Errata and Additions for Section 7.7. Structural Analysis in the First and Second Printings of Advanced Compiler Design and Implementation

Steven S. Muchnick

30 June 1998

A significant error was found in the handling of improper regions in structural analysis. Fixing it provided an opportunity to overhaul and simplify the relevant code. Thus, part of p. 204 and most of pages 205 through 215 have been replaced, as follows (we first give the modifications to the figures and then the modifications to the accompanying text):

P. 204: In line 2, change "In particular" to "For example"; in line -9, change "preorder" to "postorder"; in lines -4 and -3, change "the end" to "both ends"; and, in line -2, change "successor" to "predecessor and successor".

P. 205, Figure 7.38: Insert before line 1 the line "`Succ, Pred: Node ⟶ set of Node`"; in line 4, change "`Structof`" to "`StructOf`"; and change lines -3 through -1 to the following:

```
|| postorder traversal of the flowgraph
PostCtr, PostMax: integer
Post: integer ⟶ Node
Visit: Node ⟶ boolean
```

Pp. 206–207: Replace the contents of Figures 7.39 and 7.40 by the following:

```
procedure Structural_Analysis(N,E,entry)
   N: in set of Node
   E: in set of (Node × Node)
   entry: in Node
begin
   m, n, p: Node
   rtype: RegionType
   NodeSet, ReachUnder: set of Node
   StructOf := StructType := Structures := StructNodes := ∅
   CTNodes := N; CTEdges := ∅
   repeat
      Post := ∅; Visit := ∅
      PostMax := 0; PostCtr := 1
      DFS_Postorder(N,E,entry)
      while |N| > 1 & PostCtr ≤ PostMax do
         n := Post(PostCtr)
         || locate an acyclic region, if present
         rtype := Acyclic_Region_Type(N,E,n,NodeSet)
         if rtype ≠ nil then
            p := Reduce(N,E,rtype,NodeSet)
```

```
                if entry ∈ NodeSet then
                   entry := p
                fi
             else
                || locate a cyclic region, if present
                ReachUnder := {n}
                for each m ∈ N do
                   if StructOf(m) = nil & Path_Back(m,n) then
                      ReachUnder ∪= {m}
                   fi
                od
                rtype := Cyclic_Region_Type(N,E,n,ReachUnder)
                if rtype ≠ nil then
                   p := Reduce(N,E,rtype,ReachUnder)
                   if entry ∈ ReachUnder then
                      entry := p
                   fi
                else
                   PostCtr += 1
                fi
             fi
          od
       until |N| = 1
    end   || Structural_Analysis

    procedure DFS_Postorder(N,E,x)
        N: in set of Node
        E: in set of (Node × Node)
        x: in Node
    begin
        y: Node
        Visit(x) := true
        for each y ∈ Succ(x) (Visit(y) = nil) do
            DFS_Postorder(N,E,y)
        od
        PostMax += 1
        Post(PostMax) := x
    end    || DFS_Postorder
```

and, in the caption of Figure 7.40, change “preorder” to “postorder”.

P. 208: Replace the content of Figure 7.41 by the following:

```
    procedure Acyclic_Region_Type(N,E,node,nset) returns RegionType
       N: in set of Node
       E: in set of (Node × Node)
       node: inout Node
       nset: out set of Node
    begin
       m, n: Node
       p, s: boolean
```

```
   nset := ∅
   || check for a Block containing node
   n := node; p := true; s := |Succ(n)| = 1
   while p & s do
      nset ∪= {n}; n := ◆Succ(n); p := |Pred(n)| = 1; s := |Succ(n)| = 1
   od
   if p then
      nset ∪= {n}
   fi
   n := node; p := |Pred(n)| = 1; s := true
   while p & s do
      nset ∪= {n}; n := ◆Pred(n); p := |Pred(n)| = 1; s := |Succ(n)| = 1
   od
   if s then
      nset ∪= {n}
   fi
   node := n
   if |nset| ≥ 2 then
      return Block
   || check for an IfThenElse
   elif |Succ(node)| = 2 then
      m := ◆Succ(node); n := ◆(Succ(node) - {m})
      if Succ(m) = Succ(n) & |Succ(m)| = 1
         & |Pred(m)| = 1 & |Pred(n)| = 1 then
         nset := {node,m,n}
         return IfThenElse
      || other cases (IfThen, Case, Proper)
      elif . . .
         . . .
      else
         return nil
      fi
   fi
end   || Acyclic_Region_Type
```

Pp. 208–209: Replace the content of Figure 7.42 by the following:

```
procedure Cyclic_Region_Type(N,E,node,nset) returns RegionType
   N: in set of Node
   E: in set of (Node × Node)
   node: in Node
   nset: inout set of Node
begin
   m: Node
   || check for a SelfLoop
   if |nset| = 1 then
      if node→node ∈ E then
         return SelfLoop
      else
         return nil
      fi
```

```
   fi
   if ∃m ∈ nset (!Path(node,m,N)) then
      || it's an Improper region
      nset := Minimize_Improper(N,E,node,nset)
      return Improper
   fi
   || check for a WhileLoop
   m := ◆(nset - {node})
   if |Succ(node)| = 2 & |Succ(m)| = 1 &
      |Pred(node)| = 2 & |Pred(m)| = 1 then
      return WhileLoop
   else
      || it's a NaturalLoop
      return NaturalLoop
   fi
end   || CyclicEntryType
```

P. 209: replace the content of Figure 7.43 by the following:

```
procedure Reduce(N,E,rtype,NodeSet) returns Node
   N: inout set of Node
   E: inout set of (Node × Node)
   rtype: in RegionType
   NodeSet: in set of Node
begin
   node := Create_Node( ), m: Node
   || replace node set by an abstract region node and
   || set data structures
   Replace(N,E,node,NodeSet)
   StructType(node) := rtype
   Structures ∪= {node}
   for each m ∈ NodeSet do
      StructOf(m) := node
   od
   StructNodes(node) := NodeSet
   return node
end   || Reduce
```

P. 210: Replace the content of Figure 7.44 by the following:

```
procedure Replace(N,E,node,NodeSet)
   N: inout set of Node
   E: inout set of (Node × Node)
   node: in Node
   NodeSet: in set of Node
begin
   || link region node into abstract flowgraph, adjust the postorder traversal
   || and predecessor and successor functions, and augment the control tree
   m, ctnode := Create_Node( ): Node
   e: Node × Node
```

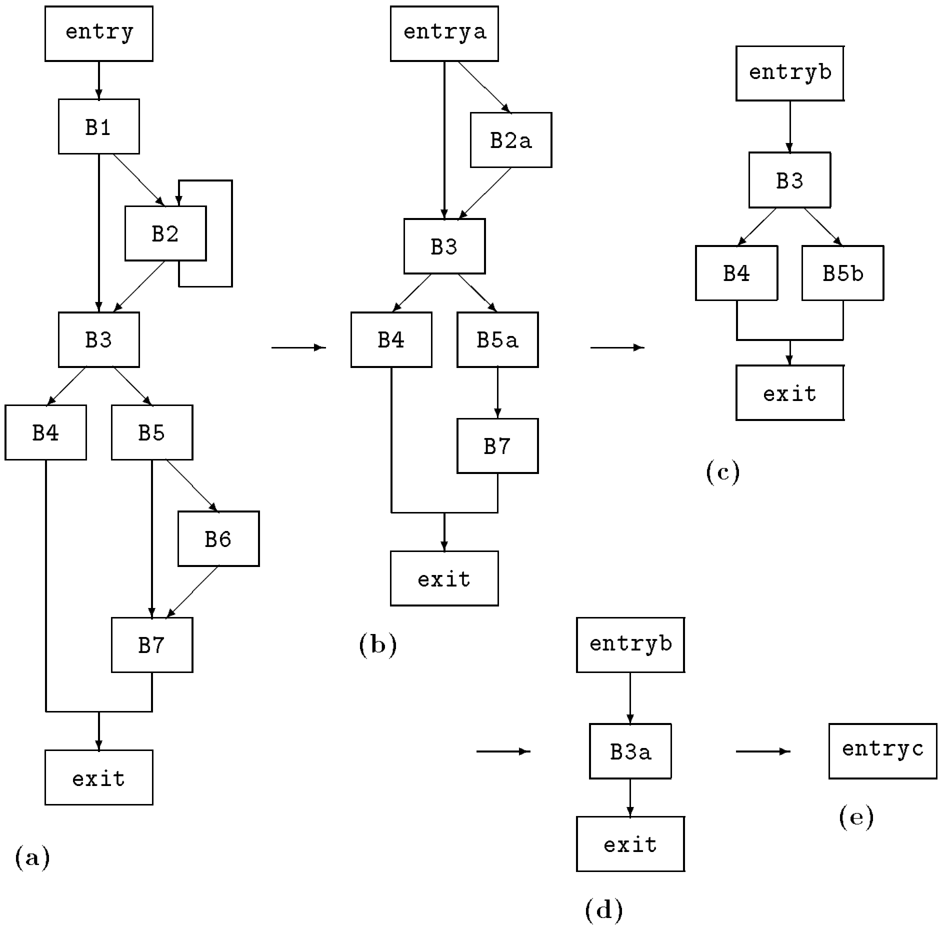
```
      N := Compact(N,node,NodeSet)
      for each e ∈ E do
         if e@1 ∈ NodeSet ∨ e@2 ∈ NodeSet then
            E -= {e}; Succ(e@1) -= {e@2}; Pred(e@2) -= {e@1}
            if e@1 ∈ N & e@1 ≠ node then
               E ∪= {e@1→node}; Succ(e@1) ∪= {node}
            elif e@2 ∈ N & e@2 ≠ node then
               E ∪= {node→e@2}; Pred(e@2) ∪= {node}
            fi
         fi
      od
      CTNodes ∪= {ctnode}
      for each n ∈ NodeSet do
         CTEdges ∪= {ctnode→n}
      od
   end   || Replace
```

P. 211: Replace the content of Figure 7.45 by the following:

```
   procedure Minimize_Improper(N,E,node,nset) returns set of Node
      N, nset: in set of Node
      E: in set of (Node × Node)
      node: in Node
   begin
      ncd, m, n: Node
      I := MEC_Entries(node,nset,E): set of Node
      ncd := NC_Domin(I,N,E)
      for each n ∈ N - {ncd} do
         if Path(ncd,n,N) & ∃m ∈ I (Path(n,m,N-{ncd})) then
            I ∪= {n}
         fi
      od
      return I ∪ {ncd}
   end   || Minimize_Improper
```

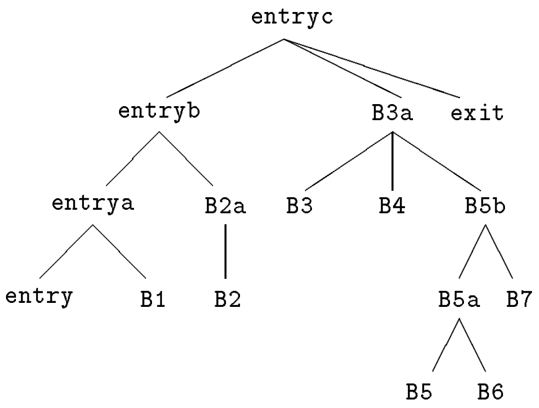
and change “Improper interval” in the caption to “Improper-interval”.

P. 212: Replace the content of Figure 7.46 by the following:



P. 212, Figure 7.47: Add a dashed line labeled "F" from `B1` to `B3`.

P. 213, Figure 7.48: Replace the content of the figure by the following:



Pp. 205–214: Replace the text by the following:

> The structural analysis algorithm `Structural_Analysis( )` given in Figure 7.39 assumes we are using the region types shown in Figures 7.35 and 7.36, plus the others described

above, although others can be used as appropriate to the language(s) being processed. The algorithm first initializes the data structures described above that record the hierarchical structure of the flowgraph and the structures that represent the control tree (`CTNodes` and `CTEdges`). Then it does a depth-first search of the flowgraph so as to construct a postorder traversal of the flowgraph's nodes. Then, in a series of passes over the flowgraph, it identifies each region and collapses it to a single abstract region node. If a reduction is performed, it repairs the sets of nodes and edges in the flowgraph and (if necessary) the postorder traversal and processes the graph again. The algorithm replaces the edges entering a region with edges to the new abstract node and edges leaving the region with edges from the new node. In parallel, it constructs the control tree.

The set `ReachUnder` is used to determine the nodes contained in a cyclic control structure. If `ReachUnder` contains a single node and there is an edge from that node to itself, then the loop is a self loop. If it contains more than one node, it can be a while loop, a natural loop, or an improper region. If the nodes in `ReachUnder` are all descendants of the first node put into it, then the loop is a while or natural loop. If it contains a nondescendant of the first node put into it, then the region is an improper one. Note that for an improper region the resulting region's entry node is *not* part of the cyclic structure, since all regions have only a single entry; the routine `Minimize_Improper( )` in Figure 7.45 determines the set of nodes making up such a region.

The computation of `ReachUnder` uses the function `Path_Back(`m`,`n`)`, which returns `true` if there is a node k such that there is a (possibly empty) path from m to k and an edge $k \rightarrow n$ that is a back edge, and `false` otherwise.

The algorithm terminates when it has reduced the flowgraph to the trivial graph with a single node and no edges. The routine `DFS_Postorder( )` given in Figure 7.40 constructs a postorder traversal of the nodes in the flowgraph. The routine `Acyclic_Region_Type(`N`,`E`,`*node*`,`*nset*`)` given in Figure 7.41 determines whether *node* is the entry node of an acyclic control structure and returns either its type or `nil` (if it is not such an entry node); the routine also stores in *nset* the set of nodes in the identified control structure.

The routine `Cyclic_Region_Type(`N`,`E`,`*node*`,`*nset*`)` given in Figure 7.42 determines whether *node* is the entry node of a cyclic control structure and either returns its type or `nil` (if it is not such an entry node); it similarly stores in *nset* the set of nodes in the identified control structure.

The routine `Reduce(`N`,`E`,`*rtype*`,`*NodeSet*`)`, defined in Figure 7.43, calls `Create_Node( )` to create a region node n to represent the identified region and sets the `StructType`, `Structures`, `StructOf`, and `StructNodes` data structures accordingly. It returns n as its value. `Reduce( )` uses `Replace( )`, which is defined in Figure 7.44, to replace the identified region by the new node, to adjust the incoming and outgoing edges and the successor and predecessor functions correspondingly, and to build the control tree represented by `CTNodes` and `CTEdges`.

The routine `Compact(`N`,`n`,`*nset*`)` used in `Replace( )` adds node n to N, inserts n into `Post( )` in the highest-numbered position of a node in *nset*, removes the nodes in *nset* from both N and `Post( )`, compacts the remaining nodes at the beginning of `Post( )`, sets `PostCtr` to the index of n in the resulting postorder, and sets `PostMax` accordingly; it returns the new value of N.

The routine `Minimize_Improper(`N`,`E`,`n`,`*nset*`)` given in Figure 7.45 is used to determine a small improper region containing n. Depending on the order of the nodes in the flowgraph given by `DFS_Postorder( )`, it limits the improper region to the smallest subgraph containing n and at least two other nodes such that (1) one of the nodes other than n dominates all the nodes in the subgraph and (2) any node on a non-empty path from n to some other node in the subgraph is also in the subgraph, or to a somewhat larger improper region that contains the smallest one as a subgraph.

`Minimize_Improper( )` uses two functions, namely, `MEC_Entries(`n`,`$nset$`,`E`)`, which returns the set of entry nodes (all members of $nset$) of the smallest multiple-entry cycle of which n is one of the entries, and `NC_Domin(`I`,`N`,`E`)`, which returns the nearest common dominator of the nodes in I. `NC_Domin( )` can easily be computed from the flowgraph's dominator tree. The function `Path(`n`,`m`,`I`)` returns `true` if there is a path from n to m such that all the nodes in it are in I and `false` otherwise.

See the discussion of Figure 7.49(b) below for an example of how the postorder affects determination of the resulting improper region. This approach almost always results in smaller improper intervals than Sharir's original method.

As an example of structural analysis, consider the flowgraph in Figure 7.46(a). Figure 7.47 shows a depth-first spanning tree for the flowgraph. The first stage of the analysis,[6] shown in Figure 7.46(b), does three reductions: `entry` followed by `B1` is recognized as a block and reduced accordingly, `B2` is recognized as a self loop and reduced, and `B5` and `B6` are recognized as an if-then and reduced. It sets the data structures as follows:

```
StructType(entrya) = Block
StructOf(entry) = StructOf(B1) = entrya
StructNodes(entrya) = {entry,B1}
StructType(B2a) = SelfLoop
StructOf(B2) = B2a
StructNodes(B2a) = {B2}
StructType(B5a) = IfThen
StructOf(B5) = StructOf(B6) = B5a
StructNodes(B5a) = {B5,B6}
Structures = {entrya,B2a,B5a}
```

The next stage, shown in Figure 7.46(c), recognizes and reduces the if-then made up of `entrya` and `B2a` and the block made up of `B5a` and `B7`. It sets the data structures as follows:

```
StructType(entryb) = IfThen
StructOf(entrya) = StructOf(B2a) = entryb
StructNodes(entryb) = {entrya,B2a}
StructType(B5b) = Block
StructOf(B5a) = StructOf(B7) = B5b
StructNodes(B5b) = {B5a,B7}
Structures = {entrya,B2a,B5a,entryb,B5b}
```

In the next stage, shown in Figure 7.46(d), `B3`, `B4`, and `B5b` are reduced as an if-then-else. The data structures are set as follows:

```
StructType(B3a) = IfThenElse
StructOf(B3) = StructOf(B4) = StructOf(B5b) = B3a
StructNodes(B3a) = {B3,B4,B5b}
Structures = {entrya,B2a,B5a,entryb,B5b,B3a}
```

In the final stage, `entryb`, `B3a`, and `exit` are reduced as a block, resulting in Figure 7.46(e). The data structures are set as follows:

```
StructType(entryc) = Block
StructOf(entryb) = StructOf(B3a) = StructOf(exit) = entryc
StructNodes(entryc) = {entryb,B3a,exit}
Structures = {entrya,B2a,B5a,entryb,B5b,B3a,entryc}
```

The resulting control tree is given in Figure 7.48.

Figure 7.49 gives two examples of flowgraphs that contain improper regions. In example (a) the routine `Minimize_Improper( )` recognizes the subgraph consisting of all nodes except `B6` as the improper interval. In (b) the improper interval(s) recognized depend on the particular postorder traversal used: if `B3` precedes `B2`, then it recognizes an improper interval consisting of `B1`, `B3`, and `B4` and then another consisting of the abstract node for that region along with `B2` and `B5`; if `B2` precedes `B3`, then it recognizes a single improper interval consisting of all five nodes.

The text of footnote 6 is as follows:

6. The figure actually shows what can be thought of as a parallel version of structural analysis that may make several reductions in each pass. This is done in the figure to save space, but could be implemented in the algorithm at the expense of significantly decreasing its understandability.

P. 215–216: Delete Exercises 7.15 and 7.17 and renumber the remaining ones as 7.15 and 7.16; in 7.16, change "a structural control-flow analysis of the routine" to "structural control-flow analyses of the routines"; and, add Exercise 7.17 as follows:

ADV 7.17 Implement in ICAN the function `MEC_Entries( )` used by `Minimize_Improper( )`.