TYPES FOR UNTYPED LANGUAGES

Matthias Felleisen (PLT)
Northeastern University

Monday, January 18, 2010
“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 ...

close 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
#lang STLC

inc : int \rightarrow int
inc(x) = x + 1

provide inc

#lang LC

require T

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

```markdown
#lang STLC

inc : int → int
inc(x) = x + 1

provide inc
```

LC

```markdown
#lang LC

require T

f(x) = ... inc(true) ...
```
#lang STLC

encode : (int → int) → int
encode(f) = \ldots f(42) + 21 \ldots

provide encode

#lang LC

require T

f(x) = \ldots x + 1 \ldots return "hello"

main(x) = \ldots encode(f) \ldots
# lang STLC

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

provide encode

# lang LC

require T
f(x) = \( x + 1 \)  return "hello"
main(x) = \( \text{encode}(f) \)
#lang STLC

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

provide encode

#lang LC

require T

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

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

```plaintext
#lang LC
inc(x) = x + 1

provide inc
```

T

```plaintext
#lang STLC

require U

f(x) = ... inc(???) .?? 10 ...
```
#lang LC

inc(x) = x + 1

provide inc

#lang STLC

require U

f(x) = ... inc(???) ??? 10 ...
#lang LC

inc(x) = x + 1

provide inc

#lang STLC

require U [inc : int → int]

f(x) = ... inc(42) + 10 ...
inc(x) = \textbf{if} \; x = 0
    \quad \text{“hello”}
  \textbf{else}
    \quad x + 1

\textbf{provide} \; \text{inc}

\textbf{require} \; \text{U} \; [\text{inc} : \text{int} \rightarrow \text{int}]

f(x) = \ldots \text{inc}(42) + 10 \ldots
\#lang LC

inc(x) = \textbf{if } x = 0 \textbf{ then } \text{“hello”} \textbf{ else } x + 1

\textbf{provide} inc

\#lang STLC

\textbf{require} \( U \[ \text{inc} : \text{int} \rightarrow \text{int} \] \)

f(x) = \ldots \text{inc(42)} + 10 \ldots
\#lang LC

\texttt{inc(x) = if } x = 0 \texttt{ "hello" else } x + 1 \texttt{ provide inc}

\#lang STLC

\texttt{require U [inc : int \rightarrow int] f(x) = ... inc(42) + 10 ...}
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))))
;; (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
;; 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)))]))
;; 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
;;; (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))))]))
;; (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)))]))
(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
$TEnv \vdash \text{expression : type}$

where type includes “true union” types

\[
\begin{align*}
TEnv(x) &= t \\
\hline \\
TEnv &\vdash x : t
\end{align*}
\]
$TEnv \vdash expression : type$

where type includes "true union" types

\[
\begin{align*}
TEnv(x) &= t \\
TEnv \vdash x : t
\end{align*}
\]

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

Monday, January 18, 2010
\[ TEnv \leftarrow \text{expression} : \text{type}; \; Filter^+, \; Filter^- \]

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)
$TEnv \vdash expression : type; Filter+, Filter-$

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)

$TEnv \vdash tst : t; Ftst+, Ftst-$

$TEnv * Ftst+ \vdash tn : t; Ftn+, Ftn-$

$TEnv * Ftst- \vdash tn : t; Fel+, Fel-$

$\underline{TEnv \vdash if \ tsp \ then \ tsp \ else \ tsp \ : t; F+, F-}$
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)
$\text{TEnv} \vdash \text{expression} : \text{type}; \text{Filter}+, \text{Filter}-; \textit{Subject}$

where

- $\textit{Subject}$ is the path that expression explores, starting at some variable
- $\text{Filter}+$ is a proposition for the case when expression evaluates to a true value
- $\text{Filter}-$ is a proposition for the case when expression evaluates to #f (false)

$\text{TEnv} \vdash \text{rator} : (d \rightarrow r; F; O); \text{Ftst}+, \text{Ftst}-; \text{Orator}$

$\text{TEnv} \vdash \text{tn} : t; \text{Ftn}+, \text{Ftn}-; \text{Orand}$

$\text{Oapp} = \text{PathRator(PathRand(x))} \text{ if } O = \text{PathRator} \text{ and } \text{Orand} = \text{PathRand(x)}$

$\text{TEnv} \vdash (\text{rator rand}) : t; F+, F-; \text{Oapp}$
$\text{TEnv} \vdash \text{expression} : \text{type}; \text{Filter}+, \text{Filter}-; \text{Subject}$

where

\text{Subject} is the path that expression explores, starting at some variable

\text{Filter}+ is a proposition for the case when expression evaluates to a true value

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

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

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
  ...
$TEnv \vdash (\lambda (x) \text{ expression}) : \text{dom} \rightarrow \text{rng}; \text{LatentFilter}^+, \text{LatentFilter}^-; \text{LatentSubject}$

where

- $\text{LatentSubject}$ is the path that $\text{body}$ explores (when applied), starting at parameter
- $\text{LatentFilter}^+$ is a proposition for the case when $\text{body}$ evaluates to a true value (when applied)
- $\text{LatentFilter}^-$ is a proposition for the case when $\text{body}$ 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)

TEnv; PropEnv ⊢ tst : t; Ftst+,Ftst-; otst
PropEnv * Ftst+ ⊢ Ftst++
TEnv * Ftst++; PropEnv * Ftst+ ⊢ tn : t; Ftn+,Ftn-; otn
PropEnv * Ftst- ⊢ Ftst--
TEnv * Ftst--; PropEnv * Ftst- ⊢ tn : t; Fel+,Fel-; oel

TEnv ⊢ if tst then tn else 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
### Statistics and Results

<table>
<thead>
<tr>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINES</strong></td>
<td>2369</td>
<td>511</td>
<td>407</td>
<td>315</td>
<td>1290</td>
<td>618</td>
</tr>
<tr>
<td><strong>INCREASE</strong></td>
<td>7%</td>
<td>25%</td>
<td>7%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
</tr>
<tr>
<td><strong>USEFUL ANN</strong></td>
<td>96</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>53</td>
<td>26</td>
</tr>
<tr>
<td><strong>λ: ANN</strong></td>
<td>34</td>
<td>22</td>
<td>2</td>
<td>8</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td><strong>OTHER ANN</strong></td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>define-struct:</strong></td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td><strong>TYPE ALIAS</strong></td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>require/typed</strong></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>Ann/100 Line</strong></td>
<td>7.3</td>
<td>18</td>
<td>6.9</td>
<td>8.6</td>
<td>6.7</td>
<td>5.0</td>
</tr>
<tr>
<td><strong>FIXES</strong></td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td><strong>FIXES/100 Line</strong></td>
<td>0.21</td>
<td>0.59</td>
<td>0.98</td>
<td>1.6</td>
<td>0.62</td>
<td>0.0</td>
</tr>
<tr>
<td><strong>PROBLEMS</strong></td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Prob/100 Line</strong></td>
<td>0.29</td>
<td>0.78</td>
<td>0.73</td>
<td>0.32</td>
<td>0.0</td>
<td>0.16</td>
</tr>
<tr>
<td><strong>DIFFICULTY</strong></td>
<td>**</td>
<td>***</td>
<td>**</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
### 11.3. STATISTICS AND RESULTS

<table>
<thead>
<tr>
<th>Metric</th>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINES</td>
<td>2369</td>
<td>511</td>
<td>407</td>
<td>315</td>
<td>1290</td>
<td>618</td>
<td>5510</td>
</tr>
<tr>
<td>INCREASE</td>
<td>7%</td>
<td>25%</td>
<td>7%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
<td>7%</td>
</tr>
<tr>
<td>Useful Ann</td>
<td>96</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>53</td>
<td>26</td>
<td>267</td>
</tr>
<tr>
<td>λ: Ann</td>
<td>34</td>
<td>22</td>
<td>2</td>
<td>8</td>
<td>22</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td>Other Ann</td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Define-Struct:</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>Type Alias</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>Require/Typed</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>ANN/100 Line</td>
<td>7.3</td>
<td>18</td>
<td>6.9</td>
<td>8.6</td>
<td>6.7</td>
<td>5.0</td>
<td>7.9</td>
</tr>
<tr>
<td>Fixes</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Fixes/100 Line</td>
<td>0.21</td>
<td>0.59</td>
<td>0.98</td>
<td>1.6</td>
<td>0.62</td>
<td>0.0</td>
<td>0.45</td>
</tr>
<tr>
<td>Problems</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Prob/100 Line</td>
<td>0.29</td>
<td>0.78</td>
<td>0.73</td>
<td>0.32</td>
<td>0.0</td>
<td>0.16</td>
<td>0.29</td>
</tr>
<tr>
<td>Difficulty</td>
<td>**</td>
<td>***</td>
<td>**</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
### 11.3. STATISTICS AND RESULTS

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

Figure 11.1: Statistics

- **Useful Ann**: This is the count of annotations of variables that recorded valuable design information about the program. It includes top-level functions, local functions, top-level constants in which the untyped program provided a comment specifying the type, and specifications of the types of mutable data, such as hash tables.

- **λ: Ann**: This is the count of uses of the λ form, which annotates the bound variables of a λ with types. With the exception of a handful of cases, this is necessary only when an anonymous function is provided as the argument to a polymorphic function such as `map`.

- **Other Ann**: This is a count of all other variable annotations and instantiations in the program, which do not appear to record useful design information.

- **define-struct**: This is the number of uses of `define-struct:` in the typed version of the program. In all cases, this is the same as the number of `define-struct` statements in the original program.
### Statistics and Results

<table>
<thead>
<tr>
<th>Metric</th>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LINES</strong></td>
<td>2369</td>
<td>511</td>
<td>407</td>
<td>315</td>
<td>1290</td>
<td>618</td>
<td>5510</td>
</tr>
<tr>
<td><strong>INCREASE</strong></td>
<td>7%</td>
<td>25%</td>
<td>7%</td>
<td>6%</td>
<td>1%</td>
<td>3%</td>
<td>7%</td>
</tr>
<tr>
<td><strong>USEFUL ANN</strong></td>
<td>96</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>53</td>
<td>26</td>
<td>267</td>
</tr>
<tr>
<td><strong>λ: ANN</strong></td>
<td>34</td>
<td>22</td>
<td>2</td>
<td>8</td>
<td>22</td>
<td>0</td>
<td>88</td>
</tr>
<tr>
<td><strong>OTHER ANN</strong></td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td><strong>define-struct:</strong></td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td><strong>TYPE ALIAS</strong></td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td><strong>require/typed</strong></td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td><strong>Ann/100 Line</strong></td>
<td>7.3</td>
<td>18</td>
<td>6.9</td>
<td>8.6</td>
<td>6.7</td>
<td>5.0</td>
<td>7.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIXES</strong></td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td><strong>FIXES/100 Line</strong></td>
<td>0.21</td>
<td>0.59</td>
<td>0.98</td>
<td>1.6</td>
<td>0.62</td>
<td>0.0</td>
<td>0.45</td>
</tr>
<tr>
<td><strong>PROBLEMS</strong></td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td><strong>Prob/100 Line</strong></td>
<td>0.29</td>
<td>0.78</td>
<td>0.73</td>
<td>0.32</td>
<td>0.0</td>
<td>0.16</td>
<td>0.29</td>
</tr>
</tbody>
</table>

| **Difficulty**    | **| ***| **| *| *| *| *|
11.3. STATISTICS AND RESULTS

<table>
<thead>
<tr>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINES</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>USEFUL ANN</td>
<td>96</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>53</td>
<td>26</td>
</tr>
<tr>
<td>λ: ANN</td>
<td>34</td>
<td>22</td>
<td>2</td>
<td>8</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>OTHER ANN</td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>define-struct:</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>TYPE ALIAS</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>require/typed</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>ANN/100 Line</td>
<td>7.3</td>
<td>18</td>
<td>6.9</td>
<td>8.6</td>
<td>6.7</td>
<td>5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXES</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>FIXES/100 Line</td>
<td>0.21</td>
<td>0.59</td>
<td>0.98</td>
<td>1.6</td>
<td>0.62</td>
<td>0.0</td>
</tr>
<tr>
<td>PROBLEMS</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PROB/100 Line</td>
<td>0.29</td>
<td>0.78</td>
<td>0.73</td>
<td>0.32</td>
<td>0.0</td>
<td>0.16</td>
</tr>
<tr>
<td>DIFFICULTY</td>
<td>**</td>
<td>***</td>
<td>**</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

Figure 11.1: Statistics
### Squad Metrics Acct Spam Sys Rand Total

<table>
<thead>
<tr>
<th>Lines</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>Sys</th>
<th>Rand</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2369</td>
<td>511</td>
<td>407</td>
<td>315</td>
<td>1290</td>
<td>618</td>
<td>5510</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>Useful Ann</td>
<td>96</td>
<td>51</td>
<td>25</td>
<td>16</td>
<td>53</td>
<td>26</td>
</tr>
<tr>
<td>λ: Ann</td>
<td>34</td>
<td>22</td>
<td>2</td>
<td>8</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>Other Ann</td>
<td>12</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>define-struct:</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Type Alias</td>
<td>13</td>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>require/typed</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Ann/100 Line</td>
<td>7.3</td>
<td>18</td>
<td>6.9</td>
<td>8.6</td>
<td>6.7</td>
<td>5.0</td>
</tr>
<tr>
<td>Fixes</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Fixes/100 Line</td>
<td>0.21</td>
<td>0.59</td>
<td>0.98</td>
<td>1.6</td>
<td>0.62</td>
<td>0.0</td>
</tr>
<tr>
<td>Problems</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Prob/100 Line</td>
<td>0.29</td>
<td>0.78</td>
<td>0.73</td>
<td>0.32</td>
<td>0.0</td>
<td>0.16</td>
</tr>
</tbody>
</table>

**Difficulty**

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

Cartwright & 1989: *Soft Typing*

Shivers 1989: from cfa to types

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

Aiken, Heinze, Henglein, and collaborators

Monday, January 18, 2010
History

Reynolds 1968: “occurrence typing”

Cartwright 1976: *Typed Lisp*

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

Shivers 1989: from cfa to types

gradual typing: Siek and Taha
History

Reynolds 1968: “occurrence typing”

Cartwright 1976: *Typed Lisp*

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

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

Shivers 1989: from cfa to types

gradual typing: Siek and Taha
Contrast

Types in Language Design & Implementation
Types in 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.