Old Dog, New Tricks

A Schemer's Guide to JavaScript Implementations

Quasiconf 2012

Andy Wingo

wingo@igalia.com

Compiler hacker at Igalia

Contract work on language implementations

V8, JavaScriptCore

Schemer

Scheme in one slide

Parse.

Expand.

Optimize.

Codegen.

Run.

JavaScript in one slide

Pre-parse.

Run.

Parse.

Codegen.

Run.

Optimize.

Codegen.

Run.

...

On optimization

Proof-driven program transformation

Scheme: Static proofs derived from interprocedural flow analysis

JS: Dynamic proofs based on run-time observations, with ability to invalidate transformations if assumptions fail to hold

Barriers to optimization in Scheme

Mutable toplevels

Separate compilation

Incomplete type information

No information on what to inline

`call/cc`

None of these inhibit JavaScript

Assume toplevels are immutable

No real separate compilation

A wealth of type information

Dynamic profiling identifies inline candidates

No `call/cc`

“Adaptive optimization”

Types and hot spots not known until runtime

Types and hot spots can change over time

Assumptions can be invalidated over time

Adaptive optimization: speculative optimization with bailout

Deoptimization for debugging

Allow multi-leveled inlining and code motion while preserving programmer's mental model of how evaluation works

Deoptimization already required by speculation failures

Other common JS optimizations

Unboxing

Common subexpression elimination

Loop invariant code motion (or loop peeling + CSE)

Range inference

Register allocation

Block reordering (?)

But to be clear: dynamic inlining is the big one

Different deployment models

Scheme implementations rarely run attacker- - controlled code

JS: Constant blinding to prevent vulnerabilities in non-JS code from using JS as a heap-spray

No threads in JS

Representation hacks

JS: NaN-boxing, sometimes

Rope strings

Dedicated regexp compilers

Matching word-at-a-time, hard to beat with a general compiler

Lazy tear-off in JSC

A static scope implementation trick

Implementing static scope

Chains: Activations on heap (!)

- Closure creation: O(1) space and time
- Free var access: O(n) time
- Not "safe for space"

Displays: Activations on stack

- Closure creation: O(n) space and time
- Free var access: O(1) time
- Mutated variables usually boxed

Lazy tear-off in JavaScriptCore

Activations on stack

Only allocate scope chain node if closure is captured

When control leaves function, tear off stack to heap, relocating pointer in scope node (no threads in JS)

Memory advantages of chains with stack discipline of display closures

Free var access still O(n) but inlinable

Living with eval

Eval only evil if it defines new locals

```
var foo = 10;

function f(s){ eval(s); return foo; }

f('var foo=20;') ⇒ 20
```

Otherwise great: a compiler available to the user

Functions in which eval appears not fully optimizable: must expose symbol tables

Inline caches

Per-caller memoization, in code

Fundamental optimization for property access

Not as needed in Scheme because not much polymorphism

Can allow efficient generic arithmetic

Can make CLOS-style generics more efficient

Clojure-like sequence protocols

Function application?

Dealing with the devil

Runtime codegen in JS has a price: C++

Most Scheme implementations are self-hosted, with AOT compiler already

Challenge: add adaptive optimization to existing Scheme implementation

Requires good AOT compiler!

JS can change the way we code

Scheme's static implementations encourage static programming

`define-integrable`

`(declare (type fixnum x))`

`(declare (safety 0) (speed 3))`

`(declare (usual-integrations))`

`include` instead of `load`

Adaptive optimization can bring back dynamicity

Summary

In Smart vs Lazy, Schemers always chose Smart

A bit of laziness won't hurt

Adaptive optimization in Scheme!

questions?

- Work: `http://www.igalia.com/compilers`
- Words: `http://wingolog.org/`
- Slides: `http://wingolog.org/pub/qc-2012-js-slides.pdf`
- Notes: `http://wingolog.org/pub/qc-2012-js-notes.pdf`