Transparent use of C++ Classes in Java Environments

Wade Holst
Ben Stephenson
The University of Western Ontario

Abstract

A mechanism is presented that provides for transparent use of C++ classes within a Java program, including within inheritance relationships. This in turn provides a means of selectively using C++ for components best suited to C++, allowing C++ legacy code to be used without re-implementation within Java programs, improving execution performance of a Java program and/or reducing memory requirements of a Java program.

The transparency is accomplished by the use of a translator called CJ that parses C++ header files and creates proxy Java classes for each C++ class that is to be provided to Java programmers. The proxy Java class contains an instance of the underlying C++ class and delegates method invocations to this underlying C++ class via the Java Native Interface.

The mapping of C++ types to Java types is addressed in detail. Java types are chosen based on their intuitiveness and the efficiency with which they can be converted to C++ types. Care is taken to provide Java programmers with the greatest possible control over primitive C++ data types from within Java. CJ addresses both single and multiple inheritance, cross-language inheritance, automatic memory management, operator overloading, default arguments, templates, special support for common C++ and Java types, pointers, references, arrays, const, and unsigned. CJ emphasizes correctness, efficiency, configurability, convenience and generality.

1 Introduction

Java and C++ are two of the most widely used object oriented programming languages. Both languages have advantages and disadvantages. C++ generates programs that execute more quickly while Java provides platform independence. C++ allows for low-level control, while Java reduces programmer burden with features like automatic memory management. Numerous other comparisons are also possible.

However, even though both languages have their advantages, in most situations a particular application must be implemented in one language or the other. This generally isn't a problem for small applications because it is relatively easy to decide whether the advantages of C++ outweigh the advantages of Java or vice-versa. However, in larger applications, consisting of multiple components, one component may be best suited to C++ while another is best suited to Java. This can put development teams in the unenviable position of having to choose a single language, knowing that whatever choice is made, desirable features will be unavailable. Furthermore, it is common for an initial choice to be made, only to later discover that the relative importance of the language strengths was incorrect. Providing true language interoperability between Java and C++ would solve these problems, providing development teams with a more powerful programming environment.

This paper describes a process that allows C++ classes to be used transparently within a Java environment. In this paper, the process is referred to as *CJ* and the program that implements the process is called *cj*. *CJ* provides the following benefits:

1. *Conjunction vs Disjunction*: Instead of programmers having to commit to a particular language for the entire project, they can implement individual components in the most appropriate language.

2. *Legacy code*: Legacy code becomes a benefit instead of a hindrance. The ability to reuse code across language barriers allows one to leverage existing C++ code bases while adding new code in Java.

3. *Improved execution speeds*: If a particular Java class has computationally intensive methods, reimplementing the class in C++ can have substantial performance benefits.

4. *Improved memory footprint*: C++ allows for data-structures that are much more space efficient than the corresponding data-structures in Java, so reimplementing in C++ can minimize the impact of memory constraints.

The remainder of the paper is structured as follows. Section 2 provides some background information necessary for subsequent sections, including related work. Section 3 describes the *CJ* architecture. Section 4 discusses a Java class hierarchy introduced by *CJ* to provide convenient and efficient Java-level access to C++ types. Section 5 discusses inheritance issues related to *CJ*. Section 6 presents results demonstrating how *CJ* can be used to improve the performance of Java programs. Section 7 discusses additional issues and outlines possible future work. Finally, Section 8 summarizes and presents conclusions.

2 Background Material

This section introduces some terminology and summarizes some Java Native Interface basics that are necessary for subsequent sections.

2.1 The Java Native Interface (JNI)

The Java Native Interface [7], hereafter referred to as the JNI, is an API that allows Java programs to interact with code written in other languages. Within this paper we will focus exclusively on how to use the JNI to communicate between C++ and Java. The following subsections consider the basic data structures provided by the JNI and the rules used by Java to determine which JNI function to invoke when a native Java method is invoked.

2.1.1 JNI data-structures, functions and types

The JNI provides a variety of data-structures and types. These data-structures represent the internal implementation of Java constructs like objects, classes, fields, methods and primitives. In the context of CJ, the `JNIEnv` structure is of particular interest. Native methods for a Java class are written using the API defined on `JNIEnv` so that access to the JVM code itself is not necessary.

The JNI also provides the types `jboolean`, `jbyte`, `jchar`, `jshort`, `jint`, `jlong`, `jfloat` and `jdouble` to refer to the JVM-specific representation of the Java primitive types `boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double` respectively. The types `jobject` and `jclass` are used by the JNI to represent a instance of a Java class and the class itself respectively. The unique identifier associated with each method has the JNI type `jmethodID`.

2.2 JNI Function Signatures

The Java Native Interface specification dictates an encoding scheme used to name native methods. The general form of the encoding is 'Java_<class>_<name>__<types>' where <types> encodes the static types of the arguments to the methods. It is not necessary to understand the details of the encoding scheme used to understand the remainder of the concepts explained in this paper. It is simply important to note that an encoding scheme is used which allows the function implementing a native method to be determined unambiguously.

2.3 Terminology

This section defines some terminology that will be used throughout the remainder of the paper.

2.3.1 Types, Classes and Methods

Much of the paper will discuss the JNI boundary code that is needed to glue C++ and Java

together. Unfortunately there is some ambiguity inherent when describing this boundary because many of the same concepts such as classes, data types and methods exist within the boundary itself and on both the Java and C++ side of the boundary. It will be useful to make a conceptual distinction between three different contexts: *C++*, *Java* and *JNI boundary*. These contexts allow us to clarify the ambiguity in referring to language constructs. We will avoid making reference to constructs such as methods and classes without clarifying the context.

2.3.2 Proxy Classes: Proper vs. Opaque

The primary purpose of CJ is to provide transparent access to C++ code within a Java environment. The idea of a *proxy Java class* is critical to our implementation. It is a Java class providing access to a C++ data-structure, and possibly, to C++ functionality as well.

All proxy classes contain a single integer field called `_jni`. This field is used to store a pointer to a C++ data-structure. The name of the proxy class is dictated by the name of the underlying C++ data structure.

Two kinds of proxy classes exist. When *CJ* is applied to a user-defined C++ class, a *proper proxy Java class* is generated that has native methods for each public and protected method within the C++ class. It is not necessary to place methods in the proper proxy Java class that correspond to private methods in the C++ class because these methods will not be invoked within any Java code. However, a class that appears as an argument type[1] within a method definition results in the generation of an *opaque proxy Java class* unless a proper proxy class associated with the C++ class already exists.

2.3.3 NativeObject

CJ provides a hierarchy of predefined Java classes that represent *semi-opaque* proxy classes. The class `NativeObject` is the root of this hierarchy. Various classes in this hierarchy have constructors and methods defined on them that provide Java programmers with facilities for creating and manipulating objects of various C++ types. Furthermore, *CJ* automatically generates new classes within this hierarchy as necessary during the creation of proper proxy classes. Proper proxy classes are not within the NativeObject hierarchy. The details of the `NativeObject` hierarchy are discussed in Section 4

2.4 Related Work

A variety of papers have discussed the idea of using the proxy concept to provide Java/C++ interoperability. Some other implementations are also available. This is not surprising given that the basic idea is simple and obvious.

One such implementation is SWIG [1]. Although there are a variety of similarities between *CJ* and SWIG, *CJ* has a different focus and addresses a variety of issues that are not considered in SWIG. In particular *CJ* intentionally chooses Java data-structures that correspond efficiently to C++ data-structures while SWIG attempts to convert arbitrary Java data-structures to appropriate C++ structures. *CJ* focuses on finding ways to provide access to arbitrary C++ methods in Java, while SWIG focuses on finding ways to pass arbitrary data-structures to C++. *CJ* allows one to transparently mix C++ and Java classes in an inheritance hierarchy while SWIG does not address inheritance issues. *CJ* does not require C++ source files[2] while SWIG requires extensive annotation of C++ source files. Related to this, *CJ* attempts to automate as much of the process as possible while SWIG requires the programmer to provide substantially more information. *CJ* also provides a variety of Java classes that make interfacing with C++ types easier. SWIG does not address this problem.

CxxWrap [3] and FooteC++ [4] also discuss integrating Java and using the idea of a proxy class. These papers do not consider many subtle issues such as pointers, references, templates, default arguments, unsigned types, const arguments, operator overloading, garbage collection etc. In contrast *CJ* has full

[1] From this point forward, discussions about arguments apply to return types as well unless otherwise noted

[2] *CJ* only requires the C++ header files.

support for garbage collection, addresses all of the aforementioned issues and provides a tool that usually performs all code generation automatically.

Microsoft JDirect [8], Instantiations' Flash Compiler [5] and Asymetrix SuperCede [2] are commercial products that offer various options for Java/C++ interoperability. Unlike *CJ*, each of these products is compiler and platform dependent. The interoperability provided by *CJ*is complete, even allowing inheritance relationships between classes written in different languages, and can be used on any platform with any C++ compiler and Java development environment that implements the JNI.

3 The CJ Architecture

The mechanism used to provide transparent access to C++ classes within Java programs is quite simple. For each C++ class to be added to the Java environment, we create a proper proxy Java class that contains a pointer to an instance of the real C++ class. Method invocations on the Java object are delegated to the C++ object it contains via the JNI. The proxy Java class and JNI code are automatically generated by *cj*.

Figure 1 shows the five steps that *CJ* performs in order to provide transparent access to the C++ class, `C`, within a Java environemnt. The five steps are:

1. generate the java source file `C.java` from the C++ file `C.hh`
2. generate `C.class` from `C.java` using `javac`
3. generate `C_jni.hh` from `C.class` using `javah`
4. generate `C_jni.cc` from `C_jni.hh` and `C.hh`
5. generate `libC.so` from `C.cc` and `C_jni.cc` using `g++`

Note that steps 2, 3 and 5 are trivial delegations to other executables while steps 1 and 4 are performed internally by *CJ*. Furthermore, note that from the user's perspective there is only one step - invoking *cj*on the C++ header file.

The rest of this section discusses how these activities are accomplished transparently, without requiring any additional actions from the programmer. During the discussion keep in mind the paradigm that this paper is focused on: An individual is writing code in a Java environment and wants to be able to use C++ classes within that Java environment transparently.

3.1 Generating the Proxy Java Class (`cj`)

The first step performed by *cj*is the creation of the java source file `C.java`. This file contains the Java source code for the proper proxy class corresponding to the class defined in `C.hh`. A Java programmer can send messages to and obtain results from this class. However, the class has no state except one special field named `_jni`. This field stores a pointer to an instance of the underlying C++ class that the Java class proxies. In order to generate a proxy Java class issues at both the method level (proxy Java methods) and the type level (proxy Java types) need to be considered. These issues are discussed in the following sub-sections.

3.1.1 Java Proxy Methods

A direct mapping exists between methods in the C++ class and methods in the corresponding proxy Java class. Specifically, for each public and protected method in the C++ class there is a corresponding method in the proxy Java class. Furthermore, for each public or protected constructor in the C++ class there is a corresponding constructor/method pair in the proxy Java class. This constructor/method pair is required because Java constructors are not permitted to be native. As a result, the generated Java constructor is a trivial delegation to its corresponding native method.

The details of what code is generated for constructors and method sis discussed in Section 3.4.

Static Methods: Methods marked as static have the same semantics in C++ as in Java.

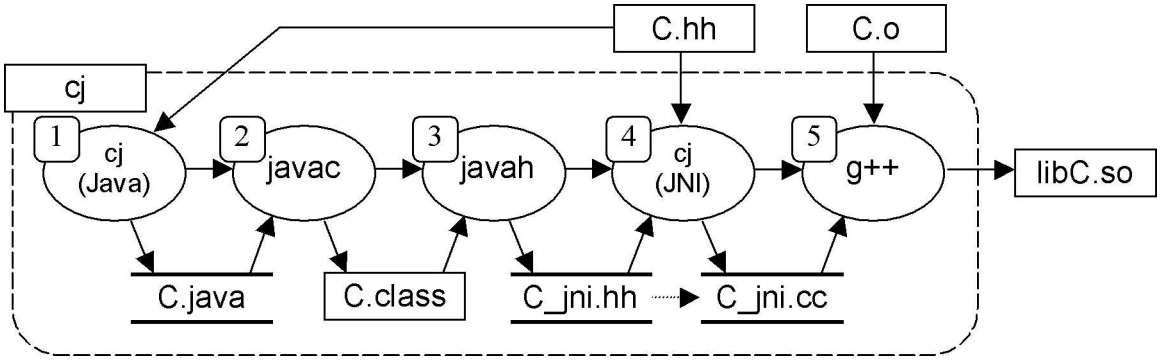

Figure 1: Dataflow Diagram for *CJ* Architecture

Namely, the method is scoped within the class, but is not associated with instances of the class. In particular, there is no *this* variable for static methods. Because the semantics are the same static methods in the C++ class are simply mapped to static methods in the proxy Java class.

Overloaded Operators: *CJ* deals with C++ operator overloading within a class by defining symbolic names for the various operators that can be overloaded. These symbolic names are used as the names of the proxy Java methods associated with the C++ operator methods. For example, the C++ method `int& operator=(int i)` is mapped to the proxy Java method `void assignTo(int i)`[3]. Although *CJ* provides intuitive names, it is possible for the user to override the default names if they do not suite the user's needs. The mapping of C++ operators to proxy Java classes does not apply to operators defined at global scope because they are not within the class being proxied.

Default Arguments: Unlike Java, C++ has the ability to specify default arguments for methods. *CJ* allows a Java programmer to make use of this facility by mapping a C++ method with *n non-default arguments followed by* k default arguments to *k + 1 proxy Java methods. One of the generated methods will have* n arguments and use the default value for the remaining *k arguments. Another method will have* n + 1 arguments. using the default values for the remaining *k - 1 arguments. This applies similarly for* n + 2, *n + 3, ..., up to* n + k, where all arguments are required and no defaults are provided.

[3]Return values of assignment operators are void in *CJ* to emphasize the inability to chain multiple assignments.

3.1.2 Proxy Java Types (C++ to Java Type Mappings)

In order to generate the proxy Java methods, a number of issues related to mapping C++ types to Java types must be considered. Figure 2 summarizes the rules that *CJ* uses to perform this type-mapping. The figure shows how primitive types like `int` are handled as well as how class types like `SomeClass` are handled. All primitive types are handled in the same manner as `int` and all class types are handled in the same manner as `SomeClass` except as noted in the section on Special Types below. The figure also shows the JNI types corresponding to Java and C++ types, and how a variable `argk` with a particular JNI type can be converted to its equivalent C++ type. In the table, `getRaw(env,argk)` is a function supplied the *CJ* that returns the C++ object stored within the JNI variable `argk`. This function is only applied to JNI variables of type `jobject` that *CJ* knows represents a proxy Java object.

The mapping between C++ and Java types is crucial for a number of reasons. First, since we are assuming that the programmer is in a Java environment using some C++ classes, we need to provide Java programmers with simple and intuitive ways of dealing with the C++ type system. If a C++ method expects an in-

C++ Type	Java Type	JNI Type	Boundary
`int`	`int`	`jint`	`argk`
`unsigned int`	`long`	`jlong`	`(argk<0)?throwException():argk`
`int*`	`IntPtr`	`jobject`	`(int*)getRaw(env,argk)`
`int[]`	`IntArr`	`jobject`	`(int*)getRaw(env,argk)`
`int&`	`IntRef`	`jobject`	`*(int *)getRaw(env,argk)`
`const int`	`final int`	`jint`	`argk`
`const int*`	`IntPtr`	`jobject`	`(const int*)getRaw(env,argk)`
`int*const`	`final IntPtr`	`jobject`	`(int*)getRaw(env,argk)`
`const int*const`	`final IntPtr`	`jobject`	`(const int*)getRaw(env,argk)`
`const int[]`	`IntArr`	`jobject`	`(const int*)getRaw(env,argk)`
`const int&`	`int`	`jint`	`argk`
`SomeClass`	`SomeClass`	`jobject`	`*(SomeClass *)getRaw(env,argk)`
`SomeClass&`	`SomeClass`	`jobject`	`*(SomeClass *)getRaw(env,argk)`
`SomeClass*`	`SomeClass`	`jobject`	`(SomeClass *)getRaw(env,argk)`
`SomeClass[]`	`SomeClassArr`	`jobject`	`(SomeClass *)getRaw(env,argk)`
`const SomeClass`	`SomeClass`	`jobject`	`*(const SomeClass *)getRaw(env,argk)`
`const SomeClass&`	`SomeClass`	`jobject`	`*(const SomeClass *)getRaw(env,argk)`
`const SomeClass*`	`SomeClass`	`jobject`	`(const SomeClass *)getRaw(env,argk)`
`const SomeClass[]`	`SomeClassArr`	`jobject`	`(const SomeClass *)getRaw(env,argk)`

Figure 2: Mapping between C++, Java and JNI types

teger pointer, a constant reference or a pass-by-value object, the Java programmer needs to know what Java objects to pass to the proxy Java method.

In addition to ease-of-use, we want to ensure that crossing the language boundary is as efficient as possible. This implies that the Java type chosen to represent a particular C++ type should allow efficient conversion to the C++ type. More specifically, we want to choose Java types whose memory layout is as close to the corresponding C++ memory layout as possible. When possible, we want to choose a Java object that has the C++ object as a subcomponent so that we can simply extract it instead of converting to an entirely different layout. Efficiency considerations also imply that we want to avoid maintaining two different copies of data-structures (one on either side of the language boundary), as this will lead to wasted time maintaining consistency.

Another important consideration is type safety. There are some differences in how C++ and Java deal with types. These differences put the convenience of particular type-mappings at odds with type safety. In such situations, *CJ* generates multiple Java proxy methods, providing a type-proper method and one or more type-convenient methods. All of these methods are native, and associated JNI code is generated by *CJ*. All such methods are type-safe, but the type-convenient methods are less efficient than the type-proper method because they require additional checks to ensure type safety.

One final priority of *CJ* is an emphasis on providing the desirable property of separate compilation. *CJ* has been designed so that changes to a C++ class requires only that the proxy class associated with that C++ class be regenerated. Currently, this is true even in superclass/subclass situations (adding a method to a class does not require proxies for subclasses to be regenerated), although this extreme form of separate compilation may be removed in the future in order to more fully support multiple inheritance. The desire for separate compilation influenced many of the issues discussed in this paper.

Primitive Types (`int`, `char`, ...): On most modern architectures the C++ types `int` and `long` both represent 32-bit integers and a C++ `short` represents a 16-bit integer. On such architectures *CJ* maps C++ `int` and `long` to Java `int`, C++ `short`, keywordfloat, and `double` to Java `short`, `float` and `double` respectively and C++ `bool` to Java `boolean`. The C++ `char` is treated specially. A Java `char` is a two-byte entity, whereas a C++ `char` is a one-byte entity. If a C++ method signature contains a `char` two different proxy Java meth-

ods are generated, one that is *type-proper and one that is* type-convenient. The type-proper method uses the Java type `byte`. The type-convenient method uses the Java type `char`. The type-convenient version is less efficient because the native code verifies that the high-order byte is clear. If it is not, an exception is thrown by the native code.

Pointers to Primitive Types (`int*`, ...): *CJ* relies on the existence of the `NativeObject` hierarchy to deal with pointers to primitive types. As mentioned in Section 2.3.3, this hierarchy consists of a collection of Java classes whose sole purpose is to provide convenient, safe, efficient and memory-managed access to various C++ data-structures.

An unfortunate ambiguity exists in how C++ pointers can be interpreted. The type `int *` can be interpreted as *pointer-to-int or as* array-of-integers. Although the low-level details are the same under both interpretations, the correct interpretation is crucial for safe usage, both when dereferencing and when deleting dynamically allocated memory.

C++ provides a mechanism to partially clarify this ambiguity. The C++ type `int arr[]` is interpreted as `int* arr` by compilers, but indicates that the variable is an array-of-integers, not a pointer-to-integer. Unfortunately, there is no corresponding syntax to unambiguously indicate pointer-to-integer. As a side note, in C++, the syntax `int arr[]` is legal, but `int[] arr` is not legal, whereas in Java, both syntaxes are legal. In subsequent discussion, references to the C++ type `int[]` should be understood as shorthand for `int arr[]` (this allows us to avoid specifying variable names in situations where the variables are irrelevant).

Yet another problem related to this ambiguity relates to differences between primitive pointers and class pointers. The C++ type `char*` almost always means array-of-chars, and `int*` is arguably more likely to mean array-of-integers than pointer-to-integer. Since C++ has reference types, the need to pass pointers to single primitive values has lessened considerable compared to C. Thus, as a general rule, primitive pointers more commonly mean *array-of-instances* than *pointer-to-instance*. On the other hand, the C++ type `FMatrix*` is more likely to mean *pointer-to-instance* than to mean *array-of-instances*. Minimizing user burden was given a higher priority than providing a consistency set of rules for interpreting pointers. As a result, *CJ* has asymmetric rules on how to interpret primitive and class pointers, as will be discussed below.

Having discussed the pointer ambiguity, we can now present *CJ*'s solution. As mentioned before, the `NativeObject` hierarchy represents a collection of opaque (and semi-opaque) Java classes that provide Java programmers access to C++ data-structures. One subclass of `NativeObject` is `Ptr`, which is an abstract superclass of a variety of semi-opaque proxy Java classes used to represent C++ pointers to primitives that are to be interpreted as *pointer-to-instance*. Another subclass of `NativeObject` is `Arr`, which is an abstract superclass of a variety of semi-opaque proxy Java classes used to represent C++ *array-of-primitives*. Because primitive pointers more often represent arrays than single instances, *CJ* by default maps `int*` to `IntArr`, `char*` to `CharArr`, etc. As well, the C++ type `int[]` is mapped to the Java type `IntArr`, and `char[]` is mapped to `CharArr`, etc. *CJ*'s heuristic for mapping primitive pointers to the *array-of-primitives* interpretation is not always correct, so *CJ* provides facilities to allow programmers to explicitly specify the interpretation desired on a per-signature basis. Furthermore, much of *CJ*'s default behavior is configurable, including how `int*`, `char*`,etc. are mapped.

Both of the alternative Java type representations also suffer from a second problem related to garbage collection. *CJ* would like to take advantage of Java finalizers to garbage collect C++ objects. However, this requires that *CJ* be able to write finalizers on proxy Java classes. Neither `Integer` or `int[]` allow for such finalizers.

By introducing special Java classes (subclasses within the `NativeObject` hierarchy) that represent primitive C++ types, we can avoid the problems mentioned above. First, the opaque proxy classes in the `NativeObject` hierarchy define constructors and methods to allow a C++ pointer to be created and manipulated. Subclasses of `Arr` have constructors to create C++ arrays, and methods to get and set par-

ticular elements. Subclasses of `Ptr` have constructors that create space for a single C++ primitive, and methods that get and set this value. Second, once `IntPtr` is available, it is a more intuitive Java counterpart to C++ `int*` than `Integer`, making the Java code that interfaces with C++ more readable to C++ and Java programmers alike. Third, finalizers can be defined on these classes. Note that it is especially important to distinguish between the two pointer interpretations when dealing with finalizers. Subclasses of `Arr` will use `delete[]` to deallocate C++ memory, while subclasses of `Ptr` will use `delete`.

Since C++ programs can set C++ pointer types to NULL , *CJ* provides type-safe facilities for setting, testing and manipulating NULL pointers. To emphasize the transparency of the language mapping, this is accomplished using Java's own `null` object. Since Java `null` can be used anywhere a class type is expected, `null` can be used for arguments with type `IntPtr`, `CharPtr`, etc. During argument marshaling (Section 3.4.2), *CJ* tests arguments to determine whether the `null` object was passed in or not.

As an example summarizing the discussion in this section, if *CJ* encounters a C++ method definition of the form:

```
int* func(int* v, int v2[])
```

it will generate the Java proxy method

```
IntPtr func(IntPtr v, IntArr v2)
```

The Java object `null` can be passed in for both of the arguments, and may be returned as the result.

Class Types (`SomeClass`, `SomeClass*`, `SomeClass&`): It is the ease with which C++ class types can be merged into Java that provides *CJ* with its power. In *CJ* any C++ class types `SomeClass`, `SomeClass*`, and `SomeClass&` are all mapped to the same proxy class `SomeClass`. Any C++ type `SomeClass[]` is mapped to `SomeClassArr` which is a subclass of `Arr` within the `NativeObject` hierarchy. This allows the Java programmer to pass objects as arguments to C++ methods without concerning themselves with the details of pointers.

Note that in the above discussion we did not specify whether a specific Java proxy class was proper or opaque. That is because special care was taken to ensure that *CJ* generates exactly the same code whether the C++ class type of an argument is made into a proper or opaque proxy class.

If *CJ* maps a C++ type to a particular Java proxy class, then *CJ* is responsible for ensuring that that proxy class actually exists. It is a simple matter for *CJ* to search the directories in the Java CLASSPATH looking for the necessary source or class file. If the necessary file is not found *CJ* automatically generates an opaque proxy class as discussed in Section 3.1.2. As a result, C++ types that look like classes can be treated as classes and full-fledged proper proxies can be created incrementally as the programmer identifies a need for Java access to particular C++ classes.

We are now in a position to explain the asymmetry between primitive pointers (and references) and class pointers (and references). The C++ type `int*` maps to `IntArr` (by default) and `int&` maps to `IntRef`, while `FMatrix*` maps to `FMatrix` (instead of `FMatrixArr`) and `FMatrix&` maps to `FMatrix` (instead of `FMatrixRef`). To justify our implementation, note that it is possible that `FMatrix` is a proper proxy class, whereas any other Java class that could represent an `FMatrix` will be an opaque proxy class. Since proper proxy classes have the entire public interface of the underlying C++ class available to them, it is preferably to provide the Java programmer with proper proxy classes where ever possible. Thus, for class types, all three C++ types map to the potentially proper proxy class `FMatrix`. This rational is not applicable to primitive types because they do not have proper proxy representations.

An important subtlety must be discussed at this point. Because the class types `SomeClass`, `SomeClass*` and `SomeClass&` all map to the same Java type `Someclass` conflicts can occur when function signatures differ only by pointers in the types. When such conflicts occur *CJ* solves the conflict by favouring pointer types over reference types over value types.

One final note on the relative importance of convenience and safety is in order here. The use of different Java proxy types for the two different pointer interpretations may place an added burden on the programmer. For class types,

CJ assumes that * means *pointer-to-instance*, and relies on the C++ programmer to have used [] in situations where *array-of-instances* is meant. *CJ* provides configuration facilities to allow users to explicitly indicate which interpretation is desired on a per-signature basis, and although the configuration does not require modification of either the source or header files, it does require a minimal amount of work on the programmers part. It was decided during the implementation of *CJ* that automatic memory management of C++ objects using the Java garbage collector was worth the increased programmer burden in this situation.

Other Pointer and Reference Types We have discussed how *CJ* deals with primitives, classes, simple pointers and simple references, but have not yet addressed types like `int**`, `char*&`, `FMatrix[][]` and `FMatrix**&`. Such types are represented by opaque proxy classes, automatically generated by *CJ* as necessary. Each * in a C++ type is converted to `Ptr`, each & is converted to `Ref`, and each [] is converted to `Arr`. Thus, *CJ* maps the C++ types cited above to the opaque proxy classes `IntPtrPtr`, `CharPtrRef`, `FMatrixArrArr` and `FMatrixPtrPtrRef` respectively.

The `const` keyword: The C++ concept of `const` is not fully supported in Java, but some mappings are possible. For example, the C++ type `const int` corresponds to `final int` in Java. However, the C++ type `const SomeClass *` in C++ does not correspond to `final SomeClass` in Java. Instead, `final SomeClass` is equivalent to the less commonly used C++ type `SomeClass *const`.

Unfortunately, there is no equivalent Java construct for the C++ type `const SomeClass *` so *CJ* maps this C++ type to the Java type `SomeClass`. *CJ* remembers which arguments were marked as `const` because it will need to typecast the extracted C++ object embedded within the Java proxy object to the appropriate C++ type during JNI code generation.

One final note about C++ `const` and Java `final` is in order here. Although `const` qualifiers do effect the signature of a C++ method, `final` qualifiers do not effect the signature of a Java method. Thus, a C++ method can have a method that differs only in the constness of a particular method, whereas this is not possible in Java. To solve this inconsistency between the languages, *CJ* only generates one method in such cases, favouring the non-const version.

The `unsigned` keyword: Java has no `unsigned` concept - all primitive types are signed. There are two strategies for dealing with this issue.

First, the unsigned component of C++ types could simply be ignored. This would preclude Java programmers from passing in the largest possible values. It also necessitates some verification code to ensure that the Java programmer does not pass a negative value across the language barrier. If a negative value is detected, an exception is thrown.

Second, each unsigned C++ type could be upgraded to the next largest Java primitive type. Thus, C++ `unsigned char` becomes Java `short`, C++ `unsigned short` becomes Java `int`, and C++ `unsigned int` becomes Java `long`. Lacking any better solution, C++ `unsigned long long` becomes Java `long` precluding the passing of the largest possible 8-byte integers from Java into C++ methods. This is similar to the approach taken by SWIG [1].

Although the second approach removes most of the range limitation posed by the first approach, it also requires not only a check for negative values, but also a check for values exceeding the range of the C++ type. Furthermore, this approach is not as intuitive to programmers wanting to easily convert C++ signatures to Java signatures.

CJ can be configured to provide either strategy for the mapping of unsigned primitives. By default, it uses the first approach, trading a more limited range for more efficiency.

Template Types: *CJ* provides a simple mechanism for dealing with C++ templates within Java programs. As an example, the C++ type `vector<int>` is mapped to the opaque proxy class `VectorOfInt`. In general, a C++ template type is converted to an opaque proxy class by changing C++ template types of the form `Name<arg1,...,argk>` to `NameOfArg1And...AndArgk` As with most

opaque proxy classes, there are no facilities provided to allow Java programmers to create or manipulate Java proxies of C++ template objects.

Special Types *CJ* provides special support for certain C++ types that are ubiquitous.

The C++ types `ostream` and `ofstream` both map to `FileOutputStream`. Note that C++ `ostream` is more general than the Java `FileOutputStream` type, and thus the C++ method can be used in more contexts than is being allowed from within Java. This limitation is due to the difficulties in finding a C++ data-structure for the general `ostream` class embedded within the data-structure for the general Java `OutputStream` class. When restricted to files, there is a common underlying implementation (both C++ and Java use file descriptors, at least in Unix). The limitations implied by this approach were considered worth the ability to address the most common kind of output streams (files). Related to this special type mapping, the C++ predefined variables `cout` and `cerr` map to `System.out` and `System.err` respectively.

The C++ types `istream` and `ifstream` both map to `FileInputStream` with the same caveats as for the output case. The C++ predefined variable `cin` maps to `System.in`.

The C++ types `const char*` and `const string &` map to the Java types `CharArr` and `String`, while the C++ types `char*`, `string` and `string &` map to the Java types `CharArr` and `StringBuffer`. However, the `String` and `StringBuffer` versions are highly inefficient. When using `String` the Java object must be copied to a C++ object because a Java string object does not store a layout that can be trivially converted to a C++ string. When using `StringBuffer`, a copy must be made both before the C++ method is called, and after it returns. The difference between these two occurs because `String` is immutable, and `StringBuffer` is mutable.

Primitive Type Identification: *CJ* generates opaque proxy classes for every C++ type that is neither a primitive nor a proper proxy class. This implies that *CJ* needs to be able to accurately identify which types are primitive and which are not. Since C++ has both typedefs and preprocessor directives, establishing whether a particular type is primitive is not as trivial as it might at first appear.

CJ has facilities to automatically analyze C++ header files looking for typedef and preprocessor defines related to types. However, the analysis is heuristic, and is not guaranteed to detect every possible mapping, and thus may not recognize a particular type as a primitive. In situations where *CJ* does not recognize a particular type as being primitive, an opaque proxy class will be generated to represent the type. Although not strictly incorrect, this decreases the efficiency by using an object to represent something that can be treated as a primitive.

To provide the greatest degree of flexibility, *CJ* provides facilities to allow users to explicitly specify primitive type mappings for situations where the heuristics applied by *CJ* fail.

Linking Native Code Into Java: When a native method is invoked in a Java program, the JVM is responsible for invoking the natively defined function associated with it. Before this can happen, the code representing the JNI function(s) must be dynamically linked into the running Java executable. This is taken care of automatically by *CJ* by placing the necessary code within a static block in the proxy Java class.

3.2 Compiling the Java Class (`javac`)

After producing the Java source file, *cj* invokes `javac` to compile it. This is necessary in order for subsequent activities (i.e. `javah`) to work properly. Note that there is very little Java code to compile, since most of the code associated with the Java class is native (and will be automatically generated in subsequent sections).

3.3 Generating The Native Signatures (`javah`)

Every native Java method implies the existence of a JNI function. If the Java method has N ar-

guments, then the JNI function will have $N+1$ or $N+2$ arguments depending on whether it is static or not. The first argument of each JNI function is a pointer to a `JNIEnv` instance, which provides an API for accessing the internals of objects and classes. For instance methods (i.e. non-static methods), the second argument is always of type `jobject` and represents the receiver of the Java method. *CJ* names this argument `jthis` (this name is used in subsequent discussion). The remaining N arguments correspond to the Java method arguments, and the JNI types of these arguments are dictated by the Java types of the corresponding Java method.

3.4 Generating the Native Source (`cj`)

JNI code representing the implementations of all native methods in the proxy Java class is automatically generated by *cj*. There are three different kinds of methods that need to be implemented: constructors, finalizers and instance methods.

Before discussing each of these, we describe some library routines useful during code generation. We also discuss *argument marshaling* and *memory management*, issues common to all three kinds of native methods.

3.4.1 CJ Library Routines

CJ uses a small collection of functions to facilitate generation. These methods are not needed by programmers using *cj*, but instead are used internally by *cj*.

- `proxyFor(env,cName,subj,javamem)`: Create a new proxy Java object whose Java class (a proxy) is identified by `cName` and whose underlying C++ object is `subj`. If the fourth argument `javamem` is boolean true, the proxy Java object is responsible for deallocating the C++ object.
- `getRaw(env,proxy)`: Obtain from a proxy Java object its underlying C++ object.
- `setRaw(env,proxy,subject)`: Insert into a proxy Java object a particular C++ data-structure.
- `jniField(env,proxy)`: Obtain from the proxy Java object a JNI-specific identifier for the field storing the underlying C++ object (i.e. the field with name `_jni` with type `int`). This method is only used by `getRaw` and `setRaw` and is not used directly by *CJ*.
- `swizzle`, `unswizzle`, `isSwizzled`: Internal methods used to indicate whether the C++ object should be deallocated by Java.
- `cjthrw(const char*)`: Throws a Java exception.

Note that none of these library routines make use of the C++ class, and instead deal with generic `void*` pointers. *CJ* type-casts the result of calling `getRaw` to the appropriate type as necessary. Since only *CJ* has access to the `_jni` field, these typecasts are always type-safe.

3.4.2 Argument Marshaling

The purpose of the JNI boundary code is to delegate behavior to the C++ side of the language barrier. For constructor methods, this implies creating a new instance of the appropriate C++ class, while for other methods it implies extracting the C++ object contained in the `jthis` Java object and sending a message to the extracted object.

Naturally, C++ methods require C++ arguments. Thus, all of the delegation code (constructor, instance method and finalizer alike) will need to generate code to take receiver and argument variables with JNI types and convert them into entities with the appropriate C++ types. This process is referred to as *argument marshaling*. A variable `argk` is converted from a JNI type to a C++ type using the code in column 4 of Figure 2. The discussion in subsequent sections will use the notation `marshal(arg)` to represent this code in situations where a generic specification is necessary. This notation is shorthand for "the code in column 4 of the row corresponding to the type of argk in Figure 2".

Figure 2 does not give all of the marshaling code that *CJ* uses because some of the code is cumbersome. This omitted code handles NULL arguments. For example, pointer

types test whether the argument is the Java `null` object, and if so, pass the C++ NULL object to the underlying C++ method. Reference and raw class types test whether the argument is the Java `null` object, and if so, throw a `NullPointerException`.

3.4.3 Memory Management

CJ takes advantage of Java garbage collection to provide automatic memory management of C++ objects created during the execution of a Java program. When a Java proxy object is garbage collected, *CJ* can arrange to have the underlying C++ object deallocated as well, by defining finalizers on proxy Java classes. However, this memory management requires some careful coordination between constructors, finalizers and instance methods.

First, if C++ objects are deleted when proxy objects are deleted, *CJ* must ensure a one-to-one correspondence between C++ and Java objects. This is done by maintaining a C++ data-structure that maps C++ pointers to Java `jobject` instances. Whenever a Java representation of a C++ data-structure is required, this map is first checked to determine if there is already such a Java object, and if so, it is returned (otherwise, a Java proxy object is created, the C++ object is embedded, the pair is added to the map, and the newly created Java object is returned). In subsequent discussion, the C++ variable `ProxyMap` represents this map. The method `jobject find(void* cobj)` searches the map for a Java object matching the C++ object `cobj`, and `void add(void* cobj, jobject jobj)` adds an object pair to the map.

Second, since not all C++ pointers are dynamically allocated, and not all dynamically allocated C++ pointers are the responsibility of the Java program, each proxy object maintains an indication of whether its underlying C++ object is to be deleted when the proxy is deleted. Where possible, this is accomplished by swizzling the C++ object pointer [6]. When this swizzling is not possible, JVM-specific implementations of *CJ* can take advantage of the layout of Java objects to find a bit that can be used to indicate this information. As a last resort, Java proxy objects can maintain a second field to store this information.

3.4.4 Constructors

When *CJ* is parsing a C++ class it maps the C++ constructors to Java constructors. Each Java constructor delegates to a native method that has the arguments as the constructor. The code in the native method first creates a new instance of the underlying C++ class by invoking the C++ constructor corresponding to the JNI function. The arguments to the C++ constructor are the marshalled versions of the JNI arguments. Once the C++ object has been created, a pointer to it is placed into the `_jni` field of the Java object. Before returning, the JNI function adds the C++ and Java objects to the ProxyMap data-structure to ensure a one-to-one correspondence between C++ and Java objects.

3.4.5 Finalizers

Each proxy class has a finalizer defined on it. When a proxy object is garbage collected, this finalizer is invoked. *CJ* generates code in this finalizer to determine if the Java proxy object is responsible for deallocating the underlying C++ object. If so, the C++ object is deleted. Note that because C++ uses two different operators to delete memory allocated as *pointer-to-instance* versus *array-of-instances*, it is critically important that *CJ* be able to identify the correct interpretation of pointers. This is the primary reason why *CJ* has both `IntPtr` and `IntArr`, and why `IntPtr` always means *pointer-to-instance* while `IntArr` always means *array-of-instances*. Note that the preceding discussion on Finalizers applies to both proper and opaque proxy classes.

3.4.6 Instance Methods

The boundary code for instance methods consists of the following activities:

1. Marshaling of the receiver and arguments to obtain C++ objects from JNI arguments (this was discussed in Section 3.4.2).

2. Invoking the appropriate C++ method on the marshaled receiver with the marshaled arguments to obtain a C++ return value.

This is straightforward, since each Java proxy method is associated with a single C++ method

3. Returning the Java construct that represents the C++ return value.

It is the handling of return values that introduces some subtleties. There are three possible kinds of return types that the JNI function can have:

1. `void`: For void returns, the JNI function does nothing besides invoke the C++ method.

2. a primitive type like `jint`: For primitive type return values, the JNI function can easily convert the C++ type to its Java equivalent.

3. the object type `jobject`: If the JNI function returns an object, it must first create a new instance of the appropriate Java proxy class, store the C++ return value within this proxy object, and return the result. There are three issues related to this process requiring additional details.

   First, *CJ* needs to address the one-to-one mapping between C++ objects and Java proxy objects required by the JNI finalizers. Rather than always creating a new Java object, the `proxyFor` library routine first looks in a `ProxyMap` hash whose keys are C++ objects and whose values are Java proxy objects. If a proxy object matching the C++ return object already exists, that object is returned as the result of the proxy Java method. If not, a new Java object is created, the C++ object is stored within it, the C++/Java object pair is recorded in the hash, and the Java object is returned.

   In addition, *CJ* needs to know if the C++ return value is under Java memory management or not. By default, *CJ* assumes not (it is better to have a memory leak than to deallocate something that was not deallocatable) and thus does not mark the proxy object as responsible for deallocating the C++ object. For the relatively rare situation where a C++ method returns an object that is the responsibility of the caller to memory manage, *CJ* provides facilities to allow the user to indicate that the return result is under Java memory management, in which case *CJ* marks the Java object appropriately.

   Finally, if the return type of the C++ method is a class (not a pointer or reference, but a class itself), then C++ automatically invokes the copy constructor of the class when returning the result. Since the result value is local to the JNI function, *CJ* must do something to provide more permanent access to the value. This is accomplished by dynamically creating a new instance of the C++ class using the copy constructor (the new instance is a copy of the result value). This new copy is then stored within a newly created Java proxy object.

3.5 Compiling JNI code into a shared library (g++)

At this point *CJ* must generate the shared library that will be used by the JVM to execute the native methods defined on the proxy class. The exact sequence of commands needed is both compiler and platform specific. As a result, the exact command used is user configurable, allowing *CJ* to be used on any platform with any C++ compiler.

4 The NativeObject Hierarchy

All opaque proxy classes provided or generated by *CJ* are subclasses within the `NativeObject` hierarchy. Some of these classes are predefined by *CJ*, and have a limited interface defined on them. These classes are thus *semi-opaque*. As well, during the creation of proper proxy classes, *CJ* may generate proxy classes within this hierarchy. Most of these classes have absolutely no functionality defined on them, and are thus truly opaque but some generated classes are semi-opaque and *CJ* generates all code necessary for them. All classes within the `NativeObject` hierarchy are placed in the `cj` package.

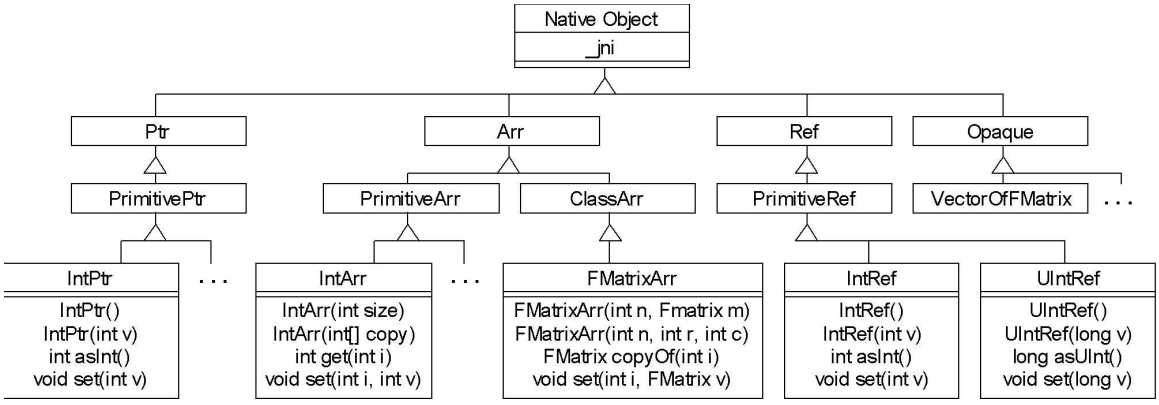

Figure 3: Native Object Hierarchy

4.1 NativeObject

`NativeObject` is the abstract superclass of all opaque proxy classes. It provides the private `_jni` instance variable.

4.2 Ptr

`Ptr` is the abstract superclass of all opaque proxy classes representing pointers to single C++ objects. In particular, instances of these classes are always interpreted as *pointer-to-instance*, never as *array-of-instances*. The subclass `PrimitivePtr` is an abstract superclass of all opaque proxy classes for *pointer-to-primitive* classes. *CJ* provides predefined subclasses of `PrimitivePtr` for the various C++ primitive types (i.e. `IntPtr`, `CharPtr`, etc.) including unsigned primitive types (i.e. `UIntPtr`, `UCharPtr`, etc.).

The subclasses of `Ptr` define a variety of constructors and methods. They cannot be defined in `Ptr` itself because the return type and/or arguments of the methods are primitive-specific. For example, the class `IntPtr` has two constructors, one with no arguments that allocates space for an integer and initializes it to the Java default of 0. The second constructor has one integer argument specifying what to initialize the allocated integer to. `IntPtr` also has two methods, `int asInt()` which returns the integer that the proxy Java object points to, and `void set(int v)`, which sets the integer to the argument value. The other primitive classes provide similar methods to read or modify the value of the primitive. Unsigned primitive types are widened as described earlier when the implications of the `unsigned` keyword were discussed.

4.3 Ref

`Ref` is the abstract superclass of all opaque proxy classes representing references to C++ primitives types. The methods in this hierarchy are identical to the `Ptr` hierarchy in all ways (except for the names of the classes).

4.4 Arr

`Arr` is the abstract superclass of all opaque proxy classes representing references to C++ array types. In particular, instances of these classes are always interpreted as *array-of-instances*, not *pointer-to-instance*. There are both `ClassArr` and `PrimitiveArr` subclasses.

Subclasses of `PrimitiveArr` declare two constructors. One of these has an integer argument indicating how many elements to allocate (which initializes each element to the default value Java dictates for the primitive associated with the array). The second constructor takes a Java array of primitives and creates a C++ copy of this array. Two methods are also defined, `get` and `set`, which return the value at an index and set the value at an index respectively.

Unlike the `PrimitiveArr`, `Ptr` and `Ref` class hierarchies, which are pre-defined by *CJ*, subclasses of `ClassArr` are not pre-defined. Instead, for each proper proxy class that *CJ* pro-

duces, a corresponding subclass of `ClassArr` may also created. Furthermore, *CJ* automatically generates a variety of constructors and methods on these semi-opaque proxy classes. In general, subclasses of `ClassArr` will have N constructors if its corresponding proper proxy class has N constructors. Each of these constructors will have one more argument than the corresponding constructor in the underlying class, representing an integer size for the array. After the size argument, the arguments of each constructor in the array match those of the corresponding constructor, and are used by *CJ* to initialize each element of the array (in this way, *CJ* does not force C++ classes to define default constructors).

4.5 Opaque

All Java classes that *CJ* generates that do not fall into one of the three hierarchies discussed in the previous sections are made subclasses of `Opaque`, emphasizing the fact that such classes have absolutely no functionality and provide no means for Java programmers to create or manipulate them. Thus, the only way that Java programmers can deal with such classes is if a proxy Java method returns an instance of such a class, in which case this instance could be used as an argument to some other Java proxy method expecting such a class. Examples of such opaque classes include `IntPtrPtr`, `FloatPtrRef`, and `PairOfIntPtrAndPerson`.

5 Inheritance in *CJ*

5.1 Java Superclass

There are times where a C++ class is standalone within an C++ environment, but is to be made a subclass of a Java class within the Java environment. From an implementation perspective, this corresponds to establishing what class the proxy Java class should extend. *CJ* provides three different ways to specify this kind of inheritance. First, a comment in the C++ class definition can be used to tell *CJ* to make the proxy Java class a subclass of another Java class. Second, a flag to *cj* can be used to specify a Java parent. Third, the parent can be specified in a configuration file.

Having discussed how the proxy Java class associated with a C++ class can be made a subclass of an existing Java class, we can now discuss what effect this has on the interaction between Java and C++. Within a Java program, we are able to create an instance of a C++ class (such an instance is a proxy Java object containing a C++ object), and we know that sending messages to such a Java object will result in sending the associated C++ method to the underlying C++ object. However, what happens if we send a message that is only defined in the Java parent?

CJ provides cross-language inheritance as long as one fundamental constraint is honoured in the implementation of Java parent class. Note that this is a non-trivial claim because the two classes may use entirely different data-structures to represent the state of the object. It is this difference in data-structures that leads to the constraint: all state must be accessed only through accessor methods. Thus, as long as the definition of a method defined in the Java parent uses the accessor methods defined in the C++ child to get and set value, the proxy class can inherit this code and expect it to work properly because *CJ* will provide reimplementations of the accessor methods, getting and setting values within the C++ data-structure rather than within the inherited Java data-structure. This functionality is demonstrated by the matrix example presented in Section 6.

It has been known for many years that accessing state directly leads to fragile code, so for good programmers the constraint mentioned above is not a constraint at all, but rather the normal way of writing code.

5.2 C++ Superclasses

c++-superclasses

5.2.1 Single Inheritance

single-inheritance

In situations where *CJ* is being applied to a C++ class `B` which is a subclass of another C++ class `A`, *CJ* generates an opaque proxy class for `A` if a proxy class for `A` does not already exist. The proper proxy class `B` being generated is then made a subclass of the proxy class `A`.

When *CJ* is generating proxy Java methods, it looks at the public methods in the C++ class. If the C++ class has a superclass, then the Java proxy class should also be able to respond to methods inherited from these C++ superclasses as well. Although *CJ* could analyze the entire C++ inheritance hierarchy and define proxy methods for all methods publicly inherited, a different strategy has been chosen. *CJ* only generates proxy methods for those methods defined locally within the C++ class being proxied. However, if the C++ parent class is also made into a proper proxy, then it will have the methods defined locally on the parent class, and the originally proxy class will inherit these methods. This approach gives the user some control over what methods are available, more accurately mirrors the inheritance structure of the C++ classes in the Java proxy hierarchy, and maintains the separate compilation property.

5.2.2 Multiple Inheritance

C++ has true multiple inheritance, whereas Java does not. In order to handle this, *CJ* could explicitly define Java proxy methods for all methods in all superclasses of the C++ class being proxied. However, this approach has the unfortunate side-effect of breaking the separate compilation property that *CJ* has so far managed to maintain. By separate compilation, we mean that changes to a C++ class that is proxied requires only that that class be recompiled with *cj*, and that no other proxy classes need to be updated. By copying methods down to handle multiple inheritance, this properly is no longer true (if a new method is defined in any C++ superclass, then proxies for all subclasses of that class must be regenerated). By default, *CJ* does not copy down methods and instead only provides single inheritance. However, a flag can be provided that tells *CJ* to accumulate all signatures of all parents and to generate proxy Java methods for all of them. When this flag is used, the separate compilation property is no longer exists, and the user is responsible for regenerating all subclasses as appropriate when new methods are added within the C++ hierarchy.

5.2.3 Virtual Methods

At first glance, one might assume that non-virtual C++ methods are mapped to final proxy Java methods. However, the C++ concept of non-virtual is not quite as restrictive as the Java concept of final. In C++, a method that is not declared virtual is uniquely identifiable at compile time, as the actual method to invoke is dictated by the static type of the variable that the receiver is stored in. Non-virtual methods do not preclude a subclass from defining the same method and hence overriding the definition from above. However, this new method will only be used in situations where a receiver is statically typed to be of this subclass.

On the other hand, a method marked as `final` in Java will result in a compiler error if subclasses attempt to override the method. Because of this difference, *CJ* does not map non-virtual C++ methods to Java final methods. All proxy methods generated by *CJ* are non-final.

6 Performance Results

7 Additional Issues and Future Work

7.1 User Configuration in *CJ*

In order to provide users with the greatest degree of flexibility, *CJ* is highly configurable. Configuration options are typically specified in a configuration file although some options can also be passed as command line arguments to *cj*. A default configuration file can be generated automatically by *CJ* that can then be modified by the user if necessary. This minimizes the amount of configuration work that must be performed before *CJ* can be used. The details of the configuration file are described in the documentation provided with the *cj* executable and are beyond the scope of this document.

7.2 Future Work

7.2.1 Semi-Opaque Proxy Classes

At the current time *CJ* does not provide an easy mechanism to allow users to upgrade opaque proxy classes to semi-opaque proxy classes. For example, a specific user may find it desirable to add functionality to the opaque proxy class `IntPtrPtrPtr`. There is nothing stopping users from adding functionality to this class, but such additions require the user to write JNI code. *CJ* may be able to help automate this process in the future.

7.2.2 Globally scoped operators

global-operators

CJ does not currently handle globally scoped operators. One commonly overloaded operator at global scope is `ostream& operator<<(ostream& os, ...)`. Support for globally scoped operators in general, and this operator in particular, would be a useful feature.

7.2.3 Minimizing the cost of boundary crossings

min-boundary

Currently, *CJ* emphasizes correctness above efficiency, and there are many tests that are applied in the boundary code to ensure correctness. Some effort should be made to minimize these costs without sacrificing correctness.

7.2.4 Platform Independent Native Code

The use of native code lessens the generality of a Java application, since it cannot be used in web applications and some other situations where dynamic loading of classes is necessary. There are, however, ways to address this problem. One simple idea would be to introduce a new attribute in `.class` files that stores a collection of architecture/URL pairs. The keys are strings representing unique encodings of architectures, and the values are URLs to shared libraries compiled for the platform identified by the key.

This idea is especially useful in situations where the C++ code can be considered to be a plug-in optimization (like our `FMatrix` example) and there exists a default Java implementation. In such situations, a virtual machine that recognizes the new attribute can check whether the current architecture is listed in the architecture/URL map, then execute the efficient C++ code, otherwise execute the slower, but still functional, Java code.

8 Conclusion

About the Authors

Blah, Blah, Blah...

9 Author's Notes

We need to translate protected methods as well because we may have the case where a C++ class is translated and then a new Java class is written that inherits from the proper proxy class.

Make sure the section on the native object hierarchy tells how new semi-opaque proxy classes are generated automatically by CJ.

Java long should map to a C++ long long int.

Why don't we use SomeClassPtr and SomeClassRef and provide the ability to convert between the types. Is it just too much work for a Java programmer?

I am concerned that I have cut too much stuff out of the section on Constructors

The stuff on static methods is WRONG. There are still N+2 arguments. The second is a jclass rather than jthis. We are generating totally wrong code here and really need to fix this to get meaningful timing results.

References

[1] David Beazley, William Fulton, Matthias Koppe, Lyle Johnson, and Richard Palmer. Swig1.3 development documentation, 2002. http://www.swig.org/Doc1.3/index.html.

[2] Netscape Communications Corporation. Supercede java edition, 1999. http://developer.netscape.com/software/tools/listing/core/java/Asynmetrix/asymetrix.html.

[3] David Deaven. Cxxwrap documentation, 2002. http://www.deaven.net/ deaven/Software/cxxwrap/doc/index.html.

[4] Bill Foote. Integrating java with c++, 2001. http://www.javaworld.com/javaworld/javatips/jw-javatip17.html.

[5] Instantiations Inc. Flash compiler, 2002. http://www.instantiations.com/SuperCede/flashcompiler.htm.

[6] Alfons Kemper and Donald Kossmann. Adaptable pointer swizzling strategies in object bases: Design, realization and quantitative analysis. Technical report, RWTH Aachen, 1993.

[7] Sheng Liang. *The Java Native Interface Programmer's Guide and Specification.* Addison–Wesley, Reading, Massachusetts, 1999.

[8] Robert McMillan. Microsoft's j/direct called death of java, 1997. http://www.javaworld.com/javaworld/jw-07-1997/jw-07-jdirect.html.