Precision in Practice: A Type-Preserving Java Compiler

Christopher League
Long Island University

Zhong Shao
Valery Trifonov
Yale University

Mobile code, pervasive networks

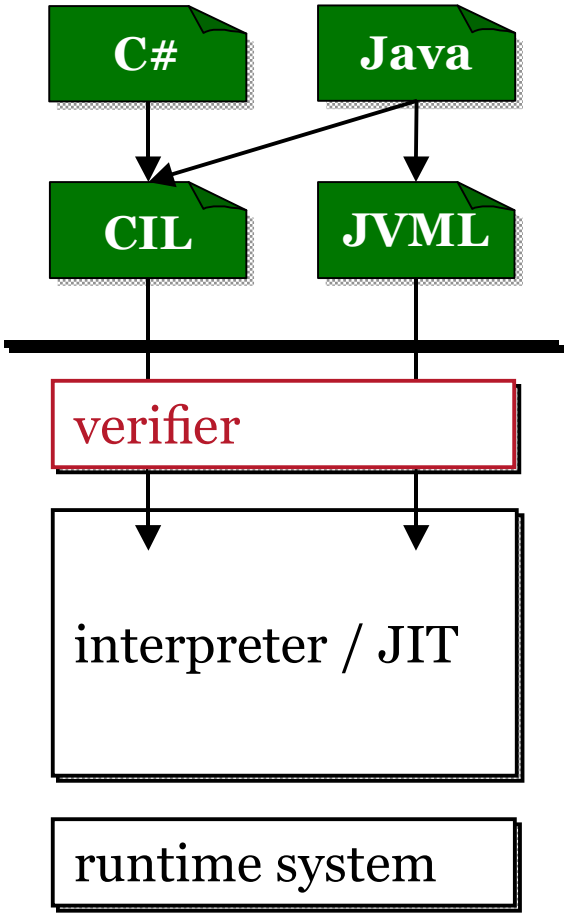
- Wireless handheld computers

- Remotely programmable devices

- Browser applets

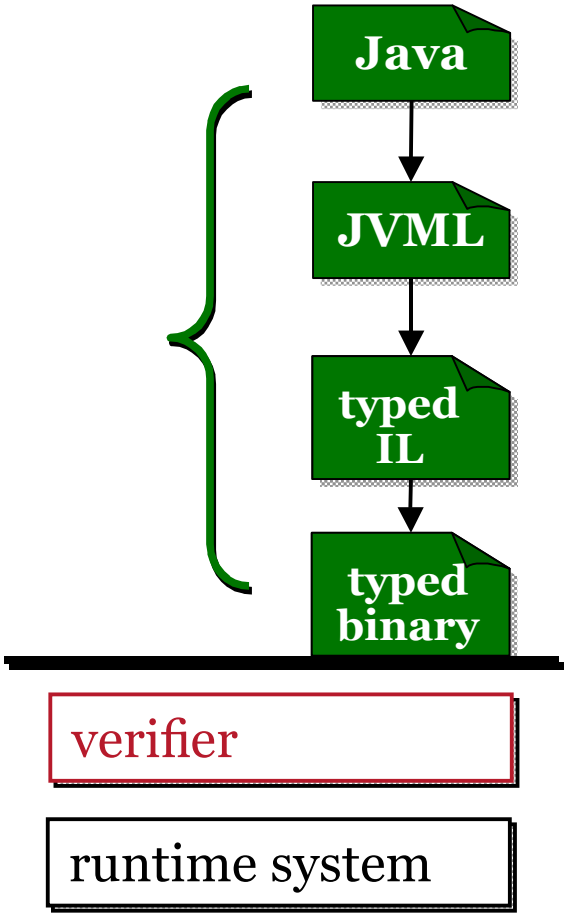
- Widely distributed computation

Verifiable distribution formats



- Quite high level
  - Atomic virtual method call
  - Limited optimizations
- Partial to OO languages
- Further compilation must be trusted

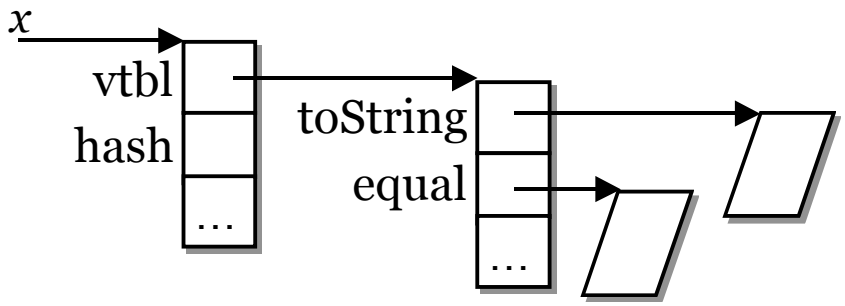
Type-preserving compiler



- Many compilers preserve some source-level type info
  - Not rigorous enough for verification!
- Lower-level code needs more sophisticated types

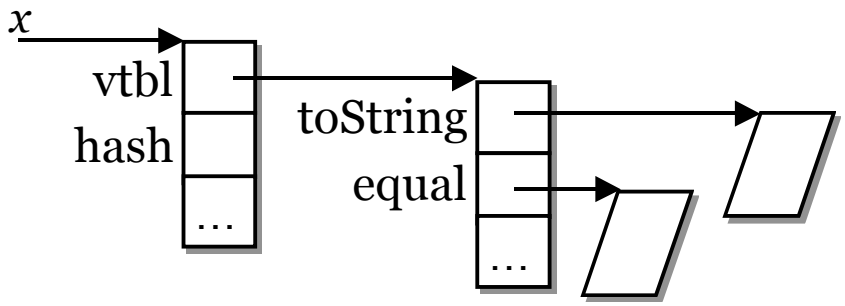
Efficient dynamic dispatch

```
public static void
example ( Object x, Object y )
{  x.toString( );

   // compiles to:
   (null check)
   let r1 = x.vtbl ;          // method suite
   let r2 = r1.toString ;     // method pointer
   r2( x )                    // "self application"

 }
```



Must ensure the call is safe

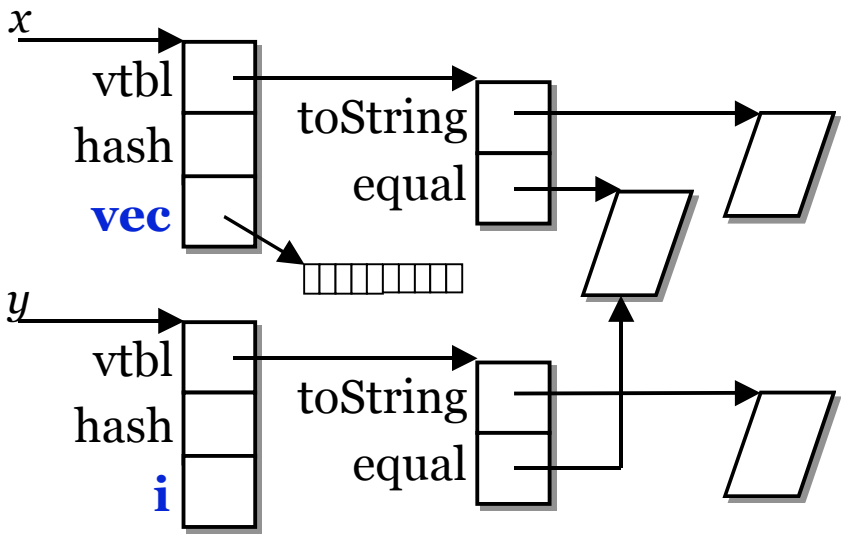
```
public static void
example ( Object x, Object y )
{  x.toString( );

   // compiles to:
   (null check)
   let r1 = x.vtbl ;
   let r2 = r1.toString ;
   r2( x ) ;                    // this is sound
   r2( y )                      // this is not!
 }
```



Any unsoundness can be exploited

Cast arbitrary integer to pointer

```
class Ref extends Object
{  public byte[ ]  vec ;
   public String toString( )
   {   vec[13] = 42 ;
       return "Ha!" ;
    }
}

class Int extends Object
{  public int  i ;
}

example( new Ref(...),
         new Int(...) );
```




```
public static void
example ( Object x, Object y )
{   let r1 = x.vtbl ;
    let r2 = r1.toString ;
    r2( y )  // wrong!
 }
```

Previous work

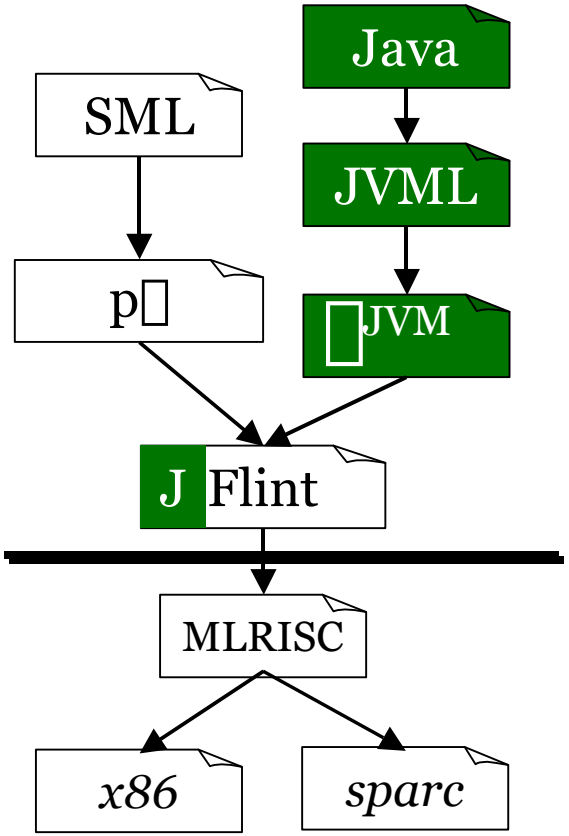
- Efficient type-theoretic encodings of Java features

  [TOPLAS '02]

- Implementation techniques for typed intermediate languages

  [ICFP '98]

Current contribution



- A prototype compiler based on our encodings
  - Front ends for both JVML and SML
  - Share optimizers and code generator
  - Run together in same runtime system

Do we compile Java or JVML?

- Java

```
x.println(y);
```

javac

- JVML byte code

```
3  aload_0    # this
4  getfield PrintStream C::x
7  dload 2
9  invokevirtual
       void PrintStream::println(double)
```

- Many details are not explicit in Java source
- Java byte code has untyped local vars & implicit data flow

Two sets of concerns

1. data & control flow, type inference
2. expanding Java primitives

- JVML byte code → JFlint

```
3  aload_0    # this
4  getfield PrintStream C::x
7  dload 2
9  invokevirtual
       void PrintStream::println(double)
```

Another IL to bridge the gap

- High-level Java primitives, types

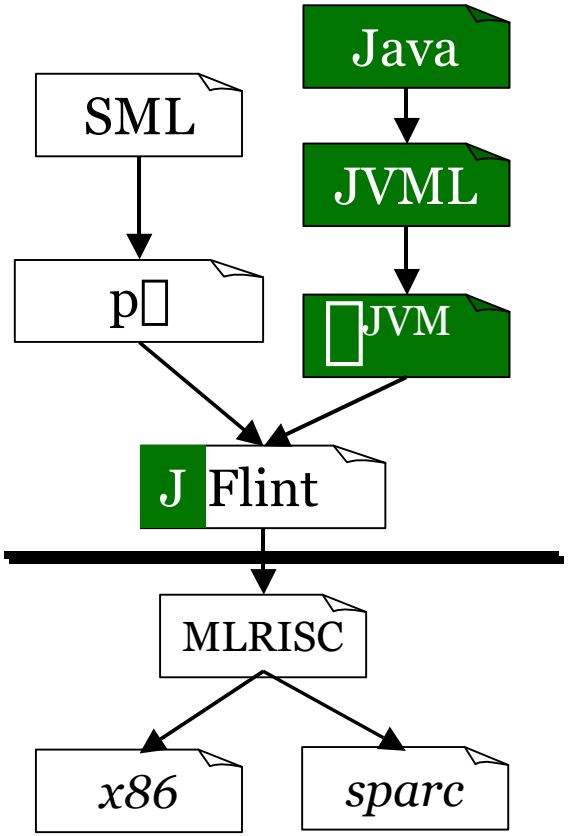
- JVML → JVM → JFlint

  - Functional control and data flow

JFlint: taming the type theory

- Before each class is compiled, we...
  - compute layout of fields & methods
  - construct all class and object types
  - keep this information in an environment
- Then, type-based translation of each JVM operation is straightforward

JFlint supports both languages well



- Its type system is impartial

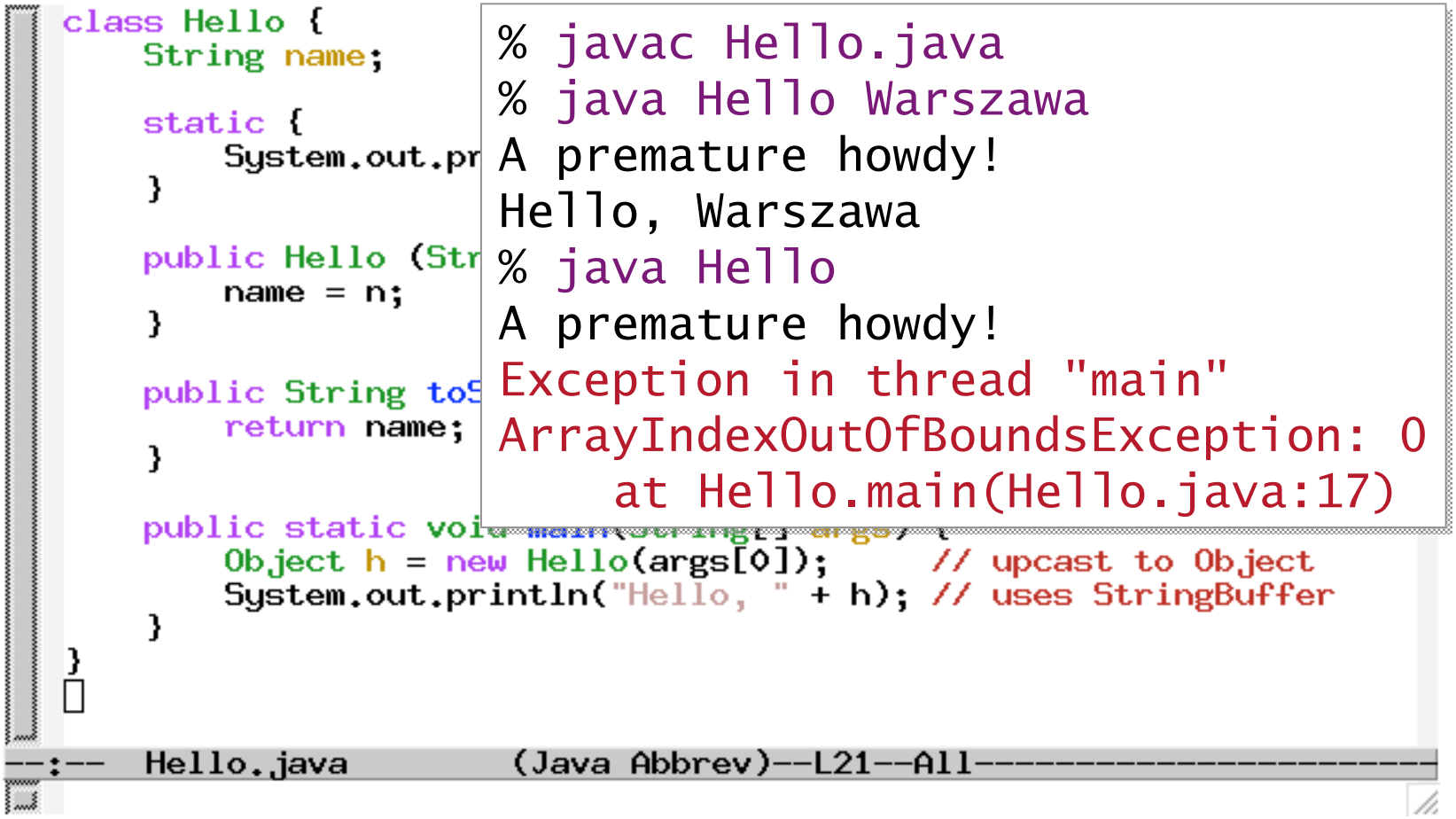
JFlint	Java	ML
	inheritance	parametric poly.
	object enc.	closures
μ	rec. classes	rec. datatypes
tags	dynamic cast	exceptions
rows	object enc.	—
records	vtable, objects	records, tuples
functions	methods	functions

Sample Java program

```
class Hello {
    String name;

    static {                          // class initializer
        System.out.println("A premature howdy!");
    }

    public Hello (String n) {    // constructor
        name = n;
    }

    public String toString() {  // overrides Object.toString
        return name;
    }

    public static void main(String[] args) {
        Object h = new Hello(args[0]);      // upcast to Object
        System.out.println("Hello, " + h); // uses StringBuffer
    }
}
```

```
--:--  Hello.java        (Java Abbrev)--L21--All-----------------------
```

Sample Java program



Demo: Java + ML

```
Standard ML of New Jersey v110.30 [JFLINT 1.2]
- Java.classPath;
val it = ["."] : string list
- Java.load "Hello";
[parsing Hello]
[parsing java/lang/Object]
[compiling java/lang/Object]
[compiling Hello]
[initializing java/lang/Object]
[initializing Hello]
A premature howdy!
val it = () : unit
-
```

Demo: Java + ML

```
- val main = Java.run "Hello";
val main = fn : string list -> unit
- main ["Warszawa", "Polska"];
Hello, Warszawa
val it = () : unit
- main [];
uncaught exception ArrayIndexOutOfBounds
  raised at: Hello.main([Ljava/lang/String;)V
-
```

Java class library

- We cannot use Java class library directly
  - Too much native code!
- We implement essential parts using
  - JVML assembly
  - ML structures & functions
    - » for final classes, such as java/lang/String
    - » a few special “native” methods

Performance @ compile time

- CaffeineMark 3.0 embedded (12 classes)
  - 1.5 sec with GNU Java compiler (gcj)
  - 2.4 sec with JFlint (60% increase)
  - + .5 sec for verification

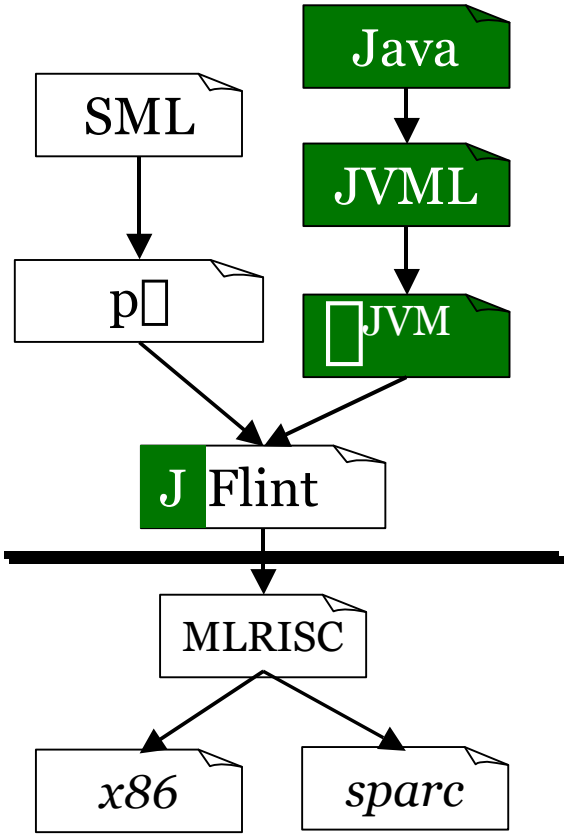
Performance @ run time

- CaffeineMark 3.0 embedded
  - 32% of the speed of gcj (with -O2)
- Why?
  - Runtime system optimized for ML:
    - » Boxed structure representations
    - » Heap-allocated activation records
  - Standard loop optimizations not implemented

Defense of performance results

- Comparison to unsafe systems is unfair
- Our typed intermediate language:
  - Safely exposes self-application code
  - Permits low-level optimizations
  - Is an effective target for Java and ML

Summary



- Type preservation is within reach for real compilers and mainstream languages