TYPES FOR UNTYPED LANGUAGES

Matthias Felleisen (PLT)
Northeastern University

> “i see you're the keynote speaker at TYPES in language design and implementation. what the f...??”
>
> -- anonymous friend from TLDI, 06/nov/2009

> “i see you're the keynote speaker at TYPES in language design and implementation. what the f...??”
>
> -- anonymous friend from TLDI, 06/nov/2009

> types are bad for pl research.
>
> -- me, many times

“i see you're the keynote speaker at TYPES in language design and implementation. what the f...??”
-- anonymous friend from TLDI, 06/nov/2009

types are bad for pl research.
-- me, many times

if i ran a company, i’d use a typed programming language.
-- me, also many times

it turns out that PLT is like running a company: start with two major components, glue them together with a few lines, and pretty soon you have several widely used apps and you need to maintain them.

it turns out that PLT is like running a company: start with two major components, glue them together with a few lines, and pretty soon you have several widely used apps and you need to maintain them.

and, of course, we're not alone: the Swedish pension system runs on Perl -- as some proudly proclaim and others equally loudly bemoan

it turns out that PLT is like running a company: start with two major components, glue them together with a few lines, and pretty soon you have several widely used apps and you need to maintain them.

and, of course, we're not alone: the Swedish pension system runs on Perl -- as some proudly proclaim and others equally loudly bemoan

and now think how many other apps run on JavaScript, Lua, Perl, PHP, Python, Ruby (on Rails), and similar scripting languages

choice 1: make fun of these languages, their lack of type systems, their mistakes with LAMBDA etc, the lack of PL background of their designers, the thousands of programmers who use them ...

choice 1: make fun of these languages, their lack of type systems, their mistakes with LAMBDA etc, the lack of PL background of their designers, the thousands of programmers who use them ...

choice 2: take the problem seriously and create a *smooth path* to add types to their programs

choice 1: make fun of these languages, their lack of type systems, their mistakes with LAMBDA etc, the lack of PL background of their designers, the thousands of programmers who use them ...

choice 2: take the problem seriously and create a *smooth path* to add types to their programs

types for untyped languages:
gradually enrich "scripts"
with explicit and sound types
without changing code

Goal: explicit types for "scripts" via a mixed typed/untyped language

problem 1: programmers will convert only small pieces ("modules") into explicitly typed form, but this "mixed" programming language should be as type sound as a plain typed language -- what does type soundness mean?

Goal: explicit types for "scripts" via a mixed typed/untyped language

problem 1: programmers will convert only small pieces ("modules") into explicitly typed form, but this "mixed" programming language should be as type sound as a plain typed language -- what does type soundness mean?

problem 2: programmers will not wish to modify code to accommodate the type checker, so the type system must accommodate the existing idioms of the scripting language, though adding type declarations to functions, structures, methods, fields etc. is acceptable -- what makes a type system practical?

TYPE SOUNDNESS FOR MIXED PROGRAMS

T

```
#lang STLC

inc : int → int
inc(x) = x + 1

provide inc
```

U

```
#lang LC

require T

f(x) = ... inc(true) ...
```

T

```
#lang STLC

inc : int → int
inc(x) = x + 1

provide inc
```

U

```
#lang LC

require T

f(x) = ... inc(true) ...
```

T

```
#lang STLC

encode : (int → int) → int
encode(f) =  ... f(42) + 21 ...

provide encode
```

U

```
#lang LC

require T

f(x) = ... x + 1 ...  return "hello"

main(x) = ... encode(f) ...
```

T

```
#lang STLC

encode : (int → int) → int
encode(f) =  ... f(42) + 21 ...

provide encode
```

U

```
#lang LC

require T

f(x) = ... x + 1 ...  return "hello"

main(x) = ... encode(f) ...
```

T

```
#lang STLC

encode : (int → int) → i
encode(f) =  ... f(42) + 2 ...

provide encode
```

bang!

U

```
#lang LC

require T

f(x) = ... x + 1 ...  return "hello"

main(x) = ... encode(f) ...
```

U

```
#lang LC

inc(x) = x + 1

provide inc
```

T

```
#lang STLC

require U

f(x) = ... inc(???) .??? 10 ...
```

U

```
#lang LC

inc(x) = x + 1

provide inc
```

T

```
#lang STLC

require U

f(x) = ... inc(???) ??? 10 ...
```

U

```
#lang LC

inc(x) = x + 1

provide inc
```

T

```
#lang STLC

require U  [inc : int → int]

f(x) = ... inc(42) + 10 ...
```

U

```
#lang LC

inc(x) = if x = 0
           "hello"
         else
           x + 1

provide inc
```

T

```
#lang STLC

require U [inc : int → int]

f(x) = ... inc(42) + 10 ...
```

U

```
#lang LC

inc(x) = if x = 0
           "hello"
         else
           x + 1

provide inc
```

T

```
#lang STLC

require U [inc : int → int]

f(x) = ... inc(42) + 10 ...
```

U

```
#lang LC

inc(x) = if x = 0
           "hello"
         else
           x + 1

provide inc
```

T

```
#lang STLC

require U [inc : int →

f(x) = ... inc(42) + 10 ...
```

bang

step 1: typed ‘modules’ must specify types for all imported variables and specify types for all exports

step 1: typed ‘modules’ must specify types for all imported variables and specify types for all exports

step 2: type checking converts these ‘interface types’ into run-time checks that ‘blame’ the violator
i.e., contracts [Findler, Felleisen ICFP 2002]

Theorem: Let P be a mixed program with checked types at **import** interpreted as contracts. Then

- P yields to a value,
- P diverges, or
- P signals an error that blames an untyped module.

Theorem: Let P be a mixed program with checked types at **import** interpreted as contracts. Then

- P yields to a value,
- P diverges, or
- P signals an error that blames an untyped module.

the "Blame Theorem"
Tobin-Hochstadt & Felleisen (2006)
improved: Wadler-Findler (2009)

question: what about polymorphism?

question: what about polymorphism?

answer 1: use generative data to protect access at run-time and get relationally parametric contracts [Guha, Matthews, Findler, Krishnamurthi, DLS 2003]

question: what about polymorphism?

answer 1: use generative data to protect access at run-time and get relationally parametric contracts [Guha, Matthews, Findler, Krishnamurthi, DLS 2003]

answer 2: this imposes a new implementation of all primitives and thus a large cost on the language -- it is an open problem and a very real one

PRACTICAL TYPES

adding types to structure fields and type declarations for functions is acceptable

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)

;; Circle → Number


(check-within
  (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)

;; Circle → Number

(check-within
  (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

```
#lang typed-scheme

(define-struct: circle ([radius : Number]))


(: circle-area (circle → Number))

(check-within
   (circle-area (make-circle 1)) pi .1)

(define (circle-area c)
  (* pi (circle-radius c) (circle-radius c)))
```

Scheme demands different types for different occurrences of parameters

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)
...

(define-struct square (length))
;; Square = (make-square Number)
...

;; Shape is one of:
;; -- Circle
;; -- Square
;; ...

;; Shape → Number
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)]
    [(square? s) (square-area s)]))
```

```
#lang scheme

(define-struct circle (radius))
;; Circle = (make-circle Number)
...

(define-struct square (length))
;; Square = (make-square Number)
...

;; Shape is one of:
;; -- Circle
;; -- Square
;; ...

;; Shape → Number
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)] ;; programmers knows s : Circle
    [(square? s) (square-area s)]))
```

```
#lang typed-scheme

(define-struct: circle ({radius : Number}))
...

(define-struct: square ({length : Number}))
...


(define-type-alias shape
  (U
   circle
   square))

(: shape-area (shape → Number))
(define (shape-area s)
  (cond
    [(circle? s) (circle-area s)] ;; and so does our type system !
    [(square? s) (square-area s)]))
```

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area
              (filter square? l))))
```

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area  ;; programmer knows: (listof square)
              (filter square? l))))
```

```
#lang scheme
...
;; (Listof shape) → Number
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0 ;; programmer also knows: (listof number)
         (map square-area  ;; programmer knows: (listof square)
              (filter square? l))))
```

```
#lang typed-scheme
...
(: sum-squares ((Listof shape) → Number))
;; compute the areas of all squares in a list of arbitrary shapes
(define (sum-squares l)
  (foldl + 0
         (map square-area  ;; and so does our type system
              (filter square? l))))
```

"occurrence typing" is also necessary for paths into data structures

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))]
    [else (+ (first l) (sum (rest l)))]))
```

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))] ;; programmer knows: (first l) = #f
    [else (+ (first l) (sum (rest l)))]))
```

```
#lang scheme

;; Atom is either Number or false

;; [Listof Atom] → Number
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))]
    [else (+ (first l) (sum (rest l)))])) ; programmer: (first l) : Number
```

```
#lang typed-scheme

(define-type-alias Atom (U Number #f))

(: sum ([Listof Atom] → Number))
;; sum the numbers in this list

(check-expect (sum (list 2 3 #f 4)) 9)

(define (sum l)
  (cond
    [(empty? l) 0]
    [(not (first l)) (sum (rest l))] ; and so does our type system
    [else (+ (first l) (sum (rest l)))]))
```

the type system needs some simple propositional reasoning

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
; Atom is either Number or #f.

;; [Listof Atom] [Listof Atom] → [Listof Number]
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else ;; programmers knows (first l) : Number, (first k) : Number
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

```
(define-type-alias Atom (U Number #f))

(: mrg ([Listof Atom] [Listof Atom] → [Listof Number]))
;; add corresponding numbers, drop false, stop at end of shortest list

(check-expect (mrg (list 1 false 2) (list 3 4 5 false 10)) (list 4 4 7))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(not (first l))
     (cons (first k) (mrg (rest l) (rest k)))]
    [(not (first k))
     (cons (first l) (mrg (rest l) (rest k)))]
    [else ;; as does our type system
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

and programmers wish to abstract over predicates and paths in such programs

```
(define-type-alias Atom (U Number #f))

(: naughty ((Pair Atom Any) → Boolean : #f @ car))
(define (naughty l) (not (first l)))

(: mrg ([Listof Atom] [Listof Atom] → [Listof Number]))

(define (mrg l k)
  (cond
    [(or (empty? l) (empty? k))
     empty]
    [(and (not (first l)) (not (first k)))
     (cons 0 (mrg (rest l) (rest k)))]
    [(naughty? l)
     (cons (first k) (mrg (rest l) (rest k)))]
    [(naughty? k)
     (cons (first l) (mrg (rest l) (rest k)))]
    [else
     (cons (+ (first l) (first k)) (mrg (rest l) (rest k)))]))
```

all of this comes with the standard polymorphic functions (ML)

plus some *very* Scheme-specific twists

```
#lang scheme

;; (A ... → Void) (Listof A) ... → Void
;; process the worklists until one of them is exhausted

(define (process-work-list f . wl)
  (cond
    [(ormap empty? wl) (void)]
    [else (begin
            (apply f (map first wl))
            (apply process-work-list f (map rest wl)))]))
```

```
#lang scheme

;; (A ... → Void) (Listof A) ... → Void
;; process the worklists until one of them is exhausted

(define (process-work-list f . wl) ;; variable number of arguments
  (cond
    [(ormap empty? wl) (void)]
    [else (begin
            (apply f (map first wl))
            (apply process-work-list f (map rest wl)))]))
```

```
#lang typed-scheme

(: process-work-list (∀ (A ...) ((A ... → Void) (Listof A) ... → Void)))
;; process the worklists until one of them is exhausted

(define (process-work-list f . wl) ;; variable number of arguments
  (cond
    [(ormap empty? wl) (void)]
    [else (begin
            (apply f (map first wl))
            (apply process-work-list f (map rest wl)))]))
```

```
#lang typed-scheme

(: process-work-list (∀ (A ...) ((A ... → Void) (Listof A) ... → Void)))
;; process the worklists until one of them is exhausted

(define (process-work-list f . wl) ;; variable number of arguments
  (cond
    [(ormap empty? wl) (void)]
    [else (begin
            (apply f (map first wl))
            (apply process-work-list f (map rest wl)))]))
```

THEORETICAL TYPES

$$TEnv \vdash expression : type$$

where *type* includes "true union" types

$$\frac{TEnv(x) = t}{TEnv \vdash x : t}$$

TEnv ⊢ *expression* : *type*

where *type* includes "true union" types

$$\frac{TEnv(x) = t}{TEnv \vdash x : t}$$

```
(: find-path ([Listof String] -> (U Path #f))
(define (find-path los) ...)
```

TEnv ⊢ *expression* : *type*; *Filter+*, *Filter-*

where

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

TEnv ⊢ *expression* : *type*; *Filter+*, *Filter-*

where

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

$$\frac{\begin{array}{c} TEnv \vdash tst : t; Ftst+,Ftst- \\ TEnv * Ftst+ \vdash tn : t; Ftn+,Ftn- \\ TEnv * Ftst- \vdash tn : t; Fel+,Fel- \end{array}}{TEnv \vdash \textbf{if}\ tst\ \textbf{then}\ tn\ \textbf{else}\ el : t; F+,F-}$$

TEnv ⊢ *expression : type; Filter+, Filter-; Subject*

where

Subject is the path that expression explores, starting at some variable

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

TEnv ⊢ *expression* : *type*; *Filter+*, *Filter-*; *Subject*

where

Subject is the path that expression explores, starting at some variable

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

$$\frac{\begin{array}{c} \mathit{TEnv} \vdash \mathit{rator} : (d \rightarrow r; F; O); \mathit{Ftst+}, \mathit{Ftst-}; \mathit{Orator} \\ \mathit{TEnv} \vdash \mathit{tn} : t; \mathit{Ftn+}, \mathit{Ftn-}; \mathit{Orand} \\ \mathit{Oapp} = \mathit{PathRator}(\mathit{PathRand}(x)) \text{ if } O = \mathit{PathRator} \;\&\; \mathit{Orand} = \mathit{PathRand}(x) \end{array}}{\mathit{TEnv} \vdash (\mathit{rator}\ \mathit{rand}) : t; F+, F-; \mathit{Oapp}}$$

TEnv ⊢ *expression* : *type*; *Filter+*, *Filter-*; *Subject*

where

Subject is the path that expression explores, starting at some variable

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

$$\frac{\begin{array}{c}TEnv \vdash rator : (d \rightarrow r; F; O); Ftst+,Ftst-; Orator \\ TEnv \vdash tn : t; Ftn+,Ftn-; Orand \\ Oapp = PathRator(PathRand(x)) \text{ if } O = PathRator \ \& \ Orand = PathRand(x)\end{array}}{TEnv \vdash (rator\ rand) : t; F+,F-; Oapp}$$

Example: assume *x* has type

```
(if (number? (first x))
    ;; path for number?: *
    ;; path for (first x): (first x)
    ;; combined: (first x)
    (add1 (first x))  ;; combine path with filter and get (first x) : Number
    ...)
```

TEnv ⊢ *(lambda (x) expression) : dom* → rng; *LatentFilter+, LatentFilter-; LatentSubject*

where

LatentSubject is the path that *body* explores (when applied), starting at parameter

LatentFilter+ is a proposition for the case when *body* evaluates to a true value (when applied)

LatentFilter- is a proposition for the case when *body* evaluates to #f (false) (when applied)

TEnv; PropEnv ⊢ *expression : type; Filter+, Filter-; Subject*

where

PropEnv is a collection of implications between filters

Subject is the path that expression explores, starting at some variable

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

TEnv; PropEnv ⊢ *expression : type; Filter+, Filter-; Subject*

where

PropEnv is a collection of implications between filters

Subject is the path that expression explores, starting at some variable

Filter+ is a proposition for the case when expression evaluates to a true value

Filter- is a proposition for the case when expression evaluates to #f (false)

$$\frac{\begin{array}{c}\mathit{TEnv};\ \mathit{PropEnv} \vdash \mathit{tst} : t;\ \mathit{Ftst}+,\mathit{Ftst}-;\ \mathit{otst} \\ \mathit{PropEnv} * \mathit{Ftst}+ \vdash \mathit{Ftst}{+}{+} \\ \mathit{TEnv} * \mathit{Ftst}{+}{+};\ \mathit{PropEnv} * \mathit{Ftst}+ \vdash \mathit{tn} : t;\ \mathit{Ftn}+,\mathit{Ftn}-;\ \mathit{otn} \\ \mathit{PropEnv} * \mathit{Ftst}- \vdash \mathit{Ftst}{-}{-} \\ \mathit{TEnv} * \mathit{Ftst}{-}{-};\ \ \mathit{PropEnv} * \mathit{Ftst}- \vdash \mathit{tn} : t;\ \mathit{Fel}+,\mathit{Fel}-;\ \mathit{oel}\end{array}}{\mathit{TEnv} \vdash \textbf{if}\ \mathit{tst}\ \textbf{then}\ \mathit{tn}\ \textbf{else}\ \mathit{el} : t;\ F+,F-}$$

Theorem: The type system satisfies the usual type soundness theorem wrt to a standard CBV semantics.

Proof

- mechanized proof for part (Isabelle/Hol, Redex)
- proof requires "dead code" rules

EXPERIENCE

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
Lines	2369	511	407	315	1290	618	5510
Increase	7%	25%	7%	6%	1%	3%	7%
Useful Ann	96	51	25	16	53	26	267
λ: Ann	34	22	2	8	22	0	88
Other Ann	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
Type Alias	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
Ann/100 Line	7.3	18	6.9	8.6	6.7	5.0	7.9
Fixes	5	3	4	5	8	0	25
Fixes/100 Line	0.21	0.59	0.98	1.6	0.62	0.0	0.45
Problems	7	4	3	1	0	1	16
Prob/100 Line	0.29	0.78	0.73	0.32	0.0	0.16	0.29
Difficulty	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

	Squad	Metrics	Acct	Spam	Sys	Rand	Total
LINES	2369	511	407	315	1290	618	5510
INCREASE	7%	25%	7%	6%	1%	3%	7%
USEFUL ANN	96	51	25	16	53	26	267
λ: ANN	34	22	2	8	22	0	88
OTHER ANN	12	7	1	1	0	1	22
define-struct:	16	2	0	0	4	1	23
TYPE ALIAS	13	7	0	2	2	3	27
require/typed	3	1	0	0	5	0	9
ANN/100 LINE	7.3	18	6.9	8.6	6.7	5.0	7.9
FIXES	5	3	4	5	8	0	25
FIXES/100 LINE	0.21	0.59	0.98	1.6	0.62	0.0	0.45
PROBLEMS	7	4	3	1	0	1	16
PROB/100 LINE	0.29	0.78	0.73	0.32	0.0	0.16	0.29
DIFFICULTY	⋆⋆	⋆⋆⋆	⋆⋆	⋆	⋆	⋆	

Metrics : inspect Scheme programs
overuses a complex data structure
for different purposes

CONCLUSION

History

History

Cartwright 1976: *Typed Lisp*

History

Reynolds 1968: "occurrence typing"

Cartwright 1976: *Typed Lisp*

History

Reynolds 1968: "occurrence typing"

Cartwright 1976: *Typed Lisp*

Cartwright & 1989: *Soft Typing*

History

Reynolds 1968: "occurrence typing"

Cartwright 1976: *Typed Lisp*

Shivers 1989: from cfa to types

Cartwright & 1989: *Soft Typing*

the Rice School of soft typing:
Fagan, Wright, Flanagan, Meunier

Aiken, Heinze, Henglein,
and collaborators

History

Reynolds 1968: "occurrence typing"

Cartwright 1976: *Typed Lisp*

Shivers 1989: from cfa to types

Cartwright & 1989: *Soft Typing*

the Rice School of soft typing:
Fagan, Wright, Flanagan, Meunier

Aiken, Heinze, Henglein,
and collaborators

the PLT school of explicit types
and sound type enrichment:
Findler, Matthews, Gray, Tobin-Hochstadt

gradual typing: Siek and Taha

History

Reynolds 1968: "occurrence typing"

Cartwright 1976: *Typed Lisp*

Shivers 1989: from cfa to types

Cartwright & 1989: *Soft Typing*

the Rice School of soft typing:
Fagan, Wright, Flanagan, Meunier

Aiken, Heinze, Henglein,
and collaborators

the PLT school of explicit types
and sound type enrichment:
Findler, Matthews, Gray, Tobin-Hochstadt

gradual typing: Siek and Taha

Foster & Hicks et al.: Ruby
Field et al.: "occurrence typing" for COBOL
Vitek & Field et al.: Thorn
Wadler: theory

Contrast

Types ***in*** Language Design & Implementation

Contrast

Types ***in*** Language Design & Implementation

Types ***after*** Language Design & Implementation:
untyped languages are more popular than typed languages
people build large systems in these languages
and ***then*** they discover the **need for types**

Contrast

Types ***in*** Language Design & Implementation

Types ***after*** Language Design & Implementation:
- untyped languages are more popular than typed languages
- people build large systems in these languages
- and ***then*** they discover the **need for types**

the game is different:
- asking programmers to rewrite code to appease a type
- checker will not work; the type system must accommodate the language

For Scheme we found these to be useful:

- "occurrence typing"
- separating positive and negative propositions
- separating out the subject of inspection
- basic propositional reasoning

Sam Tobin-Hochstadt,
Typed Scheme: From Scripts to Programs
NUPRL Dissertation, January 2010
http://www.ccs.neu.edu/scheme/pubs/

For other "scripting" languages
we may need these
and different techniques.
There's plenty of work.

THE END

Sam Tobin-Hochstadt (NUPRL)

T. Stevie Strickland (NUPRL)

Ivan Gazeau (ENS)