Jeannie: Granting Java Native Interface Developers Their Wishes

Martin Hirzel
IBM Watson Research Center
hirzel@us.ibm.com

Robert Grimm
New York University
rgrimm@cs.nyu.edu

Abstract

Higher-level languages interface with lower-level languages such as C to access platform functionality, reuse legacy libraries, or improve performance. This raises the issue of how to best integrate different languages while also reconciling productivity, safety, portability, and efficiency. This paper presents Jeannie, a new language design for integrating Java with C. In Jeannie, both Java and C code are nested within each other in the same file and compile down to JNI, the Java platform's standard foreign function interface. By combining the two languages' syntax and semantics, Jeannie eliminates verbose boiler-plate code, enables static error detection across the language boundary, and simplifies dynamic resource management. We describe the Jeannie language and its compiler, while also highlighting lessons from composing two mature programming languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.12 [*Software Engineering*]: Interoperability; D.3.4 [*Programming Languages*]: Processors

General Terms design, languages

Keywords programming language composition, Java, C, JNI, foreign function interface, xtc, modular syntax, *Rats!*

1. Introduction

Higher-level languages must interface with lower-level languages, typically C, to access platform functionality, reuse legacy libraries, and improve efficiency. For example, most Java programs execute native code, since several methods of even class `Object` at the root of Java's class hierarchy are written in C. Foreign-function interfaces (FFIs) accomplish this task, providing access to C code and data from higher-level languages and vice versa. For example, a Java project can reuse a high-performance C library for binary decision diagrams (BDDs) through the Java Native Interface [38] (JNI), which is the standard FFI for Java.

FFI designs aim for productivity, safety, portability, and efficiency. Unfortunately, these goals are often at odds. For instance, Sun's original FFI for Java, the Native Method Interface [52, 53] (NMI), directly exposed Java objects as C structs and thus provided simple and fast access to object fields. However, this is unsafe, since C code can violate Java types, notably by storing an object of incompatible type in a field. Furthermore, it constrains garbage collectors and just-in-time compilers, since changes to the data representation may break native code. In fact, NMI required a conservative garbage collector because direct field access prevents the Java virtual machine (JVM) from tracking references in native code. Finally, native code that depends on a virtual machine's object layout is hard to port to other JVMs.

In contrast to the Native Method Interface, the Java Native Interface is well encapsulated. As a result, C code is easily portable while also permitting efficient and differing Java virtual machine implementations. But JNI's reflection-like API requires verbose boiler-plate code, which reduces productivity when writing and maintaining native code. Furthermore, it is still unsafe—typing errors cannot be checked statically, nor does the JNI specification require dynamic checks. It is also less efficient than unportable FFIs that expose language implementation details.

This paper presents Jeannie, a new FFI for Java that provides higher productivity and safety than JNI without compromising the latter's portability. Jeannie achieves this by combining Java and C into one language that nests program fragments written in either language within each other. Consequently, Jeannie inherits most of the syntax and semantics from the two languages, with extensions designed to seamlessly bridge between the two. By analyzing both Java and C together, the Jeannie compiler can produce high-quality error messages that prevent many a maintenance nightmare. Our compiler is implemented using *Rats!* [29], a parser generator supporting modular syntax, and xtc [28], a language composition framework. It accepts the complete Jeannie lan-

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.

```
package my.net;
import java.io.IOException;

public class Socket {
  static {
    System.loadLibrary("Network");
  }
  protected int native_fd;
  protected native int available() throws IOException;
  …
}
```

```
#include <jni.h>
#include <sys/ioctl.h>          1
#include <errno.h>
#include <string.h>

jint
Java_my_net_Socket_available(JniEnv *env, jobject this)
{                                                       2
  int num, fd;
  jclass cls; jfieldID fid;                             3
  cls = (*env)->GetObjectClass(env, this);
  fid = (*env)->GetFieldID(env, cls, "native_fd", "I");
  fd  = (*env)->GetIntField(env, this, fid);
  if (ioctl(fd, FIONREAD, &num) != 0) {
    jclass exc =                                        4
      (*env)->FindClass(env, "Ljava/io/IOException;");
    (*env)->ThrowNew(env, exc, strerror(errno) );       5
    return 0;
  }
  return num;
}
```

Figure 1. JNI example, with Java on top and C on bottom.

guage, including Java 1.4 code and most gcc extensions to C, and its backend produces conventional JNI code. Future work will explore alternative backends that, for example, produce code optimized for specific JVMs.

Jeannie's contributions are threefold:

- Improved productivity and safety for JNI;
- Deep interoperability of full programming languages;
- Language composition lessons from a large example.

We evaluate Jeannie based on a partial port of the JavaBDD library [60] and on micro-benchmarks that exercise new language features. Our results demonstrate that Jeannie code is simpler than the corresponding JNI code, while introducing little performance overhead and maintaining JNI's portability.

2. Point of Departure: JNI

This section provides a flavor of JNI, the Java Native Interface, as described in Liang's book "The Java Native Interface: Programmer's Guide and Specification" [38]. Figure 1 illustrates how a socket abstraction can use JNI to interface with the operating system. The Java class on top declares a native method `available` with no Java body. The method is implemented by the body (2) of C function `Java_my_net_Socket_available` on bottom. A downcall from Java to C looks like any other Java method call. On the other hand, C code uses a reflection-like API for upcalls to Java methods

```
`.C {                                                   1
#include <sys/ioctl.h>
#include <errno.h>
#include <string.h>
}
package my.net;
import java.io.IOException;

public class Socket {
  protected int native_fd;
  protected native int available() throws IOException
  `{                                                    2
    int num;
    int fd = `this.native_fd ;                          3
    if (ioctl(fd, FIONREAD, &num) != 0)
      `throw new                                        4
        IOException( `_newJavaString(strerror(errno)) );  5
    return num;
  }
  …
}
```

Figure 2. Jeannie version of code from Figure 1.

and for Java field accesses. In the example, C Snippet 3 reads the Java field `this.native_fd`.

The JNI interface pointer `env` is the first argument to all C functions implementing Java native methods. The corresponding type `JNIEnv`, declared in `jni.h`, points to a struct of function pointers that are implemented by the Java virtual machine. In the struct, the field access function `GetIntField` used by Snippet 3 of Figure 1 is replicated for several other types, including `GetBooleanField` and `GetObjectField`. JNI also provides similar functions for calling methods. Snippet 4 illustrates JNI's facilities for raising Java exceptions. Since C lacks exception handling facilities, the C code must emulate the abrupt control flow with an explicit `return` statement. JNI also provides functions to check for pending Java exceptions, entering and leaving synchronization monitors, acquiring and releasing Java arrays for direct access in C, and many more. Finally, JNI functions use types such as `jint`, `jclass`, and `jfieldID`, which are also declared in `jni.h`.

Before first calling a native method, a Java virtual machine needs to link the corresponding native library. The `System.loadLibrary` call in the static initializer of class `Socket` in Figure 1 identifies the library's name, which is mapped to a shared object file (`.so` or `.dll`) on the JVM's library path. After loading the object file, the JVM locates the C function implementing `available` by following the name mangling rules of the JNI specification. Repeated calls to `System.loadLibrary` for the same library are ignored, so that the C functions for several Java classes can be included in the same library.

3. Jeannie Overview

Figure 2 shows the JNI code from Figure 1 rewritten in Jeannie. The outer Jeannie code in Figure 2 corresponds

to the Java code in Figure 1. Nested code snippets in both figures that correspond to each other are labeled with the same numbers (1 through 5). The Jeannie file starts with a `.C block of top-level C declarations included from header files (1). It continues like a normal Java file with a package declaration, imports, and one or more top-level Java class or interface declarations. Unlike in Java, native methods in Jeannie have a body (2). The backtick ` toggles to the opposite language, which is C in the case of Snippet 2. Consequently, Java code can be nested in C (Snippets 3 and 4), and C code can be nested in Java (Snippet 5), to arbitrary depth. The JNI code in Figure 1 illustrates that such deep nesting occurs naturally.

Nested Java code in Jeannie does not need to use fully qualified names: where Snippet 4 in Figure 1 refers to `java.io.IOException`, Snippet 4 in Figure 2 uses the simple name `IOException`, which Jeannie resolves using the import declaration. The Jeannie compiler verifies statically that names and types are used consistently across nested Java and C code. It also verifies that `throws` clauses declare all checked exceptions, while extending Java's exception handling discipline to nested C code and automatically releasing resources such as locks and memory on abrupt control flow. As a result, Jeannie can avoid a large class of bugs that JNI code is susceptible to.

Figure 3 illustrates the Jeannie pipeline for compiling a file. `Socket.jni` corresponds to Jeannie source code as shown in Figure 2. The pipeline first injects

```
#include <jni.h>
```

into the initial `.C block (e.g., Snippet 1 in Figure 2). It then runs the C preprocessor, which is necessary to ensure that all C code in the Jeannie file is syntactically valid and referentially complete. The result is the intermediate file `Socket.jni.i`, which serves as the input to the Jeannie compiler proper.

The Jeannie compiler converts the intermediate file into two separate files `Socket.i` and `Socket.java`, with pure preprocessed C code that uses JNI on the one hand and pure Java code with native methods and a `System.loadLibrary` call on the other hand. Jeannie's parser is generated by *Rats!* [29] from a grammar that reuses separate C and Java grammars. The name and type analysis is based on xtc [28] and reuses separate C and Java semantic analyzers. If the input program has syntactic or semantic errors, the compiler reports them to the user. Otherwise, the code generator produces two separate abstract syntax trees (ASTs) for C and Java, which are converted back into source code by pretty-printers from xtc.

From this point on, the compilation pipeline proceeds as if the JNI code had been written by hand. The platform C compiler compiles `Socket.i` together with other C files into a shared object file `Network.dll`, and the Java compiler compiles `Socket.java` together with other Java files into Java bytecode, which it may place into a file `Network.jar`.

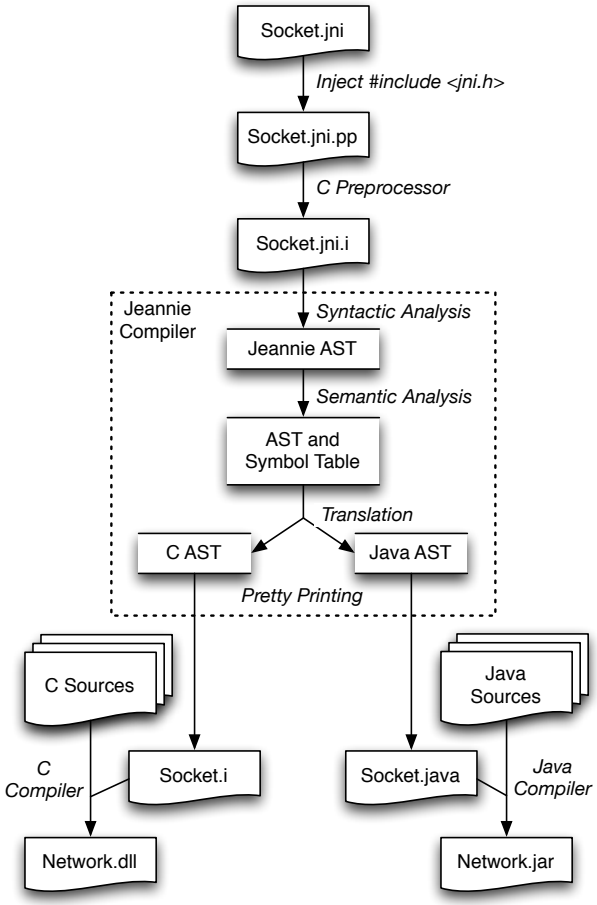

Figure 3. The Jeannie build process and compiler stages.

Later, the JVM's dynamic linker combines the code into a running application. In general, the `Network.dll` and `Network.jar` files are not self-contained, but interact with significant other code written independently of Jeannie, such as the operating system kernel or the Java standard libraries.

4. Jeannie Language

Jeannie combines Java and C. It inherits most of the syntax and semantics from the two base languages [27, 34]. The primary contribution of this paper is to show how to integrate the two languages generally and seamlessly.

4.1 Syntax

Jeannie nests C code in Java and Java code in C. It supports the complete syntax of both C and Java and relies on the *backtick operator* ` to switch between the two languages. For example, when the current language is Java,

```
`{ int x=42; printf("%d", x); }
```

denotes a C block nested in Java code. Conversely, when the current language is C,

```
`{ int x=42; System.out.print(x); }
```

denotes a Java block nested in C code. The backtick operator can also be applied to expressions; it has the same precedence as other unary prefix operators such as logical negation `!`. For example, when the current language is Java,

```
`((jboolean)feof(stdin))
```

denotes a C expression nested in Java code. In this example, the parentheses around the C cast expression are necessary because the cast operator has lower precedence than the backtick operator.

Jeannie also supports a qualified form of the backtick operator for switching between the two languages; depending on the destination language, it is denoted `` `.C `` or `` `.Java ``. The simple and qualified forms are interchangeable, allowing developers to trade off terseness and clarity as desired.

Synchronization

In Java, the `synchronized` (*m*) { *CS* } statement acquires a monitor *m*, then executes the code in the critical section *CS*, then releases *m*. Java semantics are designed so that the monitor is always released, even in the presence of abrupt control flow due to, for example, exceptions. However, since C lacks language support for synchronization, JNI provides two separate functions `MonitorEnter` and `MonitorExit`. It is the programmer's responsibility to ensure the typestate properties of a monitor object along all control paths for JNI code.

Jeannie leverages Java's monitor discipline by allowing `synchronized` statements to be nested in C code with a backtick. For example,

```
`synchronized (lock) { counter.incr(); }
```

denotes a Java `synchronized` statement nested in C code. The monitor is the object in variable `lock` and the critical section contains a method call `counter.incr()`. Of course, the critical section can also use a backtick to switch back into C. For example,

```
`synchronized(flock) `{ fprintf(fd, s); }
```

denotes a `synchronized` statement whose body is a C block.

Exception Handling

In Java, a `try`/`catch`/`finally` statement first executes the `try` block. If the `try` block abruptly terminates with an exception and there is a matching `catch` block, it then executes that exception handler. Either way, it then executes the `finally` block (if present). In contrast, C does not support exceptions. Instead, programmers rely on error codes and explicit control transfer through `goto` and `return`.

Jeannie leverages Java's exception handling discipline by allowing Java `try` statements to be nested in C code with a backtick. For example,

```
`try{ f(); } catch (Exception e) { h(e); }
```

denotes a Java `try` statement nested in C code. Of course, any of the `try`, `catch`, and `finally` blocks can use a backtick to switch back into C. C code can also throw an exception by using a Java `throw` statement with a backtick. For example,

```
`throw new Exception("boo");
```

tries to startle the program by throwing an exception from C code. To complete integration of Java's exception handling discipline with C, Jeannie also allows C functions to declare exceptions with a `` `throws `` clause. For example,

```
void
printf(char* f, ...) `throws IOException;
```

declares a C function `printf` that may signal a Java `IOException` (instead of returning a status code).

String and Array Access

The Jeannie syntax presented so far already supports C code accessing Java arrays and strings. For example, when the current language is C,

```
cChar = `javaString.charAt(3);
cFloat = `javaArray[3];
`(javaArray[3] = `sqrt(cFloat));
```

reads a character from a Java string, reads a floating point number from a Java array, and writes that number's square root back into the Java array. While succinct, this idiom for accessing arrays and strings may be inefficient. For example,

```
for (jint i=0, n=`ja.length; i<n; i++)
  s += `ja[`i];
```

repeatedly crosses language boundaries as it sums up the elements of Java array `ja`.

To speed up bulk access to strings and arrays, Jeannie provides the `_copyFromJava` and `_copyToJava` built-ins. Comparable to C's `memcpy` and Java's `System.arrayCopy`, the two functions copy regions of strings and arrays, but across languages. For example,

```
_copyFromJava(ca, cStart, ja, jStart, len)
```

copies the array elements `ja[jStart]` through `ja[jStart+len-1]` to `ca` starting at `ca[cStart]`. Java strings are treated as read-only character arrays and can be copied in their native UTF-16 encoding or converted into UTF-8 encoding. To let C code allocate an appropriately sized buffer, the `_getUTF8Length` built-in returns the length of a Java string in UTF-8 encoding. Furthermore, to convert C strings into Java strings, the `_newJavaString` built-in illustrated in Figure 2 creates a new Java string from a C string. Consistent with C99's practice for evolving the language, Jeannie prefixes new C built-ins and keywords with an underscore, thus reducing the likelihood of naming conflicts with existing code. The `jeannie.h` header file provides convenience macros such as

Example Code	Nonterminal	Parsing Expression
`` `.C { #include <stdio.h> } class A {} ``	*File*	= `` `.C { C.Declarations } Java.File ``
C nested in Java	Modifications to Java grammar	
`` `{ int x=42; printf("%d", x); } ``	*Java.Block*	+= ... / *CInJava C.Block*
`` `((jboolean)feof(stdin)) ``	*Java.UnaryExpression*	+= ... / *CInJava C.UnaryExpression*
`` `.C ``	*CInJava*	= `` `.C / ` ``
Java nested in C	Modifications to C grammar	
`` `{ int x=42; System.out.print(x); } ``	*C.Block*	+= ... / *JavaInC Java.Block*
`` `new HashMap(); ``	*C.UnaryExpression*	+= ... / *JavaInC Java.UnaryExpression*
`` `java.util.Map ``	*C.TypeSpecifier*	+= ... / *JavaInC Java.QualifiedIdentifier*
`` f(char *s) `throws IOException ``	*C.FunctionDeclarator*	:= *C.DirectDeclarator* (*C.ParameterDeclaration*) ⟨ *JavaInC Java.ThrowsClause* ⟩?
	C.Statement	+= ...
`` `synchronized(act) { act.deposit(); } ``		/ *JavaInC Java.SynchronizedStatement*
`` `try { f(); } catch (Exception e) { h(e); } ``		/ *JavaInC Java.TryStatement*
`` `throw new Exception("boo"); ``		/ *JavaInC Java.ThrowStatement*
`` `.Java ``	*JavaInC*	= `` `.Java / ` ``
New C constructs	Modifications to C grammar	
	C.Statement	+= ...
`_with (jint* ca = `ja) { sendMsg(ca); }`		/ `_with` (*WithInitializer*) *C.Block*
`_abort ca;`		/ `_abort` *C.Identifier* ;
`_commit ca;`		/ `_commit` *C.Identifier* ;
	WithInitializer	=
`msg->data = `ja`		*C.AssignmentExpression*
`jint* ca = `v.toArray()`		/ *C.Declaration*

Figure 4. Jeannie syntax. For exposition, parsing expressions are somewhat simplified when compared to the actual grammar. Literals are set in monospace font, without double quotes.

```
#define copyFromJava _copyFromJava
```

to eliminate unsightly underscores from most programs.

To avoid manual buffer management and enable optimizations that avoid copying altogether, Jeannie's `_with` statement provides a more disciplined way of accessing Java strings and arrays from C code. For example,

```
_with (jint* ca = `ja) {
  for (jint i=0, n=`ja.length; i<n; i++)
    s += ca[i];
}
```

acquires a copy of Java array `ja`'s contents, sums up its elements, and then releases the copy while also copying back the contents. Following the example of Java's `synchronized` statement, Jeannie's `_with` statement uses syntactic nesting to enforce the proper pairing of acquire and release operations on a resource. By default, the string or array is released when control reaches the end of a `_with` block; Jeannie also supports leaving the statement abruptly by using a `_commit ca` or `_abort ca` statement.

Formal Syntax

Figure 4 shows the Jeannie grammar. The first column shows example code for each construct, while the other columns specify the syntax using parsing expressions and the grammar modification facilities of *Rats!* [29]. For example,

Java.Block += ... / *CInJava C.Block*

modifies the Java grammar: the nonterminal *Java.Block*, in addition (+=) to recognizing Java blocks (...) now recognizes a backtick (*CInJava*) followed by a C block (*C.Block*). As another example, the rule

C.FunctionDeclarator := *C.DirectDeclarator* (*C.ParameterDeclaration*) ⟨ *JavaInC Java.ThrowsClause* ⟩?

modifies the C grammar: the nonterminal *C.FunctionDeclarator*, instead of (:=) recognizing just a C function declarator, now recognizes a C function declarator followed by an optional backtick and Java `throws` clause.

The Jeannie grammar's start symbol is *File*. As illustrated in Figure 2, a Jeannie source file starts with C declarations, which typically come from header files, followed by the usual contents of a Java source file, i.e., the package, import, and top-level class and interface declarations.

One modification to the C grammar not yet discussed is the ability to use a backticked Java type name as a C type specifier. For example, when the current language is C,

```
const `java.util.Map m = ...;
```

defines a C variable `m` that contains a constant opaque reference to a Java `Map`. As illustrated by this example, C code may not only use Java types but also combine them with C qualifiers such as `const`.

Most of Jeannie's syntax has been designed for orthogonality. By adding only a small number of rules to existing C

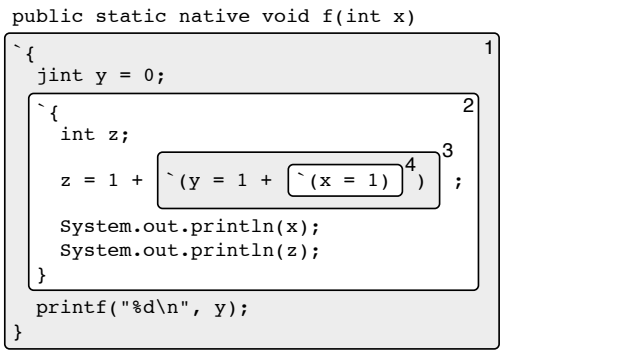

Figure 5. Example code with deep nesting and assignments.

and Java grammars, it enables the seamless integration of the two languages. However, Jeannie's syntax does trade orthogonality for efficiency for string and array accesses. Furthermore, since JNI targets C and Jeannie includes the complete C language, it is possible to use JNI within Jeannie code. While generally discouraged, this does allow for incremental ports of JNI code to Jeannie.

4.2 Dynamic Semantics

For pure Java and pure C code, the dynamic semantics of Jeannie are just the dynamic semantics of the respective base language. This section clarifies the behavior of variables, arrays, and abrupt control flow at the language boundary.

Formal Parameters and Local Variables

Consider the Jeannie code in Figure 5. Block 1 is a C block. It can use the local variable `y` and an implicit parameter `JNIEnv* env` for direct JNI calls. Block 2 is a Java block. It can use the formal parameter `x`, the local variable `z`, and an implicit parameter `T this`, with `T` being the class that declares method `f`. Expression 3 is a C expression in the scope of Block 1 and can hence use `y` and `env`. Expression 4 is a Java expression in the scope of Block 2 and can hence use `x`, `z`, and `this`.

Consistent with the rule of least surprise, all Jeannie code in the same activation of a method observes the same state. Consequently, in Block 2 of Figure 5, the statement

```
System.out.println(x);
```

prints `1`, since the preceding Expression 4 assigns that value to `x`. Likewise, in Block 1, the statement

```
printf("%d\n", y);
```

prints 2, since the preceding Expression 3 assigns that value to `y`.

The Jeannie language does not specify the dynamic semantics of unobserved state. For example, the C code in Figure 5 never applies the address-of operator `&` to `y`. Consequently, an implementation need not store `y` at the same address for Block 1 and Expression 3. Likewise, the Java code never uses the implicit parameter `this`. Consequently, an implementation need not pass it around. By only defining the runtime semantics of observed state, Jeannie enables different implementation and optimization strategies. Notably, a backend more sophisticated than our current one may perform inlining and register allocation across language boundaries similar to [51].

Non-Local Variables and Garbage Collection

JNI distinguishes between local and global references. Local references are formal parameters and local variables in C that refer to Java objects. They pose little challenge for garbage collection: a Java virtual machine simply treats references passed into C code as additional roots for the duration of a native method call. Global references are static variables and heap-allocated data structures in C that refer to Java objects. By definition, static variables and heap-allocated data structures may hold a Java reference even after a native method returns, thus raising the issue of how to prevent the referenced objects' premature collection.

JNI addresses this problem by requiring explicit acquire and release operations through `NewGlobalRef` and `DeleteGlobalRef`. The JVM's garbage collector then treats these references as additional roots. However, this approach suffers from all the problems of manual memory management: leaks, dangling references, and double deletes. The alternative of using syntactic nesting to enforce the pairing of acquire and release operations does not work for global references. After all, they are intended to escape a language's stack discipline.

Instead, Jeannie recommends that programmers do *not* store Java references in C's static variables or heap-allocated data structures. Rather, they should only store them in fields of Java classes and objects. For example, to access a static field `f` of class `C`, a developer merely writes `` `C.f ``, and, to access an instance field `g` of object `o`, the developer writes `` `o.g ``. In either case, a Java reference is only a backtick away, while taking full advantage of garbage collection. Future work may explore an escape analysis that warns users when the compiler cannot prove that local references indeed stay local.

String and Array Access

Figure 6 shows how programmers should choose between Jeannie's different string and array access features. While the expression semantics of regular Java capture the dynamic semantics of backticked string or array accesses, the `_copyFromJava`, `_copyToJava`, `_newJavaString`, and `_with` constructs merit discussion.

The `_copyFromJava` and `_copyToJava` built-ins perform their copy operation unless the Java reference is `null` or the Java indices are out of bounds; in those cases, the built-ins throw an appropriate exception. In contrast and consistent with C semantics, an invalid C pointer or index produces undefined behavior. Trying to copy a string *to* Java is a static error. When copying a string *from* Java, the pointer can

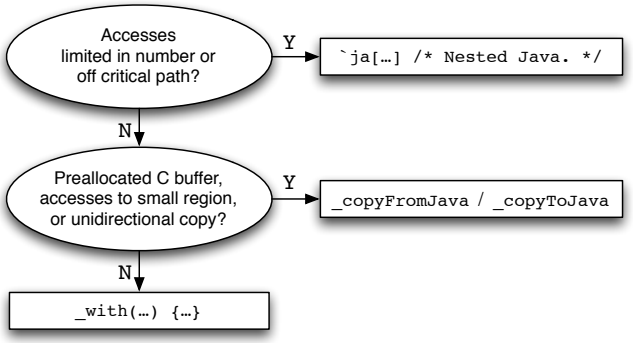

Figure 6. Decision diagram for how to access arrays and strings (modeled after similar diagrams in the JNI specification [38, §3.2.7, §3.3.4]).

```
public static native int
replace(char[] chars, char oldC, char newC)
`{
  jchar old = `oldC , new = `newC ;

  jint  len = `chars.length ;

  _with (jchar* s = `chars ) {
    for (int i=0; i<len; i++) {
      if (old == s[i]) {
        s[i] = new;
        return i;
      }
    }
    _abort s;
  }
  return -1;
}
```

Figure 7. Example code for `_with` statement. Note that `new` is not a keyword in C code even though it is for Java.

be of type `jbyte*` or `jchar*`. In the former case, the result is a UTF-8 encoded C array; in the latter case, it is UTF-16 encoded just like Java strings. In both cases, the length argument counts Java characters in the Java string, while the return value counts C characters in the C string. Comparable to `_copyFromJava`, the `_newJavaString` built-in accepts a C string of type `jbyte*` or `jchar*`, converting from UTF-8 to UTF-16 for the former.

As shown in Figure 4, the `_with` statement takes the following form:

```
_with (ca = ja) { ... }
```

If *ja* is `null`, the `_with` statement signals a `NullPointerException`. Otherwise, it initializes *ca* to point to a copy of *ja*. As for the built-in copy functions, strings are UTF-8 encoded if *ca* is of type `jbyte*` and UTF-16 encoded if *ca* is of type `jchar*`. Independent of encoding, modifying a string leads to undefined behavior.

The `_commit` and `_abort` statements initiate an abrupt control transfer to the code immediately following a `_with` statement. A `_commit` statement copies any changes back into *ja*, whereas `_abort` discards them. For example, in Figure 7, the `_abort` statement indicates that no changes need to be copied back to `ja`. Both `_commit` and `_abort` release any resources necessary for implementing the `_with` statement, notably the copy's memory.

Control flow can also leave a `_with` statement through completion of the block, a `return` statement, or an exception. The first two cases represent an implicit `_commit`, while the third case represents an implicit `_abort`. In all three cases, Jeannie code behaves exactly as if the `_commit` or `_abort` was present, including releasing any resources. For example, in Figure 7, the `return i` statement inside the loop represents an implicit `_commit`, and the modified character array is copied back before control leaves the `_with` statement. Thanks to the explicit `_abort` at the end of the `_with` statement's block, the array is not copied back if no character was replaced.

Abrupt Control Flow

Jeannie has several language features that abruptly change control flow: explicit function or method returns, exceptions, array aborts and commits, and `break`, `continue`, and `goto` statements. To avoid unnecessary complexity, `break`, `continue`, and `goto` statements must not cross language boundaries or `_with` statements. When abrupt control flow leaves `synchronized`, `_with`, or `try` blocks, Jeannie releases any locks or arrays while also executing any `finally` blocks—just as it does for regular control flow. Function and method returns go back to the call site, just as in C and Java, and exceptions propagate to the dynamically closest matching `catch` clause, just as in Java.

4.3 Static Semantics

Similar to Jeannie's dynamic semantics, the static semantics of Jeannie are largely just the static semantics of the respective base language—with one crucial difference: the Jeannie compiler checks that Jeannie code is well-formed and well-typed for both base languages at the same time and across the language boundary. In other words, Jeannie not only replaces JNI's reflection-like API with direct language support, but also enforces the static semantics of the combined language, thus significantly reducing the potential for software errors.

Most of the compile-time semantic rules for C and Java also apply to nested C and Java code in Jeannie. For instance, Jeannie resolves Java class names relative to imports and matches C functions to their prototypes, which are typically declared in header files included at the beginning of a Jeannie file. Furthermore, Jeannie verifies that all checked exceptions are either caught locally or declared as thrown by the enclosing function or method. Next, Jeannie checks that Java members are in fact accessible, i.e., that references to fields, methods, and member types obey their visibility (`private`, `protected`, `public`, or default).

In Jeannie, native methods of a Java class must have a body and that body must be a backticked C block. Native

methods also declare an implicit C parameter `JNIEnv* env`, so that C code has access to JNI's API. Consequently, explicit parameters of native methods cannot have the name `env`. As discussed above under dynamic semantics, Jeannie provides this feature to facilitate incremental conversion of JNI code to Jeannie, though its use is discouraged in general.

Typing

Jeannie defines several type equivalences between Java and C types, denoted as $JT \equiv CT$. Java's primitive types are equivalent to their corresponding typedefs in `jni.h`. For example:

```
char ≡ jchar
 int ≡ jint
```

Similarly, Java arrays of primitive types are equivalent to their typedefs in `jni.h`, including:

```
char[] ≡ jcharArray
 int[] ≡ jintArray
```

Additionally, Jeannie honors the type equivalences provided by JNI for reference types, such as:

```
    java.lang.Object ≡ jobject
  java.lang.Object[] ≡ jobjectArray
java.lang.Throwable ≡ jthrowable
```

See the JNI specification for a full list [38, §12.1].

Jeannie extends the C type system by introducing an opaque reference type for every Java class or interface. The Java type and C type are equivalent. For example:

```
java.io.IOException ≡ `java.io.IOException
```

Jeannie has the same rules for resolving simple type names to fully qualified type names as Java. For example, C code in Jeannie can use the type specifier `` `IOException `` for class `java.io.IOException` if the current file is part of package `java.io` or if it has the appropriate import declaration.

C assignments, variable initializers, function invocations, and `return` statements can implicitly widen opaque references to Java classes or interfaces. For example, when the current language is C, the second assignment in

```
`java.util.HashMap h = ...;
`java.util.Map     m = h;
```

is legal because class `HashMap` implements interface `Map`.

When a Java expression is nested in C code, Jeannie type-checks the C code as if the Java expression had the equivalent C type. Likewise, when a C expression is nested in Java code, Jeannie type-checks the Java code as if the C expression had the equivalent Java type. However, C pointers, structs, and unions have no equivalent in Java, and a Jeannie compiler flags an error when a program attempts to use them in Java code.

In addition to these type equivalence and widening rules, each base language has its own implicit and explicit conversions, which are observed by the Jeannie compiler. For example, when the current language is Java,

```
if (`((jboolean)feof(stdin))) return;
```

first explicitly casts the return value of the function call `feof(stdin)` to `jboolean` and then relies on Jeannie's type equivalence rules to implicitly convert the C typedef `jboolean` into the Java primitive type `boolean`. That type, in turn, is the expected type for the `if` statement's condition.

Jeannie treats Java reference types in C no more and no less opaquely than JNI. For example, Sun's default `jni.h` header defines `jobject` to be a pointer to an undefined C struct `_jobject`. This prevents mistakes caused by inattentiveness, since dereferencing a variable of type `jobject` leads to a compile-time error. However, it cannot prevent programmers from casting `jobject` to a pointer to another, defined C struct and then accessing the object's contents. In other words, C code in Jeannie is just as weakly typed as any C code, including JNI. At the same time, Jeannie's support for opaque Java references and implicit widening operations within nested C code considerably reduces the need for using explicit and potentially unsafe C casts.

Storage Model

To ensure well-defined semantics, primary identifiers require the context of the declaring language. For example, the following code is malformed:

```
int x = 42;
`f(x); ⇒ error: 'x' used in wrong language
```

More importantly, backticked expressions in Jeannie always return an r-value. For example, the following code is malformed:

```
`x[0] = y; ⇒ error: 'x[0]' not an l-value
```

Such cross-language assignments fundamentally have ambiguous semantics. For instance, assume that the example's language is C and that `x` is a Java array. Should Jeannie just execute the assignment, as pure C would? Or, should Jeannie first perform Java's checks against `null` references and out-of-bounds indices? Additionally, cross-language assignments raise implementation issues. For instance, should the above C assignment execute a write barrier for the Java garbage collector? To avoid these issues, Jeannie requires that operations on l-values occur in the language of the l-value. For instance,

```
`(x[0] = `y);
```

correctly performs the assignment in nested Java code.

String and Array Access

Each of Jeannie's `_copyFromJava`, `_copyToJava`, and `_with` constructs for accessing strings and arrays requires a C pointer *ca* and a reference to a Java string or array *ja*. The reference *ja* has an opaque C type and is the result

of a C expression. In practice, this C expression usually is a backticked Java expression, such as `` `v.toArray() `` or `` `o.toString() ``. If *ja* is a `jstring`, then *ca* must be `jbyte*` for UTF-8 encoding or `jchar*` for UTF-16 encoding. Otherwise, *ja* must be a Java array, such as `jintArray`, and *ca* must be of the corresponding pointer type, such as `jint*`. In contrast to the copy built-ins and the `_with` statement, the `_newJavaString` built-in requires only a C pointer *ca*, which must be `jbyte*` or `jchar*`.

For the two copy built-ins

```
_copyFromJava(ca, tgt, ja, src, len)
_copyToJava  (ja, tgt, ca, src, len)
```

the *tgt* and *src* indices as well as the *len* element count must be of type `jint`. The result of the copy built-ins is also of type `jint` and indicates the number of elements actually copied. We did consider a single `_copy` built-in that relies on parameter ordering to determine whether to copy from or to Java. But unfortunately, C's `memcpy` and Java's `System.arrayCopy` disagree on whether to put the source or target parameter first. As a result, Jeannie programmers might become confused about a single `_copy` construct's correct usage. By providing two separate constructs, the Jeannie compiler can statically determine parameter ordering errors based on argument types. Either way, it rejects attempts to copy into Java strings.

In a `_with` statement

```
_with (ca = ja) { ... }
```

the C pointer *ca* either must be newly declared or must be modifiable, i.e., not `const`. The identifier *ca* in `_commit` *ca* and `_abort` *ca* statements must be the formal of a directly enclosing `_with` statement. It is illegal to jump in to or out of `_with` statements using a `break`, `continue`, or `goto` statement. However, control may leave a `_with` statement through a `return`, exception, or by completing the block.

Abrupt Control Flow

Besides preventing `break`, `continue`, or `goto` statements from crossing a `_with` boundary, Jeannie also prevents them from crossing a language boundary. In contrast, regular function or method returns and thrown exceptions may cross language boundaries. When a C `return` statement returns from a Java method, a Jeannie compiler converts the returned C value to its equivalent Java type and then checks that it conforms to the method's return type. Similarly, when a Java `return` statement returns from a C function, a Jeannie compiler converts the returned Java value to its equivalent C type and then checks that it conforms to the function's return type. A thrown exception must be either (1) unchecked, i.e., a subclass of Java's `RuntimeException` or `Error`, (2) caught locally, or (3) declared as thrown by the enclosing function or method.

5. Pragmatics

This section explores the pragmatics of the Jeannie compiler, focusing on two main issues. The first issue is how to translate Jeannie code to Java and C. The second issue is how to achieve scalable composition [45]. Fundamentally, language composition is only practical if the development effort for realizing the composed language is commensurate with the newly added features and *not* with the entire language. The Jeannie compiler largely achieves this goal and directly reuses grammars, semantic analyzers, and pretty printers for Java 1.4 and C with most gcc extensions. In fact, many of these components predate the Jeannie project and have been developed by different programmers for the different languages.

5.1 Syntactic Analysis

Any real-world code written in C relies on the preprocessor to include header files, resolve conditionals, and expand macros, notably for platform-specific declarations. For example, Sun's default `jni.h` header declares several types such as `jint` and macros such as `JNI_TRUE`, with several declarations conditional on the computer architecture (32-bit or 64-bit) and language (C or C++). Furthermore, before preprocessing, C code need not be well-formed, since the preprocessor considers only tokens and not syntactic units. Consequently, as shown in Figure 3, Jeannie invokes the C preprocessor before parsing the resulting code, and Java code is also subject to preprocessing. To some developers, this may represent a welcome feature; for others, it may lead to undesirable results due to inadvertent name capture. This issue is not unique to Jeannie, as preprocessing poses a challenge for any tool accepting C code. One solution would be to develop a "Jeannie-aware" version of the preprocessor that ignores Java code. A more general solution would be to integrate the preprocessor's features with the language [40].

The actual Jeannie grammar is written for the *Rats!* parser generator [29], which is well-suited to this task for three reasons. First, *Rats!* has a module system, and, as shown in Figure 4, the Jeannie grammar reuses existing C and Java grammars as modules, modifies several productions, and adds a few new ones. Second, *Rats!* is scannerless, i.e., integrates lexical analysis with parsing. Consequently and as illustrated in Figure 7 for `new`, Java tokens and C tokens are only recognized in the respective language's context. In other words, the Jeannie parser can recognize code written without regard for Jeannie's union of Java and C, notably code included from platform-specific header files. Third, *Rats!* has built-in support for parser state, which lets the Jeannie parser resolve ambiguities between `typedef` names and other names in C code without complicating the Java grammar.

Rats! grammars specify not only a language's syntax but also the structure of its abstract syntax tree (AST) through productions marked as `generic`. The corresponding AST

```
public Node getThisDotField(String name) {
  Node v$1 = GNode.create("ThisExpression", null);
  Node v$2 = GNode.
    create("SelectionExpression", v$1, name);
  return v$2;
}
```

Figure 8. Example method created by xtc's concrete syntax tool.

nodes are instances of a generic node class `GNode`. The generic node's name is the unqualified nonterminal on the left-hand of a production, though individual alternatives can override this default. The generic node's children are the values of non-void expressions on the right-hand side. Since the Jeannie grammar reuses existing Java and C grammars, it also reuses their AST declarations. However, since several productions in the Java and C grammars have the same names, the Jeannie AST may also contain generic nodes with the same name but different semantics. As a result, semantic analysis needs to track the current backtick nesting depth during AST traversal.

Discussion

Thanks to *Rats!*, the development of Jeannie's syntactic analysis phase was largely straightforward. However, we did encounter two issues when trying to write code processing the resulting ASTs; the solutions to both problems, in turn, depend on *Rats!* and are useful beyond Jeannie. First, a grammar is a rather circuitous starting point for understanding a language's AST, since the grammar contains many expressions that do not contribute to the AST such as punctuation and layout. To address this issue, we modified *Rats!* to automatically generate the necessary documentation: given the corresponding command line flag, it loads a grammar, removes any expression that does not contribute to the AST, and then pretty prints the reduced grammar, optionally as hyperlinked HTML.

Second, Jeannie's translation phase needs to create new AST fragments for Java and C from scratch, but doing so programmatically in handwritten code is tedious and error-prone. To address this issue, we added support for concrete syntax [14] to xtc. Our concrete syntax tool builds on variants of the C and Java grammars created with *Rats!*' module system; both grammars recognize individual declarations, statements, or expressions with embedded pattern variables. The tool converts the corresponding AST into Java code that recreates the AST, while also replacing the AST nodes of pattern variables with Java variables. For example, our tool translates the Java code template

```
getThisDotField { this.#name }
```

into the method shown in Figure 8, which takes a parameter `name` representing the template's only pattern variable and returns a generic node representing the field access expression.

5.2 Semantic Analysis

Comparable to the syntactic analysis phase, the Jeannie compiler's semantic analysis phase builds on separate semantic analysis phases for Java and C. Each author separately developed one of the semantic analyzers, with little coordination on their internals. At the same time, both semantic analyzers share facilities for (1) traversing abstract syntax trees through visitors, (2) tracking identifiers through a symbol table, and (3) representing types and their annotations. Before discussing the composition of the two semantic analysis phases, we explore the common facilities. They are provided by the xtc (eXTensible C) toolkit [28] and also used by other source-to-source transformation tools including *Rats!* [29] and C4 [22, 61].

Visitors

The visitor design pattern enables type-safe tree traversal through double dispatch [24]. Visitors implement a common interface `V` that has a `visit(N)` method for every distinctly typed node `N`, and nodes implement an `accept(V)` method that invokes the visitor on the node. While effective, we have also found the visitor design pattern to be limiting. Notably, it does not support visit methods that accept a superclass of several nodes, thus leading to code duplication when processing several types of nodes in the same way. Furthermore, it cannot dispatch on properties of a node, in particular, the names of generic nodes created by inline AST declarations for *Rats!*. Finally, changes to an AST's structure tend to ripple through the entire code base, since the visitor design pattern requires changing the common interface `V` and thus all classes that implement `V`.

xtc addresses these concerns by providing a more expressive alternative through dynamic visitor dispatch [9]. Under this model, the appropriate visit method is dynamically selected and invoked through Java reflection; method resolutions are cached to reduce the overhead of future dispatches. Intuitively, for a given node `N`, xtc's visitor dispatch selects the visit method with the closest supertype `N'` as its only argument (which includes `N` itself). In practice, xtc's visitor dispatch also considers interfaces, starting with the interfaces implemented by the node's class. For generic nodes, it precedes type-based resolution with a name-based resolution step, seeking a method `visit`*Name*`(GNode)` for generic nodes with name *Name*. While xtc's visitor dispatch provides a simple solution that meets our needs, it does eschew type safety for expressivity; other efforts have explored different trade-offs between expressivity, safety, and complexity [30, 42, 58].

Symbol Table

xtc's symbol table maps identifiers to types, organizes the mapping into hierarchically nested scopes, partitions each

```
public void visit(Node n) {
  table.enter(n);
  for (Object o : n)
    if (o instanceof Node)
      dispatch((Node)o);
  table.exit(n);
}
```

Figure 9. Example code for synchronizing generic AST traversal with the current symbol table scope.

scope amongst different namespaces, and supports the obvious `define`, `isDefined`, and `lookup` operations. It also tracks the current scope and supports both absolute and relative name resolution. Probably its most interesting feature is support for synchronizing AST traversal with the current scope. When creating a new scope, a visitor invokes the symbol table's `mark` method on each node corresponding to the new scope. That method uses `Node`'s support for arbitrary properties to annotate the node with the scope's identity. Before and after processing a node, other visitors then call the symbol table's `enter` and `exit` methods, which rely on the node's annotations to update the current scope. When combined with xtc's dynamic visitor dispatch, this feature supports generic AST traversal that is also synchronized with the symbol table.

For example, the C analyzer verifies labels, which may be used before they are defined, in two passes. During the first pass, it collects all label definitions in the symbol table. During the second pass, it checks that each label appearing in a `goto` statement or label address expression has been defined. (The latter construct is a gcc extension to access a label's address as a value.) The second pass is implemented as a visitor with only *three* methods. The first two methods visit `goto` statements and label address expressions, checking that each label is defined. Figure 9 shows the third method, which visits all other AST nodes. It simply enters the node's scope, iterates over its children, dispatching the visitor on children that are also nodes, and finally exits the node's scope again.

Type Representation

xtc represents typing information for Java and C in a single, unified class hierarchy, whose common interface is defined by the base class `Type`. Leafs of the hierarchy correspond to distinct kinds of types, and intermediate classes capture major categories of types. For example, both `IntegerT` and `FloatT` are subclasses of `NumberT`, while both `ArrayT` and `PointerT` are subclasses of `DerivedT`. In addition to capturing each type's inherent properties, such as the integral kind for `IntegerT` or the pointed-to type for `PointerT`, every type instance can have several annotations. They capture a type's source language and location, scope, compile-time constant value, memory shape, and other attributes including C qualifiers such as `const` and Java modifiers such as

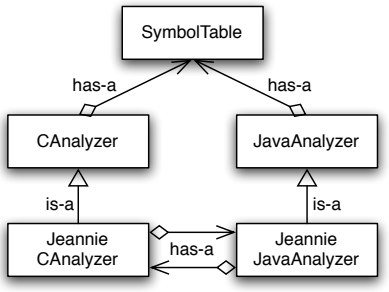

Figure 10. Class diagram for Jeannie's semantic analyzer.

`public`. The common interface defined by `Type` provides methods to

- determine a type's kind such as `isClass`,
- convert to a particular kind such as `toFunction`,
- read annotations such as `hasScope` and `getConstant`,
- add annotations such as `attribute`.

The base class `Type` also defines a Java enum over all type kinds; it is accessible through `tag` to facilitate efficient case analysis with Java `switch` statements.

The overall hierarchy of types comprises one base class, six intermediate classes, and 21 leaf classes, representing the relative richness of the combined type systems of Java and C. To make this information manageable, the hierarchy of types is supplemented by language-specific classes whose methods directly capture the corresponding language standard's instructions such as "The type of each of the operands of a multiplicative operator must be a primitive numeric type, or a compile-time error occurs" [27, §15.17] or "If both the operands have arithmetic type, the usual arithmetic conversions are performed" [34, §6.5.8]. The hierarchy of types is also supplemented by a hierarchy of references, which represent the shapes of l-values' memory regions, are used by the C analyzer to track compile-time constants even across (de)reference operations, and are based on the corresponding feature of CIL [43].

Combined Semantic Analyzer

With the semantic analyzers for Java and C relying on the same visitor framework, symbol table, and type representation hierarchy (but otherwise having independent internal structures), the composition of the two analyzers is relatively straightforward. As illustrated in Figure 10, the Jeannie compiler's semantic analysis phase builds on subclasses of the two analyzers, with each subclass implementing additional visit methods for that particular language's extensions. Both visitors reference each other and, on encountering a backtick operator, dispatch the other visitor on the backticked construct. Both visitors also reference the same symbol table, thus ensuring that mappings between identifiers and types are shared and that either visitor correctly enters and exits scopes while traversing the AST. Finally, both visitors di-

rectly reference types for the other language, since all types are represented as subclasses of the same base class `Type`.

For example, when processing Jeannie's opaque reference types, that is, Java types appearing in C's type specifiers, the extended analyzer for C defers to the extended analyzer for Java to determine the appropriate Java type representation. It then continues processing the variable declaration with the original, non-extended code. Associating a variable with a Java type in the C analyzer does not result in a typing error—as long as the variable is only referenced in constructs that support Java types and that have been appropriately extended in the Jeannie analyzer, notably initializers and assignments.

To process C's function declarators, the extended analyzer for C overrides the `visitFunctionDeclarator` method. The extended version first delegates to the original method, which ignores the generic node's extra child, and then processes the `throws` clause if present. The type representation for C functions `FunctionT` can seamlessly store the extra information, since its superclass `FunctionOrMethodT` already captures the union of Java methods and C functions, i.e., a return type, name, list of parameter types, and list of exception types.

Discussion

We did encounter one obstacle while combining the two semantic analyzers. The C analyzer relies on a nested class to factor out the state machine necessary for processing C's declaration specifiers such as `const` or `int`. The Jeannie compiler extends this nested visitor with support for Java types. But, as originally written, the C analyzer created instances of this class through a `new` expression—thus preventing substitution with the extended version. We addressed this issue by introducing an explicit (and overridable) factory method. Our fix certainly illustrates the utility of the factory design pattern; though the issue itself could be avoided with dedicated language support for composition [45].

While the composition itself raised only one, relatively minor, concern, developing the shared facilities and semantic analyzers themselves has proven to be a little more challenging. In particular, we encountered three significant issues. First, while Java's and C's primitive types are seemingly similar, they have subtle differences. Notably, C treats the boolean type `_Bool` introduced in C99 as an integer, while Java does not treat its boolean type `boolean` as an integer. As a result, `xtc`'s type framework must distinguish between the two languages when testing for integral or arithmetic types and when performing integral promotion. We originally tried including these operations in the common interface for `Type`, but then factored them into separate classes that provide language-specific operations without cluttering `Type`'s interface. Following the lesson from the previous paragraph, the language-specific operations are implemented by instance methods and the corresponding classes are accessed through factory methods, with the result that operations can be easily overridden.

Second, C's integers have only relative rank restrictions, while Java's integers have fixed in-memory representations. At the same time, either analyzer must contain an interpreter for a substantial subset of the language's expression syntax to track compile-time constant values. Moreover, the Jeannie analyzer must be able to map between Java and C integers. `xtc`'s type framework addresses these concerns by including (1) a class capturing the local platform's type limits and (2) support for performing arithmetic operations on compile-time constants independent of the underlying representation. The class representing local type limits can easily be regenerated by compiling and running a simple C program. In our tests of 32-bit Mac OS X, Linux, and Windows systems, the limits are the same.

Third, while C translation units must be self-contained, i.e., must incorporate declarations for all referenced types and variables after preprocessing, Java compilation units may reference types and variables in other source and binary files. While resolution to external classes is lazy, i.e., performed only when encountering an appropriate name, it still requires that the Java analyzer be able to locate source as well as binary files and determine their type signatures. For source files, the Java analyzer parses each file and then performs a lightweight analysis to determine only the signature. For binary files, our implementation can fortunately avoid most of the complexity of resolving and inspecting class files because it is also written in Java and can use reflection. Jeannie's analyzer leverages these facilities to resolve three code formats: Jeannie source, Java source, and Java binary.

5.3 Translation

As described in Section 4, the Jeannie language has been carefully designed to allow for different implementations. For this paper, we built a compiler that translates Jeannie code into fairly straightforward JNI code. Consequently, code generated by our Jeannie compiler has the same portability properties as handwritten JNI code: it makes no assumptions about the implementation of the Java virtual machine.

The translator converts a Jeannie AST into two separate ASTs, one for Java and one for C. As illustrated in Figure 3, the input Jeannie AST combined with a symbol table holds the results of semantic analysis. The output Java and C ASTs are converted to source code using `xtc` pretty printers, which are implemented as visitors. The code generator itself also is a visitor. In the spirit of scalable composition, it only has explicit visit methods for a small subset of AST nodes, using a catch-all visit method similar to the one shown in Figure 9 for processing intermediate AST nodes unaffected by Jeannie semantics.

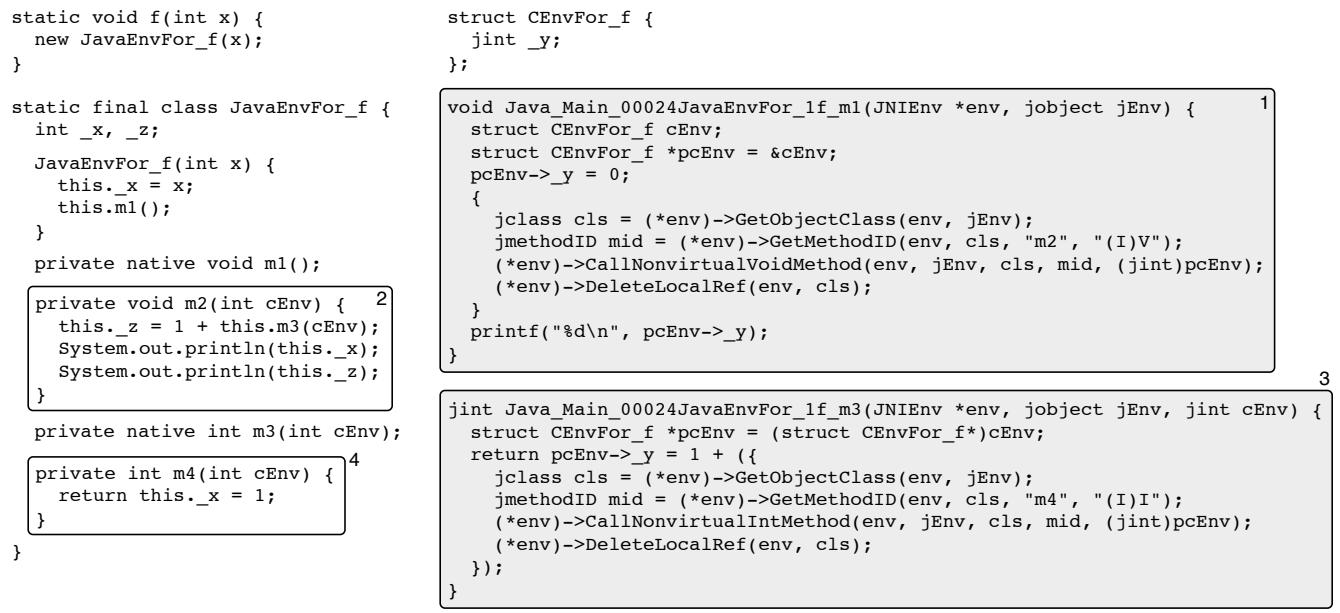

Figure 11. Jeannie-generated Java code (left) and C code (right) for the example in Figure 5, omitting abrupt control flow.

Environments

Consider the Jeannie code example from Figure 5. Snippets 1 and 3 are C code, whereas Snippets 2 and 4 are Java code. The translator turns each snippet into a method as shown in Figure 11: native methods `m1` and `m3` for the C snippets and Java methods `m2` and `m4` for the Java snippets. Snippets 1 and 2 are blocks, so the corresponding methods `m1` and `m2` return `void`. Snippets 3 and 4 are expressions, so the corresponding methods `m3` and `m4` return non-void values. The formal `JNIEnv* env` for the C functions implementing native methods is the JNI interface pointer.

In our translation scheme, nested snippets are executed through method calls. For example, Java Snippet 2 in Figure 5 contains C Snippet 3, and, consequently, Java method `m2` in Figure 11 calls native method `m3`. Likewise, C Snippet 3 contains Java Snippet 4, and the C function implementing native method `m3` calls Java method `m4` through JNI. Since JNI's method invocation API requires a class pointer and method ID, the translator isolates the corresponding declarations and actual method invocation in a statement expression. A statement expression `({ ... })` is a gcc extension to C that introduces a new scope within an expression; its value is the result of executing the last statement.

Consistent with Jeannie's semantics, C Snippets 1 and 3 need to share the same dynamic C state, and Snippets 2 and 4 need to share the same dynamic Java state. For example, Snippet 4 modifies Java variable `x`, and Snippet 2 observes that state in the `System.out.println(x)` statement. Our translator addresses this issue by reifying each base language's dynamic state through explicit environments.

The Java environment is implemented as a member class that is a sibling of the original method and allocated on the heap on calls to that method. For example, in Figure 11, class `JavaEnvFor_f` contains fields `_x` and `_z`. Field `_x` corresponds to the formal parameter `x` of `f` and is initialized in the constructor. Field `_z` corresponds to the local variable `z` and is initialized in Snippet 2. The environment's instance methods are the methods synthesized for nested snippets, which ensures that all methods have access to the environment's state through the implicit `this` for Java code and an explicit `jobject jEnv` for C code. The environment class itself is static for static methods and non-static for instance methods, thus providing access to the enclosing instance only for instance methods, which is the desired behavior.

The C environment is implemented as a global C struct and allocated on the stack in the function implementing the outermost C block. For example, `struct CEnvFor_f` in Figure 11 contains field `_y`, which corresponds to the local variable `y` of Snippet 1 in Figure 5. A pointer to the C environment is passed to all methods implementing nested code snippets. Since Java does not support pointer types, the corresponding parameter is declared to be a Java integer and, as illustrated for `m3`, cast back to an actual pointer in C code. Note that the example code uses a 32-bit Java `int` because the targeted architecture is 32-bit.

Of course, the translator avoids name clashes of synthesized methods and fields with members of existing classes by generating names distinct from any names free in the body of the original Jeannie method.

String and Array Access

The translation of the `_copyFromJava` and `_copyToJava` built-in functions is straightforward. It simply uses one of JNI's copy functions such as `GetStringUTFRegion` or `SetIntArrayRegion`, which is selected based on the type of the Java string or array and, in the case of strings, the type of the C pointer to distinguish between UTF-8 and UTF-16 encodings. Similarly, the `_getUTF8Length` and `_newJavaString` built-ins are translated into the corresponding JNI functions such as `GetStringUTFLength` and `NewStringUTF`.

In contrast, the translation of a `_with` statement

`_with (`*ca* `=` *ja*`) { ... }`

is more involved. To capture `_with`'s semantics, the Jeannie translator introduces two additional bindings in a method's C environment:

- `_caJavaObject`, which caches a C reference to the Java object resulting from expression *ja* and is used to release the string or array;
- `_caReleaseMode`, which can be `JNI_COMMIT` or `JNI_ABORT` and is updated by translated `_commit` and `_abort` statements.

Again, synthesized names avoid accidental name capture and may thus differ from the names shown here. To actually acquire and release arrays, the translation uses the same JNI copy functions discussed above, allocating the C array on the stack. To acquire and release strings, it uses JNI functions such as `GetStringChars` and `ReleaseStringChars`, which, depending on the JVM, may provide direct access to a string's contents. This does not violate `_with` statement semantics because Java strings are immutable and thus should not be updated by native code.

Abrupt Control Flow

Consistent with Jeannie's dynamic semantics as described in Section 4.2, the `return i` statement in Figure 7 abruptly transfers control to the method's caller while also releasing the array `s`. To implement these semantics, the translator introduces two additional bindings in a method's Java environment:

- `_returnResult` to hold the return value of non-void methods;
- `_returnAbrupt` to indicate whether a method's dynamic execution has initiated a return.

Figure 12 shows the complete translation of Figure 7's `return i` statement. It first sets `_returnResult` and `_returnAbrupt` in the Java environment. It then sets `_sReleaseMode` in the C environment; as described in Section 4.2 under "string and array access", the `return` statement is an implicit `_commit`. Finally, it jumps to label `release_s` at the end of the translated `_with` statement. After releasing the array `s`, the code at the end of the translated `_with` statement checks whether `_returnAbrupt` is true and, if so, re-initiates the pending abrupt method termination.

Abrupt termination checks occur not only at the end of `_with` statements, but also at the language boundary, since the method returns generated for abrupt control flow do not propagate beyond the method calls generated for nested code snippets. Notably, abrupt termination checks are generated for calls from Java to native methods implementing C snippets, if the snippets contain `return` statements. They are also generated for calls from C to Java methods implementing Java snippets, if the snippets may throw exceptions or contain `return` statements.

Figure 13 shows an abrupt termination check in C generated for a nested Java expression. The Java expression itself is executed by method `m1`. The abrupt termination check first caches the method's result in variable `tmp`. It then checks for a pending Java exception or a `return` flagged by `_returnAbrupt`. If so, it propagates the abrupt control flow through a `return` (or a jump to the end label of a surrounding `_with`). Otherwise, the statement expression simply evaluates to the cached result `tmp`.

Discussion

Our translation of Jeannie focuses on completeness. It correctly implements the whole language, but the resulting code may be inefficient, because it tends to be more general than necessary. We are aware of several opportunities for optimization. For instance, when a nested code snippet only reads but never writes a variable, it suffices to pass that variable's value down through a parameter instead of placing it in an environment class or struct. In many cases, the translator could thus avoid reifying environments altogether.

Furthermore, when a Java snippet nested in C code performs only a single Java operation that has a corresponding JNI function, the translator can directly use the JNI function instead of synthesizing an entire method wrapping the operation. Typically, the JNI function will be more efficient than an upcall from C to Java. After transforming a Java operation into a direct JNI function call, the translator may also be able to skip a language transition. More specifically, when the current language is C, code of the form

`` `( `*java-op* `( `` *c-expr* `)) ``

can be translated into code of the form

`({ T tmp=`*c-expr*`; (*env)->`*jni-function*`(tmp); })`

where the C expression is evaluated before the Java operation rather than in a callback from within the operation.

Finally, a `_with` statement can avoid copying an array and instead provide direct access to its contents if the observable behavior is the same. Notably, this is the case if the `_with` statement's block cannot abort and other threads cannot prematurely observe any modifications. JNI already

```
{
  jclass cls = (*env)->GetObjectClass(env, jEnv);
  jfieldID fidResult = (*env)->GetFieldID(env, cls, "_returnResult", "I");
  (*env)->SetIntField(env, jEnv, fidResult, pcEnv->_i);
  jfieldID fidAbrupt = (*env)->GetFieldID(env, cls, "_returnAbrupt", "Z");
  (*env)->SetBooleanField(env, jEnv, fidAbrupt, JNI_TRUE);
  pcEnv->_sReleaseMode = JNI_COMMIT;
  (*env)->DeleteLocalRef(env, cls);
  goto release_s;
}
```

Figure 12. C code generated for the "`return i`" statement from Figure 7.

```
({
  jclass cls = (*env)->GetObjectClass(env, jEnv);
  jmethodID mid = (*env)->GetMethodID(env, cls, "m1", "(I)I");
  jint tmp = (*env)->CallNonvirtualIntMethod(env, jEnv, cls, mid, (jint)pcEnv);
  jfieldID fid = (*env)->GetFieldID(env, cls, "_returnAbrupt", "Z");
  (*env)->DeleteLocalRef(env, cls);
  if ((*env)->ExceptionCheck(env) || (*env)->GetBooleanField(env, jEnv, fid))
    return;
  tmp;
})
```

Figure 13. C code generated for a nested Java expression with abrupt termination check, simplified for readability.

Description	Pseudo-code
Empty C block in Java	`` `{} ``
Empty Java block in C	`` `{} ``
Constant C expr. in Java	`` `1 ``
Constant Java expr. in C	`` `1 ``
Exception in C in Java	`try `{ `throw } catch {}`
Direct array read	`s += `a[`i]`
Direct array write	`` `(a[`i] = `1) ``
With array read	`_with (a) { s += a[i]; }`
With array write	`_with (a) { a[i] = 1; }`

Table 1. Summary of micro-benchmarks.

allows for direct access to (some) arrays through the, for example, `GetPrimitiveArrayCritical` and `ReleasePrimitiveArrayCritical` functions. But it relies on the programmer to restrict the code bracketed by these functions. In contrast, Jeannie's `_with` statement is more general and enables direct access as an optimization when the code, in fact, observes the necessary restrictions. We will investigate the relative priority of these and other optimizations in future work.

6. Evaluation

We evaluated our Jeannie compiler by porting part of the JavaBDD library [60] to Jeannie. JavaBDD defines an object-oriented API for manipulating binary decision diagrams and can interface with several implementations, including the BuDDy library written in C [39]. Our port replaces the JNI-based wrapper for BuDDy with the corresponding Jeannie version; this entailed rewriting 92 native methods across three classes. Additionally, we implemented a set of micro-benchmarks, which, as shown in Table 1, exercise all major language features new to Jeannie. Each micro-benchmark is implemented in Jeannie as well as plain Java and C using JNI. The JNI versions do not use explicit environments, thus incorporating the optimization discussed at the end of Section 5.3.

Program	Jeannie	Java	C
BuDDy wrapper	1,433	660	1,340
Micro-benchmarks	110	102	74

Table 2. Lines of code for Jeannie and JNI versions.

To demonstrate portability, all experiments run on two platforms with different Java virtual machines and operating systems. The first platform ("HS/OS X") is Apple's port of Sun's HotSpot JVM running on a MacBook Pro laptop with Mac OS X. The second platform ("J9/Linux") is IBM's J9 JVM running on an off-the-shelf Intel PC with Linux. Despite the different Java virtual machines and operating systems, all benchmarks seamlessly run across the two platforms. Portability is a strong point of JNI, and the Jeannie compiler preserves it.

Benchmark	HS/OS X	J9/Linux
11-Queens	1.04	1.04
Rubik's Cube	1.15	1.14
Empty C block in Java	0.98	1.23
Empty Java block in C	1.86	1.41
Constant C expr. in Java	1.05	1.06
Constant Java expr. in C	1.45	1.36
Exception in C in Java	1.05	0.95
With array read	1.09	1.65
With array write	1.19	1.06

Table 3. Speedup of JNI over Jeannie code.

To provide an indication of productivity, Table 2 compares the number of non-commenting, non-empty lines of code (LoC) between the Jeannie and JNI versions of the BuDDy wrapper and the micro-benchmarks. It does not count the C header files automatically generated by `javah` for the JNI versions. The LoC counts show that Jeannie is more concise than JNI, with JNI requiring 40% more code for the BuDDy wrapper and 60% more code for the micro-benchmarks. Additionally, the JNI versions require picking all the right API functions, with mistakes manifesting themselves as either linker or, worse, runtime errors. Jeannie not only prevents these mistakes in the first place, but also performs additional static checks across the language boundary. Finally, the JNI versions incur a higher maintenance cost, since they are spread over three separate files (`.h`, `.c`, and `.java`) instead of just one for Jeannie.

To quantify the overhead of the Jeannie compiler's translation scheme, Table 3 shows the speedup of the JNI versions over the Jeannie versions in a head-to-head performance comparison. The first two benchmarks exercise the BuDDy library through N-Queens and Rubik's Cube solvers distributed with JavaBDD. Speedup numbers are based on three iterations for each of the BuDDy benchmarks and one million iterations for each of the micro-benchmarks. The results show that the translation scheme's overhead is reasonable, especially for the end-to-end benchmarks using BuDDy, which see an overhead of at most 15%. By comparison, the micro-benchmarks show more variance, ranging from no speedup to overheads of several tens of percent for most micro-benchmarks to an outlier at 1.86. For comparison, the Jeannie versions of the 11-Queens and Rubik's Cube solvers have a speedup between 1.16 and 1.28 over a pure Java implementation of JavaBDD. We expect this speedup to increase, and JNI's speedup over Jeannie to decrease, as we implement the optimizations discussed in Section 5.3.

Initial testing exhibited a performance anomaly for one micro-benchmark. Further testing showed that this anomaly was caused by creating too many local references, which the garbage collector uses to track pointers from C to Java objects. Consequently changing the Jeannie translator to explicitly and eagerly release local references eliminated the anomaly. This experience demonstrates an additional advantage of Jeannie over JNI. Once an optimization has been implemented in the Jeannie compiler, all Jeannie code benefits through a simple recompilation. In contrast, JNI requires manually updating all C code—in this particular case, all method and field accesses through JNI's reflection-like API.

Access	HS/OS X	J9/Linux
Read	74.2	69.6
Write	74.7	83.8

Table 4. Speedup of array accesses through `_with`.

To quantify the benefits of the `_with` statement, Table 4 shows the speedup of accessing an array through `_with` over backticked subscript expressions. Each experiment either reads or writes each element in a 1,000 element `int` array. Overall, Jeannie obtains roughly a factor 75 speedup by providing the `_with` construct, which clearly justifies this non-orthogonal extension to the two base languages.

Finally, we consider the Jeannie compiler itself. The Jeannie grammar adds four modules with 250 LoC to the existing grammars for Java and C. The Java grammar, in turn, comprises 8 modules with 800 LoC, and the C grammar comprises 10 modules with 1,200 LoC. The Jeannie compiler's Java code, excluding any machine-generated code, comprises 1,900 non-commenting source statements (NCSS), which approximately corresponds to 1,900 semicolons and open braces and is a more conservative measure than the lines of code used above. By comparison, the C analyzer alone comprises 4,200 NCSS, the Java analyzer alone also comprises 4,200 NCSS, and the common type representation comprises 2,600 NCSS. Based on these statistics, we conclude that our composition of existing compiler components for Java and C was successful; the Jeannie compiler's complexity directly reflects the complexity of Jeannie's added features and not of the two embedded languages. Additionally, the compiler's performance was reasonable in our interactions; it did not feel noticeably slower than other compilers in the tool chain.

7. Related Work

Work related to Jeannie can be categorized into work on (1) the Java native interface in particular and foreign function interfaces in general, as well as on (2) combined languages and support for implementing language extensions and compositions.

7.1 Bridging Java, Other Languages with C

Besides Jeannie, several other efforts have explored implications of the Java native interface. Out of these, Janet [15, 36] comes closest to Jeannie by embedding C in Java as well; it even relies on the backtick operator to switch between Java and C. However, Janet performs only limited semantic analysis for Java and none for C. In fact, even its syntactic analysis for C is incomplete, tracking only curly braces and embedded Java expressions. As a result, Janet can abstract away

explicit JNI invocations, but also suffers from two significant limitations. First, Janet supports only Java embedded in C embedded in Java, with the embedded Java code limited to a few, simple expressions that have direct equivalents in JNI. Second, while Janet verifies that methods and fields referenced from embedded Java do, in fact, exist, it does not determine their types nor does it determine the correctness of the embedded C code. As a result, program bugs are detected either when compiling the resulting C code or, worse, at runtime. Unlike Janet, Jeannie fully composes the two languages and thus is more expressive, while also ensuring the correctness of programs.

Other work on JNI has explored the safety, efficiency, and portability implications of interfacing a safe, high-level language with an unsafe, low-level language. In particular, Furr and Foster [23] have developed a polymorphic type inference system for JNI, which statically ensures that literal names used in JNI invocations are consistent with the dynamic types of C pointers. Alternatively, Tan et al. [54] have extended CCured [19] to ensure that C code cannot subvert the safety of Java code. Their system relies on a combination of static analysis and runtime checks, thus guaranteeing safety. In the opposite direction, Tan and Morrisett [55] show how to improve the reach of Java analyses in the presence of native code. To this end, they introduce three new JVM bytecodes that model the behavior of C code and present a tool that automatically extracts models from native code. Next, Stepanian et al. [51] explore how to reduce the performance overhead of JNI invocations for small native methods such as `Object.hashCode` by inlining the code in the virtual machine's just-in-time (JIT) compiler. Finally, the Java JNI bridge [18] supports interfacing with a dynamic binary translator to run C code compiled for a different architecture than the JVM's. These efforts are largely orthogonal to our own work, though Jeannie's combined language is more amenable to semantic analysis and code generation than JNI with its reflection-like callbacks into the Java runtime.

Of course, JNI itself represents only one point in the design space of foreign function interfaces (FFIs). As discussed in detail in [46], FFIs provide both mechanism and policy for bridging between a higher-level language and C. The *mechanism* typically entails glue code to convert between the different data representations and calling conventions. For example, like JNI, the FFIs for O'Caml [37] and SML/NJ [33] as well as the Swig FFI generator for scripting languages [7] rely on glue code written in C. However, the FFI for the Scheme-based esh shell [47], the Haskell FFI [17], NLFFI for SML/NJ [10], and Charon for Moby [21] extend the higher-level language so that glue code can be written in the higher-level language.

The *policy* determines how to abstract C's low-level data structures and functions in the higher-level language. For most FFIs, this policy is fixed. For example, JNI enforces an object-oriented view of C code and data. However, FIG for Moby [46] allows for domain-specific policies by tailoring the FFI based on developer annotations. Similarly, Exu for Java [35] allows for customized policies through a meta-object protocol. In contrast, PolySPIN [6] and Mockingbird [2] hide most of an FFI's mechanism and policy by automatically mapping between data structures and functions of languages with recursive (but not polymorphic) types. Likewise, Jeannie hides most of JNI's mechanism and policy, but it eschews the type mapping machinery for a direct embedding of languages within each other.

7.2 Combined Languages and Their Pragmatics

Language composition has a long history back to literal C code appearing in specifications for lex and yacc and shell code appearing in specifications for sed and awk. However, these systems largely treat snippets of the embedded language as opaque strings. More recently, web scripting environments, such as PHP [5] or JSP [48], ensure that embedded HTML markup is at least syntactically well-formed. Next, XJ [31], XTATIC [26], Cω [8], and LINQ [41] take language composition one step further, combining a programming language with a data query language while also providing an integrated semantics. Jeannie shares the integration of syntax and semantics with the latter efforts, but also differs in scope: to our knowledge, we are the first to fully combine two mature, real-world programming languages.

To effectively combine or even to "just" extend programming languages, developers require tool and/or language support. While Jeannie relies on *Rats!* and `xtc` to make the composition of Java and C practical, a considerable body of work explores support for modular syntax, frameworks for language extension and composition, and in-language macro or meta-programming systems.

A first attempt at providing modular syntax was motivated by the embedding of a data language within a programming language [16]; it is, however, limited by the use of LL parsing, which is *not* closed under composition. In contrast, SDF2 [14, 57] achieves full modularity for syntactic specifications and has several similarities with *Rats!*. Notably, both systems support the organization of grammar fragments into modules and the composition of modules through parameterized definitions, called grammar mixins in SDF2 [12]. Both systems also integrate lexical analysis with parsing and support the inline declaration of abstract syntax trees.

At the same time, SDF2 and *Rats!* differ in the underlying formalism and parsing technique, thus leading to different trade-offs. SDF2 builds on CFGs and GLR parsing, which support arbitrary left-recursion but also lead to unnecessary ambiguities, thus requiring explicit disambiguation [56]. *Rats!* builds on PEGs and packrat parsing, which avoid unnecessary ambiguities through ordered choices but do not support left-recursion. It mitigates this limitation somewhat by translating directly left-recursive productions into equivalent right-iterations. *Rats!* also differs from SDF2 in its support for parser state: it enables parsers that pro-

duce a single AST, even if the language is context-sensitive, while SDF2's C parser requires non-trivial post-processing to disambiguate a forest of ASTs [1]. Finally, as shown in [29], *Rats!*-generated parsers are notably faster than SDF2-generated parsers.

While modular syntax helps with the composition of languages, developers still need support for realizing their languages' semantics. Notable alternatives to `xtc` include Polyglot [44], JastAdd [20], CIL [43], and Stratego/XT [13]. Polyglot and JastAdd provide frameworks for extending Java; they have been carefully designed to promote extensibility and maximize code reuse. Similarly, CIL provides a framework for extending C, but with less emphasis on extensibility and more emphasis on the analysis of C programs. All three frameworks have been used to realize major language extensions, including J& with Polyglot [45], Java 1.5 on top of version 1.4 with JastAdd [32], and CCured with CIL [19]. Stratego/XT is not targeted at extensions of a particular programming language. Rather, it provides a domain-specific language for specifying program transformations in general; it is also integrated with other language processing tools such as SDF2. As a language extension/composition framework, `xtc` is not yet as fully developed as these systems; however, as illustrated in this paper, it is sufficiently expressive to carry the full composition of Java and C into Jeannie.

Unlike the previous efforts, which distinguish between the syntactic and semantic aspects of language extension and composition, macro systems—such as MS^2 for C [59], Dylan [49], the Java Syntactic Extender [3], MacroML [25], Maya for Java [4], Template Haskell [50], and the language-independent `metafront` [11]—combine both aspects into a single specification. For these systems, a macro defines, *at the same time*, how to express a language extension, i.e., the syntax, and how to translate it to a more basic version of the language, i.e., the semantics. Overall, macros tend to hide many of the complexities of grammars, abstract syntax trees, and semantic analysis from developers and thus are more accessible and precise. However, as a result, they are also less expressive and more suitable for relatively targeted language modifications.

8. Conclusions

This paper presented Jeannie, a new language design for integrating Java with C. Jeannie nests both Java and C code within each other in the same file and supports a relatively straightforward translation to JNI, the Java platform's foreign function interface. By fully combining the syntax and semantics of both languages, Jeannie eliminates verbose boiler-plate code, enables static error detection even across the language boundary, and simplifies dynamic resource management. As a result, Jeannie provides improved productivity and safety when compared to JNI, while also retaining the latter's portability. At the same time, the Jeannie language does not require an implementation with JNI, and future work will explore how to optimize the translation for specific Java virtual machines, thus improving efficiency as well.

We implemented our Jeannie compiler based on *Rats!* [29], a parser generator supporting modular syntax, and `xtc` [28], a language composition framework. The compiler directly reuses grammars, semantic analyzers, and pretty printers for Java 1.4 and C with most gcc extensions. As a result, the complexity of the Jeannie compiler is largely commensurate with the newly added features and not the entire language. However, our experiences also illustrate the need for paying careful attention to shared infrastructure, notably the type representation, and to provide software hooks for overriding default behaviors. The open source release of our Jeannie compiler is available at `http://cs.nyu.edu/rgrimm/xtc/`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CNS-0448349 and CNS-0615129 and by the Defense Advanced Research Projects Agency under Contract No. NBCH30390004. We thank Joshua Auerbach, Rodric Rabbah, Gang Tan, David Ungar, and the anonymous reviewers for their feedback on earlier versions of this paper.

References

[1] R. Anisko, V. David, and C. Vasseur. Transformers: a C++ program transformation framework. Tech. Report 0310, Laboratoire de Recherche et Développement de l'Epita, Le Kremlin-Bicêtre cedex, France, May 2004.

[2] J. Auerbach, C. Barton, M. Chu-Carroll, and M. Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *Proc. 19th IEEE International Conference on Distributed Computing Systems*, pp. 393–402, May 1999.

[3] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 31–42, Oct. 2001.

[4] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in Java. In *Proc. 2002 ACM Conference on Programming Language Design and Implementation*, pp. 270–281, June 2002.

[5] S. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lefdorf, A. Zmievski, and J. Ahto. *PHP Manual*. PHP Documentation Group, Feb. 2004. `http://www.php.net/manual/`.

[6] D. J. Barrett, A. Kaplan, and J. C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *Proc. 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 147–155, Oct. 1996.

[7] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proc. 4th USENIX Tcl/Tk Workshop*, July 1996.

[8] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. In *Proc. 19th European Conference on Object-Oriented Programming*, vol. 3586 of *LNCS*, pp. 287–311, July 2005.

[9] J. Blosser. Java tip 98: Reflect on the visitor design pattern. *JavaWorld*, July 2000. `http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`.

[10] M. Blume. No-longer foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1):36–52, Sept. 2001.

[11] C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The `metafront` system: Extensible parsing and transformation. *Electronic Notes in Theoretical Computer Science*, 82(3):592–611, Dec. 2003.

[12] M. Bravenboer, É. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. In *Proc. 2006 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 209–228, Oct. 2006.

[13] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2005.

[14] M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proc. 2004 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 365–383, Oct. 2004.

[15] M. Bubak, D. Kurzyniec, and P. Łuszczek. Creating Java to native code interfaces with Janet extension. In *Proc. SGI Users's Conference*, pp. 283–294, Oct. 2000.

[16] L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Tech. Report 121, Digital Equipment, Systems Research Center, Feb. 1994.

[17] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. P. Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2003.

[18] M. Chen, S. Goldenberg, S. Srinivas, V. Ushakov, Y. Wang, Q. Zhang, E. Lin, and Y. Zach. Java JNI bridge: A framework for mixed native ISA execution. In *Proc. International IEEE Symposium on Code Generation and Optimization*, pp. 65–75, Mar. 2006.

[19] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proc. 2003 ACM Conference on Programming Language Design and Implementation*, pp. 232–244, June 2003.

[20] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proc. 2007 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2007.

[21] K. Fisher, R. Pucella, and J. Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1):3–19, Sept. 2001.

[22] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. `patch (1)` considered harmful. In *Proc. 10th Workshop on Hot Topics in Operating Systems*, pp. 91–96, June 2005.

[23] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *Proc. 15th European Symposium on Programming*, pp. 309–324, Mar. 2006.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Jan. 1995.

[25] S. E. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *Proc. 2001 ACM International Conference on Functional Programming*, pp. 74–85, Sept. 2001.

[26] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for XTATIC. In *Proc. 14th International Conference on Compiler Construction*, vol. 3443 of *LNCS*, pp. 43–58, Apr. 2005.

[27] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, June 2000.

[28] R. Grimm. xtc (eXTensible C). `http://cs.nyu.edu/rgrimm/xtc/`.

[29] R. Grimm. Better extensibility through modular syntax. In *Proc. 2006 ACM Conference on Programming Language Design and Implementation*, pp. 38–51, June 2006.

[30] C. Grothoff. Walkabout revisited: The runabout. In *Proc. 17th European Conference on Object-Oriented Programming*, vol. 2743 of *LNCS*, pp. 101–125, July 2003.

[31] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *Proc. 14th International World Wide Web Conference*, pp. 278–287, May 2005.

[32] G. Hedin. The JastAdd extensible Java compiler. `http://jastadd.cs.lth.se/web/extjava/index.shtml`.

[33] L. Huelsbergen. A portable C interface for Standard ML of New Jersey. Tech. report, AT&T Bell Laboratories, Jan. 1996. `http://www.smlnj.org/doc/SMLNJ-C/smlnj-c.ps`.

[34] International Organization for Standardization. Information Technology—Programming Languages—C. ISO/IEC Standard 9899:TC2, May 2005.

[35] A. Kaplan, J. Bubba, and J. C. Wileden. The Exu approach to safe, transparent and lightweight interoperability. In *Proc. 25th IEEE Computer Software and Applications Conference*, pp. 393–400, Oct. 2001.

[36] D. Kurzyniec. Creating Java to native code interfaces with Janet extension. Master's thesis, University of Mining and Metallurgy, Kraków, Poland, Aug. 2000. `http://janet-project.sourceforge.net/papers/janet_msc.pdf`.

[37] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.09. `http://caml.inria.fr/`, Oct. 2005.

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

[39] J. Lind-Nielsen. BuDDy. `http://buddy.sourceforge.net/`.

[40] B. McCloskey and E. Brewer. ASTEC: A new approach to refactoring C. In *Proc. 10th European Software Engineering Conference*, pp. 21–30, Sept. 2005.

[41] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proc. 2006 ACM SIGMOD International Conference on Management of Data*, p. 706, June 2006.

[42] T. Millstein. Practical predicate dispatch. In *Proc. 2004 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 345–364, Oct. 2004.

[43] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. 11th International Conference on Compiler Construction*, vol. 2304 of *LNCS*, pp. 213–228, Apr. 2002.

[44] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Conference on Compiler Construction*, vol. 2622 of *LNCS*, pp. 138–152. Springer, Apr. 2003.

[45] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 2006 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 21–36, Oct. 2006.

[46] J. Reppy and C. Song. Application-specific foreign-interface generation. In *Proc. 5th International Conference on Generative Programming and Component Engineering*, pp. 49–58, Oct. 2006.

[47] J. R. Rose and H. Muller. Integrating the Scheme and C languages. In *Proc. 1992 ACM Conference on LISP and Functional Programming*, pp. 247–259, June 1992.

[48] M. Roth and E. Pelegrí-Llopart. JavaServer Pages specification version 2.0. Tech. report, Sun Microsystems, Nov. 2003.

[49] A. Shalit. *The Dylan Reference Manual*. Addison-Wesley, Sept. 1996.

[50] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, Dec. 2002.

[51] L. Stepanian, A. D. Brown, A. Kielstra, G. Koblents, and K. Stoodley. Inlining Java native calls at runtime. In *Proc. 1st ACM/USENIX Conference on Virtual Execution Environments*, pp. 121–131, June 2005.

[52] Sun Microsystems. Java native interface specification, release 1.1, Jan. 1997.

[53] Sun Microsystems. Integrating native methods into Java programs. `http://java.sun.com/docs/books/tutorialNB/download/tut-native1dot0.zip`, May 1998.

[54] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang. Safe Java native interface. In *Proc. 2006 IEEE International Symposium on Secure Software Engineering*, pp. 97–106, Mar. 2006.

[55] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *Proc. 2007 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Oct. 2007.

[56] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proc. 11th International Conference on Compiler Construction*, vol. 2304 of *LNCS*, pp. 143–158. Springer, Apr. 2004.

[57] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Sept. 1997.

[58] J. Visser. Visitor combination and traversal control. In *Proc. 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 270–282, Oct. 2001.

[59] D. Weise and R. Crew. Programmable syntax macros. In *Proc. 1993 ACM Conference on Programming Language Design and Implementation*, pp. 156–165, June 1993.

[60] J. Whaley. JavaBDD. `http://javabdd.sourceforge.net/`.

[61] M. Yuen, M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Making extensibility of system software practical with the C4 toolkit. In *Proc. AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies*, Mar. 2006.