The Design and Implementation of Typed Scheme

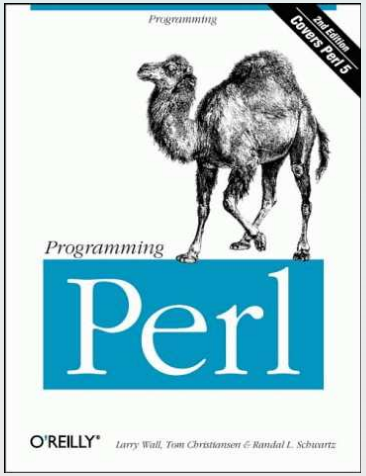
Sam Tobin-Hochstadt and Matthias Felleisen

Northeastern University

The PL Renaissance

The PL Renaissance

Programming languages are flourishing



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 ()
                             (localize id))))])
  (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)))])
   (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 ()
                              (localize id))))])
  (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)))])
   (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

Program Evolution

How can we make our script evolve into a mature program?

By adding statically checked design information piece by piece.

What do we need

1. Module by module migration
2. Easy integration with untyped code
3. Sound guarantees from the type system
4. Avoid rewriting code

Typed Scheme

Why PLT Scheme?

PLT Scheme is a scripting language.

- Untyped, Modular, Built to script libraries

We are facing the same dilemma.

- Lots of untyped code to maintain

Why PLT Scheme?

PLT Scheme has advantages for implementing program evolution.

- Modules [Flatt 02]
- Software Contracts [Findler 02]

Why PLT Scheme?

Lessons from PLT Scheme apply elsewhere.

Modular migration

1. Module by module migration

Hello World in PLT Scheme

print

```
#lang scheme

(printf "Hello World")
```

Modular migration

1. Module by module migration

Hello World in Typed Scheme

```
#lang typed-scheme

(printf "Hello World")
```

print

Modular migration

1. Module by module migration

Simple Arithmetic in PLT Scheme

arith

```
#lang scheme


(define (sq x)
  (* x x))
```

Modular migration

1. Module by module migration

Simple Arithmetic in Typed Scheme

arith

```
#lang typed-scheme

(: sq (Num → Num))
(define (sq x)
  (* x x))
```

Modular migration

1. Module by module migration

Multi-module programs

arith

```
#lang typed-scheme

(: sq (Num → Num))
(define (sq x)
  (* x x))
(provide sq)
```

run

```
#lang typed-scheme

(require arith)
(sq 11) ; => 121
```

Integration with plain PLT Scheme

2. Easy integration with untyped code

Typed to Untyped

arith

```
#lang typed-scheme

(: sq (Num → Num))
(define (sq x)
  (* x x))
(provide sq)
```

run

```
#lang scheme

(require arith)
(sq 11)
(sq "eleven")
```

Integration with plain PLT Scheme

3. Sound guarantees from the type system

Typed to Untyped

```
#lang typed-scheme                arith

(: sq (Num → Num))
(define (sq x)
  (* x x))
(provide sq)
```

```
#lang scheme                      run

(require arith)
(sq 11)
(sq "eleven") ; => contract violation
```

Integration with plain PLT Scheme

2. Easy integration with untyped code

Untyped to Typed

arith

```
#lang scheme


(define (sq x)
  (* x x))
(provide sq)
```

run

```
#lang typed-scheme

(require/typed sq (Num → Num) arith)
(sq 11)
(sq "eleven")
```

Integration with plain PLT Scheme

3. Sound guarantees from the type system

Untyped to Typed

```
#lang scheme                                   arith

(define (sq x)
  (* x x))
(provide sq)
```

```
#lang typed-scheme                             run

(require/typed sq (Num → Num) arith)
(sq 11)
(sq "eleven") ; => type error
```

4. Avoid rewriting code

PLT Scheme programmers do not write with any particular type system in mind.

So Typed Scheme must capture their informal reasoning.

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang scheme

; Shape = Position ∪ Circle ∪ Rectangle ∪ ...

; Shape → Num
; what is the area of shape s?
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang scheme

; Shape = Position ∪ Circle ∪ Rectangle ∪ ...

; Shape → Num
; what is the area of shape s?
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang scheme

; Shape = Position ∪ Circle ∪ Rectangle ∪ ...

; Shape → Num
; what is the area of shape s?
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang typed-scheme

(define-type-alias Shape
  (∪ Position Circle Rectangle ...))
(: shape-area (Shape → Num))
; what is the area of shape s?
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(define-type-alias Shape
  (∪ Position Circle Rectangle ...))
(: shape-area (Shape → Num))
; what is the area of shape s?
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Shape

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Shape

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Position

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Shape

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Circle

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

s : Circle

4. Avoid rewriting code

Occurrence Typing

```
#lang typed-scheme

(: shape-area (Shape → Num))
(define (shape-area s)
  (cond
    [(position? s) 0]
    [(circle? s) (* (sqr (circle-radius s) 2) pi)]
    ...))
```

circle? : (Any → Bool : Circle)

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang scheme

(map rectangle-area
        (filter rectangle? list-of-shapes))
```

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang typed-scheme

(map rectangle-area
        (filter rectangle? list-of-shapes))
```

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang typed-scheme

(map rectangle-area
     (filter rectangle? list-of-shapes))
```

rectangle? : (Any → Boolean : Rectangle)

4. Avoid rewriting code

How PLT Scheme programmers reason

```
#lang typed-scheme

(map rectangle-area
        (filter rectangle? list-of-shapes))
```

```
filter : (∀ (a b)
              ((Listof a) (Any → Bool : b)
               → (Listof b)))
```

4. Avoid rewriting code

- Polymorphism
- Union Types
- Recursive Types
- Structures
- Occurrence Typing

4. Avoid rewriting code

Polymorphism

Union Types

Recursive Types

Structures

Occurrence Typing

Refinement Types

Refinement Types

Any predicate can define a type

SQL

```
#lang typed-scheme

(: sql-safe? (String → Bool))
(define (sql-safe? s) ...)
(: query ((Refinement sql-safe?) → Result))
(define (query s)
  (string-append
   "SELECT from Data where k = " s ";"))
```

Refinement Types

Any predicate can define a type

```
#lang typed-scheme                                    SQL

(: sql-safe? (String → Bool))
(define (sql-safe? s) ...)
(: check (String → (Refinement sql-safe?)))
```

Refinement Types

Any predicate can define a type

SQL

```
#lang typed-scheme

(: sql-safe? (String → Bool))
(define (sql-safe? s) ...)
(: check (String → (Refinement sql-safe?)))
(define (check s)
  (if (sql-safe? s)
      s
      (error "not safe")))
```

Refinement Types

Any predicate can define a type

SQL

```
#lang typed-scheme

(: sql-safe? (String → Bool))
(define (sql-safe? s) ...)
(: check (String → (Refinement sql-safe?)))
(define (check s)
  (if (sql-safe? s)
      s
      (error "not safe")))
```

sql-safe? : (String -> Bool : (Refinement sql-safe?))

Refinement Types

Any predicate can define a type

SQL

```
#lang typed-scheme

(: sql-safe? (String → Bool))
(define (sql-safe? s) ...)
(: check (String → (Refinement sql-safe?)))
(define (check s)
  (if (sql-safe? s)
      s
      (error "not safe")))
```

x : (String -> Bool : (Refinement x))

Does it work?

Formal Validation

We have a formal model of Typed Scheme

- Enjoys standard soundness properties

We have an implementation of this model using PLT Redex

- 500 lines

We have a verified model in Isabelle/HOL

- 5000 lines

Real Validation

Implemented in PLT Scheme

- Support PLT Tools (Check Syntax, Debugger, etc)
- Integrates with macro and module system
- Standard libraries available

Real Validation

Ported thousands lines of existing code

- Games, scripts, libraries, educational code
- Even parts of DrScheme
- Not by the original author
- Very few changes to the code

Future Work

Exploiting Soft Typing

Soft typing attempted to typecheck untyped programs without programmer help.

This project was hindered by complex type languages and unpredictable errors.

We believe that using Typed Scheme as a target will help.

Onward to JavaScript?

Other scripting languages present new challenges.

- Object systems
- No contract systems
- Weak module systems

And new opportunities.

Thank You

Implementation, Documentation

`http://www.plt-scheme.org`

PLT Redex Model, Isabelle Model

`http://www.ccs.neu.edu/~samth`

Thanks to Ryan Culpepper, Matthew Flatt, Ivan Gazeau