Higgs

A monitoring JIT for JavaScript

Maxime Chevalier-Boisvert

Dynamic Language Team

Université de Montréal

Higgs

- Tracing JIT for JavaScript
  - Written in D + JS
  - Second iteration of Tachyon compiler
  - Simpler, more straightforward design
  - More dynamic optimization strategy
- 3 main components:
  - Interpreter
  - Type monitoring system
  - Tracing JIT
- Current status:
  - ES5 interpreter complete, GC complete
  - Type monitoring implementation started

Previous Work: Tachyon

- Method-based JS compiler
- Compiles JS down to x86/x86-64
- Compiler itself written in (extended) JS
  - Even the assembler
- Self-hosting: Tachyon can compile itself
- Goal: static analysis + dynamic reoptimization
  - Similar to Brian Hackett's type inference work

Previous Work: Type Analysis

- Goal: more accuracy, reasonable cost for server-side JIT or offline compilation
- Path-sensitive type analysis of JS
- Recency types (Altucher & Landi, POPL 95)
- Decoupled fixed-point algorithm
- Results:
  - Accuracy close to state of the art
  - Some cases very hard to analyze without context
  - Cost still fairly high, even for offline compilers
- My conclusion:
  - A simpler and more dynamic strategy is needed

Higgs: Goals and Non-Goals

- Goal:
  - Demonstrate soundness and effectiveness of novel compiler architecture and dynamic language optimizations
- Non-goals:
  - Supporting every JS program
  - Competing on parsing/compilation speed
  - Beating V8/FF on every benchmark
- Simplifying assumptions
  - For now, Higgs targets x86-64 only
  - Targets programs with medium-long running-times
    - e.g.: games, server-side environments

Why a tracing JIT?

- Have tracing JITs gone out of fashion?
  - V8, IonMonkey are method-based
  - Are method-based JITs better?
- Important advantages:
  - Simple design
  - Incremental construction
  - Inlining comes “for free”
  - Code invalidation (architectural elegance?)
  - Bonus: fast compilation

The Higgs Interpreter

- Reference implementation and JIT fallback
- Fast prototyping: simplicity, extensibility prioritized over speed
- Implements register-based VM + GC
- Interprets a low-level IR
- JS primitives implemented in extended JS
- GC is semi-space, stop-the-world, copying
- Runtime, stdlib, GC ported over from Tachyon

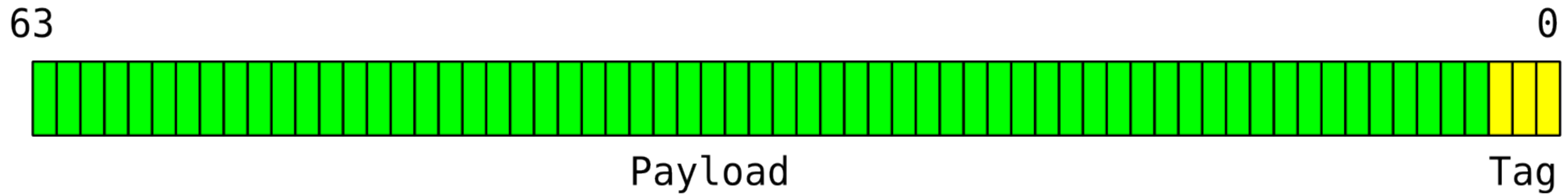
Why a low-level IR?

- Simplifies the interpreter
  - Deals with simple, low-level ops
    - e.g.: imul, fmul, load, store, call, ret
  - Knows little about JS semantics
- Simplifies the JIT
  - Less duplicated functionality in interpreter and JIT
  - Avoids implicit dynamic dispatch in IR ops
    - e.g.: the + operator in JS has lots of implicit branches!
- JS primitives of runtime are JS functions
  - Tracing through high-level opcodes is problematic
  - Similar idea to Tamarin-Tracing design (Forth opcodes)

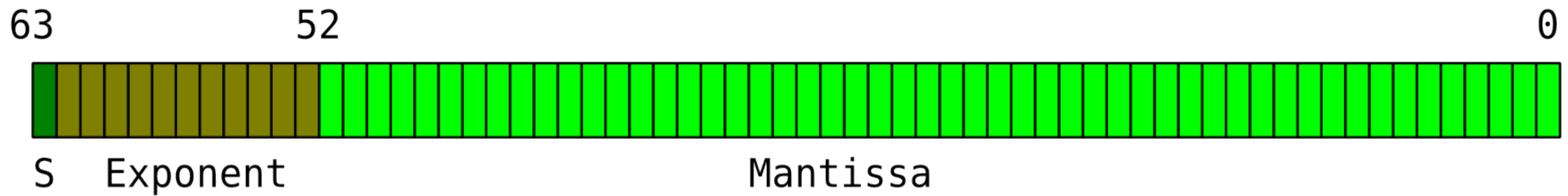
Double-Word Tagging

- Higgs uses double-word tagging of values
  - No tag bits, no NaN tagging
  - One value word (64-bit) + one type tag byte
- Downside: size, two stack pointers, two arrays
- Upsides:
  - Values accessible directly, no unboxing
  - Modern CPUs have multiple execution units
  - In many cases, can completely ignore type info
  - JIT performance favored over interpreter performance

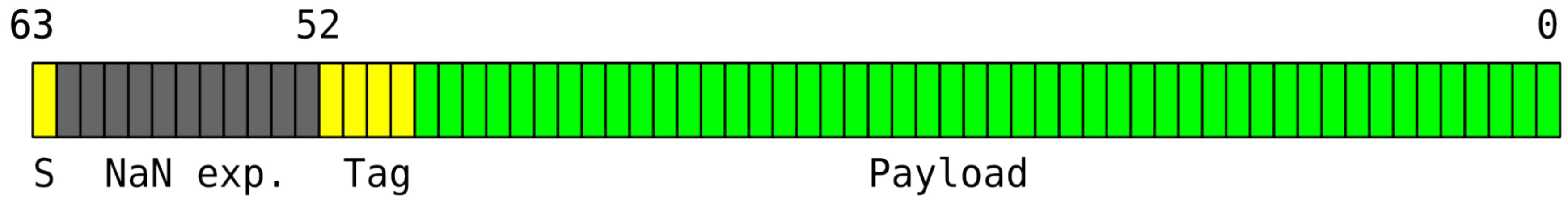
Tag Bit Scheme



IEEE Double

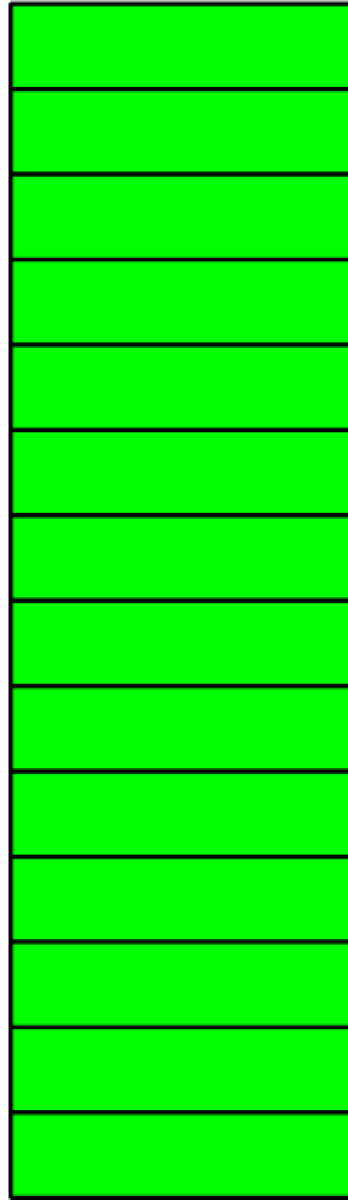
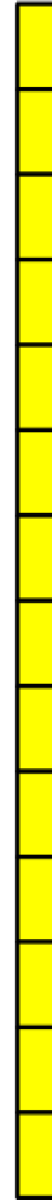


NaN Boxing



Double-Word Tagging

Data Stack Type Stack

wsp →   ← tsp

Redefinable Runtime Library

- Runtime functions are redefinable at run-time
- Allows for things like operator overloading
- Allows for things like *load('runtime.js')*
- *Side effect of simpler design*
  - Runtime functions are like any other JS function
- We can optimize away the cost
  - Inlined like any other function

Profiling

- Unlike a static compiler, an interpreter or VM can observe a program's execution
- Can gather useful info for optimization
  - Programs tend to have repetitive behaviors
- Profiling incurs cost: cost/accuracy tradeoff
- In practice, modern VMs do statistical profiling
  - e.g.: using inline caches to gather type profile
  - e.g.: approximate call graph construction

Monitoring

- Opinion: modern VMs still don't make effective use of opportunities afforded by profiling
- An interpreter can *fully* observe a program's execution
  - *Any* property of an executing program can be observed
  - Massive amounts of data are available
- Thought experiment: if you were to pause a program's execution, you could gather the current types of *all* variables and object fields
- Monitoring is non-statistical profiling
  - Type monitoring: gather a fully accurate type profile.

Monitoring in Higgs

- Interpreter *monitors* the types of all object fields by recording all types written using special monitoring instructions
- Tracing JIT uses type profile to compile optimized traces relying on type observations
- If type optimizations become invalidated, whole traces can be invalidated
  - Easy: nullify trace pointer, exit trace if needed

Gambling with Types

- The system should be designed so that most type optimizations will not be invalidated
  - Can maximize the chance of this by making smarter optimization choices (heuristically)
  - Can avoid repeatedly making the same mistakes
- Brian Hackett showed some amount of invalidation/recompilation is not catastrophic
  - Recompilation probability tends to tail off with time

Overview of the Higgs Model

- Program execution begins
- Interpreter builds type profile through monitoring
- Interpreter records “hot” traces
- Traces passed to optimizer
  - Makes observations based on type profile
  - Simplifies/optimizes recorded traces
- Machine code generation
- Interpreter branches to compiled trace code
- Monitoring continues, potentially invalidating traces

```
function init()
{
    // Initialization of an array with integer values
    arr = new Array(1000);
    for (var i = 0; i < arr.length; ++i)
        arr[i] = i;
    return arr;
}

// Code operating on the array. This is the part
// of the program we will specifically try to optimize.
//
// Optimizing more complex examples such as matrix
// multiplication and FFT involves similar challenges.
function compute(arr)
{
    sum = 0;
    for (var i = 0; i < arr.length; ++i)
        sum += arr[i];

    return sum;
}
```

IR of the *compute* Function

```
COMPUTE:
sum = 0;
i = 0;

LOOP_TEST:
n = getProp(arr, 'length'); // l = a.length
t = ge(i, n);
if t goto LOOP_END          // while (i < a.length)

LOOP_BODY:
v = getProp(arr, i);
sum = add(sum, v);          // sum += arr[i]
i = add(i, 1)               // i += 1
goto LOOP_TEST

LOOP_END:
return sum
```

Recording Traces

```
COMPUTE:
sum = 0;
i = 0;

LOOP_TEST:                        // Trace recording begins here
n = getProp(arr, 'length');
t = ge(i, n);
if t goto LOOP_END

LOOP_BODY:
v = getProp(arr, i);
sum = add(sum, v);
i = add(i, 1)
goto LOOP_TEST                    // This backwards branch can
                                  // trigger trace recording if
LOOP_END:                         // executed often enough
```

Primitive Calls

```
COMPUTE:
sum = 0;
i = 0;

LOOP_TEST:
n = getProp(arr, 'length');    // Primitive calls will be
t = ge(i, n);                  // inlined into the trace
if t goto LOOP_END

LOOP_BODY:
v = getProp(arr, i);
sum = add(sum, v);
i = add(i, 1)
goto LOOP_TEST

LOOP_END:
```

```
//LOOP_TEST:

    //n = getProp(a, 'length');
    if !is_array (a)                => exit trace
    n = get_array_len(a)

    //t = ge(i, n);
    if !is_int(i)                   => exit trace
    if !is_int(n)                   => exit trace
    t = ge_int32(i, n)

    if t goto LOOP_END

//LOOP_BODY:

    //v = getProp(a, i);
    if !is_array (a)                => exit trace
    if !is_int(i)                   => exit trace
    v = get_array_elem(a, i)

    //sum = add(sum, v);
    if !is_int(sum)                 => exit trace
    if !is_int(v)                   => exit trace
    sum = add_int32(sum, v)
    if int_overflow                 => exit trace

    //i = add(i, 1)
    if !is_int(i)                   => exit trace
    i = add_int32(i, 1)
    if int_overflow                 => exit trace

    goto LOOP_TEST                 // Return to the trace head
```

```
//LOOP_TEST:

    //n = getProp(a, 'length');
    if !is_array (a)            => exit trace
    n = get_array_len(a)

    //t = ge(i, n);
    if !is_int(i)               => exit trace
    if !is_int(n)               => exit trace
    t = ge_int32(i, n)

    if t goto LOOP_END

//LOOP_BODY:

    //v = getProp(a, i);
    if !is_array (a)            => exit trace
    if !is_int(i)               => exit trace
    v = get_array_elem(a, i)

    //sum = add(sum, v);
    if !is_int(sum)             => exit trace
    if !is_int(v)               => exit trace
    sum = add_int32(sum, v)
    if int_overflow             => exit trace

    //i = add(i, 1)
    if !is_int(i)               => exit trace
    i = add_int32(i, 1)
    if int_overflow             => exit trace

    goto LOOP_TEST              // Return to the trace head
```

Trace Optimization

- Through monitoring, we can <u>observe</u> that
  - *arr* is an array of integers at the entry to *compute*
  - The length of *arr* is 1000
  - Integers in *arr* reside in range [0, 999]
- From JS semantics, we have that:
  - *i*, *sum* are integer values at initialization
  - The result of int+int is int
  - Array lengths are always Uint32
- By type propagation, we can infer that:
  - *i*, *sum* will remain Uint32 throughout their lifetime

```
//LOOP_TEST:

    //l = getProp(a, 'length');
    //if !is_array (a) => exit        We know a is an array on trace entry
    n = get_array_len(a)

    //t = ge(i, n);
    //if !is_int(i) => exit           We know i is integer on trace entry
    //if !is_int(n) => exit           Array lengths are always UInt32
    t = ge_uint32(i, n)

    if t goto LOOP_END

//LOOP_BODY:

    //v = getProp(a, i);
    //if !is_array (a) => exit        As before
    //if !is_int(i) => exit           As before
    v = get_array_elem(a, i)

    //sum = add(sum, v);
    //if !is_int(sum) => exit         Type known before trace
    //if !is_int(v) => exit           from monitoring, know v to be int
    sum = add_int32(sum, v)           // Type of v does not change here
    if int_overflow => exit trace

    //i = add(i, 1)
    //if !is_int(i) => exit           As before
    i = add_int32(i, 1)               // Type of i does not change here
    if int_overflow => exit           Because i < 0xFFFFFFFF

    goto LOOP_TEST                    // Return to the trace head
```

Invalidation

- When traces make type observations, there is a danger: the type profile could change
- In our example, should someone store a string in *arr*, the type of array elements would change
  - e.g.: arr[0] = “foo”
  - *arr* now contains both strings and integers
- If this happens, optimized code should be invalidated (easy in a tracing JIT)
- Code only reoptimized if called again

The Cost of Monitoring

- There's a problem with monitoring
- Need to maintain a fully-accurate type profile
- Must account for all property writes
  - Invalid optimizations will cause bugs
- Recording every property write would be slow
- Must save time, but still account for everything

```
p = { x:1.5, y:1.5 } // 2D point
v = { x:0.2, y:0.3 } // Velocity vector

// Move our point in function of time
for (;;)
{
    dT = deltaTime()
    p.x = p.x + v.x * dT // Don't want to check
    p.y = p.y + v.y * dT // both of these writes!
}
```

```
p = { x:1.5, y:1.5 } // x, y: float
v = { x:0.2, y:0.3 } // x, y: float

// Move our point in function of time
for (;;)
{
    dT = deltaTime()      // dT: float
    p.x = p.x + v.x * dT // x = float + float * float
    p.y = p.y + v.y * dT // y = float + float * float
}
```

Monitoring Cleverly

- Don't have to record *every* property write
  - Once we make an observation about a type, only care to know that this observation remains valid
  - Only need to monitor writes that may violate past observations
  - Observations provide information we can use to optimize (and remove) monitoring code
- Building a walled garden: type observations are used justify the validity of other observations
  - Minimal set of checks to guarantee safety

Type Inference

- How is the design of Higgs different from Mozilla's type inference?
  - Higgs does not do global type analysis
  - No fixed-point, no complex TI algorithm
- Higgs does linear type propagation along traces
  - Like constant propagation, simple algorithm
- Access to potentially more accurate information than global type analysis can provide
  - Type analyses are forced to be conservative
  - Imprecisions slip into the model
  - May overgeneralize type sets

Continuous Optimization

- Programs may perform different tasks at different times
- Long-running programs may be worth re-optimizing
- e.g.: GIMP-like image editor, operates on different image formats
  - Want to re-optimize image-processing kernels
- Could rebuild type profiles, partially or entirely
  - e.g.: scan the heap, like garbage collection

Conclusion

- Higgs will use monitoring to extract accurate type information from programs
- Goal: optimize programs to a greater extent than otherwise achievable
- Cost of accurate type profiling can be offset (optimized away) by the JIT
- Higgs makes interesting technical choices along the way, explores the space of JIT design possibilities

Visiting Mozilla

- Visiting Mountain View office until Feb 15th
- Here to learn about Mozilla JIT technology
  - Optimizations done (both high and-low-level)
  - Specifics of your tracing JIT, how to do this well
  - Catching details I might be overlooking
- If you'd like to discuss compilers, programming languages or microcontrollers let's talk
  - e.g.: /((food|beer) talk)+/

github.com/maximecb/Higgs

maximechevalierb@gmail.com

maximecb on #ionmonkey

pointersgonewild.wordpress.com

Love2Code on twitter