Polymorphic Type Inference for the JNI*

Michael Furr and Jeffrey S. Foster

University of Maryland, College Park
{furr,jfoster}@cs.umd.edu

Abstract. We present a multi-lingual type inference system for checking type safety of programs that use the Java Native Interface (JNI). The JNI uses specially-formatted strings to represent class and field names as well as method signatures, and so our type system tracks the flow of string constants through the program. Our system embeds string variables in types, and as those variables are resolved to string constants during inference they are replaced with the structured types the constants represent. This restricted form of dependent types allows us to directly assign type signatures to each of the more than 200 functions in the JNI. Moreover, it allows us to infer types for user-defined functions that are parameterized by Java type strings, which we have found to be common practice. Our inference system allows such functions to be treated polymorphically by using instantiation constraints, solved with semi-unification, at function calls. Finally, we have implemented our system and applied it to a small set of benchmarks. Although semi-unification is undecidable, we found our system to be scalable and effective in practice. We discovered 155 errors and 36 cases of suspicious programming practices in our benchmarks.

1 Introduction

Foreign function interfaces (FFIs) allow programs to call functions written in other languages. FFIs are important because they let new languages access system libraries and legacy code, but using FFIs correctly is difficult, as there is usually little or no compile-time consistency checking between native and foreign code. As a result, programs that use FFIs may produce run-time typing errors or even violate type safety, thereby causing program crashes or data corruption.

In this paper we develop a multi-lingual type inference system that checks for type safe usage of Java's FFI to C, called the Java Native Interface (JNI).[1] In the JNI, most of the work is done in C "glue code," which translates data between the two languages and then in turn invokes other routines, often in system or user libraries. Java primitives can be accessed directly by C glue code because they have the same representations as C primitives, e.g., a Java integer can be given the C type `int`. However all Java objects, no matter what class they are from, are assigned a single opaque type `jobject` by the JNI. Since `jobject`

* This research was supported in part by NSF CCF-0346982 and CCF-0430118

[1] The JNI also contains support for C++, but our system only analyzes C code.

contains no further Java type information, it is easy for a programmer to use a Java object at a wrong type without any compile-time warnings.

We present a new type inference system that embeds Java type information in C types. When programmers manipulate Java objects, they use JNI functions that take as arguments specially-formatted strings representing class names, field names, and method signatures. Our inference system tracks the values of C string constants through the code and translates them into Java type annotations on the C `jobject` type, which is a simple form of dependent types.

We have found that JNI functions are often called with parameters that are not constants—in particular, programmers often write "wrapper" functions that group together common sequences of JNI operations, and the Java types used by these functions depend on the strings passed in by callers. Thus a novel feature of our system is its ability to represent partially-specified Java classes in which class, field, and method names and type information may depend on string variables in the program. During type inference these variables are resolved to constants and replaced with the structured types they represent. This allows us to handle wrapper functions and to directly assign type signatures to the more than 200 functions in the JNI.

Our system also supports polymorphism for functions that are parameterized by string variables and Java types. Our type inference algorithm generates instantiation constraints [1] at calls to polymorphic functions, and we present an algorithm based on semi-unification [2, 3] for solving the constraints.

We have implemented our system and applied it to a small set of benchmarks. Although semi-unification is undecidable, we found our algorithm to be both scalable and effective in practice. In our experiments, we found 155 errors and 36 suspicious but non-fatal programming mistakes in our benchmarks, suggesting that programmers do misuse the JNI and that multi-lingual type inference can reveal many mistakes.

In summary, the main contributions of this paper are as follows:

- We develop a multi-lingual type inference system that embeds Java types inside of C. Our system uses a simple form of dependent types so that Java object types may depend on the values of C string constants and variables.
- We present a constraint-based type inference algorithm for inferring multi-lingual types for C glue code. During inference, as string variables are resolved to constant strings they are replaced with the structured Java types they represent. Our system uses instantiation constraints to model polymorphism for JNI functions and user-defined wrapper functions.
- We describe an implementation of our type inference system. We have applied our implementation to a small set of benchmarks, and as a result, we found a number of bugs in our benchmark suite.

This work complements and extends our previous work on the OCaml-to-C FFI [4]. Our previous system was monomorphic and worked by tracking integers and memory offsets into the OCaml heap. Our previous system also did not model objects, which clearly limits its applicability to the JNI. Our new system

can model accesses to Java objects using string constants and variables, and performs parametric polymorphic type inference.

2 Background

In this section we motivate our system by briefly describing the JNI [5]. The JNI is typically used to access low-level C libraries which are impractical to recode in Java. Libraries may have a significant amount of development time invested and interfacing it with Java via the JNI avoids duplicated programming work. Also, low-level operating system functions are typically only provided by means of a C library (libc) and so the JNI must be used to access them. To call a C function from Java, the programmer first declares a Java method with the `native` keyword and no body. When this native method is invoked, the Java runtime finds and invokes the correspondingly-named C function. Since Java and C share the same representation for primitive types such as integers and floating point numbers, C glue code requires no special support to manipulate them. In contrast, Java objects, such as instances of `Object`, `Class`, or `int[]`, are all represented with a single opaque C type `jobject` (often an alias of `void *`), and glue code invokes functions in the JNI to manipulate `jobject`s. For example, to get the object `Point.class`, which represents the class `Point`, a programmer might write the following C code[2]:

```
jobject pointClass = FindClass("java/awt/Point");
```

Here the `FindClass` function looks up a class by name. The resulting object `pointClass` is used to access fields and methods, as well as create new instances of class `Point`. For example, to access a field, the programmer next writes

```
jfieldID fid = GetFieldID(pointClass, "x", "I");
```

After this call, `fid` contains a representation of the location of the field `x` with type `I` (a Java `int`) in class `Point`. This last parameter is a terse encoding of Java types called a *field descriptor*. Other examples are `F` for float, `[I` for array of integers, and `Ljava/lang/String;` for class `String`. Notice this is a slightly different encoding of class names than used by `FindClass`. Our implementation enforces this difference, but we omit it from the formal system for simplicity.

Finally, to read this field from a `Point` object `p`, the programmer writes

```
jobject p = ...;
int y = GetIntField(p, fid);
```

The function `GetIntField` returns an `int`, and there is one such function for each primitive type and one function `GetObjectField` for objects.

Thus we can see that a simple field access that would be written `int y = p.x` in Java requires three JNI calls, each corresponding to one internal step of the

[2] The JNI functions discussed in this section are actually invoked slightly differently and take an additional parameter, as discussed in Section 4.

```
int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj, fid);
}
```

Fig. 1. JNI Wrapper Function Example

JVM: getting the type of the object, finding the offset of the field, and retrieving its contents. And while a Java compiler only accepts the code `y = p.x` if it is type correct, errors in C glue code, such as typos in the string `java/awt/Point`, `x`, or `I`, will produce a run-time error. There are also several other places where mistakes could hide. For example, the programmer must be careful to maintain the dependence between the type of `x` and the call to `GetIntField`. If the type of `x` were changed to `float`, then the call must also be changed, to `GetFloatField`, something that is easy to overlook. Moreover, since `pointClass` and `p` both have type `jobject`, either could be passed where the other is expected with no C compiler warning, which we have seen happen in our benchmarks. Invoking a Java method is similar to extracting a field. We omit the details due to lack of space.

One common pattern we have seen in JNI code is wrapper functions that specialize JNI routines to particular classes, fields, or methods. For example, Fig. 1 contains a function `my_getIntField` that extracts an integer field from an object. This routine invokes the JNI function `GetObjectClass`, which returns an object representing the class of its argument (as opposed to `FindClass`, which looks up a class by name). Calling `my_getIntField` is safe if the first parameter has an integer field whose name is given by the second parameter. Thus this function is parameterized by the object and by the name of the field, but not its type. Since this wrapper function might be called multiple times with different objects and different field names, we need to perform polymorphic type inference, not only in the types of objects but also in the values of string parameters.

3 Type System

In this section, we describe our multi-lingual type inference system. The input to our system is a collection of Java classes and a C program. Our type inference system analyzes the C program and generates *constraints* that describe how it uses Java objects. We also add constraints based on the type signatures of native methods and the actual class definitions from Java. We then solve the constraints to determine whether the C code uses Java objects consistently with the way they are declared, and we report any discrepancies as type errors.

Extracting type information from Java class files is relatively straightforward. Thus most of the work in our system occurs in the C analysis and constraint resolution phases, so these are the focus of this section.

$$
\begin{array}{rl}
ct ::= & \texttt{void} \mid \texttt{int} \mid \texttt{str}\{s\} \mid (\texttt{ct} \times \cdots \times \texttt{ct}) \rightarrow \texttt{ct} \\
\mid & jt\ \texttt{jobject} \mid (f, o)\ \texttt{jfieldID} \mid (m, o)\ \texttt{jmethodID} \\
jt ::= & \alpha \mid \mathsf{JVoid} \mid \mathsf{JInt} \mid o \mid jt\ \mathsf{JClass} \mid\ \mathsf{JTStr}\{s\} \\
o ::= & \{s; F; M\} \\
s ::= & \nu \mid \textit{"Str"} \\
f ::= & s : jt \\
F ::= & \phi \mid \emptyset \mid \langle f; \cdots ; f \rangle \circ F \\
m ::= & s : (jt \times \cdots \times jt) \rightarrow jt \\
M ::= & \mu \mid \emptyset \mid \langle m; \cdots ; m \rangle \circ M
\end{array}
$$

Fig. 2. Type Language

3.1 Multi-Lingual Types

To check that C glue code uses Java types correctly, we must reconstruct the Java types for C variables that are declared with types like `jobject`, `jfieldID`, and `jmethodID`. Before giving our type language formally, consider the function `my_getIntField` in Fig. 1. Our system will give this function the following type, which corresponds to the informal description given at the end of Section 2:

$$\texttt{my\_getIntField} : \{\nu; \langle \nu_{field} : \mathsf{JInt} \rangle \circ \phi; \mu\}\ \texttt{jobject} \times \texttt{str}\{\nu_{field}\} \rightarrow \mathsf{JInt}\ \texttt{jobject}$$

This is the type of a function that takes two parameters and returns a Java integer (which has the same representation as a C `int`). Note we use the constructor `jobject` to denote any Java type, not just objects. The second parameter is a C string whose contents are represented by the variable ν_{field}. The first parameter is an object that contains a field with name ν_{field} and Java type `int`. Here we have embedded Java type information into the bare C type in order to perform checking. The type of `my_getIntField` places no constraints on the class name ν of the first parameter or its other fields ϕ and methods μ. In order to infer this type, we need to track intermediate information about `cls` and `fid` as well. A detailed explanation of how this type is inferred is given in Section 3.4.

Our formal type grammar is given in Fig. 2. C types `ct` include void, integers, string types $\texttt{str}\{s\}$ (corresponding to C types `char *`), and function types. In type $\texttt{str}\{s\}$, the string s may be either a known constant string *"Str"* or a type variable ν that is later resolved to a constant string. For example, in Fig. 1, `field` is given type $\texttt{str}\{\nu_{field}\}$, and ν_{field} is then used as a field name in the type of `my_getIntField`.

Our type language embeds a Java type jt in the C type `jobject`. In order to model the various ways the JNI can be used to access objects, we need a richer representation of Java types than just simple class names. For example, the wrapper function in Fig. 1 may be safely called with mutually incompatible classes as long as they all have the appropriate integer field. Thus our Java types include type variables α, the primitives JVoid and JInt, and *object descriptions* o of the form $\{s; F; M\}$, which represents an object whose class is named s with *field set* F and *method set* M. Since our inference system may discover the fields and methods of an object incrementally, we allow these sets to grow with the

$$\mathit{FindClass} : (\texttt{str}\{\nu\}) \rightarrow \mathsf{JTStr}\{\nu\}\ \mathsf{JClass}\ \texttt{jobject}$$

$$\mathit{GetObjectClass} : (\{\nu;\phi;\mu\}\ \texttt{jobject}) \rightarrow \{\nu;\phi;\mu\}\ \mathsf{JClass}\ \texttt{jobject}$$

$$\mathit{GetFieldID} : (o\ \mathsf{JClass}\ \texttt{jobject} \times \texttt{str}\{\nu_2\} \times \texttt{str}\{\nu_3\}) \rightarrow (f,o)\ \texttt{jfieldID}\ \texttt{jobject}$$
$$\text{where } o = \{\nu_1; \langle f \rangle \circ \phi; \mu\} \text{ and } f = \nu_2 : \mathsf{JTStr}\{\nu_3\}$$

$$\mathit{GetIntField} : (o\ \texttt{jobject} \times (f,o)\ \texttt{jfieldID}\ \texttt{jobject}) \rightarrow \mathsf{JInt}\ \texttt{jobject}$$
$$\text{where } o = \{\nu_1; \langle f \rangle \circ \phi; \mu\} \text{ and } f = \nu_2 : \mathsf{JInt}$$

Fig. 3. Sample JNI Type Signatures. All unconstrained variables are quantified.

composition operator $\circ$. A set is *closed* if it is composed with $\emptyset$, and it is *open* if it is composed with a variable ϕ or μ. Since we never know just from C code whether we have accessed all the fields and methods of a class, field and method sets become closed only when unified with known Java types.

Instances of Java's `Class` class are essential for using the JNI, and therefore we distinguish them from other objects with the type jt JClass. For example, in Fig. 1 the variable `cls` is given type $\{\nu; \langle \nu_{field} : \mathsf{JInt} \rangle \circ \phi; \mu\}$ JClass `jobject`, meaning it is the class of `obj`. This separation is required so that an instance object is not passed to a function like `GetFieldID` which requires a class object.

Lastly, Java objects whose types are described by string s have type $\mathsf{JTStr}\{s\}$. During inference, when the value of s is determined, $\mathsf{JTStr}\{s\}$ will be replaced by the appropriate type. For example, initially the result type of `my_getIntField` is determined to be $\mathsf{JTStr}\{\text{“}\texttt{I}\text{”}\}$, which is immediately replaced by JInt.

The types (f,o) `jfieldID` and (m,o) `jmethodID` represent intermediate JNI values for extracting field f or method m. The name of a field or method is a string s. For example, `fid` in Fig. 1 has type $(\nu_{field} : \mathsf{JInt}, o)$ where o is our representation of `obj`. We include o so that we can check that this field identifier is used with an object of the correct type.

Given this type grammar, we can precisely describe the types of the JNI functions. Fig. 3 gives the types for the functions we have seen so far in this paper. For instance, the function `GetIntField` takes an object with a field f and a `jfieldID` describing f, and returns an integer. Notice that the object type o is open, because it may have other fields in addition to f. As we discussed in Section 2, it is important that these functions be polymorphic. In the type signatures in Fig. 3, any unconstrained variables are implicitly quantified.

3.2 Constraint Generation

The core of our system is a type inference algorithm that traverses C source code and infers the types in Fig. 2. Due to lack of space, we omit explicit typing rules for the source language, since they are mostly standard. Each C term of type `jobject` is initially assigned type α `jobject` for fresh α, and similarly for

`jfieldID` and `jmethodID`. As we traverse the program, we generate *unification constraints* of the form $a = b$ (where a and b range over `ct`, jt, etc) whenever two terms must have the same type. For example, for the code in Fig. 1, we unify the type of `cls` with the return type of `GetObjectClass`. Since the only way to manipulate `jobject` types is through the JNI, in general our analysis only generates constraints at assignment, function definition, and function application. Because we use unification for jt types rather than subtyping, our inference algorithm could fail to unify two Java objects where one object is a subtype of the other. However, we did not find this to be a problem in practice (see Section 4).

To support polymorphism, we use *instantiation constraints* of the form $a \preceq_i b$ to model substitutions for quantified variables for each instantiation site i [2, 3]. Formally, we use the following two rules for generalization and instantiation:

$$\text{(Let)}\quad \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma[f \mapsto (t_1, fv(\Gamma))] \vdash e_2 : t_2}{\Gamma \vdash \text{let } f = e_1 \text{ in } e_2 : t_2} \qquad \text{(Inst)}\quad \frac{\Gamma(f) = (t, \vec{\alpha}) \qquad (t, \vec{\alpha}) \preceq_i (\beta, \vec{\alpha}) \qquad \beta \text{ fresh}}{\Gamma \vdash f_i : \beta}$$

In (Let), we represent a polymorphic type as a pair $(t, \vec{\alpha})$, where t is the base type and $\vec{\alpha}$ is the set of variables that may *not* be quantified. Then in (Inst), we generate an instantiation constraint $(t, \vec{\alpha}) \preceq_i (\beta, \vec{\alpha})$ where i is unique to this instantiation site. This constraint requires that there exist a substitution S_i such that $S_i(t) = \beta$. Our type rule also demands that $\vec{\alpha} \preceq_i \vec{\alpha}$, i.e., S_i does not instantiate the free variables of the environment. The main advantage to this notation for polymorphism is that it allows us to traverse the source code in any order. In particular, we can generate instantiation constraints for a call to function f before we have seen a definition of f. A full discussion of these rules is beyond the scope of this paper and can be found elsewhere [3].

We have formalized a checking version of our type system in terms of lambda-calculus extended with strings, let-bound polymorphism, and primitives representing Java objects. We also include as primitive a set of JNI functions for operating on Java objects, and the small-step semantics for the language includes a reduction rule for each of these functions. We believe it is straightforward to prove that the reduction rules preserve solutions.

Theorem 1 (Soundness). *If $\Gamma \vdash e : ct$, then there exists a value v such that $e \rightarrow^* v$ and $\Gamma \vdash v : ct$.*

Proof. The proof is available in our companion technical report [6].

In our implementation, we do not keep track of the set $fv(\Gamma)$ for functions. Since C does not have nested functions, we simply issue warnings at any uses of global variables of type `jobject`. In general we have found that programmers use few globals when interacting with the JNI. We do not issue warnings for global `char *` types since these are common in C programs. While this makes our implementation unsound, doing so would generate a very high rate of false positives. Our current implementation also does not check for global variables of type `jfieldID`, `jmethodID`, or any Java types embedded in C data structures.

3.3 Constraint Solving

Our type inference algorithm generates a set of constraints that we need to solve in order to decide if the program is well-typed. To solve the constraints, we use a variant of Fähndrich et al's worklist algorithm for semi-unification [2].

To simplify our constraint resolution rules below, whenever we refer to field and method sets we always assume they have been flattened so that they are composed with either $\emptyset$ or a variable. During the process of unification, unknown strings ν will be replaced by known constant strings $\texttt{str}\{\textit{"Str"}\}$. As this happens, we need to ensure that our object types still make sense. In particular, Java classes may not contain two fields with the same name but different types.[3] Thus we also impose a well-formedness constraint on all field sets: any two fields with the same name in a field set must have the same type. Formally, for a field $f = s : jt$ we define $\mathit{fname}(f) = s$ and $\mathit{ftype}(f) = jt$. Then a field set $\langle f_1; \cdots ; f_n \rangle$ is well-formed if $\mathit{fname}(f_i) = \mathit{fname}(f_j) \Rightarrow \mathit{ftype}(f_i) = \mathit{ftype}(f_j)$ for all i, j. Methods however, unlike fields, may be overloaded in Java, and so we do not apply the above well-formedness condition to them.

During constraint solving, our system may unify a string variable ν with a constant string *"Str"*. When this occurs, our system uses the *Eval* function shown below to convert a type $\mathsf{JTStr}\{s\}$ into the Java type it represents:

$$
\begin{aligned}
\mathit{Eval}(\ \mathsf{JTStr}\{\text{“V”}\}) &\Rightarrow \mathsf{JVoid} \\
\mathit{Eval}(\ \mathsf{JTStr}\{\text{“I”}\}) &\Rightarrow \mathsf{JInt} \\
\mathit{Eval}(\ \mathsf{JTStr}\{\text{“Ljava/lang/String;”}\}) &\Rightarrow \{\text{“java/lang/String”}; \cdots ; \cdots\} \\
\mathit{Eval}(\ \mathsf{JTStr}\{\text{“Ljava/awt/Point;”}\}) &\Rightarrow \{\text{“java/awt/Point”}; \cdots ; \cdots\} \\
&\vdots
\end{aligned}
$$

We use a similar function to convert the string representation of a method signature into a list of Java types.

We express constraint solving in terms of rewrite rules, shown in Fig. 4. Given a set of constraints C, we apply these rules exhaustively, replacing the left-hand side with the right-hand side until we reach a fixpoint. Technically because we use semi-unification this algorithm may not terminate, but we have not found a case of this in practice. The complete list of rewrite rules is long and mostly standard, and so Fig. 4 contains only the interesting cases. The exhaustive set of rules may be found in our companion technical report [6].

In Fig. 4, the (Closure) rule unifies two terms b and c when they are both instantiations of the same variable a at the same instantiation site. Intuitively, this rule enforces the property that substitution S_i must replace variable a consistently [3]. The rule (JTStr Ineq) applies the usual semi-unification rule for constructed types. Since the substitution S_i renames the left-hand side to yield the right-hand side in a constraint $\preceq_i$, the right-hand side must have the same shape. Thus in (JTStr Ineq), we unify jt with $\mathsf{JTStr}\{\nu\}$ where ν is fresh and then propagate the semi-unification constraint to s and ν.

[3] Although an overloaded field via inheritance is possible, their manipulation in C is not supported by our system and was not observed in our benchmarks.

In (FieldSet InEq), we match up the fields of two non-empty field sets. We directly propagate instantiation constraints to fields f_k and f'_j with the same field name or variable. We then make an instantiation constraint to F' from the remaining fields of the left-hand side for which there does not exist a field with the same name on the right-hand side. Recall that we assume the sets have been flattened, so that F' is either a variable or $\emptyset$. We then generate the analogous constraint for F and the right-hand side. Notice that this process may result in a field set with multiple fields with variables ν for names. Our implicit well-formedness requirement from Section 3 will handle the case where these variables are later unified with known strings.

We omit the rules for method sets due to lack of space. These rules are similar to the field set rules except for one important detail. Suppose we have an open method set $\langle x : \alpha \to \alpha \rangle \circ \mu$ which is unified with a closed method set $\langle x : \mathsf{JInt} \to \mathsf{JInt}; x : \mathsf{JVoid} \to \mathsf{JVoid} \rangle \circ \emptyset$. Since the method `x` is overloaded, we do not know if α should unify with JInt or JVoid. Therefore, if this occurs during unification, our tool emits a warning and removes the constraint. Also, we could unify return types of methods with otherwise equal signatures, but do not do so in our current implementation.

The next three rules handle strings. (Str Sub) replaces one string variable by another. (Str Resolve) uses *Eval* to replace occurrences of $\mathsf{JTStr}\{\nu\}$ with their appropriate representations. (Str Eq) and (Str Neq) test string constants for equality.

Because $\mathsf{JTStr}\{\}$ encodes its type as a string, we use a slightly different rewrite rule for this case. In rule (JTStr Sub), if a $\mathsf{JTStr}\{\}$ type is unified with a type variable, then the variable is replaced as normal. However, if a $\mathsf{JTStr}\{s\}$ type is unified with a void type as in (JTStr Void), then we add the constraint that the $s =$ "V", since $Eval(s)$ must produce a void type. We use a similar rule for integers. Similarly, the rule (JTStr Obj) adds the constraint that s must have the same name as the object it unifies with.

3.4 Example

In this section, we demonstrate our inference system on the `my_getIntField` function from Fig. 1. Initially our analysis assigns each parameter and local variable of this function a fresh type (we omit the variable `fid` for brevity as it only provides redundant constraints on `my_getIntField`):

$$\texttt{obj} : \alpha_{obj}\ \texttt{jobject} \quad \texttt{cls} : \alpha_{cls}\ \texttt{jobject} \quad \texttt{field} : \texttt{str}\{\nu_{field}\}$$

The first line of the function calls the `GetObjectClass` function (call this instantiation site 1). After looking up its type in the environment (shown in Fig. 3 with quantified type variables ν, ϕ, and μ), we add the following constraints:

$$\{\nu; \phi; \mu\}\ \texttt{jobject} \preceq_1 \alpha_{obj}\ \texttt{jobject}$$
$$\{\nu; \phi; \mu\}\ \mathsf{JClass}\ \texttt{jobject} \preceq_1 \alpha_{cls}\ \texttt{jobject}$$

$$\text{(Closure)} \quad C \cup \{a \preceq_i b\} \cup \{a \preceq_i c\} \Rightarrow C \cup \{a \preceq_i b\} \cup \{b = c\}$$

$$\text{(JTStr Ineq)} \quad C \cup \{\ \mathsf{JTStr}\{s\} \preceq_i jt\} \Rightarrow C \cup \{jt = \ \mathsf{JTStr}\{\nu\}\} \cup \{s \preceq_i \nu\} \quad \nu \text{ fresh}$$

$$\begin{aligned}\text{(FieldSet InEq)} \quad C \cup \{\langle f_1; \cdots; f_n\rangle \circ F \preceq_i \langle f'_1; \cdots; f'_m\rangle \circ F'\} \Rightarrow \\ C \cup \{\mathit{ftype}(f_k) \preceq_i \mathit{ftype}(f'_j) \mid \mathit{fname}(f_k) = \mathit{fname}(f'_j)\} \\ \cup\{\langle f_k \mid \nexists j.\, \mathit{fname}(f_k) = \mathit{fname}(f'_j)\rangle \preceq_i F'\} \\ \cup\{F \preceq_i \langle f'_j \mid \nexists k.\, \mathit{fname}(f_k) = \mathit{fname}(f'_j)\rangle\}\end{aligned}$$

$$\text{(Str Sub)} \quad C \cup \{\nu_1 = \nu_2\} \Rightarrow C[\nu_1 \mapsto \nu_2]$$

$$\text{(Str Resolve)} \quad C \cup \{\nu = \text{“}\mathit{Str}\text{”}\} \Rightarrow C[\nu \mapsto \text{“}\mathit{Str}\text{”}][\ \mathsf{JTStr}\{\text{“}\mathit{Str}\text{”}\} \mapsto \mathit{Eval}(\text{“}\mathit{Str}\text{”})]$$

$$\text{(Str Eq)} \quad C \cup \{\text{“}\mathit{Str}_1\text{”} = \text{“}\mathit{Str}_2\text{”}\} \Rightarrow C \qquad \mathit{Str}_1 = \mathit{Str}_2$$

$$\text{(Str Neq)} \quad C \cup \{\text{“}\mathit{Str}_1\text{”} = \text{“}\mathit{Str}_2\text{”}\} \Rightarrow \mathit{error} \qquad \mathit{Str}_1 \neq \mathit{Str}_2$$

$$\text{(JTStr Sub)} \quad C \cup \{\alpha = \ \mathsf{JTStr}\{s\}\} \Rightarrow C[\alpha \mapsto \ \mathsf{JTStr}\{s\}]$$

$$\text{(JTStr Void)} \quad C \cup \{\mathsf{JVoid} = \ \mathsf{JTStr}\{s\}\} \Rightarrow C \cup \{s = \text{“}\mathtt{V}\text{”}\}$$

$$\text{(JTStr Obj)} \quad C \cup \{\{\nu; F; M\} = \ \mathsf{JTStr}\{s\}\} \Rightarrow C \cup \{s = \nu\}$$

Fig. 4. Selected Constraint Rewrite Rules

Applying our constraint rewrite rules yields the following new constraints:

$$\begin{array}{cc} \alpha_{obj} = \{\nu_2; \phi_2; \mu_2\} & \alpha_{cls} = \{\nu_3; \phi_3; \mu_3\}\ \mathsf{JClass} \\ \nu \preceq_1 \nu_2 & \nu \preceq_1 \nu_3 \\ \phi \preceq_1 \phi_2 & \phi \preceq_1 \phi_3 \\ \mu \preceq_1 \mu_2 & \mu \preceq_1 \mu_3 \\ \nu_2, \phi_2, \mu_2 \text{ fresh} & \nu_3, \phi_3, \mu_3 \text{ fresh} \end{array}$$

Then rule (Closure) in Fig. 4 generates the constraints $\nu_2 = \nu_3$, $\phi_2 = \phi_3$, and $\mu_2 = \mu_3$ to require that the substitution corresponding to this call is consistent. Next, `my_getIntField` calls `GetFieldID`, and after applying our constraint rules, we discover (among other things): $\phi_2 = \langle \nu_{field} :\ \mathsf{JTStr}\{\text{“I”}\}\rangle \circ \phi_4$ with ϕ_4 fresh. Now since we have a JTStr with a known string, our resolution rules call *Eval* to replace it, yielding $\phi_2 = \langle \nu_{field} : \mathsf{JInt}\rangle \circ \phi_4$. The last call to `GetIntField` generates several new constraints, but they do not affect the types. Thus after the function has been analyzed, it is given the type

$$\texttt{my\_getIntField} : \{\nu_1; \langle \nu_{field} : \mathsf{JInt}\rangle \circ \phi_4; \mu\}\ \texttt{jobject} \times \texttt{str}\{\nu_{field}\} \rightarrow \mathsf{JInt}\ \texttt{jobject}$$

In other words, this function accepts any object as the first parameter as long as it has an integer field whose name is given by the second parameter, exactly as intended.

4 Implementation and Experiments

We have implemented the inference system described in Section 3 in the form of two tools used in sequence during the build process. The first tool is a lightweight Java compiler wrapper. The wrapper intercepts calls to `javac` and records

the class path so that the second tool can retrieve class files automatically. The wrapper itself does not perform any code analysis. The second tool applies our type inference algorithm to C code and issues warnings whenever it finds a type error. Our tool uses CIL [7] to parse C source code and the OCaml JavaLib [8] to extract Java type information from compiled class files.

Our implementation contains some additional features not discussed in the formal system. In addition to the type `jobject`, the JNI contains a number of typedefs (aliases) for other object types, such as `jstring` for Java `String`s. These are all aliases of `jobject`, and so their use is not required by the JNI, and they do not result in any more checking by the C compiler. Our system does not require their use either, but since they are a form of documentation we enforce their intended meaning, e.g., values of type `jstring` are assigned a type corresponding to `String`. We found 3 examples in our benchmarks where programmers used the wrong alias. The JNI also defines types `jvoid` and `jint`, representing Java voids and integers, as aliases of the C types `void` and `int`, respectively. Our system does not distinguish between the C name and its `j`-prefixed counterpart.

Rather than being called directly, JNI functions are actually stored in a table that is passed as an extra argument (usually named `env`) to every C function called from Java, and this table is in turn passed to every JNI function. For example, the `FindClass` function is actually called with `(*env)->FindClass(env,...)`. Our system extracts FFI function names via syntactic pattern matching, and we assume that the table is the same everywhere. Function pointers that are not part of the JNI are not handled by our system, and we do not generate any constraints when they are used in a program.

The JNI functions for invoking Java methods must take a variable number of arguments, since they may be used to invoke methods with any number of parameters. Our system handles the commonly-used interface, which is JNI functions declared to be varargs using the `...` convention in C. However, the JNI provides two other calling mechanisms that we do not model: passing arguments as an array, and passing arguments using the special `va_list` structure. We issue warnings if either is used.

Although our type system is flow-insensitive, we treat the types of local variables flow-sensitively. Each assignment updates the type of a variable in the environment, and we add a unification constraint to variables of the same name at join points in the control flow graph. See [4] for details.

Lastly, our implementation models strings in a very simple way to match how they are used in practice in C glue code. We currently ignore string operations like `strcat` or destructive updates via array operations. We also assume that strings are always initialized before they are used, since most compilers produce a warning when this is not the case.

We ran our analysis tool on a suite of 11 benchmarks that use the JNI. Fig. 5 shows our results. All benchmarks except pgpjava are glue code libraries that connect Java to an external C library. The first 7 programs are taken from the Java-Gnome project [9], and the remaining programs are unrelated. For each

Program	C LOC	Java LOC	Time (s)	Errs	Warnings	False Pos	Impr
libgconf-java-2.10.1	1119	670	2.4	0	0	10	0
libglade-java-2.10.1	149	1022	6.9	0	0	0	1
libgnome-java-2.10.1	5606	5135	17.4	45	0	0	1
libgtk-java-2.6.2	27095	32395	36.3	74	8	34	18
libgtkhtml-java-2.6.0	455	729	2.9	27	0	0	0
libgtkmozembed-java-1.7.0	166	498	3.3	0	0	0	0
libvte-java-0.11.11	437	184	2.5	0	26	0	0
jnetfilter	1113	1599	17.3	9	0	0	0
libreadline-java-0.8.0	1459	324	2.2	0	0	0	1
pgpjava	10136	123	2.7	0	1	0	1
posix1.0	978	293	1.8	0	1	0	0
Total				155	36	44	22

Fig. 5. Experimental Results

program, Fig. 5 lists the number of lines of C and Java code, the analysis time in seconds (average of 3 runs), and the number of messages reported by our tool, divided manually into four categories as described below. The running time includes the C code analysis (including extracting Java types from class files) but not the parsing of C code. The measurements were performed on a 733 MHz Pentium III machine with 1GB of RAM.

Our tool reported 155 errors, which are programming mistakes that may cause a program to crash or to emit an unexpected exception. Surprisingly, the most common error was declaring a C function with the wrong arity, which accounted for 68 errors (30 in libgtk and 38 in libgnome). All C functions called from Java must start with one parameter for the JNI environment and a second parameter for the invoking object or class. In many cases the second parameter was omitted in the call, and hence any subsequent arguments would be retrieved from the wrong stack location, which could easily cause a program crash.

56 of the errors were due to mistakes made during a software rewrite. Programs that use the JNI typically use one of two techniques to pass C pointers (e.g., GUI window objects) through Java: they either pass the pointer directly as an integer, or they embed the pointer as an integer field inside a Java object. Several of the libraries in the Java-Gnome project appear to be switching from the integer technique to the object technique, which requires changing Java declarations in parallel with C declarations, an error-prone process. Our tool detected many cases when a Java native method specified an `Object` parameter but the corresponding C function specified an integer parameter, or vice-versa. This accounted for 4 errors in libgnome, 25 in libgtk, and 27 in libgtkhtml.

Type mismatches accounted for 17 of the remaining errors. 6 errors occurred because a native Java method was declared with a `String` argument, but the C code took a byte array argument. In general Java strings must be translated to C strings using special JNI functions, and hence this is a type error. Another

type error occurred because one C function passed a (non-array) Java object to another C function expecting a Java array. Since both of these are represented with the type `jobject` in C, the C compiler did not catch this error.

Finally, 14 errors were due to incorrect namings. 11 of these errors (9 in jnetfilter and 2 in libgtk) were caused by calls to `FindClass` with an incorrect string. Ironically, all 9 jnetfilter errors occurred in code that was supposed to construct a Java exception to throw—but since the string did not properly identify the exception class, the JVM would throw a `ClassNotFound` exception instead. The remaining 3 errors were due to giving incorrect names to C functions corresponding to Java native methods; such functions must be given long names following a particular naming scheme, and it is easy to get this wrong.

We also ran our tool on a development Java 1.6 compiler, code-named Mustang. Our tool produced 480 messages, one of which was an actual error in which the JNI glue code did not properly distinguish between the types `int` and `long`. If used on a big-endian 64-bit machine, the C function would access only the higher 32 bits of the value, creating a runtime error [10]. The remaining messages were all false positives or imprecision messages. This program is not present in Figure 5 because we had to play special tricks to use our tool on the source code due to its intricate bootstrapping build process and were therefore unable to calculate an accurate running time.

Most of the errors we found are easy to trigger with a small amount of code. In cases such as incorrectly-named function, errors would likely be immediately apparent as soon as the native method is called. Thus clearly many of the errors are in code that has not been tested very much, most likely the parts of libraries that have not yet been used by Java programmers.

Our tool also produced 36 warnings, which are suspicious programming practices that do not actually cause run-time errors. One warning arose when a programmer called the function `FindClass` with a field descriptor of the form `Ljava/lang/String;` rather than a fully qualified class name `java/lang/String`. Technically this is an error [5], but the Sun JVM we tested allows both versions, so we only consider this a warning. Another example that accounted for 2 warnings was due to incorrectly declaring a function to return the wrong type, but then returning the correct type in the function body.

Finally, 33 warnings were due to the declaration of C functions that appear to implement a specific native method (because they have mangled names), but do not correspond to any native Java method. In many cases there was a native method in the Java code, but it had been commented out or moved without deleting the C code. This will not cause any run-time errors, but it seems helpful to notify the programmer about this dead code.

Our tool also produced 44 false positives, which are warnings about correct code, and 22 imprecision warnings, which occurred when the analysis had insufficient information about the value of a string. All of the false positives were caused by the use of subtyping inside of C code, which our analysis does not model precisely because it uses unification. 16 of the warnings were due to unification failures with partially specified methods that could not be resolved.

The other 6 warnings occurred when the programmer called a Java method by passing arguments via an array, which our analysis does not model.

5 Related Work

In prior work, we presented a system for inferring types for programs that use the OCaml-to-C FFI [4]. Our new work on the JNI differs in several ways. To infer types for the JNI, we need to track string values through the source code, whereas for the OCaml FFI we mainly needed to track integers and pointer offsets. Our new system can directly assign polymorphic types to JNI functions and to user-written wrapper functions, in contrast to our previous system which did not track sufficient interprocedural information to allow this and was not polymorphic. Our new system also includes support for indexing into records using strings for field names and method signatures, and those strings may not be known until constraint resolution. Our previous system did not support objects.

Recently several researchers have developed sophisticated techniques to track strings in programs [11–14]. One goal of these systems is to check that dynamically-generated SQL queries are well-formed by creating a language which describes all possible strings for a given expression. For purposes of checking clients of the JNI, we found that simple tracking of strings is sufficient.

Nishimura [15] presents an object calculus which can statically infer kinded-types for first-class method names. Their system has similar restrictions to ours like not supporting inheritance or overloaded methods. Our work differs in that we are typing C code and must analyze the value of C strings instead of working with a pure object calculus.

There are many different foreign function interfaces with various design trade-offs [16–20, 5]. We believe that the techniques we develop in this paper and in our previous work [4] can be adapted to many FFIs.

An alternative to using FFIs directly is to use automatic interface generators to produce glue code. SWIG [21] generates glue code based on an interface specification file. Exu [22] provides programmers with a light-weight system for automatically generating JNI-to-C++ glue code for the common cases. Mockingbird [23] is a system for automatically finding matchings between two types written in different languages and generating the appropriate glue code. Our benchmark suite contained custom glue code that was generated by hand.

In addition to the JNI, there has been significant work on other approaches to object-oriented language interoperation, such as the commercial solutions COM [24], SOM [25] and CORBA [26]. Barret [27] proposes the PolySPIN system as an alternative to CORBA. All of these systems check for errors mainly at run-time (though in some cases interface generators can be used to provide some compile-time checking). The Microsoft common-language runtime (CLR) [28, 29] provides interoperation by being the target of compilers for multiple different languages, and the CLR includes a strong static type system. However, the type system only checks CLR code, and not unmanaged code that it may invoke.

Grechanik et al [30] present a framework called ROOF that allows many different languages to interact using a common syntax. This system includes both run-time and static type checking for code that uses ROOF. It is unclear whether ROOF supports polymorphism and whether it can infer types for glue code in isolation.

6 Conclusion

We have presented a multi-lingual type inference system for checking that programs use the JNI safely. Our system tracks the values of C strings to determine what names and type descriptors are passed to JNI functions. Thus we are able to infer what type assumptions C glue code makes about Java objects and check whether they are consistent with the actual Java class definitions. Our system treats wrapper functions and JNI functions polymorphically, allowing them to be parametric even in string arguments. Using an implementation of our system, we found many errors and suspicious practices in our suite of benchmarks. Our results suggest that our static checking system can be an important part of ensuring that the JNI is used correctly, and we believe that the same ideas can be applied to other object-oriented FFIs.

References

1. Rehof, J., Fähndrich, M.: Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In: Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages, London, United Kingdom (2001)
2. Fähndrich, M., Rehof, J., Das, M.: Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In: Proceedings of the 2000 ACM Conference on Programming Language Design and Implementation, Vancouver B.C., Canada (2000)
3. Henglein, F.: Type Inference with Polymorphic Recursion. ACM Transactions on Programming Languages and Systems **15** (1993) 253–289
4. Furr, M., Foster, J.S.: Checking Type Safety of Foreign Function Calls. In: Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation, Chicago, Illinois (2005) 62–72
5. Liang, S.: The Java Native Interface: Programmer's Guide and Specification. Addison-Wesley (1999)
6. Furr, M., Foster, J.S.: Polymorphic Type Inference for the JNI. Technical Report CS-TR-4759, University of Maryland, Computer Science Department (2005)
7. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: Compiler Construction, 11th International Conference, Grenoble, France (2002)
8. Cannasse, N.: (Ocaml javalib) `http://team.motion-twin.com/ncannasse/javaLib/`.
9. Java-Gnome Developers: (Java bindings for the gnome and gtk libraries) `http://java-gnome.sourceforge.net`.
10. Furr, M., Foster, J.S.: Java SE 6 "Mustang" Bug 6362203 (2005) `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6362203`.

11. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Static Analysis, 10th International Symposium, San Diego, CA, USA (2003)
12. DeLine, R., Fähndrich, M.: The Fugue Protocol Checker: Is your software Baroque? Technical Report MSR-TR-2004-07, Microsoft Research (2004)
13. Gould, C., Su, Z., Devanbu, P.: Static Checking of Dynamically Generated Queries in Database Applications. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK (2004) 645–654
14. Thiemann, P.: Grammar-Based Analysis of String Expressions. In: Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Language Design and Implementation, Long Beach, CA, USA (2005)
15. Ernst, S.N.: Static Typing for Dynamic Messages. In: Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages, San Diego, California (1998)
16. Blume, M.: No-Longer-Foreign: Teaching an ML compiler to speak C "natively". In: BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability, Firenze, Italy (2001)
17. Finne, S., Leijen, D., Meijer, E., Jones, S.P.: Calling hell from heaven and heaven from hell. In: Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming, Paris, France (1999) 114–125
18. Fisher, K., Pucella, R., Reppy, J.: A framework for interoperability. In: BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability, Firenze, Italy (2001)
19. Huelsbergen, L.: A Portable C Interface for Standard ML of New Jersey. `http://www.smlnj.org//doc/SMLNJ-C/smlnj-c.ps` (1996)
20. Leroy, X.: The Objective Caml system (2004) Release 3.08, `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`.
21. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: USENIX Fourth Annual Tcl/Tk Workshop. (1996)
22. Bubba, J.F., Kaplan, A., Wileden, J.C.: The Exu Approach to Safe, Transparent and Lightweight Interoperability. In: 25th International Computer Software and Applications Conference (COMPSAC 2001), Chicago, IL, USA (2001)
23. Auerbach, J., Barton, C., Chu-Carroll, M., Raghavachari, M.: Mockingbird: Flexible stub compilation from paris of declarations. In: Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA (1999)
24. Gray, D.N., Hotchkiss, J., LaForge, S., Shalit, A., Weinberg, T.: Modern Languages and Microsoft's Component Object Model. cacm **41** (1998) 55–65
25. Hamilton, J.: Interlanguage Object Sharing with SOM. In: Proceedings of the Usenix 1996 Annual Technical Conference, San Diego, California (1996)
26. Object Management Group: Common Object Request Broker Architecture: Core Specification. Version 3.0.3 edn. (2004)
27. Barrett, D.J.: Polylingual Systems: An Approach to Seamless Interoperability. PhD thesis, University of Massachusetts Amherst (1998)
28. Hamilton, J.: Language Integration in the Common Language Runtime. ACM SIGPLAN Notices **38** (2003) 19–28
29. Meijer, E., Perry, N., van Yzendoorn, A.: Scripting .NET using Mondrian. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary (2001)
30. Grechanik, M., Batory, D., Perry, D.E.: Design of large-scale polylingual systems. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, Scotland, UK (2004) 357–366