Growing Software
From Scripts to Programs

Sam Tobin-Hochstadt
March 2, 2011
Oregon State University
The Rise Of Scripting

A brief tour
Python
Pensionssystemet

Det svenska pensionssystemet består av tre huvuddelar, den statliga allmänna pensionen, tjänstepensionen och den frivilliga pensionen. AP-fondernas förvaltning är den del av den allmänna pensionen.

Pensionssystemet kan liknas vid en pyramid där den allmänna pensionen utgör basen, därefter tjänstepensionen och överst det frivilliga privata pensionssparandet.

Allmän pension
Quick hack to critical system: *The paradigmatic scripting story*

Started as a backup system
Ended managing billions in assets
“whipitupitude” — Larry Wall
via web

Common Lisp

C++

Ruby

Scala

Java
Addressing the Challenge
Non-Solutions

Waterfall development of spec and code

Replace all scripting languages

Omniscient program analysis
Non-Solutions

- Waterfall development of spec and code
- Replace all scripting languages
- Omniscient program analysis

The all-too-common result: rewrite in C++/Java
What is a solution?

What we want: a robust, maintainable program

Where we are: a quick but overgrown script
What we want: a robust, maintainable program

Where we are: a quick but overgrown script

Existing PL technology: **Types** as lightweight specifications

- Robustness via static enforcement
- Maintainability via checked specs
- Evolution via refactoring support
What is a solution?

What we want: a robust, maintainable program in a **typed sister language**

Where we are: a quick but overgrown script
What we want: a robust, maintainable program in a **typed sister language**

Where we are: a quick but overgrown script

Choose a component

Add type annotations

What is a solution?
What we want: a robust, maintainable program in a **typed sister language**

Where we are: a quick but overgrown script

Choose a component

Add type annotations

Check types statically
What we want: a robust, maintainable program in a typed sister language.

What is a solution?

Where we are: a quick but overgrown script.

Choose a component

Add type annotations

Check types statically

Safely Interoperate
What we want: a robust, maintainable program in a **typed sister language**.

What is a solution?

Where we are: a quick but overgrown script.

Choose a component

Add type annotations

Check types statically

Safely Interoperate
My Research Methodology

1. Discover a challenge in the real world
2. Study the challenge in a controlled but realistic environment
   - Formally analyze the problem
   - Implement the solution in a production system
3. Validate the solution in theory & practice
4. Bring the solution to the broader community
Racket

A descendant of Lisp & Scheme

15 years of development

20+ current developers

Used in dozens of companies, 120 universities, 200 schools

500,000 line code base

Ideal environment for investigating script to program evolution
Typed Racket

A typed dialect of Racket

Publicly distributed for 4+ years

Used in key Racket systems

Used in multiple companies and several college courses

Supports dozens of existing libraries

A testbed for scripts-to-programs research
(define (main stx trace-flag super-expr deser-id-expr name-id ifc-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-localised? #f]
    [localise/set-flag (lambda (id)
      (let ([id2 (localise id)])
        (set! any-localised? #t)
        id2))]
    [bind-local-id (lambda (id)
      (let ([id (syntax-local-infer-name stx)]
        (bound-identifier localized-id)))]
    [lookup-localise (lambda (id)
      (bound-identifier-mapping-get localized-map id (lambda ()
        ; If internal & external names are distinguished, we need to fall back to localise:
        (localise id))))])

; ----- Expand definitions -----
(let ([defn-and-exprs (expand-all-forms stx defn-and-exprs def-ctx bind-local-id)])

[bad (lambda (meg expr)
  [raise-syntax-error #f meg stx expr])]

[class-name (if name-id
  (syntax-e name-id)
  (let ([a (syntax-local-infer-name stx)])
    (if (syntax e)
      (syntax-e e)
      a)))]

; ----- 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 augment override augment rename-super inherit inherit/super inherit/inner rename/inner inspect)

  [[form meq idp ...]
    (and (identifier? #form)
      (or (free-identifier? #form (quote-syntax -init))
        (free-identifier? #form (quote-syntax -init-field)))])))

+) + 900 more lines
+ 900 more lines
(: main : Stx Bool Expr (or #f Id) ... -> Expr)

(define (main stx trace-flag super-expr
deser-id-expr name-id
ifc-exprs defn-and-exprs)

(let-values ([this-id (quote (quote-syntax 'this))])
  [the-obj (quote-syntax 'self)])
(let* ([def-ctx (syntax-local-make-definition-context)])
  [localized-map (make-bound-identifier-mapping)]
  [any-localized? #f]
  [localise/set-flag (lambda (id)
     (let ((id2 (localise id)))
       (unless (eq? id id2)
         (set! any-localized? #t))
       id2))]
  [bind-local-id (lambda (id)
      (let ((loc (syntax-local-infer-name stx))
        (bound-identifier localised-ids id))
        (if (syntax? loc)
          (syntax-e loc)
          loc))]
  [lookup-local (lambda (id)
      (bound-identifier-mapping-get localized-map id
      (lambda () ; If internal & external names are distinguished, ; we need to fall back to localise: (localise id)])))

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

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

+ 900 more lines
Safe Interoperation
Modular Programs,
Modular Checking

```javascript
require(['some/module',
    'text!some/module.html',
    'text!some/module.css'],
function(module, html, css) {
    return style_with(html, css);
});
```
Modular Programs, Modular Checking

```javascript
require(["some/module",
  "text!some/module.html"],
  function(module, html, css) {
      return style_with(html, css);
  });

import os.system
system.output("hello world")
```
Modular Programs,
Modular Checking

```javascript
require(['some/module', 'text!some/module.html', 'text!some/module.css'], function(module, html, css) {
  return style_with(html, css);
});

import os.system
system.output('hello world')

module DogsRelated
  NBR_OF_DOGS_NEEDED = 5
  class Dog
    def bark
      puts 'Woof...
    end
  end
end
```
Modular Programs,
Modular Checking

```javascript
require(['some/module',
          'text!some/module.html'],
          function(module, html, css) {
            return style_with(html, css);
          });
import os.system
system.output("hello world")

module DogsRelated
NBR_OF_DOGS_NEEDED = 5
class Dog
  def bark
    puts "Woof...
  end
end

render :: Data -> Graphic
function render(d) {
  let d1 = process(d);
  return transform(d1);
}
```
Making Interoperation Safe

Typed Module

Untyped Module

Untyped Module

Untyped Module
Making Interoperation Safe

Typed Module

Untyped Module

Dynamic Type-Enforcing Boundary
Making Interoperation Safe

Typed Module

Dynamic Type-Enforcing Boundary

Untyped Module

Typed Module

Untyped Module

Typed Module
Making Interoperation Safe

- Typed Module
- Untyped Module
- Dynamic Type-Enforcing Boundary
- Typed Module
- Typed Module
Dynamically Enforcing Types

<table>
<thead>
<tr>
<th>Static Type</th>
<th>Synthesized Dynamic Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>is_numeric</td>
</tr>
<tr>
<td>Listof[String]</td>
<td>s.all(is_string)</td>
</tr>
</tbody>
</table>
Dynamically Enforcing Types

<table>
<thead>
<tr>
<th>Static Type</th>
<th>Synthesized Dynamic Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>is_numeric</td>
</tr>
<tr>
<td>Listof[String]</td>
<td>s.all(is_string)</td>
</tr>
<tr>
<td>InFile -&gt; OutFile</td>
<td>preconditions/postconditions</td>
</tr>
</tbody>
</table>
#lang racket

(define (add5 x) (+ x 5))

#lang racket

(require server)
(add5 7)
(define (add5 x) (+ x 5))

(require server)
(add5 "seven")

+: expected number, but got "seven"
#lang typed/racket

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

#lang racket

(require server)
(add5 "seven")

+: expected number, but got "seven"
#lang typed/racket

(: add5 : Number → Number)
(define (add5 x) (x + 5))

#lang racket

(require server)
(add5 "seven")

+: expected number, but got “seven”
#lang typed/racket

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

---

#lang racket

(require server)
(add5 "seven")

client broke the specification on add5
#lang racket

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

---

#lang typed/racket

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

(add5 7)

server interface broke the specification on add5
## Dynamically Enforcing Types

<table>
<thead>
<tr>
<th>Static Type</th>
<th>Synthesized Dynamic Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>is_numeric</td>
</tr>
<tr>
<td>Listof[String]</td>
<td>s.all(is_string)</td>
</tr>
<tr>
<td>InFile -&gt; OutFile</td>
<td>preconditions/postconditions</td>
</tr>
<tr>
<td>(ℝ -&gt; ℝ) -&gt; (ℝ -&gt; ℝ)</td>
<td></td>
</tr>
</tbody>
</table>
## Dynamically Enforcing Types

<table>
<thead>
<tr>
<th>Static Type</th>
<th>Synthesized Dynamic Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>is_numeric</td>
</tr>
<tr>
<td>Listof[String]</td>
<td>s.all(is_string)</td>
</tr>
<tr>
<td>InFile -&gt; OutFile</td>
<td>preconditions/postconditions</td>
</tr>
<tr>
<td>((\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R}))</td>
<td>higher-order contracts</td>
</tr>
</tbody>
</table>

[Findler & Felleisen ICFP 02]
#lang typed/racket

(: deriv : (R -> R) -> (R -> R))  
(define (deriv f) (lambda (x) ...))

---

#lang racket

(require server)  
(define cos (deriv sin))  
(cos "bad")
#lang typed/racket

(: deriv : (R -> R) -> (R -> R))

(define (deriv f) (lambda (x) ...))

---

#lang racket

(require server)

(define cos (deriv sin))

(cos "bad")

client broke the specification on deriv
#lang typed/racket

(: deriv : (R -> R) -> (R -> R))
(define (deriv f) (lambda (x) ...))

#lang typed/racket

(require server)
(define cos (deriv sin))
(cos "bad")

typechecker: incorrect argument to deriv
Typed & Untyped

Catch errors dynamically at the boundary

Key Elements

Automatically Synthesizing Dynamic Checks from Types
[DLS 06]

Multi-language Infrastructure
[PLDI 11]

More Efficient, More Expressive Contracts
[Work in progress]
Static Guarantees from Blame

server interface broke the specification on add5

client broke the specification on add5

client broke the specification on deriv
Static Guarantees from Blame

server interface broke the specification on add5

client broke the specification on add5

client broke the specification on deriv

Contracts and blame give us a soundness theorem:

**Dynamic type errors always blame the untyped modules**

[DLS 2006]
Static Guarantees from Blame

Contracts and blame give us a soundness theorem:

Dynamic type errors always blame the untyped modules

[DLS 2006]
Why Multilanguage Soundness?

- Support local reasoning
- Static guarantee only depends on typed modules
- Tunable levels of checking
Types for Untyped Languages
All programmers reason about their programs
All programmers reason about their programs

Type systems capture programmer reasoning
Programs in Lua don’t use the Java type system
Java  
Lua  
Programs in Lua don’t use the ML type system  
Perl  
Python  
Ruby  
C#  
Lua  
Clojure  
Haskell  
Scala  
Javascript  
Java  
PHP  
C++  
Pascal
Solution: design a type system based on the existing idioms of the language
Types for Existing Programs

- Unions, Structures, Polymorphism
- Occurrence Typing
- Refinement Types
- Variable-Arity
- Numerics
- Standard
- [POPL 08]
- [ICFP 10]
- [HOSC 11]
- [ESOP 09]
- in preparation
Types for Existing Programs

- Unions, Structures, Polymorphism
- Occurrence Typing
- Refinement Types
- Variable-Arity
- Numerics

- Standard
- [POPL 08]
- [ICFP 10]
- [HOSC 11]
- [ESOP 09]
- in preparation
Dynamic Type Tests

```javascript
if (typeof x === "number") {
    return x + 1;
}
else if (typeof x === "function") {
    return x();
}
else if (typeof x === "object") {
    return x.length;
}
else
    return 0;
```
if (typeof x === "number") {
    return x + 1;
}
else if (typeof x === "function") {
    return x();
}
else if (typeof x === "object") {
    return x.length;
}
else
    return 0;

Dynamic Type Tests

if isinstance(x,Numeric):
    print x + 1
elif isinstance(x,String):
    print x
else:
    print "Nothing"
Dynamic Type Tests

```javascript
if (typeof x === "number") {
    return x + 1;
}
else if (typeof x === "function") {
    return x();
}
else if (typeof x === "object") {
    return x.length;
}
else
    return 0;
```

```java
if (isinstance(x, Numeric)):
    if isinstance(x, Numeric):
        print x + 1
    else if (type(x) === "String") {
        print x
    } else:
        print "Nothing"
```
;; sum : BT -> Number
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+
                  (sum (left bt))
                  (sum (right bt)))]))
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
    (cond [(number? bt) bt]
          [else (+
                  (sum (left bt))
                  (sum (right bt))))]))
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+
            (sum (left bt))
            (sum (right bt)))]))

bt : BT
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+
              (sum (left bt))
              (sum (right bt)))]))
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+
               (sum (left bt))
               (sum (right bt)))]))

bt : BT
bt : Number
number? : Any Number -> Bool
Number 
Any 
Bool 
Number 
Any 
Bool
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+ (sum (left bt))
                  (sum (right bt))))])
(define-type BT (U Number (Pair BT BT)))

(: sum : BT -> Number)
(define (sum bt)
  (cond [(number? bt) bt]
        [else (+
              (sum (left bt))
              (sum (right bt))))]))
(map rectangle-area
  (filter rectangle? list-of-shapes))

filter :

\[ \forall \alpha \beta. (\alpha \to \text{Bool}) (\text{Listof } \alpha) \to (\text{Listof } \beta) \]
(map rectangle-area
  (filter rectangle? list-of-shapes))

filter :
(Shape \rightarrow Bool) (Listof Shape) \rightarrow (Listof Rect)

\forall \alpha \beta. (\alpha \xrightarrow{\beta} Bool) (Listof \alpha) \rightarrow (Listof \beta)
filter : 
(Shape $\xrightarrow{\text{Rect}}$ Bool) (Listof Shape) $\rightarrow$ (Listof Rect)

$\forall \alpha \beta. (\alpha \xrightarrow{\beta} \text{Bool}) (\text{Listof } \alpha) \rightarrow (\text{Listof } \beta)$
Key Idea 1:
A logic to prove facts about variables and types
Key Idea 1:
A logic to prove facts about variables and types

Key Idea 2:
An environment of general propositions

\[
\text{L-SUB} \\
\Gamma \vdash \tau_x \quad \vdash \tau \prec \sigma \\
\overline{\quad \Gamma \vdash \sigma_x \quad \Gamma \vdash x : \tau \quad \#f_x | \#f_x ; x}
\]
Key Idea 1: A logic to prove facts about variables and types

Key Idea 2: An environment of general propositions

Result: Rich type system that can follow sophisticated reasoning
Soundness: if $e : \tau$ and $e \to v$, then $v : \tau$

In other words, we can trust our types.
Validation: Existing Code

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squad</td>
<td>1,500</td>
</tr>
<tr>
<td>Metrics</td>
<td>3,000</td>
</tr>
<tr>
<td>Acct</td>
<td>4,500</td>
</tr>
<tr>
<td>Spam</td>
<td>6,000</td>
</tr>
<tr>
<td>SVN</td>
<td></td>
</tr>
<tr>
<td>PRNG</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>13,500</td>
</tr>
</tbody>
</table>

- Original Code
- New Code
Validation: Existing Code

5500 lines of code

7% increase

Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Original Code</th>
<th>New Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squad</td>
<td>1500</td>
<td>1720</td>
</tr>
<tr>
<td>Metrics</td>
<td>3000</td>
<td>3000</td>
</tr>
<tr>
<td>Acct</td>
<td>4500</td>
<td>4500</td>
</tr>
<tr>
<td>Spam</td>
<td>6000</td>
<td>6000</td>
</tr>
<tr>
<td>SVN</td>
<td>7500</td>
<td>8000</td>
</tr>
<tr>
<td>PRNG</td>
<td>9000</td>
<td>9500</td>
</tr>
<tr>
<td>Total</td>
<td>5500</td>
<td>5870</td>
</tr>
</tbody>
</table>

Legend:
- Original Code
- New Code
Validation: Existing Code

5500 lines of code
7% increase

25 errors found
13 minor edits required
1 component left untyped

Program

<table>
<thead>
<tr>
<th>Squad</th>
<th>Metrics</th>
<th>Acct</th>
<th>Spam</th>
<th>SVN</th>
<th>PRNG</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>Original Code</td>
<td>New Code</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5500</td>
<td>5400</td>
<td>5400</td>
<td>5400</td>
<td>5400</td>
<td>5400</td>
<td>5500</td>
</tr>
</tbody>
</table>

7% increase
Validation: Comparative

fun balance T (B, T(R, T(R, a, x, b), y, c), z, d) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, T(R, a, x, T(R, b, y, c)), z, d) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, a, x, T(R, T(R, b, y, c), z, d)) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T (B, a, x, T(R, b, y, T(R, c, z, d))) = T(R, T(B, a, x, b), y, T(B, c, z, d))
| balance T body = T body

(define (balance tree)
  (match tree
    [(T B (T R (T R a x b) y c) z d) (T R (T B a x b) y (T B c z d))]
    [(T B (T R a x (T R b y c)) z d) (T R (T B a x b) y (T B c z d))]
    [(T B a x (T R (T R b y c) z d)) (T R (T B a x b) y (T B c z d))]
    [(T B a x (T R b y (T R c z d))) (T R (T B a x b) y (T B c z d))]
    [else tree]])))

[Prashanth Thesis 2011]
Contracts to Dynamically Enforce Types

Blame for Soundness

DLS 2006, STOP 2009
Contracts to Dynamically Enforce Types

Blame for Soundness

Type System for Language Idioms

Validation on Existing Programs

Contracts to Dynamically Enforce Types

Blame for Soundness

Type System for Language Idioms

Validation on Existing Programs

Multilanguage Development Infrastructure

Scheme 2007, PLDI 2011
Developing a solution

Locate an existing problem

Typed

Untyped
Developing a solution

Locate an existing problem

Develop a rigorous design

Typed

Untyped

\[
\text{ST-Abs} \quad \frac{\Gamma, x : t \vdash_{ST} e : s; e'}{\Gamma \vdash_{ST} (\lambda x : t.e) : (t \rightarrow s); (\lambda x : t.e')}
\]
Developing a solution

Locate an existing problem

Develop a rigorous design

Validate by implementation & experiment

**Typed**

**Untyped**

\[ \text{ST-Abs} \]

\[
\Gamma, x : t \vdash_{ST} e : s; e' \\
\Gamma \vdash_{ST} (\lambda x : t.e) : (t \rightarrow s); (\lambda x : t.e')
\]
Developing a solution

Locate an existing problem

Develop a rigorous design

Validate by implementation & experiment

Developed a solution implementation & experiment
t
v
position, where rules trivially follows. Further, by examination of the reductio way.

This concludes the case. The others are proved in a similar way.

If Validate by

Locate an

Proof Sketch

Given the soundness of

existing

by hypothesis, it must satisfy its cast, and have type have label consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent (since the only other possibility is a redex).

Therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.

consistent. If it is a type-annotated abstraction, it must have label row contract, and untyped abstractions are not consistent, and thus therefore, the whole application must have been consistent, then it must be been a typed abstraction, since it is the argument of a blessed application and satisfied.
Developing a solution

Locate an existing problem

Transfer Lessons to Other Languages

Typed

Untyped

Validate by implementation & experiment

Develop a rigorous design
The Way Forward

Bringing the solution to the broader community
Next Stop: JavaScript

Language Infrastructure

Contracts

Modules

In collaboration with
Next Stop: JavaScript

Language Infrastructure

Contracts

Modules

In collaboration with
Modules on the Web

```
module $ = "http://jquery.com/jquery.js";

$(document).ready(function() {
    alert("hello world");
});
```

Naming  Scoping  Pre-fetching, parsing, compiling
Sandboxing  Cross-Origin Security
Beyond Types ...

Where we are: a quick but overgrown script

What we want: a robust maintainable program
Beyond Types ...

What we want: reliable, effective software

What we want: a robust maintainable program

Where we are: a quick but overgrown script
Beyond Types ...

What we want: reliable, effective software

Robust Communication

What we want: a robust maintainable program

Where we are: a quick but overgrown script
Beyond Types ...

What we want: reliable, effective software

Robust Communication
Parallel Performance

What we want: a robust maintainable program
Where we are: a quick but overgrown script
Beyond Types ...

What we want: reliable, effective software

What we want: a robust, maintainable program

Where we are: a quick but overgrown script

Robust Communication

Parallel Performance

Trustworthy Security
Beyond Types ...

What we want: reliable, effective software

What we want: a robust maintainable program

Where we are: a quick but overgrown script

Robust Communication

Parallel Performance

Verified Correctness

Trustworthy Security
The Big Picture

Scripts *can* become robust programs

.... modularly, soundly, and effectively

New challenges and new opportunities
The Big Picture

Scripts can become robust programs

... modularly, soundly, and effectively

New challenges and new opportunities

Thank you