acmqueue

The Challenge of Cross-language Interoperability

Interfacing between languages is increasingly important

David Chisnall

Interoperability between languages has been a problem since the second programming language was invented. Solutions have ranged from language-independent object models such as COM (Component Object Model) and CORBA (Common Object Request Broker Architecture) to VMs (virtual machines) designed to integrate languages, such as the JVM (Java Virtual Machine) and CLR (Common Language Runtime). With software becoming ever more complex and hardware less homogeneous, the likelihood of a single language being the correct tool for an entire program is lower than ever. As modern compilers become more modular, there is potential for a new generation of interesting solutions.

In 1961 the British company Stantec released a computer called the ZEBRA, which was interesting for a number of reasons, not least of which was its data flow-based instruction set. The ZEBRA was quite difficult to program with the full form of its native instruction set, so it also included a more conventional version, called Simple Code. This form came with some restrictions, including a limit of 150 instructions per program. The manual helpfully informs users that this is not a severe limitation, as it is impossible that someone would write a working program so complex that it would need more than 150 instructions.

Today, this claim seems ludicrous. Even simple functions in a relatively low-level language such as C have more than 150 instructions once they are compiled, and most programs are far more than a single function. The shift from writing assembly code to writing in a higher-level language dramatically increased the complexity of programs that were possible, as did various software engineering practices.

The trend toward increased complexity in software shows no sign of abating, and modern hardware creates new challenges. Programmers in the late 1990s had to target PCs at the low end that had an abstract model a lot like a fast PDP-11. At the high end, they would have encountered an abstract model like a very fast PDP-11, possibly with two to four (identical) processors. Now, mobile phones are starting to appear with eight cores with the same ISA (instruction set architecture) but different speeds, some other streaming processors optimized for different workloads (DSPs, GPUs), and other specialized cores.

The traditional division between high-level languages representing the class that is similar to a human's understanding of the problem domain and low-level languages representing the class similar to the hardware no longer applies. No low-level language has semantics that are close to a programmable data-flow processor, an x86 CPU, a massively multithreaded GPU, and a VLIW (very long instruction word) DSP (digital signal processor). Programmers wanting to get the last bit of performance out of the available hardware no longer have a single language they can use for all probable targets.

Similarly, at the other end of the abstraction spectrum, domain-specific languages are growing

more prevalent. High-level languages typically trade generality for the ability to represent a subset of algorithms efficiently. More general-purpose high-level languages such as Java sacrifice the ability to manipulate pointers directly in exchange for providing the programmer with a more abstract memory model. Specialized languages such as SQL make certain categories of algorithms impossible to implement but make common tasks within their domain possible to express in a few lines.

You can no longer expect a nontrivial application to be written in a single language. High-level languages typically call code written in lower-level languages as part of their standard libraries (for example, GUI rendering), but adding calls can be difficult.

In particular, interfaces between two languages that are not C are often difficult to construct. Even relatively simple examples, such as bridging between C++ and Java, are not typically handled automatically and require a C interface. The Kaffe Native Interface[4] did provide a mechanism for doing this, but it was not widely adopted and had limitations.

The problem of interfacing between languages is going to become increasingly important to compiler writers over the coming years. It presents a number of challenges, detailed here.

OBJECT MODEL DIFFERENCES

Object-oriented languages bind some notion of code and data together. Alan Kay, who helped develop object-oriented programming while at Xerox PARC, described objects as "simple computers that communicate via message passing." This definition leaves a lot of leeway for different languages to fill in the details:

- Should there be factory objects (classes) as first-class constructs in the language?
- If there are classes, are they also objects?
- Should there be zero (e.g., Go), one (e.g., Smalltalk, Java, JavaScript, Objective-C), or many (e.g., C++, Self, Simula) superclasses or prototypes for an object?
- Is method lookup tied to the static type system (if there is one)?
- Is the data contained within an object of static or dynamic layout?
- Is it possible to modify method lookup at runtime?

The question of multiple inheritance is one of the most common areas of focus. Single inheritance is convenient, because it simplifies many aspects of the implementation. Objects can be extended just by appending fields; a cast to the supertype just involves ignoring the end, and a cast to a subtype just involves a check—the pointer values remain the same. Downcasting in C++ requires a complex search of the inheritance graph in the run-time type information via a runtime library function.

In isolation, both types of inheritance are possible to implement, but what happens if you want, for example, to expose a C++ object into Java? You could perhaps follow the .NET or Kaffe approach, and support direct interoperability with only a subset of C++ (Managed C++ or C++/CLI) that supports single inheritance only for classes that will be exposed on the Java side of the barrier.

This is a good solution in general: define a subset of one language that maps cleanly to the other but can understand the full power of the other. This is the approach taken in Pragmatic Smalltalk:[5] allow Objective-C++ objects (which can have C++ objects as instance variables and invoke their methods) to be exposed directly as if they were Smalltalk objects, sharing the same underlying representation.

This approach still provides a cognitive barrier, however. If you want to use a C++ framework

directly, such as LLVM from Pragmatic Smalltalk or .NET, then you will need to write single-inheritance classes that encapsulate the multiple-inheritance classes that the library uses for most of its core types.

Another possible approach would be to avoid exposing any fields within the objects and just expose each C++ class as an interface. This would, however, make it impossible to inherit from the bridged classes without special compiler support to understand that some interfaces came along with implementation.

Although complex, this is a simpler system than interfacing between languages that differ on what method lookup means. For example, Java and Smalltalk have almost identical object and memory models, but Java ties the notion of method dispatch to the class hierarchy, whereas in Smalltalk two objects can be used interchangeably if they implement methods with the same names.

This is a problem encountered by RedLine Smalltalk,[1] which compiles Smalltalk to run on JVM. Its mechanism for implementing Smalltalk method dispatch involves generating a Java interface for each method and then performing a cast of the receiver to the relevant interface type before dispatch. Sending messages to Java classes requires extra information, because existing Java classes don't implement this; thus, RedLine Smalltalk must fall back to using Java's Reflection APIs.

The method lookup for Smalltalk (and Objective-C) is more complex, because there are a number of second-chance dispatch mechanisms that are either missing or limited in other languages. When compiling Objective-C to JavaScript, rather than using the JavaScript method invocation, you must wrap each Objective-C message send in a small function that first checks if the method actually exists and, if it doesn't, calls some lookup code.

This is relatively simple in JavaScript because it handles variadic functions in a convenient way: if a function or method is called with more arguments than it expects, then it receives the remainder as an array that it can expect. Go does something similar. C-like languages just put them on the stack and expect the programmer to do the write with no error checking.

MEMORY MODELS

The obvious dichotomy in memory models is between automatic and manual deallocation. A slightly more important concern is the difference between deterministic and nondeterministic destruction.

It is possible to run C with the Boehm-Demers-Weiser garbage collector[3] without problems in many cases (unless you run out of memory and have a lot of integers that look like pointers). It is much harder to do the same for C++, because object deallocation is an observable event. Consider the following code:

```
{
      LockHolder l( mutex );
      /* Do stuff that requires mutex to be locked */
}
```

The `LockHolder` class defines a very simple object; a mutex passes into the object, which then locks the mutex in its constructor and unlocks it in the destructor. Now, imagine running this same code in a fully garbage-collected environment—the time at which the destructor runs is not defined.

This example is relatively simple to get right. A garbage-collected C++ implementation is required

to run the destructor at this point but not to deallocate the object. This idiom is not available in languages that were designed to support garbage collection from the start. The fundamental problem with mixing them is not determining who is responsible for releasing memory; rather, it is that code written for one model expects deterministic operation, whereas code written for the other does not.

There are two trivial approaches to implementing garbage collection for C++: the first is to make the `delete` operator invoke destructors but not reclaim the underlying storage; the other is to make `delete` a no-op and call destructors when the object is detected as unreachable.

Destructors that call only `delete` are the same in both cases: they are effectively no-ops. Destructors that release other resources are different. In the first case, they run deterministically but will fail to run if the programmer does not explicitly delete the relevant object. In the second case, they are guaranteed to run eventually but not necessarily by the time the underlying resource is exhausted.

Additionally, a fairly common idiom in many languages is a self-owned object that waits for some event or performs a long-running task and then fires a callback. The receiver of the callback is then responsible for cleaning up the notifier. While it's live, it is disconnected from the rest of the object graph and so appears to be garbage. The collector must be explicitly told that it is not. This is the opposite of the pattern in languages without automatic garbage collection, where objects are assumed to be live unless the system is told otherwise. (Hans Boehm discussed some of these issues in more detail in a 1996 paper.[2])

All of these problems were present with Apple's ill-fated (and, thankfully, no longer supported) attempt to add garbage collection to Objective-C. A lot of Objective-C code relies on running code in the `-dealloc` method. Another issue was closely related to the problem of interoperability. The implementation supported both traced and untraced memory but did not expose this information in the type system. Consider the following snippet:

```
void allocateSomeObjects (id * buffer, int count)
{
       for (int i=0 ; i<count ; i++)
       {
              buffer [i] = [SomeClass new];
       }
}
```

In garbage-collected mode, it is impossible to tell if this code is correct. Whether it is correct or not depends on the caller. If the caller passes a buffer allocated with `NSAllocateCollectable(),` with `NSScannedOption` as the second parameter, or with a buffer allocated on the stack or in a global in a compilation unit compiled with garbage-collection support, then the objects will last (at least) as long as the buffer. If the caller passes a buffer that was allocated with `malloc()` or as a global in a C or C++ compilation unit, then the objects will (potentially) be deallocated before the buffer. The *potentially* in this sentence makes this a bigger problem: because it's nondeterministic, it is hard to debug.

The ARC (Automatic Reference Counting) extensions to Objective-C do not provide complete garbage collection (they still allow garbage cycles to leak), but they do extend the type system to

define the ownership type for such buffers. Copying object pointers to C requires the insertion of an explicit cast containing an ownership transfer.

Reference counting also solves the determinism problem for acyclic data. In addition, it provides an interesting way of interoperating with manual memory management: by making `free()` decrement the reference count. Cyclic (or potentially cyclic) data structures require the addition of a cycle detector. David F. Bacon's team at IBM has produced a number of designs for cycle detectors[8] that allow reference counting to be a full garbage-collection mechanism, as long as pointers can be accurately identified.

Unfortunately, cycle detection involves walking the entire object graph from a potentially cyclic object. Some simple steps can be taken to lessen this cost. The obvious one is to defer it. An object is only potentially part of a cycle if its reference count is decremented but not deallocated. If it is later incremented, then it is not part of a garbage cycle (it may still be part of a cycle, but you don't care yet). If it is later deallocated, then it is acyclic.

The longer you defer cycle detection, the more nondeterminism you get, but the less work the cycle detector has to do.

EXCEPTIONS AND UNWINDING

These days, most people think of exceptions in the sense popularized by C++: something that is roughly equivalent to `setjmp()` and `longjmp()` in C, although possibly with a different mechanism.

A number of other mechanisms for exceptions have been proposed. In Smalltalk-80, exceptions are implemented entirely in the library. The only primitive that the language provides is that when you explicitly return from a closure, you return from the scope in which the closure was declared. If you pass a closure down the stack, then a return will implicitly unwind the stack.

When a Smalltalk exception occurs, it invokes a handler block on the top of the stack. This may then return, forcing the stack to unwind, or it may do some cleanup. The stack itself is a list of activation records (which are objects) and therefore may do something more complex. Common Lisp provides a rich set of exceptions too, including those that support resuming or restarting immediately afterward.

Exception interoperability is difficult even within languages with similar exception models. For example, C++ and Objective-C both have similar notions of an exception, but what should a C++ catch block that expects to catch a `void*` do when it encounters an Objective-C object pointer? In the GNUstep Objective-C runtime[6], we chose not to catch it after deciding not to emulate Apple's behavior of a segmentation fault. Recent versions of OS X have adopted this behavior, but the decision is somewhat arbitrary.

Even if you do catch the object pointer from C++, that doesn't mean that you can do anything with it. By the time it's caught, you've lost all of the type information and have no way of determining that it is an Objective-C object.

Subtler issues creep in when you start to think about performance. Early versions of VMKit[7] (which implements Java and CLR VMs on top of LLVM) used the zero-cost exception model designed for C++. This is *zero cost* because entering a try block costs nothing. When throwing an exception, however, you must parse some tables that describe how to unwind the stack, then call into a personality function for each stack frame to decide whether (and where) the exception should be caught.

This mechanism works very well for C++, where exceptions are rare, but Java uses exceptions to report lots of fairly common error conditions. In benchmarks, the performance of the unwinder was a limiting factor. To avoid this, the calling convention was modified for methods that were likely to throw an exception. These functions returned the exception as a second return value (typically in a different register), and every call just had to check that this register contained 0 or jump to the exception handling block if it did not.

This is fine when you control the code generator for every caller, but this is not the case in a cross-language scenario. You might address the issue by adding another calling convention to C that mirrors this behavior or that provides something like the multiple-return-values mechanism commonly used in Go for returning error conditions, but that would require every C caller to be aware of the foreign language semantics.

MUTABILITY AND SIDE EFFECTS

When you start to include functional languages in the set with which you wish to interoperate, the notion of mutability becomes important. A language such as Haskell has no mutable types. Modifying a data structure in place is something that the compiler may do as an optimization, but it's not something exposed in the language.

This is a problem encountered by F#, which is sold as a dialect of OCaml and can integrate with other .NET languages, use classes written in C#, and so on. C# already has a notion of mutable and immutable types. This is a very powerful abstraction, but an immutable class is simply one that doesn't expose any fields that are not read only, and a read-only field may contain references to objects that (via an arbitrary chain of references) refer to mutable objects whose state may be changed out from under the functional code. In other languages, such as C++ or Objective-C, mutability is typically implemented within the class system by defining some classes that are immutable, but there is no language support and no easy way of determining whether an object is mutable.

C and C++ have a very different concept of mutability in the type system provided by the language: a particular reference to an object may or may not modify it, but this doesn't mean that the object itself won't change. This, combined with the deep copying problem, makes interfacing functional and object-oriented languages a difficult problem.

Monads provide some tantalizing possibilities for the interface. A monad is an ordered sequence of computational steps. In an object-oriented world, this is a series of message sends or method invocations. Methods that have the C++ notion of const (i.e., do not modify the state of the object) may be invoked outside of the monad, and so are amenable to speculative execution and backtracking, whereas other methods should be invoked in a strict sequence defined by the monad.

MODELS OF PARALLELISM

Mutability and parallelism are closely related. The cardinal rule for writing maintainable, scalable, parallel code is that no object may be both mutable and aliased. This is trivial to enforce in a purely functional language: no object is mutable at all. Erlang makes one concession to mutability, in the form of a process dictionary—a mutable dictionary that is accessible only from the current Erlang process and so can never be shared.

Interfacing languages with different notions of what can be shared presents some unique

problems. This is interesting for languages intended to target massively parallel systems or GPUs, where the model for the language is intimately tied to the underlying hardware.

This is the issue encountered when trying to extract portions of C/C++/Fortran programs to turn into OpenCL. The source languages typically have in-place modification as the fastest way of implementing an algorithm, whereas OpenCL encourages a model where a source buffer is processed to generate an output buffer. This is important because each kernel runs in parallel on many inputs; thus, for maximum throughput they should be independent.

In C++, however, ensuring that two pointers do not alias is nontrivial. The `restrict` keyword exists to allow programmers to provide this annotation, but it's impossible in the general case for a compiler to check that it is correctly used.

Efficient interoperability is very important for heterogeneous multicore systems. On a traditional single-core or SMP (symmetric multiprocessing) computer, there is a one-dimensional spectrum between high-level languages that are close to the problem domain and low-level languages that are close to the architecture. On a heterogeneous system, no one language is close to the underlying architecture, as the difficulty of running arbitrary C/C++ and Fortran code on GPUs has shown.

Current interfaces—for example, OpenCL—are a long way from ideal. The programmer must write C code to manage the creation of a device context and the movement of data to and from the device, and then write the kernel in OpenCL C. The ability to express the part that runs on the device in another language is useful, but when most of the code for simple operations is related to the boundary between the two processing elements rather than the work done on either side, then something is wrong.

How to expose multiple processing units with very different abstract machine models to the programmer is an interesting research problem. It is very difficult to provide a single language that efficiently captures the semantics. Thus, this problem becomes one of interoperability between specialized languages. This is an interesting shift in that domain-specific languages, which are traditionally at the high-level end of the spectrum, now have an increasing role to play as low-level languages.

THE VM DELUSION

The virtual machine is often touted as a way of addressing the language interoperability problem. When Java was introduced, one of the promises was that you would soon be able to compile all of your legacy C or C++ code and run it in JVM alongside Java, providing a clean migration path. Today, Ohloh.net (which tracks the number of lines of code available in public open source repositories) reports 4 billion lines of C code, and around 1.5 billion each of C++ and Java. While other languages such as Scala (almost 6 million lines of code tracked by Ohloh.net) run in JVM, legacy low-level languages do not.

Worse, calling native code from Java is so cumbersome (in terms of both cognitive and runtime overhead) that developers end up writing applications in C++ rather than face calling into a C++ library from Java. Microsoft's CLR did a little better, allowing code written in a subset of C++ to run; it makes calling out to native libraries easier but still provides a wall.

This approach has been a disaster for languages such as Smalltalk that don't have large companies backing them. The Smalltalk VM provides some advantages that neither CLR nor JVM provides in the form of a persistence model and reflective development environment, but it also forms a very

large PLIB (programming language interoperability barrier) by dividing the world into things that are inside and things that are outside the box.

This gets even more complex once you have two or more VMs and now have the problem of source-language interoperability and the (very similar) problem of interoperability between the two VMs, which are typically very low-level programming languages.

THE PATH FORWARD

Many years ago the big interoperability question of the day was C and Pascal—two languages with an almost identical abstract machine model. The problem was that Pascal compilers pushed their parameters onto the stack left to right (because that required fewer temporaries), whereas C compilers pushed them right to left (to ensure that the first ones were at the top of the stack for variadic functions).

This interoperability problem was largely solved by the simple expedient of defining calling conventions as part of the platform ABI (application binary interface). No virtual machine or intermediate target was required, nor was any source-to-source translation. The equivalent of the virtual machine is defined by the ABI and the target machine's ISA.

Objective-C provides another useful case study. Methods in Objective-C use the C calling convention, with two hidden parameters (the object and the selector, which is an abstract form of the method name) passed first. All parts of the language that don't trivially map to the target ABI or ISA are factored out into library calls. A method invocation is implemented as a call to the `objc_msgSend()` function, which is implemented as a short assembly routine. All of the introspection works via the mechanism of calls to the runtime library.

We've used GNUstep's Objective-C runtime to implement front ends for dialects of Smalltalk and JavaScript in LanguageKit. This uses LLVM, but only because having a low-level intermediate representation permits optimizations to be reused between compilers: the interoperability happens in the native code. This runtime also supports the blocks ABI defined by Apple; therefore, closures can be passed between Smalltalk and C code.

Boehm GC (garbage collector) and Apple AutoZone both aimed to provide garbage collection in a library form, with different requirements. Can concurrent compacting collectors be exposed as libraries, with objects individually marked as nonmovable when they are passed out to low-level code? Is it possible to enforce mutability and concurrency guarantees in an ABI or library? These are open problems, and the availability of mature libraries for compiler design makes them interesting research questions.

Perhaps more interesting is the question of how many of these can be sunk down into the hardware. In CTSRD (Crash-worthy Trustworthy Systems R&D), a joint project between SRI International and the University of Cambridge Computer Laboratory, researchers have been experimenting with putting fine-grained memory protection into the hardware, which they hope will provide more efficient ways of expressing certain language memory models. This is a start, but there is a lot more potential for providing richer feature sets for high-level languages in silicon, something that was avoided in the 1980s because transistors were scarce and expensive resources. Now transistors are plentiful but power is scarce, so the tradeoffs in CPU design are very different.

The industry has spent the past 30 years building CPUs optimized for running languages such as C, because people who needed fast code used C (because people who designed processors optimized

them for C, because...). Maybe the time has come to start exploring better built-in support for common operations in other languages. The RISC project was born from looking at the instructions that a primitive compiler generated from compiling C code. What would we end up with if we started by looking at what a native JavaScript or Haskell compiler would emit?

REFERENCES

1. Allen, S. 2011. RedLine Smalltalk. Presented at the International Smalltalk Conference.
2. Boehm, H.-J. 1996. Simple garbage-collector-safety. *ACM SIGPLAN Notices* 31(5):89-98.
3. Boehm, H.-J., Weiser, M. 1988. Garbage collection in an uncooperative environment. *Software Practice and Experience* 18(9): 807-820.
4. Bothner, P., Tromey, T. 2001. Java/C++ integration; http://per.bothner.com/papers/UsenixJVM01/CNI01.pdf. http://gcc.gnu.org/java/papers/native++.html
5. Chisnall, D. 2012. Smalltalk in a C world. In *Proceedings of the International Workshop on Smalltalk Technologies:* 4:1-4:12.
6. Chisnall, D. 2012. A New Objective-C Runtime: from Research to Production. *ACM Queue*. http://queue.acm.org/detail.cfm?id=2331170
7. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B. 2010. VMKit: a substrate for managed runtime environments. *ACM SIGPLAN Notices* 45(7): 51-62.
8. Paz, H., Bacon, D. F., Kolodner, E. K., Petrank, E., Rajan, V. T. 2005. An efficient on-the-fly cycle collection. In *Proceedings of the 14th International Conference on Compiler Construction*: 156-171. Berlin, Heidelberg: Springer-Verlag.

Portions of this work were sponsored by DARPA (Defense Advanced Research Projects Agency) and AFRL (Air Force Research Laboratory), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DAVID CHISNALL is a researcher at the University of Cambridge, where he works on programming language design and implementation. He spent several years consulting in between finishing his Ph.D. and arriving at Cambridge, during which time he also wrote books on Xen and the Objective-C and Go programming languages, as well as numerous articles. He also contributes to the LLVM, Clang, FreeBSD, GNUstep, and Étoilé open source projects, and he dances the Argentine tango.