Functional Java Bytecode*

Christopher LEAGUE Valery TRIFONOV Zhong SHAO

Computer Science Department, Yale University
POB 208285, New Haven, CT 06520 USA
{*league, trifonov, shao*}*@cs.yale.edu*

ABSTRACT

We describe the design and implementation of λJVM, a functional representation of Java bytecode that makes data flow explicit, verification simple, and that is well-suited for translation into lower-level representations such as those used in optimizing compilers. It is a good alternative to stack-based Java bytecode for virtual machines or ahead-of-time compilers which optimize methods and produce native code. We use λJVM as one component in a sophisticated type-preserving compiler for Java class files. Though our implementation is incomplete, preliminary measurements of both compile and run times are promising.

Keywords: Java, type-preserving compilation, intermediate languages, typed lambda calculus.

1. MOTIVATION

The Java™ platform allows code from an untrusted producer to be transmitted to a consumer in a form that can be *verified* [10, 18]. Analogously, much recent work in type theory focuses on using type systems and logics to reason about the safety and security of low-level object code [22, 20, 4]. These systems have several potential advantages over the Java platform. Since they use lower-level code, they can better support different kinds of source languages. While many kinds of compilers target the Java virtual machine, they must often resort to Java primitives which are awkward or inefficient. JVML cannot express raw data layouts or optimizations.

For the especially security-conscious, a more severe problem is that the Java platform has an enormous trusted computing base (TCB). In addition to the verifier, we must trust that the just-in-time compiler, a considerably large and complex piece of software, does not introduce security-critical bugs. In systems based on proof-carrying code or typed machine language, the compiler can be removed from the TCB.

The FLINT project at Yale aims to build a type-preserving compiler infrastructure to generate low-level typed object code for multiple source languages [26]. The first generation of our FLINT intermediate language is widely distributed as a key component of the Standard ML of New Jersey compiler [27, 28]. We have recently extended FLINT to support a sophisticated type-preserving front end for Java [16, 17].

This paper describes λJVM, an intermediate language we designed as a midpoint between JVML and FLINT. We believe it is a good alternative to stack-based Java bytecode for virtual machines or ahead-of-time compilers which optimize methods and produce native code. It is already in a form that, like static single assignment [1], makes data flow explicit. In addition, λJVM is cleaner to specify and simpler to verify than JVML.

*This work was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9901011 and CCR-0081590. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

2. DESIGN

λJVM is a simply-typed lambda calculus [5] expressed in A-normal form [6] and extended with the types and primitive instructions of the Java virtual machine. The syntax is given in figure 1. We use terms e in place of the bytecode for method bodies; otherwise the class file format remains the same. A-normal form ensures that functions and primitives are applied to values only; the `let` syntax binds intermediate values to names. A nested algebraic expression such as $(3 + 4) \times 5$ is expressed in A-normal form as `let` $x = 3 + 4$; `let` $y = x \times 5$; `return` y.

By simply-typed, we mean that polymorphic types and user-defined type constructors are banned. Types include integers `I`, floats `F`, and the rest of Java's primitive types. `V` is the void type, used for methods or functions which do not return a value. Class or interface names c also serve as types. c^0 indicates an *uninitialized* object of class c. (We describe our strategy for object initialization verification in §4.) The set type $\{\overline{c}\}$ represents a union. Normally we can treat $\{a, b, c\}$ as equivalent to the name of the class or interface which is the *least common ancestor* of a, b, and c in the class hierarchy. (For interfaces, however, a usable ancestor does not always exist; see §4.) Finally, $(\overline{\tau}) \to \tau$ is the type of a λJVM function with multiple arguments.

Values include names x (introduced by `let`), constants of various types, the null constant (`null`$[\tau]$ for a given array or object type τ), and, finally, anonymous functions $\lambda\,(\overline{x : \tau})\,e$. The names and types of arguments are written inside the parentheses, and followed by e, the function body.

Terms include two binding forms: `letrec` binds a set of mutually recursive functions; `let` $x = p$; e executes the primitive operation p, binds the result to x, and continues executing e. If we are uninterested in the result of a primop (or it does not produce a result), the sequencing form p; e may be used instead of `let`. Conditional branches are used for numeric comparisons and for testing whether reference values are null. For brevity, we omit the `lookupswitch` and `tableswitch` of the Java virtual machine. Finally, the base

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

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

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

Primops $p ::= \texttt{new}\ c \mid \texttt{chkcast}\ c\ v \mid \texttt{instanceof}\ c\ v$
$\mid \texttt{getfield}\ fd\ v_o \mid \texttt{putfield}\ fd\ v_o\ v$
$\mid \texttt{getstatic}\ fd \mid \texttt{putstatic}\ fd\ v$
$\mid \texttt{invokevirtual}\ md\ v_o\ (\overline{v})$
$\mid \texttt{invokeinterface}\ md\ v_o\ (\overline{v})$
$\mid \texttt{invokespecial}\ md\ v_o\ (\overline{v})$
$\mid \texttt{invokestatic}\ md\ (\overline{v})$
$\mid bo[\tau]\ v\ v \mid \texttt{neg}[\tau]\ v \mid \texttt{convert}[\tau_0, \tau_1]\ v$
$\mid \texttt{newarray}[\tau]\ v_n \mid \texttt{arraylength}[\tau]\ v_a$
$\mid \texttt{aload}[\tau]\ v_a\ v_i \mid \texttt{astore}[\tau]\ v_a\ v_i\ v_o$

Branches $br ::= \texttt{eq} \mid \texttt{ne} \mid \texttt{lt} \mid \texttt{le} \mid \texttt{gt} \mid \texttt{ge}$

Binops $bo ::= br \mid \texttt{add} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{and} \mid \texttt{or} \mid \ldots$

Field descriptor $fd ::= \tau\ c.f$

Method descriptor $md ::= c.m(\overline{\tau})\,\tau$

Figure 1: Syntax of λJVM method bodies. Meta-variable c ranges over class and interface names; x ranges over value names; i, r, and s range over integer, floating point, and string constants, respectively. The overline ($\overline{v}$) indicates zero or more delimited occurrences.

cases can return (with an optional value), call a function, or throw an exception.

The primitive operations cover those JVM instructions which are not for control flow or stack manipulation. They may be grouped into three categories: object, numeric, and array. Object primops include `new`, the dynamic cast and `instanceof` predicate, field accesses, and method calls. Field and method descriptors include the class name and type, just as they do in the JVM. Numeric primops are the usual arithmetic and conversions; branches, when used as primops, return boolean values. Array primops create, subscript, update, and return the length of an array. We use square brackets for type parameters which, in JVML, are part of the instruction. Thus, `iadd` is expressed as `add[I]`, `fmul` is `mul[F]`, and `i2f` is `convert[I,F]`. We omit multi-dimensional arrays, `monitorenter` and `monitorexit` for brevity.

There are three important things to note about λJVM. First, it is *functional*. There is no operand stack, no local variable assignment, and all data flow is explicit. Second, it is impossible to call a function and continue executing after it returns: `let` $y = f(x)$`;` ... is not valid syntax. Therefore all function calls can be implemented as jumps. (Tail call optimization is standard practice in compilers for functional languages.) This makes λJVM functions very lightweight; more akin to basic blocks than to functions in C.

Third, functions are first class and lexically scoped. We will use higher-order functions to implement subroutines, and they may be useful for exception handlers as well. Importantly, functions in λJVM cannot escape from the method in which they are declared. Except for the entry point of a method, call sites of all λ-functions are known. This means a compiler is free to use the most efficient calling convention it can find. Typically, each higher-order function is represented as a closure – a pair of a function pointer and an environment containing values for the free variables [15]. This representation is convenient, consistent, and compatible with separate compilation, but many other techniques are available. Our implementation currently uses *defunctionalization* [25].

Kelsey and Appel [13, 3] have observed that A-normal form for functional programs is equivalent to static single assignment (SSA) form used in many optimizing compilers to make analyses clean and efficient. This is why λJVM is preferable to stack-based Java bytecode for virtual machines (or ahead-of-time compilers) that optimize methods and produce native code – it is already in a format suitable for analysis, optimization, and code generation. Furthermore, as we discuss in §4, type checking for λJVM is far simpler than standard class file verification.

3. TRANSLATION

In this section, we describe how to translate JVM bytecode to λJVM. Figure 2(a) contains a simple Java method which creates objects, invokes a method, and updates a loop counter. Suppose that `IntPt` and `ColorPt` are both subclasses of `Pt`. With this example, we will demonstrate set types and mutable variable elimination.

Figure 2(b) shows the bytecode produced by the Sun Java compiler. The first step in transforming the bytecode to λJVM is to find the basic blocks. This method begins with a block which allocates and initializes an `IntPt`, initializes `j`, then jumps directly to block `C`. `C` does the loop test and either jumps to `B` (the loop body) or falls through and returns. The loop body creates a `ColorPt`, updates the loop counter, and then falls through to the loop test.

Next, data flow analysis must infer types for the stack and local variables at each program point. This analysis is also needed during bytecode verification. In the beginning, we know that local variable 0 contains the method argument `i` and the stack is empty. (For virtual methods, local 0 contains `this`.) Symbolic execution of the first block reveals that, upon jumping to `C`, local 1 contains an `IntPt` and local 2 contains an `int`. We propagate these types into block `C`, and from there into block `B`. During symbolic execution of `B`, we store a `ColorPt` into local 1. Since the current type of local 1 is `IntPt`, we must unify these. Fortunately, we can unify these in λJVM without even knowing where they fit into the class hierarchy – we simply place them into a set type. Now local 1 has type `{IntPt, ColorPt}`. If two types cannot be unified (`int` and `IntPt`, for example), then the variable is marked as unusable (`void`). Block `C` is a successor of `B`, and since the type of local 1 has changed, we must check it again. Nothing else changes, so the data flow is complete and we know the types of the locals and the stack at the start of each block.

Next we use symbolic execution to translate each block to a λ-function. The type annotations within each λ-binding come directly from the type inference. For each instruction which pushes a value onto the operand stack, we push a value (either a fresh name or a constant) onto the symbolic stack. For each instruction which fetches its operands from the stack, we harvest the values from the symbolic stack and emit the corresponding primop. Figure 2(c) shows the resulting code. The method is a λ-function with an argument *i*. *B* and *C* are functions implementing the basic blocks of the same name, and the code of the first block follows. The loop counter is updated by passing a

(a) Java source

```
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;
}
```

(b) Java VM bytecode

```
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
```

(c) λJVM code

```
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 invokevirtual Pt.draw()V p ( );
                return.
       B = λ(p : {IntPt, ColorPt}, j : I)
           let q = new ColorPt;
           invokespecial ColorPt.<init>(I)V q (j);
           let k = mul[I] j 2;
           C(q, k).
let r = new IntPt;
invokespecial IntPt.<init>(I)V r (i);
C(r, 1)
```

Figure 2: A sample method expressed as Java source (a), in the stack-based JVM bytecode (b), and in λJVM (c).

new value to function C each time around the loop. Since the argument `i` is unchanged in the method, we have lifted its binding so that the other two blocks are within its scope. (Our implementation does not currently do this, but it is a simple transformation.)

We used this rather simple example to illustrate the basic principles, but two JVML features prove quite challenging: subroutines and exception handlers.

Subroutines

The Java compiler uses subroutines to implement `finally` blocks [18]. Other compilers that target JVML could, of course, use them for other reasons. The `jsr` instruction pushes a return address onto the stack and transfers control to the specified label. The `ret` instruction jumps back to the return address in the specified local variable.

Subroutines pose three major challenges. First, they are "polymorphic over the types of the locations they do not touch" [29]. As long as a subroutine ignores local 2, say, it could contain an integer at one call site and a float at another. Second, since return addresses can be stored in local variables, subroutine calls and returns need not obey a stack discipline. Indeed, they need not return at all. In Java, we need only to place a `break` or `continue` inside a `finally` block to produce a subroutine which ignores its return address and jumps elsewhere. Finally, a subroutine might update a local variable. Since locals are not mutable in λJVM, the subroutine must explicitly pass the new value back to the caller.

We solve these problems using the *continuation-passing* idiom from functional programming. The subroutine takes a higher-order function (called the *return continuation*) in place of a return address. Any values the subroutine might change are passed to the return continuation as arguments; any free variables in the continuation are preserved across the call.

An example is worthwhile. The subroutine `S` in the following code has two call sites. In the first, local 1 is uninitialized; in the second, it contains a string. The subroutine either updates local 0 and returns normally or jumps directly to the end of the method.

```
public static f(I)V
   jsr S
   ldc "Hello"          S: astore_2  ; ret addr
   astore_1                iload_0
L: jsr S                   ifeq R
   aload_1                 iinc 0 -1
   invoke println          ret 2
   goto L               R: return
```

Bytecode verification is much trickier in the presence of subroutines, and our type inference phase is no different. We must unify the types of locals at different call sites, and decide which are passed to the subroutine, which are passed back to the caller, and which are otherwise preserved across the call. A translation of the example appears below.

```
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 subroutine S takes an argument r of type `(I)` → `V`; this is the return continuation. In one branch, it returns from the method, in the other, it jumps to the continuation, passing the new value of local 0. Now consider the two call sites of S. Inside L the string s is a free variable of the functional argument, so it is preserved across the call.

This solution works quite well. We used Jasmin, a JVML assembler [19], to generate a series of convoluted test cases which do not arise from typical Java compilers. Our code translated all of them correctly. We emphasize again that these higher-order functions can be compiled away quite efficiently in λJVM since all call sites are known.

Exception handlers
Unfortunately, exception handlers do not fit as nicely into λJVM. We have many ideas, but have not yet settled on a solution. The exception table in JVML is not structured like the try/catch blocks of Java. Protected regions need not be properly nested, nor do they necessarily correspond to blocks induced by non-exceptional control flow. One can use exceptions and handlers to implement arbitrary jumps and loops. These features complicate analysis, to be sure, but they are tractable.

The primary difficulty is in propagating the values of local variables into the handler. Since control can jump to the handler from outside the current method, local values used by the handler cannot so easily be kept in registers. This problem is all too familiar to those who optimize Java bytecode in the presence of exceptions [11].

One solution requires adding mutable cells to λJVM. Any locals used in an exception handler could then be stored in memory rather than passed around in registers. The analysis required to implement this scheme is far from trivial, particularly if we want to reduce loads and stores. Compiling Java directly, we could use the structure of the try blocks to guide storage of local variables. Unfortunately this structure is not present in the class file and may not even exist.

Another possibility is to use continuation-passing style at the method level. Once we translate to FLINT, null pointer checks and the like are exposed, so it is possible for locally-raised exceptions to jump directly to the handler. To accommodate externally-raised exceptions, we could pass an *error continuation* to each method we invoke. Values which must be propagated to the handler are simply free variables in the error continuation. This technique should work quite well in FLINT, but we are not yet certain how best to express such handlers in λJVM.

4. VERIFICATION

The JVM specification [18] defines a conservative static analysis for verifying the safety of a class file. Code which passes verification should not, among other things, be able to corrupt the virtual machine which executes it. One of the primary benefits of λJVM is that verification reduces to simple type checking. Most of the analysis required for verification is performed during translation to λJVM. The results are then preserved in type annotations, so type checking can be done in one pass. Our type checker is less than 260 lines of ML code, excluding the λJVM data structure definitions.

Two of the most complex aspects of class file verification are subroutines [29] and object initialization [7]. We have already seen how subroutines disappear, but let us explore in detail the problem of object initialization.

Object initialization
Our explanation of the problem follows that of Freund and Mitchell [7]. In Java source code, the `new` syntax allocates and initializes an object simultaneously:

```
Pt p = new Pt(i);   p.draw();
```

In bytecode, however, these are separate instructions:

```
new Pt                          ; alloc
dup
iload_0
invokespecial Pt.<init>(I)V     ; init
invokevirtual Pt.draw()V        ; use
```

Between allocation and initialization, the pointer can be duplicated, swapped, stored in local variables, etc. Once we invoke the initializer, all instances of the pointer become safe to use. We must track these instances with some form of alias analysis. The following code creates two points; the verifier must determine whether the drawn point is properly initialized.

```
1: new Pt
2: dup
3: new Pt
4: swap
5: invokespecial Pt.<init>()V
6: pop
7: invokevirtual Pt.draw()V
```

This code would be incorrect without the `pop`.

Lindholm and Yellin [18] describe the conservative alias analysis used by the Sun verifier. The effect of the `new` instruction is modeled by pushing the `Pt` type onto the stack along with an 'uninitialized' tag and the offset of the instruction which created it. To model the effect of the initializer, update all types with the same instruction offset, marking them as initialized. Finally, uninitialized objects must not exist anywhere in memory during a backward branch.

In λJVM, many aliases disappear with the local variable and stack manipulations. Every value has a name and a type. The `new` primop introduces a name with uninitialized object type c^0. The initializer then updates the type of its named argument in the environment. After translating the previous example to λJVM, it is clear that the drawn object is initialized:

```
let x = new Pt;
let y = new Pt;
invokespecial Pt.<init>()V x;
invokevirtual Pt.draw()V x;
```

After `invokespecial`, the type environment contains $x \mapsto$ `Pt` and $y \mapsto \texttt{Pt}^0$.

Aliases can occur in λJVM when the same pointer is passed as two arguments to a basic block. Translating the Java statement `new Pt (f? x : y)` to JVML introduces a branch between the `new` and the `invokespecial`. A naïve translation to λJVM might introduce an alias because the same pointer exists in two locations across basic blocks. Our inference algorithm must employ the same technique as the Sun verifier – mark the uninitialized object arguments with the offset of the `new` instruction. Then, we can recognize arguments that are aliases and coalesce them.

Subtyping and set types
Two other interesting aspects of the λJVM type system are the subtype relation and the set types. The subtype relation ($\tau \leqslant \tau'$) handles numeric promotions such as `I` $\leqslant$ `F`. On class and interface names it mirrors the class hierarchy. The rules for other types are as follows:

$$\frac{c \in \{\overline{c}\}}{c \leqslant \{\overline{c}\}} \qquad \frac{\{\overline{c_1}\} \subseteq \{\overline{c_2}\}}{\{\overline{c_1}\} \leqslant \{\overline{c_2}\}}$$

$$\frac{\tau \leqslant \tau'}{\tau[] \leqslant \tau'[]} \qquad \frac{c \leqslant c' \quad \forall c \in \{\overline{c}\}}{\{\overline{c}\} \leqslant c'} \ (*)$$

where the curly typewriter braces `{ · }` are the λJVM set types and the Roman braces $\{\cdot\}$ are standard set notation.

The set elimination rule $(*)$ is required when a value of set type is used in a primop with a field or method

descriptor. In function C of figure 2(c), for example, p is used as the self argument for method `draw` in class `Pt`. The type of p is `{IntPt, ColorPt}`, so the type checker requires `IntPt` $\leqslant$ `Pt` and `ColorPt` $\leqslant$ `Pt`.

In our example, we could have used the super class type `Pt` in place of the set `{IntPt, ColorPt}`, but with interfaces and multiple inheritance, this is not always possible. Both Goldberg and Qian have observed this problem [9, 24]; the following example is from Knoblock and Rehof [14]:

```
interface SA { void saMeth(); }
interface SB { void sbMeth(); }
interface A extends SA, SB { ... }
interface B extends SA, SB { ... }

public static void (boolean f, A a, B b) {
    if (f) { x = a; }
    else { x = b; }
    x.saMeth();
    x.sbMeth();
}
```

What is the type of `x` after the join? The only common super type of `A` and `B` is `Object`. But then the method invocations would not be correct. We must assign to `x` the set type `{A, B}`. For the first method invocation, the type checker requires that `{A, B}` $\leqslant$ `SA`. For the second invocation, `{A, B}` $\leqslant$ `SB`. These subtyping judgments are easily derived from the interface hierarchy (`A` $\leqslant$ `SA`, `B` $\leqslant$ `SA`, `A` $\leqslant$ `SB`, and `B` $\leqslant$ `SB`) using the set elimination rule ($*$).

We utilize subtypes either by subsumption (if v has type τ and $\tau \leqslant \tau'$ then v also has type τ') or as explicit coercions (`let` x = `convert`$[\tau, \tau']$ v; ... where $\tau \leqslant \tau'$). Our type checker accepts the former but can automatically insert explicit coercions as needed.

5. IMPLEMENTATION

We have implemented the translation from Java class files to λJVM. After parsing the class file into a more abstract form, methods are split into basic blocks. Next, type inference determines the argument types for each basic block. It determines subroutine calling conventions and eliminates aliased arguments. Finally, the translation phase uses the results of type inference to guide conversion to λJVM. Our type checker verifies that the translation produced something sensible and checks those JVM constraints that were not already handled during parsing and inference. Although we translate exception-handling code, jumps to the handlers are currently omitted – exceptions can be thrown, but not caught. Subroutine calls to `finally` blocks work fine in non-exceptional cases.

The next stage in our application compiles λJVM to FLINT. We developed a type-theoretic encoding of Java objects and classes which, at run time, behaves just like a standard implementation using per-class dispatch tables and self-application [16, 17]. Higher-order functions in λJVM are eliminated using defunctionalization [25]. The implementation of this stage is still in progress; we can currently handle primitive types, subroutines, inheritance, fields, static and virtual method invocation, and arrays of primitive types.

Our FLINT/ML compiler [26] then performs standard optimizations, closure conversion, and code generation. Thus we can compile toy Java programs through λJVM into FLINT and then into native code. Intermediate code type-checks after every phase. Preliminary measurements of both compilation and run times are promising, but some work remains before we can run real benchmarks.

Because our application does not require it, we have not implemented serialization for λJVM programs. We could borrow the byte codes from JVML for primops, and then use relative instruction offsets for representing `let`-bound names. Or we could follow Amme et al. [2], who describe two innovative encoding techniques – referential integrity and type separation – in which only well-formed programs can be specified. That is, programs are well-formed by virtue of their encoding.

6. RELATED WORK

Katsumata and Ohori [12] translate a subset of JVML into a λ-calculus by regarding programs as *proofs* in different systems of propositional logic. JVML programs correspond to proofs of the sequent calculus; λ-programs correspond to natural deductions. Translations between these systems yield translations of the underlying programs. This is a very elegant approach – translated programs are type-correct by construction. Unfortunately, it seems impossible to extend it to include JVML subroutines and exceptions.

Gagnon et al. [8] give an algorithm to infer static types for local variables in JVML. Since they do not use a single-assignment form, they must occasionally split variables into their separate uses. Since they do not support set types, they insert explicit type casts to solve the multiple interface problem described above.

Amme et al. [2] translate Java to SafeTSA, an alternative mobile code representation based on SSA form. Since they start with Java, they avoid the complications of subroutines as well as the multiple interface problem. Basic blocks must be split wherever exceptions can occur, and control-flow edges are added to the `catch` and `finally` blocks. Otherwise, SafeTSA is similar in spirit to λJVM.

7. CONCLUSION

We have described the design and implementation of λJVM, a functional representation of Java bytecode which makes data flow explicit, verification simple, and which is well-suited for translation into lower-level representations such as those used in optimizing compilers.

λJVM is particularly successful as an intermediate point between Java bytecode and the kind of A-normal form typed λ-calculus typically used for compiling functional languages [21, 23, 27]. It supports a critical separation of concerns by abstracting away the details of JVML and providing a model of control and data flow closer to that of a functional language. Given an encoding of objects and classes in the target language, the Java primitives of λJVM can then be compiled away [16, 17].

We use λJVM as one component of a type-preserving compiler for Java class files. Though the implementation is still in progress, we can compile simple Java programs compile successfully through λJVM into FLINT, and then into native code.

ACKNOWLEDGMENTS

We wish to thank Stefan Monnier for insightful discussions on closure representations and exception handling. John Garvin implemented the class file parser and basic block algorithm. Daniel Dormont contributed preliminary code

for analyzing exception handlers. They both spotted and fixed many bugs.

REFERENCES

[1] B. Alpern, M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proc. Symp. on Principles of Program. Lang.*, pages 1–11, January 1988.

[2] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proc. Conf. on Program. Lang. Design Impl.* ACM, 2001.

[3] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, April 1998.

[4] A. W. Appel. Foundational proof-carrying code. In *Proc. IEEE Symp. on Logic in Computer Science (LICS)*, June 2001.

[5] H. Barendregt. Typed lambda calculi. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford, 1992.

[6] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. Conf. on Program. Lang. Design Impl.*, pages 237–247, Albuquerque, June 1993.

[7] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *Trans. Program. Lang. and Syst.*, 21(6):1196–1250, 1999.

[8] E. Gagnon, L. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Proc. Static Analysis Symp.*, 2000.

[9] A. Goldberg. A specification of Java loading and bytecode verification. In *Conf. on Computer and Comm. Security*, pages 49–58. ACM, 1998.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Mass., 2nd edition, 2000.

[11] M. Gupta, J. Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *Proc. 14th European Conf. on Object-Oriented Program.*, Cannes, June 2000.

[12] S. Katsumata and A. Ohori. Proof-directed decompilation of low-level code. In *Proc. 10th European Symp. on Programming (ESOP)*, volume 2028 of *LNCS*, Geneva, April 2001.

[13] R. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Proc. Workshop on Intermediate Representations*, pages 13–22. ACM, March 1995.

[14] T. Knoblock and J. Rehof. Type elaboration and subtype completion for Java bytecode. In *Proc. Symp. on Principles of Program. Lang.*, pages 228–242, 2000.

[15] P. Landin. The mechanical evaluation of expressions. *Computer J.*, 6(4):308–320, 1964.

[16] C. League, Z. Shao, and V. Trifonov. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. Funct. Program.*, pages 183–196, New York, September 1999. ACM.

[17] C. League, V. Trifonov, and Z. Shao. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop Found. Object-Oriented Lang.*, London, January 2001.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[19] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[20] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. Workshop on Compiler Support for Syst. Soft.*, pages 25–35, New York, May 1999. ACM.

[21] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Proc. Workshop on Compiler Support for Syst. Soft.*, New York, 1996. ACM.

[22] G. C. Necula. Proof-carrying code. In *Proc. Symp. on Principles of Program. Lang.*, pages 106–119, New York, January 1997. ACM.

[23] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: A technical overview. In *Proc. UK Joint Framework for Inform. Tech.*, December 1992.

[24] Z. Qian. A formal specification for Java Virtual Machine instructions for objects, methods, and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 271–312. Springer, 1999.

[25] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. 25th ACM Nat'l Conf.*, pages 717–740, Boston, 1972.

[26] Z. Shao. An overview of the FLINT/ML compiler. In *Proc. Int'l Workshop on Types in Compilation*, Amsterdam, June 1997.

[27] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proc. Conf. on Program. Lang. Design Impl.*, pages 116–129, New York, June 1995. ACM.

[28] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *Proc. Int'l Conf. Funct. Program.*, pages 313–323, New York, September 1998. ACM.

[29] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. Symp. on Principles of Program. Lang.*, pages 149–160, San Diego, January 1998. ACM.