Checking Type Safety of Foreign Function Calls

MICHAEL FURR and JEFFREY S. FOSTER
University of Maryland, College Park

Foreign function interfaces (FFIs) allow components in different languages to communicate directly with each other. While FFIs are useful, they often require writing tricky, low-level code and include little or no static safety checking, thus providing a rich source of hard-to-find programming errors. In this paper, we study the problem of enforcing type safety across the OCaml-to-C FFI and the Java Native Interface (JNI). We present O-Saffire and J-Saffire, a pair of multilingual type inference systems that ensure C code that uses these FFIs accesses high-level data safely. Our inference systems use *representational types* to model C's low-level view of OCaml and Java values, and singleton types to track integers, strings, memory offsets, and type tags through C. J-Saffire, our Java system, uses a polymorphic, flow-insensitive, unification-based analysis. Polymorphism is important because it allows us to precisely model user-defined wrapper functions and the more than 200 JNI functions. O-Saffire, our OCaml system, uses a monomorphic, flow-sensitive analysis, because while polymorphism is much less important for the OCaml FFI, flow-sensitivity is critical to track conditional branches, which are used when "pattern matching" OCaml data in C. O-Saffire also tracks garbage collection information to ensure that local C pointers to the OCaml heap are registered properly, which is not necessary for the JNI. We have applied O-Saffire and J-Saffire to a set of benchmarks and found many bugs and questionable coding practices. These results suggest that static checking of FFIs can be a valuable tool in writing correct multilingual software.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages, Verification

Additional Key Words and Phrases: foreign function interface, FFI, foreign function calls, representational type, multilingual type system, multilingual type inference, flow-sensitive type system, dataflow analysis, OCaml, Java, JNI, Java Native Interface

1. INTRODUCTION

Many programming languages contain a *foreign function interface* (FFI) that allows programs to invoke functions written in other languages. FFIs are important for high-level languages, because they allow programs to access a multitude of system and user libraries that would be otherwise unavailable. Moreover, different languages make different programming idioms either harder or easier and have various performance tradeoffs, and thus the ability to write multilingual programs has important software engineering benefits.

Unfortunately, FFIs are difficult to use correctly, especially when there are mismatches between native and foreign type systems, data representations, and runtime environments. In all of the FFIs we are aware of, there is little or no consistency checking between foreign and native code [Blume 2001; Finne et al. 1999; Huelsbergen 1996; Leroy 2004; Liang 1999]. As a consequence, adding an FFI to a safe language potentially provides a rich source of operations that can violate safety in subtle and difficult-to-find ways.

In this paper, we present O-Saffire and J-Saffire[1], a pair of type inference systems that ensure type safety across the OCaml FFI and the Java Native Interface (JNI), respectively. In these FFIs, both of which interface to C, most of the work is done in C "glue code," which translates data between the high-level language and C, and then invokes other routines, often in system or user libraries. It is easy to make mistakes in glue code, both because it is low-level and because the C compiler does not check that glue code accesses high-level data at the right types. For example, we found OCaml FFI glue code that uses integers as pointers and vice-versa, and JNI glue code that accesses non-existent classes. These errors can result in silent memory corruption or run-time exceptions, and they are often extremely difficult to debug.

O-Saffire and J-Saffire detect these kinds of problems by performing constraint-based type inference on C glue code. The first challenge in analyzing glue code is tracking types from OCaml or Java through C. In both FFIs, all high-level types are conflated to a single type in C, either `value` for OCaml or `jobject` for Java. Thus we perform inference by extending these to *multilingual types* of the form mt `value` and jt `jobject`. Here mt is an OCaml type and jt is a Java type. O-Saffire and J-Saffire compare the inferred types in glue code to type information extracted from OCaml or Java, and report any inconsistencies to the user.

As it turns out, C glue code has a richer view of types than is available in the high-level language. Thus the types mt and jt above are actually *representational types* that model C's low-level view of OCaml and Java data. In particular, OCaml glue code can observe that many high-level types have the same physical representation. For example, the value of OCaml type `unit` has the same representation as the OCaml integer 0, nullary OCaml data constructors are represented using integers, and OCaml records and tuples can be silently injected into sum types if they have the right dynamic tag. Thus our representational types model OCaml data as tagged unions of primitive and pointer types of various shapes. JNI glue code cannot manipulate Java objects directly, but it accesses fields and methods by specifying their names and types with C strings, and thus JNI glue code can be polymorphic in ways not allowed in Java except with reflection. Our representational types model Java types using strings, which may be quantified variables, to label classes, fields, and methods.

O-Saffire and J-Saffire use similar core ideas but are structured slightly differently, because the OCaml FFI and the JNI provide different data access mechanisms. OCaml glue code directly manipulates OCaml data using integer arithmetic and pointer operations, and can form pointers into the middle of OCaml records and tuples. Moreover, since C (unsurprisingly) provides no linguistic support for pattern matching OCaml data types, C glue code must explicitly perform dynamic type tag tests. To model these operations, O-Saffire uses an iterative, intraprocedural, flow-sensitive dataflow analysis to track integer values, represented with singleton types, as well as offset and tag information. Our dataflow analysis is fairly simple, which turns out to be sufficient in practice because most programs use the FFI in a simple way, in part to avoid making mistakes.

The OCaml FFI also gives C low-level control over references to the OCaml heap,

[1]Saffire = Static Analysis of Foreign Function InteRfacEs

which is managed by the OCaml garbage collector. To avoid memory corruption, before a C program calls OCaml (which might invoke the garbage collector), it must notify the OCaml runtime system of any pointers it has to the OCaml heap. This is easy to forget to do, especially when the OCaml runtime is called indirectly. O-Saffire uses *effects* to track functions that may invoke the OCaml garbage collector, and O-Saffire ensures that pointers to the OCaml heap are registered as necessary.

In contrast, JNI glue code must invoke special functions to manipulate `jobject`s, which are opaque, and pointers to the Java heap are automatically registered with the garbage collector. JNI functions take as arguments specially-formatted strings that identify class, field, and method names as well as their types. We have found that JNI glue code uses strings fairly directly, e.g., string constants are passed around the program without manipulating them (e.g., via substring or concatenation). Thus J-Saffire uses flow-insensitive unification to track string values, which it represents with singleton types.

We have found that, unlike OCaml glue code, JNI glue code often contains wrapper functions that group together common operations, and the Java types used by these functions depend on the strings passed in by callers. Thus representational types inferred by J-Saffire can model 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. Moreover, J-Saffire performs polymorphic type inference, including polymorphism in types representing string values, which allows J-Saffire to precisely analyze wrapper functions and directly assign universal type signatures to the more than 200 functions in the JNI.

We have proven that restricted versions of O-Saffire and J-Saffire are sound, modulo certain features of C such as out-of-bounds array accesses and type casting.

We have implemented both O-Saffire and J-Saffire and applied them to a set of 11 and 12 benchmarks respectively. In our experiments, we found many outright errors (24 for O-Saffire and 156 for J-Saffire) and suspicious but non-fatal programming mistakes (22 for O-Saffire and 124 for J-Saffire). We have reported all of the errors to developers, and they have been confirmed. Both O-Saffire and J-Saffire run efficiently in practice, usually takes only a few second for analysis.

As far as we are aware, ours is the first work that attempts to check richer properties on the foreign language side between two general-purpose programming languages, and we believe that our core ideas are applicable to other FFIs as well. Our results suggest that multilingual type inference is a beneficial, practical addition to a language with a foreign function interface.

In summary, the main contributions of this paper are:

—We develop *multilingual, representational types* that embed OCaml and Java type information into C in a way that matches C's low-level view of high-level data. As a result, we are able to check that C glue code uses high-level data and functions at the right type.

—We present multilingual type inference systems O-Saffire and J-Saffire for the OCaml FFI and the JNI, respectively. Our type systems use singleton types to track the values of integers and strings through C glue code. We infer these types either using standard unification, which is sufficient for the JNI, or using

a flow-sensitive dataflow analysis, which tracks integer values, offset, and tag information for the OCaml FFI. Additionally, O-Saffire uses effects to ensure that garbage collector invariants are obeyed in the foreign language, and J-Saffire supports polymorphism for JNI functions and user-defined wrapper functions.

—We show that restricted versions of O-Saffire and J-Saffire are sound, so that multilingual programs that pass our type systems will not violate type safety.

—We describe implementations of O-Saffire and J-Saffire, along with experiments in which we apply our implementations to a number of benchmarks. We found many bugs and questionable coding practices as a result.

O-Saffire and J-Saffire were both presented separately in earlier conference versions [Furr and Foster 2005a; 2006b]. The current paper gives a more cohesive presentation of both systems, comparing and contrasting them as they are developed. This paper also includes checking versions of O-Saffire and J-Saffire, which were omitted from the conference papers, and a more algorithmic presentation of type inference. We include sketches of our soundness theorems, including a detailed operational semantics; these theorems were only stated without proof and without semantics in the conference versions. Our discussion of the JNI inference system includes more details on methods and polymorphism. Finally, we include a new discussion section that explores the relationship between the two systems, discusses the applicability of these ideas to other FFIs, and sketches future work.

2. BACKGROUND

In a typical use of an FFI, the high level language invokes a C routine, which in turn invokes a system or user library routine. The C routine usually contains "glue" code to manipulate data from the high-level language and translate between the different data representations of the two languages. Glue code is structured somewhat differently for OCaml and Java, since each provides a different view of foreign data types. OCaml's FFI exposes a very low level view of OCaml types, corresponding to exactly how they are represented in memory. To use the OCaml FFI, the C programmer must directly manipulate OCaml values using bit-shifting operations and pointer dereferences. OCaml provides C macros for these purposes, but their use is not mandatory and does not provide any safety checking. The JNI, on the other hand, provides a more opaque interface. Java objects are modeled as pointers that must be passed to JNI functions to be manipulated, and classes, fields, methods are all described using strings. The JNI includes over 200 functions, and it is easy to call the wrong function or to make typos in string-valued parameters, neither of which produces a compiler warning. We begin by discussing how each interface is typically used by a C programmer.

2.1 The OCaml FFI

Fig. 1 shows the basic OCaml and C source language types. OCaml types include unit and int types, product types (records or tuples), and sum types. Sums are composed of type constructors S, which may optionally take an argument. OCaml also includes types for updatable references and functions. Other OCaml types such as objects and polymorphic variants are not supported by our system; see Section 5.1 for a discussion. C includes types `void`, `int`, pointer types constructed

$$
\begin{array}{rcl}
\mathit{mltype} & ::= & \mathsf{unit} \mid \mathsf{int} \mid \mathit{mltype} \times \mathit{mltype} \mid S + \cdots + S \\
& \mid & \mathit{mltype}\ \mathsf{ref} \mid \mathit{mltype} \rightarrow \mathit{mltype} \\
S & ::= & \mathit{Constr} \mid \mathit{Constr}\ \mathsf{of}\ \mathit{mltype}
\end{array}
$$

(a) OCaml Type Grammar

$$
\begin{array}{rcl}
\mathit{ctype} & ::= & \texttt{void} \mid \texttt{int} \mid \mathit{ctype}\ \texttt{*} \mid \mathit{ctype} \times \ldots \times \mathit{ctype} \rightarrow \mathit{ctype} \\
& \mid & \texttt{value}
\end{array}
$$

(b) C Type Grammar

Fig. 1. OCaml and C source type languages

with postfix `*`, and functions. C also includes the type `value`, to which all OCaml data is assigned.

To invoke a C function called *c_name*, an OCaml program must contain a declaration of the form

$$\mathsf{external}\ \ f : \mathit{mltype} = \text{“}\mathit{c\_name}\text{”}$$

where *mltype* is an OCaml function type. When the OCaml program calls f, the OCaml runtime invokes the corresponding C function declared as

```
value c_name(value arg1, ..., value argn);
```

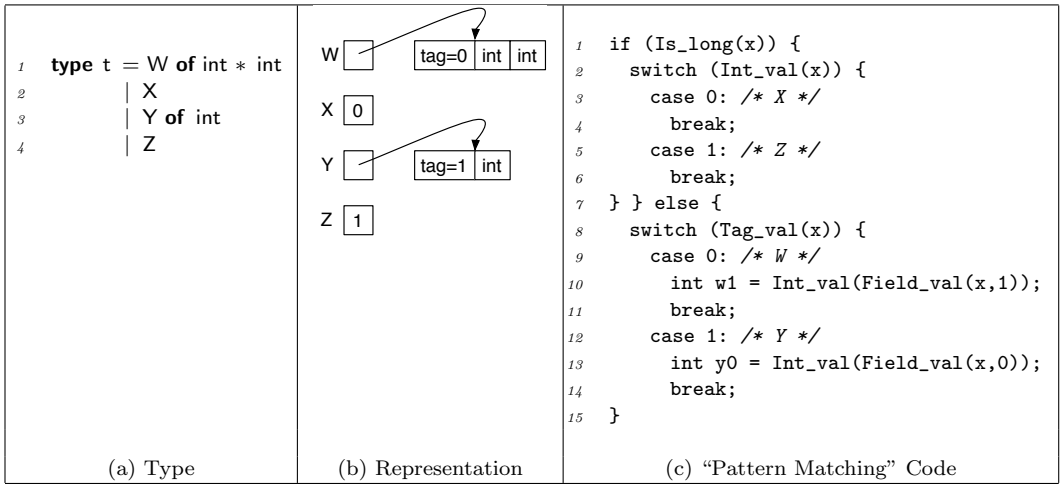
Although different OCaml types have different physical representations, there is no protection in C from mistakenly using data at the wrong type. As an example, consider the OCaml sum type declaration shown in Fig. 2(a). This type has nullary (no-argument) constructors X and Z and non-nullary constructors W and Y.

Each nullary constructor in a sum type is numbered from 0 and is represented in memory directly as that integer, as shown in Fig. 2(b). Thus to C functions, nullary constructors look just like OCaml ints, e.g., X:t and 0:int are identical. Additionally, the value of type unit is also represented by the OCaml integer 0.

The low-order bit of such *unboxed* values is always set to 1 to distinguish them from pointers. C routines use the macro `Val_int` to convert to tagged integers and `Int_val` to convert back. There are no checks, however, to ensure that these macros are used correctly or even at all. In particular, in the standard OCaml distribution the type `value` is a typedef (alias) of `long`. Thus one could mistakenly apply `Int_val` to a pointer, or apply `Val_int` to a `value`. We found several examples of these sorts of mistakes in our experiments.

Each non-nullary constructor in a sum type is also numbered separately from 0. These constructors are represented as *boxed* values or pointers to *structured blocks* on the heap. A structured block is an array of `value`s preceded by a header that contains, among other things, a *tag* with the constructor number. Fig. 2(b) shows the representations of W and Y for our example type t. Products that are not part of a sum are represented as structured blocks with tag 0. For example, W (1,2) and (1,2) both have the same representation.

Boxed values are manipulated using the macro `Field(`x`,`i`)`, which expands to `*((value*)`x`+`i`)`, i.e., it accesses the ith element in the structured block pointed to by x. There are no checks to prevent a programmer from applying `Field` to



(a) Type (b) Representation (c) "Pattern Matching" Code

Fig. 2. Example OCaml type

an unboxed `value` (notice the type cast in the macro expansion) or from accessing past the end of a structured block.

OCaml provides several macros for testing tags and for checking boxedness of a `value`. We illustrate these tests in Fig. 2(c), which contains glue code to perform "pattern matching" on a C value `x` of OCaml type `t`. The macro `Is_long()` on line 1 checks whether `x` is a pointer by examining its low-order bit. If it is unboxed, `Int_val()` on line 2 is used to extract the tag, and thus on line 3 we know `x` is X, and on line 5 we know it is Z. Otherwise, we know on line 8 that `x` is boxed, and so `Tag_val()` is used to extract the tag from the structured block. Then on line 10, since `x` must be constructed from W, we may access the second field of the structured block, and on line 13, `x` is Y, and so we may access its first field.

Notice that because C has no linguistic support for OCaml pattern matching, what would be a simple match expression in OCaml has been expanded into its constituent parts, with all of the dynamic tag tests explicitly coded by the programmer. This is a common phenomenon in FFIs, where operations that have direct syntactic support in one language must be clumsily reproduced in the other, and we will see a similar phenomenon in the JNI. Analyzing this kind of "pattern matching" code is the principal challenge in type inference for OCaml glue code.

In addition to using OCaml data at the correct type, C FFI functions that call the OCaml runtime must notify the garbage collector of any C pointers to the OCaml heap. To do so, C functions use macros `CAMLparam` and `CAMLlocal` to register parameters and locals, respectively. If a function registers any such pointers, it must call `CAMLreturn` upon exiting to release the pointers. We have found in our experiments that it is easy to forget to use these macros, and discovering these errors with testing would be difficult.

2.2 The Java FFI (JNI)

Fig. 3 shows the basic Java source language types, along with the C types used by the FFI. Java includes void and integer types as well as class types that contain fields

$$
\begin{array}{rcl}
\mathit{jtype} & ::= & \mathsf{void} \mid \mathsf{int} \mid \mathsf{class}\ \mathit{id}\ \{\mathit{field}^*\ \mathit{method}^*\} \\
\mathit{field} & ::= & \mathit{id} : \mathit{jtype} \\
\mathit{method} & ::= & \mathit{id} : \mathit{jtype} \times \cdots \times \mathit{jtype} \to \mathit{jtype}
\end{array}
$$

(a) Java Type Grammar

$$
\begin{array}{rcl}
\mathit{ctype} & ::= & \texttt{void} \mid \texttt{int} \mid \mathit{ctype}\ \texttt{*} \mid \mathit{ctype} \times \ldots \times \mathit{ctype} \to \mathit{ctype} \\
& \mid & \texttt{jobject} \mid \texttt{jfieldID} \mid \texttt{jmethodID}
\end{array}
$$

(b) C Type Grammar

Fig. 3. Java and C source type languages

and methods. The base C type language is as before, but this time it is extended to include the type `jobject` for Java objects, and `jfieldID` and `jmethodID`, which identify fields and methods in the JNI.

Unlike the OCaml FFI, which exposes OCaml data representations to C, users of the JNI do not directly manipulate Java data. However, using the JNI is still complicated, and it is easy to make many kinds of mistakes. To call a C function from Java, the programmer first declares a Java method with the native keyword and no body, for example:

```
package edu.cs.umd;
class Foo { native void bar(int x, float y); }
```

When the native method bar is invoked, the Java runtime finds and invokes the correspondingly-named C function. Unlike OCaml, the C function name is not up to the programmer, but is strictly specified by the JNI. To avoid ambiguity, the name includes not only the Java method name, but also its enclosing class and package. If the method is overloaded, then the types of the parameters are appended to its name using field descriptors, which are discussed below. For our example, assuming bar is overloaded, the corresponding C function must be

```
void Java_edu_cs_umd_Foo_bar__IF(jint x, jfloat y);
```

As this example shows, the C function names can be long and cryptic and are easy to misspell, and as the JVM dynamically loads C libraries at runtime, there is no check during compilation or linking that these names are correctly specified.

Unlike in the OCaml FFI, Java and C share the same representation for primitive types such as integers and floating point numbers. Thus C glue code requires no special support to manipulate them—the types `jint` and `jfloat` used above are simply typedefs of `int` and `float`. 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");
```

[2]The JNI functions discussed in this section are actually invoked slightly differently and take an additional parameter, as discussed in Section 5.3.

```
jobject my_getObjectField(jobject obj, char *field) {
  jobject cls = GetObjectClass(obj);
  jfieldID fid = GetFieldID(cls, field, "java/lang/Object");
  return GetObjectField(obj, fid);
}
```

Fig. 4. JNI wrapper function example

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* [Lindholm and Yellin 1997]. Other examples are `F` for float, as we saw earlier, `[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`, which omits the initial `L` and trailing semicolon. Our implementation enforces this difference, but we omit it from our 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 JVM: getting the type of the object, finding the offset of the field, and retrieving its contents. This is similar to what happens with OCaml pattern matching in C, which is also expanded out into its constituent parts in glue code.

Moreover, 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. First the programmer calls `GetMethodID`, which accepts a string that encodes a list of parameter types to retrieve a `methodID`. Then they call the appropriate dispatch function, `Call<Type>Method`, where `<type>` is the return type of the method. Like the field functions above, there is one such function for each primitive Java type and one for objects.

One common pattern we have seen in JNI code is wrapper functions that specialize JNI routines to particular classes, fields, or methods. Fig. 4 shows an example

wrapper function `my_getObjectField` that extracts a field of type Object 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). The contents of the field are extracted via a call to `GetObjectField`, which behaves the same as `GetIntField` but returns a Java object. Calling `my_getObjectField` is safe if the first parameter has an Object field whose name is given by the second parameter. Thus this function is parameterized by the object from which to extract the field 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, to precisely analyze this code we need polymorphism, not only in the types of objects but also in the values of string parameters. Note that our experience with OCaml glue code shows that it most often is not polymorphic, and thus our OCaml FFI type inference system is monomorphic.

Finally, like OCaml, Java is a garbage collected language. However, unlike OCaml, Java does not require the C programmer to register all local C references to Java objects. Instead, since all such references are returned from JNI function calls, they are automatically registered until the C code returns execution to the JVM. An early release mechanism is also available. Thus the JNI provides a layer of safety to the C programmer in exchange for (possibly) less efficient garbage collection while in C code.

2.3 Saffire

As the previous discussion illustrates, using an FFI correctly requires careful programming, and there is little or no compiler support for preventing errors. The goal of Saffire is to address this problem by providing compile-time type checking across these FFIs. In both the OCaml FFI and the JNI, most of the work occurs in C glue code rather than in the high-level language, and so the focus of Saffire is on analyzing glue code. There are several major challenges:

—In these FFIs, most or all foreign types are conflated to only one C type, either `value` for OCaml or `jobject` for Java. Thus our first step is to develop an extended type system to make the high-level language types available to our analysis of glue code. In particular, Saffire will infer types of the form *mt* `value` and *jt* `jobject`, where *mt* and *jt* are OCaml and Java types, respectively.

—C glue code can use data in ways either not possible or not common in the high-level language. For instance, C glue code can freely use the OCaml integer 0 where types unit, int, or an unboxed nullary constructor are expected, and the "pattern matching" code in Fig. 2(c) can be applied to any data type with the same structure, even if it has a different name (or, in fact, even if it has fewer constructors). In the JNI, the wrapper function in Fig. 4 can be applied to any object with the appropriate integer field, which could only be achieved with reflection within Java. Our solution is to make types *mt* and *jt* richer than source-level types *mltype* and *jtype*, so that we can model C's view of the high-level language types directly.

—Determining how C code uses data from the high-level language requires tracking detailed information that depends on the particular FFI programming idioms. For example, to analyze the "pattern matching" code in Fig. 2(c), we need to

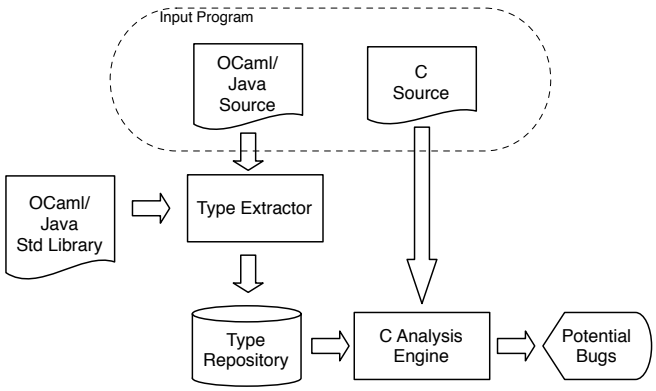


Fig. 5. Architecture of O-Saffire and J-Saffire

track the values of integers and model conditional branches precisely, and to compute the type of `Field(`x`,`i`)` we need to know the value of pointer offset i. On the other hand, the JNI instead uses strings to find classes, fields, and methods, and tends to use polymorphic wrapper functions more. Our approach is to use the appropriate style of analysis for each system. For the OCaml FFI, we use a monomorphic data flow analysis to track integers and pointers, and for the JNI we use a polymorphic unification-based analysis to track strings.

In the next sections, we present two type inference systems, O-Saffire for the OCaml FFI (Section 3) and J-Saffire for the JNI (Section 4). Fig. 5 gives a overview of the architecture of the systems, which is the same for both. The input is a program written in OCaml and C or Java and C. We begin by extracting high-level type information from the source code and standard library code, and we create a database containing all of the high-level language types. (Our implementation of J-Saffire actually creates this database on demand, but the concept is the same.) Since OCaml and Java are both type and memory safe, we know that these types are used correctly in the high-level source code. We then take the C source code and analyze it, inferring the shapes of the high-level types used by the C code. Then we compare the inferred types against the types declared in OCaml and Java in the type repository, and issue warnings if there are any type errors. Note that we need to perform inference rather than pure checking to account for intermediate functions that are not directly called from the high-level language. For the OCaml FFI, we also check that pointers to the OCaml heap are registered with the garbage collector as we perform type inference.

3. O-SAFFIRE: TYPE INFERENCE FOR THE OCAML FFI

In this section, we present O-Saffire, our multilingual type inference system for the OCaml FFI. As discussed in Section 2.3, O-Saffire has two phases: first we extract type information from OCaml, and then we perform type inference on glue code to find any inconsistencies with the extracted OCaml types. We begin our presentation with the second phase, first developing our multilingual type language (Section 3.1) and a type checking system for glue code (Section 3.2), and then

$$
\begin{array}{rcl}
e & ::= & n \mid \texttt{*}e \mid e \ aop \ e \mid lval \mid e \texttt{ +}_p e \mid \texttt{Val\_int } e \mid \texttt{Int\_val } e \\
lval & ::= & x \mid \texttt{*}(e_1 +_p e_2) \\
aop & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{==} \mid \cdots \\
s & ::= & L\texttt{:}\ s \mid s\ \texttt{;}\ s \mid lval \texttt{ := } e \mid lval \texttt{ := } f(e, \ldots, e) \mid \texttt{goto } L \\
 & \mid & \texttt{if } e \texttt{ then } L \mid \texttt{if\_unboxed}(x) \texttt{ then } L \\
 & \mid & \texttt{if\_sum\_tag}(x) == n \texttt{ then } L \\
 & \mid & \texttt{if\_int\_tag}(x) == n \texttt{ then } L \\
 & \mid & \texttt{return } e \mid \texttt{CAMLreturn}(e) \\
d & ::= & ctype\ x = e \mid \texttt{CAMLprotect}(x) \\
f & ::= & \texttt{function } ctype\ f(ctype\ x, \ldots, ctype\ x)\ d^*\ s \\
 & \mid & \texttt{function } ctype\ f(ctype\ x, \ldots, ctype\ x) \\
\mathcal{P} & ::= & f^*
\end{array}
$$

Fig. 6. Simplified C grammar

proving it sound (Section 3.3). Finally, we present the type extraction process and give an algorithmic presentation of type inference (Section 3.4). Sections 5.1 and 5.2 discuss our implementation and experimental results.

Source Language. Fig. 6 presents the C-like language we use for glue code in our formal system. This language is based on the intermediate representation of CIL [Necula et al. 2002], which we used in our implementation. In this language, expressions e are side-effect free. We include integers n, pointer dereferences $*e$, as well as the usual arithmetic operators. L-values $lval$ are the restricted subset of expressions that can appear on the left-hand side of an assignment, namely variables x and pointer dereferences.

Expressions include pointer arithmetic $e_1 \texttt{ +}_p e_2$ for computing the address of offset e_2 from the start of the structured block pointed to by e_1. In actual C source code, pointer arithmetic can be distinguished from other forms using standard C type information. Our type system currently only supports pointer arithmetic on `value` types, since our focus is on FFI safety, not general C type safety, which can be addressed with other techniques [Chandra and Reps 1999; Necula et al. 2002]. Our formal system allows `value`s to be treated directly as pointers, though in C they must first be cast to `value *`. We also include the `Val_int` and `Int_val` conversion functions as primitives. Note that we omit the address-of operation `&`. Variables whose addresses are taken are treated as globals by the implementation, and uses of `&` that interact with `*` can be eliminated. Finally, the code in our benchmark suite rarely stores foreign data directly into C `struct` types, and thus we omit C structures from our formal system for simplicity. Implementation details about these heuristics and omitted C constructs are discussed in Section 5.1.

In our source language, statements s can be associated with a label L, and sequencing is written with semicolon. We also have assignment statements $lval \texttt{ := } e$ and $lval \texttt{ := } f(e, \ldots, e)$, the latter of which stores in $lval$ the result of invoking function f with the specified arguments. A branch $\texttt{goto } L$ unconditionally jumps to the statement labeled L; we assume that labels are unique within a function, and jumping across function boundaries is not allowed. A conditional branch $\texttt{if } e \texttt{ then } L$

jumps to the statement labeled L if the integer e evaluates to a non-zero number. Loop constructs and switch statements are omitted because they can be transformed into `if` and `goto` statements.

We include three primitive conditional tests for inspecting a `value` at run time. The conditional `if_unboxed`(x) checks to see whether x is not a pointer, i.e., its low-order bit is 1. The conditional `if_sum_tag`(x) tests the runtime tag of a structured block pointed to by x, and `if_int_tag`(x), tests the runtime value of unboxed variable x. In actual C source code, these tests are made by applying `Is_long`, `Tag_val`, or `Int_val`, respectively, and then checking the result.

Statements also include `return` e, which exits the current function and returns the value of e. The special form `CAMLreturn` is used for returning from a function and releasing all variables registered with the garbage collector. This statement should be used in place of `return` if and only if local variables have been registered by `CAMLprotect`, our formalism for `CAMLlocal` and `CAMLparam`. We restrict occurrences of `CAMLprotect` to the top of a function so that the set of registered variables is constant throughout the body of a function.

Programs $\mathcal{P}$ consist of a sequence of function declarations and definitions f. We omit global variables, and our implementation forbids (via a warning message) `value`s from being stored in them and does not check if they are correctly registered with the garbage collector. We assume all local variables are defined at the top-level of the function.

3.1 Multilingual Types

We begin our discussion by developing multilingual, representational types to model C's view of OCaml data. Fig. 7(a) presents the basic, flow-insensitive types in our multilingual type language, which integrates and generalizes the basic OCaml and C types in Fig. 1. Our grammar for C types `ct` embeds extended OCaml types mt in the type `value`, so that we can track OCaml type information through C. Additionally, we augment function types with an effect GC to track which functions may cause a garbage collection, as discussed below. Our grammar for OCaml types mt includes type variables α, which we treat monomorphically and will be solved for in our inference system, as well as function types.

All of the other OCaml types from Fig. 1(a)—unit, int, products, sums, and references—are modeled with a *representational type* (Ψ, Σ). In this type, Ψ describes the unboxed values the type may represent. For a sum type, Ψ is an exact value n counting the number of nullary constructors of the sum. Integers have the same physical representation as nullary constructors but could have any value, so for integers Ψ is $\top$. In our inference algorithm, Ψ may also be a variable ψ to be solved for. The Σ component of a representational type describes its possible boxed values, if any, i.e., it describes the shapes of the structured blocks the type represents. Σ is a sequence of products Π, one for each non-nullary constructor of the type. The position of each Π in the sequence corresponds to its constructor tag number, and each Π itself contains the types of the elements of the structured block. For example, the OCaml type `t` in Figure 2(a) has representational type

$$(2,\ \ (\top, \emptyset) \times (\top, \emptyset) \times \emptyset\ \ +\ \ (\top, \emptyset) \times \emptyset\ \ +\ \ \emptyset)$$

Here, $\Psi = 2$ since `t` has two nullary constructors (X and Z), and Σ contains two

$$
\begin{array}{rcll}
\texttt{ct} & ::= & \texttt{void} \mid \texttt{int} \mid mt\ \texttt{value} \mid \texttt{ct}\ \texttt{*} & \textit{(C types)} \\
& \mid & \texttt{ct} \times \cdots \times \texttt{ct} \rightarrow_{GC} \texttt{ct} & \\
mt & ::= & \alpha \mid mt \rightarrow mt \mid (\Psi, \Sigma) & \textit{(ML types)} \\
\Psi & ::= & \psi \mid n \mid \top & \textit{(unboxed value types)} \\
\Sigma & ::= & \sigma \mid \emptyset \mid \Pi + \Sigma & \textit{(sum types)} \\
\Pi & ::= & \pi \mid \emptyset \mid mt \times \Pi & \textit{(product types)} \\
GC & ::= & \gamma \mid \mathsf{gc} \mid \mathsf{nogc} & \textit{(GC effects)}
\end{array}
$$

(a) Flow-Insensitive Types

$$
\begin{array}{rcll}
B & ::= & \mathsf{boxed} \mid \mathsf{unboxed} \mid \top \mid \bot & \textit{(boxedness)} \\
I & ::= & n \mid \top \mid \bot & \textit{(pointer offset)} \\
T & ::= & n \mid \top \mid \bot & \textit{(tag/value)}
\end{array}
$$

(b) Flow-Sensitive Components of $\texttt{ct}\{B, I, T\}$

Fig. 7. OCaml/C multilingual type language

product types, the integer pair type $(\top, \emptyset) \times (\top, \emptyset) \times \emptyset$ for W, and the boxed integer type $(\top, \emptyset) \times \emptyset$ for Y.

Notice in Fig. 2(c) that our C code to examine a value of type `t` does not by itself fully specify the type of `x`. For example, the type could have another nullary constructor or non-nullary constructor that is not checked for. Thus our grammars for Σ and Π include variables σ and π that range over sums and products [Rémy 1989], which we use to allow sum and product types to grow during inference. Only when an inferred type is unified with an OCaml type can we know its size exactly.

Our type language also annotates each function type with a garbage collection effect GC, which can either be a variable γ (used in inference), gc if the function may invoke the OCaml runtime (and thus the garbage collector), or nogc if it definitely will not. GC naturally forms the two-point lattice with order $\mathsf{nogc} \sqsubseteq \mathsf{gc}$. Note that we reserve $\leq$ for the total ordering over the integers and use $\sqsubseteq$ for other partial orders. O-Saffire ensures that all necessary variables are registered before calling a function with effect gc.

Flow-Sensitive Types. Recall the example code in Fig. 2(c) for testing the tags of a `value`. In order to analyze such a program, we need to track information that may differ at each program point—for example, on each of the branches after a boxedness test, we know whether or not the tested value is boxed. Thus, O-Saffire extends the basic types `ct` to types of the form $\texttt{ct}\{B, I, T\}$, where B tracks boxedness (i.e., the result of `if_unboxed`), I tracks an offset into a structured block, and T tracks the type tag of a structured block or the value of an integer. In O-Saffire, B, I, and T are flow-sensitive, while `ct` is flow-insensitive.

Fig. 7(b) gives grammars for B, I, and T, which form lattices with the orders $\bot \sqsubseteq \mathsf{boxed} \sqsubseteq \top$ and $\bot \sqsubseteq \mathsf{unboxed} \sqsubseteq \top$ for B, and $\bot \sqsubseteq n \sqsubseteq \top$ for I and T. In our type rules, $\top$ is used for an unknown type and $\bot$ is used for unreachable code, for example following an unconditional branch. We extend arithmetic on integers to offsets I as $\top\ aop\ n = \top$, $\bot\ aop\ I = \bot$, and similarly for T. We also extend the lattice orders to types $\texttt{ct}\{B, I, T\}$ in the natural way. We define $\texttt{ct}\{B, I, T\} \sqsubseteq \texttt{ct}'\{B', I', T'\}$ if $\texttt{ct} = \texttt{ct}'$, $B \sqsubseteq B'$, $I \sqsubseteq I'$, and $T \sqsubseteq T'$. We use $\sqcup$

to denote the least upper bound operator, and we extend $\sqcup$ to types $\texttt{ct}\{B, I, T\}$ similarly. Notice that B, I, and T do not appear in the grammar for `ct` in Fig. 7, and thus we do not track them for values stored in the heap. In our experience, this is sufficient in practice.

The meaning of $\texttt{ct}\{B, I, T\}$ depends on `ct`. If `ct` is `value`, then B represents whether the data is boxed or unboxed. If B is `unboxed`, then T represents the value of the data (which is either an integer or nullary constructor), and I is always 0. For example, on line 3 of Fig. 2(c), `x` has type $\texttt{ct}\{\texttt{unboxed}, 0, 0\}$, since it represents the constructor `X`. If B is `boxed`, then T represents the tag of the structured block and I represents the offset into the block. For example, on line 12 of Fig. 2(c), `x` has type $\texttt{ct}\{\texttt{boxed}, 0, 1\}$ since it represents constructor `Y`.

Otherwise, if `ct` is `int`, then B is $\top$, I is 0, and T tracks the value of the integer, either $\bot$ for unreachable code, a known integer n, or an unknown value $\top$. For example, the C integer 5 has type $\texttt{int}\{\top, 0, 5\}$. Finally, for all other `ct` types, $B = T = \top$ and $I = 0$.

We say that a `value` is *safe* if it is either unboxed or a pointer to the first element of a structured block, and we say that any other `ct` that is not `value` is also safe. In O-Saffire, data with a type where $I = 0$ is safe. A safe `value` can be used directly at its type, and for boxed types the header can be checked with the regular dynamic tag tests. This is not true of a `value` that points into the middle of a structured block. O-Saffire only allows offsets into OCaml data to be calculated locally within a function, and so we require that any data passed to another function or stored in the heap is safe. Since all data passed from OCaml is safe, this property also holds for functions called directly from OCaml.

3.2 Type Checking Glue Code

We now present O-Saffire's type checking rules for C glue code. In our type checking rules, we assume we have concrete multilingual type information for the program, meaning that types in general have no variables (e.g., sum and product types all end with $\emptyset$). We divide our type rules into those for expressions and those for statements.

Type Checking for Expressions. Fig. 8 give our type checking rules for expressions. These rules include type environments Γ, which map variables to types $\texttt{ct}\{B, I, T\}$, and a *protection set* P, which contains those variables that have been registered with the garbage collector by `CAMLprotect`. Our rules for expressions prove judgments of the form $\Gamma, P \vdash e : \texttt{ct}\{B, I, T\}$, meaning that in type environment Γ and protection set P, the C expression e has type `ct`, boxedness B, offset I, and tag/value T.

We discuss the rules briefly. In all of the rules, we assume that the program is correct with respect to the standard C types, and that full C type and `ct` type information is available. Thus some of the rules apply to the same source construct but are distinguished by the types of subexpressions.

The rule (Int Exp) gives integers the appropriate type, and (Var Exp) looks up the type of a variable in the type environment. (AOP Exp) performs the operation *aop* on T and T' in the types. (Val Deref Exp) extracts a field from a structured block. Here e must have a known tag m and offset n, so that we can determine the

$\boxed{\Gamma, P \vdash e : \texttt{ct}\{B, I, T\}}$

$$\frac{}{\Gamma, P \vdash n : \texttt{int}\{\top, 0, n\}}\ (\textsc{Int Exp})$$

$$\frac{x \in dom(\Gamma)}{\Gamma, P \vdash x : \Gamma(x)}\ (\textsc{Var Exp})$$

$$\frac{\Gamma, P \vdash e_1 : \texttt{int}\{\top, 0, T\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, T'\}}{\Gamma, P \vdash e_1\ aop\ e_2 : \texttt{int}\{\top, 0, T\ aop\ T'\}}\ (\textsc{AOP Exp})$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\texttt{boxed}, n, m\} \quad mt = (\Psi, \Pi_0 + \cdots + \Pi_m + \cdots + \emptyset) \quad \Pi_m = mt_0 \times \cdots \times mt_n \times \cdots \times \emptyset}{\Gamma, P \vdash *e : mt_n\ \texttt{value}\{\top, 0, \top\}}\ (\textsc{Val Deref Exp})$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, 0, \top\} \quad mt = (0, \Pi_0 + \emptyset) \quad \Pi_0 = mt_0 \times \cdots \times \emptyset}{\Gamma, P \vdash *e : mt_0\ \texttt{value}\{\top, 0, \top\}}\ (\textsc{Val Deref Boxed Exp})$$

$$\frac{\Gamma, P \vdash e : \texttt{ct}\ *\{\top, 0, \top\}}{\Gamma, P \vdash *e : \texttt{ct}\{\top, 0, \top\}}\ (\textsc{C Deref Exp})$$

$$\frac{\Gamma, P \vdash e_1 : \texttt{ct}\ *\{\top, 0, \top\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, 0\}}{\Gamma, P \vdash e_1 +_p e_2 : \texttt{ct}\ *\{\top, 0, \top\}}\ (\textsc{Add C Exp})$$

$$\frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\texttt{boxed}, n, n'\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (\Psi, \Pi_0 + \cdots + \Pi_{n'} + \cdots + \emptyset) \quad \Pi_{n'} = mt_0 \times \cdots \times mt_{n+m} \times \cdots \times \emptyset}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, n + m, n'\}}\ (\textsc{Val Add Exp})$$

$$\frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\top, 0, \top\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (0, \Pi_0 + \emptyset) \quad \Pi_0 = mt_0 \times \cdots \times mt_m \times \cdots \times \emptyset}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, m, 0\}}\ (\textsc{Val Add Boxed Exp})$$

$$\frac{\Gamma, P \vdash e : \texttt{int}\{\top, 0, T\} \qquad T + 1 \leq \Psi}{\Gamma, P \vdash \texttt{Val\_int}\ e : (\Psi, \Sigma)\ \texttt{value}\{\texttt{unboxed}, 0, T\}}\ (\textsc{Val Int Exp})$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\texttt{unboxed}, 0, T\}}{\Gamma, P \vdash \texttt{Int\_val}\ e : \texttt{int}\{\top, 0, T\}}\ (\textsc{Int Val Exp})$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, 0, T\} \qquad mt = (\Psi, \emptyset)}{\Gamma, P \vdash \texttt{Int\_val}\ e : \texttt{int}\{\top, 0, T\}}\ (\textsc{Int Val Unboxed Exp})$$

$$\frac{\Gamma, P \vdash f : \texttt{ct}_1 \times \cdots \times \texttt{ct}_n \rightarrow_{GC'} \texttt{ct} \quad \Gamma, P \vdash e_i : \texttt{ct}_i\{B_i, 0, T_i\} \qquad i \in 1..n \quad \Gamma, P \vdash cur\_func : \cdot \rightarrow_{GC} \cdot \qquad GC' \sqsubseteq GC \quad \texttt{gc} \sqsubseteq GC \Rightarrow (ValPtrs(\Gamma) \cap live(\Gamma)) \subseteq P}{\Gamma, P \vdash f(e_1, \ldots, e_n) : \texttt{ct}\{\top, 0, \top\}}\ (\textsc{App})$$

Fig. 8. Type checking for C expressions

appropriate field type. Notice that the resulting B and T information is $\top$, since they are unknown, but the offset is 0, since we will get back safe OCaml data. Rule (Val Deref Boxed Exp) handles the case when B and T are $\top$, which occurs when records or tuples are accessed directly without first testing their boxedness. This rule requires that the type have one, non-nullary constructor and no nullary constructors. We could similarly add a rule when B is boxed but T is $\top$, which is included in our implementation, but this adds no interesting issues.

The rule (C Deref Exp) checks a C pointer dereference. Notice that B and T are always $\top$ for C pointers. (Add C Exp) performs pointer arithmetic on C types other than `value`. Note that in order to ensure soundness, we only allow pointer arithmetic on C pointers with an offset of 0 in our formal system. (Val Add Exp) computes an offset into a structured block. Notice that it must be possible

to safely dereference the resulting pointer, as the offset cannot be larger than the width of the block. While this is not strictly necessary (we could wait until the actual dereference to enforce the size requirement), it seems like good practice not to form invalid pointers. We use (Val Add Boxed Exp) for computing offsets into tuples that are not part of sums. Similar to (Val Deref Boxed Exp), we allow B and T to be $\top$, but require that the type have one, non-nullary constructor and no nullary constructors.

(Val Int Exp) and (Int Val Exp) translate between C and OCaml integers. When we form an OCaml integer from a C integer, we require $T + 1 \leq \Psi$, meaning that the resulting representational type have at least $T + 1$ constructors (Ψ is the count of the constructors, which are numbered from 0). Similarly to (Val Deref Boxed Exp), (Int Val Unboxed Exp) handles the case where a value is used immediately as an integer without a boxedness test.

Finally, (App) type checks a function call. Technically, function calls are not expressions in our grammar, but we put this rule here to make the rules for statements a bit more compact. To invoke a function, the actual and the formal types must match. Notice that the B_i and T_i are discarded, but we require that all actual arguments are safe ($I_i = 0$). Additionally, we require that $GC' \sqsubseteq GC$, since if f might call the garbage collector, so might the current function cur_func, where the type of cur_func is bound in the environment at function definition.

The last hypothesis of (App) requires that if this function may call the garbage collector, then every variable that points into the OCaml heap and is still live must have been registered with a call to `CAMLprotect`. Here $\mathit{ValPtrs}(\Gamma)$ is the set of all variables in Γ with a type (Ψ, Σ) `value` where $|\Sigma| > 0$, i.e., the set of all variables that are pointers into the OCaml heap. The set $\mathit{live}(\Gamma)$ is all variables live at the program point corresponding to Γ. We omit the computation of *live*, since it is standard.

Type Checking for Statements. Unlike our type rules for expressions, which have no side effects, our type rules for statements are flow-sensitive, which we model by allowing the type environment to vary from one statement to another, even in the same scope. This allows us to precisely track facts about local variables. To support branches, our rules use a *label environment* G mapping labels to type environments, where $G(L)$ is the environment at the beginning of statement L.

Since type environments are flow-sensitive, some of our type rules need to constrain type environments to be compatible with each other. Let $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$. Then we define $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in \mathit{dom}(\Gamma)$, and we define $(\Gamma \sqcup \Gamma')(x) = \Gamma(x) \sqcup \Gamma'(x)$ for all $x \in \mathit{dom}(\Gamma)$. Also, for the fall-through case for an unconditional branch our rules need to reset all flow-sensitive information to $\bot$ to remove all flow-sensitive constraints from the type environment. This is required so that the types in the environment will correctly join with those in the incoming environment of a subsequent label statement. We define $\mathit{reset}(\Gamma)(x) = \mathtt{ct}\{\bot, \bot, \bot\}$, where $\Gamma(x) = \mathtt{ct}\{B, I, T\}$.

The top part of Fig. 9 gives our type rules for statements, which prove judgments of the form $\Gamma, G, P \vdash s, \Gamma'$, meaning that in type environment Γ, label environment G, and protection set P, statement s type checks, and after s the new environment is Γ'.

$\boxed{\Gamma, G, P \vdash s : \Gamma'}$

$$\text{(Seq Stmt)}\quad \frac{\Gamma, G, P \vdash s_1, \Gamma' \qquad \Gamma', G, P \vdash s_2, \Gamma''}{\Gamma, G, P \vdash s_1 \,;\, s_2, \Gamma''}$$

$$\text{(Lbl Stmt)}\quad \frac{G(L), G, P \vdash s, \Gamma' \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G, P \vdash L\texttt{:}\ s, \Gamma'}$$

$$\text{(Goto Stmt)}\quad \frac{\Gamma \sqsubseteq G(L)}{\Gamma, G, P \vdash \texttt{goto}\ L, reset(\Gamma)}$$

$$\text{(Ret Stmt)}\quad \frac{\Gamma, P \vdash e : \texttt{ct}\{B, 0, T\} \qquad P = \emptyset \qquad \Gamma \vdash cur\_func : \cdot \rightarrow_{GC} \texttt{ct}}{\Gamma, G, P \vdash \texttt{return}\ e, reset(\Gamma)}$$

$$\text{(CAMLRet Stmt)}\quad \frac{\Gamma, P \vdash e : \texttt{ct}\{B, 0, T\} \qquad P \neq \emptyset \qquad \Gamma, P \vdash cur\_func : \cdot \rightarrow_{GC} \texttt{ct}}{\Gamma, G, P \vdash \texttt{CAMLreturn}(e), reset(\Gamma)}$$

$$\text{(LSet Stmt)}\quad \frac{\Gamma, P \vdash \texttt{*}(e_1 \texttt{+}_p e_2) : \texttt{ct}\{\top, 0, \top\} \qquad \Gamma, P \vdash e_3 : \texttt{ct}\{B, 0, T\}}{\Gamma, G, P \vdash \texttt{*}(e_1 \texttt{+}_p e_2) \texttt{:=} e_3, \Gamma}$$

$$\text{(VSet Stmt)}\quad \frac{\Gamma, P \vdash e : \texttt{ct}\{B, I, T\} \qquad x \in P \Rightarrow I = 0}{\Gamma, G, P \vdash x \texttt{:=} e, \Gamma[x \mapsto \texttt{ct}\{B, I, T\}]}$$

$$\text{(If Stmt)}\quad \frac{\Gamma, P \vdash e : \texttt{int}\{\top, 0, T\} \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G, P \vdash \texttt{if}\ e\ \texttt{then}\ L, \Gamma}$$

$$\text{(If unboxed Stmt)}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}\{B, 0, T\} \qquad \Gamma[x \mapsto mt\ \texttt{value}\{\texttt{unboxed}, 0, T\}] \sqsubseteq G(L)}{\Gamma, G, P \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L, \Gamma[x \mapsto mt\ \texttt{value}\{\texttt{boxed}, 0, T\}]}$$

$$\text{(If sum tag Stmt)}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}\{\texttt{boxed}, 0, T\} \qquad mt = (\Psi, \Pi_0 + \cdots + \Pi_n + \cdots + \emptyset) \qquad \Gamma[x \mapsto mt\ \texttt{value}\{\texttt{boxed}, 0, n\}] \sqsubseteq G(L)}{\Gamma, G, P \vdash \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$$\text{(If int tag Stmt)}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}\{\texttt{unboxed}, 0, T\} \qquad mt = (\Psi, \Sigma) \qquad n + 1 \leq \Psi \qquad \Gamma[x \mapsto mt\ \texttt{value}\{\texttt{unboxed}, 0, n\}] \sqsubseteq G(L)}{\Gamma, G, P \vdash \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L, \Gamma}$$

$\boxed{\Gamma, P \vdash s, \Gamma', P' \text{ and } \Gamma \vdash f, \Gamma'}$

$$\text{(Var Decl)}\quad \frac{ctype = |\texttt{ct}| \qquad \Gamma, P \vdash e : \texttt{ct}\{B, I, T\}}{\Gamma, P \vdash ctype\ x = e, \Gamma[x \mapsto \texttt{ct}\{B, I, T\}], P}$$

$$\text{(CAMLProtect Decl)}\quad \frac{\Gamma, P \vdash x : mt\ \texttt{value}\{B, I, T\}}{\Gamma, P \vdash \texttt{CAMLprotect}(x), \Gamma, P \cup \{x\}}$$

$$\text{(Fun Decl)}\quad \frac{\texttt{ct}_f = \texttt{ct}_1 \times \ldots \times \texttt{ct}_n \rightarrow_{GC} \texttt{ct}_0 \qquad f \in dom(\Gamma) \Rightarrow \texttt{ct}_f = \Gamma(f) \qquad ctype_i = |\texttt{ct}_i| \qquad i \in 0..n}{\Gamma \vdash \texttt{function}\ ctype_0\ f(ctype_1\ x, \ldots, ctype_n\ x), \Gamma'[f \mapsto \texttt{ct}_f]}$$

$$\text{(Fun Defn)}\quad \frac{\begin{array}{c}\Gamma_0 = \Gamma[x_i \mapsto \texttt{ct}_i\{\top, 0, \top\}, cur\_func \mapsto \Gamma(f)] \qquad P_0 = \emptyset \qquad ctype_i = |\texttt{ct}_i| \qquad i \in 1..n \\ \Gamma_{j-1}, P_{j-1} \vdash d_j, \Gamma_j, P_j \qquad j \in 1..m \qquad \Gamma_m, G, P_m \vdash s, \Gamma' \\ \Gamma(f) = \texttt{ct}_1 \times \ldots \times \texttt{ct}_n \rightarrow_{GC} \texttt{ct}_0\end{array}}{\Gamma \vdash \texttt{function}\ ctype\ f(ctype_1\ x_1, \ldots, ctype_n\ x_n)\ d_1 \ldots d_m;\ s, \Gamma}$$

Fig. 9. Type checking for C statements, declarations, and definitions

(Seq Stmt) is straightforward, and (Lbl Stmt) and (Goto Stmt) constrain the type environment $G(L)$ to be compatible with the current environment Γ. (Ret Stmt) checks that the type of e is the same as the return type of the current function. We also require that e is safe and that P is empty so that no variables were registered with the garbage collector. (CAMLRet Stmt) is identical to (Ret Stmt) except that we require P to be non-empty, since it must be paired with at least one `CAMLprotect` declaration. In each of (Goto Stmt), (Ret Stmt), and (CAMLRet Stmt), we use *reset* to compute a new, unconstrained type environment following these statements, since they are unconditional branches.

(LSet Stmt) typechecks writes to memory. We abuse notation slightly and allow e_3 on the right-hand side to be either an expression or a function call, which is checked with rule (App) in Fig. 8. Notice that since we do not model such heap writes flow-sensitively, we require that the type of e_3 is safe, and that the output type environment is the same as the input environment. In contrast, (VSet Stmt) models writes to local variables, which are treated flow-sensitively. Again, we abuse notation and allow the right-hand side to be a function application checked with (App). We also perform an extra check to ensure that any variable registered with the garbage collector is safe.

The rule (If Stmt) models a branch on a C integer. (If unboxed Stmt) models one of our three dynamic tag tests. At label L, we know that local variable x is `unboxed`, and in the else branch (the fall-through case), we know x is `boxed`. We can only apply `if_unboxed` to expressions known to be safe. In particular, in the else branch we must know the offset of the `boxed` data is 0, to allow us to do further tag tests.

Similarly, in (If sum tag Stmt) we set x to have tag n at label L. Notice that this test is only valid if we already know (e.g., by calling `if_unboxed`) that x is boxed and at offset 0, since otherwise the header cannot be read. In the else branch, nothing more is known about x. In either case, we require that if this test is performed, then mt must have at least n possible tags. While omitting this last requirement would not create a runtime error, it may imply a coding error, since the program would be testing for more constructors than are defined by the type. Therefore our heuristic is to warn about this case by including that clause in our rules. In (If int tag Stmt), variable x is known to have value n at label L. Analogously with the previous rule, we require x to be `unboxed`, and with the constraint $n+1 \leq \Psi$ we require that x must have at least $n+1$ nullary constructors. Our implementation also includes variations on (If sum tag Stmt) and (If int tag Stmt) that allow $B = \top$ in exchange for stricter constraints on mt, but we omit these rules since they add no new issues.

The bottom part of Fig. 9 gives type rules for declarations and definitions. For declarations, we use judgments of the form $\Gamma, P \vdash s, \Gamma', P'$, where Γ' and P' are the output environment and protection set. (Var Decl) binds a local variable to the environment, and the protection set does not change. Here we define $|\cdot| : \texttt{ct} \to \mathit{ctype}$ to be the operation of removing mt annotations from `ct`'s to yield *ctype*s, defined in the natural way. (CAMLProtect Decl) takes a variable in the environment and adds it to the protection set P.

The last two rules, (Fun Decl) and (Fun Defn), handle function declarations

$$\frac{}{\Gamma, G \vdash (), \Gamma}\ (\textsc{Empty Stmt}) \qquad \frac{n \sqsubseteq T}{\Gamma \vdash n : \texttt{int}\{\top, 0, T\}}\ (\textsc{Int Exp}) \qquad \frac{\Gamma(l) = \texttt{ct} *\{\top, 0, \top\}}{\Gamma \vdash l : \Gamma(l)}\ (\textsc{Loc Exp})$$

$$\frac{n+1 \leq \Psi \qquad \mathsf{unboxed} \sqsubseteq B \qquad n \sqsubseteq T}{\Gamma \vdash \{n\} : (\Psi, \Sigma)\ \texttt{value}\{B, 0, T\}}\ (\textsc{ML Int Exp})$$

$$\frac{\Gamma(\{l+n\}) = (\Psi, \Sigma)\{B, I, T\} \qquad \mathsf{boxed} \sqsubseteq B \qquad n \sqsubseteq I \qquad m \sqsubseteq T \qquad \Sigma = \Pi_0 + \cdots + \Pi_j \qquad m \leq j \qquad \Pi_m = mt_0 \times \cdots \times mt_k \qquad n \leq k}{\Gamma \vdash \{l+n\} : \Gamma(\{l+n\})}\ (\textsc{ML Loc Exp})$$

Fig. 10. Type rules for values and the empty statement

and definitions. We again use the $|\cdot|$ operator to check that the source types of the parameters match the annotated `ct` types. (Fun Decl) adds the type of f to the output environment, checking that it matches any previous declarations. (Fun Defn) checks the type of a function definition. Note that for simplicity, we assume all functions are declared before use, and we check the type of the function matches the type previously assigned to it. We also bind cur_func to the type of the current function, and we assume that all parameters are safe, which is enforced in (App).

3.3 Soundness

We now sketch a proof of soundness for a slightly simplified version of our multilingual type checking system that omits function calls, and `CAMLprotect` and `CAMLreturn`. We believe these features can be added without difficulty, though with more tedium. We omit some details of the proofs, which can be found in full in a companion technical report [Furr and Foster 2006a].

The first step is to extend our grammar for expressions to include new semantic values: C locations l, OCaml integers $\{n\}$, and OCaml locations $\{l+n\}$, which represents a pointer to the OCaml heap with base address l and offset n. We write $\{l+-1\}$ for the location of the type tag in the header of an OCaml block. We define the syntactic values v as these three forms plus C integers n:

$$v ::= l \mid \{n\} \mid \{l+n\} \mid n$$

As is standard, in our soundness proof we overload Γ so that in addition to containing types for variables, it contains types for C locations and OCaml locations. We also add the empty statement () to our grammar for statements. The type checking rules for these new forms are given in Fig. 10. Rules (Empty Stmt), (Int Exp), and (Loc Exp) are as expected. In rule (ML Int Exp), we assign an OCaml integer {n} a representational type. Note that although our type system does use $\sqsubseteq$ at joins, otherwise the system does not include subsumption. Thus to support preservation, we integrate a notion of subsumption into (ML Int Exp) to allow $\{n\}$ to be assigned a more general type. This rule requires that $\{n\}$'s type have at least $n+1$ constructors, but places no constraint on the boxed component of the type. Similarly, rule (ML Loc Exp) assigns {l + n} its type in Γ, which must be a representational type that has a tag m, where the mth component of the sum has a product with at least n components. In our proof, our inductive hypothesis will add additional conditions on Γ so that $\{l+n\}$ is consistent with the tag $\{l+-1\}$.

$$R ::= [] \mid *R \mid R \mathit{\ aop\ } e \mid v \mathit{\ aop\ } R \mid R +_p e \mid v +_p R \mid \texttt{Val\_int}\ R \mid \texttt{Int\_val}\ R$$
$$\mid \quad R\ \texttt{;}\ s \mid \texttt{if}\ R\ \texttt{then}\ L \mid *R\ \texttt{:=}\ e \mid *v\ \texttt{:=}\ R \mid x\ \texttt{:=}\ R$$

(a) Reduction Contexts

(o-var) $\langle S_C, S_{ML}, V, R[x]\rangle \to \langle S_C, S_{ML}, V, R[v]\rangle$ if $V(x) = v$

(o-ml-add) $\langle S_C, S_{ML}, V, R[\{l + n_1\} +_p n_2]\rangle \to \langle S_C, S_{ML}, V, R[\{l + n\}]\rangle$ if $n = n_1 + n_2$

(o-c-add) $\langle S_C, S_{ML}, V, R[l +_p 0]\rangle \to \langle S_C, S_{ML}, V, R[l]\rangle$

(o-c-deref) $\langle S_C, S_{ML}, V, R[*l]\rangle \to \langle S_C, S_{ML}, V, R[v]\rangle$ if $S_C(l) = v$

(o-ml-deref) $\langle S_C, S_{ML}, V, R[*\{l + n\}]\rangle \to \langle S_C, S_{ML}, V, R[v]\rangle$ if $S_{ML}(\{l + n\}) = v$

(o-aop) $\langle S_C, S_{ML}, V, R[n_1 \mathit{\ aop\ } n_2]\rangle \to \langle S_C, S_{ML}, V, R[n]\rangle$ if $n = n_1 \mathit{\ aop\ } n_2$

(o-valint) $\langle S_C, S_{ML}, V, R[\texttt{Val\_int}\ n]\rangle \to \langle S_C, S_{ML}, V, R[\{n\}]\rangle$

(o-intval) $\langle S_C, S_{ML}, V, R[\texttt{Int\_val}\ \{n\}]\rangle \to \langle S_C, S_{ML}, V, R[n]\rangle$

(b) Expressions

(o-label) $\langle S_C, S_{ML}, V, L : s; s'\rangle \to \langle S_C, S_{ML}, V, s; s'\rangle$

(o-goto) $\langle S_C, S_{ML}, V, \mathit{goto}\ L; s\rangle \to \langle S_C, S_{ML}, V, D(L)\rangle$

(o-c-assign) $\langle S_C, S_{ML}, V, *l := v; s\rangle \to \langle S_C[l \mapsto v], S_{ML}, V, s\rangle$

(o-ml-assign) $\langle S_C, S_{ML}, V, *\{l + n\} := v; s\rangle \to \langle S_C, S_{ML}[\{l + n\} \mapsto v], V, s\rangle$

(o-var-assign) $\langle S_C, S_{ML}, V, x := v; s\rangle \to \langle S_C, S_{ML}, V[x \mapsto v], s\rangle$

(o-if) $\langle S_C, S_{ML}, V, \texttt{if}\ n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, D(L)\rangle$ if $n \neq 0$

(o-if2) $\langle S_C, S_{ML}, V, \texttt{if}\ n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, s\rangle$ if $n = 0$

(o-ifsum) $\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, D(L)\rangle$ if $(S_{ML}(\{l + -1\})) = n$ and $V(x) = \{l + 0\}$

(o-ifsum2) $\langle S_C, S_{ML}, V, \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, s\rangle$ if $(S_{ML}(\{l + -1\})) \neq n$ and $V(x) = \{l + 0\}$

(o-ifi) $\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, D(L)\rangle$ if $V(x) = \{n\}$

(o-ifi2) $\langle S_C, S_{ML}, V, \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, s\rangle$ if $V(x) \neq \{n\}$

(o-iflong) $\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(x)\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, D(L)\rangle$ if $V(x) = \{n\}$

(o-iflong2) $\langle S_C, S_{ML}, V, \texttt{if\_unboxed}(x)\ \texttt{then}\ L; s\rangle \to \langle S_C, S_{ML}, V, s\rangle$ if $V(x) = \{l + 0\}$

(c) Statements

Fig. 11. Small-step semantics rules

Fig. 11(a) defines reduction contexts R, which specify the order of evaluation in our semantics. Here, each expression contains a hole [] that shows what must be evaluated next. Statements such as $\texttt{if\_unboxed}(x)$ are not present as they do not contain any sub-expressions. We use the notation $R[e]$ to mean the reduction context R where the hole is replaced by e.

Our operational semantics uses three stores to model updatable references: S_C maps C locations to values, S_{ML} maps OCaml locations to values, and V maps local variables to values. Our small-step operational semantics for expressions is

shown in Fig. 11(b), which defines a reduction relation of the form

$$\langle S_C, S_{ML}, V, e\rangle \to \langle S'_C, S'_{ML}, V', e'\rangle$$

Here, an expressions e in state S_C, S_{ML}, and V, reduces to a new expression e' and yields new stores S'_C, S'_{ML}, and V' following the style of Felleisen and Hieb[Felleisen and Hieb 1992]. We define $\to^*$ as the reflexive, transitive closure of $\to$.

We discuss the reduction rules briefly. Rule (`o-var`) looks up a variable in the variable store V. (`o-ml-add`) performs pointer arithmetic on a OCaml location. Similarly, (`o-c-add`) performs pointer arithmetic on a C location. However, as mentioned earlier, we only allow offsets of 0 for C locations in this system. (`o-c-deref`) and (`o-ml-deref`) each dereference a location by looking up its value in the appropriate store. (`o-aop`) performs arithmetic on two C integers, and (`o-valint`) and (`o-intval`) convert between C integers and OCaml integers.

Our operational semantics for statements is shown in Fig. 11(c). To model branches, we also include a global statement store D, which maps labels L to a sequence of statements s. That is, $D(L)$ returns the sequence of statements that would be executed after a jump to L. The rule (`o-label`) evaluates to the statement following a label. (`o-goto`) unconditionally jumps to the label L, looking up the next statement to execute in D. (`o-c-assign`), (`o-ml-assign`), and (`o-var-assign`) update a C location, OCaml location, and local variable, respectively, by modifying the store S_C, S_{ML}, or V, as appropriate. (`o-if`) jumps to label L when n is nonzero, and (`o-if2`) executes the fall-through statement otherwise. The remaining conditional rules are similar, jumping to label L when x has the correct dynamic tag (`o-ifsum`), the correct integer value (`o-ifi`), or is unboxed (`o-iflong`).

Since Γ contains type information about the stores, we must ensure that this information correctly types values in the stores. Therefore, we introduce a notion of *compatibility*:

Definition 3.1 *Compatibility.* Γ is said to be compatible with S_C, S_{ML}, and V, written $\Gamma \sim \langle S_C, S_{ML}, V\rangle$, if

(1) $dom(\Gamma) = dom(S_C) \cup dom(S_{ML}) \cup dom(V)$
(2) For all $l \in S_C$ there exists `ct` such that $\Gamma \vdash l : \texttt{ct}\ *\{\top, 0, \top\}$ and $\Gamma \vdash S_C(l) : \texttt{ct}\{\top, 0, \top\}$.
(3) For all $\{l + n\} \in S_{ML}$ there exist Ψ, Σ, j, k, m, $\Pi_0, \ldots, \Pi_j$, and $mt_0, \ldots, mt_k$ such that
   —$\Gamma \vdash \{l + n\} : (\Psi, \Sigma)\ \texttt{value}\{\texttt{boxed}, n, m\}$
   —$\Sigma = \Pi_0 + \cdots + \Pi_j$, $m \le j$
   —$\Pi_m = mt_0 \times \cdots \times mt_k$, $n \le k$
   —$\Gamma \vdash S_{ML}(\{l + n\}) : mt_n\ \texttt{value}\{\top, 0, \top\}$
   —$S_{ML}(\{l + -1\}) = m$
(4) For all $x \in V$, $\Gamma \vdash V(x) : \Gamma(x)$

This definition first ensures that every location in each of our stores is given a type by Γ. Second, it requires that the types of all C locations correctly correspond to the type of the value to which they point. Third, an OCaml location $\{l + n\}$ must have the correct type: it must point to a sum type with at least m non-nullary

constructors, the m^{th} constructor must be a structured block with header tag m and contain at least n values, and the n^{th} value must have the same type as the value stored at $S_{ML}(\{l+n\})$. Finally, the value stored in a variable x must have the same type as x itself.

We begin by showing that given any well-typed expression that is not a value, one of the reduction rules from Fig. 11(b) applies, and the result of the reduction preserves the type of the expression.

LEMMA 3.2 PROGRESS AND PRESERVATION FOR EXPRESSIONS. *If e is an expression and $\Gamma \vdash e : \texttt{ct}\{B, I, T\}$ and $\Gamma \sim \langle S_C, S_{ML}, V\rangle$, then either e is a value or there exists e' such that*

(1) $\langle S_C, S_{ML}, V, e\rangle \to \langle S_C, S_{ML}, V, e'\rangle$, *and*

(2) $\Gamma \vdash e' : \texttt{ct}\{B, I, T\}$

PROOF SKETCH. Proceed by induction on the structure of e. If e has a subexpression that is reduced, then the conclusion holds by induction. Otherwise we assume all subexpressions are values and proceed by case analysis. If $e = x$, then by compatibility of Γ and V, we can apply `(o-var)` and preservation holds. If $e = *e_1$, we have three possibilities, depending on which type rule applies. If (C DEREF EXP) applies, then e_1 is a C pointer, and by compatibility with S_C we can reduce using `(o-c-deref)` and preservation holds. Otherwise either (VAL DEREF EXP) or (VAL DEREF BOXED EXP) applies, and in both cases e_1 must be an OCaml location. Thus by compatibility with S_{ML}, for both cases we can reduce using `(o-ml-deref)` and preservation holds.

If $e = e_1 +_p e_2$, then one of (ADD C EXP), (VAL ADD EXP), or (VAL ADD BOXED EXP) must apply. If (ADD C EXP) applies then e_1 is a C location and e_2 must be the integer 0. Therefore, we can take a step with `(o-c-add)` and preservation is implied by (LOC EXP). In the other two cases e_1 must be an OCaml location $\{l+n\}$, and e_2 must be a C integer. Therefore we can reduce using `(o-ml-add)`, and the respective type rule and compatibility ensure the extracted value is of the right type, implying preservation. If $e = e_1$ *aop* e_2, then the (AOP EXP) must apply, and so we can reduce using `(o-aop)`, and preservation follows because the reduction result is an integer.

If $e =$ `Val_int` e_1, then (VAL INT EXP) must apply, and we can reduce using `(o-valint)`, and preservation holds via (ML INT EXP). If $e =$ `Int_val` e_1, then we can reduce using `(o-intval)`, and preservation holds via (INT EXP). □

We next show progress and preservation for statements. However, first recall that typing judgments for statements include label environments G, which map labels L to type environments Γ. Thus, when we branch to a label L, we need to ensure that the next statement executed ($D(L)$) is well-typed under the environment provided by G (i.e., $G(L)$). Thus we introduce a notion of compatibility of G with our statement store D, similar to the $\sim$ relation defined above:

Definition 3.3 *L-Compatibility.* A statement store D is said to *L-compatible* with a label environment G, written $D \sim_L G$, if for all $L \in D$ there exists a Γ such that $G(L), G \vdash D(L), \Gamma$.

As mentioned above, whenever we branch to a label L, the next statement to be evaluated is $D(L)$. This is only valid if the statement to which D maps L is a labeled statement. Formally:

Definition 3.4 *Well-Formedness of D.* D is said to be *well-formed* if for all $L \in D$, $D(L)$ is a statement of the form $L : s$.

Recall from Section 3.2 that we define $\Gamma \sqsubseteq \Gamma'$ if $\Gamma(x) \sqsubseteq \Gamma'(x)$ for all $x \in dom(\Gamma) \cap dom(\Gamma')$. Since compatibility is an important property to preserve in our progress and preservation lemma for statements, we first present a lemma that shows that store compatibility follows this relation.

LEMMA 3.5. *If* $\Gamma_1 \sqsubseteq \Gamma_2$ *and* $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$ *then* $\Gamma_2 \sim \langle S_C, S_{ML}, V\rangle$

Several statements in our source language contain a label L which the program may branch to. Therefore we first present a lemma that shows that progress and preservation hold for this common case:

LEMMA 3.6. *If* $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$ *and* $D \sim_L G$ *and* D *is well-formed and* $\Gamma_1 \sqsubseteq G(L)$*, then for any statement* s *such that* $\Gamma_1, G \vdash s, \Gamma_2$ *and*

$$\langle S_C, S_{ML}, V, s\rangle \rightarrow \langle S'_C, S'_{ML}, V', D(L)\rangle$$

there exist Γ_3, s' *such that*

(1) $\langle S_C, S_{ML}, V, s\rangle \rightarrow \langle S_C, S_{ML}, V, L : s'\rangle$,
(2) $G(L) \sim \langle S_C, S_{ML}, V\rangle$*, and*
(3) $G(L), G \vdash L : s', \Gamma_3$

Finally, we show progress and preservation for statements. All of our statements will be reduced in one of three ways that correspond to the three possible conclusions below. Either (1) the statement contains a sub-expression which can be reduced, (2) the statement is part of a sequence $s_1; s_2$ and reduces to the second statement, or (3) the statement makes a branch to a label. Each conclusion is similar in that it ensures that at every step of the program: (a) it is possible to take a step, (b) the stores are still compatible with the type environments at that step, and (c) the new statement is still well-typed.

LEMMA 3.7 PROGRESS AND PRESERVATION FOR STATEMENTS. *If* s *is a statement and* $\Gamma_1, G \vdash s, \Gamma_2$ *and* $\Gamma_1 \sim \langle S_C, S_{ML}, V\rangle$ *and* $D \sim_L G$ *and* D *is well-formed, then either* $s = ()$ *or* $s = s_1; s_2$ *and one of the following must hold:*

(1) There exist Γ'_1, s'_1 *such that*
  (a) $\langle S_C, S_{ML}, V, s_1; s_2\rangle \rightarrow \langle S_C, S_{ML}, V, s'_1; s_2\rangle$
  (b) $\Gamma'_1 \sim \langle S_C, S_{ML}, V\rangle$
  (c) $\Gamma'_1, G \vdash s'_1; s_2, \Gamma_2$
(2) There exist $\Gamma'_1, S'_C, S'_{ML}, V'$ *such that*
  (a) $\langle S_C, S_{ML}, V, s_1; s_2\rangle \rightarrow \langle S'_C, S'_{ML}, V', s_2\rangle$
  (b) $\Gamma'_1 \sim \langle S'_C, S'_{ML}, V'\rangle$
  (c) $\Gamma'_1, G \vdash s_2, \Gamma_2$
(3) There exist Γ_4, s_3 *such that*
  (a) $\langle S_C, S_{ML}, V, s_1; s_2\rangle \rightarrow \langle S_C, S_{ML}, V, L : s_3\rangle$

(*b*) $G(L) \sim \langle S_C, S_{ML}, V\rangle$
(*c*) $G(L), G \vdash L : s_3, \Gamma_4$

Proof Sketch. Proceed by case analysis on s_1. If $s_1 = L : s'$, then we can reduce using `(o-label)`, and then by (Lbl Stmt) and Lemma 3.5 we have conclusion (1). If $s_1 = \texttt{goto}\ L$ we can reduce with `(o-goto)`, and by (Goto Stmt) and Lemma 3.6, we satisfy conclusion (3).

If $s_1 = e_1 := e_2$ then we have several sub-cases depending on e_1 and e_2. If either e_1 or e_2 are not a value then we can show (1) using Lemma 3.2. If e_1 and e_2 are values, then e_1 must be one of x, a C location l, or an OCaml location $\{l + n\}$. In each case we will show conclusion (2). If $e_1 = x$ then we can apply `(o-var-assign)` to take a step. The type rule (VSet Stmt) must apply and therefore Γ is updated along with V (from `(o-var-assign)`), which preserves compatibility. If e_1 is a C location l, then we can take a step with `(o-c-assign)` and (LSet Stmt) implies compatibility with S_C. Otherwise, if e_1 is an OCaml location $\{l + n\}$, then we can take a step with `(o-ml-assign)` and compatibility with S_{ML} is again implied by (LSet Stmt).

If $s_1 = \texttt{if}\ e\ \texttt{then}\ L$ then we have several sub-cases depending on e. If e is not a value, then we can show conclusion (1) using Lemma 3.2. If e is a value, then it must be an integer n by (If Stmt) and (Int Exp). If $n \neq 0$, then we can reduce using `(o-if)`, and using Lemma 3.6 we can show conclusion (3). Otherwise, if $n = 0$, we can reduce using `(o-if2)` and show conclusion (2).

If $s_1 = \texttt{if\_sum\_tag}(x) == n\ \texttt{then}\ L$ then the type rule (If sum tag Stmt) applies. Therefore, when we look up x in V, it must be a OCaml location $\{l + 0\}$ by compatibility. If the tag is n (i.e., $\{l + -1\} = n$), then we can reduce using `(o-ifsum)`, and using Lemma 3.6 we can show conclusion (3). Otherwise $\{l + -1\} \neq n$, and we can reduce using `(o-ifsum2)` and show conclusion (2).

If $s_1 = \texttt{if\_int\_tag}(x) == n\ \texttt{then}\ L$ then the type rule (If int tag Stmt) applies. Therefore when we look up x in V, it must be an OCaml integer $\{m\}$ by compatibility. If $n = m$ then we can reduce using `(o-ifi)`, and by Lemma 3.6 we can show conclusion (3). If $n \neq m$, then we can reduce using `(o-ifi2)` and show conclusion (2).

Finally, if $s_1 = \texttt{if\_unboxed}(x)\ \texttt{then}\ L$ then the type rule (If unboxed Stmt) applies, and x must have an OCaml type (mt `value`). Therefore when we look up x in V, it can either be an OCaml integer $\{n\}$ or an OCaml location $\{l + 0\}$ by compatibility. If $V(x) = \{n\}$ then we can reduce using `(o-iflong)`, and by Lemma 3.6 we can show conclusion (3). Otherwise, if $V(x) = \{l + 0\}$ we can reduce using `(o-iflong2)` and show conclusion (2). □

To show soundness, we prove that every statement either diverges or eventually reduces to ().

Theorem 3.8 Soundness. *If* $\Gamma \vdash s, \Gamma'$ *and* $\Gamma \sim \langle S_C, S_{ML}, V\rangle$ *and* $D \sim_L G$ *and* D *is well-formed, then either* $\langle S_C, S_{ML}, V, s\rangle$ *diverges, or* $\langle S_C, S_{ML}, V, s\rangle \to^* \langle S'_C, S'_{ML}, V', ()\rangle$.

Proof. By Lemma 3.7 we can continually reduce the statement and reestablish our compatibly assumptions. Therefore either this process will continue forever, or there exists s' such that $\langle S_C, S_{ML}, V, s\rangle \to^* \langle S'_C, S'_{ML}, V', s'\rangle$ and for all s'',

$$\Phi(mltype_1 \to \cdots \to mltype_n) =$$
$$\rho(mltype_1)\ \texttt{value} \times \cdots \times \rho(mltype_{n-1})\ \texttt{value} \to_\gamma \rho(mltype_n)\ \texttt{value} \qquad \gamma \text{ fresh}$$

$$\begin{aligned}
\rho(\mathsf{unit}) &= (1, \emptyset) \\
\rho(\mathsf{int}) &= (\top, \emptyset) \\
\rho(mltype\ \mathsf{ref}) &= (0, \rho(mltype) \times \emptyset\ +\ \emptyset) \\
\rho(mltype_1 \to mltype_2) &= \rho(mltype_1) \to \rho(mltype_2) \\
\rho(L_1 \mid L_2\ \text{of}\ mltype) &= (1, \rho(mltype) \times \emptyset\ +\ \emptyset) \\
\rho(mltype_1 \times mltype_2) &= (0, \rho(mltype_1) \times \rho(mltype_2) \times \emptyset\ +\ \emptyset)
\end{aligned}$$

Fig. 12. Translation rules for OCaml types

$\langle S_C, S_{ML}, V, s' \rangle \not\to \langle S'_C, S'_{ML}, V', s'' \rangle$. Since s' is well typed by Lemma 3.7, it must be () or else we could apply Lemma 3.7 again and produce s''. □

3.4 Type Inference

Finally, we present our type inference algorithm. Unlike the type checking system, we do not assume that we have OCaml types everywhere. Instead, as described in Fig. 5, we assume we are given an OCaml program and an unannotated C program, and then proceed in two stages. First we convert the source types of FFI functions as declared in OCaml into our multilingual types. Then we perform type inference on the C code, beginning in a type environment containing the converted types, and check for any potential type errors. We discuss each stage in turn.

Translating OCaml Types to Representational Types. The first stage of our inference algorithm is to translate each external function type declared in OCaml into our multilingual types. We only examine OCaml type information and not code because the OCaml type system ensures there are no type errors. We then store the converted types in an offline type repository, which is used during the second stage of our algorithm.

We convert external declarations using the type translation Φ given in Fig. 12, which translates OCaml function types into representational types. In this definition, we implicitly assume that $mltype_n$ is not constructed with $\to$, i.e., the arity of the translated function type is $n-1$. Φ is defined in terms of helper function ρ. The function ρ gives unit and int both pure unboxed types, with no Σ component. Since unit is a singleton type, we know its physical representation is the value 0, and we assign it type $(1, \emptyset)$. This is the same as the representational type for a degenerate sum type with a single nullary constructor, e.g., type t' = A, which is correct because that one nullary constructor has the same representation as unit. As we have seen before, int is translated to $(\top, \emptyset)$, making it incompatible with any sum type. The ρ function encodes mutable references as a boxed type with a single non-nullary constructor of size 1. Regular function types are converted to *mt* function types.

Finally, sum types are handled by counting the nullary constructors and mapping each non-nullary constructor to a product type representing its arguments. Rather than give the general case for sums and products, we illustrate the translation with two sample cases. In the definition of ρ in Fig. 12, we show the translation of a sum type with one nullary constructor and one non-nullary constructor. Product types

```
1                                  // x : α value{⊤, 0, ⊤}
2  if_unboxed(x) {                 //   α = (ψ, σ)
3                                  // x : ... {unboxed, 0, ⊤}
4     if_int_tag(x) == 0           //   1 ≤ ψ
5       /* X */                    // x : ... {unboxed, 0, 0}
6     if_int_tag(x) == 1           //   2 ≤ ψ
7       /* Z */                    // x : ... {unboxed, 0, 1}
8  } else {
9                                  // x : ... {boxed, 0, ⊤}
10    if_sum_tag(x) == 0           //   σ = π0 + σ'
11      /* W */                    // x : ... {boxed, 0, 0}
12      int w1 = Int_val(*(x+1)); //   π0 = α1 × (⊤, ∅) × π0'
13    if_sum_tag(x) == 1           //   σ = π0 + π1 + σ''
14      /* Y */                    // x : ... {boxed, 0, 1}
15      int y0 = Int_val(*(x+0)); //   π1 = (⊤, ∅) × π1'
16 }                               // x : ... {⊤, 0, ⊤}
```

Fig. 13. "Pattern matching" code with inferred types and constraints

are handled by making an appropriate boxed type with no nullary constructors and a single non-nullary constructor of the appropriate shape.

Consider the following OCaml program, which declares a foreign function:

$$\begin{array}{l}\mathsf{type\ s = P\ of\ int\ ref \mid R \mid Q}\\ \mathsf{external\ f}_{ML} : \mathsf{int} \to \mathsf{s} \to \mathsf{unit} = \text{“}f_C\text{”}\end{array}$$

The Φ function converts this function type as follows. The first argument of f_C has type int and is thus represented as $(\top, \emptyset)$. The second argument has type s, which has one non-nullary constructor and two nullary constructors. The non-nullary constructor takes as its argument an int ref, which corresponds to the representational type $(0, (\top, \emptyset) \times \emptyset + \emptyset)$. Therefore the representational type for s is $(2, (0, (\top, \emptyset) \times \emptyset + \emptyset) \times \emptyset \ + \ \emptyset)$. Finally, the return type of f_C is unit, which is represented as $(1, \emptyset)$ as in Fig. 12. Therefore, the multilingual type of this function, which is stored in the type repository, is as follows, where γ is a fresh variable to be solved for in the next stage of analysis:

$$f_C : (\top, \emptyset) \times (2, (0, (\top, \emptyset) \times \emptyset + \emptyset) \times \emptyset \ + \ \emptyset) \to_\gamma (1, \emptyset)$$

The second stage of our analysis then assigns this signature to the function f_C in the C code and ensures that it is consistent with body of f_C.

Type Inference Example. To motivate the discussion of the next phase of type inference, Fig. 13 shows the inference process for the example from Fig. 2(c), which has been rewritten in our formal language. Assume this code starts with x initialized to some data passed from OCaml. To enhance readability we omit labels and jumps, and instead show control-flow with indentation. We have annotated the example with the types assigned by our inference rules. The variable x begins on line 1 with an unknown type α value$\{\top, 0, \top\}$. B and T are $\top$ here because the boxedness and tag of x are unknown at this program point, and I is set to zero since all data passed from OCaml is safe. Upon seeing the if_unboxed call, α unifies with the representational type (ψ, σ), where ψ and σ are variables to be solved for based on

the constraints generated in the remaining code. On the true branch, we give `x` an `unboxed` type but still an unknown tag. Since the flow-insensitive part of `x`'s type does not change (it is always α `value` throughout the function), we elide it from here on in the figure. Line 4 checks the unboxed constructor for `x` and adds the constraint $1 \leq \psi$, since `x` can only be from a sum with at least 1 nullary constructor. Thus on line 5, `x` is now fully known, and can safely be used as the nullary type constructor X. Similarly, on line 7, `x` is known to be the constructor Z, and we generate the constraint $2 \leq \psi$ from the tag test on line 6.

On the false branch of the `if_unboxed` test, our type rules give `x` a boxed type with offset 0 (since `x` is safe). After testing the tag of `x` against 0 on line 10, we know that `x` has at least one non-nullary constructor, which we enforce with the constraint $\sigma = \pi_0 + \sigma'$. On line 11, then, `x` can be safely treated as the constructor W (tag 0). Line 12 accesses the second field of `x` and treats it as an integer. Therefore we unify π_0 with a product with at least two types, the second of which is the integer type $(\top, \emptyset)$. Similarly, on line 14 we know that `x` has constructor Y (tag 1), and on line 15, we know the constructor has at least one field, which is an integer. Finally, on line 16, we join the branches together and lose information about the boxedness or tag of `x`.

When we solve the unification constraints on α, π_0, and π_1 and inequality constraints on ψ, we discover $\alpha = (\psi, \quad \alpha_1 \times (\top, \emptyset) \times \pi_0' \quad + \quad (\top, \emptyset) \times \pi_1' \quad + \quad \sigma'')$ with $2 \leq \psi$, which correctly unifies with our original type OCaml type `t`, which corresponds to representational type $(2, \quad (\top, \emptyset) \times (\top, \emptyset) \times \emptyset \quad + \quad (\top, \emptyset) \times \emptyset \quad + \quad \emptyset)$. After unification, we therefore also discover that $\alpha_1 = (\top, \emptyset)$ and that π_0', π_1', and σ'' are all $\emptyset$.

Type Inference for Glue Code. The second phase of our inference algorithm infers types for C source code, incorporating the representational types gathered in the first phase. Recall that our system uses types of the form $\texttt{ct}\{B, I, T\}$, where `ct` represents the structure of OCaml data and is flow-insensitive, whereas B, I, and T are flow-sensitive. During inference, O-Saffire generates *constraints* on flow-insensitive type structure, and uses data flow analysis to infer the flow-sensitive B, I, and T. We generate four kinds of constraints C:

$$C \ ::= \ \tau = \tau' \mid T + 1 \leq \Psi \mid GC \sqsubseteq GC' \mid \mathsf{gc} \sqsubseteq GC' \Rightarrow P \subseteq P'$$

From left to right, we have equality constraints of the form $\tau = \tau'$, where τ ranges over `ct`, mt, Π, and Σ; inequality constraints $T + 1 \leq \Psi$ that give lower bounds on the number of primitive tags of a representational type; inequality constraints $GC \sqsubseteq GC'$ among garbage collection effects; and conditional constraints $\mathsf{gc} \sqsubseteq GC' \Rightarrow P \subseteq P'$, used to check that C pointers to the OCaml heap are correctly registered with the garbage collector.

Fig. 14 shows our type inference rules for expressions, which are almost the same as the type checking rules in Fig. 8, except that they introduce fresh variables where appropriate. When we apply these rules to an expression, we view any constraints C listed as side-conditions of the hypotheses as being *generated*, and at the end of type inference we gather all the generated constraints and solve them.

Fig. 14 only lists rules that are different than their type checking counterparts. For example, in the rule (Val Deref Exp), the exact number of non-nullary

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\texttt{boxed}, n, m\} \quad mt = (\psi, \pi_0 + \cdots + \pi_m + \sigma) \quad \pi_m = \alpha_0 \times \ldots \times \alpha_n \times \pi \quad \psi, \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash *e : \alpha_n\ \texttt{value}\{\top, 0, \top\}} \text{ (Val Deref Exp)}$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, 0, \top\} \quad mt = (0, \pi_0 + \emptyset) \quad \pi_0 = \alpha_0 \times \cdots \times \alpha_n \times \pi \quad \alpha_i, \pi_0, \pi \text{ fresh}}{\Gamma, P \vdash *e : \alpha_n\ \texttt{value}\{\top, 0, \top\}} \text{ (Val Deref Boxed Exp)}$$

$$\frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\texttt{boxed}, n, n'\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (\psi, \pi_0 + \cdots + \pi_{n'} + \sigma) \quad \pi_{n'} = \alpha_0 \times \cdots \times \alpha_{n+m} \times \pi \quad \psi, \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, n+m, n'\}} \text{ (Val Add Exp)}$$

$$\frac{\Gamma, P \vdash e_1 : mt\ \texttt{value}\{\top, 0, \top\} \quad \Gamma, P \vdash e_2 : \texttt{int}\{\top, 0, m\} \quad mt = (0, \pi_0 + \emptyset) \quad \pi_0 = \alpha_0 \times \cdots \times \alpha_m \times \pi \quad \pi_i, \sigma, \alpha_i, \pi \text{ fresh}}{\Gamma, P \vdash e_1 +_p e_2 : mt\ \texttt{value}\{\texttt{boxed}, m, 0\}} \text{ (Val Add Boxed Exp)}$$

$$\frac{\Gamma, P \vdash e : \texttt{int}\{\top, 0, T\} \quad T + 1 \leq \psi \qquad \psi, \sigma \text{ fresh}}{\Gamma, P \vdash \texttt{Val\_int}\ e : (\psi, \sigma)\ \texttt{value}\{\texttt{unboxed}, 0, T\}} \text{ (Val Int Exp)}$$

$$\frac{\Gamma, P \vdash e : mt\ \texttt{value}\{\top, 0, T\} \quad mt = (\psi, \emptyset) \qquad \psi \text{ fresh}}{\Gamma, P \vdash \texttt{Int\_val}\ e : \texttt{int}\{\top, 0, T\}} \text{ (Int Val Unboxed Exp)}$$

$$\frac{\Gamma, P \vdash f : \texttt{ct}_1 \times \cdots \times \texttt{ct}_n \to_{GC'} \texttt{ct} \quad \Gamma, P \vdash e_i : \texttt{ct}'_i\{B_i, 0, T_i\} \qquad \texttt{ct}_i = \texttt{ct}'_i \qquad i \in 1..n \quad \Gamma, P \vdash \mathit{cur\_func} : \cdot \to_{GC} \cdot \qquad GC' \sqsubseteq GC \quad \texttt{gc} \sqsubseteq GC \Rightarrow (\mathit{ValPtrs}(\Gamma) \cap \mathit{live}(\Gamma)) \subseteq P}{\Gamma, P \vdash f(e_1, \ldots, e_n) : \texttt{ct}\{\top, 0, \top\}} \text{ (App)}$$

Fig. 14. Expression type rules modified for inference

constructors may not be known, and therefore we generate a constraint $mt = (\psi, \pi_0 + \cdots + \pi_m + \sigma)$ to unify mt with a representational type whose sum component ends in variable σ and thus may grow during inference. Only when we unify this type with a known OCaml type will the exact size be fixed. As another example, in (Val Int Exp), we generate the constraint $T + 1 \leq \psi$ to require that e have at least $T + 1$ non-nullary constructors.

Note that none of our expression type inference rules allow $I = \top$. This can occur after a join point when a variable has been used to point to two different offsets of a block on two separate branches. In practice, if I is ever $\top$, O-Saffire emits a message warning that it does not have enough information to analyze the expression and proceeds to optimistically check the rest of the program in search of further errors. Examples of this are discussed in Section 5.2.

Unlike our type rules for expressions, our type rules for statements are somewhat non-standard: They allow the type environment to change from one program point to another, and they include a label environment G used to model jumps. Thus we use dataflow analysis to infer types for statements, and in lieu of more traditional type inference judgments we present statement inference as a pair of algorithms. We define an algorithm InferFunc to perform inference on a function body by iteratively applying another algorithm InferStmt, which infers types for statements. To integrate the expression type inference rules in Fig. 14 into the al-

Algorithm 1 INFERFUNC(Γ, C, f) – Type inference for C functions

Input: A type environment Γ, a constraint set C, and a function f of the form

$$\texttt{function}\ ctype_0\ f(ctype_1\ x_1, \ldots, ctype_n\ x_n)\ d_1 \ldots d_m;\ s_1; \ldots; s_k$$

Side effects: Updates constraint set C

1: $\texttt{ct}_i \leftarrow \eta(ctype_i)$ for $i \in 0..n$
2: $C \leftarrow C \cup \{\Gamma(f) = \texttt{ct}_1 \times \ldots \times \texttt{ct}_n \rightarrow_\gamma \texttt{ct}_0\}$ where γ fresh
3: $\Gamma' \leftarrow \Gamma[x_i \mapsto \texttt{ct}_i\{\top, 0, \top\}, cur\_func \mapsto \Gamma(f)]$
4: $P \leftarrow \emptyset$
5: **for** $i \in 1..m$ **do**
6: **switch** d_i
7: **case** $ctype_x\ x = e$**:**
8: $\texttt{ct}\{B, I, T\} \leftarrow$ INFEREXPR(C, Γ, P, e)
9: $\Gamma' \leftarrow \Gamma'[x \mapsto \texttt{ct}\{B, I, T\}]$
10: **case** `CAMLprotect`(x)**:**
11: $P \leftarrow P \cup \{x\}$
12: **end switch**
13: **end for**
14: $\forall L \in$ body of $f, G(L) \leftarrow reset(\Gamma')$
15: $\forall s \in$ body of $f, out_\Gamma[s] \leftarrow reset(\Gamma')$
16: $out_\Gamma[start] = \Gamma'$
17: $W \leftarrow \{s_1\}$
18: **while** W is not empty **do**
19: remove a statement s from W
20: $in_\Gamma[s] \leftarrow \bigsqcup_{p \in pred[s]} out_\Gamma[p]$
21: $G_{old} \leftarrow G$
22: $\Gamma_{new} \leftarrow$ INFERSTMT$(C, in_\Gamma[s], P, G, s)$
23: **if** $out_\Gamma[s] \neq \Gamma_{new}$ or $G \neq G_{old}$ **then**
24: add all successors of s to W
25: $out_\Gamma[s] \leftarrow \Gamma_{new}$
26: **end if**
27: **end while**

gorithm, we use the notation INFEREXPR(C, Γ, P, e) to mean the type $\texttt{ct}\{B, I, T\}$ such that $\Gamma, P \vdash e : \texttt{ct}\{B, I, T\}$ according to the rules in Fig. 14. Any constraints generated while applying these rules are added to C as a side effect.

Algorithm 1 defines INFERFUNC(Γ, C, f), which takes as input a type environment Γ (containing its type and the type of other functions in the program), a constraint set C (added to during inference), and a function definition f, and performs type inference on the function body. To perform inference on a program, we apply this algorithm to each function definition in order, building up a set of constraints that are solved at the end. In essence, this algorithm takes the place of the (FUN DEFN) rule in Fig. 9. We omit the part of the algorithm for handling function declarations, since it simply adds those declarations to the current global type environment.

We assume we have access to a control-flow graph for the body of f, and that in the body of the function $s_1; \ldots; s_k$, each s_i is not itself a sequence statement (but may be labeled). We also assume that s_1 has a distinguished predecessor named *start*. The algorithm INFERFUNC computes a type environment $out_\Gamma[s]$ that holds immediately after each statement s.

The algorithm begins on line 1 by translating *ctype*s, which are present in the

C code, to `cts`, which include OCaml type information. Since we do not know the OCaml types yet, we introduce fresh variables everywhere using the function η:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l} \eta(\texttt{void}) & = & \texttt{void} & \eta(\texttt{value}) & = & \alpha\ \texttt{value} & \alpha \text{ fresh} \\ \eta(\texttt{int}) & = & \texttt{int} & \eta(\mathit{ctype}\ \texttt{*}) & = & \eta(\mathit{ctype})\ \texttt{*} & \end{array}$$

We do not translate C function types because they are not first class in our language. Then line 2 adds a constraint that the `cts` for parameters and return unify with the existing function type, and line 3 creates an initial type environment Γ' that includes the formal parameters and the name cur_func.

Next, lines 4–13 handle local variable declarations. Lines 7–9 bind local variables in Γ', using InferExpr to infer types of initializers. Lines 10–11 add variables to the protection set P. Then line 14 uses *reset* to initialize the label environment G to map each label L to the initial type environment with all of the flow-sensitive elements set to $\bot$. Then we initialize the dataflow facts for each statement in the CFG, where the *start*, an empty node preceding s_1, ends in environment Γ'. Lastly, we initialize the worklist with the first statement.

The heart of the algorithm is lines 18–27, which iterate over the worklist. We remove a statement from the worklist, and then join the type environments from all predecessors. Line 21 saves a copy of the current label environment G, and then we call InferStmt to infer the output environment Γ_{new} after statement s is executed. As we discuss below, this may also update C and G. If G is updated, or if Γ_{new} is different than $out_\Gamma[s]$, then we place all successors of s on the worklist and update $out_\Gamma[s]$.

Algorithm 2 defines InferStmt(C, Γ, P, G, s), which performs type inference on a statement s. The other inputs to the algorithm are the constraint set C, an initial type environment Γ, a protection set P, and a label environment G. The algorithm is a simple case analysis depending on s.

Lines 2–4 infer the type of a labeled statement, checking the inner statement in environment $G(L)$, where G has been imperatively updated to join the initial environment Γ with the previous $G(L)$. If G is actually changed by this case, then Algorithm 1 places the successors of s on the worklist, which will cause them to be visited. Lines 5–7 handle `goto`, imperatively updating the G and returning an environment with all flow-sensitive components set to $\bot$.

Lines 8–14 handle the two forms of `return`, ensuring that `CAMLreturn` is used if and only if variables have been registered with `CAMLprotect` (which places the variables in P). This case uses InferExpr to infer the type of e, and generates a constraint unifying this type with the return type of the function. Note that the returned value must be safe. Lines 15–20 handle assignment through a pointer, unifying the types of the left- and right-hand sides on line 19. Line 18 ensures that only safe data is stored in the heap. Lines 21–24 handle assignment to a local variable, which updates the type of x in the output type environment and ensures that if x has been registered with the garbage collector, then it is safe.

The remaining four cases on lines 25–48 handle branches. In each case, we update the label environment G by joining the current $G(L)$ with information determined by the conditional test, if any. The last three cases also require $I = 0$, so that the tests can be performed safely, and update the output environment to also reflect the conditional. Lastly, the cases for `if_sum_tag` and `if_int_tag` require that x

Algorithm 2 InferStmt(C, Γ, P, G, s) – Type inference for C statements

Input: An initial type environment Γ, a label environment G, a protection set P, a statement s, and a constraint set C
Output: A type environment that holds at the end of statement s
Side effects: Updates constraint set C, label environment G

switch s
case $L : s'$**:**
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma]$
  return InferStmt$(C, G(L), P, G, s')$
case `goto` L**:**
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma]$
  return $reset(\Gamma)$
case `return` e **or** `CAMLreturn`(e)**:**
  fail if $(s = \texttt{return}\ e \wedge P \neq \emptyset) \vee (s = \texttt{CAMLreturn}(e) \wedge P = \emptyset)$
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr(C, Γ, P, e)
  fail if $I \neq 0$
  $\Gamma(cur\_func) = \texttt{ct}_1 \times \ldots \times \texttt{ct}_n \rightarrow_{GC} \texttt{ct}_0$
  $C \leftarrow C \cup \{\texttt{ct} = \texttt{ct}_0\}$
  return $reset(\Gamma)$
case $*(e_1 +_p e_2)$ `:=` e_3**:**
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr$(C, \Gamma, P, *(e_1 +_p e_2))$
  $\texttt{ct}'\{B', I', T'\} \leftarrow$ InferExpr(C, Γ, P, e_3)
  fail if $(I \neq 0 \vee I' \neq 0)$
  $C \leftarrow C \cup \{\texttt{ct} = \texttt{ct}'\}$
  return Γ
case x `:=` e**:**
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr(C, Γ, P, e)
  fail if $(x \in P \wedge I \neq 0)$
  return $\Gamma[x \mapsto \texttt{ct}\{B, I, T\}]$
case `if` e `then` L**:**
  InferExpr(C, Γ, P, e) *(C compiler ensures result is int)*
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma]$
  return Γ
case `if_unboxed`(x) `then` L**:**
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr(C, Γ, P, x)
  fail if $I \neq 0$
  $\Gamma' \leftarrow \Gamma[x \mapsto \texttt{ct}\{\texttt{unboxed}, I, T\}]$
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma']$
  return $\Gamma[x \mapsto \texttt{ct}\{\texttt{boxed}, I, T\}]$
case `if_sum_tag`$(x) == n$ `then` L**:**
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr(C, Γ, P, x)
  fail if $I \neq 0$ or $B \neq \texttt{boxed}$
  $C \leftarrow C \cup \{\texttt{ct} = (\psi, \pi_0 + \cdots + \pi_n + \sigma)\ \texttt{value}\}$ ψ, π_i, σ fresh
  $\Gamma' \leftarrow \Gamma[x \mapsto \texttt{ct}\{\texttt{boxed}, 0, n\}]$
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma']$
  return Γ
case `if_int_tag`$(x) == n$ `then` L**:**
  $\texttt{ct}\{B, I, T\} \leftarrow$ InferExpr(C, Γ, P, x)
  fail if $I \neq 0$ or $B \neq \texttt{unboxed}$
  $C \leftarrow C \cup \{\texttt{ct} = (\psi, \sigma)\ \texttt{value})\} \cup \{n + 1 \leq \psi\}$ ψ, σ fresh
  $\Gamma' \leftarrow \Gamma[x \mapsto \texttt{ct}\{\texttt{unboxed}, 0, n\}]$
  $G \leftarrow G[L \mapsto G(L) \sqcup \Gamma']$
  return Γ
end switch

is boxed or unboxed, respectively, and add equality constraints to C to ensure the type of x has the right shape.

Constraint Solving. Algorithm 1 clearly terminates, because updates monotonically increase facts about B, I, and T, which are finite height lattices. After we have applied Algorithm 1 to all of the functions in our program, we are left with a constraint set C. We solve the equality constraints $\tau = \tau'$ with ordinary unification. When solving a constraint $(\Psi, \cdot) = (\Psi', \cdot)$, we require that Ψ and Ψ' are the same, i.e., if $\Psi = n$, then Ψ does not unify with $\top$, as required by our type rules. Several rules also produce inequality constraints of the form $T + 1 \leq \Psi$. Recall that these ensure that nullary constructors can only be used with a sum type that is large enough. Thus in this constraint, if T is negative, we require $\Psi = \top$, since negative numbers are never constructors. After the unification constraints have been solved, we can walk through the list of $T + 1 \leq \Psi$ constraints and check whether they are satisfiable.

Finally, we are left with constraints involving garbage collection effects. The atomic subtyping constraints $GC \sqsubseteq GC'$ can be solved via graph reachability. We can think of the constraint $GC \sqsubseteq GC'$ as an edge from GC to GC'. Such edges form a call graph, i.e., there is an edge from GC to GC' if the function with effect GC is called by the function with effect GC'. To determine whether a function with effect variable γ may call the garbage collector, we simply check whether there is a path from gc to γ in this graph, and using this information we ensure that any conditional constraints $\mathsf{gc} \sqsubseteq GC' \Rightarrow P \subseteq P'$ from (App) are satisfied for gc functions.

4. J-SAFFIRE: TYPE INFERENCE FOR THE JNI

In this section, we present J-Saffire, our multilingual type inference system for the Java Native Interface (JNI). The overall design of J-Saffire is the same as O-Saffire, but the systems differ substantially in detail because of differences between the JNI and the OCaml FFI. First, although we found little use of objects in the OCaml FFI, objects are critical for the JNI. Indeed, J-Saffire's main focus is on inferring representations of Java object types. Second, while the OCaml FFI uses integer tags and pointer offsets to access OCaml data, in the JNI, fields, methods, and classes are all described using specially-formatted strings, and those strings typically do not change during execution. Thus J-Saffire uses a flow-insensitive analysis to track string values through glue code. Finally, we found that parametric polymorphism is important for the JNI, because it allows J-Saffire to model user-defined wrapper functions precisely and directly assign type signatures to the multitude of JNI functions, rather than give separate type rules for each function, as O-Saffire does. Thus J-Saffire uses instantiation constraints, discussed later, to perform parametric polymorphic type inference. This last point is especially important for our implementation as we do not need to handle each JNI function specially, but rather simply write down a type signature for each.

Source Language. In Section 3, we described O-Saffire in terms of a source language that was very close to C, which was helpful because C language constructs (pointer arithmetic, conditional tests, etc.) are directly used to manipulate OCaml

$$
\begin{array}{rcl}
e & ::= & x \mid f_i \mid \lambda x.e \mid \textit{“Str”} \mid \delta_i(e, \ldots, e) \mid e\ e \mid \text{let } f = e \text{ in } e \\
\delta & ::= & \texttt{FindClass} \mid \texttt{GetObjectClass} \mid \texttt{GetFieldID} \mid \texttt{GetMethodID} \\
& \mid & \texttt{GetObjectField} \mid \texttt{CallObjectMethod}_n \mid \ldots
\end{array}
$$

Fig. 15. JNI source language

```
1  let my_getObjectField = λobj.λfield.
2    let cls = GetObjectClass1(obj) in
3    let fid = GetFieldID2(cls, field, "java.lang.Object") in
4      GetObjectField3(obj, fid)
5  in ...
```

Fig. 16. JNI wrapper function from Fig. 4 in formal language

data in the OCaml FFI. However, as we saw in Section 2.2, the JNI is much more opaque, and glue code does almost all of its work by invoking JNI functions. Thus to make J-Saffire simpler to present, and soundness easier to prove, we describe J-Saffire in terms of the language in Fig. 15, which is the lambda calculus extended with primitive strings *“Str”* and JNI functions δ, which are not curried. In our source language, let introduces polymorphism, and let-bound variables f are annotated with an *instantiation site* i when they are used in the program text. JNI functions δ are also polymorphic, and hence include an instantiation site. We discuss this further in Section 4.2.

We list a few JNI functions as examples. The function $\texttt{FindClass}(s)$ returns the Class object of the class named by string s. The function $\texttt{GetObjectClass}(o)$ returns the Class of its argument o. The function $\texttt{GetFieldID}(c, f, t)$ returns a field identifier for the field named f of type t in class c, and similarly $\texttt{GetMethodID}(c, m, t)$ returns a method identifier for the method named m of type t in class c. The function $\texttt{GetObjectField}(o, fid)$ returns the object (i.e., non-primitive) field identified by fid of object o, and lastly, the function $\texttt{CallObjectMethod}_n(o, m, x_1, \ldots, x_n)$ invokes method m of object o with arguments x_1 through x_n and returns the object result. In the JNI, `CallObjectMethod` is a varargs function, and in our formalism we assume we have variations $\texttt{CallObjectMethod}_n$ for each possible arity n.

4.1 Multilingual Types

Like O-Saffire, J-Saffire uses a multilingual type language when performing type inference on glue code. In the JNI, the key type is `jobject`, which is the C type given to all Java objects, and data of this type can only be manipulated by passing it to JNI functions. Thus our strategy is to extend `jobject` with a representational type to model C's view of Java objects. The JNI also uses field and method identifiers, which have C types `jfieldID` and `jmethodID`, respectively, and so J-Saffire needs to extend those types as well with information on which fields and methods they represent.

Before we present our multilingual type language formally, consider again the example wrapper function `my_getObjectField` from Section 2.2, shown in Fig. 16 in our formal language grammar. Recall that this function takes two arguments,

$$
\begin{array}{rcll}
\texttt{ct} & ::= & \alpha \mid \texttt{str}\{s\} \mid (\texttt{ct} \times \cdots \texttt{ct}) \to \texttt{ct} & \textit{(C types)} \\
& \mid & jt\ \texttt{jobject} & \\
& \mid & (f, o)\ \texttt{jfieldID} & \\
& \mid & (m, o)\ \texttt{jmethodID} & \\
s & ::= & \nu \mid \textit{“Str”} & \textit{(Singleton strings)} \\
jt & ::= & \iota \mid o \mid jt\ \mathsf{JClass} & \textit{(Java types)} \\
o & ::= & \{s; F; M\} & \textit{(Rep. types for objects)} \\
f & ::= & s : jt & \textit{(Field typing)} \\
F & ::= & \phi \mid \emptyset \mid \langle f; \cdots; f \rangle \circ F & \textit{(Field set)} \\
m & ::= & s : [s; sig] & \textit{(Method typing)} \\
sig & ::= & \psi \mid (jt \times \cdots jt) \to jt & \textit{(Method sig)} \\
M & ::= & \mu \mid \emptyset \mid \langle m; \cdots; m \rangle \circ M & \textit{(Method set)} \\
\sigma & ::= & \forall \vec{\vartheta}.\texttt{ct} & \textit{(Type scheme)}
\end{array}
$$

Fig. 17. Java/C multilingual type language

an object and the name of one of its Object fields, and then returns that field. Notice that in order to assign this function a type, we need to specify that the value of the second argument names a field of the first argument; that field must be a java.lang.Object; and any other fields or methods of the object type are unconstrained. J-Saffire gives `my_getObjectField` the following type:

$$\{\nu; \langle \nu_{field} : o_{ret} \rangle \circ \phi; \mu\}\ \texttt{jobject} \to \texttt{str}\{\nu_{field}\} \to o_{ret}\ \texttt{jobject}$$

where $o_{ret} = \{\textit{“java.lang.Object”}; \langle f_1; \cdots; f_n \rangle; \langle m_1; \cdots; m_k \rangle\}$ is a representational type describing instances of java.lang.Object. This representational types has three parts: the class name *“java.lang.Object”*; the field set $\langle f_1; \cdots; f_n \rangle$; and the method set $\langle m_1; \cdots; m_k \rangle$. These latter two sets come from the definition of Object in Java.

The second parameter is a C string whose contents are represented by the type variable ν_{field}. We use a variable because when the function is created, the actual contents of the string are unknown (and in fact are likely to vary with different calls to the function). Such variables may be later unified with a constant string once the contents of the string becomes known, creating a singleton type. Lastly, the first parameter is a Java object with representational type $\{\nu; \langle \nu_{field} : o_{ret} \rangle \circ \phi; \mu\}$. In this type, the class name ν is an unconstrained variable; the field set must contain a field named ν_{field} of type o_{ret}, but then may contain anything else, which is represented by variable ϕ; and the method set is unconstrained, as represented by the variable μ. In order to infer this type, J-Saffire also needs to track intermediate information about `cls` and `fid` as well.

Grammar for Java/C Multilingual Types. Our formal multilingual type grammar is given in Fig. 17. Our type language does not include integer or void types because our source language does not contain any values of these types. This simplifies the presentation of our system by restricting communication through the JNI to Java object types. Our implementation (Section 5.3) does handle primitives.

The type language in Fig. 17 has type variables α, singleton string types $\texttt{str}\{s\}$, and (possibly uncurried) function types. The type $\texttt{str}\{s\}$ is our formalism for the C type `char *`. In this type, the string s may be either a type variable ν to be solved for, or a known constant *“Str”*. For example, in `my_getObjectField`, the parameter `field` is given type $\texttt{str}\{\nu_{field}\}$, and the parameter `obj` has a field

named ν_{field}.

The C types `ct` also include `jobject`, `jfieldID`, and `jmethodID`, extended to contain Java type information. The type jt `jobject` represents a Java object with Java type jt. The types (f, o) `jfieldID` and (m, o) `jmethodID` represent intermediate JNI values for extracting field f or method m from object of type o. We include o so that we can check that this field identifier is used with an object of the correct type. For example, `fid` in `my_getObjectField` has type $(\nu_{field} : o_{ret}, o_{obj})$, and thus can be used to extract a field named ν_{field} of type o_{ret} from an instance of type o_{obj}.

Our grammar for Java type jt includes type variables ι, a *representational type* o for Java objects, and a class type jt JClass. Representational types o have the form $\{s; F; M\}$, which represents an instance of the class named s with *field set* F and *method set* M. Our type checking system will implicitly assume that, if s is a known string in a representational type $\{s; F; M\}$, then F and M match the field and method information from Java. In a representational type, a field set is a list of field typings $s : jt$, which means the field named s has Java type jt. Notice that s may be a variable or a known string. Similarly, a method set is a list of method typings $s : [s'; sig]$, where s is the method name, s' is the method descriptor (a string describing the method argument and return types), and sig is the method signature. Method signatures sig may be a variable ψ representing an unknown signature, or $(jt_1 \times \cdots jt_n) \to jt_0$, where $jt_1 \ldots jt_n$ are the argument types and jt_0 is the return type. Our type checking system will implicitly assume that the method descriptor, if it is a known string, correctly describes its associated method signature.

When performing inference, J-Saffire may discover the fields and methods of an object incrementally, and so we allow these sets to grow with the composition operator $\circ$. We say that a set is *closed* if it is composed with the empty set of methods $\emptyset$, 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 the class name s is a known string. This is similar to O-Saffire, in which sums Σ and products Π may grow during inference.

Returning to our last jt type, a *class type* jt JClass is the type J-Saffire gives to a Class object that represents the Java type jt. Recall that instances of Class are essential for using the JNI. For example, when J-Saffire infers a type for `my_getObjectField`, the local variable `cls`, which holds the class of `obj`, is given the type o_{obj} JClass `jobject`, where o_{obj} is the representational type for `obj`. J-Saffire needs to know what classes instances of Class represent in order to determine types for fields and methods.

Finally, we include universal polymorphic type schemes σ of the form $\forall\vec{\vartheta}.$`ct`. Here to reduce notation, we use ϑ to range over any of the variables in our type system (α, ν, ι, ϕ, ψ, or μ).

Example JNI Function Types. Given this type grammar, we can precisely describe the types of the JNI functions. Fig. 18 gives polymorphic type signatures for the functions mentioned in Fig. 16. `FindClass` takes a string ν and returns the class object for the class named ν. Note that although the field and method

$$\begin{aligned}
\texttt{FindClass}: &\ \forall \nu, \phi, \mu.\ (\texttt{str}\{\nu\}) \to \{\nu; \phi; \mu\}\ \text{JClass}\ \texttt{jobject} \\
\texttt{GetObjectClass}: &\ \forall \nu, \phi, \mu.\ (\{\nu; \phi; \mu\}\ \texttt{jobject}) \to \{\nu; \phi; \mu\}\ \text{JClass}\ \texttt{jobject} \\
\texttt{GetFieldID}: &\ \forall \nu_i, \nu_f, \phi_i, \mu_i. \\
&\quad (o_1\ \text{JClass}\ \texttt{jobject} \times \texttt{str}\{\nu_f\} \times \texttt{str}\{\nu_2\}) \to (f, o_1)\ \texttt{jfieldID} \\
&\qquad \text{where } o_1 = \{\nu_1; \langle f \rangle \circ \phi_1; \mu_1\} \text{ and } f = \nu_f : \{\nu_2; \phi_2; \mu_2\} \\
\texttt{GetMethodID}: &\ \forall \nu_i, \nu_m, \phi_i, \mu_i, \psi. \\
&\quad (o_1\ \text{JClass}\ \texttt{jobject} \times \texttt{str}\{\nu_m\} \times \texttt{str}\{\nu_2\}) \to (m, o_1)\ \texttt{jmethodID} \\
&\qquad \text{where } o_1 = \{\nu_1; \phi_1; \langle m \rangle \circ \mu_1\} \text{ and } m = \nu_m : [\nu_2; \psi] \\
\texttt{GetObjectField}: &\ \forall \nu_i, \nu_f, \phi_i, \mu_i.\ (o_1\ \texttt{jobject} \times (f, o_1)\ \texttt{jfieldID}) \to o_2\ \texttt{jobject} \\
&\quad \text{where } o_1 = \{\nu_1; \langle f \rangle \circ \phi_1; \mu_1\},\ f = \nu_f : o_2, \text{ and } o_2 = \{\nu_2; \phi_2; \mu_2\} \\
\texttt{CallObjectMethod}_n: &\ \forall \iota_i, \nu_i, \nu_m, \phi_i, \mu_i.\ (o_1\ \texttt{jobject} \times (m, o_1)\ \texttt{jmethodID} \times \\
&\qquad \iota_1\ \texttt{jobject} \times \cdots \times \iota_n\ \texttt{jobject}) \to o_2\ \texttt{jobject} \\
&\quad \text{where } o_1 = \{\nu_1; \phi_1; \langle m \rangle \circ \mu_1\},\ m = \nu_m : (\iota_1 \times \cdots \times \iota_n) \to o_2, \text{ and } o_2 = \{\nu_2; \phi_2; \mu_2\}
\end{aligned}$$

Fig. 18. Sample JNI type signatures

$$\begin{aligned}
\theta(C) &= \{\text{“}C\text{”}; \\
&\quad \langle \text{“}f_1\text{”} : \theta(C_{f_1}); \cdots \rangle \circ \emptyset; \\
&\quad \langle \text{“}m_1\text{”} : \theta((C_{p_1} \ldots C_{p_n})C_r); \cdots \rangle \circ \emptyset\} \\
&\qquad \text{where } C \text{ is declared as} \\
&\qquad \text{“class } C\ \{C_{f_1}\ f_1;\ \ldots C_r\ m_1(C_{p_1}\ p_1, \ldots, C_{p_n}\ p_n); \ldots\}\text{”} \\
\theta((C_1 \ldots C_n)C_r) &= [\text{“}(C_1 \ldots C_n)C_r\text{”}; \theta(C_1) \times \cdots \theta(C_n) \to C_r]
\end{aligned}$$

Fig. 19. Translation rules for Java types

sets of the resulting type are universally quantified, our type checking system requires that representational types are consistent with types from Java. Hence if ν is instantiated to a known string, then ϕ and μ will be constrained appropriately.

`GetObjectClass` takes an instance object as a parameter and returns the class of that object. The function `GetFieldID` takes a class object and two strings describing a field (ν_f) and its type (ν_2) and returns the `jfieldID` for the field ν_f in that class. Note that the type of the class object and the object type embedded in the `fieldID` must be the same. Also, the object type o_1 is open, because it may have other fields in addition to ν_f. Similarly, `GetMethodID` returns the `methodID` for the named method and type. The signature of the method must match ν_2, the string passed as the second argument. `GetObjectField` takes an object and a `jfieldID` with matching types and extracts the contents of the field described by the `jfieldID`. `CallObjectMethod` is similar, calling the method described by the `jmethodID`.

4.2 Type Checking Glue Code and Soundness

Since our source language is close to lambda calculus, type checking for J-Saffire is fairly standard. The one catch is types of the form $\{s; F; M\}$ or $[s; sig]$. Recall that in these types, s is a field descriptor (a class name, in our formalism) or a method

$\boxed{\Gamma \vdash e : \mathtt{ct}}$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\textsc{Var}) \qquad \frac{\Gamma[x \mapsto \mathtt{ct}] \vdash e : \mathtt{ct}'}{\Gamma \vdash \lambda x.e : \mathtt{ct} \to \mathtt{ct}'}\ (\textsc{Lam}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{ct} \to \mathtt{ct}' \qquad \Gamma \vdash e_2 : \mathtt{ct}}{\Gamma \vdash e_1\ e_2 : \mathtt{ct}'}\ (\textsc{App})$$

$$\frac{}{\Gamma \vdash \text{“}Str\text{”} : \mathtt{str}\{\text{“}Str\text{”}\}}\ (\textsc{String}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{ct}_1 \qquad fv(\Gamma) \cap \vec{\vartheta} = \emptyset \qquad \Gamma[f \mapsto \forall \vec{\vartheta}.\mathtt{ct}_1] \vdash e_2 : \mathtt{ct}_2}{\Gamma \vdash \text{let } f = e_1 \text{ in } e_2 : \mathtt{ct}_2}\ (\textsc{Let})$$

$$\frac{\Gamma(f) = \forall \vec{\vartheta}.\mathtt{ct}}{\Gamma \vdash f_i : \mathtt{ct}[\vec{\vartheta} \mapsto \vec{t}]}\ (\textsc{Inst}) \qquad \frac{T(\delta) = \forall \vec{\vartheta}.(\mathtt{ct}_1, \ldots, \mathtt{ct}_n) \to \mathtt{ct} \qquad \Gamma \vdash e_j : \mathtt{ct}_j[\vec{\vartheta} \mapsto \vec{t}] \qquad j \in 1..n}{\Gamma \vdash \delta_i(e_1, \ldots, e_n) : \mathtt{ct}[\vec{\vartheta} \mapsto \vec{t}]}\ (\textsc{Delta})$$

Fig. 20. Type checking rules

descriptor, respectively. When s is a known string, these types only make sense if they correspond to the actual classes from Java. We formalize this by defining a function θ in Fig. 19 that translates field and method descriptors into our multilingual type grammar, assuming we have all necessary class declarations. A field descriptor C is translated to a representational type with the appropriate field and method sets. Method descriptors, which contain parentheses to distinguish them from field descriptors, are translated into a method type with the corresponding signature. Note that we elide the details of separators between class names in method signatures.

We say that $\{s; F; M\}$ or $[s; sig]$ is *well-formed* if either s is a variable, or if $s =$ *“str”* and either $\{s; F; M\} = \theta(str)$ or $[s; sig] = \theta(str)$, respectively. In our type checking system, we implicitly assume that all types are well-formed.

Fig. 20 gives our type checking rules. Like our O-Saffire checking rules, we assume we have concrete, multilingual type information for the program. The rules (Var), (Lam), and (App) are standard. In (String), string constants are given the corresponding singleton string type. Rule (Let) introduces Hindley-Milner style universal types, which are instantiated by (Inst). In (Inst), we use t to range over the kinds of types variables may be instantiated to ($\mathtt{ct}$, s, jt, F, sig, and M), and we implicitly assume that type variables are always instantiated to the right kind. In the checking system, the index i on an occurrence of f is not used. Finally, JNI functions are typed using the (Delta) rule. We write $T(\delta)$ for the type scheme for JNI function δ, e.g., as shown in Fig. 18. Since these functions are not curried, (Delta) combines both (Inst) and (App) to type check the application of the JNI function.

Soundness. We now sketch a proof of soundness for our type checking system. The first step is to extend our language so that we can represent the values returned by JNI functions. Fig. 21(a) shows our source language with values v called out, and with four new kinds of values. *object*(C) represents an instance of class C. Here we use a completely opaque representation of the Java object, which is sufficient

$$
\begin{array}{rcl}
e & ::= & v \mid x \mid f_i \mid \delta_i(e, \ldots, e) \mid e\ e \mid \text{let } f = e \text{ in } e \\
v & ::= & \lambda x.e \mid \text{“Str”} \mid \mathit{object}(C) \mid \mathit{class}(C) \mid \mathit{fid}(F, C_F, C) \mid \mathit{mid}(M, S_M, C) \\
\delta & ::= & \ldots
\end{array}
$$

(a) Source language with new values

$$
\frac{o = \theta(C)}{\Gamma \vdash \mathit{object}(C) : o\ \texttt{jobject}}\ (\textsc{Obj}) \qquad \frac{o = \theta(C)}{\Gamma \vdash \mathit{class}(C) : o\ \mathsf{JClass}\ \texttt{jobject}}\ (\textsc{Class})
$$

$$
\frac{f = \text{“}F\text{”} : \theta(C_F) \qquad o = \theta(C) \qquad f \in o}{\Gamma \vdash \mathit{fid}(F, C_F, C) : (f, o)\ \texttt{jfieldID}}\ (\text{FID}) \qquad \frac{m = \text{“}M\text{”} : \theta(S_M) \qquad o = \theta(C) \qquad m \in o}{\Gamma \vdash \mathit{mid}(M, S_M, C) : (m, o)\ \texttt{jmethodID}}\ (\text{MID})
$$

(b) New type rules

Fig. 21. Language and type rules with opaque Java values

because these values may only be manipulated via JNI functions. $\mathit{class}(C)$ represents an instance of java.lang.Class that is known to describe class C. $\mathit{fid}(F, C_F, C)$ represents a field identifier for a field named F of type C_F inside of class C. Finally, $\mathit{mid}(M, S_M, C)$ represents a method identifier for the method with name M and method descriptor S_M that resides inside class C.

Fig. 21(b) gives the type checking rules for these four new values. In all of the type rules, we use θ to return the corresponding representational type or method type for a given class name. (Obj) and (Class) are straightforward. In (FID), we write $f \in o$ to mean that the field typing f is present in object description o, and similarly for $m \in o$ in (MID).

We give an operational semantics for our language in Fig. 22. Part (a) of this figure defines reduction contexts, with the hole [] specifying which subexpression to evaluate next. Part (b) defines a reduction relation of the form $R[e] \to R[e']$, where the expression $R[e]$ evaluates to the expression $R[e']$, and we write $\to^*$ for the reflexive, transitive closure of $\to$.

The rules (β) and (let) are standard. The remaining rules describe the behavior of JNI functions. Since Java objects are opaque in our system, our operational semantics for JNI functions simply produce new opaque objects of the right type. The rule (δ-FC) converts a string into the corresponding Java class object, assuming the class exists in Java. The rule (δ-GOC) returns the class of an instance object. Rules (δ-$GFID$) and (δ-$GMID$) retrieve the field ID or method ID of a class member, creating the appropriate opaque object. These rules only apply if the field or method exists in the object's class, which we test using θ. (Using θ is not necessary, but it is convenient, because it already can process both field and method descriptors.) Rule (δ-GOF) extracts the field described by its second argument from its first argument. Notice that the field must exist in the class, and the class of the field identifier must match the object. Similarly, rule (δ-COM) invokes a Java method of arity n from the instance class. That method must exist in the class, and all the arguments must be of the right type (we do not permit subtyping). Note that although instances of java.lang.Class are objects, and hence we could allow a $\mathit{class}(C)$ to be used wherever an $\mathit{object}(\mathit{java.lang.Class})$ is expected, our semantics

$$R ::= [] \mid R\ e \mid v\ R \mid \text{let } f = R \text{ in } e \mid \delta(R, e, \ldots, e) \mid \delta(v, R, \ldots, e) \mid \cdots$$

(a) Reduction Contexts

$$(\beta) \quad R[(\lambda x.e)\ v] \rightarrow R[e[x \mapsto v]]$$

$$(let) \quad R[\text{let } f = v \text{ in } e] \rightarrow R[e[f \mapsto v]]$$

$$(\delta\text{-}FC) \quad R[\texttt{FindClass}_i(\text{“}C\text{”})] \rightarrow R[class(C)] \quad \text{if } C \text{ is a declared Java class}$$

$$(\delta\text{-}GOC) \quad R[\texttt{GetObjectClass}_i(object(C))] \rightarrow R[class(C)]$$

$$(\delta\text{-}GFID) \quad R[\texttt{GetFieldID}_i(class(C), \text{“}F\text{”}, \text{“}C_F\text{”})] \rightarrow R[fid(F, C_F, C)] \quad \text{if } C \text{ is declared as “class } C\ \{\ldots C_F\ F; \ldots\}\text{”}$$

$$(\delta\text{-}GMID) \quad R[\texttt{GetMethodID}_i(class(C), \text{“}M\text{”}, \text{“}S_M\text{”})] \rightarrow R[mid(M, S_M, C)] \quad \text{if } S_M = (C_1 \ldots C_n)C_r \text{ and } C \text{ is declared as “class } C\ \{\ldots C_r\ M(C_1\ p_1, \ldots, C_n\ p_n); \ldots\}\text{”}$$

$$(\delta\text{-}GOF) \quad R[\texttt{GetObjectField}_i(object(C), fid(F, C_F, C))] \rightarrow R[object(C_F)] \quad \text{if } C \text{ is declared as “class } C\ \{\ldots C_F\ F; \ldots\}\text{”}$$

$$(\delta\text{-}COM) \quad R[\texttt{CallObjectMethod}_{ni}(object(C), mid(M, S_M, C), v_1, \ldots, v_n)] \rightarrow R[object(C_r)] \quad \text{if } S_M = (C_1 \ldots C_n)C_r \text{ and } C \text{ is declared as “class } C\ \{\ldots C_r\ M(C_1\ p_1, \ldots, C_n\ p_n); \ldots\}\text{”} \text{ and } \forall i.v_i = object(C_i)$$

(b) Small-step Semantics

Fig. 22. Operational semantics

forbids this for simplicity.

Finally, we can show soundness by proving progress and preservation. We begin by presenting the usual substitution lemmas; we omit the proofs of these lemmas, because they are completely standard.

Lemma 4.1 Substitution. *If* $\Gamma[x \mapsto \texttt{ct}'] \vdash e : \texttt{ct}$ *and* $\Gamma \vdash e' : \texttt{ct}'$ *then* $\Gamma \vdash e[x \mapsto e'] : \texttt{ct}$.

Lemma 4.2 Polymorphic Substitution. *If* $\Gamma[f \mapsto \forall\vec{\beta}.\texttt{ct}'] \vdash e : \texttt{ct}$ *and* $\Gamma \vdash e' : \texttt{ct}'$ *and* $fv(\Gamma) \cap \vec{\beta} = \emptyset$ *then* $\Gamma \vdash e[f \mapsto e'] : \texttt{ct}$.

Given these lemmas, we can prove that types are preserved under reduction in the semantics. Note that unlike our proof for O-Saffire, for J-Saffire, here we prove progress and preservation separately.

Lemma 4.3 Preservation. *If* $\Gamma \vdash e : \texttt{ct}$ *and* $e \rightarrow e'$, *then* $\Gamma \vdash e' : \texttt{ct}$.

Proof Sketch. By induction on the structure of e. If a subterm inside of e is reduced, then we apply induction. Otherwise we proceed by case analysis on the reduction. For (β) we use the substitution lemma, and for (let) we use the polymorphic substitution lemma. Otherwise $e = \delta(v_1, \ldots, v_n)$.

If the reduction was $(\delta\text{-}FC)$, then we have $e \rightarrow class(C)$, and using the type of

`FindClass` from Fig. 18 with (Delta), we have

$$\frac{\begin{array}{l}\text{(Delta)}\\ T(\delta) = \forall \nu, \phi, \mu.(\texttt{str}\{\nu\}) \to \{\nu; \phi; \mu\}\ \mathsf{JClass}\ \texttt{jobject} \qquad \Gamma \vdash \text{“}C\text{”} : \texttt{str}\{\text{“}C\text{”}\}\end{array}}{\Gamma \vdash \texttt{FindClass}_i(\text{“}C\text{”}) : \{\text{“}C\text{”}; \phi'; \mu'\}\ \mathsf{JClass}\ \texttt{jobject}}$$

for some ϕ' and μ'. By our assumption of well-formedness, $\{\text{“}C\text{”}; \phi'; \mu'\} = \theta(C)$. And since by (Class) we have $\Gamma \vdash \mathit{class}(C) : \theta(C)\ \mathsf{JClass}\ \texttt{jobject}$, we then have $\Gamma \vdash \mathit{class}(C) : \{\text{“}C\text{”}; \phi'; \mu'\}\ \mathsf{JClass}\ \texttt{jobject}$, which is what we wanted to show.

The other reduction steps are similar. For (δ-*GOC*), we again use (Class) to show that the class object produced by reduction has the same type as is returned by the JNI call. For (δ-*GFID*) and (δ-*GMID*), we use (FID) and (MID), respectively. Lastly, for (δ-*GOF*) and (δ-*COM*), we use (Obj) to show that the object produced by reduction has the type we expect. There is surprisingly little to show for these last two cases; because we assume we have taken a reduction step, we need not check anything except the type of the retrieved field or method result. □

Next, we show that for any well typed expression e, one of the reduction rules can always be applied:

Lemma 4.4 Progress. *For every closed expression e, if $\Gamma \vdash e : \texttt{ct}$, then either e is a value, or there exists an e' such that $e \to e'$.*

Proof Sketch. By induction on the structure of e. If e is a value, then we are done. If e is an application $e_1\ e_2$, then we either apply induction or show that e_1 must be a function based on its type, and hence we can take a step with (β). If e is a let, then we either use induction or take a step with (*let*). Otherwise e is of the form $\delta(e_1, \ldots, e_n)$. If some e_i is not a value, we can apply induction. Otherwise, e is of the form $\delta(v_1, \ldots, v_n)$, and we proceed by case analysis on the JNI function δ. We illustrate two of the cases, which are all very similar.

If $\delta = \texttt{FindClass}$, then by assumption we have

$$\frac{\begin{array}{l}\text{(Delta)}\\ T(\delta) = \forall \nu, \phi, \mu.(\texttt{str}\{\nu\}) \to \{\nu; \phi; \mu\}\ \mathsf{JClass}\ \texttt{jobject} \qquad \Gamma \vdash v : \texttt{str}\{s\}\end{array}}{\Gamma \vdash \texttt{FindClass}_i(v) : \{s; \phi'; \mu'\}\ \mathsf{JClass}\ \texttt{jobject}}$$

for some ϕ' and μ'. Since v is a value, it must be of the form “C”, and thus $s =$ “C” by (String). But then by our well-formedness assumption, $\{s; \phi'; \mu'\} = \{\text{“}C\text{”}; \phi'; \mu'\} = \theta(C)$, and hence C must be a valid Java class. But then we can take a step using (δ-*FC*).

If $\delta = \texttt{GetObjectField}$, then by assumption we have

$$\frac{\begin{array}{l}\text{(Delta)}\\ \begin{array}{c} T(\delta) = \forall \nu_i, \nu_f, \phi_i, \mu_i.\ (o_1\ \texttt{jobject} \times (f, o_1)\ \texttt{jfieldID}) \to o_2\ \texttt{jobject} \\ o_1 = \{\nu_1; \langle f \rangle \circ \phi_1; \mu_1\} \end{array} \\ f = \nu_f : o_2 \qquad o_2 = \{\nu_2; \phi_2; \mu_2\} \qquad \Gamma \vdash v_1 : o_1'\ \texttt{jobject} \qquad \Gamma \vdash v_2 : (f', o_1')\ \texttt{jfieldID}\end{array}}{\Gamma \vdash \texttt{GetObjectField}_i(v_1, v_2) : o_4}$$

Since the v_i are values, we have $v_1 = \mathit{object}(C)$ and $v_2 = \mathit{fid}(F, C_F, C')$, for some C, F, C_F, and C'. By (Obj), we have $o_1' = \theta(C)$, and by (FID) we have $o_1' = \theta(C')$, and thus $C = C'$. Then also (FID) we have $\text{“}F\text{”} : \theta(C_F) \in \theta(C)$, and thus class C has a field F of type C_F. Therefore we can take a step with (δ-*GOF*). □

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\textsc{Var}) \qquad \frac{\Gamma[x \mapsto \alpha] \vdash e : \mathtt{ct} \qquad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : \alpha \to \mathtt{ct}}\ (\textsc{Lam}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{ct} \qquad \Gamma \vdash e_2 : \mathtt{ct}' \qquad \mathtt{ct} = \mathtt{ct}' \to \alpha \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\ e_2 : \alpha}\ (\textsc{App})$$

$$\frac{}{\Gamma \vdash \text{“}Str\text{”} : \mathtt{str}\{\text{“}Str\text{”}\}}\ (\textsc{String}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{ct}_1 \qquad \Gamma[f \mapsto (\mathtt{ct}_1, fv(\Gamma)] \vdash e_2 : \mathtt{ct}_2}{\Gamma \vdash \text{let } f = e_1 \text{ in } e_2 : \mathtt{ct}_2}\ (\textsc{Let})$$

$$\frac{\Gamma(f) = (\mathtt{ct}, \vec{\vartheta}) \qquad \vec{\vartheta} \preceq_i \vec{\vartheta} \qquad \mathtt{ct} \preceq_i \alpha \qquad \alpha \text{ fresh}}{\Gamma \vdash f_i : \alpha}\ (\textsc{Inst}) \qquad \frac{T(\delta) = (\mathtt{ct}_1, \ldots, \mathtt{ct}_n) \to \mathtt{ct} \qquad \Gamma \vdash e_i : \mathtt{ct}'_i \qquad \mathtt{ct}_i \preceq_i \mathtt{ct}'_i \qquad \mathtt{ct} \preceq_i \alpha \qquad \alpha \text{ fresh}}{\Gamma \vdash \delta_i(e_1, \ldots, e_n) : \alpha}\ (\textsc{Delta})$$

Fig. 23. Type rules modified for inference

Finally, we state our soundness theorem:

THEOREM 4.5 SOUNDNESS. *If* $\Gamma \vdash e : \mathtt{ct}$*, then there exists a value* v *such that* $e \to^* v$ *and* $\Gamma \vdash v : \mathtt{ct}$*.*

PROOF. The result follows directly from Lemma 4.3 and 4.4. □

Full proof details are available in a companion technical report [Furr and Foster 2006a].

4.3 Type Inference for the JNI

Since our type system for the JNI is flow-insensitive, we use conventional notation to present our inference algorithm. Fig. 23 shows our type inference rules. We perform type inference by applying these rules to our program and generating a set of *constraints*, which are given in the hypotheses of the rules. We then solve the constraints to find a valid typing for the program, if any.

We discuss the rules briefly. Rule (VAR) is the same as the checking rule. In rule (LAM), we create a fresh variable for the domain of the function type, and in rule (APP) we generate an *equality constraint* to ensure the function's domain type matches its actual argument type. Notice that, as was the case in the type checking system, we require equality of types and have no subsumption. Thus we could fail to unify two Java objects when one object is a subtype of the other. We did not find this to be a problem in practice, however (see Section 5.3). Rule (STRING) gives a singleton string type to the expression.

In our type checking system for the JNI, we introduce polymorphism when typing the expression let $f = e_1$ in e_2. To infer a type for this expression, we would normally first infer a type for e_1, then generalize its type to a type scheme $\forall\vec{\vartheta}.\mathtt{ct}$, which we would then bind to f while typing e_2. However, while this approach is possible to use for C, in our experience it is awkward in practice. C code is not structured as nested let blocks. Instead, function definitions in C are all at the top level and may call each other arbitrarily, assuming appropriate prototypes are supplied to satisfy the compiler's declaration-before-use requirement. Using the standard approach to

Program and Types	Constraints	Substitutions and Derived Constrs
$\lambda y.\text{let } f = \lambda x.(x, y) \text{ in}$ $\quad f_2\ (f_1\ \text{“}s\text{”})$ $y\ :\ \beta$ $f\ :\ (\alpha \to \alpha \times \beta, \{\beta\})$	$\alpha \to \alpha \times \beta\ \preceq_1\ \texttt{str}\{\text{“}s\text{”}\} \to \gamma$ $\beta\ \preceq_1\ \beta$ $\alpha \to \alpha \times \beta\ \preceq_2\ \gamma \to \eta$ $\beta\ \preceq_2\ \beta$	$S_1\ =\ \{\alpha \mapsto \texttt{str}\{\text{“}s\text{”}\}; \beta \mapsto \beta\}$ $\gamma\ =\ \texttt{str}\{\text{“}s\text{”}\} \times \beta$ $S_2\ =\ \{\alpha \mapsto \texttt{str}\{\text{“}s\text{”}\} \times \beta; \beta \mapsto \beta\}$ $\eta\ =\ (\texttt{str}\{\text{“}s\text{”}\} \times \beta) \times \beta$

Fig. 24. Polymorphism example

let-polymorphism, we would need to discover the dependencies between functions and then visit the functions in the program in the appropriate order.

Instead, we take an approach based on semiunification [Henglein 1993; Fähndrich et al. 2000], which we think is a more elegant solution and is easier in practice. In rule (Let), we represent a polymorphic type as a pair $(\texttt{ct}, \vec{\vartheta})$, where $\texttt{ct}$ is the base type and $\vec{\vartheta}$ is the set of variables that may *not* be quantified. For comparison, the pair $(\texttt{ct}, \vec{\vartheta})$ represents the type scheme $\forall\overrightarrow{\vartheta'}.\texttt{ct}$ where $\vartheta' = fv(\texttt{ct}) - \vartheta$. The key feature of this new version of (Let) is that, in practice, we need not know the full type of the expression e_1 (whose type is generalized[3]) when we form this pair. We could initially pick its type to be a fresh variable, which we then equate with the actual type of e_1 later on during typing, and inference would still proceed correctly.

In rule (Inst), we need to instantiate a polymorphic type $(\texttt{ct}, \vec{\vartheta})$. Since in practice we may not know the type of $\texttt{ct}$ yet, we generate *instantiation constraints* relating the base type to this particular instance, which occurs at instantiation site i. Instantiation constraints have the form $a\ \preceq_i\ b$, where a and b range over $\texttt{ct}$, jt, etc. This constraint says that there must exist a substitution S_i such that $S_i(a) = b$. Thus in rule (Inst), we generate a constraint $\texttt{ct} \preceq_i \alpha$ to require that there is some substitution S_i such that $S_i(\texttt{ct}) = \alpha$. We also need to ensure that non-quantifiable variables are not instantiated, which we enforce with the *self-instantiation* constraint $\vec{\vartheta}\ \preceq_i\ \vec{\vartheta}$. This constrains the substitution S_i to rename every variable in ϑ to itself, i.e., to not rename $\vec{\vartheta}$. Note that two constraints $a \preceq_i b$ and $c \preceq_j d$ do not interact if $i \neq j$.

The last rule, (Delta), simply combines (Inst) and (App) to apply a JNI function to its arguments at instantiation site i. Here we assume that T maps each JNI function to a plain type in which all variables are quantified, and hence we write its type as just $(\texttt{ct}_1, \ldots, \texttt{ct}_n) \to \texttt{ct}$ instead of pairing it with $\emptyset$, and we do not generate a self-instantiation constraint.

A full discussion of this style of polymorphic type inference is beyond the scope of this paper and can be found elsewhere [Henglein 1993]. However, we illustrate the basic idea with the simple program shown in the left box in Fig. 24. Note that we have numbered the instantiation sites, and we assume we have added pairs to the language for the sake of exposition. The program begins with the introduction of the lambda-bound variable y, which has type β as shown. The variable y is free in the scope of the enclosed let and thus cannot be quantified. Therefore we assign f the type $(\alpha \to \alpha \times \beta, \{\beta\})$ with (Let), also as shown.

[3]In practice, the only place we generalize a type is at function boundaries and therefore implicitly enforce the value restriction required for soundness in the presence of mutation.

Within the body of the let, we use (Inst) and (App) for each function application, generating the constraints shown in the middle box of the figure. At instantiation site 1, we generate the constraints $\alpha \to \alpha \times \beta \preceq_1 \texttt{str}\{\textit{"s"}\} \to \gamma$ and $\beta \preceq_1 \beta$. (We've simplified away an extra equality constraint here.) Then at instantiation site 2, we require that this second instance of f instantiate to type $\gamma \to \eta$, where η is a fresh variable, so that it takes as input the output of the first application of f.

To solve these constraints, we need to find substitutions S_1 and S_2 for the instantiation sites and solve any implied equality constraints. We describe our formal algorithm for this below, and here we reason informally. The results are shown in the right box of the figure. First, we observe that instantiation 1 replaces α with $\texttt{str}\{\textit{"s"}\}$ and β with itself, giving us S_1 as shown. Then since $S_1(\alpha \times \beta) = \gamma$, we can conclude that γ has the form $\texttt{str}\{\textit{"s"}\} \times \beta$. Next we observe that instantiation 2 replaces α with γ, and thus it replaces α with $\texttt{str}\{\textit{"s"}\} \times \beta$. And as before, $S_2(\beta) = \beta$ by the self-instantiation constraint. Then since $S_2(\alpha \times \beta) = \eta$, we have $(\texttt{str}\{\textit{"s"}\} \times \beta) \times \beta = \eta$.

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` or global structures containing fields of type `jobject`. In general we have found that programmers use few globals when interacting with the JNI. When encountering a global of type `char *`, our implementation does track their contents and therefore they can not be used in JNI function calls as special strings. We also do not check for global variables of type `jfieldID` or `jmethodID` in our current implementation.

Constraint Resolution. Our type inference rules generate unification constraints (rule (App)) and instantiation constraints (rules (Inst) and (Delta)). To solve these constraints, we use a variant of Fähndrich et al.'s worklist algorithm for semi-unification [Fähndrich et al. 2000].

To simplify the constraint resolution rules we present 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. Also, 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 are still well-formed. In particular, if we ever form a type $\{\textit{"C"}; F; M\}$, we check that F and M are the correct field and method sets for class C, and if we form a method signature $[\textit{"}S_M\textit{"}; \textit{sig}]$, we ensure that *sig* matches method descriptor S_M.

To improve our error reporting, we also enforce a finer well-formedness condition on representational types whose class names are not known. In particular, we require that in a field set, any two fields with the same name must have the same type.[4] 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, where the $=$ on *fname* is syntactic equality. In other words, $\textit{"s"} = \textit{"t"}$ if $s = t$, and $\nu = \nu'$ if ν and ν' are the same variable. Whenever we replace

[4]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.

$$
\begin{array}{rll}
\text{(Closure)} & C \cup \{a \preceq_i b\} \cup \{a \preceq_i c\} \quad \Rightarrow & C \cup \{a \preceq_i b\} \cup \{b = c\} \\
\text{(Obj InEq)} & C \cup \{\{s; F; M\} \preceq_i jt\}\} \quad \Rightarrow & \\
 & \quad C \cup \{jt = \{\nu; \phi; \mu\}\} \cup \{s \preceq_i \nu\} \cup \{F \preceq_i \phi\} \cup \{M \preceq_i \mu\} & \nu, \phi, \mu \text{ fresh} \\
\text{(FieldSet InEq)} & C \cup \{\langle f_1; \ldots; f_n\rangle \circ F \preceq_i F'\} \Rightarrow & \\
 & \quad C \cup \{\langle f_1'; \ldots; f_n'\rangle \circ \phi = F'\} \cup \{F \preceq_i \phi\} \cup \{f_1 \preceq_i f_1'\} \cup \cdots \cup \{f_n \preceq_i f_n\} & \\
 & \qquad \text{where } f_i' = \nu_i : \iota_i \text{ with } \phi, \nu_i, \iota_i \text{ fresh} & \\
\text{(MethSet InEq)} & C \cup \langle m_1; \ldots; m_n\rangle \circ M \preceq_i M' \Rightarrow C \cup \{\langle m_1'; \ldots; m_n'\rangle \circ \mu = M'\} & \\
 & \qquad \cup \{M \preceq_i \mu\} \cup \{m_1 \preceq_i m_1'\} \cup \cdots \cup \{m_n \preceq_i m_n'\} & \\
 & \qquad \text{where } m_i' = \nu_i : [\nu_i'; \psi_i] \text{ with } \mu, \nu_i, \nu_i', \psi_i \text{ fresh} & \\
\text{(Str Resolve)} & C \cup \{\nu = \text{“}Str\text{”}\} \quad \Rightarrow & C[\nu \mapsto \text{“}Str\text{”}] \\
 & & \text{and verify well-formedness} \\
\text{(Str Sub)} & C \cup \{\nu_1 = \nu_2\} \quad \Rightarrow & C[\nu_1 \mapsto \nu_2] \\
 & & \text{and verify well-formedness} \\
\text{(Str Eq)} & C \cup \{\text{“}Str_1\text{”} = \text{“}Str_2\text{”}\} \quad \Rightarrow & C \qquad Str_1 = Str_2 \\
\text{(Str Neq)} & C \cup \{\text{“}Str_1\text{”} = \text{“}Str_2\text{”}\} \quad \Rightarrow & \mathit{error} \qquad Str_1 \neq Str_2
\end{array}
$$

Fig. 25. Selected constraint rewrite rules

a ν by a constant string during constraint resolution, we re-check well-formedness and generate any implied equality constraints. Methods, however, unlike fields, may be overloaded in Java, and so we do not apply the above well-formedness condition to method sets.

We express constraint solving in terms of rewrite rules, shown in Fig. 25. 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 straightforward, and so Fig. 25 contains only the interesting cases. The exhaustive set of rules may be found in our companion technical report [Furr and Foster 2006a].

In Fig. 25, the (Closure) rule unifies two terms b and c when they are both instantiations of the same variable a at the same instantiation site. This rule enforces the property that substitution S_i must replace variable a consistently [Henglein 1993]. The rule (Obj 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 (Obj Ineq), we unify jt with $\{\nu; \phi; \mu\}$, where ν, ϕ, and μ are fresh variables, and then propagate the semi-unification constraint to the components.

In (FieldSet InEq), we instantiate one field set to another. For each field typing f_i in the field set on the left-hand side of the constraint, we make a new typing f_i' with free variables. Then we unify F' with a new field set containing the f_i' composed with a fresh variable ϕ, and the propagate the instantiation constraint to produce $F \preceq_i \phi$ and $f_i \preceq_i f_i'$ for all i. If it turns out that this creates any duplicate field in the field set F', then our well-formedness condition will later unify them.

(MethSet InEq) behaves similarly to (FieldSet InEq), except it creates fresh method signatures m_i' and propagates the constraints to them. Recall that we

do not have a well-formedness condition on method names. To understand why, suppose an open method set $\langle x : \iota \to \iota \rangle \circ \mu$ is unified with a closed method set $\langle x : Obj_1 \to Obj_1; x : Obj_2 \to Obj_2 \rangle \circ \emptyset$. Since the method x is overloaded, we do not know if ι should unify with Obj_1 or Obj_2. Therefore, if this occurs during unification, J-Saffire emits a warning and removes the constraint. However, our unification algorithm does allow method sets with multiple copies of the same method signature to unify as long as they contain no free type variables. We could also unify return types of methods with otherwise equal signatures, but do not do so in our current implementation.

The last four rules handle strings. (Str Resolve) replaces a string variable by a constant, which triggers a well-formedness check on all types. In particular, for representational types $\{s; F; M\}$, if s becomes a known constant we unify F and M with the corresponding field and method sets, and similarly for $[s; sig]$. We also ensure that the same name appears at most once in a field set. (In practice, only those representational types and field sets that may be affected are verified). (Str Sub) replaces one string variable by another, and again, we check well-formedness. (Str Eq) and (Str Neq) unify string constants.

The remainder of the rewrite rules (not shown) either replace type variables with terms, match up field or method sets, or match up type constructors and propagate constraints recursively to the constructor parameters.

Example. We illustrate our type inference system on the `my_getObjectField` function from Fig. 16. We begin by assigning each parameter a fresh type:

$$\texttt{obj} : \alpha_{obj} \qquad \texttt{field} : \alpha_{field}$$

The second line of the function calls the `GetObjectClass` function. After looking up its type in the environment (shown in Fig. 18 with quantified type variables ν, ϕ, and μ) and assigning its return type the fresh type α_{cls}, we add the following constraints for the function call using (Delta):

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

In order for a substitution to map the type variables on the left hand side of the constraint, the type on the right hand side must have the same shape as the type on the left. Therefore, our constraint rewrite rules first unify α_{obj} and α_{cls} with fresh types of the correct shape, and then propagate the instantiation constraints to the new types:

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

Then rule (Closure) in Fig. 25 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_getObjectField` calls `GetFieldID`, and after applying our type inference

and constraint rewrite rules, we discover (among other things):

$$\begin{array}{rcll} \alpha_{field} & = & \texttt{str}\{\nu_{field}\} & \nu_{field} \text{ fresh} \\ \phi_2 & = & \langle \nu_{field} : \{ \textit{“java.lang.Object”}; \ldots \} \rangle \circ \phi_4 & \phi_4 \text{ fresh} \end{array}$$

The last call to `GetObjectField` generates several new constraints, but they do not affect the types. Thus after `my_getObjectField` has been analyzed, it is given the type

$$\begin{array}{c} \{\nu_2; \langle \nu_{field} : \{ \textit{“java.lang.Object”}; \ldots \} \rangle \circ \phi_4; \mu_2\}\ \texttt{jobject} \rightarrow \texttt{str}\{\nu_{field}\} \\ \rightarrow \{ \textit{“java.lang.Object”}; \ldots \}\ \texttt{jobject} \end{array}$$

In other words, this function accepts any object as the first parameter as long as it has a field with type java.lang.Object whose name is given by the second parameter, exactly as intended.

5. IMPLEMENTATIONS

Next we describe our implementations of O-Saffire and J-Saffire, as well as the results of applying them to a variety of benchmarks. Section 6 compares and contrasts the two systems.

5.1 O-Saffire

Our implementation of O-Saffire consists of two separate tools, one for OCaml and one for C. The first tool uses the `camlp4` preprocessor to analyze OCaml source programs, extract the type signatures of any foreign functions, and then apply the type translation function Φ discussed in Section 3.4. Because ultimately C foreign functions will see the physical representations of OCaml types, O-Saffire resolves all types to a concrete form. In particular, type aliases are replaced by their base types, and opaque types are replaced by the concrete types they hide, when available. If the concrete type is not available, the opaque type is assigned a fresh type variable, and O-Saffire simply checks to ensure it is used consistently. As each OCaml source file is analyzed, O-Saffire incrementally updates a central type repository with the newly extracted type information, beginning with a pre-generated repository from the standard OCaml library.

The second tool, built using CIL [Necula et al. 2002], performs the bulk of the analysis. This part of O-Saffire takes as input the central type repository and a set of C source programs to which it applies the type inference algorithm from Section 3.4. In order to analyze constructs such as `if_unboxed`, O-Saffire uses syntactic pattern matching to identify tag and boxedness tests in the code. In particular, O-Saffire recognizes comparisons `if ((e & 1) == 0)` and its negation as tests for boxedness. O-Saffire also recognizes several patterns for checking the value of a primitive type or a tag, including `if(e == n)` and `if (e-1)` (a comparison to zero). Other patterns O-Saffire recognizes include tests `if(Int_val(e) == 0)`, `if (e == Val_unit)`, `if (e == Val_true)`, and `if (e != Val_int(0))`. Programmers often use this last test to check for a non-empty list or a supplied option value. All of these patterns are also recognized when prefixed by the unary negation (`!`) operator.

One feature of C that we have not fully discussed is the address-of operator. Our implementation models address-of in different ways, depending on the usage. Any local variable with an integer type (or local structure with a integer field) that has its

address computed is given the type $\texttt{int}\{\top, 0, \top\}$ everywhere. This conservatively models the fact that the variable may be updated arbitrarily through other aliases. It has been our experience that variables used for indexing into `value` types rarely have their address taken, so this usually does not affect our analysis. Similarly, we produce a warning for any variable of type `value` whose address is taken (or any variable containing a field of type `value`), as well as for any global variable of type `value`.

Recall that in our formal system, we do not allow pointer arithmetic on C pointer types. In our implementation, we allow potentially unsafe pointer arithmetic for non-`value` types as our focus is on FFI safety, not general C type safety, which could be addressed with other techniques [Chandra and Reps 1999; Necula et al. 2002]. We also treat function pointers conservatively. When encountering a call through a C function pointer, O-Saffire currently issues a warning and does not generate typing constraints on the parameters or return type.

Sometimes it is useful to pass C data and pointers to OCaml. For example, glue code for a windowing library might want to return pointers representing windows or buttons to OCaml. There are two main ways to accomplish this. C pointers can be safely passed directly to OCaml, because the OCaml garbage collector does not follow pointers to memory outside its own heap. Alternately, pointers can be passed in *custom blocks* that encapsulate C data and include pointers to routines for deallocation, comparison, serialization, and hashing, which will be used appropriately by the OCaml runtime. In either case, it is up to the programmer to ensure that such data is not treated as OCaml data or vice-versa.

O-Saffire supports these behaviors with a very simple heuristic. If a `value` is cast directly to a C pointer type, our analysis unifies the type of the OCaml value with a *custom* type and therefore ensures that the OCaml value is always used as a C pointer. However, we do not distinguish between two different C types so long as they are both pointers. Since a programmer can easily cast one pointer type to another arbitrarily in C, enforcing a stronger heuristic at the language boundary did not seem beneficial. On the other hand, O-Saffire issues a warning at a cast of a `value` type directly to a primitive type (such as an integer), since that is most likely an error in our experience. Our heuristic for other casts is to ignore all casts between C types that do not involve `value`.

We model C `struct`s in one of two ways. Any `struct s` stored on the C stack (i.e., a local or parameter) is expanded one level so that all of its fields are treated as local variables (e.g., `s.f1`, `s.f2`, etc). If an OCaml value is stored in or read from a heap-allocated `struct`, we emit a warning, which only occurred twice in our benchmarks. In general, we found that C `struct` types rarely contain OCaml data.

In addition to the types we have described so far, OCaml also includes objects, polymorphic variants, and universally quantified types. O-Saffire treats object types in the same way as opaque types, with no subtyping between different object types. We have not seen objects used in FFI C code. O-Saffire does not handle polymorphic variants, which are used in FFI code, and this leads to some false positives in our experiments. Unlike traditional sum types which use a sequentially numbered tag in the header of a structured block, polymorphic variants store a hash of their constructor name as the first element of the block. This is particularly difficult to

Program	C-loc	O-loc	Ext	Time	Err	Wrn	FPos	Imp
apm-1.00	124	156	4	0.01s	0	0	0	0
camlzip-1.01	139	820	9	0.01s	0	0	0	1
ocaml-mad-0.1.0	139	38	3	0.01s	1	0	0	0
ocaml-ssl-0.1.0	187	151	14	0.02s	4	2	0	0
ocaml-glpk-0.1.1	305	147	30	0.03s	4	1	0	1
gz-0.5.5	572	192	29	0.02s	0	1	0	1
ocaml-vorbis-0.1.1	1183	443	7	0.07s	1	0	0	2
ftplib-0.12	1401	21	17	0.06s	1	2	0	1
lablgl-1.00	1586	1357	324	0.40s	4	5	140	20
cryptokit-1.2	2173	2315	24	0.03s	0	0	0	1
lablgtk-2.2.0	5998	14847	1307	3.83s	9	11	74	48
				Total	24	22	214	75

Fig. 26. Experimental results

handle because we have seen FFI code use binary search to select the proper tag for the block, and our analysis is not sophisticated enough to support this style of tag test.

O-Saffire translates universally quantified type variables to the representational type $(\psi, \pi+\sigma)$ where ψ, π and σ are fresh variables with the constraint $1 \leq \psi$. Since a polymorphic type could be either boxed or unboxed, this prevents a C function from using the polymorphic type directly as an integer or a boxed type without at least performing a boxedness test. Our current implementation also cannot infer universally quantified types for C "helper" functions that are polymorphic in OCaml `value` parameters. Unlike JNI glue code, such OCaml glue functions appear to be rare in practice, as we only saw 4 such functions in our benchmark suite. We could extend our implementation to use the same technique discussed in Section 4.3 to infer polymorphic signatures, but instead we take a heuristic approach: In O-Saffire, the programmer can annotate functions to indicate that calling them should yield no constraints between formal and actual arguments, and we added such annotations for the 4 polymorphic functions we found.

Finally, a common technique for error handling in C glue code is to raise an OCaml exception, and when this occurs, the OCaml runtime pops the entire C function stack and returns control to OCaml. However, in order to throw an exception, the exception itself must be allocated on the OCaml heap. Thus it is safe for a function not to register its local references to data on the OCaml heap and then allocate and throw an exception. To avoid false positives in this situation, our implementation tracks functions which never return, and assigns them the effect nogc, since all C references are no longer live when such a function is called.

5.2 OCaml Experiments

We ran O-Saffire on several programs that utilize the OCaml foreign function interface. The programs we looked at are actually glue libraries that provide an OCaml API for system and third-party libraries. All of the programs we analyzed were from a tested, released version, though we believe O-Saffire is also useful during development.

Fig. 26 gives a summary of our benchmarks and results. For each program, we

list the lines of C code, OCaml code, and number of external OCaml function declarations. The fifth column gives the running time in seconds (five run average) for our C code analysis on an AMD Athlon 4600 processor with 4GB of memory. Recall from Section 3.4 that we do not directly analyze OCaml function bodies. The cost of extracting OCaml types is negligible and therefore omitted. The running time only contains the time spent in our analysis, which begins once CIL has constructed the AST of the program. All of the running times are very fast. The slowest benchmark is lablgtk, which takes significantly more time than the others. We believe this is due to the large number of global variables pulled in to the code via system headers, which slows down operations on type environments.

The next three columns list the number of errors found, the number of warnings for questionable programming practice, and the number of false positives, i.e., warnings for code that appears to be correct. The last column shows the number of places where the implementation warned that it did not have precise flow-sensitive information (see below). The total number of warnings issued by O-Saffire is the sum of these four columns.

We found 24 outright errors in the benchmarks. One source of errors was forgetting to register C references to the OCaml heap before invoking the OCaml runtime. This accounts for one error in each of ftplib, lablgl, and lablgtk. Similarly, the one error in each of ocaml-mad and ocaml-vorbis was registering a local parameter with the garbage collector but then forgetting to release it, thus possibly leaking memory or causing subtle memory corruption.

The 19 remaining errors are type mismatches between the C code and the OCaml code. For instance, 5 of the lablgtk errors and all ocaml-glpk and ocaml-ssl errors were due to using `Val_int` instead of `Int_val` or vice-versa. Another error was due to one FFI function mistreating an optional argument as a regular argument. In particular, the function directly accessed the option block as if it were the contents of the option type rather than the option type itself. Thus, the C code will most likely violate type safety.

Another lablgtk error was due to a C pointer of type `t` being initialized as the value `(t*)Val_unit`, which is unsafe—recall that `Val_Unit` is represented as the OCaml integer 0, which is stored with a tag in memory as the value 1. This mistake is particularly dangerous because on one program path, the variable is passed to a gtk library function that expects either a valid pointer or `NULL`, but `Val_Unit` is neither of those. The remaining two gtk errors were due to a type mismatch between OCaml and the C. The OCaml source code contained a declaration type t = {len: int}, defining the type t to be a record with a single element. However, the C code manipulated data with this type as if it had a second field containing a C pointer, which it does not. The remaining lablgl errors were similar cases in which data was described with one shape in OCaml and manipulated with a different shape in C.

In addition to the 24 errors, O-Saffire reported 22 warnings corresponding to questionable coding practices. A common mistake was declaring the last parameter in an OCaml signature as type unit even though the corresponding C function omits that parameter in its declaration:

OCaml : external f : int → unit → unit = "f"
C : `value f(value x);`

While this does not usually cause problems on most systems, it is not good practice, since the trailing unit parameter is placed on the stack. The warnings reported for ftplib, ocaml-glpk, ocaml-ssl, lablgl, and lablgtk were all due to this case.

The warning in gz is an interesting abuse of the OCaml type system. The gz program contains an FFI function to *seek* (set the file position) on file streams, which either have type `input_channel` or `output_channel`. However, instead of taking a sum type as a parameter to allow both kinds of arguments, the function is declared with a polymorphic type as its parameter.

```
OCaml : external seek : int → 'a → unit = "seek"
    C : value seek(value  pos, value  chan) {
          FILE  *strm = Field(chan,0);
          fseek(strm,...);
        }
```

Clearly using `chan` in this way is very dangerous, because OCaml will allow *any* argument to be passed to this function, including unboxed integers. In this case, however, only the right types are passed to the function, and it is encapsulated so no other code can access it, and therefore we classify this as questionable programming practice rather than an error.

O-Saffire also reported a number of false positives, i.e., warnings for code that seems correct. One source of false positives is polymorphic variants, which we do not handle. The other main source of false positives is due to pointer arithmetic disguised as integer arithmetic. Recall that the type `value` is actually a typedef for `long`. Therefore if v is an OCaml value that has been unified with a C pointer type, then both $((t*)v + 1)$ and $(t*)(v + \text{sizeof}(t))$ are equivalent. However, our system will not type check the second case because direct arithmetic is being performed on a `value` type.

Finally, in several of the benchmarks there are a number of places where O-Saffire issued a warning because it does not have precise enough information to compute a type. For instance, this may occur when computing the type of $e_1 \texttt{+}_p e_2$ if e_2 has the type $\texttt{int}\{\top, 0, \top\}$, since the analysis cannot determine the new offset. 54 of the imprecision messages occurred when the type of an expression included B, I, or T as $\top$ when a more precise type was needed during inference. We also classify as imprecision warnings those that are for occurrences of global `value` types, uses of function pointers, and occurrences of C `struct` types containing `value` types stored in the heap. However, these cases occurred only 10, 8, and 2 times, respectively. One interesting direction for future work would be eliminating these warnings and instead adding run-time checks to the C code.

We have reported the O-Saffire results to the authors of the programs in our benchmark suite and confirmed that all of the errors reported were in fact previously unknown bugs.

5.3 J-Saffire

Similarly to O-Saffire, our implementation of J-Saffire is composed of two tools. The first tool is a lightweight Java compiler wrapper that 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. Note that this is different from O-Saffire, which could eagerly translate OCaml types because only

types occurring in external declarations could be used in C. The JNI, however, allows C code to look up Java classes dynamically via FindClass. Thus the second tool in J-Saffire translates Java types and applies our type inference algorithm to C code, issuing warnings whenever it finds a type error. This part of J-Saffire uses CIL to parse C source code and the OCaml JavaLib [Cannasse 2004] to extract Java type information from compiled class files.

In addition to the type `jobject`, the JNI contains several primitive types, including `void` and `int`, which we did not model in our formal system. To model these additional types, our implementation has slightly richer handling of field and method descriptors. In the JNI, these descriptors may include class names, which we included in our formalism, or strings such as `I` and `V` to represent primitives `int` and `void`. Thus in our implementation, we introduce a new type constructor $\mathsf{JTStr}\{\nu\}$, which is a place holder for the jt type described by ν. When ν is unified with a constant string, the type $\mathsf{JTStr}\{\nu\}$ is immediately replaced with the concrete type it describes. This was not necessary in our formalism, because we could use a representational type $\{\nu;\ \phi;\ \mu\}$ instead, but that type would only represent an object and not a primitive.

Another type that we omitted from our formal system is Java arrays. In Java, arrays are a subtype of Object, and so we model them in our implementation as a Java class with a unique name. We represent the array contents as a field in this object so that its type is available when the array is dereferenced.

The JNI also contains a number of typedefs (aliases) for more specific object types, such as `jstring` for Java Strings. 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. J-Saffire 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 14 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. J-Saffire does not distinguish between the C name and its `j`-prefixed counterpart. Our implementation also forbids casting between `jobject` (or one of its typedefs) to any other C type and produces a warning if this occurs.

Rather than being called directly, JNI functions are actually stored in a table that is passed as an extra argument (typically 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,...)`. J-Saffire extracts FFI function names via syntactic pattern matching, and we assume that the table is the same everywhere. J-Saffire ignores calls to function pointers that are not part of the JNI, and issues a warning whenever it encounters one.

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. J-Saffire 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.

Program	C-loc	J-loc	Ext	Time	Err	Wrn	FPos	Imp
libgconf-java-2.10.1	1119	670	93	1.32s	0	0	10	0
libglade-java-2.10.1	149	1022	6	0.64s	0	0	0	1
libgnome-java-2.10.1	5606	5135	599	6.53s	45	0	0	1
libgtk-java-2.6.2	27095	32395	3201	1.04s	74	8	36	18
libgtkhtml-java-2.6.0	455	729	72	0.65s	27	0	0	0
libgtkmozembed-java-1.7.0	166	498	23	0.66s	0	0	0	0
libvte-java-0.11.11	437	184	36	0.67s	0	26	0	0
jnetfilter	1113	1599	105	5.38s	9	0	0	0
libreadline-java-0.8.0	1459	324	17	0.63s	0	0	0	1
pgpjava	10136	123	12	1.11s	0	1	0	1
posix1.0	978	293	26	0.70s	0	1	0	0
Java Mustang compiler	532k	1974k	2495	630s	1	88	96	2620
				Total	156	124	142	2642

Fig. 27. Experimental results

Although our formal type system is flow-insensitive, J-Saffire treats 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, similarly to O-Saffire.

Lastly, J-Saffire 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.

5.4 JNI Experiments

We ran J-Saffire on a suite of 12 benchmarks that use the JNI. Fig. 27 shows our results. The first 7 programs are taken from the Java-Gnome project [Java-Gnome Developers 2005], and the remaining programs are unrelated. The last program is a development Java 1.6 compiler, code-named Mustang (we used build 61). All benchmarks except pgpjava and Mustang are glue code libraries that connect Java to an external C library. For each program, Fig. 27 lists the number of lines of C code and Java code, and the number of `native` methods. Next we list the analysis time in seconds (average of 5 runs), and the number of messages reported by J-Saffire, manually divided into the same four categories as the O-Saffire experiments. The running time includes the C code analysis (including extracting Java types from class files) but not the parsing of C code or the compilation time. The measurements were performed on an AMD Athlon 4600 processor with 4GB of RAM. In a prior conference version [Furr and Foster 2006b], we reported the number of messages for Mustang to be much lower. We since discovered that we had not reported the imprecision messages for this benchmark and have also fixed a few bugs in the implementation which have changed some of our results.

J-Saffire reported 156 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 a private 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. J-Saffire 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 18 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. The one error in Mustang occurred when the C source code did not properly distinguish between the types `int` and `long` in a C function signature. If used on a big-endian 64-bit machine, the C function would access only the higher 32 bits of the value on the stack, creating a runtime error [Furr and Foster 2005b].

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. As mentioned in Section 2.2, such functions must be given long names following a particular naming scheme, and it is easy to get this wrong.

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.

J-Saffire also produced 124 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 [Liang 1999], but the Sun JVM we tested allows both versions, so we only consider this a warning.

13 of the warnings were due to using an inappropriate type alias for a JNI type, such as declaring an arbitrary object with the type `jstring`. Since these aliases

are just typedefs for `jobject`, their misuse will not cause any ill effects, but this is a poor practice.

Finally, 110 warnings were due to the declaration of C functions that appear to implement a specific native method (because they have specially formatted 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.

J-Saffire also produced 140 false positives, which are warnings about correct code. 27 of the Mustang false positives, 34 of the libgtk false positives, and all of the false positives from the rest of the benchmarks were due to subtyping inside of C code, which our analysis does not model precisely because it uses unification. The remaining 2 false positives in libgtk and 31 of the Mustang messages occurred when an object of type jt JClass unified with an instance of java.lang.Class. This is safe in the JNI, but since we use a richer representation of classes in J-Saffire, these two types are incompatible in our system. Of the remaining false positives (all in Mustang), 32 occurred because of the type of Java object returned by a function was selected using a `switch` statement that tested the contents of a string. These occurred in internal functions of Mustang that directly manipulate Java objects, so it is no surprise that our analysis was unable to handle this style of code.

Finally, J-Saffire emitted 2642 imprecision warnings, which occurred when the analysis was unable to analyze a particular program point. The vast majority of these cases were in Mustang. 36 of the imprecision warnings (of which 14 are from Mustang) were due to unification failures with partially specified methods that could not be resolved. 707 of the messages (of which 701 are from Mustang) were due to code using parts of the JNI API that our analysis does not model, such as using an alternative calling convention of passing method arguments packed in an array. The remaining messages all came from Mustang. 115 of these were due to expressions that our analysis was unable to handle, such as directly manipulating the `jobject struct` type, which is assumed to be opaque in our system. The other 1784 messages were due to the use of function pointers, which J-Saffire does not model. The Java compiler appears to be atypical in the style of JNI code it uses, since the other benchmarks reported few, if any such warnings (which is not surprising). Therefore, we did not investigate adding support for these features in our initial implementation, but this may be worth exploring in a future version.

We have reported the results to the authors of the programs in our benchmark suite for J-Saffire and confirmed that all of the errors reported were in fact previously unknown bugs.

6. DISCUSSION AND FUTURE WORK

We have presented two type inference systems for checking type safety of programs that use the OCaml FFI and the JNI. Our inference systems rely on two key ideas. The first is to refine the types given to high-level data in C glue code. Instead of the single, conflated type assigned by the FFI, our analyses use multilingual, representational types that model C's view of OCaml and Java data. The other key idea of our analyses is to use singleton types to track values through C glue

code in order to determine how high-level data is being accessed by the glue code.

The need for value tracking is a fundamental part of FFI analysis, because it allows us to model FFI operations precisely. In the high-level language, operations like pattern matching and field access are distinguished syntactically using constructs that contain non-first class identifiers. For example, in Java a field access is written $e.f$, where e is an arbitrary expression but f is an identifier for a field. C glue code cannot simply use C's corresponding field access operator for Java data because the C compiler does not have object layout information. Thus instead in the JNI, object fields are accessed via a series of function calls in which the field name is passed as a string. This makes field names first class in C glue code, which means that in order to determine what fields are accessed by a JNI operation we need to track string values. We similarly needed to track integer values in glue code for the OCaml FFI in order to track manipulations of sum and product types.

We found that we were able to determine string and integer values with sufficient precision using fairly simple analyses—a flow-insensitive, context-sensitive analysis for strings in the JNI, and a flow-sensitive, context-insensitive analysis for integers and offsets in the OCaml FFI. We suspect these analyses were sufficient because glue code seems complicated on the surface, and since programmers are naturally worried about making mistakes in this code, they use the FFI in a simple way.

Despite this, it is not surprising that programmers still make type errors in glue code. In both FFIs, type violations can succeed silently—this is almost always the case in the OCaml FFI, while in the JNI some kinds of type errors result in run-time exceptions. Thus programmers are left to rely on testing to find bugs, which can cover only a limited number of executions. Another approach programmers could take is to wrap high-level data in C types and use C's type system to enforce type safety. Although this would be possible (e.g., making one C `struct` for each kind of high-level data), we did not find any code that tried this, perhaps because it may be too complicated given the perceived benefit, and it would require updating each time a type signature changed.

We also found that, although in many ways the OCaml FFI and the JNI are similar, their different design decisions resulted in different kinds of mistakes by programmers. In particular, in the OCaml FFI programmers need to remember to explicitly register pointers to the OCaml heap, while the JNI automatically registers such pointers. As a result OCaml programs can introduce bugs by forgetting to register pointers, although they can also reduce overhead by not bothering to register pointers that will be dead before the garbage collector is invoked.

Aside from these fundamental design choices, even seemingly trivial FFI design decisions have consequences. For example, the poor choice of names for the OCaml FFI macros `Val_int` and `Int_val` (do you remember which is which?) leads to mistakes, whereas in the JNI there is no such confusion because C and Java integers have the same representation. On the other hand, the JNI's baroque naming convention for native functions results in many errors, whereas we found no cases where the OCaml FFI programmer mismatched function names between C and OCaml. In general, writing programs that use an FFI is just like any other software engineering task: if the programmer has an opportunity to make a mistake because of something slightly complicated or confusing, he or she will make that

mistake at times.

6.1 Future Improvements to O-Saffire and J-Saffire

There are several aspects of the OCaml and Java FFIs that our systems do not currently handle. Both FFIs include array types that can be accessed by C code, but our analyses do not perform bounds checks. One approach to this problem is inserting dynamic checks at runtime [Tan et al. 2006]. In our experiments, we found a number of false positives in O-Saffire due to polymorphic variants, and in J-Saffire due to the lack of subtyping on `jobject`s. We believe that both of these could be addressed with more sophisticated analyses.

Our analyses also do not model control flow via exceptions. In the OCaml FFI, an exception unwinds both the OCaml and the C stack to the point of the closest handler. Thus the C programmer must be careful to free any C resources in use when an exception is raised. In the JNI, exceptions unwind only the Java stack up to the last native method or the last handler, whichever comes first. Thus JNI glue code must be careful to always check for exceptions after calling almost any JNI function. We leave enforcement of these requirements to future work.

Finally, our analyses also do not track foreign data stored in global C variables. In both OCaml and Java, if a foreign data reference is stored in a global, the glue code must register that reference as a global root to the garbage collector. Similarly, when the global variable is overwritten or becomes dead, the programmer must explicitly remove the global root. Currently O-Saffire and J-Saffire simply issue warnings when they see global variables that may hold foreign data and do not attempt to track the contents of the global. To verify that global references are handled correctly would likely require a new global analysis, including constructing a multilingual call graph to track the lifetimes of objects through the entire program.

6.2 Future Work for Other FFIs

We believe that our basic approach can also be applied to other FFIs. One interesting target is the Haskell FFI [Jones 2001], which is somewhat different than the OCaml or Java FFI. In the Haskell FFI, all glue code is written in Haskell rather than C. The programmer imports foreign (typically C) functions into the Haskell namespace, giving them Haskell type signatures. Assuming these are correctly specified, Haskell's type system will then enforce type safety at native calls. However, just as manipulating high level types in C code is cumbersome in the OCaml FFI and the JNI, manipulating C types in Haskell is also messy and error-prone. For instance, to access a field of a C `struct`, the programmer must use the low-level, type-unsafe function `peekByteOff`, which extracts a field at a given numeric offset. While an FFI pre-processor can help automate many of these low-level calls, we believe our approach could also ensure that Haskell accesses C data safely.

Another possible target is dynamically typed languages such as Python, which has an FFI to C. To analyze such a program we would most likely need to use a soft typing system [Wright and Cartwright 1994] to infer types for Python data. One challenge for Python is that it uses reference counting for memory management, and thus C glue code must explicitly increment and decrement counts for references to Python data. As this is easy to do incorrectly, we suspect an analysis to look for these kind of bugs would find many mistakes.

Finally, another kind of code that has the same flavor as FFIs is code that uses reflection. Reflection APIs typically treat data as "foreign" until its type has been examined, at which point it can be cast to a native type. For example, Java's reflection API looks very similar to the JNI, except that objects accessed by reflection are instances of `Object` rather than the completely opaque type `jobject`. In Java's reflection API, class, field and method names are looked up using strings, intermediate field values are stored in a non-parameterized `Field` class, and result types must be explicitly downcast from `Object` to the correct type. We believe that our singleton type system could be used to track strings through Java reflective code, and we could then statically ensure that many uses of reflection are type safe. We leave this as an open problem for future work.

7. RELATED WORK

Most languages include a foreign function interface, typically to C, since it runs on many platforms. For languages whose semantics and runtime systems are close to C, foreign function interfaces can be fairly straightforward. For languages that are further from C, FFIs are more complicated, and there are many interesting design points with different tradeoffs [Blume 2001; Finne et al. 1999; Huelsbergen 1996; Leroy 2004; Liang 1999; Jones 2001]. For example, Blume [Blume 2001] proposes a system allowing arbitrary C data types to be accessed by ML. Fisher et al [Fisher et al. 2001] have developed a framework that supports exploration of many different foreign interface policies that free the programmer from having to write glue code to translate between data representations. While various interfaces allow more or less code to be written natively (and there is a general trend towards more native code rather than glue code), the problem of validating usage of the interface on the foreign language side still remains.

In work concurrent to ours, Tan et al [Tan et al. 2006] present a system that inserts dynamic checks into C glue code to ensure that the JNI is used safely. Because the checks are performed at runtime, their system can only verify type and memory safety of particular program executions. However, they are also able to check for more kinds of errors than our static system, such as out of bounds array accesses.

Our technique of tracking pointer offsets for the OCaml-to-C FFI bears some resemblance to systems that use physical type checking for C [Chandra and Reps 1999; Necula et al. 2002], in that both must be concerned with detailed memory representations. However, our system is considerably simpler than full-fledged physical type checking systems, because OCaml data given type `value` is typically only used in restricted ways.

Our string-tracking mechanism for code that uses the JNI is relatively simple, supporting polymorphism but treating strings themselves as opaque. Recently several researchers have developed more sophisticated techniques to track strings in programs [Christensen et al. 2003; DeLine and Fähndrich 2004; Gould et al. 2004; Thiemann 2005]. These systems include support for modeling string manipulation routines, because one of their goals is to check that dynamically-generated SQL queries are well-formed. For purposes of checking clients of the JNI, we have found that our simple tracking of strings as constants is sufficient.

Nishimura [Nishimura 1998] presents an object calculus that can statically infer kinded types for first-class method names, which is similar to our inference of Java object types for the JNI. Nishimura's system has similar restrictions to ours such as 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, and we have implemented our system and applied it to a number of benchmarks.

Trifonov and Shao [Trifonov and Shao 1999] use effects to reason about the safety of interfacing multiple safe languages with different runtime resource requirements in the same address space. Their focus is on ensuring that code fragments in the various languages have access to necessary resources while preserving the languages' semantics, which differs from our goal of checking types and GC properties in FFIs.

Matthews et al.[Matthews and Findler 2007] study the semantics of mixing the typed and untyped lambda calculus to better reason about their interaction. They focus on high level properties such as ensuring soundness of the typed language in the presence of foreign values, while our focus is on detecting errors in low-level glue code.

There are a number of alternatives to using FFIs directly. One technique is to use automatic interface generators to produce glue code. SWIG [Beazley 1996] generates glue code based on an interface specification file. This has the advantage of eliminating the need for custom glue code (and thus eliminating safety violations), but it exposes all of the low level types to the high level language, creating a possibly awkward interface. Exu [Bubba et al. 2001] provides programmers with a light-weight system for automatically generating JNI-to-C++ glue code for the common cases. Mockingbird [Auerbach et al. 1999] 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 COM [Gray et al. 1998], SOM [Hamilton 1996] and CORBA [Object Management Group 2004]. Barrett [Barrett 1998] 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. Grechanik et al [Grechanik et al. 2004] 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.

Another approach to avoiding foreign function interfaces is to compile all programs to a common intermediate representations. For example, the Microsoft common-language runtime (CLR) [Hamilton 2003; Meijer et al. 2001] provides interoperation by being the target of compilers for different languages, and the CLR includes a strong static type system. While this solution avoids some of the difficulties that can arise with FFIs, it does not solve the issue of interfacing with programs in non-CLR languages or with unmanaged (unsafe) CLR code.

8. CONCLUSION

We have presented O-Saffire and J-Saffire, a pair of multilingual type inference systems for checking the type safety of programs that use the OCaml-to-C FFI and the JNI, respectively. Both systems focus their analysis efforts on C glue code, ensuring that it obeys the high-level language types. Since these FFIs conflate almost all high-level types to a single C type, to perform inference we use extended multilingual types that embed the high-level types into C types. O-Saffire and J-Saffire use representational types to capture C's view of high-level data. In O-Saffire, our representational types model a union of unboxed and boxed data, which covers integers, updatable references, data types, and the unit type. Then during type inference, we use an intraprocedural data flow analysis algorithm to track the currently-known information about a representational type, which allows us to analyze the C code equivalent of OCaml pattern matching. In J-Saffire, representational types model an object as an instance of a class with a given name, which may be a string variable, and with a subset of its fields and methods, the names and types of which could also be unknown string variables. Since strings tend to be used fairly simply in JNI glue code, J-Saffire uses unification to track strings, but includes a polymorphic inference component to model wrapper functions precisely, even allowing functions to be polymorphic in the values of string arguments. O-Saffire also uses effects to track garbage collection information and ensure that C pointers to the OCaml heap are registered with the garbage collector. We applied our implementations of O-Saffire and J-Saffire to a number of benchmarks, finding many errors and suspicious coding practices. Our results suggest that our static checking system can be an important part of ensuring that foreign function interfaces are used correctly.

Acknowledgments

This research was supported in part by NSF CCF-0346982 and CCF-0430118. We would like to thank the anonymous reviewers for their helpful comments.

REFERENCES

Auerbach, J., Barton, C., Chu-Carroll, M., and Raghavachari, M. 1999. Mockingbird: Flexible stub compilation from paris of declarations. In *Proceedings of the 19th International Conference on Distributed Computing Systems*. Austin, TX, USA.

Barrett, D. J. 1998. Polylingual Systems: An Approach to Seamless Interoperability. Ph.D. thesis, University of Massachusetts Amherst.

Beazley, D. M. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *USENIX Fourth Annual Tcl/Tk Workshop*.

Blume, M. 2001. 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.

Bubba, J. F., Kaplan, A., and Wileden, J. C. 2001. The Exu Approach to Safe, Transparent and Lightweight Interoperability. In *25th International Computer Software and Applications Conference (COMPSAC 2001)*. Chicago, IL, USA.

Cannasse, N. 2004. Ocaml javalib. `http://team.motion-twin.com/ncannasse/javaLib/`.

Chandra, S. and Reps, T. W. 1999. Physical Type Checking for C. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toulouse, France, 66–75.

Christensen, A. S., Møller, A., and Schwartzbach, M. I. 2003. Precise Analysis of String Expressions. In *Static Analysis, 10th International Symposium*. San Diego, CA, USA.

DeLine, R. and Fähndrich, M. 2004. The Fugue Protocol Checker: Is your software Baroque? Tech. Rep. MSR-TR-2004-07, Microsoft Research. Jan.

Fähndrich, M., Rehof, J., and Das, M. 2000. 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.

Felleisen, M. and Hieb, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103,* 2, 235–271.

Finne, S., Leijen, D., Meijer, E., and Jones, S. P. 1999. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*. Paris, France, 114–125.

Fisher, K., Pucella, R., and Reppy, J. 2001. A framework for interoperability. In *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*. Firenze, Italy.

Furr, M. and Foster, J. S. 2005a. Checking Type Safety of Foreign Function Calls. In *Proceedings of the 2005 ACM Conference on Programming Language Design and Implementation*. Chicago, Illinois, 62–72.

Furr, M. and Foster, J. S. 2005b. Java SE 6 "Mustang" Bug 6362203. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6362203`.

Furr, M. and Foster, J. S. 2006a. Checking Type Safety of Foreign Function Calls. Tech. Rep. CS-TR-4845, University of Maryland, Computer Science Department. Dec.

Furr, M. and Foster, J. S. 2006b. Polymorphic Type Inference for the JNI. In *15th European Symposium on Programming*. Vienna, Austria. To appear.

Gould, C., Su, Z., and Devanbu, P. 2004. Static Checking of Dynamically Generated Queries in Database Applications. In *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, Scotland, UK, 645–654.

Gray, D. N., Hotchkiss, J., LaForge, S., Shalit, A., and Weinberg, T. 1998. Modern Languages and Microsoft's Component Object Model. *cacm 41,* 5 (May), 55–65.

Grechanik, M., Batory, D., and Perry, D. E. 2004. Design of large-scale polylingual systems. In *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, Scotland, UK, 357–366.

Hamilton, J. 1996. Interlanguage Object Sharing with SOM. In *Proceedings of the Usenix 1996 Annual Technical Conference*. San Diego, California.

Hamilton, J. 2003. Language Integration in the Common Language Runtime. *ACM SIGPLAN Notices 38,* 2 (Feb.), 19–28.

Henglein, F. 1993. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems 15,* 2 (Apr.), 253–289.

Huelsbergen, L. 1996. A Portable C Interface for Standard ML of New Jersey. `http://www.smlnj.org//doc/SMLNJ-C/smlnj-c.ps`.

Java-Gnome Developers. 2005. Java bindings for the gnome and gtk libraries. `http://java-gnome.sourceforge.net`.

Jones, S. P. 2001. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineernig theories of software construction*, T. Hoare, M. Broy, and R. Steinbruggen, Eds. IOS Press, 47–96.

Leroy, X. 2004. The Objective Caml system. Release 3.08, `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`.

Liang, S. 1999. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley.

Lindholm, T. and Yellin, F. 1997. *The Java Virtual Machine Specification*. Addison-Wesley.

Matthews, J. and Findler, R. B. 2007. Operational Semantics for Multi-Language Programs. In *Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*. Nice, France, 3–10.

Meijer, E., Perry, N., and van Yzendoorn, A. 2001. Scripting .NET using Mondrian. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*. Budapest, Hungary.

Necula, G., McPeak, S., Rahul, S. P., and Weimer, W. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction, 11th International Conference*. Grenoble, France.

Necula, G., McPeak, S., and Weimer, W. 2002. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*. Portland, Oregon, 128–139.

Nishimura, S. 1998. Static Typing for Dynamic Messages. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*. San Diego, California.

Object Management Group 2004. *Common Object Request Broker Architecture: Core Specification*, Version 3.0.3 ed. Object Management Group.

Rémy, D. 1989. Typechecking records and variants in a natural extension of ML. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*. Austin, Texas, 77–88.

Tan, G., Appel, A. W., Chakradhar, S., Raghunathan, A., Ravi, S., and Wang, D. 2006. Safe Java Native Interface. In *Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*. Arlington, Virginia, USA.

Thiemann, P. 2005. 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.

Trifonov, V. and Shao, Z. 1999. Safe and Principled Language Interoperation. In *8th European Symposium on Programming*, D. Swierstra, Ed. Lecture Notes in Computer Science, vol. 1576. Springer-Verlag, Amsterdam, The Netherlands, 128–146.

Wright, A. K. and Cartwright, R. 1994. A practical soft type system for scheme. In *Proceedings of the Conference on Lisp and Functional Programming*. 250–262.

Received Month Year