**520 — Principles of Programming Languages**

**33: Iterators**

Christian Collberg  
collberg@cs.arizona.edu  
Department of Computer Science  
University of Arizona

---

**CLU-style Iterators**

Iterators were pioneered by CLU, a (dead) class-based language from MIT.

```plaintext
setsum = proc(s:intset) returns(int)
  sum : int := 0
  for e:int in intset$elmts(s) do
    sum := sum + e
  end
  return sum
end setsum
```

---

**CLU-style Iterators...**

- Procedure *setsum* computes the sum of the elements in a set of integers.
- *setsum* iterates over an instance of the abstract type *intset* using the *intset$elmts* iterator.
- Each time around the loop, *intset$elmts* yields a new element, suspends itself, and returns control to the loop body.

```plaintext
intset = cluster is create, elmts,...
  rep = array[int]
  elmts = iter(s:cvt) yields(int)
    i : int := rep$low(s)
    while i <= rep$high(s) do
      yield (s[i])
      i = i + 1
end
end elmts
end intset
```
A CLU cluster is a typed module; a C++ class, but without inheritance.

CLU makes a clear distinction between the abstract type (the cluster as seen from the outside), and its representation (the cluster from the inside). The rep clause defines the relationship between the two.

---

```
elmts = iter(s:cvt) yields (int)
i : int := rep$low(s)
while i <= rep$high(s) do
  yield (s[i])
i = i + 1
end
end elmts
```

---

`s:cvt` says that the operation converts its argument from the abstract to the representation type.

`rep$low` and `rep$high` are the bounds of the array representation.

`yield` returns the next element of the set, and then suspends the iterator until the next iteration.

Iterators may be nested and recursive.

---

```
array = cluster [t: type] is ...
elmts = iter(s:array[t]) yields (t)
  for i:int in int$from(to(
    array[t]$low(a),
    array[t]$high(a)) do
    yield (a[i])
  end
end elmts
end array
```

---

```
array = cluster [t: type] is ...
elmts = iter(s:cvt) yields (int)
  for i:int in array$elmts(s) do
    yield (i)
  end
end elmts
```
CLU-style Iterators...

- Iterators may invoke other iterators.
-CLU supports constrained generic clusters (like Ada's generic packages, only better).

Iterator Implementation

Iter1 = iter ( ... )
... yield x
(1) ...
end
end Iter1
P = proc ( ... )
  for i in Iter1(...) do
    S
  end
end P

Calling an iterator is the same as calling a procedure. Arguments are transferred, an activation record is constructed, etc.

Returning from an iterator is also the same as returning from a procedure call.
Iterator Implementation...

- When an iterator yields an item, its activation record remains on the stack. A new activation record (called a resume frame) is added to the stack.
- The resume frame contains information on how to resume the iterator. The return address-entry in the resume frame contains the address in the iterator body where execution should continue when the iterator is resumed.

Nested Iterators...

- Since iterators may be nested, a procedure may have several resume-frames on the stack.
- A new resume frame is inserted first in the procedure’s iterator chain.
- At the end of the for-loop body we resume the first iterator on the iterator chain:
  1. The first resume frame is unlinked.
  2. We jump to the address contained in the removed frame’s return address entry.

```plaintext
for i in Iter1(...) do
  for j in Iter2(...) do
    S
  end
end
```

When we get to the end of Iter2’s body we return as from a normal call. Iter1 may generate a new item and P may again start up Iter2.
Simpler Iterator Implementation

```plaintext
Iter = iter ( ... )
    while ... do
        yield x
    end
end

begin
    for i in Iter(...) do
        print(i);
    end
end
```

Simpler Iterator Implementation...

```plaintext
PROCEDURE Iter ( 
    Success, Fail : LABEL;
    VAR Resume : LABEL; VAR Result : T);
BEGIN
    WHILE ... DO
        ResumeLabel:
            Result := x;
            Resume := ADDR(ResumeLabel);
            GOTO Success
    END;
    GOTO Fail;
END
```

Simpler Iterator Implementation...

```plaintext
VAR Result : T;
VAR Resume : LABEL;
BEGIN
    Iter(ADDR(SuccesLabel), ADDR(FailLabel),
        Resume, Result);
    SuccessLabel:
        WRITE Result;
        GOTO Resume;
    FailLabel:
END;
```

Icon Generators

Procedures are really generators; they can return 0, 1, or a sequence of results. There are three cases

- **fail** The procedure fails and generates no value.
- **return e** The procedure generates one value, e.
- **suspend e** The procedure generates the value e, and makes itself ready to possibly generate more values.

```plaintext
procedure To(i,j)
    while i <= j do {
        suspend i
        i+:= 1
    }
end
```
Readings and References

4. Todd A. Proebsting: *Simple Translation of Goal-Directed Evaluation*, PLDI’97, pp. 1–6. This paper describes an efficient implementation of Icon iterators.

Summary

- Sather (a mini-Eiffel) has adopted an iterator concept similar to CLU’s, but tailored to OO languages.
- Iterators function (and can be implemented as) coroutines. Smart compilers should, however, take care to implement “simple” iterators in a more direct way (See the Sather paper).
- Inline expansion of iterators may of course be helpful, but the same caveats as for expansion of procedures apply: code explosion, cache overflow, extra compilation dependencies.