Deep and Shallow Types

Thesis Defense

Ben Greenman    2020-12-17
Matthias Felleisen
Amal Ahmed
Jan Vitek
Shriram Krishnamurthi
Fritz Henglein
Sam Tobin-Hochstadt
Deep and Shallow Types

Thesis Defense

Ben Greenman 2020-12-17
If I reproduce somebody's guess in my work ...
me living far away ...
it means that there really is something in it.
Ershov
Great Idea:

mixing typed and untyped code

Gradual Typing

Migratory Typing

Multi-Language Semantics

Hybrid Typing
The Basics

typed code
  more constraints, strong guarantees

untyped code
  more freedom, for better or worse

mixed-typed code
  combine both ... somehow
Q. What happens at the boundaries?
Q. What happens at the boundaries?

Does the type `Num` keep out the letter "A"?

#lang untyped
(f "A")

#lang typed
(define (f (n : Num))
  (+ n 1))
Gradual Typing

Migratory Typing

Multi-Language Semantics

Hybrid Typing

research landscape
research landscape ... over 200 publications
research landscape ... over 200 publications

Q. What happens at the boundaries?
Q. What happens at the boundaries?

research landscape ... over 200 publications

6+ ideas for boundaries
research landscape
research landscape

language landscape ... many implementations
Mixed-Typed Design Space
Q. Does the type `Num` keep out the letter "A"?
Q. Does the type `Num` keep out the letter "A"?

A. Yes!

A. No
Q. Can the type `(-> Num)` detect bad functions?
Q. Can the type `(-> Num)` detect bad functions?

A. Yes

A. No
Q. What happens at the boundaries?

A. Nothing  A. Spot-checks  A. Everything!  A. ...
Q. Why?
Q. Why?  A. Performance!
Q. Why? A. Performance!

Q. Where's the data?
Mixed-Typed Design Space
Lively, but Disorganized!
My Research brings order to the design space

- How to assess type guarantees
- How to measure performance
* How to measure performance  
  (the problem)  
* How to assess type guarantees  
  (solution space)
- How to measure performance (the problem)
- How to assess type guarantees (solution space)

Thesis Preview:
Deep and Shallow types can interoperate
How to measure performance

(the problem)
Typed Racket

- Mature, strong mixed-typed language
- Home of severe performance costs
6 Arrays

by Neil Toronto <ntoronto@racket-lang.org>

**Performance Warning:** Indexing the elements of arrays created in untyped Racket is currently 25-50 times slower than doing the same in Typed Racket, due to the overhead of checking higher-order contracts. We are working on it.

For now, if you need speed, use the typed/racket language.
... More Costs

warning on use trie functions in #lang racket?

johnbclements
to Racket Users

This program constructs a trie containing exactly two keys; each
execution takes 12 seconds. The code below is:

#lang untyped
(require pfdts/trie)

(define t (trie ....))
(time (bind t ....))

12 seconds
... More Costs

```
warning on use trie functions in #lang racket?

johnbclements to Racket Users

This program constructs a trie containing exactly two keys; each
key is a string 18 in the length of (len (loop (digby t 0))).

#lang untyped
(require pfds/trie)
(define t (trie ....))
(time (bind t ....))

12 seconds

#lang typed
(require pfds/trie)
(define t (trie ....))
(time (bind t ....))

1 ms!
```
Typed Racket, Performance

- Clearly, problems exist
Typed Racket, Performance

- Clearly, problems exist

Need a way to measure!
Step 1: Benchmarks
Step 1: Benchmarks

Collected small, useful programs
Step 1: Benchmarks

Collected small, useful programs

Added types, if missing
Step 1: Benchmarks

3.6 jpeg Description

author: Andy Wingo
source: github.com/wingo/racket-jpeg
dependencies: math/array (typed) and rnrs/bytevectors-6 (untyped)

Parses a bytestream of JPEG data to an internal representation, then serializes the result.

```
0. bit-ports.rkt  2. huffman.rkt  4. main.rkt  6. ../../../base/untyped.rkt
1. exif.rkt  3. jfif.rkt  5. ../../../base/math/array.rkt
```
Step 2: How to Measure
Step 2: How to Measure

What to measure = all configurations

3 components  ➔  8 configurations
Step 2: How to Measure

What to measure = all configurations

6 components ➤

64 configurations
Step 2: How to Measure

What to measure = all configurations

Q. How to study?  Q. How to scale?
Step 2: How to Measure

What to measure = all configurations

Q. How to study? Q. How to scale?

A. Focus on the programmer ...
Step 2: How to Measure

A. Count D-deliverable configs
Step 2: How to Measure

A. Count $D$-deliverable configs

If $D=4$, then count configs with at most $4x$ overhead

= 50%
Step 2: How to Measure

A. Count D-deliverable configs
Step 2: How to Measure

A. Count $D$-deliverable configs

$D$-deliverable $\sim$ Bernoulli random variable

linear-size sampling works
Step 3: Summarize with a Picture
Step 3: Summarize with a Picture
Step 3: Summarize with a Picture

quadU

10 samples of 140 configs

D = 1x 2x 10x 20x
Performance Method
Performance Method

1. collect mixed-typed benchmarks

2. count D-deliverable configs (or sample)

3. plot results
Applications:

Typed Racket

Reticulated Python
Typed Racket  some results from our 21 benchmarks

- jpeg
- suffixtree
- take5
- synth

2x  20x  2x  20x
Typed Racket some results from our 21 benchmarks
Reticulated Python different benchmarks

spectralnorm

2x  20x

pystone

2x  20x

chaos

2x  20x

go

2x  20x
Reticulated Python different benchmarks

spectralnorm 2x 20x

pystone 2x 20x

Not so bad
Q. Is Reticulated better, overall?
Natural Transient

type soundness

gradual guarantee

blame theorem
<table>
<thead>
<tr>
<th>Natural</th>
<th>Transient</th>
</tr>
</thead>
<tbody>
<tr>
<td>type soundness</td>
<td>✔️</td>
</tr>
<tr>
<td>gradual guarantee</td>
<td>✔️</td>
</tr>
<tr>
<td>blame theorem</td>
<td>✔️</td>
</tr>
</tbody>
</table>
Q. Can the type \((\rightarrow \text{Num})\) detect bad functions?
Q. Can the type $\rightarrow\text{Num}$ detect bad functions?

A. Natural = Yes

A. Transient = No
expects Num, Str ...
#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
  (+ 1 acc))

#lang typed
(: t-fold-file
  (-> Path Num
      (-> Num Str Num)
      Num))

(define t-fold-file u-fold-file)
#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
  (+ 1 acc))

#lang typed
(: t-fold-file
  (-> Path Num
      (-> Num Str Num)
      Num))
(define t-fold-file u-fold-file)

#lang untyped
(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)

#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
  (+ 1 acc))

#lang typed
(: t-fold-file
  (-> Path Num
      (-> Num Str Num) Num))
(define t-fold-file u-fold-file)

Q. Do types protect the callback?
A. Transient = No
A. Natural = Yes

#lang untyped
(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)

- But Natural and Transient disagree
Need to measure type guarantees

- But Natural and Transient disagree
How to assess type guarantees
Co-Natural  Forgetful  Erasure
Natural    Amnesic    Transient
| Co-Natural | Natural | Amnesic | Forgetful | Transient | Erasure |

0. before = sound vs. unsound
0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries
Complete Monitoring vs. Type Soundness
Complete Monitoring  vs. Type Soundness

Q. Do types protect the callback?

TS  =/> Yes

CM  => Yes

#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
  (+ 1 acc))

#lang typed
(: t-fold-file
  (-> Path Num
      (-> Num Str Num)
      Num))

(define t-fold-file u-fold-file)

TS  nothing

CM  Num , Str

#lang untyped
(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)

CM  Num , Str
Shallow types are sound.

Deep types protect untyped code, too.
0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries
0. before = sound vs. unsound
1. Complete Monitoring ~ types guard all boundaries
2. Blame Soundness ~ errors are accurate
0. before = sound vs. unsound

1. Complete Monitoring ~ types guard all boundaries

2. Blame Soundness ~ errors are accurate

3. Blame Completeness ~ errors are exhaustive
0. before = sound vs. unsound

1. Complete Monitoring  ~  types guard all boundaries

2. Blame Soundness  ~  errors are accurate

3. Blame Completeness  ~  errors are exhaustive

4. Error Preorder  ~  head-to-head test
type soundness

complete monitoring

blame soundness

blame completeness

error preorder
<table>
<thead>
<tr>
<th></th>
<th>Natural</th>
<th>C</th>
<th>F</th>
<th>Transient</th>
<th>A</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>type soundness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>y</td>
<td>✓</td>
</tr>
<tr>
<td>complete monitoring</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>blame soundness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>h</td>
<td>✓</td>
</tr>
<tr>
<td>blame completeness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>error preorder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Natural</td>
<td>C</td>
<td>F</td>
<td>Transient</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>=</td>
<td>&lt;</td>
</tr>
</tbody>
</table>
My Research brings order to the design space

- How to assess type guarantees
- How to measure performance
Goal: mixed-typed code with strong guarantees

Problem: high performance overhead
Goal: mixed-typed code with strong guarantees

Problem: high performance overhead

Q. What to do?
Goal: mixed-typed code with strong guarantees
Problem: high performance overhead

Q. What to do?

a. build a new language
a. build a new compiler
a. improve the current compiler
Goal: mixed-typed code with strong guarantees
Problem: high performance overhead

Q. What to do?

a. build a new language
a. build a new compiler
✓a. improve the current compiler
Goal: mixed-typed code with strong guarantees
Problem: high performance overhead

Q. What to do?

a. build a new language

a. build a new compiler

✓ a. improve the current compiler

- re-use type system
- add new semantics
Deep and Shallow types can interoperate.
preserving their formal properties

Programmers can use these types to:
– strengthen Shallow guarantees
– avoid unimportant Deep errors
– lower runtime costs
Unpublished Results
Plan:
Plan:
- combine Natural + Transient
Plan:
- combine Natural + Transient
- extend TR

1. new model

2. new language
**Model**  Deep + Shallow + Untyped

\[
\begin{align*}
\mathbf{s} &= x \mid i \mid (s, s) \mid \lambda x. \mathbf{s} \mid \lambda x: \mathbf{T}. \mathbf{s} \mid \\
&\quad \ldots
\end{align*}
\]

\[
\begin{align*}
\mathbf{T} &= \ldots
\end{align*}
\]

\[
\begin{align*}
\mathbf{L} &= \ldots
\end{align*}
\]
**Model**  Deep + Shallow + Untyped

\[ s = x \mid i \mid (s, s) \mid \lambda x. s \mid \lambda x:T. s \mid \text{unop } s \mid \text{binop } s \mid \text{app } s \mid \ldots \]

\[ T = \ldots. \]

\[ L = \ldots. \]
Model  Deep + Shallow + Untyped

\[ s = x \mid i \mid (s, s) \mid \lambda x. s \mid \lambda x: T. s \mid \text{unop } s \mid \text{binop } s \ s \mid \text{app } s \ s \mid \text{module } L \ s \]

\[ T = \ldots. \]

\[ L = \ldots. \]
Model Deep + Shallow + Untyped

\[ s = x \mid i \mid (s, s) \mid \lambda x. s \mid \lambda x:T. s \mid \text{unop } s \mid \text{binop } s s \mid \text{app } s s \mid \text{module } L s \]

\[ T = \text{Nat} \mid \text{Int} \mid T \times T \mid T \to T \]

\[ L = \ldots \]
Model Deep + Shallow + Untyped

\[ s = x \mid i \mid (s, s) \mid \lambda x. s \mid \lambda x:T. s \mid \text{unop } s \mid \text{binop } s s \mid \text{app } s s \mid \text{module } L s \]

\[ T = \text{Nat} \mid \text{Int} \mid T \times T \mid T \to T \]

\[ L = \text{Deep} \mid \text{Shallow} \mid \text{Untyped} \]
Model  Deep + Shallow + Untyped

\[ s = x \mid i \mid (s, s) \mid \lambda x. s \mid \lambda x: T. s \mid \text{unop } s \mid \text{binop } s \, s \mid \text{app } s \, s \mid \text{module } L \, s \]

\[ T = \text{Nat} \mid \text{Int} \mid T \times T \mid T \rightarrow T \]

\[ L = \text{Deep} \mid \text{Shallow} \mid \text{Untyped} \]
Model  Boundaries

- Deep = wrap, or fully check
- Shallow = spot-check inputs
- Untyped

Diagram:
- Deep \(\rightarrow\) wrap \(\rightarrow\) Shallow
- Shallow \(\rightarrow\) scan
- Untyped \(\rightarrow\) noop
1. new model
1. new model

- **Type Soundness**
  - types predict outcomes

- **Complete Monitoring**
  - Deep types predict behaviors
2. new language
Typed Racket Compiler

Expand ➔ Typecheck ➔ Generate Contracts ➔ Optimize

2. new language
Shallow Racket

Expand ➤ Typecheck ➤ Generate Contracts ➤ Optimize

2. new language
Shallow Racket

- Expand
- Typecheck
- Generate Contracts
- Optimize

- Insert Checks

2. new language
Shallow Racket

Expand  ➔ Typecheck  ➔ Generate Contracts  ➔ Optimize

Insert Checks

2. new language
Insert Checks  types to shapes

design choice: enforce full type constructors
**Insert Checks** types to shapes

design choice: enforce full type constructors

<table>
<thead>
<tr>
<th>Type</th>
<th>shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>number?</td>
</tr>
<tr>
<td>(Listof Num)</td>
<td>list?</td>
</tr>
<tr>
<td>(U Num Sym)</td>
<td>(or number? symbol?)</td>
</tr>
<tr>
<td>(-&gt; Num Num)</td>
<td>(and procedure?</td>
</tr>
<tr>
<td></td>
<td>(arity-includes 1))</td>
</tr>
</tbody>
</table>
Optimize

- apply
- box
- dead-code
- extflonum
- fixnum
- float-complex
- float
- list
- number
- pair
- sequence
- string
- struct
- vector
Optimize

- apply
- box
- dead-code
- extflonum
- fixnum
- float-complex
- float
- list
- number
- pair
- sequence
- string
- struct
- vector
- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs
Shallow to Deep = stronger guarantees
Shallow to Deep = stronger guarantees

#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
 (+ 1 acc))

#lang shallow
(: t-fold-file
 (-> Path Num
  (-> Num Str Num) Num))
(define t-fold-file u-fold-file)

#lang untyped
(define (u-fold-file path acc f)
 ; read str from path
 ... (f str acc) ...)

nothing
Shallow to Deep = stronger guarantees

#lang untyped
(t-fold-file "file.txt" 0 count)
(define (count acc str)
  (+ 1 acc))

#lang deep
(: t-fold-file
  (-> Path Num
      (-> Num Str Num)
      Num))
(define t-fold-file u-fold-file)

#lang untyped
(define (u-fold-file path acc f)
  ; read str from path
  ... (f str acc) ...)

Num, Str
Shallow to Deep = stronger guarantees

Deep protects all boundaries
Deep to Shallow = fewer errors
Deep to Shallow = fewer errors

[racket] error: Attempted to use a higher-order value passed as `Any` in untyped code:

68 views

mailoo

to us...@racket-lang.org

Hello,

I'm new to racket, and even more with typed/racket.

I play a little with the "Any" type (due to 'dynamic-require' which
Deep to Shallow = fewer errors

#lang deep
(: b Any)
(define b (box 42))

#lang untyped
(set-box! b 0)
Deep to Shallow = fewer errors

>Error: attempted to use higher-order value passed as Any
Deep to Shallow = fewer errors

Error: attempted to use higher-order value passed as Any

OK
Deep to Shallow = fewer errors

Shallow can run almost all type-correct code

Error: attempted to use higher-order value passed as Any

#lang shallow
(: b Any)
(define b (box 42))

#lang untyped
(set-box! b 0)

OK

#lang shallow
(: b Any)
(define b (box 42))
Better Performance

~ 2 sec. Untyped baseline
Better Performance

#lang untyped
....
#lang untyped
....
~ 2 sec. Untyped baseline

#lang untyped
....
#lang deep
....
~ 13 sec. Mixed : Shallow wins

#lang untyped
....
#lang shallow
....
~ 4 sec.
Better Performance

Untyped baseline

~ 2 sec.

Mixed: Shallow wins

~ 13 sec.

~ 4 sec.

Typed: Deep wins

< 2 sec.

~ 5 sec.
Better Performance

quadU

Deep + Shallow = maximize D-deliverable cfgs.
Better Performance

jpeg 2x 20x

take5 2x 20x

suffixtree 2x 20x

synth 2x 20x

quadU 2x 20x

sieve 2x 20x
New Migration Plan

1. Deep, until slow
2. Shallow, to fix boundaries
3. Deep, or mix, at end
New Migration Plan

What % of paths are D-deliverable at each step?
New Migration Plan

% of 3-deliverable paths
New Migration Plan

% of 3-deliverable paths

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Deep or Shallow</th>
<th>Deep and Shallow</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>suffixt</td>
<td>0%</td>
<td>12%</td>
</tr>
<tr>
<td>take5</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>sieve</td>
<td>0%</td>
<td>100%</td>
</tr>
<tr>
<td>fsmoo</td>
<td>0%</td>
<td>50%</td>
</tr>
<tr>
<td>dungeon</td>
<td>0%</td>
<td>67%</td>
</tr>
</tbody>
</table>
Better Together

How many configs do best with a mix?
# Better Together

How many configs do best with a mix?

| Benchmark | $D+S \geq D|S$ |
|-----------|-------------|
| fsm       | 37%         |
| morsecode | 25%         |
| jpeg      | 37%         |
| kcfa      | 55%         |
| zombie    | 6%          |
| zordoz    | 46%         |
Thesis Statement

Deep and Shallow types can interoperate. preserving their formal properties

Programmers can use these types to:
- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs
Deep and Shallow types can interoperate.

✓ preserving their formal properties

Programmers can use these types to:

- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs
Deep and Shallow types can interoperate.

- preserving their formal properties

Programmers can use these types to:

- strengthen Shallow guarantees
- avoid unimportant Deep errors
- lower runtime costs
Contributions

1. performance analysis method
2. design analysis method
3. scaled-up Transient
4. Deep + Shallow
<table>
<thead>
<tr>
<th></th>
<th>Natural</th>
<th>C</th>
<th>F</th>
<th>Transient</th>
<th>A</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>type soundness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>y</td>
<td>✓</td>
</tr>
<tr>
<td>complete monitoring</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>blame soundness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>h</td>
<td>✓</td>
</tr>
<tr>
<td>blame completeness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>error preorder</td>
<td>Natural</td>
<td>C</td>
<td>F</td>
<td>Transient</td>
<td>A</td>
<td>E</td>
</tr>
</tbody>
</table>
Optimization

- apply
- box
- dead-code
- extflonum
- fixnum
- float-complex
- float
- list
- number
- pair
- sequence
- string
- struct
- vector
### Better Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Worst Deep</th>
<th>Worst Shallow</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg</td>
<td>23x</td>
<td>2x</td>
</tr>
<tr>
<td>suffixtree</td>
<td>31x</td>
<td>6x</td>
</tr>
<tr>
<td>take5</td>
<td>32x</td>
<td>3x</td>
</tr>
<tr>
<td>synth</td>
<td>49x</td>
<td>4x</td>
</tr>
<tr>
<td>quadU</td>
<td>60x</td>
<td>8x</td>
</tr>
<tr>
<td>sieve</td>
<td>10x</td>
<td>2x</td>
</tr>
</tbody>
</table>
## Transient Blame

Quite Bad!

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Shallow Blame</th>
<th>Worst</th>
<th>Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>jpeg</td>
<td>46x</td>
<td>23x</td>
<td></td>
</tr>
<tr>
<td>suffixtree</td>
<td>&gt;189x</td>
<td>31x</td>
<td></td>
</tr>
<tr>
<td>take5</td>
<td>51x</td>
<td>32x</td>
<td></td>
</tr>
<tr>
<td>synth</td>
<td>&gt;1440x</td>
<td>49x</td>
<td></td>
</tr>
<tr>
<td>quadU</td>
<td>560x</td>
<td>60x</td>
<td></td>
</tr>
<tr>
<td>sieve</td>
<td>out of memory</td>
<td></td>
<td>10x</td>
</tr>
</tbody>
</table>
Shallow cannot run 1/2

problem: inst changes shape

```scheme
#lang deep
(require/typed racket/base
  (cdr (All (A) A)))

(define fake-str : String
  (inst cdr String))

(string-length fake-str)
```
Shallow cannot run 2/2

problem: occurrence-type side effect

#lang deep

(require/typed racket/base
  (values (-> Any Any : String)))

(define x : Any 0)

(define fake-str : String
  (if (values x)
      x
      (error 'unreachable)))
- conditionally weaken Deep -- Shallow, if escapes
- noop Deep -- Shallow, if S can wrap

**Model**  Other Ideas

- Deep
- Shallow
- Untyped

wrap
wrap
wrap
wrap
noop
scan
Deep to Shallow = simpler behavior

#lang untyped
(index-of '(a b) 'a)
Deep to Shallow = simpler behavior

#lang deep
(: index-of
  (-> (Listof T) T (Maybe Num)))

(index-of '(a b) 'a)

Untyped 0  Deep #f  Shallow 0
Deep to Shallow = simpler behavior

#lang shallow
(: index-of
  (-> (Listof T) T (Maybe Num)))

(index-of '(a b) 'a)
Deep to Shallow = simpler behavior

#lang shallow
(: index-of
  (-> (Listof T) T (Maybe Num)))

No wrappers = fewer surprises

Untyped 0  Deep #f  Shallow 0