
ORACLE®

ORACLE

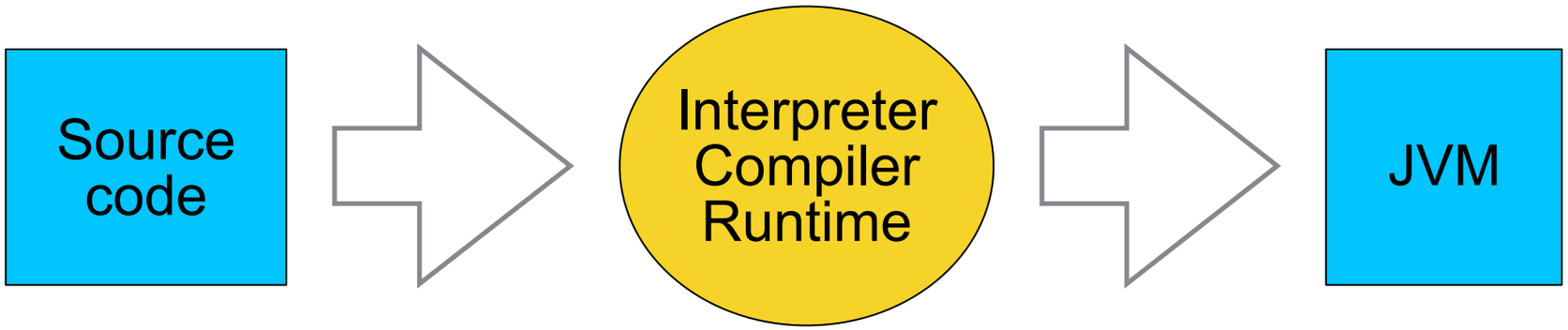
Towards JVM Dynamic Languages Toolchain

Insert Picture Here

- 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.

Why?



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.

What can Nashorn do today?

- Parameter type specialized compilation
- Gradual deoptimization with on-stack code replacement
  - a.k.a. optimistic typing
- Splitting large methods and array initializers
- Static code analysis
- Compiler optimizations

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
```

Deoptimizing compilation: arithmetic overflow

- can’t just use IMUL.

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

```
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.

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
```

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)|0` (logical or coerces to 32-bit int in JS)

Type inference

- 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

Handling undefined

```
function f(x) {
    var i;
    print(i);
    var sum = 0;
    for(i = 0; i < x; ++i) {
        sum += i;
    }
}
print(f(5))
```

- “undefined” is a type in the type calculator with a single value.
- When known undefined is loaded, we just load it as a constant.
- Otherwise, it’s an object. But that’s rare.

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.

Parameters with multiple types

```
function f(x, y) {
  x = x * 2.1;
  return x * y;
}
print(f(5, 6))
```

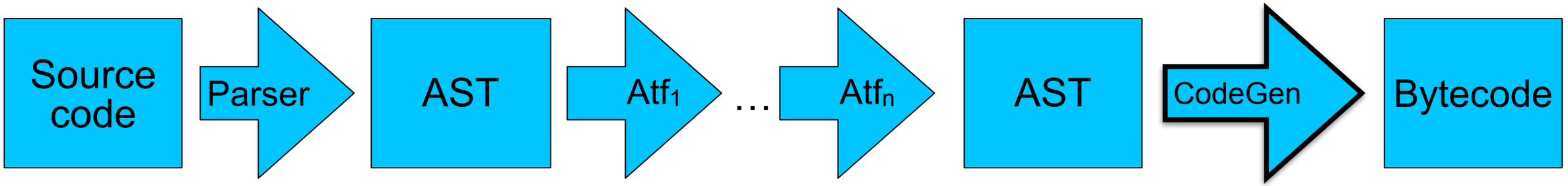
- Currently, storage for locals of multiple types is contiguous.
- Method prologue is emitted to shift incoming parameters to the right as necessary.

```
public static f(Object;II)D
  iload 2
  istore 4

  iload 1
  i2d
  ldc 2.1
  dmul

  dstore 2
  dload 2
  iload 4
  i2d
  dmul
  dreturn

  local x     L0   L2   1    I
  local :this L0   L4   0    Object;
  local x     L2   L4   2    D
  local y     L0   L4   4    I
```

Dead code elimination

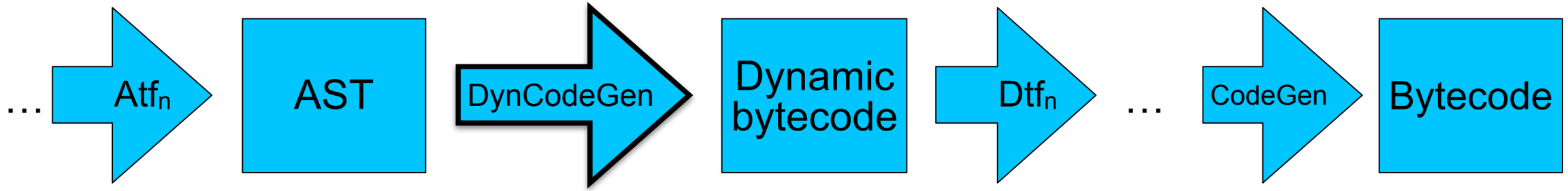
- Due to type specialization functions are compiled only on first invocation.
- Dead stores are eliminated, as well as some side-effect free parts of their right sides.
  - Since we don’t have full liveness analysis on AST, we resorted to weaker “type liveness” analysis for variables to avoid unnecessary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

Nashorn compiler pipeline



- After parser, there are lots of AST transformation steps (lowering, splitting, symbol assignment, 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.

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:getProp|getElem|getMethod: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
```

Code regeneration reasons

- foo might turn out not to be an int (but a double, or even an object)
- multiplication can overflow.
- If needed, function type during regeneration will also evolve from int to long or double.

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 can bridge that.
  - In dynamic bytecode, you can symbolically reference those variables as if they were locals.
  - JVM code generator will emit necessary "load scope, get variable as property from it" sequence, complete with optimism if needed.

Dynamic bytecode to JVM bytecode

- Types are inferred; statically non-provable types are optimistically presumed; some 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.

Things you don’t even think about

- Remember how we capture local variables in the optimism exception handler?
- Know what we can’t capture?
- Stack state.
- We’ll add artificial stores into temporary local variable slots for anything that needs to remain on stack, e.g.

Things you don’t even think about

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

- If squaring y overflows, we already have square of x on stack, and it must be captured and restored.
- If a value on stack isn’t already a load of a local variable, we store it into a new temporary and load it back on stack, thus stashing them away in a place accessible to the exception handler.

Things you don’t even think about

```
  public static f(Object;II)I
    try L0 L1 L2 UnwarrantedOptimismException
    try L3 L4 L5 UnwarrantedOptimismException
    try L4 L6 L2 UnwarrantedOptimismException

    0     iload 1
    1     iload 1
L0 [bci=2]
    2     invokedynamic imul(II)I
L1 [bci=7]
    7     istore 3
    8     iload 3
    9     iload 2
   10     iload 2
L3 [bci=11]
   11     invokedynamic imul(II)I
L4 [bci=16]
   16     invokedynamic iadd(II)I
L6 [bci=21]
   21     ireturn
```

```
L5 [bci=22]
   22     iconst_4
   23     anewarray Object
   26     iload 3
   27     invokedynamic populateArray([Object;I)[Object;
   32     goto L8
L2 [bci=35]
   35     iconst_3
   36     anewarray Object
L8 [bci=39]
   39     aload 0
   40     iload 1
   41     iload 2
   42     invokedynamic populateArray([Object;Object;II)
   47     iconst_0
   51     invokestatic RewriteException.create(…);
   54     athrow

    local :this          L7   L5   0    Object;
    local x              L7   L5   1    I
    local y              L7   L5   2    I
```

Could be improved

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

- Generates:

```
iload 1
iload 1
invokedynamic imul(II)I

iload 1
pop
istore 5
iload 5
iload 1

iload 2
iload 2
invokedynamic imul(II)I
```

- Could be:

```
iload 1
iload 1
invokedynamic imul(II)I

istore 5
iload 1

iload 2
iload 2
invokedynamic imul(II)I
```

Could be improved

- We detect the need to store the value too late (after load of the next one was emitted).
- Hard to solve with AST, as it doesn't have the stack-machine model in it. Easier with a CFG of basic blocks in an already stack-machine language.
- We could also avoid synthetic stores for all values that are constants or pure functions of local variables. (Currently: locals only).

Linking aspect

- There is also a linking aspect involving MethodHandles.catchException combinator that jumps into deoptimizing recompilation and runs a rest-of method when a function exits with RewriteException.
- Won’t go into details here.

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 an input code representation to work on.
  - 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 you 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.

Hardware and Software

Engineered to Work Together


ORACLE®