Clean: Removing Useless Control Flow

Original C code written by *Robert Shillner*, August 1992
Webbed by *John Lu*, June 1994

June 15, 1994

Contents

List of Figures

Chapter 1

Introduction

1.1 Why

User code rarely has useless control flow, but optimized code will often have it. Optimizing passes, such as dead code elimination, constant propagation, loop-invariant code motion, node splitting, *etc.*, will change the control flow of a problem and/or remove code from basic blocks. This will create opportunities to simplify the control-flow graph (CFG). Since many optimizations create useless control flow, **clean** was written to clean up the opportunities that they create. As such, **clean** is intended to be run after other optimization passes.

1.2 What's Done

clean is a simple housekeeping optimization which removes empty blocks and other useless control flow. **clean** performs four transformations:

1. Empty block removal
2. Upward propagation of branches
3. Coalescing of blocks
4. Redundant-Branch folding

The first three deal with blocks ending in a jump operation (*i.e.,* `JMPl`). The last one deals with blocks ending with a branch.

1.2.1 Empty Block Removal

In Figure 1.1, we have an opportunity to remove an empty block. We define an empty block as a block containing only a jump operation. An empty block C is removed by taking C's labels and in-edges from C and putting them in D. Thus, all branches to C have been retargeted to D, avoiding the extra jump.

It is important that the successors of an empty block are processed before the empty block itself is processed. For example, if an empty block X with an empty successor Y is processed, X's labels and in-edges will be transferred to Y, only to be transferred again to Y's successor when Y is processed. However, if Y is processed first, Y's labels and in-edges are moved to its successor, and then X's labels and in-edges are moved to X's successor (which is the block that was Y's successor). By doing this, each label and edge is moved only once.

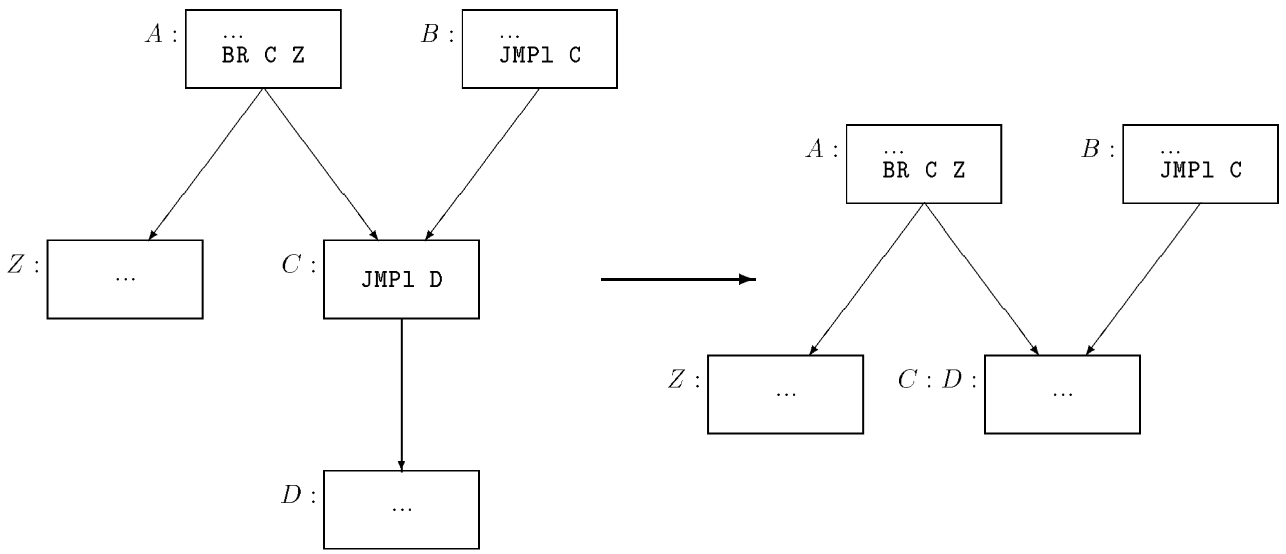


Figure 1.1: Empty Block Removal

1.2.2 Upward Propagation of Branches

In Figure 1.2, we have an opportunity to upward propagate a branch. For this transformation, we require a block ending in a jump whose successor has one instruction. This instruction must have a branch operation, but may also have other operations. Block A satisfies these conditions, so the edge connecting A and B is removed, A's jump operation is changed into a `NOP`, the branch is copied into A, and copies of B's outedges are placed in A.

Again, it is important that the successors of a block are processed before the block itself is processed. We want to be sure that any branch that is propagated is not redundant (see Subsection 1.2.4). Otherwise, branches could be copied which would later need to be folded, thus creating extra work.

Notice that we do not propagate jumps upward because they will be taken care of when we process empty blocks. If we have a jump to a jump, the second block must be an empty block. Since we process blocks in post-order, the predecessor block's jump will be retargeted when we process the successor block (see Subsection 1.2.1). When the predecessor block is processed, the opportunity to upward propagate a jump will no longer exist.

1.2.3 Coalescing of Blocks

In Figure 1.3, we can coalesce blocks B and C, since B ends in a jump, and C has only one predecessor. The edge connecting the two blocks is eliminated, and the two blocks are joined into one by taking all of the code, labels, and in-edges from B and placing them in C.

Once again, it is important that the successors of a block X are processed before X is. If X and its successor Y are coalesced, and Y and its successor Z can be coalesced, then merging X and Y before merging Y and Z will cause the in-edges and labels of X to be moved twice. Processing the successors of X before processing X solves this problem.

1.2.4 Redundant-branch Folding

In Figure 1.4, we have an opportunity to fold a redundant branch. Each target of a branch is checked, and if all of the targets are the same, the branch is transformed into a jump. The redundant edges from block A

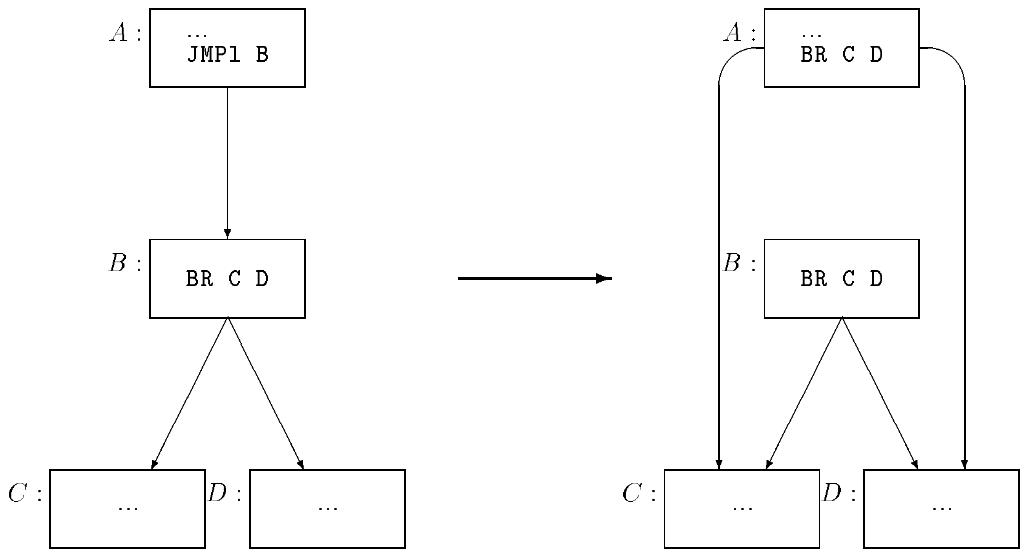


Figure 1.2: Upward Propagation of Branches

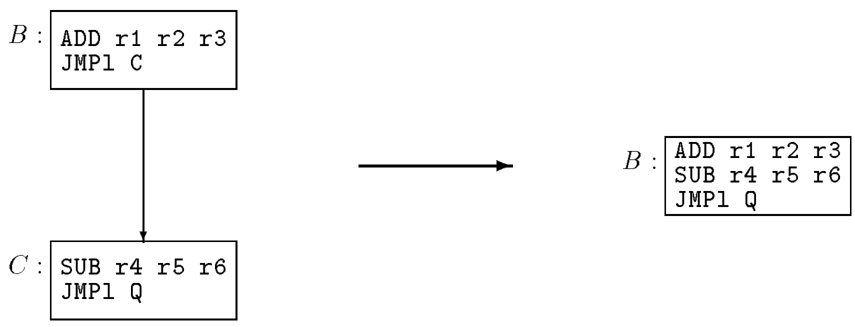


Figure 1.3: Coalescing of Blocks

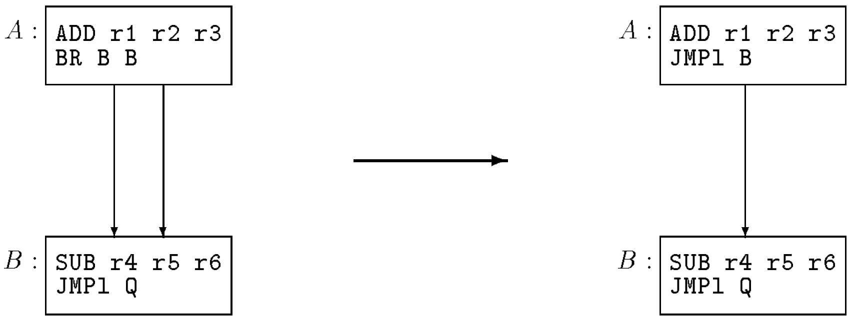


Figure 1.4: Redundant-branch Folding

to B are deleted.

It is also important in this transformation that the successors of a block be processed first. Suppose a block C ends in a branch with targets A and B. A is an empty block whose successor is B. If A is processed first, then the targets of C would both be B, and the redundant branch would be removed in one pass. If C is processed before A, C's branch will not be folded. Another pass would be needed. See Figures 3.5 and 3.6 for an example where the successors are not processed first.

1.3 Usage

clean can be invoked from the command line:

clean [`-cdt`] [*filename*]

The flags have the following effect:

- `-c` Remove all comments.
- `-d` Print some debugging output to `stderr`. Up to three -d's are useful. The first prints a message each time a transformation is applied. The second outputs the final CFG to `stderr`. The third outputs the original CFG.
- `-t` Increment the maximum number of times the `Clean` routine will be called. Adding more -t's will increment the maximum by one for each `t`. It may be useful to call the routine `Clean` more than once, because calling `Clean` may expose other cleaning opportunities (see Section 3.2). More calls to the `Clean` routine are performed only if the previous call did not stabilize, (*i.e.,* made a change to the CFG). By default, the routine is called up to three times. Three was chosen because the second pass usually produces no changes.

clean reads `ILOC`input from *filename* (or from `stdin` if no *filename* is specified) and prints its output to `stdout`.

1.4 Overall Structure

This is a quick overview of how the various parts of the program fit together. The whole program is contained in one file, `Clean.c`.

```
"Clean.c" 5a ≡
    ⟨Include Files 31⟩
    ⟨Globals 5b, ... ⟩
    ⟨Prototypes 36⟩
    ⟨The main routine 5c⟩
    ⟨Functions 8, ... ⟩
    ◇
```

1.5 The `main` Routine

The `main` routine has seven steps:

1. Parse the command line, setting flags for user-specified options and setting `inputFile` to the name of the input file. If no input file is specified, `inputFile` will be set to `NULL`, which tells the function `Block_Init` to use `stdin`.
2. Read the `ILOC`input and construct the CFG, using the `shared` routine `Block_Init`.
3. Initialize the memory allocation Arena, `cleanArena`. Arenas are described by Hanson [2]. We allocate memory in Arenas rather than using `malloc`, since this allows us to deallocate large portions of memory quickly and easily. If we are careful to put data objects with similar death times into the same Arena, we can deallocate all of them at once by destroying the Arena.

   ```
   ⟨Globals 5b⟩ ≡
       Arena cleanArena = 0;
       ◇
   ```

   Macro defined by scraps 5b, 6a, 7b.
   Macro referenced in scrap 5a.

4. Output the original CFG if desired.
5. Call the routine `Clean` once for each `-t` on the command line or until the CFG stabilizes, whichever comes first.
6. Output the processed CFG if desired.
7. Output the cleaned code.

```
⟨The main routine 5c⟩ ≡
    Void main(Int argc, Char **argv)
    {
      Char *inputFile;

      ⟨Parse the command line and set inputFile 6b⟩
      Block_Init(inputFile);
      cleanArena = Arena_Create();
      if (debug >= 3) Block_Dump_All_Simple();
      ⟨Perform calls to the Clean routine 7c⟩
      if (debug >= 2) Block_Dump_All_Simple();
      Block_Put_All(stdout);
    }
    ◇
```

Macro referenced in scrap 5a.

We need some global variables to handle the command line options. The variable `debug` is controlled by the `-d` option. The variable `iterations` is controlled by the `-t` option. We have an error string that is printed if there is a parsing error. There is also a global variable `keep_comments` that is defined in the shared library and is controlled by the `-c` option.

⟨Globals 6a⟩ ≡

```
Unsigned_Int debug = 0;
static Unsigned_Int iterations = 0;
Char errorString[] =
        "Usage: %s [-cdt] [filename]\n\n"
        "-c keep comments\n"
        "-d debug level\n"
        "-t number of cleaning passes\n";
```

◇

Macro defined by scraps 5b, 6a, 7b.
Macro referenced in scrap 5a.

We parse the command line by looping over the entries in `argv`. `argNumber` is the current index into `argv`, and `currChar` is a pointer into `argv`[`argNumber`]. We assume that any input beginning with the character '`-`' is a list of flags, and all others are filenames. If more than one filename is entered, we print out the error message and halt. If the number of iterations is not specified, we give `iterations` the default value of three.

⟨Parse the command line and set `inputFile` 6b⟩ ≡

```
{
  Unsigned_Int argNumber;
  Char *currChar;
  inputFile = NULL;

  for (argNumber = 1; argNumber < argc; argNumber++) {
    currChar = argv[argNumber];

    if (*currChar++ == '-')
        ⟨Parse flags list 7a⟩
    else if (inputFile == NULL)
      inputFile = argv[argNumber];
    else {
      fprintf(stderr, errorString, argv[0]);
      ABORT;
    }
  }

  if (iterations == 0) iterations = 3;
}
```

◇

Macro referenced in scrap 5c.

The flags list is parsed by examining each character in the list. For each valid flag, there is a corresponding global variable. If we hit a character which is not in the set of recognizable flags, we print out the usage message and halt.

⟨Parse flags list 7a⟩ ≡

```
while (*currChar) {
  switch (*currChar++) {
    case 'd': debug++;
              break;
    case 'c': keep_comments = FALSE;
              break;
    case 't': iterations++;
              break;
    default:  fprintf(stderr, errorString, argv[0]);
              ABORT;
  }
}
```

◇

Macro referenced in scrap 6b.

We may need to call the routine `Clean` more than once because a call to `Clean` may expose other opportunities for cleaning. But, more calls to `Clean` would be useless, if the previous call to `Clean` left the CFG unchanged. We use a global boolean flag `Changed` to detect this.

⟨Globals 7b⟩ ≡

```
Boolean Changed;
```

◇

Macro defined by scraps 5b, 6a, 7b.
Macro referenced in scrap 5a.

We know that we will call the `Clean` routine at least once, so we place one call outside the `for`-loop and start the counter, `i`, at one. Since the routine `Clean` may modify the CFG, we need to call `Block_Order` between calls to the routine `Clean` in order to maintain the DFS-tree for the CFG. We stop when we reach the number of iterations specified or when the CFG does not change, whichever comes first.

⟨Perform calls to the `Clean` routine 7c⟩ ≡

```
{
  Unsigned_Int i;

  Changed = FALSE;
  Clean();
  for (i = 1; Changed && (i < iterations) ; i++) {
    Block_Order();
    Changed = FALSE;
    Clean();
  }
}
```

◇

Macro referenced in scrap 5c.

Chapter 2

Details

This chapter assumes familiarity with ILOC, described in [1]. The topics relevant to **clean** are: operations, instructions, blocks, and edges.

2.1 The Clean Routine

The routine `Clean` performs a single postorder pass on the CFG. Transformations that apply to a block as it is being visited are performed. The body of the `for`-loop applies the appropriate optimization to a block by switching on the first operation of the last instruction. The code, `->inst`, gets the first instruction of the block. The code, `->prev_inst`, gets the last instruction, since instructions form a doubly-linked circular list. The code, `->operations[0]`, gets the first operation. The code, `->opcode`, gets the opcode. Blocks with jumps are handled by `ProcessJMPl`. Blocks with branches are handled with `ProcessBranch`.

⟨Functions 8⟩ ≡

```
Void Clean(Void)
{
  Block *currBlock;

  ForAllBlocks_Postorder (currBlock) {
    switch (currBlock->inst->prev_inst->operations[0]->opcode) {

    case JMPl:
      ProcessJMPl(currBlock);
      break;

    case BR:
    case BRgt: case BRge: case BRlt: case BRle: case BReq: case BRne:
    case JMPr:
      ProcessBranch(currBlock);
      break;

    default:
      break;
    }
  }
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

2.2 Branches

Redundant-branch folding is the only optimization that can be applied to a block that ends with a branch. Such blocks are handled by **ProcessBranch**, which will change a branch in which all targets are the same into a jump. The variable **succ** is used to point to the first successor of **currBlock**.

⟨Functions 9a⟩ ≡
```
static Void ProcessBranch(Block *currBlock)
{
  Block *succ;

  ⟨Decide if targets are identical 9b⟩
  ⟨Create and Install Jump 10a⟩
  ⟨Reset Edges 10b⟩
  ⟨Process new jump 11a⟩
}
```
◇

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

We check to see if the targets of the branch are identical by examining all successors. If not, there is nothing we can do, so we return. The code **currBlock->succ->succ** may look confusing, but the first **->succ** points to the list of successor edges. The second **->succ** points to the successor block of the first edge of the list of successors of the current block. If we do have an opportunity to fold the branch, we set the global **Changed** flag to **TRUE**.

⟨Decide if targets are identical 9b⟩ ≡
```
{
  Edge *succEdge;

  succ = currBlock->succ->succ;
  Block_ForAllSuccs (succEdge, currBlock) {
    if (succEdge->succ != succ)
      return;
  }
  Changed = TRUE;
}
```
◇

Macro referenced in scrap 9a.

Next, we create a new jump operation with the appropriate target, and use it to replace the old branch operation. There are two cases in finding the target for the new jump operation. If the old branch was a **JMPr** operation, the targets of the branch are held in the list argument of the operation. Since they all point to the same block, we take the first item in the list of labels for the **JMPr** and put it in the argument slot used by the target of the jump. If the old branch was not a **JMPr** (*i.e.,* it was a **BR** or **BR(cc)**), the targets are held in the constants section of the **arguments** array. Then we just use the first constant of **oldBranch** as the target of the jump.

⟨Create and Install Jump 10a⟩ ≡

```
{
  Operation *oldBranch, *newBranch;

  if (debug) fprintf(stderr,
                     "clean: Folded a redundant branch in: %s.\n",
                     Label_Get_String(currBlock->labels->label));
  oldBranch = currBlock->inst->prev_inst->operations[0];
  newBranch = Operation_Allocate(cleanArena, 1);
  newBranch->opcode = JMPl;
  newBranch->constants = 1;
  newBranch->referenced = 1;
  newBranch->defined = 1;
  newBranch->comment = oldBranch->comment;
  newBranch->source_line_ref = oldBranch->source_line_ref;

  if (oldBranch->opcode == JMPr)
    newBranch->arguments[0] = oldBranch->arguments[oldBranch->defined];
  else
    newBranch->arguments[0] = oldBranch->arguments[0];

  currBlock->inst->prev_inst->operations[0] = newBranch;
}
◇
```

Macro referenced in scrap 9a.

Next, we must restructure the CFG so that there is only a single edge from `currBlock` to `succ`. To remove the redundant edges between `currBlock` and `succ`, we search `succ`'s list of predecessors for the first edge that is connected to `currBlock`. We set `currBlock`'s list of successors to this edge and set this edge's `next_succ` field to `NULL`. Then, we still need to walk down `succ`'s list of predecessors and remove all subsequent edges connected to `currBlock`. This is done by using two pointers, `currEdge` and `prevEdge`. The edge `currEdge` points to the edge currently being examined. The edge `prevEdge` points to edge that preceeds `currEdge` on the list. When we find an edge that connects `succ` and `currBlock`, we must reset `prevEdge` to the next edge in the list that does not connect `succ` and `currBlock`. We find this edge with a `for`-loop and decrement `succ->pred_count` once for each edge skipped.

⟨Reset Edges 10b⟩ ≡

```
{
  Edge *prevEdge,*currEdge;

  for (currEdge = succ->pred; currEdge->pred!=currBlock; currEdge = currEdge->next_pred);
  currBlock->succ = currEdge;
  currEdge->next_succ = NULL;

  for(prevEdge = currEdge, currEdge = currEdge->next_pred;
      currEdge;
      prevEdge = currEdge, currEdge = currEdge->next_pred)
    if (currEdge->pred == currBlock) {
      for (; currEdge && currEdge->pred == currBlock; currEdge = currEdge->next_pred)
        succ->pred_count--;
      prevEdge->next_pred = currEdge;
      if (!currEdge)
        break;
    }
}
◇
```

Macro referenced in scrap 9a.

After restructuring the CFG, the block that we are processing ends with a jump. Thus, if `ProcessBranch` successfully folds a redundant-branch, its last action will be to call `ProcessJMPl` on the current block.

⟨Process new jump 11a⟩ ≡
```
ProcessJMPl(currBlock);
```
◇

Macro referenced in scrap 9a.

2.3 Jumps

Blocks ending with a jump are handled by the function `ProcessJMPl`. Three transformations can be performed on blocks ending with a jump: empty block removal, upward propagation of branches, and coalescing of blocks. `ProcessJMPl` tests for each of these three cases and calls an appropriate helper function if one of the cases is satisfied.

⟨Functions 11b⟩ ≡
```
static Void ProcessJMPl(Block *currBlock)
{
  Block *succ = currBlock->succ->succ;

  ⟨Process empty block 12a⟩
  ⟨Coalesce blocks 13a⟩
  ⟨Upward propagate a conditional 12b⟩
}
```
◇

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

2.3.1 Ordering

The optimizations are attempted in this order: empty block removal, coalescing of blocks, and upward propagation of branches. To see why these optimizations are done in this order, consider what happens when we have an opportunity to do more than one optimization of the three.

2.3.1.1 Empty Block and Upward Propagation of Branches

If an empty block is followed by a block consisting of only a branch operation, we could either process the empty block or upward propagate the branch. If we propagate the branch, we will have two branches and potentially more work. If we eliminate the empty block, we only have one copy of the branch. Thus, it is important that empty blocks are processed before we attempt do upward propagation of branches.

2.3.1.2 Empty Block and Coalescing

If we have two blocks that can be coalesced and the predecessor is an empty block, we could do either coalescing of blocks or empty block removal. In both cases, all the in-edges and labels to the first block will be moved to the second. The resulting graph structure will be identical. Thus, it is unimportant which optimization we do.

2.3.1.3 Upward Propagation of Branches and Coalescing

If we have two blocks that can be coalesced and the and the successor consists of only a branch operation, we can either upward propagate the branch or coalesce blocks. In either case, the resulting graph structure will be the same, but upward propagation of branches would result in the successor block having an unreachable branch. Maintaining predecessor counts is more difficult if an unreachable branch may be created. By performing upward propagation of branches last, we avoid this possibility.

2.3.2 Empty Block

If a block is empty, we can move all entries into the block to the block's successor. By doing this, we remove a jump operation from any path that goes through this block. If the block can be optimized, we set the global `Changed` flag to `TRUE` and call `MergeBlocks`, which moves all in-edges and labels of the current block to its successor and removes the edge between the current block and its successor.

⟨Process empty block 12a⟩ ≡

```
if (IsEmptyBlock(currBlock)) {
  if (debug) fprintf(stderr, "clean: Removed an empty block: %s\n",
                     Label_Get_String(currBlock->labels->label));
  Changed = TRUE;
  MergeBlocks(currBlock, succ);
  return;
}
```

◇

Macro referenced in scrap 11b.

2.3.3 Upward Propagation of Branches

If we have a jump to a branch, we can remove the jump operation and place a copy of the branch instruction at the end of the block. This will save us a jump operation on any path through the block. We look for this by checking that the successor block has one instruction which has a branch operation. The first operation of the first non-`NOP` instruction of the successor is `succ->inst->next_inst->operations[0]`. Remember that the first instruction of a block is always a `NOP`. We check to make sure this is a branch operation.

If the block satisfies these conditions, we set the global `Changed` flag to `TRUE` and call `HoistBranch`, which removes the jump and places a deep copy of the branch as the last instruction. It also modifies the CFG to reflect the changes to the current block.

`IsBranchOperation` returns `TRUE` only for `BR`, `BR(cc)`, and `JMPr` operations. Note that `JMPl` instructions are not upward propagated, since these opportunities will be handled when empty blocks are processed.

⟨Upward propagate a conditional 12b⟩ ≡

```
if (IsBranchOperation(succ->inst->next_inst->operations[0])) {
  if (debug) fprintf(stderr,
                     "clean: Propagated a branch from %s into %s.\n",
                     Label_Get_String(succ->labels->label),
                     Label_Get_String(currBlock->labels->label));
  Changed = TRUE;
  HoistBranch(currBlock);
  return;
}
```

◇

Macro referenced in scrap 11b.

2.3.4 Coalescing Blocks

We can coalesce two blocks if the `pred_count` field of the successor block is one. These blocks behave like a single basic block.[1] Thus, we can merge the code in the two blocks into one block and remove the intermediate jump.

If the block can be optimized, we set the global `Changed` flag to `TRUE` and call `ConcatenateBlocks`, which moves all the instructions in the current block into the successor. All labels in the subsequent block are also removed, and the labels and in-edges of the predecessor block are moved to the successor block.

[1] A basic block is a straight-line piece of code such that any instruction in the block executes only if every instruction in the block executes.

⟨Coalesce blocks 13a⟩ ≡

```
    if (succ->pred_count == 1) {
      if (debug) fprintf(stderr, "clean: Concatenated two blocks: %s and %s.\n",
                         Label_Get_String(currBlock->labels->label),
                         Label_Get_String(succ->labels->label));
      Changed = TRUE;
      ConcatenateBlocks(currBlock);
      return;
    }
```

◇

Macro referenced in scrap 11b.

2.4 Support Functions

There are seven support functions used by **clean**. The first, `CopyOperation`, copies operations. Two of them, `IsBranchOperation` and `IsEmptyBlock`, are `Boolean` functions. The other four, `HoistBranch`, `ConcatenateBlocks`, `MergeBlocks`, and `RemovePred`, manipulate the CFG.

2.4.1 `CopyOperation`

`CopyOperation` copies any operation. It is used in `HoistBranch` when the instruction with a branch is copied.

⟨Functions 13b⟩ ≡

```
static Operation *CopyOperation(Operation *oldOp)
{
  int          size;
  Operation    *newOp;

  ⟨Allocate space for operation 13c⟩
  ⟨Copy operation 14a⟩
  return(newOp);
}
```

◇

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

In order to allocate space for the new operation, the size of the `arguments` field must be determined. If there are no list arguments, then the size is simply the `defined` field. Otherwise, there will be `opcode_specs[oldOp->opcode].lists` number of `NULL`-terminated lists. We just count each argument until the last expected `NULL` is found.

⟨Allocate space for operation 13c⟩ ≡

```
{
  int lists;
  Unsigned_Int2 *p;

  lists =  opcode_specs[oldOp->opcode].lists;
  size = oldOp->defined;
  for (p = &(oldOp->arguments[size]); lists; ++p, ++size)
    if (!*p)
      --lists;
  newOp = Operation_Allocate(cleanArena,size);
}
```

◇

Macro referenced in scrap 13b.

Next, we must fill in the fields of the new operation.

⟨Copy operation 14a⟩ ≡

```
{
  Unsigned_Int2 *p,*q;

  newOp->opcode = oldOp->opcode;
  newOp->comment = oldOp->comment;
  newOp->source_line_ref = oldOp->source_line_ref;
  newOp->constants = oldOp->constants;
  newOp->constants = oldOp->constants;
  newOp->referenced = oldOp->referenced;
  newOp->defined = oldOp->defined;
  newOp->critical = oldOp->critical;
  for (p=newOp->arguments, q=oldOp->arguments; size; p++, q++, size--)
    *p=*q;
}
◇
```

Macro referenced in scrap 13b.

2.4.2 IsBranchOperation

`IsBranchOperation` returns `TRUE` if an operation is a `BR`, `BR(cc)`, or `JMPr` operation. It is used in the test for upward propagation of branches in `ProcessJMPl`. Notice that it will return `FALSE` for `JMPl` (see Subsection 1.2.2 for an explanation).

⟨Functions 14b⟩ ≡

```
static Boolean IsBranchOperation(Operation *operation)
{
  Opcode_Names opcode = operation->opcode;

  return ((opcode == JMPr) ||
          (opcode_specs[opcode].details & CONDITIONAL));
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

2.4.3 IsEmptyBlock

`IsEmptyBlock` is called in `ProcessJMPl` when we test for an empty block. `IsEmptyBlock` uses the standard interators to insure that the only operations are `JMPl`s and `NOP`s. Since there is exactly one control flow operation in any block, we know this is a sufficient test.

⟨Functions 15a⟩ ≡

```
static Boolean IsEmptyBlock(Block *currBlock)
{
  Inst *currInst;
  Operation **currOperPtr;

  Block_ForAllInsts (currInst, currBlock) {
    Inst_ForAllOperations (currOperPtr, currInst) {
      if ((*currOperPtr)->opcode != NOP &&
          (*currOperPtr)->opcode != JMPl)
        return FALSE;
    }
  }

  return TRUE;
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

2.4.4 HoistBranch

`HoistBranch` removes the jump in `currBlock` and adds a copy of the branch that is in the successor. It is called by `ProcessJMPl` when a jump to a branch is found.

⟨Functions 15b⟩ ≡

```
static Void HoistBranch(Block *currBlock)
{
  Block *succ = currBlock->succ->succ;
  Inst  *newInst,*succInst = succ->inst->prev_inst;

  ⟨Allocate instruction 15c⟩
  ⟨Copy operations 16a⟩
  ⟨Install instruction and remove jump operation 16b⟩
  ⟨Remove redundant edges 16c⟩
  ⟨Add new edges 17a⟩
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

First, we allocate space to copy the instruction the branch is in. To do this, we need to count the number of operations in the old instruction. `Count` is initially one since we `NULL`-terminate the array of operations.

⟨Allocate instruction 15c⟩ ≡

```
{
  int Count;
  Operation **op;

  Count = 1;
  Inst_ForAllOperations(op,succInst)
    Count++;
  newInst = Inst_Allocate(cleanArena,Count);
}
◇
```

Macro referenced in scrap 15b.

Next, we will copy operations. Note that `oldOp` and `newOp` are not pointers to operations but pointers to pointers to operations.

⟨Copy operations 16a⟩ ≡

```
{
  Operation **oldOp,**newOp;

  newOp=newInst->operations;
  Inst_ForAllOperations(oldOp,succInst) {
    *newOp=CopyOperation(*oldOp);
    newOp++;
  }
  *newOp = NULL;
}
◇
```

Macro referenced in scrap 15b.

Now, we install the instruction as the last instruction of `currBlock` and remove the jump operation.

⟨Install instruction and remove jump operation 16b⟩ ≡

```
{
  Inst *firstInst,*oldLastInst;
  Operation *jmplOp;

  firstInst = currBlock->inst;
  oldLastInst = firstInst->prev_inst;
  newInst->next_inst = firstInst;
  newInst->prev_inst = oldLastInst;
  firstInst->prev_inst = newInst;
  oldLastInst->next_inst = newInst;
  jmplOp = oldLastInst->operations[0];
  jmplOp->opcode = NOP;
  jmplOp->constants = 0;
  jmplOp->defined = 0;
  jmplOp->referenced = 0;
}
◇
```

Macro referenced in scrap 15b.

Now, we need to reset the edges. First, we remove the edge between `currBlock` and `succ`. This is done by setting the linked list of `currBlock`'s successors to `NULL` and calling `RemovePred` to remove `currBlock` from `succ`'s predecessor list. Note that we do not have to worry about the possibility that `succ` could become unreachable, thus affecting predecessor counts, because block coalescing is done first. See Subsubsection 2.3.1.3.

⟨Remove redundant edges 16c⟩ ≡

```
  currBlock->succ = NULL;
  RemovePred(succ, currBlock);
◇
```

Macro referenced in scrap 15b.

Since we copied `succ`'s branch into `currblock`, all of `succ`'s successors are now also `currblock`'s successors. We add the newly required edges by iterating over `succ`'s successors and putting an edge between the successor and `currBlock`.

⟨Add new edges 17a⟩ ≡

```
{
  Edge *succEdge;
  Edge *newEdge;

  Block_ForAllSuccs (succEdge, succ) {
    ⟨Make edge 17b⟩
    ⟨Add edge to currBlock's list of successors 17c⟩
    ⟨Add edge to successor's list of predecessors 17d⟩
  }
}
◇
```

Macro referenced in scrap 15b.

The edge to be added will be a copy of `succEdge`, except that the predecessor field will point to `currBlock`.

⟨Make edge 17b⟩ ≡

```
  newEdge = Arena_GetMem(cleanArena, sizeof(Edge));
  newEdge->pred = currBlock;
  newEdge->succ = succEdge->succ;
  newEdge->edge_extension = NULL;
◇
```

Macro referenced in scrap 17a.

Each edge is installed in `currBlock`'s successor list by inserting it at the head of the list.

⟨Add edge to currBlock's list of successors 17c⟩ ≡

```
  newEdge->next_succ = currBlock->succ;
  currBlock->succ = newEdge;
◇
```

Macro referenced in scrap 17a.

We find the target of `newEdge`, `newSucc`, and add `newEdge` to `newSucc`'s list of predecessors. Predecessor counts are also maintained while adding predecessors.

⟨Add edge to successor's list of predecessors 17d⟩ ≡

```
{
  Block *newSucc;

  newSucc = newEdge->succ;
  newEdge->next_pred = newSucc->pred;
  newSucc->pred = newEdge;
  newSucc->pred_count++;
}
◇
```

Macro referenced in scrap 17a.

2.4.5 ConcatenateBlocks

`ConcatenateBlocks` concatenates `currBlock` with its successor, `succ`. This is done by moving the instructions in `currBlock` into `succ`. The block, `succ`, must have `currBlock` as its sole predecessor. `ConcatenateBlocks` is called by `ProcessJMPl` when coalescing blocks. The variable `lastInst` is points to the last instruction of `currBlock`.

⟨Functions 18a⟩ ≡

```
static Void ConcatenateBlocks(Block *currBlock)
{
  Block *succ = currBlock->succ->succ;
  Inst *lastInst = currBlock->inst->prev_inst;

  ⟨Splice predecessor into successor 18b⟩
  ⟨Remove jump and labels 18c⟩
  ⟨Call MergeBlocks 19a⟩
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

We will move everything between the second instruction and and last instruction of the current block to its destination between the first and second instructions of the successor block. Remember, the first instruction of a block is a `NOP`.

⟨Splice predecessor into successor 18b⟩ ≡

```
{
  Inst *succFirstInst;
  Inst *succSecondInst;
  Inst *secondInst;

  secondInst = currBlock->inst->next_inst;
  succFirstInst = succ->inst;
  succSecondInst = succ->inst->next_inst;

  succFirstInst->next_inst = secondInst;
  secondInst->prev_inst = succFirstInst;
  succSecondInst->prev_inst = lastInst;
  lastInst->next_inst = succSecondInst;
}
◇
```

Macro referenced in scrap 18a.

Then, we convert the jump into a `NOP`. This is the whole purpose of this optimization. We do not remove the whole instruction, because the jump operation may be in a multi-operation instruction. All labels to the successor are also removed.

⟨Remove jump and labels 18c⟩ ≡

```
{
  Operation *jmplOp;

  jmplOp = lastInst->operations[0];
  jmplOp->opcode = NOP;
  jmplOp->constants = 0;
  jmplOp->referenced = 0;
  jmplOp->defined = 0;
}

  succ->labels = NULL;
◇
```

Macro referenced in scrap 18a.

Now, we have an empty block preceeding another block, the exact situation where the function `MergeBlocks` is needed, so we call `MergeBlocks`.

⟨Call MergeBlocks 19a⟩ ≡

```
MergeBlocks(currBlock, succ);
```

◇

Macro referenced in scrap 18a.

2.4.6 MergeBlocks

MergeBlocks merges the sequential blocks topBlock and bottomBlock by moving all labels and incoming edges of topBlock to bottomBlock and removing the edge between topBlock and bottomBlock. Any code in topBlock is lost. The block topBlock should have no outgoing edges other than to bottomBlock. MergeBlocks is called by ProcessJMP1 when an empty block is found. It is called by ConcatenateBlocks after ConcatenateBlocks has moved all the instructions of a block into its successor.

⟨Functions 19b⟩ ≡

```
static Void MergeBlocks(Block *topBlock, Block *bottomBlock)
{
  ⟨Move labels to successor 19c⟩
  ⟨Maintain the CFG 20a⟩
  ⟨Update predecessor counts 20b⟩
}
```

◇

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

The first thing we do is reset the labels of the two blocks. All labels of topBlock are moved to bottomBlock. We do this by extending topBlock's linked list of labels with bottomBlock's linked list of labels. Note that all reachable blocks not currently being processed have at least one label. Then we reset bottomBlock's pointer to its linked list of labels to topBlock's. topBlock should no longer have any labels, so its pointer to a linked list of labels is set to NULL.

⟨Move labels to successor 19c⟩ ≡

```
{
  Label *lastLabel;

  for (lastLabel = topBlock->labels;
       lastLabel->next;
       lastLabel = lastLabel->next);

  lastLabel->next = bottomBlock->labels;
  bottomBlock->labels = topBlock->labels;
  topBlock->labels = NULL;
}
```

◇

Macro referenced in scrap 19b.

Now, we remove the edge from topBlock to bottomBlock with a call to RemovePred. We also reset the edges that went into topBlock to point to bottomBlock. While we are resetting the succ field of these edges to bottomBlock, we calculate a pointer to the last edge in this linked list, lastEdge. We use this pointer to help create a single linked list of of bottomBlock's and topBlock's predecessors, and set bottomBlock->pred to this list. topBlock should have no sucessors or predecessors, so we set its pred and succ fields to NULL. Notice how the pointer surgery required to reset the linked list of edges and labels is similar.

⟨Maintain the CFG 20a⟩ ≡

```
{
  Edge *predEdge;
  Edge *lastEdge;

  RemovePred(bottomBlock, topBlock);

  Block_ForAllPreds (predEdge, topBlock) {
    predEdge->succ = bottomBlock;
    lastEdge = predEdge;
  }

  lastEdge->next_pred = bottomBlock->pred;
  bottomBlock->pred = topBlock->pred;

  topBlock->pred = NULL;
  topBlock->succ = NULL;
}
```

◇

Macro referenced in scrap 19b.

Now, we need to update the predecessor counts for both blocks. All of `topBlock`'s predecessors have become `bottomBlock`'s predecessors, so we just increment `bottomBlock`'s count by `topBlock`'s count. We do not have to subtract one for `topBlock`, since `RemovePred` already does that. `topBlock` has no predecessors, so we set its `pred_count` field to zero.

⟨Update predecessor counts 20b⟩ ≡

```
bottomBlock->pred_count += topBlock->pred_count;
topBlock->pred_count = 0;
```

◇

Macro referenced in scrap 19b.

2.4.7 RemovePred

`RemovePred` removes the edge from `sourceBlock` to `destBlock` from the predecessor list of `destBlock`. `RemovePred` is called in `HoistBranch` and `MergeBlocks`. Note that `destBlock` will still be on `sourceBlock`'s successor list. This is because when `RemovePred` is called in `HoistBranch` and `MergeBlocks`, the predecessor block will need to have no successors.[2] This can be more easily done by just setting the predecessor's list of successors to `NULL`.

There are two cases for finding the edge to be deleted. If the edge is first in the list we must reset the `pred` field of `destBlock` to the next edge. Otherwise, we use two pointers, `prevEdge` and `currEdge`, to locate the edge to be deleted. We iterate over the list of edges until `currEdge` points to the appropriate edge, and `prevEdge` points to its predecessor. Then we set `prevEdge`'s `next_pred` field to `currEdge`'s next edge. Since there must be an edge to remove, we do not test to make sure that `currEdge` is not `NULL`.

[2] In `HoistBranch`, setting the predecessor's list of successors is actually done before the call to `RemovePred`, but this does not matter.

⟨Functions 21⟩ ≡

```
static Void RemovePred(Block *destBlock, Block *sourceBlock)
{
  Edge *prevEdge,*currEdge = destBlock->pred;

  destBlock->pred_count--;
  if (currEdge->pred == sourceBlock)
    destBlock->pred = destBlock->pred->next_pred;
  else
    do {
      prevEdge = currEdge;
      currEdge = currEdge->next_pred;
      if (currEdge->pred == sourceBlock) {
        prevEdge->next_pred = currEdge->next_pred;
        break;
      }
    }
    while (TRUE);
}
◇
```

Macro defined by scraps 8, 9a, 11b, 13b, 14b, 15ab, 18a, 19b, 21.
Macro referenced in scrap 5a.

Chapter 3

Running Time

Our analysis of the running time of **clean** is divided into two Sections. In the first Section, we examine the running time for one call of the routine `Clean`. This analysis does not take into account the size of the basic blocks. Only the structure of the CFG is considered. In the second Section, we examine how many passes are needed to completely clean a graph.

3.1 One Pass

In this Section, Out will be the number of outgoing edges of a block, In will be the number of incoming edges to a block, $InSucc$ will be the number of incoming edges to a successor block, $OutSucc$ will be the number of outgoing edges from the successor block, N will be the number of nodes in the CFG, and E will be the number of edges.

3.1.1 Branch Folding

Branch folding takes O(Out) time to recognize, since we must examine the target of each branch to insure that they are the same. Branch folding takes O($InSucc$) amount of time to perform, since it takes O($InSucc$) time to find and remove the old edges connecting the block and its successor. Removing the edges only takes O(Out) time to perform, but these edges must be found among all the incoming edges to the successor block by a linear search. Since O($InSucc$) is no smaller than O(Out), performing branch folding requires O($InSucc$) time to perform. After we replace the branch with a jump, we may need more time to process the jump.

3.1.2 Empty Block

Empty block removal takes a constant amount of time to recognize. All that is done is to make sure that there is only one instruction with a single jump operation. Empty block removal takes O($In + InSucc$) time to perform. The O(In) term comes from the time needed to reset the target of the incoming edges. The O($InSucc$) term comes from the call to `RemovePred`, which performs a linear search of the incoming edges to the successor block in order to remove the edge that came from the empty block.

3.1.3 Upward Propagation of Branches

Upward propagation of branches takes a constant amount of time to recognize. We just look for a jump to a block with a lone branch operation. It takes O($OutSucc + InSucc$) amount of time to perform. We replace the jump with a copy of the branch instruction and create $OutSucc$ new edges to join with the targets of the branch. The O($InSucc$) term comes from the linear search required to remove the old edge connecting the block and its successor.

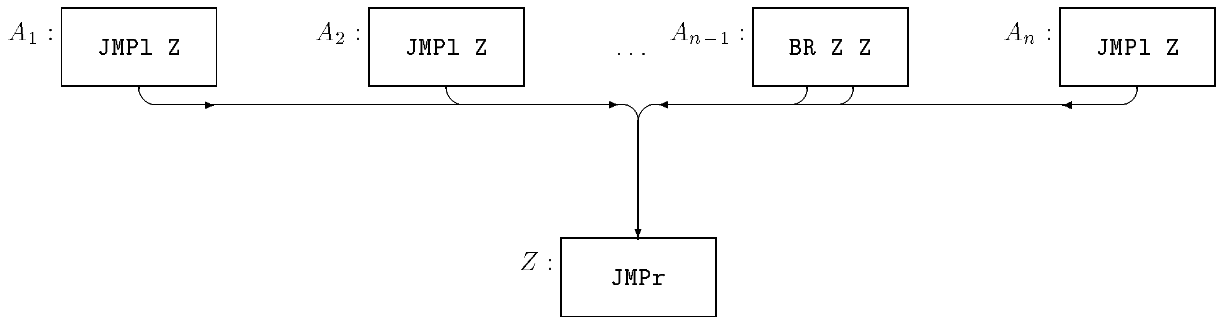


Figure 3.1: O(E^2) Running Time for One Pass, Example One

3.1.4 Coalescing Blocks

Coalescing blocks takes a constant amount of time to recognize. We just look for a block whose successor has only one predecessor. When **clean** coalesces blocks, it moves all code and labels in the top block into the bottom block, which takes constant time with respect to edges and nodes. After this, we have an empty block preceding another block, so we process the blocks with a call to `MergeBlocks`. The time bound is the same as for empty block removal, but since we know that there is only one in-edge for the successor, the time bound simplifies to O(In).

3.1.5 Analysis

There are two reasons why **clean** may take a non-linear time to perform one pass. Removing incoming edges and upward propagating a branch may require time proportional to the size of the graph. These are due to the *InSucc* and *OutSucc* terms from the previous Subsection.

In Figure 3.1, we have a sub-graph that can require O(E^2) time to process due to the *InSucc* term in the time to perform empty block removal and branch folding. There are five incoming edges into block Z. The three edges from the empty blocks must be removed, and one edge from the block A_{n-1} must be removed from the list of predecessors in block Z. If the edge is from an empty block, we would expect to have to search through half the edges before finding the edge that needs to be removed. If the edge is being removed due to redundant-branch folding, then we search through all the edges, removing all edges whose predecessor is the block with the redundant-branch, except one.

In Figure 3.2, we have a graph that takes O(E^2) time to process due to the *OutSucc* term in the time required to perform upward propagation of branches. For each of the A_i blocks, **clean** will upward propagate the bottom branch and create n new edges for the new connections. This will be done for each block, so n^2 new edges will be created.

Other than these two reasons, the graph takes O($N + E$) time to process. The only other non-constant operations we do are moving edges in empty block removal and block coalescing, and examining edges for redundancy in redundant-branch folding. Each edge will be examined at most once to check if it is from a redundant-branch, so this cannot cause non-linear behavior. Moving an edge cannot cause non-linear behavior, since an edge will be moved at most twice by one call to the routine `Clean`.

If an edge is moved by block coalescing, then it cannot be moved again. We will not process the successor block later in the pass, since the edge that joined the predecessor and the successor cannot be a backedge. Remember, any block with a backedge entering it must have at least two predecessors, since every block must have one forward edge entering it. We can only coalesce two blocks when the parent is the only predecessor

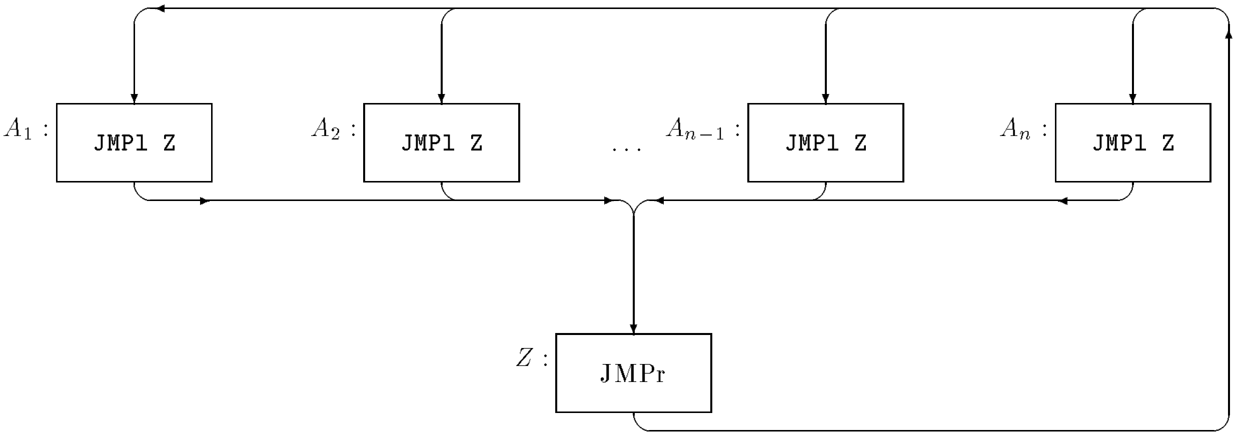


Figure 3.2: O(E^2) Running Time for One Pass, Example Two

of the child.

We cannot have two successive empty blocks whose exiting edges are backedges, since any block with an entering backedge must have an exiting forward edge. Thus, after the second empty block is processed, the edges will not be moved again.

3.2 Multiple Passes

There are two cases when multiple passes are *required* to completely clean the graph:

1. Backedge to a redundant-branch
2. Two backedges that will lead to the same block

An arbitrary number of passes *may* be needed for a certain class of graphs.

3.2.1 Backedge to a Redundant-branch

Consider Figure 3.3. Since blocks are processed in postorder, block C will be processed first. Block C's successor, block A, only has a branch, so we can upward (in this picture downward) propagate block A's branch to block C. Later in the pass, we will fold block A's redundant-branch. In the next call to the routine `Clean`, block C's copy of the redundant-branch will be folded into a jump (see Figure 3.4). Note that we cannot process the hoisted branch in the same pass in which it was hoisted (see Appendix C).

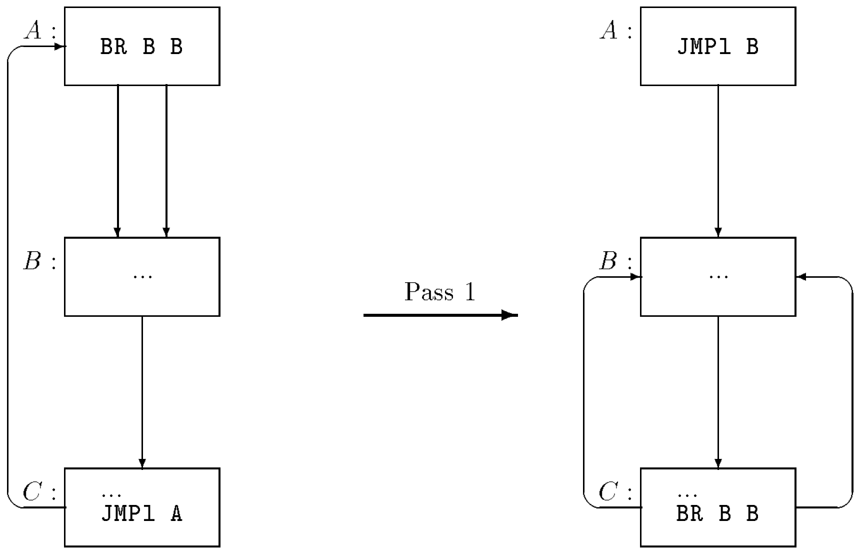


Figure 3.3: Backedge to Redundant-branch: Pass 1

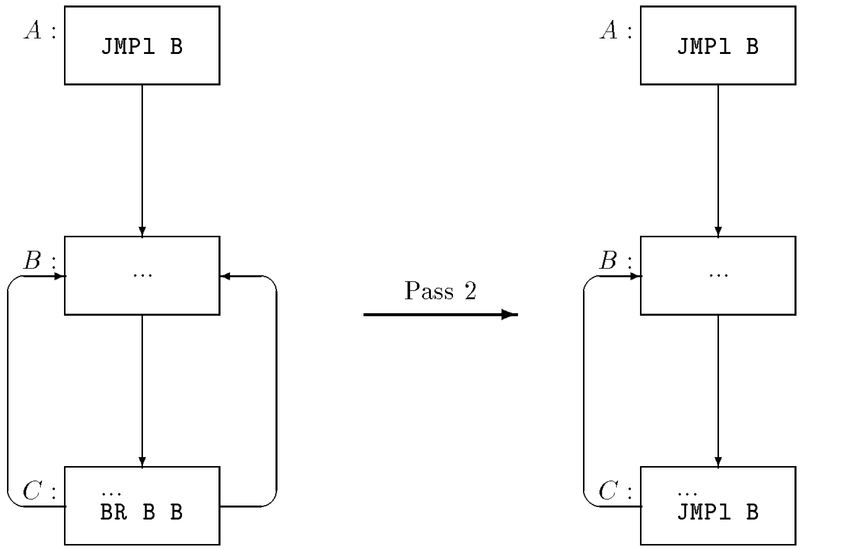


Figure 3.4: Backedge to Redundant-branch: Pass 2

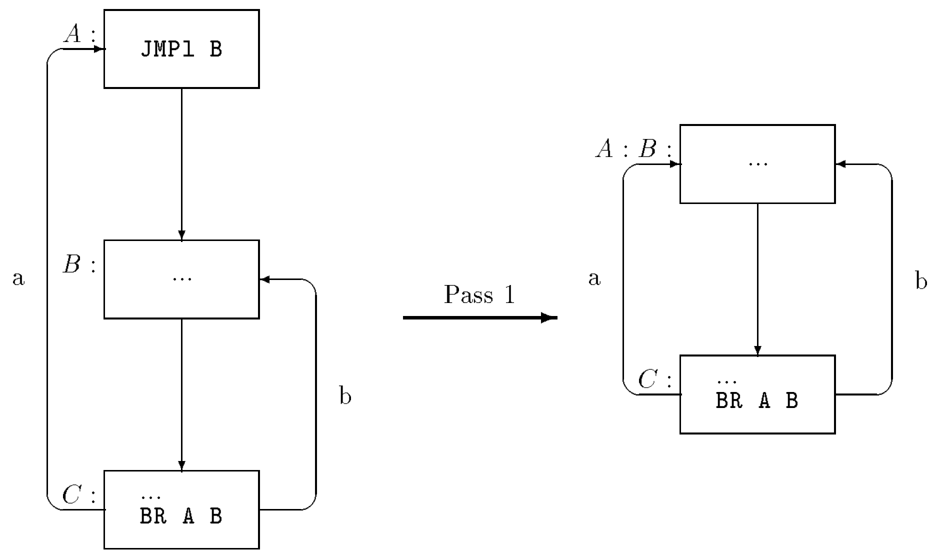


Figure 3.5: Two Backedges That Will Lead to Same Block: Pass1

3.2.2 Two Backedges That Will Lead to the Same Block

Consider Figure 3.5. On the first call to the routine `Clean`, nothing will be done to block C, since its out-edges, A and B, do not have the same destination. However, block A is an empty block, so when it is processed, edge A will be moved to block B. Now, we have an opportunity for redundant-branch folding on block C in the second pass (see Figure 3.6).

3.2.3 Arbitrary Number of Passes May Be Needed

In this Section, we will show that an arbitrary number of passes may be needed. In fact, the number of passes needed may grow linearly with the size of the CFG. This behavior depends on the depth-first search tree that `Block_Order` creates. Interestingly, if the depth-first tree is selected correctly, all of these graphs can be cleaned in a single pass. In Figure 3.7, we have a graph that may need three passes to completely clean. In this graph, there is only one redundant-branch, the second one from the bottom. However, all of these branches will eventually become redundant. But, because of our carefully constructed depth-first search tree, the only block with an obvious optimization opportunity will not be processed until all the other branches are processed. Thus, any new exposed opportunities cannot be taken in this pass. The resulting graph exhibits the same pattern as the original graph and has the same problem (see Figure 3.8). The graph will finally be cleaned on the third pass, as shown in Figure 3.9. This example can be easily be extended to *possibly* need more passes by increasing the height of the stack of branches. Note that this graph does not *require* multiple passes, since it can be cleaned in one pass, if the path from the `JMPr` to the bottom block is taken in the depth-first search tree.

3.3 Results

We tested the performance of **clean** on the `fmm` and `spec` benchmarks by running **combine**, **cprop**, **dead**, and then **clean** on them. Cleaning always stabalized in two passes, and thus only one pass produced any changes. The time to perform cleaning was always less than one-hundredth of a second for each of the files in the benchmarks.

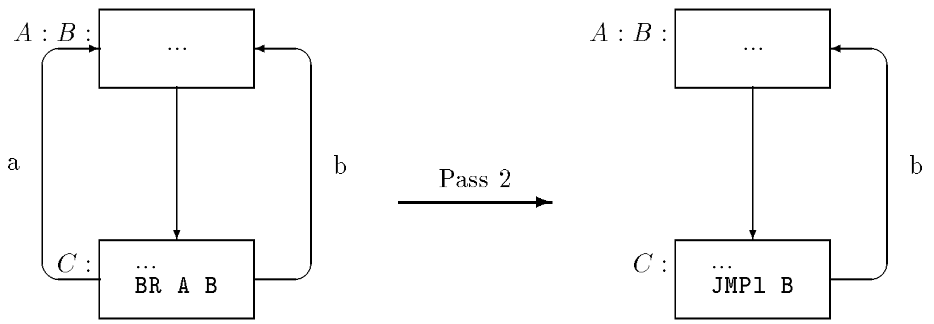


Figure 3.6: Two Backedges That Will Lead to Same Block: Pass2

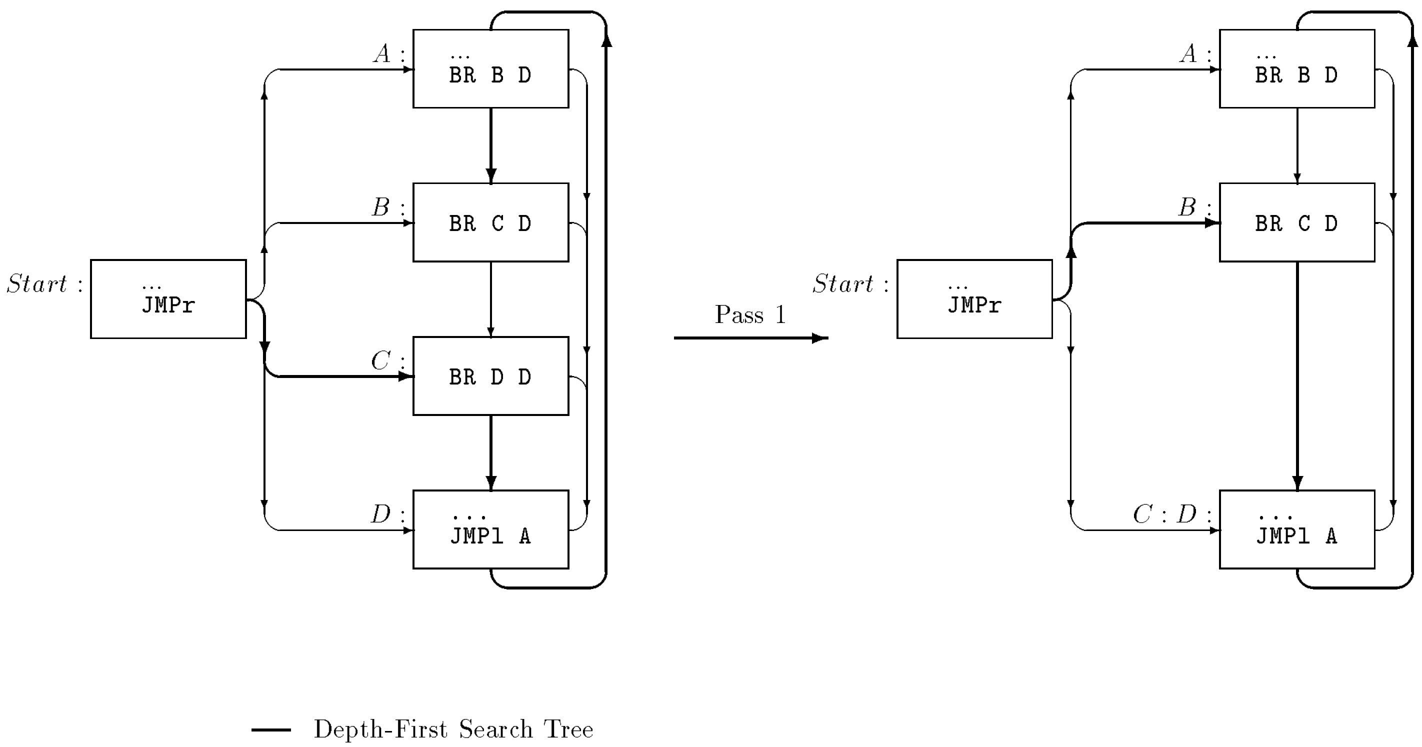
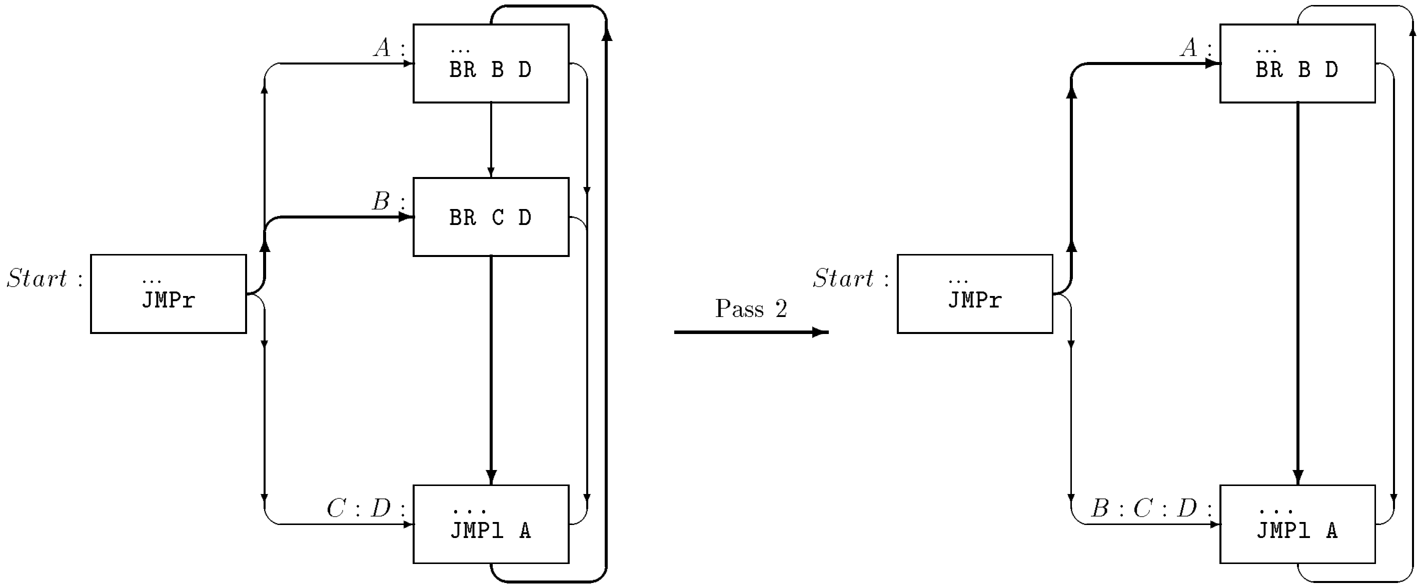
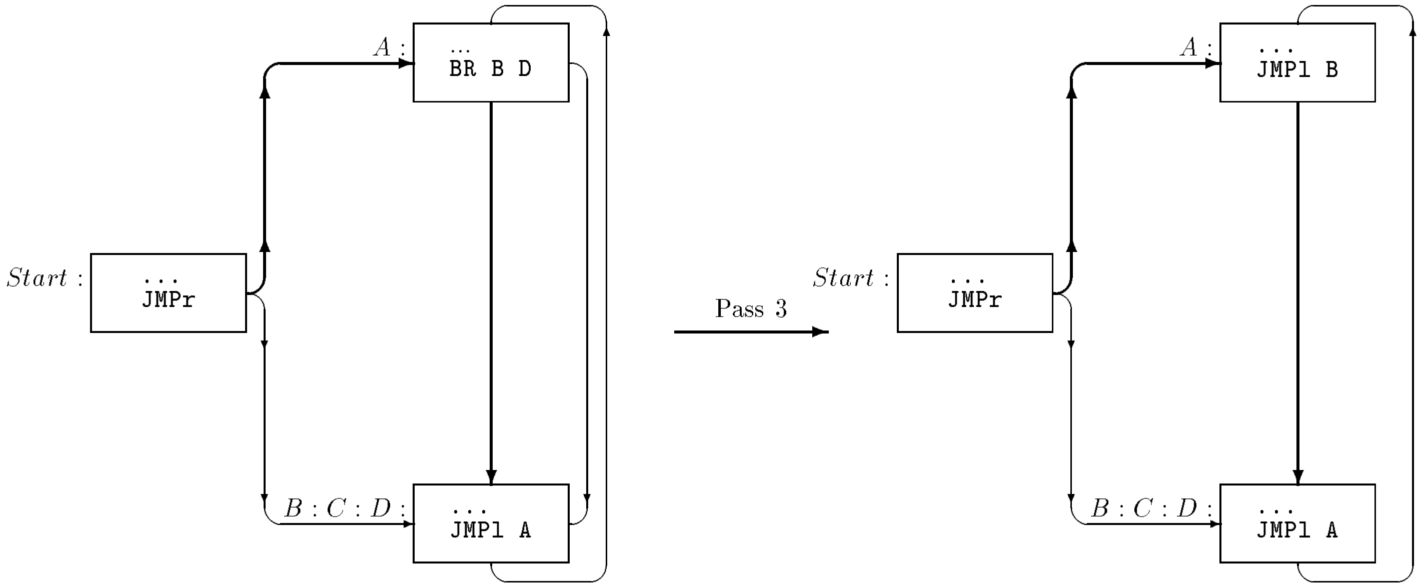


Figure 3.7: Arbitrary Number of Passes: Pass 1



— Depth-First Search Tree

Figure 3.8: Arbitrary Number of Passes: Pass 2



— Depth-First Search Tree

Figure 3.9: Arbitrary Number of Passes: Pass 3

Appendix A

Makefile

This Appendix contains the `Makefile`. The **clean** program can be made by typing `make all`. On the RS/6000, we can use dynamically loaded libraries to save lots of recompilation in the event of library maintanence. For other machines, disable the `-L` option.

"Makefile" 29 ≡

```
CC=xlc

ENV = -O

ifeq ($(ENV),-g)
FLAGS = $(ENV) -DRUNTIME_CHECK
else
FLAGS = $(ENV)
endif

OBJS=Clean.o
SHAREDDIR = /compiler/installed/shared
INC=-I. -I$(SHAREDDIR)/include
LIBS=$(SHAREDDIR)/archive/shared$(ENV).a
HDRS=$(SHAREDDIR)/web/shared.h

DYNAMIC=-L$(SHAREDDIR)/archive

all : clean

clean : $(OBJS)
	$(CC) $(FLAGS) -o clean $(OBJS) $(LIBS) $(DYNAMIC)

Clean.o : Clean.c $(HDRS)
	$(CC) $(FLAGS) -c Clean.c $(INC)
```
◇

File defined by scraps 29, 30.

The `clean.dvi` document can be read by typing `make view`. Because `clean.tex` and `clean.aux` are dependant on each other via `clean.w`, this command may leave incorrect page references.

"Makefile" 30 ≡

```
clean.aux : clean.tex
        latex clean.tex
        nuweb clean.w
        latex clean.tex

clean.bbl : clean.aux
        bibtex clean
        latex clean

clean.tex : clean.w
        nuweb clean.w

clean.dvi : clean.tex clean.aux clean.bbl
        latex clean.tex

view : clean.dvi
        xdvi clean.dvi &
```

◇

File defined by scraps 29, 30.

Appendix B

Include Files

We use the standardized include file that allows access to all of the `shared` code.

⟨Include Files 31⟩ ≡

```
#include "/compiler/installed/shared/web/shared.h"
◇
```

Macro referenced in scrap 5a.

Appendix C

Why Hoisted Branches are not Processed

When we hoist a branch into a block, we may create a new opportunity to do some more cleaning. Although we have just created another branch operation, we should not call `ProcessBranch` on this block, because we assume that the branch has already been processed. Remember, it came from a successor of the block we are considering, and we process things in postorder. This assumption is false if the edge was a backedge. In this case, we may have to make another pass (see Section 3.2.1). If **clean** did call `ProcessBranch`, it might go into an infinite loop.

In the graph in Figure C.1, the first pass will never finish processing the CFG if we allow hoisted branches to be processed in the same pass in which they are hoisted. When we process block F, we can do nothing since the two targets are different. When we process block E, an empty block, we move all of E's labels and in-edges into block F. Now, we have created an opportunity to hoist a branch into block D. This branch is also a redundant branch (see Figure C.2). If we process hoisted branches, the branch in block D will be converted into a jump (see Figure C.3). This will result in the same CFG we started with in C.2 and thus cause an infinite loop.

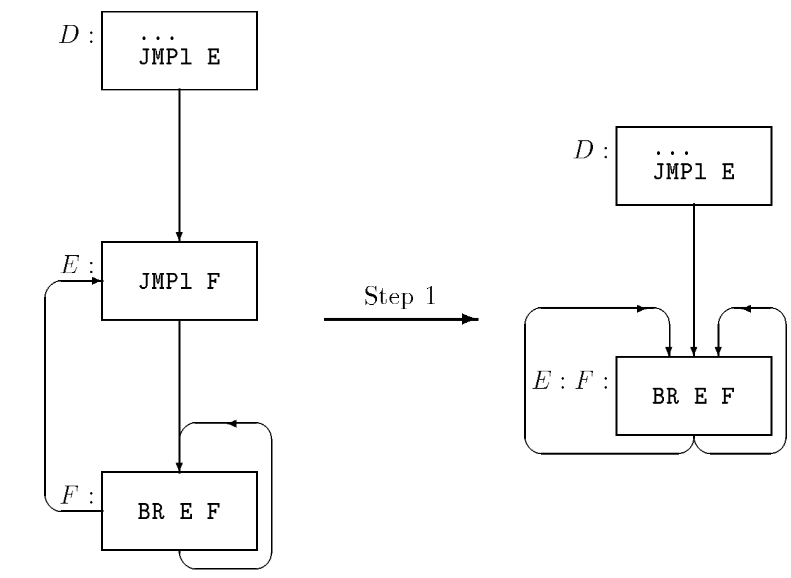


Figure C.1: CFG That Can Cause Infinite Loop: Step 1

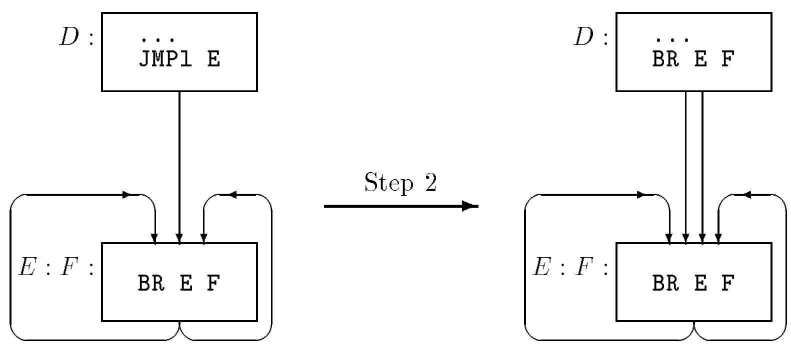


Figure C.2: CFG That Can Cause Infinite Loop: Step 2

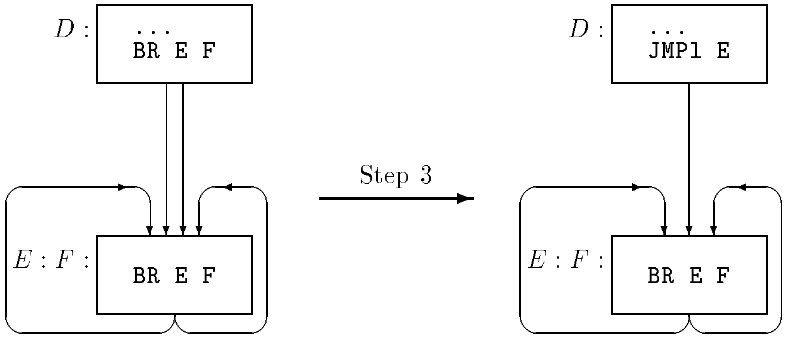


Figure C.3: CFG That Can Cause Infinite Loop: Step 3

Appendix D

Memory Use

There is only one Arena created for **clean**, so all memory allocated is reclaimed at the same time, the end of the program.

Appendix E

Prototypes

These are the prototypes for all the functions in **clean**.

⟨Prototypes 36⟩ ≡

```
Void Clean();

static Operation *CopyOperation(Operation *op);

static Void ProcessJMPl(Block *currBlock);
static Void ProcessBranch(Block *currBlock);

static Void HoistBranch(Block *currBlock);
static Void ConcatenateBlocks(Block *currBlock);

static Boolean IsBranchOperation(Operation *operation);
static Boolean IsEmptyBlock(Block *currBlock);

static Void MergeBlocks(Block *topBlock, Block *bottomBlock);
static Void RemovePred(Block *destBlock, Block *sourceBlock);
```

◇

Macro referenced in scrap 5a.

Appendix F

Indices

F.1 Index of File Names

F.2 Index of Macro Names

F.3 Index of Identifiers

Bibliography

[1] Preston Briggs. The massively scalar compiler project. Technical Report ONR grant N00014-91-J-1989, Rice University, July 1995. Not yet available.

[2] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software – Practice and Experience*, 20(1):5–12, January 1990.