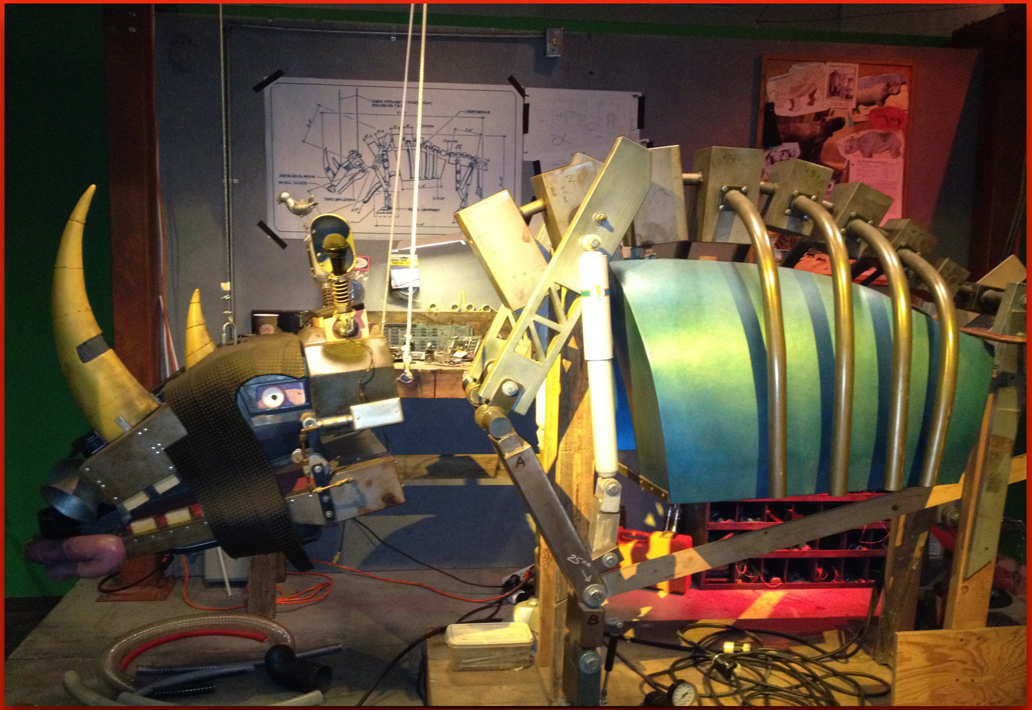
ORACLE®

Nashorn: Implementing a Dynamic Language Runtime on JVM

- Attila Szegedi
  Principal Member of Technical Staff



The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

What is Nashorn?

- Nashorn is an ECMAScript 5.1 runtime on top of JVM.
- Open source: all development happens in OpenJDK.
- Ships as standard part of Oracle's Java SE starting with version 8.
- Accessible through standard javax.script.* API, or directly through jdk.nashorn.api.scripting package.
- Command line: $JAVA_HOME/bin/jjs
- Has no interpreter currently; compiles to Java bytecode on-the-fly.

Why Nashorn?

- Full ECMAScript 5.1 compliance.
- Modern codebase.
- Security minded.
- Proving ground for invokedynamic.
- Laying groundwork for general dynamic languages platform support.

Release Schedule So Far

- Java 8: first release
- Java 8u20: security fixes and smaller improvements
- Java 8u40: performance release + some ES6 features
- Java 9: …

8u40: Performance Focus

- It’s easy to write a slow language runtime.
- You can spend a lifetime writing optimizations in your runtime.
- We want to go far, and then generalize the benefits.
- Hence, this talk will mostly be about runtime performance.

What can Nashorn do today?

- Parameter type specialized compilation
- Gradual deoptimization with on-stack code replacement
  - a.k.a. “optimistic typing”
- Static code analysis
- Compiler optimizations
- Take advantage of runtime context during compilation

Parameter type specialized compilation

- Here’s code versions for f generated when invoked with int and double:

```
function square(x) {
    return x*x;
}
print(square(500));
print(square(500.1));
```

```
public static square(Object;I)I
  0     iload 1
  1     iload 1
  2     invokedynamic imul(II)I
  7     ireturn

public static square(Object;D)D
  0     dload 1
  1     dload 1
  2     dmul
  3     dreturn
```

Parameter type specialized compilation

- Here’s code version for f generated when invoked with object:

```
function square(x) {
    return x*x;
}

var a = {
  valueOf: function() {
     return 500;
  }
};

print(square(a));
```

```
public static square(Object;Object;)D
 0 aload 1
 1 invokestatic JSType.toNumber(Object;)D
 4 aload 1
 5 invokestatic JSType.toNumber(Object;)D
 8 dmul
 9 dreturn
```

Parameter type specialized compilation

- toNumber is invoked twice: object-to-number can have side effects!

```
function square(x) {
    return x*x;
}
var i = 500;
var a = {
  valueOf: function() {
     return i++;
  }
}

print(square(a))
```

```
public static square(Object;Object;)D
 0 aload 1
 1 invokestatic JSType.toNumber(Object;)D
 4 aload 1
 5 invokestatic JSType.toNumber(Object;)D
 8 dmul
 9 dreturn
```

Parameter type specialized compilation

- works with higher order functions, too, as the JS function object is a holder for all its type specializations, and parameter types propagate.

```
function square(x) {
    return x*x;
}

function apply(f, x) {
    return f(x);
}

print(apply(square, 500));
print(apply(square, 500.1));
```

Deoptimizing compilation: arithmetic overflow

- can’t just use IMUL.

```
function square(x) {
    return x*x;
}
print(square(1 << 17));
```

```
public static square(Object;I)I
 0     iload 1
 1     iload 1
 2     invokedynamic imul(II)I
 7     ireturn
```

Deoptimizing compilation: arithmetic overflow

```
public static square(Object;I)I
  try L0 L1 L2 UnwarrantedOptimismException

  0    iload 1
  1    iload 1
L0
  2    invokedynamic imul(II)I [static ‘mathBootstrap']
L1
  7    ireturn
L2
  8    iconst_2
  9    anewarray Object
 12    aload 0
 13    iload 1
 14    invokedynamic populateArray([Object;Object;I)[Object;[‘populateArrayBootstrp']
 23    invokestatic RewriteException.create(UnwarrantedOptimismException;
                                 [Object;)RewriteException;
 26    athrow
```

Deoptimizing compilation: arithmetic overflow

```
public static int mulExact(final int x, final int y, final int programPoint)
    throws UnwarrantedOptimismException
{
    try {
        return Math.multiplyExact(x, y);
    } catch (final ArithmeticException e) {
        throw new UnwarrantedOptimismException((long)x * (long)y, programPoint);
    }
}
```

- Math.multiplyExact() is intrinsified by HotSpot
- We must perform the operation and return the result in the exception

Deoptimizing compilation: property type

```
function f(a) {
    return a.foo * 2;
}
f({foo: 5.5})
```

Deoptimizing compilation: property type

```
 public static f(Object;Object;)I
   try L0 L1 L2 UnwarrantedOptimismException
   try L3 L4 L2 UnwarrantedOptimismException

   0    aload 1
L0
   1    invokedynamic dyn:getProp|getElem|getMethod:foo(Object;)I [static “bootstrap” pp=2]
L1
   6    iconst_2
L3
   7    invokedynamic imul(II)I [static “mathBootstrap” pp=3]
L4
  12    ireturn
L2
  13    iconst_2
  14    anewarray Object
  17    aload 0
  18    aload 1
  19    invokedynamic populateArray([Object;Object;Object;)[Object;
  ...
```

Deoptimizing compilation: property type

```
public static f(Object;Object;)D
  0     aload 1
  1     invokedynamic dyn:getProp|getElem|getMethod:foo(Object;)D [static 'bootstrap']
  6     ldc 2.0
  9     dmul
 10     dreturn
```

- Since first argument is now double, second is also widened to double.
  - Static analysis FTW!
- Multiplication can no longer overflow, so implicitly it also becomes non-optimistic in a single deoptimizing recompilation pass.

Okay, But How Do We Deoptimize Running Code?

- To deoptimize running code, we must be able to:
  - recompile it on the fly, and
  - replace running code on top of the stack.

Okay, But How Do We Deoptimize Running Code?

- To deoptimize running code, we must be able to:
  - recompile it on the fly, and
  - replace running code on top of the stack.
- We achieve this with a pure bytecode solution (runs on any JVM) that
  - throws an exception where type assumptions are too narrow,
  - links call site in caller with exception handler that derails into compiler,
  - recompiles a new version of the code with wider type,
  - compiles a separate one-shot continuation version of the code too,
  - jumps into the continuation variant to resume execution.

Deoptimizing Compilation: rest-of method

```
  public static f(RewriteException;)D
    0    goto L0
    3    nop
        ...
    8    athrow
L2
    9    ldc 2.0
   12     dmul
   13     dreturn
L0
   14     aload 0
   15     astore 2
   16     aload 0
   17     invokevirtual RewriteException.getByteCodeSlots()[Object;
   20     dup
   21     iconst_0
   22     aaload
   23     astore 0
   25     iconst_1
   26     aaload
   27     astore 1
   32     invokevirtual RewriteException.getReturnValueDestructive()Object;
   35     invokestatic JSType.toNumber(Object;)D
   38     goto L2
```

Deoptimizing Compilation: rest-of method

- In extremely simplified terms, if a function can be deoptimized, we link a MethodHandles.catchException() combinator into every call site that roughly does this:

```
try {
   return invoke_function(params);
} catch (RewriteException e) {
   return do_the_thing(fndata, e));
}
```

- … where “do_the_thing” means recompile the function, store the new version as the current one for later invocations, invalidate already linked call sites (incl. this one), generate the continuation version, and jump into it.

Tricky stuff

- Deoptimizing recompilation in presence of recursive (incl. mutually recursive) invocations: function is on stack below top, too!
- Not exhausting the stack on cascading recompiles. For that we actually use:

```
try {
   return invoke_function(params);
} catch (RewriteException e) {
   return fold(invoke, do_the_thing_return_continuation(fndata, e)));
}
```

- BTW, as a super-heavy user of invokedynamic, we push for optimizations in it with the relevant JVM teams.
  - “Hey, we need catchCombinator to be super fast in the fast path. Yeah, no matter how many or what type parameters function has.”

Tricky stuff

- Of course, getting every piece of this logic right can mess with your sanity.

Wrestling with Nashorn's parser for three days, this is how I feel. Illustrate my work with Chtulhu Gloom cards now.

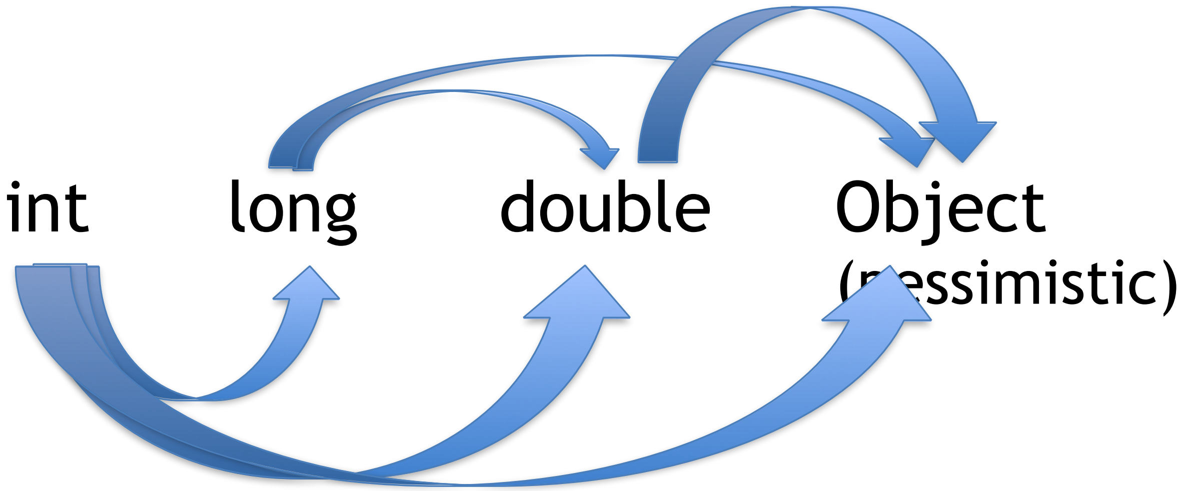
Tricky stuff

- Of course, getting every piece of this logic right can mess with your sanity.

Lagergren
@lagergren

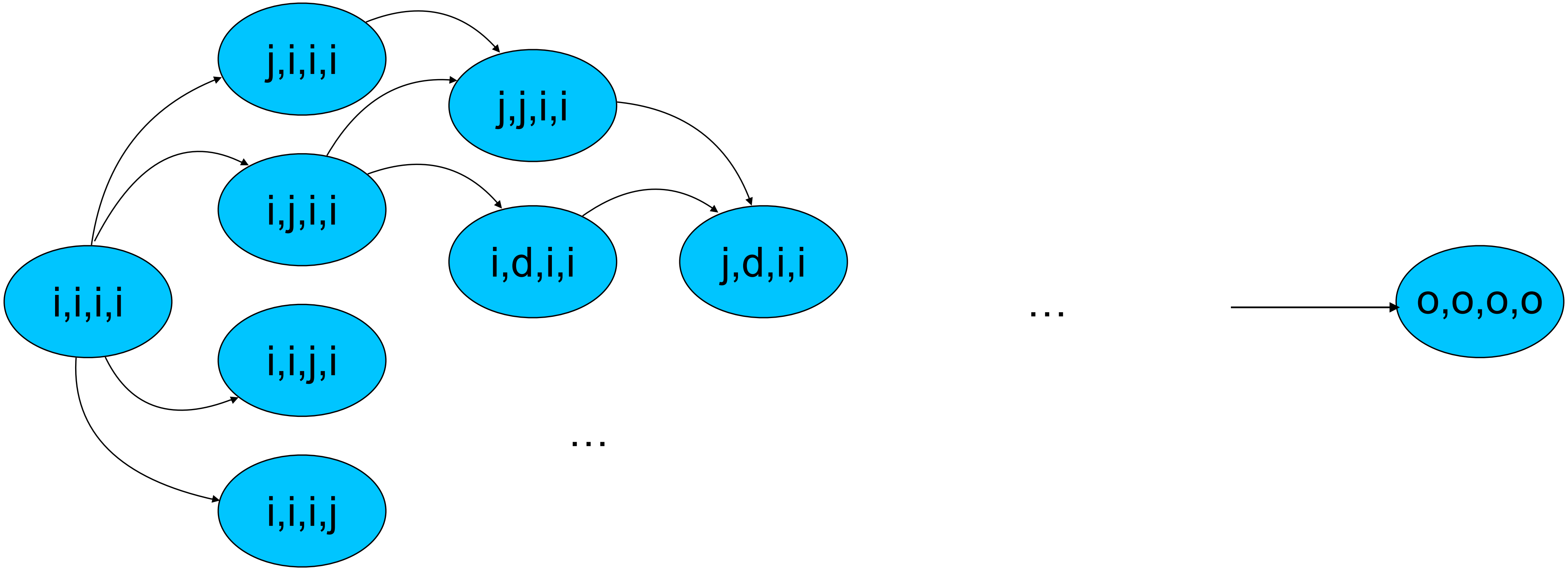
Today: JavaScript object shape problem #fetalposition /cc @shipilev @viktorklang @asz

Nashorn’s (Extremely Simple) Type Hierarchy



int long double Object (pessimistic)

Deoptimization is stepping in a lattice of type tuples



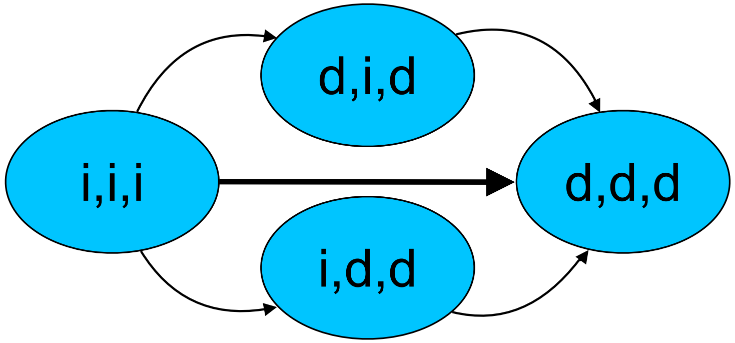
Sometimes we don’t want to be optimistic

- Static analysis can help us avoid optimism when it’s not needed.
- E.g. overflow semantics is sufficient because an operator would coerce anyway: **(i*i)l0** (logical or coerces to 32-bit int in JS)

It’s compile time! No, it’s run time! No, it’s both!

- We’re in a unique position with a dynamic language as compile and run times are not separated.
- We use it to leap ahead in walking the type lattice instead of stepping by one on each recompilation.

It’s compile time! No, it’s run time! No, it’s both!



```
function f(o) {
    var x = o.x;
    var y = o.y;
    return x * y;
}

print(f({x: 1.1, y: 2.1}));
```

- When we deoptimize because o.x is double, we also peek at o.y type.
- The runtime values are available to the compiler!
- We only evaluate side-effect free expressions.

It’s compile time! No, it’s run time! No, it’s both!

- Savings can be significant; here’s an Octane box2d snippet:

```
aa.prototype.SolveVelocityConstraints = function() {
        var d, h = 0,
            l = this.m_bodyA,
            j = this.m_bodyB,
            o = l.m_linearVelocity,
            q = l.m_angularVelocity,
            n = j.m_linearVelocity,
            a = j.m_angularVelocity,
            c = l.m_invMass,
            g = j.m_invMass,
            b = l.m_invI,
            e = j.m_invI;
```

Hey, There’s Static type Inference in Your Dynamic Language Runtime

- Static analysis in a dynamic language is great when you’re targeting a statically typed runtime.
- Nashorn calculates types of expressions and local variables. Local variable types are propagated from def to use sites.
  - Handles tricky control flow situations. Examples:
    - Control transfer into catch blocks
    - break from a finally
  - Operates on AST
  - Recognizes patterns (e.g. value of expression is used as an object/fn)

Try blocks - multiple type liveness

```
(function f() {
    var a = 1; // def as int
    try { // (def as double implicitly, int remains live)
        print(a); // use as int
        a = 3; // def as int (and double implicitly)
        print(a); // use as int
        a = 2.1; // def as double (int no longer live)
        print(a); // use as double
    } catch(e) {
        print(a); // use as double
    }
    print(a); // use as double
})();
```

Variables with multiple types

```
function f(x) {
    var i;
    var y;
    if(x) {
        i = 1;
        y = i * 2; // i int here
    }
    return i; // i object here
}
print(f(5))
```

- Even then, we preserve it as int within the branch range and add an implicit boxing before the join point.
- This preservation of narrower types in ranges is true for all types.
- Variables can be stored in slots of different types at different times.
- Here, we emit a synthetic “else” block that initializes i to Undefined.INSTANCE.

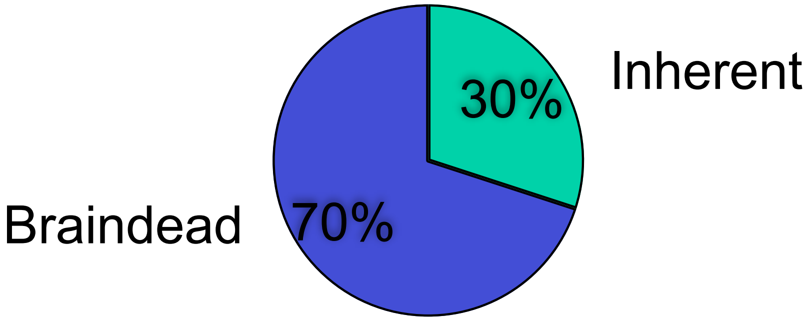
Dead code elimination

- Due to type specialization functions are compiled only on first invocation.
- Dead stores are eliminated, as well as some side-effect free parts of their right sides.
  - Since we don’t have full liveness analysis on AST, we resorted to weaker “type liveness” analysis for variables to avoid unnecessary conversions at control flow joins. Almost as a side effect, partial dead store elimination came out of it.

Type Liveness Analysis is not a Luxury

```
var y = 0;
 for(var i = 0; i < 10; ++i) {
     var x = i * 2;
     y += x;
 }
 print(y);
```

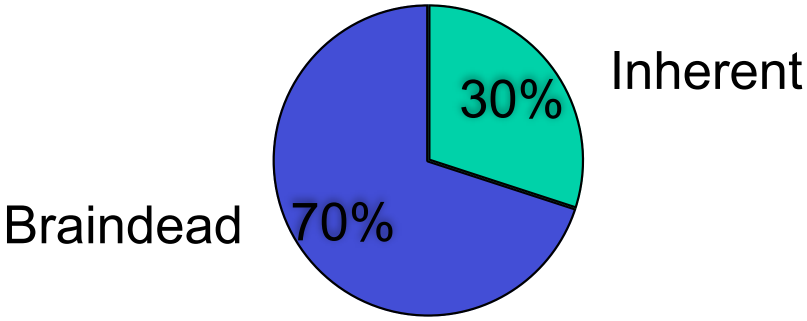
- 30% inherent type proliferation in a dynamic language
- 70% JavaScript's braindead local variable scoping rules.



Type Liveness Analysis is not a Luxury

```
var y = 0;
 for(var i = 0; i < 10; ++i) {
     var x = i * 2;
     y += x;
 }
 print(y);
 print(x);
```

- 30% inherent type proliferation in a dynamic language
- 70% JavaScript's braindead local variable scoping rules.

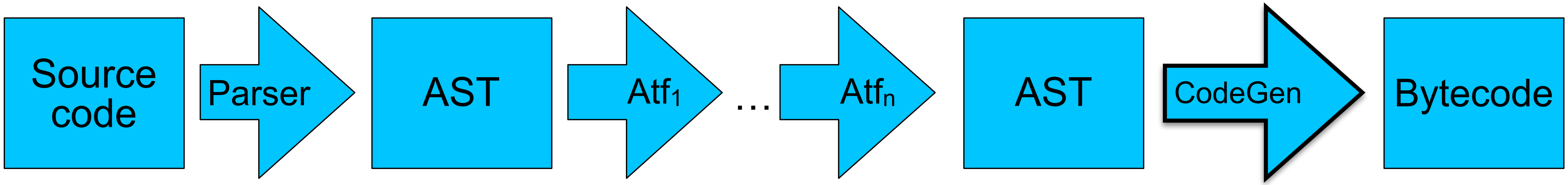


Things I Don’t Have Time To Talk About

- Object representation
- Array representation (short story: we strive to maintain compact arrays of homogenous primitive types whenever we can so we can use native array element getters/setters)
- Efficient linking of builtins (Array.push, Array.pop, String.charAt, etc.)
  - Switchpoint-guarded direct constant linking to implementation.
- Efficient linking of “fn.apply(this, arguments)” and similar recurring patterns.

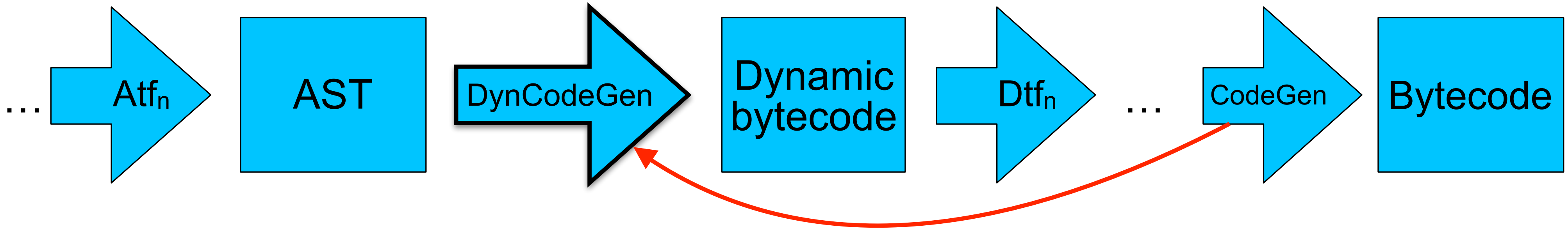
The Bright Future We Want To Create

Nashorn compiler pipeline



- After parser, there are lots of AST transformation steps (constant folding, lowering, splitting, symbol assignment, static type calculation)
- Code generator translate AST to JVM bytecode. It's heavy machinery.
  - Emits code for optimistic operations, continuation handling, loads vs. stores, self-assignment stores (++, += operations), split functions control flow handover, …

Separate AST linearization

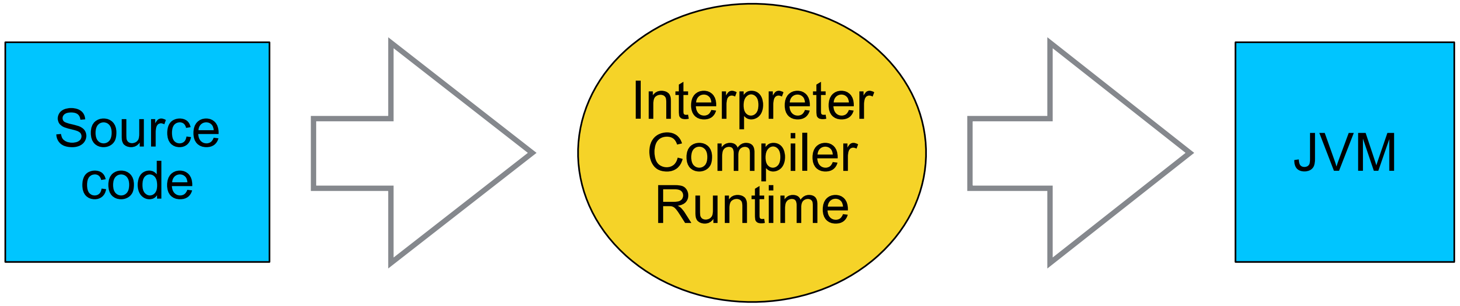


- Code generator heavy lifting doesn’t need to produce JVM bytecode.
- It could produce “dynamically typed” bytecode.
- Separate optimization steps could work on this bytecode.
  - Some optimizations work better on a basic-block representation.
- Finally, a light straightforward new codegen could emit JVM bytecode from dynamic bytecode.

Benefits of new intermediate representation

- Some calculations are impossible or near impossible on AST, but easy and well understood on basic blocks.
  - E.g. liveness analysis needs backwards control flow traversal.
- AST is language dependent, bytecode isn't. Or less so.
- The proposed dynamic bytecode is still very close to JVM bytecode, so can be targeted by many languages.
  - This is the reusability/toolchain idea.
- Similar in idea to LLVM's IR.

Why?



Source code

Interpreter Compiler Runtime

JVM

Assumption: you talk to JVM in bytecode

Reuse

- V8 has 972k lines of code
- Nashorn has 213k lines of code
  - Because underlying Java platform provides lots of services
- Imagine your language runtime tapping some of Nashorn’s 213k LOC.

Arithmetic example:

- Source code:

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

- Dynamic bytecode:

```
f(INT x, INT y)
  LOAD x
  LOAD y
  MUL
  RETURN
```

- JVM bytecode:

```
public static f(Object;II)I
  iload 1
  iload 2
  invokedynamic imul(II)I ['mathBootstrap']
  ireturn

catch UnwarrantedOptimismException
  iconst_3
  anewarray Object
  aload 0
  iload 1
  iload 2
  invokedynamic populateArray([Object;Object;II)[Object;
  invokestatic RewriteException.create(...)RewriteException;
  athrow

  local :this         L3   L2   0    Object;
  local x            L3   L2   1    I
  local y            L3   L2   2    I
```

Benefits of new intermediate representation

- You can emit symbolic identifiers for variables; final codegen will take care of mapping to JVM local variable slots.
- It’ll infer types for variables and other values (with pluggable language specific rules for operations).
- It’ll emit optimistic operations when needed, set up exception handlers, and emit continuation restart methods.
- It’ll infer JVM return types for functions.

Property getter example:

- Source code:

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

- Dynamic bytecode:

```
f(OBJECT x, INT y)
  LOAD x
  GETPROP foo
  LOAD y
  MUL
  RETURN
```

- JVM bytecode:

```
public static f(Object;II)I
  aload 1
  invokedynamic dyn:getPropIgetElemIgetMethod:foo(Object;)I
  iload 2
  invokedynamic imul(II)I ['mathBootstrap']
  ireturn

catch UnwarrantedOptimismException
  iconst_3
  anewarray Object
  aload 0
  aload 1
  iload 2
  invokedynamic populateArray([Object;Object;Object;I)[Object;
  invokestatic RewriteException.create(...)RewriteException;
  athrow

  local :this      L3   L2   0    Object;
  local x         L3   L2   1    Object;
  local y         L3   L2   2    I
```

Hypothetical lexical scope getter example:

- Source code:

```
function f(x, y) {
  function g(z) {
    return x.foo * z;
  }
  return g(z);
}
```

- Dynamic bytecode:

```
g(INT y)
  LOAD x
  GETPROP foo
  LOAD y
  MUL
  RETURN
```

- JVM bytecode:

```
public static f$g(ScriptFunction;Object;I)I
  aload 0
  invokevirtual ScriptFunction.getScope()ScriptObject;
  astore 3
  aload 3
  invokedynamic dyn:getProp|getElem|getMethod:x(Object;)Object;
  invokedynamic dyn:getProp|getElem|getMethod:foo(Object;)I
  iload 2
  invokedynamic imul(II)I
  ireturn

catch UnwarrantedOptimismException
  …
```

Lexical scope

- Most dynamic languages have the concept of accessing variables from outer lexical scope.
- JVM bytecode doesn’t.
- We thought we can bridge that.
  - In dynamic bytecode, you could symbolically reference those variables as if they were locals.
  - JVM code generator would emit necessary “load scope, get variable as property from it” sequence, complete with optimism if needed.
  - JVM code has no nesting, though, so this would be too hard.

Dynamic bytecode to JVM bytecode

- Types are inferred; statically non-provable types are optimistically presumed; optimistic operations are emitted as indy invocations

```
LOAD x     → ILOAD 3 or
             DLOAD 3 etc.

LOAD x     → ALOAD 2 // scope
             INVOKEDYNAMIC getprop:x

MUL        → IMUL or
             INVOKEDYNAMIC imul

ALOAD x       ALOAD 5
GETPROP y →   INVOKEDYNAMIC getprop:y
```

Other things we can do automatically

- Splitting of large methods into less than 64k JVM bytecode chunks.
- Current Nashorn AST splitter makes conservative size estimates.
- Dynamic bytecode size much better approximates JVM bytecode size.
- Passing local variables used by split chunks needs artificial lexical scope objects.
- Split functions are also subject to deoptimizing recompilation.
  - Trickier as possibly multiple stack frames need to be saved/restored.

Summary

- Some things are hard:
  - static analysis, type-specialized on-demand compilation, optimistic types with on-stack-replacement deoptimizing recompilation, etc.
- We have solved a bunch of those with Nashorn, and would like to offer them as a reusable library.
- The library needs a suitable code representation as its input.
  - AST is too high level, JVM bytecode is too statically typed.
  - We think something “almost bytecode”, but with optional types.

Summary

- We want to give people a framework that:
  - takes “dynamic bytecode" as its input with given parameter types
  - proves JVM types of expressions statically
  - where they can’t be proven or could overflow, inserts optimistic operations
  - allows for running various optimizing transforms on it
  - emits the final JVM bytecode, with all the smarts presented earlier.
  - helps you with method handle combinators for linking call sites to functions that can get deoptimized.

ORACLE®