ILEA: Inter-Language Analysis across Java and C *

Gang Tan
Boston College
gtan@cs.bc.edu

Greg Morrisett
Harvard University
greg@eecs.harvard.edu

Abstract

Java bug finders perform static analysis to find implementation mistakes that can lead to exploits and failures; Java compilers perform static analysis for optimization. If Java programs contain foreign function calls to C libraries, however, static analysis is forced to make either optimistic or pessimistic assumptions about the foreign function calls, since models of the C libraries are typically not available.

We propose ILEA (stands for Inter-LanguagE Analysis), which is a framework that enables existing Java analyses to understand the behavior of C code. Our framework includes: (1) a novel specification language, which extends the Java Virtual Machine Language (JVML) with a few primitives that approximate the effects that the C code might have; (2) an automatic specification extractor, which builds models of the C code. Comparing to other possible specification languages, our language is expressive, yet facilitates construction of automatic specification extractors. Furthermore, because the specification language is based on the JVML, existing Java analyses can be easily migrated to utilize specifications in the language. We also demonstrate the utility of the specifications generated, by modifying an existing non-null analysis to identify null-related bugs in Java applications that contain C libraries. Our preliminary experiments identified dozens of null-related bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logic and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Reliability, Verification

* This work was supported in part by AFOSR Grant F49620-01-1-0298. Any opinions, findings, and conclusions or recommendations in this publication are those of the authors and do not reflect the views of this agency.

Keywords Inter-language analysis, Java Native Interface, JNI, JVML, Specification extraction

1. Introduction

Compilers have long used static analysis to determine when it is safe to perform optimization. Today, when security and reliability are as much of a concern as performance, many tools (c.f., [11, 34, 19, 21, 5]) rely on static analysis to find implementation errors that can lead to exploits and failures. In both settings, the precision of a given analysis is limited by a combination of the approximations used for modeling code and data, as well as the *horizon* of the analysis—the size of the program unit that is analyzed (i.e., whole program, module, class, method, etc.)

When looking for bugs in security-critical code, high precision is needed to ensure all potential flaws are found and that false positives are minimized. Similarly, in contexts where high performance is desired, high precision enables advanced optimizations. In both settings, extending the horizon of an analysis can have a significant impact. For example, recent studies show that inter-procedural analysis can result in significant performance gains over intra-procedural techniques, even in the context of just-in-time Java compilers [22, 31, 33, 32].

Unfortunately, the horizon of most analyses is limited *a priori* to code written in a single language, and yet most real applications consist of a mixture of code written in different languages. For example, many real Java applications utilize the Java Native Interface (JNI [24]) to access C libraries. Indeed, almost any useful Java application mixes Java and native C code since many classes provided by a Java platform are in fact just wrappers for underlying native C code. For instance, the often-used java.io.FileInputStream class in Sun's JDK uses the JNI to invoke operating-system functions to perform file operations. Another example is the classes under java.util.zip, which are just wrappers that invoke the popular Zlib C library. For functionality and convenience, any Java platform contains a significant amount of native C code. For example, Sun's JDK 1.5.0 contains over 700,000 lines of native C/C++ code.[1]

[1] We measured the lines of C/C++ code under the directories j2se/src/share/native, j2se/src/solaris/native, and j2se/src/windows/native in Sun's JDK 1.5.0. Around 80,000 lines are in C++, while the rest are in C.

Since native C code in Java applications is plentiful, one may wonder how existing Java analyses treat a native-method call from Java. Typically, they make one of two choices: Either they make an optimistic assumption, which essentially assumes the native method call is a nop, or they make a pessimistic assumption, which assumes anything can happen in the native method. Bug finders tend to make the former choice, to avoid too many false positives (which also means that they may miss some real bugs). Java optimizers tend to make the latter choice, as they need to guarantee the correctness of the optimizations that they perform.

This paper asks the question "how can we extend the horizon of existing Java analyses to cover native C code?". A solution to this problem enables a Java bug-finder to search for bugs across Java and C, and enables a Java optimizer to catch more optimization opportunities.

The basic approach that we advocate is to use a separate tool to construct a *specification* (i.e., model) of the C code, and then extend the analysis so that it understands the specification. Given this general approach, two questions remain: "What specification language would be the best for all analyses?" and "How do we generate specifications for native C methods?"

1.1 Possible specification languages for native C code

The native C code itself, is in some sense, the most accurate specification in that it loses no information. However, the actual code is a poor "model" for two reasons: First, to use the model, we must modify each Java analysis so that it understands C code, which isn't portable and often demands a complete reworking of abstract domains. Second, many parts of the C code may not be relevant to a given Java analysis. As an example, the fdlibm library (a math library in C) used by the classes in java.lang.StrictMath has over 8,500 lines of C code. Almost all the code in fdlibm perform pure computation and makes no modification to the Java heap. Thus, the vast majority of the 8,500 lines of code is irrelevant for e.g., a points-to analysis.

Another possible specification language is some annotation language, designed for the particular property that a Java analysis is interested in. For example, one could use the immutability specifications advocated by Pechtchanski and Sarkar [32] to annotate side-effect-free native methods. If a Java bug finder intends to search for bugs of null dereferences across native code, one could use the nonnull specifications (as in [7]) to annotate the return values and parameters of native methods. As a last example, Guyer and Lin's annotation language [17] could be used for specifying data flow properties. However, this approach is also problematic. As with C code, we must teach existing analyses to understand the syntax and semantics of the annotations. Furthermore, when a new property of interest comes along, we must invent a new kind of annotation. In short, any simple set of annotations will be incomplete as a specification language.

To achieve completeness, we could specify the behavior of C code using pre- and post-conditions in the style of Hoare logic, and as advocated by both Spec# [1] and ESC/Java [11]. Here, we effectively model the C code as a *relation* that captures connections between the parameters, the return value, and the initial and final Java heaps, but that abstracts over the actual steps of the C code, and the state outside the Java abstract machine. The approach is more flexible than simple annotations because we can provide a range of specifications, from simple summaries to a (relatively) complete characterization of the effect of the C code on the Java heap. For example, we can summarize the effect of *any* C procedure by using the everywhere-defined relation (which effectively tells an analysis nothing), or if we have a powerful logic, we can capture a precise specification of the C code, at least with respect to the state of the JVM. The key problem with logic-based summaries, is that it may require significant effort to change an existing Java analysis to understand and utilize the specifications, as they are in a different language from Java or C.

The approach that we advocate is to use a slightly extended Java Virtual Machine Language (JVML) to model the C code. The key advantage is that existing analyses already understand the syntax and semantics of the JVML language, and our proposed extensions are relatively modest. Furthermore, if the native C code can be faithfully compiled into the JVML, then we can build a completely accurate model. Of course, we cannot hope to compile all C code, but as we argue below, our extensions make it possible to build useful, conservative models of the behavior of the C code, at least with respect to the inputs, outputs, and the changes to the Java heap. In effect, our extended JVML code is an alternative representation of a relational specification and thus enjoys the expressiveness and portability of the logic-based approach.

2. Extended JVML

Our proposed specification language extends the JVML with a few extra primitives for approximating the behavior of C code that cannot be faithfully compiled into Java. For instance, suppose we wish to model a C function that invokes a system call such as `getpid`. We could approximate the behavior of this call by using JVML code that returns a random number. Of course, there are no JVML instructions that will truly give us back a random number, so we add a new operation, choose τ, which returns a random τ value.

Now to model a higher-level system call, such as gettimeofday(), which might allocate a Java object of type Time { int sec; int msec; }, we could use JVML code that does effectively:

```
new Time(choose(int), choose(int))
```

We also add an operation mutate(x:object), which has the effect of causing some (type-preserving) mutation to the object referenced by x. So to model a C function that might mutate any object in the heap, we could do:

```
Object x = choose(Object);
mutate(x);
```

The existing JVML instructions, together with choose and mutate, can specify a variety of properties. We present a few examples.

1) The program can mutate existing objects of any type but cannot allocate:

```
for (int i = 0; i < choose(int); i++) {
   Object x = choose(Object);
   mutate(x);
}
```

2) The program can allocate arbitrary number of objects of type T:

```
for (int i = 0; i<choose(int); i++) {
  new(T);
}
```

3) The program can access any object of type T, but does not modify anything, nor does it access objects of incompatible types:

```
for (int i = 0; i<choose(int); i++) {
  T x = choose T;
  boolean b = x.equals(x);
}
```

We have presented these examples using Java source syntax, for its simplicity and clarity. Nevertheless, we base our specification language on the JVML, for the expressiveness of the JVML. Furthermore, users can still write specifications in Java source syntax, and a separate tool can convert them into JVML specifications.

To make our specification language precise, we have extended the syntax and semantics of the JVML$_f$ model, formalized by Freund and Mitchell [13]. A summary of the syntax for our extended-JVML model is given in Figure 1. The change[2] over the syntax of JVML$_f$ is the addition of the following instructions:

- choose τ: choose a random object of type τ and place it on the operand stack.
- mutate: mutate the object (or array) whose reference is on the top of the operand stack.
- top: may have any type-preserving effect on the JVM heap (including mutation and allocation).

[2] We also add a label instruction, and use abstract labels instead of offsets in jump instructions. We then use the Jasmin [26] assembler to compute the offsets of labels; see later discussion.

instruction	I	$::=$	choose τ | mutate | top
		|	label l | ifeq l | goto l
		|	new σ | nop | push v | pop
		|	store d | load d | add
		|	newarray τ | arraylength
		|	arrayload τ | arraystore τ
		|	return | returnval
		|	...
JVML type	τ	$::=$	int | void | Array τ | σ | ...
class name	σ	$::=$	java/lang/Object
		|	java/lang/String | ...
label	l		
var id	d		

Figure 1. Extended JVML syntax (An excerpt)

This set of new primitives is designed considering the possible effects that a C function may have on the JVM heap. Note that we do not have a separate primitive for allocating objects, since the JVML instruction "new σ" already covers the job of allocation. From the JVM's perspective, the changes to the JVM heap is the most relevant effect that C code might have. For analyses such as dead-code elimination, other effects such as the I/O effect of the C code may also be relevant. More primitives can be added to accommodate those, but we restrict ourselves to the JVM-heap effect in this paper.

We have a top primitive in the language, for a couple of reasons. First, it serves as a safe exit for cases that are difficult or impossible to specify accurately. It provides a coarse, yet safe, specification. Second, although we focus on the JVM-heap effect in this paper, the semantics of top can be easily changed to have any effect, such as the I/O effect.

Existing JVML instructions are partial functions; their semantics deterministically maps a JVM state to the next JVM state. The new primitives, on the other hand, are non-deterministic, and thus denote relations between JVM states. For example, mutate(x) relates a JVM state to any new state that is the result of performing (type-preserving) mutations to the object. To specify a C function, we use existing JVML instructions for characterizing the function's deterministic part, and leave its non-deterministic part for the new primitives.

The idea of using the JVML plus a few relational primitives as the specification language has many benefits:

- This specification language is *expressive*: It can model the input/output effects of *well-behaved* C code. Intuitively, well-behaved C code respects well-typedness of Java heaps; we formalize the notion of well-behaved C code later. We regard expressiveness a considerable advantage over other specification languages.

- Any existing analysis on the JVML will be easily carried over to our specification language. The additional work is to handle the new primitives, which is usually an easy augmentation. For example, in an intra-procedural analysis, the treatment of top should be no different from the treatment of procedural calls. The smooth transition for existing analyses is demonstrated by our experience with modifying an existing nonnull analysis, which will be detailed in Section 4.
- The new primitives make it possible to automatically extract a range of models for C code. In particular, an extractor always has safe exits through the new primitives. In Section 3, we describe one specification extractor, which takes full advantage of the new primitives; for example, the extractor does not model C memory, so it always return "choose int" for any integer from the C memory.
- It also supports by-hand construction of models. Programmers can easily specify coarse models of C libraries by writing small snippets of (extended) Java code which are then compiled to extended JVML specifications. (Of course, verifying that the model is correct would be generally impossible.)

2.1 Models of our extended JVML language

We have informally discussed the meanings of the new JVML instructions. In this section, we make their semantics precise, by presenting their formal definitions.

We base our definitions on Freund and Mitchell's JVML model. Relevant concepts from their model are presented in Figure 2. As a summary, the dynamic semantics of JVML$_f$ is captured by the judgment $\Gamma \vdash C_0 \rightarrow C_1$, meaning that a program represented by Γ moves from configuration C_0 to C_1. A JVML$_f$ machine configuration C is a pair $A; h$, where A is a stack of activation records and h is a JVM heap. Both A and h are detailed in Figure 2; we only mention that all objects and arrays in the heap h contain runtime type tags, and Tag(h, a) returns the type tag for the object at location a in h.

We extend the semantics of JVML$_f$ for our new instructions in Figure 3. Each of the instructions has a relatively straightforward definition.

Having a model for our extended JVML, we next formalize the assumption of well-behaved C code. C code has its own world, including its memory heap, I/O behaviors, and so on. Therefore, when considering the JVM and C together, a machine state consists of a JVML configuration $A; h$ and a C world w. We write a complete machine state as $(A; h; w)$.

[3] JVML$_f$ has additional components for exception handling and object initialization; we omit them in our formalism.

$\Gamma \vdash C_0 \rightarrow C_1$	A program represented by Γ moves from configuration C_0 to C_1.
Γ	A global environment; the representation of a JVML$_f$ program.
$C = A; h$	A JVML$_f$ machine configuration.
A	A stack of activation records.
$\langle M, pc, f, stk \rangle$	One activation record[3]: M : the method reference of the current activation record. pc : the address of the next instruction to be executed. f : a map from the set of local-variable indexes to values. stk : the operand stack.
h	A JVML$_f$ heap, mapping locations to objects or arrays.
$\langle\!\langle \{\!\mid \sigma_i, l_i, \kappa_i \mid\!\} = v_i \rangle\!\rangle_{\sigma}^{i \in I}$	An object, mapping field references to values; $i \in I$ refers to the i-th field in the object.
$\{\!\mid \sigma, l, \kappa \mid\!\}$	A field reference, with class name σ, label l, and type κ.
$[\![v_i]\!]_{\mathbf{Array}\ \tau}^{i \in [0,n)}$	An array, with length n.
Tag(h, a)	The runtime type tag of a heap object or array at location a.

Judgments:

$\Gamma \vdash h$ wt	h is a well-typed heap.
$\Gamma, h \vdash v : \tau$	The value v has type τ given environment Γ and heap h.

Notations:

$\langle M, pc, f, stk \rangle \cdot A$	A stack of activation records, whose top is $\langle M, pc, f, stk \rangle$.
$v \cdot stk$	An operand stack, whose top is v.

Figure 2. Summary of the dynamic semantics of the JVML$_f$

$$\begin{array}{c} \Gamma \vdash M(pc) = \mathsf{choose}\ \tau \\ \Gamma, h \vdash v : \tau \\ \hline \Gamma \vdash \langle M, pc, f, stk\rangle \cdot A; h \to \langle M, pc+1, f, v \cdot stk\rangle \cdot A; h \end{array} \qquad \begin{array}{l} \textit{the current instruction is } \mathsf{choose}\ \tau \\ v \textit{ has type } \tau \\ \\ \end{array}$$

$$\begin{array}{c} \Gamma \vdash M(pc) = \mathsf{mutate} \\ h[b] = [\![v_i]\!]^{i \in [0,n)}_{\mathbf{Array}\ \tau} \\ o = [\![v'_i]\!]^{i \in [0,n)}_{\mathbf{Array}\ \tau}, \text{ and } \forall i \in [0,n).\ \Gamma, h \vdash v'_i : \tau \\ \hline \Gamma \vdash \langle M, pc, f, b \cdot stk\rangle \cdot A; h \to \langle M, pc+1, f, stk\rangle \cdot A; h[b \mapsto o] \end{array} \qquad \begin{array}{l} \textit{the current instruction is } \mathsf{mutate} \\ \textit{array of element type } \tau \textit{ at location } b \\ \textit{mutate the array to get } o \\ \\ \end{array}$$

$$\begin{array}{c} \Gamma \vdash M(pc) = \mathsf{mutate} \\ h[b] = \langle\!\langle \{\!|\sigma_i, l_i, \kappa_i|\!\} = v_i \rangle\!\rangle^{i \in I}_{\sigma} \\ o = \langle\!\langle \{\!|\sigma_i, l_i, \kappa_i|\!\} = v'_i \rangle\!\rangle^{i \in I}_{\sigma}, \text{ and } \forall i \in I.\ \Gamma, h \vdash v'_i : \kappa_i \\ \hline \Gamma \vdash \langle M, pc, f, b \cdot stk\rangle \cdot A; h \to \langle M, pc+1, f, stk\rangle \cdot A; h[b \mapsto o] \end{array} \qquad \begin{array}{l} \textit{the current instruction is } \mathsf{mutate} \\ \textit{object of type } \sigma \textit{ at location } b \\ \textit{mutate the object to get } o \\ \\ \end{array}$$

$$\begin{array}{c} \Gamma \vdash M(pc) = \mathsf{top} \\ \Gamma \vdash h \sqsubseteq h' \\ \hline \Gamma \vdash \langle M, pc, f, stk\rangle \cdot A; h \to \langle M, pc+1, f, stk\rangle \cdot A; h' \end{array} \qquad \begin{array}{l} \textit{the current instruction is } \mathsf{top} \\ \textit{type-preserving heap extension} \\ \\ \end{array}$$

Figure 3. Models of choose τ, mutate, and top.

DEFINITON 1. *(Rule for type-preserving heap extensions.)*

$$\frac{\Gamma \vdash h \text{ wt} \quad \Gamma \vdash h' \text{ wt} \\ \forall a \in \mathrm{dom}(h).\ a \in \mathrm{dom}(h') \wedge \mathrm{Tag}(h,a) = \mathrm{Tag}(h',a)}{\Gamma \vdash h \sqsubseteq h'}$$

DEFINITON 2. *(Well-behaved C code.) Given an initial machine state* $(A; h; w)$ *such that* $\Gamma \vdash h$ wt*, C code is well behaved if for any result of its execution, say* $(A'; h'; w')$*, we have* $A' = A$ *and* $\Gamma \vdash h \sqsubseteq h'$.

In words, we say C code is well behaved if it does not change the activation records of the JVM, and performs only type-preserving modifications to the Java heap, including allocation. (It can perform arbitrary changes to its own world.) We believe the well-behaved assumption of the C code is a reasonable one. On the other hand, it is possible to intentionally or unintentionally write ill-behaved C code, e.g., code that writes random integers to random memory addresses. In this case, we rely on our previous work, the SafeJNI [35] system, to prevent ill-behaved C code from destroying the JVM state. For instance, the SafeJNI system inserts dynamic checks to the C code so that the code is stopped if it directly writes to the memory addresses that belong to the JVM.

Our language is *expressive* in the sense that we can use a combination of the primitives to conservatively simulate the Java-heap effects of any well-behaved C program. In particular, a loop similar to the one given at the beginning of this section can be written that relates a heap to all of its well-typed mutations. When followed by a loop to allocate objects of the correct types (passing the constructors values obtained via choose), we get a program that relates a heap to any of its well-behaved relatives.

Ideally, our specification language would satisfy a (relative) *completeness* property in the sense that for any well-behaved C program, we could write an extended-JVML program that relates *only* those heaps related by the C program. Unfortunately, this property does not hold. For example, a C program could use pointer arithmetic to gain access to a particular object in the Java heap and mutate it, but the only way we can model this is through the use of "choose". While this is sound, it is not precise as we may end up choosing other objects.

In general, the only way to establish completeness would be to either add more instructions to the JVML, or to further restrict our definition of "well-behaved" C code to the point where we can always effectively compile its actions on the Java heap into deterministic JVML code. We resisted adding more instructions because this would make it harder to integrate with an analysis. We resisted strengthening the definition of "well-behaved" to admit more C code into the framework.

3. Automatic specification extraction

We have written an automatic specification extractor, which takes C code as input, and outputs its extended-JVML specification. Our specification extractor is implemented in the CIL framework [29] as a CIL feature. Before our extractor is invoked, the CIL front end converts C to the CIL intermediate language. The conversion compiles away many complexities of C, thus allowing our extractor to concentrate on a relatively clean subset of C.

Our specification extractor can deal with nearly all C features, thanks to the extra instructions; in the worst case, the top specification is issued. The extractor makes the compilation of common and easy cases precise, while leaving rare or difficult cases for approximation. One of the most difficult issues of analyzing C is how to deal with pointer aliases; our extractor depends on a points-to analysis provided by the CIL. We would like to stress that our specification extractor is not the only possible one. Our specification language allows the construction of a range of specification extractors, each of which can make its own trade-off between precision and approximation.

To present the main ideas behind our specification extractor, we have formalized the extraction process for a subset of C language, including pointers, assignments, conditional statements, and a representative set of JNI API functions. For simplicity, this subset omits loops, function calls, struct/union types, and others. We will discuss how our implementation treats these features in Section 3.2.

C syntax. The syntax for a subset of the C language representing the CIL intermediate language is given in Figure 4. There are CIL expressions, which have no side effects and no control flow, and CIL statements, which may have side effects or control flow. In an assignment statement "$lv = \delta$", the l-value lv refers to a region of storage; it is either a variable x, or $*e$, which denotes the memory slot pointed to by e. The right hand of the assignment (or δ) can be either an expression, or a JNI API call[4].

$$
\begin{array}{llcl}
\text{l-value} & lv & ::= & x \mid *e \\
\text{expression} & e & ::= & n \mid e_1 + e_2 \mid lv \\
\text{JNI API} & japi & ::= & \mathrm{GetArrayLength}(e) \\
 & & \mid & \mathrm{GetIntArrayElements}(e) \\
 & & \mid & \mathrm{ReleaseIntArrayElements}(e, e) \\
 & & \mid & \mathrm{GetObjectArrayElement}(e, e) \\
 & & \mid & \mathrm{SetObjectArrayElement}(e, e, e) \\
 & & \mid & \mathrm{NewIntArray}(e) \\
 & & \mid & \mathrm{NewObjectArray}(e, e, e) \\
\text{exp or JAPI} & \delta & ::= & e \mid japi \\
\text{statement} & s & ::= & s_1; s_2 \mid \mathsf{if}\ (e)\ s_1\ s_2 \\
 & & \mid & lv = \delta \mid japi \\
 & & \mid & \mathsf{return}(e) \mid \mathsf{return} \\
\text{C type} & t & ::= & \mathsf{int} \mid \mathsf{void} \mid t\,* \\
 & & \mid & \mathsf{jint}\,* \mid \mathsf{jobject} \\
\text{function} & fn & ::= & t\ \textit{Fun-Name}(t_1\ x_1,\ \ldots, t_n\ x_n) \\
 & & & \quad \{\ t'_1\ y_1;\ \ldots; t'_m\ y_m; \\
 & & & \quad\quad s; \}
\end{array}
$$

Figure 4. Our C Syntax

Our formalism includes a subset of JNI API functions, which are related to array operations. The meaning of these functions is explained in Table 1. Notice that the JNI treats primitive-type arrays and object arrays differently. Primitive-type arrays are accessed through direct pointers (returned by functions such as $\mathrm{GetIntArrayElements}$); object arrays are accessed on a per-element basis.

We use the symbol t for a C type, to distinguish it from a JVML type τ. The type "$\mathsf{jint}\ *$" is a pointer to an array of integers in the JVM; it is the return type of $\mathrm{GetIntArrayElements}$. The type $\mathsf{jobject}$ is the type for references to Java objects.

The CIL front end compiles away many complexities of C. For example, our syntax does not allow "if $(japi)\ s_1\ s_2$": the CIL front end creates a new local variable tmp, and transforms the statement to

$$
\begin{array}{l}
tmp = japi; \\
\mathsf{if}\ (tmp)\ s_1\ s_2;
\end{array}
$$

In a similar fashion, the CIL converts C expressions, which may have nondeterministic order of side effects[30], to CIL statements and side-effect-free CIL expressions. Effectively, the CIL linearizes the order of side effects present in the C expressions. In the following discussion, we will equate expressions with CIL expressions.

Notation. We use the notation $\vec{I}$ for a sequence of extended-JVML instructions, and similarly use $\vec{e}$ for a sequence of expressions. We use the notation $[\,]$ for the empty sequence, and use $\vec{I_1} @ \vec{I_2}$ to denote the concatenation of $\vec{I_1}$ and $\vec{I_2}$. Finally, since the specification extractor essentially performs a compilation from C to the extended-JVML, we will use the phrase "specification extraction" and "compilation" interchangeably throughout the discussion.

Compiling C functions. The rules given in Figure 5 define a compiler that maps a C function into a sequence of extended-JVML instructions $\vec{I}$. The compiler first constructs a compilation environment Φ. This construction is formalized using the definitions $\mathsf{mapTy}(-)$ and $\mathsf{mapVar}(-,-)$.

The definition $\mathsf{mapTy}(t)$ maps a C type t to a JVML type τ. This definition is partial; in general, it does not map C pointer types. The definition $\mathsf{mapVar}([t_1\ x_1; t_2\ x_2; \ldots], d)$ maps C variables to JVML local variables, starting from the index d. It maps only those variables whose types are mapped.

With the help of $\mathsf{mapTy}(-)$ and $\mathsf{mapVar}(-,-)$, we get a compilation environment Φ of the form

$$\{x_1 : (d_1, \tau_1), \ldots, x_n : (d_n, \tau_n)\}.$$

It maps C variables into its JVML local-variable indexes and types.[6] The body of the C function is then compiled under the environment Φ, and a return type τ_{ret}, which is mapped from the return type of the function.

[4] Instead of being invoked directly, JNI API functions are invoked through the function pointers stored in an interface table. We simplify this in the formalism.

[5] GetIntArrayElements in the JNI has an extra argument isCopy; we omit it in the formalism.

[6] Notice that in Figure 5, we always start from index 1, as index 0 is reserved for the self object.

JNI API Functions	**Description**
$\mathsf{int}\ \mathrm{GetArrayLength}(\mathsf{jobject}\ \mathit{array})$	Get the *array* length.
$\mathsf{jint}\ *\ \mathrm{GetIntArrayElements}(\mathsf{jobject}\ \mathit{array})$[5].	Return a pointer to the elements of the integer *array*, or null if the operation fails.
$\mathsf{void}\ \mathrm{ReleaseIntArrayElements}(\mathsf{jobject}\ \mathit{array}, \mathsf{jint}\ *\ \mathit{body})$	Release the *body* pointer to *array*.
$\mathsf{jobject}\ \mathrm{GetObjectArrayElement}(\mathsf{jobject}\ \mathit{array}, \mathsf{int}\ n)$	*array* is a reference to an array of Java objects; return the n-th element.
$\mathsf{void}\ \mathrm{SetObjectArrayElement}(\mathsf{jobject}\ \mathit{array}, \mathsf{int}\ n, \mathsf{jobject}\ \mathit{obj})$	Set the n-th element of *array* to *obj*.
$\mathsf{jobject}\ \mathrm{NewIntArray}(\mathsf{int}\ \mathit{len})$	Construct a new integer array, with length *len*.
$\mathsf{jobject}\ \mathrm{NewObjectArray}(\mathsf{int}\ \mathit{len}, \mathsf{jclass}\ t, \mathsf{jobject}\ \mathit{init})$	Construct a new object array, with length *len* and element type t; all elements are initially set to *init*.

Table 1. JNI API functions

$$\frac{\begin{array}{l}\Phi = \mathsf{mapVar}([t_1\ x_1; \ldots; t_n\ x_n; t'_1\ y_1; \ldots; t'_m\ y_m], 1)\\ \tau_{\text{ret}} = \begin{cases}\tau, & \text{if } \mathsf{mapTy}(t) = \tau\\ \mathsf{void}, & \text{otherwise}\end{cases}\\ \Phi, \tau_{\text{ret}} \vdash s \rightsquigarrow \vec{I}\end{array}}{\vdash \begin{pmatrix} t\ \mathit{Fun\text{-}Name}(t_1\ x_1,\ \ldots, t_n\ x_n)\\ \quad \{\, t'_1\ y_1;\ \ldots; t'_m\ y_m;\\ \qquad s; \} \end{pmatrix} \rightsquigarrow \vec{I}}$$

$$\begin{array}{l}\mathsf{mapVar}([t_1\ x_1; t_2\ x_2; \ldots], d)\\ \quad = \begin{cases}(\ \{x_1 : (d, \tau_1)\} \cup \mathsf{mapVar}([t_2\ x_2; \ldots], d+1)\ ), \\ \hspace{10em} \text{if } \mathsf{mapTy}(t_1) = \tau_1\\ \mathsf{mapVar}([t_2\ x_2; \ldots], d), \hspace{3em} \text{otherwise}\end{cases}\end{array}$$

$$\begin{array}{lcl}\mathsf{mapTy}(\mathsf{int}) & = & \mathsf{int}\\ \mathsf{mapTy}(\mathsf{jint}\ *) & = & \mathsf{Array\ int}\\ \mathsf{mapTy}(\mathsf{jobject}) & = & \mathsf{java/lang/Object}\end{array}$$

Figure 5. Rule to extract JVML specifications from C functions

Compiling expressions. Figure 6 presents our rules for compiling expressions. When compiling an expression e under an environment Φ, there are two possible outcomes. The first is that the compilation is successful—it is able to accurately track the result of e:

$$\Phi \vdash e \rightsquigarrow (\vec{I}, \tau)$$

In this case, the compilation produces a list of extended-JVML instructions that put the value of the expression on the top of the JVML stack, and also returns the JVML type of the value. An example is the rule for a constant n, in which it just pushes the constant onto the stack.

The second case is that the compilation fails—it cannot track the value of the expression:

$$\Phi \vdash e \rightsquigarrow ?$$

$$\boxed{\Phi \vdash e \rightsquigarrow (\vec{I}, \tau) \text{ or } \Phi \vdash e \rightsquigarrow ?}$$

$$\frac{}{\Phi \vdash n \rightsquigarrow ([\mathsf{push}\ n], \mathsf{int})}$$

$$\frac{\Phi \vdash e_1 \rightsquigarrow (\vec{I_1}, \mathsf{int}) \quad \Phi \vdash e_2 \rightsquigarrow (\vec{I_2}, \mathsf{int})}{\Phi \vdash e_1 + e_2 \rightsquigarrow (\vec{I_1} @ \vec{I_2} @ [\mathsf{add}], \mathsf{int})}$$

$$\frac{}{\Phi \vdash e_1 + e_2 \rightsquigarrow ?}$$

$$\frac{\Phi(x) = (d, \tau)}{\Phi \vdash x \rightsquigarrow ([\mathsf{load}\ d], \tau)} \qquad \frac{x \notin \mathrm{dom}(\Phi)}{\Phi \vdash x \rightsquigarrow ?}$$

$$\frac{}{\Phi \vdash *e \rightsquigarrow ?}$$

Figure 6. Rules for compiling expressions

In this case, the compilation returns "?". An example is the rule for compiling $*e$, in which it always return "?"; the rules do not model C memory.

Rules in Figure 6 are all straightforward. The only point we would like to make is that the two rules for $e_1 + e_2$ are ordered, in the sense that only if the first rule is not applicable, the second rule applies. Many rules that we will see later are in this style.

Compile or choose. Compilation of an expression may produce "?", an untracked value. In certain cases, however, a value for the expression is needed even though it may be untracked. For example, to compile "if $(e)\ s_1\ s_2$", we always need an integer value for e—this is exactly where the new JVML instruction "choose τ" can come into play. In Figure 7, we define a notion of $\mathsf{compileOrChoose}(\Phi, e, \tau)$, which tracks the exact value of e if the compilation of e succeeds in producing a value of type τ, and otherwise chooses a random value of type τ. In addition to expressions, this compile-or-choose concept is also defined on JNI API calls; we defer this definition to the place when we discuss the compilation of JNI API calls.

$$\mathsf{compileOrChoose}(\Phi, e, \tau) = \begin{cases} \vec{I}, & \text{if } \Phi \vdash e \rightsquigarrow (\vec{I}, \tau) \\ [\mathsf{choose}\ \tau], & \text{otherwise} \end{cases}$$

$$\mathsf{jvmEffects}(\Phi, e) = [\,]$$

Figure 7. The definitions of $\mathsf{compileOrChoose}(-,-,-)$ and $\mathsf{jvmEffects}(-,-)$, for expressions

Compiling statements. In Figure 8 we present rules for compiling C statements, using the judgment

$$\Phi, \tau_{\mathrm{ret}} \vdash s \rightsquigarrow \vec{I}.$$

In words, the C statement s is compiled to a sequence $\vec{I}$ of extended-JVML instructions, under Φ and τ_{ret}. The type τ_{ret} is mapped from the declared return type of the function that s belongs to. It is used in the rule for return(e) statement to inform it of the expected type of e.

Most rules in Figure 8 are straightforward. As an example, to compile "if (e) s_1 s_2", we first compile e, s_1, and s_2; then we insert appropriate comparisons, labels and jumps. Furthermore, in the case that the compilation of e fails, a "choose int" is inserted in the place of e.

There are two rules for the case of $x = \delta$. The second rule uses a $\mathsf{jvmEffects}(\Phi, \delta)$ function, which returns a sequence of instructions that reflects the possible effects of executing δ. This is because δ may be a JNI API call that has effects on the JVM heap. Since expressions have no effects, $\mathsf{jvmEffects}(\Phi, e)$ is defined to be the empty sequence in Figure 7.

The compilation rule for "$*e = \delta$" deserves explanation. It uses two auxiliary predicates described below; the computation of these predicates will be described in Section 3.2.

$\mathsf{mayPtJVM}(e)$:	expression e may point to the JVM heap
$\mathsf{mayPtTo}(e, x)$:	expression e may point to the address of variable x

If e may point to the JVM heap, then the assignment "$*e = \delta$" may modify the heap. In this case, we give it a top specification, which means the assignment may have any effect on the JVM heap. To ensure soundness of the specification, this rule is very conservative. Other extractors may produce a more precise specification, by using a more accurate analysis.

The assignment "$*e = \delta$" can modify some local variables, since $*e$ may be aliases of the local variables. This is why the rule for $*e = \delta$ also sets all the possibly affected variables to random values.

To be concrete, we present a toy C function and its extended-JVML specification in Figure 9. In the example, the variables "i" and "j" are mapped to JVML local variables with indexes 1 and 2, respectively. The pointer variable "p" is not mapped; this is why the "*p" expression in the conditional statement is approximated by the "choose int" instruction. Other than this approximation, the rest of the function is compiled faithfully.

$$\boxed{\Phi, \tau_{\mathrm{ret}} \vdash s \rightsquigarrow \vec{I}}$$

$$\frac{\Phi, \tau_{\mathrm{ret}} \vdash s_1 \rightsquigarrow \vec{I_1} \quad \Phi, \tau_{\mathrm{ret}} \vdash s_2 \rightsquigarrow \vec{I_2}}{\Phi, \tau_{\mathrm{ret}} \vdash s_1; s_2 \rightsquigarrow \vec{I_1} @ \vec{I_2}}$$

$$\frac{\begin{array}{c}\vec{I_e} = \mathsf{compileOrChoose}(\Phi, e, \mathsf{int}) \\ \Phi, \tau_{\mathrm{ret}} \vdash s_1 \rightsquigarrow \vec{I_1} \quad \Phi, \tau_{\mathrm{ret}} \vdash s_2 \rightsquigarrow \vec{I_2}\end{array}}{\Phi, \tau_{\mathrm{ret}} \vdash \text{if } (e)\ s_1\ s_2 \rightsquigarrow \begin{array}{l}\vec{I_e}@ \\ [\mathsf{push}\ 0; \mathsf{ifeq}\ \mathsf{lfalse}]@ \\ \vec{I_1}@[\mathsf{goto}\ \mathsf{lend}]@ \\ [\mathsf{label}\ \mathsf{lfalse}]@\vec{I_2}@ \\ [\mathsf{label}\ \mathsf{lend}]\end{array}} \left(\begin{array}{l}\text{lfalse and} \\ \text{lend fresh}\end{array}\right)$$

$$\frac{\begin{array}{c}\Phi(x) = (d, \tau) \\ \vec{I} = \mathsf{compileOrChoose}(\Phi, \delta, \tau)\end{array}}{\Phi, \tau_{\mathrm{ret}} \vdash x = \delta \rightsquigarrow \vec{I}@[\mathsf{store}\ d]}$$

$$\frac{x \notin \mathrm{dom}(\Phi) \quad \vec{I} = \mathsf{jvmEffects}(\Phi, \delta)}{\Phi, \tau_{\mathrm{ret}} \vdash x = \delta \rightsquigarrow \vec{I}}$$

$$\frac{\begin{array}{l}\vec{I_1} = \mathsf{jvmEffects}(\Phi, \delta) \\ \vec{I_2} = \begin{cases} [\mathsf{top}], & \text{if } \mathsf{mayPtJVM}(e) = \mathrm{true} \\ [\,], & \text{otherwise}\end{cases} \\ \mathsf{mayModVar} = \{\Phi(x) \mid \mathsf{mayPtTo}(e, x) \wedge x \in \mathrm{dom}(\Phi)\} \\ \vec{I_3} = [\mathsf{choose}\ \tau; \mathsf{store}\ d; \ldots], \text{for each } (d, \tau) \in \mathsf{mayModVar}\end{array}}{\Phi, \tau_{\mathrm{ret}} \vdash *e = \delta \rightsquigarrow \vec{I_1}@\vec{I_2}@\vec{I_3}}$$

$$\frac{\vec{I_1} = \begin{cases} \vec{I}@[\mathsf{pop}], & \text{if } \Phi \vdash \mathit{japi} \rightsquigarrow (\vec{I}, \tau) \\ \vec{I}, & \text{if } \Phi \vdash \mathit{japi} \rightsquigarrow (\vec{I}, ?)\end{cases}}{\Phi, \tau_{\mathrm{ret}} \vdash \mathit{japi} \rightsquigarrow \vec{I_1}}$$

$$\frac{\vec{I} = \begin{cases} [\mathsf{return}], & \text{if } \tau_{\mathrm{ret}} = \mathsf{void} \\ \mathsf{compileOrChoose}(\Phi, e, \tau_{\mathrm{ret}})@[\mathsf{returnval}], & \text{otherwise}\end{cases}}{\Phi, \tau_{\mathrm{ret}} \vdash \mathrm{return}(e) \rightsquigarrow \vec{I}}$$

$$\frac{\vec{I} = \begin{cases} [\mathsf{return}], & \text{if } \tau_{\mathrm{ret}} = \mathsf{void} \\ [\mathsf{choose}\ \tau_{\mathrm{ret}}; \mathsf{returnval}], & \text{otherwise}\end{cases}}{\Phi, \tau_{\mathrm{ret}} \vdash \mathrm{return} \rightsquigarrow \vec{I}}$$

Figure 8. Rules for compiling statements

[7] Note that there is a difference between the ifeq in the JVML$_f$ by Freund and Mitchell and the one in the JVML. The version in JVML$_f$ takes two operands, while the version in JVML takes only one.

(a) A TOY C FUNCTION.

```
int f(int i, int *p) {
  int j;
  if (*p) j = i;
  else j = i + i;
  return j;
}
```

(b) EXTENDED-JVML SPECIFICATION

```
   ; We use ; to start comments
   ; Phi = {i : (1, int), j : (2, int)}

   ; for if (*p) ...
  choose int
  push 0
  ifeq[7] lfalse

   ; for j = i
  load 1
  store 2
  goto lend

 label lfalse:
   ; for j = i + i
  load 1
  load 1
  add
  store 2

 label lend:
   ; for return j
  load 2
  returnval
```

Figure 9. A compilation example

Compiling JNI API calls. Java applications that interact with C code through the JNI changes the JVM state mostly through the JNI API functions. We could possibly give the top specification to all JNI API calls, but that would lose a lot of useful information. Therefore, the extractor treats JNI API calls as special, interpreting them at a higher-level of precision.

Figure 11 presents the rules for compiling JNI API calls. The compilation may result in two cases. The first case is

$$\Phi \vdash japi \rightsquigarrow (\vec{I}, \tau),$$

in which the instruction sequence $\vec{I}$ performs the side effects of the JNI API call, and also puts a value of type τ on the top of the stack.

The second case is that

$$\Phi \vdash japi \rightsquigarrow (\vec{I}, ?),$$

in which the compilation cannot track the return value of the JNI API call. However, it may still have side effects on the JVM heap, which is captured by $\vec{I}$. Note that in this case, the stack before the execution of $\vec{I}$ will be the same as the stack after.

$$\begin{aligned}&\mathsf{compileOrChoose}(\Phi, japi, \tau)\\ &\quad = \begin{cases} \vec{I}, & \text{if } \Phi \vdash japi \rightsquigarrow (\vec{I}, \tau) \\ \vec{I}@[\mathsf{pop}; \mathsf{choose}\ \tau], & \text{if } \Phi \vdash japi \rightsquigarrow (\vec{I}, \tau') \\ & \text{and } \tau' \neq \tau \\ \vec{I}@[\mathsf{choose}\ \tau], & \text{if } \Phi \vdash japi \rightsquigarrow (\vec{I}, ?) \end{cases}\end{aligned}$$

$$\begin{aligned}&\mathsf{jvmEffects}(\Phi, japi) =\\ &\quad = \begin{cases} \vec{I}@[\mathsf{pop}], & \text{if } \Phi \vdash japi \rightsquigarrow (\vec{I}, \tau) \\ \vec{I}, & \text{if } \Phi \vdash japi \rightsquigarrow (\vec{I}, ?) \end{cases}\end{aligned}$$

$$\mathsf{mayNull}(\vec{I}) = \begin{array}{l}[\mathsf{choose\ int; push\ 0; ifeq\ lfalse}]@\vec{I}@\\ [\mathsf{goto\ lend; label\ lfalse; push\ null}]@\\ [\mathsf{label\ lend}],\end{array}$$

where lfalse and lend are fresh labels.

Figure 10. The definitions of $\mathsf{compileOrChoose}(-,-,-)$ and $\mathsf{jvmEffects}(-,-)$, for JNI API calls, and the definition of $\mathsf{mayNull}(-)$

The rules in Figure 11 are straightforward translations of the semantics described in Table 1. The rules for $\mathrm{GetIntArrayElements}$ use an auxiliary definition $\mathsf{mayNull}(-)$ in Figure 10, to capture the case of null return values when the operation fails. Using null return values to indicate error conditions is very common in the JNI.

The semantics of JNI API functions is described in the JNI manual [24]. A faithful compilation involves a careful reading of the manual. This process is straightforward, although laborious.

3.1 Properties of the specification extractor

We describe a few interesting theorems about the specification extractor, based on the formalism. The proofs of these theorems are presented in appendix B.

First, the compilation is *complete* in the following sense:

THEOREM 1. *(Compilation Completeness) For any C function fn, there exists a sequence $\vec{I}$ of extended-JVML instructions such that $\vdash fn \rightsquigarrow \vec{I}$.*

Next, we show a theorem that states the list of instructions produced is always well-typed in the extended JVML. Stating this theorem requires typing rules for the new instructions; the typing rules are shown in the appendix A.

THEOREM 2. *(Well-typed Compilation) If $\vdash fn \rightsquigarrow \vec{I}$, then $\vec{I}$ is a sequence of well-typed extended-JVML instructions.*

The definition of well-typed extended-JVML instruction sequences and the proof of the theorem are sketched in appendix B.

$$\boxed{\begin{array}{l}\Phi \vdash \mathit{japi} \rightsquigarrow (\vec{I}, \tau) \\ \text{or } \Phi \vdash \mathit{japi} \rightsquigarrow (\vec{I}, ?)\end{array}}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I}, \mathsf{Array}\ \tau)}{\Phi \vdash \mathrm{GetArrayLength}(e) \rightsquigarrow (w@[\mathsf{arraylength}], \mathsf{int})}$$

$$\frac{}{\Phi \vdash \mathrm{GetArrayLength}(e) \rightsquigarrow ([\,], ?)}$$

$$\frac{\Phi \vdash e \rightsquigarrow (\vec{I}, \mathsf{Array\ int}) \quad \vec{I_n} = \mathsf{mayNull}(w)}{\Phi \vdash \mathrm{GetIntArrayElements}(e) \rightsquigarrow (\vec{I_n}, \mathsf{Array\ int})}$$

$$\frac{}{\Phi \vdash \mathrm{GetIntArrayElements}(e) \rightsquigarrow ([\,], ?)}$$

$$\frac{}{\Phi \vdash \mathrm{ReleaseIntArrayElements}(e) \rightsquigarrow ([\,], ?)}$$

$$\frac{\begin{array}{c}\Phi \vdash e_1 \rightsquigarrow (\vec{I_1}, \mathsf{Array}\ \tau) \\ \vec{I_2} = \mathsf{compileOrChoose}(\Phi, e_2, \mathsf{int})\end{array}}{\Phi \vdash \begin{array}{c}\mathrm{GetObjectArrayElement}(e_1, e_2) \\ \rightarrow (\vec{I_1}@\vec{I_2}@[\mathsf{arrayload}\ \tau], \tau)\end{array}}$$

$$\frac{}{\Phi \vdash \mathrm{GetObjectArrayElement}(e_1, e_2) \rightsquigarrow ([\,], ?)}$$

$$\frac{\begin{array}{c}\Phi \vdash e_1 \rightsquigarrow (\vec{I_1}, \mathsf{Array}\ \tau) \\ \vec{I_2} = \mathsf{compileOrChoose}(\Phi, e_2, \mathsf{int}) \\ \vec{I_3} = \mathsf{compileOrChoose}(\Phi, e_2, \tau)\end{array}}{\Phi \vdash \begin{array}{c}\mathrm{SetObjectArrayElement}(e_1, e_2, e_3) \rightarrow \\ (\vec{I_1}@\vec{I_2}@\vec{I_3}@[\mathsf{arraystore}\ \tau], ?)\end{array}}$$

$$\frac{}{\Phi \vdash \begin{array}{c}\mathrm{SetObjectArrayElement}(e_1, e_2, e_3) \rightarrow \\ ([\mathsf{choose\ (java/lang/Object);\ mutate}], ?)\end{array}}$$

$$\frac{\vec{I} = \mathsf{compileOrChoose}(\Phi, e, \mathsf{int})}{\Phi \vdash \mathrm{NewIntArray}(e) \rightsquigarrow (\vec{I}@[\mathsf{newarray\ int}], \mathsf{Array\ int})}$$

$$\frac{}{\Phi \vdash \mathrm{NewObjectArray}(e_1, e_2, e_3) \rightsquigarrow ([\mathsf{top}], ?)}$$

Figure 11. Rules for compiling JNI API calls

We also conjecture that the sequence of extended-JVML instructions is a conservative approximation of the C function, in terms of effects on the JVM heap.

CONJECTURE 1. *(Conservative Compilation) Assuming a well-behaved and well-typed C function fn. If* $\vdash$ *fn* $\rightsquigarrow \vec{I}$*, then the semantics of* $\vec{I}$ *conservatively approximates the semantics of fn: if the C function fn brings a state* $(A; h; w)$ *to* $(A'; h'; w')$*, then the dynamic semantics of* $\vec{I}$ *can bring the JVM state* $(A; h)$ *to* $(A'; h')$*.*

Proving this conjecture, however, requires modeling the CIL's intermediate language semantics; we leave it to future work.

3.2 Additional features

Having formalized a subset of our specification extractor, we discuss its other features. At a high level, the extractor tries to capture common and easy cases. An example is that comparing Java references to null is fairly common in JNI glue code, since JNI API functions often use null return values to indicate error conditions. The system compiles such null comparisons directly to ifnull and ifnonnull instructions. On the other hand, the system leaves rare or difficult cases for approximation. For example, it does not track values in C structs, unions, or arrays; these values are approximated using the choose τ instruction.

The extractor can handle nearly all C features: loops; structs; unions; arrays; global variables; type casts; function calls, including C library function calls; JNI API functions, including those that perform field accesses, Java-method invocations, catching/throwing exceptions, among many others.

Before we describe a few important features of the extractor, we would like to point out two assumptions of the extractor. First, it generates specifications that faithfully capture the JVM-heap effect, but may ignore other effects. If a Java analysis depends on other effects such as whether a Java object may be referenced, the extractor needs to be adjusted accordingly. Second, the extractor assumes a single-threaded environment. It cannot handle C code that spawns a new thread. It also does not handle JNI API functions related to threads, such as MonitorEnter and MonitorExit. Having stated these assumptions, we believe that they are limitations of the extractor, not of the specification language. For example, the specification language can specify the read access of Java objects, as demonstrated by one of the examples in Section 2. It can also specify creation of new threads by "new Thread().start()".

Accessing fields. The JNI uses a multi-step process to access a field of an object: get the class object of the object; get the field ID; then use the field ID to access the value of the field. An example of reading the value of an integer field named "x" is as follows:

```
//Get the class object
jclass cls = (*env)->GetObjectClass(env, obj);
//Get the field ID
fid = (*env)->GetFieldID(env, cls, "x", "I");
if (fid != NULL) {
   //Get the int field
   int i = (*env)->GetIntField(env, obj, fid);
};
```

Our extractor could track the local dependency information between Java-object references to compile the above code into JVML getfield instructions; it could even use the techniques by Furr and Foster [15] that perform polymorphic type inferences to track the Java types of C references interprocedurally. We decided to follow a simpler route, which compiles GetObjectClass, GetFieldID, and GetIntField into JVML instructions that use the Java reflection API.[8]

There are cases when it is difficult to even know which field the program is accessing (e.g., when the field name in GetFieldID is a computed string). In such situations, the system uses "choose τ" to approximate the value in a field of type τ, and "mutate" for the effect of writing to a field.

Pointer analysis. The compilation rule for $*e = \delta$ in Figure 8 uses two predicates: $\mathsf{mayPtTo}(-,-)$, and $\mathsf{mayPtJVM}(-)$. The first one is computed by a generic pointer-analysis engine provided by the CIL. We compute the second predicate as follows:

1. We define a C attribute jref; a type having this attribute means that values of the type may be a pointer to the JVM heap.
2. We annotate return types of certain JNI API functions to have the jref attribute; we also annotate certain C types such as jobject to have the jref attribute.
3. We implemented a module that propagates jref through the C program as a whole-program analysis. For example, if there is an assignment $p = q$, and q has the jref attribute, then p also gets the attribute. The CIL front end merges all C files into a single file and thus allows whole-program analysis.

After these steps, any pointer of jref attribute may be a pointer to the JVM heap.

More on type mapping. The JNI includes a number of Java-reference types, such as jintArray and jstring. When used in C, all these reference types are aliases of jobject. Nevertheless, our system takes advantage of these reference types to map them to more refined JVML types. For example:

$$\mathsf{mapTy}(\mathsf{jintArray}) = \mathsf{Array\ int}$$
$$\mathsf{mapTy}(\mathsf{jstring}) = \mathsf{java/lang/String}$$

[8] Certain Java analyses may not admit specifications that use the reflection API. In this situation, we can always approximate the field operations; see the following text.

Cutting down the number of functions to compile. Our specification extractor uses a simple strategy to cut down the number of functions that it compiles. It performs a JVM-effect analysis on C functions, and only if the analysis decides that a function may have JVM effects, does the extractor compile the C function. This JVM-effect analysis is based on the jref attribute and the call graph of the whole program.

Function calls. We distinguish two kinds of function calls in C: calling a JVML function, or calling another C function. Calling a JVML function in C goes through a similar process as accessing fields; thus we treat it similar to how we treat field accesses. In the worst case, calling a JVML function produces the top effect.

Calling a C function either results in a real function call, or produces a nop if that function is not compiled (which means it does not have any JVM effect).

C system-library functions. For C system-library functions that may produce effects, we hand built models, and communicated these models to our system using C attributes. Since many of C library functions do not have effects, the default model is nop and thus they do not require any annotation.

Loops. Since the environment Φ in our specification extractor is flow insensitive, compiling C loops does not pose much more difficulty. It is possible to have a more refined analysis in which the environment Φ is flow sensitive; that is, local variables have different types at different locations. A flow-sensitive version means that the system needs to deal with merging environments at join points.

Having discussed the major features of our specification extractor, in Appendix C we present the compilation of two example C functions. The specifications there are in Jasmin [26] JVML assembly syntax.

4. Utilizing specifications

We have proposed an extended-JVML language as the specification language for modeling C components in applications with mixed Java and C components. We have also described a specification extractor for automatically generating models of C components. We claim that via these models, an existing Java analysis, with small changes, can understand the behavior of C components. Therefore, a Java bug finder can catch more bugs, and a Java optimizer can capitalize on more optimization opportunities. We support this claim by next reporting our preliminary experience with a Java bug finder called Jlint[21].

Jlint is a tool that finds bugs in Java class files, in the spirit of lint-like tools [20]. It performs dataflow analysis to identify bugs in Java class files, including null dereferences and deadlocks, among others. We limit our experiments to finding null-related bugs, which is one of the most common and troublesome errors in OO programs.

C code that interacts with the JVM through the JNI is particularly prone to null-related bugs. First, many JNI API functions use null values to report errors. For example, the GetFieldID function returns null when the operation fails.[9] The following code crashes Sun's JVM, when `fid` gets null.

```
    //Get the field ID
    fid = (*env)->GetFieldID(env, cls, "x", "I");
    //Get the int field
    int i = (*env)->GetIntField(env, obj, fid);
```

The correct usage should first check if fid is non-null, before calling GetIntField.

Another source of null-related bugs is when Java code invokes a native C method with null parameters and the C method forgets to check it is non-null before dereferencing it, and vice versa.

Jlint performs inter-procedural dataflow analysis to identify null-related bugs.[10] We retrofitted Jlint so that it accepts our extended-JVML instructions. The changes to Jlint are minor, and we describe them next. At every program location, Jlint keeps a context, tracking states (null or non-null) of local variables and values in the operand stack. For top, the context will not change (since Jlint does not track the nullity of object fields); for mutate, the context will not change except for popping the top value on the stack; for choose τ, we add a value to the operand stack, and mark it as possibly null if τ is a reference type.

4.1 Preliminary experiments

We compiled a set of four Java programs that access C code through the JNI:

- java.lang.StrictMath: extracted from Sun's JDK 1.5.0; It performs basic numeric operations, and uses the JNI to invoke the methods in the C "Freely Distributable Math Library" (fdlibm).
- java.lang.zip: extracted from Sun's JDK 1.5.0; It performs compression and decompression, and uses the JNI to invoke the Zlib C library.
- posix 1.0: a package that provides JNI wrappers for accessing operating-system functions.
- libreadline-java-0.8.0: a package that provides JNI wrappers for accessing the GNU readline library.

We ran preliminary experiments on the above set of programs. We first used the specification extractor to generate extended-JVML models for the C components in the programs. The output syntax of the extractor is the assembler-like syntax of JVML defined in Jasmin [26]. We used Jasmin to convert the models generated by the extractor into Java class files; Jasmin relieves us of the work such as constructing constant pools, calculating label offsets, among others. Jasmin was changed slightly to encode the extra JVML instructions via unused JVML opcodes. We then fed the class files for the C code, together with the ones for the Java code, to the modified Jlint to identify null-related bugs in the programs.

Table 2 shows the results. For each program, Table 2 lists the number of lines of C glue/library code and the number of lines of Java code. Next, it lists specification-extraction statistics, which include time (average of 3 runs) spent on extracting the specification, the number of functions compiled vs. the number of total functions in the program, and the number of lines of code in the extended-JVML specification. Finally, the table presents the number of error messages reported by Jlint, and also the number of false positives. The experiments were performed on a Pentium M processor (1.86 GHz) with 1GB of RAM.

The modified Jlint does not report any null-related bug in the two programs that are extracted from Sun's JDK 1.5.0: java.lang.StrictMath and java.util.zip. This is perhaps because the code from JDK was developed and reviewed rigorously. The absence of null-related bugs in java.lang.StrictMath is hardly surprising. Every native method in the package accepts a double from the JVM, invokes the corresponding routine in the fdlibm library, and returns the result as a double; there are no object references involved. A typical specification, which is for the cosine function, is as follows:

```
.method  Java_StrictMath_cos(Ljava/lang/Class;D)D
    choose double
    dreturn
.end method
```

Readers may wonder whether specifications like the above convey any information at all. They certainly do. For example, this specification tells a Java optimizer that the JVM heap is not changed (or is immutable during the invocation of the native method); furthermore, no exceptions are raised. Many optimizations are enabled by just knowing these two facts. Pechtchanski and Sarkar [32] show that simple immutability specifications result in measurable speedups (5% to 10%) in certain benchmark programs.

For the remaining two programs, libreadline-java-0.8.0 and posix 1.0, the modified Jlint reports a number of null-related bugs. For posix 1.0, Jlint reports 15 bugs in total: 12 bugs are due to not checking for nullity of field IDs before using them; 3 bugs are due to not checking for nullity of the result of the JNI API functions FindClass, GetStringUTFChars, and AllocObject, respectively. Of the 15 bugs, 13 are intra-procedural (which means a possibly null value is used in the same procedure as where it is produced); the other two are inter-procedural. For libreadline-java-0.8.0, Jlint reports 10 bugs: 7 bugs are due to not checking for nullity of the result of the JNI API GetStringUTFChars; 2 bugs

[9] It fails if the specified field cannot be found, or if the class initializer fails, or if the system runs out of memory [24].

[10] More precisely, Jlint performs *limited* inter-procedural analysis in terms of tracking null values: It tracks the case of passing null values to methods as parameters, but does not track the case of possibly null return values; it also does not track the nullity of fields.

Program	Glue/Library C LOC	Java LOC	Time	Compiled/Total # of Functions	JVML LOC	Errors	FP
java.lang.StrictMath	153/8505	1128	0.69s	22/112	323	0	0
java.util.zip	262/8933	824	0.79s	19/109	1771	0	0
posix 1.0	1874/0	860	0.24s	33/33	1361	15	0
libreadline-java-0.8.0	659/1151	1196	0.13s	19/38	1130	10	1

Table 2. Preliminary experimental results for identifying null-related bugs through modified Jlint

are due to not checking for the nullity of the result of NewStringUTF. Of the 9 real bugs, 8 are inter-procedural. One bug is between Java and C: the C code calls a Java method with a possibly null parameter, and the Java method does not check for null before dereferencing the parameter. The one false positive is due to Jlint treating one control-flow path as a possible one for null dereferences, even though the control flow along that path is impossible to happen.

The preliminary experimental results show the utility of the specifications of native C code to existing Java-code analyzers: with small changes, Jlint can find null-related bugs in native C code, and propagate information from the C code back into Java.

5. Related work

Work related to this paper can be divided into two categories: (1) work related to alternative specification languages and automatic specification extraction; (2) work in the area of analyzing software applications written in multiple programming languages.

In Section 1.1, we have already mentioned much related work on alternative specification languages. Comparing to them, our specification language, in general, is expressive, yet facilitates automatic extraction of specifications. The Hoare-logic style specifications used by Spec# and ESC/Java are also expressive, and ESC/Java has a system called Houdini [10] that can automatically adds specifications to programs, similar to our specification extractor. Compared to our specifications, specifications in ESC/Java are more compact. On the other hand, to teach an existing Java analysis to accept and utilize logic-based specifications may be a nontrivial effort, while it is relatively easy to change it to accept our new instructions. Finally, it is worth remarking that the program logics used in ESC/Java and Spec# are *not* complete, as they are based on first-order predicate logic [4].

Our work belongs to the general category of analyzing software systems written in multiple programing languages, or multilingual software. Representative work in this category includes: tracking types of data passed from a strongly typed language to a weakly typed language to prevent the second language from misusing the types [14, 15]; the interoperation between two safe languages when they have different systems of computation effects such as exceptions [36]; ensuring safe interoperation between statically typed languages and dynamically typed languages [16, 25]; and interfacing different languages with C code through Foreign Function Interfaces (FFI) [2, 23, 8, 9]. One approach for reducing the number of programming errors in the JNI is to have a framework that allows programmers to develop mixed Java and C code in a single language. Such frameworks include Jeannie [18] and Janet [3]. Jeannie is particularly expressive in that it allows arbitrary Java code to be embedded in C code, and vice versa; Jeannie also performs many checks to detect programming errors such as type errors. Other analysis tools (and compilers) handle multilingual issues by compiling to a common intermediate representation. For example, the Fortify Source Code Analyzer [12] compiles Java, C, and C++ to a common intermediate representation. As another example, Microsoft compiles a variety of languages to its MSIL intermediate representation [6]. These systems have the key advantage that they can, in principle, perform extremely precise analyses across language boundaries. However, building such a general infrastructure is much more difficult than the approach we propose as it requires an analysis to work with least common denominator representations and code.

6. Conclusion and future work

In understanding multilingual software applications, which specification language to use is worth discussion. We believe that the specification language we have proposed offers a good trade-off between completeness, easy migration of existing tools, and easy construction of a range of specification extractors.

We mention some possible future work. In our framework, the specification extractor is part of the trusted computing base (TCB). If the generated specifications have mistakes, an analysis based on them may get wrong results. We believe that most of the specification extractor can be removed from the TCB using the PCC/TAL techniques [28, 27]. Second, although our framework is based on the JVML, the techniques should be applicable to .NET CLR [6]. Finally, we plan to remove the assumption of single-threaded C code in the extractor. We believe the general methodology can be extended to specify multi-threaded C code in the extended JVML. The extension will enable an analysis to find concurrency-related bugs such as deadlocks across Java and C code.

Acknowledgments

We are grateful to Martin Hirzel and the anonymous reviewers for their constructive comments.

References

[1] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Post Proceedings of International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69, 2004.

[2] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[3] M. Bubak, D. Kurzyniec, and P. Luszczek. Creating Java to native code interfaces with Janet extension. In *First Worldwide SGI Users' Conference*, pages 283–294, 2000.

[4] E. Clarke. *Completeness and incompleteness theorems for Hoare-like axiom systems.* PhD thesis, Cornell University, 1976.

[5] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software engineering (ICSE)*, pages 439–448, 2000.

[6] Ecma International. *Common Language Infrastructure (CLI)*, 4th edition, June 2006. Standard ECMA-335.

[7] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 302–312, 2003.

[8] S. Finne, D. Leijen, E. Meijer, and S. P. Jones. Calling hell from heaven and heaven from hell. In *ACM International Conference on Functional programming (ICFP)*, pages 114–125, 1999.

[9] K. Fisher, R. Pucella, and J. H. Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[10] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[12] Fortify. http://www.fortifysoftware.com/.

[13] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, 2003.

[14] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.

[15] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.

[16] K. E. Gray, R. B. Findler, and M. Flatt. Fine-grained interoperability through mirrors and contracts. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–245, 2005.

[17] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Proceedings of the Second Conference on Domain-Specific Languages*, pages 39–52, 1999.

[18] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007. To appear.

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *the Companion to the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–136, 2004.

[20] S. Johnson. Lint, a C program checker. Unix Documentation, 1977.

[21] K. Knizhnik and C. Artho. *Jlint manual*, 2002. `http://artho.com/jlint/manual.html`.

[22] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *International Conference on Compiler Construction (CC)*, pages 287–304, April 2005.

[23] X. Leroy. *The Objective Caml system.* `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[24] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[25] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *34th ACM Symposium on Principles of Programming Languages (POPL)*, pages 3–10, 2007.

[26] J. Meyer, D. Reynaud, and I. Kharon. Jasmin. `http://jasmin.sourceforge.net/`, 2004.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 85–97, 1998.

[28] G. C. Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[29] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[30] M. Norrish. *Formalising C in HOL.* PhD thesis, University

of Cambridge, 1998.

[31] I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 195–210, 2001.

[32] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 202–211, 2002.

[33] F. Qian and L. J. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.

[34] Splint. http://www.splint.org/.

[35] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java Native Interface. In *Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 97–106, 2006.

[36] V. Trifonov and Z. Shao. Safe and principled language interoperation. In *8th European Symposium on Programming (ESOP)*, pages 128–146, 1999.

A. Typing rules for the extended-JVML instructions

We use the following judgment in Freund and Mitchell [13] to specify the typing rules of our new JVML instructions:

$$\Gamma, F, S, i \vdash P : M$$

The judgment means that instruction i of P is well-typed given Γ, F, S, and M. Each component in the judgment is explained below:

- Γ: A global environment; the representation of a JVML$_f$ program.
- F: A map from addresses to variable types, which map local variables to types such that $F_i[d]$ is the type of local variable d at address i.
- S: A map from addresses to stack types. We use S_i as the type of the operand stack at address i of the program.
- P: A JVML$_f$ method, where $P[i]$ represents the JVML$_f$ instruction at location i.
- M: A method reference, which contains the method's argument types and return type, among other things.

With the judgment, the typing rules for our new JVML instructions are specified below (in the style of Freund and Mitchell):

$\Gamma, F, S, i \vdash P : M$

$P[i] = \mathsf{choose}\ \tau$
$\Gamma \vdash \tau \cdot S_i <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in \mathrm{dom}(P)$

$\Gamma, F, S, i \vdash P : M$

$P[i] = \mathsf{mutate}$
$S_i = \tau \cdot \beta$ $\tau \in \textit{Simple-Ref} \cup \textit{Array}$
$\Gamma \vdash \beta <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in \mathrm{dom}(P)$

$\Gamma, F, S, i \vdash P : M$

$P[i] = \mathsf{top}$
$\Gamma \vdash S_i <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$ $i+1 \in \mathrm{dom}(P)$

B. Properties of the specification extractor

B.1 Compilation completeness

THEOREM. *For any C function fn, there exists a sequence $\vec{I}$ of extended-JVML instructions such that $\vdash$ fn $\leadsto \vec{I}$.*

The proof of the theorem is mainly based on the following lemma.

LEMMA 3. *For all s, Φ, and τ_{ret}, there exists $\vec{I}$, such that $\Phi, \tau_{\mathrm{ret}} \vdash s \leadsto \vec{I}$.*

The proof is by induction over the syntax of s. It uses Lemma 4 below.

LEMMA 4.

(a) $\mathsf{compileOrChoose}(\Phi, \delta, \tau)$ *is a total function.*
(b) $\mathsf{jvmEffects}(\Phi, \delta)$ *is a total function.*

B.2 Well-typed compilation

To state and prove the well-typed compilation theorem, a few concepts are necessary. First, the JVML model in our paper uses abstract labels, while the model of Freund and Mitchell uses concrete addresses and offsets. To bridge the gap, we introduce a judgment

$$st \vdash \vec{I} \Rightarrow P.$$

The judgment assumes a start address st, and converts $\vec{I}$ to P, which is a function from concrete addresses to extended-JVML instructions. The conversion can be specified in two phases: the first phase collects the abstract labels defined in $\vec{I}$ and calculates their concrete addresses; the second phase replaces the labels in the jump instructions of $\vec{I}$ with offsets, based on the information collected in the first phase. The formal specification of the conversion is straightforward and we omit its detail.

Next, to prove the well-typed compilation theorem, we need to express the invariants associated with a sequence $\vec{I}$ of instructions in judgments such as $\Phi \vdash e \rightsquigarrow (\vec{I}, \tau)$. The model by Freund and Mitchell provides rules for checking complete JVML methods. However, the sequence $\vec{I}$ corresponds to only a fragment of a JVML method. Therefore, we need a concept, which defines well-typed instruction sequences, without knowing the whole method (so called compositional reasoning).

DEFINITON 3. *We write $\Gamma, F, S, i \vdash_{\mathrm{f}} P$ to mean that instruction i of fragment P is well typed under Γ, F and S.*

We put the letter "f" in the judgment to indicate that it checks a method fragment P, not a complete method. The rules for $\Gamma, F, S, i \vdash_{\mathrm{f}} P$ closely follow the corresponding rules for $\Gamma, F, S, i \vdash P : M$, except that certain requirements are removed. For example, the case for choose τ is as follows:

$\Gamma, F, S, i \vdash_{\mathrm{f}} P$

$P[i] = \mathsf{choose}\ \tau$
$\Gamma \vdash \tau \cdot S_i <: S_{i+1}$ $\Gamma \vdash F_i <: F_{i+1}$

Comparing to the previous rule, the difference is that the requirement $i + 1 \in \mathrm{dom}(P)$ is removed—P is only a fragment of a JVML method, the next address, $i + 1$, may not be in the domain of P. The requirement $i+1 \in \mathrm{dom}(P)$ in other rules is similarly dropped.[11]

DEFINITON 4. *A method fragment P is well typed under environment Γ, type information F and S, written as*

$$\Gamma, F, S \vdash_{\mathrm{f}} P,$$

if $\Gamma, F, S, i \vdash_{\mathrm{f}} P$, for all $i \in \mathrm{dom}(P)$.

Before we state the main theorem, we first introduce a few definitions.

DEFINITON 5. *Given a C function fn, we write $\Phi(fn)$ to denote the compilation environment constructed from the formal parameters and local variables of fn. It is defined in the same way as how Φ is constructed in the rule for $\vdash fn \rightsquigarrow \vec{I}$ in Figure 5.*

DEFINITON 6. *Given a compilation environment Φ, we construct a type information F as follows:*

$$F(\Phi)_i[d] = \tau, \text{ if } \Phi(x) = (d, \tau) \text{ for some } x$$

DEFINITON 7. *Let Γ_{emp} denote the empty global environment.*

[11] For the return instructions, we also remove the requirement that the type of the return value has to match the return type declared; we can prove this easily for our specification extractor as a separate lemma.

The JVML instructions generated by our specification extractor do not use program-specific information. The empty environment is sufficient to type check them.

Now we are ready to state the main theorem:

THEOREM. *(Well-typed compilation) If $\vdash fn \rightsquigarrow \vec{I}$, and $st \vdash \vec{I} \Rightarrow P$, then there exists a type information S, such that*

$$\Gamma_{\mathrm{emp}}, F(\Phi(fn)), S \vdash_{\mathrm{f}} P.$$

The proof of the theorem needs a few auxiliary lemmas for expressing the invariants of the instruction sequence $\vec{I}$ produced by the judgments in the specification extractor. For example, in $\Phi \vdash e \rightsquigarrow (\vec{I}, \tau)$, the sequence $\vec{I}$ is supposed to compute a value of type τ and put it on the top of the stack. Furthermore, everything already on the stack before the execution of $\vec{I}$ are not touched by $\vec{I}$. We next state this lemma.

DEFINITON 8. *We use the notation $(S \diamond \beta)$ to stand for a type information that adds the stack type β to the end of every stack type in S. It is defined as*

$$(S \diamond \beta)_i = S_i \bullet \beta,$$

where $S_i \bullet \beta$ is the concatenation of S_i and β.

LEMMA 5. *If $\Phi \vdash e \rightsquigarrow (\vec{I}, \tau)$ and $st \vdash \vec{I} \Rightarrow P$, then there exists S such that*

a) S_{st} is the empty stack type,
b) $S_{st+|P|} = \tau$, where $|P|$ denotes the size of P,
c) for all stack type β, we have $\Gamma_{\mathrm{emp}}, F(\Phi), S \diamond \beta \vdash_{\mathrm{f}} P$

In words, the lemma expresses that the P will eventually compute a value of type τ at the top of the stack, and it will not change any value already on the stack before P. A similar lemma can be stated for $\Phi \vdash japi \rightsquigarrow (\vec{I}, \tau)$.

LEMMA 6. *If $\Phi, \tau_{\mathrm{ret}} \vdash s \rightsquigarrow \vec{I}$, and $st \vdash \vec{I} \Rightarrow P$, then there exists S such that*

a) S_{st} is the empty stack type,
b) $S_{st+|P|}$ is the empty stack type,
c) for all stack type β, we have $\Gamma_{\mathrm{emp}}, F(\Phi), S \diamond \beta \vdash_{\mathrm{f}} P$

A similar lemma can be stated for $\Phi \vdash japi \rightsquigarrow (\vec{I}, ?)$.

C. Compilation examples

(a) AN EXAMPLE C FUNCTION.

```
int main() {
  int i, sum = 0;
  int *p = &i;
  for (i = 0; i < 10; i++) sum = sum + *p;
  return sum; }
```

(b) EXTENDED-JVML SPECIFICATION, IN JASMIN SYNTAX.

```
.method  main()I
  .limit locals 3
  .limit stack 999
      ; Jasmin uses ; to start comments
      ; Phi = {i : (1, int), sum : (2, int)}

      ; for sum = 0
    iconst_0
    istore_2
      ; for i = 0
    iconst_0
    istore_1
  loopbegin-0:
      ; for if (i < 10) ...
    iload_1
    ldc 10
    if_icmpge iffalse-4
    iconst_1
    goto endif-5
  iffalse-4:
    iconst_0
  endif-5:
    ifeq iffalse-2
    goto endif-3
  iffalse-2:
    goto loopend-1
  endif-3:
      ; for sum = sum + (*p)
    choose int
    istore_2
      ; for i = i + 1
    iload_1
    iconst_1
    iadd
    istore_1
    goto loopbegin-0
  loopend-1:
      ; for return sum
    iload_2
    ireturn
.end method
```

Figure 12. A compilation example.

```
class IntArray {
  /* declare a native method */
  private native int sumArray(int arr[]);
  public static void main(String args[]) {
    IntArray p = new IntArray();
    int arr[] = new int [10];
    for (int i = 0; i < 10; i++) arr[i] = i;
    /* call the native method */
    int sum = p.sumArray(arr);
    System.out.println("sum = " + sum);
  }
  static {
    /* load the DLL library that implements
       the native method */
    System.loadLibrary("IntArray");
  }
}
```

```
#include <jni.h>
#include "IntArray.h"
JNIEXPORT jint JNICALL
Java_IntArray_sumArray
(JNIEnv *env, jobject self, jintArray arr)
/* env is an interface pointer through which a JNI API
  function can be called; self is the reference to the object
  on which the method is invoked; arr is the reference to
  the array. */
{
  int len = (*env)->GetArrayLength(env, arr);
  int i, sum = 0;
  jint *body =
    (*env)->GetIntArrayElements(env, arr, 0);
  test if body is null
  for (i=0; i<len; i++) {
    sum += body[i];
  }
  (*env)->ReleaseIntArrayElements(env,arr,body,0);
  return sum;
}
```

Figure 13. A JNI example. Java code passes C code an array of integers and C code returns the sum of the array. On the top is the Java code; on the bottom is the C code.

```
.method  Java_IntArray_sumArray(LIntArray;[I)I
  .limit locals 9
  .limit stack 999
      ; Phi = {obj : (1, IntArray), arr : (2, int []),
      ;        len : (3, int), tmp : (4, int),
      ;        i : (5, int), sum: (6, int),
      ;        body: (7, int []), tmp___0 : (8, int [])}

      ; for tmp =
      ;     (*(((*env))->GetArrayLength))(...);
    aload_2
    arraylength
    istore 4
      ; for len = tmp;
    iload 4
    istore_3
      ; for sum = 0;
    iconst_0
    istore 6
      ; for tmp___0 =
      ;     (*(((*env))->GetIntArrayElements))(...);
    choose int
    ifeq iffalse-0
    aload_2
    goto endif-1
  iffalse-0:
    aconst_null
  endif-1:
    astore 8
      ; for body = tmp___0;
    aload 8
    astore 7

      ; for if (body == 0) ...
    aload 7
    ifnonnull iffalse-2
    iconst_0
    ireturn
    goto endif-3
  iffalse-2:
  endif-3:
      ; for i = 0;
    iconst_0
    istore 5
  loopbegin-4:
      ; for if (i < len) ...
    iload 5
    iload_3
    if_icmpge iffalse-8
    iconst_1
    goto endif-9
  iffalse-8:
    iconst_0
  endif-9:
    ifeq iffalse-6
    goto endif-7
  iffalse-6:
    goto loopend-5
  endif-7:
      ; for sum = sum + (*(body + i));
    choose int
    istore 6
      ; for i = i + 1;
    iload 5
    iconst_1
    iadd
    istore 5
    goto loopbegin-4
  loopend-5:
      ; for return sum
    iload 6
    ireturn

    nop
.end method
```

Figure 14. The extended-JVML specification for the C function in Figure 13