Efficient High-Level Iteration with Accumulators

ROBERT D. CAMERON
Simon Fraser University

Accumulators are proposed as a new type of high-level iteration construct for imperative languages. Accumulators are user-programmed mechanisms for successively combining a sequence of values into a single result value. The accumulated result can either be a simple numeric value such as the sum of a series or a data structure such as a list. Accumulators naturally complement constructs that allow iteration through user-programmed sequences of values such as the iterators of CLU and the generators of Alphard. A practical design for high-level iteration is illustrated by way of an extension to Modula-2 called Modula Plus. The extension incorporates both a redesigned mechanism for iterators as well as the accumulator design. Several applications are illustrated including both numeric and data structure accumulation. It is shown that the design supports efficient iteration both because it is amenable to implementation via in-line coding and because it allows high-level iteration concepts to be implemented as encapsulations of efficient low-level manipulations.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*control structures*

General Terms: Design, Languages

Additional Key Words and Phrases: Accumulation, accumulators, generators, iteration, iterators, loop constructs, mapping functions, Modula, reduce operator

1. INTRODUCTION

Alphard [6] and CLU [4] provide user-definable iteration mechanisms known as *generators* and *iterators*, respectively. Both these facilities allow programmers to define their own abstract sequence of values for consideration within a **for** loop. For example, in CLU one could define an iterator *IntElems* that could generate all the elements in a list of integers and could be used in a **for** loop as follows:

```
for x: integer in IntElems(list) do
  process(x);
end;
```

Just as the **for** loops of many other languages allow a loop control variable to assume values over a range of integers, generators and iterators allow loop control

This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Simon Fraser University Centre for Systems Science.
Author's address: School of Computing Science, Simon Fraser University, Burnaby, British Columbia V5A 1S6, Canada.

variables to assume values over arbitrary user-specified sequences. In particular, generators and iterators provide elegant support for abstract data types, allowing **for** loops iterating through sequences of values naturally associated with such user-defined data structures as lists, queues, stacks, and trees.

This paper proposes the *accumulator* as a complementary iteration mechanism to generators and iterators. Rather than specifying the generation of sequence values, an accumulator specifies how successive values of a sequence can be combined to calculate a single value. For example, the accumulator *Sum* could be defined to add up a series of numeric values. Given such an accumulator, the sum of the squares of the elements in an integer list could be calculated using the following *accumulation expression* (using an adapted CLU syntax):

Sum{*x* ∗ *x* **for** *x*: *integer* **in** *IntElems*(*list*)}

Alternatively, a sequence of values can be accumulated into a data structure. For example, given a list of integers, the corresponding list of their squares may be computed using an appropriate *MakeIntList* accumulator:

MakeIntList{*x* ∗ *x* **for** *x*: *integer* **in** *IntElems*(*list*)}

Just as generators and iterators allow the generation of sequences of values from data structures, accumulators in turn allow the elegant construction of data structures from sequences.

In fact, the combination of generation of sequence values and accumulation of those values into a single result already underlies a variety of high-level iteration constructs found in various functional languages. In APL [2], the *reduce* operator ("/") allows successive elements of a vector V to be accumulated in various ways, for example, as a sum using the notation "+/V" or as a product using the notation "×/V". LISP mapping functions [7] generate successive values from lists, apply functions to them, and accumulate their results back into lists. Similar features also occur in the languages FP [1], NIAL [3], and SETL [5] for iteration and accumulation over sequences, nested arrays, and sets, respectively. In each case, the high-level iteration constructs are an important contribution to the brevity and clarity of programs expressed in the language.

The viewpoint of this paper is that the benefits of high-level iteration constructs in functional languages can be carried over into imperative languages as well. In the imperative context, however, such iteration constructs are subject to further design criteria not generally relevant to functional languages. First of all, it is important that such constructs in imperative languages allow iteration and accumulation over user-defined data abstractions; typical functional languages have a single dominant data-structuring mechanism and provide high-level iteration constructs for that mechanism only. Second, efficiency of implementation is much more important in imperative languages than in functional languages, and is especially important in looping constructs whose execution may dominate overall program performance. The approach herein, then, is to achieve these objectives through abstraction mechanisms that can be programmed at a low level for efficiency and can be used at a high level for the concise expression of iterative computations.

In order to provide a concrete illustration of the incorporation of high-level iteration constructs into imperative languages, this paper presents a design that extends the Modula-2 programming language; the extended language will be called *Modula Plus.* The concepts, however, are not particular to Modula-2 and could be adapted for inclusion in a wide variety of imperative languages.

The remainder of this paper is organized as follows: Section 2 describes the design and use of iterators in Modula Plus, drawing upon and extending the existing designs in CLU and Alphard. Accumulators are then presented in Section 3, with several examples of their implementation and use. Section 4 concludes the paper with an evaluation of the iterator-accumulator approach in comparison with other high-level iteration schemes for imperative languages. Section 5 provides a summary.

2. ITERATORS AND **FOR** LOOPS IN MODULA PLUS

In Modula Plus, the **ITERATOR** construct is responsible for the generation of sequence values to be used in **FOR** loops, in the same fashion as the iterator construct of CLU and the generator construct of Alphard. Thus, the use of iterators also requires a redesign of Modula-2's **FOR** loop, again following the approach of CLU and Alphard. In addition, Modula Plus incorporates a **FIRST** loop patterned after the corresponding construct in Alphard.

2.1 Iterators

The syntax of Modula Plus iterators is as follows:

```
⟨iterator-declaration⟩ ::=
  ITERATOR ⟨name⟩ "(" ⟨parameter-list⟩ ")" ":" ⟨type-name⟩ ";"
  [⟨declarations⟩]
  [INIT ⟨statement-list₁⟩]
  LOOP ⟨statement-list₂⟩
  [FINAL ⟨statement-list₃⟩]
  END ⟨name⟩ ";"
```

The ⟨declarations⟩ section of the iterator typically contains declarations of one or more variables that are used to define the state of an iteration. When a loop involving an iterator is to be executed, the initialization statements (⟨statement-list$_1$⟩) are first executed to set up the proper initial state of the iteration. The main body of the iterator (⟨statement-list$_2$⟩) is then executed at the start of each iteration cycle to either terminate the iteration (using an **EXIT** statement) or update the state of the iteration and supply the next value for the loop control variable. In the main body of the iterator, the name of the iterator appearing on the left-hand side of an assignment is used to denote the loop control variable, so that values "assigned" to the iterator name are actually assignments to the control variable of the calling loop. Finalization statements (⟨statement-list$_3$⟩) may be specified to perform any cleanup actions necessary when the iteration terminates.

Consider, for example, the *Upto* iterator shown in Figure 1, which generates the sequence of integer values in a given range. The variable *nextValue* is used to represent the state of the iteration, indicating the next value to be given to the loop control variable. At the beginning of each iteration, the terminating condition is tested, allowing further execution only if the *final* value has not been

```
ITERATOR Upto (init, final : INTEGER) : INTEGER;
VAR nextValue : INTEGER;
INIT
  nextValue := init;
LOOP
  IF nextValue > final THEN EXIT
  ELSE
    Upto := nextValue;
    nextValue := nextValue + 1
  END
END Upto;
```

Fig. 1. The *Upto* iterator.

exceeded. If so, then the current value of *nextValue* becomes the new value for the loop control variable by assignment to *Upto*, and then the iteration state is updated for the next time around.

It may be the case that an iterator generates no sequence values at all. For example, if n has the value 0 and a loop involving the iterator-call *Upto*(1, n) is invoked, the *Upto* iterator will terminate with an **EXIT** at the beginning of the first iteration. In such a case, the main body of the calling loop is not executed at all.

Just as in CLU and Alphard, the main benefit of such programmable iterators is that sequences of values over user-defined data abstractions can be generated. For example, consider the following data type for lists of integers:

TYPE
  IntListType = **POINTER TO** *IntListCell*;
  IntListCell = **RECORD** *value*:*INTEGER*; *next*:*IntListType* **END**;

An iterator that generates all the integers from such a list can be implemented as shown in Figure 2.

2.2 FOR Loops

Modula Plus iterators are designed to be used in a modified **FOR** loop with the following syntax:

⟨for-loop⟩ ::= **FOR** ⟨iterator-list⟩ **DO** ⟨statement-list⟩ **END**
⟨iterator-list⟩ ::= ⟨iterator-section⟩ {"," ⟨iterator-section⟩}
⟨iterator-section⟩ ::= ⟨loop-variable⟩ **IN** ⟨iterator-call⟩
⟨loop-variable⟩ ::= ⟨name⟩
⟨iterator-call⟩ ::= ⟨*iterator*-name⟩ "(" ⟨argument-list⟩ ")"

A **FOR** loop may have several loop control variables, with values supplied to them in parallel by separate iterator invocations. The body of the loop (⟨statement-list⟩) is executed for each set of variable assignments so generated. The loop is terminated when any one of the iterators terminates with an **EXIT**.

Typically, a **FOR** loop will involve only one iterator, as in the following simple counting loop:

FOR *i* **IN** *Upto*(1, *n*) **DO** *SomeProc*(*i*) **END**

```
ITERATOR IntElems (x : IntListType) : INTEGER;
VAR tail : IntListType;
INIT
  tail := x;
LOOP
  IF tail = NIL THEN EXIT
  ELSE
    IntElems := tail^.value;
    tail := tail^.next
  END
END IntElems;
```

Fig. 2. The *IntElems* iterator.

Note that, if *n* is less than 1, the loop terminates without executing its body. Thus, the loop is entirely equivalent to its Modula-2 counterpart:

FOR *i* := 1 **TO** *n* **DO** *SomeProc*(*i*) **END**

Using iterators, however, the loop control variable need not be a counter, but may stand for elements from a data structure instead.

FOR *i* **IN** *IntElems*(*someList*) **DO** *SomeProc*(*i*) **END**

Parallel iteration is typically useful either for considering elements from two or more data structures in one-to-one correspondence or for setting up an associated counter while processing a data structure. For example, one could apply the procedure *SomeProc*2 to corresponding pairs of integers from two lists as follows:

```
FOR i IN IntElems(list1), j IN IntElems(list2) DO
  SomeProc2(i, j)
END
```

If the values of *list*1 and *list*2 are (1, 4, 7) and (5, 12, 6), respectively, then the *SomeProc*2 procedure will be applied successively to the argument lists (1, 5), (4, 12), and (7, 6). Alternatively, the following loop considers elements from a list together with a counter indicating the element position for up to 100 elements of the list:

```
FOR i IN IntElems(list), j IN Upto(1, 100) DO
  SomeProc2(i, j)
END
```

Unfortunately, if there are more than 100 elements in the list, only the first 100 are processed because the loop terminates when the *Upto* iterator **EXITS**.

Problems with arbitrary limits on parallel counter variables can be avoided using "infinite" or nonterminating iterators. These are simply iterators without an **EXIT** statement in their main body. For example, the *CountFrom* iterator shown in Figure 3 generates integers ad infinitum from a given initial value. Using this iterator, an unlimited number of elements from a list can be considered

```
ITERATOR CountFrom (init : INTEGER) : INTEGER;
VAR nextValue : INTEGER;
INIT
  nextValue := init;
LOOP
  CountFrom := nextValue;
  nextValue := nextValue + 1
END CountFrom;
```

Fig. 3. The *CountFrom* iterator.

in a loop with a parallel counter variable as follows:

```
FOR i IN IntElems(list1), j IN CountFrom(1) DO
  SomeProc2(i, j)
END
```

Note that the use of nonterminating iterators for parallel loop variables also improves efficiency by eliminating unwanted tests.

2.3 In-Line Coding of Iterators

In order to keep iteration as efficient as possible, Modula Plus iterators are designed to be easily implementable by in-line coding. Essentially, the strategy is to replace a **FOR** loop and its iterator calls by a simple **LOOP** with copies of the iterator bodies at the beginning of the loop. Consider, for example, the following in-line coded version of the **FOR** loop just above:

```
tail := list1;
nextValue := 1;
LOOP
  IF tail = NIL THEN EXIT
  ELSE
    i := tail^.value;
    tail := tail^.next
  END;
  j := nextValue;
  nextValue := nextValue + 1;
  SomeProc2(i, j)
END
```

The statements just before the **LOOP** are the initialization statements of the iterators with substitutions made to simulate parameter passing to the iterators. Within the **LOOP**, note that the **EXIT** statement from the *IntElems* iterator is unchanged and becomes an **EXIT** statement for the **LOOP**. Assignments to the iterator names *IntElems* and *CountFrom* have been replaced by assignments to the corresponding loop control variables *i* and *j*. Of course, appropriate copies of the local declarations from the iterators must be inserted into the program unit containing the **FOR** loop; systematic renaming is generally also required to ensure that new unique names are used for each in-line coded iterator call.

If the iterator has finalization statements, they are in-line coded by placing them after the **END** of the loop.

2.4 **FIRST** Loops

The **FIRST** loop of Modula Plus is adapted from the corresponding construct in Alphard as a convenient method for programming search loops. The syntax is as follows:

```
⟨first-loop⟩ ::=
  FIRST ⟨iterator-list⟩ SUCH THAT ⟨expression⟩
  DO ⟨statement-list₁⟩ ELSE ⟨statement-list₂⟩ END
```

The basic semantics of the loop is to search through sets of values for the loop control variables until the first such set is found for which the given ⟨expression⟩ evaluates to *TRUE*. This set of variable bindings is then made available for the execution of ⟨statement-list$_1$⟩, after which the loop is terminated. As with the **FOR** loop, successive sets of values for each loop control variable in ⟨iterator-list⟩ are generated in parallel. If any of the iterators exits before a suitable set of values is found, then the loop is terminated and ⟨statement-list$_2$⟩ is executed in an environment in which the values of all the loop control variables are undefined. Essentially, ⟨statement-list$_1$⟩ gives the actions to be taken upon a successful search, whereas ⟨statement-list$_2$⟩ is used in the case of search failure.

2.5 Comparison

Although the objectives of the iterator and generator designs in CLU, Alphard, and Modula Plus are similar, the mechanisms are different. In particular, CLU uses a coroutine mechanism for the definition of iterators that differs substantially from the Modula Plus approach. In Alphard, generators are defined using *&init* and *&next* functions in an approach that is roughly equivalent to that of Modula Plus.

From a programming perspective, the CLU design has some advantage for its more convenient definition of iterators in certain cases, but has a significant disadvantage in the way iterators may be used. The first advantage of CLU's coroutine mechanism is that it allows recursively defined iterators, such as an iterator to generate the leaves of a binary tree. In Modula Plus, such iterator programming requires the use of an explicit stack to simulate the recursion, but is still practicable. The second advantage of the coroutine approach is that it allows the definition of iterators for a high-level abstraction in terms of the iterators of a lower-level abstraction. In Modula Plus, the higher-level iterator cannot be defined in terms of the lower-level iterator, but must instead be defined in terms of the functions and procedures provided by the lower-level abstraction. In practice, however, these iterator definitions are still relatively small and straightforward in Modula Plus. The disadvantage of the CLU design is that it does not allow the style of parallel iterator combination within **for** loops that is available in Modula Plus. To allow such parallel iteration in CLU would require a more complex coroutine facility than the simple stack-based mechanism that suffices for nested and recursive, but not parallel, iterator invocation. In programming terms, then, the Modula Plus design is arguably preferable to that of CLU on the basis that it is more important to support the high-level use of abstraction mechanisms than to support high-level means for implementing them.

The Modula Plus approach is also advantageous from the efficiency perspective. In CLU, each cycle of a **for** loop imposes the cost of two coroutine activations (to and from the iterator). The corresponding cost in Modula Plus is somewhat less, essentially that of a procedure call and return. Furthermore, the Modula Plus design is directly amenable to efficient implementation via in-line coding as illustrated above, whereas the CLU design is not, both because iterators may be recursively defined and because iterators may have multiple points (**yield** points) at which a coroutine return to the calling **for** loop may occur.

3. ACCUMULATORS

An accumulator is an abstraction that allows a sequence of values to be combined to yield a single result value. The **ACCUMULATOR** construct of Modula Plus allows accumulators to be programmed as state machines in a fashion very similar to **ITERATORS**. Given such accumulator implementations, high-level iterations can be programmed using either *accumulator expressions*, which combine both iteration and accumulation, or *literal accumulations*, which provide accumulation over the values of a sequence of expressions explicitly provided by the user.

3.1 Accumulator Programming

The syntax of Modula Plus accumulators is as follows:

```
⟨accumulator-declaration⟩ ::=
  ACCUMULATOR ⟨name⟩ "(" ⟨parameter-name⟩ ":" ⟨type-name1⟩ ")"
    ":" ⟨type-name2⟩ ";"
  [⟨declarations⟩]
  [INIT ⟨statement-list1⟩]
  LOOP ⟨statement-list2⟩
  FINAL ⟨statement-list3⟩
  END ⟨name⟩ ";"
```

This construct defines an accumulator that processes a sequence of values of the type specified by ⟨*type*-name$_1$⟩ yielding an accumulated value whose type is specified by ⟨*type*-name$_2$⟩. As with iterators, the ⟨declarations⟩ typically contain one or more variables used to define the state of the accumulation. When an accumulator is invoked in a given loop, the initialization statements (⟨statement-list$_1$⟩) are first executed to set up the proper accumulator state. Subsequently, the actions specified by the main body of the accumulator (⟨statement-list$_2$⟩) are executed for each generated sequence value, wherein ⟨*parameter*-name⟩ names the current sequence value. Upon exhaustion of the sequence values, actions specified by ⟨statement-list$_3$⟩ are then executed, including a **RETURN** statement to yield the final value of the accumulation.

Using this construct, the *Sum* accumulator can be implemented in a straightforward fashion as shown in Figure 4. When a loop involving the *Sum* accumulator is invoked, the accumulator state variable *total* is initialized to 0. Each time a sequence value x is generated and passed to the accumulator, its value is added to *total*. When the sequence values have been exhausted, the final value of *total* is returned as the result of the accumulation.

```
ACCUMULATOR Sum (x : INTEGER) : INTEGER;
VAR total : INTEGER;
INIT
  total := 0;
LOOP
  total := total + x
FINAL
  RETURN total
END Sum;
```

Fig. 4. The *Sum* accumulator.

```
ACCUMULATOR Minimum (x : INTEGER) : INTEGER;
VAR MinSoFar : INTEGER;
INIT
  MinSoFar := MAXINT;
LOOP
  IF x < MinSoFar THEN MinSoFar := x END
FINAL
  RETURN MinSoFar
END Minimum;
```

Fig. 5. The *Minimum* accumulator.

Another example of an arithmetic accumulator is one that determines the minimum of a sequence of integer values as shown in Figure 5. Other arithmetic accumulators, for example, *Product* and *Maximum*, can be similarly implemented.

Accumulators for data structures can also be defined using the **ACCUMULATOR** construct. For example, the *MakeIntList* accumulator that accumulates a series of integers into a list (of type *IntListType* given previously) is defined in Figure 6. Here, the accumulator state consists of a *result* list being built up, together with a pointer *lastCell* to the last physical cell of the list. In general, each time a new sequence value x is generated, it is placed in a new list cell, *newCell*, that is created for it, and the link field *next* of the previous *lastCell* is set to point to this *newCell*. The *newCell* then becomes the new *lastCell* for the next iteration. The initial sequence value is handled as a special case, however, since no *lastCell* exists at that point. Once all the sequence values have been accumulated, the list is properly terminated by placing *NIL* in the *next* field of the final *lastCell*, and then the accumulated *result* list is returned. Note that the accumulation of lists is quite efficient when programmed this way, requiring only a small constant cost for the addition of each list element.

In general, it is important to program accumulators so that an appropriate value is returned even if the main body of the loop is not executed at all, that is, if no sequence values are accumulated. In such a case, only the **INIT** and **FINAL** portions of the accumulator are executed. For example, if there are no integers to be accumulated by *MakeIntList*, *NIL* is returned to denote the empty list. Similarly, the *Sum* and *Minimum* accumulators return 0 and *MAXINT*, respectively, when accumulating over empty sequences.

```
ACCUMULATOR MakeIntList (elem : INTEGER) : IntListType;
VAR result, lastCell, newCell : IntListType;
INIT
  result := NIL;
LOOP
  NEW(newCell);
  newCell^.value := elem;
  IF result = NIL THEN
    result := newCell
  ELSE
    lastCell^.next := newCell;
  END;
  lastCell := newCell;
FINAL
  IF result ≠ NIL THEN lastCell^.next := NIL END;
  RETURN result
END MakeIntList;
```

Fig. 6. The *MakeIntList* accumulator.

```
ACCUMULATOR All (c : BOOLEAN) : BOOLEAN;
LOOP
  IF NOT c THEN RETURN FALSE END
FINAL
  RETURN TRUE
END All;
```

Fig. 7. The *All* accumulator.

The **ACCUMULATOR** construct also allows **RETURN** statements in the main accumulator loop, making it possible to return an accumulated value before all sequence values have been considered. This is primarily of use in *BOOLEAN* accumulators such as the *All* accumulator shown in Figure 7, which returns the conjunction of a series of *BOOLEAN* values, that is, returns *TRUE* only if *all* the sequence values are true. As soon as any sequence value is known to be *FALSE*, *FALSE* is immediately returned. *TRUE* is returned only when the sequence values have been exhausted and none of them have proved to be *FALSE*. The *Any* accumulator, which returns the disjunction of a series of *BOOLEAN* values, can be similarly programmed.

3.2 Accumulator Expressions

Accumulators are primarily designed to allow the concise expression of high-level iteration in the form of *accumulator* expressions, which have the following syntax in Modula Plus:

⟨accumulator-expression⟩ ::=
  ⟨*accumulator*-name⟩ "{" ⟨expression$_1$⟩ [**SUCH THAT** ⟨expression$_2$⟩]
    FOR ⟨iterator-list⟩ "}"

Such an expression results in the application of a given accumulator ⟨accumulator-name⟩ to the sequence of values of ⟨expression$_1$⟩ evaluated for each specified set of assignments to the loop control variables. The sets of assignments considered are those generated by the ⟨iterator-list⟩, optionally filtered by ⟨expression$_2$⟩. If this filter expression is present, a set of assignments to the loop control variables is used in the evaluation of ⟨expression$_1$⟩ only if ⟨expression$_2$⟩ evaluates to *TRUE* in the context of those assignments.

For example, the *Sum* accumulator defined above could be used to calculate the sum of all integers in an integer list.

Sum{*i* **FOR** *i* **IN** *IntElems*(*list*)}

Using a filter expression, only the sum of the elements in odd positions can be calculated.

Sum{*i* **SUCH THAT** *ODD*(*j*) **FOR** *i* **IN** *IntElems*(*list*), *j* **IN** *CountFrom*(1)}

Accumulation over another accumulation expression is also possible, giving doubly nested accumulations.

Sum{*Sum*{*x*[*i*, *j*] **FOR** *j* **IN** *Upto*(1, *Cols*)} **FOR** *i* **IN** *Upto*(1, *Rows*)}

This expression represents the sum of all elements in a two-dimensional array.

In these examples, note that accumulator expressions are analogous to standard mathematical notation. For example, the following mathematical form corresponds to the double accumulation above:

$$\sum_{i=1}^{Rows} \sum_{j=1}^{Cols} x_{ij}.$$

One benefit of accumulators, then, is to allow programs to use this common and widely accepted form of mathematical notation, changing only the surface syntax.

3.3 Literal Accumulations

In Modula Plus, *literal accumulations* provide a second way of using accumulators. In this case, an accumulator is applied to a sequence of individually specified values, simply using function-call syntax with an arbitrary number of "argument" expressions.

```
⟨literal-accumulation⟩ ::=
  ⟨accumulator-name⟩ "(" ⟨expression⟩ {"," ⟨expression⟩} ")"
```

For example, the following literal accumulations evaluate to 12 and −5, respectively:

Sum(7, 2, −5, 0, 8)
Minimum(7, 2, −5, 0, 8)

Of course, literal accumulations using the *Minimum* accumulator are much more useful than those involving *Sum*, given the existence of the dyadic + operator. In essence, literal accumulations allow many of the useful forms of "polyadic" functions in LISP[7] to be elegantly carried over into imperative languages.

3.4 *BOOLEAN* Accumulations

The *BOOLEAN* accumulators *Any* and *All* often allow concise formulation and use of truth conditions that must be calculated iteratively. For example, suppose that one of two actions is to be taken depending on whether all the integers in a given list are positive or not. This can be concisely expressed using an accumulator expression as follows:

```
IF All{x > 0 FOR x IN IntElems(list)}
  THEN Action1
  ELSE Action2
END
```

Without the higher-level iteration constructs, this loop could only be programmed at a much lower level.

```
StillPositive := TRUE;
WHILE StillPositive AND (tail ≠ NIL) DO
  StillPositive := tail^.value > 0;
  tail := tail^.next
END;
IF StillPositive
  THEN Action1
  ELSE Action2
END
```

Although one could hide the explicit data accesses using appropriate data-access functions, the overall level of programming would still not be raised much.

3.5 Accumulators, Data Structures, and Abstract Data Types

In addition to numeric and *BOOLEAN* accumulations, accumulations into data structures are another major application of the **ACCUMULATOR** construct. For example, using the *MakeIntList* accumulator defined above, a list of the squares of the first *n* integers can be calculated as the value of the following expression:

```
MakeIntList{i * i FOR i IN Upto(1, n)}
```

More interesting, perhaps, are cases in which sequence values are both generated from and accumulated into data structures. For example, calculation of a list of the averages of corresponding elements in two lists could be expressed as follows:

```
MakeIntList{(a + b)/2 FOR a IN IntElems(list1), b IN IntElems(list2)}
```

This example is remarkably similar to the equivalent expression in LISP using the *mapcar* mapping function.

```
(mapcar '(lambda (a b) (divide (plus a b) 2)) list1 list2)
```

With the appropriate definitions of a *Tails* iterator for generating the successive tail sublists of a given list and an *AppendLists* accumulator for concatenating a series of lists into a single list, the *maplist*, *mapcan*, and *mapcon* mapping functions can be simulated using iterator-accumulator loops as well.[1]

[1] The *mapl* and *mapc* functions do not perform any accumulation and hence correspond to **FOR** loops using iterators only.

```
DEFINITION MODULE IntLists;

TYPE IntListType;

PROCEDURE NoIntsQ(x : IntListType) : BOOLEAN
  (* This function returns TRUE iff a given list x is empty. *);

PROCEDURE IntCount(x : IntListType) : INTEGER
  (* This function returns the number of elements in a given list x. *);

PROCEDURE FirstInt(x : IntListType) : INTEGER
  (* This function returns the first integer in a given list. *);

PROCEDURE NthInt(x : IntListType; n : INTEGER) : INTEGER
  (* This function returns the integer at position n of a list. *);

PROCEDURE TailInts(x : IntListType) : IntListType
  (* This function returns the tail sublist of a given list after its
     first element. *);

PROCEDURE EmptyIntList() : IntListType
  (* This function returns an empty integer list. *);

PROCEDURE ConsInt(i : INTEGER; x : IntListType) : IntListType
  (* This function returns the list formed by adding an element to the
     beginning of a given list. *);

PROCEDURE Append1Int(x : IntListType; i : INTEGER) : IntListType
  (* This function returns the list formed by adding an element to the
     end of a given list. *);

ITERATOR IntElems (list : IntListType) : INTEGER
  (* This iterator generates the successive elements of a list. *);

ACCUMULATOR MakeIntList (elem : INTEGER) : IntListType
  (* This accumulator accumulates successive integers into a list. *);
END IntLists.
```

Fig. 8. The *IntListType* definition module.

In dealing with data structures, accumulators and iterators in Modula Plus are designed to carry over one of the most significant benefits of CLU iterators and Alphard generators, namely, elegant support for abstract data types. In this approach, an abstract data type is specified to include not only a collection of functions and procedures for the type, but also relevant iterators and accumulators. As in Modula-2, an abstract data type specification is summarized in Modula Plus with a **DEFINITION MODULE**, such as the integer list module shown in Figure 8. Following the Modula-2 approach, implementations for the declared entities are contained in a corresponding **IMPLEMENTATION MODULE** and may be used by any program through an **IMPORT** directive.

Iterators and accumulators enhance the concept of abstract data types in two ways. First, they provide a higher level of abstraction by allowing iterative computations to be expressed more concisely than otherwise possible. Second, they provide for efficient implementations of such iterative computations, often more efficient than equivalent loops without iterators and accumulators. Whether justified or not, programmers often forsake data abstraction for the sake of efficiency; efficient implementations can conversely promote its use.

Consider, for example, the following simple iterator-accumulator loop:

```
newlist := MakeIntList{x * x FOR x IN IntElems(list1)}
```

Without the use of iterators and accumulators, the most concise expression of this iteration is probably the following:

```
newlist := EmptyIntList( );
FOR i := 1 TO IntCount(list1) DO
  x := NthInt(list1, i);
  newlist := Append1Int(newlist, x * x)
END
```

The effort to read and understand that this loop is accumulating the squares of the elements of one list into another is certainly greater than that for the iterator-accumulator loop. Furthermore, given the representation for integer lists shown previously, the efficiency of this loop is much poorer than that of the iterator-accumulator loop. Within each iteration, the average cost of the *NthInt* and *Append1Int* operations is $O(n)$, where n is the length of the lists involved. The execution cost of the entire loop is thus $O(n^2)$. On the other hand, the cost of the iterator-accumulator loop is $O(n)$, since only a constant cost is involved in both the iterator and accumulator bodies for each list element considered.

3.6 In-Line Coding of Accumulators

As with simple iterator loops, loops involving accumulators are easily implementable via source-level in-line coding. In this case, however, the simplest strategy is to replace an accumulator loop by a function call that returns the result of the accumulation. The called function is created by combining together iterator and accumulator bodies to achieve the effect of the loop. Consider, for example, the following iterator-accumulator loop:

```
MakeIntList{(a + b)/2 FOR a IN IntElems(list1), b IN IntElems(list2)}
```

The in-line coding process would replace this loop by the following function call:

```
LoopFunction1(list1, list2)
```

Note that the parameter list for the loop function consists of the parameters of all the iterator calls in the original loop. The function created to implement this loop is shown in Figure 9. The body of this function is quite similar to the code generated for an iterator-only loop, except that the components of the accumulator definition have been added as well. The initialization statements of the accumulator are included with those of the iterators just before the main **LOOP**. An assignment statement simulating parameter passing to the accumulator parameter (*elem*, here) is included inside the main **LOOP** just after the code of the iterator bodies. Following this is the unmodified code of the accumulator body. The finalization code from the accumulator is included just after the **LOOP** so that it is executed when the **LOOP** terminates. Note that **RETURN** statements are carried over unmodified from the accumulator implementation to become the appropriate **RETURN** statements for the created loop function.

```
PROCEDURE LoopFunction1(x1, x2 : IntListType) : IntListType;
VAR tail1 : IntListType;
VAR tail2 : IntListType;
VAR elem : INTEGER;
VAR result, lastCell, newCell : IntListType;
BEGIN
  tail1 := x1;
  tail2 := x2;
  result := NIL;
  LOOP
    IF tail1 = NIL THEN EXIT
    ELSE
      i := tail1^.value;
      tail1 := tail1^.next
    END;
    IF tail2 = NIL THEN EXIT
    ELSE
      j := tail2^.value;
      tail2 := tail2^.next
    END;
    elem := (a + b) / 2;
    NEW(newCell);
    newCell^.value := elem;
    IF result = NIL THEN
      result := newCell
    ELSE
      lastCell^.next := newCell;
    END;
    lastCell := newCell;
  END;
  IF result ≠ NIL THEN lastCell^.next := NIL END;
  RETURN result
END LoopFunction1;
```

Fig. 9. Function created for in-line iterator-accumulator code.

4. EVALUATION

In proposing new constructs for a language, there must be some justification that the benefits they provide outweigh the costs of the additional language complexity incurred. Furthermore, if there are alternative language mechanisms that achieve similar objectives, then the cost–benefit trade-off for the proposed constructs must compare favorably with the alternatives.

This paper takes for granted that the work in functional languages has established the inherent benefits of high-level iteration constructs. The previous sections of this paper illustrate how these benefits may be carried over into imperative languages using iterators and accumulators. Although iterator-accumulator loops do lose some conciseness compared with their functional counterparts (especially with APL notation such as "+/V"), they nevertheless considerably improve the brevity and clarity of expression for high-level iterations in imperative languages. Furthermore, iterators and accumulators have additional benefits with respect to the efficiency of loop implementations and in the ability to program iterations for user-defined data abstractions.

It is also arguable that the addition of iterators and accumulators to imperative languages such as Modula-2 does not impose a great burden in terms of increased language complexity. This is primarily because iterators and accumulators integrate quite well with existing language mechanisms. As new types of program unit, the mechanisms for defining iterators and accumulators are reasonably consistent with the procedure definition mechanism. The Modula Plus **FOR** loop is a straightforward extension of the corresponding construct in Modula-2 to use iterators and to allow parallel loop variables. Accumulator expressions are a further extension of this but still consistent with the standard approach to iteration in imperative languages using control variables. Given these extensions, **FIRST** loops and literal accumulations are natural additions that add little complexity. In addition to this construct-level integration, the philosophy of iterators and accumulators integrates well with a central philosophical tenet of Modula-2, namely, support for data abstractions whose low-level implementation details can be hidden from the programs in which they are used.

4.1 Generator Expressions

Generator expressions as defined by Wile [9] represent a powerful alternative approach to high-level iteration for imperative languages. In this approach, generators are entities that produce a sequence of values, and generator expressions are constructs that denote generators. Generator expressions include notations for constructing basic generators for simple sequences and combining forms that define new generators in terms of others. The combining forms include mapping a generator sequence through a function, curtailing a sequence when a predicate becomes true, filtering a sequence using a predicate, combining sequences in parallel, concatenating sequences, accruing a sequence of partial accumulations, and extracting final sequence values. By allowing this arbitrary combination of generators, the generator expression notation becomes more powerful than the comparatively fixed iterative combinations that iterators and accumulators provide.

Unfortunately, the benefits of generator expressions are diminished somewhat by problems of implementation efficiency and come at a considerable cost in language complexity. From the efficiency viewpoint, there seems to be no straightforward general strategy for efficient implementation of generator expressions, although such implementations may be derivable by program transformation methods [9]. The additional language complexity imposed by generator expressions arises, first, from the complicated semantics of generators and, second, from the ability to use them in arbitrary combinations. Furthermore, the power of generator expressions does not seem to buy much; practical examples of high-level iteration generally use only a few of the generator expression combining forms at a time and can also be expressed using iterator-accumulator combinations. Essentially, generator expressions are too rich for practical inclusion in languages such as Modula-2.

4.2 Expressional Loops

Another alternative for incorporating high-level iteration into imperative languages is the *expressional loop* notation (XLoop) of Waters [8], which can be applied, for example, to the Ada programming language. This approach is based on the addition of *series* data types to the language; that is, for any given base type, a series type can be constructed that comprises the possible series of objects of that base type. The notation then provides for the definition of various kinds of functions that deal with series objects. *Enumerators* and *generators* can be defined as functions from nonseries values to, respectively, finite and infinite series objects. *Map* functions can be defined to take series values as input and to produce transformed series values as output. *Reducers* take a series as input and accumulate values from the series into a single result value. As with the generator expression approach, combining forms are also available for defining new series-processing functions in terms of existing ones.

The key advantage of expressional loops over generator expressions is that certain restrictions on the notation allow expressional loops to be directly reduced to equivalent iterative code using a preprocessor. The restrictions also simplify the semantics of series functions somewhat, which is a further benefit of the notation. Compared to the Modula Plus approach, however, the restrictions make the kind of efficient accumulation exhibited by the *MakeIntList* accumulator above difficult, essentially because the functions used to define reducers in XLoop must be side-effect free.

As with generator expressions, the XLoop notation provides somewhat greater expressive power than iterators and accumulators at the expense of extra language complexity. Again, the ability to arbitrarily combine series-processing functions gives rise to both this expressive power and complexity. In the XLoop approach, there is also additional language complexity because certain aspects of the notation do not integrate well into conventional imperative languages. In particular, the approach to iteration represented by XLoop conflicts with the conventional imperative approach using **FOR** loops; the result is that an imperative language incorporating the approach has two overlapping but noncomplementary mechanisms for iteration through sequences. Furthermore, the series data type constructor behaves quite differently from other data type constructors of imperative languages, defining data objects that do not have a physical existence at any point in time.

5. SUMMARY

Accumulators and iterators are practical and efficient mechanisms for carrying over into imperative languages the high-level iterative concepts widely found in functional languages. This paper has illustrated a design for such high-level iteration in the context of an extension to Modula-2 called Modula Plus. The design supports efficient iteration both because it is amenable to implementation via in-line coding and because it allows high-level iteration concepts to be implemented as encapsulations of efficient low-level manipulations. The design integrates well into Modula-2, building on its **FOR** loop construct; it should be easily adaptable to other imperative languages that have similar counter-based loops.

ACKNOWLEDGMENTS

My thanks to David Wile for pointing out the need for the **FINAL** statements in iterator implementations.

REFERENCES

1. BACKUS, J. W. Can programming be liberated from the von Neumann style?—A functional style and its algebra of programs. *Commun. ACM* 21, 8 (Aug. 1978), 613-641.
2. IVERSON, K. E. Operators. *ACM Trans. Program. Lang. Syst. 1*, 2 (Oct. 1979), 161-176.
3. JENKINS, M. A. *The Q'Nial Reference Manual.* Queen's University, Kingston, Ontario, 1983.
4. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564-576.
5. SCHWARTZ, J. T., DEWAR, R. B. K., DUBINSKY, E., AND SCHONBERG, E. *Programming with Sets—An Introduction to SETL.* Springer-Verlag, New York, 1986.
6. SHAW, M., WULF, W. A., AND LONDON, R. L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM* 20, 8 (Aug. 1977), 553-564.
7. STEELE, G. L., JR. *Common Lisp.* Digital Press, Burlington, Mass., 1984.
8. WATERS, R. C. Expressional loops. In *Conference Record of the 11th Annual ACM Symposium on the Principles of Programming Languages* (Salt Lake City, Ut., Jan 1984). ACM, New York, 1984, pp. 1-10.
9. WILE, D. S. *Generator Expressions.* Information Sciences Institute, Univ. of Southern California, Los Angeles, 1983. (Additional information is in the 1979 report of the same name.)

Received November 1985; revised August 1987 and January 1989; accepted, January 1989