scheme in 3 Slides

Hello World

```
#lang        scheme                                    hello

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

Hello World

```
#lang typed-scheme                                    hello

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

ack

```
#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)
```

ack

```
#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)
```

Modules

ack

```
#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)))]))
```

compute

```
#lang        scheme

(require ack)

(ack 2 3)
```

ack

```
#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)))]))
```

compute

```
#lang       scheme

(require ack)

(ack 2 3)
```

ack

```
#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)))]))
```

compute

```
#lang typed-scheme

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

Modules

ack

```
#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)))]))
```

compute

```
#lang typed-scheme

(require ack)

(ack 2 3)
```

Sound Interoperation

Typed & Untyped

server

```
#lang typed-scheme

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

client

```
#lang        scheme

(require server)
(add5 7)
```

Untyped code can make mistakes

server

```
#lang typed-scheme

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

client

```
#lang       scheme

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

Untyped code can make mistakes

server

```
#lang typed-scheme

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

client

```
#lang       scheme

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

+: expects type <number> as 1st argument

Catch errors dynamically at the boundary

server

```
#lang typed-scheme

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

client

```
#lang       scheme

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

client broke the contract on add5

Catch errors dynamically at the boundary

server

```
#lang          scheme

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

client

```
#lang typed-scheme

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

server interface broke the contract on add5

Catch errors dynamically at the boundary

server

```
#lang typed-scheme

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

client

```
#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

occur

```
#lang typed-scheme

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

refine

```
#lang typed-scheme

(: check (String -> (Refinement sql-safe?)))
(define (check s)
  (if (sql-safe? s)
      s
      (error "unsafe string!")))
```

union

```
#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)))))
```

varar

```
#lang typed-scheme

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

Scheme Idioms

```
#lang        scheme                                  number?


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

Filters & Objects

```
#lang typed-scheme

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

number?

Filters & Objects

```
#lang typed-scheme

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

type: Any

number?

Filters & Objects

```
#lang typed-scheme

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

number?

type: Any

```
#lang typed-scheme

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

number?

type: Number

Filters & Objects

```
#lang typed-scheme

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

number?

type: (Any -> Boolean : Number)

Filters & Objects

```
#lang typed-scheme

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

number?

type: Any
object: x

Filters & Objects

```
#lang typed-scheme

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

number?

type: Boolean
filter: (apply-filter Number x)

Filters & Objects

```
#lang typed-scheme

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

number?

type: Boolean
filter: Number_x

Filters & Objects

```
#lang typed-scheme

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

number?

env: x:Any + Number$_x$

Filters & Objects

```
#lang typed-scheme

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

number?

env: x:Number
type: Number

Scheme Idioms

```
#lang        scheme                                          else

; s is a symbol, number or string
(define (->string s)
  (cond [(symbol? s) (symbol->string s)]
        [(number? s) (number->string s)]
        [else s]))
```

Then & Else

```
#lang typed-scheme                                    else

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

Then & Else

```
#lang typed-scheme

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

else

type: (U Symbol Number String)

Then & Else

```
#lang typed-scheme

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

else

type: Symbol

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]))
```

else

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]))
```

else

type: Number

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]))
```

else

type: String

Then & Else

```
#lang typed-scheme

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

```
type: (Any -> Boolean : Symbol | Symbol
```

else

Then & Else

```
#lang typed-scheme

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

else

type: Boolean
filter: $\text{Symbol}_s \mid \overline{\text{Symbol}_s}$

Then & Else

```
#lang typed-scheme

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

else

env: s:(U Symbol Number String) + Symbol_s

Then & Else

```
#lang typed-scheme                                          else

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

env: s:Symbol

Then & Else

```
#lang typed-scheme                                    else

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

nv: s:(U Symbol Number String) + $\overline{\text{Symbol}_s}$

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]))
```

else

env: s:(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]))
```

else

type: (Any -> Boolean : Number | Number

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]))
```

else

type: Boolean
filter: $\text{Number}_s \mid \overline{\text{Number}_s}$

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]))
```

else

env: s:(U Number String) + Number_s

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]))
```

else

env: s:Number

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]))
```

else

env: s:(U Number String) + $\overline{\text{Number}_s}$

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]))
```

else

env: s:String

and

```
#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]))
```

Logical Reasoning

```
#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]))
```

and

Logical Reasoning

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

filter: $\text{Number}_x \mid \overline{\text{Number}_x}$

and

Logical Reasoning

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

and

filter: $\text{String}_y \mid \overline{\text{String}_y}$

Logical Reasoning

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

and

filter: Number$_x$ String$_y$ |

Logical Reasoning

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

and

filter: $\text{Number}_x\ \text{String}_y \mid \text{Number}_x \supset \overline{\text{String}_y}$

Logical Reasoning

```
#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]))
```

and

env: $\text{Number}_x \supset \overline{\text{String}_y}$

filter: Number_x

Logical Reasoning

```
#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]))
```

and

env: $\text{Number}_x \supset \overline{\text{String}_y}$

filter: $\text{Number}_x \ \overline{\text{String}_y}$

Logical Reasoning

```
#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]))
```

and

env: x:Any y:(U Number String) + $\text{Number}_x\ \overline{\text{String}_y}$

Logical Reasoning

```
#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]))
```

and

env: x:Number y:Number

All Together Now

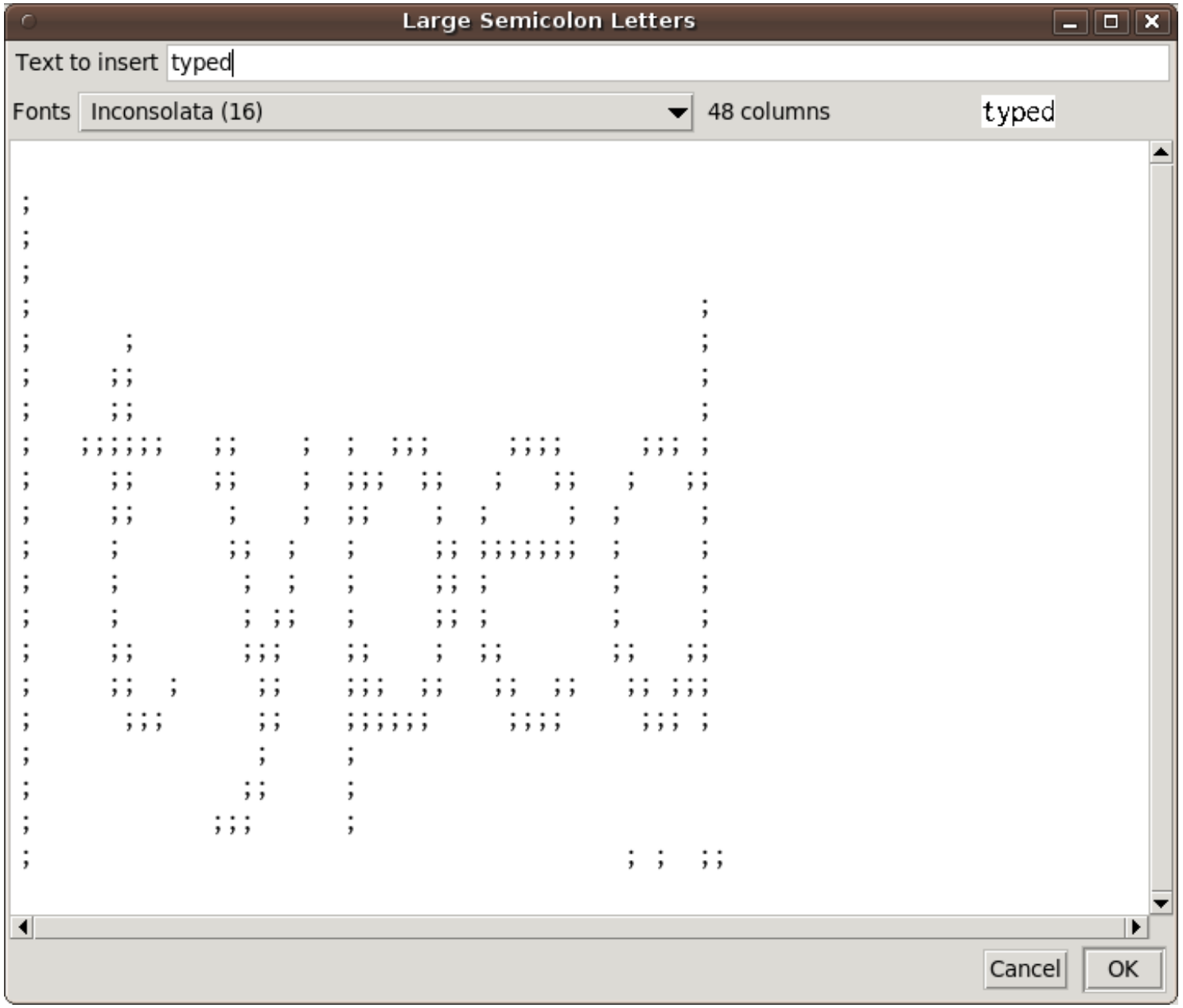
```
#lang typed-scheme

(: 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

Implementation



Implementation

```
#lang typed-scheme                                    tslide

(: subtitle-pict : (String -> Pict))
(define (subtitle-pict s)
  (text s (current-title-font) large-text-size))
```

Validation

	Squad	Metrics	Acct	Spam	System	Rand	Total
Lines	**2369**	**511**	**407**	**315**	**1290**	**618**	**5510**
Increase	**7%**	**25%**	**7%**	**6%**	**1%**	**3%**	**7%**
Fixes (Good)	**5**	**3**	**4**	**5**	**8**	**0**	**25**
Problems (Bad)	**7**	**4**	**3**	**1**	**0**	**1**	**16**

Sample Fixes

```
#lang          scheme                    assert

(+ 10 (string->number str))
```

Sample Fixes

```
#lang typed-scheme                    assert

(+ 10 (assert (string->number str)))
```

Sample Fixes

```
#lang          scheme                          div

(define (divs . args)
  (* -1 (apply / args)))
```

Sample Fixes

```
#lang typed-scheme                                    div

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

Sample Problems

cond

```
#lang        scheme

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

Sample Problems

cond

```
#lang typed-scheme

(cond [(< x 0) 'negative]
      [(= x 0) 'zero]
      [else    'positive])
```

Sample Problems

```
#lang        scheme                                          mutate

(define pr (make-pair x y))
(when (string? (pair-left pr))
  (set-pair-left! pr (string->symbol (pair-left pr))))
```

Sample Problems

mutate

```
#lang typed-scheme

(define pr (make-pair
            (if (string? x) (string->number x) x)
            y))
```

Related Work

Interlanguage Integration

ProfessorJ

- 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

- Fagan (1991), Aiken (1994)
- 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

- Siek et al (2006-2009), Wadler & Findler (2007), Herman et al (2007)

DRuby

- Furr et al (2009)

Conclusion

Thesis

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.

Support

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

- Occurence Typing
  - Done, and more
- Variable-arity Polymorphism
  - Done
- Keyword and Optional Arguments
  - Done: Use & Import/Export
  - Not Done: Definition
- Validation
  - Some Done