Pycket
A Tracing JIT For a Functional Language

Spenser Bauman\textsuperscript{1}  Carl Friedrich Bolz\textsuperscript{2}
Robert Hirschfeld\textsuperscript{3}  Vasily Kirilichev\textsuperscript{3}  Tobias Pape\textsuperscript{3}
Jeremy G. Siek\textsuperscript{1}  Sam Tobin-Hochstadt\textsuperscript{1}

\textsuperscript{1}Indiana University Bloomington, USA
\textsuperscript{2}King’s College London, UK
\textsuperscript{3}Hasso-Plattner-Institut, University of Potsdam, Germany

APLS
December 16, 2015
Problem: Racket is slow on generic code

Generic code:

```
(define (dot v1 v2)
  (for/sum ([ [e1 v1] [e2 v2]])
    (* e1 e2)))
```

```
(time (dot v1 v2)) ;; 3864 ms
```

Hand optimized:

```
(define (dot-fast v1 v2)
  (define len (flvector-length v1))
  (unless (= len (flvector-length v2))
    (error 'fail))
  (let loop ([n 0] [sum 0.0])
    (if (unsafe-fx= len n) sum
      (loop (unsafe-fx+ n 1)
        (unsafe-fl+ sum (unsafe-fl* (unsafe-flvector-ref v1 n)
                                   (unsafe-flvector-ref v2 n))))))
```

```
(time (dot-fast v1 v2)) ;; 268 ms
```
Problem: Racket is slow on contracts

```
(define/contract (dot-safe v1 v2)
  ((vectorof flonum?) (vectorof flonum?) . -> . flonum?)
  (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))

(time (dot-safe v1 v2)) ;; 8888 ms
```
Problem: Racket is slow wrt. gradual typing

*Is Sound Gradual Typing Dead?* Takikawa et al. POPL 2016

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Typed/Untyped Ratio</th>
<th>Max. Overhead</th>
<th>Mean Overhead</th>
<th>3-Deliverable</th>
<th>3/10-Usable</th>
</tr>
</thead>
<tbody>
<tr>
<td>kcfa</td>
<td>(7 modules)</td>
<td>1.00x</td>
<td>22.67x</td>
<td>32 (25%)</td>
<td>48 (38%)</td>
</tr>
<tr>
<td>tetris</td>
<td>(9 modules)</td>
<td>0.97x</td>
<td>117.28x</td>
<td>128 (25%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>snake</td>
<td>(8 modules)</td>
<td>0.92x</td>
<td>121.51x</td>
<td>4 (2%)</td>
<td>28 (11%)</td>
</tr>
<tr>
<td>synth</td>
<td>(10 modules)</td>
<td>1.03x</td>
<td>85.90x</td>
<td>15 (1%)</td>
<td>73 (7%)</td>
</tr>
</tbody>
</table>
Pycket is a tracing JIT compiler which reduces the need for manual specialization and reduces contract overhead.

```scheme
(time (dot v1 v2)) ;; 74 ms
(time (dot-fast v1 v2)) ;; 74 ms (268 ms on Racket)
(time (dot-safe v1 v2)) ;; 95 ms
```
Pycket tames overhead from gradual typing

kcfa

slowdown factor
0
20
40
60
80
100
120

number below
racket
pycket
hidden

slowdown factor
0
100
200
300
400
500

number below
racket
pycket
hidden

tetris

snake

synth

slowdown factor
0
50
100
150
200
250
300
400
500
600
700
800
900
1000

number below
racket
pycket
hidden
Idea: Apply dynamic language JIT compiler to Racket

Take: Racket + Apply: RPython Project

= Pycket
Background: Tracing JIT Compilation
Background: Tracing JIT Compilation

Program Execution Trace

Execution

Interpreter

side exit

Trace

guard
Background: The PyPy Meta-Tracing JIT

Python Interpreter → RPython Interpret & Profile → Tracing → Native Execution

Virtual Machine

Python Program Input

Optimize Code Gen.

hot loop
The Pycket Meta-Tracing JIT

- Racket Interpreter
- Virtual Machine
- Racket Program Input
- RPython Interpret & Profile
- Native Execution
- Tracing

hot loop

Optimize Code Gen.
Our Racket Interpreter: The CEK Machine

\[
e ::= x | \lambda x. e | (e e) | \text{letcc } x. e | e@e
\]

\[
\kappa ::= [] | \text{arg}(e, \rho)::\kappa | \text{fun}(v, \rho)::\kappa | \text{ccarg}(e, \rho)::\kappa | \text{cc}(\kappa)::\kappa
\]

\[
v ::= \lambda x. e | \kappa
\]

\[
\langle x, \rho, \kappa \rangle \longmapsto \langle \rho(x), \rho, \kappa \rangle
\]

\[
\langle (e_1 \ e_2), \rho, \kappa \rangle \longmapsto \langle e_1, \rho, \text{arg}(e_2, \rho)::\kappa \rangle
\]

\[
\langle v_1, \rho, \text{arg}(e_2, \rho')::\kappa \rangle \longmapsto \langle e_2, \rho', \text{fun}(v_1, \rho)::\kappa \rangle
\]

\[
\langle v_2, \rho, \text{fun}(\lambda x. e, \rho')::\kappa \rangle \longmapsto \langle e, \rho'[x \mapsto v_2], \kappa \rangle
\]

\[
\langle \text{letcc } x. e, \rho, \kappa \rangle \longmapsto \langle e, \rho[x \mapsto \kappa], \kappa \rangle
\]

\[
\langle (e_1 @ e_2), \rho, \kappa \rangle \longmapsto \langle e_1, \rho, \text{ccarg}(e_2, \rho)::\kappa \rangle
\]

\[
\langle \kappa_1, \rho, \text{ccarg}(e_2, \rho')::\kappa \rangle \longmapsto \langle e_2, \rho', \text{cc}(\kappa_1) :: \kappa \rangle
\]

\[
\langle v_2, \rho, \text{cc}(\kappa_1)::\kappa \rangle \longmapsto \langle v_2, \rho, \kappa_1 \rangle
\]

*Programming Languages and Lambda Calculi.* Flatt and Felleisen. 2007
Challenges particular to Racket

- Detect loops for trace compilation in a higher-order language without explicit loop constructs
- Reduce the need for manual specialization
- Reduce the overhead imposed by contracts
Loop finding: cyclic paths

Record cycles in control flow

Default RPython strategy
Tracing cycles in the control flow is insufficient

The CEK machine has no notion of a program counter, can try to use AST nodes instead.

```
1. (define (my-add a b) (+ a b))
2. (define (loop a b)
3.   (my-add a b)
4.   (my-add a b)
5.   (loop a b))
```

Begin tracing at a **hot** node and continue until that node is reached again

```
(+ a b)
```
Tracing cycles in the control flow is insufficient.

```
(define (my-add a b) (+ a b))
(define (loop a b)
  (my-add a b)
  (my-add a b)
(loop a b))
```

Begin tracing at a **hot** node and continue until that node is reached again.

![Arrow diagram](image)
Tracing cycles in the control flow is insufficient

1. \( \text{define } \) (my-add a b) (+ a b))
2. \( \text{define } \) (loop a b)
3. \( \text{(my-add a b)} \)
4. \( \text{(my-add a b)} \)
5. \( \text{(loop a b)} \)

Begin tracing at a **hot** node and continue until that node is reached again
Tracing cycles in the control flow is insufficient.

```
(define (my-add a b) (+ a b))
(define (loop a b)
  (my-add a b)
  (my-add a b)
  (loop a b))
```

Begin tracing at a **hot** node and continue until that node is reached again.
Newer definition: A loop is a cycle in the program’s call graph.

1. Build the callgraph during execution
2. Mark functions in a cycle as a loop
Data Structure Specialization

Unbox small, fixed-size arrays of Racket values

```
<table>
<thead>
<tr>
<th>Vals</th>
<th>*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Fixnum: 1</td>
</tr>
<tr>
<td>1</td>
<td>Flonum: 3.14</td>
</tr>
<tr>
<td>2</td>
<td>Symbol: 'a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>EnvSize3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Val0</td>
</tr>
<tr>
<td>Val1</td>
</tr>
<tr>
<td>Val2</td>
</tr>
</tbody>
</table>

Symbol: 'a
```
Specialized Mutable Objects

Optimistically specialize the representation of homogeneous containers

When a mutating operation invalidates the current strategy, the storage is rewritten — this is fortunately infrequent

[Bolz et al., OOPSLA 2013]
Pycket: What Works?

- File IO
  
  (open-input-file "list.txt")
  (open-output-file "brain.dat")

- Numeric tower
  
  number? complex? real? rational? integer? ...

- Contracts
  
  (define-contract ...)

- Typed Racket
  
  #lang typed/racket

- Primitive Functions ($\sim 900/1400$)
Pycket: What Doesn’t Work?

- FFI
- Scribble
  
  \texttt{#lang scribble/base}
- DrRacket
- Web
  
  \texttt{#lang web-server/insta}
- Threads
  
  \texttt{(thread ( () ...))}
- Lesser used primitives
# Performance Caveats

<table>
<thead>
<tr>
<th>Fast</th>
<th>Slow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tight loops</td>
<td>Branchy/irregular control flow</td>
</tr>
<tr>
<td>Numeric Computations</td>
<td>Code not easily expressed as loops</td>
</tr>
<tr>
<td></td>
<td>Interpreters</td>
</tr>
<tr>
<td></td>
<td>Short-running programs</td>
</tr>
</tbody>
</table>
Benchmarks
Overall Performance

Larceny Benchmarks

- racket
- larceny
- gambit
- system
- bigloo
- pycket

Geomean runtime

Shootout Benchmarks

- racket
- system
- pycket
Contracts and Chaperones

\[
\text{(define (dot } v1 \ \text{v2) (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))}
\]

\[
\text{(define/contract (dotc } v1 \ \text{v2) ((vectorof flonum?) (vectorof flonum?) . -> . flonum?) (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))}
\]

- Pycket supports Racket’s implementation of higher-order software contracts via \textit{impersonators} and \textit{chaperones}
- Used to support Type Racket’s implementation of gradual typing
- Overhead = Enforcement Cost + Extra Indirection

[Strickland, Tobin-Hochstadt, Findler, Flatt 2012]
Benchmarks: Contracts

Chaperone Slowdown

- Bubble
- Church
- Struct
- Ode
- Binomial

Sloonsdown

- System
- Pycket
- Racket
Future Improvements

- Improve chaperone/impersonator performance and space usage
- Explore interaction between ahead-of-time and just-in-time optimizations
- Green threads and inter-thread optimizations
- Improve performance on complicated control flow
- Support more of Racket
Dynamic language JIT compilation is a viable implementation strategy for functional languages

Novel loop detection method for trace compilation of a higher-order language

Significant reduction in contract overhead

Significant reduction in the need for manual specialization

https://github.com/samth/pysket