LLJava

Minimalist Structured Programming on the Java Virtual Machine [Work-in-Progress Research Paper]

Baltasar Trancón y Widemann
Ilmenau University of Technology
Ehrenbergstraße 29, 98693 Ilmenau
Germany
baltasar.trancon@tu-ilmenau.de

Markus Lepper
semantics GmbH
Berlin
Germany

ABSTRACT

There is a wide gap in abstraction level between Java source code and JVM bytecode. Many important software-related tasks, such as specification and implementation of code synthesis procedures, code inspection, software understanding and teaching, can benefit from an adequate, intermediate level of abstraction. Available bytecode assembly/disassembly tools are ad-hoc and fall short of the requirements regarding compositionality and clarity. We report on the design and implementation of the LLJava language that bridges the gap, based on careful analysis of bytecode information and rigorous design.

CCS Concepts

•**Software and its engineering** → **Compilers;** *Imperative languages; Control structures; Source code generation;* Software reverse engineering; Documentation;

Keywords

Java virtual machine, bytecode, assembler/disassembler, code model, ergonomics

1. INTRODUCTION

As intended by design, the Java Virtual Machine (JVM) has become an attractive target environment for more than vanilla Java programs. Experimental language extensions and complex metaprogramming systems abound, bytecode instrumentation and aspect weaving is performed routinely, and many alternative languages, both general and domain-specific in scope, are hosted on the Java runtime environment. But is the current situation convenient for the implementation, documentation and inspection of JVM-hosted languages? And if so, is Java source code or JVM bytecode the adequate representation of "trans-Java" programs?

PPPJ '16, August 29-September 02, 2016, Lugano, Switzerland

ISBN 978-1-4503-4135-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2972206.2972218

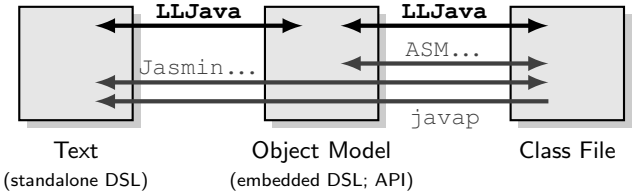

Figure 1: Formats and Tools (Outline)

Consider the implementation case first: should we compile to source code or to bytecode? In a heavyweight approach, it is possible to synthesize Java source code and hand it to the Java compiler. In a lightweight approach, it is likewise possible to produce JVM bytecode directly. There exist both object models (such as BCEL [2] and ASM [1]) and assemblers (such as Jasmin [3, 8], Oolong [5, 4], and Krakatau [6]) that handle the actual file format. Unfortunately, each of the extreme approaches has some significant drawbacks:

On the one hand, not everything legal on the JVM (as specified in [7], henceforth called JVMS) is convenient or even possible to express as a Java program, and there is no reliable way to direct `javac` to make particular choices in code generation. Furthermore, compiling Java source code to bytecode requires considerable resources, and may not be feasible for dynamic code bases or on constrained platforms.

On the other hand, bytecode generation, as raw bytes, API calls or assembly code, is burdened with an overwhelming number of low-level details to be filled in, in the right context and order. Diagnostic messages from JVM on-board bytecode verifiers are notoriously terse and hardly helpful, so the extra bits had better be consistent. Metaprogramming is obstructed by the lack of abstraction and compositionality.

The issue becomes even more pressing when documentation of language implementations and inspection of programs are considered: It is often necessary that programmers can understand the effect of compilation to the JVM in theory, and observe it in practice, respectively. Bytecode representations suffer from irrelevant verbosity and poor scalability regarding the expression of structure, from the perspective of the human reader and writer. This holds for the specified binary form but also for the largely homomorphic textual forms defined by ad-hoc tools such as `javap`. A simple verbatim translation of the binary data into some text format is insufficient for nontrivial practical needs; the level of abstraction and the applied strategies are crucial.

Here, we present efforts towards a solution, both in design and in implementation by a complete tool chain, as depicted in Figure 1. In contrast to the existing approaches, we emphasize the consistent treatment of redundancy, inference rules and abstractions in nomenclature, based on rigorous and explicit principles of analysis and design.

1.1 A Short Taxonomy of Bytecode Details

It is quite evident that the JVM bytecode as the low-level target format of a Java compiler effects more details than the Java language as its high-level source format; otherwise there would be little point in compilation. But on second thought, the details are there for a number of very different reasons. We discern four basic categories:

1. *Artifacts* of Java features defined by mostly source-level transformations (such as nested classes, enums, or annotations).
2. *Explication* of semantics (such as the disentanglement of control and data flow in expression evaluation by introduction of an operand stack).
3. *Organization* of code and metadata for efficient lookup, analysis and execution (such as operand stack limits, or exception tables for a method body).
4. *Encoding* of the complex term graph representing a class definition into a compact and platform-independent sequence of bytes (such as type expression mangling, or sharing of metadata elements in the constant pool).

We refer to the former two categories as *conceptual* details, and to the latter two as *technical* details, respectively. We postulate the working hypothesis for the present discussion:

> *A sweet spot on the continuum between extreme high- and low-level representations of the JVM capabilities is found by making conceptual details explicit, and technical details implicit, respectively.*

Existing bytecode tools and libraries already make some technical details implicit, particularly ones of category 4 (such as constant pool management), but fail to do so for others (such as exposing the user to mangled type expressions). The purpose of the present discussion, and of the LLJava project it reports on, is to complete the abstraction process, and to design a clear, convenient and self-explaining syntactic representation for the remaining conceptual details.

1.2 Motivating Example

Consider Figure 2: three similar Hello World examples retrieved from the repositories of various real-world byte code assembly language projects. Whitespace has been altered in some cases to align the corresponding content. The minor syntactic differences are indicated by superscripts. Note how syntactic traditions are mixed blithely in lines 1–2: `.class` is old-school assembler directive style, `public` is Java style (where the JVMS has `ACC_PUBLIC`), whereas finally `java/lang/Object` is JVM mangled style. More mangling is exposed in lines 4, 7, 9. All examples have inline allocation of a string constant in line 8, with no direct mention of constant pool management.

By contrast, lines 5–6 expose technical details, namely metadata on space resources used by the method. Quite tellingly, the major difference between the three examples is exactly here: The Oolong variant gives correct tight bounds for both operand stack (2) and local variables (1). The Krakatau variant gives safe loose bounds (10) for both, likely betraying the author's unwillingness to do the counting. The Jasmin variant omits the local variable bound altogether; it is trivially inferred from the method signature by the tool. Unfortunately, our experiments have revealed that the implemented inference procedure is too trivial: the distribution *jasmin-sable* 2.4.0-5 also allows omission of line 5, but unsoundly "infers" an operand stack bound of 1, which is duly refuted by bytecode verification.

For a JVM tool to produce up-to-date bytecode compatible with class file format versions greater than 50.0, a substantial amount of nontrivial type state inference must be performed anyway: The JVMS requires full type information on operand stack items and local variables of all basic blocks of control flow graphs, for efficient verification. From this tabulated type information not only resource limits, but operand types of most instructions follow implicitly. We hold that the same local type inference is naturally carried out by the human programmer. Hence a great deal of detail in bytecode instructions is technical, either in the sense of category 3 (where redundant type information is repeated for machine convenience) or category 4 (where common patterns are mapped to more compact alternative instructions).

As an illustration of consequences for language design, contrast Figure 2 with Figure 3, which mimics the same example in our proposed intermediate language LLJava. The main matter of this paper is dedicated to exploration of the design and its rationale.

1.3 Design Guidelines

The design of LLJava obeys the following requirements:

1. MUST represent all capabilities of the JVM eventually, except for arguably obscure or deprecated features; MAY impose temporary restrictions on prototypes. The limitations of the current prototype are stated at the appropriate places below.
2. MUST reuse or extrapolate Java high-level syntax where adequate; SHOULD adapt JVMS syntax where it is specified; SHOULD NOT invent new syntax without necessity. Syntactic choices are discussed and justified below.
3. MUST abstract from *holistic* information – indexing and tabulating of entities that inhibits incremental program construction. Absence of holistic constraints, *compositionality*, is an important practical requirement, in particular when operating the API to construct code automatically under program control.
4. SHOULD abstract from *ready* information – local sources; retrieved directly or by simple and specified deduction procedures. This is most evident in our *overloading* use of identical operators for analogous instructions, in contrast to JVMS vernacular where they are disambiguated lexically; see §2.3.

2. THE LLJAVA LANGUAGE

This section specifies the syntax of the LLJava language. We use EBNF operators |, ( ), [] and { } to abbreviate grammar rules. The API of the corresponding object model is implied, but too technical to be discussed here. The semantics of translation to JVM bytecode is specified informally.

```
1  .class public HelloWorld
2  .super java/lang/Object
3  .method public static main :[a] ([Ljava/lang/String;)V
4    .limit stack 2[ab] 10[c]
5    .limit[bc] locals[bc] 1[b] 10[c]
6    getstatic java/lang/System /[ab] out Ljava/io/PrintStream;
7    ldc "Hello␣World!"
8    invokevirtual java/io/PrintStream /[ab] println(Ljava/lang/String;)V
9    return
10 .end method
11 .end[b] class[b]
```

a — http://jasmin.sourceforge.net/about.html
b — https://github.com/ymasory/programming-for-the-jvm/blob/master/examples/HelloWorld.j
c — https://github.com/Storyyeller/Krakatau/blob/master/examples/hello.j

Figure 2: Synoptic Hello World example in Jasmin (a), Oolong (b), Krakatau (c)

```
1  public synthetic class HelloWorld                // NB no legal Java class (no constructor)
2    /*extends java.lang.Object*/ {
3    public static void main(java.lang.String[] /*args*/) {
4
5
6      get    static java.io.PrintStream java.lang.System.out
7      load   "Hello,␣World!"
8      invoke void /*java.io.PrintStream.*/println(java.lang.String)
9      return
10   }
11 }
```

Figure 3: Hello World example in LLJava

2.1 Types

The (pre-generic) type language of Java is adequate for the type system of the JVM; no conceptual details are added, except for the deprecated return-address type, which we do not support. Therefore we use the familiar Java syntax for all type matters. For the purpose of presentation in this paper, we abstract from the technical details of mangling and from the treatment of generic types, to be split into raw types (descriptors) and parameterized types (signatures).

Type → *RefType* | *PrimType*
RefType → *ClassType* | *Type* **[ ]**
ClassType → *QualId*
PrimType → **boolean** | **char** | **byte** | **short** | **int** | **long** | **float** | **double**
Result → **void** | *Type*

All type information in LLJava is handled on a strict per-class basis. Other classes are referenced by name only, no implicit information concerning their subclass relationship and their members is assumed or used. Thus the LLJava tools operate in a completely modular fashion, at the price that cross-class type assumptions must be explicit in the LLJava program; see for instance §2.3.4 and §2.3.7. Top-level class types are referenced by fully qualified identifiers in Java source (dotted) notation.

2.2 Class Structure

For the aspects of bytecode that concern the JVM class structure, that is everything outside of method body code, we use almost pure Java source code syntax.

2.2.1 Access Flags and Attributes

Flag → **public** | **private** | ...

For each JVMS constant `ACC_XYZ` there is a flag *xyz*. Most of these are already valid Java keywords. In the single case of a naming inconsistency, we prefer Java `strictfp` over `strict` for JVMS `ACC_STRICT`. We have extrapolated the set to contain also `bridge`, `varargs`, `synthetic` (see Figure 3, line 1) and `annotation`. We abstract from the technical details of numeric encoding as bits of a 16-bit word. Note that several syntactic flags may share a common numeric encoding, and are to be disambiguated by context.

We do not yet support explicit denotation of arbitrary bytecode attributes (JVMS §4.7). Implicit attributes that are compiled from other information are specified below.

2.2.2 Classes

ClassDef → {*Flag*} **class** *QualId*
[**extends** *ClassType*]
[**implements** *ClassType* {**,** *ClassType*}]
{ {*Member*} **}**

Flag constraints specified for classes in the JVMS apply. The flag `super`, whose absence is deprecated, is always implied. The superclass defaults to `java.lang.Object` if omitted (see Figure 3, line 2). Note that the Java keywords `enum` and `interface` are used as flags that precede, but do not replace the keyword `class`.

2.2.3 Members

Member → *Field*
Field → {*Flag*} *Type* *Id* [**=** *Literal*] **;**

Member → *Method*
Method → {*Flag*} *Result MethodName*
([*Param* {**,** *Param*}] **)**
[**throws** *ClassType* {**,** *ClassType*}]
(*Block* | **;**)
MethodName → *Id* | **<init>** | **<clinit>**
Param → *Type* [*Id*]

Flag constraints specified for methods in the JVMS apply. A literal initializer specifies a `ConstantValue` attribute of the field in bytecode. Parameters can be named for reference in the method body; anonymous parameters can not be accessed (see Figure 3, line 4). Types listed after `throws` specify the `Exceptions` attribute of the method in bytecode.

We do not yet support inner classes or annotations.

2.3 Statements

This section follows the subsection structure of JVMS, §2.11. Consequently, the syntax of method body code deviates from the Java source code syntax in many ways, thus making the essential details explicit. However, it also abbreviates the verbose syntactic style of JVMS and `javap` significantly, thus making technical details implicit.

Unlike JVM bytecode, where the code for each method is a flat sequence of instructions, we allow instructions to be organized into a hierarchical block structure. The reasons are manifold: At the syntactic level, block structure has better documentation value for complex code, and allows for symbolic rather than numeric referencing of code positions and intervals. At the object-model level, block structure simplifies compositional code synthesis, and serves as the internal model on which to normalize the control flow graph into basic blocks; a necessary step for the static analysis that recovers local type information which is abstracted from in the instruction syntax presented below.

Block → **{** {*Statement*} **}**
Statement → *Block* | *Instruction* | *Id* **:**
CodeRef → **goto** *Id* | *Block*
CodeInterval → *Id* [**–** *Id*] | *Block*

Labels apply to the following statement, and serve to reference code positions (such as branch targets) and intervals (such as exception handler scopes). Intervals are specified by a single label (ranges over the following instruction or block) or a pair of labels (inclusive start and exclusive end).

2.3.1 Load and Store Instructions

Instruction → **load** *Literal*
| (**load** | **store**) *VariableRef*
VariableRef → **this** | *Id*
Instruction → *Type Id* **;**

The `load` instruction subsumes the loading of literals from the constant pool, as well as immediate and operand values, such as in `aconst_null` or `bipush`, respectively. The type of the loaded value is inferred trivially and hence implicit. Local variables are declared with name and type; they are in scope and must be unique from the point of declaration to the end of the enclosing block. Local variables are referenced by name and numbered implicitly. We foresee, but do not yet support a notation for explicit allocation of numbered local variable slots.

2.3.2 Arithmetic Instructions

Instruction → **add** | **sub** | **neg** | **inc**
| **mul** | **div** | **rem**
| **shl** | **shr** | **ushr**
| **and** | **or** | **xor**
| **cmp** [**<** | **>**]

Arithmetic operations are named as in JVMS, omitting the initial letter that signifies operand type, which is instead inferred from the typing of the operand stack. The `NaN`-sensitive instructions `dcmp<`*op*`>` and `fcmp<`*op*`>` are written graphically.

2.3.3 Type Conversion Instructions

Instruction → **cast** *PrimType*

All conversion operations such as `l2f` are subsumed under a single `cast` instruction. The source type is inferred from the typing of the operand stack; the target type is explicit and in Java syntax. Note that `cast` is also used as an abbreviation of the related JVM `checkcast` instruction; see next subsection.

2.3.4 Object Creation and Manipulation

Instruction → **new** *RefType* {**[** **_** **]**}
| (**get** | **put**) [**static**] *FieldRef*
| (**load** | **store**) **[** **]** | **length**
Instruction → (**instanceof** | **cast**) *RefType*
FieldRef → *Type* [*QualId* **.**] *Id*

Regular and array objects are created uniformly using the `new` instruction. A type suffix `[_]` denotes a dimension of multiarray allocation. A single operation `get` denotes JVMS `getfield` as the unmarked default, or can be qualified with `static`; `set` is treated symmetrically. Array access with `load[]` and `store[]` omits the element type, which is inferred from the typing of the operand stack.

Note that the type of fields is explicit, because it cannot be inferred in a modular way during separate compilation of classes. By contrast, the owner type of the field declaration can be omitted for non-static access; the unmarked default is the class type of the reference on the operand stack.

2.3.5 Control Flow

Instruction → **try** *CodeInterval* {*Handler*}
Handler → **catch** **(** [*ClassType*] **)** *CodeRef*
Instruction → **goto** *Id* | **return** | **throw**

The content of the exception table of the `Code` attribute of a method is specified in a decentral way using the `try` operation. It refers to a code interval, which can either contain inline code, as in Java source, or cross-reference a range, as in JVMS bytecode. Analogously, handlers refer either to inline code or to a branch target. The catch-all handler is denoted as `catch()`. Note that neither `try` nor `catch` corresponds to an actual JVM instruction.

Unconditional jumps are denoted with the explicit `goto` instruction. The width of the address operand is inferred. The `return` instruction omits the result type, which is inferred from the method declaration. The same holds for `throw` trivially.

We do not support the deprecated subroutine instructions `jsr` and `ret`.

2.3.6 Conditionals

Instruction → **switch** **{** {*Case*} **}**
Case → (**case** *Integer* | **default**) **:** *CodeRef*

Both styles of JVMS `switch` instructions, table and lookup, are denoted as `switch`. Cases either contain inline code or a virtual `goto` cross-reference.

Instruction → **if** [**!**] **(** *Condition* **)** *CodeRef*
Condition → _ *RelOp* (_ | **0**)
| _ *EqOp* **null**
RelOp → *EqOp* | **<** | **<=** | **>=** | **>**
EqOp → **==** | **!=**

Conditional branches are denoted with `if` as in Java source code. The condition is written graphically, with the operand or operands to be consumed from the stack specified by _ placeholders.

Conditional branching is different from `try–catch` and `switch` in the sense that an inline code block as the target *CodeRef* is not directly useful, as it would imply that control is transferred to the next instruction (the beginning of the block) unconditionally. Therefore we interpret the forms

if (*c*) { *S* } and **if** !(*c*) { *S* }

as shorthands for the patterns

```
    if (c) goto A                 if (c) goto B
    goto B
A:  S                      and
B:                            A:  S
                              B:
```

respectively, where `A`, `B` are fresh labels. Note that on the left hand side, the first **goto** is virtual, but the second one is an actual instruction.

2.3.7 Method Access

Instruction → **invoke** [*InvokeFlag*] *MethodRef*
InvokeFlag → **static** | **interface** | **super** | **private**
| *BootstrapRef*
MethodRef → *Result* [*QualId* **.**] *MethodName*
([*Type* {**,** *Type*}] **)**

The five invocation styles of the JVM are selected by qualifiers: `invokevirtual` is the unmarked default as in Java (see Figure 3, line 5); `static` and `interface` select the eponymous styles; both `private` and `super` select `invokespecial`; a reference to a bootstrap method selects `invokedynamic`, which is not yet supported.

As for fields, method references have explicit type signatures. The owner class, if omitted, defaults to the class type of the reference on the operand stack.

A method reference usually mentions the fully qualified method name, with containing class and package. An unqualified method name is implicitly completed with the operand type as the containing class (see Figure 3, line 5).

2.3.8 Stack Manipulation

Instruction → **nop** | **pop** | **dup**

The number of operand stack slots to be manipulated is inferred from the typing of the operand stack. We do not (yet?) support "advanced" stack manipulation, such as `pop2` for pairs of single-slot values, or `dup_x`.

2.3.9 Monitor Access

Instruction → **enter** | **exit**

3. ADVANCED EXAMPLES

3.1 Coding for the Human Reader/Writer

Consider a small but nontrivial Java program as depicted in Figure 4, top. A LLJava source program that conveys equivalent information at the JVM level, not only to the machine but also to the human reader, is depicted in Figure 4, bottom. It has been "compiled" from the former by invoking `javac` followed by `javap`, and manual editing of the code dump. We expect our forthcoming LLJava disassembler tool to produce comparable output.

Note how the inline code variants of *CodeRef* and *CodeInterval*, in lines 36 and 14, respectively, are used to reflect the structured style of Java programming. Alternatively, in a more bytecode-like format, one could remove the `try–catch()` pattern, and insert the statement in line 42f. that reflects the tabular style of `Code` attribute exception tables.

3.2 Language Explication by Translation

An effective way of communicating language semantics in practical terms is by means of code generation schemes for individual language constructs. This can only be made adequately precise, if the target language has reasonable compositionality. For instance, consider the Java conditional operator (left), whose semantics can be explained precisely and concisely by a compositional translation scheme to LLJava (right):

```
x ? y : z   ⇝   {
                  x
                  if (_ == 0) goto A
                  y
                  goto B
                A: z
                B:
                }
```

4. CONCLUSION

4.1 Summary

We have presented the design of LLJava, a language comprising a textual frontend and object model that expresses the capabilities of the JVM at an intermediate level of structure and abstraction that caters for the needs of code synthesizers and human readers and writers of JVM bytecode alike.

4.2 Implementation Status

LLJava is implemented in Java and hence suitable for meta- or macro-programming in any language hosted interoperably on the JVM platform. The current prototype covers the distinct processing steps unequally:

- The forward frontend (parser) for translating LLJava source code to object model is operational.
- The forward backend (encoder) for translating LLJava object model to bytecode (class file) is mostly operational. Subtle technicalities of the `StackMapTable` attribute are a subject of current research; hence only class file version 49.0 is fully supported. Support for

version 52.0, including verification by type checking (JVMS §4.10.1), is work in progress.

- The backward frontend (unparser) for translating LLJava object model to source code is mostly operational. Current research focuses on the theoretically minor but practically important detail of finding a good graph representation for control flow (Spanning tree of inline code blocks versus labels and virtual `goto` cross-references).
- The backward backend (decoder) for translating class files to LLJava object model is only partly operational. Abstraction from technical details is the easy part. The reconstruction of local variables (name, type and scope) from access patterns and debugging information is an interesting topic of future research. We are considering a variant that is closer to the JVMS notion of local variables as numbered, untyped stack slots.

4.3 Related Work

Basic related tools have been cited in the introduction. There is also more up-to-date and technologically advanced work that deserves mention:

The practical use of abstractions from the JVM bytecode format has been demonstrated in great detail, in the context of static analysis, by the Soot framework [9] which builds on Jasmin. It offers a plethora of different, specialized intermediate representations (with whimsical names such as *baf*, *grimp* or *jimple*) which have distinct abstraction strategies, sharing a general paradigm shift to register code. However, our proposed strategy from section 1.1, which emphasizes both code synthesis and understanding for the JVM paradigm, is not among them.

5. REFERENCES

[1] *ASM*. OW2 Consortium. 2015. URL: http://asm.ow2.org/.

[2] *Byte Code Engineering Library (BCEL)*. Apache Commons. 2014. URL: https://commons.apache.org/proper/commons-bcel/index.html.

[3] T. Downing and J. Meyer. *JAVA Virtual Machine*. O'Reilly, 1996. ISBN: 978-1565921948.

[4] J. Engel. *Code for book Programming for the Java Virtual Machine*. GitHub, 2010. URL: https://github.com/ymasory/programming-for-the-jvm/.

[5] J. Engel. *Programming for the Java Virtual Machine*. Addison-Wesley, 1999. ISBN: 978-0201309720.

[6] R. Grosse. *Krakatau Bytecode Tools*. GitHub, 2015. URL: https://github.com/Storyyeller/Krakatau/.

[7] T. Lindholm et al. *The Java Virtual Machine Specification*. Java SE 7. JSR-000924. Oracle, 2013.

[8] J. Meyer and D. Reynaud. *Jasmin Home Page*. 2005. URL: http://jasmin.sourceforge.net/.

[9] R. Vallée-Rai et al. "Soot – a Java Bytecode Optimization Framework". In: *Proc. CASCON*. IBM, 1999.


```
class StreamCat {
  boolean cat(java.io.InputStream in,
              java.io.OutputStream out) {
    byte[] buf = new byte[1024];
    try {
      int c;
      while ((c = in.read(buf)) != -1)
        out.write(buf, 0, c);
      in.close();
      out.close();
      return true;
    }
    catch (java.io.IOException e) {
      return false;
    }
  }
}
```

```
class StreamCat {
  synthetic void <init>() { // implicit
    load    this
    invoke  super void <init>()
    return
  }

  boolean cat(java.io.InputStream in,
              java.io.OutputStream out) {
    byte[] buf;
    load    1024
    new     byte[]
    store   buf
T: try {
     int c;
A:   load   in
     load   buf
     invoke int read(byte[])
     dup
     store  c
     load   -1
     if     (_ == _) goto B
     load   out
     load   buf
     load   0
     load   c
     invoke void write(byte[], int, int)
     goto   A
B:   load   in
     invoke void close()
     load   out
     invoke void close()
     load   1            // true
     return
   }
C: catch (java.io.IOException) {
     java.io.IOException e;
     store  e            // funny, unused
     load   0            // false
     return
   }
// try T /*-C*/
// catch (java.io.IOException) goto C
  }
}
```

Figure 4: Stream catenation example, Java (top) and LLJava (bottom)