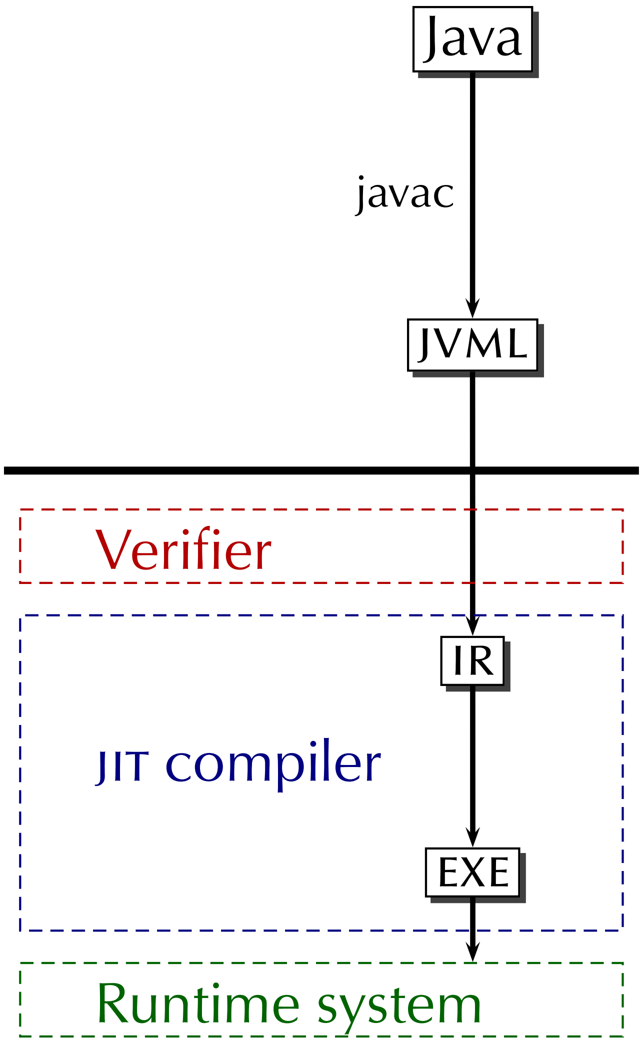
Functional Java Bytecode

Christopher League
Valery Trifonov
Zhong Shao

Yale University
USA

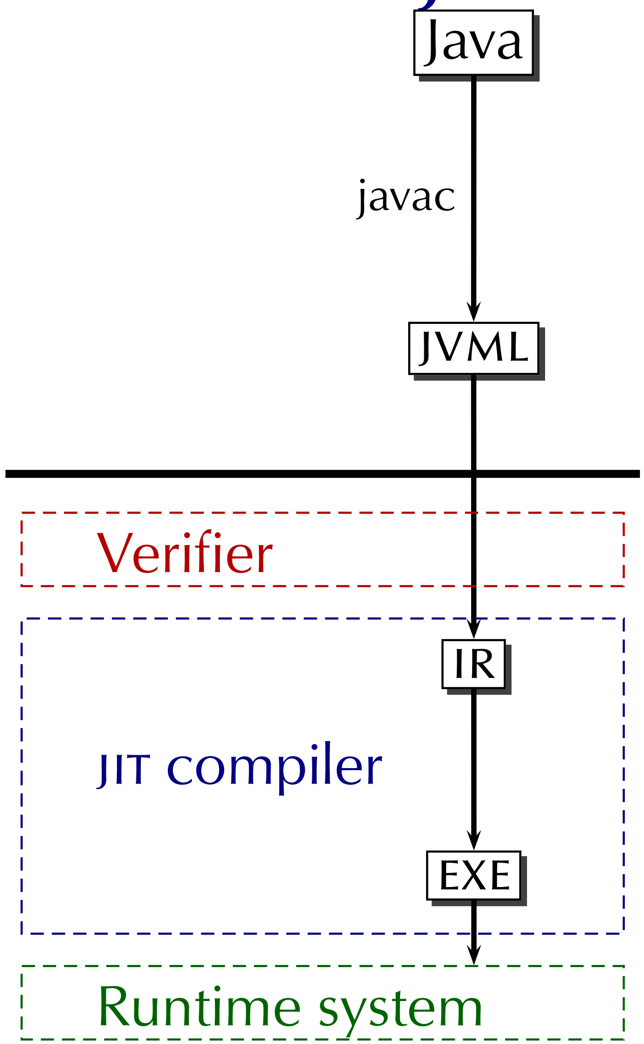
SCI · IRE workshop
24 July 2001

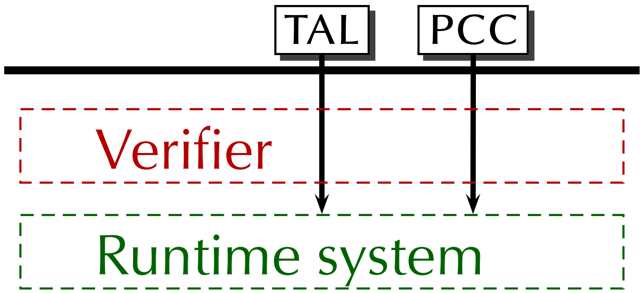
Java supports verifiable mobile code



TAL, PCC support verifiable object code

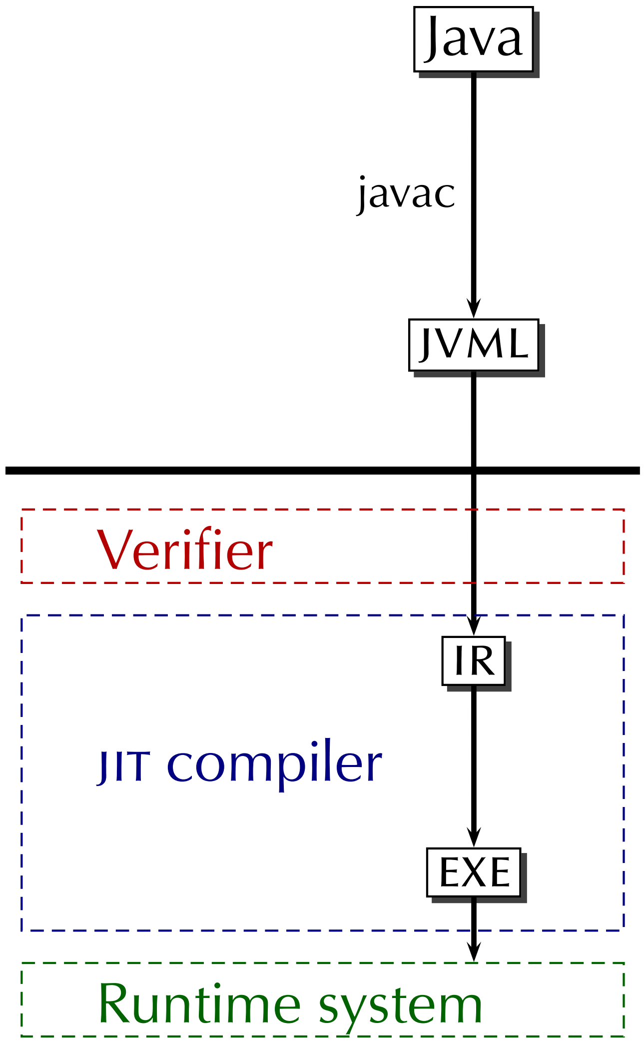
[Necula and Lee 1996]
[Morrisett *et al.* 1999]
[Appel 2001]

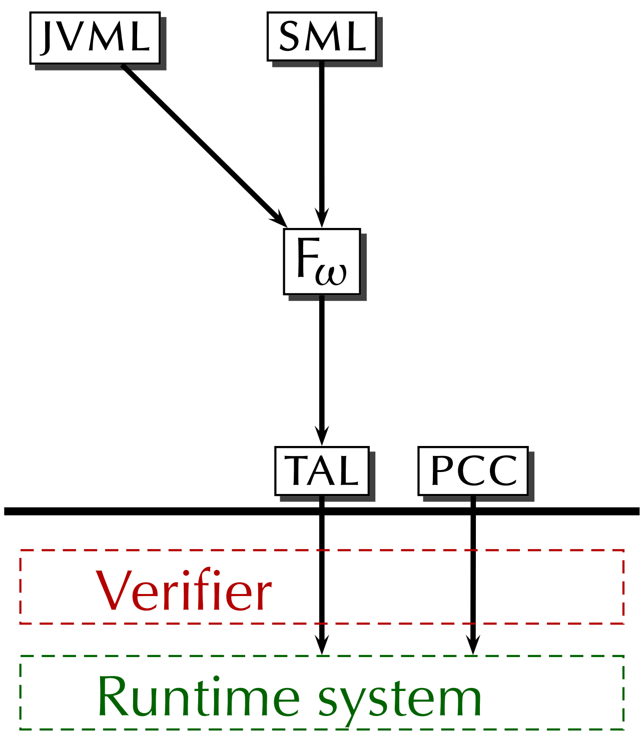




FLINT: a certifying compiler framework

[Shao *et al.* 1997, 1998]
[League *et al.* 1999, 2001]





JVML and F_ω are worlds apart

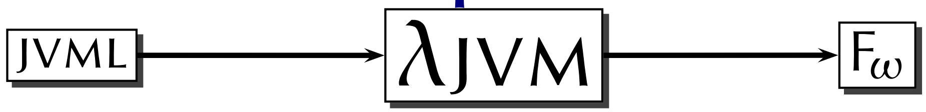
- classes, objects, methods
- access control
- inheritance, subtyping

- 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



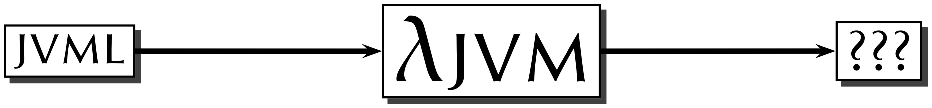
- classes, objects, methods
- access control
- inheritance, subtyping

- records, functions
- existential types
- row polymorphism

- operand stack
- untyped local variables
- subroutines

- explicit arguments
- typed, immutable bindings
- higher-order functions

λJVM is an alternative to JVML...



...for Java systems with optimizing compilers.

- explicit data flow (like SSA form)
- suitable for translation into compiler IRS

Compared to JVML:

- cleaner specification
- simpler verification

Syntax of simply-typed λ-calculus

Types $\tau ::= (\overline{\tau}) \rightarrow \tau \mid \mathtt{V} \mid \mathtt{I}$

Values $v ::= x \mid \lambda(\overline{x : \tau})e \mid i$

Terms $e ::= \mathtt{letrec}\ \overline{x = v}\mathtt{.}\ e \mid \mathtt{let}\ x = p\mathtt{;}\ e$
$\quad \mid \mathtt{return} \mid \mathtt{return}\ v \mid v(\overline{v})$

Primops $p ::= v_1 + v_2 \mid v_1 \times v_2 \mid \ldots$

A-normal form: nested expressions must be flattened

$$(3 + 4) \times 5 \implies \begin{array}{l} \mathtt{let}\ a = 3 + 4\mathtt{;} \\ \mathtt{let}\ b = a \times 5\mathtt{;} \\ \mathtt{return}\ b \end{array}$$

[Flanagan *et al.* 1993]

...extended with Java primitives & types

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

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

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

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

Branches br Binops bo Field/method descriptor f/m

Observations about λJVM

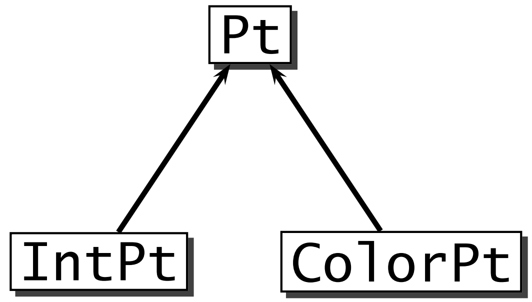
1. it is functional
   and therefore equivalent to SSA
   [Kelsey 1995]

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

```
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;
}
```

```
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
```

At B: $0 \Rightarrow \mathrm{I}, \quad 1 \Rightarrow \{\texttt{IntPt}, \texttt{ColorPt}\}, \quad 2 \Rightarrow \mathrm{I}$

At C: $0 \Rightarrow \mathrm{I}, \quad 1 \Rightarrow \{\texttt{IntPt}, \texttt{ColorPt}\}, \quad 2 \Rightarrow \mathrm{I}$

...mutually recursive functions in λJVM

```
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;
}
```

```
public static m(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)
```

```
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 JVML, 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 ($\mathtt{I} \leq \mathtt{F}$).
On set types,

$$\frac{c \in \{\overline{c}\}}{c \leq \{\overline{c}\}}$$

$$\frac{\{\overline{c_1}\} \subseteq \{\overline{c_2}\}}{\{\overline{c_1}\} \leq \{\overline{c_2}\}}$$

$$\frac{c \leq c' \quad \forall c \in \{\overline{c}\}}{\{\overline{c}\} \leq c'}$$

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

λJVM is a functional representation of Java bytecode which

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

λJVM is particularly successful as a midpoint between Java bytecode and IRS based on typed λ-calculi.