A Technique for Generic Iteration and Its Optimization

Stephen M. Watt

The University of Western Ontario

October 17, 2005

Western

ORCCA

Ontario Research Centre for Computer Algebra

Outline

- Generic iteration
- *Save/restore* vs *suspend/resume* iteration
- Previous implementations of suspend/resume
- Our implementation
- Optimization

Generic Iteration

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

```
int sum(Iter<int> *iit) {
    int s = 0;
    for ( ; !iit->empty(); iit->step())
        s += iit->value();
    return s;
}
```

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

```
struct ListLink { int first; ListLink *rest; };

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

Save/Restore Iteration

- Save and restore state of iteration in an iterator object. *E.g.* **`_i`** in AIter, **`_l`** in LIter
- Not always so simple…

A slightly more complicated example

- Hash table type and traversal:

```
typedef int   Key;
typedef char *Val;

struct HBlock {
    HBlock *next;
    int entc;
    struct {Key key; Val val;} entv[10];
};
struct HTable {
    int buckc;
    HBlock **buckv;
};
```

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

void printVals2(HTable *ht) {
    HIter hit(ht);
    for ( ; !hit.empty(); hit.step())
        print(hit.value());
}
```

Save/Restore Iterator

- Logic is *much* more complicated
- Must establish (at least informally) invariants
- How to optimize?

```
class HIter : public Iter<Val> {
    HTable *ht;
    HBlock *blk;
    int i, j;
public:
    HIter(HTable *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; }
};
```

Suspend/Resume Iterator

- “yield” in CLU, “suspend” in Icon
- Suspend/resume for same structure in Aldor:

```
generator(ht: HTable): 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;
        }
}
```

- Same clear logic as explicit traversal.

Previous Implementations

- Save/Restore:
  - Efficiency requires inlining, unravelling save/restore logic, data structure elimination
- Functional Suspend/Resume:
  - Pro: conceptually elegant, easy implementation
  - Con: efficiency, cannot do parallel traversal
- Continuation Suspend/Resume:
  - Pro: conceptually elegant
  - Con: loss of stack-based model
- Co-routine and Thread-based Suspend/Resume:
  - Pro: easy to write iterators
  - Con: efficiency, complex model

Our Implementation of Suspend/Resume

- Basic idea:
  - Make the traversal function state-free
    by lifting variables to an outer lexical level.
  - Suspension is achieved by remembering IP.

- Advantages:
  - Allows parallel iteration
  - Admits optimization
  - Can make save/restore look like suspend/resume

- This is the way *all* for loops are handled in Aldor

Example

```
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;
        }
    }
}
```

```
class HIter : public Iter<Val> {
    HTable *ht;          HBlock *blk;
    int i, j, _lab;      Val _val;
public:
    HIter(HTable *ht0) { ht = ht0; _lab = 0; }

    void step() {  switch(_lab) { case 0:
         for (i=0; i < ht->buckc; i++) {
              blk = ht->buckv[i];
              while (blk != 0) {
                   for (j=0; j<blk->entc; j++){
                        _val  = blk->entv[j].val;
                        _lab = 1;  return;  case 1: ;
                   }
                   blk = blk->next;
              }
         }
         _lab = -1;  case -1: ;
    }  }
    Val   value()  { return _val; }
    bool empty() { return _lab == -1; }
};
```

C++ Cosmetics

```
#define GI0        0
#define GIX        -1
#define GIBegin     switch(_lab){case GI0: ;
#define GIYield(L,v){_val=v;_lab=L; return; case L: ;}
#define GIReturn    {_lab=GIX; return;}
#define GIEnd       {case GIX: return;} }

template <typename V> class GIter {
protected:
    int _lab; V _val;
public:
    GIter() : _lab(GI0) { }
    V    value() { return _val; }
    bool empty() { return _lab == GIX; }
};
```

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

    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;
            }
        }
        GIReturn;
        GIEnd;
    }
};
```

Optimization

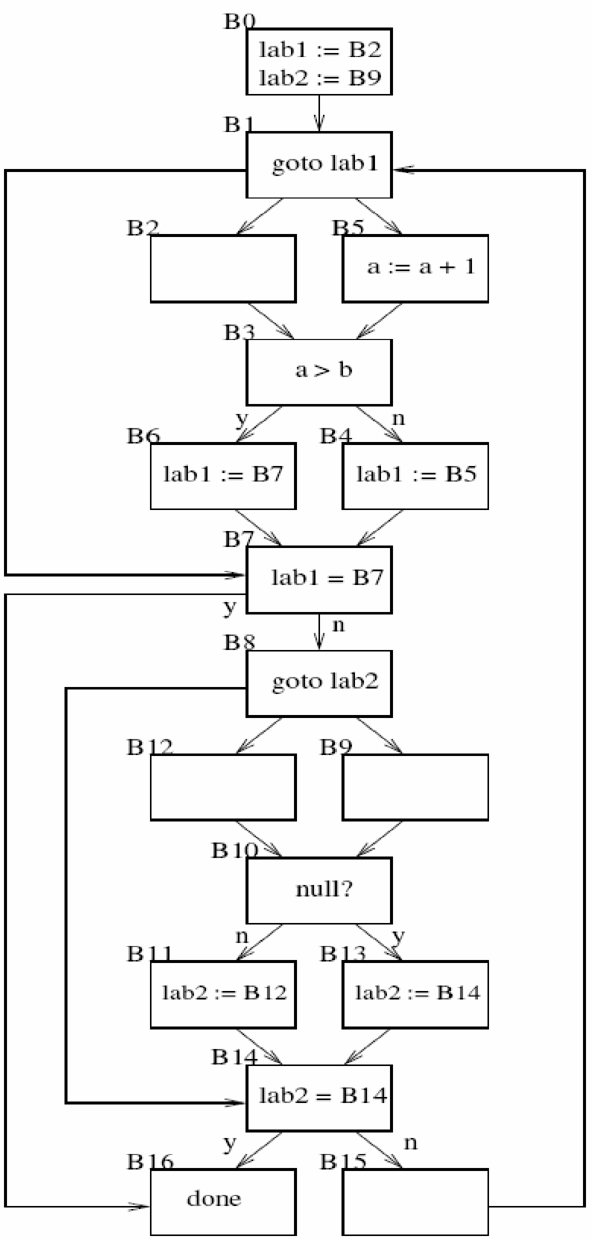
1. Perform function inlining
2. Apply data structure elimination (flattens closure envs)
3. Value numbering of vars tested to for multi-way branches (*L*oop *C*ontrol *V*ariables)
4. Repeat until LCVs dead or no change:
   - Clone blocks from loop header to blocks modifying or testing loop control variables
   - Associate distinct instances of each cloned block to that block's predecessors
   - Dataflow. Assignments to LCVs generate, and branches kill.
   - Specialize program. LCVs now have determined values in basic blocks.
5. Clean up.
   - Copy prop. CSE. Const folding. Dead var elim. Block consolidation.

Example: Parallel traversal of range and list

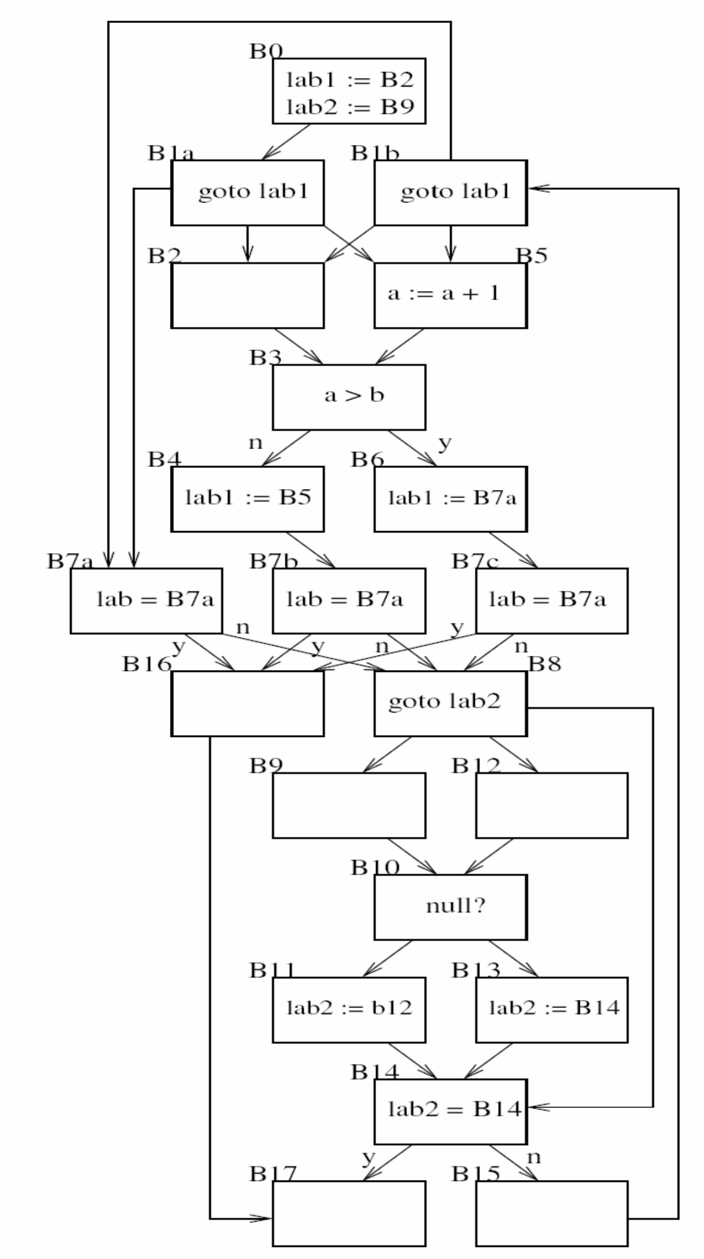
```
generator(seg:Segment Int):Generator Int == generate {
    i := a;
    while a <= b repeat { yield a; a := a + 1 }
}
generator(l: List Int): Generator Int == generate {
    while not null? l repeat { yield first l; l := rest l }
}

client() == {
    ar := array(...);
    li := list(...);
    s := 0;
    for i in 1..#ar  for e in l  repeat { s := s + ar.i + e }
    stdout << s
}
```

Inlined

```
B0:  ar := array(...);
     l := list(...);
     segment := 1..#ar;
     lab1 := B2;
     l2 := l;
     lab2 := B9;
     s := 0;
     goto B1;
B1:  goto @lab1;
B2:  a := segment.lo;
     b := segment.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
```



Split Blocks for 1st Iterator



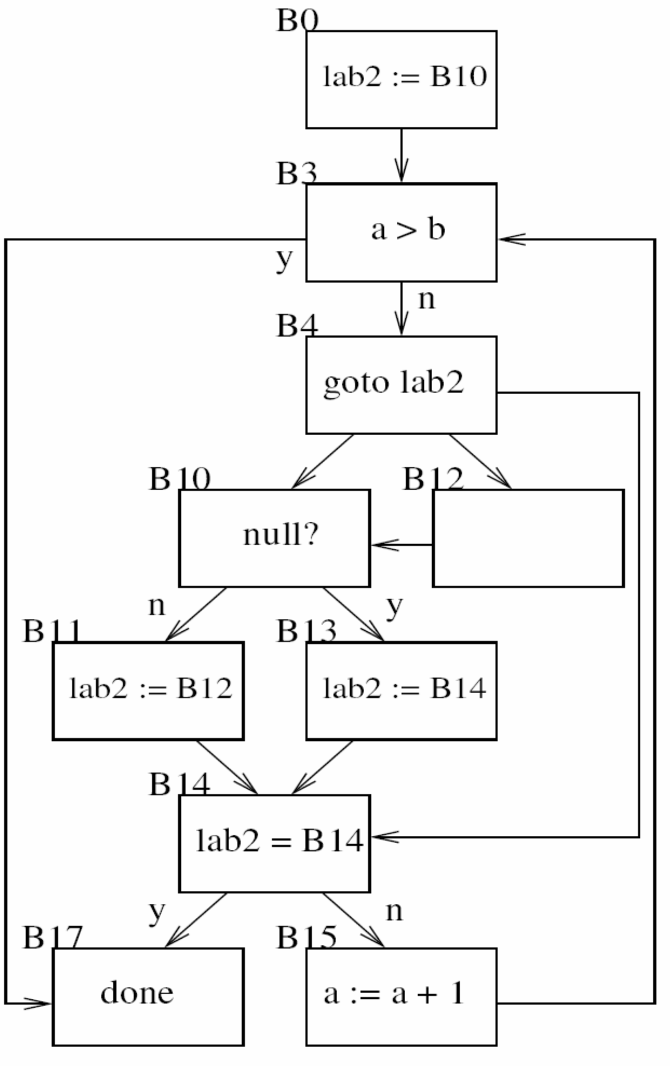
Dataflow

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

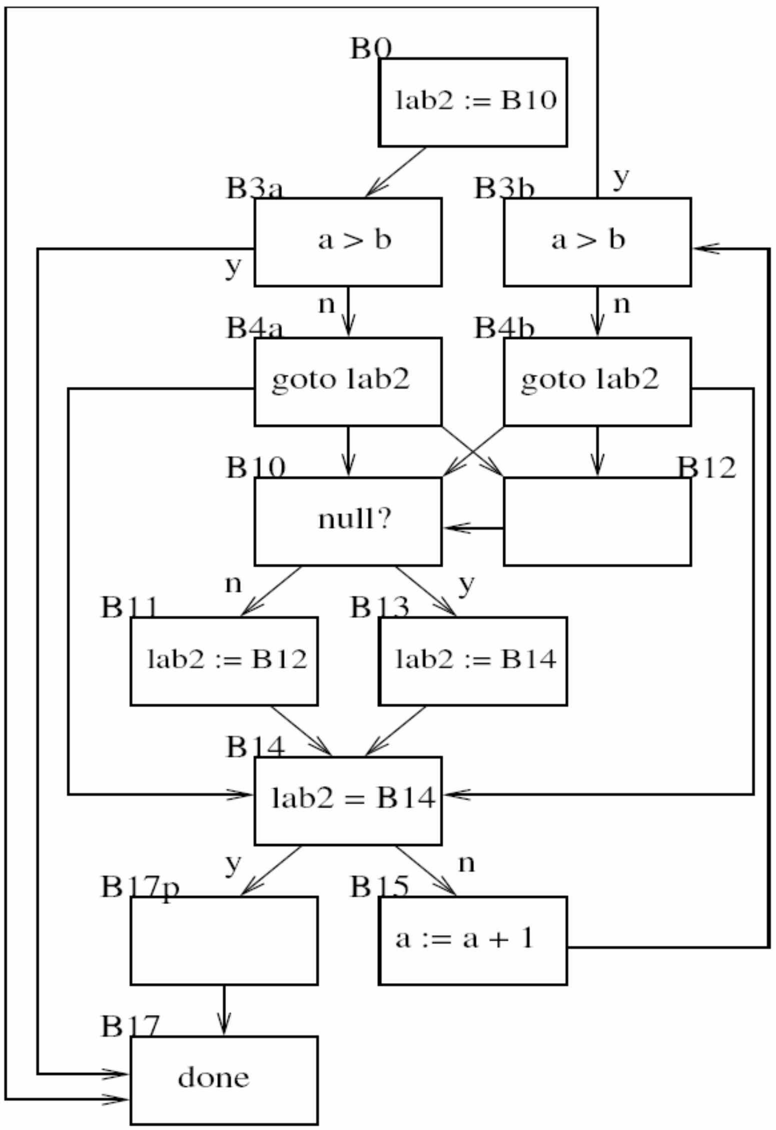
Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2	...	...	1..	1..
B1b	B15	B2 B5	...	...	.1.	.1.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B6 B4	...	...	11.	11.
B4	B3	B7b	.1.	1.1	11.	.1.
B5	B1b	B3	...	...	.1.	.1.
B6	B3	B7c	..1	11.	11.	..1
B7a	B1b	B8	...	...	.1.	.1.
B7b	B4	B8	...	...	.1.	.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

[lab1 == B2, lab1 == B5, lab1 == B7]

Resolution of 1st Iterator

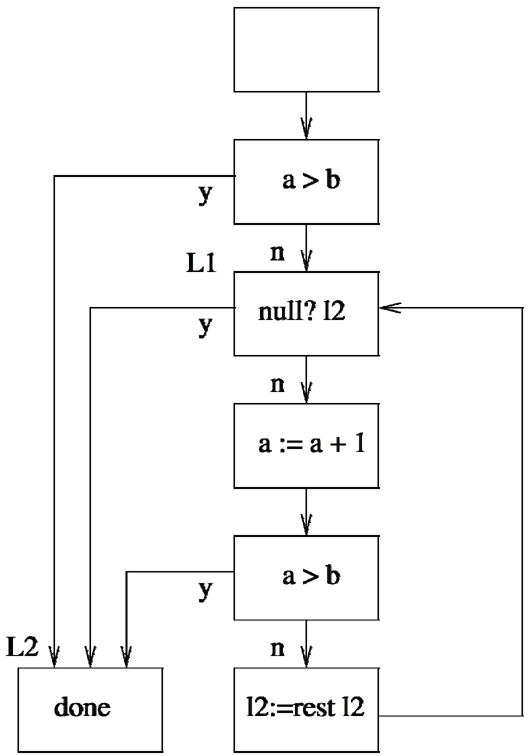


Split Blocks for 2nd Iterator



Resolution of 2nd Iterator

```
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
}
```



Conclusions

- Suspend/resume iterators are *much* easier to understand than save/restore, but have not had efficient implementation.
- Have shown a technique to implement suspend/resume iterators and a strategy to optimize the generated code.
- This is the *only* way that for loops are implemented in Aldor, giving efficient inner loops in large computer algebra library.
- Can use this to write suspend/resume iterators in terms of save/restore at source level in other languages.
(Abuse of **`switch`.**)