Dead Code Elimination

Original C code written by *Preston Briggs*, October 1991
Extended by *Robert Shillingsburg*, June 1992
Rewritten and extended by *Taylor Simpson*, June 1993

October 12, 1993

[0]This work has been supported by ARPA, through ONR grant N00014-91-J-1989.

Contents

List of Figures

Chapter 1

Introduction

Dead code elimination is an optimization which removes all operations whose results are not used (either directly or indirectly) to produce output. In figure 1.1, statement 3 is used directly to produce output. Statement 2 is used indirectly to produce output because it computes the value used by statement 3.

1)	X $\leftarrow$ 5
2)	X $\leftarrow$ 3
3)	print X

Figure 1.1: Program Containing Dead Code.

The value computed by statement 1 can never be used to produce output because it is immediately overwritten by statement 2. Therefore, statement 1 is considered *dead*, and statements 2 and 3 are considered *live*. Statement 1 can be eliminated without changing the results of the program.

Few source programs contain dead code, but many optimizations may create dead code. Consider the following example:

1)	X $\leftarrow$ 5
2)	Y $\leftarrow$ X
3)	print Y

Figure 1.2: Source Program Containing No Dead Code.

The program in figure 1.2 contains no dead code. However, if value numbering (refer to [2]) is applied to the program, the result will be:

1)	X $\leftarrow$ 5
2)	Y $\leftarrow$ X
3)	print X

Figure 1.3: Program After Value Numbering.

If the value of Y computed in statement 2 is not used elsewhere in the program, statement 2 has become dead.

Since many optimizations can introduce dead code into a program, it is natural to eliminate it during a separate optimization pass. This will simplify many optimizations by moving the task of dead code elimination to a single place.

1.1 Basic Algorithm

We use a work list algorithm similar to the one described in [2] to eliminate the dead code from an `ILOC` routine. The process begins with a set of *critical* operations which directly affect the routine's output (e.g. procedure calls and returns). Initially, only these operations are marked critical and all other operations are marked not critical. The work list is initialized to contain the operations in the critical set. The algorithm then proceeds by removing an operation O from the worklist, marking O critical, and placing onto the worklist any operations which produce results used by O and have not yet been marked critical. This process continues until the work list becomes empty, at which point those operations which have not been marked critical are assumed not to affect program output. These noncritical operations are eliminated.

```
worklist ← {operations which directly affect output}
while (worklist ≠ ∅)
    Remove an operation O from worklist
    Mark O critical
    For each item I referenced by O
        For each definition site D for I
            if (D is not marked critical)
                Add D to worklist
Eliminate all operations not marked critical
```

Figure 1.4: Basic Dead Code Elimination Algorithm.

In order to find the definition sites of the items used in critical operations, a dead code eliminator must have access to some form of use-def chain for each item. **dead** uses static single assignment (SSA) form to create the use-def chains (see chapter 2 for more information on SSA).

In addition to the basic dead code elimination functionality, **dead** folds up dead conditional branches. A dead conditional branch is a branch upon which no live code is control dependent. For example, an if statement with no live code in either part is a dead conditional branch:

```
Y ← 7
X ← 1
if (X ≠ 0)
    X ← 5
else
    X ← 1
print Y
```

Figure 1.5: Program with a Dead Conditional Branch.

If **dead** is applied to the program in figure 1.5, the if statement will be eliminated:

Y ← 7
print Y

Figure 1.6: Program with Dead Conditional Branch Eliminated.

A consequence of the use of control dependence is that **dead** preserves only partial correctness. That is, **dead** may cause some divergent programs to terminate, specifically those with infinite loops containing no live code. For example, the program in figure 1.7 will execute forever without producing any output:

X ← 1
while (X $\neq$ 0)
    X ← 5
Y ← 5
print Y

Figure 1.7: Program with Infinite Loop Containing No Live Code.

The body of the while loop in figure 1.7 contains no live code, so **dead** will remove the loop. The result will be a terminating program that prints the number 5. Infinite loops which contain live code are preserved by **dead**.

dead also identifies operations which reference uninitialized registers. When such an operation is found, the operation is removed.

1.2 Usage

dead can be invoked from the command line as follows

`dead` [`-cdt`] [*filename*]

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

The following options are available:

`-c` Remove all comments from the routine.

`-d` Print some debugging output to `stderr`. Each `d` that appears in a flags list will increase the level of debugging information provided. The following levels are available:

1. List the major phases of the program as they are entered
2. List the minor phases of the program as they are entered
3. List operations containing uninitialized registers

`-t` Print timing information for the major phases (not currently implemented).

1.3 Performance

We ran **dead** on five routines from three programs in the SPEC benchmark. For each routine, we recorded the amount of time **dead** took in each of its major phases. We ran all of the experiments on an IBM RS/6000 Model 540, running at 30MHz, with a 64KB data cache. Figure 1.8 shows the number of basic blocks, registers and operations in each routine, as well as the time (in seconds) consumed by each phase. In all cases, the conversion to SSA form is the most time consuming step.

Program	fpppp		tomcatv	doduc	
Routine	fpppp	twldrv	tomcatv	repvid	iniset
Blocks	2	333	91	127	446
Registers	9214	5902	946	831	2283
Operations	22351	16205	2690	1884	6731
Convert to SSA	0.66	1.27	0.13	0.09	0.32
Mark Critical Operations	0.31	1.43	0.05	0.03	0.09
Convert from SSA	0.10	0.07	0.01	0.01	0.04
Total	1.07	2.77	0.19	0.13	0.45

Figure 1.8: Performance Data for **dead**.

1.4 Overall structure

This is simple boilerplate that specifies how the major components of the C program will fit together. We include several of the standard compiler include files:

Compiler_Types.h Gives portable declarations for the standard data types

Parser.h Declarations for routines, variables, and macros to read, write, and manipulate the `ILOC` routine

Arena.h Declarations for the memory allocation arena data structures and routines

SparseSet.h Declarations for the sparse set data structures and routines

Dominator.h Declarations for `Dominator_CalcDom` and `Dominator_CalcPostDom`

Time.h Declarations for execution timing routines.

"dead.c" 4 ≡

```
#include <Compiler_Types.h>
#include <Parser.h>
#include <Arena.h>
#include <SparseSet.h>
#include <Dominator.h>
#include <Time.h>

⟨Type Declarations 9a, ... ⟩
⟨Macros 7a, ... ⟩
⟨Global Variables 6b, ... ⟩
⟨Prototypes 49a, ... ⟩
⟨Functions 9c, ... ⟩
⟨The main routine 6a⟩
◇
```

1.5 The `main` Routine

The `main` routine has seven steps:

1. Parse the command line, setting flags for user-specified options and setting the string `inputFile` to the name of the input file (if the user specifies no input file, `inputFile` will be left `NULL`, which tells `Block_Init` to use `stdin`).

2. Initialize the memory allocation arenas. We allocate memory in arenas rather than using `malloc`. This allows us to deallocate large protions 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.

3. Read the `ILOC` input and construct the control flow graph, using the shared routine `Block_Init`. Next, we call the shared routine `Block_Find_Infinite_Loops`. This routine ensures that there is a path from each block to the `end_block`. This property is important when calculating post-dominance. At this point, we calculate the dominance and post-dominance information for the routine. The dominance information is used during the conversion to SSA form, and the post-dominance information is used during the marking of critical operations.

4. Convert the routine into SSA form. This conversion renames all the items in the routine so that there is a unique definition point for each item in the routine. During this phase, we also construct a table containing these definition points.

5. Find and mark the critical operations. The `Block_Init` routine automatically marks certain operations as critical. For each operation marked critical, we find the definition points for each item referenced by the operation. The defining operations are also marked critical and added to the critical set. The process terminates when no more operations can be marked critical.

6. Convert the routine out of SSA form. During this phase, we restore the original names to all the items in the routine.

7. Print only those operations that have been marked critical. This is accomplished by setting the `print_all_operations` flag to `FALSE` and calling the shared routine `Block_Put_All` to output the routine.

After each phase of the program, we print some timing and memory usage information. The printing of the timing information is controlled by the `print_time` flag which will be turned on during parsing of the command line if the user specified the `-t` option. The variable `time` keeps track of the time spent along the way. The `Time_Dump` routine prints the timing information. The printing of the memory usage information is controlled by the version of the shared library which is linked with **dead**. If the debug version of the library is used, the information will be printed; otherwise, it will not. The `Arena_DumpStats` routine prints the memory usage information.

⟨The `main` routine 6a⟩ ≡

```
Void main(Unsigned_Int argc, Char **argv)
{
    Char *inputFile = NULL;
    Timer time = Time_Start();

    ⟨Parse the command line 7c⟩
    ⟨Initialize the arenas 8c⟩

    Block_Init(inputFile);
    Block_Find_Infinite_Loops();
    Dominator_CalcDom(start_block, SSAArena);
    Dominator_CalcPostDom(end_block, lastArena);
    Time_Dump(time, "Parsing required %1.2f seconds\n");
    Arena_DumpStats();

    ⟨Convert the routine to SSA form 12⟩
    Time_Dump(time, "Converting to SSA required %1.2f seconds\n");
    Arena_DumpStats();

    ⟨Find and mark critical operations 34⟩
    Time_Dump(time, "Finding critical operations required %1.2f seconds\n");
    Arena_DumpStats();

    ⟨Convert the routine out of SSA form 41⟩
    Time_Dump(time, "Converting out of SSA required %1.2f seconds\n");
    Arena_DumpStats();

    print_all_operations = FALSE;
    Block_Put_All(stdout);
    Time_Dump(time, "Printing required %1.2f seconds\n");
    Arena_DumpStats();
    free(time);

    exit(0);
}
◇
```

Macro referenced in scrap 4.

We'll need some global variables to handle the command line options. Note that the `-c` option is controlled by the `keep_comments` variable defined in the shared library. The `-d` option is controlled by the `debug` variable; it has three significant values. The `-t` option is controlled by the `time_print` variable defined in the shared library.

⟨Global Variables 6b⟩ ≡

```
static Unsigned_Int debug = 0;
◇
```

Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

⟨Macros 7a⟩ ≡

```
#define MAJOR_PHASES          1
#define MINOR_PHASES          2
#define UNINITIALIZED_REG     3
◇
```

Macro defined by scraps 7ab, 17ade.
Macro referenced in scrap 4.

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 file names. If more than one file name is entered, we assume the user entered something wrong. In this case we print out the usage message and halt.

⟨Macros 7b⟩ ≡

```
#define UsageString          "Usage: %s [-cdt] [filename]\n"
◇
```

Macro defined by scraps 7ab, 17ade.
Macro referenced in scrap 4.

⟨Parse the command line 7c⟩ ≡

```
{
    Unsigned_Int argNumber;

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

        if (*currChar++ == '-')
            ⟨Parse a flags list 8a⟩
        else if (!inputFile)
            inputFile = argv[argNumber];
        else
        {
            fprintf(stderr, UsageString, argv[0]);
            ABORT;
        }
    }
}
◇
```

Macro referenced in scrap 6a.

We parse a flags list by looking at one character at a time. 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 a flags list 8a⟩ ≡

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

Macro referenced in scrap 7c.

1.6 Memory Allocation Arenas

We allocate memory in three arenas. Each arena contains variables with similar death times. Refer to appendix A for a list of variables contained in each arena.

⟨Global Variables 8b⟩ ≡

```
static Arena phiTempsArena = NULL;
static Arena SSAArena = NULL;
static Arena lastArena = NULL;
◇
```

Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

We can create all the arenas at once because creation only allocates a small amount of memory (used for housekeeping). More memory will be allocated when it is requested from the arena.

⟨Initialize the arenas 8c⟩ ≡

```
phiTempsArena = Arena_Create();
SSAArena = Arena_Create();
lastArena = Arena_Create();
◇
```

Macro referenced in scrap 6a.

1.7 The block_extension Structure

Each basic block created by `Block_Init` contains a pointer to a `block_extension` structure. This structure allows pass-specific information to be added to each block of the routine. This scrap merely creates the structure to hold the information we need. The actual fields (and how to initialize and print them) will be introduced as they are needed.

⟨Type Declarations 9a⟩ ≡
```
struct block_extension
{
    ⟨block_extension fields 17b, ... ⟩
};
◇
```
Macro defined by scraps 9a, 16b, 22a, 23a, 24b.
Macro referenced in scrap 4.

We must create and initialize the `block_extension` structure for each block. This could be done on any early pass through the blocks. It happens to be done while we are finding live items. The initialization of the individual fields will be presented with the declarations.

⟨Create and initialize the block_extension structure 9b⟩ ≡
```
{
    Block_Extension extension =
        (Block_Extension)Arena_GetMem(lastArena,
                                      sizeof(struct block_extension));

    ⟨Initialize block_extension fields 17c, ... ⟩
    currBlock->block_extension = extension;
}
◇
```
Macro referenced in scrap 13a.

The `Block_Put_All` function allows us to pass a function that will print out the `block_extension`. This is especially useful for invoking from **dbx**. The printing of the individual fields will be presented with the declarations.

⟨Functions 9c⟩ ≡
```
static Void ExtensionPrinter(Block_Ptr block)
{
    Block_Extension block_extension = block->block_extension;

    if (block_extension)
    {
        ⟨Print block_extension fields 18, ... ⟩
    }
}
◇
```
Macro defined by scraps 9c, 27, 40c.
Macro referenced in scrap 4.

Chapter 2

Convert to SSA Form

The following code is a modification of the algorithm given in [1] to convert a routine into static single assignment (SSA) form. The primary feature of SSA is to have only one definition point for each register or tag in the routine. This property allows easy construction of use-def chains for each item in a routine. Throughout this document, we use the term *item* to refer to either a register or a memory tag.

The basic idea used in constructing SSA form is to give unique names to the targets of all assignments in the routine, and then to overwrite uses of the assignments with the new names. A complication arises in routines with more than one basic block. Values can flow into a block from more than one definition site, and each site has a unique SSA name for the item. Consider the following example:

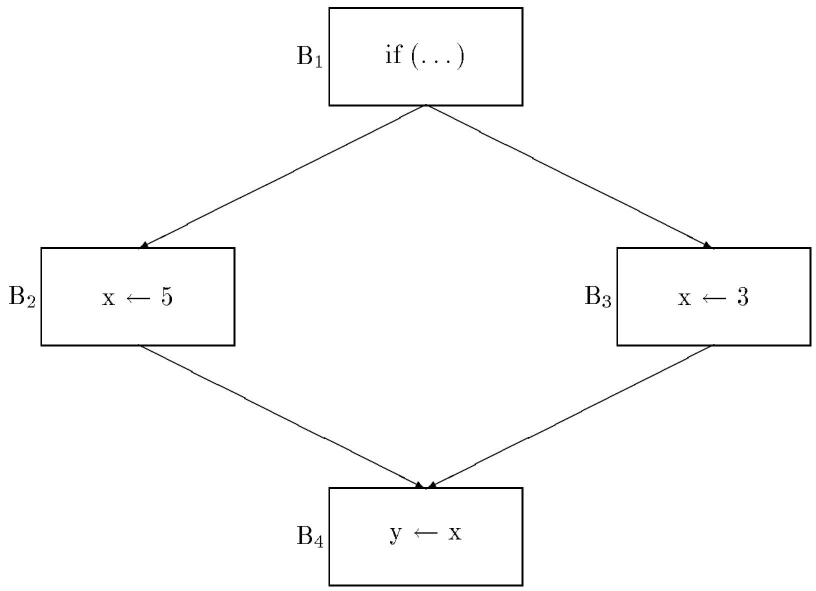

Figure 2.1: Program Before Conversion to SSA.

There are two definition points for the variable x referenced in B_4. If each definition point is overwritten with a unique SSA name, how can we correctly replace the reference to x? Special assignments called ϕ-nodes are used to solve the problem. These ϕ-nodes are placed at the routine's join points (that is, at the beginning of the basic blocks which have more than one predecessor). One ϕ-node is inserted at each join point for each item in the original routine. This item is called the ϕ-node's original item. In practice, to save space and time, ϕ-nodes are placed at only certain join points and for only certain items. We use a modified form of SSA (similar to pruned SSA), which reduces the number of ϕ-nodes used. Instead of inserting ϕ-nodes for all items in the routine, we only insert ϕ-nodes for items which are live on entry to some basic block in the routine.

The ϕ-nodes provide a single definition for an item that had more than one definition (on different control-flow paths) in the original routine. Each ϕ-node defines a new name for the original item as a function of all of the SSA names which are current at the end of the join point's predecessors. Any uses of the original item in the basic block are replaced by the new name. The number of inputs for a ϕ-node is equal to the number of predecessors of the block in which the ϕ-node appears. The ϕ-node selects the value of the input that corresponds to the block from which control is transferred and assigns this value to the result. When ϕ-nodes are properly inserted, it becomes possible for any routine to be transformed into an equivalent routine in which each item has exactly one definition point.

For the above example, the result would be:

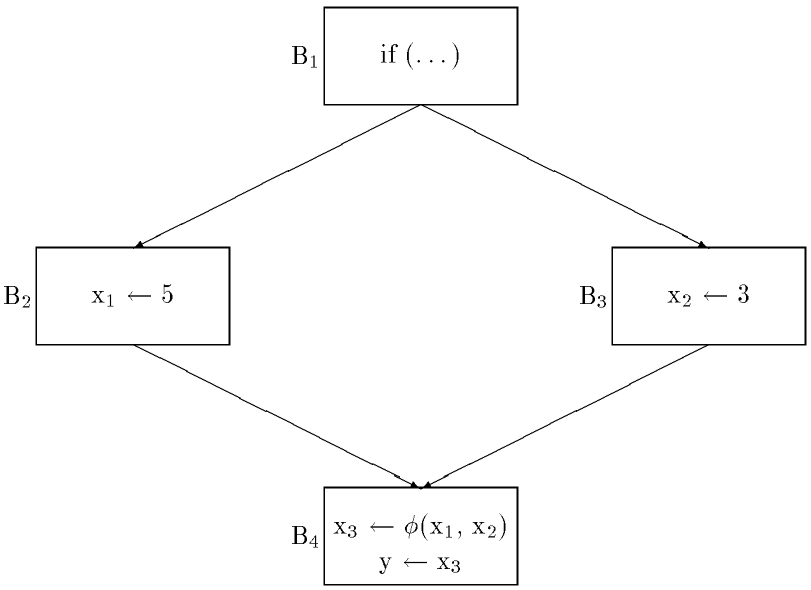

Figure 2.2: Program After Conversion to SSA.

If control is transferred from block B_2 to B_4, the ϕ-node will assign x_3 the value from x_1. If control is transferred from block B_3 to B_4, the ϕ-node will assign x_3 the value from x_2.

The process used to convert the routine to SSA form has four steps.

1. Find the interesting items (those which are live on entry to any block). The variable `liveSet` keeps track of which items we've found to be live on entry to some basic block. This set is created with a universe size equal to `register_count` + `tag_count`. The registers occupy the first `register_count` indices, and the tags occupy the last `tag_count` indices. The tag numbers are adjusted by `Block_Init` to fall within the proper range.

2. For each item in `liveSet`, find the basic blocks which contain assignments to that item. We use an array called `assignments` which is allocated to contain a list of basic blocks for each item. Only members of `liveSet` will have valid entries in the `assignments` array. Notice that the allocation also clears the memory so that all the lists are initially empty.

3. For each item x in `liveSet`, we use a worklist algorithm to place ϕ-nodes for x. Initially, the worklist contains the blocks in `assignments`$[x]$. For every block b on the worklist, we iterate over the blocks f in the dominance frontier of b, placing a ϕ-node for x in f and adding f to the worklist.

4. Give all of the items new SSA form names. Note that **dead**'s use of SSA form does not require the defined items to be actually renamed, so we rename only the referenced items.

⟨Convert the routine to SSA form 12⟩ ≡

```
{
    SparseSet liveSet =
        SparseSet_Create(phiTempsArena, register_count + tag_count);
    Block_List_Ptr *assignments =
        Arena_GetMemClear(phiTempsArena,
                          (register_count + tag_count)*sizeof(Block_List_Ptr));

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Convert the routine to SSA form\n");

    ⟨Put live items in liveSet 13a⟩
    ⟨Find lists of blocks which assign each item in liveSet 15⟩
    ⟨Insert the necessary φ-nodes 19a⟩
    ⟨Rename the items to satisfy the SSA property 21b⟩
}
◇
```

Macro referenced in scrap 6a.

2.1 Find the Items Which Are Live on Entry to Any Block

The first step in the conversion to SSA form is to find the items which are live on entry to any basic block. This step is not a standard procedure in the construction of SSA form; it is a modification which speeds up the computation and reduces the number of ϕ-nodes. We iterate through the blocks. Within each block we search for operations that reference an item defined outside that block. Each of these items is added to `liveSet`. When all blocks have been processed, `liveSet` contains all items which are live on entry to some basic block.

We use a variable **killedSet** to keep track of the items that have been killed (that is, defined) within a basic block. Clearly, once an item is defined within a basic block, any subsequent reference to that item will use that definition. Thus, only references to items not in **killedSet** are live across the basic block boundary.

⟨Put live items in `liveSet` 13a⟩ ≡
```
{
    SparseSet killedSet =
        SparseSet_Create(phiTempsArena, register_count + tag_count);
    Block *currBlock;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Put live items in liveSet\n");

    ForAllBlocks(currBlock)
    {
        Inst *currInst;
        ⟨Create and initialize the block_extension structure 9b⟩
        SparseSet_Clear(killedSet);

        Block_ForAllInsts(currInst, currBlock)
        {
            Operation **currOperPtr;
            Inst_ForAllOperations(currOperPtr, currInst)
            {
                Operation *currOper = *currOperPtr;

                ⟨Put the live registers used by currOper in liveSet 13b⟩
                ⟨Put the live tags used by currOper in liveSet 14a⟩

                ⟨Put the registers defined by currOper in killedSet 14b⟩
                ⟨Put the tags defined by currOper in killedSet 14c⟩
            }
        }
    }
}
◇
```
Macro referenced in scrap 12.

We iterate over all the used registers of **currOper**. If a register is referenced that is not in **killedSet**, we know that it is live across the basic block boundary.

⟨Put the live registers used by `currOper` in `liveSet` 13b⟩ ≡
```
{
    Unsigned_Int2 *currRegPtr;
    Operation_ForAllUses(currRegPtr, currOper)
    {
        Unsigned_Int currReg = *currRegPtr;

        if (!SparseSet_Member(killedSet, currReg))
            SparseSet_Insert(liveSet, currReg);
    }
}
◇
```
Macro referenced in scrap 13a.

The process for finding live tags is similar.

⟨Put the live tags used by currOper in liveSet 14a⟩ ≡

```
{
    Unsigned_Int2 *currTagPtr;
    Operation_ForAllRefTags(currTagPtr, currOper)
    {
        Unsigned_Int currTag = *currTagPtr;

        if (!SparseSet_Member(killedSet, currTag))
            SparseSet_Insert(liveSet, currTag);
    }
}
◇
```

Macro referenced in scrap 13a.

Once we have looked at all the referenced items, we update `killedSet` to include any registers defined by `currOper`.

⟨Put the registers defined by currOper in killedSet 14b⟩ ≡

```
{
    Unsigned_Int2 *currRegPtr;

    Operation_ForAllDefs(currRegPtr, currOper)
        SparseSet_Insert(killedSet, *currRegPtr);
}
◇
```

Macro referenced in scrap 13a.

The process for adding tags to `killedSet` is similar.

⟨Put the tags defined by currOper in killedSet 14c⟩ ≡

```
{
    Unsigned_Int2 *currTagPtr;

    Operation_ForAllDefTags(currTagPtr, currOper)
        SparseSet_Insert(killedSet, *currTagPtr);
}
◇
```

Macro referenced in scrap 13a.

2.2 Find Lists of Blocks Which Assign Each Item in `liveSet`

Once we have found the set of live items, we are ready to find the basic blocks which define each of them. Since we are iterating over all the defined items in the routine, this is a convenient time to count them. A global variable, `defCount`, is used as a counter. We will need to know the total number of definitions in the routine when we allocate sets and arrays indexed by the new SSA names.

⟨Global Variables 14d⟩ ≡

```
static Unsigned_Int defCount = 0;
◇
```

Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

The **assignments** array is built by looking at the defined items of all operations in the routine. Any defined item that is in **liveSet** causes **currrBlock** to be added to that item's assignment list.

⟨Find lists of blocks which assign each item in liveSet 15⟩ ≡

```
{
    Block *currBlock;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Find lists of blocks which assign each item in liveSet\n");

    ForAllBlocks(currBlock)
    {
        Inst *currInst;
        Block_ForAllInsts(currInst, currBlock)
        {
            Operation **currOperPtr;
            Inst_ForAllOperations(currOperPtr, currInst)
            {
                Operation *currOper = *currOperPtr;
                Unsigned_Int2 *currItemPtr;

                Operation_ForAllDefs(currItemPtr, currOper)
                {
                    Unsigned_Int currItem = *currItemPtr;

                    ⟨Add currBlock to assignments[currItem] 16a⟩
                    defCount++;
                }

                Operation_ForAllDefTags(currItemPtr, currOper)
                {
                    Unsigned_Int currItem = *currItemPtr;

                    ⟨Add currBlock to assignments[currItem] 16a⟩
                    defCount++;
                }
            }
        }
    }
}
◇
```

Macro referenced in scrap 12.

We only add `currBlock` to the list at `assignments[currItem]` if `currItem` is in `liveSet` and if `currBlock` is not already in the list. We conclude that `currBlock` is not in the list if the list is empty or `currBlock` is the first item.

⟨Add currBlock to assignments[currItem] 16a⟩ ≡
```
if (SparseSet_Member(liveSet, currItem) &&
    (!assignments[currItem] ||
     assignments[currItem]->block != currBlock))
{
    Block_List_Ptr blockListElt =
        Arena_GetMem(phiTempsArena, sizeof(Block_List_Node));

    blockListElt->block = currBlock;
    blockListElt->next = assignments[currItem];
    assignments[currItem] = blockListElt;
}
◇
```
Macro referenced in scrap 15.

2.3 Insert the Necessary ϕ-Nodes

Once the `liveSet` and the `assignments` array are computed, we are ready to put empty ϕ-nodes in the proper places.

2.3.1 The ϕ-node structure

The ϕ-nodes for each basic block are kept in a linked list. Each ϕ-node is represtented by a structure with four fields:

- `next` A pointer to the next ϕ-node in the list of ϕ-nodes for the basic block
- `oldName` The original item name
- `argCount` A counter specifying how many valid arguments have been place in the ϕ-node
- `args` An array (properly sized to contain the arguments of the ϕ-node)

Upon creation, the `argCount` field of a ϕ-node is set to zero. As arguments are added to the ϕ-node (see section 2.4.3.4), `argCount` is incremented.

⟨Type Declarations 16b⟩ ≡
```
typedef struct phiNode
{
    struct phiNode *next;
    Unsigned_Int2 oldName;
    Unsigned_Int2 argCount;
    Unsigned_Int2 args[1];      /* extensible */
} PhiNode, *PhiNodePtr;
◇
```
Macro defined by scraps 9a, 16b, 22a, 23a, 24b.
Macro referenced in scrap 4.

Notice that the `args` array is declared with a size of 1. However, ϕ-nodes have one argument for each predecessor of the basic block in which the node is located. Therefore, when we allocate a new ϕ-node the size is adjusted according to the number of arguments. We use a macro to perform the allocation. Notice that all ϕ-nodes are allocated in `lastArena`.

⟨Macros 17a⟩ ≡
```
#define AllocPhiNode(predCount) \
    Arena_GetMem(lastArena, \
                 sizeof(PhiNode) + sizeof(Unsigned_Int2)*(predCount-1))
```
◇

Macro defined by scraps 7ab, 17ade.
Macro referenced in scrap 4.

The list of ϕ-nodes for each basic block is contained in the `block_extension` structure.

⟨`block_extension` fields 17b⟩ ≡
```
struct phiNode *phiNodes;
```
◇

Macro defined by scraps 17b, 39c.
Macro referenced in scrap 9a.

⟨Initialize `block_extension` fields 17c⟩ ≡
```
extension->phiNodes = NULL;
```
◇

Macro defined by scraps 17c, 40a.
Macro referenced in scrap 9b.

Since there is a list of ϕ-nodes at each basic block, we define an iterator macro to step through them all. It takes two arguments:

1. A pointer to a ϕ-node structure (the iterator variable)
2. A pointer to the basic block

⟨Macros 17d⟩ ≡
```
#define Block_ForAllPhiNodes(phiNode, block) \
    for (phiNode = (block)->block_extension->phiNodes; \
         phiNode; \
         phiNode = phiNode->next)
```
◇

Macro defined by scraps 7ab, 17ade.
Macro referenced in scrap 4.

Another useful macro will iterate over all the arguments of a ϕ-node. It takes two arguments:

1. A pointer to an `Unsigned_Int2` (the iterator variable)
2. A pointer the the ϕ-node

⟨Macros 17e⟩ ≡
```
#define PhiNode_ForAllUses(usePtr, phiNode) \
    for (usePtr = &(phiNode)->args[0]; \
         usePtr < &(phiNode)->args[(phiNode)->argCount]; \
         usePtr++)
```
◇

Macro defined by scraps 7ab, 17ade.
Macro referenced in scrap 4.

Notice the use of the two ϕ-node iterator macros in this scrap.

⟨Print `block_extension` fields 18⟩ ≡

```
{
    PhiNodePtr node;
    Unsigned_Int2 *usePtr;

    Block_ForAllPhiNodes(node, block)
    {
        fprintf(stderr, "Phi node for item %d:\n   ",
                node->oldName);

        PhiNode_ForAllUses(usePtr, node)
            fprintf(stderr, "r%d ", *usePtr);

        fprintf(stderr, "\n");
    }
}
◇
```

Macro defined by scraps 18, 40b.
Macro referenced in scrap 9c.

2.3.2 Placing the ϕ-nodes

We use a worklist algorithm to place the ϕ-nodes for each item in `liveSet`. The `worklist` contains the basic blocks which need a ϕ-node for the current item in `liveSet`.

Initially, the **worklist** contains the basic blocks in the **assignments** array entry for the item chosen from **liveSet**. As blocks on the **worklist** are processed, blocks in the dominance frontier are added to the **worklist**. We must be careful of two things:

1. Each basic block can appear on the **worklist** at most once. The **everOnWorklist** set keeps track of which blocks have ever been on the **worklist** for the current item chosen from **liveSet**. Before placing a block on the **worklist**, we check that it is not in **everOnWorklist**. After placing a block on the **worklist**, we insert it into **everOnWorklist**.

2. Each basic block can have at most one ϕ node for each item in **liveSet**. The **hasAlready** set keeps track of which blocks already have a ϕ-node for the current item chosen from **liveSet**. Before placing a ϕ-node in a basic block, we check that the block is not in **hasAlready**. After placing a ϕ-node in a basic block, we insert the block into **hasAlready**.

⟨Insert the necessary ϕ-nodes 19a⟩ ≡

```
{
    Block_List_Ptr worklist;
    SparseSet everOnWorklist =
        SparseSet_Create(phiTempsArena, block_count + 1);
    SparseSet hasAlready =
        SparseSet_Create(phiTempsArena, block_count + 1);
    Unsigned_Int currItem;

    if (debug >= MINOR_PHASES)
        fprintf(stderr, "    Insert the necessary phi nodes\n");

    SparseSet_Forall(currItem, liveSet)
    {
        ⟨Set up the worklist for currItem 19b⟩

        ⟨Process the blocks in the worklist 20a⟩
    }
    Arena_Destroy(phiTempsArena);
}
◇
```

Macro referenced in scrap 12.

Initially, the **worklist** contains all the basic blocks in **assignments**[**currItem**]. We must also empty the **hasAlready** set and initialize the **everOnWorklist** set to contain those items in **assignments**[**currItem**].

⟨Set up the worklist for currItem 19b⟩ ≡

```
worklist = assignments[currItem];
SparseSet_Clear(hasAlready);
SparseSet_Clear(everOnWorklist);
{
    Block_List_Ptr currBlockEntry;
    for (currBlockEntry = worklist;
         currBlockEntry;
         currBlockEntry = currBlockEntry->next)
        SparseSet_Insert(everOnWorklist,
                         currBlockEntry->block->preorder_index);
}
◇
```

Macro referenced in scrap 19a.

For each basic block b on the `worklist`, we iterate over the blocks in the dominance frontier of b. For each basic block in the dominance frontier, we place a ϕ-node for `currItem` in that block and place that block on the `worklist`.

⟨Process the blocks in the `worklist` 20a⟩ ≡

```
{
    Block_List_Ptr currBlockEntry;

    for (currBlockEntry = worklist;
         currBlockEntry;
         currBlockEntry = currBlockEntry->next)
    {
        Block_List_Ptr domBlockList;
        Dominator_ForFrontier(domBlockList, currBlockEntry->block->dom_node)
        {
            Block *frontierBlock = domBlockList->block;

            ⟨Place φ-node for currItem in frontierBlock 20b⟩
            ⟨Place frontierBlock on the worklist 21a⟩
        }
    }
}
◇
```

Macro referenced in scrap 19a.

Placing a ϕ-node in a basic block requires a bit of housekeeping. First, we must make sure there is not already a ϕ-node for this item in the block. Then, we must allocate the ϕ-node, initialize the fields, and insert the ϕ-node into the block's `block_extension`. Finally, we record the fact that this block now has a ϕ-node and increment `defCount` to count the new definition.

⟨Place ϕ-node for `currItem` in `frontierBlock` 20b⟩ ≡

```
if (!SparseSet_Member(hasAlready, frontierBlock->preorder_index))
{
    PhiNodePtr phiNode = AllocPhiNode(frontierBlock->pred_count);
    Block_Extension extension = frontierBlock->block_extension;

    phiNode->oldName = currItem;
    phiNode->argCount = 0;

    phiNode->next = extension->phiNodes;
    extension->phiNodes = phiNode;

    SparseSet_Insert(hasAlready, frontierBlock->preorder_index);

    defCount++;
}
◇
```

Macro referenced in scrap 20a.

Before placing a basic block on the **worklist**, we must be sure it has never been on the **worklist**. After placing a block on the **worklist**, we must record the fact that it has been there. Notice that we insert the block after **currBlockEntry** so that it will be processed later.

⟨Place `frontierBlock` on the `worklist` 21a⟩ ≡
```
if (!SparseSet_Member(everOnWorklist, frontierBlock->preorder_index))
{
    Block_List_Ptr tmpBlockEntry =
        Arena_GetMem(phiTempsArena, sizeof(Block_List_Node));

    tmpBlockEntry->next = currBlockEntry->next;
    currBlockEntry->next = tmpBlockEntry;
    tmpBlockEntry->block = frontierBlock;

    SparseSet_Insert(everOnWorklist, frontierBlock->preorder_index);
}
◇
```
Macro referenced in scrap 20a.

2.4 Rename the Items to Satisfy the SSA Property

Having placed the appropriate ϕ-nodes, the next step in SSA building is to rename all of the items to satisfy the SSA requirement that only one definition reaches a use. This means that we have to give every definition in the routine a unique item number. The majority of the work in the step is done by the recursive function **Rename** which does the actual renaming of items and filling in of the ϕ-nodes. **Rename** takes as it argument a pointer to a basic block and performs a preorder walk over the dominator tree rooted at the argument. The renaming step is accomplished by initializing the related data structures and then invoking **Rename** with the **start_block** as its argument.

Since, we assume that all memory is defined before the routine is called, we must create pseudo-definitions within the SSA for all the memory tags (see section 2.4.2.1). Therefore, we add **tag_count** to the number of definitions in the routine.

⟨Rename the items to satisfy the SSA property 21b⟩ ≡
```
if (debug >= MINOR_PHASES)
    fprintf(stderr, "    Rename the items to satisfy the SSA property\n");

defCount += tag_count;

⟨Initialize the Rename data structures 22c, ... ⟩

Rename(start_block);

Arena_Destroy(SSAArena);
◇
```
Macro referenced in scrap 12.

2.4.1 Mapping SSA Names to Original Names

During the renaming phase, we must keep track of two pieces of information about each new name given to an item:

1. A mapping from the SSA name to the original name
2. The definition site for the item

This information is kept in a `NewName` structure which has five fields:

`defSite` A pointer to the definition site (either an operation or a ϕ-node)

`block` A pointer to the basic block in which this item is defined

`oldName` The item number as it appeared in the original routine (before conversion to SSA form)

`isPhiNode` A flag indicating if the definition site is a ϕ-node (otherwise, it is an operation)

`critical` A flag indicating if the item has been marked critical

⟨Type Declarations 22a⟩ ≡

```
typedef struct
{
    union
    {
        Operation *operation;
        PhiNode *phiNode;
    } defSite;
    Block *block;
    Unsigned_Int2 oldName;
    Boolean isPhiNode;
    Boolean critical;
} NewName, *NewNamePtr;
◇
```

Macro defined by scraps 9a, 16b, 22a, 23a, 24b.
Macro referenced in scrap 4.

We use an array of `NewName` structures indexed by the SSA names. This array is called `nameInfo`; it contains an element for each unique definition in the routine (`defCount`). The `nameInfo` array will be kept during the marking of critical code and the conversion of the routine back from SSA form, so it is a global variable which will be allocated from `lastArena`

⟨Global Variables 22b⟩ ≡

```
static NewName *nameInfo = NULL;
◇
```

Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

⟨Initialize the `Rename` data structures 22c⟩ ≡

```
nameInfo = Arena_GetMem(lastArena, (defCount + 1)*sizeof(NewName));
◇
```

Macro defined by scraps 22c, 24a, 25d.
Macro referenced in scrap 21b.

2.4.2 Mapping Original Names to SSA Names

When renaming items, we need a mapping from the original names to the new SSA names. At any point in the renaming process, we must be able to find the correct SSA name for each item in the original program. The correct SSA name is the one defined at the most recent assignment which can either be before this point in the same basic block or in another basic block. A block b may contain assignments to an item i in the original program in one of two ways:

1. b contains a ϕ-node with original item i
2. b contains an operation that defines i

When a reference to i appears in b, it should be replaced by the SSA name of the most recent definition in b if one exists. If no definition exists in b, then the reference to i should be replaced by the SSA name which was active at the end of the closest dominator of b (*i.e.,* the parent in the dominator tree). We perform the renaming in a preorder walk over the dominator tree to ensure that each parent is processed before its children. We must also ensure that the mapping of original names to SSA names at the beginning of each block is the mapping at the end of that block's parent. To accomplish this, the mapping for each item is stored in a stack data structure. The element at the top of the stack contains the most recent SSA name for the item. Whenever an item is defined for the first time within a basic block, a new name is pushed onto the stack for that item. Subsequent definitions within the same basic block will overwrite the first definition. The mappings are kept in a `NameList` structure which has two fields:

`next` Pointer to the next element lower on the stack

`newName` New (SSA) name for this item

⟨Type Declarations 23a⟩ ≡
```
typedef struct nameList
{
    struct nameList *next;
    Unsigned_Int newName;
} NameList, *NameListPtr;
◇
```
Macro defined by scraps 9a, 16b, 22a, 23a, 24b.
Macro referenced in scrap 4.

We use an array of `NameListPtr`'s indexed by the original item names called `nameStacks`. When the renaming is finished, we dispose of `nameStacks`.

⟨Global Variables 23b⟩ ≡
```
static NameListPtr *nameStacks = NULL;
◇
```
Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

Each name that is pushed onto `nameStacks` must be unique to satisfy the SSA property. We use a simple counter (`currName`) when generating new SSA names for items. The initial value for `currName` is 1 because the value 0 is used to mark the end of the list of tags in an operation. Each time a new SSA name is assigned, `currName` will be incremented.

⟨Global Variables 23c⟩ ≡
```
static Unsigned_Int currName = 1;
◇
```
Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

2.4.2.1 Generating Pseudo-Definitions for All Tags

The nameStacks array requires some special initialization. Note that the allocation clears all the entries. This is the correct value for the first register_count elements. However, we need to create pseudo-definitions for all the memory tags in the routine. The tags are located in the last tag_count elements of the array. We push a new name onto the stack for each of these elements.

Notice that we do not fill in all the fields of the nameInfo entry. This is possible because when the critical flag is set to TRUE none of the other fields are referenced.

⟨Initialize the Rename data structures 24a⟩ ≡

```
{
    Unsigned_Int numStacks = register_count + tag_count;
    Unsigned_Int oldName;

    nameStacks = Arena_GetMemClear(SSAArena,
                                   numStacks*sizeof(NameListPtr));
    for (oldName = register_count; oldName < numStacks; oldName++)
    {
        Unsigned_Int newName;
        NameListPtr newNameList =
            Arena_GetMem(SSAArena, sizeof(NameList));

        newNameList->next = nameStacks[oldName];
        nameStacks[oldName] = newNameList;
        newName = newNameList->newName = currName++;

        nameInfo[newName].oldName = oldName;
        nameInfo[newName].critical = TRUE;
    }
}
◇
```

Macro defined by scraps 22c, 24a, 25d.
Macro referenced in scrap 21b.

2.4.2.2 Saving and Restoring the State of nameStacks

Since the renaming step processes the basic blocks in preorder, we must save the state of nameStacks before processing each basic block and restore the state after processing the basic block's children. In this fashion, each basic block will begin the renaming process with the mapping that was active at the end of its parent in the dominator tree.

To facilitate the restoring of nameStacks at the completion of each basic block, we use a sizeable array containing the items which were defined in that basic block. For this, we use a StateArray sturcture. The structure has two fields:

size The number of items in the array

items Array containing the items defined in the block

⟨Type Declarations 24b⟩ ≡

```
typedef struct
{
    Unsigned_Int size;
    Unsigned_Int2 *items;
} StateArray;
◇
```

Macro defined by scraps 9a, 16b, 22a, 23a, 24b.
Macro referenced in scrap 4.

The array containing the items pushed during the current basic block is called `statesPushed`.

⟨Declare local variable `statesPushed` 25a⟩ ≡

```
StateArray statesPushed;
```

◇

Macro referenced in scrap 27.

Upon completion of a basic block, we step through the `statesPushed` array and pop the corresponding `nameStacks` element. Notice that we don't explicitly free the elements being popped. We rely on the functions `Arena_Mark` and `Arena_Release` for this (see section 2.4.3).

⟨Pop the items in `statesPushed` from `nameStacks` 25b⟩ ≡

```
{
    Unsigned_Int2 *item = statesPushed.items;
    Unsigned_Int2 *max_item = item + statesPushed.size;

    while (item < max_item)
    {
        Unsigned_Int index = *item++;
        nameStacks[index] = nameStacks[index]->next;
    }
}
```

◇

Macro referenced in scrap 27.

For efficiency, we want to ensure that each item in `nameStacks` gets pushed no more than once per basic block. A `SparseSet` called `newNameInserted` is used to keep track of which items were defined in that basic block. The set has a universe size equal to the number of items in the original program (`register_count` + `tag_count`).

Notice that `newNameInserted` is a global variable. This way, it can be created only once (before the initial invocation of `Rename`) and cleared at the beginning of each basic block.

⟨Global Variables 25c⟩ ≡

```
static SparseSet newNameInserted;
```

◇

Macro defined by scraps 6b, 8b, 14d, 22b, 23bc, 25c.
Macro referenced in scrap 4.

⟨Initialize the `Rename` data structures 25d⟩ ≡

```
newNameInserted = SparseSet_Create(SSAArena, register_count + tag_count);
```

◇

Macro defined by scraps 22c, 24a, 25d.
Macro referenced in scrap 21b.

2.4.2.3 Generating SSA Names

Updating the `nameStacks` array for each definition in the routine requires a bit of housekeeping. Therefore, we define a scrap which will be included whenever a new SSA name is created. This scrap requires the variables `newName`, `oldName`, and `newNameInserted` to be defined in the enclosing scope.

A new SSA name (`newName`) will be pushed onto `nameStacks[oldName]` only if this is the first definition for `oldName` in the basic block. If this is the case, `oldName` will be added to the `newNameInserted` set. Otherwise, the element at the top of `nameStacks[oldName]` will be overwritten.

⟨Assign `newName` the SSA name for `oldName` 26a⟩ ≡

```
if (!SparseSet_Member(newNameInserted, oldName))
{
    NameListPtr newNameList =
        Arena_GetMem(SSAArena, sizeof(NameList));

    newNameList->next = nameStacks[oldName];
    nameStacks[oldName] = newNameList;

    SparseSet_Insert(newNameInserted, oldName);
}

nameStacks[oldName]->newName = currName;
newName = currName++;
◇
```

Macro referenced in scraps 28a, 31c, 32a.

To save the state of the `nameStacks` array, we put the elements of `newNameInserted` into the `statesPushed` array.

⟨Build the `statesPushed` array from the elements of `newNameInserted` 26b⟩ ≡

```
statesPushed.size = SparseSet_Size(newNameInserted);
if (statesPushed.size)
{
    Unsigned_Int2 *item = statesPushed.items =
        Arena_GetMem(SSAArena, statesPushed.size*sizeof(Unsigned_Int2));
    Unsigned_Int name;
    SparseSet_Forall(name, newNameInserted)
        *item++ = name;
}
◇
```

Macro referenced in scrap 27.

2.4.3 Rename the Items in a Basic Block

Once the `Rename` data structures are initialized, we are ready to perform the renaming of the items in the routine. This step is accomplished by the `Rename` function. `Rename` takes as it argument a pointer to a basic block and performs a preorder walk over the dominator tree rooted at the argument. Therefore, `start_block` should be passed to the initial invocation of `Rename`.

For each basic block, there are six steps:

1. Generate new names for the items assigned by ϕ-nodes in the block.
2. For each operation in the block, replace the referenced items by their new SSA names and create new SSA names for each defined item.
3. For each successor of the block, add one argument to each ϕ-node in the successor. The argument contains the current name for the ϕ-node's original item.
4. Build the `statesPushed` array from the elements of `newNameInserted`. The `statesPushed` array provides a simpler and more compact copy of `newNameInserted` which is necessary because it will be overwritten during the next step.
5. Recur on each of the blocks children in the dominator tree.
6. Restore the original state of the `nameStacks` array by popping the names generated in this basic block.

Since the `nameStacks` array is popped at the end of each invocation of `Rename`, we use the `Arena_Mark` and `Arena_Release` routines to deallocate the memory which is no longer needed.

⟨Functions 27⟩ ≡
```
static Void Rename(Block *block)
{
    ⟨Declare local variable statesPushed 25a⟩

    SparseSet_Clear(newNameInserted);

    Arena_Mark(SSAArena);

    ⟨Make new names for items defined by φ-nodes 28a⟩
    ⟨For each operation, rename the uses and make new names for the definitions 28b⟩
    ⟨Add an argument to the φ-nodes in each successor 32b⟩
    ⟨Build the statesPushed array from the elements of newNameInserted 26b⟩
    ⟨Recur on the children of block 33⟩
    ⟨Pop the items in statesPushed from nameStacks 25b⟩

    Arena_Release(SSAArena);
}
◇
```
Macro defined by scraps 9c, 27, 40c.
Macro referenced in scrap 4.

2.4.3.1 Make New Names for Items Defined by ϕ-Nodes

The renaming of ϕ-nodes in a block is a simple matter of iterating through the list of nodes, assigning a new name to each ϕ-node assignment, and filling the fields in the `nameInfo` array entry.

⟨Make new names for items defined by ϕ-nodes 28a⟩ ≡

```
{
    PhiNodePtr phiNode;
    Block_ForAllPhiNodes(phiNode, block)
    {
        Unsigned_Int oldName = phiNode->oldName;
        Unsigned_Int newName;

        ⟨Assign newName the SSA name for oldName 26a⟩

        nameInfo[newName].oldName = oldName;
        nameInfo[newName].critical = FALSE;
        nameInfo[newName].defSite.phiNode = phiNode;
        nameInfo[newName].isPhiNode = TRUE;
        nameInfo[newName].block = block;
    }
}
◇
```

Macro referenced in scrap 27.

2.4.3.2 For Each Operation, Rename the Uses and Make New Names for the Definitions

For each operation in the block, we replace all uses with their new SSA names and generate new SSA names for the definitions.

⟨For each operation, rename the uses and make new names for the definitions 28b⟩ ≡

```
{
    Inst *currInstr;
    Operation **currOperPtr;
    Block_ForAllInsts(currInstr, block)
        Inst_ForAllOperations(currOperPtr, currInstr)
        {
            Operation *currOper = *currOperPtr;

            ⟨Replace currOper's used registers with new names 29a⟩
            ⟨Replace currOper's used tags with new names 31b⟩

            ⟨Generate new names for currOper's defined registers 31c⟩
            ⟨Generate new names for currOper's defined tags 32a⟩
        }
}
◇
```

Macro referenced in scrap 27.

Replacing the used register names is a accomplished by looking up the new name in the `nameStacks` array. However, we must be careful to check if the `nameStacks` entry is `NULL` which indicates an uninitialized registers.

⟨Replace `currOper`'s used registers with new names 29a⟩ ≡

```
{
    Unsigned_Int2 *currRegPtr;
    Operation_ForAllUses(currRegPtr, currOper)
    {
        Unsigned_Int currReg = *currRegPtr;
        NameListPtr recentName = nameStacks[currReg];

        if (recentName)
            *currRegPtr = recentName->newName;
        else
        {
            ⟨Handle uninitialized register 29b⟩
            break;
        }
    }
}
◇
```

Macro referenced in scrap 28b.

2.4.3.3 Handle an Operation Containing an Uninitialized Register

Whenever an uninitialized register is encountered, we must delete the operation. If the debug level is high enough, we will also print a warning message.

⟨Handle uninitialized register 29b⟩ ≡

```
{
    Opcode_Names opcode = currOper->opcode;

    if (debug >= UNINITIALIZED_REG)
        fprintf(stderr,
                "Operation uses an undefined register: %s %d\n",
                opcode_specs[opcode].opcode, currReg);

    ⟨Delete currOper 30a⟩
}
◇
```

Macro referenced in scrap 29a.

When deleting the operation which uses an uninitialized register, one of four things can happen:

- Branches and `JMPr`'s are turned into `JMPl`'s.
- Returns use the frame pointer argument in place of the uninitialized register.
- `JMPl`'s and calls are left alone.
- All other types of operations are turned into `NOP`'s.

Notice that all the operations that require special handling are flow of control operations. Therefore, they must appear as the first operation in the last instruction in a basic block. Since our basic blocks are stored as doubly-linked lists, we have an easy way to check if `currOper` fits this criterion.

⟨Delete `currOper` 30a⟩ ≡
```
if (currOper == block->inst->prev_inst->operations[0])
{
    if ((opcode_specs[opcode].details & CONDITIONAL) || (opcode == JMPr))
        ⟨Replace currOper with a JMPl 30b⟩
    else if (opcode != JMPl)         /*  opcode must be a RETURN  */
        ⟨Copy the frame pointer argument into the return value 30c⟩
}
else if (!(opcode_specs[opcode].details & CALL))
    ⟨Throw currOper away 31a⟩
◇
```
Macro referenced in scrap 29b.

Replace the branch with a `JMPl` to the block's first successor. Notice that we do not remove the other edges from the control-flow graph because they will never be referenced.

⟨Replace `currOper` with a `JMPl` 30b⟩ ≡
```
{
    currOper->opcode = JMPl;
    currOper->constants = 1;
    currOper->referenced = 1;
    currOper->defined = 1;
    currOper->arguments[0] = block->succ->succ->labels->label;
}
◇
```
Macro referenced in scrap 30a.

We assume that the frame pointer argument to the `RTN` is valid, and the return value is not valid. For this reason, we copy the frame pointer argument into the return value.

⟨Copy the frame pointer argument into the return value 30c⟩ ≡
```
currOper->arguments[1] = currOper->arguments[0];
◇
```
Macro referenced in scrap 30a.

To throw away an operation, we must not only change its opcode field to `NOP`, we must also set the `constants`, `referenced`, and `defined` fields to zero. This is to make sure the iterator macros work properly if they are applied to this operation in the future. We also set the `critical` field of the operation to `FALSE` just in case the `Block_Init` had marked it critical.

⟨Throw `currOper` away 31a⟩ ≡

```
{
    currOper->opcode = NOP;
    currOper->constants = 0;
    currOper->referenced = 0;
    currOper->defined = 0;
    currOper->critical = FALSE;
}
◇
```

Macro referenced in scrap 30a.

Notice that there is no check for uninitialized memory tags because we inserted the pseudo-definitions (see section 2.4.2.1) for all tags.

⟨Replace `currOper`'s used tags with new names 31b⟩ ≡

```
{
    Unsigned_Int2 *currTagPtr;
    Operation_ForAllRefTags(currTagPtr, currOper)
        *currTagPtr = nameStacks[*currTagPtr]->newName;
}
◇
```

Macro referenced in scrap 28b.

The process for generating new SSA names for the defined registers is a simple matter of finding a new name and filling in the `nameInfo` array entry.

⟨Generate new names for `currOper`'s defined registers 31c⟩ ≡

```
{
    Unsigned_Int2 *currRegPtr;
    Operation_ForAllDefs(currRegPtr, currOper)
    {
        Unsigned_Int oldName = *currRegPtr;
        Unsigned_Int newName;

        ⟨Assign newName the SSA name for oldName 26a⟩

        nameInfo[newName].oldName = oldName;
        nameInfo[newName].critical = FALSE;
        nameInfo[newName].defSite.operation = currOper;
        nameInfo[newName].isPhiNode = FALSE;
        nameInfo[newName].block = block;
    }
}
◇
```

Macro referenced in scrap 28b.

The process for generating new names for the defined tags is similar.

⟨Generate new names for currOper's defined tags 32a⟩ ≡
```
{
    Unsigned_Int2 *currTagPtr;
    Operation_ForAllDefTags(currTagPtr, currOper)
    {
        Unsigned_Int oldName = *currTagPtr;
        Unsigned_Int newName;

        ⟨Assign newName the SSA name for oldName 26a⟩

        nameInfo[newName].oldName = oldName;
        nameInfo[newName].critical = FALSE;
        nameInfo[newName].defSite.operation = currOper;
        nameInfo[newName].isPhiNode = FALSE;
        nameInfo[newName].block = block;
    }
}
◇
```
Macro referenced in scrap 28b.

2.4.3.4 Add an Argument to the ϕ-nodes in Each Successor

After we have processed all the operations in a basic block, we are ready to update the ϕ-nodes at the beginning of each of the successor blocks. We must add an argument to each ϕ-node in the successor block containing the most recent SSA name for the ϕ-node's item. Recall that the **PhiNode** structure contains an **argCount** field that indicates the current number of valid arguments to the ϕ-node. The **args** field has valid arguments in elements zero through **argCount** - 1. Therefore, to add an argument to a ϕ-node, we put the new SSA name in **args**[**argCount**] and increment **argCount**. If no SSA name exists for an item, that simply means that it is not defined along this path, and no argument is added to the ϕ-node.

⟨Add an argument to the ϕ-nodes in each successor 32b⟩ ≡
```
{
    Edge *currEdge;
    Block_ForAllSuccs(currEdge, block)
    {
        PhiNodePtr phiNode;
        Block_ForAllPhiNodes(phiNode, currEdge->succ)
        {
            NameListPtr recentName = nameStacks[phiNode->oldName];

            if (recentName)
            {
                phiNode->args[phiNode->argCount] = recentName->newName;
                phiNode->argCount++;
            }
        }
    }
}
◇
```
Macro referenced in scrap 27.

2.4.3.5 Recur on the Children of the Block

Since we calculated the dominance information while placing the ϕ-nodes, we can simply use a macro to iterate over all the children in the dominator tree.

$\langle$Recur on the children of `block` 33$\rangle \equiv$

```
{
    Block_List_Node *childBlock;
    Dominator_ForChildren(childBlock, block->dom_node)
        Rename(childBlock->block);
}
```

$\diamond$

Macro referenced in scrap 27.

Chapter 3

Find and Mark the Critical Operations

This is the heart of dead code elimination. We a use a `worklist` to keep track of the items yet to have their definitions traced. The `worklist` is implemented as a `SparseSet` with universe size equal to the number of items in the routine. Since the routine is in SSA form, this is also the number of definitions. The definitions are numbered from one to `defCount`, so we need `defCount` + 1 elements in the set. The `Block_Init` function marks certain operations as critical. The `worklist` is initialized with the items used by these operations. Then, we repeatedly remove an item from the `worklist` and mark the items used to define the item until the `worklist` is empty.

Also, when a basic block is determined to contain a critical operation, any branch operation upon which that block is control dependent is marked and added to the `worklist`.

⟨Find and mark critical operations 34⟩ ≡

```
{
    SparseSet worklist =
        SparseSet_Create(lastArena, defCount + 1);

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Find and mark critical operations\n");

    start_block->block_extension->useful = TRUE;
    end_block->block_extension->useful = TRUE;

    ⟨Initialize the worklist 35a⟩
    ⟨Process the worklist 36b⟩
}
◇
```

Macro referenced in scrap 6a.

3.1 Initialize the `worklist`

We initialize the `worklist` by iterating through the blocks, and through each operation in each block. When we hit an operation which `Block_Init` has marked critical, we place each item referenced in the critical operation on the `worklist`.

At the end of each block, we check if the block contained any critical instructions. If so, we call **MarkBlockUseful** for the block. This function will mark critical any branch operations upon which this block is control dependent (see section 3.3).

```
⟨Initialize the worklist 35a⟩ ≡
    {
        Block *currBlock;

        if (debug >= MINOR_PHASES)
            fprintf(stderr, "    Initialize the worklist\n");

        ForAllBlocks(currBlock)
        {
            Boolean criticalOpInBlock = FALSE;
            Inst *currInst;

            Block_ForAllInsts(currInst, currBlock)
            {
                Operation **currOperPtr;
                Inst_ForAllOperations(currOperPtr, currInst)
                {
                    Operation *currOper = *currOperPtr;
                    if (currOper->critical)
                    {
                        criticalOpInBlock = TRUE;
                        ⟨Put currOper's used items on the worklist 35b⟩
                    }
                }
            }

            if (criticalOpInBlock)
                MarkBlockUseful(currBlock, worklist);
        }
    }
    ◇
```

Macro referenced in scrap 34.

```
⟨Put currOper's used items on the worklist 35b⟩ ≡
    {
        Unsigned_Int2 *usePtr;
        Operation_ForAllUses(usePtr, currOper)
            ⟨Add *usePtr to the worklist 36a⟩

        Operation_ForAllRefTags(usePtr, currOper)
            ⟨Add *usePtr to the worklist 36a⟩
    }
    ◇
```

Macro referenced in scrap 35a.

Before adding an item to the **worklist**, we make sure it has never been there by checking the **critical** field of the **nameInfo** array entry.

⟨Add *usePtr to the worklist 36a⟩ ≡
```
{
    Unsigned_Int use = *usePtr;

    if (!nameInfo[use].critical)
    {
        nameInfo[use].critical = TRUE;
        SparseSet_Insert(worklist, use);
    }
}
```
◇

Macro referenced in scraps 35b, 36c, 37bc, 39a.

3.2 Process the worklist

This phase removes items from the **worklist** one at a time. The definition site of the item is found using the use-def chain in **nameInfo**. Each item used at the definition site is added to the **worklist** (if it has never been there).

This process may add more items to the **worklist**, but each item can be on the **worklist** at most once. Therefore, the **worklist** will eventually empty out, and the algorithm will terminate. At this point, all useful statements have been marked critical.

⟨Process the worklist 36b⟩ ≡
```
if (debug >= MINOR_PHASES)
    fprintf(stderr, "    Process the worklist\n");

while (SparseSet_Size(worklist))
{
    Unsigned_Int currItem = SparseSet_ChooseMember(worklist);
    SparseSet_Delete(worklist, currItem);

    if (nameInfo[currItem].isPhiNode)
        ⟨Mark the φ-node defining currItem critical 36c⟩
    else
        ⟨Mark the operation defining currItem critical 37a⟩
}
```
◇

Macro referenced in scrap 34.

When an item is defined in a ϕ-node, we simply add all the inputs of the ϕ-node to the **worklist**.

⟨Mark the ϕ-node defining currItem critical 36c⟩ ≡
```
{
    PhiNodePtr currPhiNode = nameInfo[currItem].defSite.phiNode;
    Unsigned_Int2 *usePtr;

    PhiNode_ForAllUses(usePtr, currPhiNode)
        ⟨Add *usePtr to the worklist 36a⟩
}
```
◇

Macro referenced in scrap 36b.

When an operation defining **currItem** is marked critical, each item used at the definition site is placed on the **worklist**, and **MarkBlockCritical** is called for the block containing the definition.

⟨Mark the operation defining `currItem` critical 37a⟩ ≡

```
{
    Operation *currOper = nameInfo[currItem].defSite.operation;

    currOper->critical = TRUE;

    ⟨Put currOper's used registers onto the worklist 37b⟩
    ⟨Put currOper's used tags onto the worklist 37c⟩

    MarkBlockUseful(nameInfo[currItem].block, worklist);
}
◇
```

Macro referenced in scrap 36b.

⟨Put `currOper`'s used registers onto the `worklist` 37b⟩ ≡

```
{
    Unsigned_Int2 *usePtr;
    Operation_ForAllUses(usePtr, currOper)
        ⟨Add *usePtr to the worklist 36a⟩
}
◇
```

Macro referenced in scraps 37a, 40d.

Handling the memory tags used by an operation is not as simple as handling the registers. Special attention must be paid to store operations because of aliasing. Otherwise, we place all the referenced tags onto the **worklist**.

⟨Put `currOper`'s used tags onto the `worklist` 37c⟩ ≡

```
if (opcode_specs[currOper->opcode].details & STORE)
    ⟨Trace the tag defined by currOper 39a⟩
else
{
    Unsigned_Int2 *usePtr;
    Operation_ForAllRefTags(usePtr, currOper)
        ⟨Add *usePtr to the worklist 36a⟩
}
◇
```

Macro referenced in scrap 37a.

We use special code to handle a store operation that is determined to be critical. The defined and used tags are kept in lists in which position is important. The positions are numbered starting with zero. The tag in position zero is the one that is actually written by the store. Other tags in the list represent possible aliases. The `Block_Init` function initially creates identical lists for defined and used tags. During the conversion to SSA, these get renumbered to unique values. So, the tag in position i of the used tag list is the SSA name for the memory location before the operation executes. Also, the tag in position i of the defined tag list is the SSA name for the memory location after the operation executes.

First, consider what happens during scalar stores. If `currItem` is actually defined by this operation (*i.e.,* it is in position zero), we don't need to look for any more definitions of `currItem` because `currOper` is definitely the defining operation. However, if `currItem` is one of the possible aliases for this operation (*i.e.,* it is in a position other than zero), we must add the used tag in the same position to the `worklist`. This is because aliasing is a MAY relationship. In other words, this store may affect the memory location, and it may not.

Consider the following example. Suppose A and B may be aliased:

	FORTRAN	Refenenced Tags	Defined Tags
1)	A = . . .	A_0, B_0	A_1, B_1
2)	B = . . .	B_1, A_1	B_2, A_2
	⋮		
3)	. . . = B	B_2	
4)	. . . = A	A_2	

The tag B_2 is defined on line 2 in position zero. Therefore, this store must define B_2, and no tags will be added to the `worklist`. The tag A_2 is defined on line 2 in position one. Therefore, this store may define A_2, and tag A_1 is added to the `worklist`.

Next, consider what happens during array stores. Since we are not keeping track of array subscripts, we don't know if this store affects the array in a subscript of interest or not. Therefore, we must add the used tag in the same position to the `worklist` regardless of the position. One can think of all the elements of an array as possibly being aliased to each other.

Consider the following example. Suppose A and B may be aliased:

	FORTRAN	Referenced Tags	Defined Tags
1)	A(i) = . . .	A_0, B_0	A_1, B_1
2)	B(j) = . . .	B_1, A_1	B_2, A_2
	⋮		
3)	. . . = B(k)	B_2	
4)	. . . = A(l)	A_2	

The tag B_2 is defined on line 2 in position zero. Therefore, tag B_1 is added to the `worklist`. The tag B_1 is defined on line 1 in position one. Therefore, tag B_0 is added to the `worklist`. The tag A_2 is defined on line 2 in position one. Therefore, tag A_1 is added to the `worklist`. The tag A_1 is defined on line 1 in position zero. Therefore, tag A_0 is added to the `worklist`.

Notice the use of the `SCALAR` bit in the `details` field of the operation. If `currOper` is not a scalar store or the defined tag is not in position zero, then we add the referenced tag in position `defTagPos` to the `worklist`.

⟨Trace the tag defined by `currOper` 39a⟩ ≡
```
{
    Unsigned_Int defTagPos = 0;
    ⟨Assign defTagPos to be the position of currItem 39b⟩

    if (!(opcode_specs[currOper->opcode].details & SCALAR) || defTagPos)
    {
        Unsigned_Int2 *usePtr =
            &currOper->arguments[currOper->defined + defTagPos];

        ⟨Add *usePtr to the worklist 36a⟩
    }
}
◇
```
Macro referenced in scrap 37c.

We must find the position of `currItem` in the defined-tag list for this operation. To do this, we step through the list looking for the desired tag value. Note that the positions begin with 0 and that `defTagPos` is only incremented after the test for the tag fails.

Remember that we did not overwrite the tags in the defined list during the conversion to SSA form. Therefore, we must look up the old tag value for `currItem` in the `nameInfo` array and search the list for that tag.

⟨Assign `defTagPos` to be the position of `currItem` 39b⟩ ≡
```
{
    Unsigned_Int searchTag = nameInfo[currItem].oldName;
    Unsigned_Int2 *currTagPtr;

    Operation_ForAllDefTags(currTagPtr, currOper)
    {
        if (*currTagPtr == searchTag)
            break;

        defTagPos++;
    }
}
◇
```
Macro referenced in scrap 39a.

3.3 Mark the Useful Blocks

During the marking of critical operations, we also keep track of which basic blocks contain critical operations. It is possible that some basic blocks contain no critical operations. If this is the case, we can eliminate some of the flow-of-control operations from the routine. Whenever a block b is determined to contain useful operations, all branches upon which b is control dependent are marked critical and added to the `worklist`. Once we have completed marking all critical operations, any unmarked conditional branch can be turned into a jump.

We use a field in the `block_extension` structure to keep track of which blocks have already been marked useful.

⟨`block_extension` fields 39c⟩ ≡
```
Boolean useful;
◇
```
Macro defined by scraps 17b, 39c.
Macro referenced in scrap 9a.

⟨Initialize `block_extension` fields 40a⟩ ≡

```
extension->useful = FALSE;
◇
```

Macro defined by scraps 17c, 40a.
Macro referenced in scrap 9b.

⟨Print `block_extension` fields 40b⟩ ≡

```
fprintf(stderr, "Useful: %d\n", block_extension->useful);
◇
```

Macro defined by scraps 18, 40b.
Macro referenced in scrap 9c.

The `MarkBlockUseful` takes as its arguments a `block` and a `worklist`. If the `block` hasn't been marked useful, `MarkBlockUseful` sets the `useful` field in the `block_extension` structure and marks critical all branches and blocks which control its argument `block`.

⟨Functions 40c⟩ ≡

```
static Void MarkBlockUseful(Block *block, SparseSet worklist)
{
    if (!block->block_extension->useful)
    {
        block->block_extension->useful = TRUE;

        ⟨Mark all branches on which block is control dependent 40d⟩
    }
}
◇
```

Macro defined by scraps 9c, 27, 40c.
Macro referenced in scrap 4.

⟨Mark all branches on which `block` is control dependent 40d⟩ ≡

```
{
    Block_List_Ptr controlDepPredList;
    Dominator_ForFrontier(controlDepPredList, block->post_dom_node)
    {
        Block *controlDepPred = controlDepPredList->block;
        Operation *currOper = controlDepPred->inst->prev_inst->operations[0];

        if (!currOper->critical)
        {
            currOper->critical = TRUE;

            ⟨Put currOper's used registers onto the worklist 37b⟩
            MarkBlockUseful(controlDepPred, worklist);
        }
    }
}
◇
```

Macro referenced in scrap 40c.

Chapter 4

Convert Out of SSA Form

The routine must now be converted out of SSA form. All of the necessary information is available in the `nameInfo` data structure. We perform also another important function, that of folding up conditional branches which were not marked critical. A dead branch is replaced by a jump to the closest useful post-dominator of the branch.

We also mark all jumps-to-label critical. These were not marked critical before because to do so would have been to mark all blocks ending in `JMPl` operations critical. This would have inhibited the elimination of dead conditionals.

⟨Convert the routine out of SSA form 41⟩ ≡
```
{
    Block *currBlock;

    if (debug >= MAJOR_PHASES)
        fprintf(stderr, "Convert the routine out of SSA form\n");

    ForAllBlocks(currBlock)
    {
        ⟨If currBlock has a dead conditional branch, retarget it 42a⟩
        ⟨Restore the original names to the items in currBlock 43⟩
    }
}
◇
```
Macro referenced in scrap 6a.

4.1 Retarget Dead Conditional Branches

The process of finding and marking critical operations ignored all jumps and marked only those branches upon which some useful block was control dependent. Therefore, at this point we must mark all jumps critical and change any unmarked branches into jumps. Notice that we may create blocks whose only operation is a jump. A later optimization pass will remove such blocks.

The only possible place for a branch is the first operation in the last instruction in a basic block. Since our basic blocks are stored as doubly-linked lists, we have an easy way to find the last instruction in the block. If we find a dead conditional branch, we find a new target for the operation and change it to a `JMPl`.

⟨If `currBlock` has a dead conditional branch, retarget it 42a⟩ ≡
```
{
    Operation *branch = currBlock->inst->prev_inst->operations[0];

    if (!branch->critical)
    {
        Opcode_Names opcode = branch->opcode;

        branch->critical = TRUE;

        if ((opcode_specs[opcode].details & CONDITIONAL) || (opcode == JMPr))
        {
            Block *target;

            ⟨Find new target for branch 42b⟩

            branch->opcode = JMPl;
            branch->constants = 1;
            branch->referenced = 1;
            branch->defined = 1;
            branch->arguments[0] = target->labels->label;
        }
    }
}
◇
```
Macro referenced in scrap 41.

There are two major steps for finding a new target for a branch:

1. Search up the postdominator tree for the first useful block. We know this search terminates, because `end_block`, which postdominates everything, has been marked useful.
2. If the closest useful block is the `end_block`, then we conclude that we are in an infinite loop. If this is the case, we choose `target` to be any one of `currBlock`'s successors other than `end_block`.

⟨Find new `target` for `branch` 42b⟩ ≡
```
target = currBlock->post_dom_node->parent;
while (!target->block_extension->useful)
    target = target->post_dom_node->parent;

if (target == end_block)
{
    if (end_block == currBlock->succ->succ)
        target = currBlock->succ->next_succ->succ;
    else
        target = currBlock->succ->succ;
}
◇
```
Macro referenced in scrap 42a.

4.2 Restore the Original Names

Restoring the original names is a matter of looking up the `oldName` in the `nameInfo` array entry.

⟨Restore the original names to the items in `currBlock` 43⟩ ≡

```
{
    Inst *currInst;
    Block_ForAllInsts(currInst, currBlock)
    {
        Operation **currOperPtr;
        Inst_ForAllOperations(currOperPtr, currInst)
        {
            Operation *currOper = *currOperPtr;

            if (currOper->critical)
            {
                Unsigned_Int2 *currItemPtr;

                Operation_ForAllUses(currItemPtr, currOper)
                    *currItemPtr = nameInfo[*currItemPtr].oldName;

                Operation_ForAllRefTags(currItemPtr, currOper)
                    *currItemPtr = nameInfo[*currItemPtr].oldName;
            }
        }
    }
}
◇
```

Macro referenced in scrap 41.

Appendix A

Memory Use Diagram

This diagram shows the major data structures allocated during dead code elimination and their lifetimes across the different phases. From such a graph, we can easily see a logical set of arenas in which to keep the objects, such that we minimize memory use by grouping objects with similar lifetimes into the same arena.

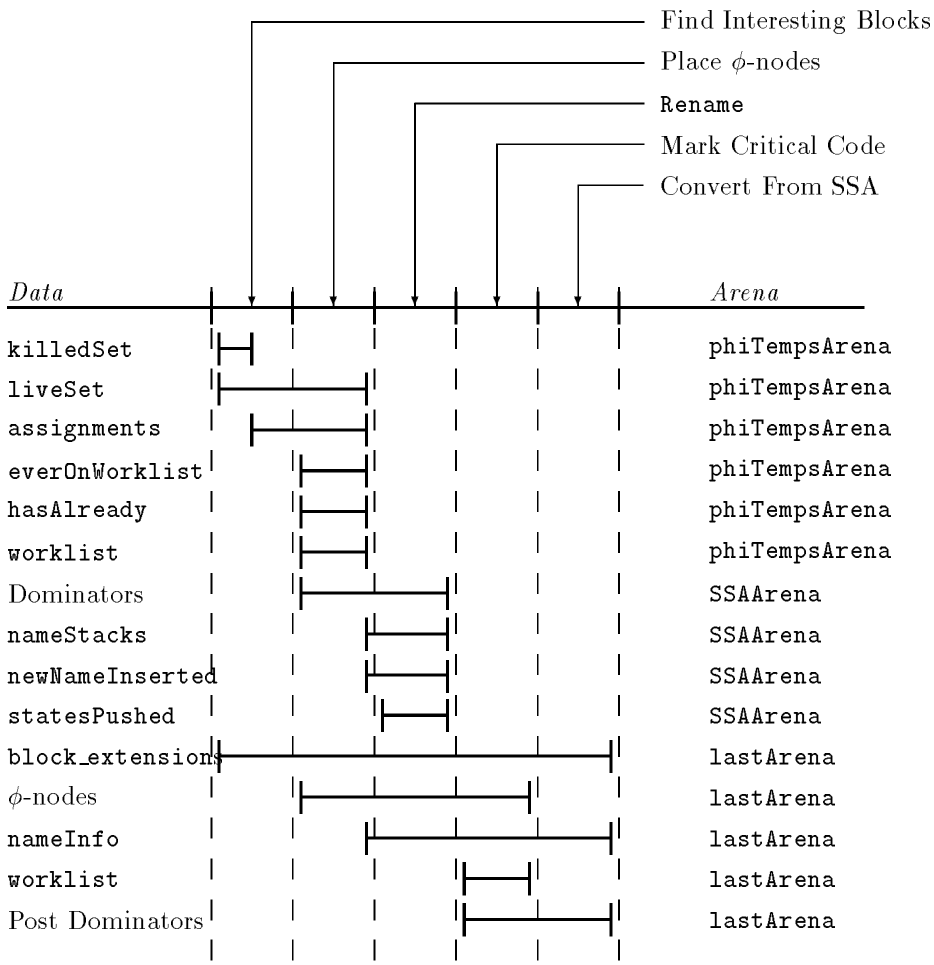

Figure A.1: Memory Use Diagram for **dead**.

Appendix B

Indices

B.1 Index of File Names

B.2 Index of Macro Names

B.3 Index of Identifiers

B.4 Function Prototypes

⟨Prototypes 49a⟩ ≡

```
static Void ExtensionPrinter(Block_Ptr block);
```

◇

Macro defined by scraps 49abc.
Macro referenced in scrap 4.

⟨Prototypes 49b⟩ ≡

```
static Void Rename(Block *block);
```

◇

Macro defined by scraps 49abc.
Macro referenced in scrap 4.

⟨Prototypes 49c⟩ ≡

```
static Void MarkBlockUseful(Block *block, SparseSet worklist);
```

◇

Macro defined by scraps 49abc.
Macro referenced in scrap 4.

Bibliography

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[2] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.