C to Java: Converting Pointers into References

Erik D. Demaine
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
eddemaine@uwaterloo.ca

Abstract

We consider the problem of converting C pointers to the less flexible concept of references. Our main application is converting scientific applications from C to Java. We provide a general method to model essentially all features of pointers using references. The model is easily implemented in Java. We give optimizations that map key facilities like arrays and structures onto the obvious Java equivalents, arrays and objects. These improvements make the conversion "optimal" for all typed pointers. For untyped pointers, we can still fall back on the general model, hence providing general automatic conversion from C to Java code, whose efficiency improves with the quality of the C code.

1 Introduction

The C language is well known for its overly flexible pointers. A pointer may represent the address of an individual value, an array, a structure, or a position in an array or structure. Pointers can be cast into other pointer types or integers, split into pieces and later rejoined, and can take part in arithmetic operations. While this gives the programmer immense flexibility, most of these abilities are considered unnecessary, and when used are dangerous and lead to code that is difficult to debug.

For this reason, many modern languages such as Java support a much more restricted model called *references*. The programmer has limited access to the references themselves; typically, one can only assign to a reference andtest whether two references point to the same object. Other uses of a reference are implicitly about the object it refers to; usually, the programmer does not explicitly dereference or take the address of an object. This leads to significantly safer and simpler code.

Since C is popular, there is a lot of code written in it. Many want to extend such programs and exploit features of more modern programming languages. For example, one may want to port an application to Java for demonstration on the World Wide Web. Since C and Java are similar in syntax, the main difficulty in converting between the two is the conversion between major concepts, in particular pointers and references.

In this paper, we discuss automatic conversion of pointers into references, with the application of converting C code into Java. Our main focus is on translating scientific applications written in C. A significant example of such a program is SuperLU [1, 2], a state-of-the-art general sparse linear solver released in 1997 and available only in C.

The rest of this paper is outlined as follows. Section 2 surveys related work. In Section 3, we give a general and flexible conversion model that provides nearly full conversion from C to Java. Section 4 discusses how this can be improved to be "optimal" for most applications. We conclude in Section 5.

2 Related Work

Much work has been done on converting Java to C for optimization purposes; see for example Harissa [3], j2c [4], JCC [5], and Toba [6]. However, little has been done on the reverse direction. Of particular relevance are c2j [7] and c2j++ [8]. Both are tools for translating C++ to Java. c2j++ is based on c2j; the main difference is that c2j++ is written in Java, whereas c2j is written in C++.

Unfortunately, the tools are limited. They essentially ignore the problem of pointers, which is the main difficulty of C-to-Java conversion. For example, they simply remove all pointer dereferences, that is, remove asterisks and convert "`->`" to "`.`". This is often correct when dealing with objects and pointers to objects, but (for example) never works for arrays. For scientific applications, arrays are perhaps the most important and common data structure, so their conversion is important. Indeed, we pay much attention to their efficient representation in Java in this paper.

Another related project is a recent Fortran-to-Java converter f2j [9]. It is of interest because it uses an intermediate representation of C, that is, it converts Fortran code into C, and converts the result to Java. However, this was done purely for convenience (since a Fortran-to-C converter was already available), and does not support general C to Java conversion; in particular, pointers are completely avoided. In comparison to our work, it also implements facilities for call-by-reference and multi-dimensional arrays, but the methods they use do not work in general (for both Fortran and C).

A couple of minor results in this paper have been partially achieved before. Section 4.1 discusses possible methods for passing basic values such as integers to functions by reference in Java. We know of one system supporting this, EPP [10]. EPP is a framework for Java preprocessors and one example, called ref [11], adds an operator "&" for passing values by reference. It implements this using one-element arrays, but in such a way that does not allow taking general addresses of basic values, unlike our methods in Section 4.1.

Section 4.1 also gives the results from a benchmark we wrote, comparing the various options for call-by-reference in Java. A related "microbenchmark" has been developed by Griswold and Phillips [12].

3 Block Model

This section defines what we call the *block model* of pointers. It allows us to abstract essentially all features of C pointers into a simple theoretical framework. The few limitations are discussed in Section 3.4.

At any time in the model, the memory consists of a collection of independent "top-level" blocks. Blocks can be dynamically created. Each block consists of a basic C value (character, integer, or floating-point value of some size), a pointer, or a structure containing several sub-blocks. A particular kind of structure of interest is the array, where each sub-block has the same type. Pointers can either be the special value `null` or refer to a (not necessarily top-level) block.

Let us introduce some notation for this model. We write $*p$ to denote the block pointed to by p. Note that p is also a block, since pointers are stored in blocks, and we associate blocks with their values. Hence, we can construct pointers to pointers, and so on. We write $b = (b_1, \ldots, b_n)$ when b is a structure block with sub-blocks $b_1, \ldots, b_n$.

We can now consider how pointers in such a model can be mapped onto references.

3.1 Pointer Comparison and Arithmetic

Suppose we want to determine whether $p \leq q$ for two pointers p and q. More generally, we will define the relation $\&b \leq \&c$, where $\&b$ denotes the *address* of a block b. This makes sense if b and c are in the same top-level block. The ordering of addresses is defined by the following two properties. (As is common, $a < b$ means $a \leq b$ and $b \not\leq a$.)

1. If $b = (b_1, \ldots, b_n)$, then $\&b_1 = \&b$ and $\&b_i < \&b_{i+1}$ for all $1 \leq i < n$.
2. The relation $\leq$ is transitive; that is, if $\&b_1 \leq \&b_2$ and $\&b_2 \leq \&b_3$ then $\&b_1 \leq \&b_3$.

If we want to define $\leq$ over all addresses, we could assign some order to the top-level blocks. For example, we could define that $\&b < \&c$ for two top-level blocks b and c if b was created before c.

Note that this ordering of addresses also makes it easy to add integers to pointers. Basically, adding n to a pointer p involves finding the n^{th} successor of p. However, we must take into account that each time we find the successor, we skip over a block of arbitrary size, whereas we want to skip over a particular number of bytes (n times the size of p's type). Similarly, we can define the difference of two pointers p and q; it is simply the n such that $p = q + n$. In general, we can support pointer arithmetic.

3.2 Stack

The block model only models heaps. What about variables allocated on the stack? One possibility is to model each stack element as a heap allocation. In fact, while this adds extra overhead, it is the most efficient general method. This is because, while Java has a stack, it does not support taking the address of anything on the stack [13]. Objects cannot be allocated on the stack (unlike C++ for example); only references to objects can be allocated on the stack. One cannot pass basic values by reference or otherwise find their addresses. Hence, all objects whose addresses are used at some point, which is all that our model is supposed to cover, must indeed be on the heap in Java. That is, if we encounter a stack object in the C source whose address is used, we must convert it into a heap allocation if we plan on converting to Java.

3.3 Implementation in Java

The block model is easy to implement in Java. We make an abstract class `Block`, which includes an entry `parent` of type `StructB`. `StructB` is a final class extending `Block`, consisting of an array of children. There are several other final classes extending `Block`. For example, `IntB` represents a block consisting of a single integer `val`, and `PointerB` represents a block pointing to some other block `target`. One can then define how to compare two pointers, increment a pointer, and in general do arithmetic on pointers, as we have described. An example of the conversion process is given in Figure 1.

This implementation has a limitation with dynamic allocation: we must know the type of block at the time of creation. This can be difficult with calls to the C `malloc` function, since only a number of bytes is passed in. Fortunately, most dynamic allocations are of the form `malloc (n * sizeof (type))`, which indicates that we should allocate a structure block with `n` entries of the specified type. In other words, we can modify the definition of `sizeof` to indicate type information. Most of the time, this will be sufficient.

```
main () {
   int a[4] = {1,2,3,0};
   printf ("%d\n", sum (a));
}
int sum (int *a) {
   int s = 0;
   while (*a != 0)
      s += *(a++);
   return s;
}
```

(a)

```
main () {
   IntB[] a_ = {new IntB (1), new IntB (2),
      new IntB (3), new IntB (0)};
   StructB a = new StructB (a_);
   System.out.println (sum (new PointerB (a)));
}
int sum (PointerB a) {
   int s = 0;
   while (((IntB) a.target).val != 0) {
      s += ((IntB) a.target).val;
      a.increment ();
   }
   return s;
}
```

(b)

Figure 1: *Example of block model. (a) Input C code. (b) Resulting Java code.*

However, in C one could call `malloc (n1 * sizeof (type1) + n2 * sizeof (type2))`. This suggests a structure block with `n1` + `n2` elements, but it does not indicate the relative order of `type1` and `type2` elements; they may be interspersed arbitrarily. To solve this case, we can create an *unknown block*, that is, an object of type `UnknownB`. Such a block stores its size in bytes and a `Block[]` array of that size. Initially all elements of the array are `null`, and as sections of the allocated space get used, the corresponding blocks get created.

This allows dynamic determination of sub-block types. It induces some overhead since we must check whether an array entry is `null` before using it, and if it is, we must create the block first. However, most dynamic allocations declare the type ahead of time, so we should rarely have to use this mechanism.

3.4 Extensions

The block model illustrates that we can automatically convert essentially any C code into Java. It supports almost all "safe" C code, modelling basic C types, pointers, and structures. Pointers are untyped, so they can be cast in the code as desired (including support for the `void *` type). However, the block model does not support union types or casting between integers and pointers. In general, such features represent extremely unwieldy type mangling that is fortunately uncommon in "well-written" code. Hence, the lack of such support is usually of limited importance.

However, there are several possible modifications to the block model that support more C features. For example, the most common form of union types, where only one entry is valid at any time, can be represented by an unknown block with one entry. Before accessing it, we check whether the entry is of the correct type, and if not replace it with a newly created block.

Another possibility, which fully supports C pointers including casting between integers, is to store all addressable memory in an `Object[]` array. Then a pointer is simply an index into this array, measured in bytes. As a result, we can trivially convert between pointers and integers, perform pointer arithmetic and comparison, etc. The main difficulty with this approach is supporting dynamic allocation of blocks with unknown types; if supported, we must check whether the array entry is `null` before every access to addressable memory. In constrast, the block model only had to do this for the blocks that had unknown types at creation time.

In conclusion, there are several possible strategies for general conversion, each with varying levels of flexibility and efficiency. Indeed, there are probably several approaches yet to be discovered, and the most appropriate model likely depends on the application. In the next section, we will see how to achieve extremely efficient conversion for the most common uses of pointers in scientific applications, such as arrays.

4 Improved Conversion

The block model is extremely general, but the code it generates can be much more complicated than required. In particular, it can be much slower than the original C.

For example, consider the conversion in Figure 1. The main problem with the resulting Java code is that it does not exploit Java's existing facilities for arrays. It creates several objects to duplicate the heap, representing it at the user level, for pointer-manipulation purposes. The reason for this is that the converter does not know, when converting sum's code, that a is an array. If it knew, it could perform the conversion in Figure 2.

```
main () {
   int a[4] = {1,2,3,0};
   printf ("%d\n", sum (a));
}
int sum (int *a) {
   int s = 0;
   while (*a != 0)
      s += *(a++);
   return s;
}
```

(a)

```
main () {
   int[] a = {1,2,3,0};
   System.out.println (sum (a));
}
int sum (int[] a) {
   int a_i = 0;
   int s = 0;
   while (a[a_i] != 0)
      s += a[a_i++];
   return s;
}
```

(b)

Figure 2: *Example of the improved conversion. (a) Input C code. (b) Resulting Java code.*

We can not expect much better code than this; it is simple and efficient. The goal of this section is to generate such a perfect mapping of C facilities onto Java features. Indeed, the methods of this section will perform the optimal conversion above.

4.1 Call-By-Reference

Passing values by reference is perhaps the most fundamental use of C pointers. In Java, values are passed by reference if and only if they are objects or arrays. The first question we must ask, then, is how to pass *basic* variables (that is, variables of basic type such as integer) by reference in Java. Let us call a variable *addressable* if its address is taken at some point in the C source. Otherwise, we do not need to worry about the variable.

The most obvious method is to replace all addressable basic variables with objects that simply contain an entry of that type (Figure 3). When such a variable is declared, it must be created using the `new` operator. If we wish to use or modify the value of the variable, we use or modify the entry in the corresponding object. The address of the variable is simply the corresponding object.

A similar method is to replace addressable basic variables with one-element arrays (Figure 4). Again, their declarations must now include creations with the `new` operator. Accessing the variable corresponds to accessing the first element of the array, and the address of the variable is the array itself. Indeed, the array method is nearly identical to the object method, replacing "`Int`" with "`int[]`."

```
public final class Int {
   public Int (int val) {
      this.val = val;
   }
   public int val;
}
```

C		Java
`int var;`	↦	`Int var = new Int ();`
`int ivar = 5;`	↦	`Int ivar = new Int (5);`
`var++;`	↦	`var.val++;`
`f (&var);`	↦	`f (var);`
`int *ptr = &var;`	↦	`Int ptr = var;`
`(*ptr)++;`	↦	`ptr.val++;`
`f (ptr);`	↦	`f (ptr);`

Figure 3: *Call-by-reference using objects.*

C		Java
`int var;`	↦	`int[] var = new int[1];`
`int ivar = 5;`	↦	`int[] ivar = {5};`
`var++;`	↦	`var[0]++;`
`f (&var);`	↦	`f (var);`
`int *ptr = &var;`	↦	`int[] ptr = var;`
`(*ptr)++;`	↦	`ptr[0]++;`
`f (ptr);`	↦	`f (ptr);`

Figure 4: *Call-by-reference using arrays.*

Objects and arrays are likely the only general methods for addressing basic variables. We could imagine achieving call-by-reference by passing copies of basic variables, instead of their pointers, and having the new values returned (e.g., through global variables for single-threaded programs). However, it does not seem possible to provide the desired semantics in general; for example, how could a pointer then be stored in a variable?

An obvious question is which of the two methods is more efficient in current Java implementations. To answer this, we ran a simple benchmark to compare the time for incrementing the first entry of an array versus incrementing the entry of an object. We also compared with the time for incrementing a basic integer variable. For accurate evaluation, we had to loop over several iterations and divide the (user CPU) time by the number of iterations. For correct evaluation, we subtracted the time to simply run through an empty loop of the same length.

The results are given in Table 1. Objects win over arrays most of the time, so we will likely choose them when possible. The difference is small on interpreted JVM's, but is more significant on the JVM with JIT (Just In Time) compilation. We likely want to leave the choice as an option to the conversion process, in case the relative performance differs on other machines.

Machine and JVM	Opt.	Integer	Object	Array
Sun SPARCstation 20	None	20.6 ns	162.6 ns	225.5 ns
Sun's 1.1.1 with JIT	-O	20.3 ns	163.4 ns	232.0 ns
IBM RS/6000 25T	None	23.4 us	92.2 us	117.0 us
IBM's 1.1.1 port	-O	25.4 us	91.2 us	116.9 us
DEC Alpha 3000/800S	None	20.8 us	169.4 us	142.1 us
DEC's 1.0.1 port	-O	20.7 us	160.7 us	164.5 us

Table 1: *Benchmark results comparing the access time for basic integers and elements of objects and arrays. The code was compiled with* `javac` *from Sun's JDK. "Opt." gives the optimization level. Curiously, higher optimization causes worse performance in some cases, and this phenomenon is reproducible.*

4.2 Arrays

Our main focus is on scientific applications, which mainly use basic numbers and arrays of numbers. Hence, these two kinds of values should receive special attention with regard to optimization. Basic C numbers are easily converted into efficient Java. The concern of this section is to map C arrays onto Java arrays whenever possible.

One problem is that a C array with entries of type t is really just a pointer to a block of type t. This makes it unclear when a pointer p in fact represents an array. If $*p$ is of basic type such as integer or even a pointer, we can simply assume that it represents an array, since pointers to basic values alone can also be represented by arrays. The other possibility is when $*p$ is a structure block, which we shall consider in Section 4.5.

Hence, we can map the C type $t*$ (pointer to type t) to the Java type $J(t*) = J(t)[\,]$ (Figure 5). Any addressable variable `v` of type t must also be mapped to this type, and references to `v` are replaced by `v[0]`, which the `&` operator converts back to `v`.

C		Java
`int a[5];`	$\mapsto$	`int[] a = new int[5];`
`int *p = a;`	$\mapsto$	`int[] p = a;`
`p++;`	$\mapsto$	Not allowed.
`*p = 5;`	$\mapsto$	`p[0] = 5;`
`p[2] = 10;`	$\mapsto$	`p[2] = 10;`
`int i;`	$\mapsto$	`int[] i = new int[1];`
`i = 5;`	$\mapsto$	`i[0] = 5;`
`p = &i;`	$\mapsto$	`p = i;`

Figure 5: *Mapping C arrays to Java arrays, requiring addressable basic integers to be mapped to arrays. Pointing to elements of an array is not allowed.*

4.3 Indexable Arrays

While Java arrays $J(t)[\,]$ adequately represent pointers to basic variables and pointers to arrays, they do not fully represent the C type $t*$. This is because Java arrays are not *translatable.* That is, we cannot make an array that refers to a subsection of an existing array. In contrast, C allows one to add an integer (such as 1) to a pointer that points to an array, causing it to point to a particular element of the array. Such practice is common (for example, in SuperLU [1, 2]), since in C it allows efficient access to individual elements of arrays; that is, repeatedly accessing `a[i]` is made faster by assigning `p = &a[i]` (or `p = a+i`) and then repeatedly accessing `*p`. We cannot make such an optimization in Java, but we still want to convert such code into Java.

Our solution, called *indexable arrays*, is to add an integer index to an array, together forming the Java notion of a pointer (to a basic type). That is, $J(t*) = \{J(t)[\,], \texttt{int}\}$, where the braces denote some kind of structure. In general, the structure will be represented by an object with two entries (Figure 6). Note that basic variables have to be mapped to arrays in the rare case when their address is stored in a pointer that also at some time points to an array.

One possible optimization, called *unrolling*, removes a level of indirection by replacing the structure with two individual variables (Figure 7). This improves efficiency, except when we return a pointer into an array from a function, in which case we must temporarily switch to the object representation.

Pointers to particular elements of arrays clearly add extra overhead that we would like to avoid for normal pointers to beginnings of arrays. That is, we do not want to force all pointers to have an

```
public class IntPointer {
   public IntPointer (int[] a, int i) {
      this.a = a;
      this.i = t;
   }
   public int[] a;
   public int i;
}
```

C		Java
`int a[5];`	↦	`int[] a = new int [5];`
`int *p = a;`	↦	`IntPointer p =` `new IntPointer (a, 0);`
`p++;`	↦	`p.i++;`
`*p = 5;`	↦	`p.a[p.i] = 5;`
`p[2] = 10;`	↦	`p.a[p.i+2] = 10;`
`int i;`	↦	`int[] i = new int[1];`
`i = 5;`	↦	`i[0] = 5;`
`p = &i;`	↦	`p.a = i; p.i = 0;`

Figure 6: *Indexable arrays using objects, requiring some basic integers to be mapped to arrays.*

C		Java
`int a[5];`	↦	`int[] a = new int[5];`
`a[2] = 5;`	↦	`a[2] = 5;`
`int *p = a;`	↦	`int[] p = a;` `int p_i = 0;`
`p++;`	↦	`p_i++;`
`*p = 5;`	↦	`p[p_i] = 5;`
`p[2] = 10;`	↦	`p[p_i+2] = 10;`
`int i;`	↦	`int[] i = new int[1];`
`i = 5;`	↦	`i[0] = 5;`
`p = &i;`	↦	`p = i;` `p_i = 0;`
`f (p);`	↦	`f (p, p_i);`

`p = g ();`	↦	`IntPointer temp = g ();`
		`p = temp.a;`
		`p_i = temp.i;`

`return p;`	↦	`return new IntPointer (p, p_i);`

Figure 7: *Unrolled indexable arrays. Returning indexable arrays from functions requires special care.*

extra index variable. Note that at conversion time, we can determine whether pointers may at some time point to a particular element of an array, and hence require the extra variable. Similarly, we can check if a variable of basic type can be addressed by simply using objects, which are typically more efficient. For function arguments we can make a special optimization, by copying the function for each type of arguments with which it is called (Figure 8). This allows effective optimization of the special cases, even if multiple cases are exploited. In the worst case, which is rare, function copying results in quadratic code expansion.

4.4 Multi-dimensional Arrays

Multi-dimensional C arrays are in fact one-dimensional arrays with syntactic sugar for accessing elements. For example, if we declare `int a[3][4]`, then both `a` and `a[2]` have types corresponding to `int *`. In contrast, Java's multi-dimensional arrays have depth. For example, `int[][] a = new int[4][]` declares `a` as a four-element array of references to arrays; each entry of `a` is initially `null`. Hence, multi-dimensional C arrays must be converted to one-dimensional Java arrays (Figure 9).

4.5 Structures

C structures quite obviously map to Java objects (Figure 10(a)). Any elements of a structure are converted as if they represented variables. For example, in Figure 10, the address of entry `i` of

C		Java
`int i;`	↦	`Int i;`
`f (&i);`	↦	`f (i);`
`int a[5];`	↦	`int[] a = new int[5];`
`f (a);`	↦	`f (a);`
`int *p = &a[2];`	↦	`int[] p = a;` `int p_i = 2;`
`f (p);`	↦	`f (p, p_i);`

`void f (int *i) {`
`*i = 5;`
`}`
	↦	`void f (Int i) { i.val = 5; }`
		`void f (int[] i) { i[0] = 5; }`
		`void f (int[] i, int i_i) { i[i_i] = 5; }`

Figure 8: *Copying functions to optimize for special cases. We make three copies of* `f` *because of the three different types of calls to* `f`*. Note the use of Java's function overloading.*

C		Java
`int a[3][4];`	↦	`int[] a = new int[3*4];`
`int b[3][4] = {{1,2},{3},{4,5,6,7}};`	↦	`int[] b = {1,2,0,0,3,0,0,0,4,5,6,7};`
`a[1][2] = b[2][1];`	↦	`a[1*4+2] = b[2*4+1];`
`int *p = a;`	↦	`int[] p = a;`
`int *q = b[2];`	↦	`int[] q = b;` `int q_i = 2*4;`

Figure 9: *Converting multi-dimensional C arrays to one-dimensional Java arrays.*

`struct T` is taken; hence, it must be converted to be addressable, which involves converting the type of entry `i` from `int` to `Int` (an object with an `int` in it). A similar transformation would occur if a structure had an indexable array as an entry, for example. In addition, we must clone (that is, duplicate) any structures that are passed by value, since objects can only be passed by reference in Java.

As a potential optimization, we can unroll structures that are not addressable (Figure 10(b)). This means that we do not have to clone anything when passing by value. Of course, we cannot pass an unrolled structure by reference. For large structures that are passed between functions a lot, unrolling may worsen efficiency. Hence, the user should be able to specify that certain structures should or should not be unrolled.

4.6 Untyped Pointers

For `void *` pointers or pointers that are type cast, called *untyped* pointers, we cannot use the efficient conversion we have discussed. Instead, we must resort to the block model of Section 3.3. Note that we can detect which pointers must be typed. If the address of some block b is stored in an untyped pointer, we must convert the entire top-level block containing b using the block model. Any pointers that directly or may indirectly point to such information must also use the block model.

Our goal is to use the block model as little as possible. Indeed, "well-written" (in particular scientific) applications should not need it at all. Our analysis of SuperLU [1, 2], a good representative of such applications, indicates that the block model is unnecessary. In general, the programmer may want to hand-convert any code that requires the block model in order to improve efficiency.

```
C                          Java
struct S {             ↦   class S implements
                                  Clonable {
   char c;                    char c;
   int i;                     int i;
};                         }
struct S x;            ↦   S x = new S ();
f (x);                 ↦   f (x.clone ());
struct S *p = &x;      ↦   S p = x;
g (p);                 ↦   g (p);
x.i = 5;               ↦   x.i = 5;
p->i++;                ↦   p.i++;
struct T {             ↦   class T {
   char c;                    char c;
   int i;                     Int i = new Int ();
};                         }
struct T y;            ↦   T y = new T ();
y.i = 5;               ↦   y.i.val = 5;
f (&y.i);              ↦   f (y.i);
```

(a)

```
C              Java
struct S {  ↦  char x_c;
   char c;     int x_i;
   int i;
} x;
f (x);      ↦  f (x_c, x_i);
x.i = 5;    ↦  x_i = 5;
x = g ();   ↦  S temp = g ();
               x_c = temp.c;
               x_i = temp.i;
struct T {  ↦  char y_c;
   char c;     Int y_i;
   int i;
} y;
y.i = 5;       y_i.val = 5;
f (&y.i);   ↦  f (y_i);
```

(b)

Figure 10: *Converting C structures to Java objects. (a) Standard. (b) Unrolled.*

5 Conclusion

We have presented a general model for complete and automatic C-to-Java conversion. We then considered optimizations to this general conversion, for the common uses of pointers in scientific applications. In particular, we can usually achieve the natural conversions of C arrays into Java arrays and C structures into Java objects.

The implementation of our models are currently under development. We are also working on a converter from C to Fortran 90, which mainly involves the same problem of converting pointers to references. In the future, we plan on extending our work to convert C++ to Java and Fortran 90.

Acknowledgment

We thank Alan George and David Taylor for their valuable comments on this paper. Alan George also motivated the development of this research. This work was supported by the Natural Sciences and Engineering Research Council (NSERC).

References

[1] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report UCB//CSD-95-883, Computer Science Division, U. C. Berkeley, July 1995. To appear in *SIAM J. Matrix Analysis and Applications.*

[2] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. *SuperLU Users' Guide*, November 1997. World Wide Web. `http://www.netlib.org/scalapack/prototype/superlu_ug.ps`.

[3] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[4] Yukio Andoh. j2c/CafeBabe. World Wide Web. `http://www.webcity.co.jp/info/andoh/java/j2c.html`.

[5] Nik Shaylor. JCC — A Java to C converter, May 1997. World Wide Web. `http://www.geocities.com/CapeCanaveral/Hangar/4040/jcc.html`.

[6] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: A Java-to-C translator. World Wide Web. `http://www.cs.arizona.edu/sumatra/toba`.

[7] Chris Laffra. C2J — A C++ to Java translator. In *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, chapter 4. Prentice Hall Computer Books, 1996.

[8] Ilya Tilevich. C2J++, C++ to Java translator written in Java, 1997. World Wide Web. `http://pacevm.dac.pace.edu/~ny971734/c2j.html`.

[9] Geoffrey Fox, Xiaoming Li, Qiang Zheng, and Wu Zhigang. A prototype of FORTRAN-to-Java converter. In *Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*, Las Vagas, Nevada, June 1997.

[10] Yuuji Ichisugi and Yves Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference*, Marina del Rey, California, December 1997. World Wide Web. `http://www.aist.go.jp/ETL/etl/bunsan/~ichisugi/doc/iscope97.ps.gz`.

[11] Yuuji Ichisugi. epplib/ref.lsp. In *EPP distribution*, 1997. World Wide Web. `http://www.aist.go.jp/ETL/etl/bunsan/~ichisugi/`.

[12] William G. Griswold and Paul Phillips. UCSD benchmarks for Java. World Wide Web. `http://www-cse.ucsd.edu/~wgg/JavaProf/javaprof.html`.

[13] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.