IS SOUND GRADUAL TYPING DEAD?

Asumu Takikawa    Daniel Feltey    Ben Greenman
Max S. New        Jan Vitek        Matthias Felleisen
<table>
<thead>
<tr>
<th><strong>GRADUAL TYPING THESIS</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. People write untyped code</td>
</tr>
<tr>
<td>2. Static types help maintain software</td>
</tr>
<tr>
<td>3. <em>Sound types</em> can be added <em>incrementally</em></td>
</tr>
<tr>
<td>4. Types <em>respect existing code</em> &amp; the result is <em>runnable</em></td>
</tr>
</tbody>
</table>
SOUND TYPES
The meaning of soundness

**Unsound Typed**

```
#lang typed/racket/unsound

; fact.rkt
(provide fact)
(: fact (-> Integer Integer))
(define (fact n)
  (if (zero? n)
    1
    (* n (sub1 n))))
```

**Untyped**

```
#lang racket

; use.rkt
(require "fact.rkt")
(fact "ill-typed call")
```
The meaning of soundness

Unsound Typed

#lang typed/racket/unsound

; fact.rkt
(provide fact)
(: fact -> Integer -> Integer)
(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n)))))

Untyped

#lang racket

; use.rkt
(require "fact.rkt")

(fact "ill-typed call")

; zero?: contract violation
;   expected: number?
;   given: "ill-typed call"
; [,bt for context]
The meaning of soundness

**Untyped**

```racket
#lang racket
; use.rkt
(require "fact.rkt")
(fact "ill-typed call")
```

**Typed**

```racket
#lang typed/racket
; fact.rkt
(provide fact)
(: fact (-> Integer Integer))
(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n))))
```
The meaning of soundness

Unsound

Typed

#lang typed/racket

; fact.rkt

(provide fact)

(define (fact n)
  (if (zero? n)
      1
      (* n (sub1 n)))))

Untyped

#lang racket

; use.rkt

(require "fact.rkt")

(fact "ill-typed call")

; fact: contract violation
;   expected: Integer
;   given: "ill-typed call"
;   in: the 1st argument of
;       (-> Integer any)
;   contract from: "fact.rkt"
;   blaming: "use.rkt"
RESULTS ARE RUNNABLE
#lang racket/base

(provide (struct-out stream)
  make-stream stream-unfold
  stream-get stream-take)

(struct stream (first rest))

(define (make-stream hd thunk)
  (stream hd thunk))

(define (stream-unfold st)
  (values (stream-first st)
    ((stream-rest st))))

(define (stream-get st i)
  (define-values (hd tl) (stream-unfold st))
  (cond [(= i 0) hd]
    [else (stream-get tl (sub1 i))]))

(define (stream-take st n)
  (cond [(= n 0) '()]
    [else
      (define-values (hd tl) (stream-unfold st))
      (cons hd (stream-take tl (sub1 n))))])

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
    (stream-get primes 99)))

(time (main))
#lang racket/base

(provide (struct-out stream)
    make-stream stream-unfold
    stream-get stream-take)

(struct stream (first rest))

(define (make-stream hd thunk)
    (stream hd thunk))

(define (stream-unfold st)
    (values (stream-first st)
        ((stream-rest st)))))

(define (stream-get st i)
    (define-values (hd tl) (stream-unfold st))
    (cond [(= i 0) hd]
          [else (stream-get tl (sub1 i))])))

(define (stream-take st n)
    (cond [(= n 0) '()]
          [else
            (define-values (hd tl) (stream-unfold st))
            (cons hd (stream-take tl (sub1 n)))]))
#lang typed/racket/base

(provide (struct-out stream)
    make-stream stream-unfold stream-get stream-take)

(struct: stream ([first : Natural]
    [rest : (-> stream)]))

(: make-stream (-> Natural (-> stream) stream)
  (define (make-stream hd thunk)
    (stream hd thunk))

(: stream-unfold (-> stream (values Natural stream) stream)
  (define (stream-unfold st)
    (values (stream-first st)
      ((stream-rest st)))))

(: stream-get (-> stream Natural Natural)
  (define (stream-get st i)
    (define-values (hd tl) (stream-unfold st))
    (cond [(= i 0) hd]
      [else (stream-get tl (sub1 i))])))

(: stream-take (-> stream Natural (Listof Natural))
  (define (stream-take st n)
    (cond [(= n 0) '()]
      [else
        (define-values (hd tl) (stream-unfold st))
        (cons hd (stream-take tl (sub1 n))))]))

#lang racket/base

(require "streams.rkt")

(define (count-from n)
  (make-stream
    n (lambda () (count-from (add1 n))))))

(define (sift n st)
  (values (hd tl) (stream-unfold st))
  (values (if (= 0 (modulo hd n)) (sift n tl))
    [else
      (make-stream (stream-first st) ((stream-rest st))))])

(define (sieve st)
  (define-values (hd tl) (stream-unfold st))
  (make-stream (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
  (printf "The ~a-th prime number is: ~a\n" 100
    (stream-get primes 99)))

(time (main))
#lang typed/racket/base

(provide (struct-out stream)
    make-stream stream-unfold
    stream-get stream-take)

(struct: stream ([first : Natural]
    [rest : (-> stream)]))

(define (make-stream hd thunk)
    stream hd thunk))

(define (stream-unfold st)
    values (stream-first st)
    ((stream-rest st))))

(define (stream-get st i)
    (define-values (hd tl) (stream-unfold st))
    (cond 
        [ (= i 0) hd]
        [else (stream-get tl (sub1 i))]))

(define (stream-take st n)
    (cond 
        [ (= n 0) '()]
        [else
            (define-values (hd tl) (stream-unfold st))
            (cons hd (stream-take tl (sub1 n))))])

#lang racket/base

(require "streams.rkt")

(define (count-from n)
    (make-stream n (lambda () (count-from (add1 n))))))

(define (sift n st)
    values (hd tl) (stream-unfold st))
    (if (= 0 (modulo hd n)) (sift n tl)]
    [else
            (make-stream
                hd (lambda () (sift n tl))))])

(define (sieve st)
    (define-values (hd tl) (stream-unfold st))
    (make-stream hd (lambda () (sieve (sift hd tl)))))

(define primes (sieve (count-from 2)))

(define (main)
    (printf "The ~a-th prime number is: ~a\n" 100
        (stream-get primes 99))

(time (main))
10x slowdown could make the software undeliverable
Anecdotes from users

“The end-product appears to be a 50%-performance hybrid due to boundary contracts” 2x

2.5x “At this point, about one-fifth of my code is now typed. Unfortunately, this version is 2.5 times slower”

“On my machine, it takes *twelve seconds* ... 12,000x ...
... the time taken is 1ms”
“As a practitioner, there are costs associated with using TR, therefore it has to provide equivalent performance improvements to be worthwhile at all.”

— Matthew Butterick
Why is it slow?

Bad programming / isolated incidents?
Bad implementation / design?
Fundamental issue with gradual typing?
To answer, we need an evaluation method
CONTRIBUTIONS OF OUR PAPER

- Evaluation method for language implementors
- Idea for graphically summarizing evaluation results
- Results of evaluating Typed Racket using the method
KEY CONCEPTS
Programmers add types incrementally
so should the evaluation method
Suffixtree benchmark with 6 modules

data  label  lcs  main  structs  ukkonen
Reminder: incremental addition of types
The performance lattice
Paths in lattice are gradual migration paths
Why are the configs useful?

Reveals the cost of boundaries in gradual programs

Shows paths from untyped to typed
Data / Label boundary is costly
When Data / Label have same color, it's more ok
The visualization has some limitations
Which one is better?
Which one is better?

Summarize by proportion of “deliverable” configurations

Version 1

Version 2
A configuration is \textbf{N-deliverable} if its overhead factor $\leq Nx$
1.1-deliverable proportion: 6%
3-deliverable proportion: 9%
5-deliverable proportion: 19%
10-d deliverable proportion: 22%
20-deliverable proportion: 38%

Even at 20x, no paths from untyped to typed
Visualize N-deliverable parameter with a CDF
SUFFIXTREE CDF

Green line is at 3x-deliverable
Shallow slope = bad
Gregor CDF

Steep slope = good
SUMMARY OF APPROACH

- Construct **performance lattices** for benchmarks
- Inspect lattices manually when feasible
- Compare lattices with **N-deliverable CDF**
Measured 12 curated benchmarks on all configs

5 are user-written libraries & programs
5 are educational programs
2 were written for this paper
Ran a total of 75844 configurations

Took 3 months to run
3-deliverable proportions
1.1-deliverable configs over all benchmarks

\[
\frac{283}{75844} \approx 0.4\%\]
3-deliverable configs over all benchmarks

\[
\frac{7992}{75844} \approx 10.5\%
\]
Bottom line: most configs not deliverable

Even with liberal 3x-deliverable criterion
SO, IS THERE HOPE?
Suffixtree improvement

9% to 19% improvement in 3-deliverability
KCFA improvement

25% to 29% improvement in 3-deliverability
Evaluation method helps implementors

Helps measure improvements between versions

Can inspect lattice for bad configs

19575 ms / 75x
Tools for avoiding GT performance pitfalls

Initial steps: contract profiler [St-Amour et al 2015]

Contracts account for 47.35% of running time (286 / 604 ms)
188 ms : build-matrix  (-> Int Int (-> any any any) Array)
88 ms : matrix-multiply-data  (-> Array Array [[...]])
10 ms : make-matrix-multiply  (-> Int Int Int (-> any any any) Array)
Evaluation method helps GT system implementors

Tools for avoiding GT performance pitfalls

Paper & Datasets:
http://www.ccs.neu.edu/racket/pubs/#popl16-tfgnvf
Evaluation method helps GT system implementors
Tools for avoiding GT performance pitfalls

Paper & Datasets:
http://www.ccs.neu.edu/racket/pubs/#popl16-tfgnvf

Thank you!
Other research implementations of gradual typing

Challenge: adapt this method to your chosen sound GT system