Babylon — Code Reflection

Paul Sandoz

JVM Language Summit
August 5–7, 2024

Babel Tower — Shirin Abedinirad, Guglielmo Torelli

https://www.shirinabedinirad.com/portfolio/babel-tower/

Overview

Status

Code models

Reflecting code of methods

Transforming code models

Reflecting code of lambdas

Status — JVMLS 2023 → JVMLS 2024

- (JVMLS 2023 video and slides)
- Created OpenJDK Project Babylon (Oct 23)
- Open sourced code reflection (Jan 24)
  - Significant progress on the implementation and design
- Published multiple articles on code reflection (Feb/Jun 24)
  - On example use cases and code reflection design (linked from project page)
- Open sourced **H**eterogenous **A**ccelerator **T**oolkit (**HAT**) (May 24)
  - A parallel programming model for GPUs, leveraging Project Panama's FFM API and code reflection, ensuring code reflection is fit-for-purpose

Code reflection

- Enhances Java reflection, giving access to Java code in *identified* method bodies and lambda bodies
  - Giving access at compile time and run time
- Enables the support of foreign programming models via *transformation*
  - Those that differ from the Java programming model
- The reflected Java code is represented *symbolically* as a **code model**
  - A code model consists of instances of Java classes, code elements and items, arranged in an immutable *tree structure* that is suitable for *analysis* and *transformation* by Java programs
  - Code model design is inspired by LLVM-based compilers and MLIR
- APIs to build, traverse, and transform code models

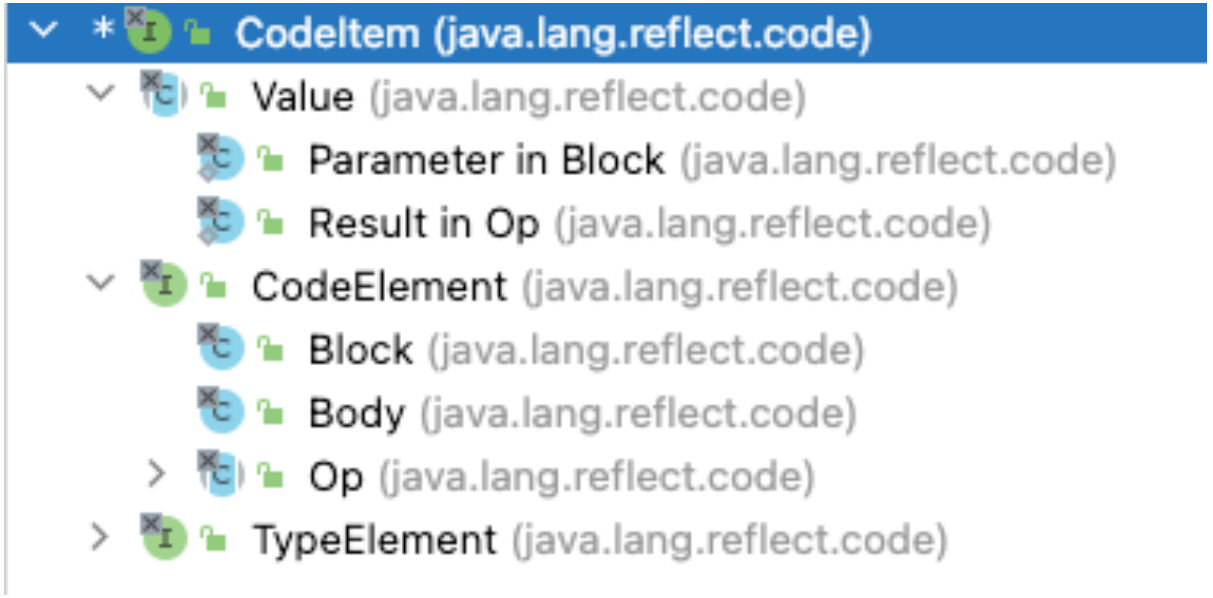
Code model design

- A code model contains operations, bodies, and blocks, that form a tree
  - An operation contains zero or more bodies. A body contains one or more blocks. A block contains a sequence of one or more operations
- Blocks and operations can declare values
  - A block can declare zero or more block parameter values
  - An operation declares an operation result value
- Values can only be assigned exactly once on declaration
  - Code models are in Static Single Assignment (SSA) form
- An operation may use values as operands
  - Only after the values have been declared and only if they dominate
- A value has a type

Code model design

- The blocks within a body are interconnected with each other and form a control flow graph
  - The last (terminating) operation in a block can jump to another block and pass values as block arguments
- Values are also interconnected with each other and form either *expression* graphs or *use* graphs
  - The relationship between an operation result and its operands are edges of an expression graph
  - The relationship between a value and its uses are edges of a use graph

Code model design — API class hierarchy



- Instances of `CodeElement` are nodes in a code model tree
- `Op` (operation) and `TypeElement` (value type) are unsealed and extensible, all other types are sealed or final
- Code item types modeling specific Java program behavior extend from `Op` and `TypeElement`

Java code models

- A Java code model is a symbolical description of a Java program
  - Contains operations and types modeling Java program behavior arranged to preserve Java program meaning
  - Java language declarations, expressions, statements, and specific behaviors, are modeled as classes extending from `Op` (operation)
  - Denotable Java types are modeled as classes extended from `TypeElement` (value type)
- The set of operations and types that model some kind of program behavior is referred to as a (code model) dialect
  - The Java dialect models Java program behavior

Java code models

- Modeling-wise somewhere between the compiler's **A**bstract **S**yntax **T**ree (AST) and bytecode
  - Easier to analyze and transform
  - "Just the right amount"
- Preserves more information than bytecode with less surface syntax details of the AST
  - Retaining code structure and denotable types
- Standardized modeling and access
  - Compiler's AST is not standardized and completely accessible
  - Bytecode is not generally accessible at runtime

Method identification and access

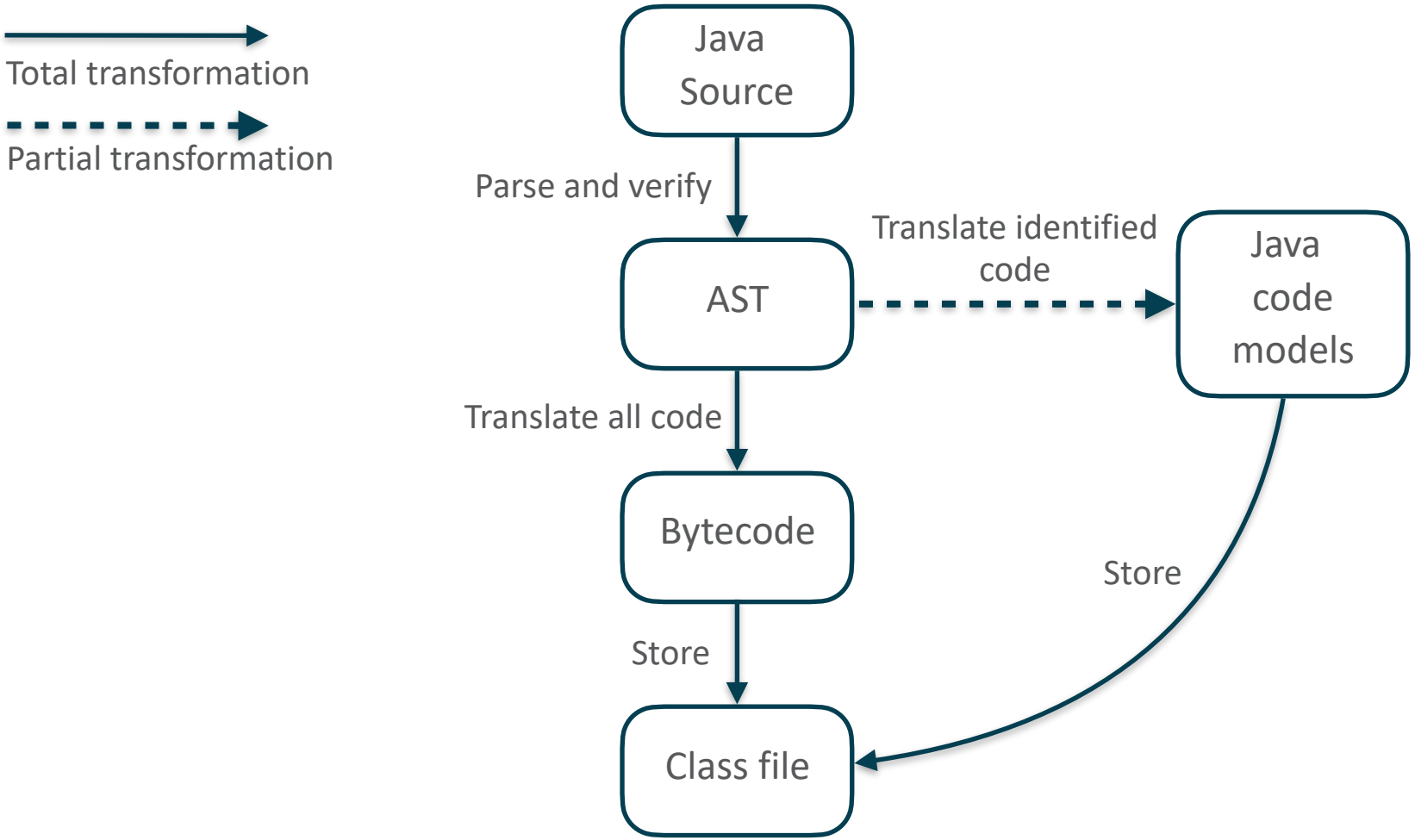
- Identify a method by annotating it

```
@CodeReflection
static double add(double a, double b) {
    return a + b;
}
```

- Access the code model using reflection

```
// Reflect on "add" method
Method addMethod = MethodExample.class.getDeclaredMethod(
        "add", double.class, double.class);
// Get "add"'s code model, not all methods are identified to reflect on their bodies
Optional<? extends Op> optionalModel = addMethod.getCodeModel();
Op addCodeModel = optionalModel.orElseThrow();
```


Source compilation



Text of code model stored in synthetic field

```
Classfile .../MethodExample.class
...
Constant pool:
...
  #53 = String             #54
  // Code model text string
  #54 = Utf8               func @\"add\" @loc=\"11:5:file:/.../MethodExample.java\" ...
  #55 = Fieldref           #7.#56
  #56 = NameAndType        #57:#58
  #57 = Utf8               jvmls24$MethodExample::add(double, double)double$op
...
{
  static final java.lang.String jvmls24$MethodExample::add(double, double)double$op;
    descriptor: Ljava/lang/String;
    flags: (0x1018) ACC_STATIC, ACC_FINAL, ACC_SYNTHETIC
...
  static {};
    descriptor: ()V
    flags: (0x0008) ACC_STATIC
    Code:
      stack=1, locals=0, args_size=0
         0: ldc           #53
         // Store to field "jvmls24$MethodExample::add(double, double)double$op":Ljava/lang/String;
         2: putstatic     #55
```

Detour — what does add's code model look like?

- We could extract the text string from the class file or we could directly convert the operation to a human readable text string — the two are the identical

```
@CodeReflection
static double add(double a, double b) {
    return a + b;
}

Op addCodeModel = optionalModel.orElseThrow();
System.out.println(addCodeModel.toText());

func @"add" @loc="11:5:file:/.../MethodExample.java" (%0 : double, %1 : double)double -> {
    %2 : Var<double> = var %0 @"a" @loc="11:5";
    %3 : Var<double> = var %1 @"b" @loc="11:5";
    %4 : double = var.load %2 @loc="13:16";
    %5 : double = var.load %3 @loc="13:20";
    %6 : double = add %4 %5 @loc="13:16";
    return %6 @loc="13:9";
};
```

Summary of method identification and access

- The source compiler
  - Translates AST node of the `add` method to a code model
  - Converts the code model to text string and assigns that string to a synthetic field in the class initializer of the class file
- The runtime
  - Accesses the synthetic field to obtain the text string
  - Parses the text string to produce a code model

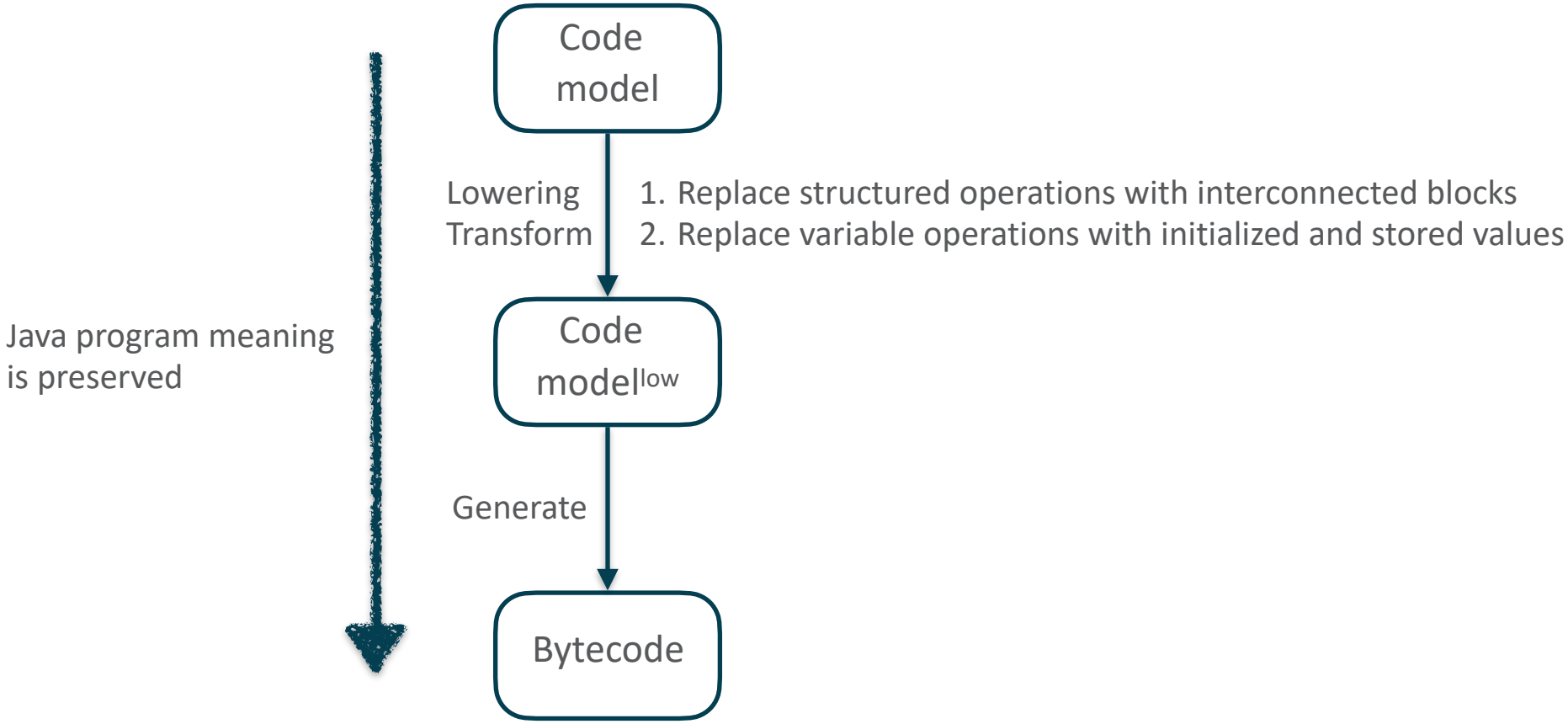
Issues with method identification and access

- We are likely misusing annotations, since they should not affect platform semantics
  - Compilation may fail if the modeling of some language construct is unsupported
  - Need a keyword?
- Storing the text string in a synthetic field is convenient for prototyping but not a good the final solution
  - Best to avoid standardizing the field name mangling and format of the text string so it can be parsed, it’s primarily intended for debugging and testing
  - An alternative solution is to transform the code model to a code model that builds itself using the code model builder API, translate that model to bytecode, embed as a synthetic method in the class file, and reference via an attribute

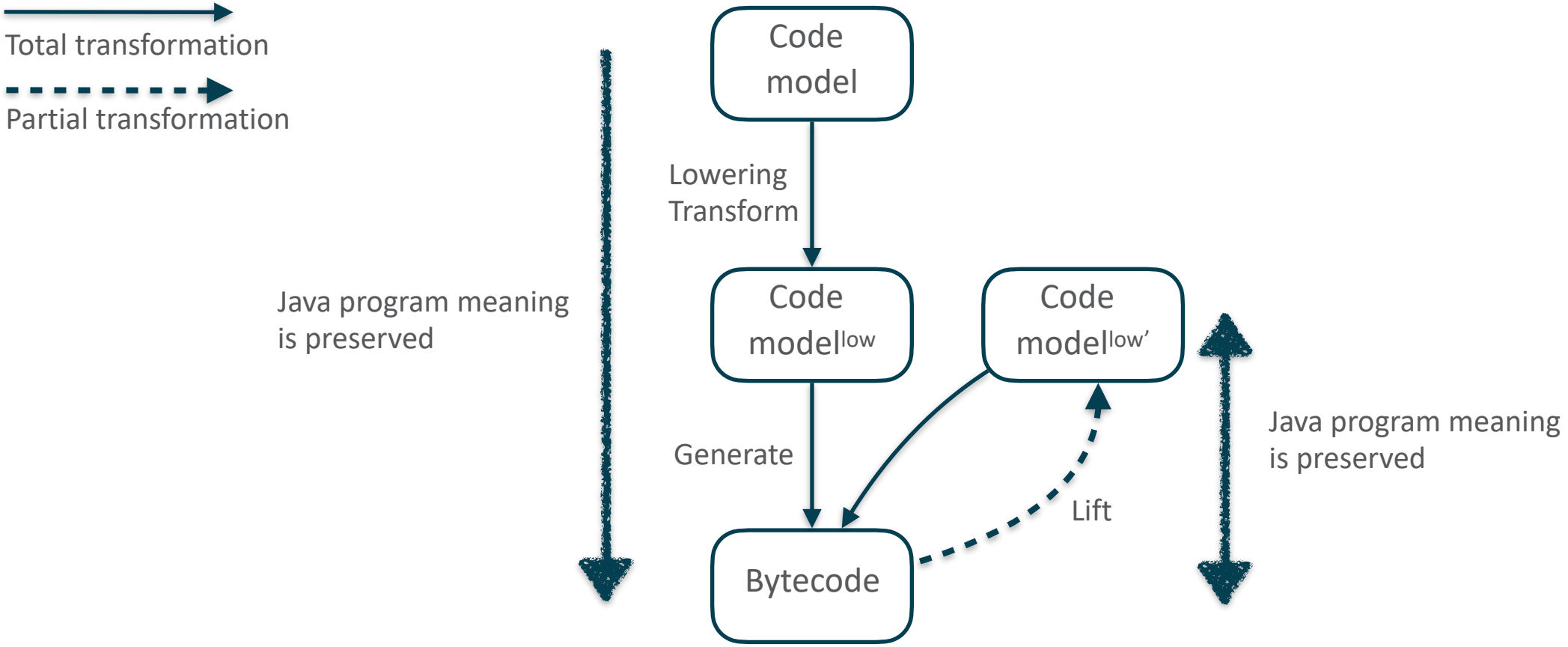
Issues with method identification and access

- There are two representations of the same code in the class file
  - Can we unify by storing the only the code model and dynamically transform it to bytecode on-demand? In some cases we will never process the bytecode
  - Bytecode and code models could get out of sync with class file transformations — but that can happen for other things too
- We also need to validate code models loaded from class files
  - Ensuring the model conforms to that produced from the compiler, as specified by the JLS
  - Code models have fundamental structural constraints, and then there are operational constraints for modeling Java language constructions

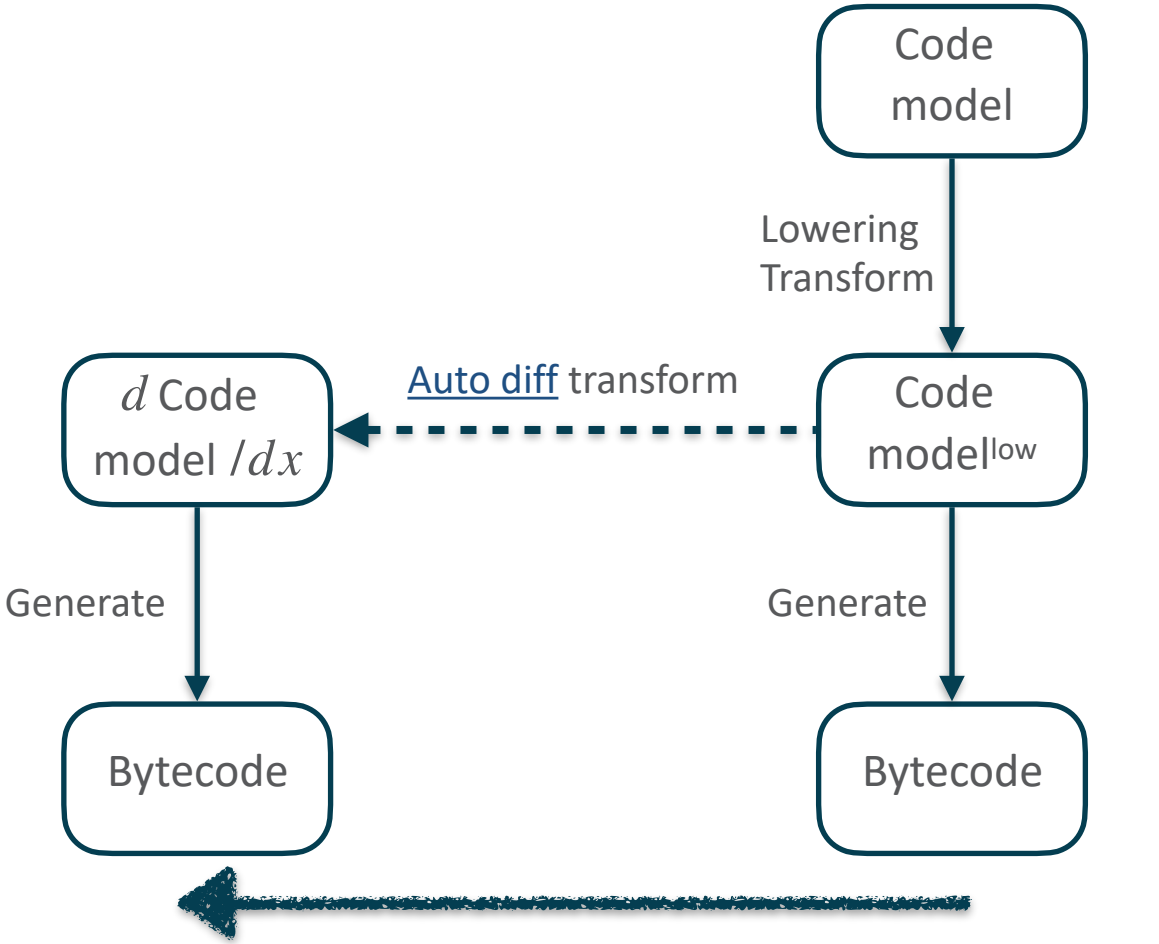
Transform to bytecode



Lift from bytecode

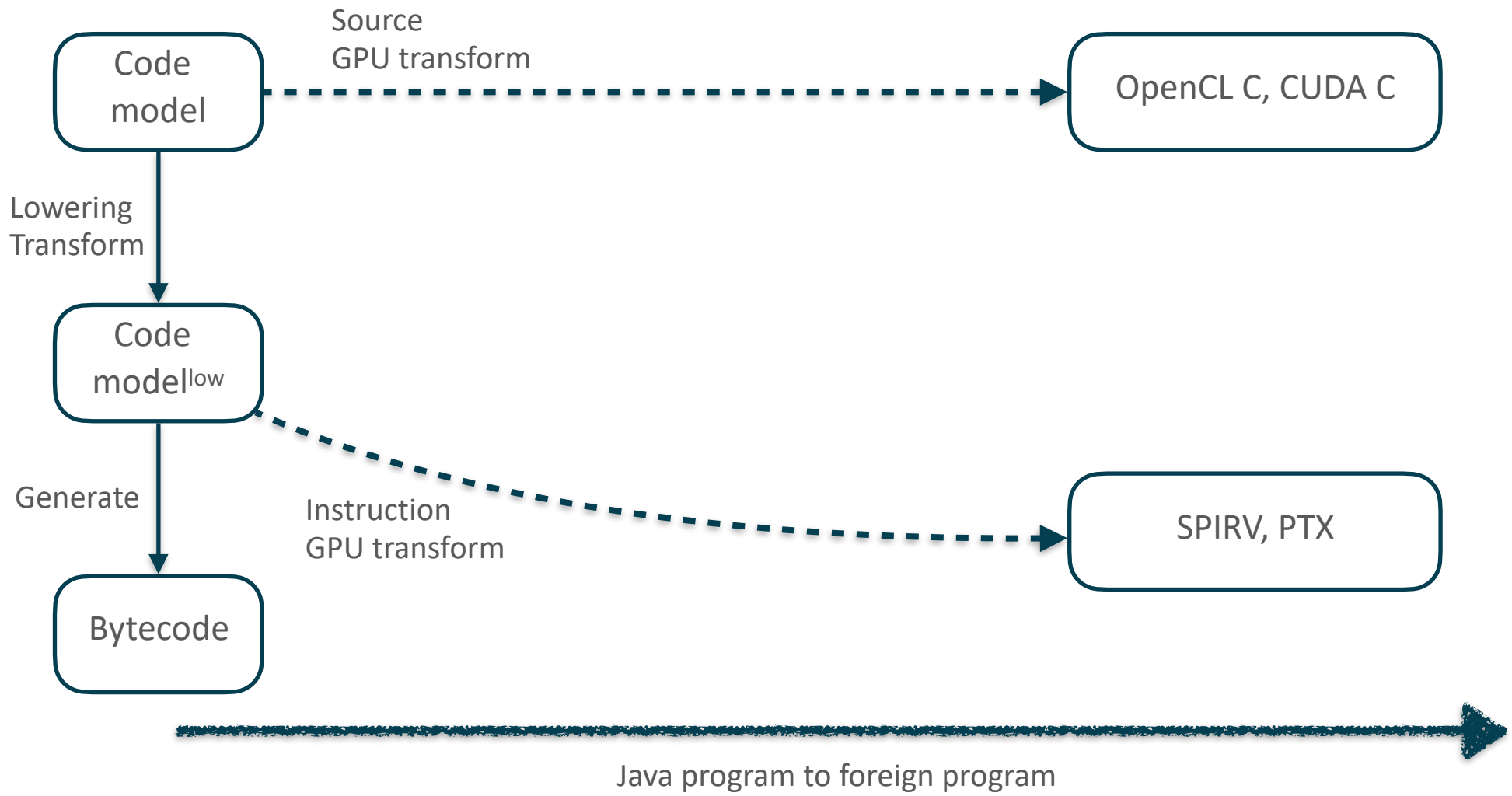


Differentiate a Java program

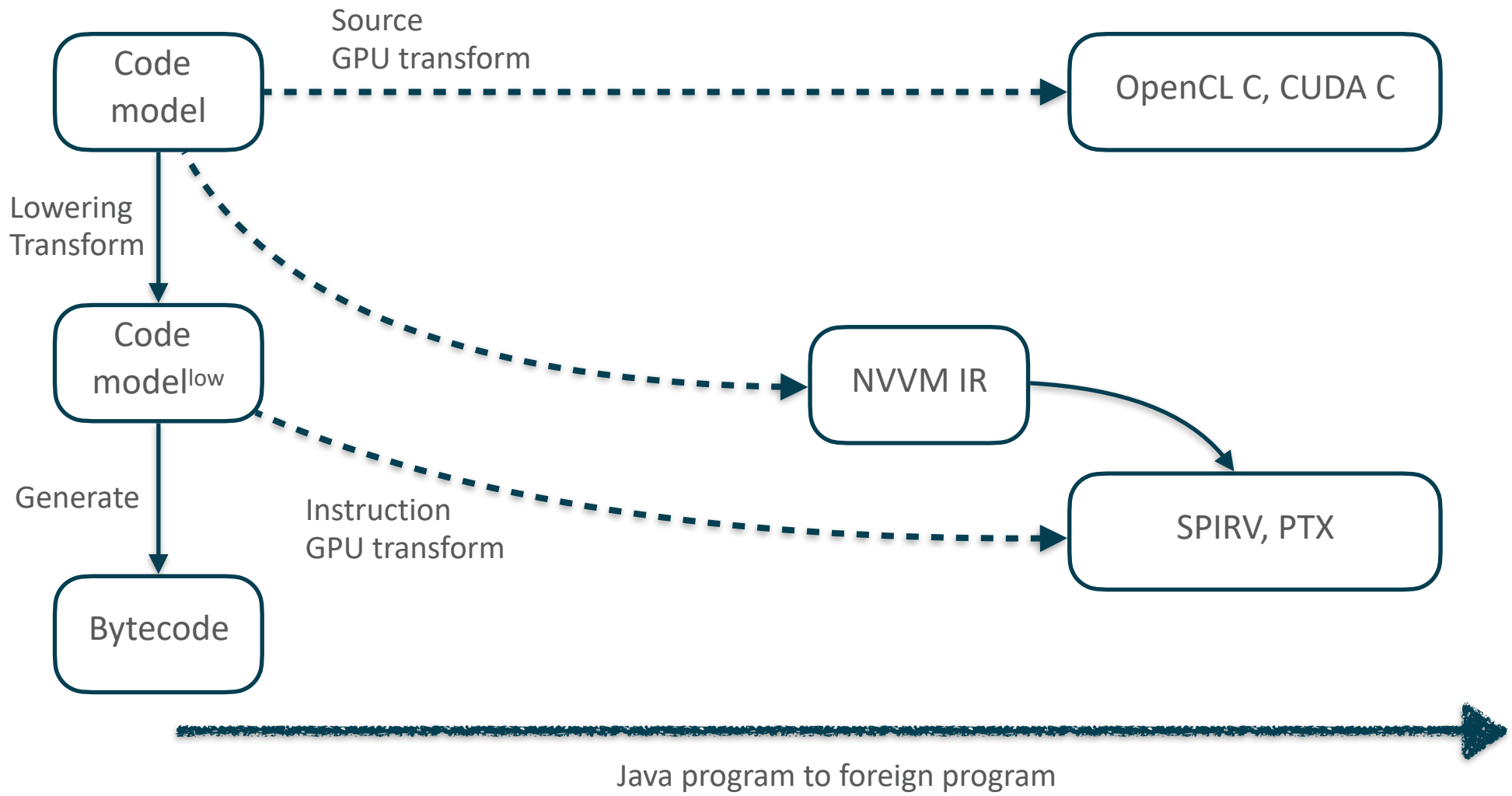


Java program to Java program with foreign programming model

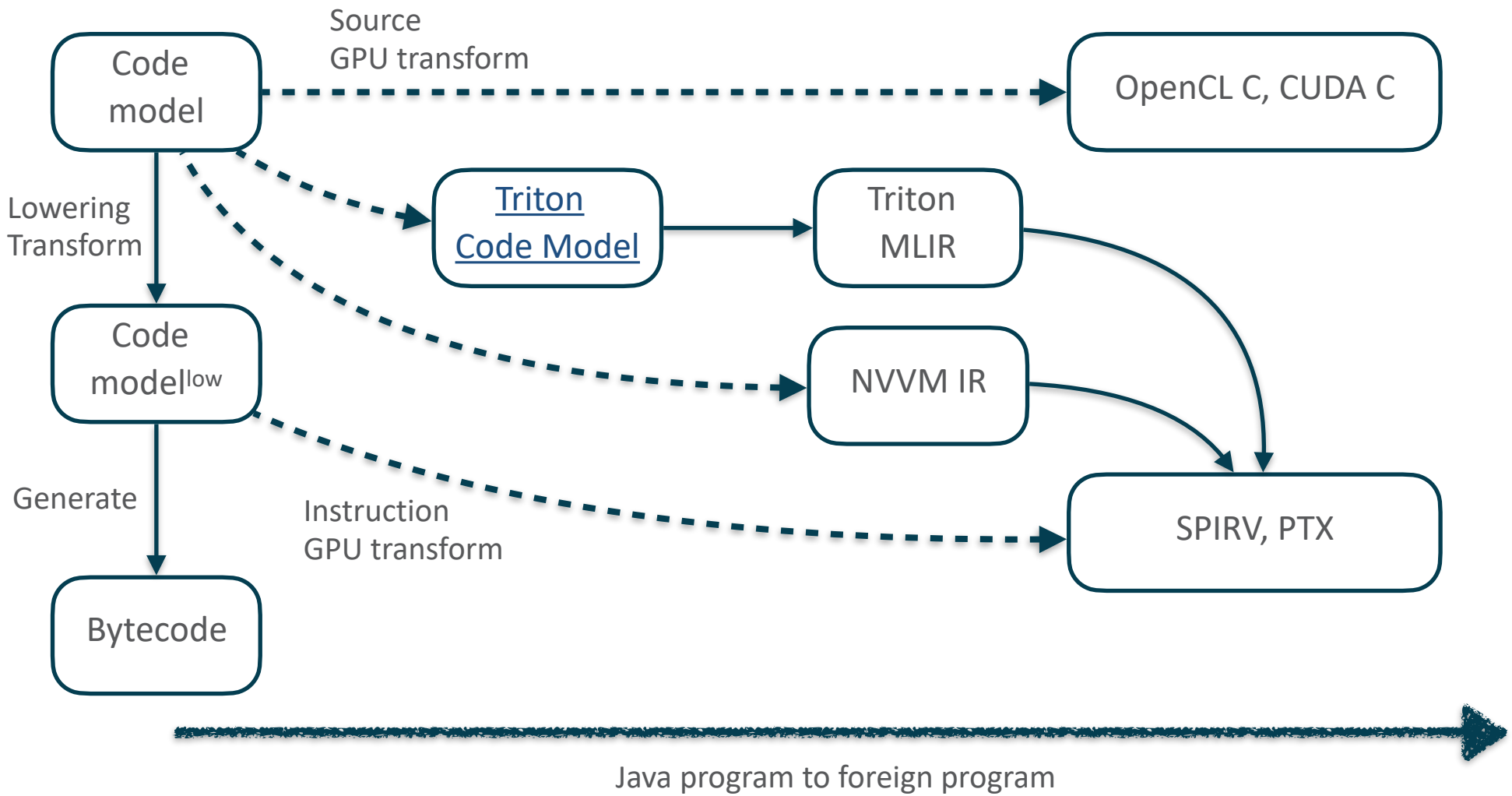
Generate GPU kernels



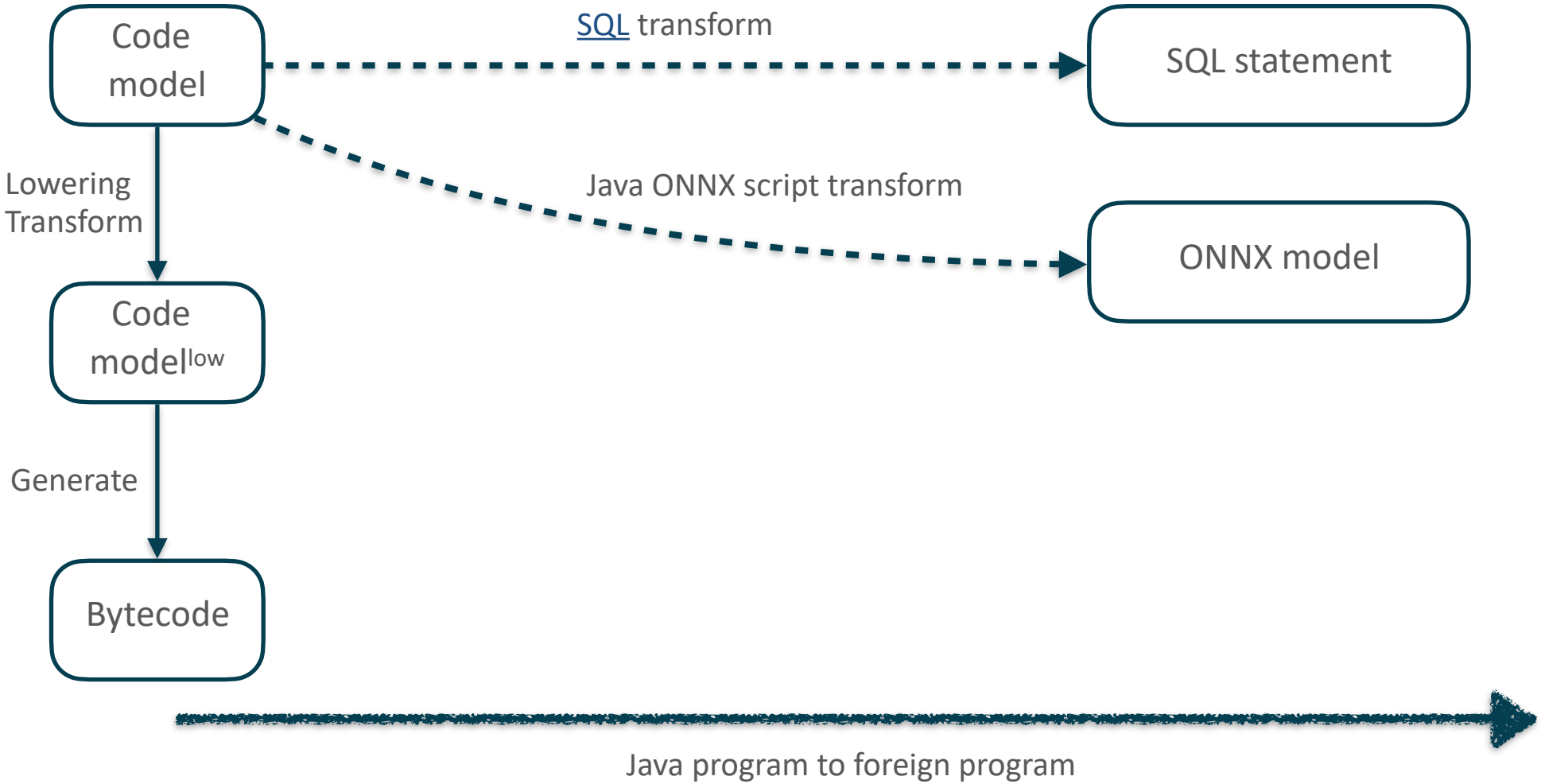
Generate GPU kernels



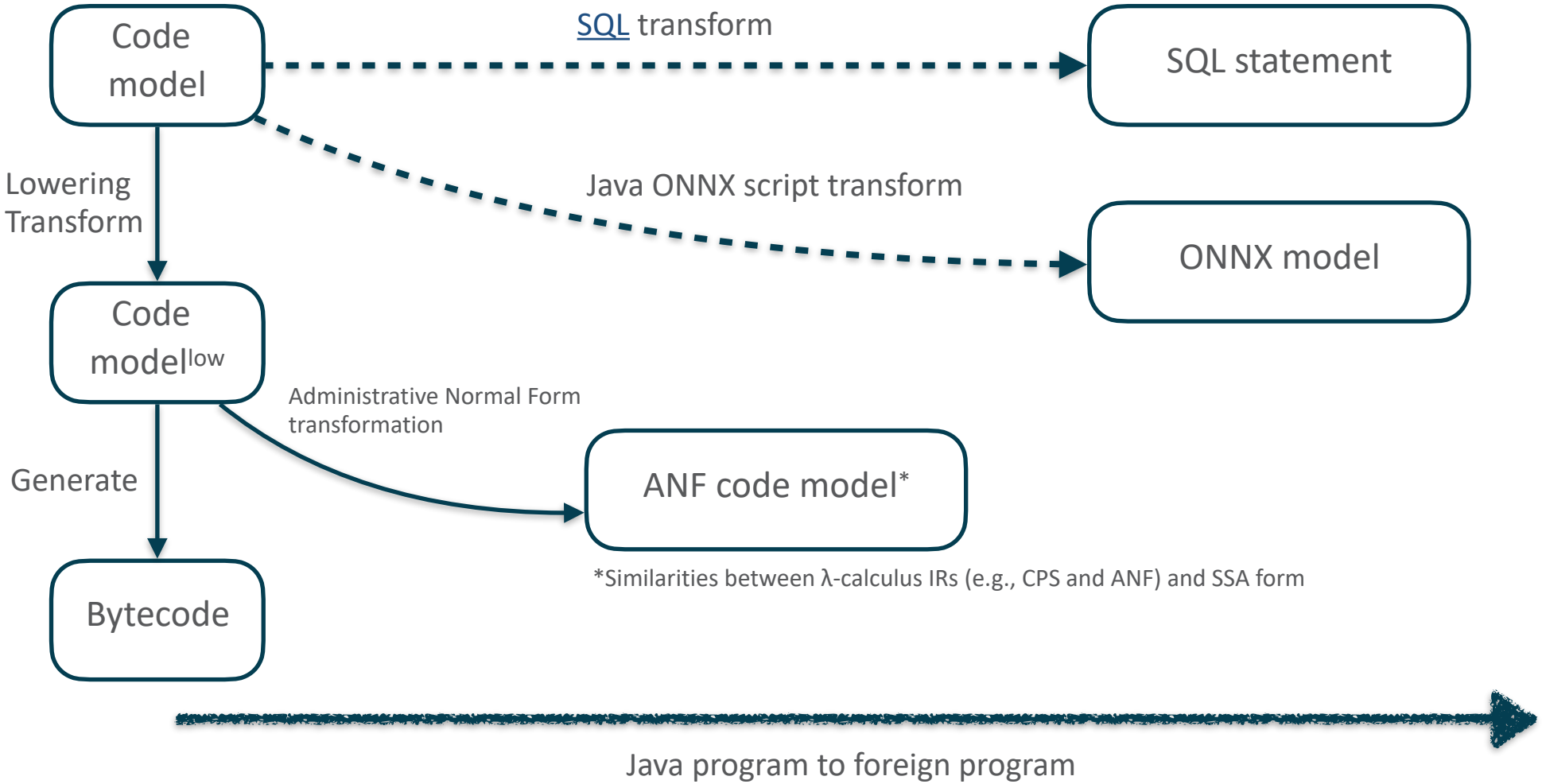
Generate GPU kernels



Interfacing with other programming domains



Interfacing with other programming domains



Transformation examples

- Lowering

```
CoreOp.FuncOp model = ...;

// Code models in, code models out
CoreOp.FuncOp loweredModel = model.transform(OpTransformer.LOWERING_TRANSFORMER);
CoreOp.FuncOp loweredSSAModel = SSA.transform(loweredModel);
```

- Bytecode generation

```
CoreOp.FuncOp model = ...;

// Bytecode generator only works with lowered models
CoreOp.FuncOp loweredModel = model.transform(OpTransformer.LOWERING_TRANSFORMER);
MethodHandle handle = BytecodeGenerator.generate(MethodHandles.lookup(), loweredModel);
```

- Lifting

```
CoreOp.FuncOp model = ...;

CoreOp.FuncOp loweredModel = model.transform(OpTransformer.LOWERING_TRANSFORMER);
byte[] classData = BytecodeGenerator.generateClassData(MethodHandles.lookup(), loweredModel);
CoreOp.FuncOp liftedLoweredModel = BytecodeLift.lift(classData, loweredModel.funcName());
```

Transformation examples

- Forward auto differentiation

```
// f(x, y)
CoreOp.FuncOp f = ...;

f = SSA.transform(f.transform(OpTransformer.LOWERING_TRANSFORMER));

Block.Parameter x = f.parameters().get(0);
CoreOp.FuncOp df_dx = ExpressionElimination.eliminate(ForwardDifferentiation.partialDiff(f, x));
MethodHandle df_dx_h = BytecodeGenerator.generate(MethodHandles.lookup(), df_dx);

Block.Parameter y = f.parameters().get(1);
CoreOp.FuncOp df_dy = ExpressionElimination.eliminate(ForwardDifferentiation.partialDiff(f, y));
MethodHandle df_dy_h = BytecodeGenerator.generate(MethodHandles.lookup(), df_dy);
```

- See test here

Lambda expression identification and access

- Identify by target typing using intersection types

```
DoubleBinaryOperator f = (DoubleBinaryOperator & Quotable) (a, b) -> a + b;
```

- Access the code model via the targeted type using a cast

```
Quoted qF = ((Quotable) f).quoted();
Op op = qF.op();
```


Lambda expression identification and access

- Identify by target typing using quotable function interface

```
@FunctionalInterface
interface QuotedDoubleBinaryOperator extends DoubleBinaryOperator, Quotable { }

QuotedDoubleBinaryOperator f = (a, b) -> a + b;
```

- Access the code model via the targeted type

```
Quoted qF = f.quoted();
Op op = qF.op();
```

- From `Quotable` we can get what code is `Quoted`
  - `Quotable` behaves similarly to `Serializable`
- `Quoted` encapsulates the lambda expression's code model and any *captured* values

Lambda expression identification and access

- Capturing lambda expression

```
public void capture(double cap) {
    QuotedDoubleBinaryOperator f = (a, b) -> a + b + cap;
    ...
```

- Access the code model and captured arguments

```
Quoted qF = f.quoted();
// Map of symbolic value to runtime value
Map<Value, Object> capturedValues = qF.capturedValues();
Op op = qF.op();
```

- The captured values can be applied to the model when processed
  - The model is static, the captured values dynamic
- No code outside the lambda expression is captured and reflected on

Uses the more flexible the lambda metafactory

```
public void capture(double);
  descriptor: (D)V
...
       0: dload_1  // Captured argument for lambda proxy class
       1: dload_1  // Captured argument for code model
       // InvokeDynamic #2:applyAsDouble:(DD)Ljvmls24/LambdaExample$QuotedDoubleBinaryOperator;
       2: invokedynamic #57,  0
       7: astore_3
       8: load_3
       // InterfaceMethod jvmls24/LambdaExample$QuotedDoubleBinaryOperator.quoted:()Ljava/lang/reflect/code/Quoted;
       9: invokeinterface #46,  1
...
BootstrapMethods:
...
  2: #227 REF_invokeStatic java/lang/invoke/LambdaMetafactory.altMetafactory:(...;
    Method arguments:
      #208 (DD)D
      #219 REF_invokeStatic jvmls24/LambdaExample.lambda$capture$2:(DDD)D
      #208 (DD)D
      // Additional flags set for quotable lambda
      #212 12
      #213 0
      // Reference to field holding the code model text string
      #222 REF_getStatic jvmls24/LambdaExample.lambda$2$op:Ljava/lang/String;
```


The code model referenced by the metafactory

```
func @"quotedLambda" (%0 : double)java.lang.reflect.code.Quoted -> {
    %1 : Var<double> = var %0 @"cap";
    %2 : java.lang.reflect.code.Quoted = quoted ()void -> {
        %3 : jvmls24.LambdaExample$QuotedDoubleBinaryOperator = lambda (%4 : double, %5 : double)double -> {
            %6 : Var<double> = var %4 @"a";
            %7 : Var<double> = var %5 @"b";
            %8 : double = var.load %6;
            %9 : double = var.load %7;
            %10 : double = add %8 %9;
            %11 : double = var.load %1";
            %12 : double = add %10 %11";
            return %12;
        };
        yield %3;
    };
    return %2;
};
```

- The generated lambda proxy class interprets this model to produce a `Quoted` instance, passing the captured values as (function) block arguments
  - The symbolic value %1, modeling the captured variable, is bound to its runtime value

Method reference identification and access

- Method references are modeled as desugared lambda expressions

```
@CodeReflection
static double add(double a, double b) {
    return a + b;
}

QuotedDoubleBinaryOperator f1 = LambdaExample::add;
QuotedDoubleBinaryOperator f2 = (a, b) -> add(a, b);

// op1 and op2 are identical in their content
Op op1 = f1.quoted().op();
Op op2 = f2.quoted().op();
```

Method reference identification and access

- We can “trampoline” to the the code model of method add

```
QuotedDoubleBinaryOperator f1 = LambdaExample::add;
CoreOp.LambdaOp lop = (CoreOp.LambdaOp) f1.quoted().op();

// Determine if this lambda is like a method reference
// by obtaining an optional invoke operation to the
// referenced method
Optional<CoreOp.InvokeOp> iop_ = lop.methodReference();
CoreOp.InvokeOp iop = iop_.orElseThrow();

// Resolve the method being invoked
Method addMethod = (Method) iop.invokeDescriptor().resolveToMember(
        MethodHandles.lookup());

// Get the invoked method's code model, as previously shown
Optional<? extends Op> optionalModel = addMethod.getCodeModel();
Op addCodeModel = optionalModel.orElseThrow();

// This will get easier with member patterns!
```


Detour — Call graph trampolining

- Create a code model for the methods in the call graph

```
@CodeReflection
static void m(int i, List<Integer> l) {
    if (i < 0) {
        return;
    }

    n(i - 1, l);
}

@CodeReflection
static void n(int i, List<Integer> l) {
    l.add(i);
    m(i - 1, l);
}
```

```
module ()void -> {
    func @"transinvoke.TransitiveInvokeModuleTest::m(int, java.util.List)void"
    (%0 : int, %1 : java.util.List<java.lang.Integer>)void -> {
        %2 : int = constant @"0"
        %3 : boolean = lt %0 %2
        cbranch %3 ^block_0_1 ^block_0_2;

      ^block_0_1:
        return

      ^block_0_2:
        branch ^block_0_3;

      ^block_0_3:
        %4 : int = constant @"1"
        %5 : int = sub %0 %4
        func.call %5 %1 @"transinvoke.TransitiveInvokeModuleTest::n(int, java.util.List)void";
        return
    };
    func @"transinvoke.TransitiveInvokeModuleTest::n(int, java.util.List)void"
    (%6 : int, %7 : java.util.List<java.lang.Integer>)void -> {
        %8 : java.lang.Integer = invoke %6 @"java.lang.Integer::valueOf(int)java.lang.Integer"
        %9 : boolean = invoke %7 %8 @"java.util.List::add(java.lang.Object)boolean"
        %10 : int = constant @"1"
        %11 : int = sub %6 %10
        func.call %11 %7 @"transinvoke.TransitiveInvokeModuleTest::m(int, java.util.List)void";
        return
    };
    unreachable;
};
```

- See test here

Structural target typing

- Target lambda expression directly to `Quoted`

```
public void structuralTarget(double cap) {
    Quoted qF = (double a, double b) -> a + b + cap;

    Map<Value, Object> capturedValues = qF.capturedValues();
    Op op = qF.op();
}
```

- Only the code model of of lambda expression generated
  - Bytecode is not generated
- Useful for quoting snippets of code that are then transformed and embedded (or inlined) when building other models
  - Let the compiler build the model snippets rather than doing so explicitly

Issues with lambda expression identification and access

- Same issues regarding code model storage
- Captured arguments list is duplicated
- Accessing `Quoted` via `Quotable::quoted`
  - Should `Quotable` be a marker interface, and we appeal to the reflection API to obtain the corresponding `Quoted` instance?
- Consider different names
  - e.g., `Reflectable` and `ReflectedLambda`?
  - Note that quoting is a more general concept in code models — any part of a model can be quoted using the `quoted` operation

`Serializable` and `Quotable` lambdas are similar

- Both trigger introspection of lambda expressions
- Both kinds of lambda proxy class expose the captured arguments
- `SerializedLambda` is to `Serializable` as `Quotable` is to `Quoted`
  - `SerializedLambda` encapsulates all the details to (re) invoke the LMF alternative bootstrap method — but only by the capturing class
  - `Quoted` encapsulates the code model — anyone with access to the quoted instance can obtain the model
  - Some developers use the former to “crack open” lambda expressions — what they really need is the latter, code reflection!
- Is there a common abstraction?

Reflective two-layered dispatch

```
interface QuotableConsumer<T> extends Consumer<T> { }

static class LayerOne {
    static void dispatch(QuotableConsumer<LayerOne> c) { ... }
    void dispatch(QuotableConsumer<LayerTwo> c) { ... }
}
static class LayerTwo { … }

public void testDispatch() {
    LayerOne.dispatch(l1 -> {
        // Layer one programming model with call graph

        l1.dispatch(l2 -> {
            // Layer two programming model with call graph
        });

        l1.dispatch(l2 -> {
            // Layer two programming model with call graph
        });
    });
}
```

Reflective two-layered dispatch

```
interface QuotableConsumer<T> extends Consumer<T>, Quotable { }

static class LayerOne {
    static void dispatch(QuotableConsumer<LayerOne> c) { ... }
    void dispatch(QuotableConsumer<LayerTwo> c) { ... }
}
static class LayerTwo { … }

public void testDispatch() {
    LayerOne.dispatch(l1 -> {
        // Layer one programming model with call graph

        l1.dispatch(l2 -> {
            // Layer two programming model with call graph
        });

        l1.dispatch(l2 -> {
            // Layer two programming model with call graph
        });
    });
}
```

- Can also include method trampolining
- Abstractly this is how the HAT programming model is arranged
- The host-compute program (l1) dispatches to one or more kernel programs (l2)

So much more left unsaid…

- Modeling of Java language constructs
  - JLS will specify what the compiler must do for identifying source, generating code models, accessing them, and storing them
  - JVMS will specify what the runtime must do to access and validate code models
- Core library APIs for code models
  - Code model structure
  - Operations and types modeling Java language constructs
  - Building, traversing, and transforming code models
  - Useful analyses and transformations
  - API design will be influenced by user-declared patterns (member patterns)


```
func @"catchThrow" @loc="14:5:file:/…/TestTry.java"
    (%0 : java.util.function.IntConsumer)void -> {
    %1 : java.lang.reflect.code.op.CoreOp$ExceptionRegion =
        exception.region.enter ^block_1 ^block_11 ^block_13;

  ^block_1:
    %2 : boolean = constant @"true" @loc="17:17";
    cbranch %2 ^block_2 ^block_5;

  ^block_2:
    exception.region.exit %1 ^block_3;

  ^block_3:
    %3 : int = constant @"2" @loc="28:22";
    invoke %0 %3
@"java.util.function.IntConsumer::accept(int)void"
@loc="28:13";
    branch ^block_4;

  ^block_4:
    branch ^block_9;

  ^block_5:
    branch ^block_6;

  ^block_6:
    %4 : java.lang.Runnable = lambda @loc="20:26" ()void -> {
        return @loc="21:17";
    };
    %5 : int = constant @"0" @loc="23:22";
    invoke %0 %5
@"java.util.function.IntConsumer::accept(int)void"
@loc="23:13";
    branch ^block_7;

  ^block_7:
    exception.region.exit %1 ^block_8;

  ^block_8:
    %6 : int = constant @"2" @loc="28:22";
    invoke %0 %6
@"java.util.function.IntConsumer::accept(int)void"
@loc="28:13";
    branch ^block_9;

  ^block_9:
    branch ^block_10;

  ^block_10:
    %7 : int = constant @"3" @loc="30:18";
    invoke %0 %7
@"java.util.function.IntConsumer::accept(int)void" @loc="30:9";
    return @loc="14:5";

  ^block_11(%8 : java.lang.IllegalStateException):
    %9 : java.lang.reflect.code.op.CoreOp$ExceptionRegion =
        exception.region.enter ^block_12 ^block_13;

  ^block_12:
    %10 : int = constant @"1" @loc="25:22";
    invoke %0 %10
@"java.util.function.IntConsumer::accept(int)void"
@loc="25:13";
    %11 : java.lang.RuntimeException = new
@"func<java.lang.RuntimeException>" @loc="26:19";
    throw %11 @loc="26:13";

  ^block_13(%12 : java.lang.Throwable):
    %13 : int = constant @"2" @loc="28:22";
    invoke %0 %13
@"java.util.function.IntConsumer::accept(int)void"
@loc="28:13";
    throw %12;
};
```