What can Scheme learn from JavaScript?

Scheme Workshop 2014

Andy Wingo

Me and Scheme

Guile co-maintainer since 2009

Publicly fumbling towards good Scheme compilers at `wingolog.org`

Scheme rules everything around me

Me and JS

2011: JavaScriptCore ("JSC", in Safari) dabbles (failure, mostly)

2012-2013: V8 (Chrome): random little things, generators, iteration

2013-2014: SpiderMonkey (Firefox): generators, iteration, block scope

Currently V8 (destructuring binding)

(Very little JS coding though!)

Scheme precedes JS

Closures

Specification people (`brendan`, `samth`, `dherman`)

Implementors (e.g. Florian Loitsch, Maciej Stachowiak)

Benchmarks (cross-compiled from Scheme!)

Practitioner language (e.g. continuations)

Scheme precedes JS

Hubris

Scheme precedes JS (?)

Hubris (?)

Scheme precedes JS (?)

Hubris (?)

How could JavaScript precede Scheme?

A brief history of JS

1996-2008: slow

2014: fastish

A brief history of JS

1996-2008: slow

2014: fastish

Environmental forcing functions

Visiting a page == installing an app

Cruel latency requirements

Why care about performance?

Expressiveness a function of speed (among other parameters)

Will programmers express idiom *x* with more or less abstraction?

60fps vs 1fps

Speed limits, expression limits

We sacrifice expressiveness and extensibility when we write fast Scheme

- Late binding vs. inlining
- Mutable definitions vs. static linking
- Top-level vs. nested definitions
- Polymorphic combinators vs. bespoke named let
- Generic vs. specific functions

We are our compilers' proof assistants, and will restrict the problem space if necessary

Lexical scope: the best thing about Scheme

Precise, pervasive design principle

Scope == truth == proof

Happy relationship to speed

Big closed scopes == juicy chunks for an optimizer to munch on

Lexical scope: the worst thing about Scheme

Limit case of big closed scope: Stalin, the best worst Scheme

We contort programs to make definitions lexically apparent, to please our compilers

With Scheme implementations like JS implementations we would write different programs

JS: speed via dynamic proof

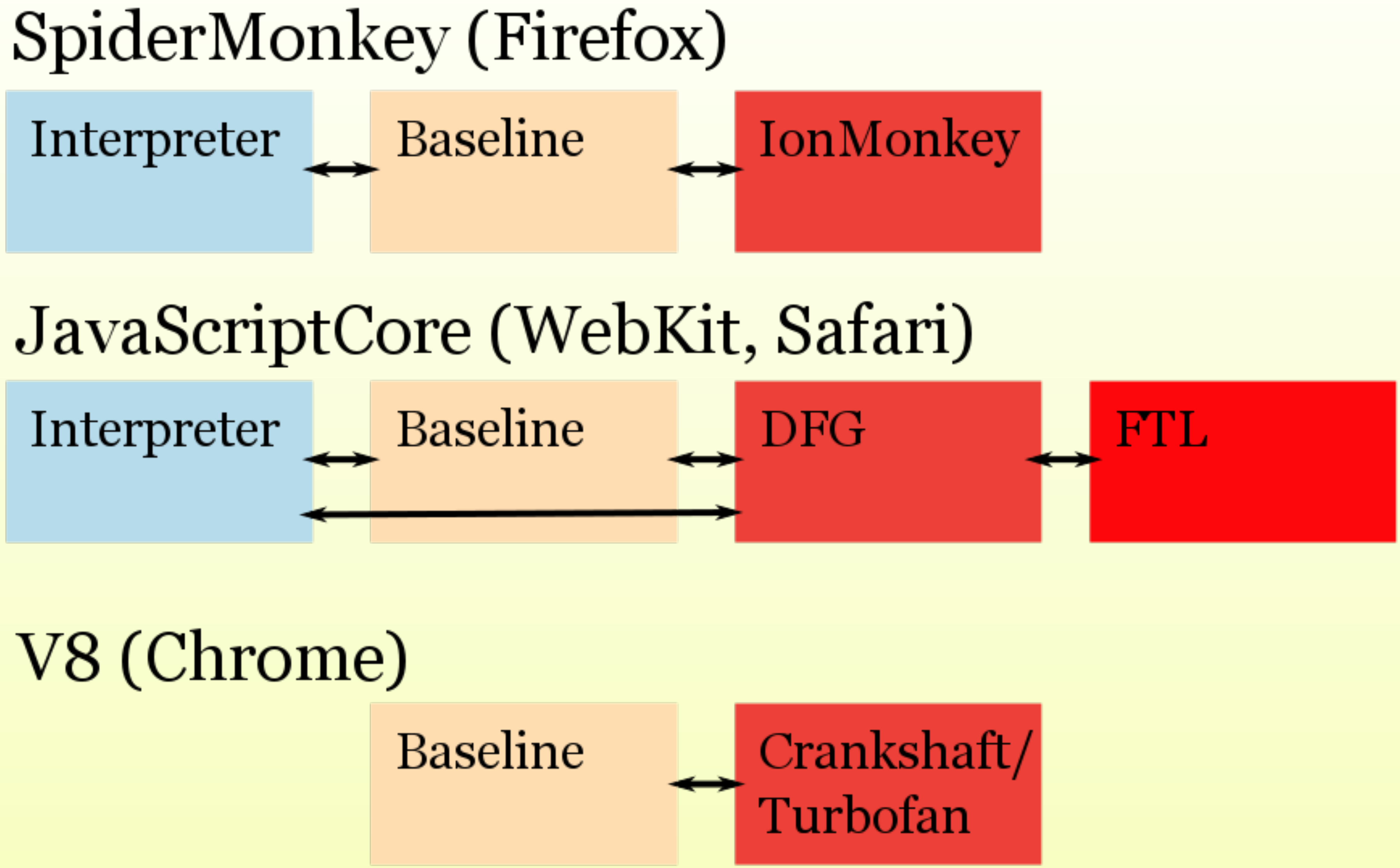
“Adaptive optimization”

A revival of compilation techniques pioneered by Smalltalk, Self, Strongtalk, Java

`expr` `ifTrue:` *`block`*

Inlining key for performance: build sizable proof term

JS contribution: *low-latency* adaptive optimization (fast start)


SpiderMonkey (Firefox)
Interpreter
Baseline
IonMonkey
JavaScriptCore (WebKit, Safari)
Interpreter
Baseline
DFG
FTL
V8 (Chrome)
Baseline
Crankshaft/
Turbofan

All about the tiers

"Method JIT compilers"; Java's HotSpot is canonical comparison

The function is the unit of optimization

Other approaches discussed later; here we focus on method JITs

All about the tiers

Conventional wisdom: V8 needs interpreter

V8 upgrading optimizing compiler

`asm.js` code can start in IonMonkey / Turbofan; embedded static proof pipeline

Optimizing compiler awash in information

Operand and result types

Free variable values

Global variable values

Sets of values: mono-, poly-, mega-morphic

Optimizations: An inventory

Inlining

Code motion: CSE, DCE, hoisting, sea-of-nodes

Specialization

- ❧ Numeric: int32, uint32, float, ...
- ❧ Object: Indexed slot access
- ❧ String: Cons, packed, pinned, ...

Allocation optimization: scalar replacement, sinking

Register allocation

Dynamic proof, dynamic bailout

Compilation is proof-driven term specialization

Dynamic assertions: the future will be like the past

Dynamic assertion failure causes proof invalidation: abort (“bailout”) to baseline tier

Bailout enables static compilation techniques (FTL)

What could Schemers do with adaptive optimization?

Example: `fmt`

```
(fmt #f
     (maybe-slashified "foo"
                       char-whitespace?
                       #\"))
⇒ "foo"
```

Hesitation to use: lots of allocation and no inlining

Compare: Dybvig doing static compilation of `format`

Example: `fmt`

With adapative optimization there would be much less hesitation

If formatting strings is hot, combinators will be dynamically inlined

Closure allocations: gone

Indirect dispatch: gone

Inline string representation details

Example: Object orientation

CLOSsy or not, doesn't matter

```
(define-generic head)
(define-method (head (obj <string>))
  (substring obj 0 1))
(head "hey")
⇒ "h"
```

Lots of indirect dispatch and runtime overhead

Example: Object orientation

If call site is hot, everything can get inlined

Much better than CLOS: optimization happens at call-site, not at callee

(Inline caches)

Example: Dynamic linking

```
(define-module (queue)
  #:use-module (srfi srfi-9)
  #:export (head push pop null))

(define-record-type queue
  (push head tail)
  queue?
  (head head)
  (tail pop))

(define null #f)
```

Example: Dynamic linking

```
(define-module (foo)
  #:use-module (queue))
(define q (push 1 null))
...
```

Observable differences as to whether compiler inlines `push` or not; can the user

- ❧ re-load the `queue` module at run-time?
- ❧ re-link separately compiled modules?
- ❧ re-define the `queue` type?

Example: Dynamic linking

Adaptive optimization enables late binding

Minimal performance penalty for value-level exports

Example: Manual inlining

```
(define-syntax define-effects
  (lambda (x)
    (syntax-case x ()
      ((_ all name ...)
       (with-syntax (((n ...) (iota (length #'(name ...)))))
         #'(begin
             (define-syntax name
               (identifier-syntax (ash 1 (* n 2))))
             ...
             (define-syntax all
               (identifier-syntax (logior name ...)))))))))

(define-effects &all-effects
  &mutable-lexical
  &toplevel
  &fluid
  ...)
```

Stockholm syndrome!

Example: Arithmetic

Generic or specific?

`fl+` or `fx+`?

Adaptive optimizations lets library authors focus on the algorithms and let the user and the compiler handle representation

Example: Data abstraction

http://mid.gmane.org/20111022000312.228558C0903@voluntocracy.org

However, it would be better to abstract this:

```
(define (term-variable x) (car x))
(define (term-coefficient x) (cdr x))
```

That would run slower in interpreters. We can do better by remembering that Scheme has first-class procedures:

```
(define term-variable car)
(define term-coefficient cdr)
```

Example: Data abstraction

Implementation limitations urges programmer to break data abstraction

Dynamic inlining removes these limitations, promotes better programs

Example: DRY Containers

Clojure's iteration protocol versus `map`, `vector-map`, `stream-map`, etc

Generic array traversal procedures (`array-ref` et al) or specific (`vector-ref`, `bytevector-u8-ref`, etc)?

Adaptive optimization promotes generic programming

Standard containers can efficiently have multiple implementations: packed vectors, cons strings

Example: Other applicables

Clojure containers are often applicable:

```
(define v '#(a b c))
(v 1) ⇒ b
```

Adaptive optimization makes different kinds of applicables efficient

Example: Open-coding

```
(define (inc x) (1+ x))
(define + -)
(inc 1) ⇒ ?
```

Example: Debugging

JS programmers expect inlining...

...but also ability to break on any source location

Example: Debugging

Adaptive optimization allows the system to go fast, while also supporting debugging in production

Hölzle's "dynamic de-optimization": tiering down

Caveats

Caveats

There are many

Method JITs: the one true way?

Tracing JITs

- Higgs (`https://github.com/maximecb/Higgs`, experiment)
- TraceMonkey (SpiderMonkey; failure)
- PyPy (mostly for Python; success?)
- LuaJIT (Lua; success)

Use existing VM?

Pycket: Implementation of Racket on top of PyPy (`http://www.ccs.neu.edu/home/samth/pycket-draft.pdf`)

Graal: Interpreter-based language implementation ("One VM to rule them all", Würthinger et al 2013)

Engineering effort

JS implementations: heaps of C++, blah

To self-host Scheme, decent AOT compiler also needed to avoid latency penalty (?)

No production self-hosted adaptive optimizers (?)

Polymorphism in combinators

Have to do two-level inlining for anything good to happen

```
(fold (lambda (a b) (+ a b)) 0 l)
⇒ (let lp ((l l) (seed 0))
    (if (null? l) seed
        (lp (cdr l)
            ((lambda (+ a b) (+ a b))
             (car l)
             seed))))
⇒ (let lp ((l l) (seed 0))
    (if (null? l) seed
        (lp (cdr l) (+ (car l) seed))))
```

Polymorphism in combinators

Polymorphism of call-site in `fold` challenging until `fold` is inlined into caller

Challenging to HotSpot with Java lambdas

Challenging to JS (`Array.prototype.foreach`; note SM call-site cloning hack)

Lack of global visibility

JIT compilation not a panacea

Some optimizations hard to do locally

- Contification
- Stream fusion
- Closure optimization

Tracing mitigates but doesn’t solve these issues

Latency, compiled files, macros

One key JS latency hack: lazy parsing/codegen

Scheme still needs an AOT pass to expand macros

Redefinition not a problem in JS; all values on same meta-level

JS doesn’t have object files; does Scheme need them?

Tail calls versus method jits

JS, Java don’t do tail calls (yet); how does this relate to dynamic inlining and method-at-a-time compilation?

How does it relate to contification, loop detection, on-stack replacement?

Pycket embeds CEK interpreter; loop detection tricky

Things best left unstolen

`undefined`, non-existent property access, sloppy mode, UTF-16, coercion, monkey-patching (or yes?), `with`, big gnarly C++ runtimes, curly braces, concurrency, ...

Next steps?

For Guile:

❧ Native self-hosted compiler

❧ Add inline caches with type feedback cells

❧ Add IR to separate ELF sections

❧ Start to experiment with concurrent recompilation and bailout

For your scheme? Build-your-own or try to reuse Graal/HotSpot, PyPy, …?

For users

Dance like no one is watching

Write lovely Scheme!

For implementors

Steal like no one is watching

Add adaptive optimization to your Schemes!

Thanks

wingo@pobox.com

wingo@igalia.com

http://wingolog.org/

@andywingo