Pycket

A functional language
and a tracing JIT

Sam Tobin-Hochstadt

Indiana University

A simple program

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

A simple program

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

506 ms (size 10000000)

A simple program

```
(define (dot u v)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

39 ms (size 1000000)

A simple program

```
(define/contract (dot u v)
  ((vectorof flonum?) (vectorof flonum?)
   . -> . flonum?)
  (for/sum ([x u] [y v])
    (* x y)))
```

1159 ms (size 1000000)

A simple program

```
(define (dot 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 v1 n)))))))
```

29 ms (size 1000000)

Success?

✓ A range of options

✓ Including fast performance

Failure?

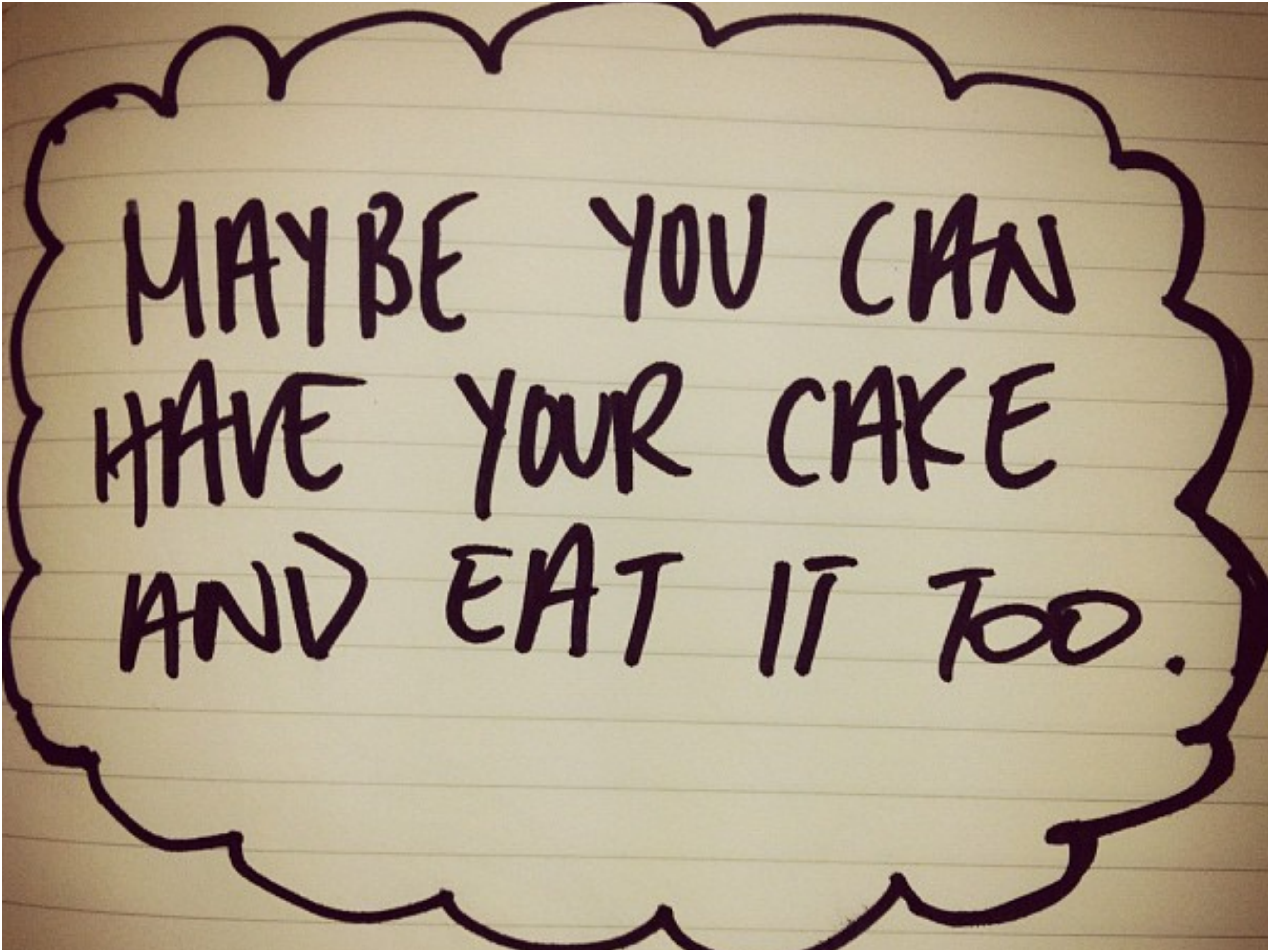
- ✗ High level or fast: pick one
- ✗ Where does this leave design?

Why are contracts hard to optimize?

```
(contract (-> integer? integer?) (lambda (x) x))
```

Why are contracts hard to optimize?

```
(chaperone-procedure
 (lambda (x) x)
 (lambda (v) (unless (integer? v) (error ’blame)) v)
 (lambda (v) (unless (integer? v) (error ’blame)) v))
```



MAYBE YOU CAN
HAVE YOUR CAKE
AND EAT IT TOO.

Enter Pycket

With added cake ...

```
(define (dot 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 v1 n)))))))
```

8 ms (size 1000000)

With added cake …

```
(define (dot u v)
  (for/sum ([x (in-vector u)]
            [y (in-vector v)])
    (fl* x y)))
```

11 ms (size 1000000)

With added cake ...

```
(define (dot u v)
  (for/sum ([x u]
            [y v])
    (* x y)))
```

12 ms (size 10000000)

With added cake ...

```
(define/contract (dot u v)
  ((vectorof flonum?) (vectorof flonum?)
    . -> . flonum?)
  (for/sum ([x u] [y v])
    (* x y)))
```

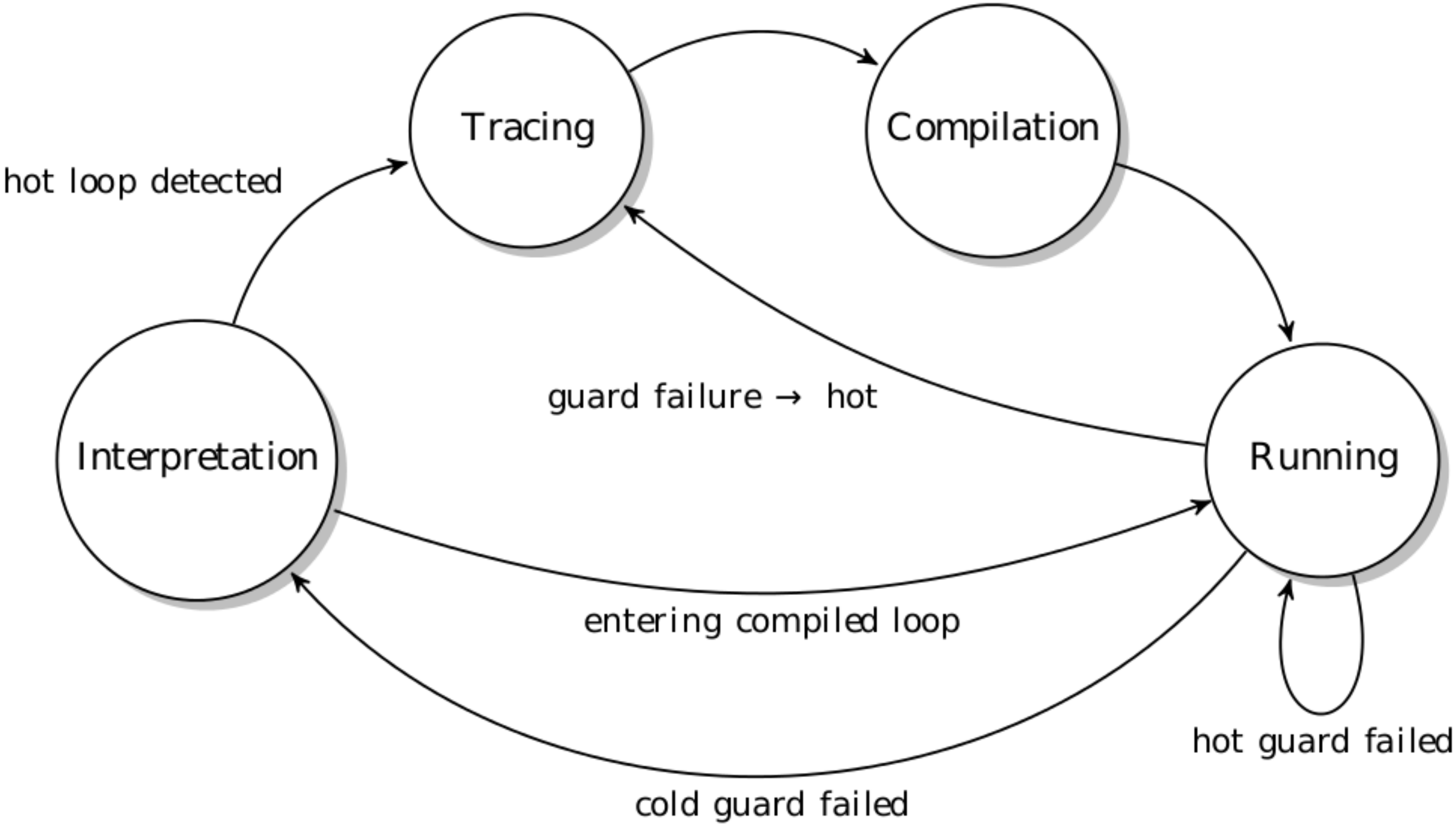
17 ms (size 1000000)

How does it work?

Tracing JIT

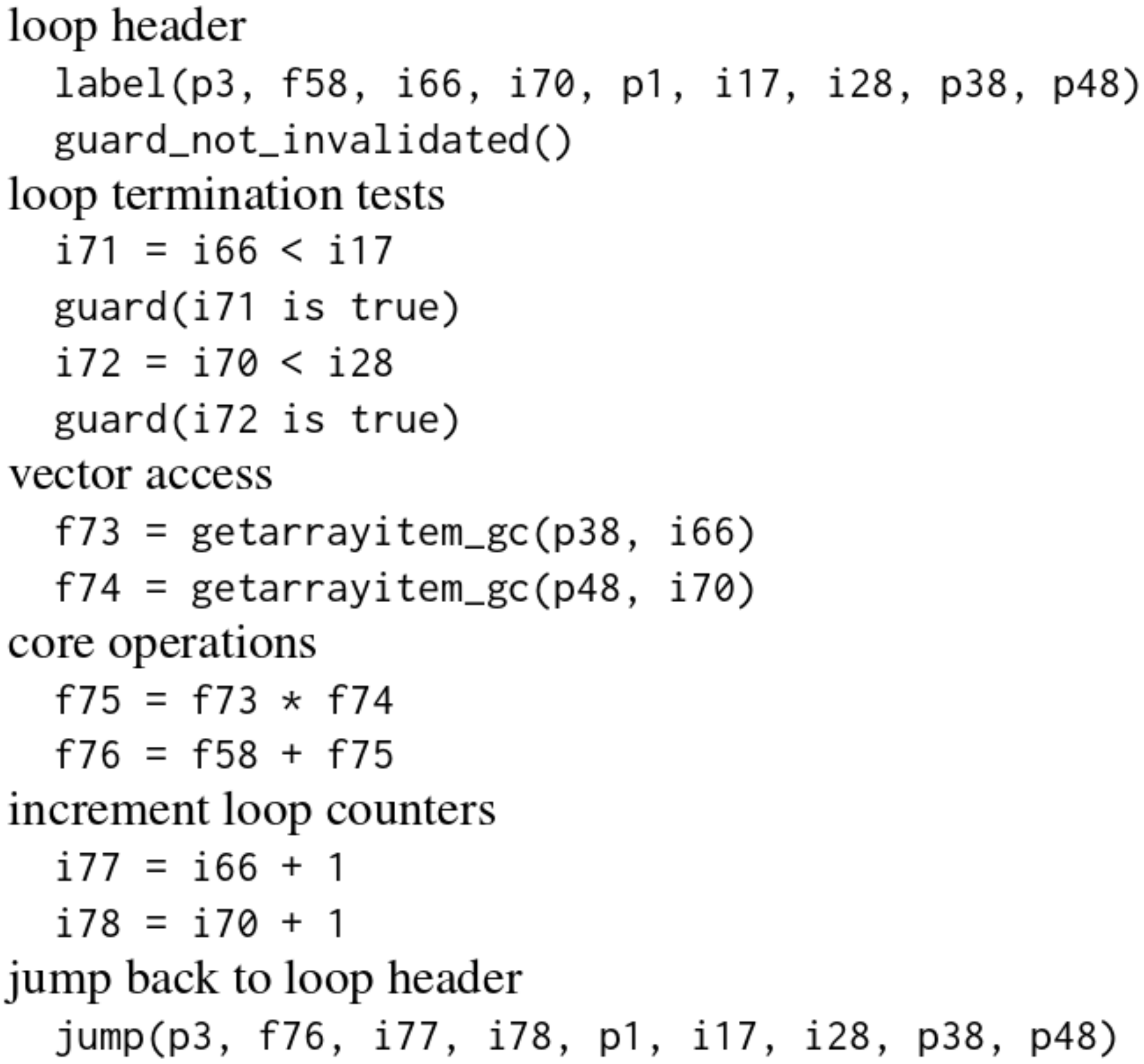
1. Interpret Program
2. Find hot loop
3. Record operations for one iteration
4. Optimize
5. Switch to new code

Tracing JIT



(Diagram from Antonio Cuni)

Dot product Inner Loop



```
loop header
  label(p3, f58, i66, i70, p1, i17, i28, p38, p48)
  guard_not_invalidated()
loop termination tests
  i71 = i66 < i17
  guard(i71 is true)
  i72 = i70 < i28
  guard(i72 is true)
vector access
  f73 = getarrayitem_gc(p38, i66)
  f74 = getarrayitem_gc(p48, i70)
core operations
  f75 = f73 * f74
  f76 = f58 + f75
increment loop counters
  i77 = i66 + 1
  i78 = i70 + 1
jump back to loop header
  jump(p3, f76, i77, i78, p1, i17, i28, p38, p48)
```

Key Optimizations

Inlining (happens for free)

Constant propagation

Allocation Removal

Meta-tracing: the magic part

We didn't write a JIT or an optimizer!

We didn't write a JIT or an optimizer!

RPython creates a JIT from an interpreter

CEK Machine

$$e ::= x \mid \lambda x.\, e \mid e\ e$$

$$\kappa ::= [] \mid \mathsf{arg}(e, \rho){::}\kappa \mid \mathsf{fun}(v, \rho){::}\kappa$$

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

CEK Advantages

Fast continuations

Tail recursion

Arbitrary size stack

CEK Advantages

Fast continuations

Tail recursion

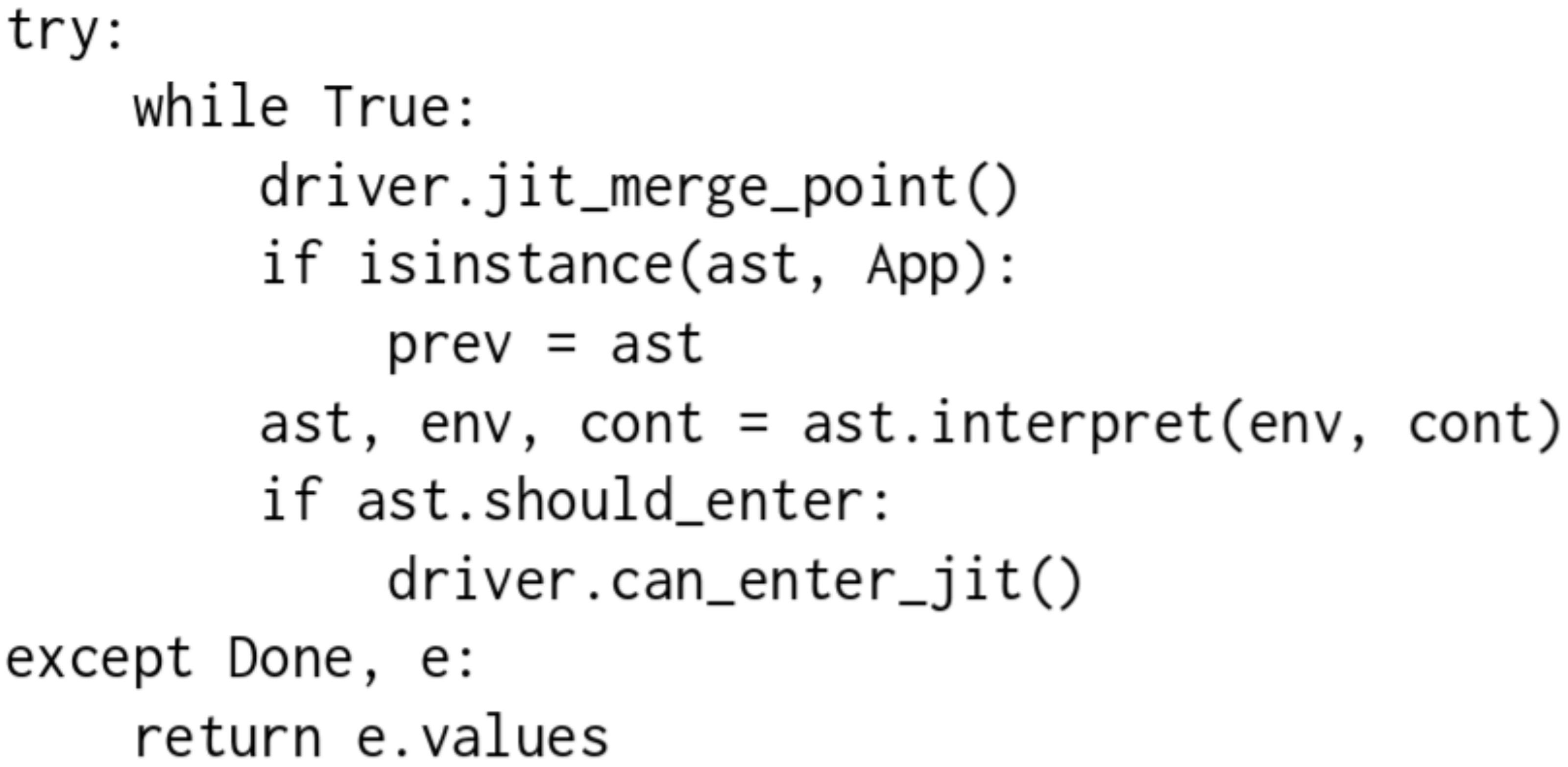
Arbitrary size stack

Allocation everywhere

From CEK to JIT

1. Whole-program type inference
2. Translation to C
3. Adding JIT based on hints

Main Interpreter Loop



```
try:
    while True:
        driver.jit_merge_point()
        if isinstance(ast, App):
            prev = ast
        ast, env, cont = ast.interpret(env, cont)
        if ast.should_enter:
            driver.can_enter_jit()
except Done, e:
    return e.values
```

Other hints

Immutable Data

Loop unrolling

Constant functions

Specialization

Optimizations

Optimization in the interpreter

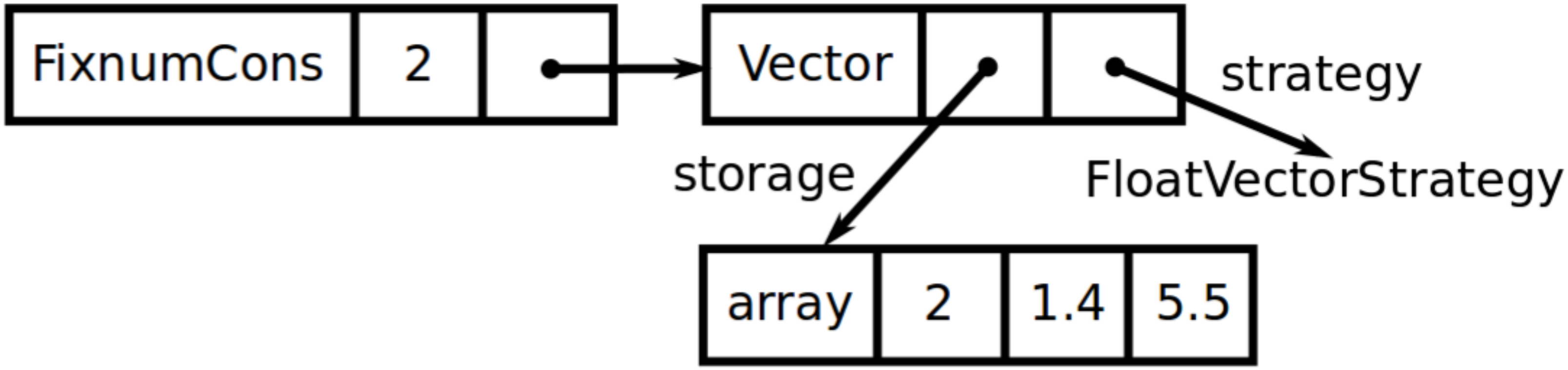
A-normalization

Assignment conversion

Environment optimization

Data structure specialization

Storage Strategies



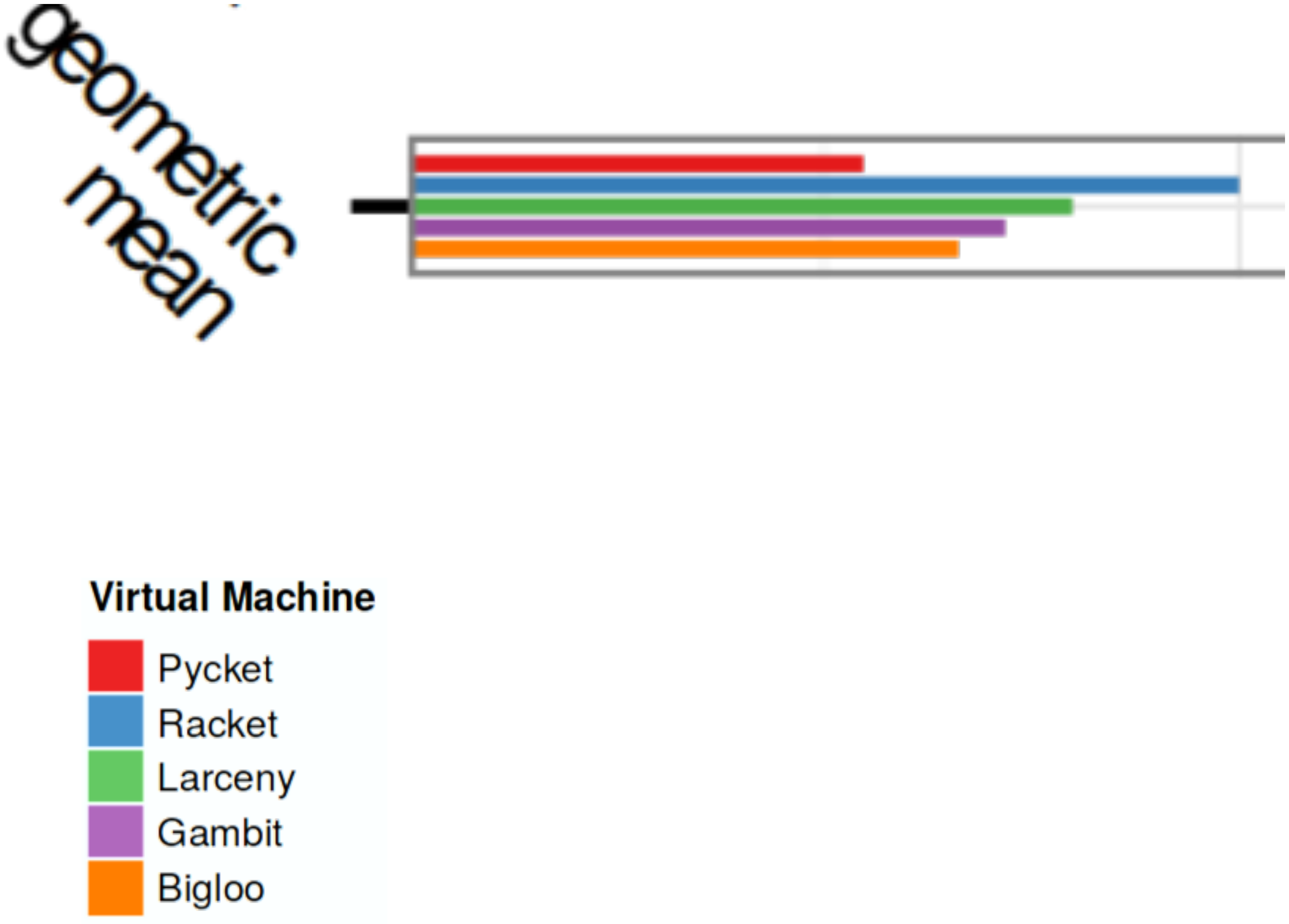
2 Optimizations we don't do

Closure conversion

Pointer tagging

How well does it work?

Scheme benchmarks



	Pycket	±	Pycket*	±	Racket	±	V8	±	PyPy	±
Bubble										
direct	640	1	656	1	1384	4	336	0	593	1
chaperone	768	0	778	1	6668	5				
proxy							105891	2579	1153	8
unsafe	496	1	550	1	955	1				
unsafe*	495	0	508	1	726	1				
Church										
direct	714	2	705	1	1243	6	2145	18	3263	14
chaperone	6079	54	8400	34	38497	66				
contract	1143	6	2310	8	10126	142	295452	1905		
proxy							53953	277	95391	848
wrap	3471	7	3213	5	4214	26	8731	45	59016	405
Struct										
direct	133	0	133	0	527	0	377	0	127	0
chaperone	134	0	134	1	5664	68				
proxy							26268	130	1168	38
unsafe	133	0	133	0	337	0				
unsafe*	133	0	133	0	337	0				
ODE										
direct	2158	6	2645	6	5476	91				
contract	2467	8	5099	8	12235	128				
Binomial										
direct	1439	8	6879	83	2931	24				
contract	17749	83	19288	61	52827	507				

github.com/samth/pycket