Interprocedural Type Specialization of JavaScript Programs Without Type Analysis

Maxime Chevalier-Boisvert
joint work with Marc Feeley

ECOOP - July 20th, 2016
Overview

- Previous work: Lazy Basic Block Versioning
  - Single pass JIT code generation technique
  - On-the-fly type-specialization, *intraprocedural* only

- **Three *interprocedural* extensions to BBV:**
  - Typed object shapes
  - Entry point versioning
  - Call continuation specialization

- Prototyped and evaluated in Higgs
  - Experimental JIT for JS, ~60KLOC
Lazy Basic Block Versioning

- JIT code generation technique
  - Fine granularity (basic block)
  - Lightweight, single pass
  - Lazy versioning
- As we compile code, accumulate facts
  - Leverage implicit type checks
- Specialize BBs based on known types
  - May compile multiple versions of blocks
  - Not duplication, but specialization
Dynamic Type Tests

- Focus: eliminating dynamic type checks
  - Dynamic languages, JS in particular
- BBV uses implicit type tests to extract type info
  - Implicit type-dispatch semantics of JS
- Type tests: primitives testing the type of a value
  - e.g. `x + y`
  - `is_int32(x)`: is `x` an integer or not?
Higgs’ Type Tags

In Higgs, all values have an associated type tag

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int32</td>
<td>32-bit integer</td>
</tr>
<tr>
<td>float64</td>
<td>64-bit floating-point value</td>
</tr>
<tr>
<td>undef</td>
<td>JS undefined value</td>
</tr>
<tr>
<td>null</td>
<td>JS null value</td>
</tr>
<tr>
<td>bool</td>
<td>true and false</td>
</tr>
<tr>
<td>string</td>
<td>Immutable JS string</td>
</tr>
<tr>
<td>object</td>
<td>Plain JS object</td>
</tr>
<tr>
<td>array</td>
<td>JS array</td>
</tr>
<tr>
<td>closure</td>
<td>JS function/closure</td>
</tr>
</tbody>
</table>
is_int32(n)?
is_int32(n)?

true

n is int32

false

B

C

D
is_int32(n)?

true

false

n is **int32**

n is **not int32**

B

C

D
is_int32(n)?

true

B

false

C

n is int32

D

n is ???

n is not int32
is_int32(n)?

true

false

n is int32

n is not int32

n is int32

n is not int32

B

C

D'

D''
is_int32(n)?
Lazy Basic Block Versioning

• Compile versions lazily: when first executed
  – Only for types seen at run-time
  – The program's behavior drives versioning
  – Interleave compilation and execution
• Avoid compiling unneeded block versions
  – unexecuted error handling is never compiled
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);

`typeof(s) = int32`
`typeof(i) = int32`
`typeof(n) = unknown`

Lazy BBV
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);

typeof(s) = int32
typeof(i) = int32
typeof(n) = unknown

Lazy BBV

check not eliminated because type of n is unknown in context
function sum(n) {
    for (var s=0, i=0; i<n; i++) s += i;
    return s;
}

sum(500);

Lazy BBV

check not eliminated because type of n is unknown in context
function sum(n) {
  for (var s=0, i=0; i<n; i++) s += i;
  return s;
}

sum(500);

typeof(s) = int32
typeof(i) = int32
typeof(n) = unknown

typeof(s) = int32
typeof(i) = int32
typeof(n) = int32

Lazy BBV

checks eliminated because types are known
Lazy BBV Results (2014)

- *Intraprocedural* lazy BBV (ECOOP 2015)
- 71% of dynamic type tests eliminated
- Measurable speedups, 21% on average
- Small, code size increase, 0.19% average
- But, can we do better?
Interprocedural Extensions
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst)
{
    if (lst == null)
        return 0
    return lst.val + sumList(lst.next)
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
Object Property Types

- Previous work treated objects like black boxes
  - Property types tested *over and over again*
- Global vars are properties of the global object
  - Every global function call involves dynamic tests
    - Trying to call a non-function throws an exception
- Would like to propagate object property types
Shapes aka “Hidden Classes”

Shape nodes:

- A
- B
- C
- D

Property slots:

- 5
- 1.5
- “foo”
- null

Shape pointer:

empty
Typed Shapes

null

string

float64

int32

A
B
C
D

empty

5
1.5
"foo"
null

A
B
C
D
Typed Shapes

• Extend shapes to store property type info
  – Type tags of properties, method identity
• Versioning based on shapes
  – Implicit shape tests extract shape info
• Permits the elimination of:
  – Missing property checks, getter/setter checks
  – Property type checks, boxing/unboxing
  – Dynamic dispatch on function calls
Interprocedural Versioning

- Previous work: intraprocedural BBV
  - Propagates info within function bodies only

- Wasted computations:
  - Objects treated as black boxes
  - Function parameters treated as unknown types
  - Return values treated as unknown types
  - Losing and re-testing value types

- Costly, particularly for recursive functions
Entry Point Specialization

- Most argument types are known at call sites
- Goal: pass arg types to callee entry points
- Key: typed shapes give us identity of callees
- Generate specialized function entry points
  - Easy: specialize the function entry blocks
  - Jump directly to specialized entry
- When callee unknown, use generic entry
  - Rare in practice and no worse than before
Call Continuation Specialization

- **Intraprocedural**: test ret value type at each call
  - Wasting cycles even when ret type is constant
- Would like to propagate ret types somehow
- Can't apply same strategy as entry point spec
  - Calls and returns are asymmetric
  - Most call sites are monomorphic (one callee)
  - Most functions have multiple callers
Speculative Optimization

• Issue: cost of testing return type is small
  – It's just one dynamic type test
• Would like to pass return type info with zero dynamic overhead
  – Avoid dynamic dispatch when returning
• Speculate that return types remain constant
  – Specialize call continuations in consequence
  – Invalidate continuations when ret types change
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst)
{
    if (lst == null)
        return 0
    return lst.val +
        sumList(lst.next)
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst)
{
    if (lst == null)
        return 0
    return lst.val +
        sumList(lst.next)
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst) // ret type int32
{
    if (lst == null)
        return 0             // int32
    return lst.val +
        sumList(lst.next)
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst) // ret type int32
{
    if (lst == null)
        return 0             // int32
    return lst.val +
        sumList(lst.next) // int32
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst) // ret type int32
{
    if (lst == null)
        return 0 // int32
    return lst.val + // int32
        sumList(lst.next) // int32
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst) // ret type int32
{
    if (lst == null)
        return 0             // int32
    return lst.val +         // int32
        sumList(lst.next)     // int32
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
function makeList(len)
{
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

function sumList(lst)
{
    if (lst == null)
        return 0
    return lst.val +
            sumList(lst.next)
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect sum')
Experimental Results
Evaluation Methodology

- Benchmarks: 26 programs from SunSpider & V8 bench
- Compared results with plain intraprocedural BBV
  - BBV + typed shapes
  - BBV + entry point versioning
  - BBV + entry point versioning + cont spec
- Metrics:
  - Type checks eliminated (precision/accuracy)
  - Execution time, compilation time
  - Total machine code size generated
  - Callee identity known (dynamic)
- Interprocedural BBV vs static type analysis
- Higgs vs commercial JavaScript VMs
Evaluation Summary

- Callee identity known for 97.5% of calls
- Return type propagated 72% of the time
- Dynamic type tests: 94.3% eliminated (vs 71%)
- Compared to intraprocedural BBV
  - Code size: +5.5% worst case
  - Compilation time: +3.7% worst case
  - Execution time: -37.6% on average
Percentage of dynamic type tests eliminated (higher is better)
Type tests eliminated, BBV vs simulated perfect analysis (higher is better)
Commercial JavaScript VMs

- Benchmarked Higgs against TraceMonkey, SpiderMonkey, V8 and Truffle/JS
- Disclaimer: Higgs lacks many opts found in commercial VMs
  - Stop-the-world, single generation copying GC
  - No LICM, GVN
  - No SIMD auto-vectorization
  - No bounds check elimination, inefficient array impl
  - No method inlining
  - No escape analysis or allocation sinking
  - On the fly register allocation, floats in GPRs (lol)
Comparative Performance

- **Mozilla TraceMonkey (1.8.5+, 2011, last pre-retired)**
  - Higgs is 2.7x faster on average
  - Higgs outperforms TM on 22/26 benchmarks
- **Oracle’s Truffle/JS (v0.9, 2015)**
  - 1000 warmup itrs: 0.69x as fast as Truffle/JS
  - No warmup: 2.2x as fast as Truffle/JS
- **Mozilla SpiderMonkey (C40.0a1, 2015)**
  - 0.37x as fast on average
  - Outperforms SM on 1/26
- **Google’s V8 (3.29.66, 2015)**
  - 0.47x as fast on average
  - Outperforms V8 on 3/26
Future Work

• BBV extensions
  – Closure variable types
  – Array element types
• Lazy incremental inlining
  – Natural way to inline with BBV
  – Partially inlining callees
  – Inlining without recompilation
• Optimizing Scheme code
  – Saleil & Feeley
Applications

- Baseline JIT
  - V8 uses its baseline JIT to gather type info
  - More precise information for optimizing JIT
- Reduce gradual typing overhead
- Static (AOT) program analysis
- Dynamic Binary Translation (DBT)
Conclusion

- Three interprocedural extensions:
  - Typed shapes, useful for JS & OO languages
  - Entry point & call continuation specialization
- Results are promising
  - 94.3% of dynamic type tests eliminated
  - More than a “perfect” static analysis
  - Large improvement over intraprocedural BBV
- Many possible extensions and applications
My ECOOP 2015 BBV talk is on YouTube
www.youtube.com/watch?v=S-aHBuoiYE0

github.com/maximecb/Higgs

maximechevalierb@gmail.com

Love2Code on twitter