Static Single Assignment Construction

Preston Briggs
Tim Harvey
Taylor Simpson

Version 1.0
July 19, 1995

Contents

List of Figures

Chapter 1

Introduction

The purpose of this document is to present a generalized implementation of a control-flow graph (CFG) static single assignment (SSA) form converter.

This document and the resultant code are intended to be part of the shared code library used in the Massively Scalar Compiler Project. The goal is to provide a central implementation of the SSA conversion algorithm described in Cytron *et al.*, which will handle all of the various "flavors" of SSA form [3].

1.1 SSA form

SSA form is a way of representing a routine which facilitates data-flow analysis. Code is said to be in SSA form if only one definition of a memory tag or register[1] appears statically in the routine. With this invariant met, we can trivially build both use-def and def-use chains; that is, we can build links between the definitions and uses of resources. These chains are used extensively in many classical optimizations, including dead-code elimination, constant propagation, and global value numbering. This method is also being used both to develop new optimizations such as assertion insertion and to improve existing optimizations such as peephole optimization.

Converting the CFG to SSA form requires that we rewrite the original code by inventing and substituting new names for each definition and substituting these new names for subsequent uses of a resource. We will refer to the process of converting the CFG into SSA form as *building* SSA. Likewise, we will refer to the process of converting the code out of SSA form as *restoring*.

To build SSA form from a straight-line piece of code is trivial; each time a resource gets defined, we give it a new name which we then substitute into subsequent references. When the resource gets redefined, we begin substituting the next new name, and so forth. For example, consider the short scrap of ILOC in the left-hand column of Figure 1.1. Conversion to SSA form results in the code shown in the right-hand column.

Before	After
iLDI 10 => $r17$	iLDI 10 => $r0$
iADD $r17$ $r17$ => $r18$	iADD $r0$ $r0$ => $r1$
iADD $r17$ $r18$ => $r17$	iADD $r0$ $r1$ => $r2$
iADD $r17$ $r18$ => $r19$	iADD $r2$ $r1$ => $r3$

Figure 1.1: Simple code and its conversion to SSA form

The problem becomes more difficult in the presence of control flow. If a resource in the original routine is defined along two paths which converge, we have no unique definition that reaches a subsequent reference.

[1] From now on, we'll refer to tags and registers collectively as *resources*.

To solve this problem, we introduce into the CFG pseudo functions, called *ϕ-nodes*, which take as arguments the names of the resources flowing into the convergence, commonly called a *join point*, and define a single resource. Subsequent uses of that resource will be replaced with the result of the ϕ-node, preserving the SSA invariant that a resource can only be defined once in the routine.

To understand the impact of ϕ-nodes, consider the code shown in Figure 1.2. Two different definitions of $r18$ reach the use of it at the join point of the two paths. So which defines $r20$?

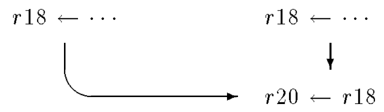


Figure 1.2: Two definitions reaching a use of a resource

If we insert a ϕ-node at the beginning of the join point and then convert the code into SSA form, we arrive at the CFG shown in Figure 1.3. We can see that our SSA invariant is kept; each use of a variable can be traced back to a single definition point.

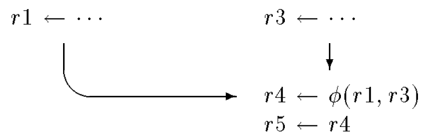


Figure 1.3: New code with SSA numbering and a ϕ-node inserted

A ϕ-node for a resource gets inserted at the join point of two distinct paths which define that resource. The set of these join points is found by examining the dominance-frontiers. This will be explained in greater detail in Chapter 2.

1.2 Flavors of SSA

Because SSA form is used for many purposes, this shared SSA module has to be flexible enough to perform a variety of analyses and build a multitude of specialized data structures.

To begin, there are two flavors of SSA form, *minimal* and *pruned*. They are distinguished by the number of ϕ-nodes inserted during building. In minimal form, ϕ-nodes are placed without regard to the liveness of the resource for which they are inserted. In other words, it is possible that a ϕ-node will be inserted for a resource which is not subsequently used. These extra ϕ-nodes do not detract from the quality of the analysis; they simply waste space.

Pruned form, however, relies on *liveness analysis* to ensure that no such dead ϕ-nodes are inserted. Since the pruned form relies on liveness analysis which the minimal form does not require, it may be slower to build. Certain applications, however, require this extra restriction on ϕ-node insertion to ensure that their analysis is accurate and the resulting code is as compact as possible.

We have developed a third flavor of SSA, which we have dubbed *semi-pruned*. Semi-pruned form inserts ϕ-nodes only for those resources which are live across block boundaries (*i.e.*, when the use of a resource occurs in a different block than the one in which the resource was defined). Thus, this method is a compromise which attempts to limit the number of dead ϕ-nodes which get inserted into the CFG while avoiding the cost of doing the pruned form's data-flow analysis.

Figure 1.4 shows the difference between the three flavors of SSA. In the original code, we define three variables, x, y, and z. Assume that x is only used in the same block in which it is defined, the variable y is used in a subsequent block prior to the join point of the two paths, and the variable z is used past the join point. The three graphs at the bottom of the figure compare the ϕ-nodes which the three flavors of SSA form insert. The leftmost graph shows the minimal SSA form. Here, ϕ-nodes get placed at the join point for all three variables. Clearly, the ϕ-nodes for the variables x and y are unnecessary; these variables are never used again. The middle graph shows the semi-pruned SSA form. In this graph, we do not place a ϕ-node for x at the join point, because it is not alive across any basic-block boundary. However, we will insert a ϕ-node for y, despite the fact that it is never used again, because it is alive across *some* block boundary, and that is the limit of the analysis used. In the rightmost graph, we see that building the pruned SSA form results in insertion of a ϕ-node for z only. For pruned form, we performed the complete analysis necessary to show that both x and y were never used again, and thus refrained from inserting ϕ-nodes for those variables.

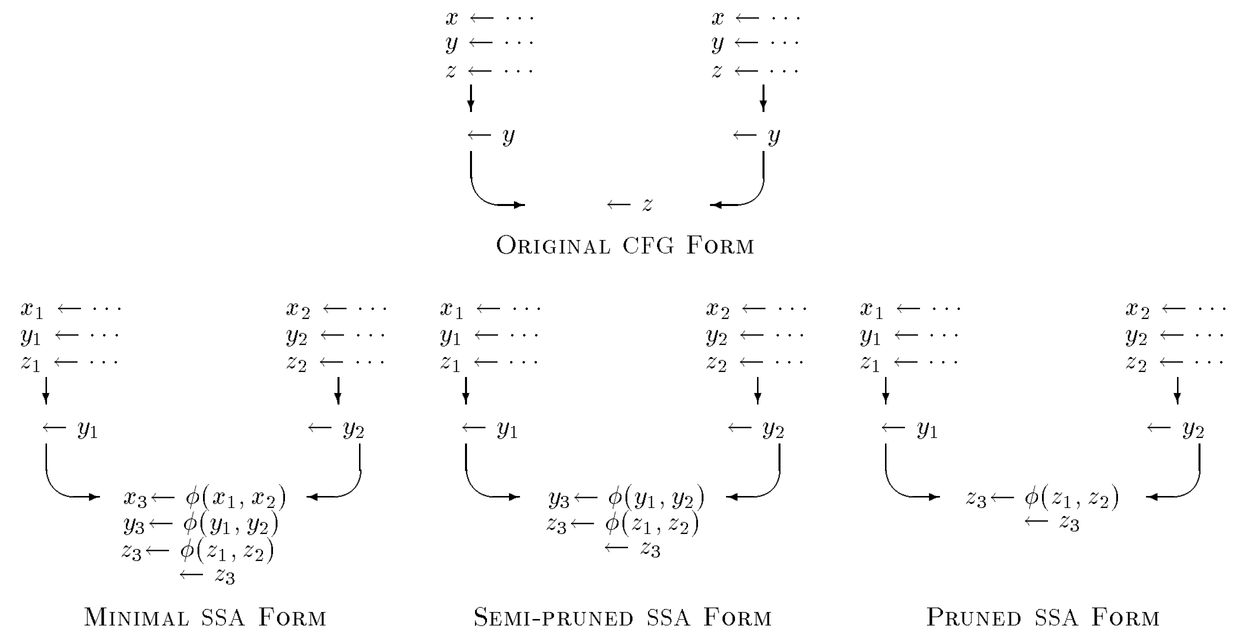


Figure 1.4: A comparison of the three flavors of SSA form

Each of the above three flavors has different uses. Cytron, *et al.* give an example where *global value numbering* can benefit from the extra, dead ϕ-nodes which minimal form contains. However, these dead ϕ-nodes constitute a waste of both time and space for optimizations like *constant propagation* and *dead code elimination*, which use the semi-pruned form. Other optimizations, such as the peephole optimizer and the register allocator, depend on the compactness of the pruned form and so must bear the extra time needed to perform the required liveness analysis.

Different applications require different data structures as well as different flavors of SSA. Those which perform forward problems will need def-use chains, while those which perform backwards problems will need use-def chains. Of course, there are applications which will require both. This code will be responsible for producing the data structures to represent both use-def and def-use chains. We will explain the structure and use of these data structures subsequently.

Finally, we must address the issue of returning the code to its original form. When we create new names for each definition of a resource, we maintain a table mapping each resource's new name back to its original name. On output, we replace each occurrence with its old name.

This will not always work, however. Optimization passes which move (or delete) instructions can alter the code so much that the original names no longer apply. For example, Figure 1.5 shows three sections of code. The first shows the original code. Notice that we remove the copy operation when we convert to SSA form, as shown in the second section of code.[2] Notice that this code still produces the same result as the original. The third column shows the code after we restore the resource names to their original names. Obviously, the variable z gets the updated value for x, instead of the first value.

$x \leftarrow 1$	$x_0 \leftarrow 1$	$x \leftarrow 1$
$y \leftarrow x$		
$x \leftarrow 2$	$x_1 \leftarrow 2$	$x \leftarrow 2$
$z \leftarrow y$	$z \leftarrow x_0$	$z \leftarrow x$
ORIGINAL CODE	SSA FORM	INCORRECTLY RESTORED

Figure 1.5: An example of code which cannot be restored to its original numbering

Thus, some applications will alter the code in such a way that we will not be able to return the resources to their original names. In this case, we will need a mechanism for converting ϕ-nodes into actual ILOC instructions. We replace a ϕ-node in some block B with a copy operation in each of B's predecessors. Since the meaning of a ϕ-node is a mapping of all of the incoming resources to a single resource, we can see that having a copy in each of the predecessor blocks (each copying that block's resource into our "ϕ-node resource") is equivalent.

Consider the example from Figure 1.3. In Figure 1.6, we show the same fragment of code with copies inserted for the ϕ-node in the top two blocks. We have made the ϕ-node obsolete by inserting these copies, allowing us to delete it without changing the meaning of the code.

$r1 \leftarrow \cdots$ $\quad$ $r3 \leftarrow \cdots$
$r4 \leftarrow r1$ $\quad$ $r4 \leftarrow r3$

$r4 \leftarrow \phi(r1, r3)$
$r5 \leftarrow r4$

Figure 1.6: Copy operations inserted to replace the ϕ-node

This has been a short overview of the different flavors of SSA that we build. In the next section, we will describe the main structure of the SSA module and provide a framework for the subsequent discussion.

1.3 Structure of the code

The code for building SSA will be part of the `shared` library code, so this document will produce both a header file and its associated body. The header file will declare all of the externally visible data structures, variables and entry points. The body of the module (the ".c" file) will contain the bodies of the code.

[2] When converting to SSA form, removal of copies can be done trivially: subsequent uses of the copy's destination are replaced by uses of the copy's source, effectively eliminating the copy operation, since its result is never used.

"SSA.h" 5a ≡

```
#ifndef SSA
#define SSA 1
#include <Shared.h>
⟨Externally-visible data structures 6a, ... ⟩
⟨Externally-visible variable declarations 6b, ... ⟩
⟨Flags for SSA_Build 12b, ... ⟩
⟨Externally-visible function prototypes 6d, ... ⟩
#endif
◇
```

"SSA.c" 5b ≡

```
#include <Shared.h>
#include "SSA.h"

⟨Externally-visible variable definitions 6c, ... ⟩
⟨Internal data structures 17d, ... ⟩
⟨Private variables 15b, ... ⟩
⟨Internal function prototypes 36a, ... ⟩
⟨Internal function bodies 37b, ... ⟩
⟨Externally-visible function bodies 15a, ... ⟩
◇
```

No single data structure could efficiently support all of the different flavors of SSA. The next section will explain the data structures which are visible to a user of this module and their use.

1.4 Externally-visible variables and data structures

In this section, we will look at the data structures and variables visible to the user.

1.4.1 The `Phi_Node` data structure and the `SSA_phi_nodes` array

We will introduce the data structures required to represent ϕ-nodes first, because we insert ϕ-nodes regardless of the flavor of SSA being built. Each ϕ-node will require the following fields:

`next` a pointer to other ϕ-nodes, since we are going to store them in a linked list

`old_name` a place to store the original (pre-SSA) name of the resource for which the ϕ-node is being inserted [3]

`new_name` a place to store the new SSA name which will get assigned to this ϕ-node

`def_type` an integer field to denote the type of the ϕ-node; this is used to determine the type of the operation to be inserted when we are replacing ϕ-nodes with copy operations

`useless` a boolean field to describe whether or not the ϕ-node has been found to be useless and thus has been removed (for more on this, see Section 2.6)

`parms` an array to hold the parameters to the ϕ-node; since this array will vary in size, it will be the last field in the data structure so that it can be sized as needed

[3] Of course, we store old names in the `SSA_name_map` array, but we will need this field before the `SSA_name_map` array gets built, so we must keep the information here as well.

⟨Externally-visible data structures 6a⟩ ≡

```
typedef struct phi_node Phi_Node;
struct phi_node
{
    Phi_Node *next;
    Unsigned_Int2 old_name;
    Unsigned_Int2 new_name;
    Unsigned_Int2 def_type; /* values defined in Shared.h */
    Boolean useless;
    Unsigned_Int2 parms[1];
};
◇
```

Macro defined by scraps 6a, 10c, 11a.
Macro referenced in scrap 5a.

Since each ϕ-node is associated with a particular block, we will allocate an array of pointers to ϕ-nodes with size equal to the number of blocks in the CFG. The array can be accessed by using a block's `preorder_index` field. If there are no ϕ-nodes associated with a block, that entry in the array will contain a `NULL` pointer.

⟨Externally-visible variable *declarations* 6b⟩ ≡

```
extern Phi_Node **SSA_phi_nodes;
◇
```

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 6c⟩ ≡

```
Phi_Node **SSA_phi_nodes;
◇
```

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

We will define a macro to enable easy traversal of the ϕ-nodes in a block. We have designed this "ForAll" macro to imitate the appearance of C's `for`-loops, while hiding the implementation details of iterating through the list of ϕ-nodes.

⟨Externally-visible function prototypes 6d⟩ ≡

```
/* Block_ForAllPhiNodes(Phi_Node *phi_node, Block *block) */
#define Block_ForAllPhiNodes(phi_node, block)                    \
        for (phi_node=SSA_phi_nodes[(block)->preorder_index];    \
             phi_node;                                           \
             phi_node=phi_node->next)
◇
```

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

As an example of the use of this macro, imagine that we want to clear all of the `new_name` fields for the ϕ-nodes in a block called `current_block`. To do this, we would use the macro as shown in Figure 1.7.

1.4.2 The `SSA_edge_maps` array

The parameters to a ϕ-node are stored in its `parms` array. Each entry in this array corresponds to a value flowing into the ϕ-node along some CFG edge. The problem, however, is to develop a mapping of edges to ϕ-node parameters so that we can tell *which* value flows along *which* edge. If we assign an ordering to the edges flowing into a block, then we can use this same ordering in the `parms` array for each ϕ-node in the block.

Thus, we will establish this ordering using an array of `Edge` pointers showing the edges flowing into a block. For each block b we create an array of `Edge` pointers corresponding to the predecessor edges of b. The ordering of the edges in this array dictates the ordering of the values in the `parms` array of each ϕ-node

```
Phi_Node *phi_runner;

Block_ForAllPhiNodes(phi_runner, current_block)
{
    phi_runner->new_name = 0;
}
```

Figure 1.7: Example use of the `Block_ForAllPhiNodes` macro

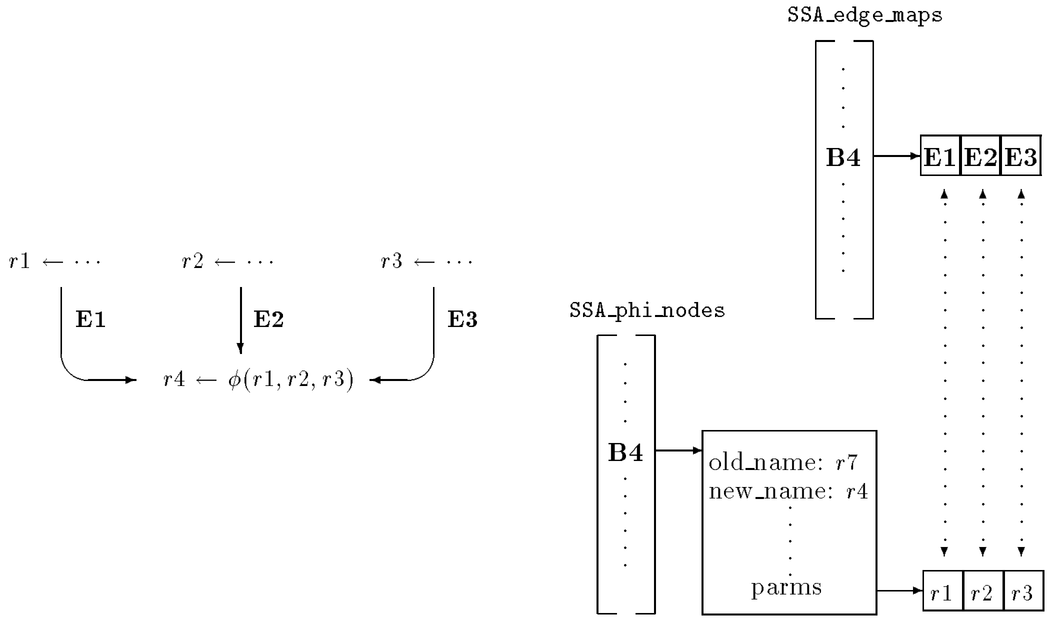


Figure 1.8: Example of the configuration of the `parms` and `SSA_edge_maps` arrays for the ϕ-node in block `B4`

in b. A value flowing along a particular edge e will be found in the ϕ-node's `parms` array at e's index in the edge-mapping array.

This will give us the mapping of ϕ-node parameters to incoming edges. For example, if we want to know the resource name flowing along some edge into a ϕ-node, we will find the index of the edge in our mapping table and use that same number as the index into the `parms` array for the resource's ϕ-node. The example in Figure 1.8 shows the design of these data structures. Note that "$B4$" is the name of the block containing the assignment to $r4$, and we use this name as the index into the various arrays.[4]

⟨Externally-visible variable *declarations* 7⟩ ≡

```
extern Edge ***SSA_edge_maps;
```

◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

[4] Of course, in the implementation itself, $B4$'s `preorder_index` field would be used.

⟨Externally-visible variable *definitions* 8a⟩ ≡

```
Edge ***SSA_edge_maps;
◇
```

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.3 The `Phi_Node_ForAllParms` and `Phi_Node_ForAllUpdatedParms` macros

Often, users will wish to iterate over the parameters to a ϕ-node. The following macro performs this function and is used in the same fashion as the `Block_ForAllPhiNodes` macro, above.

⟨Externally-visible function prototypes 8b⟩ ≡

```
/* Phi_Node_ForAllParms(Unsigned_Int2 *runner, Phi_Node *phi_node) */
#define Phi_Node_ForAllParms(runner, phi_node)                                   \
        for (runner = &((phi_node)->parms[0]);                                   \
             runner < (phi_node)->parms + SSA_block_map[(phi_node)->new_name]->pred_count; \
             runner++)
◇
```

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

Some applications, such as global value numbering, will change the `new_name` field of a ϕ-node during their normal execution. If the application changes the names of ϕ-nodes, the above macro will no longer work, because the index used for `SSA_block_map` will be incorrect and, thus, the value for `pred_count` will be incorrect. To handle this case, we will introduce a second macro, `Phi_Node_ForAllUpdatedParms`. This macro should be used whenever it is possible that a ϕ-node's `new_name` field has been changed.

⟨Externally-visible function prototypes 8c⟩ ≡

```
/* Phi_Node_ForAllUpdatedParms(Unsigned_Int2 *runner,
                               Phi_Node *phi_node, Unsigned_Int parm_count) */
#define Phi_Node_ForAllUpdatedParms(runner, phi_node, parm_count)  \
        for (runner = &((phi_node)->parms[0]);                     \
             runner < (phi_node)->parms + parm_count ;             \
             runner++)
◇
```

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

1.4.4 The `SSA_def_count` scalar value

There are four other variables which are independent of the flavor of SSA we are building. The first of these, `SSA_def_count`, is the total number of definitions in the routine plus one.[5] Since each definition will have a unique name, this variable not only tells us how large to make some of our data structures, but it also tells us what the highest name created will be. Notice that we initialize `SSA_def_count` to one instead of zero. We reserve zero to tell us when values are undefined (*e.g.,* in a ϕ-node's `parms` array). No user should adjust the value of `SSA_def_count`, as it is used during restoring. For more on this, see Section 4.1, on page 79.

There are instances (which we elaborate in Chapter 3) when this variable does not actually represent the highest name in the code, but, rather, the highest name that *might* be created. This variable is intended to be used for two purposes:

1. to create data structures guaranteed to be large enough to hold all of the resources
2. to use as a guide to creating new resource names which will not conflict with those created by this module.

A problem can arise in the first case, when the user naively iterates across an array expected to have all of its elements filled in with meaningful values. To ensure that the code works correctly, each of these arrays

[5] For reasons of clarity, we prefer to use post-incrementing on the `SSA_def_count` variable. Thus, its value will always be *one more* than the highest variable name.

must use some special value to signify unused elements, and the user must check for that value. All of the arrays in this module use either 0 or `NULL` to indicate an unused resource. Likewise, any user who creates an array based on this variable should ensure that the elements are initialized with a value which flags an unused resource name.

⟨Externally-visible variable *declarations* 9a⟩ ≡

```
extern Unsigned_Int SSA_def_count;
```

◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 9b⟩ ≡

```
Unsigned_Int SSA_def_count = 1;
```

◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.5 The `SSA_name_map` array

The second variable is used to map SSA names to their original names so that we can restore the original resource names.

⟨Externally-visible variable *declarations* 9c⟩ ≡

```
extern Unsigned_Int2 *SSA_name_map;
```

◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 9d⟩ ≡

```
Unsigned_Int2 *SSA_name_map;
```

◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.6 The `SSA_block_map` array

The third variable which does not depend on the flavor of SSA being built is the **`SSA_block_map`** array, which maps an SSA name to the block where it is defined. This will allow us to map an operation (or ϕ-node) back to the block which contains its definition.

⟨Externally-visible variable *declarations* 9e⟩ ≡

```
extern Block **SSA_block_map;
```

◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 9f⟩ ≡

```
Block **SSA_block_map;
```

◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.7 The `SSA_inst_map` array

The final variable which does not depend on the flavor of SSA being built is the **`SSA_inst_map`** array, which maps an SSA name to the instruction where it is defined. This will allow us to map an operation back to the instruction which contains it. If the resource was defined by a ϕ-node, its entry in this array will be `NULL`.

⟨Externally-visible variable *declarations* 10a⟩ ≡
```
extern Inst **SSA_inst_map;
```
◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 10b⟩ ≡
```
Inst **SSA_inst_map;
```
◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.8 The Chain data structure and SSA_use_def_chains array

Use-def chains are trivial to represent. We use an array of pointers, indexed by the SSA name. Given an SSA name, the user need only index into this array to find the definition point. This simplicity, however, is somewhat complicated by the necessities of type checking. That is, since a resource can be defined by either an `Operation` or a `Phi_Node`, we need to create a `union` data structure which can hold a pointer to either kind of structure.

⟨Externally-visible data structures 10c⟩ ≡
```
typedef struct chain_type Chain;
struct chain_type
{
    union
    {
        Operation *operation;
        Phi_Node *phi_node;
    } op_pointer;
    Boolean is_phi_node;
};
```
◇

Macro defined by scraps 6a, 10c, 11a.
Macro referenced in scrap 5a.

⟨Externally-visible variable *declarations* 10d⟩ ≡
```
extern Chain *SSA_use_def_chains;
```
◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 10e⟩ ≡
```
Chain *SSA_use_def_chains;
```
◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.9 The Chains_List data structure and SSA_def_use_chains array

For def-use chains, we will use an array similar to `SSA_use_def_chains`, but this array will store linked lists of `Chains`, since a single definition can reach many uses. We also include a field representing the resource's position in either the `arguments` array in the `Operation` data structure, or the `parms` array of the `Phi_Node` data structure. This will allow the user to follow a use to its operation or ϕ-node and to find the exact index into the respective data structure.

⟨Externally-visible data structures 11a⟩ ≡

```
typedef struct chains_list_type Chains_List;
struct chains_list_type
{
    Chain chain;
    Unsigned_Int position;
    Chains_List *next;
};
◇
```

Macro defined by scraps 6a, 10c, 11a.
Macro referenced in scrap 5a.

⟨Externally-visible variable *declarations* 11b⟩ ≡

```
extern Chains_List **SSA_def_use_chains;
◇
```

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 11c⟩ ≡

```
Chains_List **SSA_def_use_chains;
◇
```

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

1.4.10 The `Chain_ForAllUses` macro

We will also provide a macro for users to iterate easily through a particular definition's def-use chains.

⟨Externally-visible function prototypes 11d⟩ ≡

```
 /* Chain_ForAllUses(Chains_List *runner, Unsigned_Int resource_name) */
#define Chain_ForAllUses(runner, resource_name) \
    for (runner = SSA_def_use_chains[(resource_name)]; runner; runner = runner->next)
◇
```

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

It is important to note that none of the arenas which are used to store these externally-visible arrays is public. All allocated variables which are alive when `SSA_Build` completes are stored in our private arena called `permanent_memory_bank`. However, there should be no reason to deallocate any of these structures until we restore the original CFG. Thus, the call to `SSA_Restore` will not only restore the code, but will also destroy the `permanent_memory_bank` arena.

The next section will discuss the three entry points used to build SSA form and restore the code back into its original form.

1.5 External entry points

This module provides three entry points to manipulate a routine in SSA form. The first of these is `SSA_Build`, which converts the CFG into SSA form. The second entry point is `SSA_Restore`, which is the compliment to `SSA_Build`; it converts the CFG out of SSA form. Finally, there is a third entry point, `SSA_Replace_Phi_Nodes`, which replaces ϕ-nodes with copy operations.

Normal use of this module will include the initial call to `SSA_Build` to convert the CFG to SSA form. The user then has two choices: if his optimization will require the replacement of ϕ-nodes before his analysis begins, he should follow the call to `SSA_Build` with a call to `SSA_Replace_Phi_Nodes`. Otherwise, he should perform his analysis. When the pass has completed its work, the user should then call `SSA_Replace_Phi_Nodes` (if required, and if he has not already called it) and, finally, `SSA_Restore`. As we have said, the call to `SSA_Replace_Phi_Nodes` is optional and dependent on the user's needs, but `SSA_Restore` should *always* be called before the user attempts to print out the CFG.[6]

⟨Externally-visible function prototypes 12a⟩ ≡

```
extern Void SSA_Build(Unsigned_Int flags);
extern Void SSA_Replace_Phi_Nodes(Void);
extern Void SSA_Restore(Void);
◇
```

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

We will elaborate each of the three entry points in subsequent chapters. However, we will take some time in the next section to explain the flags sent in the bit-vector parameter of `SSA_Build`.

1.5.1 Controlling the flavor of SSA

We modify the flavor of SSA and specify the data structures to be built by using a bit-vector passed into `SSA_Build`. The values for each of the flags in the bit-vector will be defined in the header file for this module. The user need only combine them with the *bitwise-or* operator on the call to `SSA_Build`. For example, a user of this module who wanted to build minimal form and use-def chains would call `SSA_build` as follows:

```
SSA_Build(SSA_MINIMAL | SSA_BUILD_USE_DEF_CHAINS);
```

The first three flags control the flavor of SSA built, that is, minimal, semi-pruned, or pruned form. Exactly one of these must be specified.

⟨Flags for `SSA_Build` 12b⟩ ≡

```
#define SSA_MINIMAL      1<<0
#define SSA_SEMI_PRUNED  1<<1
#define SSA_PRUNED       1<<2
◇
```

Macro defined by scraps 12bcd, 13ab.
Macro referenced in scrap 5a.

The next two flags control whether or not def-use and/or use-def chains get built.

⟨Flags for `SSA_Build` 12c⟩ ≡

```
#define SSA_BUILD_DEF_USE_CHAINS   1<<3
#define SSA_BUILD_USE_DEF_CHAINS   1<<4
◇
```

Macro defined by scraps 12bcd, 13ab.
Macro referenced in scrap 5a.

The next flag controls whether copy operations are folded during the construction of SSA form. During the building SSA form phase, we can simply replace uses of the copy's destination with uses of the copy's source, effectively eliminating the copy operation.

Once copy operations are folded, the register names cannot be restored (by `SSA_Restore`) to their original names. The user of this code must, therefore, remember to call `SSA_Replace_Phi_Nodes` before calling `SSA_Restore`.

⟨Flags for `SSA_Build` 12d⟩ ≡

```
#define SSA_FOLD_COPIES  1<<5
◇
```

Macro defined by scraps 12bcd, 13ab.
Macro referenced in scrap 5a.

[6] The `SSA_Restore` routine understands the cases when it will not actually replace the register names; *i.e.,* when the user has folded copies and/or replaced ϕ-nodes. However, this call is still necessary, because our implementation of tags currently requires us to return them to their original names so that the parser can output them correctly. This is safe, because we use tags for informational purposes only – no amount of motion of tags will affect the result of the program.

The user also has the option of removing ϕ-nodes which serve no purpose. For more on this, see Section 2.6.

⟨Flags for SSA_Build 13a⟩ ≡

```
#define SSA_FOLD_USELESS_PHI_NODES  1<<6
```

◇

Macro defined by scraps 12bcd, 13ab.
Macro referenced in scrap 5a.

Finally, we have the ability to recognize a use of an uninitialized register while building SSA form. Each time one of these registers is found, we can print out a warning message stating where the use was in the routine and what `SSA_Build` did to correct the problem. The following flag controls the printing of these messages.

⟨Flags for SSA_Build 13b⟩ ≡

```
#define SSA_PRINT_WARNING_MESSAGES 1<<7
```

◇

Macro defined by scraps 12bcd, 13ab.
Macro referenced in scrap 5a.

Chapter 2

Building SSA Form

This chapter will present the code to convert the CFG into SSA form. This introduction is intended to provide a high-level description of the process used to build SSA form. There are two major steps in this process: inserting ϕ-nodes, and renumbering the resources in the CFG.

The process of inserting ϕ-nodes can be further broken down into separate steps.

1. Calculate the dominator tree and dominance frontiers
2. Compute liveness
3. Find the list of blocks which contain assignments to each resource
4. Insert ϕ-nodes, guided by the above analysis

The first step is to compute the dominator information. Dominance information guides the placement of ϕ-nodes, because it exposes all of the join points in the CFG. The work to build the dominance data structures is provided in our shared archive, via the entry point `Dominator_CalcDom` [2].

Inserting ϕ-nodes requires us to compute data-flow information (if the user set the `SSA_PRUNED` flag). Specifically, we need to compute the *liveness* of each resource on entry to each basic block in the routine. This analysis is done with a classical iterative approach. Note that the analyses of these first two steps are independent; therefore, the order in which they are performed can be arbitrary.

The third step in the process of inserting ϕ-nodes is to identify the blocks where each resource is defined. We perform an order-independent walk over the CFG to examine each instruction, requiring time linear in the number of operations in the code. We take advantage of this traversal to record *global* resources (unless we are building minimal SSA), those whose values are live on entry to some basic block. This information gives us an improvement in performance over the published algorithm [3].

At this point, we have a list of the resources and the blocks in which they are defined. These lists, combined with the liveness and dominator information, are used to determine the placement of ϕ-nodes for each resource. The placement of ϕ-nodes therefore has a worst-case complexity of $O(R \times N)$, where R is the number of resources and N is the number of blocks in the CFG.[1] Since not all of the fields of the ϕ-node are filled in yet, we will determine the remaining values in the next step.

With ϕ-nodes inserted into the graph, we perform a preorder walk over the dominator tree, renaming resources. This walk ensures that we always encounter the definition of a resource before any use of that resource. The actual renaming is simple; for each instruction, we rename the resources used by the instruction with their new names. Then, we assign new names to the instruction's defined resources by incrementing `SSA_def_count` for each one. The code outlining these steps is below.

[1] In the worst case, we will have to add a ϕ-node for every resource into every block.

⟨Externally-visible function bodies 15a⟩ ≡
```
Void SSA_Build(Unsigned_Int flags)
{
    ⟨Variables used to build the SSA form 17a, ... ⟩

    ⟨Parse flags and set private variables 16b, ... ⟩
    ⟨Double-check parameter values 16c⟩
    ⟨Set up memory arenas 15d⟩

    if (fold_copies)
        Dominator_CalcDom(permanent_memory_bank);
    else
        Dominator_CalcDom(dominance_memory_bank);
    ⟨Initialize SSA variables 17b, ... ⟩
    number_of_resources = register_count + tag_count;
    ⟨Perform liveness analysis, if necessary 20⟩
    ⟨Find assignments 26a⟩
    ⟨Place φ-nodes 30a⟩
    ⟨Rename resources 34⟩
    ⟨Fold useless φ-nodes, if necessary 50a⟩
    Arena_Destroy(dominance_memory_bank);
    SSA_CFG_renumbered = TRUE;
} /* SSA_Build */
◇
```
Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

As we will explain, we will reuse the code used to perform the liveness analysis both in **SSA_Build** and in **SSA_Replace_Phi_Nodes**. Because the namespace is much larger when the code is already in SSA form, we will declare the **number_of_resources** variable to guide the construction of the liveness analysis.

⟨Private variables 15b⟩ ≡
```
static Unsigned_Int number_of_resources = 0;
◇
```
Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

Consult the document explaining the shared code libraries for a full discussion of our use of arenas to handle memory allocation [1]. We will require three arenas to build the SSA form. For more, see the memory-use diagram in Appendix A, on page 81.

⟨Private variables 15c⟩ ≡
```
static Arena data_flow_memory_bank, dominance_memory_bank, permanent_memory_bank;
◇
```
Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Set up memory arenas 15d⟩ ≡
```
data_flow_memory_bank = Arena_Create();
dominance_memory_bank = Arena_Create();
permanent_memory_bank = Arena_Create();
◇
```
Macro referenced in scrap 15a.

The rest of this chapter will describe the code necessary to transform the CFG into SSA form.

2.1 Private data structures and variables

This section will describe the private data structures used to build SSA form. These data structures support the analyses to determine liveness and to control the insertion of ϕ-nodes. We will also discuss some of the variables which are shared between the procedures in this module.

The `SSA_Build` routine begins by storing into global variables the values of `flags`. These internal variables allow `SSA_Replace_Phi_Nodes` to use them without having to rely on the user to pass in the same values to both routines.

⟨Private variables 16a⟩ ≡
```
static Boolean build_minimal_form, build_semi_pruned_form, build_pruned_form,
               build_use_def_chains, build_def_use_chains,
               fold_copies, fold_useless_phi_nodes,
               complain_about_uninitialized_registers;
```
◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Parse `flags` and set private variables 16b⟩ ≡
```
build_minimal_form = (flags & SSA_MINIMAL) ? TRUE : FALSE;
build_semi_pruned_form = (flags & SSA_SEMI_PRUNED) ? TRUE : FALSE;
build_pruned_form = (flags & SSA_PRUNED) ? TRUE : FALSE;
```
◇

Macro defined by scraps 16bd.
Macro referenced in scrap 15a.

The following check ensures that exactly one of the flavors of SSA form has been requested. To do this, we first mask out all of the other flags. Since exactly one of these three flags must be set, we can simply check that the result after masking equals exactly one of the three values. If not, we print out an error message and quit.

⟨Double-check parameter values 16c⟩ ≡
```
{
    Unsigned_Int flavor = flags&(SSA_MINIMAL|SSA_SEMI_PRUNED|SSA_PRUNED);

    if (flavor != SSA_MINIMAL && flavor != SSA_SEMI_PRUNED && flavor != SSA_PRUNED)
    {
        fprintf(stderr, "Unknown type of SSA to build.  You should use exactly one of:\n");
        fprintf(stderr, "\tSSA_MINIMAL, SSA_SEMI_PRUNED, or SSA_PRUNED.\n");
        ABORT;
    }
}
```
◇

Macro referenced in scrap 15a.

The rest of the variables are straightforward and do not require any error checking.

⟨Parse `flags` and set private variables 16d⟩ ≡
```
build_def_use_chains = (flags & SSA_BUILD_DEF_USE_CHAINS) ? TRUE : FALSE;
build_use_def_chains = (flags & SSA_BUILD_USE_DEF_CHAINS) ? TRUE : FALSE;
fold_copies = (flags & SSA_FOLD_COPIES) ? TRUE : FALSE;
fold_useless_phi_nodes = (flags & SSA_FOLD_USELESS_PHI_NODES) ? TRUE : FALSE;
complain_about_uninitialized_registers = (flags & SSA_PRINT_WARNING_MESSAGES) ? TRUE : FALSE;
```
◇

Macro defined by scraps 16bd.
Macro referenced in scrap 15a.

Building the pruned SSA form requires that we avoid inserting dead ϕ-nodes, that is, ϕ-nodes which define a resource which is subsequently never used. To support this, we must determine which resources are live at the entrance of each basic block by using data-flow analysis. The results are stored in an array of `VectorSet`s, indexed by each block's `preorder_index` field. As a convenience, we add one to `block_count` when we allocate the memory for this array, since block numbers start at one instead of zero. This will be true of all of the arrays allocated using `block_count`.

⟨Variables used to build the SSA form 17a⟩ ≡
```
VectorSet *live_in = NULL;
```
◇

Macro defined by scraps 17ae, 18b.
Macro referenced in scrap 15a.

⟨Initialize SSA variables 17b⟩ ≡
```
if  (build_pruned_form)
    live_in = (VectorSet *)Arena_GetMem(data_flow_memory_bank,
                                        (block_count+1) * sizeof(VectorSet)) ;
```
◇

Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

The replacement of ϕ-nodes with copies in Chapter 3 will also rely on this liveness analysis, except that it will require *live-out* information – that is, the set of variables at the end of a block which are used in subsequent blocks. For this, we will define another array, `live_out`, which will hold this information. We make it a global array so that multiple routines can access it.

⟨Private variables 17c⟩ ≡
```
VectorSet *live_out = NULL;
```
◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

As we described in the introduction to this chapter, one of the steps in the algorithm to build SSA form consists of recording the blocks containing assignments to each resource. We will therefore create an array indexed by resource number. For each resource r, `assignments[`r`]` will contain the linked list of blocks containing assignments to r and the type of the resource in that block. By recording the type, we can infer the types of the ϕ-nodes which include a particular resource as a parameter.

The parsing routines in the `shared` library guarantee that tag numbers in the CFG are sequential and are numbered starting from `register_count`. Thus, these arrays can be accessed by simply using the resource number.

⟨Internal data structures 17d⟩ ≡
```
typedef struct assignment_node Assignment_Node;
struct assignment_node
{
    Unsigned_Int def_type; /* types defined in Shared.h */
    Block *block;
    Assignment_Node *next;
};
```
◇

Macro defined by scraps 17d, 32c, 59a, 69a.
Macro referenced in scrap 5b.

⟨Variables used to build the SSA form 17e⟩ ≡
```
Assignment_Node **assignments = NULL;
```
◇

Macro defined by scraps 17ae, 18b.
Macro referenced in scrap 15a.

⟨Initialize SSA variables 18a⟩ ≡

```
assignments =
        (Assignment_Node **)Arena_GetMemClear(data_flow_memory_bank,
                                (register_count+tag_count)*sizeof(Assignment_Node *));
```

◇

Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

While building the linked lists for the `assignments` array, we will also record *global resources*. Global resources are live across a basic block boundary (*i.e.*, the definition of a resource in one block reaches a use of that resource in another block). We implement the set of global resources using a `SparseSet`, since we need to support quick insertion of elements (avoiding duplicate entries) and efficient iteration over all the elements.

⟨Variables used to build the SSA form 18b⟩ ≡

```
SparseSet global_resources;
```

◇

Macro defined by scraps 17ae, 18b.
Macro referenced in scrap 15a.

⟨Initialize SSA variables 18c⟩ ≡

```
if (!build_minimal_form)
    global_resources = SparseSet_Create(data_flow_memory_bank, register_count+tag_count);
```

◇

Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

2.1.1 Handshaking with other `shared` code

One of the advantages of including the SSA code and data structures as part of the `shared` library is that other routines in this library can manipulate the CFG even though the code may be in SSA form. This is because they can access all of the data structures which are used to build and maintain the SSA form and make sense out of the graph.

In the past, all of the output routines, both the regular CFG printing routines (*e.g.*, `Block_Put_All`) and the debugging routines (*e.g.* `Block_Dump_All`), did not know how to print a graph in SSA form. Now, however, if `SSA_Build` sets a flag when it is called to tell the rest of the library routines that the SSA form has been built, the library routines can access the SSA data structures and "view" the CFG as they require.

We will also use this variable to limit the liveness analysis, discussed in the next section.

⟨Externally-visible variable *declarations* 18d⟩ ≡

```
extern Boolean SSA_CFG_renumbered;
```

◇

Macro defined by scraps 6b, 7, 9ace, 10ad, 11b, 18d.
Macro referenced in scrap 5a.

⟨Externally-visible variable *definitions* 18e⟩ ≡

```
Boolean SSA_CFG_renumbered = FALSE;
```

◇

Macro defined by scraps 6c, 8a, 9bdf, 10be, 11c, 18e.
Macro referenced in scrap 5b.

2.2 Finding live resources (when building *pruned* SSA)

Liveness is usually expressed in terms of the following data-flow equations:

$$\texttt{live\_in}_i \;=\; \texttt{local\_live}_i \cup (\texttt{live\_out}_i \cap \overline{\texttt{killed}_i}) \tag{2.1}$$

$$\texttt{live\_out}_i \;=\; \begin{cases} \emptyset, & \text{if } i \text{ is the exit block} \\ \displaystyle\bigcup_{j \in succ_i} \texttt{live\_in}_j, & \text{otherwise} \end{cases} \tag{2.2}$$

Where $\texttt{local\_live}_i$ is the set of exposed uses in block i, $\texttt{killed}_i$ is the set of resources defined in block i, and $\texttt{succ}_i$ is the set of successors of block i in the CFG.

These equations are somewhat complicated by the need to perform liveness analysis on the code after it has been converted to SSA form. Chapter 3 discusses the application of this. For now, it is sufficient to discuss the added complexity that ϕ-nodes introduce into the analysis.

A ϕ-node's parameters represent values which flow along a particular edge in the CFG. While the full set of a block b's ϕ-node parameters are in the `live_in` set of b, only those parameters which flow along an edge leading to a predecessor of b are in that predecessor's `live_out` set. When ϕ-nodes are taken into account, equation 2.2 becomes:

$$\texttt{live\_out}_i \;=\; \begin{cases} \emptyset, & \text{if } i \text{ is the exit block} \\ \texttt{phi\_node\_parms}_i \cup \left\{ \displaystyle\bigcup_{j \in succ_i} \texttt{live\_in}_j \right\}, & \text{otherwise} \end{cases} \tag{2.3}$$

Where $\texttt{phi\_nodes\_parms}_i$ is the set of resources which flow into i's successors' ϕ-nodes. This set is static, and if the code is not in SSA form, it is simply empty.

The resources defined by ϕ-nodes also get considered when calculating a block's `killed` set.

We solve these equations using a classical bit-vectored iterative analysis. However, we can minimize the number of bit-vectors we need. For example, when we are computing live-in information, the `live_out` set is only useful for a limited time, so we never allocate storage for it; instead, we compute it on the fly. Similarly, we do not need an explicit representation for `local_live`; we simply use `live_in` in its place. On the other hand, we must allocate an array of bit vectors to represent each block's `killed` set, as we will need its information across all of the iterations of our data-flow analysis.

We first define the `Arena` used to hold the sets needed for liveness analysis, and then define `killed`.

⟨Variables used to perform liveness analysis 19a⟩ ≡

```
Arena liveness_memory_bank = Arena_Create();
```

◇

Macro defined by scraps 19ab.
Macro referenced in scrap 20.

⟨Variables used to perform liveness analysis 19b⟩ ≡

```
VectorSet *killed = (VectorSet *)Arena_GetMem(liveness_memory_bank,
                                            sizeof(VectorSet) * (block_count + 1));
```

◇

Macro defined by scraps 19ab.
Macro referenced in scrap 20.

There are two stages to the computation of liveness information. In the first stage, we allocate and initialize the bit vectors for each basic block, building an initial approximation of `live_in` and computing `killed`. In the second stage, we perform an iterative analysis to propagate liveness information around the entire CFG.

⟨Perform liveness analysis, if necessary 20⟩ ≡
```
if (build_pruned_form)
{
    ⟨Variables used to perform liveness analysis 19a, ... ⟩

    ⟨Collect local liveness information for each block 21a⟩
    ⟨Perform iterative analysis to determine liveness 23b⟩
    Arena_Destroy(liveness_memory_bank);
}
◇
```
Macro referenced in scraps 15a, 68a.

2.2.1 Computing local liveness information

The first step in computing the local liveness information is allocating bit vectors to hold `live_in` and `killed` for each block. We then visit all of the operations in the block, computing their contribution to `live_in` and `killed`.

All of the sets in this analysis are large enough to hold the total number of registers and tags. Remember that the shared code numbers tags starting at `register_count`, so we can find elements of the set for both registers and tags without manipulation of the index.

Since the information in the `killed` sets is only used by each block's successors, there is no need to allocate a `killed` set for `end_block`.

We can tell if the CFG is already in SSA form using the global variable `SSA_CFG_renumbered`. If it is in SSA form, we need to take the ϕ-nodes into account when we compute each block's `killed` set.

⟨Collect local liveness information for each block 21a⟩ ≡

```
{
    Block *block;
    ForAllBlocks(block)
    {
        Inst *inst;
        Unsigned_Int block_num = block->preorder_index;

        VectorSet curr_live_out = NULL;
        VectorSet curr_live_in = live_in[block_num] =
              VectorSet_Create(data_flow_memory_bank, number_of_resources);

        if (SSA_CFG_renumbered)
        {
            live_out[block_num] = VectorSet_Create(permanent_memory_bank,
                                                   number_of_resources);
            curr_live_out = live_out[block_num];
        }

        if (block != end_block)
        {
            VectorSet curr_killed =
                        killed[block_num] = VectorSet_Create(liveness_memory_bank,
                                                             number_of_resources);
            if (SSA_CFG_renumbered)
                ⟨Collect φ-node defs into curr_killed 21b⟩

            Block_ForAllInsts(inst, block)
            {
                ⟨Find the contribution of inst to curr_live_in 22a⟩
                ⟨Find the contribution of inst to curr_killed 22b⟩
            }
        }
    }
    if (SSA_CFG_renumbered)
        ⟨Collect φ-node uses 23a⟩
}
◇
```

Macro referenced in scrap 20.

Computing the **killed** set for each block entails running through all of the instructions in the block. If the code is already in SSA form, we also have to iterate through the list of ϕ-nodes in the block.

⟨Collect ϕ-node defs into curr_killed 21b⟩ ≡

```
{
    Phi_Node *phi_node_runner;

    Block_ForAllPhiNodes(phi_node_runner, block)
        VectorSet_Insert(curr_killed, phi_node_runner->new_name);
}
◇
```

Macro referenced in scrap 21a.

In order to find the initial approximation of `live_in` for each block, we must examine all the resource references in each instruction (that is, all the uses of resources in all the operations comprising the instruction). If a resource is used before it is defined, we add it to `live_in`.

We add all of the referenced resources in an instruction to `live_in` *before we add any of the defined* *values to* **killed**. For example, consider the following line of ILOC code:[2]

```
ADDI 1 r1 => r2 | ADDI 1 r2 => r1
```

Since all operations within a single instruction are defined to execute in parallel, the reference of $r2$ in the second operation should cause it to be added to `live_in`, since we use its value before it gets redefined. However, if we examine the references and definitions in a single operation before moving on to the next operation in the same instruction, $r2$ would first be added to `killed`, precluding it from being inserted into `live_in`.

⟨Find the contribution of `inst` to `curr_live_in` 22a⟩ ≡

```
{
    Operation **op_runner;

    Inst_ForAllOperations(op_runner, inst)
    {
        Unsigned_Int2 *arg_runner;
        Operation *operation = *op_runner;

        Operation_ForAllUses(arg_runner, operation)
            if (!VectorSet_Member(curr_killed, *arg_runner))
                VectorSet_Insert(curr_live_in, *arg_runner);

        Operation_ForAllRefTags(arg_runner, operation)
            if (!VectorSet_Member(curr_killed, *arg_runner))
                VectorSet_Insert(curr_live_in, *arg_runner);
    }
}
```

◇

Macro referenced in scrap 21a.

Defining a resource erases its old value, or *kills* it. We examine all the operations in an instruction, adding all of the defined resources to `killed`.

⟨Find the contribution of `inst` to `curr_killed` 22b⟩ ≡

```
{
    Operation **op_runner;

    Inst_ForAllOperations(op_runner, inst)
    {
        Unsigned_Int2 *arg_runner;
        Operation *operation = *op_runner;

        Operation_ForAllDefs(arg_runner, operation)
              VectorSet_Insert(curr_killed, *arg_runner);

        Operation_ForAllDefTags(arg_runner, operation)
              VectorSet_Insert(curr_killed, *arg_runner);
    }
}
```

◇

Macro referenced in scrap 21a.

⟨Private variables 22c⟩ ≡

```
VectorSet *phi_node_uses = NULL;
```

◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

[2] In ILOC, the vertical bar ('|') construct separates operations in an instruction.

Finally, when we are computing this information for code already transformed into SSA form, we have to compute the set of values flowing along each edge. We will keep this information in the phi_node_uses array, which will be an array of vector sets indexed by block number. The members of each set represent the values passed into the ϕ-nodes of that block's successors. The variable parm_map is explained in Section 3.1.

⟨Collect ϕ-node uses 23a⟩ ≡

```
{
   Block *block; Phi_Node *phi_node_runner;

   phi_node_uses = (VectorSet *)Arena_GetMem(data_flow_memory_bank,
                                             sizeof(VectorSet) * (block_count + 1));
   ForAllBlocks(block)
   {
      Unsigned_Int block_num = block->preorder_index;
      Parm_Map_Node *node_runner = parm_map[block_num];

      phi_node_uses[block_num] = VectorSet_Create(data_flow_memory_bank, number_of_resources);
      while (node_runner)
      {
         Unsigned_Int parm_index = node_runner->index;
         Block *succ_block = node_runner->succ_block;

         Block_ForAllPhiNodes(phi_node_runner, succ_block)
             VectorSet_Insert(phi_node_uses[block_num], phi_node_runner->parms[parm_index]);
         node_runner = node_runner->next;
      }
   }
}
```

◇

Macro referenced in scrap 21a.

2.2.2 Determining global liveness iteratively

Having initialized our data-flow problem with local liveness information, we can perform a series of iterative passes over the CFG, computing each block's live_in set using the partially complete information we computed in the previous section. The process terminates when the information in the live_in sets stabilizes.

Here, we create the bit vector temp for use in the next scrap. It will be used for temporary storage, but we want to allocate it outside of the loop so that we can simply clear it at the beginning of each iteration instead of reallocating it.

⟨Perform iterative analysis to determine liveness 23b⟩ ≡

```
{
    Boolean changed;
    VectorSet temp = VectorSet_Create(liveness_memory_bank, number_of_resources);

    do
        ⟨Perform a pass of the iterative analysis, setting changed 24⟩
    while (changed);
}
```

◇

Macro referenced in scrap 20.

At each iteration, we make a postorder pass over the CFG examining each block. We take the successors of each block, find the union of their live_in sets (remember that this is the definition of live_out from equations 2.1 and 2.2 in Section 2.2, on page 19), and find the difference-union of the block's current live_in and killed sets and our calculation of live_out. If this calculation gives us a different set than the block's current `live_in` set, we copy the new set over the old one. When we make a complete pass over the CFG with no changes to any of the sets, the loop exits.

We perform each pass of the data-flow analysis by visiting the blocks in postorder (effectively working "bottom up") to ensure quickest termination. Notice the use of `temp` for both `live_out` and the new value of `live_in`.

⟨Perform a pass of the iterative analysis, setting `changed` 24⟩ ≡

```
{
    Block *block;

    changed = FALSE;
    ForAllBlocks_Postorder(block)
    {
        if (block != end_block)
        {
            Unsigned_Int block_num = block->preorder_index;
            VectorSet current_live_in = live_in[block_num];
            VectorSet current_live_out = (SSA_CFG_renumbered) ? live_out[block_num] : NULL;

            ⟨Calculate live_out into temp 25a⟩
            if (SSA_CFG_renumbered && !VectorSet_Equal(current_live_out, temp))
                VectorSet_Copy(current_live_out, temp);
            ⟨Use temp to calculate the new value of live_in, and copy it back into temp 25b⟩

            if (!VectorSet_Equal(current_live_in, temp))
            {
                changed = TRUE;
                VectorSet_Copy(current_live_in, temp);
            }
        }
    }
}
◇
```

Macro referenced in scrap 23b.

We calculate a block's `live_out` set by taking the union of all of the `live_in` sets of the block's successors and storing it in the local variable `temp`.

Notice that we do not use our `Block_ForAllSuccs` macro here, because we want to *copy* the first successor's `live_in` set to `temp`. Subsequent successors then get union-ed into `temp`. This is for efficiency; the alternative is to clear `temp` each time.

⟨Calculate `live_out` into `temp` 25a⟩ ≡

```
{
    Edge *edge = block->succ;

    VectorSet_Copy(temp, live_in[edge->succ->preorder_index]);
    for (edge=edge->next_succ; edge; edge=edge->next_succ)
        VectorSet_Union(temp, temp, live_in[edge->succ->preorder_index]);

    if (SSA_CFG_renumbered)
        VectorSet_Union(temp, temp, phi_node_uses[block->preorder_index]);
}
```

◇

Macro referenced in scrap 24.

The next step is to calculate the new value of `live_in` into `temp`. Of course, we just calculated the union of `block`'s successors' `live_in` sets into `temp`, so it initially has the value of `live_out`. We put up with a little confusion caused by the multiple uses for `temp` because the resulting code is a very efficient use of the `VectorSet`. For an explanation, recall that `VectorSet_DifferenceUnion(w, x, y, z)` computes $w \leftarrow (x \cap \overline{y}) \cup z$.

⟨Use `temp` to calculate the new value of `live_in`, and copy it back into `temp` 25b⟩ ≡

```
VectorSet_DifferenceUnion(temp, temp, killed[block_num], current_live_in);
```

◇

Macro referenced in scrap 24.

Once we have calculated the liveness sets for the blocks, we need to identify the blocks in which each resource is defined. This is the subject of the next section.

2.3 Finding assignments

We now need to accumulate the lists of blocks containing assignments to each resource. We will use the blocks in these lists as a guide to inserting ϕ-nodes.

At this time, we will also identify *global resources*, as suggested by Preston Briggs. Global resources are those resources which are defined in one block and used in another. The intent of identifying global resources is to reduce (perhaps drastically) the number of the assignment lists and the number of ϕ-nodes. This added calculation distinguishes minimal form from semi-pruned form, although, as we explain in the next paragraph, we also use this method to build the pruned form.

The published algorithm for minimal SSA form looks at the lists of blocks containing assignments for each resource, regardless of whether or not the resource is global. However, since we know that only resources live across block boundaries will have live ϕ-nodes, we can reduce the number of ϕ-nodes by only considering global resources when inserting ϕ-nodes. Even if we are building the pruned form (where we do not insert dead ϕ-nodes), this algorithm can save time, since we will look at only a subset of the resources in the routine.

In the worst case, both methods run in $O(R * B)$ time, where R is the number of resources in the routine and B is the number of blocks in the CFG. However, if the number of global resources r is very much smaller than R, we should see an improvement in the amount of time it takes to execute this code.

Note also that this method for building semi-pruned SSA does not prevent us from inserting dead ϕ-nodes. Only the liveness analysis will guarantee that. This method merely removes some of the time and space which would otherwise be required to build minimal SSA.

The code to record global resources could have been merged with the code in the last section. However, we have chosen to separate it into its own section for reasons of clarity.

⟨Find assignments 26a⟩ ≡
    ⟨Find global resources 26b⟩
    ⟨Accumulate assignment blocks for global resources 28a⟩
    ◇

Macro referenced in scrap 15a.

2.3.1 Finding global resources

To find global resources, we examine each block in the CFG. Within a block, we examine the instructions and operations in order. If an operation uses a resource that has not been defined in the block, then the resource is added to **global_resources**. Resources defined in the block are recorded in the **killed_in_block** set. We implement **killed_in_block** as a **SparseSet** so that we can quickly clear it at the beginning of each basic block. As we said above, if we are building the minimal SSA form, we do not need to do this computation.

⟨Find global resources 26b⟩ ≡

```
if (!build_minimal_form)
{
    Block *block;
    SparseSet killed_in_block = SparseSet_Create(data_flow_memory_bank,
                                                 register_count+tag_count);

    ForAllBlocks(block)
    {
        Inst *inst;

        SparseSet_Clear(killed_in_block);

        Block_ForAllInsts(inst, block)
        {
            ⟨Add exposed uses to global_resources 27a⟩
            ⟨Add all defined resources to killed_in_block 27b⟩
        }
    }
}
◇
```

Macro referenced in scrap 26a.

Within an instruction, we look at the uses in all the operations. If a use has not been killed within the block, it is added to **global_resources**. We look at all of the uses in all of the operations within a single instruction before adding any resources which may be defined by those operations to the **killed_in_block** set. This is because ILOC specifies that operations in a single instruction are performed in parallel. For more on this, see Section 2.2.1 on page 21.

⟨Add exposed uses to global_resources 27a⟩ ≡

```
{
    Operation  **op_runner;

    Inst_ForAllOperations(op_runner, inst)
    {
        Unsigned_Int2 *arg_runner; Unsigned_Int2 *tag_runner;
        Operation *operation = *op_runner;

        Operation_ForAllUses(arg_runner, operation)
        {
            Unsigned_Int arg = *arg_runner;

            if (!SparseSet_Member(killed_in_block, arg))
                SparseSet_Insert(global_resources, arg);
        }

        Operation_ForAllRefTags(tag_runner, operation)
        {
            Unsigned_Int tag = *tag_runner;

            if (!SparseSet_Member(killed_in_block, tag))
                SparseSet_Insert(global_resources, tag);
        }
    }
}
◇
```

Macro referenced in scrap 26b.

After examining all the uses in an instruction, we examine all the definitions in all the operations in that instruction, adding them to `killed_in_block`.

⟨Add all defined resources to killed_in_block 27b⟩ ≡

```
{
    Operation  **op_runner;
    Inst_ForAllOperations(op_runner, inst)
    {
        Unsigned_Int2 *arg_runner;
        Unsigned_Int2 *tag_runner;
        Operation *operation = *op_runner;

        Operation_ForAllDefs(arg_runner, operation)
            SparseSet_Insert(killed_in_block, *arg_runner);
        Operation_ForAllDefTags(tag_runner, operation)
            SparseSet_Insert(killed_in_block, *tag_runner);
    }
}
◇
```

Macro referenced in scrap 26b.

2.3.2 Building the list of blocks containing assignments to each resource

During the process of ϕ-node insertion, we will drive our worklist with lists of blocks in which the global resources (or, in the case of building minimal SSA, all resources) are defined. The following code will build these lists, loading them into our `assignments` array.

To build the **assignments** array, we examine all the blocks in the CFG, locating all the definitions in each block's instructions. For each defined resource r, we add the current block to the list in `assignments[`r`]`.

⟨Accumulate assignment blocks for global resources 28a⟩ ≡

```
{
    Block *block;

    ForAllBlocks(block)
    {
        Inst *inst;

        Block_ForAllInsts(inst, block)
            ⟨Examine all the definitions in inst 28b⟩
    }
}
◇
```

Macro referenced in scrap 26a.

We examine all the definitions in all the operations in `inst`. We will record the block into an `assignments` list only if the defined resource is in `global_resources` or if we are building minimal SSA. Additionally, we take this opportunity to increment `SSA_def_count`.

⟨Examine all the definitions in `inst` 28b⟩ ≡

```
{
    Operation  **op_runner, *operation;
    Unsigned_Int2 *arg_runner;

    Inst_ForAllOperations(op_runner, inst)
    {
        operation = *op_runner;
        Operation_ForAllDefs(arg_runner, operation)
        {
            Unsigned_Int arg = *arg_runner;

            SSA_def_count++;
            if (build_minimal_form || SparseSet_Member(global_resources, arg))
            {
                ⟨Add block to assignments[arg] 29a⟩
                assignments[arg]->def_type = Operation_Def_Type(operation, arg);
            }
        }
        Operation_ForAllDefTags(arg_runner, operation)
        {
            Unsigned_Int arg = *arg_runner;

            SSA_def_count++;
            if (build_minimal_form || SparseSet_Member(global_resources, arg))
            {
                ⟨Add block to assignments[arg] 29a⟩
                assignments[arg]->def_type = NO_DEFS; /* tags have no type */
            }
        }
    }
}
◇
```

Macro referenced in scrap 28a.

We put `block` into the list only if it is not already there. Of course, we do not need to check the whole list because the current block will only appear at the front of the list.

⟨Add `block` to `assignments[arg]` 29a⟩ ≡
```
{
    Assignment_Node *list_start = assignments[arg];

    if (!list_start || list_start->block != block)
    {
        Assignment_Node *new_node = (Assignment_Node *)Arena_GetMem(data_flow_memory_bank,
                                                                    sizeof(Assignment_Node));
        new_node->block = block;
        new_node->next = list_start;
        assignments[arg] = new_node;
    }
}
◇
```
Macro referenced in scrap 28b.

The code in the preceding sections has all been focussed on setting up the data-flow information which will tell us where to insert ϕ-nodes. The actual insertion will be discussed in the next section.

2.4 Placing ϕ-nodes

When a resource is defined along more than one path reaching a block, we need some way to maintain the SSA invariant that only one definition can reach a particular use. We do this by inserting pseudo functions called ϕ-nodes. A ϕ-node takes all of the incoming values for a particular resource as parameters and defines a single resource, thus ensuring our invariant.

The ϕ-nodes are stored in the externally-visible array `SSA_phi_nodes`, which we initialize, below.

⟨Initialize SSA variables 29b⟩ ≡
```
SSA_phi_nodes = (Phi_Node **)Arena_GetMemClear(permanent_memory_bank,
                                               sizeof(Phi_Node *) * (block_count+1));
◇
```
Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

The placement of ϕ-nodes is accomplished as follows: The `global_resources` set represents the resources which are live across block boundaries and, thus, are candidates for ϕ-nodes. For each resource r in `global_resources`, we examine the list of blocks in which it is defined, treating the list as a worklist (remember that we previously stored this list in the `assignments` array). For each block b in this worklist, we place a ϕ-node for r in each block in b's dominance frontier. We then add all of those blocks to the worklist, and continue until we have examined all of the blocks on the list. Of course, in the case of building minimal form, all of the resources are candidates, so we simply loop through the entire list of resources.

We ensure that a block never gets put on the worklist twice through the use of the `SparseSet in_list`. Before we add a block onto the worklist, we see if that block is already a member of `in_list`. If not, we add the block onto the worklist and put it in `in_list`.

Likewise, before we insert a ϕ-node into a block, we see if that block is already a member of the SparseSet has_node in order to ensure that a block does not get multiple ϕ-nodes for the same resource. We add one to block_count when we create the SparseSets because the blocks' preorder_index fields (which we use to uniquely identify each block) start numbering at one, not zero. Both of these sets get cleared for each resource.

⟨Place ϕ-nodes 30a⟩ ≡
```
{
    Unsigned_Int res;
    SparseSet in_list = SparseSet_Create(data_flow_memory_bank, block_count+1);
    SparseSet has_node = SparseSet_Create(data_flow_memory_bank, block_count+1);

    if (build_minimal_form)
        for(res=0;res<register_count+tag_count; res++)
            ⟨Place the φ-nodes for res 30b⟩
    else /* (building semi-pruned or pruned) */
        SparseSet_ForAll(res, global_resources)
            ⟨Place the φ-nodes for res 30b⟩
    Arena_Destroy(data_flow_memory_bank); /* we are done with data-flow information */
}
◇
```
Macro referenced in scrap 15a.

Below is the main code to handle the insertion of ϕ-nodes for each global resource. We begin by clearing the has_node set and initializing the in_list set with the blocks already on the worklist. We then iterate through the worklist, placing ϕ-nodes in each of the blocks' dominance frontiers.

The assignments array was built in a separate pass over the code. Our assignment_list variable, however, is the actual worklist over which we shall iterate. It is interesting to note that all blocks which are subsequently added to the worklist will never show up in the assignments array. There is no need to put them into the assignments array, because the lifetime of this array continues only until we have inserted all of the necessary ϕ-nodes, and we will never reference any element in the array (*i.e.*, any list of blocks) more than once.

Note that we always initialize the has_node set with the index of end_block. This block may well have multiple edges coming into it (*e.g.*, if the routine has multiple return statements), so it may look like a join point. However, since it has no variables, it has no need of ϕ-nodes. This initialization ensures that we never put a ϕ-node into end_block, even if we are building minimal or semi-pruned SSA form and thus do not have the data-flow analysis of the pruned form to restrict it.

⟨Place the ϕ-nodes for res 30b⟩ ≡
```
{
    Assignment_Node *assignment_list = assignments[res]; /* our worklist */

    SparseSet_Clear(has_node);
    SparseSet_Insert(has_node, end_block->preorder_index);

    ⟨Initialize the set in_list 31a⟩
    while (assignment_list)
    {
        Block *block = assignment_list->block;
        Unsigned_Int def_type = assignment_list->def_type;

        assignment_list = assignment_list->next;
        ⟨Place φ-nodes for res in block's iterated dominance frontier and update assignment_list 31b⟩
    }
}
◇
```
Macro referenced in scrap 30a.

The first thing that we have to do while placing ϕ-nodes for a resource **res** is to initialize the **in_list** set. As we said above, this set will represent the blocks in which **res** is defined and will keep us from adding a block to the worklist more than once.

We initialize **in_list** with the blocks in our initial worklist. We use each block's preorder index to uniquely identify it in the set.

⟨Initialize the set `in_list` 31a⟩ ≡

```
{
    Assignment_Node *node_runner;

    SparseSet_Clear(in_list);
    for (node_runner=assignment_list; node_runner; node_runner=node_runner->next)
        SparseSet_Insert(in_list, node_runner->block->preorder_index);
}
◇
```

Macro referenced in scrap 30b.

We insert ϕ-nodes into some number of the blocks in the dominance frontier of **block**, dependent on the following rule: If **res** is live on entry to a block (if we are building pruned SSA) and the block does not yet have a ϕ-node for **res**, then we need to add a new ϕ-node. This new ϕ-node implies extension of the dominance frontier, since it has added a new block with an assignment to **res**.

⟨Place ϕ-nodes for `res` in `block`'s iterated dominance frontier and update `assignment_list` 31b⟩ ≡

```
{
    Block_List_Node *frontier;

    Dominator_ForFrontier(frontier, block->dom_node)
    {
        Block *frontier_block = frontier->block;
        Unsigned_Int frontier_index = frontier_block->preorder_index;

        if (!SparseSet_Member(has_node, frontier_index) &&
            (!build_pruned_form || VectorSet_Member(live_in[frontier_index], res)))
        {
            SparseSet_Insert(has_node, frontier_index);
            ⟨Place a φ-node for res in frontier_block 31c⟩
            ⟨Put frontier_block into assignment_list 32b⟩
        }
    }
}
◇
```

Macro referenced in scrap 30b.

We insert a new ϕ-node into a block by simply allocating a new data structure and filling in its fields.

⟨Place a ϕ-node for `res` in `frontier_block` 31c⟩ ≡

```
{
    Phi_Node *new_node = ⟨Allocate a new phi-node 32a⟩;

    SSA_def_count++; /* for the phi-node just created */
    new_node->old_name = res;
    new_node->def_type = def_type;
    new_node->next = SSA_phi_nodes[frontier_index];
    SSA_phi_nodes[frontier_index] = new_node;
}
◇
```

Macro referenced in scrap 31b.

We use `Arena_GetMemClear` for the allocation of a ϕ-node, because we want to ensure that the elements in its `parms` array are initialized to zero.[3] Note that we also subtract one from the number of parameters needed for the `parms` array, since the structure definition already includes room for one element.

⟨Allocate a new phi-node 32a⟩ ≡
```
(Phi_Node *) Arena_GetMemClear(permanent_memory_bank,
                               sizeof(Phi_Node) +
                                   ((frontier_block->pred_count-1)*sizeof(Unsigned_Int2)));
◇
```
Macro referenced in scrap 31c.

Each block into which we insert a ϕ-node for `res` becomes a block where `res` is defined, so we have to ensure that this block gets added to `assignment_list` and its index gets added to `in_list`.

⟨Put `frontier_block` into `assignment_list` 32b⟩ ≡
```
if (!SparseSet_Member(in_list, frontier_index))
{
    Assignment_Node *new_node =
                    Arena_GetMem(data_flow_memory_bank, sizeof(Assignment_Node));

    new_node->block = frontier_block;
    new_node->def_type = def_type;
    new_node->next = assignment_list;
    assignment_list = new_node;

    SparseSet_Insert(in_list, frontier_index);
}
◇
```
Macro referenced in scrap 31b.

With ϕ-nodes inserted, we have completed all of the work necessary before renaming resources.

2.5 Renaming resources

In the final step in the translation into SSA form, we renumber all the resources so that each assignment (including ϕ-nodes) defines a unique resource name. The renumbering is handled by a recursive preorder walk over the dominator tree.

We can summarize the renaming process as follows: We declare an array of stacks (indexed by the original resource name) to hold the new names for each resource, and we use the topmost name on the stack to replace each use of that resource. We push a name onto a resource's stack each time we encounter a definition of that resource. We use the `Name` data structure to implement the name stacks.

⟨Internal data structures 32c⟩ ≡
```
typedef struct name Name;
struct name {
  Name *next;
  Unsigned_Int2 new_name;
  Unsigned_Int2 next_pushed;
};
◇
```
Macro defined by scraps 17d, 32c, 59a, 69a.
Macro referenced in scrap 5b.

[3] After we finish building SSA form, any element which is still zero will signify that the resource is not defined along the associated path in the CFG.

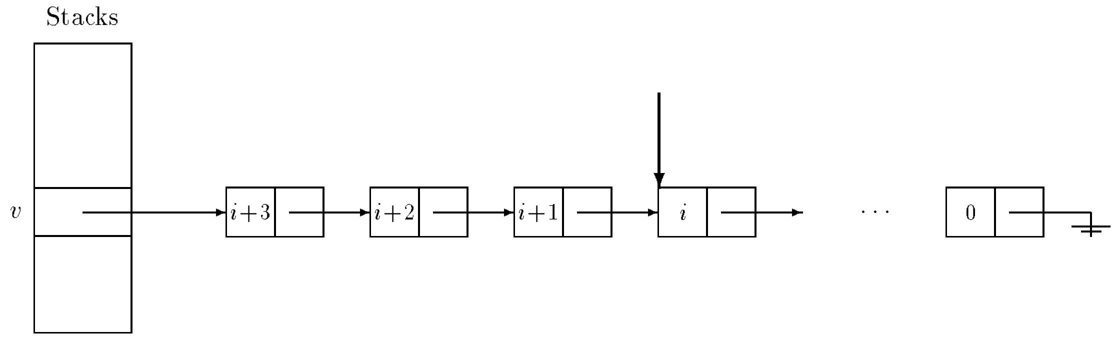


Figure 2.1: Stacks after variable v is defined three times

When we have finished processing a block (and its descendants in the dominator tree), we must restore the stacks to the same state as when we began processing the block. The method suggested by Cytron *et al.* is to iterate a second time through the instructions, this time popping a name off of the stack for each definition. Taylor Simpson developed an improvement to this method. Pushing a node for each definition in a block is wasteful. Consider a basic block that defines a variable v three times. Figure 2.1 shows the stack for v after processing the definitions in the block. The vertical arrow indicates the point where we must restore the stack. Notice that once the $i+2$ node gets pushed, the $i+1$ node can never be accessed again because any subsequent reference to v will use the name in the $i+2$ node and restoring the state of the stack will remove the $i+1$ node. Similarly, after the $i+3$ node gets pushed, the $i+2$ node will never be accessed again. We can reduce the number of nodes allocated if we overwrite the $i+1$ node with $i+2$ and then with $i+3$. However, we cannot overwrite the i node because it must remain after we have restored the state of the stack. Therefore, we will push a node onto the stack for v at the *first* definition of v in the block, but subsequent definitions of v will simply overwrite the node. We will use `SparseSet` to indicate which variables have already had a node pushed for the current block. If a variable in `pushed` is defined, we will overwrite its top-of-stack instead of pushing a new node.

⟨Private variables 33a⟩ ≡
```
static SparseSet pushed;
```
◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Initialize variables during renaming 33b⟩ ≡
```
pushed = SparseSet_Create(rename_memory_bank, register_count + tag_count);
```
◇

Macro defined by scraps 33b, 46d.
Macro referenced in scrap 34.

Since we are pushing at most one node for each variable when we process the definitions in a block, we can no longer restore the state of the stack by iterating over the operations in the block and popping a node for each definition. We could simply pop one node for each variable in `pushed`. However, this solution would require a `SparseSet` for each block (or at least one for each level in the dominator tree), and would use more memory than we can hope to save. On the other hand, we can link the nodes in `pushed` together as they are added to the set. We use the `next_pushed` field in the `Name` structure for this purpose. For each block, we maintain a list of the variables with a node to be popped. The head of the list will be held in `pushed_list`, a local variable in the `rename_resources` function. Initially the list is empty, and when a new node is pushed, it will be added to the list. Thus, restoring the state of the stacks requires popping the nodes in pushed_list. This data structure is shown in Figure 2.2. Notice that the next_pushed entries in the list are simply indices into the array of Name structures.

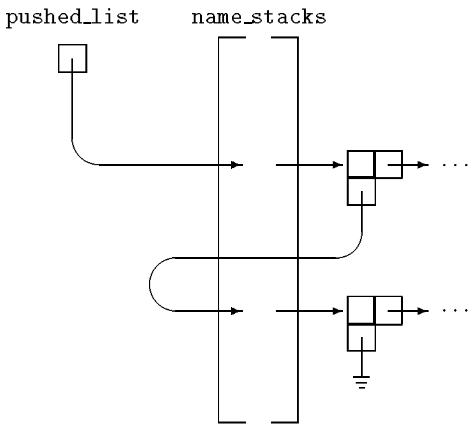


Figure 2.2: The name_stacks array, showing the connection between nodes in each stack

In summary, we use the pushed set to ensure that at most one Name node gets pushed per block, and we use the pushed_list to guide the popping of the stacks. This improvement not only keeps us from allocating superfluous Name nodes, but it also speeds up the popping phase at the end. The naive approach requires a second pass through the instructions in the code, popping a node off of each definition's stack as it is encountered. With this new method, we can simply iterate down the list of elements, popping just one node off of each stack.

The speedup of this method comes at the cost of very little memory. Since the next_pushed field is an Unsigned_Int2, it does not add to the total memory occupied by a Name structure. Therefore, the pushed set occupies the only additional memory used. We expect that the savings from allocating fewer Name nodes should compensate for this.

To perform the renaming of variables, we will use the recursive rename_resources function, defined below. Since we use Arena_Mark and Arena_Release inside rename_resources, it is important that we do not send that routine an arena which we might be using for some other purpose. Therefore, SSA_Build defines its own arena, rename_memory_bank, for just this reason.

We add tag_count to the total number of definitions (SSA_def_count) because we think of memory as having an implicit initialization upon entry of the module. This means that we have to create initial names for each of the tags and thus need an extra tag_count entries in each of the arrays to hold these initial names.

⟨Rename resources 34⟩ ≡

```
{
    Arena rename_memory_bank = Arena_Create();
    Name **name_stacks =
        Arena_GetMemClear(rename_memory_bank, (register_count+tag_count)*sizeof(Name *));

    SSA_def_count += tag_count;
    ⟨Initialize variables during renaming 33b, ... ⟩
    ⟨Initialize variables whose size depends on SSA_def_count 35a, ... ⟩
    ⟨Initialize name_stacks 36b⟩
    rename_resources(start_block, name_stacks, rename_memory_bank);
    Arena_Destroy(rename_memory_bank);
}
◇
```

Macro referenced in scrap 15a.

Having arrived at the total number of definitions in the routine, we first allocate all of the data structures which will be indexed by resource name and whose size is dependent on `SSA_def_count`. The first of these are the `SSA_block_map` and `SSA_inst_map` arrays, which map a resource to the block and instruction, respectively, in which the resource is defined.

Although arrays in C use zero-based indexing and we begin numbering our SSA names starting at one, we do not need to add one to `SSA_def_count` when allocating arrays based on its value because `SSA_def_count` is always one more than the actual number of definitions in the routine.

⟨Initialize variables whose size depends on SSA_def_count 35a⟩ ≡

```
SSA_block_map = (Block **)Arena_GetMemClear(permanent_memory_bank,
                                            sizeof(Block *) * (SSA_def_count));
SSA_inst_map = (Inst **)Arena_GetMemClear(permanent_memory_bank,
                                          sizeof(Inst *) * (SSA_def_count));
```

◇

Macro defined by scraps 35abc.
Macro referenced in scrap 34.

Next, we will allocate the `SSA_name_map` array, which stores the mapping of resource names to their original names.

⟨Initialize variables whose size depends on SSA_def_count 35b⟩ ≡

```
SSA_name_map = (Unsigned_Int2 *)Arena_GetMemClear(permanent_memory_bank,
                                                  sizeof(Unsigned_Int2) * (SSA_def_count));
```

◇

Macro defined by scraps 35abc.
Macro referenced in scrap 34.

Finally, we will allocate memory for the two arrays of chains.

⟨Initialize variables whose size depends on SSA_def_count 35c⟩ ≡

```
if (build_use_def_chains)
    SSA_use_def_chains = (Chain *)Arena_GetMemClear(permanent_memory_bank,
                                                    sizeof(Chain) * (SSA_def_count));
if (build_def_use_chains)
    SSA_def_use_chains =
            (Chains_List **)Arena_GetMemClear(permanent_memory_bank,
                                              sizeof(Chains_List *) * (SSA_def_count));
```

◇

Macro defined by scraps 35abc.
Macro referenced in scrap 34.

To choose new names, we simply increment a global counter. We initialize `new_name_counter` to one instead of zero because zero has so many other meanings (*e.g.*, zero is our end-of-list marker for tag lists in the `operation` data structure, zero is used to indicate an undefined parameter in a ϕ-node's `parms` list, *etc.*).

⟨Private variables 35d⟩ ≡

```
static Unsigned_Int new_name_counter = 1;
```

◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

Each time a new name is created, we need to push it onto a stack. To facilitate this, we use a routine called `push_name`. We define the `push_name` routine in Section 2.5.4, on page 45. The `push_name` procedure pushes a new value (**new_name**) onto the stack at a particular index (**old_name**) in the stack array. Declaring this procedure here will give us a base upon which we can build the rest of the discussion of renaming variables. The prototype is as follows:

⟨Internal function prototypes 36a⟩ ≡
```
static Void push_name(Name **stack, Arena memory_bank,
                      SparseSet pushed, Unsigned_Int2 *node_list_pointer,
                      Unsigned_Int old_name, Unsigned_Int new_name);
```
◇

Macro defined by scraps 36a, 37a, 52b, 68b, 69c.
Macro referenced in scrap 5b.

Before we can begin renaming, we have to set up the **name_stacks** array. This entails creating an initial name for each of the tags in the procedure. This is an important initialization; we detect uninitialized resources by a use of a resource when that resource's stack is empty. This is a natural way to detect the use of a register before it is defined, but it does not make sense for memory locations. We have no way to know if a particular location in memory has been defined when the procedure is entered, so we take the conservative assumption that all memory has been initialized previously. Thus, any load from that location should be valid. We encode this by pushing a name onto the top of the stack for each tag before processing, ensuring that all uses will have a valid name.

This initialization is a good example of the work we must do when creating new names. Creating new names for a resource includes not only pushing a name onto the appropriate stack in **name_stacks**, but also marking the old name in the **SSA_name_map** array and adding an entry into the **SSA_use_def_chains** array (if necessary).

Every block is created with a **NOP** operation as the first operation in the block. We use this operation as a marker for iterating through a block's list of instructions. In order to ensure that we do not accidentally dereference a Λ pointer, we make use of this marker operation to give the **defining_operation** field for each newly created tag name a non-Λ value.

Notice that for this initial call to **push_name**, we send in a dummy variable for the linked-list pointer. Since these values never get popped, there is no reason to keep a linked list of them.

⟨Initialize name_stacks 36b⟩ ≡
```
{
    Unsigned_Int i;
    Operation *NOP_operation = start_block->inst->operations[0];

    for(i = register_count; i < register_count+tag_count; i++)
    {
        Unsigned_Int new_name = new_name_counter++;
        Unsigned_Int2 dummy;

        push_name(&name_stacks[i], rename_memory_bank, pushed, &dummy, i, new_name);
        SSA_name_map[new_name] = i;
        SSA_block_map[new_name] = start_block;
        SSA_inst_map[new_name] = start_block->inst;

        if (build_use_def_chains)
        {
            SSA_use_def_chains[new_name].is_phi_node = FALSE;
            SSA_use_def_chains[new_name].op_pointer.operation = NOP_operation;
        }
    }
}
```
◇

Macro referenced in scrap 34.

Now that name_stacks is set up, we can begin recursively renaming resources.

⟨Internal function prototypes 37a⟩ ≡

```
static Void rename_resources(Block *block, Name **name_stacks, Arena work_arena);
```

◇

Macro defined by scraps 36a, 37a, 52b, 68b, 69c.
Macro referenced in scrap 5b.

As we mentioned above, this function walks over the dominator tree. At each iteration, we generate new names for all of the ϕ-nodes and examine all of the instructions in block. At each instruction, we rewrite all resource references with their new names, and invent new names for all defined resources, pushing them onto the name_stacks. We then fill in successor blocks' ϕ-nodes' parms array with the values flowing out of this block and recur on block's children in the dominator tree. The final step is to clean up this iteration by popping those names which were pushed onto name_stacks.

⟨Internal function bodies 37b⟩ ≡

```
static Void rename_resources(Block *block, Name **name_stacks, Arena work_arena)
{
    Inst *inst_runner;
    Unsigned_Int2 pushed_list = 0;

    SparseSet_Clear(pushed);
    Arena_Mark(work_arena);
    ⟨Generate new names for φ-nodes 38a⟩
    Block_ForAllInsts(inst_runner, block)
    {
        ⟨Replace used resources with new names 39a⟩
        ⟨Invent new names for defined resources and replace them in the ILOC code 44a⟩
    }
    ⟨Fill successors' φ-nodes' parms arrays 47a⟩
    ⟨Recur on block's children in the dominator tree 48b⟩
    ⟨Pop the pushed names off of the name stacks 48c⟩
    Arena_Release(work_arena);
} /* rename_resources */
```

◇

Macro defined by scraps 37b, 45b, 53a, 68c, 69d.
Macro referenced in scrap 5b.

2.5.1 Inventing new names for ϕ-nodes

The first step is to rename the resources defined by the ϕ-nodes in the current block. We iterate through the ϕ-nodes in the block, assigning the current value of new_name_counter to each one.

⟨Generate new names for ϕ-nodes 38a⟩ ≡
```
{
    Phi_Node *phi_node_runner;

    Block_ForAllPhiNodes(phi_node_runner, block)
    {
        Unsigned_Int old_name = phi_node_runner->old_name;
        Unsigned_Int new_name = new_name_counter++;

        phi_node_runner->new_name = new_name;
        SSA_name_map[new_name] = old_name;
        SSA_block_map[new_name] = block;
        ⟨Push the new φ-node name onto name_stacks 38b⟩
        if (build_use_def_chains)
        {
            SSA_use_def_chains[new_name].is_phi_node = TRUE;
            SSA_use_def_chains[new_name].op_pointer.phi_node = phi_node_runner;
        }
    }
}
◇
```
Macro referenced in scrap 37b.

Of course, we must push every new name that we create onto the name stacks. We separated this piece of code to emphasize the fact that we do not have to perform the extra check of making sure that we have not already pushed a value onto a particular stack. This is true for two reasons: this is the first time any of these names could have been pushed onto the stacks for this block, and we are guaranteed of having at most one ϕ-node for a particular variable,

⟨Push the new ϕ-node name onto `name_stacks` 38b⟩ ≡
```
push_name(&name_stacks[old_name], work_arena, pushed, &pushed_list, old_name, new_name);
SparseSet_Insert(pushed, old_name);
◇
```
Macro referenced in scrap 38a.

Once we have inserted the names of the ϕ-nodes onto the name stacks, we can rewrite resource references in this block using the new SSA names.

2.5.2 Rewriting uses of resources with SSA names

The original resource name is an index into `name_stacks`, which gives us the stack of new names created for this resource. The name at the top of the stack represents the definition which reaches this use, so we replace the use with the SSA name.

If the name stack is empty for a use of a register, it means that the operation uses an undefined register. This is divergent code, and so we are free to do anything we like with it. Most of the time, we will simply change the operation into a **NOP**, although certain operations (*e.g.*, control-flow operations) cannot be deleted from the code, so we change them into something which will allow us to continue analyzing the code. Notice that we have to break out of the loop when we encounter one of these operations. The code for fixing divergent operations is in the next section.

⟨Replace used resources with new names 39a⟩ ≡
```
{
    Operation **op_runner;

    Inst_ForAllOperations(op_runner, inst_runner)
    {
        Unsigned_Int2 *use_runner;
        Operation *operation = *op_runner;
        Unsigned_Int position = operation->constants;

        Operation_ForAllUses(use_runner, operation)
        {
            if (!name_stacks[*use_runner])
            {
                ⟨Fix divergent operation 40b⟩
                break;
            }
            else
            {
                ⟨Replace the use with the new name and build the def-use chain, if necessary 39b⟩
                position++;
            }
        }
        position=operation->defined;
        Operation_ForAllRefTags(use_runner, operation)
        {
            ⟨Replace the use with the new name and build the def-use chain, if necessary 39b⟩
            position++;
        }
    }
}
◇
```
Macro referenced in scrap 37b.

We pulled this scrap out because it can be used for both registers and tags.

⟨Replace the use with the new name and build the def-use chain, if necessary 39b⟩ ≡
```
{
    *use_runner = name_stacks[*use_runner]->new_name;
    if (build_def_use_chains)
        ⟨Add a new Chains_List node 40a⟩
}
◇
```
Macro referenced in scrap 39a.

If we are building def-use chains, we need to add a new node to the list of `Chain_List` nodes for each use of a variable in the operation.

⟨Add a new `Chains_List` node 40a⟩ ≡

```
{
    Chains_List *new_node = (Chains_List *)Arena_GetMem(permanent_memory_bank,
                                                         sizeof(Chains_List));
    Unsigned_Int resource_name = *use_runner; /* avoids derefs */

    new_node->chain.op_pointer.operation = operation;
    new_node->chain.is_phi_node = FALSE;
    new_node->position = position;
    new_node->next = SSA_def_use_chains[resource_name];
    SSA_def_use_chains[resource_name] = new_node;
}
◇
```

Macro referenced in scrap 39b.

2.5.3 Fixing divergent code

If an operation uses an undefined register, the result is meaningless. In general, we convert these operations into `NOP` operations. There are, however, three kinds of operations, the `BR` operations, `JMPr` operation, and `RTN` operations, which, for the purposes of keeping the structure of the CFG consistent, cannot be removed.

In general, we don't expect many cases of operations using uninitialized registers. However, our front end will occasionally generate such operations. For example, the front end automatically stores to memory all subroutine parameters. For output parameters, this is legal code, even though it references an undefined register. Of course, since it neither does anything useful nor affects the correctness of the code, such an operation should be deleted.

Other operations *could* affect the correctness of the code, but we have no way to differentiate between the two cases. We therefore do the minimal amount of alteration which will allow us to continue processing, and we alert the user of the divergent operation (and our solution to it) so that he can decide if the code requires attention.

We set up the error reporting to print a message explaining that we found an uninitialized register and its location. Once we decide what to do with it (at each case in the `switch` statement below), we print the solution implemented.

⟨Fix divergent operation 40b⟩ ≡

```
if (complain_about_uninitialized_registers)
    fprintf(stderr,
            "Found an uninitialized register, opcode = %s, register = %d.\n",
            opcode_specs[operation->opcode].opcode, *use_runner);

switch(operation->opcode)
{
    ⟨Handle un-removable operations 41b, ... ⟩
    default:
        ⟨Convert to NOP 41a⟩
        if (complain_about_uninitialized_registers)
            fprintf(stderr, "\tConverted the operation to a NOP.\n");
        break;
}
◇
```

Macro referenced in scrap 39a.

If the operation is one which can be deleted, we need only redefine a few of the fields of `operation` in order to make it look like a `NOP` operation. The process of converting an operation to a `NOP` is straightforward, and we will use this scrap of code to "remove" copy operations as well.

⟨Convert to `NOP` 41a⟩ ≡
```
operation->opcode = NOP;
operation->constants = 0;
operation->referenced = 0;
operation->defined = 0;
◇
```
Macro referenced in scraps 40b, 43c, 44a.

The operations we try to keep are those affecting control flow, including the `xRTN` operations. For each, we try to do something semi-reasonable.

For branches, pick one of the two destination labels, and use that as the target of a jump-to-label operation.

⟨Handle un-removable operations 41b⟩ ≡
```
case BR:
case BReq: case BRne: case BRle: case BRge: case BRlt: case BRgt:
    ⟨Convert branch to JMPl operation 41c⟩
    if (complain_about_uninitialized_registers)
        fprintf(stderr, "\tConverted the operation to a JMPl.\n");
    break;
◇
```
Macro defined by scraps 41b, 42ad, 43c.
Macro referenced in scrap 40b.

Given our CFG data structures, converting any operation into another is a simple matter of writing over some number of the fields in the original operation's data structure. The only caveat is that the original operation must have at least as many arguments as the new operation so that there is room in the `arguments` array for the new operation's arguments. Since the branch operations all have three arguments and the `JMPl` operation only has one, this is not a problem. Notice that in fixing these operations, we have randomly chosen one of the two destinations.

⟨Convert branch to `JMPl` operation 41c⟩ ≡
```
{
    Expr destination_label = operation->arguments[0];

    ⟨Rewrite operation into a JMPl operation 41d⟩
}
◇
```
Macro referenced in scrap 41b.

We are going to use this same code for the JMPr case (below), so we have pulled it out into a separate scrap.

⟨Rewrite operation into a `JMPl` operation 41d⟩ ≡
```
operation->opcode = JMPl;
operation->constants = 1;
operation->referenced = 1;
operation->defined = 1;
operation->arguments[0] = destination_label;
◇
```
Macro referenced in scraps 41c, 42b.

For the jump-to-register operation, we simply choose the first label in the list of possible destinations, and convert it to a jump-to-label operation, just like we did for branches.

⟨Handle un-removable operations 42a⟩ ≡
```
case JMPr:
        ⟨Convert jump-to-register to JMPl operation 42b⟩
        if (complain_about_uninitialized_registers)
            fprintf(stderr, "\tConverted the operation to a JMPl.\n");
        break;
```
◇

Macro defined by scraps 41b, 42ad, 43c.
Macro referenced in scrap 40b.

The list of destinations to a `JMPr` operation could be empty. If so, we will set the new jump operation to point to the first block in the CFG. Otherwise, this requires the same work as was necessary in the case of replacing a branch operation.

⟨Convert jump-to-register to `JMPl` operation 42b⟩ ≡
```
{
    Expr destination_label = operation->arguments[1];

    if (!destination_label /* the list of labels was empty */)
        ⟨Set destination_label to the label of the first block 42c⟩

    ⟨Rewrite operation into a JMPl operation 41d⟩
}
```
◇

Macro referenced in scrap 42a.

If the `JMPr`'s list of labels is empty, then we set `destination_label` to point to the label attached to the first block in the code. We use this label simply for convenience. (Remember that we have already determined that the code is divergent!)

⟨Set `destination_label` to the label of the first block 42c⟩ ≡
```
destination_label = start_block->succ->succ->labels->label;
```
◇

Macro referenced in scrap 42b.

We cannot remove `xRTN` operations, but we can force them to use sensible register values. We know that the first register defined in the `FRAME` operation (guaranteed to be the first operation that gets executed) will always be defined, so we will use that one.

⟨Handle un-removable operations 42d⟩ ≡
```
case RTN: case iRTN: case fRTN: case dRTN: case cRTN: case qRTN:
        ⟨Find a defined register and use it in place of the bad register(s) 43a⟩
        if (complain_about_uninitialized_registers)
            fprintf(stderr, "\tRewrote the RTN operation.\n");
        break;
```
◇

Macro defined by scraps 41b, 42ad, 43c.
Macro referenced in scrap 40b.

It is irrelevant which of the two registers in the RTN operation was undefined, since *both* could be undefined. Instead, we replace both registers with a register whose value we know is defined, in this case, the first register in a FRAME statement.[4]

⟨Find a defined register and use it in place of the bad register(s) 43a⟩ ≡

```
{
    Unsigned_Int legal_register_value = ⟨FRAME-defined register value 43b⟩;

    operation->arguments[0] = legal_register_value;
    /* remember that RTN operations only have _one_ argument, but the
       other flavors each have _two_ */
    if (operation->opcode != RTN)
        operation->arguments[1] = legal_register_value;
}
◇
```

Macro referenced in scrap 42d.

Finding the first register which is used in a FRAME statement is easy, although the code looks a little convoluted. We can find the FRAME statement easily because we are guaranteed that it is the first operation in the first instruction of the successor block of start_block.

To make explicit what the following code does, we take the root of the CFG (start_block), and look at the first successor edge coming out of it (succ); we look at the block to which the edge points (succ) and get its instruction list (inst); but we know that the first instruction in this list is a NOP marker-instruction, so we need to get the next instruction in the list (next_inst); finally, we know that the first operation in the instruction will be the FRAME operation, and we know that the first defined register is in position one in the operation's arguments array.

⟨FRAME-defined register value 43b⟩ ≡

```
start_block->succ->succ->inst->next_inst->operations[0]->arguments[1]
◇
```

Macro referenced in scrap 43a.

Like RTN operations, JSR operations cannot be removed from the code. The first register in JSRr operations, however, is the destination of the subroutine. A jump to an unknown location is reasonably certain to fail, so these are easy to handle. An undefined parameter (held in the rest of the JSR operation's arguments), however, may or may not cause divergence, so the best we can do is report that there might be a problem.

⟨Handle un-removable operations 43c⟩ ≡

```
case JSRr: case iJSRr: case fJSRr: case dJSRr: case cJSRr: case qJSRr:
        if (&(operation->arguments[0]) == use_runner)
        {
            fprintf(stderr,
                    "\tUndefined subroutine location. Converted the operation to a NOP.\n");
            ⟨Convert to NOP 41a⟩
        }
        else
            fprintf(stderr, "\tUnable to modify the call.\n");
        break;
case JSRl: case iJSRl: case fJSRl: case dJSRl: case cJSRl: case qJSRl:
        fprintf(stderr, "\tUnable to modify the call.\n");
        break;
◇
```

Macro defined by scraps 41b, 42ad, 43c.
Macro referenced in scrap 40b.

[4] The first register in a FRAME statement represents the value of the program counter of the calling procedure.

2.5.4 Inventing new names for defined variables

Having renamed all of the uses of resources in the operations in the current instruction, we now need to assign new names to the definitions. Notice that we process all of the uses in all of an instruction's operations before doing any of the definitions. Since multiple operations in a single instruction get executed in parallel, we have to consider all of the referenced resources as a group before we consider any of the defined resources.

When inventing new names, we push values onto `name_stacks`. If we are consuming copies, we do so at this phase. When we encounter a copy operation, we will push the source of the copy onto the destination's stack of names and remove the operation. Subsequent references to the destination will then automatically get the source's current SSA name.

We do not have a special case for handling operations which use uninitialized registers. The reason for this is that by this time, such operations have either been deleted or rewritten with the names of resources which have already been renamed using SSA names.

⟨Invent new names for defined resources and replace them in the ILOC code 44a⟩ ≡

```
{
    Operation **op_runner;

    Inst_ForAllOperations(op_runner, inst_runner)
    {
        Operation *operation = *op_runner;

        if (fold_copies && ⟨The operation is a copy 44b⟩)
        {
            ⟨Push the argument's top-of-stack element onto the result's name stack 46a⟩
            ⟨Convert to NOP 41a⟩
        }
        else
        {
            Unsigned_Int2 *def_runner;

            Operation_ForAllDefs(def_runner, operation)
                ⟨Handle each resource definition 45a⟩
            Operation_ForAllDefTags(def_runner, operation)
                ⟨Handle each resource definition 45a⟩
        }
    }
}
◇
```

Macro referenced in scrap 37b.

We can decide if an operation is a copy operation by looking its opcode up in the `opcode_specs` array. We move this into its own scrap for purposes of clarity.

⟨The `operation` is a copy 44b⟩ ≡

```
opcode_specs[operation->opcode].details & COPY
◇
```

Macro referenced in scrap 44a.

For each definition of a resource, we must:

- create a new name
- decide if we have already seen a definition of the resource in this block by seeing if it is a member of the `pushed` set:
    - if *yes*, then we replace the `new_name` field in that resource's top-of-stack element with the name we just created

  - if *no*, then we create a new stack element, add it to the resource's stack, and add the resource name into the pushed set

- add a pointer to the defining operation to the SSA_use_def_chains array to create the chain
- replace the old name in the operation with the new name
- record the mapping of the new name to the old one in SSA_name_map
- record the block in which this definition occurs.

⟨Handle each resource definition 45a⟩ ≡

```
{
    Unsigned_Int old_name = *def_runner;
    Unsigned_Int new_name = new_name_counter++;

    push_name(&name_stacks[old_name], work_arena, pushed, &pushed_list, old_name, new_name);

    if (build_use_def_chains)
    {
        SSA_use_def_chains[new_name].is_phi_node = FALSE;
        SSA_use_def_chains[new_name].op_pointer.operation = operation;
    }
    *def_runner = new_name;
    SSA_name_map[new_name] = old_name;
    SSA_block_map[new_name] = block;
    SSA_inst_map[new_name] = inst_runner;
}
◇
```

Macro referenced in scrap 44a.

Pushing a name onto the stack requires allocating a new stack node, filling in its new_name field, and adding it onto the front of both the linked list of nodes and the linked list representing the stack of names for this resource (consult Figure 2.2).

⟨Internal function bodies 45b⟩ ≡

```
static Void push_name(Name **stack, Arena memory_bank,
                      SparseSet pushed, Unsigned_Int2 *node_list_pointer,
                      Unsigned_Int old_name, Unsigned_Int new_name)
{
    Name *new_node = (Name *)Arena_GetMem(memory_bank, sizeof(Name));

    if (SparseSet_Member(pushed, old_name))
        (*stack)->new_name = new_name;
    else
    {
        new_node->new_name = new_name;
        new_node->next_pushed = *node_list_pointer;
        *node_list_pointer = old_name;
        new_node->next = *stack;
        *stack = new_node;
        SparseSet_Insert(pushed, old_name);
    }
} /* push_name */
◇
```

Macro defined by scraps 37b, 45b, 53a, 68c, 69d.
Macro referenced in scrap 5b.

Remember that if we are folding copy operations, we have to push the SSA name of the source onto the destination's name stack, so we have slightly different parameters to push_name than for other operations.

We have already renamed the copy's source argument, so we know the value to push onto the destination's name stack without having to look it up. Likewise, we also know the correct stack in name_stacks onto which to push the new name, since we have not yet changed the destination's name.

⟨Push the argument's top-of-stack element onto the result's name stack 46a⟩ ≡

```
{
    Unsigned_Int source = operation->arguments[0];
    Unsigned_Int destination = operation->arguments[1];

    push_name(&name_stacks[destination], work_arena, pushed, &pushed_list,
              destination, source);
}
◇
```

Macro referenced in scrap 44a.

2.5.5 Filling the parms array for each ϕ-node

This section describes the steps necessary to set up the parms array for the ϕ-nodes in successors of the current block.

At each iteration of SSA_Build, we need to add the current block's outgoing edges to its successors' SSA_edge_maps lists. Since we have not yet allocated these lists, we will do so now.

⟨Initialize SSA variables 46b⟩ ≡

```
SSA_edge_maps = (Edge ***)Arena_GetMem(permanent_memory_bank,
                                        sizeof(Edge **) * (block_count + 1));
{
    Block *block_runner;

    ForAllBlocks(block_runner)
        if (block_runner != start_block)
            SSA_edge_maps[block_runner->preorder_index] =
                (Edge **)Arena_GetMem(permanent_memory_bank,
                                      sizeof(Edge*) * block_runner->pred_count);
}
◇
```

Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

We need to enter the appropriate value into each successor's ϕ-nodes' parms arrays. We do this by looking at the old_name field of each ϕ-node and inserting the top of the name stack for that old_name (*i.e.*, name_stacks[old_name]) into the ϕ-node's parms array at the current index. From this, it is obvious that we need some way to keep track of the current index into the ϕ-nodes' parms and SSA_edge_maps arrays. We use an array of counters, indexed by the blocks' preorder_index fields. We use Arena_GetMemClear to allocate parms_indices because we want the counters to be initialized to zero.

⟨Private variables 46c⟩ ≡

```
static Unsigned_Int2 *parms_indices;
◇
```

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Initialize variables during renaming 46d⟩ ≡

```
parms_indices = (Unsigned_Int2 *)Arena_GetMemClear(rename_memory_bank,
                                     sizeof(Unsigned_Int) * (block_count + 1));
◇
```

Macro defined by scraps 33b, 46d.
Macro referenced in scrap 34.

With parms_indices defined, we can show the main loop for this step. We will also set up a data structure which we will use inside the SSA_Replace_Phi_Nodes routine. This code is explained in Section 3.1.

⟨Fill successors' ϕ-nodes' parms arrays 47a⟩ ≡

```
{
    Edge *edge_runner;

    Block_ForAllSuccs(edge_runner, block)
    {
        Block *succ_block = edge_runner->succ;
        Unsigned_Int succ_block_num = succ_block->preorder_index;
        Unsigned_Int parms_index = parms_indices[succ_block_num]++;

        ⟨Fill in successor's φ-nodes 47b⟩
        ⟨Add a node to block's parm_map entry for succ_block 59d⟩
        SSA_edge_maps[succ_block_num][parms_index] = edge_runner;
    }
}
◇
```

Macro referenced in scrap 37b.

For each successor block, we need to iterate through its list of ϕ-nodes. For each ϕ-node, we need to look up the current name for that resource and insert it into the ϕ-node's parms array at the current index. As we explained earlier, we use zero for a parameter along whose edge a value for this resource is not supplied.

Notice that if the resource's stack is empty, we do nothing. Remember that when we allocated each ϕ-node's parms array, we initialized all of the fields to zero. This is the value we would otherwise have to copy into the current slot in the parms array. Since it is already there, we simply continue.

⟨Fill in successor's ϕ-nodes 47b⟩ ≡

```
{
    Phi_Node *phi_runner;

    Block_ForAllPhiNodes(phi_runner, succ_block)
    {
        Name *top_of_stack = name_stacks[phi_runner->old_name];
        Unsigned_Int2 *parms_array = phi_runner->parms;

        if (top_of_stack)
        {
            Unsigned_Int new_name = top_of_stack->new_name; /* avoids derefs */

            parms_array[parms_index] = new_name;
            if (build_def_use_chains)
                ⟨Add a def-use chain for this parameter 48a⟩
        }
    }
}
◇
```

Macro referenced in scrap 47a.

The entries in the `parms` arrays are uses of those variables, so if we are building def-use chains, we need to add a def-use chain for each resource which we add to a `parms` array.

⟨Add a def-use chain for this parameter 48a⟩ ≡
```
{
    Chains_List *new_node = (Chains_List *)Arena_GetMem(permanent_memory_bank,
                                                         sizeof(Chains_List));

    new_node->chain.op_pointer.phi_node = phi_runner;
    new_node->chain.is_phi_node = TRUE;
    new_node->position = parms_index;
    new_node->next = SSA_def_use_chains[new_name];
    SSA_def_use_chains[new_name] = new_node;
}
◇
```
Macro referenced in scrap 47b.

2.5.6 Finishing up

Having renamed all of the resources in `block`, we now want to rename all of the resources in `block`'s dominator-tree children. We do this by recursively calling `rename_resources` for each child.

⟨Recur on `block`'s children in the dominator tree 48b⟩ ≡
```
{
    Block_List_Node *child_block_runner;

    Dominator_ForChildren(child_block_runner, block->dom_node)
        rename_resources(child_block_runner->block, name_stacks, work_arena);
}
◇
```
Macro referenced in scrap 37b.

The last step before returning from this iteration of recursion is to pop all of the SSA names generated by this block from the name stacks. As we said in the introduction to this section, we have improved the published method of popping names off of the name stacks.[5] In our method, we walk over our linked list of nodes, popping each one as we reach it. Notice that we do not do the obvious check to make sure that there is a node at the top of the stack before we try to pop it. This is safe because we only pop stacks onto which we know we pushed. Notice also that we do not deallocate the node; when this invocation of `rename_resources` finishes, we free all of the memory allocated at once using `Arena_Release`.

⟨Pop the pushed names off of the name stacks 48c⟩ ≡
```
while (pushed_list)
{
        Unsigned_Int next_node = name_stacks[pushed_list]->next_pushed;

        name_stacks[pushed_list] = name_stacks[pushed_list]->next;
        pushed_list = next_node;
}
◇
```
Macro referenced in scrap 37b.

2.6 Folding useless ϕ-nodes

Certain applications of SSA can have their analyses hindered if useless ϕ-nodes are present. We define a useless ϕ-node as one in which all of the parameters have the same name. When we compare the parameters, we ignore any which are zero or have the same name as the resource defined by the ϕ-node itself. A parameter

[5] The "slow" method requires iterating through the entire list of operations and ϕ-nodes, popping a name for each definition.

equal to zero indicates that no value flows along the corresponding edge, and a parameter equal to the name of the ϕ-node itself indicates simply that the value assigned to the ϕ-node has moved along some back edge unchanged. Thus, we can ignore those parameters because they have no impact on the value carried by the ϕ-node.

Useless ϕ-nodes are disruptive because some applications trace the origins of their arguments by following use-def chains back to the defining operations. Often, this tracing must stop at a ϕ-node, because a ϕ-node represents multiple values flowing together, so an application's analysis can be forced to give up at that point. However, we do not want a useless ϕ-node to inhibit this kind of analysis, so we need remove them before `SSA_Build` returns.

Useless ϕ-nodes can occur as a result of either the programmer writing the code himself or as a result of folding copies. For example, consider the code in Figure 2.3. For purposes of this example, assume that we will make a separate pass over the code to perform copy folding, and that this pass takes place after we have converted the routine to SSA form.

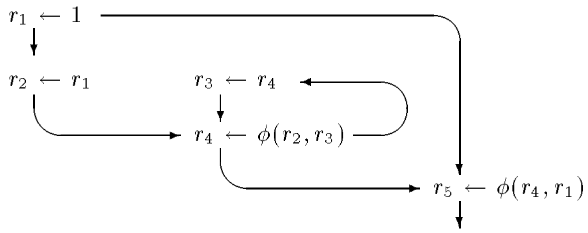


Figure 2.3: Original code in SSA form but prior to copy folding

The copy-folding pass will immediately find the copy of r_1 to r_2 and rewrite subsequent uses of r_2 with r_1, in this case, rewriting the parameter to the ϕ-node which defines r_4. Likewise, it will fold the copy of r_4 into r_3, again rewriting one of the parameters of the ϕ-node defining r_4. These changes are shown in Figure 2.4.

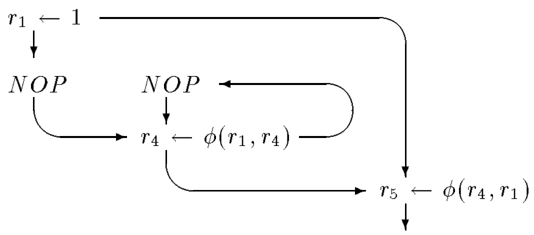


Figure 2.4: Code with copies folded and ϕ-node parameters rewritten

The pass to find useless ϕ-nodes will iterate over the graph, looking for ϕ-nodes which can be rewritten. Notice that the ϕ-node which defines r_4 fits our definition of useless, so we will eliminate it and rewrite all uses of r_4 as r_1. This will render the ϕ-node which defines r_5 useless as well. Notice that this can require multiple passes over the SSA form to find all useless ϕ-nodes, since the deletion of one ϕ-node can create the opportunity for the deletion of another.

There are two methods used to remove useless ϕ-nodes. The first is the iterative method, in which we simply walk over the graph, looking for useless ϕ-nodes. The second method uses a worklist and requires def-use chains. The expected execution time of the worklist method is better than for the iterative method, because we only have to look at a subset of the ϕ-nodes at each iteration. Of course, it also requires that def-use chains be built. We have included the code for both methods below.

```
⟨Fold useless φ-nodes, if necessary 50a⟩ ≡
    if (fold_useless_phi_nodes)
    {
        Arena useless_phi_node_memory_bank = Arena_Create();

        if (build_def_use_chains)
            ⟨Remove useless φ-nodes using the worklist method 54a⟩
        else
            ⟨Remove useless φ-nodes using the iterative method 51a⟩
        ⟨Remove the useless φ-nodes' data structures from the graph 50b⟩
        Arena_Destroy(useless_phi_node_memory_bank);
    }
    ◇
```

Macro referenced in scrap 15a.

Regardless of the method we use, we will want to remove the data structures representing the ϕ-nodes that have been found to be useless. This will keep other passes from thinking that the ϕ-node exists and is useful.

```
⟨Remove the useless φ-nodes' data structures from the graph 50b⟩ ≡
    {
        Block *block;

        ForAllBlocks(block)
        {
            Phi_Node **previous_phi_node = &SSA_phi_nodes[block->preorder_index];
            Phi_Node *current_phi_node = SSA_phi_nodes[block->preorder_index];

            while (current_phi_node)
            {
                while (current_phi_node && current_phi_node->useless)
                    current_phi_node = current_phi_node->next;

                *previous_phi_node = current_phi_node;

                if (current_phi_node)
                {
                    previous_phi_node = &current_phi_node->next;
                    current_phi_node = current_phi_node->next;
                }
            }
        }
    }
    ◇
```

Macro referenced in scrap 50a.

We should note that the idea of removing these useless ϕ-nodes is still somewhat experimental, and so the impact of this code is debatable. We still have two questions which need to be answered. The first concerns the efficacy of removing these ϕ-nodes. Put simply, we do not know if the extra expense in time is justified by the increase in accuracy. The second question concerns our definition of "useless". It is quite common for some of the parameters to a ϕ-node to be zero, which indicates that the value is undefined along the corresponding edge. We have not found a convincing argument to restrict the definition of "useless" to prohibit the removal of such ϕ-nodes.

2.6.1 Removing useless ϕ-nodes using the iterative method

In the absence of def-use chains, we can remove useless ϕ-nodes by using an iterative method accompanied by an array mapping resource names to new resource names. We initialize the mapping array to map each resource to itself. Proceeding in a reverse-postorder direction, we examine the parameters of each ϕ-node. If the ϕ-node is useless, we change the mapping of the ϕ-node's defined resource. Since a parameter to a ϕ-node can be defined by another ϕ-node, we have to iterate until we have found all of the useless ϕ-nodes (*i.e.,* until we complete a pass over the graph without deleting any ϕ-nodes). This algorithm satisfies all of the requirements for rapid algorithms, so it should halt in only a few iterations [4]. After the full set of the useless ϕ-nodes has been found, we will walk over the entire graph, using the resource map we have built to rename all of the resources.

```
⟨Remove useless φ-nodes using the iterative method 51a⟩ ≡
    {
        Unsigned_Int2 *resource_name, number_removed = 0;
        Boolean changed = TRUE;

        ⟨Initialize resource_name 51b⟩
        while (changed)
        {
            changed = FALSE;
            ⟨Search the graph for useless φ-nodes and set changed if one is found 52a⟩
        }
        ⟨Rewrite the graph using the new names in resource_name 53b⟩
        fprintf(stderr, "Number of useless phi-nodes removed: %d.\n", number_removed);
    }
    ◇
```

Macro referenced in scrap 50a.

Remember that we have to initialize the mapping of resource names to mirror the current set of names. We also initialize the zeroth element in `resource_name` so that we do not have to make any special checks in `parmcmp` (below).

```
⟨Initialize resource_name 51b⟩ ≡
    {
        Int i;

        resource_name = (Unsigned_Int2 *)Arena_GetMem(useless_phi_node_memory_bank,
                                                sizeof(Unsigned_Int2) * (SSA_def_count));
        for (i=0; i<SSA_def_count; i++)
            resource_name[i] = i;
    }
    ◇
```

Macro referenced in scraps 51a, 54a.

After we initialize **resource_name**, we iterate through all of the ϕ-nodes in the graph. For each useless ϕ-node we find, we will change its entry in **resource_name** to use the parameter of the ϕ-node.

⟨Search the graph for useless ϕ-nodes and set changed if one is found 52a⟩ ≡

```
{
    Block *block;

    ForAllBlocks_rPostorder(block)
    {
        Phi_Node *phi_node;

        Block_ForAllPhiNodes(phi_node, block)
        {
            Unsigned_Int current_name = phi_node->new_name;
            Unsigned_Int replacement_name = parmcmp(phi_node, resource_name);

            if (replacement_name && resource_name[current_name] != replacement_name)
            {
                resource_name[current_name] = replacement_name;
                changed = TRUE;
                phi_node->useless = TRUE;
                number_removed++;
            }
        }
    }
}
◇
```

Macro referenced in scrap 51a.

We modeled this function after the **strcmp** routine, except that it returns either 0 (meaning the parameters were different), or the parameter number if the parameters are all the same[6].

As we said in the introduction to this section, we do not yet know if our definition of "useless" as it applies to ϕ-nodes is correct. Specifically, if a ϕ-node has a parameter which is zero[7] but all of the other parameters are the same resource name, is that ϕ-node useless? Currently, we have written **parmcmp** to report such a ϕ-node as useless. It is a trivial matter to rewrite this routine to handle the more restrictive case.

⟨Internal function prototypes 52b⟩ ≡

```
static Unsigned_Int parmcmp(Phi_Node *phi_node, Unsigned_Int2 *resource_name);
◇
```

Macro defined by scraps 36a, 37a, 52b, 68b, 69c.
Macro referenced in scrap 5b.

[6] Ignoring parameters numbered zero or the same as the ϕ-node itself, of course.

[7] Remember that a zero in the **parms** array of a ϕ-node indicates that no value is defined along the corresponding edge for that resource.

⟨Internal function bodies 53a⟩ ≡
```
static Unsigned_Int parmcmp(Phi_Node *phi_node, Unsigned_Int2 *resource_name)
{
    Unsigned_Int2 *parm_runner;
    Unsigned_Int running_value = 0;
    Unsigned_Int phi_node_name = phi_node->new_name;

    Phi_Node_ForAllParms(parm_runner, phi_node)
    {
        Unsigned_Int next_parm = resource_name[*parm_runner];

        if (next_parm && next_parm != phi_node_name)
            if (!running_value)
                running_value = next_parm;
            else if (running_value != next_parm)
                return 0;
    }
    return running_value;
} /* parmcmp */
◇
```
Macro defined by scraps 37b, 45b, 53a, 68c, 69d.
Macro referenced in scrap 5b.

Once we have found all of the useless ϕ-nodes, we will make a pass over the entire graph, rewriting all of the resource names with their entries in `resource_name`.

⟨Rewrite the graph using the new names in `resource_name` 53b⟩ ≡
```
{
    Block *block; Inst *inst_runner; Operation **op_runner; Unsigned_Int2 *arg_runner;
    Operation *operation;

    ForAllBlocks(block)
        Block_ForAllInsts(inst_runner, block)
            Inst_ForAllOperations(op_runner, inst_runner)
            {
                operation = *op_runner;
                Operation_ForAllUses(arg_runner, operation)
                    *arg_runner = resource_name[*arg_runner];
                Operation_ForAllDefs(arg_runner, operation)
                    *arg_runner = resource_name[*arg_runner];
                Operation_ForAllRefTags(arg_runner, operation)
                    *arg_runner = resource_name[*arg_runner];
                Operation_ForAllDefTags(arg_runner, operation)
                    *arg_runner = resource_name[*arg_runner];
            }
}
◇
```
Macro referenced in scrap 51a.

2.6.2 Removing useless ϕ-nodes using the worklist method and def-use chains

The worklist method for removing useless ϕ-nodes relies on having the def-use chains. We begin by initializing the worklist to contain all of the ϕ-nodes in the routine. We will represent the worklist using a `SparseSet`, since we will need quick insertion, deletion, and iteration over the set. We remove a ϕ-node from the list and check its parameters. If it is useless, we follow its def-use chains to all of the use-sites, rewriting those uses with the useless ϕ-node's new name. Any use of this ϕ-node in another ϕ-node will cause us to put that second ϕ-node onto the worklist. The algorithm halts when the worklist empties.

Note that the order in which we look at the ϕ-nodes has no effect on the running time of the algorithm.

Also, we will use the same array, `resource_name`, as it was used above. This allows us to reuse the `parmcmp` function. Once initialized, its values will never change, so it will not affect this method.

Since we may not have built the use-def chains, we will need an array to store pointers to all of the ϕ-nodes. The `phi_node_map` array will store these pointers and will be indexed by the resource name.

⟨Remove useless ϕ-nodes using the worklist method 54a⟩ ≡

```
{
    SparseSet worklist = SparseSet_Create(useless_phi_node_memory_bank, SSA_def_count);
    Phi_Node **phi_node_map =
                (Phi_Node **)Arena_GetMem(useless_phi_node_memory_bank,
                                          sizeof(Phi_Node *) * SSA_def_count);
    Unsigned_Int number_removed = 0;
    Unsigned_Int2 *resource_name;

    ⟨Initialize resource_name 51b⟩
    ⟨Initialize phi_node_map and the worklist for removing useless φ-nodes 54b⟩
    while (SparseSet_Size(worklist))
        ⟨Examine the uses of the next φ-node on the worklist 55a⟩

    fprintf(stderr, "Number of dead phi-nodes removed: %d.\n", number_removed);
}
◇
```

Macro referenced in scrap 50a.

We initialize the worklist with all of the ϕ-nodes in the routine.

⟨Initialize `phi_node_map` and the worklist for removing useless ϕ-nodes 54b⟩ ≡

```
{
    Block *block_runner;

    ForAllBlocks(block_runner)
    {
        Phi_Node *phi_node_runner;

        Block_ForAllPhiNodes(phi_node_runner, block_runner)
        {
            Unsigned_Int phi_node_name = phi_node_runner->new_name;

            phi_node_map[phi_node_name] = phi_node_runner;
            SparseSet_Insert(worklist, phi_node_name);
        }
    }
}
◇
```

Macro referenced in scrap 54a.

Each time a node is pulled off the worklist, we will examine its parameters. If they are all the same, we will follow its def-use chains, and, for each operation using the result of this ϕ-node, we will rewrite that use as a use of the ϕ-node's parameter. For each ϕ-node using the current ϕ-node's result, we will rewrite the use and then add that ϕ-node to the worklist.

⟨Examine the uses of the next ϕ-node on the worklist 55a⟩ ≡

```
{
    Unsigned_Int next_phi_node = SparseSet_ChooseMember(worklist);
    Phi_Node *phi_node = phi_node_map[next_phi_node];
    Unsigned_Int replacement_name = parmcmp(phi_node, resource_name);

    SparseSet_Delete(worklist, next_phi_node);
    if (replacement_name)
    {
        Chains_List *use_runner, *last_in_list = NULL;

        ⟨Rewrite uses and add any new φ-nodes to the worklist 55b⟩
        ⟨Move the def-use list 55c⟩
        phi_node->useless = TRUE;
        number_removed++;
    }
}
◇
```

Macro referenced in scrap 54a.

⟨Rewrite uses and add any new ϕ-nodes to the worklist 55b⟩ ≡

```
Chain_ForAllUses(use_runner, next_phi_node)
{
    Chain *next_use = &(use_runner->chain);

    last_in_list = use_runner;
    if (next_use->is_phi_node)
        ⟨Rewrite the parameter and add it to the worklist 56a⟩
    else
        ⟨Rewrite the use 56b⟩
}
◇
```

Macro referenced in scrap 55a.

We will also need to move the list of def-use chains over to the ϕ-node's parameter. We could move each one separately, but this would be more work than necessary. Instead, we will copy the entire list at once. To do this, we will find the last node in the list of chains, reset its **next** pointer to point to the current list of def-use chains for **replacement_name**, and then point to this new head of the list of def-use nodes. We must, of course, remember to set to null the current ϕ-node's entry as well, since failure to do this will cause an infinite loop.

⟨Move the def-use list 55c⟩ ≡

```
if (last_in_list)
{
    last_in_list->next = SSA_def_use_chains[replacement_name];
    SSA_def_use_chains[replacement_name] = SSA_def_use_chains[next_phi_node];
    SSA_def_use_chains[next_phi_node] = NULL;
}
◇
```

Macro referenced in scrap 55a.

The act of changing a ϕ-node's parameter may cause it to become useless, so we add it to the worklist.

⟨Rewrite the parameter and add it to the worklist 56a⟩ ≡

```
{
    Phi_Node *target_phi_node = next_use->op_pointer.phi_node;

    target_phi_node->parms[use_runner->position] = replacement_name;
    SparseSet_Insert(worklist, target_phi_node->new_name);
}
◇
```

Macro referenced in scrap 55b.

For operations, we simply need to overwrite the appropriate argument.

⟨Rewrite the use 56b⟩ ≡

```
next_use->op_pointer.operation->arguments[use_runner->position] = replacement_name;
◇
```

Macro referenced in scrap 55b.

Chapter 3

Inserting Copies for ϕ-nodes

At the end of its processing, an application will often want the routine's original resource names restored. When we restore the original names, we render the ϕ-nodes useless, and we simply erase them. Sometimes, however, we cannot restore the resource names. In these cases, the application must call `SSA_Replace_Phi_Nodes`, which substitutes copy operations for ϕ-nodes to mimic their function.

The user should call `SSA_Replace_Phi_Nodes` if either or both of the following are true:

1. Copies have been consumed, or
2. The original register names are not going to be restored or cannot be restored, for example, because of renumbering inside the global value numbering pass.

We replace a ϕ-node in block b by inserting a copy operation into each of b's predecessors. Since the meaning of a ϕ-node is a mapping of all of the incoming resources to a single resource r, we can see that having a copy in each of the predecessor blocks (each copying that block's resource into r) is equivalent.

Consider the example in Figure 3.1. The left-hand side of the figure shows a fragment of the CFG with the code in SSA form. The right-hand side of the figure shows the same fragment with copies inserted for the ϕ-node. Note that the insertion of copy operations has made the ϕ-node obsolete, so we can discard it.

$r3 \leftarrow \cdots$ `JMP1` Join_Point	$r6 \leftarrow \cdots$ `JMP1` Join_Point	$r3 \leftarrow \cdots$ $r7 \leftarrow r3$ `JMP1` Join_Point	$r6 \leftarrow \cdots$ $r7 \leftarrow r6$ `JMP1` Join_Point
	$r7 \leftarrow \phi(r3, r6)$		$r7 \leftarrow \phi(r3, r6)$
Before		After	

Figure 3.1: The impact of inserting copies for ϕ-nodes

The process of inserting copies for ϕ-nodes will consist of iterating through the blocks in the CFG, inserting a copy for each parameter of each ϕ-node in the predecessor of the block containing the ϕ-node. This process, however, is anything but straightforward – the interaction of resources along different paths in the CFG can lead to subtle errors. To explain these problems, we have split this chapter into four sections. The first section provides the motivation for choosing the order in which we perform the copy insertion. The second section discusses the preparation of each block for copy insertion, and the third section explores the two situations which require a more sophisticated approach than the simple naive insertion of copies. Finally, the fourth section discusses the actual copy insertion algorithm.

To begin, we will show the body of the `SSA_Replace_Phi_Nodes` routine. First, however, we will define a state variable to tell us if this routine has been called. In this way, the `SSA_Restore` routine can know whether it must call this routine before doing its own processing.

⟨Private variables 58a⟩ ≡
```
static Boolean ssa_replaced_phi_nodes = FALSE;
```
◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Externally-visible function bodies 58b⟩ ≡
```
Void SSA_Replace_Phi_Nodes(Void)
{
    Arena replacement_memory_bank = Arena_Create();
    ⟨Declare the data structures for φ-node replacement 69b⟩
    Block *block;

    ⟨Compute live-out information, if necessary 68a⟩

    pushed = SparseSet_Create(replacement_memory_bank, SSA_def_count);
    insert_copies(start_block, replacement_memory_bank, copy_stats);

    Arena_Destroy(replacement_memory_bank);
    ssa_replaced_phi_nodes = TRUE;
} /* SSA_Replace_Phi_Nodes */
```
◇

Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

3.1 Choosing the order of traversal for copy insertion

We will be inserting a copy for each of a ϕ-node's parameters. Each parameter represents a value flowing in from one of the predecessors of the block containing the ϕ-node. This presents two methods for ϕ-node insertion; at each block b encountered during an order-independent iteration over the CFG, we can:

1. Insert copies into block b for all of the ϕ-nodes in b's successors, or
2. Insert copies into b's predecessors for all of the ϕ-nodes in block b.

At first glance, these two seem equivalent. The choice of which to implement seems to rely only on the data structures used to build SSA form. For example, in our implementation, the second of these two methods is faster. In method one, we have to search each successor block's `SSA_edge_maps` array to get the index of the edge between the current block b and that successor. That index, of course, is necessary to know the sources of the copy instructions to be inserted into b. This gives us an asymptotic complexity of $O(E^2)$, where E is the number of edges in the CFG. Method two saves us from having to perform these `SSA_edge_maps` searches. The tradeoff is that it requires that we do a separate pass over the CFG to ensure that we have a valid place to insert copies into each block (for more on this, see Section 3.2).

Method 2 would be faster for us, but it has problems, too. If an earlier pass of the compiler has not yet split critical CFG edges, this can lead to name conflicts in copies. That is, the resource used as the source of one copy can get overwritten by another copy. The situation where this can occur is complex; for a complete explanation, see Section 3.3, starting on page 62.

Even if the compiler splits critical edges, we cannot be guaranteed that this will be sufficient, since we have the ILOC opcode `JMPr`, jump to register. The edges coming from a `JMPr` operation cannot be split, so it is still possible to have name conflicts. Thus, our implementation will have to rely on the first method. We will iterate over the CFG; at each block b, we will insert copy operations for the entire set of b's successors' ϕ-nodes.

But we have already shown that this algorithm will have quadratic complexity. To avoid this, we will build an extra data structure called `parm_map`, which will be an array indexed by the blocks' preorder index. Each element in the array will be a linked list, where each node will contain a pointer to the successor block n and the index into n's `parms` array.

⟨Internal data structures 59a⟩ ≡

```
typedef struct parm_map_node {
    Unsigned_Int index;
    Block *succ_block;
    struct parm_map_node *next;
} Parm_Map_Node;
```

◇

Macro defined by scraps 17d, 32c, 59a, 69a.
Macro referenced in scrap 5b.

⟨Private variables 59b⟩ ≡

```
static Parm_Map_Node **parm_map;
```

◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Initialize SSA variables 59c⟩ ≡

```
parm_map = (Parm_Map_Node **)Arena_GetMemClear(permanent_memory_bank,
                                               sizeof(Parm_Map_Node *) * (block_count+1));
```

◇

Macro defined by scraps 17b, 18ac, 29b, 46b, 59c.
Macro referenced in scrap 15a.

It would be simple to do a separate pass over the CFG to build the `parm_map` lists. However, we already do such a pass during the renaming process, so it is more efficient to simply do it there. The context in which this scrap is executed is as follows: during the renaming walk over the dominator tree, each of a `block`'s successors' ϕ-nodes have their parameters filled in. The code scrap below will execute immediately following the filling in of a successor's ϕ-node parameters. It will ensure that the current successor to `block`, `succ_block`, has some ϕ-nodes, then it will add a node to `block`'s `parm_map` list, recording the pointer to `succ_block` and the current `parms_index` value, which is the index which `block`'s values were inserted into `succ_block`'s ϕ-nodes.

⟨Add a node to `block`'s `parm_map` entry for `succ_block` 59d⟩ ≡

```
if (SSA_phi_nodes[succ_block->preorder_index])
{
    Parm_Map_Node *new_node = (Parm_Map_Node *)Arena_GetMem(permanent_memory_bank,
                                                            sizeof(Parm_Map_Node));
    new_node->index = parms_index;
    new_node->succ_block = succ_block;
    new_node->next = parm_map[block->preorder_index];
    parm_map[block->preorder_index] = new_node;
}
```

◇

Macro referenced in scrap 47a.

3.2 Preparing the CFG for copy insertion

Before we can start the copy insertion, we need to ensure that we have a valid place to insert copies into each block in the CFG. The problem we are trying to avoid involves a multiple-operation instruction occurring as the last instruction in a block. Consider the segment of code in Figure 3.2.

`JMPl` Join_Point | $r3_a \leftarrow r1 + r2$ `JMPl` Join_Point | $r3_b \leftarrow r4 + r5$

$$r3_c \leftarrow \phi(r3_a, r3_b)$$

Figure 3.2: Control-flow example

In this code, we define $r3$ on two different branches, and we show the ϕ-node which results at the join point. The normal method of inserting copies is to insert them just before the last instruction in the block. Doing this would give us the code in Figure 3.3.

$r3_c \leftarrow r3_a$ $r3_c \leftarrow r3_b$

`JMPl` Join_Point | $r3_a \leftarrow r1 + r2$ `JMPl` Join_Point | $r3_b \leftarrow r4 + r5$

$$r3_c \leftarrow \phi(r3_a, r3_b)$$

Figure 3.3: Incorrect code

Clearly, this code is incorrect, since $r3_a$ and $r3_b$ are not yet defined when we copy them into $r3_c$. Thus, if a block ends in an instruction which contains more than one operation, we need to split the instruction before inserting copies for ϕ-nodes, so that the block-ending operation is the only operation in the last instruction in the block. In our example, we will transform the code as shown in Figure 3.4.

$r3_a \leftarrow r1 + r2$ $r3_b \leftarrow r4 + r5$

`JMPl` Join_Point   `JMPl` Join_Point

$$r3_c \leftarrow \phi(r3_a, r3_b)$$

Figure 3.4: Control-flow operation split off

Finally, inserting the copies will produce the code in Figure 3.5. We can see that this code will produce the same results as the original.

It may seem that we could simply insert a new block along the edge between the two blocks and load all of the new copies into it. This works in all cases except when a block ends in a `JMPr` operation, since we have no way of forcing the `JMPr` to jump to our new block instead of its original target. Because of this special case, we have to split the instruction instead.

In order to split operations into their own instructions, we allocate a new `Inst` data structure and move the branch-ending operation into it. This new instruction then gets placed as the last instruction in the

$r3_a \leftarrow r1 + r2$
$r3_c \leftarrow r3_a$
`JMPl` Join_Point

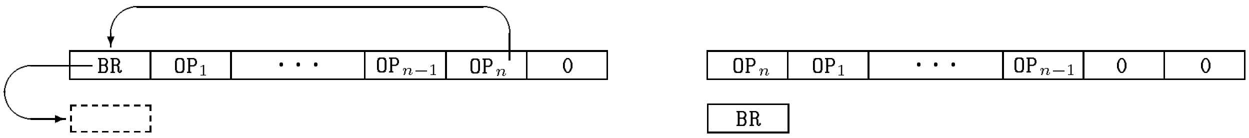
$r3_b \leftarrow r4 + r5$
$r3_c \leftarrow r3_b$
`JMPl` Join_Point

$r3_c \leftarrow \phi(r3_a, r3_b)$

Figure 3.5: Final code with copies inserted

block, maintaining our CFG invariant.

We delete the branch operation from the original instruction by moving the last operation in the instruction into the first position in the `operations` array. Remember that the `operations` array is null-terminated; deleting the operation we just moved, therefore, is a matter of inserting a zero in its place to mark the new end of the instruction's `operations` array. For more on this, consult Figure 3.6.



The Movement of Operations During Separation

The Instructions After Separation

Figure 3.6: The mechanics of operation separation

Since we know that the last entry in an instruction's `operations` array will contain the end-of-list marker, we can decide if an instruction contains more than one operation by looking for the end-of-list marker in the second position of the `operations` array.

⟨Ensure that we have a valid place to insert copies into `block` 62⟩ ≡

```
{
    Inst *last_instruction = block->inst->prev_inst;

    if (last_instruction->operations[1])
    {
        Inst *new_instruction = Inst_Allocate(block_memory_bank, 1);
        Operation **op_runner;
        Int last_operation_position = -1;

        new_instruction->operations[0] = last_instruction->operations[0];
        new_instruction->operations[1] = 0;
        Block_Insert_Instruction(new_instruction, block->inst);

        /* find the last operation in last_instruction */
        Inst_ForAllOperations(op_runner, last_instruction)
            last_operation_position++;

        last_instruction->operations[0] =
                last_instruction->operations[last_operation_position];
        last_instruction->operations[last_operation_position] = 0;
    }
}
◇
```

Macro referenced in scrap 69d.

3.3 Refining the copy-insertion algorithm

Having ensured that we have a reasonable place to put new instructions, we can insert a copy operation at the end of `block` for each of its successors' ϕ-nodes. This is a straightforward instruction insertion for a parameter of each of the successors' ϕ-nodes, subject to the following exceptions:

- If the ϕ-node represents a tag instead of a register
- If the value in the `parms` array is zero
- If the source and destination of the new copy have the same name

If any of the above hold, we will not insert a copy for that ϕ-node.

However, this naive insertion of copies can cause errors if copies were folded[1] and/or if the CFG has not had its critical edges split. Either case can result in the loss of certain necessary holders of values. The rest of this section will elaborate on the pitfalls and provide sketches of the algorithms which ensure that such errors do not occur.

3.3.1 The "Swap Problem"

A complication in the naive algorithm was pointed out to us by Cliff Click. He noticed that when folding copies, subsequent replacement of ϕ-nodes could lead to incorrect code. He and Preston Briggs subsequently reduced the problem to the example shown in Figure 3.7. For obvious reasons, we refer to this as the *Swap Problem*.

On the left side of the figure, we have a simple loop inside of which the values in two variables are swapped using a temporary. In the middle section of the sequence, we have transformed the code into SSA form without absorbing copies. On the right section of the sequence, we have inserted copies for the ϕ-nodes. This example is very simple, and it is easy to show that the code produces the same result.

[1] Note that the same naming problems caused by copy folding can also occur if an optimization pass performs aggressive renumbering, as is done by a value numberer.

$a \leftarrow 1$
$b \leftarrow 2$

$x \leftarrow a$
$a \leftarrow b$
$b \leftarrow x$
if p then

ORIGINAL CODE

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$

$a_2 \leftarrow \phi(a_1, a_3)$
$b_2 \leftarrow \phi(b_1, b_3)$
$x \leftarrow a_2$
$a_3 \leftarrow b_2$
$b_3 \leftarrow x$
if p then

SSA FORM

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1$
$b_2 \leftarrow b_1$

$x \leftarrow a_2$
$a_3 \leftarrow b_2$
$b_3 \leftarrow x$
$a_2 \leftarrow a_3$
$b_2 \leftarrow b_3$
if p then

ϕ-NODES REPLACED

Figure 3.7: ϕ-node-replacement example

Consider, however, the code in Figure 3.8. The left side of the figure shows the SSA form after folding copies. Since all of the operations in the body of the loop were copies, they have all been absorbed, and all that is left in the body are the ϕ-nodes. Compare the parameters to each ϕ-node against the ϕ-nodes in Figure 3.7. They have changed to account for copy absorption.

The right side of Figure 3.8 shows the code after we naively insert copy operations for the ϕ-nodes. This code is clearly incorrect. On the first iteration of the loop, the value of a_2 gets overwritten, and both a_2 and b_2 subsequently contain the same value.

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$

$a_2 \leftarrow \phi(a_1, b_2)$
$b_2 \leftarrow \phi(b_1, a_2)$
if p then

SSA FORM
WITH COPIES FOLDED

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1$
$b_2 \leftarrow b_1$

$a_2 \leftarrow b_2$
$b_2 \leftarrow a_2$
if p then

ϕ-NODES NAIVELY REPLACED

Figure 3.8: Incorrect ϕ-node replacement

The problem stems from the fact that the ϕ-nodes in a block are considered to execute in parallel. One solution to the problem, shown on the left side of Figure 3.9, is to take advantage of ILOC's ability to perform operations in parallel. As shown in the figure, a and b will not overwrite each other if the copies execute in parallel. Unfortunately, most machines do not include this parallel switching of values in their operation sets, so this solution does not seem to be the best one.

A second alternative, as shown on the right side of Figure 3.9, is to insert copies into temporaries for each of the values and to then insert copies from those temporaries into their destinations, as defined by the ϕ-nodes.

Naively inserting copies of values into temporaries, however, is not a feasible solution. It would double the number of copies necessary for ϕ-node replacement. Instead, we will only insert copies for temporaries

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1 \mid b_2 \leftarrow b_1$

$a_2 \leftarrow b_2 \mid b_2 \leftarrow a_2$
`if` p `then`

EXECUTING COPIES IN PARALLEL

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$t_1 \leftarrow a_1$
$t_2 \leftarrow b_1$
$a_2 \leftarrow t_1$
$b_2 \leftarrow t_2$

$t_3 \leftarrow a_2$
$t_4 \leftarrow b_2$
$a_2 \leftarrow t_4$
$b_2 \leftarrow t_3$
`if` p `then`

COPYING INTO TEMPORARIES

Figure 3.9: Possible fixes to our original example

which are necessary for correctness.

Consider again the example in Figure 3.8. The problem is that some of the parameters to the ϕ-nodes are defined by other ϕ-nodes in the same block. Notice that the copies inserted into the top block do not contain references to other resources defined by a ϕ-node. These copies have been inserted correctly – that is, they do not change the meaning of the code. It is only the copy operations inserted for *parameters which were themselves defined by ϕ-nodes in this block* which caused the problem. Thus, if we simply insert copies to temporaries for these special cases, we will get correct code.

This is slightly simplistic, however. Consider the code in Figure 3.10. Here, we do not have a cycle as in the Swap Problem, although the resource a_2 *is* used in a successive ϕ-node in the block. According to the above rule, since the ϕ-node is used as a parameter in another ϕ-node in that block, we should insert a copy to a temporary for it. Simple analysis, though, will show that if we reorder the copies, we will get correct code without the addition of a temporary, as shown in the right side of this figure.

$a \leftarrow 1$
$b \leftarrow 2$

$a_2 \leftarrow \phi(a_1, a_3)$
$b_2 \leftarrow \phi(b_1, a_2)$
`if` p `then`

ORIGINAL SSA FORM

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1$
$b_2 \leftarrow b_1$

$t \leftarrow a_2$
$a_2 \leftarrow a_3$
$b_2 \leftarrow t$
`if` p `then`

UNNECESSARY COPY TO A TEMPORARY

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1$
$b_2 \leftarrow b_1$

$b_2 \leftarrow a_2$
$a_2 \leftarrow a_3$
`if` p `then`

CORRECTLY-SCHEDULED ϕ-NODES

Figure 3.10: Simple ordering example

In some sense, the choice of how to insert copy operations for ϕ-nodes and when to insert copies to temporaries is a scheduling problem. A copy operation has two arguments, the source and the destination. We want to insert copies for a set of ϕ-nodes subject to the following restriction: to schedule a copy c, all other copy operations which include c's destination as their source must be scheduled first. That is, before we overwrite a resource, we must ensure that any other operation which needed its value has gotten it already.

Another way to look at this is to model the interaction of the copies to be inserted as a graph whose nodes represent the copies and whose edges represent a resource defined by one copy and used in another copy. If the graph is acyclic, the schedule of copies can then be found by a simple topological sort of the graph. Of course, we do not actually need to build this graph if we are careful about the data structures we use to build the schedule.

This scheduling will require three passes through the list of ϕ-nodes. In the first pass, we will count the number of times a resource is used by other ϕ-nodes. In the second pass, we will build a worklist of resources which are not used in other ϕ-nodes. The third pass will iterate over the worklist, scheduling a copy for each element in the worklist. Obviously, the copy operations whose destinations are not used by other copy operations can be scheduled immediately. Furthermore, each time we insert a copy operation, the source of that operation can be added to the worklist.

Imagine that we have a resource r which we wish to use as the source in five other copy operations. We said that the rule was that a copy redefining r cannot be inserted until all five of the other copies which use r had been inserted, in order to ensure that all of the copy operations got the same value of r before we overwrite it. But when we insert the first copy of that resource into a new destination d, we have stored r's value where we will not lose it. Each of the remaining four copy operations can use d as their source instead of r, so we are free to overwrite r.

This will ensure that the copy operations are ordered correctly, but it still does not address the Swap Problem. In the Swap Problem, we have a set of copies where each of the destinations is used as a source in another copy in the set, forming a cycle. With the above algorithm, none of the copies would ever be put on the worklist, and we would have a deadlock, much like the analogous situation in resource allocation in an operating system. Just like the resource manager, we can break this deadlock by randomly picking one of the edges in the cycle and breaking it. We do this by inserting a copy to a temporary for one of the destinations. As we pointed out in the previous paragraph, this allows us to put that copy onto the worklist, and, with the cycle broken, the rest of the copies can then be scheduled.

Figure 3.11 shows the code resulting from this algorithm when applied to the Swap Problem. It is interesting to note the similarity to the original code in Figure 3.7.

$a_1 \leftarrow 1$
$b_1 \leftarrow 2$
$a_2 \leftarrow a_1$
$b_2 \leftarrow b_1$

$t_1 \leftarrow a_2$
$a_2 \leftarrow b_2$
$b_2 \leftarrow t_1$
if p then

Figure 3.11: Code with only one copy to a temporary inserted

3.3.2 The "Lost-Copy Problem"

The Swap Problem occurs as a result of copy folding, regardless of whether or not the CFG has had its critical edges split. In contrast, the *Lost-Copy Problem* can only occur when we have folded copies *and* we have not split the critical edges in the CFG.

Consider the code on the left side of Figure 3.12. At each iteration, the loop increments a variable, and the value from the next to last iteration is then printed out. The middle column of code shows the code when translated to SSA form, but without folding copies. The result of inserting copies for the ϕ-nodes in this example is shown on the right side of Figure 3.12. Clearly, the code still produces the same result.

$x \leftarrow 1$
$y \leftarrow x$
$x = x + 1$
`if` p `then`
`RTN` y

Original Code

$x_1 \leftarrow 1$
$x_2 \leftarrow \phi(x_1, x_3)$
$y_1 \leftarrow x_2$
$x_3 = x_2 + 1$
`if` p `then`
`RTN` y_1

Code in SSA Form (Copies Not Folded)

$x_1 \leftarrow 1$
$x_2 \leftarrow x_1$
$x_2 \leftarrow \phi(x_1, x_3)$
$y_1 \leftarrow x_2$
$x_3 \leftarrow x_2 + 1$
$x_2 \leftarrow x_3$
`if` p `then`
`RTN` y_1

Copies Inserted for ϕ-nodes (Correct)

Figure 3.12: An example of the code leading to the Lost-Copy Problem

Now, however, consider the code on the left side of Figure 3.13. Here, we have translated the code to SSA form and folded copies. In the middle section of code, we have replaced the ϕ-node with copies. Clearly, the result of the code has changed; it now prints out the value of the last iteration. In contrast, the code on the right side of the figure shows the copies that would be produced had critical edges been split. This code produces the correct result.

Intuitively, the incorrect code happens because we needed to use a value outside of the scope of the ϕ-node that defined it. That is, when we rewrote uses of the copy's destination with the copy's source during the renumbering phase, we rewrote a use of that resource which lies on a path passing through a redefinition of that source. In some sense, we extended the live range of a value across a redefinition of the resource holding it.

We can prevent this by noticing that the value we are about to overwrite by the new copy is live past the point at which we inserted the copy. We can then insert a copy to a temporary prior to inserting the copy, rewriting subsequent uses of the resource with the temporary, in much the same way that the build algorithm handles copy folding. This is the idea on which we built the subsequent algorithm. This rewriting with new names mimics the name rewriting which we do when we build SSA form, implying that we have to walk the dominator tree when inserting copies. It also means that the implementation will require a stack of names similar to the `name_stacks` array in Chapter 2. Unlike the build process, however, we will only need to push names onto stacks corresponding to the registers defined by the copies we are inserting – these are the only names which will need to have their uses rewritten.

We will use live-out information computed by the data-flow analysis described in Chapter 2 to determine those registers which will require insertion of additional copies to temporaries, and then we will use a structure like the `name_stacks` array to record the temporary names that we have created.

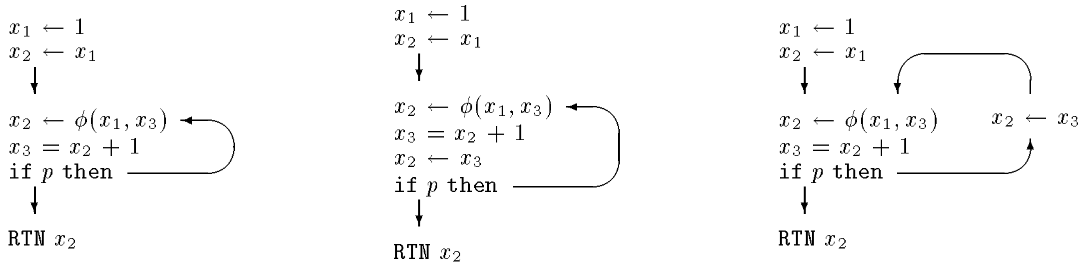
This results in the following algorithm:

At each block in a preorder walk over the dominator tree:

1. Iterate through the block, replacing uses in ϕ-nodes and instructions with any new names previously generated during the walk;

2. Build a list of the copies to be inserted into this block and use the Swap-Problem algorithm outlined in Section 3.3.1 to determine the order to insert the copies;

3. As each copy is inserted, if the copy's source is live outside of the block, push the destination's name onto the source's name stack and reset the flag showing the source is live-out;



Code in SSA Form (Copies Folded)

Copies Inserted for ϕ-nodes (Incorrect)

Copies Inserted When Critical Edges are Split (Correct)

Figure 3.13: The same example, but with copies folded

4. If the destination of the copy to be inserted is live past the end of the block, insert a copy to a temporary at the ϕ-node which defines the register.

Notice that the effort needed to solve the Lost-Copy Problem does not interfere with the algorithm used to solve the Swap Problem. The algorithm used to solve the Swap Problem is a scheduling algorithm. It provides the order in which the copies need to be inserted. The Lost-Copy-Problem algorithm works inside this scheduling algorithm.

Some clever engineering is also required to make this as efficient as possible. For example, imagine a single ϕ-node which has ten parameters and will thus cause the insertion of ten copies. If each of these new copies required a temporary to be inserted before it, we might naively create ten SSA names, one for each temporary. Not only would this result in a potentially large name explosion, but it would also lead to incorrect code. We avoid these problems by only creating a new name the first time a temporary is required for a ϕ-node. We record that name and use it for subsequent temporaries required for the same ϕ-node.

A variation on this solution comes from the observation that even though we have only defined a single name, we still may insert a large number of copies to temporaries. The workaround to this problem is to insert a copy to a temporary (when it is needed) at the top of the block to which the current ϕ-node is attached. This has the practical effect of capturing the value in question immediately after it is defined by the ϕ-node, so that it cannot be overwritten. However, we do not yet have an intuition of how this will affect register pressure. Future studies remain to determine the better of these two methods. Currently, we have implemented the single-copy method.

3.4 Inserting the copies

The solutions to the problems introduced in the last section mesh seamlessly together, since they operate at different times. The solution to the Swap Problem tells us in what order to insert the copies. The solution to the Lost-Copy Problem gets invoked after we have decided to insert a copy. By defining only a few new data structures, we can ensure that ϕ-node replacement will be done correctly and in linear time.

The Lost-Copy Problem can only occur if we have folded copies[2] *and* if the critical CFG edges have not been split. When this is the case, we will need to compute the live-out information for all of the resources in the block. We can leverage off of the liveness analysis already built for ϕ-node insertion.

[2] As we said earlier, certain optimizations can also have this effect; in this case, the easiest solution would be to allow an optimization to tell this routine that it needs to perform the following work.

⟨Compute live-out information, if necessary 68a⟩ ≡
```
if (fold_copies && !cfg_edges_split())
{
    VectorSet *live_in;

    number_of_resources = SSA_def_count;
    data_flow_memory_bank = Arena_Create();
    live_in = (VectorSet *)Arena_GetMem(data_flow_memory_bank,
                                        (block_count+1) * sizeof(VectorSet)) ;
    live_out = (VectorSet *)Arena_GetMem(block_memory_bank,
                                         (block_count+1) * sizeof(VectorSet)) ;
    ⟨Perform liveness analysis, if necessary 20⟩
    Arena_Destroy(data_flow_memory_bank);
}
◇
```
Macro referenced in scrap 58b.

Checking to see if the CFG has had its edges split requires examining the edges on every block. Remember that a critical edge is one which originates at a block with more than one outgoing edge and terminates at a block with more than one incoming edge. The function returns **TRUE** if it finds a critical edge. Because of the nature of the problem which we are solving, we need only look for the subset of unsplit critical edges which are also backedges.

⟨Internal function prototypes 68b⟩ ≡
```
static Boolean cfg_edges_split(Void);
◇
```
Macro defined by scraps 36a, 37a, 52b, 68b, 69c.
Macro referenced in scrap 5b.

⟨Internal function bodies 68c⟩ ≡
```
static Boolean cfg_edges_split(Void)
{
    Block *block; Edge *edge_runner;

    ForAllBlocks(block)
        Block_ForAllSuccs(edge_runner, block)
            if (edge_runner->back_edge)
                if (block->succ->next_succ)
                    return FALSE;

    /* if we fall through the loop, all edges have been split */
    return TRUE;
} /* cfg_edges_split */
◇
```
Macro defined by scraps 37b, 45b, 53a, 68c, 69d.
Macro referenced in scrap 5b.

Remember that the solution to the Swap Problem requires a specialized topological sort consisting of three passes over the set of copies to be inserted into a block. For the first pass of the algorithm, we build up the set of copies (that is, the set of source/destination pairs) and the count of how many times each destination is used as the source in another copy. Also, we will need the type of the register in order to know the opcode to be used. We will declare the **copy_stats** array, which will be indexed by the destination name, to record this information.

We mentioned earlier that once we copy a register $r1$ into another register $r2$, further uses of $r1$ should be replaced with a use of $r2$ in subsequent copies. We will use the **source_name_to_use** field to record this exchange.

We will implement the name stacks necessary to solve the Lost-Copy Problem using the **new_name** field in each resource's entry in the copy_stats array. Further, the first time that we insert a copy to a temporary for a register, we will record the temporary's name in the temp_name field so that we can use the same name for subsequent copy-to-temporaries for the same register.

⟨Internal data structures 69a⟩ ≡
```
typedef struct copy_data {
    Unsigned_Int2 source;
    Unsigned_Int2 destination_type;
    Unsigned_Int2 used_by_count;
    Unsigned_Int2 is_live_out;
    Unsigned_Int2 source_name_to_use;
    Unsigned_Int2 temp_name;
    Name *new_name;
} Copy_Data;
```
◇

Macro defined by scraps 17d, 32c, 59a, 69a.
Macro referenced in scrap 5b.

⟨Declare the data structures for ϕ-node replacement 69b⟩ ≡
```
Copy_Data *copy_stats = Arena_GetMemClear(replacement_memory_bank,
                                          SSA_def_count*sizeof(Copy_Data));
```
◇

Macro referenced in scrap 58b.

Remember that the solution to the Lost-Copy Problem requires us to perform a depth-first walk over the dominator tree, analogous to the walk we performed while building SSA form. As a result, we will need to declare a separate, recursive routine.

⟨Internal function prototypes 69c⟩ ≡
```
static Void insert_copies(Block *block,
                          Arena replacement_memory_bank,
                          Copy_Data *copy_stats);
```
◇

Macro defined by scraps 36a, 37a, 52b, 68b, 69c.
Macro referenced in scrap 5b.

⟨Internal function bodies 69d⟩ ≡
```
static Void insert_copies(Block *block,
                          Arena replacement_memory_bank,
                          Copy_Data *copy_stats)
{
    Unsigned_Int pushed_list = 0;
    ⟨Declare the worklists 72c, ... ⟩

    SparseSet_Clear(pushed);
    Arena_Mark(replacement_memory_bank);
    ⟨Initialize the worklists 72d, ... ⟩
    ⟨Walk through the instructions in the block, replacing uses with new names 70⟩

    ⟨Ensure that we have a valid place to insert copies into block 62⟩
    ⟨Insert copies into block from all of its successors 71c⟩
    ⟨Recur over all of the block's children in the dominator tree 71a⟩
    ⟨Pop any names added during this run 71b⟩
    Arena_Release(replacement_memory_bank);
} /* insert_copies */
```
◇

Macro defined by scraps 37b, 45b, 53a, 68c, 69d.
Macro referenced in scrap 5b.

At each block, the first step is to walk through the instructions in the block, rewriting uses with new names. Again, this is analogous to the procedure used to build SSA form, except that, instead of pushing new names onto the various stacks for every definition, whether from a ϕ-node or an operation, we only push names onto the stacks of those ϕ-nodes which have already had a copy to a temporary inserted into this block (*i.e.*, the **new_name** field is non-zero). This is mostly a convenience; when we rewrite the uses in the block, we would need some special-case code to guard against overwriting those copies to temporaries already inserted. By pushing the ϕ-node's name onto the top of its own stack, we avoid this problem. Obviously, this solution does not affect correctness – after all, whether we use the temporary's name or the ϕ-node's name in the code, up until the point where we insert a copy which overwrites the ϕ-node's value (the temporary never gets overwritten), the values are the same. When we do overwrite that value by the insertion of a copy to the ϕ-node's register, we will either push the temporary's name onto the stack to ensure that subsequent uses get the correct value (in the case of the value being live-out past that point), or it will not matter (because the value is not live-out past that point).

Notice that when we rewrite the ϕ-nodes' parameters, this does not affect the correctness of the copies that we are inserting. That is, we do not have to process the blocks in an order that allows us to ensure that any parameters that are going to need to be rewritten are. This comes about naturally as a consequence of the way we write values onto the name stacks and because we are iterating through the graph in dominator order.

⟨Walk through the instructions in the block, replacing uses with new names 70⟩ ≡

```
{
    Inst *inst_runner; Operation **op_runner; Unsigned_Int2 *arg_runner;
    Phi_Node *phi_node_runner;

    Block_ForAllPhiNodes(phi_node_runner, block)
        Phi_Node_ForAllUpdatedParms(arg_runner, phi_node_runner, block->pred_count)
            if (copy_stats[*arg_runner].new_name)
                *arg_runner = copy_stats[*arg_runner].new_name->new_name;

    Block_ForAllPhiNodes(phi_node_runner, block)
    {
        Unsigned_Int2 name = phi_node_runner->new_name;

        /* only push onto the stack if the value is live outside of
           the current block */
        if (live_out && VectorSet_Member(live_out[block->preorder_index], name))
            push_name(&(copy_stats[name].new_name),
                                 replacement_memory_bank, pushed,
                                 &pushed_list, name, name);
    }

    Block_ForAllInsts(inst_runner, block)
        Inst_ForAllOperations(op_runner, inst_runner)
            Operation_ForAllUses(arg_runner, *op_runner)
                if (copy_stats[*arg_runner].new_name)
                    *arg_runner = copy_stats[*arg_runner].new_name->new_name;
}
◇
```

Macro referenced in scrap 69d.

After we insert the copies into this block, we need to recur over the current block's children in the dominator tree. Following this, we need to pop the names off of the name stacks that were pushed on for this block. We have put these two scraps here to continue expressing the symmetry between this algorithm and the SSA building routine. For more information, consult Section 2.5.6.

⟨Recur over all of the block's children in the dominator tree 71a⟩ ≡

```
{
    Block_List_Node *list_runner;
    Dominator_ForChildren(list_runner, block->dom_node)
        insert_copies(list_runner->block, replacement_memory_bank, copy_stats);
}
◇
```

Macro referenced in scrap 69d.

⟨Pop any names added during this run 71b⟩ ≡

```
while (pushed_list)
{
    Unsigned_Int next_node = copy_stats[pushed_list].new_name->next_pushed;

    copy_stats[pushed_list].new_name = copy_stats[pushed_list].new_name->next;
    pushed_list = next_node;
}
◇
```

Macro referenced in scrap 69d.

As we said in Section 3.3, the modified topological sort we will use to schedule the insertion of the copies will require three steps. In the first step, we build up the list of copies to be inserted and some statistics about the registers used in the copies. The second step consists of initializing the worklist with those copies which can be scheduled immediately. Finally, the third step empties the worklist.

⟨Insert copies into `block` from all of its successors 71c⟩ ≡

⟨Phi-node replacement pass 1: initialize the data structures 72a⟩
⟨Phi-node replacement pass 2: set up the worklist for inserting copies for ϕ-nodes 73d⟩
⟨Phi-node replacement pass 3: iterate over the worklist, inserting copies 74a⟩
◇

Macro referenced in scrap 69d.

In the first pass over the successors' ϕ-nodes, we need to count the number of times each resource will be used in another copy, and we also need to build up the list of copy operations which will be inserted. For each copy, we record information in `copy_stats` for both the destination and the source. This will be the only pass over the ϕ-nodes themselves; future passes will be over the information built up in the `copy_stats` array.

Also, we will only insert a copy for a ϕ-node if it represents a register, if the source is not equal to zero (which represents a path down which no value flows), and if the source and destination names are different.

We will insert copies into the current block for each of its successors. We do not have to check that a successor actually has ϕ-nodes – only those successors with ϕ-nodes will be in the block's `parm_map` list. If none of a block's successors have ϕ-nodes, its `parm_map` entry will be `NULL`. Remember that the parameters to the ϕ-nodes are aligned such that the parameters in the first index in the arrays correspond to the set of values flowing along a single edge into the block, the set of parameters in the second element of a block's ϕ-nodes represent the values flowing along another incoming edge, *etc.*

⟨Phi-node replacement pass 1: initialize the data structures 72a⟩ ≡

```
{
    Parm_Map_Node *next_node = parm_map[block->preorder_index];

    while (next_node)
    {
        Unsigned_Int index = next_node->index;
        Block *succ_block = next_node->succ_block;
        Phi_Node *phi_node_runner;

        Block_ForAllPhiNodes(phi_node_runner, succ_block)
            if (⟨The current φ-node represents a register 72b⟩)
            {
                Unsigned_Int source_name = phi_node_runner->parms[index];
                Unsigned_Int destination_name = phi_node_runner->new_name;

                if (copy_stats[source_name].new_name)
                    source_name = copy_stats[source_name].new_name->new_name;

                if (source_name && destination_name != source_name)
                    ⟨Add the statistics for this copy into the copy_stats array 73a⟩
            }
        next_node = next_node->next;
    }
}
◇
```

Macro referenced in scrap 71c.

We can tell if the current ϕ-node is a register by comparing its `old_name` field to the global variable `register_count`. If it is less than `register_count`, then the ϕ-node represents a register.

⟨The current ϕ-node represents a register 72b⟩ ≡

```
phi_node_runner->old_name < register_count
◇
```

Macro referenced in scrap 72a.

We also need to keep track of the destinations in the set of copies to be inserted. When we insert a copy, we will remove the destination from this list. In the case of a cycle, none of the members will be on the worklist, but they will be in this list. In this way, we will know which copies have not yet been scheduled.

⟨Declare the worklists 72c⟩ ≡

```
SparseSet destination_list;
◇
```

Macro defined by scraps 72c, 73b.
Macro referenced in scrap 69d.

⟨Initialize the worklists 72d⟩ ≡

```
destination_list = SparseSet_Create(replacement_memory_bank, SSA_def_count);
◇
```

Macro defined by scraps 72d, 73c.
Macro referenced in scrap 69d.

When we find a ϕ-node that requires us to insert a copy, we gather information related to the new copy. This information includes not only the source and destination names of the copy, but also its `live_out` information, and its relation to other copies to be added (in its `used_by_count` field). Further, we take this opportunity to initialize the other fields in the `copy_stats` array for both the source and destination. Finally, we add the copy's destination name to the `destination_list`.

⟨Add the statistics for this copy into the `copy_stats` array 73a⟩ ≡

```
{
    copy_stats[destination_name].source = source_name;
    copy_stats[destination_name].destination_type = phi_node_runner->def_type;
    copy_stats[destination_name].is_live_out = (live_out) ?
                VectorSet_Member(live_out[block->preorder_index], destination_name)
                : FALSE;
    copy_stats[destination_name].source_name_to_use = destination_name;
    copy_stats[source_name].source_name_to_use = source_name;
    copy_stats[source_name].used_by_count++;
    SparseSet_Insert(destination_list, destination_name);
}
◇
```

Macro referenced in scrap 72a.

The third step of the algorithm requires a worklist, which we will implement using a `SparseSet`.

⟨Declare the worklists 73b⟩ ≡

```
SparseSet worklist;
◇
```

Macro defined by scraps 72c, 73b.
Macro referenced in scrap 69d.

⟨Initialize the worklists 73c⟩ ≡

```
worklist = SparseSet_Create(replacement_memory_bank, SSA_def_count);
◇
```

Macro defined by scraps 72d, 73c.
Macro referenced in scrap 69d.

Once we have initialized the `copy_stats` data structure, we will make a second pass through the candidate copy operations to find those whose `used_by_count` field is equal to zero (*i.e.,* those copy operations whose destinations are not the sources of other copy operations). These are the initial values put onto the worklist, since they can be inserted safely at any time.

⟨Phi-node replacement pass 2: set up the worklist for inserting copies for ϕ-nodes 73d⟩ ≡

```
{
    Unsigned_Int destination_runner;

    SparseSet_ForAll(destination_runner, destination_list)
        if (copy_stats[destination_runner].used_by_count == 0)
            SparseSet_Insert(worklist, destination_runner);
}
◇
```

Macro referenced in scrap 71c.

Once we have set up the `copy_stats` array and initialized the worklist, we can iterate until the worklist empties, inserting copy operations for each element on the worklist. However, emptying the worklist does not mean that we have inserted all of the necessary copies. If we find a set of ϕ-nodes whose source/destination dependencies form a cycle (*a.k.a.* the Swap Problem), we will need to break the cycle as we described earlier and iterate over the worklist again. We detect a cycle by keeping the `destination_list` set. Each time we take an element off of the worklist, we also remove that element from the `destination_list` set. If the worklist empties and there are still entries in the **destination_list** set, we know that we have one or more cycles, and we try to break them by copying to a temporary. Since there can be more than one cycle, we will perform these two steps until the **destination_list** is empty.

⟨Phi-node replacement pass 3: iterate over the worklist, inserting copies 74a⟩ ≡

```
{
    Unsigned_Int destination_list_size;

    do
    {
        while (SparseSet_Size(worklist))
            ⟨Choose a copy off of the worklist, insert it, and remove it from destination_list 74b⟩
        destination_list_size = SparseSet_Size(destination_list);
        if (destination_list_size)
            ⟨Insert a copy to a temporary to break the cycle 76⟩
    } while(destination_list_size);
}
◇
```

Macro referenced in scrap 71c.

We will insert a copy for each element on the worklist. Each time we insert a copy, we can set the source's **used_by_count** field to zero, since we now have a copy of it. We must also rewrite the source's **new_name** field to represent the fact that subsequent references to the source should be replaced by the name of the resource into which we just copied. With this done, the source can be added to the worklist, if necessary.

We only push onto the name stack for an inserted copy if the source is going to be overwritten by a copy in this block and if that source is in the **live_out** set for the block. Also, after we insert the copy, we have a register holding the value, so we can set the source's **is_live_out** entry to **FALSE**.

⟨Choose a copy off of the worklist, insert it, and remove it from destination_list 74b⟩ ≡

```
{
    Unsigned_Int destination_name = SparseSet_ChooseMember(worklist);
    Unsigned_Int source_name = copy_stats[destination_name].source;
    Inst *new_instruction = Inst_Allocate(block_memory_bank, 1);

    SparseSet_Delete(worklist, destination_name);
    if (copy_stats[destination_name].is_live_out && !copy_stats[destination_name].temp_name)
    {
        copy_stats[destination_name].is_live_out = FALSE;
        ⟨Insert a copy to a temporary to store the old value 75b⟩
    }
    ⟨Insert the copy into block 77b⟩
    copy_stats[source_name].used_by_count = 0;
    if (SparseSet_Member(destination_list, source_name) &&
                                            copy_stats[source_name].is_live_out)
        push_name(&(copy_stats[source_name].new_name),
                                    replacement_memory_bank, pushed,
                                    &pushed_list, source_name, destination_name);
    copy_stats[source_name].is_live_out = FALSE;
    copy_stats[source_name].source_name_to_use = destination_name;

    if (SparseSet_Member(destination_list, source_name))
        SparseSet_Insert(worklist, source_name);
    SparseSet_Delete(destination_list, destination_name);
}
◇
```

Macro referenced in scrap 74a.

If the destination of the copy is live outside the block, we will insert a copy to a temporary to store the value in the register prior to overwriting it. To generalize our scraps, we will introduce a new variable, `insert_at_phi_node`, which will determine where the copy gets inserted. As we said in Section 3.3.2, when we insert a copy to a temporary, we will put that operation immediately following the ϕ-node that defines the source of that copy. This flag will will tell a subsequent scrap where the copy goes.

⟨Private variables 75a⟩ ≡

```
static Boolean insert_at_phi_node = FALSE;
◇
```

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

Inserting a copy to a temporary is the same as inserting any other copy. Notice, however, the name changes that we institute so that we can again reuse scraps.

⟨Insert a copy to a temporary to store the old value 75b⟩ ≡

```
{
    Unsigned_Int source_name = destination_name;
    Unsigned_Int destination_name = (copy_stats[source_name].temp_name) ?
                copy_stats[source_name].temp_name : new_name_counter++;

    copy_stats[destination_name].destination_type = copy_stats[source_name].destination_type;
    copy_stats[source_name].used_by_count = 0;
    if (!copy_stats[source_name].temp_name)
    {
        insert_at_phi_node = TRUE;
        ⟨Insert a copy for the temp 77a⟩
    }
    copy_stats[source_name].temp_name = destination_name;
    copy_stats[source_name].source_name_to_use = destination_name;
    push_name(&(copy_stats[source_name].new_name),
                                    replacement_memory_bank, pushed,
                                    &pushed_list, source_name, destination_name);
}
◇
```

Macro referenced in scrap 74b.

When we find a cycle, we will have to insert a copy of one of the involved resources to a temporary variable. Since this clears that resource's `used_by_count` field, it can be put onto the worklist, and we will subsequently be able to insert copies for all of the resources in the cycle.

One problem remains. These temporaries will need new names. This means that we must know in advance how many of these copies to temporaries will be needed, since we will want to store them in our SSA tables. At this point in the routine, these tables have already been allocated based on the size of `SSA_def_count`. Therefore, we need some way to decide how many of these extra copies are going to be needed and increment `SSA_def_count` *before* any of the tables get allocated.

Remember that these problems were caused by folding copy operations – the act of folding copies "dirtied" the ϕ-node parameters or lengthened the live range of a value. But the only ϕ-node parameters or subsequent uses of a ϕ-node which require temporaries are the very parameters and uses rewritten as a result of copy folding. Therefore, the upper bound of copies to temporaries which will need to be inserted is equal to the number of copies in the original ILOC. To provide room for these new temporaries, we will treat copies as any other operation and increment `SSA_def_count` each time we see one during the find-assignments phase. When we are folding copies, this is overly conservative; we will probably not need that many copies to temporaries, so the SSA tables will be allocated slightly too large. We believe that the amount of unused memory, however, should be relatively small.

Notice that when choosing a name for a temporary, we increment the private variable `new_name_counter`, not the public variable `SSA_def_count`. Remember that `SSA_def_count` is the total number of definitions in the original code and, thus, the highest number assigned as a name to any resource when the code is transformed into SSA form. When we are folding copies, it represents not the actual highest resource name created, but, rather, the *upper bound* of the SSA namespace. Since this code *must* know what the last SSA name created was, we keep the private variable `new_name_counter`, which is always equal to the next name to be created.

If a copy to a temporary has already been put in for this register, then we do not have to add another one. We can tell if this is true by examining `copy_stats[source_name].temp_name`. It gets initialized to zero when the `copy_stats` array is created, and if a copy to a temporary has already been inserted into the code, then it will contain the name of the temporary instead of zero.

⟨Insert a copy to a temporary to break the cycle 76⟩ ≡

```
{
    Unsigned_Int source_name = SparseSet_ChooseMember(destination_list);
    Unsigned_Int destination_name;

    if (!copy_stats[source_name].temp_name)
    {
        destination_name = new_name_counter++;
        copy_stats[destination_name].destination_type =
                                        copy_stats[source_name].destination_type;
        copy_stats[source_name].temp_name = destination_name;
        insert_at_phi_node = TRUE;
        ⟨Insert a copy for the temp 77a⟩
    }
    else
        destination_name = copy_stats[source_name].temp_name;

    copy_stats[source_name].used_by_count = 0;
    copy_stats[source_name].is_live_out = FALSE; /* we've now got something
                                                    holding its value, so it's
                                                    not live out anymore */
    copy_stats[source_name].source_name_to_use = destination_name;
    push_name(&(copy_stats[source_name].new_name),
                                        replacement_memory_bank, pushed,
                                        &pushed_list, source_name, destination_name);
    SparseSet_Insert(worklist, source_name);
}
◇
```

Macro referenced in scrap 74a.

Finally, the remainder of this chapter will show the bookkeeping code necessary to create and insert new instructions into the CFG.

Notice that we leverage off of the following copy-insertion code when inserting copies to temporaries. This forces us to be careful with our variable names. Also, notice that we do not write the temporary name back into the ϕ-node's `parms` list. The ϕ-nodes are useless after this routine finishes, so this should not be a problem. If some application required the ϕ-nodes to be kept up to date, it would be simple matter to record the necessary information in the `Parm_Map_Node` data structure.

⟨Insert a copy for the temp 77a⟩ ≡

```
{
    Inst *new_instruction = Inst_Allocate(block_memory_bank, 1);

    ⟨Insert the copy into block 77b⟩
    SSA_name_map[destination_name] = SSA_name_map[source_name];
    SSA_block_map[destination_name] = block;
    SSA_inst_map[destination_name] = new_instruction;
    ⟨Add the use-def chain, if necessary 78c⟩
}
◇
```

Macro referenced in scraps 75b, 76.

Inserting a new operation into the CFG requires that we allocate an `Inst` data structure and an `Operation` data structure. We then fill in the fields of each and insert the new instruction into `block`. Since the `Inst` data structure can hold an arbitrary number of operations, the last element in its `operations` array is set to Λ to mark the end of the list of operations.

⟨Insert the copy into `block` 77b⟩ ≡

```
{
    Operation *new_operation = Operation_Allocate(block_memory_bank, 2);

    ⟨Fill in the fields of new_operation 77c⟩

    new_instruction->operations[0] = new_operation;
    new_instruction->operations[1] = NULL;

    ⟨Insert new_instruction into block 78b⟩
}
◇
```

Macro referenced in scraps 74b, 77a.

Once we have allocated memory to hold the new copy operation, we need to fill in the operation fields.

⟨Fill in the fields of `new_operation` 77c⟩ ≡

```
{
    ⟨Assign to new_operation→opcode the correct type of copy opcode 78a⟩

    new_operation->comment = 0;
    new_operation->source_line_ref = Expr_Install_String("0");

    new_operation->constants = 0;
    new_operation->referenced = 1;
    new_operation->defined = 2;

    new_operation->critical = TRUE;

    new_operation->arguments[0] = copy_stats[source_name].source_name_to_use;
    new_operation->arguments[1] = destination_name;
}
◇
```

Macro referenced in scrap 77b.

We can find the correct type for the new copy operation by checking the `destination_type` field in the `copy_stats` array.

⟨Assign to `new_operation→opcode` the correct type of copy opcode 78a⟩ ≡
```
switch(copy_stats[destination_name].destination_type)
{
    case INT_DEF     : new_operation->opcode = i2i; break;
    case FLOAT_DEF   : new_operation->opcode = f2f; break;
    case DOUBLE_DEF  : new_operation->opcode = d2d; break;
    case COMPLEX_DEF : new_operation->opcode = c2c; break;
    case DCOMPLEX_DEF: new_operation->opcode = q2q; break;
    default:
        fprintf(stderr, "Fatal error trying to determine the type of a phi-node.\n");
        ABORT;
        break;
}
◇
```
Macro referenced in scrap 77c.

Having inserted `new_operation` into the `operations` array of `new_instruction`, the next step is to insert `new_instruction` into the CFG by making it the next-to-last instruction in `block` (remember that the *last* instruction has to be the block-ending (*i.e.,* control flow) instruction).

⟨Insert `new_instruction` into `block` 78b⟩ ≡
```
if (insert_at_phi_node)
{
    Block_Insert_Instruction(new_instruction,
                             SSA_block_map[source_name]->inst->next_inst->next_inst);
    insert_at_phi_node = FALSE;
}
else
    Block_Insert_Instruction(new_instruction, block->inst->prev_inst);

◇
```
Macro referenced in scrap 77b.

We need to add the use-def chain, if we are building that array. Notice that we do not have similar code to build the def-use chain. We did not include this code because it is fairly bulky, and we do not see a need for it in our suite of applications.

⟨Add the use-def chain, if necessary 78c⟩ ≡
```
if (build_use_def_chains)
{
   SSA_use_def_chains[destination_name].is_phi_node = FALSE;
   SSA_use_def_chains[destination_name].op_pointer.operation = new_instruction->operations[0];
}
◇
```
Macro referenced in scrap 77a.

Chapter 4

Restoring the CFG

It is a straightforward process to restore the CFG. We simply read through the operations in the CFG (in any order), and overwrite the names of resources with their old names, which are found in the `SSA_name_map` table.

⟨Externally-visible function bodies 79⟩ ≡

```
Void SSA_Restore(Void)
{
    Block *block_runner; Inst *inst_runner; Operation **op_runner; Unsigned_Int2 *arg_runner;
    Operation *operation;

    if (fold_copies && !ssa_replaced_phi_nodes)
        SSA_Replace_Phi_Nodes();

    ForAllBlocks(block_runner)
        Block_ForAllInsts(inst_runner, block_runner)
            Inst_ForAllOperations(op_runner, inst_runner)
            {
                operation = *op_runner;
                if (!ssa_replaced_phi_nodes)
                {
                    Operation_ForAllUses(arg_runner, operation)
                        ⟨Rewrite register argument with old name 80a⟩
                    Operation_ForAllDefs(arg_runner, operation)
                        ⟨Rewrite register argument with old name 80a⟩
                }
                Operation_ForAllRefTags(arg_runner, operation)
                    ⟨Rewrite tag argument with old name 80b⟩
                Operation_ForAllDefTags(arg_runner, operation)
                    ⟨Rewrite tag argument with old name 80b⟩
            }
    Arena_Destroy(permanent_memory_bank);
    SSA_CFG_renumbered = FALSE;
} /* SSA_Restore */
◇
```

Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

4.1 How to create instructions while the code is in SSA form

We pulled the rewriting code into its own section because we must address a serious issue, namely, how does this module "restore" instructions created after SSA form was built? The problem is that if the new instruction defines a resource, that resource must have a unique name to satisfy the SSA invariant. But the usual method of restoring the CFG consists of simply looking up a resource in the **SSA_name_map** table to get the old name. Of course, any resource with a name higher than **SSA_def_count** will not be stored in this table.

To enable this code to work around the problem, an optimizer pass must follow this rule for new instructions: *defined resources of new instructions should be numbered starting at* SSA_def_count.

The **SSA_Restore** routine will rewrite each of these registers with an "old" name defined by taking the difference between the SSA name and **SSA_def_count** and adding that value to **register_count**. This effectively means that each new SSA name created after building SSA form is assigned a unique "old" name starting at **register_count**. Since all of the original register names are less than **register_count**, this will ensure that all created registers will not interfere with existing resource names.

⟨Rewrite register argument with old name 80a⟩ ≡

```
{
    Unsigned_Int argument = *arg_runner; /* saves derefs */

    if (argument < SSA_def_count)
        *arg_runner = SSA_name_map[argument];
    else
        *arg_runner = register_count + (argument - SSA_def_count);
}
◇
```

Macro referenced in scrap 79.

The case of tags is much more complex because of our implementation of tags as handles. We are still discussing how to handle creation of tags while in SSA form, and so, for now, we simply write out an error message and halt if we encounter an SSA name which represents a tag and is higher than **SSA_def_count**.

⟨Rewrite tag argument with old name 80b⟩ ≡

```
{
    Unsigned_Int argument = *arg_runner; /* saves derefs */

    if (argument < SSA_def_count)
        *arg_runner = SSA_name_map[argument];
    else
    {
        fprintf(stderr, "Creation of tags while in SSA form is currently not allowed.\n");
        ABORT;
    }
}
◇
```

Macro referenced in scrap 79.

Appendix A

Memory Use Diagram

Figure A.1 shows the lifetimes of all of the variables which get created from our arenas during execution of this code.

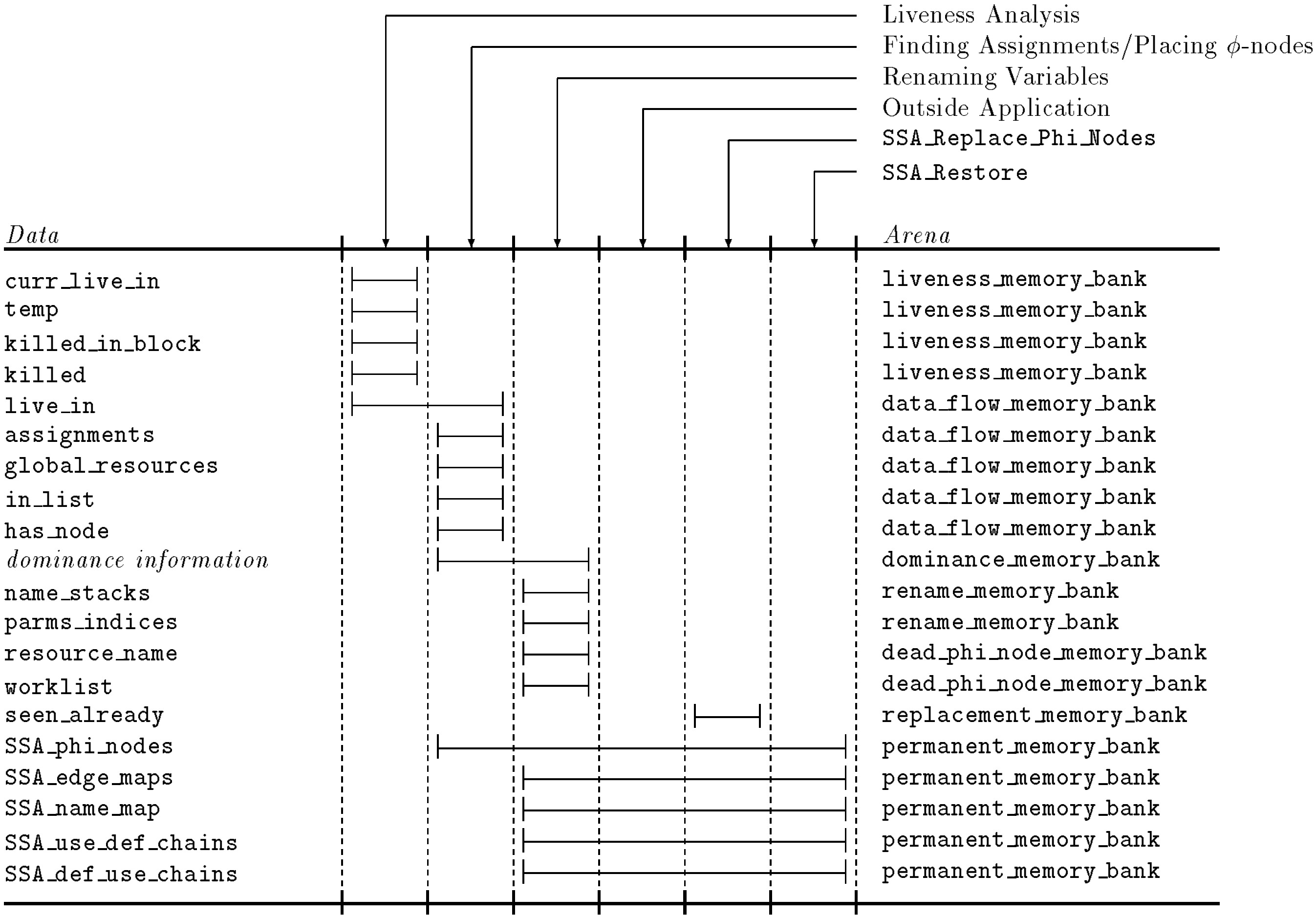


Figure A.1: Memory-use diagram for the SSA module

Appendix B

Implementation Hints

This code was developed within the framework of an experimental compiler whose value lies not only in the quality and usability of its code, but also in its clarity of presentation. As a result, we have sacrificed some efficiency for either breadth of application or greater clarity of explanation.

Obvious examples of this are the multiple checks of which flavor of SSA is being built. In a compiler driven by other concerns, one would tailor the SSA code for each application (perhaps by using preprocessor directives to control conditional compilation).

We show multiple passes over the CFG during the finding-assignments phase. We very informally consider passes over the CFG as being low-cost ("informally", because we have never done any real analysis to determine how expensive they really are). However, some of the separate passes during the finding-assignments phase can be merged to save time.

Copy insertion can take place during each iteration of `rename_resources`. We chose to do it in a separate pass over the CFG because it is somewhat complicated to explain. Instead of inserting copies into a block's predecessors as we show, one would insert copies into the current block for each of its successors. Of course, this requires that, instead of renaming a block's ϕ-nodes as the first step in `rename_resources`, a pass be made over all of the ϕ-nodes before `rename_resources` is called the first time (perhaps when the ϕ-nodes are created, for example) to assign new names to *all* of the ϕ-nodes in the entire CFG. This is necessary to ensure that the copies we insert contain references to the correct resources.

There is a potential pitfall in our implementation of the global arrays which hold block-specific information. Remember that these arrays are indexed by a block's `preorder_index` value. These values, of course, are sequential and unique, but they are not necessarily static. Certain optimization passes which add or delete blocks from the CFG can cause these fields to take on new values, rendering the SSA arrays meaningless. Currently, we have no optimizations with this problem. Future optimizations which modify the blocks' `preorder_index` fields must save the original `preorder_index` values in each block's `block_extension` structure.

Appendix C

Debugging Routines

This appendix section will show the debugging routines we used to help debug this code, as well as ones which are useful when debugging other routines which use this code. We do not pay very much attention to documenting these routines; they were simply useful at one time and may be again.

C.1 Debugging-function prototypes

The `print_phi_nodes` routine prints all of the ϕ-nodes hanging off of a specific block. The other two printing routines print *all* of the different kinds of chains in the graph.

⟨Externally-visible function prototypes 83a⟩ ≡

```
Void print_phi_nodes(Block *block);
```

◇

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

⟨Externally-visible function prototypes 83b⟩ ≡

```
Void print_def_use_chains(Void);
```

◇

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

⟨Externally-visible function prototypes 83c⟩ ≡

```
Void print_use_def_chains(Void);
```

◇

Macro defined by scraps 6d, 8bc, 11d, 12a, 83abc.
Macro referenced in scrap 5a.

C.2 Debugging-function bodies

⟨Private variables 83d⟩ ≡

```
static char *type_strings[] = {"NO_DEFS", "INT_DEF", "FLOAT_DEF", "DOUBLE_DEF",
                               "COMPLEX_DEF", "DCOMPLEX_DEF", "MULT_DEFS"};
```

◇

Macro defined by scraps 15bc, 16a, 17c, 22c, 33a, 35d, 46c, 58a, 59b, 75a, 83d.
Macro referenced in scrap 5b.

⟨Externally-visible function bodies 84a⟩ ≡

```
Void print_phi_nodes(Block *block)
{
    Phi_Node *phi_node_runner;

    fprintf(stderr, "*******PHI_NODES:\n");
    Block_ForAllPhiNodes(phi_node_runner, block)
    {
        fprintf(stderr, "\tNew_Name = %d", phi_node_runner->new_name);
        fprintf(stderr, "\tOld_Name = %d", phi_node_runner->old_name);
        ⟨Print the parms list 84b⟩
        fprintf(stderr, "\t\t********\n");
    }
} /* print_phi_nodes */
◇
```

Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

If the `parms` list is too long, we will want to print it on its own line.

⟨Print the parms list 84b⟩ ≡

```
if (block->pred_count > 10)
    fprintf(stderr, "\n\t");
else
    fprintf(stderr, "\t");

fprintf(stderr, "\tParms: [ ");
{
    Unsigned_Int i;
    for (i=0; i<block->pred_count; i++)
        fprintf(stderr, "%d ", phi_node_runner->parms[i]);
}
fprintf(stderr, "]\n");
◇
```

Macro referenced in scrap 84a.

⟨Externally-visible function bodies 85a⟩ ≡

```
Void print_def_use_chains(Void)
{
    Unsigned_Int i;

    fprintf(stderr, "\n\n PRINTING DEF-USE CHAINS\n\n");
    for(i=0; i < SSA_def_count; i++)
    {
        Chains_List *runner = SSA_def_use_chains[i];

        fprintf(stderr, "*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*\n");
        fprintf(stderr, "RESOURCE NAME: %d\n", i);
        while(runner)
        {
            Chain *next_chain = &(runner->chain);

            fprintf(stderr, "----------\n");
            ⟨Print next_chain 86⟩
            fprintf(stderr, "\tPosition: %d\n", runner->position);
            runner = runner->next;
        }
        fprintf(stderr, "*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*\n");
    }
    fprintf(stderr, "\n\n***DONE*** PRINTING DEF-USE CHAINS\n\n");
} /* print_def_use_chains */
◇
```

Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

⟨Externally-visible function bodies 85b⟩ ≡

```
Void print_use_def_chains(Void)
{
    Unsigned_Int i;

    fprintf(stderr, "\n\nPRINTING USE-DEF CHAINS\n\n");
    for(i=1; i < SSA_def_count; i++)
    {
        Chain *next_chain = &SSA_use_def_chains[i];
        Unsigned_Int resource_name = i;

        fprintf(stderr, "---------------------------------------------\n");
        fprintf(stderr, "Resource Name: %d", i);
        fprintf(stderr, "\tOriginal Name: %d\n", SSA_name_map[i]);
        fprintf(stderr, "\tBlock Name: %s\n",
                        Expr_Get_String(SSA_block_map[i]->labels->label));
        ⟨Print next_chain 86⟩
        fprintf(stderr, "---------------------------------------------\n");
    }
    fprintf(stderr, "\n\n***DONE*** PRINTING USE-DEF CHAINS\n\n");
} /* print_use_def_chains */
◇
```

Macro defined by scraps 15a, 58b, 79, 84a, 85ab.
Macro referenced in scrap 5b.

⟨Print next_chain 86⟩ ≡

```
fprintf(stderr, "\tIs_Phi_Node: ");
if (next_chain->is_phi_node)
{
    Unsigned_Int j;
    Phi_Node *phi_node = next_chain->op_pointer.phi_node;

    fprintf(stderr, "TRUE\n");
    fprintf(stderr, "\tPhi Node Parms(old_name):\n\t\t");
    for(j=0; j < SSA_block_map[phi_node->new_name]->pred_count; j++)
    {
        Unsigned_Int parm = phi_node->parms[j];

        fprintf(stderr, "%d(%d)   ", parm, (parm) ? SSA_name_map[parm] : 0);
    }
    fprintf(stderr, "\n");
}
else
{
    fprintf(stderr, "FALSE\n");
    fprintf(stderr, "\tOpcode = %s\n",
            opcode_specs[next_chain->op_pointer.operation->opcode].opcode);
}
◇
```

Macro referenced in scraps 85ab.

Appendix D

Indices

D.1 Index of File Names

D.2 Index of Macro Names

D.3 Index of Identifiers

Bibliography

[1] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994. Preliminary version available via anonymous ftp.

[2] Preston Briggs and Tim Harvey. Computing dominators and dominance frontiers. Technical report, Rice University, June 1994. Available via anonymous ftp.

[3] 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.

[4] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global flow problems. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 207–217, Boston, Massachusetts, October 1973.