A Technique for Generic Iteration and Its Optimization

Stephen M. Watt

Department of Computer Science
University of Western Ontario
London Ontario, Canada N6A 5B7
watt@csd.uwo.ca

Abstract

Software libraries rely increasingly on iterators to provide generic traversal of data structures. These iterators can be represented either as objects that maintain state or as programs that suspend and resume control. This paper addresses two problems that remain in the use of iterators today: The first problem is that iterators represented as state-saving objects in languages such as C++ or Java typically have logic that is much more complicated than control–based iterators. This paper presents a program structuring technique that allows object–based iterators to be implemented with the same clarity as control–based iterators. The second problem is that the usual implementations of control–based iterators are not sufficiently efficient for high-performance applications. This paper presents a code optimization technique that can be applied to control–based iteration to produce efficient natural loops at the machine code level. Combined, these two results allow iterators for complex data structures to be easily written and efficiently implemented.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors—compilers, optimization

General Terms Languages, Design, Performance

Keywords Generic programming, Iterators

1. Introduction

The concept of the *iterator* has been well established in computer programming languages for some thirty years now. Iterators were introduced as constructs to allow looping over abstract data structures without revealing their internal representation. They have since become one of the cornerstones of object-oriented and generic programming.

Initially, support for iterators was provided through special programming language control-flow constructs. The `yield` statement of CLU [1] and the `suspend` statement of ICON [2], are two examples. These provided a mechanism for a structure-traversing program to give a value to a loop variable, then temporarily suspend execution until a value was needed for the next iteration. This allowed a very natural implementation of iterators: an abstract type would provide an iterator that traversed the data structure, knowing its representation, and yielding values as it went. This approach had the advantage of clarity, but required non-trivial language support and could incur substantial overhead. We call these "control–based iterators."

As more traditional programming languages adopted data abstraction mechanisms, another approach to implementing iterators became more common. An iterator could be implemented as a related data structure with fields to record the state of a traversal. For example, an array iterator would record a pointer to the head of the array and the current index. Iterators for complex data structures might require several fields to save the position in the object being traversed and have *complex logic to restore the position in the traversal logic* for the subsequent iteration step. This approach had the advantage that no new language features were needed, but the programs to traverse complex data structures were quite difficult to write and to understand. We call these "object–based iterators." Today, major software libraries most often adopt this approach, for example the C++ Standard Template Library [3] and Java JDK5 [4, 5].

We note that the concept of abstract iterators quickly followed the introduction of data abstraction [6, 7]. Despite a minority opinion that they are a flawed concept [8], with the increased use of generic programming, newer programming languages, such as Aldor [9, 10, 11] (our own language), C# 2.0 [12], Python [13], Ruby [14] and Sather [15], provide native support for control–based iterators. In some cases, the use of iterators is defined in terms of library classes for object–based iterators. For example, in C#, a function returning an iterator block creates an enumerator object with a `MoveNext` method.

This paper solves two problems that remain with iterators today: how to make object–based iterators easier to understand, and how to make control–based iterators more efficient. Taken together, these solutions allow clear and efficient implementation for both object–based and control–based iterators.

Some authors use the term *iterator*, and others the term *generator*. Some make subtle distinctions between the two. For clarity, we uniformly use the terms *object–based iterator* for cases in which it is the library programmer's responsibility to save traversal state in some structure and use the *control–based iterator* for cases in which this is automatic and the programmer writes an explicit traversal with suspending execution.

The remainder of this paper is organized as follows: Section 2 presents examples of generic iteration, a non-trivial object–based iterator and a control–based iterator. Section 3 describes the most important previous implementations of iterators. We present our implementation of control–based iterators in Section 4. Section 5 then shows how our iterator implementation can be optimized with value numbering and dataflow analysis. This is illustrated with extended examples in Section 6. We present our conclusions in Section 7.

WGP'06 September 16, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-492-6/06/0009...$5.00

2. Iterator Examples

2.1 Generic Iteration

We give a simple first example of generic iteration to serve as a starting point for our discussion. In C++, we may write an abstract base clase for a simple object–based iterator as shown in Figure 1a. The `step` method is called to position the iterator at the next value, if there is one. The `empty` method tells whether there is another value available and, if so, the `value` method returns it. A function using this iteration interface is shown in Figure 1b.

Iterators for specific classes can be derived from the abstract base class. Figure 1c shows an iterator that traverses an array. Note that this iterator object has fields `_argv` and `_argc` to represent the original object to be traversed, and the field `_i` representing the state of the iteration. Similarly, the iterator shown in Figure 1d extends the abstract base class to provide iteration over a linked list. The function `sum` in Figure 1b may be passed objects of type `AIter`, `LIter` or other derived types.

2.2 A Nontrivial Object-Based Iterator

In the previous example there was only a trivial amount of state to be maintained between successive calls to the `step` method. For the array iterator it was the single array index, `_i`, and for the list iterator it was the pointer to the current position in the list, `_l`.

We now look at an only slightly more complicated data structure to see how maintaining a consistent state and continuing a traversal is not always so easy. For this we use a simple hash table type, defined in Figure 2a as a vector of buckets, each bucket being a blocked linked list. It is straightforward, as shown in Figure 2b, to write a program to iterate over all values in the hash table by making direct use of the representation. This may also be achieved using abstract iteration, as shown in Figure 2c.

While the *use* of this abstract iteration is simple, it is not so straightforward to provide an *implementation*. In Figure 2d we show one possible implementation of the hash table iterator. Between invocations of the `step` method, it is necessary to save the values that correspond to `i`, `blk` and `j` of the function `printvals1`. In writing this iterator class, one must establish (at least informally) some invariant about the state variables that is true after each call to `step`. Here we have: if `i == ht->buckc` the iterator is done, otherwise `blk->entv[j].val` is the current value.

Note, in `step`, the cascading set of tests that work from the innermost part of the data structure back to the outermost part of the data structure. For a complicated data structure, re-entering the traversal logic in this way can be awkward and difficult to manage. The state of the traversal must be deduced from the values of the fields in the iterator object and some invariant associated with the `step` method. Because all resumptions of the iterator come through the *same* entry point, this invariant must simultaneously address the state of all variables involved in the traversal. This leads to more complicated reasoning than the separate loop invariants we have with a complex explicit traversal. Not only does this make programming error-prone for the library developer—it also makes it more difficult for a compiler to deduce what must be true in loops.

2.3 Control-Based Iterators

Figure 3a shows how a control–based iterator looks using the syntax of Aldor. We have assumed the same data structure fields as in the previous example. Ignoring the syntactic details, we see that implementing this iterator is essentially the same as writing explicit loops to traverse the structure, as in the `printVals1` function.

The Aldor language keyword `generate` heads a block of code that produces a Generator object. The keyword `yield` is used to return a value and suspend the program for later resumption. These keywords are underlined for clarity in the example. In the Aldor language, we would more naturally use a representation where sub-objects were traversable with their own iterators, as shown in Figure 3b.

We note that the code implementing these iterators is completely straightforward, using an explicit traversal of the data structure. For this control–based iterator there is no need for complex reasoning about the state of the object between method calls — this is manifest in the program. For complex data structures (e.g. trees, graphs) or lightweight structures with special cases (e.g. ascending *vs* descending integer ranges), it would be normal and natural to have multiple `yield` statements in the same generator. In object–based iterators, these situations lead to convoluted programs.

```
template <typename T>
class Iter {
public:
    virtual T     value() = 0;
    virtual bool  empty() = 0;
    virtual void  step()  = 0;
};
```

(a) Iterator abstraction

```
template <typename T>
T sum(Iter<T> &it) {
    T s = 0;
    for ( ; !it.empty(); it.step())
        s += it.value();
    return s;
}
```

(b) Iterator use

```
template <typename T>
class AIter : public Iter<T> {
    T   *_argv;
    int _i, _argc;
public:
    AIter(int ac, T *av) : _argc(ac),_argv(av),_i(0) { }
    void step()  { _i++; }
    bool empty() { return _i == _argc; }
    T  value()   { return _argv[_i]; }
};
```

(c) Array iterator

```
template <typename T>
struct ListLink {
    T        first;
    ListLink *rest;
};

template <typename T>
class LIter : public Iter<T> {
    ListLink<T> *_l;
public:
    LIter(ListLink<T> *l) { _l = l; }
    void step()  { _l = _l->rest; }
    bool empty() { return _l == 0; }
    T  value()   { return _l->first; }
};
```

(d) List iterator

Figure 1. A simple example of generic iteration

```
template <typename Key, typename Val>
struct HBlock {
    HBlock *next;
    int    entc;
    struct {Key key; Val val;} entv[10];
};

template <typename Key, typename Val>
struct HTable {
    int             buckc;
    HBlock<Key,Val> **buckv;
};
```

(a) Hash table type

```
template <typename Key, typename Val>
void printVals1(HTable<Key,Val> *ht) {
    for (int i=0; i < ht->buckc; i++) {
        HBlock<Key,Val> *blk = ht->buckv[i];
        while (blk != 0) {
            for (int j=0; j < blk->entc; j++)
                blk->entv[j].val.print();
            blk = blk->next;
        }
    }
}
```

(b) Explicit traversal

```
template <typename Key, typename Val>
void printVals2(HTable<Key,Val> *ht) {
    HIter<Key,Val> hi(ht);
    for ( ; !hi.empty(); hi.step())
        hi.value().print();
}
```

(c) Generic traversal

```
template <typename Key, typename Val>
class HIter : public Iter<Val> {
    HTable<Key,Val> *ht;
    HBlock<Key,Val> *blk;
    int i, j;
  public:
    HIter(HTable<Key,Val> *ht0) {
        ht = ht0;
        i  = 0;
        j  = -1;  // ++j gives entv[0]
        // Find first non-empty block
        while (i < ht->buckc) {
            blk = ht->buckv[i];
            if (blk && blk->entc > 0) break;
            i++;
        }
        step();
    }
    void step() {
        if (++j < blk->entc) return;
        j = 0;           // Try start of a block.
        blk = blk->next; // Try next block in chain.
        if (blk && blk->entc > 0) return;
        i++;             // Try next chain.
        while (i < ht->buckc) {
            blk = ht->buckv[i];
            if (blk && blk->entc > 0) break;
            i++;
        }
    }
    Val  value() { return blk->entv[j].val; }
    bool empty() { return i == ht->buckc; }
};
```

(d) Iterator definition

Figure 2. A non-trivial object–based iterator

```
generator(ht: HTable(Key,Val)): Generator(Val) ==
generate
    for i in 0..ht.buckc-1 repeat {
        blk := ht.buckv.i;
        while not null? blk repeat {
            for j in 0..blk.entc-1 repeat
                yield blk.entv.j.val;
            blk := blk.next;
        }
    }
```

(a) Using explicit indexing

```
generator(ht: HTable(Key,Val)): Generator(Val) ==
generate
    for blk in ht.buckv repeat
        while not null? blk repeat {
            for v in blk.entv repeat
                yield v;
            blk := blk.next;
        }
```

(b) Using array iterators on sub-parts

Figure 3. Control–based iterators

3. Previous Implementations

3.1 Object-Based Iterators

With object–based iterators the burden is on the library programmer to determine how best to save the traversal state between successive iteration steps.

Different environments have different conventions for the set of methods used for traversal. In our examples we used the methods `empty`, `value` and `step`. Sometimes the act of extracting the current value is combined either with stepping to the next value or with the end test. Java 5 libraries use the method `hasNext` as an end test and the method `next` both to extract the current value and to step to the next one. C# 2.0 libraries use the method `MoveNext` to advance the iterator and perform the end test, setting the value of `Current` as a side-effect.

To obtain the same efficiency as explicit traversal of the data structure, substantial additional compiler support is required. First, the compiler must support inlining and apply it to the iterator constructor and the methods used in traversal. Second, the compiler must be able to deduce and use loop invariants to simplify these inlined methods. Finally, the compiler must support data structure elimination to convert the iterator object to a collection of temporary variables, one for each field, and allow these fields to be placed as some combination of registers and memory.

3.2 Control-Based Iterators

Functional Transformation

An elegant implementation of control–based iterators is that adopted by CLU: When the loop begins execution, the iterator is activated and a stack frame is pushed for it. The iterator then runs until it encounters the first `yield`. At this point, the body of the loop is executed, *without* popping the stack frame for the iterator. This can be implemented simply with closures, or with a more specialized treatment. The result is equivalent to a transformation of the program skeleton shown in Figure 4a to the form shown in Figure 4b. We have hypothesized support for lexically nested functions that may be passed as parameters. The control–based iterator is then implemented as a function that takes a function for the loop body as a parameter, as shown in Figure 4c.

There are two concerns with this implementation: First, the efficiency depends critically on the cost of function calls and on the compiler's ability to do cross-module inlining. Second, and

```
example() {
    ...
    for v in ht repeat { body }
    ...
}
```

(a) Original client

```
example() {
    ...
    local bodyfun(v) { body }
    iterator(ht, bodyfun);
    ...
 }
```

(b) Transformed client

```
void iterator(HTable *ht, void (*bodyfun)(Val)) {
    for (int i=0; i < ht->buckc; i++) {
        struct HBlock *blk = ht->buckv[i];
        while (blk != 0) {
            for (int j=0; j < blk->entc; j++)
                // A yield statement would go here.
                bodyfun(blk->entv[j].val);
            blk = blk->next;
        }
    }
}
```

(c) Iterator implementation

Figure 4. Functional control–based iteration

perhaps more importantly, with this implementation of control–based iterators one cannot traverse multiple objects in parallel.

Continuations

Continuations, as supported by Scheme and other programming languages, provide a general mechanism to implement non-local control flow and may easily be used to implement control–based iterators. The iterator saves a continuation of where it is to resume and the loop saves a continuation to resume the loop body after advancing the iterator.

To show how these may be used, in Figure 5 we define a function `my-generator` that yields the symbol `before-first`, then yields the elements of the argument list `xx` one by one, and finally yields the symbol `after-last`.

While continuations provide a conceptually elegant implementation of iterators, there is an overhead for their naïve use. This includes the loss of a stack–based model for function calls, implying an increase in the complexity of memory management. There has been considerable work on the optimization of continuations, but many important programming languages do not support them at all. To introduce continuations in existing frameworks would require re-developing compiler implementations and would complicate language interoperability.

Co-routines and Threads

Implementation of control–based iterators using continuations does not make use of their full generality. In fact, the swapping back and forth between a continuation for the loop body and a continuation for the iterator is equivalent to setting up a pair of co-routines. The use of threads has also been proposed to provide co-routines for more natural iterators in Java [16]. In fact, the principal benefit cited in that paper was the ability to write the traversal procedures naturally, as is our objective. The problem with this approach, again, is the overhead associated with setting up and running co-routines or threads, making this strategy unsuitable for efficient inner loops.

```
;;;;;; Interface to create and use iterators
;; Macro (generate label e1 e2 e3 ...)
(define-syntax generate (syntax-rules () (
   (_ g expr ...)
   ;; g will have pair value (continuation . yield val)
   (let ((g (cons 'dummy 'dummy)))
       (if (eq? 'continue
                (call/cc (lambda (startcn)
                    (set-car! g startcn)
                    'start )) )
           (begin expr ... (set-car! g #f)) )
       (g-step! g)
       g) )))

(define (yield g value)
   ;; (car g) saves continuation to resume loop body
   (let ((loopcn (car g)))
      (if (eq? 'yield
               (call/cc (lambda (yieldcn)
                   (set-car! g yieldcn)
                   (set-cdr! g value)
                   'yield )) )
          (loopcn 'yield) ) ) )

;;;;;; Interface to use iterators
(define (g-step! g)
   (if (car g)
       ;; (car g) saves continuation to point after yield
       (let ((yieldcn (car g)))
          (if (eq? 'continue
                   (call/cc (lambda (loopcn)
                       (set-car! g loopcn)
                       'continue )) )
              (yieldcn 'continue) ) ))
   g)

(define (g-empty? g) (not (car g)))
(define (g-value  g) (cdr g))

;;;;;; Example of definition and use
(define (my-generator xx) (generate g0
   (yield g0 'before-first)
   (do ((l xx (cdr l))) ((null? l))  (yield g0 (car l)))
   (yield g0 'after-last) ))


(do ((g (my-generator '(a b c d e f)) (g-step! g)))
    ((g-empty?  g))
  (write (g-value g)) )
```

Figure 5. Continuation control–based iteration

4. Our Technique for Control–Based Iterators

We now describe our implementation of control–based iterators, as initially developed for Aldor. This implementation has the following properties:

- It allows parallel iteration over multiple objects.
- It admits optimizations that make it no more expensive than explicit hand-written loops.
- It can be applied (in some programming languages) to make object–based iterators look like control–based iterators.

The basic idea of our control–based implementation is to make the traversal function free of local state by lifting its variables to a higher lexical level and suspending the traversal function by remembering its instruction pointer. This instruction pointer can then become one more field in an object–based iterator and resuming the traversal function consists of jumping to the saved location.

```
generator(HTable *ht) == generate {
    for (int i=0; i < ht->buckc; i++) {
        HBlock *blk = ht->buckv[i];
        while (blk != 0) {
            for (int j=0; j < blk->entc; j++)
                yield blk->entv[j].val;
            blk = blk->next;
        }
    }
}
```

(a) Original control–based iterator (pseudocode)

```
// Two distinguished values.
void  *_Start = 0, *_End = (void *)(-1);

template <typename Key, typename Val>
class Generator : public Iter<Val> {
    int i, j;
    HTable<Key, Val> *ht;
    HBlock<Key, Val> *blk;
    void * _L;       // Saves the instruction pointer.
    Val    _val;     // Saves the yielded value.
  public:
    Generator(HTable<Key,Val> *ht0) : ht(ht0), _L(_Start)
    { step(); }

    void step() {
         if (_L != _Start) goto *_L;  // GNU C++ notation
         for (i=0; i < ht->buckc; i++) {
            blk = ht->buckv[i];
            while (blk != 0) {
                for (j=0; j<blk->entc; j++) {
                    _L   = && L1;     // GNU C++ notation
                    _val = blk->entv[j].val;
                    return;
         L1:        ;
                }
                blk = blk->next;
            }
            _L = _End;
         }
    }
    bool empty() { return _L == _End; }
    Val  value() { return _val; }
};
```

(b) After transformation

Figure 6. Our approach to control–based iterators

```
#define GI0         -2   // Start label
#define GIX         -1   // Exit  label

#define GIBegin     switch (this->_L) { case GI0: ;
#define GIEnd       {this->_L=GIX; case GIX: return;} }

#define GIYield(n,v){this->_L=n;this->_val=v;return;case n:;}
#define GIReturn    {this->_L=GIX; return; }

template <typename Val>
class GIter {
protected:
    int _L;
    Val _val;
public:
    GIter() : _L(GI0) { }
    Val  value()        { return _val; }
    bool empty()        { return _L == GIX; }
    virtual void step() = 0;
};
```

(a) Cosmetic macros and iterator base class

```
template <typename Key, typename Val>
class HIter : public GIter<Val> {
private:
    HTable <Key,Val> *ht;
    HBlock <Key,Val> *blk;
    int i, j;
public:
    HIter(HTable <Key,Val> *ht0) : ht(ht0) { step(); }

    void step() {
        GIBegin;
        for (i=0; i < ht->buckc; i++) {
            blk = ht->buckv[i];
            while (blk != 0) {
                for (j=0; j < blk->entc; j++)
                    GIYield(1, blk->entv[j].val);
                blk = blk->next;
            }
        }
        GIEnd;
    }
};
```

(b) Creating an iterator

Figure 7. Realization in Standard C++

With these alterations, the natural control–based traversal function replaces the complex object–based step function. It is no longer necessary for the library programmer to provide code to work back into the middle of a complex data structure. This approach amounts to creating a specialized light-weight continuation in a programming language that does not support continuations natively.

We illustrate this with our hash table example. For those not familiar with Aldor, we use a C++ syntax, extended with `generate`, `yield` and label variables. We begin with the control–based iterator shown in Figure 6a.

The control–based iterator is transformed to an object with three methods, `step`, `empty` and `value`, sharing a common environment. The original traversal function becomes the `step` method. All local variables (`i`, `j`, `ht` and `blk`) are lifted out of the traversal function, and two new variables (`_L` and `_val`) are introduced.

Suspension is achieved by replacing each `yield` statement in the original program by

- a statement saving the resumption point in `_L`,
- a statement placing the value to `yield` in `_val`,
- a simple `return` statement, and
- a label *after* the return statement.

Resumption is achieved by adding a multi-way branch to the saved resumption point as the first statement of the traversal function. When the traversal program completes, the label variable is given the end label value as the resumption point. This is then used as a test to determine whether the iterator has completed.

With these conventions, the control–based hash table iterator is transformed to the program shown in Figure 6b. We have used the GNU C++ syntax (`&&`) for label variables. Although computed `goto`s are now typically viewed with disfavour, in fact this use *increases* the clarity of the code as compared to Figure 2d.

The Aldor implementation does not actually construct an object with methods accessing the object's fields. Instead, it equivalently constructs a triple of closures (`empty?`, `value` and `step!`) sharing an environment. *All* of the `for` loops in Aldor use iterators compiled in this way. The closures are then inlined and further optimized, as described in Section 5, to obtain the same efficiency as explicit traversal.

If the target language does not support label variables, but does support arbitrary branching, it is straightforward to re-cast the saving of the control flow point. This is done by associating an distinct identifying value with each `yield`.

We illustrate this for Standard C++ in Figure 7. We have used an integer variable instead of a label variable to save the resumption point, and have used a switch statement in place of the computed `goto`. To resume the traversal at an arbitrary point in the `step` function, we make novel use of an often undesirable property of the C++ `switch` statement: A switch's cases can occur anywhere, simply as labels in the compound statement of the switch body. Although it is different, this use of a switch statement to resume a function is reminiscent of Duff's device [17], in which a switch statement is used in loop unrolling.

The judicious use of a few macros can improve the readability of control–based iterators written this way. This is shown in Figure 7a. We arbitrarily fix `-2` and `-1` as the labels for the start and end of the traversal. This allows us to have a base class providing a common, trivial, implementation for the `empty` and `value` methods. With this, our control–based hash table iterator is simplified, as shown in Figure 7b. We note that `GIYield` may appear any number of times, provided each occurrence uses a different integer label. This is useful for iterators over complex structures. The `GIReturn` may also occur any number of times, and when it is used it causes the iterator to complete. (Subsequent calls to `empty` will return `true`.) The function body must start and end with `GIBegin` and `GIEnd`.

5. Optimization of Control-Based Iterators

Before we can adopt our implementation of control–based iterators without reservation, we must ensure that it can be at least as efficient as the usual object–based iterators, and preferably as good as hand-written loops.

The problem with our implementation, as presented, is the multi-way branch at the beginning of the `step` function. This form of computed jump breaks branch prediction on modern architectures and obscures loop invariants. Furthermore, a naïve implementation would have a number of unnecessary jumps in each loop iteration.

We use the following optimization strategy to improve code arising in our control–based implementation:

1. *Perform function inlining.* Inlining thresholds will determine whether `step`, `empty` and `value` get inlined. If they are not suitable inlining candidates, then optimizing this loop is not critical.
2. *Apply data structure elimination.* This will merge the data of the iterator objects (or the environments of the iterator function closures) into the enclosing scope's set of local/temporaries.
3. *Compute the flow graph.*
4. *Establish a value numbering* for the loop control variables. The loop control variables are the variables that are either tested to determine a multi-way branch or are the labels of a computed goto/switch. There will be one value number for each yield statement, as well as one each for the beginning and end of the iterator.
5. Repeat the following steps until loop control variables are dead, or flow graph does not change:
   (a) *Clone selected nodes.* The nodes to be cloned are those modifying or testing loop control variables with multi-way input or multi-way output. If necessary, clone all blocks from natural loop header to this node.
   (b) *Associate distinct instances* of each cloned block to each of that block's predecessors
   (c) *Apply dataflow analysis.* Assignments to loop control variables and branch tests generate and kill value numbering properties.
   (d) *Specialize the program.* The loop control variables will now have determined value in particular basic blocks. Specialize the block where this occurs.
6. *Clean up.* Copy propagation, common subexpression elimination, constant folding, dead variable elimination, block consolidation.

This optimization can be applied to low-level intermediate code and does not require any knowledge of the suspend–resume semantics. Indeed, this optimization might beneficially be applied to programs that make use of computed branches of any sort. These would include computed `goto`s of Fortran, branching through label variables, switch statements with arbitrary structure (e.g Duff's device, our device), or tail calls of function variables.

We emphasize that our implementation of control–based iterators is the *only* form of `for` loop provided in Aldor. We have relied on this optimization to remove iterator overhead since the original language implementation [9] and have found it to be effective.

```
Int ==> Integer;

generator(seg:Segment Int):Generator Int ==
generate {
   a := lo seg;
   b := hi seg;
   while a <= b repeat { yield a; a := a + 1 }
}
client() == {
    ar := array(...);
    s  := 0;
    -- n..m forms a Segment
    for i in 1..#ar repeat s := s + a.i;
    stdout << s
}
```

(a) Original program

```
client() == {
    local ar, s, i, a, b, val, lab, seg;
    ar   := array(...);
    seg  := 1..#ar;
    lab  := L0;
    step!() == {
       goto @lab;
L0:    a := seg.lo;
       b := seg.hi;
       while a <= b repeat {
          lab := L1;
          val := a;
          return;
L1:
          a := a + 1
       }
       lab := LX;
LX: }
    empty?() == { lab = LX }
    value()  == { val }
    s := 0;
    repeat {
        step!();
        if empty?() then break;
        i := value();
        s := s + a.i
    }
    stdout << s
}
```

(b) Iterator constructor inlined

Figure 8. Example 1: Integer range iteration

6. Optimization Examples

We illustrate our optimization strategy with two examples. In the first case, we show how it can be applied to a simple range counting iterator. In the second case, we show the optimization of the parallel traversal of two iterators.

6.1 Integer Range Iterator

We apply our optimization to the use of a simple integer range iterator, shown in Figure 8a. We begin by inlining the generator for the `for` loop. This gives the program in Figure 8b, shown in a pseudo-code notation. We inline the `step!`, `empty?` and `value` functions. Converting the flow-control to `goto`s gives the program with flow graph shown Figure 9.

According to our criteria, the blocks B1 and B7 must be cloned. We split B1 to B1a and B1b, with B1a having predecessor B0 and B1b having predecessor B8. We clone B7 as B7a, B7b, B7c with predecessors B4, B6 and B1a, respectively. For convenience we split B8 and B9 to B8a/8b, B9a/B9b respectively so we have a place to put dataflow information. Whenever a multi-way exit is based on the value of a loop control variable, we can put information about its value in the branch destinations. In this case, we insert the information that `lab = B7c` in B9a and `lab != B7c` in B8a. The resulting flow graph is shown in Figure 10.

We set up forward dataflow equations for the value numbering with bits 1, 2, 3 corresponding to the variable `lab` having values B2, B5, B7a respectively, and obtain the results shown in Figure 11. From this, we see that block B1a must exit to B2, because that is the only value that `lab` can have. Coming into block B7a, we see that only the second bit is on so `lab` must have the value B5. Since this is not equal to B7c, we know that block B7a must exit to B8a. Similarly B7b must exit to B9a, and B7c must exit to B8a.

We use this information to specialize the exits of the affected blocks, update the flow graph, recompute the Gen and Kill sets, and perform a second dataflow computation. This shows that B1b must exit to B5. We update the flow graph with this information. We obtain no further changes and so proceed to the clean up step. We obtain the program shown in Figure 12.

We have been able to optimize away all overhead associated with the simple control–based iterator, and obtain the same low-level code as explicit iteration written by hand.

```
client() == {
    local ar, s, i, a, b, val, lab, seg;
B0: ar := array(...);
    seg := 1..#ar;
    lab := B2;
    goto B1;
B1: goto @lab;
B2: a := seg.lo;
    b := seg.hi
    s := 0;
    gotoB3;
B3: if not (a <= b) then goto B6 else goto B4;
B4: lab := B5;
    val := a;
    goto B7;
B5: a := a + 1
    goto B3;
B6: lab := B7
    goto B7;
B7: if not not (lab = B7) then goto B9 else goto B8;
B8: i := val;
    s := s + ar.i;
    goto B1;
B9: stdout << s;
    return
}
```

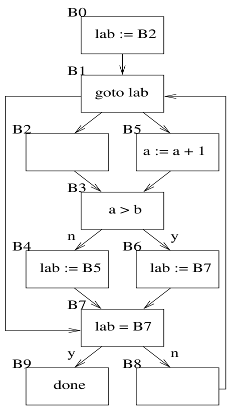


Figure 9. After inlining `empty?`, `value` and `step!`

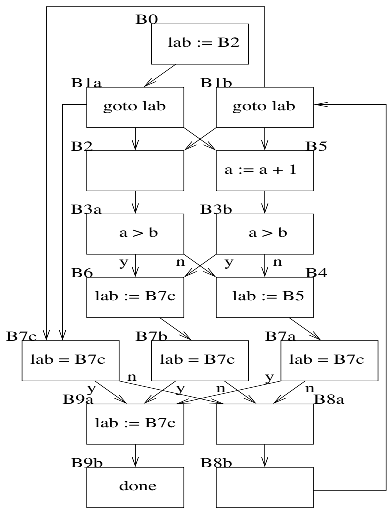


Figure 10. After cloning nodes

Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2 B5 B7c	...	...	1..	1..
B1b	B8b	B2 B5 B7c	...	...	11.	11.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B4 B6	...	...	11.	11.
B4	B3	B7a	.1.	1.1	11.	.1.
B5	B1a B1b	B3	...	...	11.	11.
B6	B3	B7b	..1	11.	11.	..1
B7a	B4	B8a B9a	...	...	.1.	.1.
B7b	B6	B8a B9a	...	...	..1	..1
B7c	B1a B1b	B8a B9a	...	...	11.	11.
B8a	B7a B7b B7c	B8b	...	..1	111	11.
B8b	B8a	B1b	...	...	11.	11.
B9a	B7a B7b B7c	B9b	..1	11.	111	..1
B9b	B9a		...	...	..1	..1

Figure 11. Dataflow results for a simple iterator.
Bits represent possible values for labels.

```
client() == {
     local ar, s, a, b;
     ar := array(...);
     a := 1;
     b := #ar;
     s := 0;
B3:  if a > b then goto B9b;
     s := s + ar[i];
     a := a + 1;
     goto B3;
B9b: stdout << s;
     return
}
```

Figure 12. Control-flow optimization complete

```
generator(seg:Segment Int):Generator Int == generate {
    a := lo seg;
    b := hi seg;
    while a <= b repeat {
        yield a;
        a := a + 1
    }
}
generator(l: List Int): Generator Int == generate {
    while not null? l repeat {
        yield first l;
        l := rest l
    }
}
client() == {
    ar := array(...);
    li := list(...);
    s := 0;
    for i in 1..#ar  for e in l  repeat {
        s := s + ar.i + e
    }
    stdout << s
}
```

Figure 13. Example 2: Parallel traversal

6.2 Parallel Iterators

We give a second example of our optimization strategy, applying it to the case of parallel traversal of two iterators, one over an integer range, the other over a linked list. The starting program is shown in Figure 13. In Aldor, parallel iteration is indicated by placing two `for` iteration controls on a single `repeat` loop. We show the two iterator-producing functions, even though they would normally be placed in other files associated with the type constructors `Segment` and `List`

The first step is to inline the iterator functions into the `client` program, represent all control flow as `goto`s, and perform data structure elimination and some boolean expression simplification. The code resulting after data structure elimination is illustrated in Figure 14. We identify B1 as the head node of a loop and `lab1` as its controlling variable. We also note that B7 has a branch that depends on the label variable `lab1`. Therefore, by our criteria, we must clone B1 and B7 into copies for each of their predecessors. This is illustrated in Figure 15.

We determine that the possible values for `lab1` are B2, B5 and B7 and give these value numbers 1, 2 and 3 respectively. A first round of dataflow analysis determines the possible values for `lab1` in each basic block, as shown in Figure 16a. We use the results of this first dataflow analysis to specialize the program and recompute the dataflow. This yields the results shown in Figure 16b. These results allow us to specialize each of the cloned nodes B1a, B1b, B7a, B7b, B7c. After clean up, we obtain the program represented by Figure 17.

```
client() == {
    local ar, l,
         a: Int, b: Int, s: Int, val1: Int,
         lab1: Label, seg: Segment Int,
         l2: List Int, val2: Int, lab2: Label;

B0:  ar := array(...);
     l := list(...);
     seg := 1..#ar;
     lab1 := B2;
     l2 := l;
     lab2 := B9;
     s := 0;
     goto B1;
B1:  goto @lab1;
B2:  a := seg.lo;
     b := seg.hi;
     goto B3;
B3:  if a > b then goto B6; else goto B4;
B4:  lab1 := B5
     val1 := a;
     goto B7;
B5:  a := a + 1
     goto B3;
B6:  lab1 := B7
     goto B7;
B7:  if lab1 = B7 then goto B16; else goto B8
B8:  i := val1;
     goto @lab2;
B9:  goto B10
B10: if null? l2 then goto B13; else goto B11
B11: lab2 := B12
     val2 := first l2;
     goto B14;
B12: l2 := rest l2
     goto B10
B13: lab2 := B14
     goto B14
B14: if lab2 = B14 then goto B16; else goto B15
B15: e := val2;
     s := s + ar.i + e
     goto B1;
B16: stdout << s
}
```

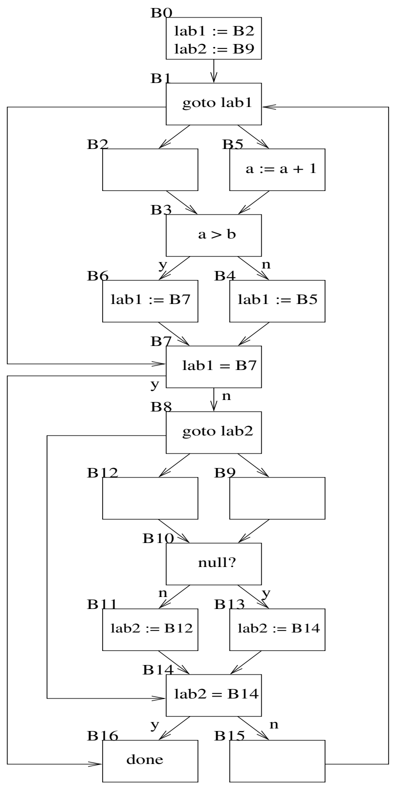


Figure 14. Inlined parallel iterators

```
client() == {
     /* lab1 := B2, B5, B7a */
B0:  ar := array(...);
     l := list(...);
     seg := 1..#ar;
     lab1 := B2;
     l2 := l;
     lab2 := B9;
     s := 0;
     goto B1a;
B1a: goto @lab1;
B1b: goto @lab1;
B2:  a := seg.lo;
     b := seg.hi;
     goto B3;
B3:  if a > b then goto B6; else goto B4;
B4:  lab1 := B5
     val1 := a;
     goto B7b;
B5:  a := a + 1
     goto B3;
B6:  lab1 := B7a
     goto B7c;
B7a:  if lab1 = B7a then goto B16; else goto B8
B7b:  if lab1 = B7a then goto B16; else goto B8
B7c:  if lab1 = B7a then goto B16; else goto B8
B8:  /* lab1 != B7 */
     i := val1;
     goto @lab2;
B9: goto B10
B10: if null? l2 then goto B13; else goto B11
B11: lab2 := B12
     val2 := first l2;
     goto B14;
B12: l2 := rest l2
     goto B10
B13: lab2 := B14
     goto B14
B14: if lab2 = B14 then goto B17; else goto B15
B15: e := val2;
     s := s + ar.i + e
     goto B1b;
B16: /* lab1 = B7 */
     goto B17/
B17: stdout << s
}
```

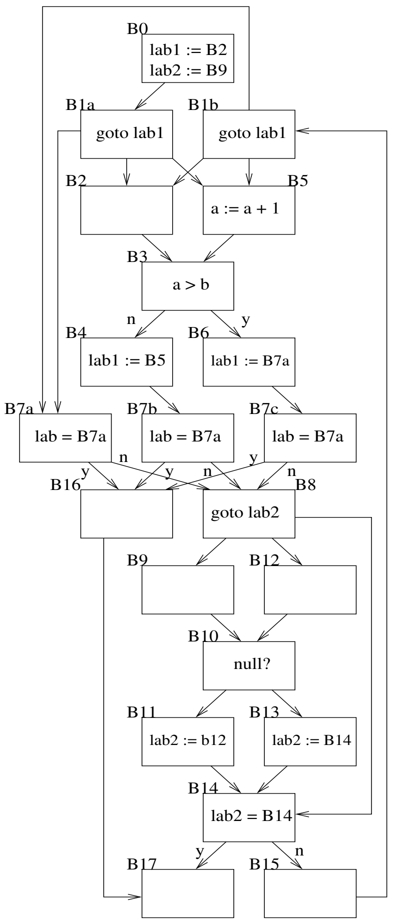


Figure 15. Split blocks for 1st iterator

Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2 B5 B7a	...	...	1..	1..
B1b	B15	B2 B5 B7a	...	...	11.	11.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B6 B4	...	...	11.	11.
B4	B3	B7b	.1.	1.1	11.	.1.
B5	B1a B1b	B3	...	...	11.	11.
B6	B3	B7c	..1	11.	11.	..1
B7a	B1a B1b	B8 B16	...	...	11.	11.
B7b	B4	B8 B16	...	...	.1.	.1.
B7c	B6	B8 B16	...	...	..1	..1
B8	B7a B7b B7c	B9 B12 B14	...	..1	111	11.
B9	B8	B10	...	...	11.	11.
B10	B9 B12	B11 B13	...	...	11.	11.
B11	B10	B14	...	...	11.	11.
B12	B8	B10	...	...	11.	11.
B13	B10	B14	...	...	11.	11.
B14	B8 B11 B13	B17 B15	...	...	11.	11.
B15	B14	B1b	...	...	11.	11.
B16	B7a B7b B7c	B17	..1	11.	111	..1
B17	B16 B14		...	...	111	111

(a) First round

Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2	...	...	1..	1..
B1b	B15	B2 B5	...	...	.1.	.1.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B6 B4	...	...	11.	11.
B4	B3	B7b	.1.	1.1	11.	.1.
B5	B1b	B3	...	...	.1.	.1.
B6	B3	B7c	..1	11.	11.	..1
B7a	B1b	B8	...	...	.1.	.1.
B7b	B4	B8	...	...	.1.	.1.
B7c	B6	B16	...	...	..1	..1
B8	B7a B7b	B9 B12 B14	...	...	.1.	.1.
B9	B8	B10	...	...	.1.	.1.
B10	B9 B12	B11 B13	...	...	.1.	.1.
B11	B10	B14	...	...	.1.	.1.
B12	B8	B10	...	...	.1.	.1.
B13	B10	B14	...	...	.1.	.1.
B14	B8 B11 B13	B17 B15	...	...	.1.	.1.
B15	B14	B1b	...	...	.1.	.1.
B16	B7c	B17	...	...	..1	..1
B17	B16 B14		...	...	.11	.11

(b) Second round

Figure 16. Dataflow for the first iterator

```
client() == {
B0:  ar := array(...);
     l := list(...);
     seg := 1..#ar;
     l2 := l;
     lab2 := B10;
     s := 0;
     a := seg.lo;
     b := seg.hi;
     goto B3;

B3:  if a > b then goto B17; else goto B4;
B4:  val1 := a;
     i := val1;
     goto @lab2;

B10: if null? l2 then goto B13; else goto B11
B11: lab2 := B12
     val2 := first l2;
     goto B14;
B12: l2 := rest l2
     goto B10
B13: lab2 := B14
     goto B14

B14: if lab2=B14 then goto B17; else goto B15
B15: e := val2;
     s := s + ar.i + e
     a := a + 1
     goto B3;
B17: stdout << s
}
```

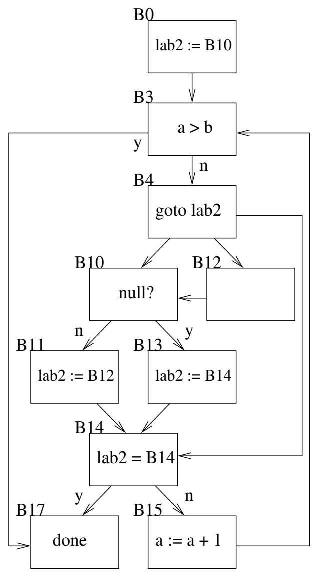
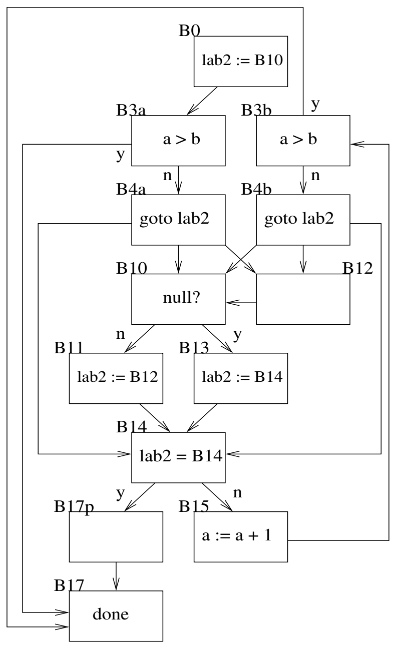
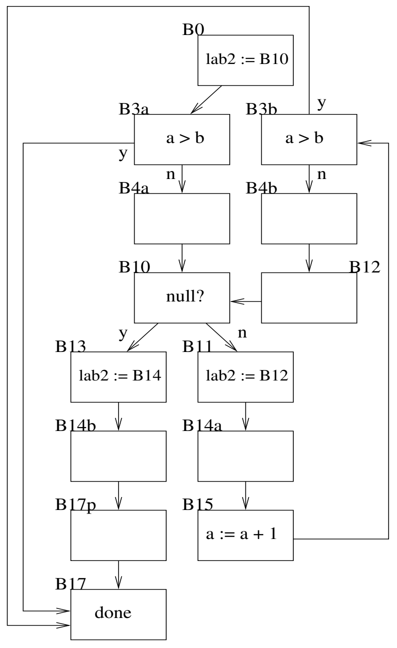


Figure 17. Resolution of 1st iterator



(a) After splitting B3 and B4



(b) After splitting B14

Figure 18. Split blocks for 2nd iterator

Having dealt with the first iterator, we continue. We recognize B3 as a loop head and B4 as a computed goto that depends on the loop variable `lab2`. We clone the blocks from the loop head to the computed goto block. This is shown in Figure 18a. We determine that the variable `lab2` may take only the values B10, B12, B14, do dataflow and specialize. Then block B14 is identified and split, as shown in Figure 18b. After dataflow, specialization and clean up, we obtain the program shown in Figure 19.

Our optimization has eliminated the overhead of the abstract control–based iterators, including that arising from obscured branch prediction. Normally, further optimizations would be applied to the program at this stage. For example, the list operations `first`, `rest` and `null?` would be inlined, and the null tests inside `first` and `rest` would be eliminated as redundant.

```
client() == {
      ar := array(...);
      l  := list(...);
      l2 := l;
      s  := 0;
      a  := 1;
      b  := #ar;
      if a > b then goto L2
L1:  if null? l2 then goto L2
      e := first l2;
      s := s + ar.a + e
      a := a + 1
      if a > b then goto L2
      l2 := rest l2
      goto L1
L2:   stdout << s
}
```

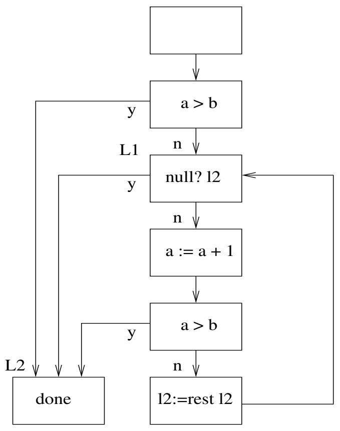


Figure 19. Resolution of 2nd iterator

7. Conclusions

Interest in iterators has renewed with the increased use of generic programming. Although they have a long history, certain practical problems relating to iterators have remained open.

In languages that do not support iterators natively, writing object–based iterators has been overly complicated for library programmers, requiring complex logic to recover previous traversal state. In languages that do support iterators natively, their performance has made them unsuitable for use in critical inner loops. In this situation, generic loop traversal is often avoided and replaced by explicit iteration over more basic values such as integer ranges or arrays. This paper solves both these problems.

First, we have shown how to write iterators clearly in a setting where only object–based iterators are supported. We have shown a straightforward way to simulate control–based iterators in with object–based iterators. With this, it is possible to use a clear programming style for iterators even in settings where control–based iterators are not directly supported. As one particular instance, we have shown a C++ programming device to write iterators in the control–based style.

Second, we have shown how to optimize the complex branching that arises with control–based iterators. We have implemented this optimization in the Aldor compiler, where it has proven very effective. In Aldor, abstract iterators serve as the *only* mechanism to iterate over a set of values and, in particular, there is no special treatment for low-level types. The compiler is relied upon to optimize away all overhead from abstract iteration. The resulting machine code is what one would write explicitly by hand.

These two results, combined, offer a useful direction for improved iterators, easily applicable to mainstream programming language implementations.

References

[1] Liskov, B., Atkinson, R., Bloom, T., Moss, J.E., Schaffert, J.C., Scheifler, R., Snyder, A.: CLU Reference Manual. Springer Verlag LNCS 114 (1981).

[2] Griswold, R.E., Hanson, D.R., Korb, J.T.: Generators in Icon. ACM Transactions on Programming Languages and Systems **3** (1981) 144–161.

[3] Musser, D.R., Derge, G. J., Saini, A.: STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library. Addison Wessley (2001).

[4] JDK 5.0 Documentation. Sun Microsystems (2004) `http://java.sun.com/j2se/1.5.0/docs` .

[5] JSR 201: Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. Sun Microsystems (2004) `http://java.sun.com/j2se/1.5.0/docs/guide/language/foreach.html` .

[6] Atkinson, R.: Toward more general iteration methods in CLU. Technical Report CLU Design Note 54, Programming Methodology Group, MIT Laboratory for Computer Science, Cambridge MA, USA (1975).

[7] Shaw, M., Wulf, W., London, R.: Abstraction and verification in Alphard: Defining and specifying iteration and generators. Communications of the ACM **20** (1977).

[8] Baker, H.G.: Iterators: Signs of weakness in object-oriented languages. ACM SIGPLAN OOPS Messenger **4** (1993) 18–25.

[9] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Steinbach, J.M., and Sutor,R.S.: A First Report on the $A^{\#}$ Compiler, Proc. International Symposium on Symbolic and Algebraic Computation (ISSAC 1994), July 20-22 1994, Oxford, England, ACM Press (1994) 25–31.

[10] Watt, S.M.: Aldor. In Grabmeier, J., Kaltofen, E., Weispfenning, V., eds.: Handbook of Computer Algebra, Springer Verlag (2003) 265–270.

[11] Aldor Users Guide. Aldor.Org (2003) `http://www.aldor.org/AldorUserGuide` .

[12] C# Version 2.0 Specification. Microsoft Corporation (2005).

[13] van Rossum, G., (editor), F.L.D.J.: Python Language Reference Manual (Release 2.3). Python Labs (2003).

[14] Thomas, D., Fowler, C., Hunt, A.: Programming Ruby: The Pragmatic Programmer's Guide (2nd Ed.) The Pragmatic Bookshelf (2004).

[15] Murer, S., Omohundro, S., Stoutamire, D., Szyperski, C.: Iteration abstraction in Sather. ACM Transactions on Programming Languages and Systems **18** (1996) 1–15.

[16] Kim, M.H.: Generalized enumeration mechanism for Java. Java Developer's Journal **3** (1998).

[17] Duff, T.: E-mail to Dennis Ritchie Thu 10 Nov 83 17:57:56 PST (1983).