Functional Java Bytecode

Christopher League
Valery Trifonov
Zhong Shao

Yale University
USA

SCIEIRE workshop
24 July 2001
Java supports verifiable mobile code

```
Java

javac

JVML

Verifier

IR

JIT compiler

EXE

Runtime system
```
TAL, PCC support verifiable object code

[Necula and Lee 1996]
[Morrisett et al. 1999]
[Appel 2001]
FLINT: a certifying compiler framework

[Shao et al. 1997, 1998]
[League et al. 1999, 2001]
JVML and $F_\omega$ are worlds apart

- classes, objects, methods
- access control
- inheritance, subtyping
- operand stack
- untyped local variables
- subroutines

- records, functions
- existential types
- row polymorphism
- explicit arguments
- typed, immutable bindings
- higher-order functions
λJVM was designed as a midpoint

- classes, objects, methods
- access control
- inheritance, subtyping

- records, functions
- existential types
- row polymorphism

- operand stack
- untyped local variables
- subroutines

- explicit arguments
- typed, immutable bindings
- higher-order functions
\( \lambda \text{JVML} \) is an alternative to \( \text{JVML} \... \)

...for Java systems with optimizing compilers.
- explicit data flow (like SSA form)
- suitable for translation into compiler IRS

Compared to \( \text{JVML} \):
- cleaner specification
- simpler verification
Syntax of simply-typed \( \lambda \)-calculus

Types \[ \tau ::= (\tau) \rightarrow \tau \mid \nu \mid 1 \]

Values \[ \nu ::= x \mid \lambda(x:\tau)e \mid i \]

Terms \[ e ::= \text{letrec } x = \nu . e \mid \text{let } x = p ; e \]
\[ \quad \mid \text{return } \mid \text{return } \nu \mid \nu(\nu) \]

Primops \[ p ::= \nu_1 + \nu_2 \mid \nu_1 \times \nu_2 \mid \ldots \]

A-normal form: nested expressions must be flattened

\[ (3 + 4) \times 5 \implies \text{let } a = 3 + 4; \]
\[ \quad \text{let } b = a \times 5; \]
\[ \quad \text{return } b \]

[Flanagan et al. 1993]
...extended with Java primitives & types

Types \( \tau ::= (\tau) \to \tau \mid V \mid I \mid L \mid F \mid D \mid \ldots \mid c \mid \tau[] \mid c^0 \mid \{\overline{c}\} \)

Values \( v ::= x \mid \lambda(x:\tau)e \mid i \mid r \mid s \mid \text{null}[\tau] \)

Terms \( e ::= \text{letrec } x = v . e \mid \text{let } x = p ; e \mid p ; e \mid \text{return } \mid \text{return } v \mid v(\overline{v}) \mid \text{throw } v \mid \text{if } \text{br}[\tau] v v \text{ then } e \text{ else } e \)

Primops \( p ::= \text{bo}[\tau] v_1 v_2 \mid \text{neg}[\tau] v \mid \text{convert}[\tau_0, \tau_1] v \mid \text{new } c \mid \text{chkcast } c v \mid \text{instanceof } c v \mid \text{getfield } f v_0 \mid \text{putfield } f v_0 v \mid \ldots \mid \text{invokevirtual } m v_0 (\overline{v}) \mid \ldots \mid \text{newarray}[\tau] v_n \mid \ldots \)

Branches \( \text{br} \quad \text{Binops } \text{bo} \quad \text{Field/method descriptor } f / m \)
Observations about $\lambda$JVM

1. it is functional
   and therefore equivalent to SSA
   [Kelsey 1995]

2. all function calls are in tail position

3. functions are first class and
   lexically scoped
   • flexible control flow,
   • yet all call sites are known!
Example: a Java ‘for’ loop becomes...

```java
public static void m(int i) {
    Pt p = new IntPt(i);
    for (int j = 1; j < i; j *= 2) {
        p = new ColorPt(j);
    }
    p.draw();
    return;
}
```
branches and jumps JVML

```java
public static void m(int i) {
    Pt p = new IntPt(i);
    for (int j = 1; j < i; j *= 2) {
        p = new ColorPt(j);
    }
    p.draw();
    return;
}
```

```java
public static m(I)V
    new IntPt
    dup
    iload_0
    invokespecial IntPt.<init>(I)V
    astore_1 ; p = new IntPt(i)
    iconst_1
    istore_2 ; j = 1
    goto C
}
```
...mutually recursive functions in λ JVM

```java
public static void m(int i) {
    Pt p = new IntPt(i);
    for (int j = 1; j < i; j *= 2) {
        p = new ColorPt(j);
    }
    p.draw();
    return;
}
```

```lambda
public static m(i: I) V = λ(i: I)
letrec
  C = λ(p:{IntPt,ColorPt}, j: I)
      if lt[I] j i then B(p,j)
      else ivirtual Pt.draw() V p();
      return.
  B = λ(p:{IntPt,ColorPt}, j: I)
      let q = new ColorPt;
      ispecial ColorPt.<init>(I) V q (j);
      let k = mul[I] j 2;
      C(q,k).
  let r = new IntPt;
  ispecial IntPt.<init>(I) V r (i);
  C(r,1)
```

```java
public static m(I)V
    new IntPt
dup
    iload_0
    invokespecial IntPt.<init>(I)V
    astore_1 ; p = new IntPt(i)
    iconst_1
    istore_2 ; j = 1
    goto C
B: new ColorPt
dup
    iload_2
    invokespecial ColorPt.<init>(I)V
    astore_1 ; p = new ColorPt(j)
    iload_2
    iconst_2
    imul
    istore_2 ; j *= 2
C: iload_2
    iload_0
    if_icmplt B ; goto B if j < i
    aload_1 ; p.draw()
    invokevirtual Pt.draw() V
    return
```
Subroutines pose major challenges

1. they are polymorphic over the types of the locations they do not touch

2. calls and returns need not obey stack discipline

3. subroutine might update a local variable

Solution: continuation-passing style
Example: method with one subroutine

public static f(I)V
    jsr S
    ldc "Hello"
    astore_1
L: jsr S
    aload_1
    invoke println
    goto L

S: astore_2 ; return addr
    iload_0
    ifeq R
    iinc 0 -1
    ret 2

R: return

At each call site, we must determine which local variables:
• should be passed to the subroutine,
• could be modified and thus should be passed back to the caller
• are ignored but must be preserved across the call.
Return address ↔ continuation

public static f(I)V
    jsr S
    ldc "Hello"
    astore_1
L: jsr S
   aload_1
    invoke println
    goto L

S: astore_2 ; return addr
    iload_0
    ifeq R
    iinc 0 -1
    ret 2
R: return

public static f(I)V = λ(n:I)
    letrec
        S = λ(i:I, r:(I)→V)
            if eq[I] i 0 then return
            else let j = add[I] i -1;
                r(j).
        L = λ(i:I, s:String)
            S(i, λ(j:I)
                invoke println s;
                L(j,s)).
        S(n, λ(j:I) L(j, "Hello"))

The higher-order functions can be compiled away efficiently since all call sites are known.
Verifying λjvm classes

Class verification reduces to simple type checking

(< 260 lines of SML code)

• all the difficult analyses are done during translation,
• results are preserved in type annotations.

Object initialization

In JVM, requires (conservative) alias analysis.

• in λjvm, aliases within a basic block are transparent;
• between basic blocks, aliased arguments are unified.
Subtyping and set types

The subtype relation \((\tau \leq \tau')\) mirrors the class hierarchy, and includes numeric promotions \((\mathbb{I} \leq \mathbb{F})\). On set types,

\[
\begin{align*}
    c & \in \{\overline{c}\} \\
    c & \leq \{\overline{c}\} \\
    \{\overline{c_1}\} & \subseteq \{\overline{c_2}\} \\
    \{\overline{c_1}\} & \leq \{\overline{c_2}\} \\
    c & \leq c' \quad \forall c \in \{\overline{c}\} \\
    \{\overline{c}\} & \leq c'
\end{align*}
\]
Related work

• Gagnon et al. [2000]
  “Efficient inference of static types for Java bytecode”
  – mutable variables, split on demand into separate uses separate uses
  – no set types; explicit type casts instead

• Katsumata and Ohori [2001]
  “Proof-directed decompilation of low-level code”
  – extremely elegant
  – does not extend to subroutines, etc.

• Amme et al. [2001] SafeTSA
  – similar in spirit, but starts with Java
  – innovative encoding techniques – largely orthogonal
Conclusion

\( \lambda jvm \) is a functional representation of Java bytecode which

- makes data flow explicit,
- makes verification simple, and
- is a good match for optimizing compiler \( IRS \).

\( \lambda jvm \) is particularly successful as a midpoint between Java bytecode and \( IRS \) based on typed \( \lambda \)-calculi.

http://flint.cs.yale.edu/