Precision in Practice: A Type-Preserving Java Compiler*

Christopher League[1], Zhong Shao[2], and Valery Trifonov[2]

[1] Long Island University · Computer Science
1 University Plaza, Brooklyn, NY 11201
christopher.league@liu.edu

[2] Yale University · Computer Science
P.O. Box 208285, New Haven, CT 06520
flint@cs.yale.edu

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 ML. 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) [1] 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 Intermediate Language (CIL) of the Microsoft .NET platform [2] includes type information and defines verification conditions for many of its instructions.

As a general distribution format, JVM class files are very high-level and quite partial to the Java language. The byte-code language (JVML) includes no facil-

* 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.

ities for specifying data layouts or expressing the results of standard optimizations. Compiling other languages for the JVM means making foreign constructs look and act like Java classes or objects. That so many translations exist [3] 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 [4] 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 or Haskell's constructor classes.

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 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) [5] and Proof-Carrying Code (PCC) [6].

Many compilers—including Marmot [7], Intel's VM [8], and NaturalBridge BulletTrain [9]—preserve *some* kind of type information in their intermediate code, but none are rigorous enough to support verification. Lower-level code requires more sophisticated type systems. As we will demonstrate, 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.

Our previous work [10, 11] 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 [12] 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. 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 [13] 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 section 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, fixed point) and rows [14] for abstracting over structure suffixes. The instruction stream includes explicit type operations that guide the verifier.
- Unlike the CIL extension [4], 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 section 4.
- Nothing about our instruction set should surprise a typical compiler developer. 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, we print the type annotations using short, intuitive names such as `InstOf[java/lang/Object]`.
- 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 [15]. We postpone discussion of other related projects to section 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. This is one reason why virtual method calls are *atomic* operations in both JVML and CIL. How to enforce the self-application invariant in lower-level code is not widely understood.

Cedilla Systems developed *Special J* [15], a proof-carrying code compiler for Java. Their paper 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 discovered that their current system indeed does *not* properly enforce the necessary invariant on self-application [16]. 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 consumer's proof checker will *accept* the malicious code given above.

Necula claims that this hole can be patched [16], but it has still not been addressed in subsequent work [17]. One weakness in the Cedilla PCC architecture 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 have different levels of granularity. 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.

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 [18]. 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. The difference is that λJVM replaces the implicit operand stack and untyped local variables with explicit data flow and fully-typed single-assignment bindings. 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 [19, 20]. Although we have not implemented them, this phase would be suitable for class hierarchy analysis and various object-aware optimizations [21] because the class hierarchy and method invocations are still explicit.

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 [22].

On JFlint, we run several contraction optimizations (inlining, common subexpression elimination, *etc.*), and type-check the code after each pass. Since method invocations are no longer atomic in JFlint, these optimizations readily lift and merge vtable accesses. A future version of the JFlint type system will even have support for optimizing array bounds checks [23].

We discard the type information before converting to MLRISC [24] 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.* [5] should apply directly, since JFlint is based on System F.

Figure 1 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.

```
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
```

Fig. 1. Compiling and running a Java program in SML/NJ.

The session in figure 1 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 wrongly 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 should be possible. Benton and Kennedy [25] 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. The SML runtime system does not yet support kernel threads, so we have ignored concurrency and synchronization.

Finally, 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 dots at left indicate erasable type operations. 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 fixed point (recursive) 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 -> InstOf[java/lang/String];
           hashCode : sj -> int;
           mj(sj) }
```

`UNFOLD` eliminates the fixed point by replacing occurrences of the bound variable `s0` with the recursive 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 -> InstOf[java/lang/String]`. 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

```
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
  | CALL   of id * value list
  | RETURN of value
  | 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
```

Fig. 2. Representation of JFlint code.

m2, say); these never match the variables in the type of r2. The precise typing rules for UNFOLD and OPEN are available elsewhere [11, 26].

After the final verification, the type operations 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. 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 2 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 very lightweight, more akin to basic blocks than to functions in C.

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 arguments and transfers control all at once; the calling convention is not explicit. It is possible to break these down and still preserve verifiability [5], but this midpoint is simpler and still quite useful for optimization.

There are two hurdles for a conventional compiler developer 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 [19]. 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 [10, 11]; some of these appear in figure 3. 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 3).

Previously, we showed the expansion of `InstOf[C]` as an existential type. Suppose `D` extends `C`; then, `InstOf[D]` is 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 [27] with 'top' subtyping [28].) 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 3 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 is easily recognized and eliminated during code generation.

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 is expensive, so it pays to cache the results. With the addition of unordered (permutable) record types and a trusted primitive for the dynamic

```
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]?;

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);
```

Fig. 3. Recipes for some λJVM primitives.

search, interface types pose no further problems. Verifying the searching and caching code in a static type system would be quite complex. As an experiment, we implemented a unique representation of interfaces for which the dynamic search is unnecessary [10].

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. Since all interface relationships are declared in Java, the itables can be created when each class is compiled, and then linked into the class vtable. Since the layout of the vtable is known at the point of upcast, dynamic search is unnecessary.

The final recipe in figure 3 illustrates 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

```
signature JTYPE = sig
 type ty
 val var    : int * int -> ty                          (* type variable *)
 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 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
```

Fig. 4. Abstract interface for JFlint type representation.

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.

This scheme also supports multiple inheritance of interfaces. Suppose interface `AB` extends both interfaces `A` (with method `a`) and `B` (with method `b`). The itable of `AB` will contain pointers to itables for each of the super interfaces: To upcast from `AB` to `B`, just open the interface object, fetch `itbl.B`, pair it with `obj`, and re-package.

Unfortunately, Java's covariant subtyping of arrays (widely considered to be a misfeature) is not directly compatible with this interface representation. Imagine casting an array of class type to an array of interface type—we would need to coerce each element! For the purpose of experimentation, we ignored the covariant array subtyping rule. In the future, we would like to find a hybrid approach that allows cheap, easily verifiable invocation of interface methods, but is still compatible with the Java specification.

4.4 Type representation

To support efficient compilation, types are represented differently from code. Figure 4 contains part of the abstract interface to our type system. Most of our types are standard: based on the higher-order polymorphic lambda calculus (see [29] 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 [30] to hide the types of additional fields and methods; these are rows.

A 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 [14].

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 [31].

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 [2, 4]). 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.

JFlint demonstrates a better approach: 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--` [32], 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

If a type-preserving compiler is to scale, types and type operations must be implemented with extreme care. The techniques of Shao, et al. made the FLINT typed IL practical enough to use in a production compiler [18]. Although different type structures arise in our Java encodings, the techniques are quite successful. 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 [33]. Since `gcj` is written in C and our compiler in SML, this performance gap can easily be attributed to linguistic differences. Verifying both the λJVM and the JFlint code adds another half second.

Run times are promising, but 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.

6 Related work

Throughout the paper, we made comparisons to the Common Intermediate Language (CIL) of the Microsoft .NET platform [2] and ILX, a proposed extension for functional language interoperability [4]. We discussed the proof-carrying code system Special J [15] at length in section 2. We mentioned `C--` [32], 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 [34]; we mentioned their extensions for interoperability earlier [25]. 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.

Wright, et al. [35] compile a Java subset to a typed intermediate language, but they use unordered records and resort to dynamic type checks because their system is too weak to type self application. Neal Glew [36] translates a simple class-based object calculus into an intermediate language with F-bounded polymorphism [27] and a special 'self' quantifier. A more detailed comparison with this encoding is available elsewhere [11, 26].

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 [7, 37]. Gagnon et al. [38] 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. [39] 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 machine 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 preclude optimizations or efficient execution. We therefore believe that precise type preservation is within reach for real-world Java systems.

References

[1] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. 2nd edn. Addison-Wesley (1999)

[2] ECMA: Common language infrastructure. Drafts of the TC39/TG3 standardization process. `http://msdn.microsoft.com/net/ecma/` (2001)

[3] Tolksdorf, R.: Programming languages for the JVM. `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html` (2002)

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

[5] Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to typed assembly language. ACM Trans. on Programming Languages and Systems **21** (1999)

[6] Necula, G.C.: Proof-carrying code. In: Proc. Symp. on Principles of Programming Languages, Paris, ACM (1997) 106–119

[7] Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B., Tarditi, D.: Marmot: an optimizing compiler for Java. Software: Practice and Experience **30** (2000)

[8] Stichnoth, J.M., Lueh, G.Y., Cierniak, M.: Support for garbage collection at every instruction in a Java compiler. In: Proc. Conf. on Programming Language Design and Implementation, Atlanta, ACM (1999) 118–127

[9] NaturalBridge: Personal comm. with Kenneth Zadeck and David Chase (2001)

[10] League, C., Shao, Z., Trifonov, V.: Representing Java classes in a typed intermediate language. In: Proc. Int'l Conf. Functional Programming, Paris, ACM (1999)

[11] League, C., Shao, Z., Trifonov, V.: Type-preserving compilation of Featherweight Java. ACM Trans. on Programming Languages and Systems **24** (2002)

[12] Bruce, K.B., Cardelli, L., Pierce, B.C.: Comparing object encodings. Information and Computation **155** (1999) 108–133

[13] Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). MIT Press (1997)

[14] Rémy, D.: Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA (1993)

[15] Colby, C., Lee, P., Necula, G.C., Blau, F., Cline, K., Plesko, M.: A certifying compiler for Java. In: Proc. Conf. on Programming Language Design and Implementation, Vancouver, ACM (2000)

[16] Necula, G.C.: Personal communication (2001)

[17] Schneck, R.R., Necula, G.C.: A gradual approach to a more trustworthy, yet scalable, proof-carrying code. In: Proc. Conf. on Automated Deduction. (2002)

[18] Shao, Z., League, C., Monnier, S.: Implementing typed intermediate languages. In: Proc. Int'l Conf. Functional Programming, Baltimore, ACM (1998) 313–323
[19] Appel, A.W.: SSA is functional programming. *ACM SIGPLAN* Notices (1998)
[20] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. on Programming Languages and Systems **13** (1991) 451–490
[21] Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Proc. European Conf. Object-Oriented Programming. (1995)
[22] League, C., Trifonov, V., Shao, Z.: Functional Java bytecode. In: Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics. (2001) Workshop on Intermediate Representation Engineering for the Java Virtual Machine.
[23] Shao, Z., Saha, B., Trifonov, V., Papaspyrou, N.: A type system for certified binaries. In: Proc. Symp. on Principles of Programming Languages. (2002)
[24] George, L.: Customizable and reusable code generators. Technical report, Bell Labs (1997)
[25] Benton, N., Kennedy, A.: Interlanguage working without tears: Blending ML with Java. In: Proc. Int'l Conf. Functional Programming, Paris, ACM (1999) 126–137
[26] League, C.: A Type-Preserving Compiler Infrastructure. PhD thesis, Yale University (2002)
[27] Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proc. Int'l Conf. on Functional Programming and Computer Architecture, ACM (1989) 273–280
[28] Castagna, G., Pierce, B.C.: Decidable bounded quantification. In: Proc. Symp. on Principles of Programming Languages, Portland, ACM (1994)
[29] Barendregt, H.: Typed lambda calculi. In Abramsky, S., Gabbay, D., Maibaum, T., eds.: Handbook of Logic in Computer Science. Volume 2. Oxford (1992)
[30] Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. ACM Transactions on Programming Languages and Systems **10** (1988) 470–502
[31] Schürmann, C., Yu, D., Ni, Z.: An encoding of F-omega in LF. In: Proc. Workshop on Mechanized Reasoning about Languages with Variable Binding, Siena (2001)
[32] Peyton Jones, S., Ramsey, N., Reig, F.: C—: a portable assembly language that supports garbage collection. In Nadathur, G., ed.: Proc. Conf. on Principles and Practice of Declarative Programming. Springer (1999) 1–28
[33] Bothner, P.: A GCC-based Java implementation. In: Proc. IEEE Compcon. (1997)
[34] Benton, N., Kennedy, A., Russell, G.: Compiling Standard ML to Java bytecodes. In: Proc. Int'l Conf. Functional Programming, Baltimore, ACM (1998) 129–140
[35] Wright, A., Jagannathan, S., Ungureanu, C., Hertzmann, A.: Compiling Java to a typed lambda-calculus: A preliminary report. In: Proc. Int'l Workshop on Types in Compilation. Volume 1473 of LNCS., Berlin, Springer (1998) 1–14
[36] Glew, N.: An efficient class and object encoding. In: Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, ACM (2000)
[37] Knoblock, T., Rehof, J.: Type elaboration and subtype completion for Java bytecode. In: Proc. Symp. on Principles of Programming Languages. (2000) 228–242
[38] Gagnon, E., Hendren, L., Marceau, G.: Efficient inference of static types for Java bytecode. In: Proc. Static Analysis Symp. (2000)
[39] Amme, W., Dalton, N., von Ronne, J., Franz, M.: 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)