Precision in Practice: A Type-Preserving Java™ Compiler*

Christopher League Zhong Shao Valery Trifonov

Technical Report YALEU/DCS/TR-1223
Yale University Computer Science Department
POB 208285, New Haven, CT 06511 USA
{league, shao, trifonov}@cs.yale.edu

pre·ci·sion *n.* **2a:** the degree of refinement with which an operation is performed. [Merriam-Webster]

Abstract

Popular mobile code architectures (Java and .NET) include verifiers to check for memory safety and other security properties. Since their formats are relatively high level, supporting a wide range of source language features is awkward. Further compilation and optimization, necessary for efficiency, must be trusted. We describe the design and implementation of a fully type-preserving compiler for Java and SML. Its strongly-typed intermediate language, provides a low-level abstract machine model and a type system general enough to prove the safety of a variety of implementation techniques. We show that precise type preservation is within reach for real-world Java systems.

1 Introduction

There is increasing interest in program distribution formats that can be checked for memory safety and other security properties. The Java Virtual Machine (JVM) [30] performs conservative analyses to determine whether the byte codes of each method are safe to execute. Its *class file* format contains type signatures and other symbolic information that makes verification possible. Likewise, the Common Intermediary Language (CIL) of the Microsoft .NET platform [31] includes type information and defines verification conditions for many of its instructions.

*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. ¶ Java is a registered trademark of Sun Microsystems, Inc. in the U.S. and other countries. CaffeineMark is a trademark of Pendragon Software.

As a general distribution format, JVM class files are very high-level and quite partial to the Java language. The bytecode language (JVML) includes no facilities for specifying data layouts or expressing many common optimizers. Compiling other languages for the JVM means making foreign constructs look and act like Java classes or objects. That so many translations exist [48] is a testament to the utility of the mobile code concept, and to the ubiquity of the JVM itself. To some extent, CIL alleviates these problems. It supports user-defined value types, stack allocation, tail calls, and pointer arithmetic (which is outside the verifiable subset). Even so, a recent proposal to extend CIL for functional language interoperability [46] added no fewer than 6 new types and 12 new instructions (bringing the total number of `call` instructions to 5) and it still does not support ML's higher-order modules [24] or Haskell's constructor classes [25].

Another problem with both of these formats is that they require further compilation and optimization to run efficiently on real hardware. Since these phases occur after verification, they are not guaranteed to preserve the verified safety and security properties. Bugs in the compiler may have security implications, so the entire compiler must be *trusted*.

The idea of type-preserving compilation, then, is to remove the compiler from the trusted code base (TCB) by propagating type information through all the compilation and optimization passes. Every representation from the source down to the object code supports verification. Object formats developed in this context include Typed Assembly Language (TAL) [34] and Proof-Carrying Code (PCC) [36, 38].

Type-preserving compilers have other widely recognized benefits. Many compiler bugs are revealed early by verifying the intermediate code after each pass. Furthermore, type information can be useful for low-level optimizations [18, 47] and accurate garbage collection [19, 49].

Many compilers preserve *some* kind of type information in their intermediate code (Marmot [20], Intel VM [45], BulletTrain [35]) but none are rigorous enough to support verification. Lower-level code requires more sophisticated type systems. As we demonstrate in the next section, annotations that merely distinguish between integers, floats, and objects of distinct classes are insufficient. Types must enforce subtle invariants, for which logical constructs (such as quantification) are useful.

Representing and manipulating these types can be over-

whelming; early results in this area reported compilation times increasing by an order of magnitude [47]. In a practical compiler, careful design and implementation of the intermediate languages is essential.

Our previous work [27, 29] developed type-theoretic encodings of many Java features. We proved useful properties, such as type preservation and decidability, but always our goal was to implement the encodings in a practical compiler. In fact, we rejected the classic object encodings [9] because their runtime penalties—superfluous indirections and function calls—were too high.

This paper describes the design and implementation of a compiler based on our encodings. To our knowledge, it is the first practical system to use a higher-order polymorphic intermediate language to compile both functional and object-oriented source languages. Additionally, it has the following features:

- Front ends for both Standard ML [32] and JVML that share optimizations and code generators. Programs from either language run together in the same interactive runtime system.
- λJVM, our high-level intermediate language (IL) in the Java front end, uses the same primitive instructions and types as JVML, but is easier to verify and more amenable to optimization (see §3).
- JFlint, our low-level generic IL, includes function declarations, arrays and structures, and the usual branches and numeric primitives. Its type system includes logical quantifiers (universal, existential, fixpoint) and rows [41] for abstracting over structure suffixes. The instruction stream includes explicit type operations that guide the verifier.
- Unlike the CIL extension [46], our design supports a pleasing synergy between the encodings of Java and ML. JFlint does not, for example, treat Java classes or ML modules as primitives. Rather, it provides a low-level abstract machine model and sophisticated types that are general enough to prove the safety of a variety of implementation techniques. We expand on this in §4.
- Nothing about our instruction set should surprise a typical compiler hacker. Type operations must appear periodically, but most occur in canned sequences that can easily be treated as macros. Although the detailed type information can be quite large, our graph representation maintains optimal sharing. Type annotations within the code are merely pointers into this graph. For debugging purposes, it is easy to print the type annotations using short, intuitive names such as `InstOf[java/lang/Object]`. With clever implementation of the type operations, compile times do not explode (see §5).
- All types are discarded after verification, leaving concise and efficient code, *exactly* as an untyped compiler would produce.

Our thesis, in short, is that precise type preservation is within the reach of practical Java systems.

The next section introduces a detailed example to elucidate some of the issues in certifying compilation of object-oriented languages, and to distinguish our approach from that of Cedilla Systems [13]. We postpone discussion of other related projects to §6.

2 Background: self-application and Special J

We begin by attempting to compile the most fundamental operation in object-oriented programming: virtual method invocation.

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

The standard implementation adds an explicit *self* argument (`this`) to each method and collects the methods into a per-class structure called a *vtable*. Each object contains a pointer to the vtable of the class that created it. To invoke a virtual method, we load the vtable pointer from the object, load the method pointer from the vtable, and then call the method, providing the object itself as `this`.

```
public static void deviant (Object x, Object y)
{ if (x is null) throw NullPointerException;
  r1 = x.vtbl;
  r2 = r1.toString;
  call r2 (x); }
```

A certifying compiler must justify that the indirect call to `r2` is safe; this is not at all obvious. Since `x` might be an instance of a subclass, the method in `r2` might require additional fields and methods that are unknown to the caller. Self-application works thanks to a rather subtle invariant. One way to upset that invariant is to select a method from one object and pass it *another* object as the self argument. For example, replace just the last instruction above with `call r2 (y)`.

This might seem harmless; after all, both `x` and `y` are instances of `Object`. It is unsound, however, and any unsoundness can be exploited. Suppose class `Int` extends `Object` by adding an integer field; class `Ref` adds a byte vector and overrides `toString`:

```
class Ref extends Object
 {  public byte[] vec;
    public String toString()
     { vec[13] = 0xFF;  return "Ha ha!"; }
 }
```

Then, calling the deviant method as follows:

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

will jump to `Ref.toString()` with `this` bound to the `Int` object. Thus, we use an arbitrary integer as an array pointer.

How does one express and enforce the self-application invariant, once the operation is expanded to lower-level code? The answer is not widely understood outside the object type theory community (see [9] for a survey). This is one reason why virtual method calls are atomic operations in both JVML and CIL.

Two years ago at PLDI, Colby, Lee, Necula, and others from Cedilla Systems presented initial results about Special J, a proof-carrying code compiler for Java [13]. They described the design, defined some of the predicates used in verification conditions, explained their approach to exceptional control flow, and gave some experimental results. Their running example was hand-optimized code including a loop, an array field, and an exception handler.

Unfortunately, their paper did not adequately describe the safety conditions for virtual method calls. In communication with the authors, we dug deeper and discovered that their current system indeed does *not* enforce the self-application invariant properly [37]. It gives the type "vtable of `Object`" to `r1` and the type "implementation of `String Object.toString()`" to `r2`. The verification condition for the call requires only that the static class of the self argument matches the static class of the object from which the method was fetched. As a result, the client's proof checker will, for example, *accept* the malicious code given above.

Necula claims that this hole can be patched [37]. Without a detailed proposal, we remain skeptical. Its existence, undetected for two years, raises suspicion about the integrity of the proof rules used by Special J. The issue is that the rules for the *source* language are part of the trusted code base. If they are unsound, all bets are off. Moreover, the rules and the code work at different levels. PCC is machine code, but its logical predicates refer specifically to Java constructs such as objects, interfaces, and methods. To support another language, an entirely new set of language-specific predicates and rules must be added to the TCB.

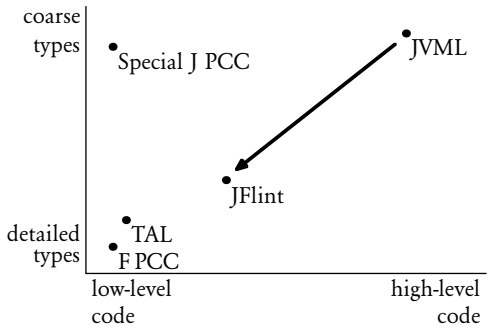

Figure 1: The landscape of verifiable languages. Off the diagonal, the "granularity gap" provides more opportunities for soundness errors.

To illustrate this discord, we chart languages by the granularity of their instructions and types (see figure 1). Code granularity is straightforward: higher-level languages have complex primitives like invokevirtual and checkcast; the lowest level has explicit registers and calling conventions. Compilers transform the former to the latter. Type granularity is less clear; intuitively, detailed types encode representations and invariants while coarse types hide or ignore them. These notions are not particularly quantifiable (and not the only dimensions that matter), but the graph is telling nonetheless.

We place the output of Special J near the upper left because its verifier and many axioms operate at the level of Java, even though the code is machine-level. JFlint, in contrast, is on the diagonal because its types encode the requisite invariants with precisely the granularity at which the code operates—that of functions and structures. JFlint is still a few steps from machine code, but both TAL [34] and *foundational* proof-carrying code (in which the typing rules are proved as lemmas in higher-order logic [4, 3]) are closely aligned with our approach.

In the absence of formal proofs of the trusted rules, we have more confidence in systems on the diagonal. While it is possible to build a correct system elsewhere, we fear the "granularity gap" provides ample opportunities for soundness errors.

In the next section, we briefly survey the architecture of our compiler. Its key strongly-typed intermediate language is the topic of section 4.

3 Architecture of our compiler

Standard ML of New Jersey is an interactive runtime system and compiler based on a strongly-typed intermediate language called FLINT [43, 44]. We extended the FLINT language of version 110.30 and implemented a new front end for Java class files. We updated the optimization phases to recognize the new features. The code generator and runtime system remain unchanged.

The Java front end parses class files and converts them to a high-level IL called λJVM. This language uses the same primitive instructions and types as JVML; on our chart in figure 1, it would be in the upper-right corner with JVML. The difference is that λJVM replaces the implicit operand stack and untyped local variables with explicit data flow and fully-typed single-assignment bindings. λJVM includes types for marking *uninitialized* objects, and set types $\{\overline{c}\}$. 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 [26].)

This alternate representation has several advantages. First, it is simpler to verify than JVML, because all the hard analyses (object initialization, subroutines, *etc.*) are performed during translation and their results preserved in type annotations. The type checker for λJVM is just 260 lines of SML code. Second, as a *functional* IL, it is (like static single assignment form) amenable to further analysis and optimization [2, 14]. Although we have not implemented them, this phase would be suitable for class hierarchy analysis and various object-aware optimizations [17, 16] because the class hierarchy and method invocations are still explicit.

We illustrate briefly 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`. 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. 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.) Next we use symbolic execution to translate each

(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 fun C (p:{IntPt,ColorPt}, j:I)
           if lt[I] j i then B(p,j)
           else invokevirtual Pt.draw()V p ();
                return;
       fun B (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).

block to a function. The type annotations within each formal parameter list come directly from the type inference. Figure 2(c) shows the resulting code. The method is a 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 variable is updated by passing a new value to function C each time around the loop.

We designed λJVM so that its control and data flow mimic that of JFlint. This means that the next phase of our compiler is simply an *expansion* of the JVML types and operations into more detailed types and lower-level code. For further details about λJVM, please see [28].

On JFlint, we run several contraction optimizations (inlining, common subexpression elimination, *etc.*), and typecheck the code after each pass. We discard the type information before converting to MLRISC [22] for final instruction selection and register allocation. To generate typed machine code, we would need to preserve types throughout the back end. The techniques of Morrisett *et al.* [34] should apply directly, since JFlint is based on System F.

Figure 3 demonstrates the SML/JFlint system in action. The top-level loop accepts Standard ML code, as usual. The JFlint subsystem is controlled via the `Java` structure; its members include:

- `Java.classPath : string list ref`
  Initialized from the `CLASSPATH` environment variable, this is a list of directories where the loader will look for class files.
- `Java.load : string -> unit`
  looks up the named class using `classPath`, resolves and loads any dependencies, then compiles the byte codes and executes the class initializer.
- `Java.run : string -> string list -> unit`
  ensures that the named class is loaded, then attempts to call its `main` method with the given arguments.
- `Java.flush : unit -> unit`
  forces the Java subsystem to forget all previously loaded classes. Normally, loaded classes persist across calls to `load` and `run`; after `flush`, they must be loaded and

```
Standard ML of New Jersey v110.30 [JFLINT 1.2]
- Java.classPath := ["/home/league/r/java/tests"];
val it = () : unit
- val main = Java.run "Hello";
[parsing Hello]
[parsing java/lang/Object]
[compiling java/lang/Object]
[compiling Hello]
[initializing java/lang/Object]
[initializing Hello]
val main = fn : string list -> unit
- main ["Duke"];
Hello, Duke
val it = () : unit
- main [];
uncaught exception ArrayIndexOutOfBounds
  raised at: Hello.main([Ljava/lang/String;)V
- ^D
```

Figure 3: Compiling and running a Java program using JFlint in the SML/NJ interactive runtime system.

initialized again.

The session in figure 3 sets the `classPath`, loads the `Hello` class, and binds its `main` method, using partial application of `Java.run`. The method is then invoked twice with different arguments. The second invocation erroneously accesses `argv[0]`; this error surfaces as the ML exception `Java.ArrayIndexOutOfBounds`.

This demonstration shows SML code interacting with a complete Java program. Since both run in the same runtime system, very fine-grained interactions are possible, but have not been our primary focus. Benton and Kennedy [6] designed extensions to SML to allow seamless interaction with Java code when both are compiled for the Java virtual machine. Their design should work quite well in our setting also.

Ours is essentially a *static* Java compiler, as it does not handle dynamic class loading or the `java.lang.reflect` API. These features are more difficult to verify using a static type system, but they are topics of active research. Also, our runtime system does not, for now, dynamically load native code. This is a dubious practice anyway; such code has free reign over the runtime system, thus nullifying any safety guarantees won by verifying pure code. Nevertheless, this restriction is unfortunate because it limits the set of existing Java libraries that we can use.

4 Overview of the JFlint IL

To introduce the JFlint language, we begin with a second look at virtual method invocation in Java: below is the expansion into JFlint of a Java method that takes `Objects` `x` and `y` and calls `x.toString()`.

```
 obedient (x, y : InstOf[java/lang/Object]?) =
  switch (x)
   case null: throw NullPointerException;
   case non-null x1:
.    <f1,m1; x2 : Self[java/lang/Object] f1 m1>
.       = OPEN x1;
.    x3 = UNFOLD x2;
     r1 = x3.vtbl;
     r2 = r1.toString;
     call r2 (x2);
```

The annotations `InstOf` and `Self` abbreviate more detailed types. The postfix `?` indicates that the arguments could be null. The code contains the same operations as before: null check, two loads, and a call. The null check is expressed as a `switch` that, in the non-null case, binds the new identifier `x1` to the value of `x`, but now with type `InstOf[java/lang/Object]` (losing the `?`). It is customary to use new names whenever values change type, as this dramatically simplifies type checking.

4.1 Type operations

The new instructions following the null check (`OPEN` and `UNFOLD`) are type operations. `InstOf` abbreviates a particular existential type (we clarify the meanings of the various types in section 4.4):

```
InstOf[java/lang/Object] =
     exists f0, m0: Self[java/lang/Object] f0 m0
```

`OPEN` eliminates the existential by binding fresh type variables (`f1` and `m1` in the example) to the hidden witness types. Likewise, `Self` abbreviates a fixpoint type:

```
Self[java/lang/Object] fi mi =
     fixpt s0:
         { vtbl : Meths[java/lang/Object] s0 mi;
           hash : int;
           fi }

Meths[java/lang/Object] sj mj =
         { toString : sj -> void;
           hashCode : sj -> int;
           mj(sj) }
```

`UNFOLD` eliminates the fixpoint by replacing occurrences of the bound variable `s0` with the fixpoint type itself. These operations leave us with a *structural* view of the object bound to `x3`; it is a pointer to a record of fields prefixed by the vtable (a pointer to a sequence of functions). Importantly, the fresh type variables introduced by the `OPEN` (`f1` and `m1`) find their way into the types of the vtable functions. Specifically, `r2` points to a function of type `Self[java/lang/Object] f1 m1 -> void`. Thus the only valid self argument for `r2` is `x2`. The malicious code of section 2 is rejected because opening `y` would introduce brand new type variables (`f2` and `m2`, say); these never match the variables in the type of `r2`.

We observe the following about the type operations. After the final verification, they are completely discarded and the aliased identifiers are renamed. This *erasure* leaves us with *precisely* the same operational behavior that we used in an untyped setting. Also, like other instructions, type manipulations yield to simple optimizations. We can, for example, eliminate redundant opens and hoist loop-invariant unfolds. In fact, using *online* common subexpression elimination, we avoid emitting redundant operations in the first place. For a series of method calls and field accesses on the same object, we would `OPEN` and `UNFOLD` it just once. Although the type operations have no runtime penalty, optimizing them is advantageous. First, fewer type operations means smaller programs and faster compilation and verification. Second, excess type operations often hide further optimization opportunities in runtime code.

4.2 Code representation

Our examples use a pretty-printed surface syntax for JFlint. Figure 4 contains a portion of the SML signature for representing such code in our compiler. Identifiers and constants comprise *values*. Instructions operate on values and bind their results to new names. Loads and stores on structures refer to the integer offset of the field. Function declarations have type annotations on the formal parameters. Non-escaping functions whose call sites are all in tail position are akin to basic blocks.

This language is closer to machine code than to JVML, but not quite as low-level as typed assembly language. Allocating and initializing a structure, for example, is one instruction: `STRUCT`. Similarly, the `CALL` instruction passes *n*

```
signature JFLINT = sig
 datatype value              (* identifiers and constants *)
  = VAR of id | INT of Int32.int | STRING ...

 datatype exp
  = LETREC of fundec list * exp
  | LET    of id * exp * exp
  | SWITCH of value * (case * exp) list
  | CALL   of id * value list
  | RETURN of value
  | PRIMOP of primop * value list       * id * exp
  | STRUCT of value list                * id * exp
  | LOAD   of value * int               * id * exp
  | STORE  of value * int * value           * exp
  ...                    (* type manipulation instructions *)
  | INST   of id * ty list              * id * exp
  | FOLD   of value * ty                * id * exp
  | UNFOLD of value                     * id * exp
  | PACK   of ty list * (value*ty) list * id * exp
  | OPEN   of value * id list * (id*ty) list * exp
  ...
 withtype fundec = id * (id * ty) list * exp
end
```

Figure 4: Representation of JFlint code. Expressions ending with `id * exp` bind their result to identifier `id` in successor `exp`.

arguments and transfers control all at once; the calling convention is not explicit. It is possible to break these down and still preserve verifiability [34], but this midpoint is simpler and still quite useful for optimization.

There are two hurdles for the usual compiler hacker in using a strongly-typed IL like JFlint. The first is simply the functional notation, but it can be understood by analogy to SSA. Moreover, it has additional benefits such as enforcing the dominator property and providing homes for type annotations [2]. The second hurdle is the type operations themselves: knowing where to insert and how to optimize them. The latter is simple; most standard optimizations are trivially type-preserving. Type operations have uses and defs just like other instructions, and type variables behave (in most cases) like any other identifier.

As for knowing what types to define and where in the code to insert the type operations: we developed recipes for Java primitives [27, 29]; some of these appear in figure 5. A thorough understanding of the type system is helpful for developing successful new recipes, but experimentation can be fruitful as long as the type checker is used as a safety net. Extending the type system without forfeiting soundness is, of course, a more delicate enterprise; a competent background in type theory and semantics is essential.

4.3 Interfaces and casts

The open-unfold sequence used in method invocation appears whenever we need to access an object's structure. Getting or setting a field starts the same way: null check, open, unfold (see the first expanded primop in figure 5).

Previously, we showed the expansion of `InstOf[C]` as an existential type. Suppose `D` extends `C`; then, `InstOf[D]` is

```
putfield C.f (x : InstOf[C]?; y : T) ⟹
 switch (x) case null: throw NullPointerException;
 case non-null x1:
.  <f3,m3; x2 : Self[C] f3 m3> = OPEN x1;
.   x3 = UNFOLD x2;
    x3.f := y;

upcast D,C (x : InstOf[D]?) ⟹
 switch (x) case null: return null : InstOf[C]?;
 case non-null x1:
.  <f4,m4; x2 : Self[D] f4 m4> = OPEN x1;
.   x2 = PACK f5=NewFlds[D] f4, m5=NewMeths[D] m4
.        WITH x1 : Self[C] f5 m5;
    return x2 : InstOf[C]?;

upcastinterface C,I (x : InstOf[C]?) ⟹
 switch (x) case null: return null : IfcObj[I]?;
 case non-null x1:
.  <f6,m6; x2 : Self[C] f6 m6> = OPEN x1;
.   x3 = UNFOLD x2;
    r1 = x3.vtbl;
    r2 = r1.I;
    y1 = { itbl = r2; obj = x2 };
.   y2 = PACK t = Self[C] f6 m6
.        WITH y1 : IfcPair[I] t;
    return y2 : IfcObj[I]?;

invokeinterface I.m (x : IfcObj[I]?; v1...vn) ⟹
 switch (x) case null: throw NullPointerException;
 case non-null x1:
.  <t; x1 : IfcPair[I] t> = OPEN x1;
    r1 = x1.itbl;
    r2 = x1.obj;
    r3 = r1.m;
    call r3 (r2, v1, ..., vn);
```

Figure 5: Recipes for some λJVM primitives. Lines preceded by dots contain type operations.

a *different* existential. In Java, any object of type `D` also has type `C`. To realize this property in JFlint, we use explicit type coercions. (This helps keep the type system simple; otherwise we would need F-bounded quantifiers [10] with 'top' subtyping [11].) λJVM marks such coercions as *upcasts*. They are expanded into JFlint code just like other operators.

An upcast should not require any runtime operations. Indeed, apart from the null test, the upcast recipe in figure 5 is nothing but type operations: open the object and repackage it to hide more of the fields and methods. Therefore, only the null test remains after type erasure: (`x == null? null : x`). This can easily be recognized and eliminated during code generation. An alternate formulation of the type system might permit fancier coercions to operate underneath the option type (`?`); we did not find them worthwhile in this case.

In Java, casts from a class to an interface type are also implicit (assuming the class implements the interface). On method calls to objects of interface type, a compiler cannot statically know where to find the interface method. Most implementations use a dynamic search through the vtable to locate either the method itself, or an embedded *itable* containing all the methods of a given interface. This search

```
signature JTYPE = sig
 type ty
 val var    : int * int -> ty          (* type variable *)
 val prim   : primtyc -> ty
 val arrow  : ty list * ty -> ty       (* function type *)
 val struct : ty -> ty                 (* structure types *)
 val row    : ty * ty -> ty
 val empty  : int -> ty
 ...                                   (* quantified types *)
 val forall : kind list * ty list -> ty
 val exists : kind list * ty list -> ty
 val fixpt  : kind list * ty list -> ty
 val lam    : kind list * ty -> ty     (* higher-order *)
 val app    : ty * ty list -> ty
end
```

Figure 6: Abstract interface for JFlint type representation. Type variables use de Bruijn notation (depth × offset) [15].

is expensive, so it pays to cache the results. A type system for verifying the dynamic search and caching code would be quite complex. Moving this particular functionality to the runtime system—as trusted code—is one way to avoid the complexity. We instead use a unique representation of interfaces for which dynamic search is unnecessary [27].

In our system, interface calls are about as cheap as virtual calls (null check, a few loads and an indirect call). We represent interface objects as a pair of the interface method table and the underlying object. To invoke a method, we fetch it from the itable and pass it the object as the self argument. This implies a non-trivial coercion when an object is upcast from a class to an interface type, or from one interface to another: fetch the itable and create the pair. But at this point, layout of the vtable is known, so a dynamic search is unnecessary.

The last two recipes in figure 5 illustrate this technique. The new type abbreviations for representing interface objects are, for example:

```
IfcObj[java/lang/Runnable] =
     exists t . IfcPair[java/lang/Runnable] t

IfcPair[java/lang/Runnable] t =
     { itbl : { run : t -> void },
       obj : t
      }
```

The existential hides the actual class of the object. Just as with virtual invocation, the interface invocation relies on a sophisticated invariant. A method from the itable must be given a compatible object as the self argument. The existential ensures that only the packaged object will be used with methods in the itable.

4.4 Type representation

To support efficient compilation, types are represented differently from code. Figure 6 contains part of the abstract interface to our type system; section 5 describes what lies behind the interface. Most of our types are standard: based on the higher-order polymorphic lambda calculus (see [5] for an overview).

A structure is a pointer to a sequence of fields, but we represent the sequence as a linked list of *rows*. Any *tail* of the list can be replaced with a type variable, providing a handle on suffixes of the structure. The `InstOf` definition used an existential quantifier [33] to hide the types of additional fields and methods; these are rows.

The universal quantifier—precisely the inverse—allows outsiders to provide types; in our encoding, it models inheritance. Subclasses provide new types for the additional fields and methods. *Kinds* classify types and provide bounds for quantified variables. They ensure that rows are composed properly by tracking the structure offset where each row begins [41].

Our object encodings rely only on standard constructs, so our type system is rooted in well-developed type theory and logic. The soundness proof for a similar system is a perennial assignment in our semantics course. The essence was even formalized in machine-checkable form using Twelf [42].

4.5 Synergy

Judging from the popular formats, it appears that there are just two ways to support different kinds of source languages in a single type-safe intermediate language. Either favor one language and make everyone else conform (JVM) or incorporate the union of all the requested features (CIL, ILX [31, 46]). CIL instructions distinguish, for example, between loading functions *vs.* values from objects *vs.* classes. ILX adds instructions to load from closure environments and from algebraic data types.

The (overlooked) third approach is to provide a low-level abstract machine model and general types capable of proving safety of various uses of the machine primitives. Structures in JFlint model Java objects, vtables, classes, and interfaces, plus ML records and the value parts of modules. Neither Java nor ML has a universal quantifier, but it is useful for encoding both Java inheritance and ML polymorphism. The existential type is essential for object encoding but also for ML closures and abstract data types.

We believe this synergy speaks well of our approach in general. Still, it does not mean that we can support all type-safe source languages equally well. Java and ML still have much in common; they work well with precise generational garbage collection and their exceptions are similar enough. Weakly typed formats, such as C-- [39, 40], are more ambitious in supporting a wider variety of language features, including different exception and memory models. Practical type systems to support that level of flexibility are challenging; further research is needed.

5 Implementation concerns and experimental results

If a type-preserving compiler is to scale well, extreme care must be taken in implementing the type representations and operations. Naïveté here can lead to exponential increases in compile time. Several years ago, we presented a series of techniques which, taken together, made the FLINT typed IL practical enough to use in a production compiler [44].

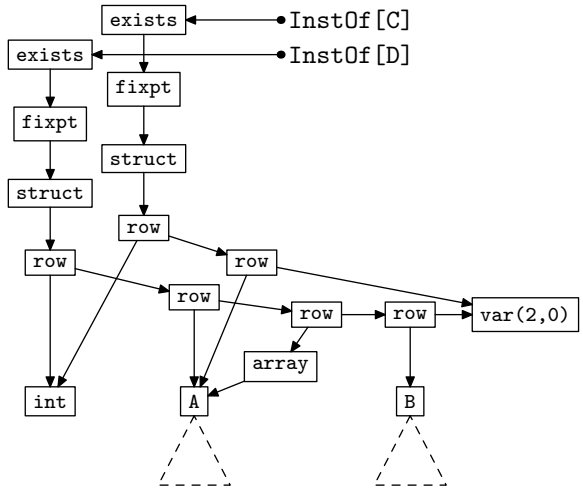

Figure 7: A fragment of the compiler's graph representation of types. Here, `D` extends `C` with two new fields. Common subterms are maximally shared.

Different type structures arise in our Java encodings, but the techniques are as successful as ever.

We represent types as directed acyclic graphs. Part of what the type representation interface (figure 6) hides is automatic hashing to ensure maximal sharing in the graph representation. Figure 7 shows a fragment of the graph representation of the `InstOf` types for class `C` and `D` (which extends `C`). Type variables are represented as pairs of integers which represent the lexical binding depth and offset. This means that types which differ only in their variable names *share* the same representation. Therefore, the equivalence of two types in normal form is simply pointer equivalence.

A common operation on types is the substitution of types for variables. Replacing variables using assignment is not an option because so much of the graph is shared. We create the instantiated type lazily, memoizing each result so that the next time we need the same substitution it is available immediately.

With all these tricks, compile and verification times remain reasonable. A full type-preserving compile of the 12 classes in the CaffeineMark™ 3.0 embedded series takes 2.4 seconds on a 927 MHz Intel Pentium III Linux workstation. This is about 60% more than `gcj`, the GNU Java compiler [8]. Since `gcj` is written in C and our compiler in SML, the gap is easily attributed to linguistic differences. Verifying both the λJVM and the JFlint code adds another half second.

Run times are promising, and can be improved. (Our goal, of course, is to preserve type safety; speed is secondary.) CaffeineMark runs at about a third the speed in SML/NJ compared to `gcj -O2`. There are several reasons for this difference. First, many standard optimizations, especially on loops, have not been implemented in JFlint yet. Second, the code generator is still heavily tuned for SML; structure representations, for example, are more boxed than they should be. Finally, the runtime system is also tuned for SML; to support `callcc`, every activation record is heap-allocated and subject to garbage collection.

Benchmarking is always fraught with peril. In our case, meaningful results are especially elusive because we can only compare with compilers that differ in *many* ways besides type preservation. To obtain more meaningful statistics and prove wider applicability, we are considering a project to integrate our technology into an existing just-in-time compiler, perhaps using the Open Runtime Platform from Intel [12]. We believe it is possible, using the techniques we have presented, to move most of an optimizing JIT out of the trusted code base without appreciable impact on performance.

6 Related work

Throughout the paper, we made comparisons to the Common Intermediary Language (CIL) of the Microsoft .NET platform [31] and ILX, a proposed extension for functional language interoperability [46]. We discussed the proof-carrying code system Special J [13] at length in section 2. We mentioned C-- [39], the portable assembly language, in section 4.5. Several other systems warrant mention.

Benton et al. built MLj, an SML compiler targeting the Java Virtual Machine [7]; we mentioned their extensions for interoperability earlier [6]. Since JVML is less expressive than JFlint, they monomorphize SML polymorphic functions and functors. On some applications, this increases code size dramatically. JVML is less appropriate as an intermediate format for functional languages because it does not model their type systems well. Polymorphic code must either be duplicated or casts must be inserted. JFlint, on the other hand, completely models the type system of SML.

Morrisett et al. developed compilers for various type-safe source languages which emit typed assembly language. Neal Glew describes an encoding for a toy object calculus [23], but has not shown it to scale to a robust implementation of a real class-based object-oriented language.

Many researchers use techniques reminiscent of those in our λJVM translation format. Marmot converts bytecode to a conventional high-level IL using abstract interpretation and type elaboration [20, 26].

Gagnon et al. [21] 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.

Amme et al. [1] translate Java to SafeTSA, an alternative mobile code representation based on SSA form. Since they start with Java, they avoid the complications of subroutines and set types. 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 our type-preserving compiler for both Java and SML. Its strongly-typed intermediate language provides a low-level abstract ma-

chine model and a type system general enough to prove the safety of a variety of implementation techniques. This approach produces a pleasing synergy between the encodings of both languages.

We have shown that type operations can be implemented efficiently and do not get in the way of optimizations or efficient execution. We therefore believe that precise type preservation is within reach for real-world Java systems, including those with just-in-time compilers.

References

[1] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proc. Conf. on Programming Language Design and Implementation*. ACM, 2001.

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

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

[4] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proc. Symp. on Principles of Programming Languages*, pages 243–253, Boston, January 2000. ACM.

[5] Henk Barendregt. Typed lambda calculi. In Samson Abramsky, Dov Gabbay, and Tom Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford, 1992.

[6] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proc. Int'l Conf. Functional Programming*, pages 126–137, Paris, 1999. ACM.

[7] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proc. Int'l Conf. Functional Programming*, pages 129–140, Baltimore, September 1998. ACM.

[8] Per Bothner. A GCC-based Java implementation. In *Proc. IEEE Compcon*, 1997.

[9] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2):108–133, 1999.

[10] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Int'l Conf. on Functional Programming and Computer Architecture*, pages 273–280. ACM, September 1989.

[11] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proc. Symp. on Principles of Programming Languages*, Portland, January 1994. ACM.

[12] Michał Cierniak, Guei-Yuan Lueh, and James Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proc. Conf. on Programming Language Design and Implementation*, Vancouver, June 2000. ACM.

[13] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. A certifying compiler for Java. In *Proc. Conf. on Programming Language Design and Implementation*, Vancouver, June 2000. ACM.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[15] N. G. de Bruijn. A survey of the project AUTOMATH. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[16] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, October 1996. ACM.

[17] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. European Conf. Object-Oriented Programming*, 1995.

[18] Amer Diwan, Kathryn S. McKinley, and Eliot Moss. Type-based alias analysis. In *Proc. Conf. on Programming Language Design and Implementation*, pages 106–117, Montréal, June 1998. ACM.

[19] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically typed language. In *Proc. Conf. on Programming Language Design and Implementation*, pages 273–282. ACM, 1992.

[20] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, 2000.

[21] Etienne Gagnon, Laurie Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proc. Static Analysis Symp.*, 2000.

[22] Lal George. Customizable and reusable code generators. Technical report, Bell Labs, 1997.

[23] Neal Glew. An efficient class and object encoding. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications*. ACM, October 2000.

[24] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proc. Symp. on Principles of Programming Languages*, pages 341–354. ACM, 1990.

[25] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *J. Functional Programming*, 5(1):1–35, 1995.

[26] Todd Knoblock and Jakob Rehof. Type elaboration and subtype completion for Java bytecode. In *Proc. Symp. on Principles of Programming Languages*, pages 228–242, 2000.

[27] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *Proc. Int'l Conf. Functional Programming*, pages 183–196, Paris, September 1999. ACM.

[28] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[29] Christopher League, Valery Trifonov, and Zhong Shao. Type-preserving compilation of Featherweight Java. In *Proc. Int'l Workshop Found. Object-Oriented Languages*, London, January 2001.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[31] Microsoft Corp., *et al.* Common language infrastructure. Drafts of the ECMA TC39/TG3 standardization process. `http://msdn.microsoft.com/net/ecma/`.

[32] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[33] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[34] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Programming Languages and Systems*, 21(3):528–569, May 1999.

[35] NaturalBridge. Personal communication with Kenneth Zadeck and David Chase, August 2001.

[36] George C. Necula. Proof-carrying code. In *Proc. Symp. on Principles of Programming Languages*, pages 106–119, New York, January 1997. ACM.

[37] George C. Necula. Personal communication, September and October 2001.

[38] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. Conf. on Programming Language Design and Implementation*, pages 333–344, Montréal, June 1998. ACM.

[39] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: a portable assembly language that supports garbage collection. In Gopalan Nadathur, editor, *Proc. Conf. on Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 1–28. Springer, 1999.

[40] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proc. Conf. on Programming Language Design and Implementation*, pages 285–298. ACM, 2000.

[41] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA, 1993.

[42] Carsten Schürmann, Dachuan Yu, and Zhaozhong Ni. An encoding of F-omega in LF. In *Proc. Workshop on Mechanized Reasoning about Languages with Variable Binding*, Siena, June 2001.

[43] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. Conf. on Programming Language Design and Implementation*, pages 116–129. ACM, June 1995.

[44] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proc. Int'l Conf. Functional Programming*, pages 313–323, Baltimore, September 1998. ACM.

[45] James M. Stichnoth, Guei-Yuan Lueh, and Michał Cierniak. Support for garbage collection at every instruction in a Java compiler. In *Proc. Conf. on Programming Language Design and Implementation*, pages 118–127, Atlanta, May 1999. ACM.

[46] Don Syme. ILX: extending the .NET Common IL for functional language interoperability. In *Proc. BABEL Workshop on Multi-Language Infrastructure and Interoperability*. ACM, 2001.

[47] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. Conf. on Programming Language Design and Implementation*, New York, 1996. ACM.

[48] Robert Tolksdorf. Programming languages for the JVM. `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[49] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. Conf. on Lisp and Funct. Prog.*, pages 1–11. ACM, 1994.