A Transient Semantics for Typed Racket

Ben Greenman
Lukas Lazarek
Christos Dimoulas
Matthias Felleisen
2022-04-12

<Programming> 6.2
Context = Gradual Typing

High-level goal: mix typed and untyped code
Context = Gradual Typing

High-level goal: mix typed and untyped code

Typed Function

```
function add1(n : Num)
    n + 1
```

Untyped Caller

```
add1("hola")
```
**Context = Gradual Typing**

**High-level goal**: mix typed and untyped code

**Typed Function**

```
function add1(n : Num)
    n + 1
```

**Untyped Caller**

```
add1("hola")
```

**Central question**: what should types mean at run-time?
What Should Types Mean?

Three leading strategies:
What Should Types Mean?

Three leading strategies:

**Guarded**
Types enforce behaviors

**Transient**
Types enforce top-level shapes

**Optional**
Types enforce nothing
Example

Typed Function

function add1(n : Num) {
  n + 1
}

Untyped Caller

add1("hola")
Example

Typed Function

```
function add1(n : Num)
    n + 1
```

Untyped Caller

```
add1("hola")
```

**Guarded** and **Transient**

```
Error: expected Num
```

**Optional**

```
"hola" + 1
```
Example 2

Untyped Array

```
arr = ["A", 3]
```

Typed Client

```
nums : Array(Num) = arr
ums[0]
```
Example 2

**Guarded** and **Transient** agree, but for different reasons ...

Untyped Array

```
arr = ["A", 3]
```

Typed Client

```
nums : Array(Num) = arr
nums[0]
```

**Guarded** and **Transient**

```
Error: expected Array(Num)
```

Optional

```
"A"
```
Example 2+

Guarded and Transient agree, but for different reasons ...

... and they disagree for an untyped client
Example 2+

Guarded and Transient agree, but for different reasons ...

... and they disagree for an untyped client

Untyped Array

```javascript
arr = ["A", 3]
```

Typed Interface

```javascript
nums : Array(Num) = arr
```

Unyped Client

```javascript
nums[0]
```
Example 2+

Guarded and Transient agree, but for different reasons ...

... and they disagree for an untyped client

Untyped Array

```
arr = ["A", 3]
```

Typed Interface

```
ums : Array(Num) = arr
```

Unyped Client

```
ums[0]
```

Guarded

```
Error: expected Array(Num)
```

Transient and Optional

```
"A"
```
Guarded
Types enforce behaviors

Transient
Types enforce top-level shapes

Optional
Types enforce nothing
Typed Racket has **Guarded** types ... and a big problem

- **Guarded**: Types enforce behaviors
- **Transient**: Types enforce top-level shapes
- **Optional**: Types enforce nothing
Guarded Types are Expensive!
Guarded Types are Expensive!

Typed Racket

- **Strong types**: Type soundness
  + Complete monitoring
- **High overheads** are common on the GTP Benchmarks
- **Worst cases**: 25x, 1400x
Guarded Types are Expensive!

Typed Racket

- **Strong types**: Type sound + Complete monitoring
- **High overheads** are common on the GTP Benchmarks
- **Worst cases**: 25x, 1400x

Q. Is Sound Gradual Typing Dead?
Guarded gradual types are too slow

What to do?
Guarded gradual types are too slow

**What to do?**

1. **Improve the compiler**
   Collapsible Contracts  [OOPSLA '18]
Guarded gradual types are too slow

**What to do?**

1. **Improve the compiler**
   Collapsible Contracts  [OOPSLA '18]

2. **Remove checks statically**
   Corpse Reviver  [POPL '21]
Guarded gradual types are too slow

**What to do?**

1. **Improve the compiler**
   Collapsible Contracts [OOPSLA'18]

2. **Remove checks statically**
   Corpse Reviver [POPL'21]

3. **Build a new compiler**
   Pycket [OOPSLA'17]
Guarded gradual types are too slow

What to do?

1. **Improve the compiler**
   Collapsible Contracts  [OOPSLA '18]

2. **Remove checks statically**
   Corpse Reviver  [POPL '21]

3. **Build a new compiler**
   Pycket  [OOPSLA '17]

4. **Use weaker types**
   Today!
Hope to **reduce costs across the board** without changing the surface language

- Same code, types, and type checker
- Different run-time behavior

4. **Use weaker types**
   Today!
The Inspiration: Reticulated Python

**Transient** semantics ~ enforce types with tag checks

**No** contract wrappers
The Inspiration: Reticulated Python

**Transient** semantics ~ enforce types with tag checks

**No** contract wrappers

Performance is not bad! [POPL '17]

<6x overhead for Transient
<18x overhead for Transient with blame
Research Questions

RQ0. How to add *transient types* to Typed Racket?
Research Questions

RQ0. How to add *transient types* to Typed Racket?

RQ1. Can Transient *scale to a rich type system*?
RQ2. Can we *adapt an existing compiler* to do so?
Research Questions

Implications for other gradual languages, especially Optional ones that wish to strengthen their types

RQ1. Can Transient scale to a rich type system?
RQ2. Can we adapt an existing compiler to do so?
Two Type Systems
Two Type Systems

\[
T := \text{Dynamic} \\
\text{Int} \\
\text{Ref } T \\
T \rightarrow T \\
\text{Class } \{ T \ldots \} \\
\ldots \text{ a few more}
\]
Two Type Systems

\[
T := \text{Dynamic}
\]
\[
\text{Int}
\]
\[
\text{Ref } T
\]
\[
T \rightarrow T
\]
\[
\text{Class } \{ T \ldots \}
\]
\[
\text{.... a few more}
\]

\[
T := \text{Any (the top type)}
\]
Two Type Systems

\[ T := \text{Dynamic} \]
\[ \text{Int} \]
\[ \text{Ref} \ T \]
\[ T \rightarrow T \]
\[ \text{Class} \{ \ T \ldots \} \]
\[ \ldots \text{a few more} \]

\[ T := \text{Any (the top type)} \]
\[ \text{Integer} \]
\[ \text{Natural} \]
Two Type Systems

T := Dynamic
    Int
    Ref T
    T -> T
    Class { T ... }
    .... a few more

T := Any (the top type)
    Integer
    Natural
    (Vectorof T)
    (Vector T ...)

Any
Two Type Systems

T := Dynamic
   Int
   Ref T
   T -> T
   Class { T ... }
   .... a few more

T := Any (the top type)
   Integer
   Natural
   (Vectorof T)
   (Vector T ...)
   (-> T ... (Values T ...))
Two Type Systems

T := Dynamic
    Int
    Ref T
    T -> T
    Class { T ... }
    .... a few more

T := Any (the top type)
    Integer
    Natural
    (Vectorof T)
    (Vector T ...)
    (-> T ... (Values T ...))
    (Class T ...)

(Any)
Two Type Systems

\[ T := \text{Dynamic} \]
\[ \quad \text{Int} \]
\[ \quad \text{Ref } T \]
\[ \quad T \rightarrow T \]
\[ \quad \text{Class } \{ T \ldots \} \]
\[ \quad \ldots \text{ a few more} \]

\[ T := \text{Any (the top type)} \]
\[ \quad \text{Integer} \]
\[ \quad \text{Natural} \]
\[ \quad \text{(Vectorof } T) \]
\[ \quad \text{(Vector } T \ldots) \]
\[ \quad \text{(-} \rightarrow T \ldots \text{ (Values } T \ldots)) \]
\[ \quad \text{(Class } T \ldots) \]
\[ \quad \text{(All } X T) \]
\[ \quad \text{(Union } T \ldots) \]
\[ \quad \text{(Rec } X T) \]
Two Type Systems

T := Dynamic
    Int
    Ref T
    T -> T
    Class { T ... }
    .... a few more

T := Any (the top type)
   Integer
   Natural
   (Vectorof T)
   (Vector T ...)
   (-> T ... (Values T ...))
   (Class T ...)
   (All X T)
   (Union T ...)
   (Rec X T)
   .... many more
Two Compilers
Two Compilers

Typecheck + Elaborate

Python
Two Compilers

Typecheck + Elaborate

Python

Expand

Typecheck

Guard Boundaries

Optimize

Racket
Two Compilers

Python

Typecheck + Elaborate

(replace only the guard pass)

Racket

Expand

Typecheck

Guard Boundaries

Insert Transient Checks

Optimize

Racket
Challenges
Challenges

Enforcing types

- All
- Union
- Rec
Challenges

- Enforcing types
  - All
  - Union
  - Rec

+ Generalize tag checks to **shape checks** +
Challenges

Enforcing types
- All
- Union
- Rec

+ Generalize tag checks to *shape checks*

Optimizing typed code
Challenges

Enforcing types
- All
- Union
- Rec
- Generalize tag checks to **shape checks**

Optimizing typed code
- Trust only shapes
Challenges

Enforcing types
- All
- Union
- Rec

+ Generalize tag checks to **shape checks** +

Navigating expanded code

Optimizing typed code

+ Trust only shapes +
Challenges

Enforcing types
- All
- Union
- Rec

+ Generalize tag checks to \texttt{shape checks} +

Navigating expanded code
- + Typechecker must leave \texttt{evidence} +

Optimizing typed code
- + Trust only shapes +
Challenges

Enforcing types
- All
- Union
- Rec

+ Generalize tag checks to **shape checks** +

Navigating expanded code
+ Typechecker must leave **evidence** +

Optimizing typed code
+ Trust only shapes +

Minimizing costs
Challenges

Enforcing types
- All
- Union
- Rec
+ Generalize tag checks to *shape checks* +

Navigating expanded code
+ Typechecker must leave *evidence* +

Optimizing typed code
+ Trust only shapes +

Minimizing costs
+ E.g. reduce codegen +
Challenges

Enforcing types
- All
- Union
- Rec

+ Generalize tag checks to \textit{shape checks} +

Navigating expanded code

+ Typechecker must leave \textit{evidence} +

Optimizing typed code

+ Trust only shapes +

Minimizing costs

+ E.g. reduce codegen +
Navigating expanded code

+ Typechecker must leave **evidence** +
Navigating expanded code

(for/sum ([byte (open-input-file "my.txt")]) byte)

+ Typechecker must leave evidence +
Navigating expanded code

+ Typechecker must leave evidence +

(for/sum ([byte (open-input-file "my.txt")])
  byte)

(define seq (make-seq (open-input-file "my.txt")))
(define (for-loop result pos)
  (if (not (seq.use-pos? pos))
      result
      (let ([byte (seq.get-val pos)])
        (for-loop (if (or (not seq.use-val?)
                         (seq.use-val? byte))
                  (+ result byte)
                  result)
        (seq.next-pos pos))))
(for-loop 0 seq.init)
Navigating expanded code

+ Typechecker must leave evidence +

\[
\text{(for/sum ([byte (open-input-file "my.txt")]) byte)}
\]

\[
\text{(define seq (make-seq (open-input-file "my.txt")))}
\]

\[
\text{(define (for-loop result pos)}
\]

\[
\text{(if (not (seq.use-pos? pos)) result)}
\]

\[
\text{(let ([byte (seq.get-val pos)])}
\]

\[
\text{(for-loop (if (or (not seq.use-val?) (seq.use-val? byte))}
\]

\[
\text{(seq.use-val? byte))}
\]

\[
\text{(+ result byte)}
\]

\[
\text{result)}
\]

\[
\text{(seq.next-pos pos)))))}
\]

\[
\text{(for-loop 0 seq.init)}
\]

Don't want to check every function call!
How's performance?

Does it match the worst cases for Reticulated?

<6x overhead for Transient
<18x overhead for Transient with blame
Worst Case Overhead vs. Untyped
# Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>Transient</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>gregor</td>
</tr>
<tr>
<td>morsecode</td>
<td>jpeg</td>
</tr>
<tr>
<td>sieve</td>
<td>lnm</td>
</tr>
<tr>
<td>snake</td>
<td>mbta</td>
</tr>
<tr>
<td>suffixtree</td>
<td>quadT</td>
</tr>
<tr>
<td>tetris</td>
<td>quadU</td>
</tr>
<tr>
<td>acquire</td>
<td>synth</td>
</tr>
<tr>
<td>dungeon</td>
<td>take5</td>
</tr>
<tr>
<td>forth</td>
<td>zombie</td>
</tr>
<tr>
<td>fsm</td>
<td>zordoz</td>
</tr>
<tr>
<td>fsmoo</td>
<td></td>
</tr>
</tbody>
</table>
## Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Transient</th>
<th>T+Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>1x</td>
<td>&gt;540x</td>
<td>gregor</td>
</tr>
<tr>
<td>morsecode</td>
<td>3x</td>
<td>&gt;250x</td>
<td>jpeg</td>
</tr>
<tr>
<td>sieve</td>
<td>4x</td>
<td>&gt;220x</td>
<td>lnm</td>
</tr>
<tr>
<td>snake</td>
<td>8x</td>
<td>&gt;1000x</td>
<td>mbta</td>
</tr>
<tr>
<td>suffixtree</td>
<td>6x</td>
<td>&gt;190x</td>
<td>quadT</td>
</tr>
<tr>
<td>tetris</td>
<td>10x</td>
<td>&gt;720x</td>
<td>quadU</td>
</tr>
<tr>
<td>acquire</td>
<td>1x</td>
<td>34x</td>
<td>synth</td>
</tr>
<tr>
<td>dungeon</td>
<td>5x</td>
<td>75x</td>
<td>take5</td>
</tr>
<tr>
<td>forth</td>
<td>6x</td>
<td>48x</td>
<td>zombie</td>
</tr>
<tr>
<td>fsm</td>
<td>2x</td>
<td>230x</td>
<td>zordoz</td>
</tr>
<tr>
<td>fsmoo</td>
<td>4x</td>
<td>100x</td>
<td></td>
</tr>
</tbody>
</table>
## Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Transient</th>
<th>T+Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>1x</td>
<td>&gt;540x</td>
<td>gregor</td>
</tr>
<tr>
<td>morsecode</td>
<td>3x</td>
<td>&gt;250x</td>
<td>jpeg</td>
</tr>
<tr>
<td>sieve</td>
<td>4x</td>
<td>&gt;100x</td>
<td></td>
</tr>
<tr>
<td>snake</td>
<td>8x</td>
<td>&gt;100x</td>
<td></td>
</tr>
<tr>
<td>suffixtree</td>
<td>6x</td>
<td>&gt;220x</td>
<td></td>
</tr>
<tr>
<td>tetriss</td>
<td>10x</td>
<td>&gt;1200x</td>
<td></td>
</tr>
<tr>
<td>acquire</td>
<td>1x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dungeon</td>
<td>5x</td>
<td>75x</td>
<td>take5</td>
</tr>
<tr>
<td>forth</td>
<td>6x</td>
<td>48x</td>
<td>zombie</td>
</tr>
<tr>
<td>fsm</td>
<td>2x</td>
<td>230x</td>
<td>zordoz</td>
</tr>
<tr>
<td>fsmoo</td>
<td>4x</td>
<td>100x</td>
<td></td>
</tr>
</tbody>
</table>

*Transient alone is not so bad*

*T+Blame gets expensive*
Blame: The Idea
Blame: The Idea

$$\lambda x. \ "B"$$

$$f : \text{Num} \rightarrow \text{Num}$$

$$f(2)$$
When a typed/untyped interaction goes wrong, blame shows where to start debugging
Blame: The Idea

When a typed/untyped interaction goes wrong, blame shows where to start debugging.

Guarded wrappers can attach precise blame info to values.
Blame: The Idea

When a typed/untyped interaction goes wrong, blame shows where to start debugging.

Guarded wrappers can attach precise blame info to values

Transient has no wrappers, but keeps a global map on the side
Blame: The Idea

\[ \lambda x. \text{"B"} \]

\[ f : \text{Num} \to \text{Num} \]

\[ f(2) \]

When a **typed/untyped** interaction goes wrong, **blame** shows where to start debugging.

**Guarded** wrappers can attach precise blame info to values.

**Transient** has no wrappers, but keeps a **global map on the side**.

... a large map gets expensive.
### Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Transient</th>
<th>T+Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>1x</td>
<td>&gt;540x</td>
<td></td>
</tr>
<tr>
<td>morsecode</td>
<td>3x</td>
<td>&gt;250x</td>
<td></td>
</tr>
<tr>
<td>sieve</td>
<td>4x</td>
<td>&gt;220x</td>
<td></td>
</tr>
<tr>
<td>snake</td>
<td>8x</td>
<td>&gt;1000x</td>
<td></td>
</tr>
<tr>
<td>suffixtree</td>
<td>6x</td>
<td>&gt;190x</td>
<td></td>
</tr>
<tr>
<td>tetris</td>
<td>10x</td>
<td>&gt;720x</td>
<td></td>
</tr>
<tr>
<td>acquire</td>
<td>1x</td>
<td>34x</td>
<td></td>
</tr>
<tr>
<td>dungeon</td>
<td>5x</td>
<td>75x</td>
<td></td>
</tr>
<tr>
<td>forth</td>
<td>6x</td>
<td>48x</td>
<td></td>
</tr>
<tr>
<td>fsm</td>
<td>2x</td>
<td>230x</td>
<td></td>
</tr>
<tr>
<td>fsmoo</td>
<td>4x</td>
<td>100x</td>
<td></td>
</tr>
<tr>
<td>gregor</td>
<td>2x</td>
<td>23x</td>
<td></td>
</tr>
<tr>
<td>jpeg</td>
<td>2x</td>
<td>38x</td>
<td></td>
</tr>
<tr>
<td>lnm</td>
<td>1x</td>
<td>29x</td>
<td></td>
</tr>
<tr>
<td>mbta</td>
<td>2x</td>
<td>37x</td>
<td></td>
</tr>
<tr>
<td>quadT</td>
<td>7x</td>
<td>34x</td>
<td></td>
</tr>
<tr>
<td>quadU</td>
<td>8x</td>
<td>320x</td>
<td></td>
</tr>
<tr>
<td>synth</td>
<td>4x</td>
<td>220x</td>
<td></td>
</tr>
<tr>
<td>take5</td>
<td>3x</td>
<td>33x</td>
<td></td>
</tr>
<tr>
<td>zombie</td>
<td>31x</td>
<td>560x</td>
<td></td>
</tr>
<tr>
<td>zordoz</td>
<td>3x</td>
<td>220x</td>
<td></td>
</tr>
</tbody>
</table>
### Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Transient</th>
<th>T+Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>1x</td>
<td>&gt;540x</td>
<td>gregor</td>
</tr>
<tr>
<td>morsecode</td>
<td>3x</td>
<td>&gt;250x</td>
<td>jpeg</td>
</tr>
<tr>
<td>sieve</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>snake</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>suffixtree</td>
<td>1x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tetris</td>
<td>5x</td>
<td>75x</td>
<td>take5</td>
</tr>
<tr>
<td>acquire</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dungeon</td>
<td>6x</td>
<td>48x</td>
<td>zombie</td>
</tr>
<tr>
<td>forth</td>
<td>2x</td>
<td>230x</td>
<td>zordoz</td>
</tr>
<tr>
<td>fsm</td>
<td>4x</td>
<td>100x</td>
<td></td>
</tr>
<tr>
<td>fsmoo</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Why is T+Blame so much worse than Reticulated?**

1. Larger, longer-running benchmarks
2. No dynamic type
Roadblock

T+Blame is too expensive!

Future: can run-time support reduce the cost?
Overall Performance
Overall Performance

Gradual types should support all mixed-typed configurations

N components => $2^N$ configurations
Overall Performance

Gradual types should support all mixed-typed configurations

N components => $2^N$ configurations

1x 2x 4x 10x 20x

synth
Overall Performance

Gradual types should support all mixed-typed configurations

N components => $2^N$ configurations

At $x=10$, count the % of configurations that run at most 10x slower than untyped
Overall Performance
Overall Performance

Guarded: % of fast-enough points

- synth
- take5
- quadU
- jpeg
- suffixtree
- dungeon

x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points
Overall Performance

Guarded vs Transient: % of fast-enough points

- synth
- take5
- quadU
- jpeg
- suffixtree
- dungeon

**x** axis = [1x, 20x] (sets a limit for "fast enough")

**y** axis = % of all gradually-typed points
Overall Performance

Guarded vs Transient: % of fast-enough points

x axis = [1x, 20x] (sets a limit for "fast enough")

y axis = % of all gradually-typed points
Overall Performance

Guarded vs Transient: % of fast-enough points

**Transient** ~ low costs in general

**Guarded** ~ high cost, but only for interactions

Future: systematically explore combinations

\[ x \text{ axis} = [1x, 20x] \text{ (sets a limit for "fast enough")} \quad y \text{ axis} = \% \text{ of all gradually-typed points} \]
In Conclusion

RQ: Can transient types:
    - scale to a rich type system
    - in the context of an existing compiler?
In Conclusion

RQ. Can **transient types**:  
- scale to a rich type system  
- in the context of an existing compiler?

Yes! ... without blame
In Conclusion

RQ. Can transient types:
- scale to a rich type system
- in the context of an existing compiler?

Yes! ... without blame

... and with some tailoring

✓ Overall performance is much improved
Reminder: **Transient** is a promising way to **strengthen** unsound **Optional** types

**Guarded** > **Transient** < **Optional**

Lots of potential clients!
The End
### Worst Case Overhead vs. Untyped

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Guarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>1x</td>
<td>&gt;540x</td>
</tr>
<tr>
<td>morsecode</td>
<td>3x</td>
<td>&gt;250x</td>
</tr>
<tr>
<td>sieve</td>
<td>4x</td>
<td>&gt;220x</td>
</tr>
<tr>
<td>snake</td>
<td>8x</td>
<td>&gt;1000x</td>
</tr>
<tr>
<td>suffixtree</td>
<td>6x</td>
<td>&gt;190x</td>
</tr>
<tr>
<td>tetris</td>
<td>10x</td>
<td>&gt;720x</td>
</tr>
<tr>
<td>acquire</td>
<td>1x</td>
<td>34x</td>
</tr>
<tr>
<td>dungeon</td>
<td>5x</td>
<td>75x</td>
</tr>
<tr>
<td>forth</td>
<td>6x</td>
<td>48x</td>
</tr>
<tr>
<td>fsm</td>
<td>2x</td>
<td>230x</td>
</tr>
<tr>
<td>fsmoo</td>
<td>4x</td>
<td>100x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transient</th>
<th>T+Blame</th>
<th>Guarded</th>
</tr>
</thead>
<tbody>
<tr>
<td>gregor</td>
<td>2x</td>
<td>23x</td>
</tr>
<tr>
<td>jpeg</td>
<td>2x</td>
<td>38x</td>
</tr>
<tr>
<td>lnm</td>
<td>1x</td>
<td>29x</td>
</tr>
<tr>
<td>mbta</td>
<td>2x</td>
<td>37x</td>
</tr>
<tr>
<td>quadT</td>
<td>7x</td>
<td>34x</td>
</tr>
<tr>
<td>quadU</td>
<td>8x</td>
<td>320x</td>
</tr>
<tr>
<td>synth</td>
<td>4x</td>
<td>220x</td>
</tr>
<tr>
<td>take5</td>
<td>3x</td>
<td>33x</td>
</tr>
<tr>
<td>zombie</td>
<td>31x</td>
<td>560x</td>
</tr>
<tr>
<td>zordoz</td>
<td>3x</td>
<td>220x</td>
</tr>
</tbody>
</table>


# Optimizations

<table>
<thead>
<tr>
<th>Topic</th>
<th>Ok for Transient?</th>
<th>Topic</th>
<th>Ok?</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>y</td>
<td>list</td>
<td>y</td>
</tr>
<tr>
<td>box</td>
<td>y</td>
<td>number</td>
<td>y</td>
</tr>
<tr>
<td>dead-code</td>
<td>N</td>
<td>pair</td>
<td>N</td>
</tr>
<tr>
<td>extflonum</td>
<td>y</td>
<td>sequence</td>
<td>y</td>
</tr>
<tr>
<td>fixnum</td>
<td>y</td>
<td>string</td>
<td>y</td>
</tr>
<tr>
<td>float-complex</td>
<td>y</td>
<td>struct</td>
<td>y</td>
</tr>
<tr>
<td>float</td>
<td>y</td>
<td>vector</td>
<td>y</td>
</tr>
</tbody>
</table>

https://prl.ccs.neu.edu/blog/2020/01/15/the-typed-racket-optimizer-vs-transient
Example: Retic. and Dyn

Most of the local variables get the Dynamic type and skip blame-map updates.

```python
def permutations(iterable: List(int)) -> List(List(int)):
    pool = tuple(iterable)
    n = len(pool)
    r = n
    indices = list(range(n))
    cycles = list(range(n-r+1, n+1))[::-1]
    result = [ [pool[i] for i in indices[:r]] ]
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                ....
```

No Wrappers = Simpler

(define b : (Boxof Char)
  (box #\X))

(define any : Any b)

(set-box! any #\Y)

Guarded

Transient

Error

OK
Limitation

Neither **Guarded** nor **Transient** TR allows occurrence types at a boundary

```
(require/typed racket/function
  (identity (-> Any Boolean : String)))
;; ^ Not permitted!

(define x : Any 0)

(define fake-str : String
  (if (identity x)
    (ann x String)
    (error 'unreachable)))

(string-length fake-str)
```