Sather Iters: Object-Oriented Iteration Abstraction

Stephan Murer, Stephen Omohundro, and Clemens Szyperski
The International Computer Science Institute
1947 Center St, Suite 600
Berkeley, CA 94704
Email: {murer, om, szyperski}@icsi.berkeley.edu

TR-93-045

August 1993

Abstract

Sather iters are a powerful new way to encapsulate iteration. We argue that such iteration abstractions belong in a class' interface on an equal footing with its routines. Sather iters were derived from CLU iterators but are much more flexible and better suited for object-oriented programming. 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 how to implement them in terms of coroutines and then show how to transform this implementation into efficient code.

1 Introduction and Motivation

Sather is an object-oriented language developed at the International Computer Science Institute [10]. It has clean and simple syntax, parameterized classes, object-oriented dispatch, multiple inheritance, strong typing, and garbage collection. It was originally derived from Eiffel but aims to achieve the performance of C or C++ without sacrificing elegance or safety. The initial release "Sather 0.2" of the compiler, debugger, class library, and development environment were made available by anonymous FTP (ftp.icsi.berkeley.edu) in May, 1991 and it was retrieved by several hundred sites. Feedback from these users and our own use has led to the design of "Sather 1.0" which incorporates a number of new language constructs. This paper describes *iters*, a new form of iteration abstraction.

Sather 0.2 had a fairly conventional `until ... loop ... end` statement. While this suffices for the most basic iterative situations, we felt the need for a more general construct. The Sather 0.2 libraries made heavy use of *cursor* objects to iterate through the contents of container objects [11]. While these work quite well in certain circumstances, they have a number of problems which are described in section 3. That section also describes approaches based on riders, closures, streams, series, generators, coroutines and blocks.

We also felt a need to *encapsulate* the common operation of iterating through a structure. Typical loops (such as Sather `until` loops) initialize some iteration variables and then repeatedly execute the body of the loop, update the variables in some way, and test for the end of the loop. An important simple example is the code to step through the elements of an array. 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 are sometimes called "*fencepost*" errors and they are very common. The code to step through more complex containers (such as hash tables) typically must rely on the detailed internal structure of the container. We also found that virtually the same code for stepping through structures was repeated in many places. Each of these points argues for making the iteration operation a part of the interface of a container class rather than a part of the code in a client of the class. Another goal for the iter design was the wish to allow iters to be programmed in an active style, i.e. without the need for inverting every control structure, as is required for cursors and most other constructs.

Beyond its application in Sather, the iter-based loop construct fits well into other block-structured programming languages. Its encapsulation of iter states decouples separate iteration processes and allows the nesting thereof.

The name "*iter*" (short for "iterator") and the initial design were derived from a construct in the CLU language [7]. A CLU iter is like a routine except that it may "`yield`" in addition to returning. They may only be called in the head of a special "`for`" loop construct. The loop is executed once each time the iter yields a value. When the iter returns, the loop exits. While CLU iters can deal with the

simplest iteration situations, such as stepping through the elements of arrays and other containers, they have several limitations which Sather iters remove:

- *One iterator per loop:* There is no simple way to step through two structures simultaneously.
- *No way to modify elements:* While CLU iters support the retrieval of elements from a structure, there is no straightforward way to add or modify elements.
- *Iter arguments are loop invariant:* There is no clean way to pass data which changes during a loop to an iter.
- *Dispatched iter calls:* CLU does not support object-oriented dispatch for routines or iters.

We wanted Sather iters to retain the clean design of CLU iters while removing these limitations. Similar to CLU, Sather iters look like routines except that they may `yield` or `quit` instead of returning. They also may mark one or more of their arguments as variable by appending an exclamation point to the type specifier of the argument. Marked arguments are evaluated each time they are encountered in the execution of a loop. Such arguments may be used to pass data to the iter which varies on each iteration. In contrast to CLU iters which may only generate a sequence of values, these arguments allow a class to define iters which modify successive elements of a structure, i.e. to "consume" a sequence of values.

The rest of this paper is organized as follows. Section 2 introduces the iter syntax and gives some simple examples for motivation. Section 3 compares iters with other constructs serving similar purposes. Section 4 introduces a basic implementation based on coroutines and uses this to define the operational semantics of iters. Finally, section 5 shows how to implement iters efficiently.

2 Simple Examples

The Sather loop statement has the simple form: "`loop ... end`". Iters may only be called within loop statements. When an iter 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 iter is maintained, however. The next execution of the iter call will cause execution of the statement following the last executed `yield` statement. The local variables and arguments not marked with exclamation points retain their previous values. When an iter quits instead of yielding, the loop is immediately broken and execution continues with the statement which follows the loop.

In practice, this construct is quite powerful, yet simple to use. Every class automatically inherits the three iters: `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 as long as 'pred' is true, quits once 'pred' is false.
  loop if pred then yield else quit end end end
```

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

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

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

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

Note that the arbitrary placement of the `while` iter also makes structured programming of loops with a single conditional exit in their middle, so-called "$n1/2$ loops", easy.

The `INT` class defines a number of useful iters including `upto!`. In Sather iters (and routines), the predefined local variable `res` is used to hold the value to be yielded (to be returned) next.

```
upto!(limit:INT):INT is
  -- Yield successive integers from self up to "limit".
  res:=self; loop while!(res<=limit); yield; res:=res+1 end end
```

To add up the integers from 10 to 20, one might say:

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

A useful iter for computing this kind of sum is:

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

Using this, the values may be added with:

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

(Note that this is equivalent to writing:

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

This explains the meaning of iter calls within expressions used as variable arguments in other iter calls.) With a similar iter for products, 20! can be computed with:

```
loop f:=product!(1.upto!(20)) end
```

Most container classes define iters to yield and modify the contained elements. For example, the `ARRAY{T}` class defines:

```
elts!:T is
  -- Successively yield the elements of self.
  loop res:=self[0.upto!(asize-1)]; yield end end;

set_elts!(x:T!) is
  -- Set the elements of self to successive values of "x".
  loop self[0.upto!(asize-1)]:=x; yield end end
```

These are also examples of using nested iters, where the iter `upto` generates a stream of indices used by `elts` and `set_elts` to index the array. (Nesting iters allows the formation of new iters by abstracting from existing ones.) Then, to set the elements of an array `a:ARRAY{INT}` to the constant value 7, you simply write:

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

To double the elements:

```
loop a.set_elts!(2*a.elts!) end
```

To make `b` be a copy of `a`:

```
loop b.set_elts!(a.elts!) end
```

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

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

Other classes similarly define iters as part of their interfaces. For example, hash tables can yield their elements and trees and graphs have iters to yield their nodes in depth-first and breadth-first orders.

The interface of a class includes iters on an equal footing with routines. As with routines, iters may define conditionally compiled preconditions and postconditions. Any preconditions are checked on each call to the iter. Postconditions are checked when the iter yields but not when it quits. As with routines, iters may be called by object-oriented dispatch, delaying the particular choice of iter until runtime. Also,

Sather has a construct called "*bound routines*" which is similar to closures. It binds together a reference to a routine and values for some of its arguments for later call. Bound iters allow many of the powerful constructs based on higher-order functions to be applied to iters. For example, it is easy to define iters which concatenate two argument iters or filter iters which yield only the values of an argument iter which satisfy a predicate.

Iters 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 more structured loop constructs. In a similar way, Sather iters pass control back either to the point of call or to the end of the calling loop. They are initialized when the loop is entered and signal their return by breaking the loop.

3 Comparison with Other Approaches

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

3.1 Generalized Control Structures

The idea of generalizing iteration control structures goes back to early work such as the generators and "possibility lists" of Conniver [9]. 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 such 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 iters.

However, experience with the "hairy control structures" of Conniver [9] has been found to lead to unintelligible programs. We agree with Hewitt who has 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" [5, page 341]. As an alternative, Hewitt proposes lazy evaluation (using an explicit `delay` pseudo-function). While lazy evaluation allows the efficient handling of multiple recursive data structures, it also poses a particularly difficult problem for efficient implementations.

The Common Lisp `loop` macro [12] 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.

3.2 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 [16] and generalized in Ethos [14]. 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 and "seen-node" hash table for traversing trees or graphs. Note that Ellis and Stroustrup [3] 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. Cursors often require the explicit creation and garbage collection of the 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 un-handy 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.

Compared to the problems of cursors, iters are a part of the container class itself. The iter implementation manages the use of storage and can often use the stack instead of the heap. The state of iters is confined to a single loop. Iters maybe arbitrarily nested and support recursion just like routines. Finally, even though the Sather language doesn't have explicit pointers, the array iters compile into efficient code based on pointer arithmetic.

3.3 Streams, Series, and Generators

Iters also share many characteristics with *streams* [1]. One natural class of iters are those of the form "`it!:T`" which have a return value but no arguments and yield a potentially infinite stream of values. Another natural 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 iters suspend and transfer control when yielding corresponds well to the lazy evaluation semantics of streams.

The Sieve of Eratosthenes for generating successive prime numbers has been used to show the power of the stream concept [1, 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. This solution may be implemented using bound iters. However, the iter semantics allows the following much simpler implementation (cf. Fig. 1).

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

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

Figure 1: Sieve of Eratosthenes

The iter `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 calls the next higher `sieve` for all other values. Feeding `sieve` with a stream of integers starting at 2 leads to a recursive iter that yields `true` only on prime numbers. While this is not likely to be the most efficient way to implement the Sieve of Eratosthenes in Sather, it hints at the expressive power of iters.

There is, however, an important difference between streams and iters: Whereas streams may be passed around in a half-consumed state, the state of an iter is confined to its calling "loop"-statement. It is not possible to suspend iteration in one loop and to resume it with the same internal state in another loop.

Common Lisp [12] is considering the incorporation of *series* or *generators* as two proposals for defining iterative constructs. These constructs appear to require special compilation and have a rather complex semantics. They include a large number of built-in operations. These operations may each be implemented with iters and encapsulated in classes.

3.4 Coroutines

A different approach is to view all the iters and the body of a loop as *communicating sequential processes* [6] tightly coupled by communication through the arguments and results of the iters. Since there is neither preemption nor true parallel execution among iters, we may model iters and the loop body as *coroutines* [15] (section 4). Iters 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, the flow of control is structured by the "loop"-statement in the case of iters.

3.5 Blocks, Closures, and Lambda Expressions

Traditionally, iteration abstraction is supported in object-oriented languages by providing anonymous *blocks* [4], lambda expressions [1], or closures [13]. 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 iters, the control is shared by the iter and the calling loop. For example, either the iter 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 [5, page 344-347]. A simple solution using iters is shown in Fig. 2.

```
class TREE{T} is
  attr key:T
  attr left,right: SAME

  inorder!:T is    -- yields elements in order
    if self \= void then
      loop res:=left.inorder!; yield end;
      res:=key; yield;
      loop res:=right.inorder!; yield end
    end
  end

  closed_inorder!:T is    -- yields elements in order, then yields void
    loop res:=inorder!; yield end;
    res:=void; yield
  end

  same_fringe(other:SAME):BOOL is
    -- returns whether 'self' and 'other' carry an equal
    -- ordered sequence ('fringe') of elements
    loop e1:T:=inorder!; e2:T:=other.inorder!;
      until!((e1=void) or (e1\=e2))
    end;
    res := e1=e2
  end
end
```

Figure 2: The Same Fringe Problem

The iter `inorder` will yield each tree's elements in the proper order. Using a general technique for closing such iters, iter `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. (In a Sather library one could have a generic iter that can be used to close arbitrary iters this way.) The `same_fringe` routine steps through the elements of both trees simultaneously, stopping when 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.

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, this use of blocks or closures is impractical, while the situation is easily handled by the iter construct. This advantage cannot be materialized 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 iters.) 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 binding of iters to loops solves this problem syntactically.

4 Operational Semantics and Basic Implementation

In this section we show how to translate the Sather iter construct to an assumed C++-like language supporting coroutines. This approach allows us to semi-formally define the operational semantics of iters in terms of the well-understood coroutine semantics [8]. In a later section we show how various optimization steps may be used to eliminate the coroutines, leading to more efficient iter implementations in many important cases.

4.1 Elements of the Iter Construct

In the previous sections we informally introduced the iter construct, the elements of which we will describe more precisely below.

- *loop statement*: A control structure delimited using the keyword pair `loop ... end` causing repeated execution of the enclosed statements. Loop termination is controlled by iters called from within the loop.

- *iter method*: A routine qualified by an exclamation point following its name. Iter methods can only be invoked from within a loop statement. Additionally to all constructs allowed within a plain routine but return statements, an iter method may contain yield and quit statements and may have constant parameters as described below.

- *iter call*: A textual call to an iter from within a loop statement. Denoted by the name of the iter followed by a list of arguments.

- *constant parameter*: Parameters of iter methods of a type that is not marked using an exclamation point. The argument passed to a constant parameter is *supposed to not change its value* after the first execution of the corresponding iter call and before the corresponding loop statement terminates. To ensure a defined iteration state during loop execution, parameters used for method dispatching are implicitly assumed to be constant parameters. Constant parameters allow an implementation to avoid redundant evaluations of corresponding iter arguments.

- *yield statement*: The yield statement, denoted by the keyword `yield`, may only be used in the body of an iter method. Its execution causes return parameters and control to be passed back to the calling loop statement, resuming execution just after the iter call.

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

Each textual iter call maintains the state of execution of its iter. When a loop is first entered, the execution state of all enclosed iter calls is reinitialized. The first time each iter call is encountered in the execution of the loop, each of the arguments is evaluated. On subsequent calls, however, constant parameters retain their earlier values. Only the expressions for arguments which are marked with an exclamation point are re-evaluated. When an iter is first called, it begins execution with the first statement in its body. If a yield statement is executed, control is returned to the caller and the current values of return parameters, if any, are returned. A subsequent call on the iter resume execution with the statement following this yield statement. If an iter 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.

4.2 Mapping Iters to Coroutines

We begin by considering non-dispatched calls to iters, i.e. ones close to routine calls, and add dispatched iter calls only later. Also, we begin by ignoring constant parameters.

Consider the following Sather loop statement containing three statements S_1, S_2, and S_3 not containing iter calls, plus two calls to iters I_1 and I_2 passing arguments A_1 and A_2, respectively.

`loop` S_1 `;` $I_1(A_1)$ `;` S_2 `;` $I_2(A_2)$ `;` S_3 `end`

This is translated into the following code fragment:

```
int iter1(PARAMS_I1, coroutine) { ...}
int iter2(PARAMS_I2, coroutine) { ...}

void loop(int brk)
{ coroutine it1 = new_coroutine(iter1);
  coroutine it2 = new_coroutine(iter2);
  while(true) {
    S_1; it1->transfer(A_1, current_coroutine);
    if(brk) break;
    S_2; it2->transfer(A_2, current_coroutine);
    if(brk) break;
    S_3;
  }
}
```

The statements S_1, S_2, and S_3, the iter parameter declarations $PARAMS_{I1}$ and $PARAMS_{I2}$, and the iter arguments A_1 and A_2 need to be expanded to reflect the actual Sather code. A new coroutine activation is created using `new_coroutine` which takes the implementing routine as an argument. The call `c->transfer` transfers control to the coroutine `c`. Here, we assumed that a coroutine declared with parameters expects these parameters to be passed with each transfer of control to it. Executing the loop corresponds to creating a coroutine with routine `loop` and transferring to it.

The extension to more than two iter calls and more complex structures of the loop body is straightforward: Just insert the appropriate coroutine transfer for each iter call.

Next, we need to show how to translate iter methods into coroutines. Consider the Sather iter `M` with some parameter list `pars`:

```
M!(PARAMS) is S_1; yield; S_2; quit; S_3 end;
```

This is translated into a coroutine `cm`:

```
int cm(pars, coroutine loop)
{ S_1; loop->transfer(1);
  S_2; loop->transfer(0);
  /* this point is never reached:  S_3 is never executed in M */
}
```

4.3 Mapping Constant Parameters

In a next step, we introduce constant parameters. Clearly, constant parameters do not effect the translation of iter methods, but merely that of loop statements, or more precisely, that of loop statements containing calls to iters with constant parameters. Consider an iter method `IO` with a split parameter list, the first part being constant parameters:

```
IO!(CONST_PARAMS; PARAMS);
begin
  S1; yield; S2; quit; S3
end;
```

The following loop statement contains a call to an iter I_{const} having constant parameters A_{const} in addition to regular parameters A. For the sake of simplicity, the loop contains only a single iter call.

```
loop S1; I_const(A_const, A); S2 end
```

This loop translates into the coroutine shown below. The idea is to use a freshly introduced local variable `params` to hold copies of the constant arguments, and to unroll the first loop iteration. Thereby, the original arguments A_{const} are indeed evaluated once and in the correct context. Subsequent calls of the iter method use the cached value in `params`.

```
int itero(CONST_PARAMS_IO, PARAMS_IO, coroutine) { ...}

void loop(int brk)
{ CONST_PARAMS_IO params;
  coroutine ito = new_coroutine(itero);
  S1; params = A_const; ito->transfer(params, A, current_coroutine);
  while(true) {
    S1; it1->transfer(params, A, current_coroutine);
    if(brk) S2;
  }
}
```

Again, S_1, S_2, $PARAMS_{IO}$, etc. need to be filled in.

4.4 Mapping Dispatched Iter Calls

In a final step we generalize our construct by allowing dispatched iter calls. At this point the introduction of constant parameters becomes essential: A parameter that

an iter call is dispatched on must follow the semantics of constant arguments. Otherwise, each loop iteration could dispatch to a different iter implementation, effectively destroying our iter semantics.

Note that while the Sather language uses single dispatch methods only, the iter mechanism would readily fit into other languages based on multiple dispatch (sometimes called multi-methods [2]). Dispatched iters are implemented by relying on the special semantics of constant parameters. When first evaluating the dispatching arguments, the corresponding iter implementation is located, and the coroutine created. Subsequent transfers always refer to the same iter implementation, hence making the dynamic dispatch for these superfluous.

The support of object-oriented dispatch enables iters to be used to couple objects of abstract types within a loop construct. Thus, providing sufficiently general iters for generating and consuming elements of abstract container classes allows these iters to take the role of a *lingua franca* among such classes.

4.5 Exception Handling and Iters

Exception handling is not directly within the scope of this paper. However, there is an important interaction between loop statements and exceptions. Since a loop statement bounds the lifetime of its enclosed iter calls, its termination may involve some cleanup operations. For example, if the iter calls are directly implemented using coroutines, the associated work-spaces have to be deallocated. This terminating action of a loop statement has to be considered when allowing non-local exits such as exception raising.

Another subtle problem results if an iter raises an exception that is caught within the corresponding loop statement. An example in pseudo-Sather is shown in Fig. 3.

```
bad_guy! is
  exception:=#MY_EXCEPTION; raise end

troubled_guy is
  loop
    protect bad_guy! against MY_EXCEPTION then end
  end
end
```

Figure 3: Interaction of Iters and Exceptions

This situation causes many semantical difficulties. For example, should the aborted iter be restarted the next time around the loop? If so, in what state? To avoid this dilemma, the static semantics of Sather does not allow catching exceptions between iter calls and corresponding loop statements.

5 Optimized Implementations

Clearly, the basic implementation described above and based on coroutines covers all cases, but is not efficient enough to replace lightweight looping constructs. For example, the simple Sather code used to add two vectors

```
loop c.set_elts(a.elts + b.elts) end
```

should really be compiled into something close to what the following C++ fragment would:

```
{ int l = min(a.asize, b.asize, c.asize);
  /* once allocated, the size of Sather arrays is invariant */
  for(int i=0, i<l, i++) {c[i] = a[i] + b[i]}
}
```

In other words, whenever iters are used to perform a simple task, it is important to get rid of all routine calls.

In this section we introduce a quite general way to get rid of non-dispatched, non-recursive iter calls. Thereafter, we use a simple example to show that this technique indeed optimizes a very common iter form. Finally, we give some hints on how to optimize the implementation of recursive and dispatched iters.

5.1 Optimizing Non-recursive Non-dispatched Iters

To begin with, consider the above vector addition loop and the translation to coroutines (cf. section 4). An iter coroutine has exactly $j + 1$ entry points if it contains j yield statements. This can be used to transform an arbitrary non-recursive iter coroutine into a plain routine: The entry point to be used is considered the state of a finite state machine. The resulting routine has an additional input and an additional output parameter to be used to signal the entry point to use and the one to be used on the next iteration, respectively. Also, the local state of the iter has to be anchored in the caller's space and passed to the iter routine as an additional parameter.

After transforming a non-recursive iter into a plain routine containing a state machine, this can be inlined into the loop statement if the iter call is non-dispatched (or the dispatch can be statically eliminated by the compiler). To allow further optimizations of the resulting loop code, the Cartesian product of the individual state spaces is formed. While incrementally adding one iter state space after the other to the Cartesian product, unreachable product states can be eliminated. In turn, the loop is expanded by inserting the single minimized product state machine. Finally, product states that are only reached once when beginning the iteration (or just before exiting it) can be moved out of the iteration.

For further illustration we show how to apply this technique to a particularly important form of iters, i.e. the ones conforming to the following pattern, where P,

U, V, W, and Q are arbitrary non-yielding Sather statement sequences, and $COND$ is an arbitrary Sather expression:

```
I! (PARAMS) is
  P;
  loop U;
    if COND then V; yield; W end;
    Q
  end
end;
```

Note that all iters given in section 2 are covered by this pattern, as are many of the important iters in the Sather library.

Iters of this form have exactly two entry points, corresponding to two states of the resulting state machine. The state machine is entered in state 0 and switches to state 1 after the first iteration, where it stays until the loop terminates. The actions corresponding to the two states are:

```
/* State 0: */
  P; U; if(COND) V;
  while(!COND) Q;
  state = 1;

/* State 1: */
  W;
  do Q until(COND);
  V;
```

In the following Sather loop, assume that IT_1 and IT_2 are arbitrary non-dispatched iter calls to iters of the form introduces above, while S_1, S_2, and S_3 are arbitrary Sather statement sequences not containing iter calls:

```
loop S1; IT1; S2; IT2; S3 end
```

We notice that the state machines corresponding to IT_1 and IT_2 both begin in state 0 and after the first iteration are both in state 1 where they remain till the loop terminates. Hence, the Cartesian product states (0,1) and (1,0) are unreachable. The resulting loop can be unwound once to move code corresponding to states 0 out.

```
  S1; IT1(state0); S2; IT2(state0); S3; /* evaluate constant arguments */
  while(true) { S1; IT1(state1); S2; IT2(state1); S3 }
exit:  ...
```

Since each of the statements could lead to a loop termination, we assume that every occurrence of `quit` in IT_1 or IT_2 is replaced by `goto exit`.

Also, the generalization to more than two iter calls is straightforward. Care must be taken if the loop body is structured in a more complicated way: In this case some of the product states eliminated above may actually be reachable.

Our current implementation work is focussing on the optimization of recursive iters and some of the more complex loop structures. In appropriate circumstances, recursive iters can use more efficient stack structures than the direct coroutine implementation would.

6 Conclusions

We have presented Sather iters as a new construct for encapsulating iteration and shown several simple examples of its use. The real power, however, can only really be seen in the context of a large system of classes. The Sather 0.2 libraries contained several hundred classes and we have used iters extensively in converting them to Sather 1.0. In many cases, the use of iters allowed us to discover powerful new abstractions for interacting with a class. Because much of the iteration bookkeeping now occurs in the iter definitions rather than in each loop, many classes got dramatically smaller. Iteration intensive classes such as those for vectors and matrices sometimes dropped to less than one-third their former size! Using iters, the bodies of many routines were reduced to a single line of code.

In addition to the examples described in this paper, we have found many other ways of using iters. For example, they provide a natural *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. We are finding that by using iters our code gets simpler, easier to read, and less buggy. The interfaces to our classes become more concise. In practice, the construct is easy to use and understand. We are excited by the simplicity and power it brings to Sather and feel that other languages could similarly benefit from it.

7 Acknowledgements

Many people were involved in the Sather 1.0 design discussions. Ari Huttunen in particular made suggestions which improved the design of iters. Jerry Feldman, Chu-Cheow Lim, Heinz Schmidt, and David Stoutamire also made useful suggestions. Last but not least, we would like to thank Urs Hölze for careful reading and valuable comments on the structure of the paper.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Craig Chambers. Object-oriented Multi-Methods in Cecil. In *Proceedings of the Sixth European Conference on Object-Oriented Programming (ECOOP'92)*, Utrecht, 1992.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[4] Adele Goldberg and David Robson. *Smalltalk-80, The Language and its Implementation.* Addison-Wesley, 1985.

[5] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

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

[7] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

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

[9] Drew V. McDermott and Gerald Jay Sussman. The Conniver reference manual. Technical Report Artificial Intelligence Memo 259a, MIT, May 1974.

[10] Stephen Omohundro. Sather provides nonproprietary access to object-oriented programming. *Computers in Physics*, 6(5):444–449, 1992.

[11] Stephen Omohundro and Chu-Cheow Lim. The Sather language and libraries. Technical Report TR-92-017, International Computer Science Institute, March 1992.

[12] Guy L. Steele Jr. *Common LISP, The Language.* Digital Press, 2 edition, 1990.

[13] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for Extended Lambda Calculus. Technical Report Artificial Intelligence Memo 349, MIT, December 1975.

[14] Clemens A. Szypersky. *Insight* ETHOS*: On Object-Orientation in Operating Systems*, volume 40 of *Informatik-Dissertationen ETH Zürich.* Verlag der Fachvereine, Zurich, 1992.

[15] Niklaus Wirth. *Programming in Modula-2.* Springer, 1983.

[16] Niklaus Wirth and Jürg Gutknecht. *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley, 1992.