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

ICFP
August 31st 2015

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

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

```
(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
```

```
(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
```

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

---

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

Idea: Apply dynamic language JIT compiler to Racket

Take: Racket

+

Apply: RPython Project

PyPy

=

Major Challenges

- 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

Design

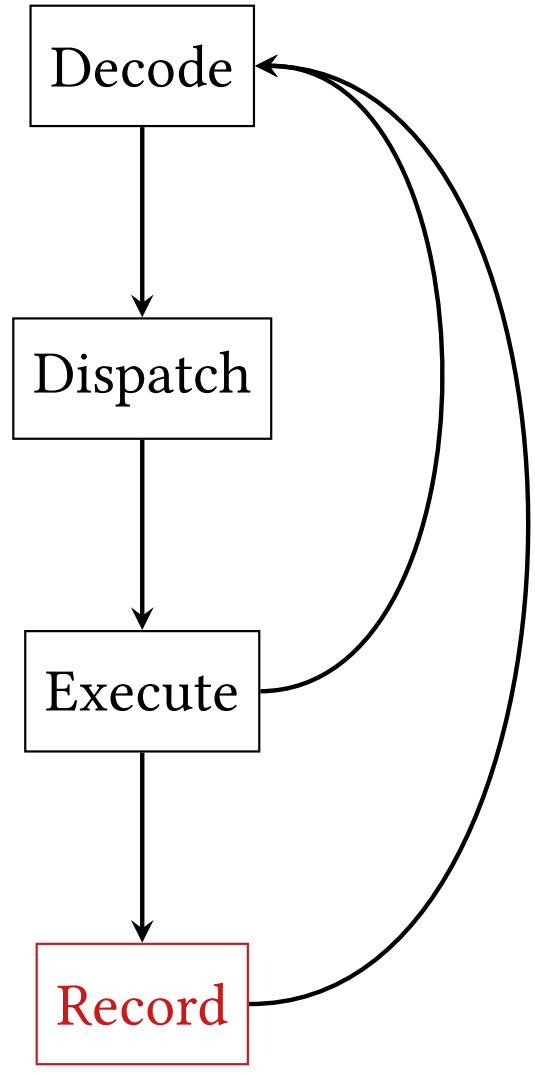
CEK Machine

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

$$\begin{aligned}
\langle x, \rho, \kappa\rangle &\longmapsto \langle \rho(x), \rho, \kappa\rangle \\
\langle (e_1\ e_2), \rho, \kappa\rangle &\longmapsto \langle e_1, \rho, \mathsf{arg}(e_2, \rho)::\kappa\rangle \\
\langle v, \rho, \mathsf{arg}(e, \rho')::\kappa\rangle &\longmapsto \langle e, \rho', \mathsf{fun}(v, \rho)::\kappa\rangle \\
\langle v, \rho, \mathsf{fun}(\lambda x.\, e)::\kappa\rangle &\longmapsto \langle e, \rho'[x \mapsto v], \kappa\rangle
\end{aligned}$$

From CEK Machine to Machine Code

1. Record emulation instructions during interpretation
2. Generate code for recorded instruction sequence
3. Bail to interpreter when control flow diverges

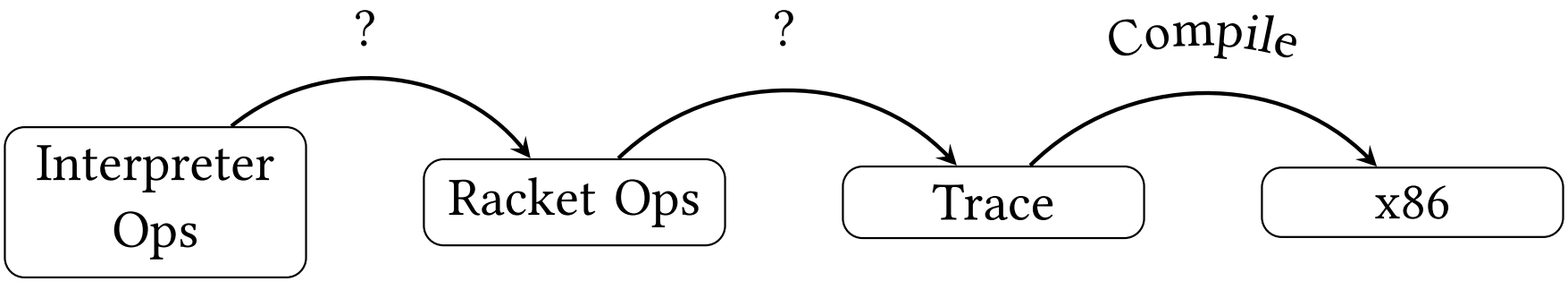


Tracing

Unit of compilation = loops

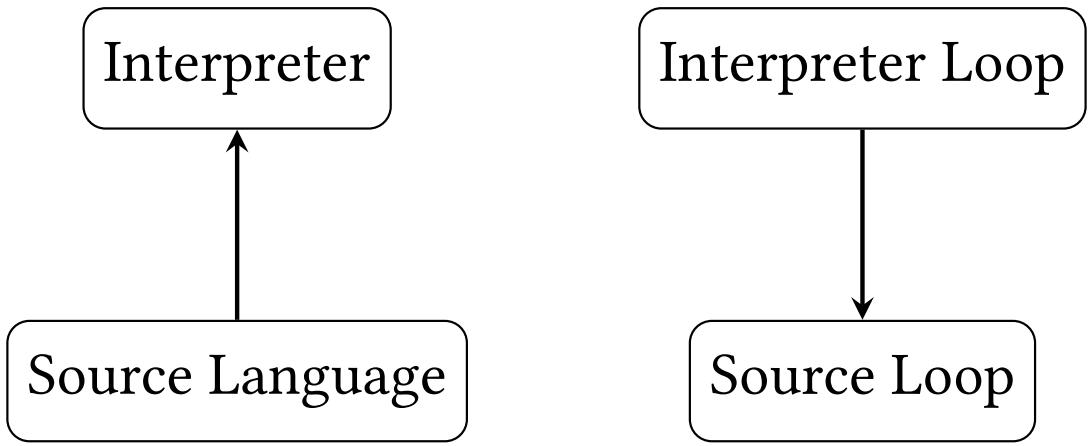
dot becomes

```
label(acc, idx1, idx2, len1, len2, arr1, arr2)
guard(idx1 < len1)
guard(idx2 < len2)
val1     = getarrayitem_gc(arr1, idx1)
val2     = getarrayitem_gc(arr2, idx2)
prod     = val1 * val2
acc_new  = acc + prod
idx1_new = idx1 + 1
idx2_new = idx2 + 1
jump(acc_new, idx1_new, idx2_new, len1, len2, arr1, arr2)
```


?
?
Compile
Interpreter Ops
Racket Ops
Trace
x86

Meta-Tracing

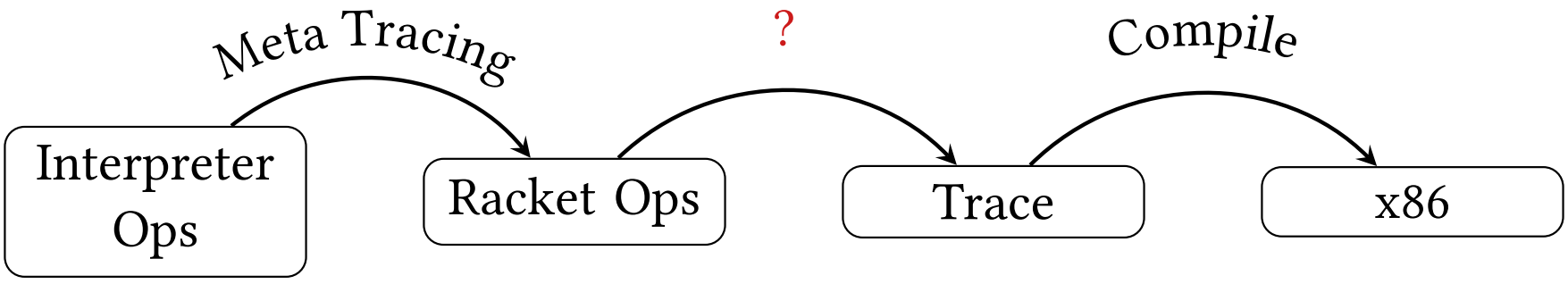
Meta-tracing: Trace the interpreter rather than the source language



[Bolz, Cuni, Fijałkowski, Rigo 2009]

Tracing The Racket Level

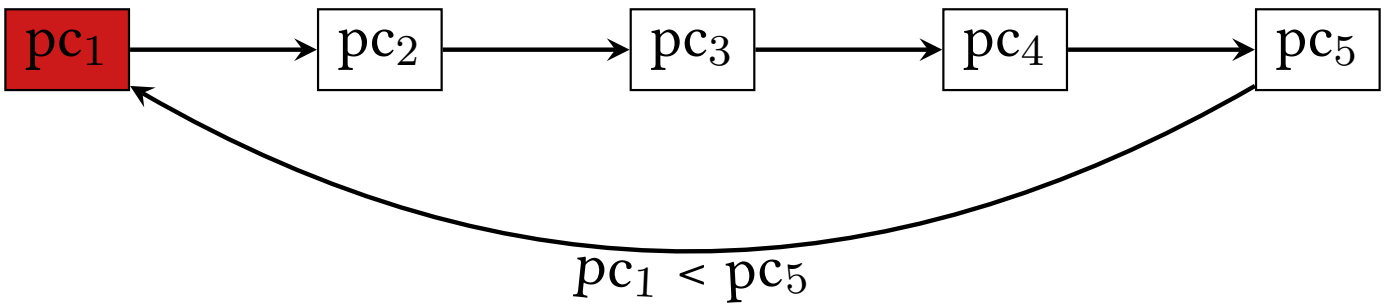
How to expose the high level interpreter structure to the JIT?



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

```
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))
```

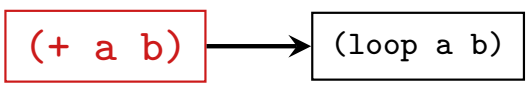
Begin tracing at a hot node and continue until that node is reached again

`(+ 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))
```

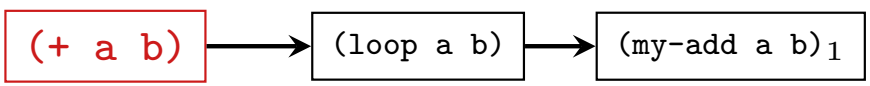
Begin tracing at a hot node and continue until that node is reached again



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))
```

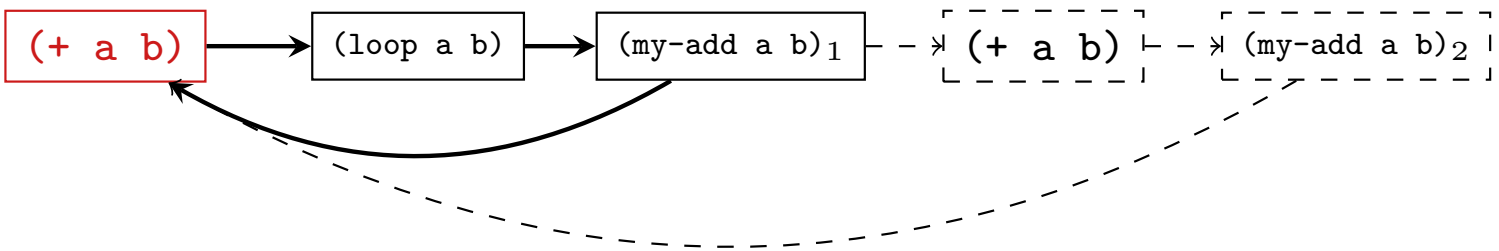
Begin tracing at a hot node and continue until that node is reached again



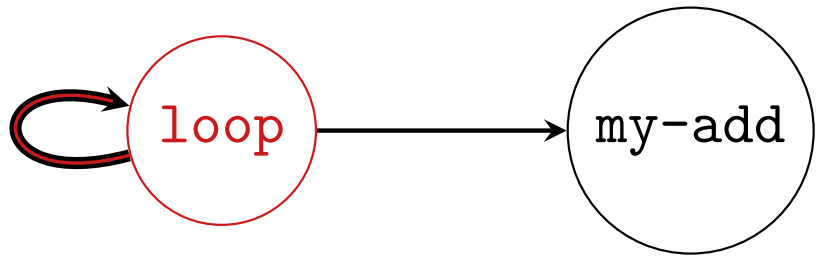
Tracing cycles in the control flow is insufficient

```
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

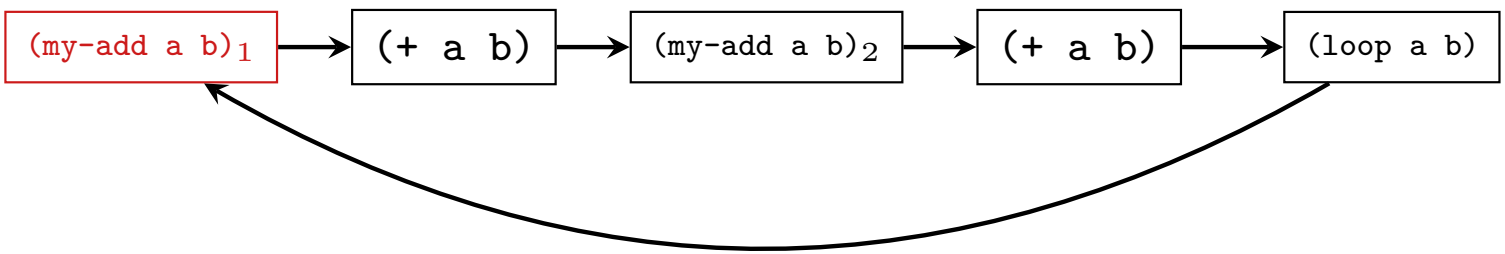


The Callgraph



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

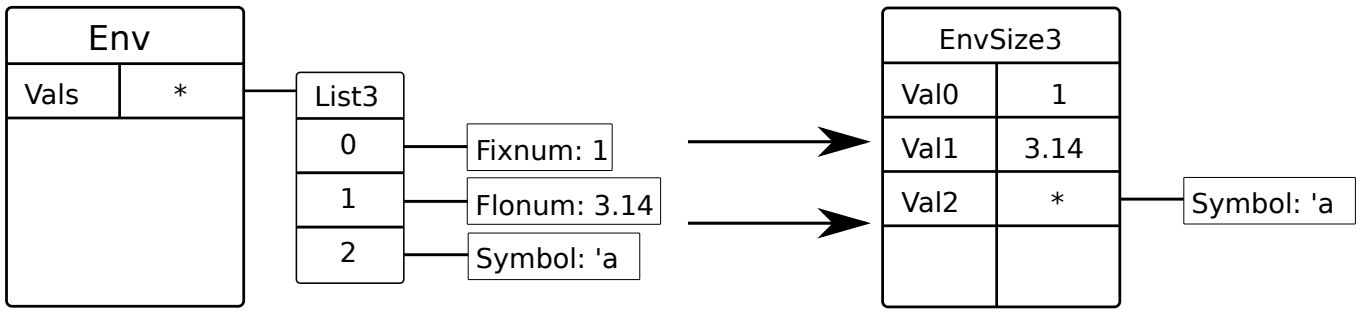
1. Build the callgraph during execution
2. Mark functions in a cycle as a loop



Data Specialization

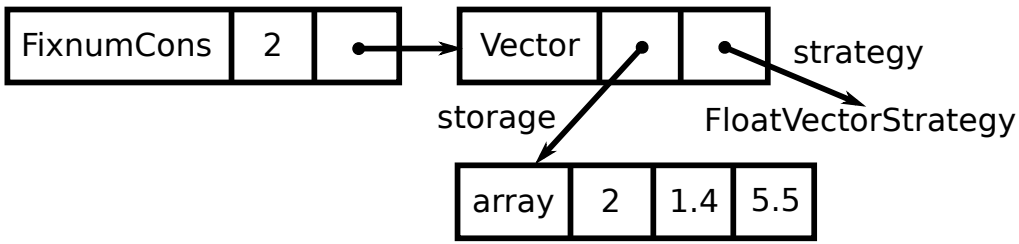
Data Structure Specialization

Unpack small, fixed-size arrays of Racket values unboxing fixnums and flonums



Specialized Mutable Objects

Optimistically specialize the representation of homogeneous containers

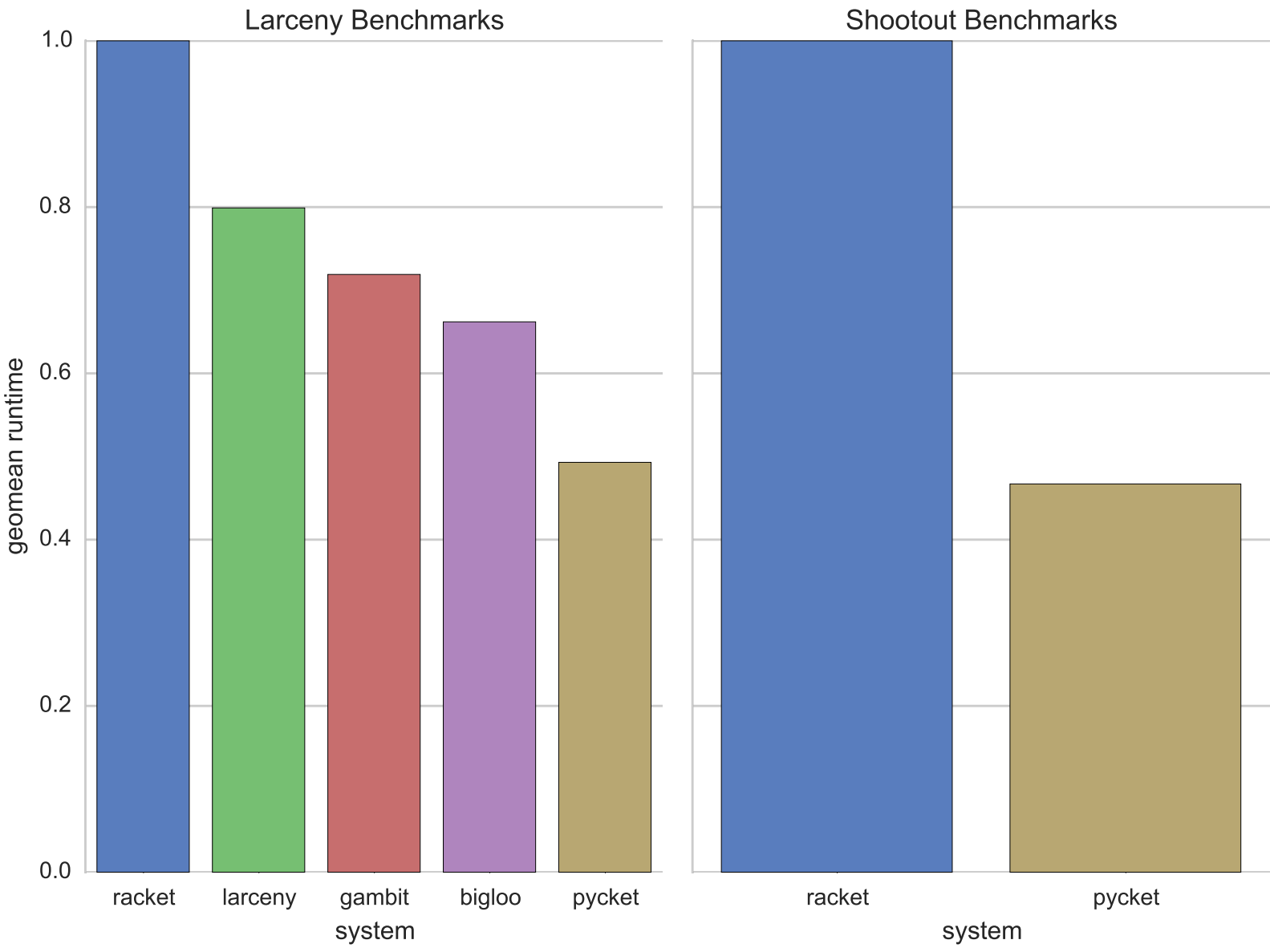


When a mutating operation invalidates the current strategy, the storage is rewritten — this is fortunately infrequent

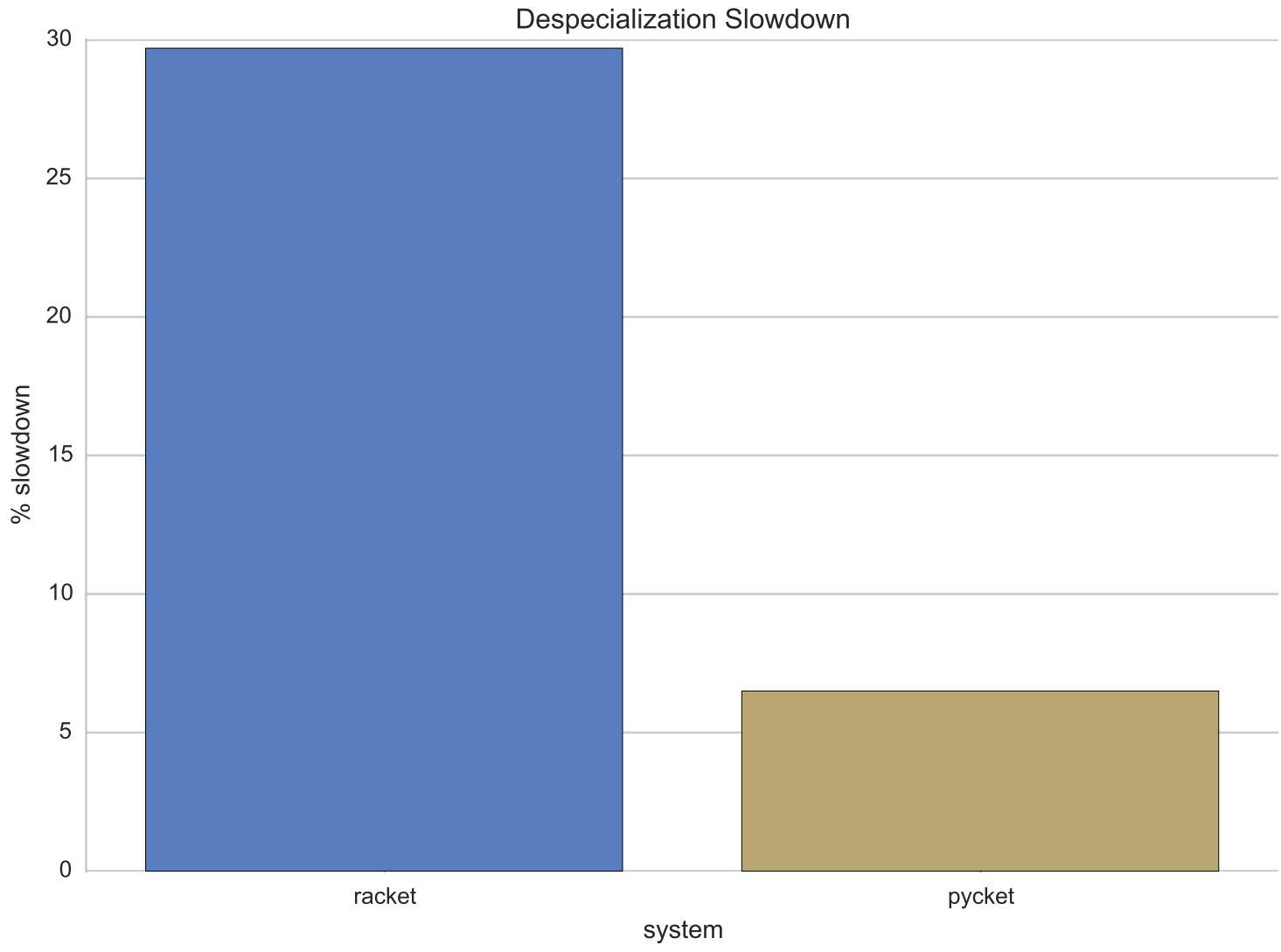
[Bolz, Diekmann, Tratt 2013]

Benchmarks

Overall Performance



Specialization



Contracts and Chaperones

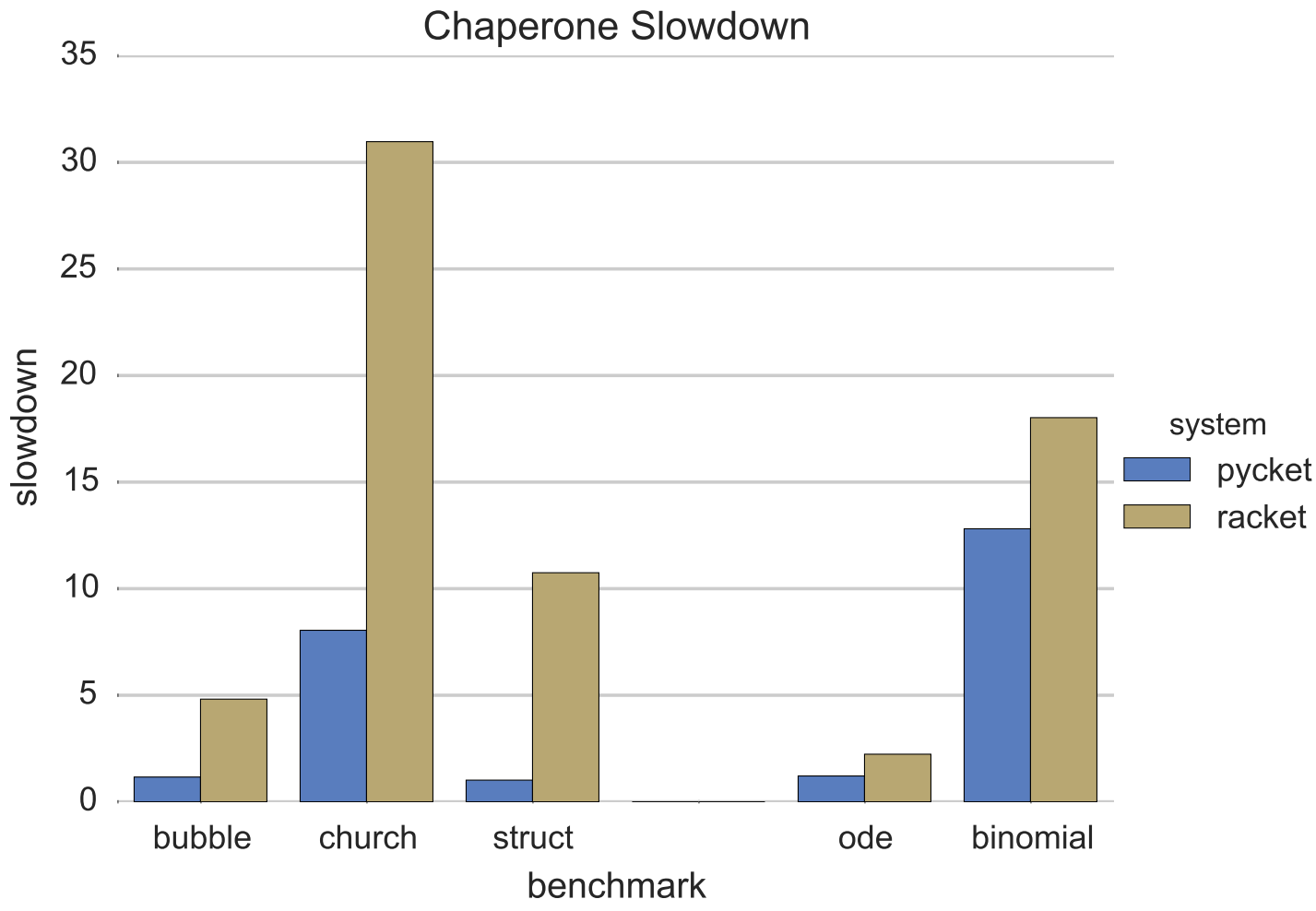
```
  (define (dot v1 v2)
    (for/sum ([e1 v1] [e2 v2]) (* e1 e2)))

(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



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