28 YEARS OF TYPES FOR UNTYPED LANGUAGES

Matthias Felleisen, PLT & NUPRL
A Personal Walk through Type Land

I am an untyped academic (1987)

type inference, à la Hindley & Milner

type inference, à la Heintze & Jaffar

explicit static types for untyped languages

modularity & contracts

incremental & idiomatic

functional & object-oriented

performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Suzuki, Borning & Ingalls 1981/82

modularity & contracts

incremental & idiomatic

performant (?)

functional & object-oriented

explicit static types for untyped languages

type inference, à la Hindley & Milner

type inference, à la Heintze & Jaffar

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Suzuki
Borning & Ingalls 1981/82

Thatte ‘92

type inference, à la Hindley & Milner

type inference, à la Heintze & Jaffar

explicit static types for untyped languages

modularity & contracts

incremental & idiomatic

functional & object-oriented

performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

- Suzuki
- Brachia & Griswold '93
- Borning & lngalls 1981/82
- Thatte '92

Type inference, à la Hindley & Milner

- type inference, à la Heintze & Jaffar

explicit static types for untyped languages

- modularity & contracts
- performant (?)

incremental & idiomatic

functional & object-oriented

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Type inference, à la Hindley & Milner

Type inference, à la Heintze & Jaffar

I still am an untyped academic (2016).

Suzuki

Brachia & Griswold '93

Henglein '94

Borning & Ingalls 1981/82

Thatte '92

modularity & contracts

incremental & idiomatic

explicit static types for untyped languages

performant (?)

functional & object-oriented
A Personal Walk through Type Land

I am an untyped academic (1987)

- Suzuki
- Borning & Ingalls 1981/82
- Thatte '92
- Henglein '94
- Brachia & Griswold '93
- Aiken & Wimmers 1991/94
- Henglein '94

- type inference,
  à la Hindley & Milner

- type inference,
  à la Heintze & Jaffar

- explicit static types
  for untyped languages

- modularity
  & contracts

- incremental
  & idiomatic

- functional &
  object-oriented

- performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

- Suzuki
- Brachia & Griswold '93
- Thatte '92
- Henglein '94

Aiken & Wimmers 1991/94

- type inference, à la Hindley & Milner
- type inference, à la Heintze & Jaffar

Gray & Flatt '05

- explicit static types for untyped languages
- modularity & contracts

- incremental & idiomatic
- functional & object-oriented

performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Suzuki
Borning & Ingalls 1981/82

Thatte '92

Henglein '94

Brachia & Griswold '93

type inference, à la Hindley & Milner

Aiken & Wimmers 1991/94

type inference, à la Heintze & Jaffar

Gray & Flatt '05

functional & object-oriented

modularity & contracts

incremental & idiomatic

explicit static types for untyped languages

Matthews & Findler '05

performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

- Brachia & Griswold '93
- Aiken & Wimmers 1991/94
- Gray & Flatt '05
- Matthews & Findler '05
- Siek & Taha '06

- type inference, à la Hindley & Milner
- type inference, à la Heintze & Jaffar
- modularity & contracts
- explicit static types for untyped languages
- incremental & idiomatic
- functional & object-oriented
- performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

- Suzuki
- Borning & Ingalls 1981/82
- Thatte '92
- Henglein '94
- Brachia & Griswold '93

- type inference, à la Hindley & Milner
- type inference, à la Heintze & Jaffar

- Aiken & Wimmers 1991/94
- Henglein '94

- modularity & contracts
- explicit static types for untyped languages
- functional & object-oriented
- incremental & idiomatic

- Gray & Flatt '05
- Matthews & Findler '05
- Siek & Taha '06

- contracts
- performant (?)

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

HL soft typing, sound and at scale

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

soft typing, with modules via contracts

functional & object-oriented

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

soft typing, with modules via contracts

I still am an untyped academic (2016).

Andrew Wright
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Cormac Flanagan

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

soft typing, with modules via contracts

functional & object-oriented

Andrew Wright

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987).

Mike Fagan

HL soft typing, sound and at scale

Cormac Flanagan

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

functional & object-oriented

I still am an untyped academic (2016).

Philippe Meunier

Andrew Wright
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Cormac Flanagan

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

functional & object-oriented

I still am an untyped academic (2016).

Philippe Meunier

Sam Tobin-Hochstadt

Andrew Wright
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Cormac Flanagan

soft typing, with accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

functional & object-oriented

I still am an untyped academic (2016).

Andrew Wright

Philippe Meunier

Sam Tobin-Hochstadt

Asumu Takikawa
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Cormac Flanagan

soft typing, with accessible type errors

Philippe Meunier

soft typing, with modules via contracts

Andrew Wright

interlanguage refactoring

Sam Tobin-Hochstadt

incremental & idiomatic

Typed Racket

Ben Greenman

functional & object-oriented

I still am an untyped academic (2016)

Asumu Takikawa
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

I still am an untyped academic (2016).

Andrew Wright

Cormac Flanagan

soft typing, sound and at scale

soft typing, accessible type errors

interlanguage refactoring

incremental & idiomatic

Typed Racket

Ben Greenman

Philippe Meunier

Sam Tobin-Hochstadt

Asumu Takikawa

functional & object-oriented

I still am an untyped academic (2016).
A Personal Walk through Type Land

I am an untyped academic (1987)

Mike Fagan

HL soft typing, sound and at scale

Andrew Wright

Cormac Flanagan

soft typing, accessible type errors

Philippe Meunier

interlanguage refactoring

Sam Tobin-Hochstadt

Typed Racket

I still am an untyped academic (2016).

Typeful Programming

 Luca Cardelli

Abstract

There exists an identifiable programming style based on the widespread use of type information handled through mechanical typechecking techniques. This typeful programming style is in a sense independent of the language it is embedded in; it adapts equally well to functional, imperative, object-oriented, and algebraic programming, and it is not incompatible with relational and concurrent programming.
Robert “Corky” Cartwright

User-Defined Data Types as an Aid to Verifying LISP Programs

ICALP 1976, pp. 228–256.
Write functional LISP, instead of imperative Algol:
- write functional programs
- describe them with user-defined types
- use these types to prove theorems

Functional programs are theories of first-order logic.
Write functional LISP, instead of imperative Algol:
- write functional programs
- describe them with user-defined types
- use these types to prove theorems

Functional programs are theories of first-order logic.

When I arrived at Rice in 1987:

“let's add types to Scheme.”
What does “adding types to Scheme” mean? Why is it hard?

;; Representing Russian dolls and computing their depth
;; RussianDoll = ‘doll u (cons RussianDoll ‘())

;; RussianDoll -> Natural
(define (depth r)
  (cond
   [(symbol? r) 0]
   [else (+ 1 (depth (first r)))]))

(depth ‘doll) ;; -> 0
(depth ‘((doll))) ;; -> 3
What does “adding types to Scheme” mean? Why is it hard?

```scheme
;; Representing propositions and checking tautology
;; Proposition = Boolean u [Boolean -> Proposition]

;; Proposition -> Boolean
(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```
type proposition = InL of bool | InR of (bool -> proposition)

let rec is_tautology p =
  match p with
  | InL b -> b
  | InR p -> is_tautology(p true) && is_tautology(p false)

is_tautology (InR(fun x -> InL true))
is_tautology (InR(fun x -> InR(fun y -> or (InL x) (InL y))))
type proposition = InL of bool | InR of (bool -> proposition)

let rec is_tautology p = match p with
  | InL b -> b
  | InR p -> is_tautology(p true) && is_tautology(p false)

is_tautology (InR(fun x -> InR(fun y -> or (InL x) (InL y))))

My idea: add a universal type to the program and add injections and projections where needed. That’s a practical version of Scott’s view that untyped languages are unityped.
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
  s &= (t \rightarrow \text{int}) \\
  t &= (v \rightarrow \text{char}) \\
  v &= \text{double}
\end{align*}
\]

\[
\begin{align*}
  s &= ((\text{double} \rightarrow \text{char}) \rightarrow \text{int}) \\
  t &= (\text{double} \rightarrow \text{char}) \\
  v &= \text{double}
\end{align*}
\]
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
  s &= (t \rightarrow \text{int}) \\
  t &= (v \rightarrow \text{char}) \\
  v &= \text{double}
\end{align*}
\]

unification ~ Gaussian elimination

\[
\begin{align*}
  s &= ((\text{double} \rightarrow \text{char}) \rightarrow \text{int}) \\
  t &= (\text{double} \rightarrow \text{char}) \\
  v &= \text{double}
\end{align*}
\]

think of all missing type declarations as variables, derive system of equations
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
\text{s} &= (t \rightarrow \text{int}) \\
\text{t} &= (v \rightarrow \text{char}) \\
\text{v} &= \text{double}
\end{align*}
\]
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
  s &= (t \to \text{int}) \\
  t &= (v \to \text{char}) \\
  v &= \text{double}
\end{align*}
\]

\[
\begin{align*}
  s &= ((\text{double} \to \text{char}) \to \text{int}) \\
  t &= (\text{double} \to \text{char}) \\
  v &= \text{double}
\end{align*}
\]

\[
\begin{align*}
  s &\subseteq \{ \text{dom : } t, \text{rng : } \text{int} \} \cup \{ \text{num : } 0 \} \\
  t &\subseteq \{ \text{dom : } v, \text{rng : } \text{char}, \text{num : } 0 \} \\
  v &\subseteq \text{double}
\end{align*}
\]
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
  s &= (t \rightarrow \text{int}) \\
  t &= (v \rightarrow \text{char}) \\
  v &= \text{double}
\end{align*}
\]
Fagan uses a record type algebra à la Remy [POPL ’88] instead of the ->, +, * algebra and then run Hindley Milner.

\[
\begin{align*}
  s &= (t \to \text{int}) \\
  t &= (v \to \text{char}) \\
  v &= \text{double} \\
  s &= ((\text{double} \to \text{char}) \to \text{int}) \\
  t &= (\text{double} \to \text{char}) \\
  v &= \text{double}
\end{align*}
\]
Representing Russian dolls and computing their depth

\[
\text{RussianDoll} = \text{`doll u (cons RussianDoll `())}
\]

\[\text{RussianDoll -> Natural}\]

\[
\text{(define (depth r)}
\]

\[
\text{(cond}
\]

\[
\text{[(symbol? r) 0]}
\]

\[
\text{[else (+ 1 (depth (first r)))]]}
\]

\[
\text{(depth `doll) ;; -> 0}
\]

\[
\text{(depth `(((doll))) ;; -> 3}
\]

\[
\text{[\[\mu (rd)(U `doll (cons RussianDoll `()))]\]
\]

\[
\text{->}
\]

\[
\text{Natural]}
\]
Fagan’s “soft typer” works on all of our “hard” examples

;; Representing propositions and checking tautology
;; Proposition = Boolean u [Boolean -> Proposition]

;; Proposition -> Boolean

(define (tautology? p)
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
Fagans’ soft typer *cannot*
- present types in an accessible manner
- deal with more than small toy programs
- cope with anything but the core functional language
Fagans’ soft typer *cannot*
- present types in an accessible manner
- deal with more than small toy programs
- cope with anything but the core functional language

Can we deal with
- 1,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an “actionable” manner
Wright’s 1992-1993 engineers Soft Scheme (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization
Wright’s 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization
Wright’s 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

\[
s \subseteq \{ \text{dom} : t, \text{rng} : \text{int} \} \cup \{ \text{num} : 0 \}
\]
\[
t \subseteq \{ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0 \}
\]
\[
v \subseteq \text{double}
\]
Wright’s 1992-1993 engineers Soft Scheme (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

\[ s \subseteq \{ \text{dom : t, rng : int} \} \cup \{ \text{num : 0} \} \]
\[ t \subseteq \{ \text{dom : v, rng : char, num : 0} \} \]
\[ v \subseteq \text{double} \]

\[ s = \ldots \quad \gamma = \emptyset \]
\[ t = \ldots \quad \delta = \emptyset \]
\[ v = \text{double} \quad \varepsilon = \emptyset \]
Wright’s 1992-1993 engineers **Soft Scheme** (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

\[
\begin{align*}
  s & \subseteq \{ \text{dom} : t, \text{rng} : \text{int} \} \cup \{ \text{num} : 0 \} \\
  t & \subseteq \{ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0 \} \\
  v & \subseteq \text{double}
\end{align*}
\]
Wright’s 1992-1993 engineers Soft Scheme (based on Chez) into an ML-like variant of Scheme with idiomatic type inference

- modify type algebra (add in set!, call/cc)
- improve implementation of type algebra
- report type errors at source level
- use types for optimization

\[ \begin{align*}
  s & \subseteq \{ \text{dom} : t, \text{rng} : \text{int} \} \cup \{ \text{num} : 0 \} \\
  t & \subseteq \{ \text{dom} : v, \text{rng} : \text{char}, \text{num} : 0 \} \\
  v & \subseteq \text{double}
\end{align*} \]

\[ \begin{align*}
  s &= \ldots \\
  y &= \emptyset \\
  t &= \ldots \\
  \delta &= \emptyset \\
  v &= \text{double} \\
  \epsilon &= \emptyset
\end{align*} \]

chez program.ss -o3
My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, “extensible den. semantics”)

Shriram Krishnamurthi’s starter project

Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)
My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, “extensible den. semantics”)

RESULT: It works.

Shriram Krishnamurthi’s starter project

Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)
<table>
<thead>
<tr>
<th><strong>My first sabbatical (1993-94)</strong></th>
<th><strong>Shriram Krishnamurthi’s starter project</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Write many 1,000 line programs in SML and Soft Scheme (Foxnet, “extensible den. semantics”)</td>
<td>Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)</td>
</tr>
<tr>
<td>RESULT: It works.</td>
<td>Type errors in SML/NJ were torture.</td>
</tr>
<tr>
<td>My first sabbatical (1993-94)</td>
<td>Shriram Krishnamurthi’s starter project</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Write many 1,000 line programs in SML and Soft Scheme (Foxnet, “extensible den. semantics”)</td>
<td>Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)</td>
</tr>
<tr>
<td>RESULT: It works.</td>
<td>Type errors in SML/NJ were torture.</td>
</tr>
<tr>
<td></td>
<td>Soft Scheme’s were violations of the Geneva convention.</td>
</tr>
</tbody>
</table>
My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, “extensible den. semantics”)

RESULT: It works.

Soft Scheme supports my module’s but is not modular.

Shriram Krishnamurthi’s starter project

Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

Soft Scheme’s were violations of the Geneva convention.
My first sabbatical (1993-94)

Write many 1,000 line programs in SML and Soft Scheme (Foxnet, "extensible den. semantics")

RESULT: It works.

Soft Scheme supports my module’s but is not modular.

Shriram Krishnamurthi’s starter project

Analyze Sitaram’s SLaTeX (now a benchmark) with Soft Scheme (3,500 lines of real-world code)

Type errors in SML/NJ were torture.

Soft Scheme’s were violations of the Geneva convention.

Undergraduates cannot use Soft Scheme in PL course.
Errors matter.
Errors matter.

Developers matter.
Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar’s SBA.
Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar’s SBA

\[
\begin{align*}
\text{double} \subseteq \text{dom}(t) \\
t \subseteq \text{rng}(v) \\
\text{dom}(v) \subseteq \text{double}
\end{align*}
\]
Flanagan uses a regular type algebra, but solves inequations instead of equations, via an approach inspired by Heintze & Jaffar’s SBA.

double \subseteq \text{dom}(t) \\
t \subseteq \text{rng}(v) \\
\text{dom}(v) \subseteq \text{double}

\begin{align*}
s &= \ldots \\
t &= \ldots \\
v &= (\to \text{double} \ (\to \text{double} \ldots))
\end{align*}

transitive closure through constructors

the solution is a least-fix point in a lattice
Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar’s SBA.

\[ \text{double} \subseteq \text{dom}(t) \]
\[ t \subseteq \text{rng}(v) \]
\[ \text{dom}(v) \subseteq \text{double} \]

\[ \text{Listof} \ X \] \(\subseteq\) \text{dom(first @1)} \checkmark

\[ \text{Pairof} \ Y \ Z \] \(\subseteq\) \text{dom(first @2)} \xmark

\[ s = \ldots \]
\[ t = \ldots \]
\[ v = (\rightarrow \text{double} \ (\rightarrow \text{double} \ldots)) \]

The solution is a least-fix point in a lattice.

Transitive closure through constructors.

Compare with specifications for primitive operations.
Flanagan uses a regular type algebra, but solves *inequations* instead of equations, via an approach inspired by Heintze & Jaffar’s SBA.

\[
\text{double} \subseteq \text{dom}(t) \\
t \subseteq \text{rng}(v) \\
\text{dom}(v) \subseteq \text{double}
\]

\[
\text{the solution is a least-fix point in a lattice}
\]

\[
\text{transitive closure through constructors}
\]

\[
[\text{Listof } X] \subseteq \text{dom(first @1)} \quad \times
\]

\[
\text{because (first '()) raises an exn}
\]

\[
[\text{Pairof } Y \ Z] \subseteq \text{dom(first @2)} \quad \checkmark
\]

\[
\text{compare with specifications for primitive operations}
\]

\[
s = \ldots \\
t = \ldots \\
v = (\rightarrow \text{double} \ (\rightarrow \text{double} \ \ldots))
\]
• HM performs in near-linear time in practice
• HM is easy to understand in principle
• HM “smears” origin information across solution due to bi-directional flow

• SBA performs in linear time up to 2,500 loc
• SBA is also easy to explain to programmers
• SBA pushes information only along actual edges in the flow graph
- HM performs in near-linear time in practice.
- HM is easy to understand in principle.
- HM “smears” origin information across solution due to bi-directional flow.
- SBA performs in linear time up to 2,500 loc.
- SBA is also easy to explain to programmers.
- SBA pushes information only along actual edges in the flow graph.

And we can visualize those!
;; reachable : sgn graph -> graph
;; to produce a graph whose visited fields are marked
;; true if the nodes are reachable from a-node
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
  (letrec ((reachable
    (lambda (a-node)
      (cond
        [(boolean? a-node) graph]
        [(node-visited a-node) (void)]
        [else (begin
                    (set-node-visited! a-node true)
                    (reachable (node-next a-node)))])))))

|| TEST SUITE -----------------------------------------------
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)

(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))

(map node-name
  (reachable (first the-graph) the-graph))
;; reachable : sgn graph -> graph
;; to produce a graph whose visited fields are marked
;; true if the nodes are reachable from a-node
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
 (letrec ((reachable
            (lambda (a-node)
                (cond
                  [(boolean? a-node) graph]
                  [(node-visited a-node) (void)]
                  [else (begin
                          (set-node-visited! a-node true)
                          (reachable (node-next a-node))))])))

  (cond
   [(empty? graph) empty]
   [else (begin
            (for-each (lambda (n) (set-node-visited! n false)) graph)
            (reachable (first graph))))])

#| TEST SUITE |
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)

(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) (lookup 'A the-graph) the-graph))

#| map node-name
(map reachable (first the-graph the-graph))

Welcome to MrSpidey, version 102/16.
CHECKS:
  map check in file "graph-spidey.ss": line 93, column 2
  first check in file "graph-spidey.ss": line 94, column 18
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
;;; false if not
;;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
  (letrec ((reachable
             (lambda (a-node)
               (cond
                 [(boolean? a-node) graph]
                 [(node-visited a-node) (void)]
                 [else (begin
                         (set-node-visited! a-node true)
                         (reachable (node-next a-node)))])))
    (cond
      [(empty? graph) empty]
      [else (begin
              (for-each (lambda (n) (set-node-visited! n false)) graph)
              (reachable [first graph]))])))

; TEST SUITE -----------------------------------------------
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)

(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph))
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph))
(not (route-exists? (lookup 'F the-graph) (lookup 'A the-graph) the-graph))

(map node-name

  (reachable [first the-graph] the-graph)

  (map
   (lambda (y1) (structure:node sym bool (union y1 false)))
   (union nil void (ocns y1 (listof y1))))

Welcome to MRSpikey, version 102/16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 16
TOTAL CHECKS: 2 (of 55 possible checks is 3.5%)
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define (reachable a-node graph)
  (letrec ((reachable
    (lambda (a-node)
      (cond
        [[(boolean? a-node) graph]
         [(node-visited a-node) (void)]
         [else (begin
           (set-node-visited! a-node true)
           (reachable (node-next a-node))))])))
    (cond
      [[(empty? graph) empty]
       [else (begin
         (for-each (lambda (n) (set-node-visited! n false)) graph)
         (reachable (first graph)))]))))

(* | TEST SUITE --------------------------*
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph)

(map node-name
  (reachable (first the-graph) the-graph)
  (rec
    ((y1 (structure:node sym bool (union y1 false)))
     (union nil void (ccns y1 (listof y1)))))
)

Welcome to MsSpidey, version 162/16.
CHECKS: 
  map check in file "graph-spidey.ss": line 93, column 2
  first check in file "graph-spidey.ss": line 94, column 16
TOTAL CHECKS: 2 (of 56 possible checks is 3.5%)
;; false if not
;; effect: to mark all those nodes in graph that are reachable from a-node
(define reachable a-node graph)
(letrec ((reachable
    (lambda (a-node)
      (cond
        [(boolean? a-node) graph]
        [(node-visited a-node) (void)]
        [else (begin
           (set-node-visited! a-node true)
           (reachable (node-next a-node))))])))
  (cond
    [(empty? graph) empty]
    [else (begin
      (for-each (lambda (n) (set-node-visited! n false)) graph)
      (reachable (first graph))))])

(* | TEST SUITE -----------------------------*
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph)
(not (route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph)
(not (route-exists? (lookup 'F the-graph) (lookup 'G the-graph) the-graph)

(map node-name

  (reachable (first the-graph) the-graph)

  (rec
   ((y1 (structure:node str bool (union y1 false)))]
   (union nil (void (void (y1 (listof y1)))))

Welcome to MsSpidey, version 1.0.16.
CHECKS:
map check in file "graph-spidey.ss": line 93, column 2
first check in file "graph-spidey.ss": line 94, column 16
TOTAL CHECKS: 2 (of 55 possible checks is 3.5%)
(define (reachable a-node graph)
  (letrec ((reachable
    (lambda (a-node)
      (cond
        [(empty? graph) empty]
        [else (begin
          (for-each (lambda (n) (set-node-visited! n false)) graph)
          (reachable (first graph))))]))))

(* TEST SUITE -----------------------------------------------)
(route-exists? (lookup 'A the-graph) (lookup 'B the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'C the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'E the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'F the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph)
(route-exists? (lookup 'A the-graph) (lookup 'D the-graph) the-graph)

(map node-name
    (reachable (first the-graph) the-graph)
    I(union nil void (cons y1 (listof y1))))

Welcome to MrSpidey, version 102/16.
CHECKS:
  map check in file "graph-spidey.ss": line 93, column 2
  first check in file "graph-spidey.ss": line 94, column 16
TOTAL CHECKS: 2 (of 58 possible checks is 3.5%)
the source of void
the source of void

the flow of void to first
Flanagan can deal with
- 2,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an “actionable” manner
Flanagan can deal with
- 2,000 lines of code
- full Scheme (assignment, continuations)
- explain types
- report errors in an “actionable” manner

Can we deal with
- get juniors and seniors to use it (future devs)
- improve precision (e.g., arity of functions)
- “modules” (independently developed pieces)?
Even with juniors and seniors, the good news...
The not so good news

;; Natural Symbol -> S-expression

(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff))]))

(wrap 3 'pizza) ;; -> '(((pizza)))
(wrap 2 'doll) ;; -> '(((doll)))
The not so good news

;; Natural Symbol -> S-expression
(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff))]))

(wrap 3 'pizza) ;; -> '(((pizza)))
(wrap 2 'doll) ;; -> '(((doll)))

~ (list depth stuff) = arg
The not so good news

(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff)]))

(wrap 3 'pizza) ;; -> '(((pizza)))
(wrap 2 'doll) ;; -> '(((doll)))
The not so good news

(define (wrap depth stuff)
  (cond
    [(zero? depth) stuff]
    [else (list (wrap (- depth 1) stuff))]))

(wrap 3 'pizza) ;; -> '(((pizza)))
(wrap 2 'doll) ;; -> '(((doll)))

~ (list depth stuff) = arg

~ (second arg)

~ (first arg)

Selectors Make Set-Based Analysis Too Hard

Philippe Meunier, Robert Bruce Findler†, Paul Steckler and Mitchell Wand

Department of Computer Science
University of Chicago
Chicago, IL 60637
robb@cs.uchicago.edu

College of Computer and Information Science
Northeastern University
Boston, MA 02115
{meunier, steck, wand}@ccs.neu.edu
;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... ))))]
    ;; --- client code
    ... )
  ... (extract (dispatch-table k) m)...
The bad news

;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))])
    ;; --- client code
    ... )
...
(extract (dispatch-table k) m)...)
The bad news

;;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... ))))])
    ;; --- client code
    ... )
  ...
  (extract (dispatch-table k) m)...

;;; Natural -> Table
(define (dispatch-table n)
  (define v (make-vector n))
  (for ((i v)) (vector-set! v i ...))
    ;; --- client code
    ... )
  ...
  (extract (dispatch-table k) m)...)
The bad news

;;; Natural -> Table
(define (dispatch-table n)
  (let ([v (build-vector n (lambda (i) (lambda (x) ... )))]))
    ;; --- client code
    ...
  ...)

... (extract (dispatch-table k) m)...
The bad news

Small syntactic changes without semantic meaning imply large changes to inferred types
The worse news

It's not only $n^8$, it's also whole-program only.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1 min</td>
</tr>
<tr>
<td>2,000</td>
<td>2 min</td>
</tr>
<tr>
<td>3,000</td>
<td>3 min</td>
</tr>
<tr>
<td>3,500</td>
<td>20 min</td>
</tr>
<tr>
<td>40,000</td>
<td>10 hrs</td>
</tr>
</tbody>
</table>
The worse news

1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs

It's not only $n^8$, it's also whole-program only.

Components
The worse news

It's not only $n^8$, it's also whole-program only.

Components

Constraints

$\{ s \subseteq \text{dom: } t, \text{rng: } \text{int} \}$
$\{ t \subseteq \text{dom: } v, v \text{ (num: 0)} \}$
$\{ v \subseteq \text{double} \}$

$\{ u \subseteq \text{dom: } s \}$
$\{ i \subseteq \text{dom: } v, v \text{ (num: 0)} \}$
$\{ w \subseteq \text{double} \}$

$\{ h \subseteq \text{dom: } t, \text{rng: } \text{int} \}$
$\{ t \subseteq \text{integer} \}$
$\{ j \subseteq \text{double} \}$
The worse news

It's not only $n^8$, it's also whole-program only.

Components

Constraints

Solution

explicit sets & set mismatches
It's not only $n^8$, it's also whole-program only.

1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs

Components

Constraints

Solution

explicit sets & set mismatches
The worse news

It's not only $n^8$, it's also whole-program only.

Components

Constraints

Solution

Now we can work with 1 module and get on-line analysis
The worse news

1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs

What's worse, we can't just add another module.

Components

<table>
<thead>
<tr>
<th>Components</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>s ∈ (dom : t, rng : int)</td>
<td>u ∈ (dom : s)</td>
</tr>
<tr>
<td>t ∈ (dom : v) u (num : 0)</td>
<td>i ∈ (dom : v) u (num : 0)</td>
</tr>
<tr>
<td>y ∈ double</td>
<td>w ∈ double</td>
</tr>
<tr>
<td>h ∈ (dom : t, rng : int)</td>
<td>t ∈ integer</td>
</tr>
<tr>
<td>j ∈ double</td>
<td></td>
</tr>
</tbody>
</table>

Constraints

It costs $O(n^2)$ space (writing, reading) to store graph constraints.

Solution

explicit sets & set mismatches
The worse news

1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs

What's worse, we can't just add another module.

Components

It isn't really modular in the sense of ML's structures.

Solution

It costs $O(n^2)$ space (writing, reading) to store graph constraints.

explicit sets & set mismatches
Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints.
Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints.
Meunier exploits Eiffel-style contracts (generalized to a higher-order setting) to describe module interfaces, derive constraints.

It works in theory. We never got it to work well in practice.

Use contracts in lieu of signatures.

1,000 lines ~ 1 min
2,000 lines ~ 2 min
3,000 lines ~ 3 min
3,500 lines ~ 20 min
40,000 lines ~ 10 hrs

Solution

$ \subseteq \{ \text{dom: } \text{t}, \text{rng: } \text{int} \}$
$ t \subseteq \{ \text{dom: } \text{v}, \{ \text{num: } 0 \} \}$
$ w \subseteq \text{double}$

$ h \subseteq \{ \text{dom: } \text{t}, \text{rng: } \text{int} \}$
$ t \subseteq \text{integer}$
$ j \subseteq \text{double}$

explicit sets & set mismatches
Modularity matters.

- Our code base has grown to 500,000 loc.
- A language renaissance has spread Untyped Languages beyond their niche uses.
Modularity matters.

- Our code base has grown to 500,000 loc.
- A language renaissance has spread Untyped Languages beyond their niche uses.

Signatures matter.

- Nobody ought to read an entire module to understand its services.
- Racket programmers use contracts as signatures.
Can we add types to this code \textbf{without} the ML-style projections/injections?

```scheme
;; Representing Russian dolls and computing their depth
;; RussianDoll = `doll u (cons RussianDoll `())

;; RussianDoll -> Natural
(define (depth r)
  (cond
   [(symbol? r) 0]
   [else (+ 1 (depth (first r)))]))

(deepth `doll) ;; -> 0
(deepth `(((doll))) ;; -> 3
```
Can we add types to this code without the ML-style projections/injections?

;; Representing Russian dolls and computing their depth

TYPE RussianDoll = 'doll u (cons RussianDoll '())

(define (depth r : RussianDoll) : Natural
  (cond
    [(symbol? r) 0]
    [else (+ 1 (depth (first r)))]))

(depth 'doll) ;; -> 0
(depth '(((doll))) ;; -> 3
Can we add types to this code **without** the ML-style projections/injections?

```Scheme
;; Representing propositions and checking tautology

TYPE Proposition = Boolean u [Boolean -> Proposition]

(define (tautology? p : Proposition) : Boolean

  (cond
   [(boolean? p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y)))))
```
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level
Tobin-Hochstadt \textit{incrementally} and \textit{idiomatically} adds types to existing large systems at appropriate granularity level.

\textbf{ASSUME} a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

Gray, Findler, Flatt
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level.

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable “components” (files, packages, classes, modules) with clear boundaries.
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level.

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable “components” (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a “by need” basis plus the addition of typed components — incrementality.

Gray, Findler, Flatt
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level.

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable “components” (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a “by need” basis plus the addition of typed components — incrementality.

WANTED annotations should go on variables and other names and should not disturb existing code — idiomaticity.
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level

ASSUME a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

ASSUME identifiable “components” (files, packages, classes, modules) with clear boundaries.

WANTED a framework for component-by-component addition of type annotation on a “by need” basis plus the addition of typed components — incrementality

WANTED annotations should go on variables and other names and should not disturb existing code — idiomaticity

WANTED the type annotations ought to be useful and meaningful — type soundness

Gray, Findler, Flatt
Tobin-Hochstadt incrementally and idiomatically adds types to existing large systems at appropriate granularity level.

**ASSUME** a large system written in an untyped language. Translating it into a typed language is prohibitively expensive.

**ASSUME** identifiable “components” (files, packages, classes, modules) with clear boundaries.

**WANTED** a framework for component-by-component addition of type annotation on a “by need” basis plus the addition of typed components — *incrementality*

**WANTED** annotations should go on variables and other names and should **not** disturb existing code — *idiomaticity*

**WANTED** the type annotations ought to be **useful** and **meaningful** — *type soundness*

Gray, Findler, Flatt

And all of this works for (almost) the full language — *coverage*
Representing Russian dolls and computing their depth

(define-type RussianDoll (U 'doll [cons RussianDoll '()])))

(define (depth {r : RussianDoll}) : Natural
  (cond
    [(symbol? r) 0]
    [else (+ 1 (depth (first r)))]))

(depth 'doll) ;; → 0
(depth '((((doll))) ;; → 3

Typed Racket satisfies “idiomaticy”
Typed Racket satisfies “idiomaticy”

;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) : Boolean
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
;; Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) : Boolean
  (cond
   [(boolean? p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))])

(tautology? true)
(tautology? (lambda (x) (lambda (_) (or x y))))

Typed Racket satisfies “idiomaticy”

no projection needed

no injection needed
Representing propositions and checking tautology

(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) : Boolean
  (cond
   [(boolean? p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
Typed Racket satisfies “idiomaticy” via `flow propositions`

```
;; Representing propositions and checking tautology
(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition}) :
  (cond
    [(boolean? p) p]
    [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```
Typed Racket satisfies “idiomaticy” via `flow propositions``

```racket
;;; Representing propositions and checking tautology
(define-type Proposition (U Boolean [Boolean -> Proposition]))
(define (tautology? {p : Proposition}):
  (cond
   [(boolean? p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))]))
(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

boolean? : Any -> Boolean: 
``and if it is true, the given value belongs to Boolean``

p is **not** a Boolean, ergo it must be in [Boolean -> Proposition]
Typed Racket satisfies “idiomaticy” via `flow propositions`

```racket
;; Representing propositions and checking tautology
(define-type Proposition (U Boolean [Boolean -> Proposition]))

(define (tautology? {p : Proposition})
  (cond
   [(boolean? p) p]
   [else (and (tautology? (p true)) (tautology? (p false)))]))

(tautology? true)
(tautology? (lambda (x) (lambda (y) (or x y))))
```

- boolean? : Any -> Boolean: 
  "`and if it is true, the given value belongs to Boolean"

- p applied to true is OK

- p is not a Boolean, 
  ergo it must be in [Boolean -> Proposition]
Typed Racket satisfies “idiomaticity” via ``flow propositions``

\[ \Gamma \vdash e : \tau | (p^+,p^-) \]
Typed Racket satisfies “idiomaticy” **via ``flow propositions``**

**IN type environment** (the type of variables in e)

**the expression e HAS**

**type τ**

\[ \Gamma \vdash e : \tau \mid (p^+, p^-) \]
Typed Racket satisfies "idiomaticity" via "flow propositions"

IN type environment (the type of variables in e)

the expression e HAS type \( \tau \)

\[ \Gamma \vdash e : \tau \mid (p+, p-) \]

and if e evaluates to a Truish value, we KNOW \( p^+ \)
Typed Racket satisfies “idiomaticy” via `flow propositions`.

\[ \Gamma \vdash e : \tau \mid (p^+, p^-) \]

IN type environment (the type of variables in \( e \))

the expression \( e \) HAS type \( \tau \)

and if \( e \) evaluates to a Truish value, we KNOW \( p^+ \)

and if \( e \) evaluates to a False value, we KNOW \( p^- \)
Typed Racket satisfies “idiomaticity” via `flow propositions`.

IN type environment (the type of variables in e)

The expression e HAS type \( \tau \)

\( \Gamma \vdash e : \tau \mid (p^+, p^-) \)

and if e evaluates to a Truish value, we KNOW \( p^+ \)

and if e evaluates to a False value, we KNOW \( p^- \)

The **knowledge** deals with plain values and paths into values:

- (odd? n) \( \sim \) if this yields False, n is even
- (prime? (second l)) \( \sim \) if this yields True, we know l has the shape [one, two, ?] and two is a prime number.
Typed Racket satisfies “idiomaticity” via `flow propositions`.

In type environment (the type of variables in e)

The expression e HAS type $\tau$

and if e evaluates to a Truish value, we KNOW $p^+$

and if e evaluates to a False value, we KNOW $p^-$

The logic can cope with the usual Boolean primitives in a programming language: and, or, not, if (conditionals), etc.

The knowledge deals with plain values and paths into values:
- $(\text{odd? } n) \rightarrow$ if this yields False, n is even
- $(\text{prime? } (\text{second } l)) \rightarrow$ if this yields True, we know l has the shape [one, two, ?] and two is a prime number.
Typed Racket satisfies “incrementality”

```racket
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(over? s) (+ (area (over-bot s)) (area (over-top s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(plain? s) (plain-area s)]
   [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(over? s) (+ (area (over-bot s)) (area (over-top s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```
;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(over? s) (+ (area (over-bot s)) (area (over-top s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```

```
;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(over? s) (+ (area (over-bot s)) (area (over-top s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```
Typed Racket satisfies “incrementality”

```racket
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(plain? s) (plain-area s)]
   [else (+ (area (over-top s)) (area (over-bot s)))]
   [apply + (map rect-area (filter rect? s))])))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

Racket has always been a family of languages.
Typed Racket satisfies “incrementality”

Racket has always been a family of languages

Racket modules already specify their implementation language

(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(plain? s) (plain-area s)]
   [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))

(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(plain? s) (plain-area s)]
   [(over? s) (+ (area (over-top s)) (area (over-bot s)))]
   [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
Typed Racket satisfies “incrementality”

Racket has always been a family of languages

Racket modules already specify their implementation language

#lang racket
Typed Racket satisfies “incrementality”

Racket has always been a family of languages

Racket modules already specify their implementation language

Adding #lang typed/racket is easy

#lang racket
Typed Racket satisfies “incrementality” at the module level.
Typed Racket satisfies “incrementality” at the module level

(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect  = (make-rect Posn Number Number)
;; Circ  = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ (area (over-bot s)))
      [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]
    [else (apply + (map rect-area (filter rect? s)))]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
Typed Racket satisfies “incrementality” at the module level

```racket
(define-struct rect (nw width height))
(define-struct circ (cntr radius))
(define-struct over (top bot))

;; Shape = Plain | (make-over Shape Shape) | [Listof Plain]
;; Plain = Rect | Circ
;; Rect = (make-rect Posn Number Number)
;; Circ = (make-circ Posn Number)

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
   [(over? s) (+ (area (over-bot s)))
   [else (apply + (map rect-area (filter rect? s)))])])

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Plain -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
   [(rect? s) (rect-area s)]
   [(circ? s) (rect-area s)])
  
;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

```racket
(define-stuct rect (nw width height))
(define-stuct circ (cntr radius))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))
```
Typed Racket satisfies “incrementality” at the module level

How do typed/racket communicate with racket
Typed Racket satisfies “incrementality” at the module level.
Typed Racket satisfies “incrementality” at the module level

```
;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Shape -> Number
;; the area of all rectangles in this s
(define (area s)
  (cond
    [(over? s) (+ (area (over-bot s)))
     (area (over-top s))]
    [else (apply + (map rect-area (filter rect? s)))]))

;; Any -> Boolean
;; is this p a plain shape?
(define (plain? p)
  (or (rect? p) (circ? p)))

;; Shape -> Number
;; the area of this plain shape s
(define (plain-area s)
  (cond
    [(rect? s) (rect-area s)]
    [(circ? s) (rect-area s)]))

;; Rect -> Number
;; the area of this rectangle r
(define (rect-area s)
  (* (rect-width s) (rect-height s)))
```

Who’s responsible for which part of the communication?

How do typed/racket communicate with racket?
Typed Racket satisfies “incrementality” at the module level.

Who’s responsible for which part of the communication?

How do typed/racket communicate with racket?
Typed Racket satisfies “incrementality” at the module level

(\( f (\lambda (x) \text{"howdy"}) \))

Who’s responsible for which part of the communication?

How do typed/racket communicate with racket?
Typed Racket satisfies “incrementality” at the module level

\[
(f \ (\lambda (x) \ "howdy"))
\]

Who’s responsible for which part of the communication?

How do typed/racket communicate with racket?

Do we need to discover this “miscommunication”?
Typed Racket satisfies “incrementality” at the module level

(f (λ (x) "howdy"))

How do typed/racket communicate with racket

Who’s responsible for which part of the communication?

Do we need to discover this “miscommunication”?

If so, who should we blame for the miscommunication?
Typed Racket satisfies "incrementality" at the module level

\[
(f \ (\lambda \ (x) \ "howdy"))
\]
Typed Racket satisfies “incrementality” at the module level

(f (λ (x) "howdy"))

If you think it’s acceptable to let this kind of mistake slip, welcome to industrial-strength, modern day C++ reincarnation. (This can’t possibly happen.)
Typed Racket satisfies “incrementality” at the module level

(f (λ (x) "howdy"))

If you think it’s acceptable to let this kind of mistake slip, welcome to industrial-strength, modern day C++ reincarnation. (This can’t possibly happen.)

If you think that this kind of miscommunication deserves the programmer’s attention, you want “type sound” interactions.
Typed Racket satisfies “incrementality” at the module level

If you think it’s acceptable to let this kind of mistake slip, welcome to industrial-strength, modern day C++ reincarnation. (This can’t possibly happen.)

If you think that this kind of miscommunication deserves the programmer’s attention, you want “type sound” interactions.

And if you want soundness, the run-time check ought to blame this connection between the two arrows.
Typed Racket satisfies "soundness" at the module levels via the compilation of types to higher-order contracts.

(f (λ (x) "howdy"))
Typed Racket satisfies "soundness" at the module levels via the compilation of types to higher-order contracts.
Typed Racket satisfies “**soundness**” at the module levels via the compilation of types to higher-order contracts.
Typed Racket satisfies "soundness" at the module levels via the compilation of types to higher-order contracts.
Findler introduced higher-order contracts [ICFP 2002]

Dimoulas developed elegant, flexible technique for proving the soundness of mixed systems [ESOP 2012]

Theorem

For all mixed programs $e \in \text{Racket} \oplus \text{Type Racket}$, one of these statements holds:
- $\text{eval}(e)$ is a value
- $\text{eval}(e)$ is a known exception from TR
- $\text{eval}(e)$ is a contract error blaming a specific boundary between a typed and an untyped module
- $\text{eval}(e)$ diverges.
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str
      ...))))
```

```racket
... (add-search analysis-presentation%)
```
Typed Racket can cope with (almost) all linguistic constructs from Racket

```
#lang racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

A function from class to class

```
… (add-search analysis-presentation%)
```

Asumu Takikawa
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))

... (add-search analysis-presentation%)
```

A function from class to class

Asumu Takikawa
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang racket
;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...
    ))
)
```

A function from class to class

... (add-search analysis-presentation%)

exported ...

... and used in a separate module
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))

... (add-search analysis-presentation%)...
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang racket
;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
                  ...
  )))

Yes, this is real-world code.

Yes, you can do this is Python, too.

don't need explicit exported ...

... and used in a separate module
```

... (add-search analysis-presentation%)...
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang typed/racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str))
    ...)))
```

```racket
... (add-search analysis-presentation%)...```
Typed Racket can cope with (almost) all linguistic constructs from Racket

;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...
    )))
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...))
```

What kind of types do classes have?

What contracts do these types compile to?
Typed Racket can cope with (almost) all linguistic constructs from Racket

```racket
#lang typed/racket
;; a mixing that adds search capabilities
(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))
```

```racket
#lang racket
... (add-search analysis-presentation%)...```
Typed Racket can cope with (almost) all linguistic constructs from Racket

Innovations needed:
- class types, with row polymorphism
- sealing contracts for enforce polymorphism
- innovative soundness proof

(define (add-search %)
  (class %
    (inherit text)
    (field [state #f])
    (define/public (search str)
      ...)))

Gradual Typing for First-Class Classes
Asumu Takikawa  T. Stephen Strickland  Christos Dimoulas
Sam Tobin-Hochstadt  Matthias Felleisen
PLT, Northeastern University
{asumu, sstrickl, chrdimo, samth, matthias}@ccs.neu.edu

Towards Practical Gradual Typing
Asumu Takikawa¹, Daniel Feltey¹, Earl Dean², Matthew Flatt³, Robert Bruce Findler⁴, Sam Tobin-Hochstadt², and Matthias Felleisen¹

¹ Northeastern University
Boston, Massachusetts
asumu@ccs.neu.edu, dfeltey@ccs.neu.edu, matthias@ccs.neu.edu

² Indiana University
Bloomington, Indiana
samth@cs.indiana.edu, edean@cs.indiana.edu

³ University of Utah
Salt Lake City, Utah
mflatt@cs.utah.edu

⁴ Northwestern University
Evanston, Illinois
robbymflatt@cs.northwestern.edu
Typed Racket can cope with (almost) all linguistic constructs from Racket

Innovations needed:
- class types, with row polymorphism
- sealing contracts for enforce polymorphism
- innovative soundness proof

Translating theory into practice:
- design for usability
- implementation engineering
- performance evaluation

```
(define (add-search %)
  (class %
    ...)))
```

#lang racket

... (add-search analysis-presentation%)...
Design matters.

- Typed Racket is *incremental*.
- Typed Racket is *idiomatic*.
- Typed Racket is *sound*.
- Typed Racket *covers it all*.
- Does it work?
- Does it really work?
- Truthfully?
- No cheating?
Design matters. Evaluation matters even more.

- Typed Racket is incremental. ✅
- Typed Racket is idiomatic. ✅
- Typed Racket is sound. ✅
- Typed Racket covers it all. ✅

- Does it work?
- Does it really work?
- Truthfully?
- No cheating?
Design needs feedback loop.
Design needs feedback loop.

Two kinds of evaluation:
- formative
- summative
Design needs feedback loop.

Two kinds of evaluation:
- formative
- summative

Three aspects to design evaluation:
- effort of adding annotations
- usability with (future) dev
- performance of mixed systems
Design needs feedback loop.

Two kinds of feedback:
- idea level (back to drawing board)
- realization level (previously

Three aspects to design evaluation:
- effort of adding annotations
- usability with (future) dev
- performance of mixed systems

Two kinds of evaluation:
- formative
- summative
Design needs feedback loop.
Design needs feedback loop.

Effort of adding type annotations:
- FP style calls for 3-5% changes
- OOP style needs 10-15% changes
- mostly annotations, some changes to code to get around the type checker
Design needs feedback loop.

Usability of Typed Racket:
- TR devs are easily proficient
- seniors in a PL course
- real-world users

Effort of adding type annotations:
- FP style calls for 3-5% changes
- OOP style needs 10-15% changes
- mostly annotations, some changes to code to get around the type checker
Design needs feedback loop.

Usability of Typed Racket:
- TR devs are easily proficient
- seniors in a PL course
- real-world users

Effort of adding type annotations:
- FP style calls for 3-5% changes
- OOP style needs 10-15% changes
- mostly annotations, some changes to code to get around the type checker
Greenman create and evaluate all possible mixed configurations of existing multi-module systems

WHICH MODULE WILL A PROGRAMMER EQUIP WITH

A  B  C
Greenman create and evaluate all possible mixed configurations of existing multi-module systems

 WHICH MODULE WILL A PROGRAMMER EQUIP WITH

Ben Greenman
Greenman create and evaluate all possible mixed configurations of existing multi-module systems

WHICH MODULE WILL A PROGRAMMER EQUIP WITH
Greenman create and evaluate all possible mixed configurations of existing multi-module systems
Greenman create and evaluate all possible mixed configurations of existing multi-module systems.

We don’t know. All $2^N$ of these configurations are feasible.

Which module will a programmer equip with?
POPL 2016 and Journal of Functional Programming [in preparation]

- ~20 modular programs with ~100,000 configurations.
- 90% of those impose a penalty of 3x or more.
- many configurations impose a 10x penalty
- some configurations cost as much as 100x of the baseline

Typed Racket’s contract impose a high run-time cost on mixed system performance.
Typed Racket’s contract impose a high run-time cost on mixed system performance.

**POPL 2016 and Journal of Functional Programming [in preparation]**

- ~20 modular programs with ~100,000 configurations.
- 90% of those impose a penalty of 3x or more.
- many configurations impose a 10x penalty
- some configurations cost as much as 100x of the baseline

---

**Is Sound Gradual Typing Dead?**

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, Matthias Felleisen
Northeastern University, Boston, MA
Premature Death?

- Practical evaluations are critical for the design feedback loop.
- They focus our mind and our research efforts.
Premature Death?  Research is when it can fail.

- Practical evaluations are critical for the design feedback loop.
- They focus our mind and our research efforts.
Lessons Learned
Lessons Learned

- the goal
- the nature of the question
- level of granularity
- type inference vs explicit static type
- the role of evaluation
Lessons Learned

- the goal
- the nature of the question
- level of granularity
- type inference vs explicit static type
- the role of evaluation
- do developers care?
Lessons Learned

- The goal
- The nature of the question
- Level of granularity
- Type inference vs explicit static type
- The role of evaluation
- What's in it for you?
- Do developers care?
Lessons Learned

- the goal
- the nature of the question
- level of granularity
- type inference vs explicit static type
- the role of evaluation
- and then, then we go into details
- what's in it for you?
- do developers care?
Why do we add types to untyped languages?
Lessons Learned

Why do we add types to untyped languages?

the goal

Is it about bug finding?

Is it about IDE mechanics?

Is it about execution speed?
Lessons Learned

the goal

Why do we add types to untyped languages?

Is it about bug finding?

Is it about IDE mechanics?

Is it about execution speed?

It is about communicating yourself and others developers in the future.
Lessons Learned

Why do we add types to untyped languages?

Is it about bug finding?

Is it about IDE mechanics?

Is it about execution speed?

the goal

It is about communicating yourself and others developers in the future.

Challenge ~ how to gather evidence for that?
Lessons Learned

What are we investigating?

the nature of the question
Lessons Learned

What are we investigating?

the nature of the question

Is it about λ calculus?

Is it about new languages?

Is it about industrial languages and needs?
Lessons Learned

What are we investigating?

- Is it about λ calculus?
- Is it about new languages?
- Is it about industrial languages and needs?

We use **Racket** for two reasons:
- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must
Lessons Learned

What are we investigating?

Should we aim for soundness?

the nature of the question

Is it about λ calculus?

Is it about new languages?

Is it about industrial languages and needs?

We use Racket for two reasons:
- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must
Lessons Learned

What are we investigating?

Is it about $\lambda$ calculus?

Is it about new languages?

Is it about industrial languages and needs?

Should we aim for soundness?

Absolutely! If academics don’t, nobody will as the numerous designs of hybrid languages in industry show (exception: C#).

We use Racket for two reasons:
- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must
Lessons Learned

**Challenge** ~ can we make it work? What does a compromise look like?

What are we investigating?

Should we aim for soundness?

Absolutely! If academics don’t, nobody will as the numerous designs of hybrid languages in industry show (exception: C#).

We use **Racket** for two reasons:
- it is useful to, and representative of, industrial untyped languages
- but it is academic and we change it if we must

Is it about λ calculus?

Is it about new languages?

Is it about industrial languages and needs?

the nature of the question
What do programmers want when they add types?

level of granularity
What do programmers want when they add types?

Lessons Learned

level of granularity

Expressions?

Classes?

Functions?

Modules?
What do programmers want when they add types?

Typed Racket bets on modules, for two reasons:
- typically small enough for conversion
- large enough to keep cost of contracts low
Lessons Learned

I was wrong.

What do programmers want when they add types?

Typed Racket bets on modules, for two reasons:
- typically small enough for conversion
- large enough to keep cost of contracts low
Lessons Learned

I was wrong.

What do programmers want when they add types?

the “Eli experience” with TypeScript

Typed Racket bets on modules, for two reasons:

▸ typically small enough for conversion
▸ large enough to keep cost of contracts low
What do programmers want when they add types?

Typed Racket bets on modules, for two reasons:
- typically small enough for conversion
- large enough to keep cost of contracts low

I was **wrong**.

the performance evaluation is disastrous (until proven otherwise)

the “Eli experience” with TypeScript

level of granularity

Expressions?

Classes?

Functions?

Modules?
Lessons Learned

Does type inference work for Untyped Languages?

type inference vs explicit static type
Lessons Learned

Does type inference work for Untyped Languages?

- Hindley-Milner?
- Local?
- Set-based?
- Modules?

type inference vs explicit static type
Lessons Learned

Does type inference work for Untyped Languages?

Probably not:
- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- ... and isn't compositional

type inference vs explicit static type

Hindley-Milner?
Local?
Set-based?
Modules?
Lessons Learned

But 1: “run time” inference (see work by Shriram Krishnamurthi and Jeff Foster)

Does type inference work for Untyped Languages?

Probably not:
- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn’t compositional

Hindley-Milner?
Local?
Set-based?
Modules?
type inference vs explicit static type
Lessons Learned

**But 1:** “run time” inference (see work by Shriram Krishnamurthi and Jeff Foster)

**But 2:** IDE tools that assist “conversion”

**Does type inference work for Untyped Languages?**

**Probably not:**
- Type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn’t compositional

**type inference vs explicit static type**

Hindley-Milner?  
Set-based?  
Local?  
Modules?
Lessons Learned

But 3: the syntax system necessitates more than plain local inference

But 2: IDE tools that assist “conversion”

But 1: “run time” inference (see work by Shriram Krishnamurthi and Jeff Foster)

Does type inference work for Untyped Languages?

Probably not:
- type inference needs an explicit type language
- HM inference by itself is extremely brittle
- HM inference for Untyped PLs cannot explain errors
- SBA inference cannot deal with modules
- … and isn’t compositional

type inference vs explicit static type
Lessons Learned

How important is the evaluation process for this field?

the role of evaluation
Lessons Learned

How important is the evaluation process for this field?

Expressiveness
Effectiveness
Usability
Performance

the role of evaluation
Lessons Learned

How important is the evaluation process for this field?

Our “business” is design, evaluation is imperative:
- calculi help with soundness
- existing body of code is critical
- but we are academic so preserve flexibility

the role of evaluation

Expressiveness
Effectiveness
Usability
Performance
Lessons Learned

**Challenge ~** how can academic teams create and maintain a PL?

How important is the evaluation process for this field?

Our “business” is **design**, evaluation is imperative:
- calculi help with soundness
- existing body of code is **critical**
- but we are academic so preserve flexibility

**the role of evaluation**
Lessons Learned

Even academics care in PL, ought to care whether the “developer on the street” will eventually care.

Do developers care?
Lessons Learned

Even academics care in PL ought to care whether the “developer on the street” will eventually care.

do developers care?

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.
Lessons Learned

Even academics care in PL ought to care whether the “developer on the street” will eventually care.

PL has failed to gather data that support soundness and sound design.

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.
Lessons Learned

Even academics care in PL ought to care whether the “developer on the street” will eventually care.

PL fails to make the argument (even) at the “theoretical” level of courses.

PL has failed to gather data that support soundness and sound design.

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.
Lessons Learned

**Challenge ~ how can academic PL improve its teaching?**

Even academics care in PL ought to care whether the “developer on the street” will eventually care.

PL has failed to gather data that support soundness and sound design.

Obviously developers care. People built big systems in Untyped, people discover problems with this approach, and industry is mimicking the incremental/gradual approach to typing.
what's in it for you?
What's in it for you?

The area provides a rich field of challenging problems, ranging from the incredibly theoretical to the highly practical.
Lessons Learned

what's in it for you?

Practical grounding matters.

The area provides a rich field of challenging problems, ranging from the incredibly theoretical to the highly practical.
Lessons Learned

Practical grounding matters.

The area provides a rich field of challenging problems, ranging from the incredibly theoretical to the highly practical.

Take a the long-term view (Wright, Flanagan, Krishnamurthi, Tobin-Hochstadt).

What’s in it for you?
Soft Typists

Robert “Corky” Cartwright, Mike Fagan, Andrew Wright

The MrSpidey Crew

Cormac Flanagan, Shriram Krishnamurthi, Matthew Flatt

Contractors

Robby Findler, Christos Dimoulas Philippe Meunier, Stevie Strickland

Typed Racketeers

Sam Tobin-Hochstadt, Vincent, St-Amour, Asumu Takikawa

Evaluators

Ben Greenman, Max New, Jan Vitek