Pycket
A Tracing JIT For a Functional Language

Spenser Bauman[1] Carl Friedrich Bolz[2]
Robert Hirschfeld[3] Vasily Kirilichev[3] Tobias Pape[3]
Jeremy G. Siek[1] Sam Tobin-Hochstadt[1]

[1]Indiana University Bloomington, USA

[2]King's College London, UK

[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

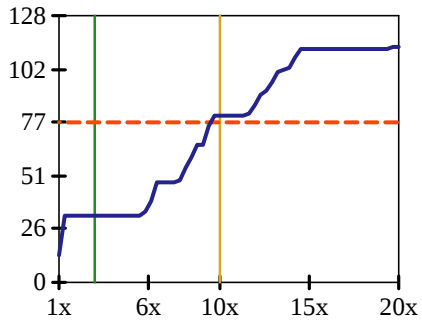
```
(define/contract (dot-safe v1 v2)
 ((vectorof flonum?) (vectorof flonum?) . -> . flonum?)
 (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))

(time (dot-safe v1 v2)) ;; 8888 ms
```

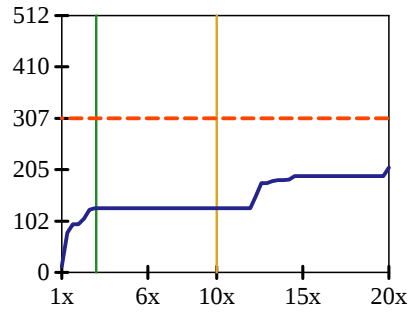
Problem: Racket is slow wrt. gradual typing

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

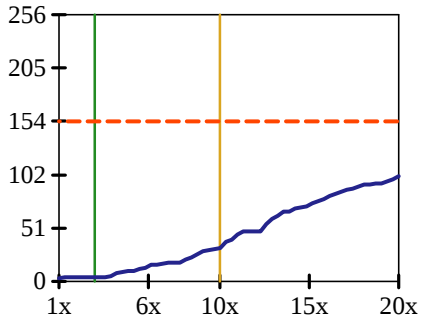
kcfa	(7 modules)
typed/untyped ratio	1.00x
max. overhead	22.67x
mean overhead	9.23x
3-deliverable	32 (25%)
3/10-usable	48 (38%)



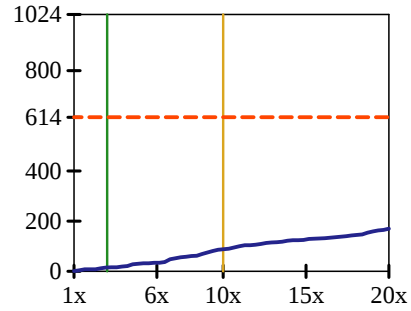
tetris	(9 modules)
typed/untyped ratio	0.97x
max. overhead	117.28x
mean overhead	33.34x
3-deliverable	128 (25%)
3/10-usable	0 (0%)



snake	(8 modules)
typed/untyped ratio	0.92x
max. overhead	121.51x
mean overhead	32.30x
3-deliverable	4 (2%)
3/10-usable	28 (11%)



synth	(10 modules)
typed/untyped ratio	1.03x
max. overhead	85.90x
mean overhead	39.69x
3-deliverable	15 (1%)
3/10-usable	73 (7%)

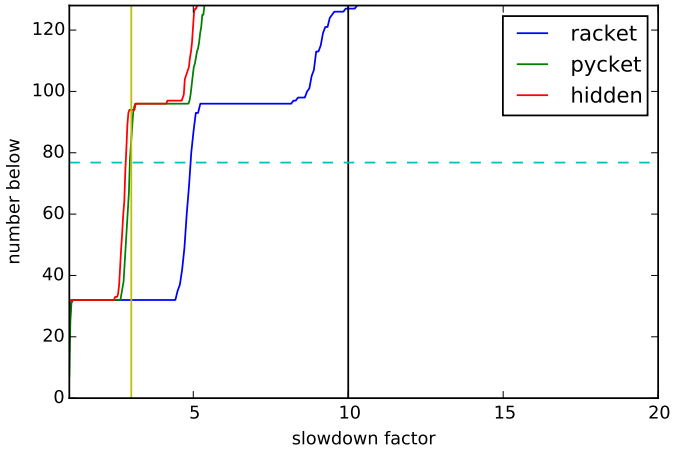


Pycket is a tracing JIT compiler which reduces the need for manual specialization and reduces contract overhead.

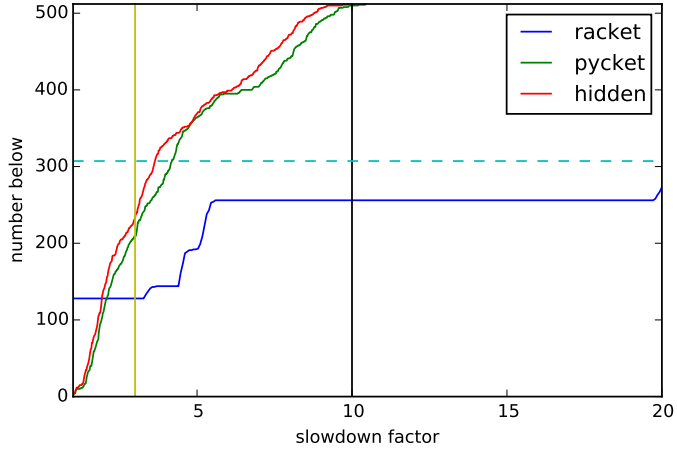
```
(time (dot v1 v2))       ;; 74 ms
(time (dot-fast v1 v2)) ;; 74 ms  (268 ms on Racket)
(time (dot-safe v1 v2)) ;; 95 ms
```

Pycket tames overhead from gradual typing

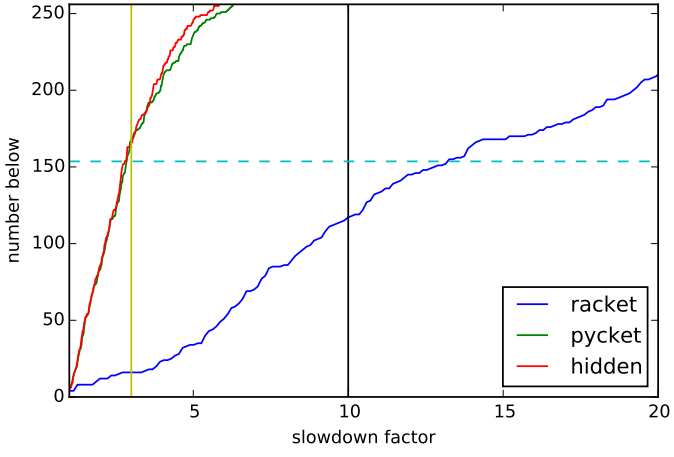
kcfa



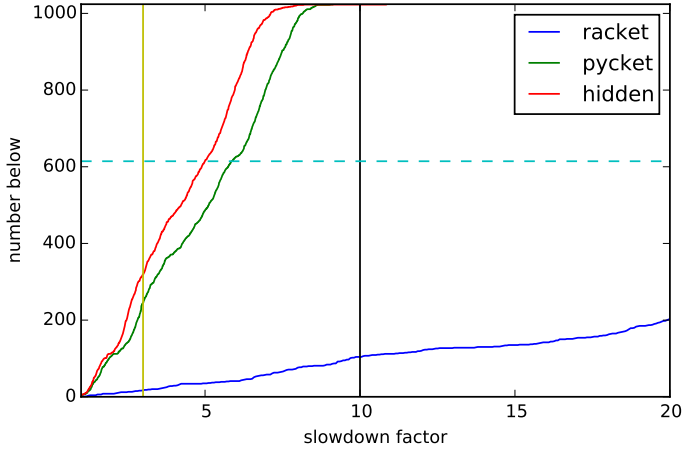
tetris



snake



synth



Idea: Apply dynamic language JIT compiler to Racket

Take: Racket

+

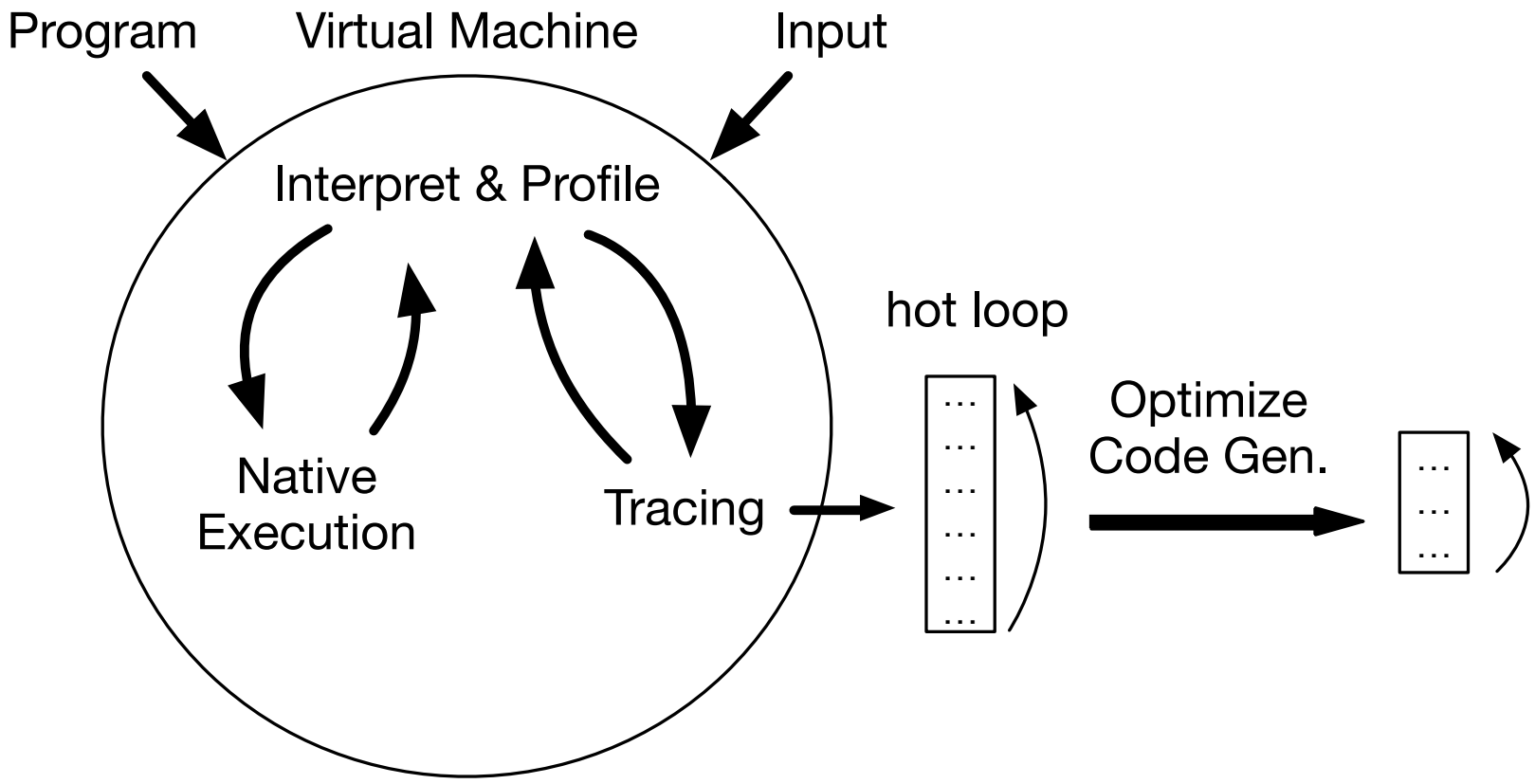
Apply: RPython Project

pypy

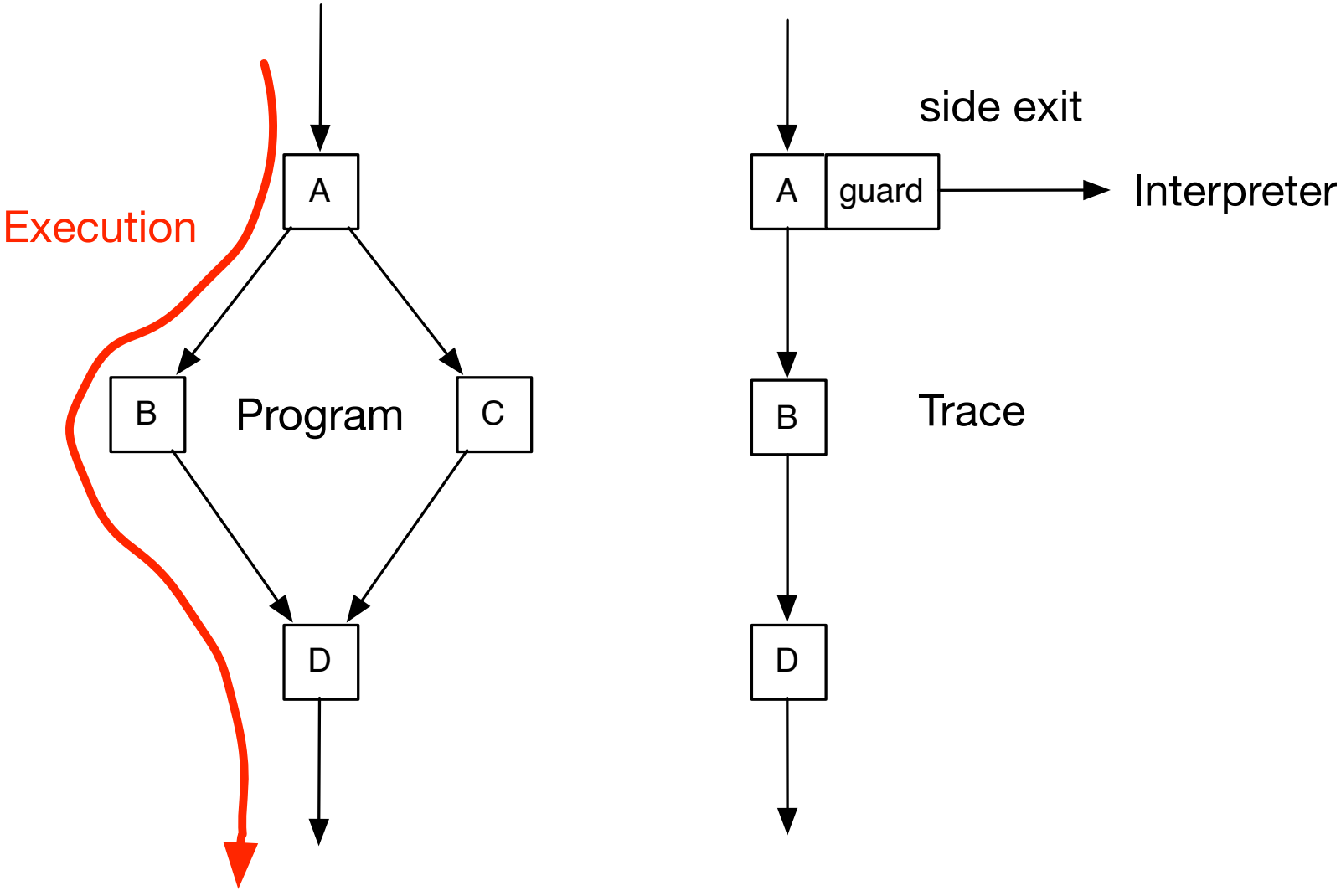
=

Pycket

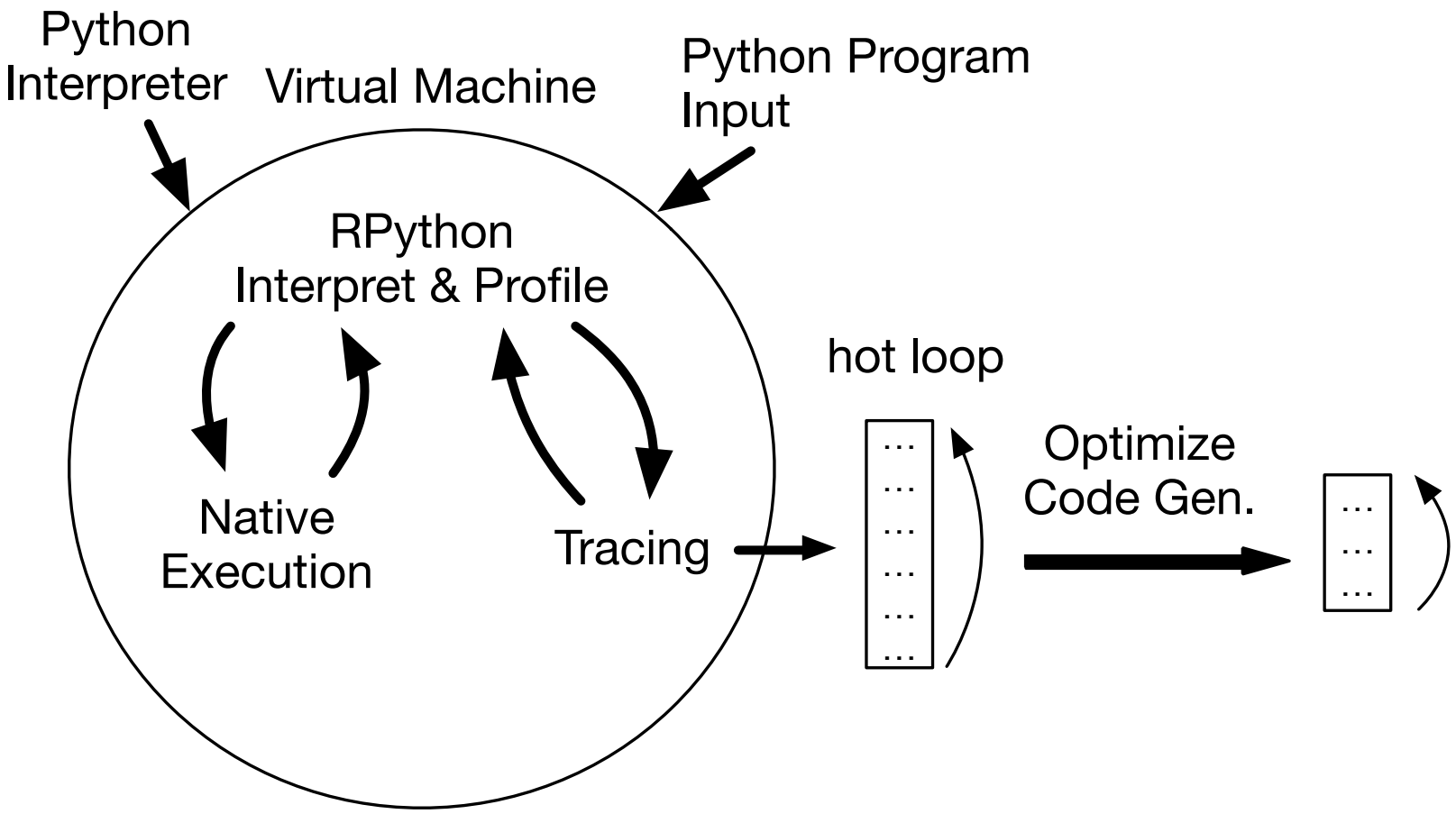
Background: Tracing JIT Compilation



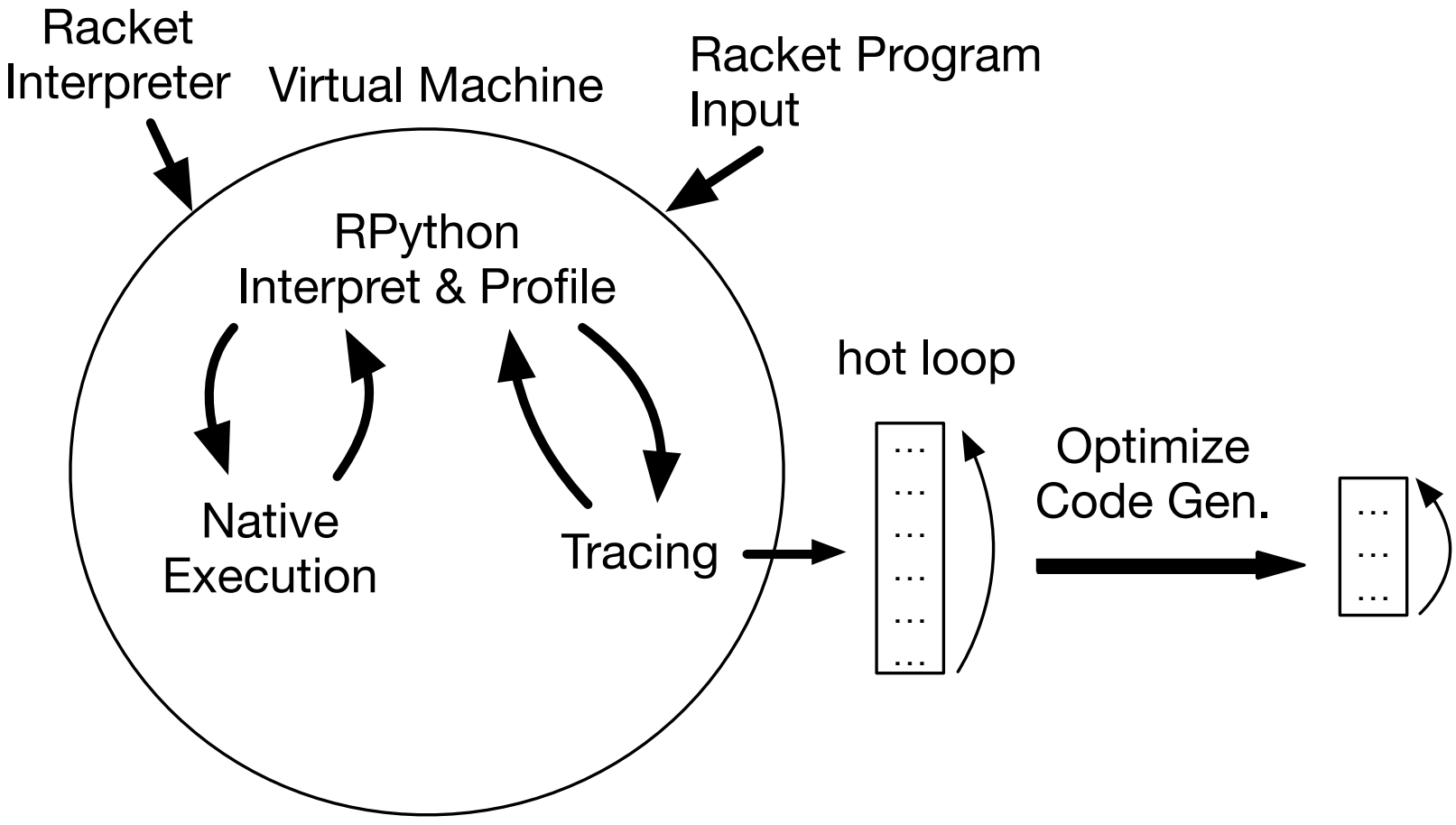
Background: Tracing JIT Compilation



Background: The PyPy Meta-Tracing JIT



The Pycket Meta-Tracing JIT



Our Racket Interpreter: The CEK Machine

$$e ::= x \mid \lambda x.\, e \mid (e\ e) \mid \mathsf{letcc}\ x.\, e \mid e@e$$
$$\kappa ::= [] \mid \mathsf{arg}(e, \rho)::\kappa \mid \mathsf{fun}(v, \rho)::\kappa \mid \mathsf{ccarg}(e, \rho)::\kappa \mid \mathsf{cc}(\kappa)::\kappa$$
$$v ::= \lambda x.\, e \mid \kappa$$

$$\langle x, \rho, \kappa\rangle \longmapsto \langle \rho(x), \rho, \kappa\rangle$$

$$\langle (e_1\ e_2), \rho, \kappa\rangle \longmapsto \langle e_1, \rho, \mathsf{arg}(e_2, \rho)::\kappa\rangle$$
$$\langle v_1, \rho, \mathsf{arg}(e_2, \rho')::\kappa\rangle \longmapsto \langle e_2, \rho', \mathsf{fun}(v_1, \rho)::\kappa\rangle$$
$$\langle v_2, \rho, \mathsf{fun}(\lambda x.\, e, \rho')::\kappa\rangle \longmapsto \langle e, \rho'[x \mapsto v_2], \kappa\rangle$$

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

$$\langle (e_1 @ e_2), \rho, \kappa\rangle \longmapsto \langle e_1, \rho, \mathsf{ccarg}(e_2, \rho)::\kappa\rangle$$
$$\langle \kappa_1, \rho, \mathsf{ccarg}(e_2, \rho')::\kappa\rangle \longmapsto \langle e_2, \rho', \mathsf{cc}(\kappa_1) :: \kappa\rangle$$
$$\langle v_2, \rho, \mathsf{cc}(\kappa_1)::\kappa)\rangle \longmapsto \langle v_2, \rho, \kappa_1\rangle$$

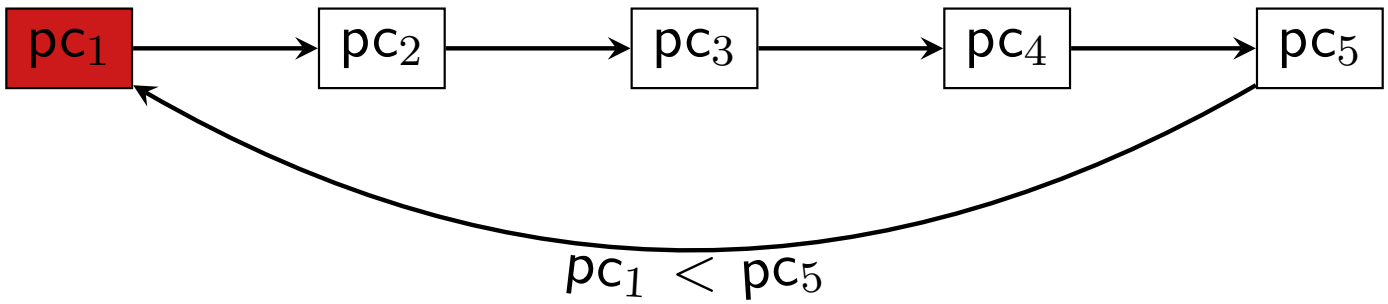
Programming Languages and Lambda Calculi. Flatt and Felleisen. 2007

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))
```

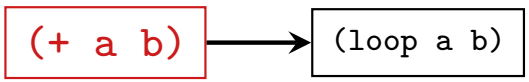
Begin tracing at a hot node and continue until that node is reached again

`(+ a b)`

Tracing cycles in the control flow is insufficient

```
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))
```

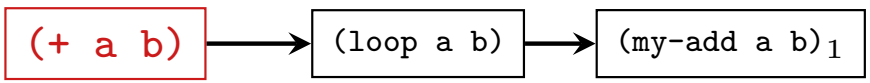
Begin tracing at a hot node and continue until that node is reached again



Tracing cycles in the control flow is insufficient

```
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))
```

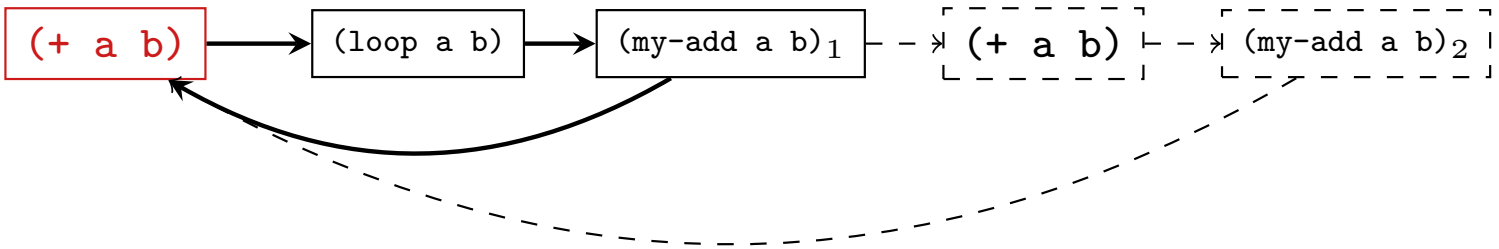
Begin tracing at a hot node and continue until that node is reached again



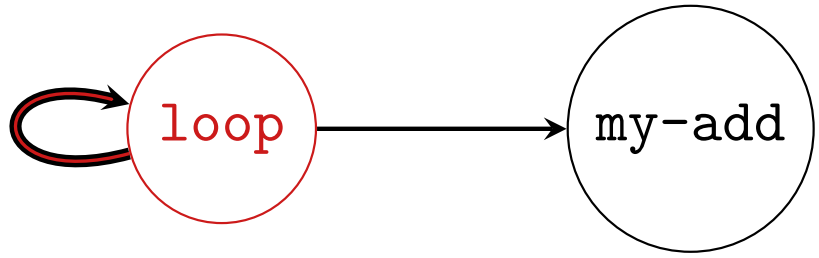
Tracing cycles in the control flow is insufficient

```
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

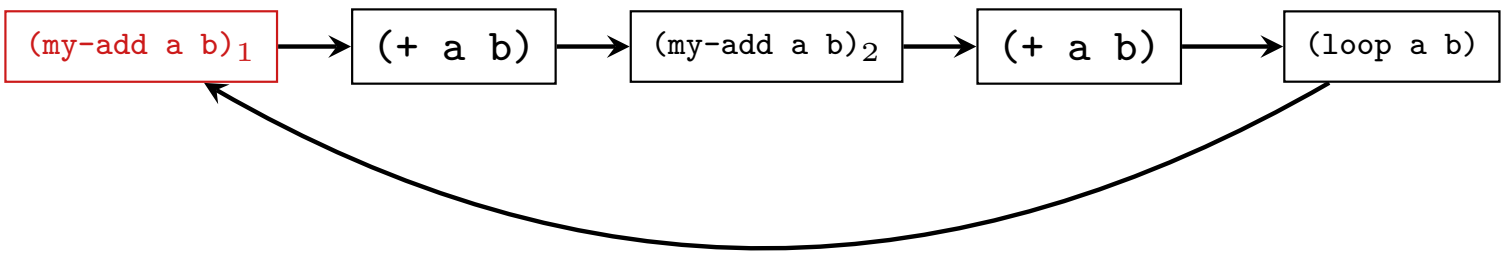


The Callgraph



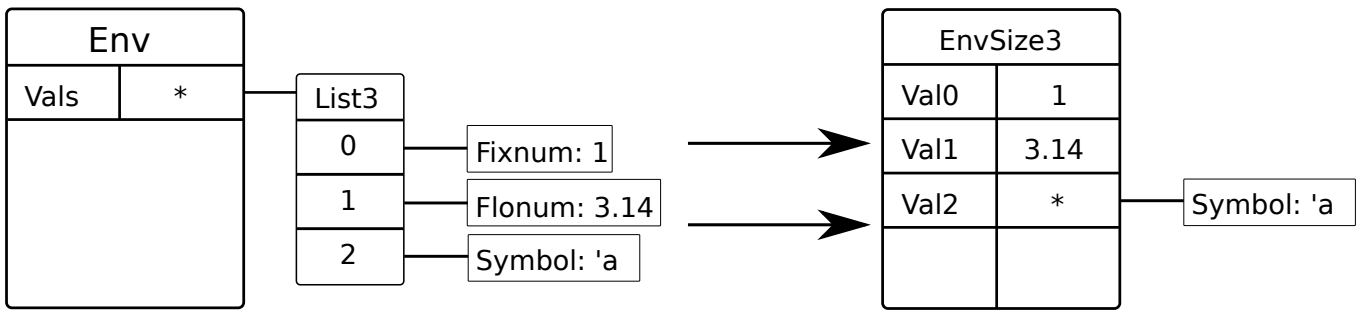
Newer definition: A *loop* is a cycle in the program's call graph.

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



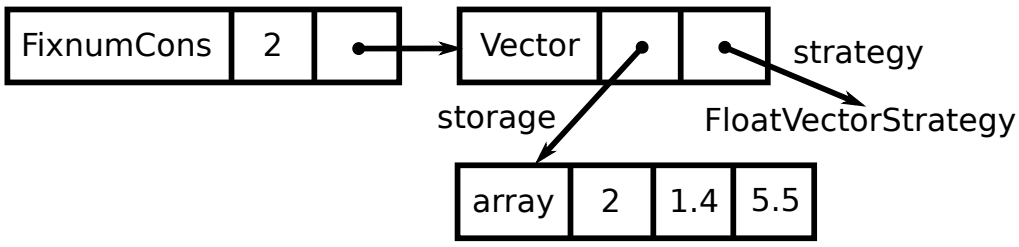
Data Structure Specialization

Unbox small, fixed-size arrays of Racket values



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

  ```
  #lang scribble/base
  ```
- DrRacket
- Web

  ```
  #lang web-server/insta
  ```
- Threads

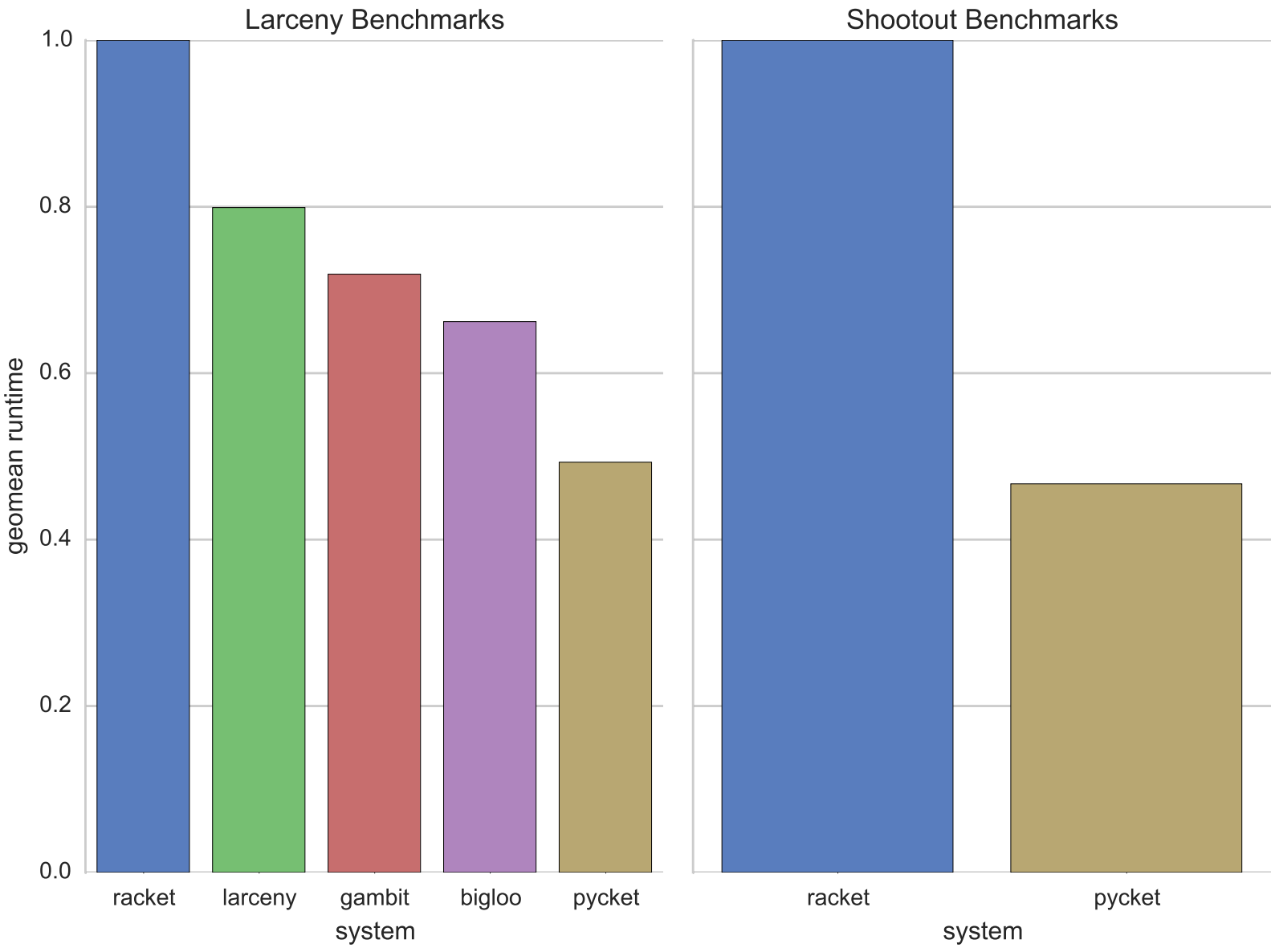
  ```
  (thread ( () ...))
  ```
- Lesser used primitives

Performance Caveats

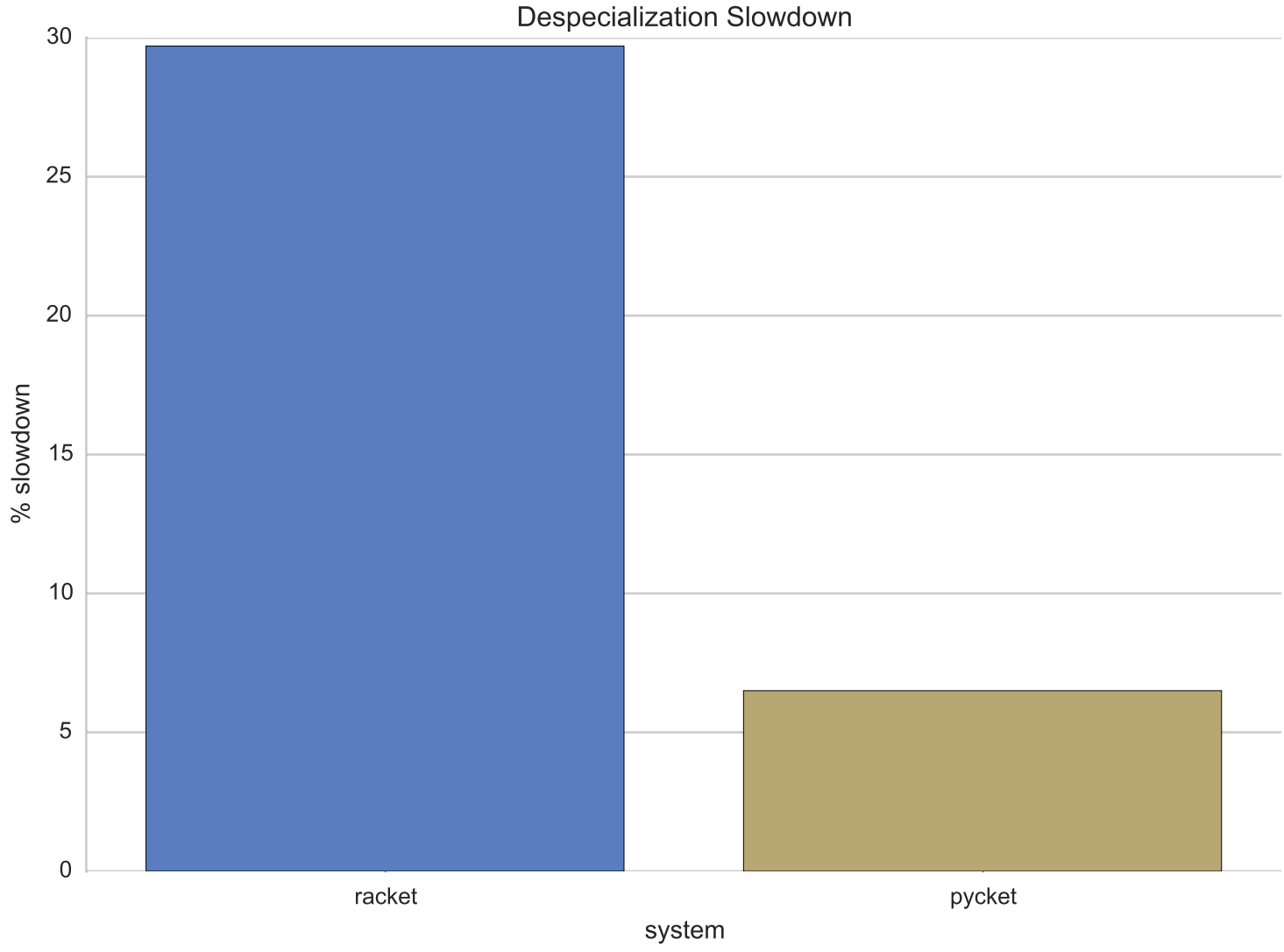
Fast	Slow
Tight loops	Branchy/irregular control flow
Numeric Computations	Code not easily expressed as loops
	Interpreters
	Short-running programs

Benchmarks

Overall Performance



Specialization



Contracts and Chaperones

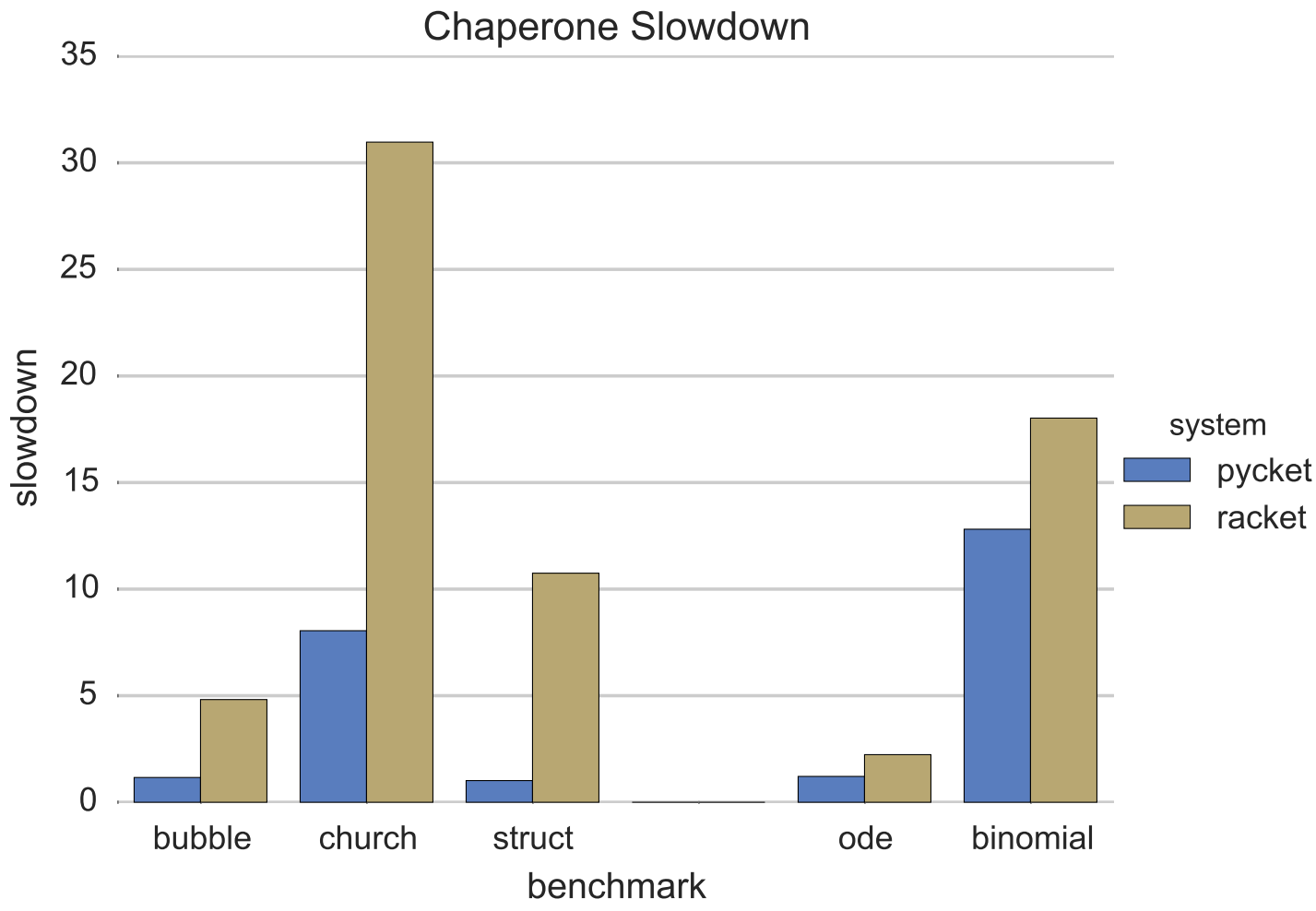
```
(define (dot v1 v2)
  (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))

(define/contract (dotc v1 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 *impersonators* and *chaperones*
- Used to support Type Racket's implementation of gradual typing
- Overhead = Enforcement Cost + Extra Indirection

[Strickland, Tobin-Hochstadt, Findler, Flatt 2012]

Benchmarks: Contracts



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

Thank You

- 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/pycket