Comprehensive exam position paper: Multi-language programming and interoperability

Geoffrey C. Hulette

August 4, 2010

Contents

1 Introduction

Increasingly, programs are being written in more than one programming language. For example, modern web-based applications may involve Java or Ruby running on the server, use SQL to access a database, render pages in PHP, and deliver content encoded in both HTML and Javascript. Usually, languages in these programs interact in an *ad hoc* manner. For example, it is common for SQL programs to be encoded in the character strings of a language like Java, and then invoked through an API function call. These methods are expedient but crude; in particular they may obstruct or preclude automated static analyses.

At the same time, there is a growing need to give programmers better access so-called "legacy" codes, many of which are written in disused languages. Automated tools are required to expose these codes *naturally*, i.e. with the high-level types, semantics, and idioms of modern languages. Without these tools, programmers are left writing tedious "glue code" to provide the connective logic. Writing this sort of code often requires a deep understanding of the various languages involved, and so is either highly specialized or else error-prone work.

Both of these problems can be addressed by *multi-language programming* techniques. The overarching goal of multi-language programming is to allow programmers the convenience of choosing whichever language is best suited to a particular task, without having to worry about how that language will interact (or not interact) with others in the application. Ideally, multi-language programming should be *seamless* [14], i.e. a programmer working in one language never needs to be aware that their code may interact with a different

language.

In this report, we will see several different approaches to multi-language programming with examples. We will examine the advantages and trade-offs of each approach, and conclude with some ideas for future work.

1.1 Goals

While the various approaches to multi-language programming employ different methods, but share common goals.

1.1.1 Scalability

In the context of multi-language programming, *scalability* refers to the ability of a system to accommodate a growing number of participating languages. For n languages, where each language must potentially interoperate with every other language, there are about n^2 interfaces that must be considered [41]. Assuming a non-trivial amount of work required for each interface, systems that fail to improve on this quadratic scaling factor can expect to incorporate no more than a handful of languages. As we will see in Section 4, interface definition languages are a popular approach that reduces the scaling factor to $2n$, although at some cost to other goals.

1.1.2 Natural programming model

Ideally multi-language programming should be completely transparent from the programmer's point of view. In practice, this is rarely achieved. Nevertheless, it is desirable to minimize the amount of required "glue code". In

particular, programmers should be free to work with the native types, semantics, and idioms of their preferred language as much as possible. These constructs can then be mapped, preferably in an automated way, to the corresponding constructs in other languages. Function calls, for example, seem to be a popular point of abstraction to bridge language barriers; the approaches in Sections 3 and 4 both use the syntax and semantics of function calls to hide inter-language machinery from the programmer.

1.1.3 Efficiency

Multi-language programming loses much of its attraction if it forces programs to pay a high price in performance. Multi-language programming tools should try to minimize the costs of, for example, marshalling and unmarshalling data structure representations (see Section 4). However, efficiency concerns may often be at odds with the goal of providing natural programming constructs.

1.1.4 Types and safety

Multi-language programming techniques are often used to connect a high-level, type-safe language to a low-level, unsafe language. In particular, many high-level languages feature connections to C (see Section 3). Ideally, systems that provide these sorts of connections are able to isolate the unsafe portions of the code from the safe ones. Otherwise, multi-language programming may remove a key benefit of using a type-safe language in the first place.

2 Operating system facilities

A common model for multi-language programming is to code an application as a set of programs that run as separate processes. These processes communicate via the operating system, and each one could potentially be written in a different language. In this model, no special inter-language facilities are required. Instead, the operating system's data exchange facilities, such as interprocess communication, the file system, sockets, or pipes are used to bridge language barriers [56]. For example, one program written in Java could write a sequence of files to the file system and another program written in C could read them. These programs might even run concurrently as part of a single logical application. This model of multi-language programming is simple but powerful, because modern operating system facilities are generally reliable and scalable.

This approach requires that each language provide routines to expose the operating system's functions. Most general-purpose programming languages provide access to the file system, and many allow programmers to access other operating system functions as well. Operating system routines are often written by creating simple wrappers over functions in the C standard library. The wrappers are coded in terms of a foreign function interface (see Section 3).

As a multi-language programming model, the operating system is not without drawbacks. First, use of operating system functions may tie the application to that operating system. Alternatively, the application may require complex logic to adjust itself for each operating system. For example, even a seemingly straightforward mechanism like file path names may have different

interpretations under different operating systems, and failure to account for these differences may cause the program to work in one but fail on another. Second, operating systems do not generally provide for function calls across processes.[1] This limits code reuse, because code cannot be shared across languages without a foreign function interface (see Section 3). Third, operating system routines usually operate on bytes, machine words, and the like – not on high-level types. This effectively makes the programmer responsible for translating data structures from a high-level representation into a sequence of bits, and back. This process is called *marshalling*. Marshalling routines can be tedious to write and error prone. When marshalling must be recoded several times across multiple languages, the chance of introducing errors is multiplied.

Some of these issues may be mitigated by using standard file formats for data exchange. Popular file formats, such as PNG for image data [4] or HDF5 for matrix and scientific data [25], are specified independently of a particular language. For example, HDF5 specifies in great detail how to encode the native types of C++, Fortran, and other languages into a language-neutral bit representation. HDF5 also provides its own types in those languages to represent matrices. Often, robust implementations of readers and writers for popular formats exist in many different languages. In the ideal case, use of a popular file format reduces the application programmer's responsibility to selecting an existing reader/writer implementation. If an implementation does not exist, however, implementing a fully compliant reader/writer may

[1] Many operating systems do provide for remote procedure calls; since RPC is usually designed to work independently of the operating system, we consider it separately in Section 4.

be difficult if the format is very complex. Furthermore, standard file formats do not exist for every conceivable data structure. Even when they do exist, standard file formats may be overly general and heavyweight for the specific data structure at hand.

2.1 Discussion

Using the operating system for multi-language programming is scalable, because each language needs only to provide one set of readers and writers to interoperate. So, for n languages, counting readers and writers separately, the scaling factor is $2n$. This is much better than n^2. However, as discussed above, the work involved in implementing each reader and writer may be substantial.

In a certain sense, operating system routines do provide a natural programming model, insofar as programmers are likely to be familiar with common operating system routines and idioms. There is no support for translating high-level types, however, or for reconciling the semantics of different languages. Those things must be done entirely by hand, if at all.

3 Foreign function interfaces

A foreign function interface (FFI) allows a program written in one language to invoke routines and/or access data in another language. The term FFI is somewhat misleading; in addition to function calls, FFIs may allow method invocations for object-oriented languages, or hooks for a low-level language to work with high-level data. FFIs interface between exactly two languages.

This restriction differentiates them from approaches that allow more than two languages, such as language-neutral intermediate representations (Section 4) or interface definition languages (Section 5). FFIs are directional, and we say that the *source* language initiates calls to the *target* language. Many FFIs, however, also support callbacks from the target language to the source language.

FFIs are a common feature in mainstream programming languages because they provide two important capabilities. First, FFIs allow high-level languages to use low-level functions and capabilities not definable in the high-level language itself. For example, a high-level language may not be able to call the operating system directly, or may require low-level access to hardware features for performance-critical routines. Second, FFIs are used to wrap and expose existing libraries of routines so that they need not be rewritten in the high-level language.

FFIs must reconcile the runtime environments and application binary interfaces of two different languages. This may present a variety of challenges, depending on the features of those languages. We will now consider some examples of FFI systems and the issues they face.

3.1 Examples

3.1.1 Fortran Bind(C)

Fortran 2003 introduced an FFI that allows bidirectional interoperability between Fortran and C [54, 53]. Since Fortran and C are similar, the FFI is straightforward. BindC introduces a set of *interoperable types* in Fortran

that represent C's primitive types (e.g. `int`, `float`, and so on). Values of these types may be passed back and forth between C and Fortran with no conversion needed. Derived types, such as structures, arrays, and pointers are interoperable if their component types are interoperable. To expose a C function to Fortran, the C function is defined as usual and linked into the Fortran program. The Fortran program must declare a correspondingly named function marked with the `BIND(C)` keyword. The function's signature (i.e. its argument and return types) must consist only of interoperable types, and it is the programmer's responsibility to ensure that the declared signature in Fortran matches the C definition.

`BIND(C)` is a bi-directional FFI, so this process also works in reverse. If a Fortran function is defined with an interoperable signature, then the C program can access it by linking in the Fortran code and declaring a C function with the corresponding name and signature.

3.1.2 Java Native Interface

The Java Native Interface (JNI) [43] is an FFI from Java to C. The JNI uses glue code in C functions to provide access to Java object data, and to invoke methods. When it is invoked, the C function is passed a special pointer (called `JNIEnv`) that acts as a reference to the JVM. This pointer is used by special JNI functions that allow C code to look up classes and types by name, instantiate objects, invoke methods, and so on. Interaction occurs dynamically, and no static type checking is performed. One benefit of this dynamic approach is binary compatibility. Because the JNI uses Java's type reflection mechanism [28] to work with JVM entities by name, the C code

will not need to be recompiled even if the JVM implementation changes.

The JNI uses glue code extensively. Primitive Java types are passed by value and mapped to corresponding C primitive types, while objects are passed by reference (except arrays, see below). The references appear to C as opaque pointers and must be manipulated exclusively through glue code. There are two kinds of references: local and global. Local references are valid only until the C function returns to Java. When control has returned to the JVM, locally referenced objects may be garbage collected. Global references are never be garbage collected until they are explicitly released. This implies that the C code must take care to release global references or else incur memory leaks. A global reference is created from a local one. Object-type arguments to JNI functions are always local.

Because objects are accessed indirectly through references and glue code, the garbage collector is free to move objects around in memory at any time. The data may even be moved during the C call if the garbage collector is running on a separate thread – the JNI standard requires that implementations take this possibility into account.

Java arrays are treated specially for performance reasons. Native code is able to access arrays directly, circumventing the usual glue code. This makes array manipulation fast. But, it means the garbage collector must take care not to move the data while the native code is accessing it. There are three ways to accomplish this task. First, the programmer can ask the JNI to "pin" the array; this tells the garbage collector to leave it in place until the programmer unpins it. This method is effective and easy to program, but can significantly complicate the garbage collection algorithm. Therefore not all

JVM implementations support pinning. Second, the JNI can copy the array's contents to a buffer so that the C code can work on it locally, and then copy it back. While the copied array is being modified, the garbage collector is free to relocate the original array. This method works when pinning is not available, but large or frequent array copies may be costly. Third, the native code can enter a critical region that temporarily suspends the Java garbage collection thread. While the garbage collector is suspended, the native code can work on the array undisturbed. However, within the critical region the native code is restricted from blocking.

Extra care must be taken when using arrays in multi-threaded JNI programs. For example, consider what would happen if two Java threads tried to invoke a native method on the same array concurrently, using the method of copying the array to a local buffer. The program would contain a race condition – the first method would complete and write the array back to Java, and then the second method would complete, and overwrite the first array. Or, if instead the program uses the critical region method, it must take care that other threads will not exhaust Java's available memory while the garbage collector is suspended. The JNI enables native code to acquire and release locks via a glue code interface to the usual Java synchronization mechanism, but otherwise does not provide special support for multi-threaded code.

The JNI supports callbacks from C to Java. The C program can use glue code to acquire object references, and pass these references to a Java method. The method signatures are obtained and invoked dynamically, via the `JNIEnv` data structure.

The JNI allows C code to throw Java exceptions. When an exception is

thrown from C, the native function exits immediately, control returns to Java, and the exception is processed normally. Because the JNI allows callbacks, it is possible that an exception thrown from a Java method will be encounter a C function on its way up the stack. In this case, control returns to C. The exception does not interrupt the native code, but is recorded in the `JNIEnv` data structure. Therefore, it is the responsibility of the C function to check whether there are any pending exceptions after control returns from a callback. If an exception is detected, the C function can handle it or else re-throw it.

3.1.3 Python Ctypes

Ctypes [40, 5] is one of several Python FFIs to C; it is interesting because it uses dynamic instead of static libraries. Dynamic libraries are files containing compiled C functions, formatted in such a way that the code can be loaded and executed by other programs at runtime. Dynamic libraries include a symbol table so that functions can be found by name.

The Ctypes Python module is initialized with a dynamic library, and it generates Python wrapper code to expose each function in the library as a Python method. Dynamic libraries do not contain type information, so the programmer must explicitly set the number and types of the arguments as well as the return type for each function.

Ctypes knows how to convert simple data representations between C and Python. Primitive C types are converted to equivalent Python types. Structures in C are converted recursively into Python objects. Arrays may be converted if they contain only primitive types. Structures and arrays are

passed to Python as references to the C heap or stack. The references are just pointers, but Python is prohibited from manipulating them directly. Instead, Ctypes provides Python glue code to manipulate each kind of data structure through the pointer.

Ctypes can construct a C function pointer from a Python routine, allow C to call back into Python. The documentation suggests that Python objects passed to C should have references held in Python, to prevent them from being garbage collected [5].

3.1.4 Matlab MEX files

The Matlab programming environment provides MEX files [3], a simple but effective FFI that allows Matlab to call C, C++, or Fortran functions. MEX foreign functions must be specially written, using the MEX API, to decode and manipulate Matlab's matrix representation. Then, the functions must be declared within Matlab. It is the programmer's responsibility to ensure that the type signatures of Matlab declarations and the C definitions match.

3.1.5 SML/NJ

Standard ML of New Jersey (SML/NJ) provides an FFI to C [35]. It is an interesting example because the two languages are quite different. The representations even of basic data types are different in SML. In particular, data types in C depend on the compiler, where in SML they are fixed by the standard. The FFI avoids this problem by parameterization, using meta-information about the C compiler. The meta-information includes details such as the size of an integer, byte ordering, calling convention, and so on.

The parameterized FFI then exposes a set of types to ML programs representing those used by the C compiler, along with conversion routines to and from common SML types.

SML/NJ manages memory with a garbage collector. As we saw in the JNI, this can cause problematic interactions. For example, a C program using a pointer into the ML heap must ensure that the dereferenced data will not be moved or disposed of. SML/NJ's implementation supports "pinning" memory, i.e. explicitly instructing the garbage collector to temporarily leave the data in place. So, C functions that wish to work with ML data directly must pin the data first.

Functions exported to SML from a C file must be marked with a special macro, which enables SML/NJ to look up the function's address by name. Since compiled C function libraries do not include type information, C functions must be registered in SML at runtime, and assigned the appropriate argument and return types. The FFI provides an ML type representing a pointer onto the C heap. It also provides C glue code to manipulate ML data structures via a pointer onto the ML heap. These pointer types allow complex data to pass across the language boundary.

SML/NJ's FFI provides callbacks to ML functions. A callback's arguments and return types are restricted to the C-compatible data types, including pointers to ML data. ML callbacks are created from regular ML functions, i.e. closures. The conversion is accomplished by dynamically registering an ML closure with the FFI – this creates a "bundle" at a fixed address, which contains code to invoke the closure. The bundle's address is presented to C as a function pointer. With this scheme, the closure may be

relocated by the garbage collector – when this happens, the bundle's contents are updated to reflect the closure's new address. The bundle itself remains at its original address, and so the function pointer remains valid in C.

3.1.6 Haskell 98

The Haskell 98 FFI [20] is part of the Haskell 98 standard. The FFI includes some placeholders intended to facilitate calls from Haskell to any external language, but only the binding to C is defined in detail. Haskell presents some interesting challenges to integration with C. Like ML, Haskell features first-class functions, a strong and static type system, and automatic memory management. In addition, it has call-by-need semantics, and distinguishes functions that may cause side effects from so-called "pure" functions.

A foreign function is exposed by declaring its type signature in the Haskell source code, along with the keywords `foreign import`. This tells the Haskell compiler that the function definition will be found in a C library.

The Haskell 98 FFI restricts the type signature of foreign functions to a set of *basic types* that map unambiguously between Haskell and C. Basic types include the usual primitive types, such as integers and floating point numbers. Basic types also include a set of "raw" types such as `Word32`, that are independent of the machine architecture and C compiler. Finally, basic types include several varieties of pointers. First, there are regular pointers to the C heap, parameterized with another basic type describing the dereferenced data. Second, there are C function pointers. Third, there are "stable" pointers, which are references to Haskell expressions guaranteed to never be deleted or moved by the garbage collector until they are explicitly

released. Stable pointers may be safely stored in C, without worry that they will be invalidated when control returns to Haskell. Finally, there are "foreign" pointers, a type representing a pair of a regular pointer onto the C heap along with a function pointer. The function pointer should point to a finalizer function, to be called by the Haskell when the object is garbage collected. Foreign pointers allow C data objects to be memory managed by Haskell.

The Haskell 98 FFI permits callbacks. A Haskell callback must be a function declared with the `foreign export` keywords, and defined in Haskell. The callback function is always evaluated strictly (not lazily) if invoked from C. To invoke a Haskell callback, the calling C function should be declared "safe" using the `safe` keyword. Safe foreign functions entail some extra overhead to call, but they guarantee that the Haskell runtime will be in a consistent state if a callback is invoked. Unsafe foreign functions are faster, but callback behavior is undefined. Haskell functions used as callbacks should not throw exceptions; the runtime behavior in this case is undefined.

Haskell uses monadic types to represent functions that may cause side effects. Using type inference, the Haskell compiler can (usually) construct this type information from the Haskell code. For foreign functions, however, Haskell cannot infer the type information. So, by default, the FFI must conservatively assume that all foreign functions may cause side effects. The programmer may override this assumption, asserting that an imported foreign function is pure. Pure functions have two benefits. First, the FFI assigns the foreign function a less restrictive type (i.e. it does not wrap the return type in the IO monad). Second, the Haskell runtime is free to memoize invocations

of pure function, including pure foreign functions.

3.1.7 GreenCard

GreenCard [38] is a different approach to designing a Haskell FFI to C. GreenCard focuses on providing an easy way to generate *wrappers*, i.e. layers of code that expose pre-existing libraries of C functions. Usually, such libraries are provided as a pair of files – one a pre-compiled library of functions; the other a "header" file of C declarations, including the names, arguments, and return types of each function. Where the Haskell 98 FFI requires that programmers manually declare foreign function signatures, GreenCard uses the header file to automatically generate appropriate types and glue code for each function.

Ideally, GreenCard could extract all the required information from only the header file. Unfortunately there are ambiguities. For example, C programmers generally use the type `char *` to represent null-terminated strings. But `char *` may also represent a pointer to a single character, or to an array of bytes. C does not distinguish between these cases, but Haskell, due to its strict type system, does. To resolve these ambiguities, GreenCard augments the header file with *annotations*. An annotations contains a Haskell type declaration for a C function, and the correspondence between the declared C and Haskell types resolves any mapping ambiguities. GreenCard includes a default mapping of types from C types to Haskell. If the defaults are satisfactory for a particular function, then no annotation is required.

GreenCard uses a type translation mechanism (see Section 6) called "Data Interface Schemes" (DIS). DISs are like macros or functions that describe how

a type in C is converted to its Haskell equivalent. DISs have a flexible syntax, with common cases handled by simple directives, and uncommon cases with arbitrary code. For example, there is a DIS directive for converting a C `enum` to an equivalent Haskell variant type.

Another DIS directive is used for C pointer return values. Many C functions that return a pointer will return the value `NULL` in case of a failure, or else a valid pointer. The DIS maps the pointer type to a Haskell `Maybe` type. The value will be `None` in case the pointer is `NULL`, and `Just x` otherwise, where `x` is the dereferenced pointer value.

GreenCard assumes by default that C functions may cause side effects, and wraps them in the `IO` monad. The programmer may override this assumption if they know the function is pure.

Since its purpose is wrapping existing C libraries, GreenCard restricts itself to calling C from Haskell. So, GreenCard does not include callbacks. It therefore avoids issues with garbage collection and exceptions.

3.1.8 CHASM

CHASM [52] is a restricted kind of FFI between C++ and Fortran 90 (F90). F90 programs have a compiler-dependent interface, because the F90 standard leaves many decisions to the implementation. CHASM generates wrappers around F90 procedures that present a consistent and well-defined interface. This allows C++ code to call F90 procedures, without having to modify the calls to suit a particular F90 compiler. CHASM's static analysis uses the Program Database Toolkit (PDT) [46] to parse and query Fortran and C++ programs.

There are two important facets of F90's procedure interface that are left to the implementation. The first is the way that arrays are passed to functions. The F90 standard declares that arrays are passed by "descriptor," without specifying the descriptor's exact representation. So, depending on the F90 compiler, a procedure might expect a simple pointer for a descriptor, or an integer handle, or some data structure. The second compiler-dependent aspect of F90 is procedure names. Some compilers, for example, store F90 procedures names using only capital letters, while others precede names with an underscore. If a caller does not know the naming convention, it cannot find the F90 procedure.

CHASM solves this problem by abstracting each Fortan 90 procedure, using a compiler-independent wrapper function. The wrapper uses a well-defined, compiler-independent naming scheme for Fortran procedures, and within the wrapper it calls the compiler-dependent name.

The wrapper function also presents a compiler-independent interface for passing arrays to procedures. The wrapper procedure has the same number and types of arguments as the wrapped procedure and except for arrays, these are passed through unchanged. The wrapper replaces each array descriptor argument, however, with an integer handle representing the array. This handle is an index into a globally-scoped table, maintained by CHASM. The table relates integer handles to array references. An array only needs to be registered in the table if it passes through one of the wrapper functions. Therefore, the wrappers contain all the logic needed to maintain the tables.

CHASM also includes a C++ class that encapsulates a Fortran array. CHASM will generate stubs for calling F90 procedures in C++ as well, and

these wrappers automatically convert the integer handle to the array class.

3.2 Glue code

The examples in this section have illustrated that we can categorize FFI systems according to the way they use *glue code*. Glue code is a term for program logic that helps to connect or reconcile two different representations of data or code. In general, frameworks endeavor to minimize glue code. As we have seen, FFIs are rarely able to exclude glue code altogether, but there are ways to mitigate the burden this places on programmers.

Glue code in FFIs serves to overcome semantic ambiguities where the two languages interact. These issues include pinning arrays as in the JNI, or registering the appropriate C function type signatures at runtime, as seen in SML/NJ and Ctypes. The most common use of glue code, though, is to allow the data types of one language to be interpreted and manipulated by the other. In the JNI, for example, programmers must use an API to pick apart Java objects passed to C functions. This kind of glue code is often used to address the problem of *type mapping* (see Section 6), where the "natural" data types of one language are mapped to a convenient and/or natural analog in the other. For example, many FFIs map between some high-level data type for strings and C's `char *` representation.

Many systems that we will examine in Sections 6 and 7, such as [33], attempt to automatically or semi-automatically generate FFI glue code.

FFIs may require glue code in the source language, the target language, or occasionally both. The former is useful for generating high-level "wrapper" functions for existing libraries, because the library source code may be

inconvenient or impossible to modify. This is the approach taken by Haskell GreenCard and the Haskell 98 FFI, Ctypes, and SML/NJ. The latter option, glue code in the target language, may be preferable when foreign functions are written with interoperability in mind. It allows low-level functions to work directly on complex, high-level data types, possibly avoiding data conversion. This approach is taken by the JNI and Matlab's MEX files. Fortran BindC requires little glue code because Fortran and C are fairly similar.

3.3 Type safety

A FFI cannot make strong guarantees about type safety if one of its interacting languages is unsafe. In particular, a program in an otherwise type-safe language employing a FFI to C is no safer than C itself.

Glue code should, in principle, be able to check for some kinds of safety violations at runtime. In practice, this does not seem to be a popular FFI feature. This is probably because one of the foremost benefits of a C FFI is the speed of C code, which runtime checks could degrade. None of the FFIs above include runtime safety checks.

3.4 Garbage collection

Automatic memory management, i.e. a garbage collector [11], in one or both languages presents challenges for FFIs. The most common problem is notifying a foreign garbage collector that a reference to one of its objects is held, so that the object will not be released. Usually, this must be done explicitly. Depending on the scheme, the object may have to be explicitly deallocated as well.

In addition, garbage collectors may move objects around in memory. If the garbage collector runs on a separate thread (e.g. , in Java), the object could even be moved while the main thread is operating on the data in a foreign function. In this case, the result would be a disaster – the data would literally be moved out from under the running program.

A seemingly simple solution is to suspend garbage collection for the duration of an FFI call. This is insufficient in general. Pointers to foreign objects stored across separate FFI invocations may still be invalidated (i.e. moved or deallocated) when control returns from the FFI invocation. If the FFI includes callbacks, the FFI must usually resume garbage collection when the foreign function calls back; in this case, even local pointers might be invalidated by the time control returns to the foreign code.

A better solution, used in the JNI and SML/NJ, is to "pin" data objects used by the foreign language. This tells the garbage collection algorithm that the data should not be moved or deallocated, until the pin is released. Unfortunately, pinning may complicate the garbage collector implementation. In cases where pinning is not or cannot be implemented, an alternative is to copy the data from the source language to a buffer in the target language, and then copy it back after the function completes. However, the overhead of copying may be substantial for large objects, or if it is called frequently. Moreover, copying becomes more complicated if the target language stores a pointer to the buffer; the FFI must then reconcile the two buffers on every entry or exit from the foreign runtime.

3.5 Exceptions

Exceptions are a common feature of high-level languages that can be difficult to map properly into low-level language semantics. Usually, FFIs connect higher-level languages to lower-level ones, so exceptions only cause problems if the FFI permits callbacks. In the absence of callbacks, there is no way for a thrown exception to reach a foreign function.

If the FFI has callbacks and the high-level language has exceptions, there are two approaches. The first, expedient option is to declare the program's behavior undefined when an exception reaches a foreign function. This approach is used in Ctypes and the Haskell 98 FFI. The second option, used for example in the JNI, is to set an exception flag when control returns to a foreign function. The foreign function must check the flag to see if the callback returned normally, or via an exception. If an exception was thrown, the foreign function may re-throw the exception. Or it may handle it, resetting the flag to indicate the exceptional condition was resolved, and returning normally.

3.6 Discussion

The primary goal of most FFIs that we have seen is to provide efficient access to C from a higher-level language. This goal is practical – most higher-level languages otherwise would sacrifice some or all of the low-level capabilities that C offers.

Efficient access to C is generally at the cost of the other interoperability goals. Targeting C precludes strong type checking and safety guarantees; in

fact, use of an FFI will generally compromise these properties in a high-level language that features them. Furthermore, as we have seen, the programming model for FFIs may not be as natural as possible. While function calls seem like a good enough abstraction, FFIs frequently require glue code to reconcile language differences. This glue code generally obscures the main program logic, and ensures that the interoperability is not seamless. Finally, it is clear that FFIs are not scalable, because by definition an FFI connects exactly two languages.

4 Interface definition languages

Interface definition languages (IDLs) are a popular approach to interoperability [39]. An IDL describes an interface to a software *component* (see below), in terms that are abstracted as much as possible from the underlying implementation language, operating system, architecture, network protocol, and so on. The goal of this abstraction is to allow the software to be reused in many contexts. Here, we are interested in the ways that IDLs permit components written in different languages to interact.

A key concept in IDL-based systems is *marshalling*. IDLs are used to generate code that implements the abstract interface they describe in some target language. Among other things, this involves deciding which types in the target map to those of the IDL. IDLs usually specify some binary format for data in their type system so that it can be moved from place to place and interpreted in different languages or systems. The process of translating data from a language's native representation to that of the IDL is called

`marshalling`. The reverse process, translating from the IDL representation to a native data format, is called `unmarshalling`).

Although details vary, IDL systems often work by generating *skeleton* and *stub* code. Skeleton code implements a template of the IDL-specified interface in some target language. The skeleton handles unmarshalling the arguments, invoking the correct function, and marshalling the return value. The skeleton contains *hooks*, i.e. spaces for an implementation of the interface functions to be filled in by the programmer. The stub code presents the interface in the target language, usually as a set of callable functions. Stub code marshals the arguments, finds and invokes the corresponding skeleton function, and unmarshals the return value. The stub and skeleton code work together to hide the complexities of the framework from the programmer.

There are many different IDLs, a fact which in itself deters from the IDLs goal of maximal interoperability. This proliferation reflects the challenge of designing an abstract interface language that is interoperable with many different languages, while easy to use in any given language.

In this section we give an overview of components and remote procedure calls, two branches of software design where IDLs play an important role. Then we provide examples of several IDL-based systems, and conclude with a discussion of the reasons why IDLs are popular and some of the weaknesses they entail.

4.1 Components

A software component, generally, is a piece of code that is intended to be reused. The exact definition is a matter of some debate [34]. A reasonable,

inclusive definition might be "a physical packaging of executable software with a well-defined and published interface" [34]. Software components are generally designed to be composed and reused by third parties, i.e. by programmers other than those who wrote the component itself.

Most component systems use IDLs to describe component interfaces [62]. IDLs allow components to be written in whatever language is most appropriate or convenient, while still presenting an interface that other components can consume. This approach to multi-language programming potentially allows many different languages to interoperate, in contrast to FFIs which allow only two [29].

Component-oriented software engineering describes a method for building software that consists entirely of composing systems of components [12, 41]. Components in this model are classified by their role and origins [62]. The most general-purpose components are those needed by many different kinds of applications (e.g. database access services, graphics, and so on). An application programmer rarely creates these kinds of components from scratch, since they are often available from third parties, and may be difficult to create. A middle tier of component generality encompasses those components that are needed by many applications within a particular domain. For example, many medical applications may need to access DICOM format images [2]. These components may expose a somewhat less general interface, if widely-used standards exist for the domain. These kinds of components are also rarely written solely by an application programmer, but large application developers may often contribute to or improve the components. Finally, there are application-specific components. These are always written by the

application developer, and may be difficult to reuse outside the application context for which they are created. These components may implement things like a specific graphical user interface for the application.

A *component framework* is required to instantiate components, manage their execution and interoperation, and provide semantics for component composition [49]. As we will see in the examples, the exact definition of a component is usually tied to the framework in which it operates [34].

Component-oriented application composition has several important advantages over more traditional techniques [49]. First, it has shown some promise as a means to managing the complexity of larger applications by them to be decomposed into smaller parts and facilitating code reuse. Second, they increase flexibility, as component-based applications may be easily recomposed and/or augmented with additional capabilities in response to changing requirements. The main drawback to component-based software is the increased time and effort required to design the abstract interface, and to ensure that the component conforms to the expectations of the framework. Component frameworks may also entail some performance costs. Finally, use of an IDL often entails restrictions on the form of the component's exposed interface [39]. We examine this last drawback in more detail in Section 4.5.

4.2 Examples of component frameworks

In this section we examine component frameworks that provide connections between components written in multiple languages. There are other important component frameworks, including Enterprise Java Beans (EJB) [23, 51] and Microsoft's Component Object Model (COM) [1] that we omit here be-

cause they do not support multi-language programming.

4.2.1 CORBA

The Common Object Request Broker Architecture (CORBA) [62] is a large and popular component framework standard. Components in CORBA are called "objects," although they are quite different than the notion of objects in object-oriented programming languages. In particular, CORBA components have a unique identifier, are created once, and then are accessed only through an interface which is defined in CORBA's IDL (described below).

The CORBA standard describes a framework that is based on an "Object Request Broker" (ORB), which acts as a backplane for inter-component communication. Every component in a CORBA system has access to the ORB. Many CORBA ORB implementations provide support inter-ORB communication, even to other ORB implementations [62]. The ORB has many functions, including services that allow components to be looked up either by name or by interface.

The ORB encapsulates almost all aspects of a component, isolating them from other components except for the IDL-defined interface. Hidden properties include the network location of the component, implementation details including the programming language used to write the component, and the execution state of the component (uninitialized, idle, or currently servicing a request).

CORBA's IDL is simple by design, so that as many languages as possible may be used to write CORBA components. The IDL is used to construct an *interface*, which components may then choose to *provide*. An interface which

can be thought of a set of functions with names and associated argument and return types. When a component provides an interface, it implements the functions described in the interface, using the appropriate types. CORBA can generate skeleton and stub code from an interface for any language that the implementation supports.

The IDL allows functions to take any fixed number of arguments, and to return a single value. Each argument and the return value must have a type, and the available types are specified by the IDL. The types include a set of precisely specified primitive numeric types (e.g. 8-, 16-, 32-, and 64-bit integers, booleans, 32- and 64-bit floating point numbers), and both ASCII and Unicode characters. There are fixed- and variable-length strings and lists. There are also constructed types, such as structure and union types similar to those in C, and a special "any" wildcard type. Finally, there is an interface reference type for CORBA interfaces, which can be used to pass typed references to components. Since many languages do not support pointers, the IDL intentionally lacks explicit pointer types.

The CORBA IDL requires that each function argument be marked as `in`, `out`, or `inout`, to indicate its purpose. Arguments marked `in` are effectively passed by value, and changes to the value within the function will not be propagated back to the caller. Arguments marked `out` are references; their initial value from the caller is ignored, but changes within the function will modify the referenced value in the caller's context. Arguments marked as `inout` are treated like `out` arguments, but their value at input is not ignored.

For a programming language to support CORBA, a mapping must be defined from CORBA's type system to the language's type system, and vice-

versa. Because the CORBA IDL specifies a fairly limited and common set of types, this mapping is straightforward for many languages. For example, in a C compiler, a CORBA string maps to a `char *`. In C++, it may be mapped to a `std::string`.

Language interoperability is realized by the CORBA IDL's type system, which acts essentially as a commonly-understood, intermediate format for data exchange across languages. Note that since the IDL includes a notion of references to other components, it also allows CORBA to act as a kind of FFI between any two languages that the implementation supports, allowing function calls from one language to another.

In practice, many popular languages provide support CORBA integration in the form of a mapping from their basic types to CORBA's IDL [62]. This support makes CORBA a practical choice for multi-language programming scenarios that require interoperation of two languages which do not have a dedicated FFI. It is also useful for situations where more than two languages are required.

4.2.2 CCA

The Common Component Architecture (CCA) [12] is a component framework for high-performance scientific applications. CCA is strongly influenced by CORBA, and the two systems have much in common.

Scientific application developers are good candidates to adopt component-based software engineering. Reuse of highly complex and specialized scientific codes is highly desirable, and the nature of scientific research, especially the need for repeatable experiments, encourages sharing [41]. The CCA was

created to address these requirements.

CCA describes component interfaces using an IDL called called SIDL [58]. SIDL is essentially an extension of CORBA's IDL that adds types of particular interest to scientists. These include multi-dimensional arrays (with either fixed or dynamic size) and complex numbers. A tool called Babel implements a SIDL parser, and can generate stub and skeleton code in C, C++, Fortran 77, Fortran 90, and Python.

In CCA, interfaces are called *ports*. A component may *provide* a port, which means that the component implements functions that match those described in the port. A component may also declare that it *uses* a port. This means that the component requires an implementation of a component that provides that port to be loaded in the component environment. This system allows for different components that provide the same functionality (i.e. provide the same port) to easily be swapped in and out of an application [12].

CCA does not currently support type mapping (see Section 6) beyond the default primitive and structure maps in CORBA, although recent work has been moving towards this goal [36].

4.2.3 Cactus

Cactus [26, 27] is a component framework with a focus on scientific applications. It does support components written multiple languages, but it does not use an IDL. Instead, it limits the languages that components may be written in to C, C++, and Fortran, and uses their existing interoperability facilities for inter-language component communication. In particular, C++ can call C functions directly since C++ is derived from C and shares much

of its architecture, and Fortran has a bi-directional FFI to C (see Section 3).

4.3 Remote Procedure Calls

Remote procedure call (RPC) is an approach to inter-process communication, where each participating processes is assumed to be running on a separate computer connected by a network [32, 18, 61, 31]. The mechanism disguises itself as a procedure call, but after the call is made, the RPC system packages the function and its arguments in a binary message, and sends it to be handled by some other process. How the receiving process is chosen and located is a feature of the particular RPC system, and in general the receiver may be running on a different machine. After the message is received, the function and arguments are decoded, the function is executed, and the return value is passed back to the calling process. When the caller receives the response, it decodes the value and returns it via the regular function call return mechanism. From the caller's point of view, this entire process is indistinguishable from a regular function call.

RPC's straightforward semantics have made it a popular approach to distributed programming [61]. Also, and more importantly for our purposes, RPC systems usually allow for multi-language programming [32]. Their use of IDLs to describe exposed procedures has the effect of abstracting language-specific details, allowing for RPC to work across any language that supports the particular RPC protocol.

In fact, RPC can be thought of as a general approach to handling heterogeneity in computing systems [50], since it can be designed to abstract different operating systems, machine architectures, programming languages,

and networking, all under the guise of a simple procedure call.

Both component frameworks and most RPC systems make use of IDLs [32]. In RPC, the units of interoperability described by the IDL are procedures or functions rather than components, but the approach is very similar.

Two models of RPC semantics are popular: blocking and non-blocking [61]. The blocking model uses strives to emulate the semantics of a normal procedure call, i.e. from the point of view of the caller, execution is suspended until the call returns. In the non-blocking model, the call is still made normally, but for the caller, execution continues immediately and does not wait for the remote call to return. Later, the caller can use the RPC's API to check whether or not a response has been received, and to collect the return value if one is available. Many systems provide both blocking and non-blocking calls. For the purposes of language interoperability, the distinction is immaterial.

RPC designs generally favor hiding the details of the network communication from the caller. Ideally, the caller need not even be aware of whether a call is remote or local. This abstraction principle is somewhat leaky, however. First, RPC calls must generally avoid passing arguments that are tied to the local address space (e.g. a pointer to an array), since the callee might be located in a separate address space. Second, the possibility of network errors implies that RPC systems must handle failures that regular procedure calls do not [18, 50]. For example, the correct semantics for RPC are unclear if the network stops functioning entirely, and must be defined by the implementation or standard. More subtle network failures are also possible, and RPC systems must be careful in the design of their protocols to ensure that, for example, failures do not cause a single call to result in more than

one execution of a remote function [59].

There are several issues in RPC systems that we will not examine here because they are not relevant to our discussion of multi-language programming. In particular, RPC systems are often concerned with their performance under different network configurations and parameters, as well as the security implications of exposing program functions on the network [61].

4.4 Examples of RPC systems

In this section we will examine several RPC systems and show how they facilitate multi-language programming. One important RPC system, Java Remote Method Invocation (RMI) is omitted because it is tied to Java and does not directly support multiple languages [65].

4.4.1 Sun RPC

Sun RPC introduced the "External Data Representation" (XDR), an influential and at the time innovative IDL design [21]. Sun RPC was originally designed to enable network distributed function calls to and from C, and not necessarily for inter-language function calls. So, XDR's data types look a lot like those of C. In addition to the usual primitive types (integers, floating point, characters, and so on), XDR supports C-style structures and unions, although these may only be one level deep (e.g. no `structs` within `structs`). XDR also supports fixed-length strings, as well as fixed- and variable-length arrays of primitives. Finally, XDR supports an "opaque" data type, that represents a sequence bytes guaranteed not to be modified by marshalling and unmarshalling [60]. XDR does not support explicit pointer types, since a raw

pointer value has no valid interpretation outside its local address space. The mappings to and from these types and their representations in C are fixed by XDR.

XDR was designed for C, and C's native data representation is tied to the machine architecture and compiler. This explains why XDR's binary data representation is so precisely specified, since it must accommodate marshalling between data formats that may have different integers lengths, different endianess, different ways of packing `structs`, and so on. This is also why XDR is a useful system for interlanguage communication – by abstracting the data transport representation from the in-language representation, and keeping its supported set of types minimal and fairly universal, XDR becomes a language-independent standard.

4.4.2 ILU

Inter-Language Unification (ILU) goes beyond traditional RPC systems in two ways. First, it explicitly focuses on facilitating multi-language programming through an RPC-based approach. Second, it creates a system where objects may be passed by reference through the IDL interface [37]. Objects in ILU are not merely static collections of data; as in OOP languages, object instances have a unique identity, may have methods, and those methods may be invoked anywhere a valid reference to the object is held, even from other languages.

ILU works much like Sun RPC, but adds a notion of *modules*. In an application, there may be only one instance of a module, and a module has exactly one interface, specified in the IDL. Modules reside permanently in

one address space, on one machine, and each module is written using one language. An application consists of a set of module instances.

In addition to regular procedures, modules may expose objects through their interface. Objects have a type, which is specified by another IDL interface with a set of methods. Object instances are owned by the module that exported the object's type. However, a *reference* to the object may be obtained by other modules. This reference may be used to invoke the object's methods, which are executed via RPC to the owner module.

ILU provides a garbage collection scheme for objects. The module that owns an object instance is responsible for disposing of the object when there are no longer any references to the object. To manage this in ILU's distributed environment, ILU must arrange for the owner module to keep track of external modules holding references to its object, and periodically query them over the network to see if the reference is still held.

The ILU implementation described in the paper supports modules written in Modula-3, FORTRAN 77, C++, Lisp, and Python [37].

4.4.3 XML-RPC

XML-RPC [9] was designed to be a simple RPC protocol that would be easy to support from a number of languages. It uses an XML format instead of binary to represent a marshalled data, and provides two formats: one for function call requests and another for responses. XML-RPC does not provide a dedicated IDL; instead it describes a set of data representations in terms of XML, and expects implementations to define their own mapping from the language to those representations.

XML-RPC provides XML encodings for the usual primitive number, boolean, and character types, as well as variable-length strings and arrays of primitive types, and structures of arbitrary nested depth.

XML-RPC specifies an extra field in the response encoding, the presence of which indicates that an error occurred during the function execution. The interpretation of the error code depends on the implementation.

4.4.4 Web Services

Web services are a general-purpose approach to "distributed services" [6, 22]. In practice, web services are used to provide RPC functionality on top of web protocols and standards like XML and HTTP.

The design of web services encompasses three orthogonal components required for distributed services. These are a communication protocol to encode data (analogous to a format for marshalled data in RPC), a way to describe services (analogous to IDLs), and a method for discovering services on the network.

SOAP is the most common communication protocol used in web services [6, 22]. Like XML-RPC, it is based on XML, and describes formats for requests (function invocations) and responses (function returns). SOAP is more complex than XML-RPC, and specifies formats for meta-data like security credentials.

The XML standard has a specification, called XSD [17], for encoding primitive and structured data in an XML format. These include numbers, booleans, structures, arrays, and so on, as well as some higher-level types like dates and times. XSD also supports construction of new data types

from this primitive set using sequencing, discriminated union, and restriction. This last mechanism is unique, and interesting: it specifies or restricts ranges of valid values that the underlying type may take. While decidedly more complex and verbose than the data encoding used by XML-RPC, XSD has the advantage that encoded data can be checked for correctness at runtime using XML processing tools that are available for a wide variety of platforms and languages. For example, integers can be verified to be within the representable range, strings can be checked to contain only valid Unicode characters, and so on.

A second XML format, called WSDL [22] provides a way to describe the web service interface. WSDL, then, acts like the IDL in a traditional RPC mechanism. A WSDL specification defines a set of valid *messages* in terms of XSD types. These messages are used by a set of *ports*, each of which contains a set of *operations*. Each operation describes a sequence of valid request and response messages. These operations might be similar to regular RPC semantics (i.e. a caller request, followed the callee's response), asynchronous messaging (i.e. just the request, with no response expected), or some other protocol entirely. WSDL therefore trades simplicity for flexibility; it can describe very complex protocols.

A WSDL description also contains a concrete binding, that tells the service what encoding and transport protocols to use (e.g. SOAP over HTTP). The concrete binding also maps operations to URLs, which gives them a globally unique endpoint for communication.

Like an IDL, the WSDL specification is processed through a tool that generates a skeleton for the implementation of the service in the desired

target language. A stub implementation can also be generated from the WSDL. Typically, WSDL for a service is made available on the internet, and clients who wish to use the service may download the WSDL interface and generate stubs in the language of their choice.

While complex, the XML standards that web services are built upon are standardized and implemented in a wide variety of languages and systems.

4.4.5 Thrift

Thrift [57] is a recent RPC system developed at Facebook, with a focus on multi-language interoperability. It uses an IDL that, in addition to the usual primitive and structured types, includes data structures, such as maps and sets, common in modern "scripting" languages (e.g. Python, Ruby).

Thrift abstracts the marshalled representation of this data with a functional interface. So, marshalled data structures may be constructed or picked apart using and API with functions like `writeInt`, `writeStruct`, or `endStruct`. This API has been ported to C++, Java, Python, PHP, and Ruby, so Thrift can marshall data to and from a variety of languages.

Instead of using the API functions, most users of Thrift describe data structures in an IDL. The IDL is processed in the usual way, producing marshalling and unmarshalling routines for the desired language. These routines are generated with calls to the API. The benefit of this approach is that the API routines may be rewritten, and the underlying data representation altered, without requiring any changes to the client code.

The IDL can describe functions, and these have the same semantics as RPC. Functions that have no return value may be marked with the `async`

keyword, which allows callers of that function to continue execution without waiting for the call to return.

4.5 Discussion

Relatively little effort is required, in general, to implement a mapping of a language's basic data types to those of an IDL, and to enable the IDL tools to generate stub and skeleton code for that language. This allows IDL-based approaches to multi-language programming to scale well – if you have n IDL-mapped languages, then you have n^2 language bindings, since any language in the set can interoperate with any other [39].

The cost of this approach is a lack of specificity to any particular language and set of types. In particular, domain-specific data types (e.g. complex numbers, matrices, images, and so on), will need to be expressed in terms of the simple primitives and structures that most IDLs offer, and not the more natural high-level types that some languages may offer. IDLs cannot easily get around this limitation, because the types they define must constitute, in some sense, a *lowest common denominator* type system across any and all languages that wish to participate [39]. If an IDL included say, a type for images, then each language would have to be able to decode that image type into meaningful data (or accept an incomplete mapping, but this defeats the purpose of interoperability). As we will see in Section 6, customizable *type maps* can help to resolve this issue.

IDL-based interoperability systems usually require marshalling for complex types, even for communication between components written in the same language. Therefore, use of an IDL may be inefficient compared to other ap-

proaches.

5 Language-neutral intermediate representations

Language-neutral intermediate representations (neutral IRs) are, in some sense, similar to IDL-based approaches. Both work by constructing a "common ground" that participating languages must be able to interface with. This approach connects each language, by transitivity, to every other language that targets the same IR.

In the language-neutral IR approach, each language is compiled to some target language (the IR) that is general enough to encode the data representations and semantics for a variety of languages. The IR serves as the mechanism for interoperability. The nature of the IR determines how interoperability mechanisms are exposed in each languages. Neutral IRs, then, can be seen as providing a framework or mechanism for multi-language interoperability, on top of which other strategies (including FFIs or language integration) can be applied.

5.1 Examples

The following systems all feature multi-language interoperability facilitates by a language-neutral IR.

5.1.1 UNCOL

As far back as 1958, there was interest in developing a "universal compiler IR," called UNCOL [47]. UNCOL was more a concept than an actual proposal, and it was never successfully designed. The idea was proposed as a means to reduce the effort required to write compilers, which was at the time considerable. Machine architectures at the time were not standardized, and language features like records, pointers, and data types were still novel. A universal IR was thought to be a partial solution – languages could target the IR, which would then be portable to many hardware architectures. Language interoperability was hardly considered, but was mentioned as a potential extra benefit.

5.1.2 Microsoft Common Language Infrastructure

Microsoft's Common Language Runtime (CLR) [10, 30] is a language-neutral IR and execution specification, similar to Java's virtual machine bytecode [45], but designed to support many different languages. The CLR supports compilation and execution of procedural, functional, or object-oriented paradigms languages. CLR's language-neutral bytecode format is called Common Intermediate Language (CIL), and it has a language-neutral type system called the Common Type System (CTS).

Like the JVM, the CLR provides high level services like garbage collection, a class loader with security features, and an extensive (and language-independent) class library providing many common data structures and algorithms.

CLR programs are self-describing, with extensive annotations describing

the types, fields that are read-only, and so on. These annotations are packaged along with the compiled CIL code. This allows compiled CIL code to be distributed independently of the source program that generated it, but still inspected and used by other CIL modules.

When executed, the platform-independent CIL is translated to platform-specific native code. CIL is a stack-based model, and although it retains type annotations as meta-data to be used by compilers and other tools, the execution engine ignores the types entirely.

The CIL is essentially a stack-based form of assembly language. Unlike assembly languages, however, it is not tied to a particular ISA, and is designed to be interpretable on any modern CPU.

Although similar to JVM bytecode, CIL has some important differences that facilitate its language neutrality. Unlike the JVM, unsafe CIL codes can be generated and are permitted to be executed. This enables languages like C++ that feature unsafe pointer arithmetic to interoperate. Also unlike JVM bytecode, CIL permits global variables, function pointers, and the ability to pass primitive parameters by reference. The garbage collection algorithm is required to support pinning data (see Section 3), which guarantees that pointers may be used on memory managed data.

The CTS provides a language-neutral type system for the CLR. There are two kinds of types in CTS: value and reference types. Value types are bit sequences (e.g. integers), allocated on the stack. The value type describes what operations are appropriate, but values do not carry their type information, so type checking must be done statically. References types are similar to object references in Java. Values of reference types carry their type information

with them, and so can be checked dynamically.

Like Java, reference types are part of an inheritance hierarchy, and must inherit from exactly one parent. Value types exist outside the hierarchy. The CLR has a notion of *interfaces*, which are sets of methods. Inheriting from an interface implies that the reference type implements all the methods described in that interface. Inheritance from multiple interfaces is allowed, and the sub-typing rules permit a reference type that inherits an interface to be used wherever that interface is expected.

Notably, although the CTS defines inheritance behavior, it intentionally omits method overloading. This is because different languages have different overloading rules. It is relatively easy for languages that wish to support overloading to implement it themselves by mangling method names.

To support language interoperability, CLR defines a Common Language Spec (CLS), which is a subset of the CTS type system. At a minimum, CLS compliant languages must be able to import and use CLS types, which include object types as well as a set of primitives. Notably, CLS-compliant languages need not be able to *extend* object types via inheritance, or even to define new object types. CLS specifies some other constraints as well, for example it mandates an interoperable naming scheme for identifiers, and disallows architecture-specific primitive types. Compliance with CLS is not required, it guarantees interoperability with any other CLS compliant code. In some ways CLS is like an IDL, but an exceptionally rich one.

Error handling in CLR is done through exceptions, so compliant languages are required to either handle or tolerate exceptions that are thrown to them. Exceptions are implemented by a two-pass stack unwinding: first

the stack is searched for a handler, and then the second pass performs cleanup before invoking the handler. Exception information is stored in fixed-format table that precedes each method entry in the CIL format. The table contains a pointer to the handler as well as a discriminator indicating whether the handler may re-throw the exception or simply terminate. Exception handling is late-bound; no work is done until the exception is thrown, and then the stack is searched. With this system, an exception can be raised in one language and caught in another.

Targeting a particular language to the CLR can be easy or difficult, depending on how well the language's concepts map onto the CLR's infrastructure. For example, the CLR does not support nested functions, multiple inheritance, or callcc, so languages with these facilities must either omit the feature, or transform the feature into CLR-compatible terms. Moreover, in order to interoperate with other languages in the CLR, a language must support the minimum requirements of the CLS. This can be awkward. For example, support for SML required an extension to the language to support object types.

5.1.3 LLVM

LLVM [42] specifies an intermediate code representation based on a 3-address, RISC-like architecture, but abstracted from actual hardware. It also provides a compiler infrastructure to transform that representation, including compilation to real hardware.

LLVM's representation is language independent; it is only slightly richer than a RISC-like assembly language. It does include a type system, but

the types may be treated as annotations, and do not prevent definition of programs that ignore the type information. LLVM does not impose any particular runtime requirements on programs, and does not provide high-level runtime features (e.g. garbage collection) directly. This makes LLVM a very different kind of system than the CLR or JVM, which provide many high-level features but also usually require that compliant languages use them.

The LLVM IR has just 31 opcodes, but most have overloaded semantics based on the types of their arguments. LLVM does not permit type coercion; types must be cast explicitly to other types if desired, so language features like implicit coercions must be compiler directed. LLVM's IR uses an infinite set of virtual registers. The definition of a register always dominates its use, i.e. registers have single-assignment semantics. The IR includes load and store opcodes to access a heap. Control flow in LLVM is explicit: functions are basic blocks, and each block ends in either a branch, a return, or one of two exception opcodes (see below).

LLVM's type system is designed to be language-independent. Every register and heap object has an explicit type, and opcodes work differently with different types. The type system includes the usual set of primitive types, as well four kinds of derived types. These are pointers, arrays, structures, and functions.

LLVM is designed to support weakly-typed languages (e.g. C) so declared type information in an LLVM program may not be reliable. In particular, there is an unsafe opcode that will cast any type to any other type. The cast opcode is the only way to convert types; this implies that programs without

the cast opcode are typesafe. [2] Heap address arithmetic uses a dedicated opcode (instead of the general-purpose addition opcodes) that preserves type information.

LLVM uses a flexible, language-independent scheme for exceptions. The IR includes two special opcodes called invoke and unwind. Invoke works like a regular function call, but takes an extra basic block argument that represents an exception handler. When the unwind opcode is executed, it works up through the call stack until an invoke opcode is found. Execution then transfers control to the exception handler block of that invoke.

Type information allows a range of aggressive transformations that would not otherwise be possible, e.g. reordering fields; optimizing memory management. These can only be done with *reliable* type information however, so they include an algorithm (Data Structure Analysis) that uses the declared types as speculative, and conservatively checks whether load/stores are consistent.

5.1.4 Moby

Moby is a functional programming language that supports interoperability with C through its compiled intermediate representation, called BOL [24, 55]. BOL is more expressive than Moby itself; in particular, it can also be used as an IR for C. The two languages have very different features, but can interoperate using BOL.

The BOL IR framework serves as a *mechanism* for interoperability, but does not define a *policy*. The distinction is important. The policy determines how low-level data structures are represented and manipulated in the

[2] Lack of the cast opcode does not prevent memory errors, such out-of-bounds memory or array accesses, from occurring.

high-level language. The mechanism, by contrast, exists to reify the policy. Ideally, if the interoperability mechanism is both flexible and powerful, it may support many different kinds of policies.

BOL is an extended, low-level lambda calculus. It has a weak type system, designed to be almost equivalent to that of C, but lacking C's recursive types. Type constructors in BOL include enumerations, pointers, arrays, and structures. BOL code can make C function calls and work with C data types directly. No marshalling is required because there is a direct mapping from BOL's types to those of C.

Types in Moby can be defined in terms of BOL types; this is Moby's primitive types are defined. Primitive Moby functions can also be defined in BOL; this can be used, for example, to wrap the C standard library for use in Moby, since BOL can call C functions directly.

There are two separate ways to access C from Moby, representing two distinct interoperability policies. The first is an IDL-based approach. Tools are used to parse a C header file and map the function signatures to Moby's type system. The header file may include some extra annotations to disambiguate the mapping of pointer types. Stubs are generated in BOL code that call the appropriate C functions, but map the types to Moby's high-level representations. Some marshalling code, also in BOL, may be generated if needed to implement the mapping.

The second interoperability policy is called Charon, which implements a type-safe embedding of C into the Moby language using phantom types. This allows Moby to manipulate C data structures and call C functions directly, without sacrificing type safety. Charon was inspired by, and is very similar

to, NLFFI [19], which is discussed in Section 7.1.1.

5.1.5 Whirl

Whirl is the IR used in the Pro64/Open64 compiler suite [8, 48, 44] and its many offshoots. Whirl is designed to be used as the input and output of every internal compiler pass. This makes it easy to reorder optimization passes in the compiler, which is important because the optimal order of the optimization passes with respect to some performance criteria may be different for different applications.

Whirl was designed to support a fixed set of languages, namely C, C++, Java, and FORTRAN 77, and it may be adequate for other languages as well.

Whirl code may be designated as being at a "level" with higher levels being closer to the source language, and lower levels closer to the machine architecture. The process of compilation, then, is a gradual translation of a program from the highest level to the lowest one. Each level is well-defined, i.e. there are constructs in the higher-level IR that must be eliminated before the level can be reduced.

At the highest level, Whirl code is very close the original language, and at this stage it is even possible to translate from Whirl back to the original language. Whirl supports this high-level representation through constructs that are specific to different language semantics. For example, Whirl contains a `DO_LOOP` element that corresponds to Fortran's loop semantics, as well as `DO_WHILE` and `WHILE_DO` elements for C. At the highest level, language-specific types are preserved as well.

While Whirl does provide a language-neutral IR for the languages it sup-

ports, it would have to be extended for an additional language to participate. Moreover, Whirl does not offer any special support for interoperability between the languages it supports. For example, Whirl does not bother to define an IR representation of C code calling a Java function, since it does not accept a source language that would admit this construct in the first place.

5.1.6 SUIF

SUIF [66] is a compiler infrastructure that includes a flexible IR component. SUIF is particularly focused on facilitating parallelizing transformations. To detect the data dependencies required for parallel transformations, SUIF uses a fairly high-level intermediate representation. Lower-level IRs often erase high-level representations of loops and conditionals, and array accesses are expressed as pointer arithmetic, and this makes certain data dependence analyses more difficult or impossible.

The SUIF IR includes standard RISC-like operations, but also higher-level representations for various loops, conditional statements, and array accesses. The loops and conditionals representations are similar to those in an abstract syntax tree, but are designed to be independent of a particular language.

Because it preserves high-level constructs, SUIF may be a good choice for a language-independent IR. At the moment, the SUIF distribution includes front-ends for both C and Fortran.

5.2 Discussion

C is something of a *lingua franca* among programming languages, not unlike a universal intermediate IR. As we saw in Section 3, most mainstream languages have some kind of ability to interoperate with C. C enjoys this status for at least two reasons. First, it is a highly desirable language to interoperate with owing to the vast numbers of existing libraries it can access. Second, C may be used for programming tasks that high-level languages are ill-suited for, such as coding of performance-sensitive algorithms that can take advantage of specialized hardware. In some ways, though, it makes for a poor language-neutral IR, because it lacks high-level features like garbage collection and exceptions.

In contrast to C, frameworks like the CLR have lots of high-level features, and support a more coherent form of interoperability because those features can be shared across languages. By the same token, however, requiring that these features be supported (or at least tolerated) by participating languages may constrain language implementations, and prevent different but useful approaches. This in turn may limit the utility of multi-language programming, reducing language specialization to a matter of syntax in the most extreme scenario.

Language-neutral IRs provide a mechanism that facilitates multi-language interoperability and programming. Some systems, like the CLR, use this mechanism to provide multi-language interoperability by designing participating languages with certain unified language semantics. Other systems, like BOL and Moby, use the neutral IR as a framework for standard approaches like FFIs. Each case, therefore, meets our goals for multi-language

interoperability differently, but the use of a neutral IR impacts the goal of scalability. Like IDLs, neutral IRs allow any number of languages to interact through a single common representation – this reduces the number of required interfaces for n languages to $2n$. However, the cost to add each language amounts to the cost of writing a full compiler that targets the neutral IR.

Whether a neutral IR can accommodate multi-language type safety, offer a natural programming model, and operate efficiently depends on the system in question. The CLR, for example, goes to great lengths to fulfill these goals. It largely succeeds, at the cost of restricting language features. Systems like Moby take a different approach, allowing different interoperability policies to be layered on top of BOL. These policies, then, determine whether and how interoperability goals are fulfilled.

6 Type mapping

In multi-language programming, there is always the question of how to relate the data types in one language to the types in another. For low-level, primitive types such as integers and floating point numbers, the mapping is usually obvious because both languages are likely to have to types that correspond very closely, if not exactly. The question becomes more complicated, however, for higher-level data types. For example, consider the following C `struct` and function declaration:

```
struct Point {
  double x, y;
```

```
};

double norm(struct Point *p);
```

How would an IDL or FFI expose `norm` to, say, Java? One possibility would map the `Point` structure to the `java.awt.Point` class, and construct a `norm` method in Java like so:

```
double norm(java.awt.Point p);
```

There are other options, as well. For example, `Point` could be mapped to `java.awt.geom.Point2D.Double`, or even to a new point class, defined by recursively mapping individual elements of the structure to corresponding new instance variables. A *type map* is some data structure or algorithm that allows these cases to be disambiguated. Type mapping is a common feature across many kinds of multi-language interoperability systems, because it is difficult in general to match types across languages. These systems may be quite simple and/or inflexible. In particular, many systems require that user-defined types by translated to some simpler type with a known cross-language mapping, in order to be passed across language boundaries. The examples we cover here are more involved. In particular, high-level type mapping systems allow arbitrary user-defined types to be mapped to across language boundaries in ways that are, to some extend, customizable by the programmer.

6.1 Examples

In this section we will examine some examples of systems that have popular or interesting approaches to high-level type mapping.

6.1.1 SWIG

SWIG [15] is a "wrapper generator", intended to generate tedious glue code needed for higher-level languages to interact with lower-level ones. It has a particular focus on connecting so-called "scripting" languages, such as Python, Ruby, and Perl, to pre-existing libraries written in C. SWIG can be configured to generate any kind of glue code, so the exact output will depend on the FFI system used. To call a C function from these languages, glue code must be generated that converts the function arguments from the scripting language representation to some representation in C, invokes the C function, checks for possible error codes, converts the return value to a high-level representation, and finally transfers control back to the high-level language.

To direct the code generation, the SWIG tool takes an interface file as input. The interface consists of ANSI C function prototypes and variable declarations. SWIG handles simple primitive type conversions automatically. By default, SWIG does not perform sophisticated type mapping. Instead, it simply converts pointer types in C to an "opaque pointer" type in the high-level language. Opaque pointers are represented as the pointer's contents (i.e. an address in memory) encoded as hexadecimal strings. Null pointers are converted to the string value "NULL". This scheme treats pointers as opaque handles, which may only be used by passing them to (appropriately

typed) wrapped C functions.

If a more involved mapping of pointer types is desired, SWIG has a notion of type mapping functions [7]. A type mapping function is a block of glue code that implements a conversion from a low-level C data type to some high-level type in the scripting language, or vice-versa. In general, to convert from C to a high-level language and back requires a pair of symmetric type maps. The code in a typemap can be arbitrarily complex. For example, it could pick apart a C structure, field by field, instantiate a new object in Python, set some fields in the Python object instance, and then return the object.

To insert the code block that implements a type conversion, SWIG must be able to recognize the data types to which it applies. SWIG uses a regular expression pattern, paired with each type map definition, to match types lexically. SWIG extends the pattern matching to account for things like aliased types. Once a type is matched, the type mapping code will be inserted into the generated glue code at the appropriate point to convert the data type.

Parameterizing SWIG with a set of typemaps defines a type mapping *policy*. By changing the typemaps, the policy will be changed. SWIG is distributed with a set of default type maps, but users are free to define their own and augment or replace the defaults.

6.1.2 FIG

FIG is a tool used to wrap existing C libraries for Moby [55]. Since Moby already uses BOL as a language-neutral IR (see Section 5.1.4), the mechanism for interoperability with C is already in place. Fig's job is to fix a policy that maps types in C to types in Moby, and is able to generate the BOL code

that implements the conversion.

The input to FIG is a C header file along with a script that is used to guide the translation of types. Internally, FIG represents the contents of the header file (i.e. function type declarations) as a set of typed terms, and transforms the terms from C to Moby through the application of *term rewriting* rules (see Appendix A). Typemaps in FIG are built from rewriting rules whose input terms are BOL types, and whose output is either another BOL type or $\perp$, which indicates failure. The use of BOL's language-neutral type representations allows type terms to be uniformly represented in BOL, while still translating between Moby and C.

A typemap in FIG is a 4-tuple, consisting of a high-level type T_m, a low-level type T_c, a marshalling function that converts a value of type T_m to a value of type T_c, and an unmarshalling function that converts values from T_c back to T_m. The marshalling and unmarshalling functions are defined in BOL. When FIG needs to generate conversion code for a type, it simply checks to see if a registered typemap takes that type as input, and then inserts the marshalling or unmarshalling code to perform the conversion.

FIG also defines a combinator language that allows simple typemaps to be composed into more complex conversions. The composition rules are based on term rewriting *strategies* (see Appendix A). Strategies allow flexible composition of typemaps, including simple constructions like sequencing (i.e. the output of one typemap is sent to the input of another), or more complex rules like mapping a typemap across the elements of tupled types. Typemap combinators are a useful and powerful feature that distinguishes FIG from other approaches to typemapping.

FIG also has a pattern matching feature, not unlike that of SWIG, that can select terms based on identifiers or other lexical constructs. This can be used, for example, to disambiguate cases when two typemaps may apply to the same type, or to apply certain typemaps only when particular naming conventions are being used in the header file.

6.1.3 PolySPIN

PolySPIN is a multi-language interoperability system designed for object-oriented languages [14]. It seeks to make interoperability for any number of OOP languages totally *seamless*, i.e. such that a programmer need never be aware that they are working with objects written in a different language. PolySPIN rejects the IDL approach, since it requires programmers to define an interface in terms of the IDL. Even if the interface is automatically generated, it does supports only a limited type system, and so using the interface is unnatural when high-level types, including objects, are involved.

PolySPIN's approach to interoperability is based on automatically generated mappings between high-level types, using a process called "type matching". Matching objects are pairs of inter-language object types that have equivalent, or roughly equivalent, interfaces. PolySPIN implements this check using *signature matching* [67]. Strict signature matching works by equating a set of primitive types across languages (e.g. a Java `int` is defined to match a C++ `int`), and then recursively equating constructed types, including method signatures and objects. Matching can also be relaxed in various ways; for example, an object may match another object if they match strictly except that the first object contains additional methods that are not

in the second object. Matching criteria in PolySPIN can be defined by programmers, and may include lexical criteria such as a type's identifier.

To make use of matching, PolySPIN introduces a system that binds a unique name to each object in a multi-language application. The methods of each object are modified so that they consult this binding whenever they are invoked. If the underlying object is instantiated locally in the calling language, the method is invoked normally. But it is also possible that the object is acting a facade to a matched object in another language. In this case, when a method is invoked, PolySPIN redirects the method call back to the underlying object in the original language. Facade objects may be obtained by passing regular objects across language boundaries, for example in a method's arguments. In any case, when an object is passed to a different language, a matching type is found, and is used as a facade to the original object.

It is worth noting one drawback of PolySPINs approach – because PolySPIN must modify the methods of each class, the full program source code must be available. So, existing libraries for which only the compiled code is available cannot be used with PolySPIN.

6.2 Discussion

The systems in this section illustrate the use of type maps in multi-language programming. Type maps are best viewed as a feature of these systems, rather than a general approach to interoperability. The goal of high-level type maps, generally, is to make programming in multi-language applications more natural by allowing programmers to use familiar high-level types instead of a

restricted, predefined set interoperable types. As with all marshalling code, high-level type maps may be at odds with efficiency, since they potentially introduce extraneous transformations.

7 Language integration

Integrating two or more languages involves reconciling the syntax and semantics of those languages. The combination may produce an entirely new language, or one language may be entirely expressible in terms of another.

7.1 Examples

The following systems are examples of systems that achieve language interoperability through an integration.

7.1.1 NLFFI

NLFFI embeds C's type system within SML, allowing for data-level interoperability with C code [19]. It allows SML to call C functions, without the need to map types or marshall data, since ML can just pass the arguments in the representation that C expects. The tradeoff is that, since C's type system is not safe, ML programs that use NLFFI are no longer guaranteed to be safe either.

NLFFI uses a type system "trick" (see below) to set up a one-to-one correspondence between types in C and a set of types defined in ML. The ML compiler is modified to generate C code for expressions that have these

types. The embedding is rather complicated; here, we give a few examples of the kinds of techniques they use.

NLFFI relies heavily on *phantom types*, which are type constructors used only to construct other types, and not assigned values. Consider the following ML type:

```
sig type bin
  type binary
  type 'a dg0
  type 'a dg1
end
```

These types can be used as a kind of "type language," to construct binary numbers. For example, the type `binary dg1 dg0 dg0` represents 100_2, or 4 in base 10. In NLFFI the technique is only slightly more complex. Specifically, NLFFI uses a decimal number representation instead of binary, and introduces extra type parameters to prohibit leading zeros, which ensures that each number corresponds to a unique type.

For example, to encode C's fixed-size array types, NLFFI uses phantom types in a one-to-one relation to non-negative integers to type fixed-length arrays, by using the phantom type to indicate the array's length. With this scheme, C arrays of the same length will have the same ML type, which conforms to C's typing rules.

As another example, NLFFI encodes C's `const` pointer type parameter by including a tag in the C pointer type. The tag itself is a phantom type that corresponds to two values, `const` or non-`const`. NLFFI provides functions

to read and write data from a pointer, and these functions take the pointer type as an argument. While the read function is polymorphic in the `const` tag, writing is strict. This technique allows ML to statically reject code that attempts to write to a `const` pointer.

NLFFI does not address some of the usual aspects of multi-language programming. In particular, it does not allow C to access SML's data or functions. NLFFI is therefore a one-way integration.

7.1.2 Jeannie

Jeannie [33] combines the syntax and semantics of C and Java, and allows each to be embedded in the other recursively. Programmers switch between languages using a special operator (Jeannie uses the backtick character to switch from Java to C, or from C to Java). The Jeannie compiler produces two separate programs, one in Java and the other in C, and connects them with automatically generated JNI code (see Section 3.1.2).

The complete syntax of each language is supported, but the switch operator must be used explicitly to change from one language to the other. The switch operator may be applied to either statements or expressions. Jeannie adds some extra syntax for convenience, such as synchronized blocks and exception constructs like `try`, `throw`, and `catch` for C, as well as extra functions, such as a version of `memcpy` that copies a block of memory from Java's heap to C's.

Jeannie statically type checks the separate Java and C code according to their own typing rules. It also adds static checks across language boundaries, which are not normally performed on JNI code. For example, there are cross-

language checks to ensure that checked exceptions are either caught locally or declared, and that private and protected Java methods are not called from C. Because it incorporates C, Jeannie is not type safe. Cross-language checks are an improvement over hand-written JNI in this regard, however.

Jeannie uses the same type equivalences defined in the JNI, but also checks that they are used correctly. C constructs like explicit pointers, `struct`s and `union`s have no equivalents in Java, and so Jeannie checks to make sure they do not cross over to Java without being explicitly marshalled.

Operators like `break` and `continue` cannot divert control flow across language boundaries, since the JNI does not support it. Other control flow operators, such as `throw`, are able to divert control flow because they have a semantics in C that is defined by the JNI (see Section 3.1.2).

7.1.3 MLj

MLj [16] is a Standard ML (SML) compiler that generates Java bytecode. It includes an extension to SML, allowing it to call Java code. The "semantic gap" between the two languages is fairly small. Java and SML both have strong typing, similar basic types, and checked bounds. They use similar exception semantics. Both prohibit explicit pointers and have automatic memory management. There are differences, however. Java's objects and inheritance subtyping have no counterpart in ML. Java lacks parametric polymorphism and support for closures.

The goal of MLj's extension is to allow ML to call Java methods and handle Java objects as first-class entities, i.e. store them and use them as arguments and/or return values from ML functions. Furthermore, they allow

Java classes to be constructed within MLj, and these classes may use ML objects freely. The complete formal semantics are given in [16]. Here, we highlight some of the interesting points.

MLj extends SML with Java types and terms. Primitive types in Java are equated with their respective ML types. So, in MLj, a ML `int` and a Java `int` refer to the same type. Some basic classes, like ML's `string` and Java's `java.lang.String` are also equated. Java's package hierarchy is equated with ML's modules, and Java's `import` syntax is mapped to ML's module `open`. Conveniently, Java's package syntax and ML's module syntax are identical, and so they work without modification.

Unlike in ML, which requires that values be bound when they are declared, Java allows reference types (i.e. objects) to be null. Therefore, MLj wraps Java object types within an option[3] type. In this scheme, a null reference is given the value `None`, while valid values are constructed with `Just`. This approach is similar to how the Haskell 98 FFI and GreenCard represent null pointer return values in C.

MLj allows certain implicit *coercions* to make the code more natural. Here, a coercion is an implicit type cast and conversion from one datatype to another. In particular, MLj allows a widening coercion on Java objects that implements Java inheritance rules. Specifically, for Java object types `T1` and `T2`, if `T1 <: T2`[4], then whenever `T1` `T2` is expected, it will be coerced to `T2`. This coercion is always well-defined for Java objects. Explicit casts, notably downcasts (i.e. a cast from `T2` to `T1`, where `T1 <: T2`), are supported for Java objects. As in Java, an invalid cast will throw an exception.

[3]ML's option type is a variant with two constructors: `None` or `Just` α.
[4]The notation `<:` is read "is a subtype of."

MLj also permits coercion from an option type `option` α to α, which simplifies programming with the null pointer mapping described above. This coercion will throw a Java `NullPointerException` if the value is `None`.

Java methods may be "overloaded," i.e. share the same identifier within a single class, and distinguished by their argument types. At an overloaded method call site, the specific method is determined by examining the supplied argument types. Overloading can be ambiguous, however. Consider the following Java class `A`, with overloaded method `foo`:

```
class A {
  void foo(String s) {System.out.println("1");}
  void foo(Object o) {System.out.println("2");}
  public static void main(String [] argv) {
    A a = new A();
    a.foo("hello");
  }
}
```

The call to `foo` in `main` is ambiguous; the value `"hello"` can be typed as either a `String` and an `Object`, because `String <: Object`. Java handles this ambiguity by choosing the most specific method with respect to an ordering on types (roughly, the ordering is determined by the inheritance hierarchy, with extra rules for primitives). So, in Java, the example above will output `1`. Note that a *unique* most specific method may not exist, and in this case the Java compiler will terminate with an error.

Unfortunately, ML's type inference algorithm cannot accommodate the "most specific" technique of selecting overloaded methods, and so MLj dis-

cards it. The specific overloaded method must be unambiguously selected by the programmer, using ML's type annotation syntax if necessary to disambiguate argument types.

MLj's compiler produces Java bytecode, which is executed within a single Java virtual machine (JVM). Therefore, MLj's ML and Java portions share a garbage collection system, and so the special rules for pinning arrays and so on that we saw in Section 3 are not needed. By the same token, in MLj Java and ML share a call stack, and since ML's `exn` type and Java's `java.lang.Exception` type are equated, exceptions may flow freely up the stack from one language to the other.

7.2 Discussion

Integrated languages are a bit of a catch-all category for systems that achieve interoperability by combining two or more languages. The approach is clearly not scalable, because considerable effort is required even to define the syntax and semantics of a language that combines two other languages, let alone implement a compiler. However, the result can accommodate a very natural programming style. It is especially interesting that the examples in this section are able to go beyond function calls as the main abstraction for interaction. They focus, instead, on expressions. MLj is even able to maintain the type safety guarantees of both ML and Java. Whether or not a given integrated language system is efficient depends on the system in question. Jeannie, for example, generates JNI code, and so is no more efficient than the JNI itself.

8 Conclusion

While there has been much work on multi-language programming models and systems, there is still room for improvement.

IDLs and FFIs remain the most popular approaches to multi-language programming. FFIs in particular are ubiquitous, but often difficult to use. IDL-based approaches are also popular, but the requirement that native types be mapped into the restricted subset of interoperable types supported by the IDL is a serious limitation. Specifically, it limits the ability of IDL systems to present a natural programming model. Type-mapping may help to alleviate this issue, allowing programmers to express interfaces in terms of user-defined high-level types. However, the marshalling procedures required by type-maps incurs a cost in terms of performance. This problem may be aggravated by the relatively seamless interoperability that type maps afford – if programmers are not aware that what appears to be a normal function call will actually marshal a large data structure, they might inadvertently cripple performance.

This problem suggests a need for type-map optimization. It may be possible to optimize the type-maps at runtime. Consider a component written in C that calls another C component. Existing component systems will marshall arguments in this case, but this is clearly unnecessary if the two C programs share a common representation for each argument. If the system could recognize that this particular pair of type-maps, which convert C to the IDL, and then the IDL back to C, is redundant, the intermediate step and concomitant marshalling could be eliminated. A generalized version of this idea, where an automated system may reason about and attempt to optimize

type-maps depending on how they are constructed, may be the basis for a promising line of future research.

Another investigation might examine the language integrations for domain-specific languages (DSLs). DSLs are often convenient, expressive, and highly optimized for a particular task, but ill-suited to general-purpose programming. Perhaps DSLs could be integrated within general-purpose languages, using techniques similar to the examples seen in Section 7. The DSL could then be called upon when needed, and robustly supported by tools. For example, an inline DSL expression could be type-checked, taking into account the surrounding program context. Integrated DSLs might represent a novel and powerful use of multi-language programming.

A Term rewriting

Term rewriting is a formal technique for reducing *terms* to some normal form with respect to a set of rules [13]. Our interest in term rewriting stems from its utility in program transformation [63].

A.1 Overview

Terms are built from *variables* and *constructors* [13]. Variables are placeholders for other terms. Constructors are functions from terms to a single term, with some unique constant symbol and arity $n \geq 0$. We adopt the convention that constructors symbols begin with uppercase letters (e.g. `Cons`), while variables are usually represented by a single lowercase letter (e.g. x). If a constructor has zero arity, we refer to it as a *constant symbol*. For example, x and y are both terms, and if `Cons` has arity 2, then $\texttt{Cons}(x, y)$ is also a term. A *signature* is a set of function symbols with associated arities. Notably, the terms of a properly defined signature can be used to represent the abstract syntax tree of a program.

Rewrite rules have the form $t_1 \rightarrow t_2$, and define a transformation of a term matching the *pattern* t_1 to a term t_2. If the pattern t_1 contains variable terms, these will be bound as a result of a successful pattern match. These bindings may then be used in the construction of t_2.

Given a term t and a set of rewrite rules R, we say that t contains a *redex* s if s is a subterm of t and if s matches one or more rewrite rules in R. A term is in *normal form* with respect to R if it contains no redices.

The task of most term rewriting systems is the reduction of terms to a

normal form with respect some rules R by applying rewrite rules to subterms until no more rules may be applied. In general, a term may contain many redices; one challenge for term rewriting implementations is to devise a *strategy* for deciding when and where to apply each rule within a term [63].

A.2 Properties

If repeated application of a set of rewrite rules is guaranteed to eventually reduce to a normal form, then the rules are said to *terminate*. For example, consider the rule set containing the single rule $\texttt{F(x)} \rightarrow \texttt{F(F(x))}$. It is easy to see that this rule will never terminate, because every application grows the term and the redex still applies. Unsurprisingly, given a set of rules, termination is undecidable in general [13].

If a set of rules is terminating, and repeated application of a set of rules will always produce a *unique* normal form, then the rules are said to be *confluent*. Like termination, confluence is undecidable in general. However, for rules which are known to terminate, confluence is decidable [13].

A.3 System S

System S is a language that provides the basic machinery for term rewriting [64]. It can be used to implement different term rewriting strategies, that is, methods for deciding which redices to reduce, and when to reduce them relative to other redices.

Terms in System S are defined formally, but the rules for constructors, variables, and pattern matching are very similar to the canonical definitions given above, so we omit the details here.

In System S, a rule s is a binary relation $\xrightarrow{s} \subseteq \mathcal{S} \times (\mathcal{S} \cup \bot)$, where $\mathcal{S}$ is a set of terms and $\bot$ is a special term indicating failure. Specifically, we say that $t \xrightarrow{s} t'$ succeeds if and only if $(t, t') \in \xrightarrow{s}$, and $t' \neq \bot$. If $(t, t') \in \xrightarrow{s}$, but $t' = \bot$, we say it fails. If $(t, t') \notin \xrightarrow{s}$ then $t \xrightarrow{s} t'$ is undefined in s.

Atomic rules are just primitive defined rules $t \to t'$, and rules may be defined to fail. System S introduces a set of compositions on rules that allow new rules to be created from existing ones. The complete semantics for these compositions are given in [64], but they include things like as a sequencing operator ;, where for rules $s1$ and $s2$, $s1; s2$ applies $s1$ first and, if it succeeds, applies $s2$ to the result (and fails otherwise). Other combinators include deterministic and non-deterministic choice, a fixed-point operator that allows recursive composition, as well as a set of operators to traverse subterms by enumerating them.

References

[1] COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx.

[2] The DICOM standard. http://medical.nema.org/.

[3] MEX-files Guide. http://www.mathworks.com/support/tech-notes/1600/1605.html.

[4] Portable Network Graphics (PNG) Specification, Second Edition. http://www.w3.org/TR/PNG/.

[5] Python v2.6.4 documentation: ctypes – A foreign function library for Python. http://docs.python.org/library/ctypes.html.

[6] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[7] Swig documentation: Typemaps. http://www.swig.org/Doc1.3/Typemaps.html.

[8] WHIRL intermediate language specification. http://www.open64.net/documentation/.

[9] XML-RPC Specification. http://www.xmlrpc.com/spec.

[10] Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma-international.org/publications/standards/Ecma-335.htm, June 2006.

[11] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[12] Robert C. Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois C. McInnes, Steve R. Parker, and Brent A. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 13–23, 1999.

[13] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[14] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *In ACM SIGSOFT'96, Fourth Symposium on the Foundations of Software Engineering*, pages 147–155, 1996.

[15] David M. Beazley. SWIG: an easy to use tool for integrating scripting languages with C and C++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, July 1996.

[16] Nick Benton and Andrew Kennedy. Interlanguage working without tears: blending SML with Java. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 126–137, New York, NY, USA, 1999. ACM.

[17] Paul V. Biron and Ashok Malhotra. XML schema part 2: Datatypes second edition. W3C recommendation, World Wide Web Consortium, October 2004.

[18] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[19] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C natively. In *Electronic Notes in Theoretical Computer Science*, volume 59, 2001.

[20] Manuel Chakravarty, Sigbjorn Finne, Fergus Henderson, Marcin Kowalczyk, Daan Leijen, Simon Marlow, Erik Meijer, Sven Panne, Simon Peyton Jones, Alastair Reid, Malcolm Wallace, and Michael Weber. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2003.

[21] George Coulouris, Jean Dolllimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*, chapter Sun RPC, pages 138–144. International Computer Science. Addison Wesley, 2 edition, 1994.

[22] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, March/April 2002.

[23] Linda DeMichiel and Michael Keith. JSR-000220 Enterprise JavaBeans 3.0 Final Release Specification. http://java.sun.com/products/ejb/docs.html, May 2006.

[24] Kathleen Fisher, Riccardo Pucella, and John Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1):3–19, September 2001.

[25] M. Folk and B.R. Barkstrom. Attributes of file formats for long-term preservation of scientific and engineering data in digital libraries, 2003.

[26] Tom Goodale. *Workflows for e-Science*, chapter Expressing Workflow in the Cactus Framework, pages 416–427. Springer London, 2007.

[27] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Masso, Edward Seidel, and John Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall PTR, second edition, June 2000.

[29] Mark Grechanik, Don Batory, and Dewayne E. Perry. Design of large-scale polylingual systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 357–366, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Jennifer Hamilton. Language integration in the Common Language Runtime. *ACM SIGPLAN Notices*, 38(2):19–28, 2003.

[31] Per Brinch Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[32] R. Hayes and R.D. Schlichting. Facilitating mixed language programming in distributed systems. *IEEE Transactions on Software Engineering*, 13:1254–1264, 1987.

[33] Martin Hirzel and Robert Grimm. Jeannie: granting Java native interface developers their wishes. In *Proceedings of the 2007 OOPSLA conference*, volume 42, pages 19–38, New York, NY, USA, 2007. ACM.

[34] Jon Hopkins. Component primer. *Communications of the ACM*, 43(10):27–30, October 2000.

[35] Lorenz Huelsbergen. A portable C interface for Standard ML of New Jersey. Technical report, AT&T Bell Laboratories, January 1996.

[36] Geoffrey C. Hulette, Matthew J. Sottile, Robert Armstrong, and Benjamin Allan. Onramp: enabling a new component-based development paradigm. In *CBHPC '09: Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, pages 1–10, New York, NY, USA, 2009. ACM.

[37] Bill Janssen and Mike Spreitzer. ILU: Inter-language unification via object modules. In *Workshop on Multi-language Object Models*, Portland, OR, August 1994.

[38] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green card: a foreign language interface for Haskell. In *ACM SIGPLAN Haskell Workshop (in conjuction with ICFP97)*, February 1997.

[39] A. Kaplan, J. Ridgway, and J.C. Wileden. Why IDLs are not ideal. *Software Specification and Design, International Workshop on*, 0:2, 1998.

[40] Guy K. Kloss. Automatic C library wrapping – Ctypes from the trenches. *The Python Papers*, 3(3), 2008.

[41] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processes*, Portsmouth, VA, March 2001.

[42] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[43] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 2002.

[44] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. OpenUH: an optimizing, portable OpenMP

compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2006.

[45] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Prentice Hall, 2nd edition, April 1999.

[46] Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende, Forschungszentrum Juelich, Reid Rivenburgh, Craig Rasmussen, and Bernd Mohr. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[47] Stavros Macrakis. From Uncol to ANDF: Progress in standard intermediate languages. White paper, Open Software Foundation, 1993.

[48] Mike Murphy. NVIDIA's experience with Open64. *Open64 Workshop at CGO '08*, 2008.

[49] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys (CSUR)*, 27(2):262–264, 1995.

[50] David Notkin, Andrew P. Black, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan. Interconnecting heterogeneous computer systems. *Commun. ACM*, 31(3):258–273, 1988.

[51] Debu Panda, Reza Rahman, and Derek Lane. *EJB 3 in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[52] C. E. Rasmussen, K. A. Lindlan, B. Mohr, J. Striegnitz, and Forschungszentrum Jlich. CHASM: Static Analysis and Automatic Code Generation for Improved Fortran 90 and C++ Interoperability. In *In Proceedings of the Los Alamos Computer Science Symposium 2001 (LACSI'01*, 2001.

[53] John Reid. The new features of Fortran 2000. *SIGPLAN Fortran Forum*, 21(2):1–31, 2002.

[54] John Reid. The new features of Fortran 2003. *SIGPLAN Fortran Forum*, 26(1):10–33, 2007.

[55] John Reppy and Chunyan Song. Application-specific foreign-interface generation. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 49–58, New York, NY, USA, 2006. ACM.

[56] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 2008.

[57] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, Palo Alto, CA, April 2007.

[58] Brent A. Smolinski, Scott Kohn, Noah Elliott, and Nathan Dykman. *Computing in Object-Oriented Parallel Environments*, volume 1732, chapter Language Interoperability for High-Performance Parallel Scientific Components, pages 61–71. Springer Berlin / Heidelberg, 1999.

[59] Alfred Z. Spector. Performing remote operations efficiently on a local computer network. *Commun. ACM*, 25(4):246–260, 1982.

[60] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832 (Draft Standard), 1995. Obsoleted by RFC 4506.

[61] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, 1990.

[62] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[63] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57:109 – 143, 2001. WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming.

[64] Eelco Visser and Zine el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422 – 441, 1998. International Workshop on Rewriting Logic and its Applications.

[65] J. Waldo. Remote procedure calls and Java remote method invocation. *Concurrency, IEEE*, 6(3):5–7, Jul-Sep 1998.

[66] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: an infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Not.*, 29(12):31–37, 1994.

[67] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a key to reuse. In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 182–190, New York, NY, USA, 1993. ACM.