MEASURING RETICULATED PYTHON

Ben Greenman and Zeina Migeed
Northeastern University
The stages of grief are: Denial, Acceptance, Bargaining, Depression, Anger.
Reticulated Python

- Gradual typing for Python  [DLS 2014]
- Static type checking
- Dynamic type enforcement
- Formal model is type is sound  [POPL 2017]
def f(n):
    return n*(n+1) // 2

def get_numbers(count):
    nums = []
    for i in range(1, 1+count):
        nums.append(f(i))
    return nums

get_numbers(4)
# [1, 3, 6, 10]
def \( f(n:\text{Int}) \rightarrow \text{Int} \):
    return \( n \times (n+1) \) \( \div 2 \)

def get_numbers(count:\text{Int}) \rightarrow \text{List(\text{Int})}:
    nums = []
    for \( i \) in \text{range}(1, 1+\text{count}):
        nums.append(f(i))
    return nums

get_numbers(4)
# [1, 3, 6, 10]
Example Program, Partially Typed

```python
def f(n:Int):
    return n*(n+1) // 2

def get_numbers(count)->List(Int):
    nums = []
    for i in range(1, 1+count):
        nums.append(f(i))
    return nums

get_numbers(4)
# [1, 3, 6, 10]

f("not a number")
# Static type error

get_numbers("not a number")
# Dynamic type error
```
Reticulated Python

• Gradual typing for Python [DLS 2014]
• Static type checking
• Dynamic type enforcement
• Formal model is type is sound [POPL 2017]
STAGE I: GRIEF
def f(n: Int):
    return n*(n+1) // 2

def get_numbers(count) -> List(Int):
    nums = []
    for i in range(1, 1+count):
        nums.append(f)  # typo!
    return nums

def apply_first(funs):
    return funs[0](10)

get_numbers(4)
# [<fun>, <fun>, <fun>, <fun>]
apply_first(get_numbers(4))
# 55
Another Something Weird

```python
@fields({'dollars': Int,
         'cents': Int})
class Cash:
    dollars = 0
    cents = 0

def add_dollars(self, dollars):
    self.dollars += dollars

def get_cash() -> Cash:
    c = Cash()
    c.add_dollars(3.14159)
    return c

get_cash()
# Cash(3.14159, 0)
```
STAGE II: DENIAL
Type Soundness

If \( e \) has type \( T \), then either:

- \( e \) reduces to a value \( v \) with type \( T \)
- \( e \) raises an error due to a partial primitive
- \( e \) diverges
If $e$ has type $T$, then either:

- $e$ reduces to a value $v$ with type $\lfloor T \rfloor$
  - e.g. $\lfloor \text{Int} \to \text{Int} \rfloor = \to$
- $e$ raises a blame error
- $e$ diverges
Big Types in Little Runtime
Open-World Soundness and Collaborative Blame for Gradual Type Systems

Michael M. Vitousek    Cameron Swords    Jeremy G. Siek
Indiana University, USA
{mvitouse,cwords,jsiek}@indiana.edu

Abstract
Gradual typing combines static and dynamic typing in the same language, offering programmers the error detection and strong guarantees of static types and the rapid prototyping and flexible programming idioms of dynamic types. Many gradually typed languages are implemented by translation into an untyped target language (e.g., Typed Clojure, TypeScript, Gradualtalk, and Reticulated Python). For such languages, it is desirable to support typed code interacts: the consistency relation plays the role that type equality usually does in the type system. Types are consistent if they are equal up to the presence of $\star$.

Most existing gradually typed languages operate by translating a surface language program into an underlying target language, which is then executed. For many gradually-typed systems such as Typed Racket and TypeScript, the target language is a dynamically typed programming language, and gradually-typed programs are compiled to gradually typed with bound code in the dynamic
Corollary 5.5.1 (Type soundness). If $\emptyset \vdash e_s \leadsto e : T$ then $\emptyset; \emptyset \vdash e : \llbracket T \rrbracket$ and either:

- $\langle e, \emptyset, \emptyset \rangle \rightarrow^* \langle v, \sigma, B \rangle$ and $\emptyset; \Sigma \vdash v : \llbracket T \rrbracket$ and $\Sigma \vdash \sigma$, or
- $\langle e, \emptyset, \emptyset \rangle \rightarrow^* \text{BLAME}(L)$, or
- for all $\varsigma$ such that $\langle e, \emptyset, \emptyset \rangle \rightarrow^* \varsigma$, have that $\varsigma = \langle e', \sigma, B \rangle$ and exists $\varsigma'$ such that $\langle e', \sigma, B \rangle \rightarrow \varsigma'$. 
\[ [T] = S \]

\[ [\ast] = \ast \quad [\text{int}] = \text{int} \]

\[ [T_1 \rightarrow T_2] = \rightarrow \quad [\text{ref } T] = \text{ref} \]

\[ T \triangleright T \]

\[ \text{ref } T \triangleright \text{ref } T \quad \ast \triangleright \ast \rightarrow \ast \]

\[ T_1 \rightarrow T_2 \triangleright T_1 \rightarrow T_2 \]

\[ T_1 \sim T_2 \]

\[ \text{int } \sim \text{int} \quad \ast \sim T \quad T \sim \ast \]

\[ \frac{T_1 \sim T_2}{\text{ref } T_1 \sim \text{ref } T_2} \quad \frac{T_1 \sim T_3}{T_1 \rightarrow T_2 \sim T_3 \rightarrow T_4} \quad \frac{T_2 \sim T_4}{\text{ref } T_1 \sim \text{ref } T_2} \]

**Figure 3.** Translation from \( \lambda^{\rightarrow} \) to \( \lambda^{\downarrow} \).
STAGE III: ANGER, BARGAINING, DEPRESSION
What are Reticulated Types Good For?

- Protect invariants? No
- Reliable documentation? No
- Enable optimizations? No

Any untyped code

=>

No compositional reasoning!
STAGE IV: ACCEPTANCE
Interoperability & Performance
Interoperability

```python
def get_numbers(count)->List(Int):
    ....
    return proxy(nums, List(Int))
```

- The proxy must be compatible with existing code

```python
nums.append(
len(nums)
nums is nums
```
def get_numbers(count) -> List(Int):
    ....
    return proxy(nums, List(Int))

• Allocation cost
• Traverse, recursively proxy
• Interpose on future operations
Measuring Typed Racket

- 20 programs
- Measured all gradually-typed configurations
- How many 20-deliverable?
Measuring Typed Racket

**Worst-Case Overhead**

<table>
<thead>
<tr>
<th>Program</th>
<th>Overhead</th>
<th>Program</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire</td>
<td>5</td>
<td>quadBG</td>
<td>4</td>
</tr>
<tr>
<td>dungeon</td>
<td>10</td>
<td>quadMB</td>
<td>139</td>
</tr>
<tr>
<td>forth</td>
<td>27</td>
<td>sieve</td>
<td>43</td>
</tr>
<tr>
<td>fsm</td>
<td>1527</td>
<td>snake</td>
<td>32</td>
</tr>
<tr>
<td>fsmoo</td>
<td>233</td>
<td>suffixtree</td>
<td>29</td>
</tr>
<tr>
<td>gregor</td>
<td>2</td>
<td>synth</td>
<td>47</td>
</tr>
<tr>
<td>kcfa</td>
<td>5</td>
<td>take5</td>
<td>1</td>
</tr>
<tr>
<td>lnm</td>
<td>1</td>
<td>tetris</td>
<td>34</td>
</tr>
<tr>
<td>mbta</td>
<td>1</td>
<td>zombie</td>
<td>292</td>
</tr>
<tr>
<td>morsecode</td>
<td>1</td>
<td>zordoz</td>
<td>1</td>
</tr>
</tbody>
</table>
Measuring Typed Racket

Frequently an order-of-magnitude slowdown
Measuring Reticulated

- 19 **different** programs
- Measured all *function-level* configurations
- How many 20-deliverable?

![Bar Chart](chart.png)
Measuring Reticulated

- 19 **different** programs
- Measured all *function-level* configurations
- How many 10-deliverable?
# Measuring Reticulated Worst-Case Overhead

<table>
<thead>
<tr>
<th>Function</th>
<th>Worst-Case Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>futen</td>
<td>1</td>
</tr>
<tr>
<td>http2</td>
<td>3</td>
</tr>
<tr>
<td>slowSHA</td>
<td>2</td>
</tr>
<tr>
<td>call_method</td>
<td>7</td>
</tr>
<tr>
<td>call_method_slots</td>
<td>8</td>
</tr>
<tr>
<td>call_simple</td>
<td>3</td>
</tr>
<tr>
<td>chaos</td>
<td>3</td>
</tr>
<tr>
<td>fannkuch</td>
<td>1</td>
</tr>
<tr>
<td>float</td>
<td>3</td>
</tr>
<tr>
<td>go</td>
<td>7</td>
</tr>
<tr>
<td>meteor</td>
<td>2</td>
</tr>
<tr>
<td>nbody</td>
<td>1</td>
</tr>
<tr>
<td>nqueens</td>
<td>1</td>
</tr>
<tr>
<td>pidigits</td>
<td>1</td>
</tr>
<tr>
<td>pystone</td>
<td>2</td>
</tr>
<tr>
<td>spectralnorm</td>
<td>8</td>
</tr>
<tr>
<td>Espionage</td>
<td>5</td>
</tr>
<tr>
<td>PythonFlow</td>
<td>7</td>
</tr>
<tr>
<td>take5</td>
<td>1</td>
</tr>
</tbody>
</table>
Measuring Reticulated

Never an order-of-magnitude slowdown
STAGE V: MOVING ON
Q1. Is Reticulated’s soundness practical?
Q2. Can Typed Racket soundness be performant?
Q3. Is Typed Racket soundness portable?
Q4. Is there a useful, “efficient” Soundness 3.0?