Project Babylon

Intro and Update

Steve Dohrmann

April 24, 2024

Acknowledgement

- Much of the material in these slides is heavily inspired by slides from Paul Sandoz's excellent JVMLS '23 Babylon presentation. He has graciously given permission for such use.
- The slides here reflect my understanding and experience with Babylon and code reflection. Any mistakes are mine alone.

Motivation – why something is needed

Specialized programming domains*, e.g.,

- GPU kernels, and kernel call graphs
- Differentiable programs
- Machine learning models
- Parallel graph programs
- Probabilistic programming
- SQL statements (or anything C# LINQ can do)
- Vectorizable programs

It should be easy for developers to write Java code for such domains and combine it with traditional Java code

*List is from Paul Sandoz's JVMLS Babylon presentation, side #3

Example – Differentiable programs

$$E(x) = \frac{(ax - b)^2}{2}$$

```
// expression APIs
Op.func(“err”, Op.var(a), Op.var(b), Op.var(x))
.body(
    Op.return(Op.expr(
        Op.pow(
            Op.var(a).mul(Op.var(x)).sub(Op.b),
            Op.constant(2d))
        .div(Op.constant(2d)))))

// embedded strings
Op.Func e = Op.func(“(a * x – b) ^ 2) / 2”); // types?

// Java method
public static double err(double a, double b, double x) {
    return Math.pow(a * x – b), 2) / 2;
}
```

Motivation – how to implement support?

1. Adding direct Java support for these specialized programming models:
   - would be very complicated
   - models would overlap and likely conflict
   - would take a long time
   - new models appear relatively quickly
   - would not serve all Java programs, e.g.,
     - not all programs can run on a GPU,
     - not all programs are differentiable

2. Tools to implement such support as a library are not yet there in Java
   - Existing reflection APIs do not give sufficiently deep access
   - Access to AST is not fully supported, and is specific to each Java compiler
   - Starting with bytecode is too low-level, a lossy start for constructs and types

Babylon wants to add code reflection tools to support strategy #2

Code Reflection – what is needed

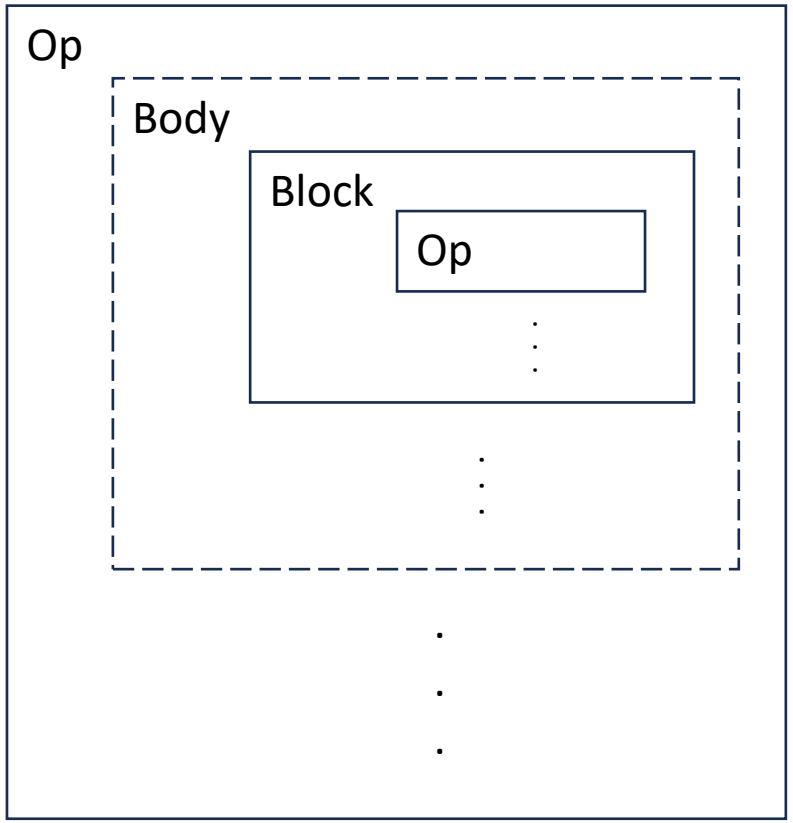
- What do we want to do with Java programs?
  - Analyze them
  - Transform them
    - to other Java code
    - to some non-Java code
- We need:
  1. a public, supported data structure that holds code elements of the program we want to represent
  2. an API with which to construct and manipulate the structure and its elements

Elements of the code reflection model

- Operation
  - name
  - zero or more operands
  - zero or more attributes
  - a result type
  - zero or more bodies
- Body
  - zero or more input parameters
  - a yield type
  - one or more blocks
- Block
  - one or more ops

CodeElements

- Concisely: a tree structure of op -> body* -> block+ -> op+
- This recursive structure lets the model:
  - be used for both high-level (compound) and low-level (leaf) operations
  - scale well with complexity



Java core and auxiliary operations

- Core operations model Java's fundamental elements, e.g.,
  - e.g., assignment, arithmetic, logic
- Auxiliary operations model higher-level Java constructs
  - e.g., for loops, try statements, switch expressions, conditionals
  - each auxiliary op can be lowered to core ops

Example – Java code model

```
@CodeReflection
public static int loop(int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += i;
  }
  return sum;
}
```

```
Method method = cls.getDeclaredMethod(loop”, int.class);
CoreOps.FuncOp javaFunc = method.getCodeModel().get();
System.out.println(javaFunc.toText());
```

------- Java high-level dialect -------

```
func @"loop" (%0 : int)int -> {
    %1 : Var<int> = var %0 @"size";
    %2 : int = constant @"0";
    %3 : Var<int> = var %2 @"sum";
    java.for
        ()Var<int> -> {
            %4 : int = constant @"0";
            %5 : Var<int> = var %4 @"i";
            yield %5;
        }
        (%6 : Var<int>)boolean -> {
            %7 : int = var.load %6;
            %8 : int = var.load %1;
            %9 : boolean = lt %7 %8;
            yield %9;
        }
        (%10 : Var<int>)void -> {
            %11 : int = var.load %10;
            %12 : int = constant @"1";
            %13 : int = add %11 %12;
            var.store %10 %13;
            yield;
        }
        (%14 : Var<int>)void -> {
            %15 : int = var.load %3;
            %16 : int = var.load %14;
            %17 : int = add %15 %16;
            var.store %3 %17;
            java.continue;
        };
    %18 : int = var.load %3;
    return %18;
};
```

----- Java low-level dialect -----

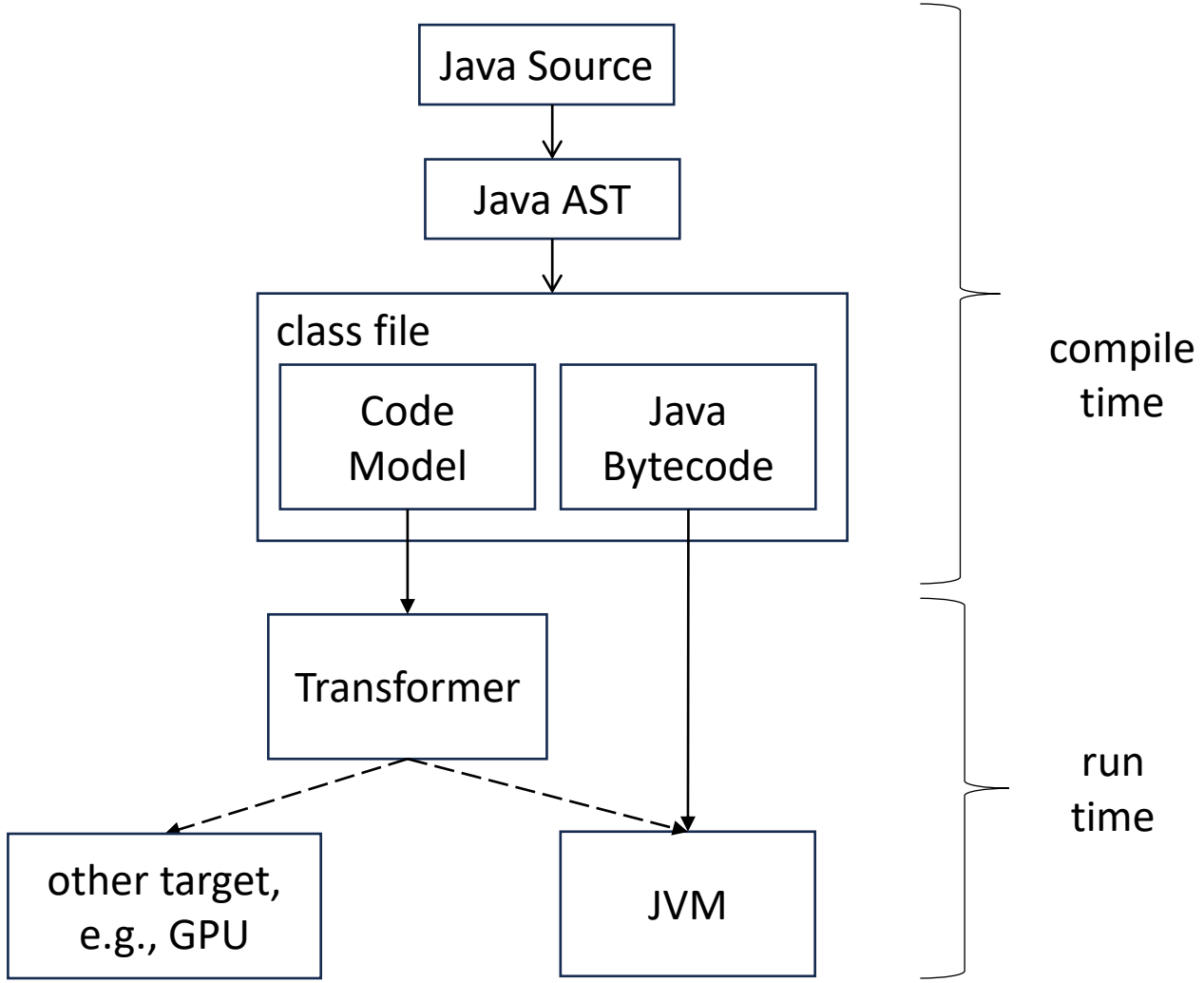
```
func @"loop" (%0 : int)int -> {
    %1 : Var<int> = var %0 @"size";
    %2 : int = constant @"0";
    %3 : Var<int> = var %2 @"sum";
    %4 : int = constant @"0";
    %5 : Var<int> = var %4 @"i";
    branch ^block_0;
  ^block_0:
    %6 : int = var.load %5;
    %7 : int = var.load %1;
    %8 : boolean = lt %6 %7;
    cbranch %8 ^block_1 ^block_2;
  ^block_1:
    %9 : int = var.load %3;
    %10 : int = var.load %5;
    %11 : int = add %9 %10;
    var.store %3 %11;
    branch ^block_3;
  ^block_3:
    %12 : int = var.load %5;
    %13 : int = constant @"1";
    %14 : int = add %12 %13;
    var.store %5 %14;
    branch ^block_0;
  ^block_2:
    %15 : int = var.load %3;
    return %15;
};
```

What can be done code models?

- Traverse up and down the tree structure
- Construct code elements using a builder
- Examine dependency relationships between elements
  - Query dominance relationships between Blocks
  - Get a list of Ops that use another Op's result
- Transform one model to another by traversing a tree and building another by copying, removing, or replacing ops
- Serialize and deserialize models
- Together these support building, analysis, transformation, and storage
- Notes
  - Currently the "roots" of code models are methods and lambda expressions
  - A model is only be provided if the Java code compiles and the caller has permission to access the model
  - Code elements are immutable once constructed / built

What happens at compile time and run time

- Code models for @CodeReflection-annotated methods are generated and stored in class files
- If access checks allow, a transformer can request the code model for a method
- The transformer may then, for example:
  - translate the method's code model to another language for execution on a non-JVM target
  - transform the method into an alternate Java method and execute that on the JVM



Does this change Java?

- A normal Java call to a method will execute the Java code exactly as the Java language and JVM specify
- The ability to code-reflect on a method is rooted in the presence of a @CodeReflection annotation in the source
- Doing something with the method, e.g., executing a transformed version, will require an explicit user action, e.g., a method call on the transformer
- Not unlike the FFM API, Code Reflection can help Java developers participate outside Java's traditional programming domains without changing the Java behavior we all count on

Updates since JVMLS

- Public GitHub repository with code reflection API implementation
  - ongoing expansion of modeled Java language elements
  - extensible types, e.g., in support of code reflection dialects
  - LINQ example code
  - Java Triton frontend code
  - Java to SPIR-V example code
- Blog articles on
  - auto-differentiation
  - the LINQ example
  - the Triton example

SPIR-V Example – What is SPIR-V

- Standard Portable Intermediate Representation
- An abstract assembly language, similar to LLVM
- Designed for parallel computing and graphics by Khronos Group
- Accepted by Vulcan, OpenGL, OpenCL, and oneAPI frameworks
- Exchanged in binary form
- More information at https://www.khrnos.org/spir

Example – translate Java to SPIR-V binary

```
@CodeReflection
public static int loop(int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += i;
  }
  return sum;
}


// Example Java code to translate “loop” method to SPIR-V dialect then to SPIR-V binary
public void test() throws Exception {
    String methodName = “loop";
    Method method = LoopTest.class.getDeclaredMethod(methodName, int.class);

    CoreOps.FuncOp javaFunc = method.getCodeModel().get();
    SpirvOps.FuncOp spirvFunc = TranslateToSpirvModel.translateFunction(javaFunc);
    MemorySegment spirvBinary = SpirvModuleGenerator.generateModule(methodName, spirvFunc);

    System.out.println(javaFunc.toText());
    System.out.println(spirvFunc.toText());
    System.out.println(SpirvModuleGenerator.disassembleModule(spirvBinary));
}
```

Example – three dialects

```
------- Java high-level dialect -------
func @"loop" (%0 : int)int -> {
    %1 : Var<int> = var %0 @"size";
    %2 : int = constant @"0";
    %3 : Var<int> = var %2 @"sum";
    java.for
        ()Var<int> -> {
            %4 : int = constant @"0";
            %5 : Var<int> = var %4 @"i";
            yield %5;
        }
        (%6 : Var<int>)boolean -> {
            %7 : int = var.load %6;
            %8 : int = var.load %1;
            %9 : boolean = lt %7 %8;
            yield %9;
        }
        (%10 : Var<int>)void -> {
            %11 : int = var.load %10;
            %12 : int = constant @"1";
            %13 : int = add %11 %12;
            var.store %10 %13;
            yield;
        }
        (%14 : Var<int>)void -> {
            %15 : int = var.load %3;
            %16 : int = var.load %14;
            %17 : int = add %15 %16;
            var.store %3 %17;
            java.continue;
        };
    %18 : int = var.load %3;
    return %18;
};
```

```
----- Java low-level dialect -----
func @"loop" (%0 : int)int -> {
    %1 : Var<int> = var %0 @"size";
    %2 : int = constant @"0";
    %3 : Var<int> = var %2 @"sum";
    %4 : int = constant @"0";
    %5 : Var<int> = var %4 @"i";
    branch ^block_0;
  ^block_0:
    %6 : int = var.load %5;
    %7 : int = var.load %1;
    %8 : boolean = lt %6 %7;
    cbranch %8 ^block_1 ^block_2;
  ^block_1:
    %9 : int = var.load %3;
    %10 : int = var.load %5;
    %11 : int = add %9 %10;
    var.store %3 %11;
    branch ^block_3;
  ^block_3:
    %12 : int = var.load %5;
    %13 : int = constant @"1";
    %14 : int = add %12 %13;
    var.store %5 %14;
    branch ^block_0;
  ^block_2:
    %15 : int = var.load %3;
    return %15;
};
```

```
------- SPIR-V dialect -------
spirv.function_loop (%0 : int)int -> {
    %1 : int = spirv.function parameter;
    %2 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @size;
    %3 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @sum;
    %4 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @i;
    spirv.store %2 %1;
    %5 : int = spirv.constant @”0”;
    spirv.store %3 %5;
    %6 : int = spirv.constant @”0”;
    spirv.store %4 %6;
    spirv.br ^block_0;
  ^block_0:
    %7 : int = spirv.load %4;
    %8 : int = spirv.load %2;
    %9 : int = spirv.lt %7 %8;
    spirv.brcond %9 ^block_1 ^block_2;
  ^block_1:
    %10 : int = spirv.load %3;
    %11 : int = spirv.load %4;
    %12 : int = spirv.iadd %10 %11;
    spirv.store %3 %12;
    spirv.br ^block_3;
  ^block_3:
    %13 : int = spirv.load %4;
    %14 : int = spirv.constant @”1”;
    %15 : int = spirv.iadd %13 %14;
    spirv.store %4 %15;
    spirv.br ^block_0;
  ^block_2:
    %16 : int = spirv.load %3;
    return %16;
};
```

Example – SPIR-V dialect -> binary

```
------- SPIR-V dialect -------
spirv.function_loop (%0 : int)int -> {
    %1 : int = spirv.function parameter;
    %2 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @size;
    %3 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @sum;
    %4 : spirv.pointer<int, CrossWorkgroup> = spirv.variable @i;
    spirv.store %2 %1;
    %5 : int = spirv.constant @"0";
    spirv.store %3 %5;
    %6 : int = spirv.constant @"0";
    spirv.store %4 %6;
    spirv.br ^block_0;
  ^block_0:
    %7 : int = spirv.load %4;
    %8 : int = spirv.load %2;
    %9 : int = spirv.lt %7 %8;
    spirv.brcond %9 ^block_1 ^block_2;
  ^block_1:
    %10 : int = spirv.load %3;
    %11 : int = spirv.load %4;
    %12 : int = spirv.iadd %10 %11;
    spirv.store %3 %12;
    spirv.br ^block_3;
  ^block_3:
    %13 : int = spirv.load %4;
    %14 : int = spirv.constant @"1";
    %15 : int = spirv.iadd %13 %14;
    spirv.store %4 %15;
    spirv.br ^block_0;
  ^block_2:
    %16 : int = spirv.load %3;
    return %16;
};
```

```
------- SPIR-V Module -------
      OpCapability Addresses
      OpCapability Linkage
      OpCapability Kernel
      OpCapability Int8
      OpCapability Int64
 %1 = OpExtInstImport "OpenCL.std"
      OpMemoryModel Physical64 OpenCL
      OpEntryPoint Kernel %9 "loop" %2 %3 %4 %5
      OpName %20 "size"
      OpName %21 "sum"
      OpName %22 "i"
      OpName %27 "bool"
      OpName %31 "float"
      OpDecorate %4 Constant
      OpDecorate %2 BuiltIn GlobalInvocationId
 %8 = OpTypeInt 64 0
 %7 = OpTypeVector %8 3
 %6 = OpTypePointer Input %7
 %2 = OpVariable %6 Input
 10 = OpTypeInt 32 0
%11 = OpTypeFunction %10 %10
%19 = OpTypePointer Function %10
%23 = OpConstant %10 0
%24 = OpConstant %10 0
%27 = OpTypeBool
%31 = OpTypeFloat 32
%34 = OpConstant %10 1
 %9 = OpFunction %10 DontInline %11
%18 = OpFunctionParameter %10
%12 = OpLabel
%20 = OpVariable %19 Function
%21 = OpVariable %19 Function
%22 = OpVariable %19 Function
      OpStore %20 %18 Aligned 8
      OpStore %21 %23 Aligned 8
      OpStore %22 %24 Aligned 8
      OpBranch %13
%13 = OpLabel
%25 = OpLoad %10 %22 Aligned 4
%26 = OpLoad %10 %20 Aligned 4
%28 = OpSLessThan %27 %25 %26
      OpBranchConditional %28 %14 %16
%14 = OpLabel
%29 = OpLoad %10 %21 Aligned 4
%30 = OpLoad %10 %22 Aligned 4
%32 = OpIAdd %10 %29 %30
      OpStore %21 %32 Aligned 8
      OpBranch %15
%15 = OpLabel
%33 = OpLoad %10 %22 Aligned 4
%35 = OpIAdd %10 %33 %34
      OpStore %22 %35 Aligned 8
      OpBranch %13
%16 = OpLabel
%36 = OpLoad %10 %21 Aligned 4
      OpReturnValue %36
      OpFunctionEnd
```

Babylon Resources

See the Babylon Project page (openjdk.org/projects/babylon) for pointers to :

- the dev mailing list
- the GitHub repo
- blog articles showing example use of code reflection
  - automatic differentiation
  - C# LINQ
  - Triton GPU programming
- Paul Sandoz's JVMLS code reflection presentation
- Gary Frost's JVMLS Java GPU programming presentation

Notices and Disclaimers

End