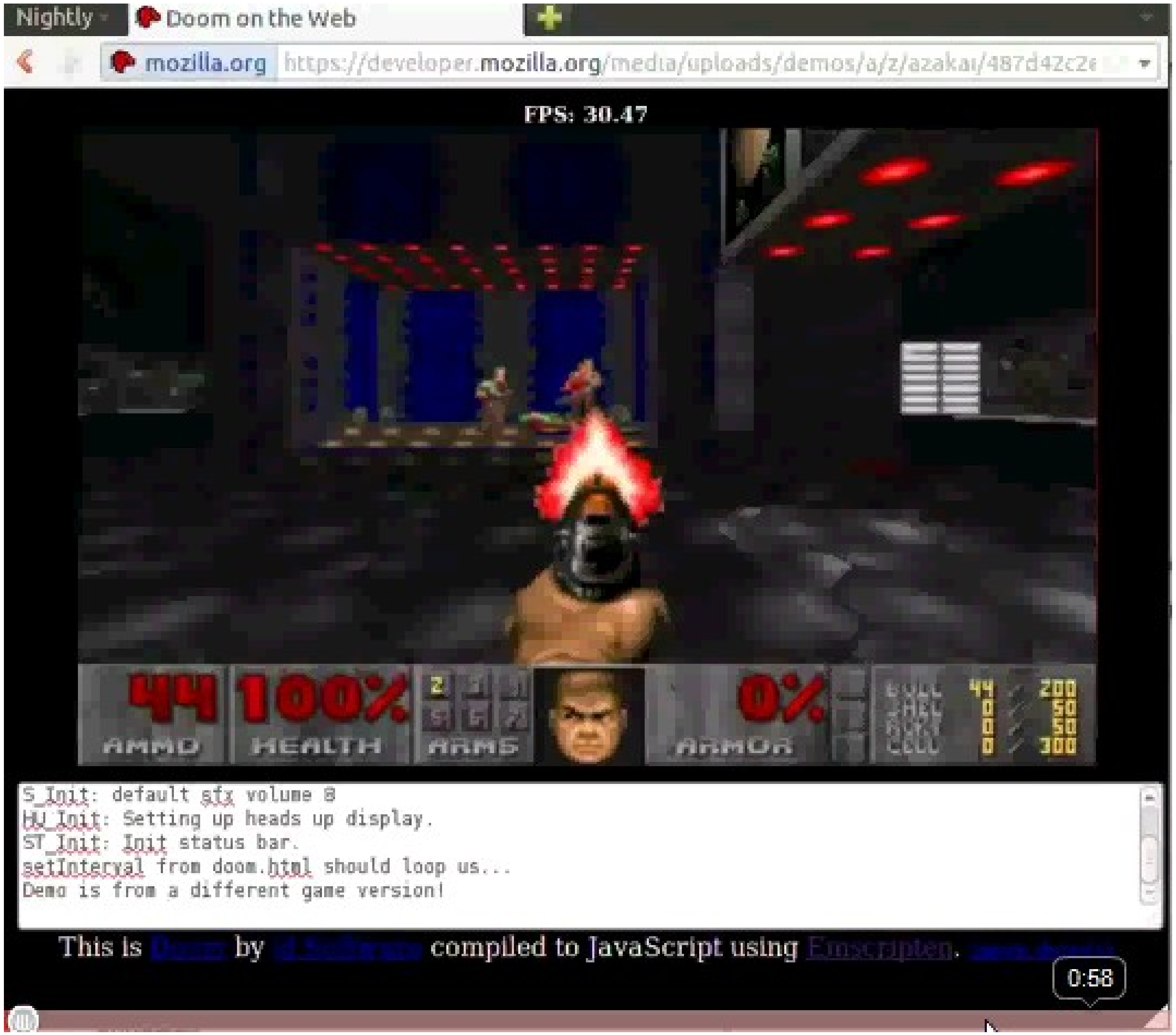
JavaScript Performance: JITs in Firefox

Peformance & Genericity Seminar
LRDE

Nicolas Pierron

Power of the Web



Outline

- What is Javascript.
- Javascript Implementations.
  - Values (Objects, strings, numbers)
  - Running Javascript
    - Interpreter (SpiderMonkey)
    - Method JIT (JägerMonkey)
    - Adaptive JIT (IonMonkey)

Core Web Components

- HTML provides content
- CSS provides styling
- JavaScript provides a programming interface

What is JavaScript?

- C-like syntax
- De facto language of the web
- Interacts with the DOM and the browser

C-Like Syntax

- Braces, semicolons, familiar keywords

```
if (x) {
    for (i = 0; i < 100; i++)
        print("Hello!");
}
```

Think LISP or Self, not Java

- Untyped - no type declarations
- Multi-Paradigm – objects, closures, first-class functions
- Highly dynamic - objects are dictionaries

No Type Declarations

- Properties, variables, return values can be anything:

```
function f(a) {
    var x = "string";
    if (a)
        x = a + 72.3;
    return x;
}
```

```
if a is:
    number: add;
    string: concat;
```

Closure

- Functions may be returned, passed as arguments:

```
function f(a) {
    return function () {
        return a;
    }
}
var m = f(5);
print(m());
```

Objects

- Objects are dictionaries mapping keys (strings) to values
- Properties may be deleted or added at any time!

```
var point = { x : 5, y : 10 };
delete point.x;
point.z = 12;
```


mozilla

Prototypes

- Every object can have a prototype object
- If a property is not found on an object, its prototype is searched instead
- … And the prototype’s prototype, etc..

```
var proto = { key : "value" };
var object = { __proto__ : proto };
print(object.key);
```

Numbers

- JavaScript specifies that numbers are IEEE-754 64-bit floating point
  - 1 bit: sign
  - 11 bits: exponent

Numbers

- Engines use 32-bit integers to optimize
- Must preserve semantics: integers overflow to doubles

```
var x = 0x7FFFFFFF;
x++;
```

```
int x = 0x7FFFFFFF;
x++;
```

	JavaScript	C++
x	2147483648	-2147483648

mozilla

Outline

- What is Javascript.
- Javascript Implementations.
  - Values (Objects, strings, numbers)
  - Running Javascript
    - Interpreter (SpiderMonkey)
    - Method JIT (JägerMonkey)
    - Adaptive JIT (IonMonkey)

Values

- The runtime must be able to query a value’s type to perform the right computation.
- When storing values to variables or object fields, the type must be stored as well.

Values

	Boxed	Unboxed
Purpose	Storage	Computation
Examples	(INT32, 9000)	(int)9000
	(STRING, “hi”)	(String *) “hi”
	(DOUBLE, 3.14)	(double)3.14
Definition	(Type tag, C++ value)	C++ value

- Boxed values required for variables, object fields
- Unboxed values required for computation

Boxed Values

- Need a representation in C++
- Easy idea: 96-bit struct
  - 32-bits for type
  - 64-bits for double, pointer, or integer
- Too big! We have to pack better.

Boxed Values

- We could use pointers, and tag the low bits
- Doubles would have to be allocated on the heap
  - Indirection, GC pressure is bad
- There is a middle ground...

Nunboxing

IA32, ARM

- Values are 64-bit
- Doubles are normal IEEE-754
- How to pack non-doubles?
- 51 bits of NaN space!

Nunboxing

IA32, ARM

Type	Payload
`0x400c0000`	`0x00000000`

63 … 0

- Full value: 0x400c000000000000
- Type is double because it's not a NaN
- Encodes: (Double, 3.5)

Nunboxing

IA32, ARM

Type	Payload
`0xFFFF0001`	`0x00000040`

63 … 0

- Full value: `0xFFFF000100000040`
- Value is in NaN space
- Value is (Int32, `0x00000040`)

Punboxing

x86-64

63 Type	47 Payload 0
1111 1111 1111 1111 1	000 .. 0000 0000 0000 0000 0100 0000

- Full value: `0xFFFF800000000040`
- Value is in NaN space
- Bits >> 47 == `0x1FFFF` == INT32
- Bits 47-63 masked off to retrieve payload
- Value(Int32, `0x00000040`)

	Nunboxing	Punboxing
Fits in register	**NO**	**YES**
Trivial to decode	**YES**	**NO**
Portability	32-bit only*	x64 only

** Some 64-bit OSes can restrict mmap() to 32-bits*

Outline

- What is Javascript.
- Javascript Implementations.
  - Values (Objects, strings, numbers)
  - Running Javascript
    - Interpreter (SpiderMonkey)
    - Method JIT (JägerMonkey)
    - Adaptive JIT (IonMonkey)

Running JavaScript

- Interpreter – Runs code, not fast
- Basic JIT – Simple, untyped compiler
- Advanced JIT – Typed compiler

Interpreter

- Good for code that runs once
- Giant switch loop
- Handles all edge cases of JS semantics

Interpreter

```
while (true) {
    switch (*pc++) {
        case OP_ADD:
            ...
        case OP_SUB:
            ...
        case OP_RETURN:
            ...
    }
}
```

Interpreter

```
case OP_ADD: {
    Value lhs = POP();
    Value rhs = POP();
    Value result;
    if (lhs.isInt32() && rhs.isInt32()) {
        int left = rhs.toInt32();
        int right = rhs.toInt32();
        if (AddOverflows(left, right, left + right))
            result.setInt32(left + right);
        else
            result.setNumber(double(left) + double(right));
    } else if (lhs.isString() || rhs.isString()) {
        String *left = ValueToString(lhs);
        String *right = ValueToString(rhs);
        String *r = Concatenate(left, right);
        result.setString(r);
    } else {
        double left = ValueToNumber(lhs);
        double right = ValueToNumber(rhs);
        result.setDouble(left + right);
    }
    PUSH(result);
    break;
```


mozilla

Interpreter

- Very slow!
  - Lots of opcode and type dispatch
  - Lots of interpreter stack traffic
- Just-in-time compilation solves both

Just In Time

- Compilation must be very fast
  - Should not introduce noticeable pauses
- People care about memory use
  - Should not JIT everything
  - Should not create bloated code
  - May have to discard code at any time

Outline

- What is Javascript.
- Javascript Implementations.
  - Values (Objects, strings, numbers)
  - Running Javascript
    - Interpreter (SpiderMonkey)
    - Method JIT (JägerMonkey)
    - Adaptive JIT (IonMonkey)

Basic JIT

- JägerMonkey in Firefox 4
- Every opcode has a hand-coded template of assembly (registers left blank)
- Method at a time:
  - Single pass through bytecode stream!
  - Compiler uses assembly templates corresponding to each opcode

Bytecode

```
function Add(x, y) {
   return x + y;
}
```

→

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
RETURN
```

Interpreter

```
case OP_ADD: {
    Value lhs = POP();
    Value rhs = POP();
    Value result;
    if (lhs.isInt32() && rhs.isInt32()) {
        int left = rhs.toInt32();
        int right = rhs.toInt32();
        if (AddOverflows(left, right, left + right))
            result.setInt32(left + right);
        else
            result.setNumber(double(left) + double(right));
    } else if (lhs.isString() || rhs.isString()) {
        String *left = ValueToString(lhs);
        String *right = ValueToString(rhs);
        String *r = Concatenate(left, right);
        result.setString(r);
    } else {
        double left = ValueToNumber(lhs);
        double right = ValueToNumber(rhs);
        result.setDouble(left + right);
    }
    PUSH(result);
    break;
```


mozilla

Assembling ADD

- Inlining that huge chunk for every ADD would be very slow
- Observation:
  - Some input types much more common than others

Integer Math – Common

```
var j = 0;
for (i = 0; i < 10000; i++) {
    j += i;
}
```

Weird Stuff – Rare!

```
var j = 12.3;
for (i = 0; i < 10000; i++) {
    j += new Object() + i.toString();
}
```

Assembling ADD

- Only generate code for the easiest and most common cases
- Large design space
  - Can consider integers common, or
  - Integers and doubles, or
  - Anything!

Basic JIT - ADD

```
if (arg0.type != INT32)
    goto slow_add;
if (arg1.type != INT32)
    goto slow_add;
```

Basic JIT - ADD

```
if (arg0.type != INT32)
    goto slow_add;
if (arg1.type != INT32)
    goto slow_add;
R0 = arg0.data
R1 = arg1.data
```

Register Allocator

Basic JIT - ADD

```
if (arg0.type != INT32)
    goto slow_add;
if (arg1.type != INT32)
    goto slow_add;
R0 = arg0.data;
R1 = arg1.data;
R2 = R0 + R1;
if (OVERFLOWED)
    goto slow_add;
```

Slow Paths

```
slow_add:
    Value result = runtime::Add(arg0, arg1);
```

Inline

```
  if (arg0.type != INT32)
      goto slow_add;
  if (arg1.type != INT32)
      goto slow_add;
  R0 = arg0.data;
  R1 = arg1.data;
  R2 = R0 + R1;
  if (OVERFLOWED)
      goto slow_add;
rejoin:
```

Out-of-line

```
slow_add:
    Value result = runtime::Add(arg0, arg1);
    R2 = result.data;
    goto rejoin;
```

Rejoining

Inline

```
if (arg0.type != INT32)
    goto slow_add;
if (arg1.type != INT32)
    goto slow_add;
R0 = arg0.data;
R1 = arg1.data;
R2 = R0 + R1;
if (OVERFLOWED)
    goto slow_add;
R3 = TYPE_INT32;
rejoin:
```

Out-of-line

```
slow_add:
    Value result = runtime::Add(arg0, arg1);
    R2 = result.data;
    R3 = result.type;
    goto rejoin;
```

Final Code

Inline

```
  if (arg0.type != INT32)
     goto slow_add;
  if (arg1.type != INT32)
     goto slow_add;
  R0 = arg0.data;
  R1 = arg1.data;
  R2 = R0 + R1;
  if (OVERFLOWED)
     goto slow_add;
  R3 = TYPE_INT32;
rejoin:
  return Value(R3, R2);
```

Out-of-line

```
slow_add:
   Value result = runtime::Add(arg0, arg1);
   R2 = result.data;
   R3 = result.type;
   goto rejoin;
```

Basic JIT – Object Access

```
function f(x) {
    return x.y;
}
```

- x.y is multiple dictionary searches!
- How can we create a fast-path?
  - We could try to cache the lookup, but
  - We want it to work for >1 objects

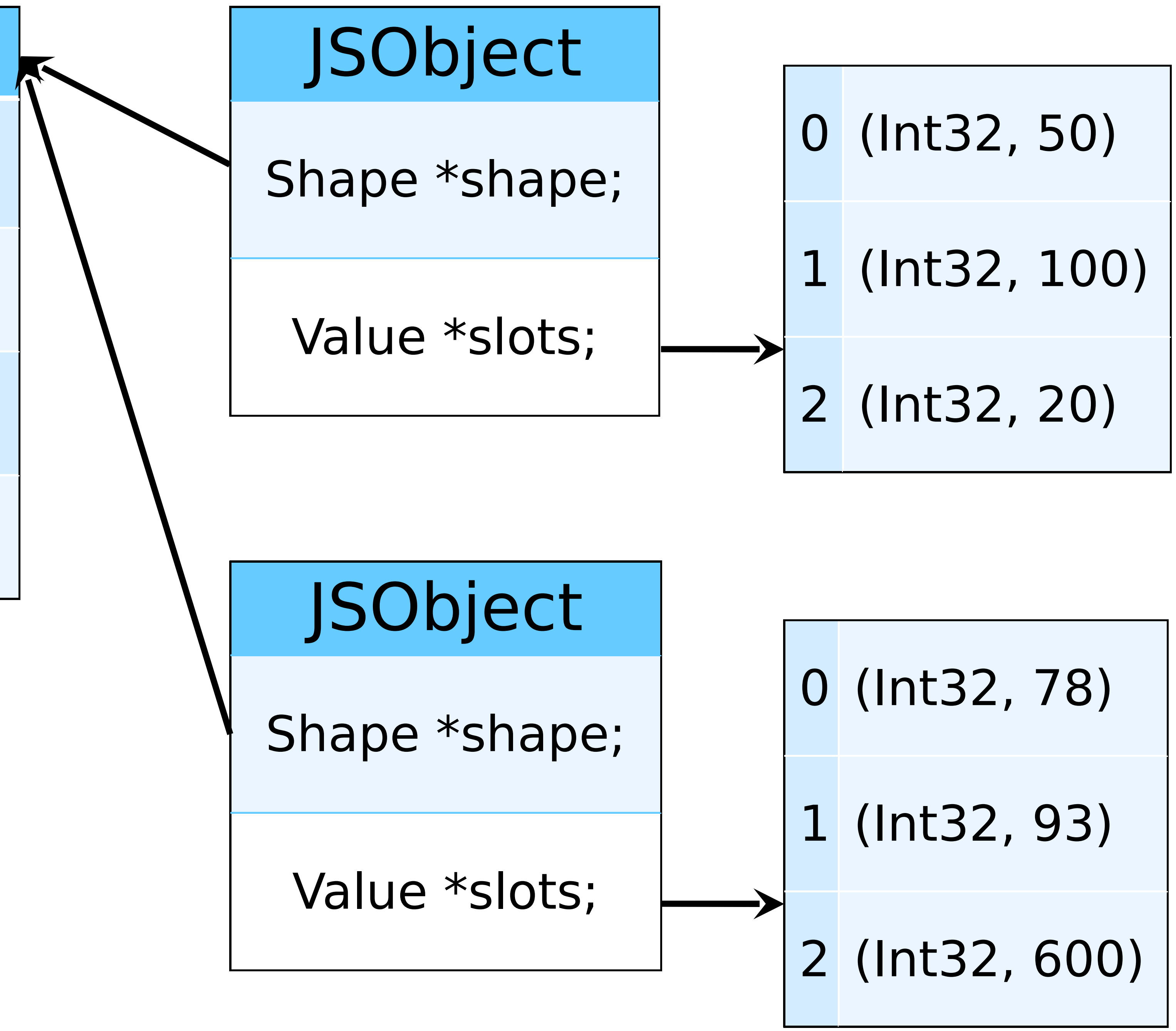
Object Access

- Observation
  - Usually, objects flowing through an access site look the same
- If we knew an object's layout, we could bypass the dictionary lookup

Object Layout

Shape	
Property Name	Slot Number
x	0
y	1
z	2

```
var obj = { x: 50,
            y: 100,
            z: 20
};
...
var obj2 = { x: 78,
             y: 93,
             z: 600
};
```



Familiar Problems

- We don’t know an object’s shape during JIT compilation
- ... Or even that a property access receives an object!
- Solution: leave code “blank,” lazily generate it later

Inline Caches

```
function f(x) {
    return x.y;
}
```

Inline Caches

- Start as normal, guarding on type and loading data

```
if (arg0.type != OBJECT)
  goto slow_property;
JSObject *obj = arg0.data;
```

Inline Caches

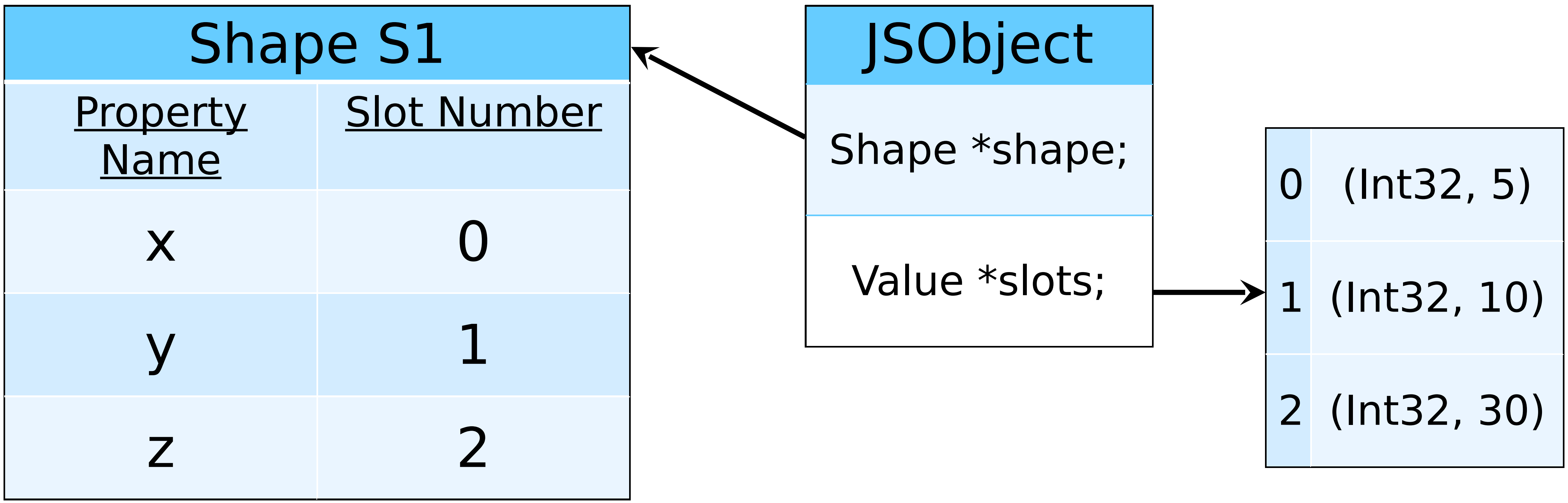
- No information yet: leave blank (nops).

```
if (arg0.type != OBJECT)
  goto slow_property;
JSObject *obj = arg0.data;

  goto property_ic;
```

Property ICs

```
function f(x) {
    return x.y;
}


f({x: 5, y: 10, z: 30});
```

Property ICs



Property ICs

Shape S1	
Property Name	Slot Number
x	0
y	**1**
z	2

Inline Caches

- Shape = S1, slot is 1

```
if (arg0.type != OBJECT)
  goto slow_property;
JSObject *obj = arg0.data;

 goto property_ic;
```

Inline Caches

- Shape = S1, slot is 1

```
if (arg0.type != OBJECT)
  goto slow_property;
JSObject *obj = arg0.data;
```

```
goto property_ic;
```

Inline Caches

- Shape = S1, slot is 1

```
if (arg0.type != OBJECT)
  goto slow_property;
JSObject *obj = arg0.data;
if (obj->shape != S1)
  goto property_ic;
R0 = obj->slots[1].type;
R1 = obj->slots[1].data;
```

Polymorphism

```
function f(x) {
    return x.y;
}
f({ x: 5, y: 10, z: 30});
f({ y: 40});
```

- What happens if two different shapes pass through a property access?

Polymorphism

Main Method

```
  if (arg0.type != OBJECT)
    goto slow_path;
  JSObject *obj = arg0.data;
  if (obj->shape != SHAPE1)
    goto property_ic;
  R0 = obj->slots[1].type;
  R1 = obj->slots[1].data;
rejoin:
```

Polymorphism

Main Method

```
  if (arg0.type != OBJECT)
    goto slow_path;
  JSObject *obj = arg0.data;
  if (obj->shape != SHAPE1)
    goto property_ic;
  R0 = obj->slots[1].type;
  R1 = obj->slots[1].data;
rejoin:
```

Generated Stub

```
stub:
  if (obj->shape != SHAPE2)
    goto property_ic;
  R0 = obj->slots[0].type;
  R1 = obj->slots[0].data;
  goto rejoin;
```

Polymorphism

Main Method

```
if (arg0.type != OBJECT)
  goto slow_path;
JSObject *obj = arg0.data;
if (obj->shape != SHAPE1)
  goto stub;
R0 = obj->slots[1].type;
R1 = obj->slots[1].data;
rejoin:
```

Generated Stub

```
stub:
  if (obj->shape != SHAPE2)
    goto property_ic;
  R0 = obj->slots[0].type;
  R1 = obj->slots[0].data;
  goto rejoin;
```

Chain of Stubs

```
if (obj->shape == S1) {
    result = obj->slots[0];
} else {
    if (obj->shape == S2) {
        result = obj->slots[1];
    } else {
        if (obj->shape == S3) {
            result = obj->slots[2];
        } else {
            if (obj->shape == S4) {
                result = obj->slots[3];
            } else {
                goto property_ic;
            }
        }
    }
}
```

Code Memory

- Generated code is patched from inside the method
  - Self-modifying, but single threaded
- Code memory is always rwx
  - Concerned about protection-flipping expense

mozilla

Basic JIT Summary

- Generates simple code to handle common cases
- Inline caches adapt code based on runtime observations
- Lots of guards, poor type information

Optimizing Harder

- Single pass too limited: we want to perform whole-method optimizations
- We could generate an IR, but without type information...
- Slow paths prevent most optimizations.

Outline

- What is Javascript.
- Javascript Implementations.
  - Values (Objects, strings, numbers)
  - Running Javascript
    - Interpreter (SpiderMonkey)
    - Method JIT (JägerMonkey)
    - Adaptive JIT (IonMonkey)

Code Motion?

```
function f(x, n) {
    var sum = 0;
    for (var i = 0; i < n; i++)
        sum += x + n;
    return sum;
}
```

The addition looks loop invariant.
Can we hoist it?

Code Motion?

```
function f(x, n) {
    var sum = 0;
    var temp0 = x + n;
    for (var i = 0; i < n; i++)
        sum += temp0;
    return sum;
}
```

Let’s try it.

mozilla

Wrong Result

```
var global = 0;
var obj = {
    valueOf: function () {
        return ++global;
    }
}
f(obj, 10);
```

Original code returns 155
Hoisted version returns 110!

Ideal Scenario

- Actual knowledge about types!
- Remove slow paths
- Perform whole-method optimizations

Advanced JITs

- Make optimistic guesses about types
  - Guesses must be informed
    - Don’t want to waste time compiling code that can’t or won’t run
  - Using type inference or runtime profiling
- Generate an IR, perform textbook compiler optimizations

Optimism Pays Off

- People naturally write code as if it were typed
- Variables and object fields that change types are rare

IonMonkey

- Work in progress
- Constructs high and low-level IRs
- Applies type information using runtime feedback
- Inlining, GVN, LICM, LSRA

```
function f(x, y) {
   return x + y + x;
}
```

→

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

Type Oracle

- Need a mechanism to inform compiler about likely types
- Type Oracle: given program counter, returns types for inputs and outputs
- May use any data source – runtime profiling, type inference, etc

Type Oracle

- For this example, assume the type oracle returns “integer” for the output and both inputs to all ADD operations

mozilla

Build SSA

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

```
v0 = arg0
v1 = arg1
```

Build SSA

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

```
v0 = arg0
v1 = arg1
v2 = iadd(v0, v1)
```


mozilla

Build SSA

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

```
v0 = arg0
v1 = arg1
v2 = iadd(v0, v1)
```


mozilla

Build SSA

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

```
v0 = arg0
v1 = arg1
v2 = iadd(v0, v1)
v3 = iadd(v2, v0)
```

Build SSA

GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN

```
v0 = arg0
v1 = arg1
v2 = iadd(v0, v1)
v3 = iadd(v2, v0)
v4 = return(v3)
```

SSA (bad types)

Boxed
Unboxed

```
v0 = arg0
v1 = arg1
v2 = iadd(v0, v1)
v3 = iadd(v2, v0)
v4 = return(v3)
```

Intermediate SSA

- SSA does not type check
- Type analysis:
  - Makes sure SSA inputs have correct type
  - Inserts conversions, value decoding

Type Checks

```
v0 = arg0
v1 = arg1
v2 = unbox(v0, INT32)
v3 = unbox(v1, INT32)
v4 = iadd(v2, v3)
v5 = unbox(v0, INT32)
v6 = iadd(v4, v5)
v7 = return(v6)
```

Optimization (GVN)

```
v0 = arg0
v1 = arg1
v2 = unbox(v0, INT32)
v3 = unbox(v1, INT32)
v4 = iadd(v2, v3)
v5 = unbox(v0, INT32)
v6 = iadd(v4, v5)
v7 = return(v6)
```

Optimization (GVN)

```
v0 = arg0
v1 = arg1
v2 = unbox(v0, INT32)
v3 = unbox(v1, INT32)
v4 = iadd(v2, v3)
v5 = unbox(v0, INT32)
v6 = iadd(v4, v5)
v7 = return(v6)
```


mozilla

Optimized SSA

```
v0 = arg0
v1 = arg1
v2 = unbox(v0, INT32)
v3 = unbox(v1, INT32)
v4 = iadd(v2, v3)

v5 = iadd(v4, v2)
v6 = return(v5)
```

Allocate Registers

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Code Generation

SSA

Native Code (Assembly)

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Code Generation

SSA

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
```

Code Generation

SSA

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
```


mozilla

Code Generation

SSA

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
```

Code Generation

SSA

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(5)
```

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
```

Code Generation

SSA

```
0: stack arg0
1: stack arg1
2: r1    unbox(0, INT32)
3: r0    unbox(1, INT32)
4: r0    iadd(2, 3)
5: r0    iadd(4, 2)
6: r0    return(4)
```

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

Generated Code

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

Generated Code

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

- Bailout exits JIT
- May recompile function

Generated Code

Native Code (Assembly)

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

C Code (Assembly)

```
r1 = arg0;
r0 = arg1;
r0 = r0 + r1;
r0 = r0 + r1;
return r0;
```


mozilla

Guards Still Needed

- Object shapes for reading properties
- Types of…
  - Arguments
  - Values from the heap, returned from C++
- Math
  - Overflow, Divide by Zero, Multiply by -0
  - NaN is falsey in JS, truthy in ISAs

Advanced JITs

- Full type speculation - no slow paths
  - Can move and eliminate code
- If speculation fails, method is recompiled or deoptimized
- Can still use techniques like inline caching!

To Infinity, and Beyond

- Advanced JIT example has four guards:
  - Two type checks
  - Two overflow checks
- Better than Basic JIT, but we can do better
  - Interval analysis can remove overflow checks
  - Type Inference!

Type Inference

- Whole program analysis to determine types of variables
- Hybrid: both static and dynamic
- Replaces checks in JIT with checks in virtual machine
  - If VM breaks an assumption held in any JIT code, that JIT code is discarded

Conclusions

Javascript

```
function f(x, y) {
   return x + y + x;
}
```

Bytecode

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

Conclusions

JägerMonkey

```
    if (arg0.type != INT32)
        goto slow_add;
    if (arg1.type != INT32)
        goto slow_add;
    r0 = arg0.data;
    r1 = arg1.data;
    r2 = r0 + r1;
    if (OVERFLOWED)
        goto slow_add;
    r3 = TYPE_INT32;
rejoin:
  ...
slow_add:
    Value result = runtime::Add(arg0, arg1);
    r2 = result.data;
    r3 = result.type;
    goto rejoin;
```

Bytecode

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

Conclusions

IonMonkey

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

Bytecode

```
GETARG 0 ; fetch x
GETARG 1 ; fetch y
ADD
GETARG 0 ; fetch x
ADD
RETURN
```

Conclusions

IonMonkey

```
if (arg0.type != INT32)
  goto BAILOUT;
r1 = arg0.data;
if (arg1.type != INT32)
  goto BAILOUT;
r0 = arg1.data;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
r0 = r0 + r1;
if (OVERFLOWED)
  goto BAILOUT;
return r0;
```

Futur of IonMonkey ?

```
r1 = arg0;
r0 = arg1;
r0 = r0 + r1;
r0 = r0 + r1;
return r0;
```

Questions?