Iteration Abstraction in Sather

STEPHAN MURER, STEPHEN OMOHUNDRO,
DAVID STOUTAMIRE, and CLEMENS SZYPERSKI
International Computer Science Institute

Sather extends the notion of an iterator in a powerful new way. We argue that iteration abstractions belong in class interfaces on an equal footing with routines. Sather iterators were derived from CLU iterators but are much more flexible and better suited for object-oriented programming. We retain the property that iterators are *structured*, i.e. strictly bound to a controlling structured statement. We motivate and describe the construct along with several simple examples. We compare it with iteration based on CLU iterators, cursors, riders, streams, series, generators, coroutines, blocks, closures, and lambda expressions. Finally, we describe experiences with iterators in the Sather compiler and libraries.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*control structures*; *coroutines*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*control primitives*

General Terms: Languages, Design

Additional Key Words and Phrases: General control structures, iteration abstraction, Sather

1. INTRODUCTION AND MOTIVATION

Sather is an object-oriented language developed at the International Computer Science Institute [Stoutamire and Omohundro 1995]. It has clean and simple syntax, parameterized classes, object-oriented dispatch (late binding), multiple inheritance, strong typing, and garbage collection. It was originally derived from Eiffel but aims to achieve the performance of C without sacrificing elegance or safety. The first version of the language ("Sather 0") was released in May, 1991. Feedback from users and our own use led to the design of "Sather 1" which incorporated a number of new language constructs. This paper describes Sather *iterators*, a form of iteration abstraction.

The original Sather had a fairly conventional `until ... loop ... end` statement. While this suffices for the most basic iterative tasks, we felt the need for a more general construct. As with C++, Sather 0 libraries made heavy use of *cursor* objects to iterate through the contents of container objects [Omohundro and Lim 1992]. While these work quite well in certain circumstances, they have a number of problems, described in detail in section 4. That section also describes approaches based on riders, closures, streams, series, generators, coroutines and blocks.

Authors' present addresses: Stephan Murer, Credit Suisse / Os1, CH-8070 Zurich, Switzerland; `stephan.murer@ska.com`. Stephen Omohundro, NEC Research Institute, Inc., 4 Independence Way, Princeton, NJ 08540; `om@research.nj.nec.com`. David Stoutamire, The International Computer Science Institute, 1947 Center St, Suite 600, Berkeley, CA 94704; `davids@icsi.berkeley.edu`. Clemens Szyperski, School of Computing Science, Queensland University of Technology, GPO Box 2434, Brisbane, QLD 4001, Australia; `c.szyperski@qut.edu.au`.

Like the language designers of CLU [Liskov and Guttag 1986], we felt a need to *encapsulate* the common operation of iterating through a data structure. Typical loops (such as Sather `until` loops) initialize some iteration variables and then repeatedly execute the body of the loop, updating the variables in some way, and testing for the end of the loop. Important examples of this pattern arise when stepping through the elements of container objects. The code for initializing, updating, and testing iteration variables is often complex and error prone. Errors having to do with the initialization or termination of iteration ("*fencepost*" errors) are very common. The code to step through complex containers such as hash tables typically must utilize the detailed internal structure of the container, sometimes causing duplication of virtually the same code in many places. Each of these observations argues for making the iteration operation a part of the interface of the container class rather than a part of the code in the client of the container. Another goal for the iterator design was to allow iterators to be programmed in an active style, without having to explicitly encode the control structure as state, as is required for cursors and most other iteration constructs.

Beyond its use in Sather, the iterator-based loop construct fits well into other structured programming languages. Its encapsulation of iterator states decouples separate iteration processes and allows them to be nested.

The name "*iterator*" and the initial design were derived from the iterator construct in the CLU language. A CLU iterator is like a routine except that it may "`yield`" in addition to returning. It may only be called in the head of a special "`for`" loop construct. The loop is executed once each time the iterator yields a value. Upon termination of the iterator, the loop exits. While CLU iterators can deal with the simplest iteration situations, such as stepping through the elements of arrays and other containers, they have several limitations which Sather iterators remove:–

—*One iterator per loop:* There is no simple way to step through two structures simultaneously in CLU.

—*No way to modify elements:* While CLU iterators support the retrieval of elements from a structure, there is no straightforward way to add or modify elements.

—*Iterator arguments are loop invariant:* There is no clean way to pass loop variant values to an iterator.

We wanted Sather iterators to retain the clean design of CLU iterators while removing these limitations. Similar to CLU, Sather iterators look like routines except that they may `yield` or `quit` instead of returning from a call. (Routines in Sather are the equivalent of methods [Goldberg and Robson 1985] or member functions [Ellis and Stroustrup 1990] in other object-oriented languages. The name "routine" comes from Sather's roots in Eiffel [Meyer 1988]. In Sather parlance, "method" means either a routine or an iterator.) Like C and Eiffel, method arguments in Sather are passed by value. Sather iterators extend CLU iterators in two important ways:–

— Multiple iterators may be invoked within a single loop, because they may occur as expressions anywhere in the loop and are not restricted to the loop head. This allows stepping through multiple structures simultaneously. The loop terminates as soon as the first of the iterators terminates.

— In contrast to CLU, Sather provides *hot iterator arguments* which are reevaluated each time the control is passed back to the iterator. Such arguments may be used to pass data to the iterator which varies on each iteration. In contrast to CLU iterators which may only generate a sequence of values, these arguments allow classes to define iterators that modify successive elements of a structure, i.e. to "consume" a sequence of values. In Sather , iterator arguments are hot unless they are declared with the `once` keyword, but `self` is never hot.

The rest of this paper is organized as follows. Section 2 introduces the Sather iterator syntax and gives some simple examples for motivation. Section 3 describes iterators in more detail. Section 4 compares iterators with other constructs that serve similar purposes. Section 5 describes experience with iterators in the Sather compiler and libraries.

2. EXAMPLES

The Sather loop statement has the simple form: "`loop ... end`". Iterators may only be called within loop statements. When an iterator is called, its body is executed until it either *quits* or *yields*. If it yields, any return value is returned to the loop and execution continues as if it were a routine call. The execution state of the iterator is maintained, however. The next call to the iterator will transfer control to the statement following the `yield`. The local variables and once arguments retain their previous values. When an iterator quits instead of yielding, the loop is immediately broken and execution continues with the statement which follows the loop. In order to be able to distinguish visually between iterator and routine calls, the language requires iterator names to end with an exclamation mark. This alerts the reader to all places where the control flow may change. This is helpful because iterators may occur as an expression anywhere in the loop.

2.1 Trivial Iterators

Every class is automatically provided with the three iterators: `while!(BOOL)`, `until!(BOOL)` and `break!` which may be used to obtain several standard forms of loop functionality. For example, `while!` is defined as:–

```
while!(pred:BOOL) is
   -- Yields while 'pred' is true, then quits.
   loop
      if pred then yield
      else quit
      end
   end
end
```

It may be used to obtain the standard "`while ... do`" behaviour:

```
i:=0;
loop while!(i<size);
   ...use i ...
   i:=i+1
end
```

The `while!` iterator takes a single Boolean argument which is evaluated on each iteration of the loop. As long as the argument evaluates to `true`, the iterator yields. This is an example of an iterator that yields without returning a value. It is merely used to control the loop. Once the argument evaluates to `false`, the iterator quits and breaks the loop. By placing the `while!` iterator at the end of the loop, a "`do ... while`" form is possible:–

```
i:=0;
loop
  ... use i ...
  i:=i+1;
  while!(i<size)
end
```

Note that the arbitrary placement of the `while` iterator also makes it easy to implement loops with a single conditional exit in their middle in a structured way (these are the so-called "$n1/2$ loops").

2.2 Integer Iterators

The integer class `INT` defines a number of useful iterators including `upto!`. In Sather iterators (and routines), the implicit argument `self` denotes the object on which the routine is invoked. In the `upto!` iterator, `self` is of type `INT`. (Keep in mind that `self` is always implicitly a once argument of any iterator.)

```
upto!(once limit:INT):INT is
  -- Yield successive integers from self up to and including 'limit'.
  r:INT:=self;
  loop
    while!(r<=limit);
    yield r;
    r:=r+1
  end
end
```

The `upto!` iterator can be used in place of the explicit initialization, increment, and termination test in the previous loops. For example, to sum the integers from 10 to 20, one might say:–

```
x:=0; loop x:=x + 10.upto!(20) end
```

A useful iterator for computing this kind of sum is:–

```
sum!(summand:INT):INT is
  -- Yield the sum of the previous values of 'summand'.
  r:INT:=0;
  loop
    r:=r+summand;
    yield r
  end
end
```

`sum!` can again be used in place of the explicit initialization in the previous loop, so the values could instead be summed by executing:–

```
loop x:=sum!(10.upto!(20)) end
```

This example also shows that iterators can be used as a part of expressions, just like functions. The results yielded by `upto!` are directly used as the arguments for `sum!`. Note that the same loop could be written more verbosely as:–

```
loop i:INT:=10.upto!(20); x:=sum!(i) end
```

2.3 Container Iterators

Most container classes in the Sather libraries define iterators to yield and modify the contained elements. While some containers such as arrays are very simple, other containers have complex implementation (such as hash tables, which are iterated over throughout the Sather compiler). The iteration construct used by client code appears the same for either container.

Arrays are variable size objects in Sather. `asize` returns the size of the current array and bracketed index expressions with values between 0 and `asize` $-$ 1 serve to access array elements. The `ARRAY{T}` class of arrays with elements of type `T` includes the following iterators:–

```
ind!:INT is -- Yield all indices of self.
   loop yield 0.upto!(asize-1) end
end;

elt!:T is -- Successively yield the elements of self.
   loop yield [ind!] end
end;

set!(x:T) is -- Set the elements to successive values of 'x'.
   loop [ind!]:=x; yield end
end
```

These are also examples of nested iterators, where the iterator `ind!` generates a stream of indices used by `elt!` and `set!` to index into the array. (Nesting iterators allows the formation of new iterators by abstracting from existing ones.) To set the elements of an array `a:ARRAY{INT}` to the constant value 7, you simply write:–

```
loop a.set!(7) end
```

To double the elements write:–

```
loop a.set!(2*a.elt!) end
```

If `b` is another object of type `ARRAY{T}`, we copy `a` into `b` (stopping at the end of the shorter of the two):–

```
loop b.set!(a.elt!) end
```

To compute the sum of the products of the elements of two such arrays:–

```
loop x:=sum!(a.elt!*b.elt!) end
```

Other examples providing `set!` and `elt!` composition iterators include sparse and distributed (cf. pSather [Murer et al. 1993]) arrays with complex underlying data structures. Although there are routines to access and write each element separately, the knowledge that a whole set of contiguous elements is to be written allows for a much more efficient implementation of the corresponding iterators compared to the single element accessor routines. Consider, for example, a block-wise distributed matrix with a central directory pointing to blocks which are matrices themselves. A row or column iterator for such a matrix yields all elements of the same block before consulting the directory for the next block containing elements of the same row or column. Accessing the row or column element-wise requires a directory access for each element leading to a potential bottleneck in a parallel implementation, or requiring the explicit implementation of suitable caching heuristics.

Many other classes similarly define iterators as part of their interfaces. For example, hash tables are able to yield their elements, and trees and graphs have iterators to yield their nodes in depth-first and breadth-first orders (see `inorder!` in Figure 2). Note that while simple array iterators are merely convenient, iterators over more complex data structures with a hidden internal representation are an indispensable tool for reasons of both efficiency and abstraction.

3. DETAILS OF THE ITERATOR CONSTRUCT

In the previous sections we informally introduced the iterator construct, the elements of which we will describe more precisely below. We begin by defining the key terms and conclude by defining the construct using these terms.

Loop statement. A control structure delimited using the keywords `loop` and `end`, causing repeated execution of the enclosed statements. Loop termination is controlled by iterators called from within the loop.

Iterator method. A method whose name ends in an exclamation point. Iterator methods can only be invoked from within loop statements. In addition to all constructs allowed within routines (except return statements), iterators may contain yield and quit statements and may have once arguments as described below.

Iterator call. A textual call to an iterator from within a loop statement. Denoted by the name of the iterator (which includes an exclamation point) followed by a list of arguments. An iterator call is always bound to the innermost lexically enclosing loop statement.

Once argument. Arguments of iterator methods marked with the `once` keyword at definition. The actual argument passed to a once argument is *not intended to change its value* after the first execution of the corresponding iterator call and before the corresponding loop statement terminates. Only the value obtained during the first call to the iterator is used in subsequent calls. To ensure a defined iteration state during loop execution, the only argument used for method dispatching, `self`, implicitly is a once argument. Once arguments allow an implementation to avoid redundant evaluations of corresponding iterator arguments in the calling context.

Hot argument. Arguments of iterator methods not marked with the `once` keyword. The expression for the actual argument is evaluated at every call.

Yield statement. The yield statement, denoted by the keyword `yield`, may only be used in the body of an iterator method. Its execution causes control and any return values to be passed back to the calling loop statement, resuming execution just after the iterator call.

Quit statement. The quit statement, denoted by the keyword `quit`, may only be used in the body of an iterator method. Its execution causes the corresponding loop statement to terminate immediately. Exiting from the body of an iterator method is considered an implicit execution of a quit statement.

Each textual iterator call maintains the state of execution of its iterator. When a loop is first entered, the execution state of all enclosed iterator calls is initialized. The first time each iterator call is encountered in the execution of the loop, each of the arguments is evaluated. On subsequent calls, however, once arguments retain their earlier values; only the expressions for hot arguments are re-evaluated. When an iterator is first called, it begins execution with the first statement in its body. If a yield statement is executed, control is passed back to the caller and the current value of the return parameters, if any, are returned. A subsequent call on the iterator resumes execution with the statement following this yield statement. If an iterator executes `quit` or reaches the end of its body, control passes immediately to the end of the enclosing loop in the caller. In this case no values are returned.

The interface of a class includes iterators on an equal footing with routines. As with routines, iterators may define conditionally compiled preconditions and postconditions. Preconditions are checked on each call to the iterator. Postconditions are checked when the iterator yields but not when it quits. As with routines, iterators may be defined in abstract classes which define interfaces that the compiler checks for conformance. Iterators may then be called by object-oriented dispatch, delaying the particular choice of iterator until runtime. This allows for abstract iteration over collections without knowing the implementing data structure at compile time.

Sather provides general non-resumable exception constructs. There is an important interaction between loop statements and exceptions. Since a loop statement bounds the lifetime of its enclosed iterator calls, its termination may involve some cleanup operations. For example, when a loop exits any space allocated for iterator calls must be deallocated. This terminating action of a loop statement has to be considered when allowing non-local exits such as exception raising. For this reason, Sather `protect` statements (which are similar to `try` statements in other languages) may only contain iterator calls if they also contain the surrounding loop. Similarly, `yield` is not permitted within a `protect` statement, which prevents the creation of dynamically protected regions which overlap instead of properly nesting in time.

4. COMPARISON WITH OTHER APPROACHES

We have discussed the ways in which Sather iterators generalize CLU iterators. In this section we compare Sather iterators with cursors, riders, streams, series, generators, coroutines, closures, and blocks.

4.1 Generalized Control Structures

The idea of generalizing iteration control structures goes back to early work such as the generators of IPL-V [Newell and Tonge 1960] or the generators and "possibility lists" of Conniver [McDermott and Sussman 1974]. Conniver includes activation records (called "frames") as first-class objects. It has a notion of pattern- or generator-defined possibility lists, where "`TRY_NEXT`" is used to get the next value from a list. Special tokens in possibility lists cause an associated generator to be invoked. This is a means for lazily computing lists of values. A generator yields new values and has the option of maintaining its state ("`AU_REVOIR`") or of quitting ("`ADIEU`"), similar to the `yield` and `quit` statements in Sather. Finally, the use of first-class frames allows generators to have side-effects in their caller environment. This can be used to simulate variable arguments and stream-consuming iterators.

However, experience with the "hairy control structures" of Conniver has been found to lead to unintelligible programs. We agree with Hewitt [1977, page 341] who found, "that we can do without the paraphernalia of 'hairy control structures' (such as possibility lists, non-local gotos, and assignments of values to the internal variables of other procedures in Conniver". As an alternative, Hewitt proposes lazy evaluation (using an explicit `delay` pseudo-function). While lazy evaluation allows for the effective handling of multiple recursive data structures, it also poses a particularly difficult problem for efficient implementations.

The Common Lisp `loop` macro [Steele Jr. 1990] is a generalized iteration control structure. While it contains about every iteration primitive that the authors could imagine (somewhat following the PL/1 tradition), all of these are built-in features ("loop clauses") of the "Loop Facility". The language definition explicitly states that "there is currently no specified portable method for users to add extensions to the Loop Facility". This prevents the use of the `loop` macro to support encapsulation of data structure specific iteration procedures.

4.2 Cursors, Riders, and C++ Iterators

As mentioned above, cursor objects are a way of encapsulating iteration without additional language constructs. *Riders* are a similar idea introduced in Oberon [Wirth and Gutknecht 1992] and generalized in Ethos [Szyperski 1992]. The idea is to define objects that point into a container class and may be used to retrieve successive elements. Their interfaces include routines to create, initialize, increment, and test for completion. The attributes of the cursor object maintain the current state of the iteration. This may be as simple as a single index into arrays, or as complex as a traversal stack or hash table recording the nodes that have already been visited for traversing trees or graphs. Note that Ellis and Stroustrup [1990] call the use of cursor objects in C++ "iterators".

We found that while cursors work quite well in certain circumstances, they can also become quite cumbersome. They require maintaining a parallel cursor object hierarchy alongside each container class hierarchy. Normally it is required to explicitly create and garbage collect cursor objects. Cursors can be semantically confusing since they maintain a location in a container for an indefinite period of time during which the container may change. Since the storage associated with a cursor is explicit, it is inconvenient to use them to describe nested or recursive control structures. Because cursors explicitly describe their implementation, they prevent a number of important optimizations on inner loops.

Iterators avoid these problems of cursors, because they are a part of the container class itself. The state of iterators is maintained only for the duration of a single loop. Iterators may be arbitrarily nested and support recursion just like routines. The iterator implementation manages the use of storage and can use the stack instead of the heap unless there are multiple recursive iterators.

Finally, even though the Sather language doesn't have explicit pointers, the array iterators can compile into efficient code based on pointer arithmetic.

4.3 Streams and Series

Iterators also share many characteristics with *streams* [Abelson et al. 1985]. One class of iterators are those of the form "`it!:T`" which have a return value but no arguments and yield a potentially infinite stream of values. Another class are those of the form "`it!(T)`" which have a single argument but no return value and which accept a potentially infinite stream of values. The way in which iterators suspend and transfer control when yielding corresponds well to the lazy evaluation semantics of streams. The main difference between iterators and streams is that on each invocation an iterator must consume one input and produce one output. Iterators are therefore always one-to-one mappings within a given loop.

The Sieve of Eratosthenes for generating successive prime numbers has been used to show the power of the stream concept [Abelson et al. 1985, pages 267–269]. While it is a conceptually simple algorithm, the control flow is rather complex. The stream solution is based on a stream which takes a stream argument and filters out later elements which are divisible by the first element. Iterators allow the following implementation (cf. Figure 1). Note that `d.divides(aprime)` in the example is a predicate that returns true if *d* divides *aprime* without remainder and false otherwise.

```
sieve!(aprime:INT):BOOL is
  -- Sieve out successive primes.
  d:INT:=aprime;
  yield true;
  loop
    if d.divides(aprime) then yield false
    else yield sieve!(aprime)
    end
  end
end;

primes!:INT is -- Yield successive primes.
  r:INT:=2;
  loop
    if sieve!(r) then yield r end;
    r:=r+1
  end
end
```

Fig. 1. Sieve of Eratosthenes

The iterator `sieve` tests the stream of values passed to it and yields `true` for the first value in this stream, `false` for all later multiples of this value, and recursively calls the next higher `sieve` for all other values. Feeding `sieve` with a stream of integers starting at 2 leads to a recursive iterator that yields `true` only on prime numbers. While this is not likely to be the most efficient or pedagogical way to implement the Sieve of Eratosthenes in Sather, it hints at the expressive power of iterators.

There is, however, an important difference between streams and iterators. Whereas streams may be passed around in a half-consumed state, the state of an iterator is confined to its calling "loop"-statement, and even more so, to a single call point within that loop. It is not possible to suspend iteration in one loop and to resume it with the same internal state in another loop. A variant of Sather under development at the University of Karlsruhe, Germany ("Sather-K" [Goos 1994]) does generalize iterators by introducing first class stream objects that retain their state across loop termination.

For Common Lisp the incorporation of *series*, *generators* and *gatherers* has been proposed for defining iterative constructs [Steele Jr. 1990]. These constructs are complex and include a large number of built-in operations. In Sather, these operations may be implemented with iterators and encapsulated in classes.

4.4 Coroutines and Generators

A different approach is to view all the iterators and the body of a loop as *communicating sequential processes* [Hoare 1985] tightly coupled by communication channels in the form of arguments and results of the iterators. Since there is neither preemption nor true parallel execution among iterators, we may model iterators and the loop body as *coroutines* [Wirth 1983].

More precisely, iterators may be thought of as *structured coroutines*. In many languages, coroutines can call other coroutines in an arbitrary fashion. Structured programming replaced the undisciplined transfer of control by "`goto`" statements with structured loop constructs. Iterators are more structured than coroutines with respect to the freedom in passing control. While a suspending coroutine may transfer control to any other waiting coroutine, in the case of iterators the flow of control is structured by the "loop"-statement. Iterators pass control back either to the point of call within the calling loop or to the end of that loop. They are initialized when the loop is entered and signal their return by breaking the loop.

Coroutines with this added restriction of always returning to their caller are sometimes called *generators*, *semicoroutines*, or *hierarchical coroutines* [Marlin 1980]. Iterators share this property. However, just as for streams, a generator's activation state is not bound to a single call point within a single loop. Instead, a generator may be left in a half-consumed state at one point and picked up at another. As mentioned in Section 4.3, this is not possible with iterators.

4.5 Blocks, Closures, and Lambda Expressions

Traditionally, iteration abstraction is supported in object-oriented languages by providing anonymous *blocks* [Goldberg and Robson 1985], lambda expressions [Abelson et al. 1985], or closures [Sussman and Steele Jr. 1975]. The *container classes* provide methods to apply a block to all or part of their elements. The execution of such block-based iterations is controlled by the container class. With iterators, the control is shared by the iterator and the calling loop. For example, either the iterator or the loop body may abort the iteration.

This difference in control becomes apparent when trying to iterate synchronously through multiple data structures. Consider the task of comparing the elements of two trees according to a pre-order traversal. This is the classical "same fringe problem" as defined by Hewitt [1977, page 344-347]. A simple solution using iterators is shown in Figure 2.

```
class TREE{T} is
  attr key:T;
  private attr left,right: TREET;
  inorder!:T is -- yields elements in order
    if void(self) then
      loop yield left.inorder! end;
      yield key;
      loop yield right.inorder! end
    end
  end;
  closed_inorder!:T is -- yields in order, then void
    loop yield inorder! end;
    yield void
  end;
  same_fringe(other:TREET):BOOL is
    -- returns whether 'self' and 'other' carry an
    -- equal ordered sequence ("fringe") of elements
    loop
      e1:T:=closed_inorder!;
      e2:T:=other.closed_inorder!;
      if void(e1) then return void(e2) end;
      if e1/=e2 then return false end
    end;
    return true
  end
end
```

Fig. 2. The *Same Fringe Problem*: traversing two binary trees in in-order sequence to determine whether they contain the same elements.

The iterator `inorder!` will yield each tree's elements in the proper order. Using a general technique for closing such iterators, iterator `closed_inorder!` uses `inorder!` to yield the same sequence, but yields `void` before quitting to indicate that the end of the structure has been reached. The `same_fringe` routine steps through the elements of both trees simultaneously, stopping when it is determined that either both trees have equal fringes, or a difference has been found, or one of the trees has a shorter fringe than the other one. If the trees kept track of their size, the `closed_inorder!` iterator would not be necessary.

In this kind of situation with more than one structure, it is not possible to pass the body of the routine to one of the trees for execution. Thus, in cases requiring the traversal of multiple structures, the use of blocks or closures is impractical, while the situation is easily handled by the iterator construct. This approach cannot be used in CLU, which allows only a single iterator per loop.

Closures can be used to implement generators; and multiple generators within a common loop can be used to traverse multiple structures simultaneously. (This is also possible in Sather using bound routines or bound iterators.) However, as for cursors, closures have the disadvantage of an unbounded lifetime of the closure state. While this may be compensated for by extensive compile-time analysis, the explicit association of iterators to loops solves this problem syntactically.

4.6 Summary of Comparisons[1]

The four constructs most commonly used for iteration are streams, cursors, built-in looping constructs (of which the Common Lisp variant is the most complex), and explicitly passing the loop body (as a closure, block, or otherwise).

The main points when comparing these constructs with Sather iterators are that cursors are harder to use and less suitable for compiler-based optimizations (but can be passed around), built-in loop constructs are not general enough, and passing closures does not effectively support the simultaneous traversal of multiple data structures. Streams and in particular generators come closest to Sather iterators, but are not bound to a specific loop.

5. EXPERIENCE

The real power of iterators can only really be seen in the context of a large system of classes. This section describes our experience with iterators in the freely distributed Sather compiler and libraries over the last year. The Sather 0 libraries contained several hundred classes, and iterators were used extensively in converting them to Sather 1. In many cases, the use of iterators allowed us to discover powerful new abstractions for interacting with a class. Because much of the iteration bookkeeping now occurs in the iterator definitions rather than in each loop, many classes became dramatically smaller. Iteration intensive classes such as those for vectors and matrices sometimes dropped to less than one-third their former size! Using iterators, the bodies of many routines could be reduced to a concise single line of code.

Table I. Number of iterators in loops in the Sather compiler and libraries (does not include built-in iterators.)

Number of iterators	0	1	>1
Number of loops	285	532	163
Percentage	29	54	17

The Sather 1 compiler is written entirely in Sather. Table I shows the number of iterators found in the 980 loops occurring in the 1.0.7 version of the Sather compiler and libraries, about 39000 lines of code. The built-in iterators `break!`, `until!` and `while!` are not counted, and could be replaced by the traditional constructs in other languages. The majority used at least one nontrivial iterator, validating the importance of iterators to this code. Only 17 percent used two or more within a loop, so CLU iterators could have been used for most cases. However, even in these cases Sather iterators eliminate the need for a distinguished loop header and allow iterators to be used directly as expressions, simplifying the code.

Because CLU iterators do not allow hot arguments, it isn't possible to express iterators such as `set!`. Of the loops, 151 (15 percent) used one or more iterators with hot arguments which could not have been expressed in CLU.

Cursor objects in any language have the problem that their semantics are usually not defined if their container is modified while they iterate. Sather iterators have the same problem, and many of the iterators in the libraries will fail if the underlying

[1]Based on a suggestion by one of the anonymous referees.

data structure is modified. There is presently no way for the compiler to detect such situations. Inserting extra runtime checks is also problematic; one must define conditions which indicate a misuse of the container iterators, for example by setting a flag while iterating and checking it whenever the container is modified. In pSather (the parallel extension to Sather) and concurrent systems in general, one has to ensure a reader-writer locking which is appropriate to the data structure. These solutions are tedious and error-prone. Fortunately, in our experience such bugs have not arisen frequently in practice, although this may not hold true for novice Sather programmers. We believe that this positive observation is largely due to the structured nature of iterators, guaranteeing a limited lifetime of the iteration state.

Iterators are a powerful construct and it is possible to write obscure and hard to understand code using them. The Sieve of Eratosthenes was an example of this. Novices have had the misunderstanding that iterator calls share state simply because they have the same name. For example, the code

```
loop
   #OUT + elt! + ", " + elt! + "\n"
end
```

prints two identical columns each with all the elements of `self`, rather than the elements distributed into two shorter columns[2]. Another pitfall has been the failure to foresee that in the following code, `foo` is evaluated one time more than `set!`, because it is `set!` which actually terminates the loop:–

```
loop b.set!(foo) end
```

Although we have seen these misconceptions arise in a number of individuals learning Sather, they need only be explained once.

Finally, although Sather supports higher-order functions, and hence programming in an applicative style, we have found that iterators often provide a cleaner solution. Iterators provide a convenient *lingua franca* for transmitting data between disparate data structures without having to allocate space for an intermediate container object such as an array or linked list.

6. CONCLUSIONS

We have presented Sather iterators, a new approach to encapsulating iteration, and have shown several simple examples of their use. Iterators eliminate common errors by combining initialization, progression and termination into one abstraction that does not need to be managed by client code. Iterators have proven very useful in practice, and are used extensively in both the standard Sather libraries as well as user code. We have found that by using iterators our code becomes simpler, easier to read, and less error prone. The interfaces to our classes become more concise and most cursor classes can be abolished. We are excited by the simplicity and power iterators bring to Sather and feel that other language designs could benefit as well.

Sather documentation and the compiler and libraries are available at

http://www.icsi.berkeley.edu/Sather

[2] In Sather-K, the latter could be expressed by reusing a stream object for both calls.

ACKNOWLEDGMENTS

Many people were involved in the Sather 1 design discussions. Ari Huttunen in particular made suggestions which improved the design of iterators. Jerry Feldman, Chu-Cheow Lim, and Heinz Schmidt also made useful suggestions. Furthermore, we would like to thank Urs Hölzle and Robert Griesemer for careful reading and for providing valuable comments on the structure of the paper. Last but not least we would like to thank the anonymous reviewers for numerous comments, which led to substantial improvements in the revised version of the paper.

REFERENCES

Abelson, H., Sussman, G. J., and Sussman, J. 1985. *Structure and Interpretation of Computer Programs.* MIT Press.

Ellis, M. A. and Stroustrup, B. 1990. *The Annotated C++ Reference Manual.* Addison-Wesley.

Goldberg, A. and Robson, D. 1985. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley.

Goos, G. 1994. Sather-K. Tech. Rep. 8/94, Faculty of Computer Science, University of Karlsruhe.

Hewitt, C. 1977. Viewing control structures as patterns of passing messages. *Artificial Intelligence 8*, 323–364.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice Hall.

Liskov, B. and Guttag, J. 1986. *Abstraction and Specification in Program Development.* MIT Press.

Marlin, C. D. 1980. *Coroutines: A Programming Methodology, a Language Design, and an Implementation.* Springer-Verlag, Berlin.

McDermott, D. V. and Sussman, G. J. 1974. The Conniver reference manual. Tech. Rep. Artificial Intelligence Memo 259a (May), MIT.

Meyer, B. 1988. *Object-oriented Software Construction.* Prentice-Hall.

Murer, S., Feldman, J. A., Lim, C.-C., and Seidel, M.-M. 1993. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Tech. Rep. TR-93-028 (December), International Computer Science Institute.

Newell, A. and Tonge, F. 1960. An introduction to Information Processing Language V. Tech. Rep. Paper P-1929, The RAND Corporation (presented at the ACM National Conference, Boston, MA 1959).

Omohundro, S. and Lim, C.-C. 1992. The Sather language and libraries. Tech. Rep. TR-92-017 (March), International Computer Science Institute.

Steele Jr., G. L. 1990. *Common LISP, The Language* (2nd ed.). Digital Press.

Stoutamire, D. and Omohundro, S. 1995. Sather 1.1. Tech. Rep. at http://www.icsi.-berkeley.edu/Sather, International Computer Science Institute.

Sussman, G. J. and Steele Jr., G. L. 1975. Scheme: An Interpreter for Extended Lambda Calculus. Tech. Rep. Artificial Intelligence Memo 349 (December), MIT.

Szyperski, C. A. 1992. *Insight* Ethos*: On Object-Orientation in Operating Systems*, Volume 40 of *Informatik-Dissertationen ETH Zürich.* Verlag der Fachvereine, Zurich.

Wirth, N. 1983. *Programming in Modula-2.* Springer.

Wirth, N. and Gutknecht, J. 1992. *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley.

Received ...; revised ... and ...; accepted ...