ACM *OOPS Messenger* **4**,3 (July 1993), 18-25.

Iterators: Signs of Weakness in Object-Oriented Languages

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 (FAX)

The appearance of *iterators* in an object-oriented language appears to be inversely related to the power of the language's intrinsic control structures. Iterator classes are used for the sole purpose of enumerating the elements of an abstract collection class without revealing its implementation. We find that the availability of higher-order functions and function closures eliminates the need for these *ad hoc* iterator classes, in addition to providing the other benefits of "mostly functional programming". We examine a purely functional iteration protocol for the difficult task of comparing the collections of leaves on two non-isomorphic trees—the so-called "samefringe" problem—and find that its type signature requires recursive (cyclic) functional types. Although higher-order "member functions" and recursive (cyclic) functional types are unusual in object-oriented languages, they arise quite naturally and provide a satisfying programming style.

A. INTRODUCTION

As a long-time Lisp programmer experienced in the use of `mapcar` and friends, I was perplexed by the appearance of "iterators" in various object-oriented programming languages—e.g., CLU [Liskov77] and C++ [Stroustrup86]. Since all of the usual forms of iteration can be trivially simulated using mapping functions, I couldn't understand the need for the apparently *ad hoc* notion of an *iterator*, whose sole function was to provide the machinery to enumerate the objects in a collection without revealing the implementation of the collection. However, recent experience with C and C++ has finally revealed the source of my cognitive dissonance—these languages lack the ability to write and use simple mapping functions like Lisp's `mapcar`! In particular, while C and C++ offer the ability to pass a function as an argument to another function (i.e., a "funarg"), the functions being passed cannot contain references to variables of intermediate scope (i.e., the functions are not function closures), and therefore they are essentially useless for most mapping processes. On the other hand, most languages of the Pascal family—e.g., Ada and Modula—provide for the creation and passing of function closures and therefore do not require iterators for simple iteration processes.[1]

The primary purpose of this paper is a discussion of the expressive power of various languages, but efficiency is always a concern, especially with constructs that must examine large collections. It should therefore be pointed out that the expression of a loop in terms of mapping functions, function closures and recursive function-calling does not mean inefficiency. It is well-known how to compile "tail-recursive" functions into highly efficient iteration [Steele78], and high quality Lisp systems have been "open-coding" mapping functions for at least 20 years.

B. MAPPING FUNCTIONS

The Lisp language was the first language to make extensive use of recursion and mapping functions instead of Fortran and Algol-like `DO`/`FOR` iteration constructs. For example, the Lisp function `mapc` "maps" a function "down" a list—i.e., it applies the function successively to every element in the list. Here is one definition for `mapc`:

```
(defun mapc[2] (function list)                        ; tail-recursive loop.
  (if (null list) nil
    (progn (funcall function (car list))
           (mapc function (cdr list)))))
```

For those people find it more productive to think of an iterative process instead of a mapping process, Common Lisp provides the macro `dolist`. The following fragment shows two different ways to print all of the elements of a list.

```
(dolist (x l) (print x))                       ; prints elements of list l.

(mapc #'(lambda (x) (print x)) l)     ; or more simply (after eta conversion),

(mapc #'print l)                               ; prints elements of list l.
```

Regardless of the syntax, the two forms are equivalent, as the following definition of the `dolist` macro shows.

[1] Ada83 [Ada83] does not allow the passing of subprograms as arguments to other subprograms, but does allow the passing of functions as arguments to generic subprograms; this capability will suffice for the emulation of iterators.

[2] This definition is not quite the same as Common Lisp `mapc`.


```
(defmacro dolist ((var list) &body forms)      ; approximation to CL 'dolist'.
  `(mapc #'(lambda (,var) ,@forms) ,list))
```

Mapping functions are not limited to Lisp lists, as our definition of the Lisp `dotimes` iteration macro shows.

```
(defmacro dotimes[3] ((var limit) &body forms)
  `(maptimes #'(lambda (,var) ,@forms) ,limit))

(defun maptimes (function n)
  (labels
    ((myloop (i)              ; local tail-recursive function to perform loop.
       (if (= i n) nil
         (progn (funcall function i)
                (myloop (1+ i))))))
    (myloop 0)))

(maptimes #'print 10)                          ; prints numbers from 0 through 9.

(dotimes (i 10) (print i))                     ; prints numbers from 0 through 9.
```

It should be clear that the more complicated iteration forms of Pascal, C, C++, etc., can all be emulated in the same fashion using mapping functions.

C. GENERATORS AND CO-ROUTINES

Iterators are a 1980's version of the 1950's concept of *generators*. A generator is a subroutine which is called like a function, but returns a "new" value every time it is called (i.e., it is emphatically *not* a mathematical function). The "random number generator" is prototypic of a generator.

Generators were heavily investigated in the late 1960's and early 1970's in the context of Artificial Intelligence programs which had to "hypothesize and test". By hiding the mechanism for hypothesizing a solution inside a generator, the program could be made more modular and understandable.

Generators have much in common with *co-routines*, and many generators are programmed as co-routines. Co-routines are most generally thought of as two independent programs that communicate by means of an intermediate buffer. Whenever the consumer of the generated objects requires another object, it takes one from the buffer, and if the buffer is currently empty, it waits for the generator to generate new objects and place them into the buffer. The generator works to fill up the buffer, and when it is full, it goes to sleep until room is once again available for new objects. The buffer size is usually finite, and may actually be zero, in which case the generator is activated directly by the consumer upon every request for a new object.

One is tempted to think that every co-routine generator can be emulated by a function which has access to some persistent local state (e.g., an Algol-60 "own" variable), but this is not the case. The standard counter-example is the "samefringe" problem, which can be solved by two co-routined "fringe" generators, but cannot be solved by a function with only a bounded amount of persistent local state.

The "fringe" of a finite tree is the simple enumeration of its leaves in left-to-right order. The "samefringe" problem is the problem of deciding whether two finite trees have the same fringe. The "samefringe" problem is not purely academic, but arises in the implementation of equality in sequence classes which have "lazy" concatenation semantics. In such lazy sequences, the concatenation operation is very fast, because it simply binds together two subsequences without examining the subsequences at all. The tradeoff comes when the sequences must be enumerated; the abstract sequence is obtained by examining the "leaves" of the concatenation tree in order. In other words, the abstract sequence is simply the "fringe" of the tree of concatenations. If one constructs at least one of the fringes as an explicit data structure, the samefringe problem is easily solved.

```
(defun fringe (x &optional (c nil))                       ; fringe of tree x.
  (if (atom x) (cons x c) (fringel x c)))

(defun fringel (xl c)                             ; fringe of list of trees xl.
  (if (null xl) c (fringe (car xl) (fringel (cdr xl) c))))

(defun samefringe1 (x y)            ; construct both fringe(x) and fringe(y).
  (equal (fringe x) (fringe y)))

(defun samefringe2 (x y)                           ; construct only fringe(x).
  (cfringe (fringe x) y #'(lambda (lx) (null lx))))
```

[3]This definition is not quite the same as Common Lisp `dotimes`.


```
(defun cfringe (lx y c)
  (if (atom y) (and (consp lx) (eql (car lx) y) (funcall c (cdr lx)))
    (cfringel lx y c)))

(defun cfringel (lx yl c)
  (if (null yl) (funcall c lx)
    (cfringe lx (car yl) #'(lambda (lx) (cfringel lx (cdr yl) c)))))
```

The problem with explicitly constructing the fringes is that in the case that one (or both) of the fringes is very large and the fringes don't match, we will have performed a great deal of construction work for nothing. We would like a "lazier" method for examining the fringes which does not perform any construction unless and until it is required.

The samefringe problem can be solved without constructing either of the fringes as explicit data structures by utilizing a "continuation-passing" style of programming. The reason for using this style lies in the unbounded amount of storage required to hold the stacks used to explore each tree; any generator for a fringe must have access to an unbounded amount of storage. Co-routines can also solve the samefringe problem because each of the co-routines has its own tree-tracing stack. Clearly, any "iterator" solution to the samefringe problem must give the iterator enough storage to emulate these stacks.

```
(defun eof (c) (funcall c nil t nil))        ; empty generator, send eof flag.

(defun samefringe (x y)                                     ; "lazy" samefringe.
  (samefringec #'(lambda (c) (genfringe x c #'eof))
               #'(lambda (c) (genfringe y c #'eof))))

(defun samefringec (xg yg)  ; check equality of leaves generated by genfringe.
  (funcall xg                  ; We don't need Scheme 1st class continuations.
   #'(lambda (x eofx xg)   ; receive 1st elt., eof flag, & generator for rest.
       (funcall yg
        #'(lambda (y eofy yg)
            (or (and eofx eofy)      ; equal if both reach eof simultaneously.
                (and (eql x y) (samefringec xg yg))))))))

(defun genfringe (x consumer genrest)  ; call consumer with leaves found in x.
  (if (atom x) (funcall consumer x nil genrest)   ; send 1st elt. & ¬eof flag.
    (genfringel x consumer genrest)))

(defun genfringel (xl consumer genrest)
  (if (null xl) (funcall genrest consumer)
    (genfringe (car xl) consumer
     #'(lambda (consumer) (genfringel (cdr xl) consumer genrest)))))
```

We don't claim that this style of programming is particularly perspicuous, although it can be understood with practice. It does show how a relatively complex task can be solved in a completely functional way—i.e., there are no assignments in this code, nor are there any explicit conses which must later be garbage-collected. If all of the function closures which are created are stack-allocated (as if by C's `alloca`, see [Baker92CONS]), then this solution doesn't do any heap allocation at all. It must, of course, utilize an amount of storage which is proportional to the sizes of the tree portions being compared, but all of this storage can be stack-allocated.

A more perspicuous functional solution can be obtained using the idea of "lazy evaluation" from functional languages. The idea is to compute the fringe lazily by producing a lazy list of the tree leaves, which can be elegantly created by means of "lazy consing" [Friedman76]. These lazy lists are gradually coerced by an equality function until either a mismatch is found, or both fringes have been completely explored. We can emulate laziness in Lisp by surrounding any lazy arguments with a `lambda` having no parameters, which is stripped off when necessary by calling this `lambda` with no arguments. Furthermore, we can utilize Common Lisp's global macro facilities to insert the `lambda`'s for us, and use its local macro facilities to insert the `funcall`'s necessary to force the evaluation of the lazy arguments. For `lsamefringe`, we only need `fringe`'s second argument to be lazy.

```
(defmacro lcons (x y)          ; a cons which is lazy in its second argument.
  `(cons ,x #'(lambda () ,y)))

(defmacro lcdr (x) `(funcall (cdr ,x)))

(defmacro fringe (x &optional (c nil))      ; make fringe lazy in its 2nd arg.
  `(lfringe ,x #'(lambda () ,c)))     ; this macro must appear before fringel.
```

```
(defun lfringe (x lc)
  (symbol-macrolet ((c '(funcall lc))) ; accessing 'c' forces arg. evaluation.
    (if (atom x) (lcons x c)        ; cdr of new cons is unevaluated argument.
      (fringel x c))))                              ; fringel is defined above.

(defun lequal (x y)             ; works only for 1-level list with lazy cdr.
  (or (and (null x) (null y))
      (and (consp x) (consp y)
           (eql (car x) (car y)) (lequal (lcdr x) (lcdr y))))))

(defun lsamefringe (x y) (lequal (fringe x) (fringe y)))
```

Unfortunately, our lazy evaluation implementation uses "upward funargs", which are typically heap-allocated, and therefore may produce a lot of transient garbage. Nevertheless, in the most common case where the fringes are unlikely to match, the amount of consing for a lazy samefringe is far less than that for "strict" (non-lazy) samefringe.

D. ITERATORS

Iterators provide a means to "generate" (enumerate) all of the elements in a collection, but without revealing the structure and implementation of this collection. Below is an example of the use of iterators to print the elements of a collection of C `int`'s.

```
int i,first(void),next(void); boolean last(void);

for (i=first(); !last(); i=next()) {printf("%d\n",i);}
```

Iterator functions for a collection can be "member functions" of the C++ collection class itself, but this policy creates a problem if one wishes to enumerate the same collection for different purposes at the same time—e.g., if one wishes to enumerate the Cartesian product of a collection with itself. The problem is that the "state" of the iterator must be stored somewhere, and the only two places are in the instance of the collection or the class of the collection. If the state is stored in the class, then only one iterator can be active for the entire class, while if the state is stored in the instance, then only one iterator can be active for the collection. Neither of these policies provides the flexibility required for our Cartesian product problem.

The solution usually proposed for C++ is to provide every collection class with a "friend" class called its "iterator class". Because the iterator class is a friend of the collection class, it is allowed to "see" its implementation, yet any state required for an iteration is stored in an iterator instance which is distinct from the collection instance itself. Since multiple iterators can be active on the same collection instance, iterators can solve the Cartesian product problem.

E. EMULATING MAPPING IN C AND C++

Before we can show how mapping can replace iterators, we first explore what goes wrong when we attempt to program a mapping function—e.g. `maptimes`—in C. We first show `maptimes` itself.

```
void maptimes(fn,limit) void (*fn)(int); int limit;
{int i; for (i=0; i<limit; i++) (*fn)(i);}
```

So far, so good. C allows us to pass a function to another function, which can then call the passed function with a locally defined argument. We can use our `maptimes` function to print the numbers from 0 through 9.

```
static void myprint(i) int i; {printf("%d\n",i);}

...; maptimes(&myprint,10); ...
```

Already, we can see one problem of C—it has no "anonymous" functions, so we have to come up with a name for a function, define the function with that name, and then use this name in the call to `maptimes`, even though the function is used only once. This problem is an irritation, but not a show-stopper.

The real show-stopper for C (and C++) is the fact that we cannot define *local* functions to pass to `maptimes`, which local functions would have access to the variables in the local environment. Suppose, for example, that we wished to sum the elements of a vector of `int`'s with a simple `for`-loop, as in the following code:

```
int a[10] = { ... } /* vector of int's. */

...; {int sum=0; for (i=0; i<10; i++) sum+=a[i]; printf("sum=%d\n",sum);} ...
```

Within the loop, we need access to `i`, `sum`, and `a`. If we attempt to program this loop using our `maptimes`, we get the following code:

```
int a[10] = { ... }
```



```
static void myloop(i) int i; {sum+=a[i];}
...; {int sum=0; maptimes(&myloop,10); printf("sum=%d\n",sum);} ...
```

The problem with this `maptimes` version is that the use of the variable name "`sum`" within the body of `myloop` does not refer to the `sum` later declared locally, but refers to some globally-scoped (or at least file-scoped) variable named "`sum`". We could easily define such a global `sum` in this particular program, but we would then lose the benefits of locally-defined and locally-scoped temporary variables.

Before giving up on the C language completely, we should try one final ploy, which is suggested by the implementation of the Lisp `dotimes` macro. The ploy is to attempt to program a `dotimes` macro using the C preprocessor macro facility. The hope is to have an expansion for `dotimes` such that any reference to `sum` within the expansion will refer to any locally-defined `sum`, rather than a global `sum`. Here is our attempt at a C `dotimes` macro:

```
#define dotimes(var,limit,body) {int var; for(var=0; var<limit; var++) body;}
...; {int sum=0; dotimes(i,10,{sum+=a[i];}); printf("sum=%d\n",sum);} ...
```

But this solution works![4] Where's the problem? The problem is that C macros are globally-scoped and are not controlled or inherited by class definitions. Thus, one cannot provide a generic `iterate-collection` macro as a "virtual macro" of the "collection" class, which would then be defined for each collection subclass as a local C macro. In other words, macros in C++ live in a totally different name space and are not aware of any of the class structure of C++.[5]

As these examples show, C (and C++) is weak in a number of ways. It does not support proper function closures, nor does it have a macro facility powerful enough to properly support something as simple as a `dotimes` macro.

F. MAPPING 'FUNCTIONS' IN ADA83

Ada83 does not have first-class functional arguments like Pascal, but it does have generic subprograms which can take subprograms as generic formal parameters. More importantly, it is possible to define these generic subprograms in packages defining "private types", which are the Ada83 incarnations of abstract data types. By exporting these generic subprograms, an abstract collection type can provide iteration capabilities without a separate iterator type.

```
generic type element_type is private;       -- Element type is generic formal.
package collection_package is
  type collection is private;             -- Define a private collection type.
  generic with procedure body(e: element_type);
    procedure iterate(c: collection);
  pragma inline(iterate);
private ...
  end collection_package;

with collection_package;
procedure main is
  -- Define or inherit 'my_element_type' type.
  package my_collection is new collection_package(my_element_type);
  local_list: my_collection.collection;
  begin
    ...
    declare    -- The lack of anonymous procedures in Ada is truly irritating.
      procedure iterate_body1(e: my_element_type) is ... ;
      procedure iteration1 is new my_collection.iterate(interate_body1);
      pragma inline(iterate_body1,iteration1);
      begin
        iteration1(local_list); -- iterate over local list of elements.
      end;
    ...
  end main;
```

We don't intend to imply that the above code is attractive, but it can be just as efficient as a typical for-loop. Ada83 generics are usually implemented by means of macro-expansion, so that the iterator generic itself is expanded in-line

[4]Well, it mostly works. Watch out for commas in the "loop" body that aren't surrounded by parentheses (ugh!).

[5]In short, C macros are déclassé.

much like our C `dotimes` macro, above. Furthermore, the body of the iteration which is passed as a procedure can be declared with a pragma `inline`, thus informing the compiler to substitute the body in place of the subroutine call in the middle of the iteration procedure. Finally, the local generic procedure instance itself can be declared `inline`, eliminating that subprogram call, as well. In other words, if the Ada compiler takes seriously the `inline` pragmas, then the resulting code for our generic iterators should be as efficient as `for`-loops.

Interestingly, Ada83 cannot use C++-style iterator classes, because Ada83 packages have no friends. In other words, any iterator types used in Ada83 must be defined in the same package that defines the collection class itself. Given the choice of iterator types or iterator generics, we feel that iterator generics provide cleaner semantics and fewer problems, especially in the context of Ada's multiple parallel tasks.

Besides its awkward syntax, which can be corrected by a decent macro system or CASE tool, our scheme for Ada83 iterator generics has some other rough edges. Unlike normal subprograms, Ada83 generics cannot be overloaded, or even renamed, although their instances can. This means that the only way to obtain unqualified access to the name of a generic is via a `use` clause, which is considered dangerous because it makes everything in a package visible without qualification, and any collisions can cause previously visible names to disappear!

G. WHY USE MAPPERS INSTEAD OF ITERATORS?

The major problem with iterators is that they are non-functional functions—they depend upon an internal state and side-effects to operate. It is very difficult to prove *anything* about functions with state, and therefore such functions inhibit most compiler optimizations. But the most damning criticism of iterators occurs in the context of multitasking and parallel threads. When using parallel threads, all side-effects must be protected by locks, in order to ensure the consistency of interpretation of the contents of shared objects with state. While most iterators in C++ are locally-defined for a particular loop, and therefore not shared with any parallel process, it is difficult for a compiler to prove this. As a result, one is either left with an inefficient program, or a potentially dangerous program, depending upon whether the compiler inserts locks to be on the safe side, or leaves them out in the name of efficiency.

These problems do not arise with the higher-order mapping functions. As we have shown, mapping functions can be completely functional (i.e., they have no shared state or assignments), even for complex iterations like those for comparing the fringes of non-isomorphic trees. This means that an arbitrary number of mappers can transparently access the same collections without interference—at least so long as the collections are not updated. If the collections are also read-only, or functional, then there is no possibility of interference under any circumstances. Mappers create function closures during their operation, and these temporary objects hold the "state" of the iteration, much as iterator objects do in C++. The difference is that these closure objects are created automatically during the execution of the mapping functions, rather than having to be explicitly created by the programmer, as in C++, and the side-effect-free nature of mappers is obvious to a compiler based on a cursory syntactic scan.

H. FUNCTIONAL ITERATION PROTOCOLS

Since we reject the non-functional protocols associated with C++-style iterators, we must exhibit a functional iteration protocol which can replace it. One possible functional iteration protocol is modelled after the `genfringe` routine from section C above. In order to better understand it, let us derive the "type signature" that `genfringe` would have in a strongly-typed language like ML [Harper86].

The `genfringe` function accepts 3 arguments and returns one result, so it has a signature of the form A×B×C->D, where A,B,C,D are ML-style "type variables". The first argument to `genfringe` (`x`) is the collection type itself—in this case Lisp's `list` type. The second argument to `genfringe` (`customer`) is obviously a function of 3 arguments itself, where the first argument is Lisp's `list` type, the second argument is a `boolean`, and the third argument is the same as `genfringe`'s third argument; the result of calling this functional argument is the same as the result of calling `genfringe` itself Therefore, a more precise typing for `genfringe` is `list`×(`list`×`boolean`×C->D)×C->D. In order to obtain the best typing for `genfringe`, we must type `genfringel`. A first approximation to the type of `genfringel` is `list`×(`list`×`boolean`×C->D)×C->D—i.e., the same as the type of `genfringe` itself! But we can see from the recursive calling structure of `genfringel` that C=(`list`×`boolean`×C->D)->D. In other words, C is no longer an ML-style type variable, but a recursive/cyclic type.[6] Interestingly, however, D remains an ML-style type variable which can be replaced by any type whatsoever, depending upon the nature of the functions which call `genfringe`; in the case of `samefringec`, D becomes the actual type `boolean`.

The fact that typing our generator function produces a recursive/cyclic type should not be surprising, since collection types themselves are recursive/cyclic types (i.e., they can hold an indefinite number of elements of the base type). If one wants to be pedantic, one can say that the cyclic functional type of our generator is the functional analog to C++-style iterator classes, so we haven't gotten rid of iterator classes after all. However, our cyclic generator types do

---

[6]ML can infer a cyclic type if the "occur check" is removed from the unification algorithm, but don't try to print it!

have some pleasant properties: 1) they involve only the collection type itself, the boolean type, and the cyclic generator type itself, so *they do hide the implementation of the collection class*; 2) the result type is a type variable, so *these generators can be used in any mapping context*; and 3) they are derived in a "purely functional" way, with *no data structures defined for holding a local iterator state*.

Recursive types have long been staples of strongly-typed languages like Pascal, Ada, C, C++, etc. However, the recursive types in these languages are always recursive *data* types [Aho86,s.6.3], not recursive *functional* types, like those of our functional iteration protocols. Recursive functional types are no more difficult to understand or implement than recursive data types; the apparent reason for their absence from most languages seems to be lack of imagination regarding their benefits. In fact, one cannot execute iterative or recursive programs without the use of recursive functional types, because the lambda calculus Y operator [Barendregt84] [Gabriel88] used to implement recursion has the ultimate recursive functional type.

Unfortunately, Ada83 (along with almost every other strongly typed language) does not support recursive types other than recursive data types, and therefore cannot support the cyclic generator types of our functional generators, which are more powerful than the simple Ada mappers of section F. One can still program powerful functional iteration protocols in Ada83, but these now require the publication of a second "iteration state" abstract type for every collection type—i.e., Ada83 still requires a form of iterator types. Each iterator function must accept such an iterator type as a "current state" argument, and produce the "next state" iterator type as one of its results; since the state is accepted and manipulated explicitly, the iterator subprogram has no hidden state and hence it is functional. Unfortunately, in the case of the samefringe problem, the requirement to make the state of the iteration explicit means the generation and manipulations of explicit stacks instead of the far more perspicuous recursive programming used in our examples, above. In other words, the lack of recursive functional types in Ada can turn simple "pop quiz" programming problems into full-blown group projects. This is not the way to obtain higher software productivity.[7]

Below we show how functional generator protocols can be used to program approximations to standard mapping functions such as Common Lisp's `every` and `mapcar`, which will work on arbitrary sequences, including trees.

```
(defun eof (c) (funcall c nil t nil))                  ; same as in section C.

(defmethod gen ((x tree) consumer genrest)
  (labels ((genfringel (xl consumer)
             (if (null xl) (funcall genrest consumer)
               (gen (the tree (car xl)) consumer
                #'(lambda (consumer) (genfringel (cdr xl) consumer))))))
    (if (atom x) (funcall consumer x nil genrest)
      (genfringel x consumer))))

(defmethod gen ((x list) consumer genrest)
  (if (null x) (funcall genrest consumer)
    (funcall consumer (car x) nil
     #'(lambda (consumer) (gen (the list (cdr x)) consumer genrest)))))

(defmethod gen ((x vector) consumer genrest)
  (labels ((myloop (i consumer)
             (if (= i (length x)) (funcall genrest consumer)
               (funcall consumer (aref x i) nil
                #'(lambda (consumer) (myloop (1+ i) consumer))))))
    (myloop 0 consumer)))

(defun every (fn x y)8            ; true only if all fn applications are true.
  (labels
   ((myloop (xg yg)
     (funcall xg
      #'(lambda (x eofx xg)
          (funcall yg
           #'(lambda (y eofy yg)
               (or (and eofx eofy) (and (funcall fn x y) (myloop xg yg)))))))))
   (myloop #'(lambda (c) (gen x c #'eof)) #'(lambda (c) (gen y c #'eof)))))
```

[7]I defy CASE tool advocates to show how their CASE tools can help in the solution of this problem. In other words, no CASE tool can make up for fundamental weaknesses in a programming language.

[8]Our `every` handles exactly 2 sequences; do *n* sequences as an exercise (Hint: lazily iterate on `every` itself!).


```
(defun samefringe (x y) (every #'eql x y))          ; example of every's use.
(defun mapcar (fn x y)  ; accepts 2 generic sequences, returns list of values.
  (labels
   ((myloop (xg yg)
     (funcall xg
      #'(lambda (x eofx xg)
          (funcall yg
           #'(lambda (y eofy yg)
               (if (or eofx eofy) nil
                 (cons (funcall fn x y) (myloop xg yg)))))))))
   (myloop #'(lambda (c) (gen x c #'eof)) #'(lambda (c) (gen y c #'eof)))))
```

I. CONCLUSIONS

We have shown how the ability to define higher-order functions (functions which take other functions as arguments) and the ability to define local functions which are closed over their free lexical variables can be used to provide iteration capabilities for abstract collections without the need to define an iterator class for each such collection class. These higher-order functions can then be provided as member functions of the collection class itself to provide the needed iteration capability. We have shown an example of such a higher-order iterator definition in Ada83, which is not normally considered to be an object-oriented programming language.

The popular object-oriented programming language C++ has neither functional closures, nor a first-class macro capability (which might be used to emulate our Ada83 scheme), which makes C++ too weak to provide proper iteration facilities within the collection class itself. Thus, the *ad hoc* scheme of pairing every collection class with its own iterator friend class is forced by the weakness of C++ control structures.

Iterators are far less powerful a facility than higher-order functions, so the *ad hoc* solution of iterator classes does nothing to solve more complex problems like samefringe which are easily solved using higher-order functions.

We have shown that functional mapping can be performed by means of functional generators whose type signatures are recursive/cyclic types, but recursive *functional* types rather than the more common recursive *data* types. While these generator types are analogous to C++-style iterator classes, our generator types are not *ad hoc*, and have just the properties one would expect from an abstract generator type—they hide the implementation of the collection class, and they have a generic type signature which can be used in a wide variety of contexts.

In conclusion, the attempt by object-oriented languages to avoid the "complexities" of higher-order functions and recursive functional types appears to have backfired. "Iterators" may look like an easy way out, but they do nothing to solve hard problems like samefringe. Higher-order functions can not only solve samefringe, but solve it in a way that is far easier to understand than any solution involving explicit state.

J. REFERENCES

Ada83. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Printing Off., Wash., DC, 1983.

Aho, A.V., *et al*. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

Baker, H.G. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *ACM Ada Letters XI*,5 (July/Aug. 1991), 79-90.

Baker, H.G. "CONS Should Not CONS its Arguments, or A Lazy Alloc is a Smart Alloc". *ACM Sigplan Not. 27*,3 (March 1992),24-34.

Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, New York, 1984.

Cohen, Ellis S. "Updating Elements of a Collection in Place". *ACM Ada Letters 6*,1 (1986),55-62.

Coyle, C., and Grogono, P. "Building Abstract Iterators Using Continuations". ACM *Sigplan Not. 26*,2 (Feb. 1991), 17-24.

Eckel, Bruce. "C++ Into the Future". *Unix Review 10*,5 (May 1992), 26-35.

Friedman, D.P., and Wise, D.S. "CONS Should not Evaluate its Arguments". In S. Michaelson and R. Milner (*eds*.), *Automata, Languages and Programming*, Edinburgh U. Press, Edinburgh, Scotland, 1976, 257-284.

Gabriel, R.P. "The Why of Y". *Lisp Pointers 2*,2 (Oct./Dec. 1988), 15-25.

Harper, R., *et al*. "Standard ML". TR ECS-LFCS-86-2, Comp. Sci. Dept., Edinburgh, UK, March 1986.

Liskov, B., *et al*. "Abstraction mechanisms in CLU". *CACM 20*,8 (Aug. 1977),564-576.

Milner, R. "A theory of type polymorphism in programming". *JCSS 17*,3 (1978), 348-375.

Shaw, M., *et al*. "Abstraction and verification in Alphard: Defining and specifying iteration and generators". *CACM 20*,8 (Aug. 1977),553-564.

Steele, Guy L. *Rabbit: A Compiler for Scheme*. AI Memo 474, MIT, May 1978.

Steele, Guy L. *Common Lisp, the Language; 2nd Ed*. Digital Press, Bedford, MA, 1990,1029p.
Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.