Elimination of Partial Redundancies

Written by *Preston Briggs*
Rewritten and extended by *Rob Shillingsburg*

October 12, 1993

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

Contents

List of Figures

Chapter 1

Introduction

This is **partial**, an optimizer pass for the Massively Scalar Compiler. It consumes and produces ILOC, performing partial redundancy elimination to remove redundant and partially redundant computations. Partial redundancy elimination is described in [2] and [1].

Partially redundant computations are computations which are redundant along some execution paths and not redundant along other paths. For example, in the fragment in Figure 1.1 the computation in the join block is partially redundant, because that computation is redundant along the left branch but not redundant along the right branch. We can improve this code by moving the computation from the join block into the right side of the branch. The result of this transformation is that while the number of instructions executed along the right branch is unchanged, the left branch executes one less instruction than before.

Another application of partial redundancy elimination is the hoisting of loop-invariant code. The fragment in Figure 1.1 contains a loop-invariant computation of `r30`. The computation of `r30` is partially redundant: it is redundant along the loop's back edge, but not redundant along the edge from `enter`. We can improve this code by moving the computation of `r30` from the loop into the "landing pad" block.

These are simple examples of the transformations performed by **partial**. In general, **partial** moves code upward if at least one execution path profits from the move and no path suffers from it. In addition, **partial** also removes operations which are redundant within a single block.

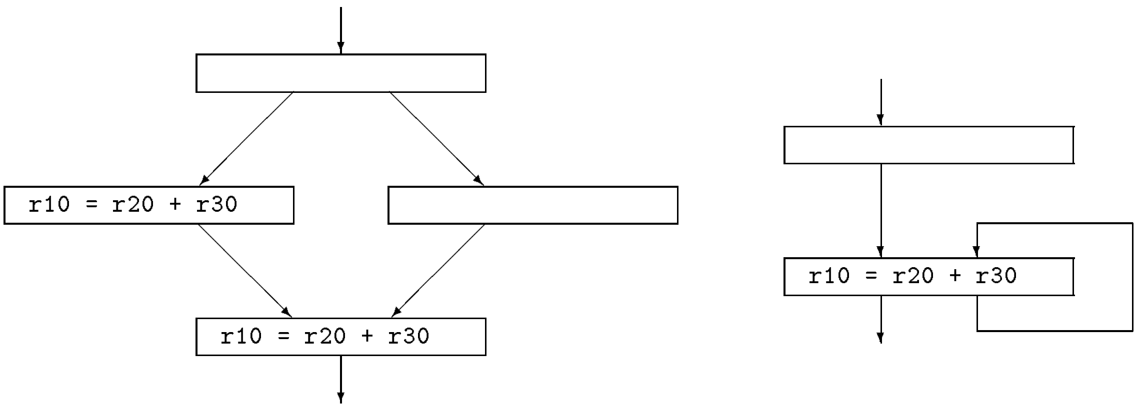

Figure 1.1: Partially redundant and loop-invariant code.

1.1 Usage

To use **partial**, type

`partial` {*options*} [*filename*]

The command line may contain the following, in any order:

- The name of an ILOC file, from which **partial** takes its input. If no file is specified, **partial** reads from `stdin`. Only one file may be specified.
- The `-c` flag, which means chunk away comments from the ILOC file.
- The `-d` flag, which means print debugging output. Up to three `-d`'s are allowed. The output, depending on the number of flags, is (1) placement information, (2) global and placement information, or (3) local, global and placement information.
- The `-t` flag, which prints timing information to the standard error.

The command line flags may be concatenated, as in `-cdd`. **partial** always prints its output on `stdout`. Debugging output and error messages go to `stderr`.

1.2 Terminology

Before we can discuss our implementation of partial redundancy elimination, we need to develop some terminology.

> **Definition** An *expression opcode* is an ILOC opcode which defines one or more result registers as a pure function of one or more constant or register operands. However, copies (*e.g.,* `i2i`) are not considered expression opcodes.

We can identify the expression opcodes by inspecting the `details` field of the opcode data structure; the expression opcodes have the `EXPR` bit set. Some examples of expression opcodes are `iADD`, `iLDI`, `iCMPeq`, `iABS` and `fSQRT`. Some opcodes that are not expression opcodes are `FRAME`, `BRne`, `bSTrr`, `iJSR` and `iRTN`.

In addition, we consider loads from non-constant memory (*e.g.,* `iLDrr`, `dLDor`) to be a special type of expression. These load opcodes have the `LOAD` bit set in their `details` fields. We do not lump the non-constant loads with the "normal" expressions because these load opcodes are not a pure function of their operands; they also depend on memory, which may be modified by stores and calls. Loads from constant memory (*e.g.,* `iLDI`, `bCON`) are considered "normal" expressions (and have `EXPR` set), because they are a function of their register operands only; the memory they access cannot be modified by other operations.

The following pair of definitions concerns the registers used by the routine:

> **Definition** An *expression register* (or *expression*) is the result register of any operation which has an expression opcode.

> **Definition** A *variable register* (or *variable*) is any register that is not an expression register.

Our intent here is that expressions are temporary registers, used during the computation of a source language variable; expression registers are not live between basic blocks. Variable registers, however, correspond roughly to source language variables; they tend to be live across multiple basic blocks. Each expression register may have *subexpressions*:

> **Definition** A *subexpression* of an expression e is an expression register which is either (1) an operand of e, or (2) an operand of some other subexpression of e.

Finally, we need a more precise meaning for the term *computation*:

> **Definition** A *computation* of an expression register e is any operation which defines e.

1.3 Register naming rules

The partial redundancy elimination process requires that we impose some extra demands on the input ILOC routine; these demands occur in the form of rules by which the registers must be named.

Rule 1 *All instructions must contain exactly one operation.*

Rule 1 allows us to make simplifying assumptions about the code. Ordinarily, ILOC may contain multiple operations within each instruction, a facility which is used for, among other things, superscalar instruction scheduling. However, in order to keep our implementation simple, we disallow multiple-operation instructions.

Rule 2 *Multiple computations of the same expression register must use the same operator with the same operands, and computations of two different expression registers must not use the same operator with the same operands.*

Rule 2 exists so that each "right-hand side" in the routine is assigned a name. During analysis, we can talk about a computation by talking about the expression register defined by that computation. Note that the names of variable registers are unrestricted.

Rule 3 *The register number of an expression register must be higher than the register numbers of its expression operands.*

Rule 3 allows us to quickly discover which expression registers depend on each other. We discuss this process further in Section 3.3.

Rule 4 *No expression may be live across a basic block boundary.*

Rule 4 prevents **partial** from moving code into unsafe locations. When **partial** moves code upward, it will not move an operation above a definition of one of the operands of the operation. However, **partial** will blithely (and incorrectly) move an operation above a use of the result register of the operation. We can prevent this type of error by not allowing any expression registers to be live across basic block boundaries: since **partial** moves code to the top or bottom of a block only, no operation defining a register r can be moved between a definition of r and the corresponding use of e. Figure 1.2 contains an example which illustrates the problems caused by expression registers which are live across basic blocks.

1.4 Theory

The partial redundancy elimination process uncovers information about the routine using classical data-flow analysis. In order to record this information, we attach a number of abstract sets to each basic block in the control-flow graph. Each of these data-flow sets can be thought of as an array of Boolean values, one value for each expression register in the routine. Each data-flow set represents a property; an expression register's entry in the set is true if the property holds for the register in the block to which the set is attached. The data-flow information we collect during partial redundancy elimination falls naturally into three categories: local information, global information and placement information, which we will explain in turn.

1.4.1 Local information

Each block has three data-flow sets, `altered`, `avloc` and `antloc`, known collectively as the *local information*. The local information is computed by inspecting the operations present in each basic block; subsequent data-flow sets are computed by inspecting the local information.

Dependence

We say that an expression e *depends* on a variable v if v is an operand of e or if v is an operand of any of e's subexpressions. We denote dependence with the "member" symbol: $v \in e$ means the expression e depends on the variable v. Since the register naming rules require expressions to have larger register numbers than their subexpressions, we can calculate dependence in one pass through the code.

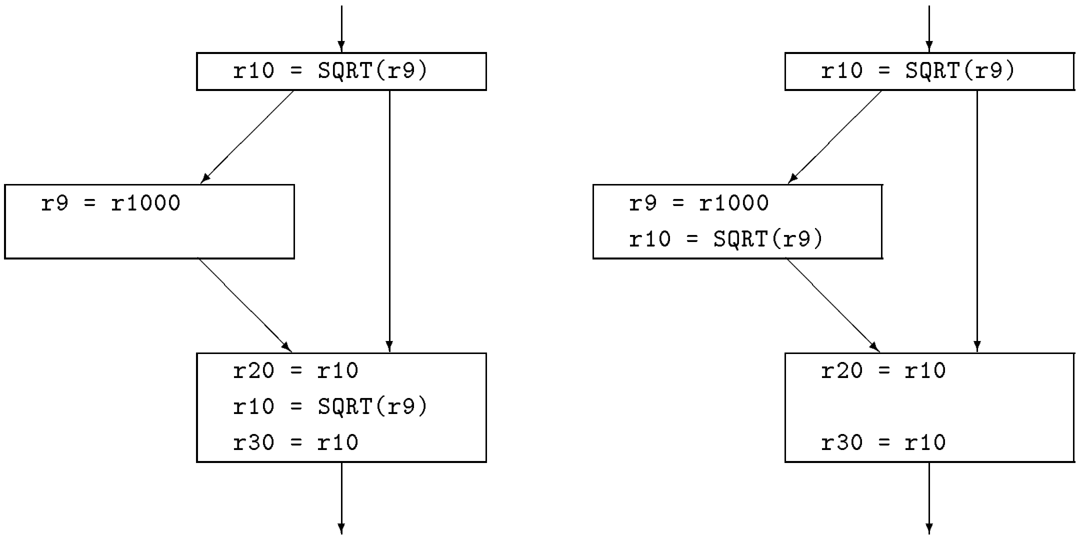

Figure 1.2: Incorrect code as transformed by **partial**. In the left-hand fragment, the expression register `r10` is live from the block `enter` to the block `join`. This code is transformed by **partial** into the fragment on the right. In the right-hand fragment, the value assigned to the variable `r20` along the left branch is different from that in the left-hand fragment.

Locally altered expressions

We say that an expression e is *altered* in a block b if any variable $v \in e$ is computed in b. The data-flow set `altered`[1] contains the register numbers of all expressions which are altered in a block.

Locally available and anticipable expressions

We say that an expression e is *locally available* in a block b if e is computed in b and no variable $v \in e$ is computed between the last computation of e in b and the end of b. Likewise, we say that an expression e is *locally anticipable* in a block b if e is computed in b and no variable $v \in e$ is computed between the beginning of b and first computation of e in b. The local information sets `avloc` and `antloc` contain the register numbers of the locally available and locally anticipable expressions in a block.

1.4.2 Global information

Each block b also has *global information*, consisting of six sets: `avin`, `avout`, `antin`, `antout`, `pavin` and `pavout`, representing, respectively, global availability on entry and exit from b, global anticipability on entry and exit from b, and partial availability on entry and exit from b.

Global availability

An expression e is *globally available* at a point p if a computation of e placed at p would be redundant on all paths from the start block to p. Global availability is a forward, intersection data-flow problem; its data-flow equations[2] follow:

[1] We can see that $\neg$`altered` is roughly analogous to the `transp` of Morel and Renvoise. The difference between $\neg$`altered` and `transp` is that Morel and Renvoise consider an expression e to be dependent on another register r only if r is an operand of e, whereas we consider $r \in e$ if r is an operand of e or of any subexpression of e.

[2] We use the notation $\texttt{set}_i$ to denote the data-flow set `set` attached block i. In addition, we use the symbols Π and Σ to indicate intersections and unions of groups of sets, respectively.

$$
\begin{aligned}
\texttt{avin}_i &= \begin{cases} \mathit{false}, & \text{if } i \text{ is the entry block;} \\ \displaystyle\prod_{j \in pred(i)} \texttt{avout}_j, & \text{otherwise} \end{cases} && (1.1) \\
\texttt{avout}_i &= (\texttt{avin}_i - \texttt{altered}_i) \cup \texttt{avloc}_i && (1.2)
\end{aligned}
$$

Global anticipability

An expression e is *globally anticipable* at a point p if a computation of e placed at p would be redundant on all paths from p to the end block. Global anticipability is a backward, intersection data-flow problem; its data-flow equations are:

$$
\begin{aligned}
\texttt{antout}_i &= \begin{cases} \mathit{false}, & \text{if } i \text{ is the exit block;} \\ \displaystyle\prod_{j \in succ(i)} \texttt{antin}_j, & \text{otherwise} \end{cases} && (1.3) \\
\texttt{antin}_i &= (\texttt{antout}_i - \texttt{altered}_i) \cup \texttt{antloc}_i && (1.4)
\end{aligned}
$$

Partial availability

An expression e is *partially available* at a point p if a computation of e at p would be redundant on some paths from the start block to p. Partial availability is a forward, union data-flow problem; its equations are:

$$
\begin{aligned}
\texttt{pavin}_i &= \begin{cases} \mathit{false}, & \text{if } i \text{ is the entry block;} \\ \displaystyle\sum_{j \in pred(i)} \texttt{pavout}_j, & \text{otherwise} \end{cases} && (1.5) \\
\texttt{pavout}_i &= (\texttt{pavin}_i - \texttt{altered}_i) \cup \texttt{avloc}_i && (1.6)
\end{aligned}
$$

1.4.3 Placement information

The final class of data-flow information is called the *placement information*. Each basic block's placement information consists of the sets `constTerm`, `ppin`, `ppout`, `insertOut`, `insertConst`, `insertIn`, and `delete`. In addition, each control-flow edge has a placement information set called `edgeInsert`.

Possible placement

The first placement information set, `constTerm`, is constant for each block. It is a subexpression used in the calculation of `ppin` and `ppout`, and we extract it in order to speed the data-flow analysis.

The next two placement information sets, `ppin` and `ppout`, represent *possible placement* of an expression on entry or exit of a block. Possible placement of an expression e at a point p means that:

- a partially redundant computation of e is present on all paths from p to the end block, and
- a computation of e at p would reach the first such partially redundant computation on each such path, making those computations fully redundant, and
- if p is the entrance of a block, e is partially available at p.

Possible placement is a backward, intersection data-flow problem. Its values flow backward from blocks containing partially redundant computations, stopping where the possible placement conditions no longer hold. The upshot of possible placement is that a computation of a partially redundant expression can typically be placed at or near the highest point in the routine where possible placement of that expression holds. Inserting a computation of the expression in this position and deleting the original (by now fully redundant) computation yields an improved routine. The possible placement data-flow equations are:

$$\texttt{ppout}_i = \begin{cases} \mathit{false}, & \text{if } i \text{ is the exit block;} \\ \displaystyle\prod_{j \in succ(i)} \texttt{ppin}_j & \text{otherwise} \end{cases} \tag{1.7}$$

$$\texttt{constTerm}_i = \texttt{antin}_i \cap \texttt{pavin}_i \tag{1.8}$$

$$\texttt{ppin}_i = \texttt{constTerm}_i \cap (\texttt{antloc}_i \cup \texttt{ppout}_i) \tag{1.9}$$

Insertion and deletion information

The remaining sets of the placement information concern actual points of insertion and deletion of computations. They are:

`insertOut`[3] is the set of expressions to be inserted at the exit of a block.

$$\texttt{insertOut}_i = (\overline{\texttt{ppin}_i} \cup \texttt{altered}_i) \cap \texttt{ppout}_i \cap \overline{\texttt{avout}_i} \tag{1.10}$$

`insertConst` is a subexpression used to compute the `insertIn` and the `edgeInsert` sets.

$$\texttt{insertConst}_i = \overline{\texttt{ppout}_i \cup \texttt{avout}_i} \tag{1.11}$$

`insertIn` is the set of expressions which *may* be inserted at the entry of a block.

$$\texttt{insertIn}_i = \texttt{ppin}_i \cap \prod_{j \in pred(i)} \texttt{insertConst}_j \tag{1.12}$$

`delete` is the set of expressions whose first occurrences in a block are to be deleted from the block.

$$\texttt{delete}_i = \texttt{antloc}_i \cap \texttt{ppin}_i \cap \overline{\texttt{insertIn}_i} \tag{1.13}$$

`edgeInsert` is the set of expressions which are to be inserted in a new block created by splitting an edge.

$$\texttt{edgeInsert}_{ij} = \texttt{ppin}_j \cap \texttt{insertConst}_i \cap \overline{\texttt{insertIn}_j} \tag{1.14}$$

1.4.4 Algorithm

The partial redundancy elimination algorithm follows:

1. Calculate the correct values for each of the local, global and placement information sets.
2. For each block b:
   (a) remove from b the first occurrence of each expression in $\texttt{delete}_b$,
   (b) insert onto edges $e = (a, b)$ expressions in $\texttt{edgeInsert}_{ab}$,
   (c) insert at the entry of b expressions in $\texttt{insertIn}_b - \texttt{antloc}_b$, and
   (d) insert at the exit of b expressions in $\texttt{insertOut}_b$.

Program	fpppp		tomcatv	doduc	
Routine	fpppp	twldrv	tomcatv	repvid	iniset
Blocks	2	333	91	127	446
Registers	9214	5902	946	831	2283
Operations	22351	16205	2690	1884	6731
Local	1.97	1.32	0.07	0.08	0.19
Global	0.01	0.27	0.02	0.03	0.15
Placement	0.01	0.26	0.02	0.02	0.18
Update	0.11	3.87	0.17	0.21	2.10
Total	2.10	5.72	0.28	0.34	2.62

Figure 1.3: Performance data for **partial**.

1.5 Performance

We've run a few experiments with **partial** to discover how long it takes to process different types of routines. We tested **partial** on five routines collected from three programs in the SPEC benchmark suite. We chose two of the routines, `fpppp` and `twldrv`, because of their large sizes. We chose `iniset` because of its high ratio of basic blocks to ILOC operations. We chose the other routines as being representative of smaller examples.

We ran **partial** on each routine and recorded the amount of time **partial** took in each of its four major steps: finding local information, finding global information, finding placement information, and updating the code. We ran all of the experiments on an IBM RS/6000 Model 540, running at 30MHz, with a 64Kbyte data cache.

Figure 1.5 shows the number of basic blocks, registers and ILOC operations in each test routine, as well as the timings (in seconds) for each of **partial**'s algorithmic steps. In each case, the most time-consuming step was either the collecting of local information or the updating of the code. The collection of local information was longer in the routines with many registers, while the updating of the code took longer in the routines with many basic blocks.

[3] The `insertOut` data-flow set corresponds to the `insert` set introduced in [2]. We have renamed it to comply with the naming convention used for the other data-flow sets; *i.e.,* use the suffix `-In` for sets which hold at the entrance of a block, and use `-Out` for sets which hold at the exit of a block.

Chapter 2

Overall structure

Our C program, `partial.c`, is constructed from the following components.

```
"partial.c" 8a ≡
    ⟨Include files 8b, ... ⟩
    ⟨Type declarations 8c, ... ⟩
    ⟨Global variables 10b, ... ⟩
    ⟨Prototypes 48a, ... ⟩
    ⟨Functions 36a, ... ⟩
    ⟨The main routine 9a⟩
    ◇
```

2.1 Preparatory material

This optimizer pass, like all of the other components of the Massively Scalar Compiler, uses the portable data types defined in the header file `Compiler_Types.h`. It also uses the ILOC parser, which is defined by `Parser.h`. In addition, **partial** makes use of two abstract set implementations, `SparseSets` and `VectorSets`, defined by `SparseSet.h` and `VectorSet.h`, respectively. Finally, we want to be able to print timing information, so we need `Time.h`.

```
⟨Include files 8b⟩ ≡
    #include <Compiler_Types.h>
    #include <Arena.h>
    #include <Parser.h>
    #include <SparseSet.h>
    #include <VectorSet.h>
    #include <Time.h>
    ◇
```

Macro defined by scraps 8b, 41a.
Macro referenced in scrap 8a.

We plan to keep various pieces of information in the block extension, which is a `struct` associated with each block in the control-flow graph. Here we declare the "frame" of the structure; we will declare and describe the fields of the extension structure as we need them.

```
⟨Type declarations 8c⟩ ≡
    struct block_extension {
      ⟨Block extension fields 17a, ... ⟩
    };
    ◇
```

Macro defined by scraps 8c, 31a, 40.
Macro referenced in scrap 8a.

2.2 The main routine

First, we parse the command line, noting user-specified options and setting **inputFileName** to the name of the input file. If the user specifies no input file, the parser will recognize the **NULL** value of **inputFileName** and use **stdin**. Then we read the ILOC input and construct the control-flow graph, using the Parser routine **Block_Init**. Then we perform partial redundancy elimination. When we are finished optimizing the routine, we set **print_all_operations** to be **FALSE**, so that **Block_Put_All** will delete the operations we have marked non-critical. We then print the resultant routine to **stdout**.

We want to be able to print timing information for each of the major steps of **partial**. To do this, we start up a **Timer** variable, **overallTime**, at the beginning of the program, and we dump its value after parsing, optimization and printing. In addition, so that we may get an idea of **partial**'s memory performance, we make calls to **Arena_DumpStats** after parsing and optimization.

⟨The main routine 9a⟩ ≡
```
Void main(Unsigned_Int argc, Char **argv)
{
  Char *inputFileName = NULL;
  Timer overallTime = Time_Start();

  ⟨Parse the command line 10a⟩
  Block_Init(inputFileName);
  Time_Dump(overallTime, "Parsing took: %1.2f secs\n"); Arena_DumpStats();
  ⟨Perform partial redundancy elimination 9b⟩
  Time_Dump(overallTime, "Optimization took: %1.2f secs\n"); Arena_DumpStats();
  print_all_operations = FALSE;
  Block_Put_All(stdout);
  Time_Dump(overallTime, "Printing took: %1.2f secs\n");
  exit(0);
}◇
```
Macro referenced in scrap 8a.

We'll perform partial redundancy elimination using the following steps. Note the **Timer** variable called **optimizationTime**; we'll dump its value inside each of the steps of the algorithm.

⟨Perform partial redundancy elimination 9b⟩ ≡
```
{
  Timer optimizationTime = Time_Start();

  ⟨Collect local information 12a⟩
  ⟨Collect global information 20⟩
  ⟨Collect placement information 27a⟩
  ⟨Update the code 33⟩
}◇
```
Macro referenced in scrap 9a.

2.3 Parsing the command line

The command line's possible contents are detailed in the introduction to **partial**. We parse the command line by looping over the entries in `argv`. We define `argNumber` to be the index of the current `argv` entry, and we define `currChar` to be a pointer to the current character in the entry. We assume that all command-line strings beginning with a hyphen are flags and that all other strings are input filenames.

⟨Parse the command line 10a⟩ ≡
```
{
  Unsigned_Int argNumber;

  for (argNumber = 1; argNumber < argc; argNumber++) {
    Char *currChar = argv[argNumber];
    if (*currChar++ == '-') {
      ⟨Parse a command-line option 10d⟩
    } else {
      ⟨Parse the input file name 11⟩
    }
  }
}◇
```
Macro referenced in scrap 9a.

If we encounter a `-c` on the command line, we set the parser control flag `keep_comments` to `FALSE`. If we encounter a `-t`, we toggle the printing of timing information toggling the value of `time_print`. Each time we encounter a `-d` on the command line, we increment our global variable `debug`.

⟨Global variables 10b⟩ ≡
```
static Unsigned_Int debug = 0;
◇
```
Macro defined by scraps 10b, 13a, 38.
Macro referenced in scrap 8a.

If we find an error, we'll execute the following:

⟨Print usage message and abort 10c⟩ ≡
```
fprintf(stderr, "Usage: %s [-cdddt] [filename]\n", argv[0]);
fprintf(stderr, "  -c\tChunk away comments from the ILOC file. \n");
fprintf(stderr, "  -d\tPrint out placement information.\n");
fprintf(stderr, "  -dd\tPrint out placement and global information.\n");
fprintf(stderr, "  -ddd\tPrint out all data flow information.\n");
fprintf(stderr, "  -t\tPrint timing information. \n");
ABORT;◇
```
Macro referenced in scraps 10d, 11.

We parse a command-line flag by switching on each letter after the initial hyphen.

⟨Parse a command-line option 10d⟩ ≡
```
while (*currChar) {
  switch (*currChar++) {
    case 'c': keep_comments = FALSE; break;
    case 'd': debug++; break;
    case 't': time_print = TRUE; break;
    default: ⟨Print usage message and abort 10c⟩
  }
}◇
```
Macro referenced in scrap 10a.

We assume that any string that doesn't begin with a hyphen is a file name. If `inputFileName` is still `NULL` we know that this is the first file name on the command line, so we can proceed. If we are trying to read a second file name, we print the `usageString` and exit.

⟨Parse the input file name 11⟩ ≡

```
if (inputFileName == NULL) {
  inputFileName = argv[argNumber];
} else {
  ⟨Print usage message and abort 10c⟩
}◇
```

Macro referenced in scrap 10a.

Chapter 3

Collection of local information

These are the steps involved in finding the local information. We use the `Timer` called `localTime` to acquire timings for the steps of collecting local information; we dump `localTime`'s value inside each step. The other variables defined here are discussed in Section 3.1.

⟨Collect local information 12a⟩ ≡
```
{
  Timer localTime = Time_Start();
  SparseSet expressions, variables, memDependent;
  VectorSet vecVariables;
  LimitedSet *dependences;

  ⟨Create first, second and fourth arenas 12b⟩
  ⟨Initialize data structures for local information 13b⟩
  ⟨Find expressions 14a⟩
  ⟨Find variables 15b⟩
  ⟨Find dependences for expressions 15c⟩
  ⟨Find dependences for variables 16b⟩
  ⟨Find local information 17b⟩
  ⟨Debugging output: Local information 43a⟩
  Arena_Destroy(firstArena);
  Time_Dump(optimizationTime, "  Local info took: %1.2f secs\n");
}◇
```
Macro referenced in scrap 9b.

Arenas 1, 2 and 4 begin their lives here. For more information on **partial**'s management of memory, see Appendix A.

⟨Create first, second and fourth arenas 12b⟩ ≡
```
firstArena = Arena_Create();
secondArena = Arena_Create();
fourthArena = Arena_Create();◇
```
Macro referenced in scrap 12a.

3.1 Data structures

We need to construct and fill in some data structures before we can uncover the local information. We'll have two `SparseSet`s, called `expressions` and `variables`, which will hold the register numbers of the expressions and variables. In addition, we will keep a `VectorSet` called `vecVariables`, which contains exactly the same elements as `variables` but in a `VectorSet` format.

Another `SparseSet`, called `memDependent`, will contain the register numbers of the expression registers which are dependent upon some non-constant memory; that is,

$\texttt{memDependent} = \{e \mid \exists v \in e \text{ where } v \text{ is defined by a load from non-constant memory}\}$

We will also have an array of `LimitedSet`s called `dependences`.[1] The `dependences` array will serve two functions: first, an entry whose index is the register number of an expression register e will hold the set of variables v on which e depends plus the subexpressions e' of e. Secondly, an entry in `dependences` whose index is the register number of a variable register v will hold the set of expressions e which depend on v. More succinctly,

$$\texttt{dependences}[e] = \{v \mid v \in e\} \cup \{e' \mid e' \text{ is a subexpression of } e\}$$

$$\texttt{dependences}[v] = \{e \mid v \in e\}$$

Finally, there is a structure called `expressionDefPoints`, which is an array of pointers to operations. Its entries will point to the definition points of each expression register. We will need this structure during the final stages of optimization, so we declare it globally:

⟨Global variables 13a⟩ ≡

```
static Operation **expressionDefPoints;
◇
```

Macro defined by scraps 10b, 13a, 38.
Macro referenced in scrap 8a.

The sets `expressions`, `variables`, `vecVariables` and `memDependent` have a universe size of `register_count` entries; `expressionDefPoints` is `register_count` entries long as well. The `dependences` array is also `register_count` entries long, and each of its entries is a set which is `register_count` entries long; consequently, `dependences` is the largest data structure used by **partial**.

⟨Initialize data structures for local information 13b⟩ ≡

```
{
  Unsigned_Int i;

  expressions = SparseSet_Create(firstArena, register_count);
  variables = SparseSet_Create(firstArena, register_count);
  vecVariables = VectorSet_Create(firstArena, register_count);
  memDependent = SparseSet_Create(firstArena, register_count);
  expressionDefPoints = Arena_GetMem(block_memory_bank,
                                     sizeof(Operation *) * register_count);
  dependences = Arena_GetMem(firstArena, sizeof(LimitedSet) * register_count);
  for (i = 0; i < register_count; i++) {
    dependences[i] = LimitedSet_Create(firstArena, register_count);
  }
  Time_Dump(localTime, "    Local info initialization took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

[1] We use `LimitedSets` because of their fast union and `ForAll` operations. For more information on `LimitedSets`, see Appendix B.

3.2 Expressions and variables

Having initialized the data structures, our first major tasks are to find the set of expressions, to construct `expressionDefPoints`, and to begin work on `memDependent`. We accomplish these objectives by inspecting each operation in the routine during a walk of the control-flow graph. We also take advantage of this pass over the blocks to create an extension for each block and mark each operation `critical`.

⟨Find expressions 14a⟩ ≡

```
{
  Block *block;

  ForAllBlocks (block) {
    Inst *inst;

    block->block_extension = Arena_GetMem(block_memory_bank, sizeof(struct block_extension));
    Block_ForAllInsts (inst, block) {
      Operation **operPtr;

      Inst_ForAllOperations (operPtr, inst) {
        Operation *operation = *operPtr;

        operation->critical = TRUE;
        ⟨Inspect this operation 14b⟩
      }
    }
  }
  Time_Dump(localTime, "    Expressions took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

For each operation in the routine, we find the `details` field of the operation's opcode and `definedReg`, the first register defined by the operation (if any). If inspection of the `details` field reveals that the operation is an expression (`EXPR`) or a load from non-constant memory (`LOAD`), then we know that `definedReg` is an expression register and that we should put it in the `expressions` set if it is not already there.

⟨Inspect this operation 14b⟩ ≡

```
{
  Unsigned_Int details = opcode_specs[operation->opcode].details;
  Unsigned_Int definedReg = operation->arguments[operation->referenced];

  if ((details & (LOAD | EXPR)) && !SparseSet_Member(expressions, definedReg)) {
    ⟨Put the expression in the set of expressions 15a⟩
  }
}◇
```

Macro referenced in scrap 14a.

While putting the expression register `definedReg` into `expressions`, we also fill in the expression's entry in `expressionDefPoints`. In addition, if the operation is a load from non-constant memory, we designate `definedReg` as being `memDependent`. Since this piece of code is executed exactly once for each expression, we won't waste time here if the expression has multiple definitions.

⟨Put the expression in the set of expressions 15a⟩ ≡

```
{
  SparseSet_Insert(expressions, definedReg);
  expressionDefPoints[definedReg] = operation;
  if (details & LOAD) {
    SparseSet_Insert(memDependent, definedReg);
  }
}◇
```

Macro referenced in scrap 14b.

Having found the set of expressions, we can now find `variables` = `vecVariables` = ¬`expressions`.

⟨Find variables 15b⟩ ≡

```
{
  Unsigned_Int i;

  for (i = 0; i < register_count; i++) {
    if (!SparseSet_Member(expressions, i)) {
      SparseSet_Insert(variables, i);
      VectorSet_Insert(vecVariables, i);
    }
  }
  Time_Dump(localTime, "    Variables took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

3.3 Dependence

Now that we have the sets of expressions and variables, we can calculate the values for the `dependences` data structure. The first task is to find the dependence information for expressions. To do this, we look at each expression register `expressionReg` in ascending order (we sort `expressions` first) and find the dependences for `expressionReg`. We will need to make use of a `SparseSet` called `subexpressions` during the calculation of `expressionReg`'s dependences; we create `subexpressions` at this scope in order to avoid extra initializations inside the loop body.

⟨Find dependences for expressions 15c⟩ ≡

```
{
  Unsigned_Int expressionReg;
  SparseSet subexpressions = SparseSet_Create(firstArena, register_count);

  SparseSet_Sort(expressions);
  SparseSet_Forall (expressionReg, expressions) {
    ⟨Find dependences for expressionReg 16a⟩
  }
  Time_Dump(localTime, "    Expression dependences took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

To find the dependences for a particular `expressionReg`, we first find `subexpressions`, the set of immediate subexpressions of `expressionReg`. We find `subexpressions` by inspecting each register `use` referenced in the operation which defines `expressionReg` (this operation is `expressionDefPoints[expressionReg]`). We first add `use` to `dependences[expressionReg]`, because `use` is an operand of `expressionReg`. Then, if `use` is an expression register, we also add `use` to `subexpressions`.

Once we have found `subexpressions`, we inspect each register `subexpressionReg` in `subexpressions`. For each `subexpressionReg`, we add `dependences[subexpressionReg]` to `dependences[expressionReg]`. Then, if `subexpressionReg` is in `memDependent`, we put `expressionReg` into `memDependent`.

In summary, the `Operation_ForAllUses` loop installs into `dependences` the *immediate* dependence data for expression registers. The inner `SparseSet_ForAll` loop then takes the closure of the immediate dependence information by using just-installed immediate information as well as the already-closed dependence information of the lower-numbered registers. This section of code works because the register numbers of expression registers are greater than the register numbers of their operands; thus, one (ordered) pass through the `expressions` set is sufficient to take the closure of the expression dependence information.

The calculation of dependences for all expressions runs in $O(n^2)$ time, where n is the number of registers in the routine. This is because:

- We execute this section of code $O(n)$ times, once for each expression register.
- We expect the `SparseSet_ForAll` loop to have only 1 or 2 iterations.
- The `LimitedSet_Union` operation requires $O(n)$ time to execute.

⟨Find dependences for `expressionReg` 16a⟩ ≡

```
{
  Unsigned_Int subexpressionReg;
  Unsigned_Int2 *usePtr;

  SparseSet_Clear(subexpressions);
  Operation_ForAllUses (usePtr, expressionDefPoints[expressionReg]) {
    Unsigned_Int use = *usePtr;

    LimitedSet_Insert(dependences[expressionReg], use);
    if (SparseSet_Member(expressions, use)) SparseSet_Insert(subexpressions, use);
  }
  SparseSet_Forall (subexpressionReg, subexpressions) {
    LimitedSet_Union(dependences[expressionReg],
                     dependences[expressionReg],
                     dependences[subexpressionReg]);
    if (SparseSet_Member(memDependent, subexpressionReg)) {
      SparseSet_Insert(memDependent, expressionReg);
    }
  }
}◇
```

Macro referenced in scrap 15c.

Having found the dependences for expressions, we next fill in the dependence information for the variables (recall that `dependences[`v`]` $= \{e \mid v \in e\}$). We look at each expression register `expressionReg` and, for each variable register `variable` $\in$ `expressionReg`, we add `expressionReg` to `dependences[variable]`,[2] the list of expressions which depend on `variable`.

⟨Find dependences for variables 16b⟩ ≡

```
{
  Unsigned_Int expressionReg, variable;

  SparseSet_Forall (expressionReg, expressions) {
    LimitedSet_ForAll (variable, dependences[expressionReg]) {
      if (SparseSet_Member(variables, variable)) {
        VectorSet_Insert(dependences[variable]->vector, expressionReg);
      }
    } LimitedSet_EndForAll;
  }
  Time_Dump(localTime, "    Variable dependences took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

[2] Although `dependences[variable]` is a `LimitedSet`, we can insert items directly into its `VectorSet`, since we do not plan to use `LimitedSet_ForAll` on `dependences[variable]`.

3.4 Local information

Now that the dependence information is complete we may begin finding the local information. We keep the local information sets, **altered**, **avloc** and **antloc**, in the extension of each block.

⟨Block extension fields 17a⟩ ≡

```
VectorSet altered;
VectorSet avloc;
VectorSet antloc;
◇
```

Macro defined by scraps 17a, 21a, 27b, 30ac.
Macro referenced in scrap 8c.

In order to provide a shorthand method of accessing the information, we also keep the local information in locally-scoped sets of the appropriate names. For each block in the routine, we create the local information sets for the block, then find the local information, then make sure that the **avloc** and **antloc** sets contain only expressions, because availability and anticipability do not make sense for variables.

⟨Find local information 17b⟩ ≡

```
{
  Block *block;

  ForAllBlocks (block) {
    VectorSet altered = block->block_extension->altered
      = VectorSet_Create(fourthArena, register_count);
    VectorSet avloc = block->block_extension->avloc
      = VectorSet_Create(secondArena, register_count);
    VectorSet antloc = block->block_extension->antloc
      = VectorSet_Create(block_memory_bank, register_count);

    ⟨Find local info in each block 18a⟩
    VectorSet_Difference(avloc, avloc, vecVariables);
    VectorSet_Difference(antloc, antloc, vecVariables);
  }
  Time_Dump(localTime, "    Altered, avloc and antloc took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 12a.

To find local information in a block, we examine each operation in the block,[3] We are concerned with operations that define registers and operations that may modify memory (that is, stores and calls).

⟨Find local info in each block 18a⟩ ≡
```
{
  Inst *inst;

  Block_ForAllInsts (inst, block) {
    Operation **operPtr;

    Inst_ForAllOperations (operPtr, inst) {
      Operation *operation = *operPtr;
      Opcode_Names opcode = operation->opcode;

      if (opcode_specs[opcode].details & (STORE|CALL)) {
        ⟨Process an operation that may modify memory 18b⟩
      }
      if (opcode_specs[opcode].defs) {
        ⟨Process an operation that defines a register 19a⟩
      }
    }
  }
}◇
```
Macro referenced in scrap 17b.

If this operation is a store or a call, we mark all the expressions which are **memDependent** (*i.e.*, dependent on memory) as **altered** and not **avloc** in this block.

⟨Process an operation that may modify memory 18b⟩ ≡
```
{
  Unsigned_Int reg;

  SparseSet_Forall (reg, memDependent) {
    VectorSet_Insert(altered, reg);
    VectorSet_Delete(avloc, reg);
  }
}◇
```
Macro referenced in scrap 18a.

[3] Register naming rule 1 ensures that there will be only one operation in each instruction, so we do not have to take the usual precautions concerning the parallelism of operations within an instruction.

Once we know that an operation defines a register, we iterate through the registers defined by the operation. If `definedReg` is a variable, we put the expressions which depend on `definedReg` into the `altered` set for this block, because a variable on which they depend has been computed in the block. If `definedReg` is already in the `avloc` set for this block, then the operation is redundant (because the expression has already been computed in the block), and so we remove the operation by marking it non-critical. Otherwise, we mark `definedReg` locally available (and possibly locally anticipable), because the expression is computed in the block.

⟨Process an operation that defines a register 19a⟩ ≡

```
{
  Unsigned_Int2 *defPtr;

  Operation_ForAllDefs (defPtr, operation) {
    Unsigned_Int definedReg = *defPtr;

    if (SparseSet_Member(variables, definedReg)) {
      ⟨Put into altered the expressions which depend on definedReg 19b⟩
    } else if (VectorSet_Member(avloc, definedReg)) {
      operation->critical = FALSE;
    } else {
      ⟨Mark definedReg locally available 19c⟩
    }
  }
}◇
```

Macro referenced in scrap 18a.

Recall that if `definedReg` is a variable, `dependences[definedReg]` is the set $\{e \mid \texttt{definedReg} \in e\}$. Therefore, when we encounter a computation of a variable register, we add `dependences[definedReg]` to `altered` and remove `dependences[definedReg]` from `avloc`.

⟨Put into `altered` the expressions which depend on `definedReg` 19b⟩ ≡

```
VectorSet_Union(altered, altered, dependences[definedReg]->vector);
VectorSet_Difference(avloc, avloc, dependences[definedReg]->vector);◇
```

Macro referenced in scrap 19a.

When we encounter a computation of an expression register `definedReg`, we add `definedReg` to the `avloc` set for the current block. Then, if it has not been `altered` in the block, we put `definedReg` into the block's `antloc` set.

⟨Mark `definedReg` locally available 19c⟩ ≡

```
VectorSet_Insert(avloc, definedReg);
if (!VectorSet_Member(altered, definedReg)) {
  VectorSet_Insert(antloc, definedReg);
}◇
```

Macro referenced in scrap 19a.

Chapter 4

Collection of global information

Recall from the discussion of theory (Section 1.4) that we describe each of the three types of global information in terms of data-flow equations. The solutions to the equations are similar, and we will solve all of them using the depth-first iterative algorithm.

The iterative algorithm has two steps: an initialization phase and an iterative phase. In the initialization phase, we initialize the global information sets to values appropriate for the desired solution. That is, for union problems we make the sets smaller than the solution, and for intersection problems we make the sets larger than the solution. In the iterative phase, we recalculate the information until the system stabilizes. We execute the iterative algorithm once for each type of data-flow information: availability, anticipability and partial availability.

The set **tempSet** is used in each calculation, so we'll just declare and create it at this outer scope level.

⟨Collect global information 20⟩ ≡
```
{
  VectorSet tempSet = VectorSet_Create(secondArena, register_count);
  thirdArena = Arena_Create();

  ⟨Initialize the availability data-flow problem 21b, ... ⟩
  ⟨Find availability using the depth-first iterative algorithm 22a⟩

  ⟨Initialize the anticipability data-flow problem 23a, ... ⟩
  ⟨Find anticipability using the depth-first iterative algorithm 23c⟩

  ⟨Initialize the partial availability data-flow problem 24b, ... ⟩
  ⟨Find partial availability using the depth-first iterative algorithm 25b⟩

  ⟨Debugging output: Global information 43b⟩
  Arena_Destroy(secondArena);
  Time_Dump(optimizationTime, "  Global info took: %1.2f secs\n");
}◇
```
Macro referenced in scrap 9b.

We keep the global information sets in the block extension; we use the names introduced in Section 1.4. There is also a dirty bit, which we use during the solving of the data-flow problems to indicate whether the global information for a block needs to be recomputed.

⟨Block extension fields 21a⟩ ≡
```
VectorSet avin, avout;
VectorSet antin, antout;
VectorSet pavin, pavout;
Boolean dirty;
◇
```
Macro defined by scraps 17a, 21a, 27b, 30ac.
Macro referenced in scrap 8c.

4.1 Global availability

Global availability is a forward problem, so the first initializations we do concern the start block. First, we create its **avout** set, initializing it to the empty set, because the start block contains no code. **avin** for the start block is not used, so we set it to **NULL**. We set the start block's **dirty** field to **FALSE**, so that the start block will not be processed in the iterative algorithm's loop.

⟨Initialize the availability data-flow problem 21b⟩ ≡
```
{
  Block_Extension extension = start_block->block_extension;

  extension->avout = VectorSet_Create(fourthArena, register_count);
  extension->avin = NULL;
  extension->dirty = FALSE;
}◇
```
Macro defined by scraps 21bc.
Macro referenced in scrap 20.

Next, we initialize the rest of the blocks in the control-flow graph. For each block other than the start block, we create the **avin** and **avout** sets. We then initialize **avin** to the 1-vector and **avout** to **altered** → **avloc**,[1] that is, all expressions except those which are in **altered** but not in **avloc**. We set the **dirty** field to **TRUE** so that the block will be processed in the iterative algorithm's loop.

⟨Initialize the availability data-flow problem 21c⟩ ≡
```
{
  Block *block;
  Unsigned_Int i;

  for (block = preorder_block_list[i = 2]; block; block = preorder_block_list[++i]) {
    Block_Extension extension = block->block_extension;
    VectorSet avin = extension->avin = VectorSet_Create(secondArena, register_count);
    VectorSet avout = extension->avout = VectorSet_Create(fourthArena, register_count);

    VectorSet_Complement(avin, avin);
    VectorSet_Imply(avout, extension->altered, extension->avloc);
    extension->dirty = TRUE;
  }
}◇
```
Macro defined by scraps 21bc.
Macro referenced in scrap 20.

The depth-first iterative algorithm will solve the availability system in a number of iterations proportional to the maximum loop nesting depth of the routine. We keep looping over the control-flow graph in reverse

[1] Recall that $A \rightarrow B = \overline{A} \cup B$.

postorder (forward problem), recomputing the availability information until the information stops changing. We save some time here by recomputing availability information only if the block has been marked **`dirty`**. Blocks are marked dirty when availability information in their predecessors changes.

⟨Find availability using the depth-first iterative algorithm 22a⟩ ≡

```
{
  Boolean changed;

  do {
    Block *block;

    changed = FALSE;
    ForAllBlocks_rPostorder(block) {
      Block_Extension extension = block->block_extension;
      if (extension->dirty) ⟨Recalculate availability information for block 22b⟩
    }
  } while (changed);
}◇
```

Macro referenced in scrap 20.

The first step in recalculating the availability information for a block is to calculate the availability on entry. We do this by taking the intersection of all of the block's predecessors' **`avout`** sets; we know there must be at least one predecessor because we know that this block is not the start block. We store the result of the recalculation in **`tempSet`**.

In the second step we recalculate the **`avout`** set for the block. However, since **`avout`** changes only when **`avin`** changes, we don't have to recalculate **`avout`** if **`tempSet`** is equal to **`avin`**. If the newly-calculated **`tempSet`** is different from the old **`avin`**, then we copy **`tempSet`** into **`avin`**. Then we recalculate **`avout = avin - altered`** ∪ **`avloc`**. Finally we mark the **`changed`** flag **`TRUE`** and mark all of the successors of the block **`dirty`**.

At some point during the entire process, we need to set the block's **`dirty`** field to **`FALSE`**; we do it at the beginning.

⟨Recalculate availability information for block 22b⟩ ≡

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

  extension->dirty = FALSE;
  VectorSet_Copy(tempSet, edge->pred->block_extension->avout);
  for (edge = edge->next_pred; edge; edge = edge->next_pred) {
    VectorSet_Intersect(tempSet, tempSet, edge->pred->block_extension->avout);
  }
  if (!VectorSet_Equal(extension->avin, tempSet)) {
    changed = TRUE;
    VectorSet_Copy(extension->avin, tempSet);
    VectorSet_DifferenceUnion(extension->avout, extension->avin,
                              extension->altered, extension-> avloc);
    Block_ForAllSuccs (edge, block) {
      edge->succ->block_extension->dirty = TRUE;
    }
  }
}◇
```

Macro referenced in scrap 22a.

4.2 Global anticipability

Global anticipability is a backward problem, so the first initializations we do concern the end block. First, we create its **`antin`** set, initializing it to the empty set, because the end block contains no code. **`antout`** for the end block is not used, so we set it to NULL. We set the end block's dirty field to FALSE, so that the end block will not be processed in the iterative algorithm's loop.

⟨Initialize the anticipability data-flow problem 23a⟩ ≡
```
{
  Block_Extension extension = end_block->block_extension;

  extension->antin = VectorSet_Create(thirdArena, register_count);
  extension->antout = NULL;
  extension->dirty = FALSE;
}◇
```
Macro defined by scraps 23ab.
Macro referenced in scrap 20.

Next, we initialize the rest of the blocks in the control-flow graph. For each block other than the end block, we create the antout and antin sets. We then initialize antout to the 1-vector and antin to altered → antloc, that is, all expressions except those which are in altered but not in antloc. We set the dirty field to TRUE so that the block will be processed in the iterative algorithm's loop.

⟨Initialize the anticipability data-flow problem 23b⟩ ≡
```
{
  Block *block;
  Unsigned_Int i;

  ForAllBlocks (block) {
    if (block != end_block) {
      Block_Extension extension = block->block_extension;
      VectorSet antout = extension->antout = VectorSet_Create(secondArena, register_count);
      VectorSet antin = extension->antin = VectorSet_Create(thirdArena, register_count);

      VectorSet_Complement(antout, antout);
      VectorSet_Imply(antin, extension->altered, extension->antloc);
      extension->dirty = TRUE;
    }
  }
}◇
```
Macro defined by scraps 23ab.
Macro referenced in scrap 20.

Once again, the depth-first iterative algorithm keeps looping over the control-flow graph in postorder (backward problem), recomputing the anticipability information until the information stops changing. For this backward problem, blocks are marked dirty when anticipability information in their successors changes.

⟨Find anticipability using the depth-first iterative algorithm 23c⟩ ≡
```
{
  Boolean changed;

  do {
    Block *block;

    changed = FALSE;
    ForAllBlocks_Postorder(block) {
      Block_Extension extension = block->block_extension;
      if (extension->dirty) ⟨Recalculate anticipability information for block 24a⟩
    }
  } while (changed);
}◇
```
Macro referenced in scrap 20.

The first step in recalculating the anticipability information for a block is to calculate the anticipability on exit. We do this by taking the intersection of all of the block's successors' `antin` sets; we know there must be at least one successor because we know that this block is not the end block. We store the result of the recalculation in `tempSet`.

In the second step we recalculate the `antin` set for the block. However, since `antin` changes only when `antout` changes, we don't have to recalculate `antin` if `tempSet` is equal to `antout`. If the newly-calculated `tempSet` is different from the old `antout`, then we copy `tempSet` into `antout`. Then we recalculate `antin = antout - altered` ∪ `antloc`. Finally we mark the `changed` flag `TRUE` and mark all of the predecessors of the block `dirty`.

At some point during the entire process, we need to set the block's `dirty` field to `FALSE`; we do it at the beginning.

⟨Recalculate anticipability information for `block` 24a⟩ ≡

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

  extension->dirty = FALSE;
  VectorSet_Copy(tempSet, edge->succ->block_extension->antin);
  for (edge = edge->next_succ; edge; edge = edge->next_succ) {
    VectorSet_Intersect(tempSet, tempSet, edge->succ->block_extension->antin);
  }
  if (!VectorSet_Equal(extension->antout, tempSet)) {
    changed = TRUE;
    VectorSet_Copy(extension->antout, tempSet);
    VectorSet_DifferenceUnion(extension->antin, extension->antout,
                              extension->altered, extension->antloc);
    Block_ForAllPreds (edge, block) {
      edge->pred->block_extension->dirty = TRUE;
    }
  }
}◇
```

Macro referenced in scrap 23c.

4.3 Partial availability

Partial availability is a forward problem, so the first initializations we do concern the start block. First, we create its `pavout` set, initializing it to the empty set, because the start block contains no code. `pavin` for the start block is not used in this data-flow problem, but since it is used in the calculation of the placement information, we initialize it to the empty set. We set the start block's `dirty` field to `FALSE`, so that the start block will not be processed in the iterative algorithm's loop.

⟨Initialize the partial availability data-flow problem 24b⟩ ≡

```
{
  Block_Extension extension = start_block->block_extension;

  extension->pavout = VectorSet_Create(secondArena, register_count);
  extension->pavin = VectorSet_Create(thirdArena, register_count);
  extension->dirty = FALSE;
}◇
```

Macro defined by scraps 24b, 25a.
Macro referenced in scrap 20.

Next, we initialize the rest of the blocks in the control-flow graph. For each block other than the start block, we create the **pavin** and **pavout** sets. We then initialize **pavout** to **avloc**. We set the **dirty** field to **TRUE** so that the block will be processed in the iterative algorithm's loop.

⟨Initialize the partial availability data-flow problem 25a⟩ ≡

```
{
  Block *block;
  Unsigned_Int i;

  for (block = preorder_block_list[i = 2]; block;  block = preorder_block_list[++i]) {
    Block_Extension extension = block->block_extension;
    VectorSet pavin = extension->pavin =  VectorSet_Create(thirdArena, register_count);
    VectorSet pavout = extension->pavout =  VectorSet_Create(secondArena, register_count);

    VectorSet_Copy(pavout, extension->avloc);
    extension->dirty = TRUE;
  }
}◇
```

Macro defined by scraps 24b, 25a.
Macro referenced in scrap 20.

Once again, the depth-first iterative algorithm keeps looping over the control-flow graph in reverse postorder (forward problem), recomputing the partial availability information until the information stops changing. For forward problems, blocks are marked dirty when partial availability information in their predecessors changes.

⟨Find partial availability using the depth-first iterative algorithm 25b⟩ ≡

```
{
  Boolean changed;

  do {
    Block *block;
    changed = FALSE;
    ForAllBlocks_rPostorder(block) {
      Block_Extension extension = block->block_extension;
      if (extension->dirty) {
        ⟨Recalculate partial availability information for block 26⟩
      }
    }
  } while (changed);
}◇
```

Macro referenced in scrap 20.

The first step in recalculating the partial availability information for a block is to calculate the partial availability on entry. We do this by taking the union of all of the block's predecessors' **pavout** sets; we know there must be at least one predecessor because we know that this block is not the start block. We store the result of the recalculation in **tempSet**.

In the second step we recalculate the **pavout** set for the block. However, since **pavout** changes only when **pavin** changes, we don't have to recalculate **pavout** if **tempSet** is equal to **pavin**. If the newly-calculated **tempSet** is different from the old **pavin**, then we copy **tempSet** into **pavin**. Then we recalculate **pavout = pavin - altered** ∪ **avloc**. Finally we mark the **changed** flag **TRUE** and mark all of the successors of the block **dirty**.

At some point during the entire process, we need to set the block's `dirty` field to `FALSE`; we do it at the beginning.

⟨Recalculate partial availability information for `block` 26⟩ ≡

```
{
  Edge *edge = block->pred;
  extension->dirty = FALSE;
  VectorSet_Copy(tempSet, edge->pred->block_extension->pavout);
  for (edge = edge->next_pred; edge; edge = edge->next_pred) {
    VectorSet_Union(tempSet, tempSet, edge->pred->block_extension->pavout);
  }
  if (!VectorSet_Equal(extension->pavin, tempSet)) {
    changed = TRUE;
    VectorSet_Copy(extension->pavin, tempSet);
    VectorSet_DifferenceUnion(extension->pavout, extension->pavin,
                              extension->altered, extension-> avloc);
    Block_ForAllSuccs (edge, block) {
      edge->succ->block_extension->dirty = TRUE;
    }
  }
}◇
```

Macro referenced in scrap 25b.

Chapter 5

Collection of placement information

The last remaining class of data-flow information is the placement information. Because some of the placement information sets depend on other placement information sets, we compute the placement information in three stages. We first find `ppin` and `ppout`, using the depth-first iterative algorithm in its two-step (initialize and solve) form. Next we find `insertConst` and `insertOut`, and finally we compute `edgeInsert`, `insertIn`, and `delete`.

⟨Collect placement information 27a⟩ ≡
```
    ⟨Initialize the data-flow problem for ppin and ppout 28a, ... ⟩
    ⟨Calculate ppin and ppout 29a⟩
    ⟨Debugging output: ppin and ppout 44a⟩
    Arena_Destroy(thirdArena);

    ⟨Find insertConst and insertOut 30b⟩
    Arena_Destroy(fourthArena);

    ⟨Find insertIn, delete and edgeInsert 31b⟩
    ⟨Debugging output: Update sets 44b⟩
    Time_Dump(optimizationTime, "  Placement info took: %1.2f secs\n");
    ◇
```
Macro referenced in scrap 9b.

5.1 Possible placement

We keep `ppin` and `ppout` in the block extension. Also in the block extension is `constTerm`, a set used in the calculation of `ppin` and `ppout`.

⟨Block extension fields 27b⟩ ≡
```
    VectorSet ppin, ppout, constTerm;
    ◇
```
Macro defined by scraps 17a, 21a, 27b, 30ac.
Macro referenced in scrap 8c.

Since the calculation of **ppin** and **ppout** is a backward problem, we will initialize the end block first. Its **constTerm** set is not used, so we set it to **NULL**. Conversely, although the **ppout** set is not used in this data-flow problem, it is used in the calculation of **insertConst**, so we initialize it to the empty set. Since the end block contains no code, we initialize its **ppin** set to be empty. We also set its **dirty** flag to **FALSE** so that the end block will not be processed during the iterative algorithm's loop.

⟨Initialize the data-flow problem for ppin and ppout 28a⟩ ≡

```
{
  Block_Extension extension = end_block->block_extension;

  extension->constTerm = NULL;
  extension->ppout = VectorSet_Create(fourthArena, register_count);
  extension->ppin = VectorSet_Create(block_memory_bank, register_count);
  extension->dirty = FALSE;
}
◇
```

Macro defined by scraps 28ab.
Macro referenced in scrap 27a.

We continue initializing the data-flow problem by iterating over the rest of the blocks, creating and initializing their **constTerm**, **ppin** and **ppout** sets. We also set each block's **dirty** field to **TRUE**. **constTerm** is initialized to **antin** ∩ **pavin**, **ppout** is made to be an empty vector, and **ppin** is initialized to **antloc** ∩ **constTerm**.

⟨Initialize the data-flow problem for ppin and ppout 28b⟩ ≡

```
{
  Block *block;

  ForAllBlocks (block) {
    if (block != end_block) {
      Block_Extension extension = block->block_extension;
      VectorSet constTerm = extension->constTerm
                          = VectorSet_Create(thirdArena, register_count);
      VectorSet ppout = extension->ppout
                      = VectorSet_Create(fourthArena, register_count);
      VectorSet ppin = extension->ppin
                     = VectorSet_Create(block_memory_bank, register_count);

      VectorSet_Intersect(constTerm, extension->antin, extension->pavin);
      VectorSet_Intersect(ppin, extension->antloc, constTerm);
      extension->dirty = TRUE;
    }
  }
}◇
```

Macro defined by scraps 28ab.
Macro referenced in scrap 27a.

The initializations are complete and we may now begin solving the data-flow system. We keep looping over the **dirty** blocks in postorder (this is a backward problem), recalculating **ppin** and **ppout** until the data-flow information stops changing. We declare and create **tempSet** in this outer scope to avoid extra initializations.

⟨Calculate ppin and ppout 29a⟩ ≡
```
{
  Boolean changed;
  VectorSet tempSet = VectorSet_Create(thirdArena, register_count);

  do {
    Block *block;
    changed = FALSE;
    ForAllBlocks_Postorder (block) {
      Block_Extension extension = block->block_extension;

      if (extension->dirty) ⟨Recalculate ppin and ppout for block 29b⟩
    }
  } while (changed);
}◇
```
Macro referenced in scrap 27a.

The recalculation of **ppin** and **ppout** occurs in two steps. First, we calculate **tempSet** as the intersection of the **ppin** sets of the successors of the block. Second, if the value of **tempSet** is different from the old value of **ppout**, we copy **tempSet** into **ppout**. We then note that the data-flow information has **changed** and recompute **ppin** = (**ppout** ∪ **antloc**) ∩ **constTerm**. Then, since **ppin** was changed in this block, we mark all of **block**'s predecessors **dirty**.

At some point in the process, we need to set **dirty** to **FALSE**; we do it at the beginning.

⟨Recalculate ppin and ppout for block 29b⟩ ≡
```
{
  Edge *edge = block->succ;

  extension->dirty = FALSE;
  VectorSet_Copy(tempSet, edge->succ->block_extension->ppin);
  for (edge = edge->next_pred; edge; edge = edge->next_pred)
    VectorSet_Intersect(tempSet, tempSet, edge->succ->block_extension->ppin);

  if (!VectorSet_Equal(tempSet, extension->ppout)) {
    changed = TRUE;
    VectorSet_Copy(extension->ppout, tempSet);
    VectorSet_Union(extension->ppin, extension->ppout, extension->antloc);
    VectorSet_Intersect(extension->ppin,
                        extension->ppin, extension->constTerm);
    Block_ForAllPreds(edge, block)
      edge->pred->block_extension->dirty = TRUE;
  }
}◇
```
Macro referenced in scrap 29a.

5.2 Insertion and deletion information

Now that we are finished finding **ppin** and **ppout**, the next task is to find the insertion and deletion information. Because some the the insertion and deletion data-flow sets depend on other insertion and deletion sets, we well compute these sets in two stages.

5.2.1 Computing insertConst and insertOut

The first step is to find **insertConst** and **insertOut**. We keep these sets in the block extension.

⟨Block extension fields 30a⟩ ≡
```
VectorSet insertConst, insertOut;
◇
```
Macro defined by scraps 17a, 21a, 27b, 30ac.
Macro referenced in scrap 8c.

For each block we create and install **insertConst** and **insertOut**. Then, we set **insertConst** to $\neg$(**ppout** $\cup$ **avout**), and we set **insertOut** to ((**ppin** $\rightarrow$ **altered**) $\cap$ **ppout**) $-$ **avout**, as specified in Equation 1.4.3

⟨Find insertConst and insertOut 30b⟩ ≡
```
{
  Block *block;

  ForAllBlocks (block) {
    Block_Extension extension = block->block_extension;
    VectorSet insertConst = extension->insertConst
                          = VectorSet_Create(block_memory_bank, register_count);
    VectorSet insertOut = extension->insertOut
                        = VectorSet_Create(block_memory_bank, register_count);

    VectorSet_Union(insertConst, extension->ppout, extension->avout);
    VectorSet_Complement(insertConst, insertConst);

    VectorSet_Imply(insertOut, extension->ppin, extension->altered);
    VectorSet_Intersect(insertOut, insertOut, extension->ppout);
    VectorSet_Difference(insertOut, insertOut, extension->avout);
  }
}◇
```
Macro referenced in scrap 27a.

5.2.2 Computing edgeInsert, insertIn and delete

The final major step in finding the placement information is to find **edgeInsert**, **insertIn** and **delete**. We keep **insertIn** and **delete** in the block extension.

⟨Block extension fields 30c⟩ ≡
```
VectorSet insertIn, delete;
◇
```
Macro defined by scraps 17a, 21a, 27b, 30ac.
Macro referenced in scrap 8c.

We keep the **edgeInsert** set in the edge extension. We will create an edge extension for each edge just before the calculation of **edgeInsert**.

⟨Type declarations 31a⟩ ≡
```
struct edge_extension {
  VectorSet edgeInsert;
};
◇
```
Macro defined by scraps 8c, 31a, 40.
Macro referenced in scrap 8a.

To find **edgeInsert**, **insertIn** and **delete**, we iterate over all the blocks. For each block we find **insertIn** (and **edgeInsert** for the block's inedges). Then we find **delete** for the block. Finally, we difference **insertIn** and **antloc**. We do this because the only remaining use of **insertIn** is the insertion of expressions into blocks where **insertIn** − **antloc** is true; we'll just go ahead and do the computation now.

⟨Find insertIn, delete and edgeInsert 31b⟩ ≡
```
{
  Block *block;

  ForAllBlocks (block) {
    Block_Extension extension = block->block_extension;
    VectorSet insertIn = extension->insertIn
                       = VectorSet_Create(block_memory_bank, register_count);

    ⟨Find insertIn and edgeInsert for block 31c, ... ⟩
    ⟨Find delete for block 32b⟩
    VectorSet_Difference(insertIn, insertIn, block->block_extension->antloc);
  }
}◇
```
Macro referenced in scrap 27a.

We set **insertIn** equal to **ppin** intersected with the **insertConst** sets of each of the block's predecessors. While we are inspecting the inedges of the block, we create an edge extension for each inedge and create and install an **edgeInsert** set in each edge extension. We calculate **edgeInsert** to be the **ppin** set of the successor intersected with the **insertConst** set of the predecessor.

⟨Find insertIn and edgeInsert for block 31c⟩ ≡
```
{
  Edge *predEdge;

  VectorSet_Copy(insertIn, extension->ppin);
  Block_ForAllPreds (predEdge, block) {
    Edge_Extension edgeExtension = predEdge->edge_extension
                                 = Arena_GetMem(block_memory_bank,
                                                sizeof(struct edge_extension));
    VectorSet predInsertConst = predEdge->pred->block_extension->insertConst;
    VectorSet edgeInsert = edgeExtension->edgeInsert
                         = VectorSet_Create(block_memory_bank, register_count);

    VectorSet_Intersect(insertIn, insertIn, predInsertConst);
    VectorSet_Intersect(edgeInsert, extension->ppin, predInsertConst);
  }
}◇
```
Macro defined by scraps 31c, 32a.
Macro referenced in scrap 31b.

Recall from Section 1.4 that $\texttt{edgeInsert}_{ij}$ = $\texttt{ppin}_j \cap \texttt{insertConst}_i - \texttt{insertIn}_j$. So far, however, we have only calculated $\texttt{edgeInsert}_{ij}$ = $\texttt{ppin}_j \cap \texttt{insertConst}_i$. Now that we have finished calculating **insertIn**, we can complete our computation of `edgeInsert`. (We extracted this loop so that we could avoid executing it if `insertIn` proved to be empty.)

⟨Find `insertIn` and `edgeInsert` for `block` 32a⟩ ≡

```
{
  Edge *predEdge;

  if (VectorSet_Size(insertIn))
    Block_ForAllPreds (predEdge, block)
      VectorSet_Difference(predEdge->edge_extension->edgeInsert,
                           predEdge->edge_extension->edgeInsert,
                           extension->insertIn);
}◇
```

Macro defined by scraps 31c, 32a.
Macro referenced in scrap 31b.

When we have found `insertIn` and `edgeInsert`, we can find `delete` = `antloc` ∩ `ppin` − `insertIn`.

⟨Find `delete` for `block` 32b⟩ ≡

```
{
  VectorSet delete = block->block_extension->delete
                   = VectorSet_Create(block_memory_bank, register_count);

  VectorSet_Intersect(delete, extension->antloc, extension->ppin);
  VectorSet_Difference(delete, delete, insertIn);
}◇
```

Macro referenced in scrap 31b.

At this point all of the analysis has been done. The only remaining task is to update the code according to the information we have gathered.

Chapter 6

Updating the code

To update the routine, we iterate over the blocks in any convenient order. For each block, we first remove the expressions marked `delete`. Then we insert expressions marked `edgeInsert` in the inedges of the block. Then we insert expressions marked `insertIn` and `insertOut` at the beginning and end of each block, respectively. When we have finished with all the blocks, we call `Block_Order`, a shared routine which updates the block lists (after new blocks were added on some edges). The block lists need to be updated for `Block_Put_All` to work correctly.

⟨Update the code 33⟩ ≡

```
{
  Block *block;

  ForAllBlocks (block) {
    VectorSet delete = block->block_extension->delete;
    VectorSet insertIn = block->block_extension->insertIn;
    VectorSet insertOut = block->block_extension->insertOut;

    ⟨Remove expressions marked delete from the top of block 34a⟩
    ⟨Insert expressions on inedges 34b⟩
    ⟨Insert expressions marked insertIn at the top of block 37b⟩
    ⟨Insert expressions marked insertOut at the bottom of block 37c⟩
  }
  Block_Order();
  Time_Dump(optimizationTime, "  Update took: %1.2f secs\n");
}◇
```

Macro referenced in scrap 9b.

To remove expressions marked `delete`, we iterate through each operation in each instruction in each block. If the operation defines a register and that register is in the `delete` set for the block, we mark the operation non-critical. Since we want to remove only the *first* instance of each expression marked for deletion in each block, we remove the entry from the `delete` set after we mark the operation non-critical.

⟨Remove expressions marked `delete` from the top of `block` 34a⟩ ≡

```
{
  Inst *inst;

  Block_ForAllInsts (inst, block) {
    Operation **operPtr;

    Inst_ForAllOperations (operPtr, inst) {
      Operation *operation = *operPtr;

      if (opcode_specs[operation->opcode].defs == 1 &&
          VectorSet_Member(delete,
                           operation->arguments[operation->referenced])) {
        operation->critical = FALSE;
        VectorSet_Delete(delete, operation->arguments[operation->referenced]);
      }
    }
  }
}◇
```

Macro referenced in scrap 33.

To insert the proper expressions on the inedges of the current block, we first iterate over each edge.[1] If the `edgeInsert` set for the edge has no members, we simply skip to the next edge. If the edge's `edgeInsert` set is nonempty and the control-flow operation at the source of the edge is a `JMPl` or `BR(xx)`, we insert the `edgeInsert` expressions on the edge. If the control-flow operation is a `JMPr`, we cannot insert expressions on the edge (`JMPr` edges are unsplittable), so we add the contents of the `edgeInsert` set to the block's `insertIn` set for later insertion at the top of the block.

⟨Insert expressions on inedges 34b⟩ ≡

```
{
  Edge *edge, *nextEdge;
  VectorSet edgeInsert;

  for (edge = block->pred; edge; edge = nextEdge) {
    nextEdge = edge->next_pred;
    edgeInsert = edge->edge_extension->edgeInsert;
    if (VectorSet_Size(edgeInsert)) {
      Opcode_Names branchOpcode =
        edge->pred->inst->prev_inst->operations[0]->opcode;

      if (branchOpcode != JMPr) {
        ⟨Insert expressions on edge 35⟩
      } else {
        VectorSet_Union(insertIn, insertIn, edgeInsert);
      }
    }
  }
}◇
```

Macro referenced in scrap 33.

[1] We don't use the `Block_ForAllPreds` construct here because the edge list can be modified within the loop.

To put the expressions in `edgeInsert` onto the edge, we first split the edge using the Parser routine `Block_Split_Edge`. This routine creates a new block and inserts it on the given edge, leaving the edge as a predecessor of the new block and creating a new edge to connect the new block to the successor of the original edge. The new block's instruction list contains two instructions, each with one operation: a `NOP` and a `JMPl`.

After we call, `Block_Split_Edge` we need to increment `block_count`, so that our subsequent call to `Block_Order` will work properly. We also need to mark the `JMPl` operation in the new block `critical`, so it won't be deleted by `Block_Print_All`.

For each register `reg` in the `edgeInsert` set of the edge, we call the function `CopyAndInsertOperation` to insert a copy of an expression which defines `reg` into the block which was created by the splitting of the edge. The arguments we pass to `CopyAndInsertOperation` are `expressionDefPoints[reg]` (a pointer an operation which defines `reg`), and `lastInst` (a pointer to the instruction containing the `JMPl` operation in the new block).

`VectorSet_ForAll` iterates over the members of `edgeInsert` in increasing order, so we can be assured that definitions of lower-numbered registers are inserted in the block before the definitions of higher-numbered registers. This is important because the higher-numbered registers may depend on the lower-numbered registers.

⟨Insert expressions on edge 35⟩ ≡

```
{
  Unsigned_Int reg;
  Inst *lastInst;

  Block_Split_Edge(edge);
  block_count++;

  lastInst = edge->succ->inst->prev_inst;
  lastInst->operations[0]->critical = TRUE;

  VectorSet_Forall (reg, edgeInsert) {
    CopyAndInsertOperation(expressionDefPoints[reg], lastInst);
  }
}◇
```

Macro referenced in scrap 34b.

`CopyAndInsertOperation` takes two arguments, an `Operation *` called `operation` and an `Inst *` called `insertBefore`. `CopyAndInsertOperation` makes a replica of the operation, makes a brand new (empty) instruction, inserts the replica as the only operation in the new instruction, then inserts the new instruction into a list of instructions just in front of `insertBefore`.

We begin by creating a new `Inst` structure. We then create an `Operation` structure of the proper size and copy the fields from the given operation to the new structure. We use the local variable `numArgs` to count the number of arguments in the operation; we need `numArgs` to allocate an operation of the proper size and to copy the correct number of arguments.

Finally, we insert the new instruction into the list of instructions (just before **insertBefore**) and insert the new operation into the new instruction.

⟨Functions 36a⟩ ≡

```
Void CopyAndInsertOperation(Operation *operation, Inst *insertBefore)
{
  Inst *newInst = Inst_Allocate(block_memory_bank, 1);
  Operation *newOp;      /* New copy of the operation. */
  Unsigned_Int numArgs;  /* Total number of args for this operation. */

  ⟨Create a new operation newOp of the proper size 36b⟩
  ⟨Copy the fields of the operation structure 37a⟩

  newInst->next_inst = insertBefore;
  newInst->prev_inst = insertBefore->prev_inst;
  newInst->prev_inst->next_inst = newInst;
  insertBefore->prev_inst = newInst;

  newInst->operations[0] = newOp;
  newInst->operations[1] = NULL;
}◇
```

Macro defined by scraps 36a, 41bc, 42.
Macro referenced in scrap 8a.

To create the new operation, we first count the number of arguments to the operation. We know that the **operation->defined** is the total number of constant, referenced and defined arguments. To this we add the number of arguments in lists, which we find by counting each element in each list, including the terminating NULLs. Once we know the number of arguments, we allocate an operation of the proper size, using the parser routine **Operation_Allocate**.

⟨Create a new operation `newOp` of the proper size 36b⟩ ≡

```
{
  Unsigned_Int numLists = opcode_specs[operation->opcode].lists;
  Unsigned_Int i;

  numArgs = operation->defined;
  for (i = 0; i < numLists; i++) {
    while (operation->arguments[numArgs++]);
  }

  newOp = Operation_Allocate(block_memory_bank, numArgs);
}◇
```

Macro referenced in scrap 36a.

We copy the fields of the operation in two phases. First, we copy the named fields, one by one, making sure that the new operation is marked `critical`. Then we copy the vector of `numArgs` arguments.

⟨Copy the fields of the operation structure 37a⟩ ≡

```
{
  Unsigned_Int i;
  Unsigned_Int2 *newArg = newOp->arguments;
  Unsigned_Int2 *oldArg = operation->arguments;

  newOp->opcode = operation->opcode;
  newOp->comment = operation->comment;
  newOp->source_line_ref = operation->source_line_ref;
  newOp->constants = operation->constants;
  newOp->referenced = operation->referenced;
  newOp->defined = operation->defined;
  newOp->critical = TRUE;

  for (i = 0; i < numArgs; i++) {
    *newArg++ = *oldArg++;
  }
}◇
```

Macro referenced in scrap 36a.

We have now finished inserting expression computations on edges. Next, we insert computations of the expressions in `insertIn` at the beginnings of the blocks, using `CopyAndInsertOperation` as before.

⟨Insert expressions marked `insertIn` at the top of `block` 37b⟩ ≡

```
{
  Unsigned_Int reg;
  Inst *firstInst = block->inst->next_inst;

  VectorSet_Forall (reg, insertIn) {
    CopyAndInsertOperation(expressionDefPoints[reg], firstInst);
  }
}◇
```

Macro referenced in scrap 33.

We then insert expressions which are in `insertOut` at the ends of blocks, again making use of the function `CopyAndInsertOperation`.

⟨Insert expressions marked `insertOut` at the bottom of `block` 37c⟩ ≡

```
{
  Unsigned_Int reg;

  Inst *lastInst = block->inst->prev_inst;
  VectorSet_Forall (reg, insertOut) {
    CopyAndInsertOperation(expressionDefPoints[reg], lastInst);
  }
}◇
```

Macro referenced in scrap 33.

We're finished!! The shared routine `Block_Put_All` will print the output to `stdout`.

Appendix A

Memory management

Our implementation of **partial** uses five arenas, four of which we declare globally. We name the arenas in the order in which they are destroyed.

⟨Global variables 38⟩ ≡
```
static Arena firstArena, secondArena, thirdArena, fourthArena;
```
◇

Macro defined by scraps 10b, 13a, 38.
Macro referenced in scrap 8a.

A diagram detailing **partial**'s use of memory appears in Figure A.1. In the diagram, the phase named "Placement information 1" corresponds to the calculation of `ppin` and `ppout`. "Placement information 2" is the calculation of `insertConst` and `insertOut`. "Placement information 3" is calculation of `insertIn`, `edgeInsert` and `delete`.

Note that while arenas 1 through 4 are global variables declared by **partial**, the arena `block_memory_bank` is declared by the shared library, in the header file `Parser.h`. This arena is created by `Block_Init` and is never destroyed. Therefore, we can use `block_memory_bank` to hold data structures which outlive the last point at which data is created, that is, data structures which remain useful after we the beginning of the phase named "Placement Information 3."

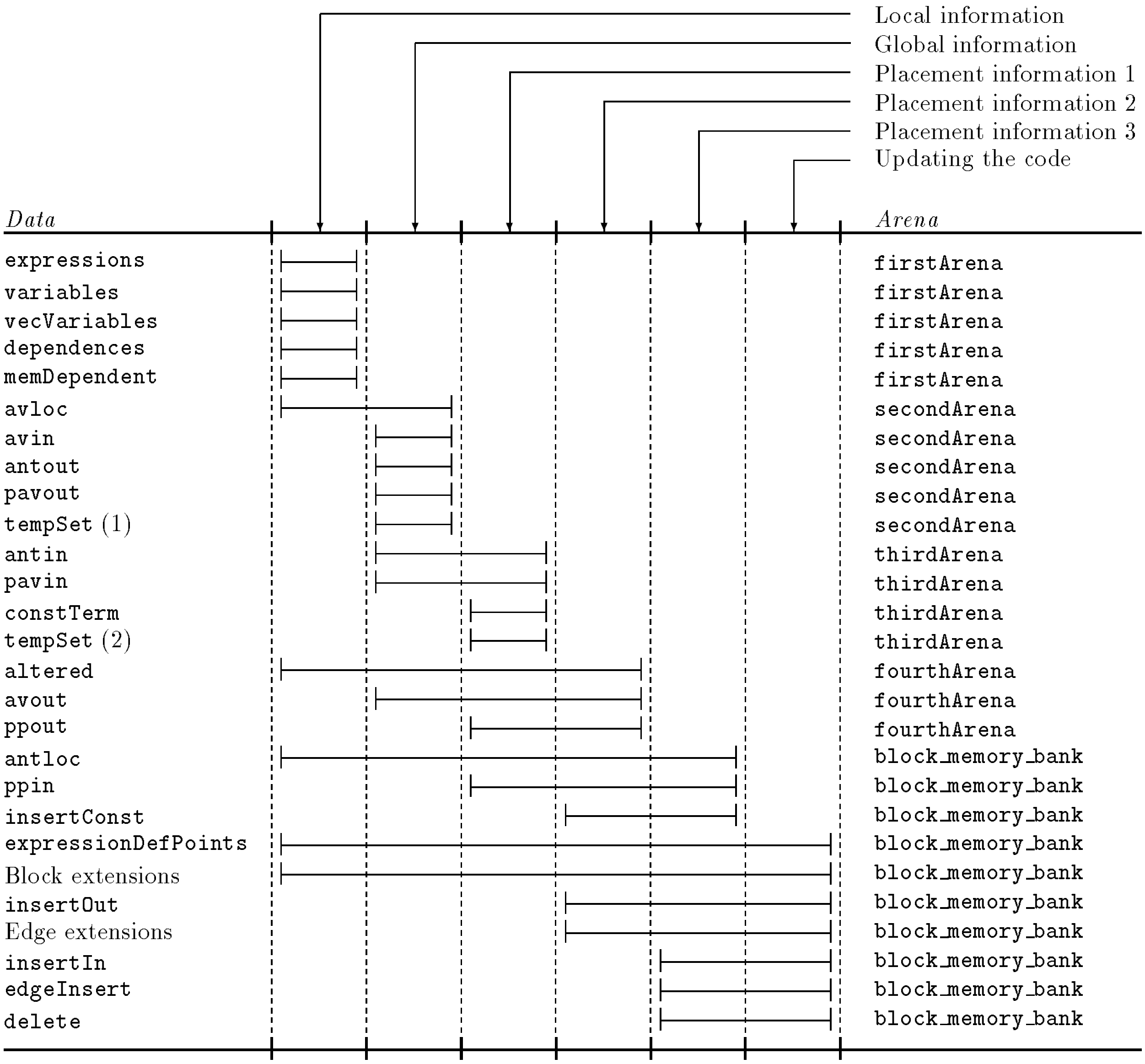

Figure A.1: Memory-use diagram for **partial**.

Appendix B

Monotonically-increasing sets

Although `VectorSet`s provide fast logical operations (such as union, intersect and difference), their *ForAll* construction is notoriously slow.[1] We desire a set implementation that combines the speed of the `VectorSet` logical operations with a *ForAll* operation which runs in time proportional to the number of members in the set.

An easy way to get a fast *ForAll* is to keep a linked list of set members alongside a normal `VectorSet`. For logical operations, we just use regular `VectorSet` function calls. For *ForAll*, we run down the linked list. This strategy, while simple, introduces limitations on the functionality of the set. In particular, it would be expensive to remove arbitrary members from the linked list. However, adding members to the set and taking the union of two sets are still reasonably cheap operations.

Since the expression-register entries of the `dependences` array (see Chapter 3) need only the *Insert*, *Union* and *ForAll* operations, we can speed things up by making `dependences` be an array of our new type of set. The variable-register entries of `dependences` do not need the *ForAll* operation, so we can simply use the `VectorSet` funcion calls – there is no need to keep up the linked list.

We'll call this new kind of set a `LimitedSet`. We'll define the `LimitedSet` structure to contain an ordinary `VectorSet`, as well as a pointer to a list of members, and an `Arena` from which more member structures may be allocated. Each member of the linked list contains an integer `element` and a `next` pointer.

We do not intend to "hide" the fields of the `LimitedSet` structure. We will provide *Insert* and *Union* operators, but if there is no need to keep the linked list in sync with the contents of the `VectorSet` (*i.e.,* if we do not plan to use *ForAll*), then we may freely perform `VectorSet` operations on the `vector` field of the `LimitedSet` structure.

⟨Type declarations 40⟩ ≡
```
typedef struct limitedSet {
  VectorSet vector;
  struct limitedSetElement *list;
  Arena arena;
} *LimitedSet;

typedef struct limitedSetElement {
  Unsigned_Int element;
  struct limitedSetElement *next;
} LimitedSetElement;
```
◇

Macro defined by scraps 8c, 31a, 40.
Macro referenced in scrap 8a.

We provide the *ForAll* mechanism in a fashion similar to that used by `VectorSets`: an iterator macro. However, because `LimitedSet_ForAll` declares a hidden variable in a private scope, each `LimitedSet_ForAll`

[1] $O(n)$, where n is the universe size of the set. Furthermore, the proportionality constants are large, so `VectorSet_ForAll` is slower that a `SparseSet_ForAll` even on dense sets.

must be matched with a `LimitedSet_EndForAll`.

⟨Include files 41a⟩ ≡
```
#define LimitedSet_ForAll(elt,set)                      \
{                                                       \
  LimitedSetElement *___listElement;                    \
  for (___listElement = set->list; ___listElement;      \
       ___listElement = ___listElement->next) {         \
    elt = ___listElement->element;

#define LimitedSet_EndForAll }}
```
◇

Macro defined by scraps 8b, 41a.
Macro referenced in scrap 8a.

We create a set by allocating the set structure, initializing the `VectorSet`, `NULL`ing the list of members and recording the arena.

⟨Functions 41b⟩ ≡
```
static LimitedSet LimitedSet_Create(Arena arena, Unsigned_Int universeSize)
{
  LimitedSet set = Arena_GetMem(arena, sizeof(struct limitedSet));

  set->vector = VectorSet_Create(arena, universeSize);
  set->list = NULL;
  set->arena = arena;

  return set;
}
```
◇

Macro defined by scraps 36a, 41bc, 42.
Macro referenced in scrap 8a.

To insert an element into a `LimitedSet`, we first check the bit vector to see if the element is already a member. If not, we insert it into the bit vector and put it into the linked list.

⟨Functions 41c⟩ ≡
```
static Void LimitedSet_Insert(LimitedSet set, Unsigned_Int element)
{
  if (!VectorSet_Member(set->vector, element)) {
    LimitedSetElement *listElement =
         Arena_GetMem(set->arena, sizeof(LimitedSetElement));

    listElement->element = element;
    listElement->next = set->list;
    set->list = listElement;
    VectorSet_Insert(set->vector, element);
  }
}
```
◇

Macro defined by scraps 36a, 41bc, 42.
Macro referenced in scrap 8a.

To unite two `LimitedSet`s and store the results into a third, we simply insert each element of each source set into the destination set, providing that the source and destination sets are different.

⟨Functions 42⟩ ≡

```
static Void LimitedSet_Union(LimitedSet dest, LimitedSet src1, LimitedSet src2)
{
  Unsigned_Int element;

  if (dest != src1) {
    LimitedSet_ForAll (element, src1) {
      LimitedSet_Insert(dest, element);
    } LimitedSet_EndForAll;
  }

  if (dest != src2) {
    LimitedSet_ForAll (element, src2) {
      LimitedSet_Insert(dest, element);
    } LimitedSet_EndForAll;
  }
}
```

◇

Macro defined by scraps 36a, 41bc, 42.
Macro referenced in scrap 8a.

Appendix C

Debugging output

These pieces of code are called from various strategic points in the program to print out the data sets for debugging purposes. We have split the information into different groups because not all information has the same lifetime. We want to be sure that the information we print actually exists!

⟨Debugging output: Local information 43a⟩ ≡

```
if (debug >= 3) {
  Block *block;

  fprintf(stderr, "***********Local Information **************\n");
  ForAllBlocks (block) {
    fprintf(stderr, "Block: %d.\n", block->preorder_index);

    fprintf(stderr, "\tAltered: "); VectorSet_Dump(block->block_extension->altered);
    fprintf(stderr, "\tAvloc: "); VectorSet_Dump(block->block_extension->avloc);
    fprintf(stderr, "\tAntloc: "); VectorSet_Dump(block->block_extension->antloc);
  }
  fprintf(stderr, "\n");
}◇
```

Macro referenced in scrap 12a.

⟨Debugging output: Global information 43b⟩ ≡

```
if (debug >= 2) {
  Block *block;

  fprintf(stderr, "***********Global Information **************\n");
  ForAllBlocks (block) {
    fprintf(stderr, "Block: %d.\n", block->preorder_index);

    fprintf(stderr, "\tAvin: "); VectorSet_Dump(block->block_extension->avin);
    fprintf(stderr, "\tAntin: "); VectorSet_Dump(block->block_extension->antin);
    fprintf(stderr, "\tPavin: "); VectorSet_Dump(block->block_extension->pavin);
    fprintf(stderr, "\tAvout: "); VectorSet_Dump(block->block_extension->avout);
    fprintf(stderr, "\tAntout: "); VectorSet_Dump(block->block_extension->antout);
    fprintf(stderr, "\tPavout: "); VectorSet_Dump(block->block_extension->pavout);
  }
  fprintf(stderr, "\n");
}◇
```

Macro referenced in scrap 20.

⟨Debugging output: ppin and ppout 44a⟩ ≡

```
if (debug >= 1) {
  Block *block;

  fprintf(stderr, "*********** Ppin, Ppout and ConstTerm **************\n");
  ForAllBlocks (block) {
    fprintf(stderr, "Block: %d.\n", block->preorder_index);

    fprintf(stderr, "\tConstTerm: "); VectorSet_Dump(block->block_extension->constTerm);
    fprintf(stderr, "\tPpin: "); VectorSet_Dump(block->block_extension->ppin);
    fprintf(stderr, "\tPpout: "); VectorSet_Dump(block->block_extension->ppout);
  }
  fprintf(stderr, "\n");
}◇
```

Macro referenced in scrap 27a.

⟨Debugging output: Update sets 44b⟩ ≡

```
if (debug >= 1) {
  Block *block;

  fprintf(stderr, "***********Update Sets **************\n");
  ForAllBlocks (block) {
    fprintf(stderr, "Block: %d.\n", block->preorder_index);

    fprintf(stderr, "\tInsertConst: "); VectorSet_Dump(block->block_extension->insertConst);
    fprintf(stderr, "\tInsertOut: "); VectorSet_Dump(block->block_extension->insertOut);
    fprintf(stderr, "\tInsertIn: "); VectorSet_Dump(block->block_extension->insertIn);
    fprintf(stderr, "\tDelete: "); VectorSet_Dump(block->block_extension->delete);
  }
  fprintf(stderr, "\n");
}◇
```

Macro referenced in scrap 27a.

Appendix D

Indices

D.1 Files

"`partial.c`" Defined by scrap 8a.

D.2 Macros

D.3 Identifiers

D.4 Function prototypes

`CopyAndInsertOperation` takes two arguments, an `Operation *` called `operation` and an `Inst *` called `insertBefore`. `CopyAndInsertOperation` makes a replica of the operation, makes a brand new (empty) instruction, inserts the replica as the only operation in the new instruction, then inserts the new instruction into a list of instructions just in front of `insertBefore`.

⟨Prototypes 48a⟩ ≡

```
static Void CopyAndInsertOperation(Operation *operation, Inst *insertBefore);
```

◇

Macro defined by scraps 48abc, 49.
Macro referenced in scrap 8a.

`LimitedSet_Create` creates and returns a new `LimitedSet` with the given universe size. The memory for the new set is allocated from `arena`; furthermore, subsequent calls to `LimitedSet_Insert` will cause memory to be allocated from `arena`.

⟨Prototypes 48b⟩ ≡

```
static LimitedSet LimitedSet_Create(Arena arena, Unsigned_Int universeSize);
```

◇

Macro defined by scraps 48abc, 49.
Macro referenced in scrap 8a.

`LimitedSet_Insert` inserts an element into a `LimitedSet`.

⟨Prototypes 48c⟩ ≡

```
static Void LimitedSet_Insert(LimitedSet set, Unsigned_Int element);
```

◇

Macro defined by scraps 48abc, 49.
Macro referenced in scrap 8a.

`LimitedSet_Union` unites two sets and puts the result in a destination set. Since `LimitedSet`s are monotonically increasing in nature, the destination set is not cleared before the union operation. Aliasing of the three arguments to `LimitedSet_Union` *is* allowed.

⟨Prototypes 49⟩ ≡

```
static Void LimitedSet_Union(LimitedSet dest, LimitedSet src1, LimitedSet src2);
```

◇

Macro defined by scraps 48abc, 49.
Macro referenced in scrap 8a.

Bibliography

[1] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[2] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.