1 Mem2Reg

1.1 功能简介

编译器为了简化前端设计，往往引入 alloca/load/store 等指令。在处理栈或局部变量时，编译器前端通过 alloca 指令将每个局部变量映射到栈上一个对应的空间，将读写映射为对应空间的 load/store 指令。这种方案简化了前端设计，但是在 IR 上需要进行相关优化时，消除这些不必要的内存操作。消除 alloca/load/store 等操作的优化被称为 Mem2Reg Pass。

```
int main(void) {
    int a;
    a = 1 + 2;
    a = a * 4;
    return 0;
}
```

图 1-1 Mem2Reg 示例代码

观察如图 1-1 所示的代码，前端通过 alloca/load/store 处理局部变量 a，得到图 1-2 的 IR。为了方便，将 IR 中涉及 alloca/store/load 的变量为内存变量，如%op0；其他变量称为寄存器变量，如%op1。在这个代码片段中：

函数“main”为局部变量 int a 分配了一个整数的内存空间（%op0），完成前端 int a 的执行，[%op0 = alloca i32]。

并将 1 + 2 的计算结果存储在其中，完成 a = 1 + 2 语句的执行，[%op1 = add i32 1, 2; store i32 %op1, i32* %op0]。

然后将该值从%op0 中加载到%op2，最终又将%op2 * 4 的结果存入%op0 中，完成前端 a = a * 4 的执行，[%op2 = load i32, i32* %op0; %op3 = mul i32 %op2, 4; store i32 %op3, i32* %op0]。

```
define i32 @main() {
label_entry:
    %op0 = alloca i32
    %op1 = add i32 1, 2
    store i32 %op1, i32* %op0
    %op2 = load i32, i32* %op0
    %op3 = mul i32 %op2, 4
    store i32 %op3, i32* %op0
    ret i32 0
}
```

图 1-2 前端生成的 IR 表示

但是通过分析，不难发现该 IR 代码存在以下问题：

1）存在许多不必要的 alloca/load/store 指令；大量的这些指令，不仅拖慢了程序的运行性能，也导致编译器对 IR 代码的分析更加复杂（需要跟踪记录变量在栈空间上的位置）。

2）不是严格的 SSA 形式，对%op0 的 load 可视为对该标识符的使用，对%op0 的 store 即为对该标识符的定义。该 IR 表示中存在对%op0 的多次 store(定义)，使得该 IR 代码不是标准的 SSA 形式的 IR。

为此编译器引入 Mem2Reg Pass 来消除这些 alloca/load/store 指令，将内存变量提升为寄存器变量。Mem2Reg 是一个转换型 Pass，全称为“memory to register”，旨在消除不必要的内存操作，并用寄存器操作替代它们，并最终获得一个标准的 SSA 形式的 IR。这种转换是有必要，因为寄存器的访问速度远快于内存独写，从而可以有效改善程序的性能。通过分析变

量的使用情况，Mem2Reg 识别出将内存变量提升为寄存器变量的机会。对图 1-2 应用 Mem2Reg 之后，得到图 1-3 所示标准 SSA 形式的 IR 代码。

```
define i32 @main() {
label_entry:
    %op1 = add i32 1, 2
    %op3 = mul i32 %op1, 4
    ret i32 0
}
```

图 1-3 运行 Mem2Reg 之后的 IR

对图 1-2 执行 Mem2Reg 过程大致如下：

1）识别可以优化的 alloca 指令

在这里编译器发现%op0 = alloca i32 是可以被 Mem2Reg Pass 优化的。编译器发现%op0 有一条对应的使用%op2 = load i32, i32* %op0 [line 6]；两条对应的定义，store i32 %op1, i32* %op0 [line 5]和 store i32 %op3, i32* %op0 [line 8]；对于%op0 的多个定义，编译器使用栈的数据结构来记录内存寄存器%op0 的每个定义的生存周期，即每个定义从生成到销毁的范围，实现基于栈的到达定义分析。初始情况如下所示，%op0 对应的栈为空，如图 1-4。

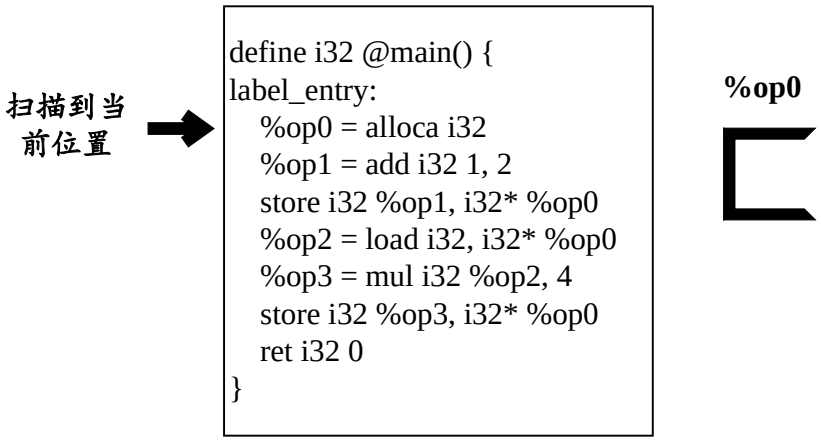


图 1-4 Mem2Reg 运行实例截图 1

接下来扫描该基本块中的所有代码。对于涉及%op0 的 load 指令，则直接将对应 load 值推入对应的栈中。对于涉及%op0 的 store 指令，则直接使用栈顶的元素进行代替。

a）扫描到 store i32 %op1, i32* %op0，将%op1 推入栈中，如图 1-5。

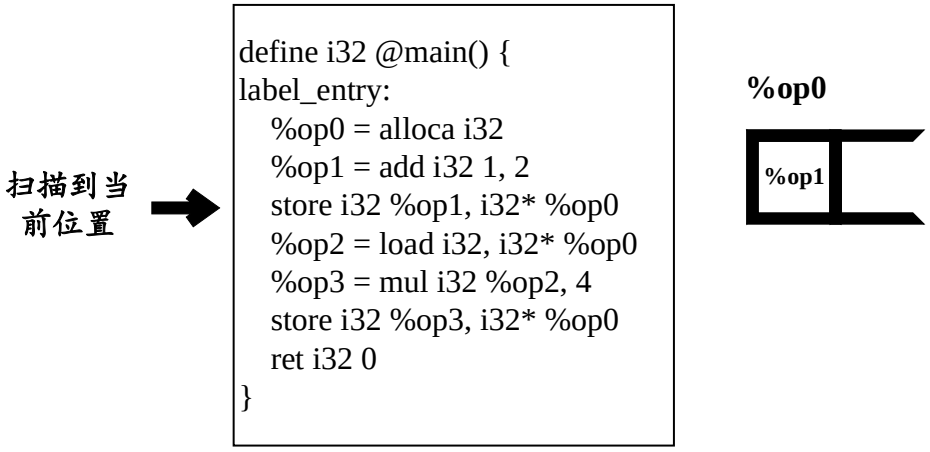


图 1-5 Mem2Reg 运行实例截图 2

b）扫描到%op2 = load i32, i32* %op0 时，则直接用栈顶元素代替%op2。注意这里的代

替不是%op2＝%op1，而是更进一步则将所有用到%op2 的地方直接用%op1 进行代替，即查询%op2 的 DU 链，将所有%op2 的使用替换为%op1，如图 1-6。

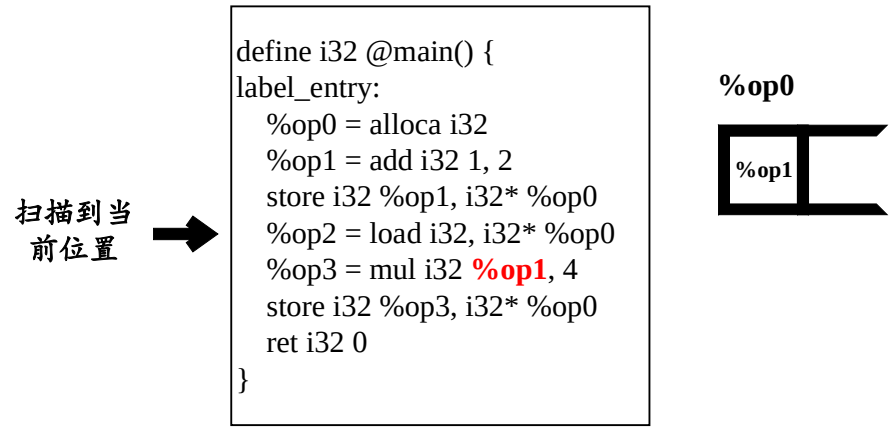


图 1-6 Mem2Reg 运行实例截图 3

c）扫描到 store i32 %op3, i32* %op0，将%op3 推入栈中，如图 1-7。

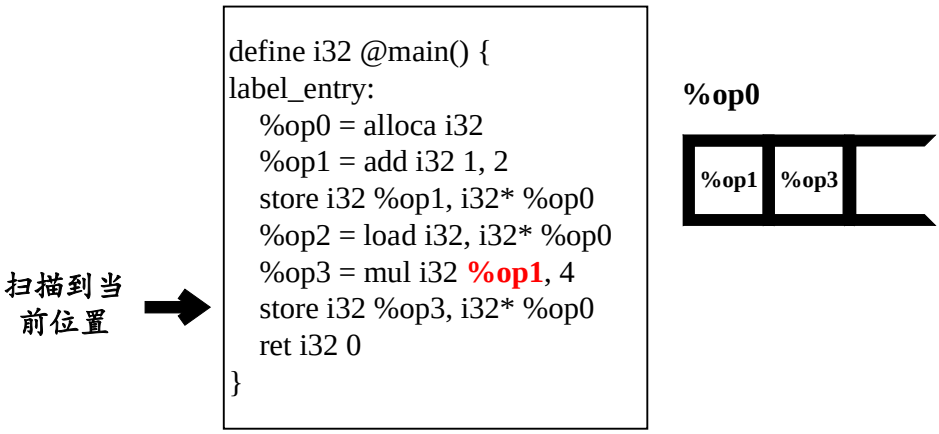


图 1-7 Mem2Reg 运行实例截图 4

2）扫描完毕后，删除%op0 相关的 allocc/load/store 指令，至此编译器完成对内存变量%op0 的提升，得到如图 1-3 所示结果。

对仅涉及单个基本块的 Mem2Reg 是相对较为简单的。但是当内存变量涉及多个基本块、涉及到分支语句时，则情况变得复杂的多。

考虑一个更为常见的例子，如图 1-8。其生成的 IR 如图 1-9。该 IR 中，内存变量%2 的 load/store 涉及多个基本块。特别考虑基本块 6 中的%7＝load i32, i32* %2 指令，由于基本块 5 和基本块 8 都会对%2 进行定义，且都会跳转到基本块 6，从而导致%7＝load i32, i32* %2 的最终结果不确定，所以基于栈的到达定义分析并不能完美解决这种情况，为此编译器引入 phi 函数。

```
int main() {
    int cond;
    int x;
    cond = 1;
    if (cond > 0)
        x = 1;
    else
        x = -1;
```

```
    return x;
}
```

图 1-8 if-else 样例代码

```
define dso_local i32 @main() {
    %1 = alloca i32
    %2 = alloca i32
    store i32 1, i32* %1
    %3 = load i32, i32* %1
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %8

5:
    store i32 1, i32* %2
    br label %6

6:
    %7 = load i32, i32* %2
    ret i32 %7

8:
    %9 = sub i32 0, 1
    store i32 %9, i32* %2
    br label %6
}
```

图 1-9 涉及分支语句的 IR

phi 函数主要作用是在程序的控制流图中的分支节点处，合并不同路径上变量的定义。在 SSA 形式中，每个变量只能有唯一的定义。但是在存在分支语句的情况下，变量可能会在不同的分支路径上具有不同的定义。这时就需要 phi 函数来处理这种情况。对于基本块 6 中的%7 = load i32, i32* %2 则可以使用 phi 函数%7 = phi i32 [1, %5], [%9, %8]进行代替。其含义是，寄存器%7 的值根据控制流赋值。如果控制流是从基本块 5 跳转到基本块 6，则%7 取值 1；如果控制流是从基本块 8 跳转到了基本块 6，则%4 取%9 的值。需要注意一点，基本块内的所有 phi 函数都必须在该基本块的开头，且 phi 函数是并行取值的。编译器通过插入必要的 phi 函数来更好的处理涉及循环和 if 语句的局部变量，从而将内存变量提升为寄存器变量，保证 IR 严格符合 SSA 形式。但是，哪些内存变量需要插入 phi 函数？在哪里插入 phi 函数？插入的 phi 函数填写什么内容？这些问题都将在后续章节进行探讨。

1.2 Mem2Reg 算法执行

Mem2Reg 算法实现通常分为两个步骤：插入 phi 函数和变量重命名。插入 phi 函数为涉及多个基本块的内存变量的生成对应的 phi 函数。变量重命名则在插入 phi 函数之后，确保所有变量都符合 SSA 形式。

1.2.1 插入 phi 函数

先定义两个函数方便进一步讨论：

$J(S)$：S为基本块的集合，$J(S)$为基本块集合S的汇合点（join block）。对于一个基本块集

合 S，如果一个基本块是 S 中的至少两个基本块的汇合点，那么这个基本块是这个集合的汇合点。

$Defs(v)$：v为变量，$Defs(v)$为定义变量v所有基本块的集合。注意，这里不要求 IR 为 SSA 格式，所以变量v可能会存在多个定义。

如果从变量v的一个定义点d出发，存在一条路径到达程序中的另一点p，并且在该路径上不存在对v的其他定义语句，那么称之为对v的定义d能够到达p，说明在p处对v的取值可能是在d处定义的。如果路径上存在一个对v的赋值语句，即在路径上有一个定义点d'也对v进行了定义，那么定义点d就被d'“杀死”了。

如果一个程序中没有一个位置可以被同一个变量的两个及以上的定义到达，那么这个程序满足单一到达定义属性（single reaching-definition property）。在这样的程序中，如果v的定义d能够到达p，那么在p处对v的取值一定是在d处定义的。插入 phi 函数的目标是生成满足单一到达定义的程序。如果在每个基本块的开头对基本块中的变量所使用的变量都分别设置一个 phi 函数，在 phi 函数里合并来自多个不同前驱基本块的对同一个变量的多个定义，显然是满足单一到达定义的要求的，但这样会生成大量不必要的 phi 函数，产生很多的冗余代码，因此算法也要尽可能地减少 phi 函数的数量。

可以发现，只有在来自多个基本块的控制流汇合到一个基本块中时，才会出现同一个变量的多个不同定义，这样的有多个前驱的基本块即为汇合点（join block）。对于一个基本块集合S，如果一个基本块是S中的至少两个基本块的汇合点，那么这个基本块是这个集合的汇合点。基本块集合S的汇合点集记为$J(S)$。

如果基本块n_1和n_2中都对v进行了定义，那么需要在$J(n_1, n_2)$对v设置一个$phi(n_1, n_2)$函数,因为在$J(n_1, n_2)$ 中v在不同的分支路径上具有不同的定义。推广一下，定义变量v的所有基本块集合为$Defs(v)$，需要在$J(Defs(v))$中的所有基本块的开头v设置 phi 函数。不过，请注意，这些新插入的 phi 函数也是对v的定义，它们对应的 phi 函数需要在$J(Defs(v) \cup J(Defs(v))$中设置。不过可以注意到$J(S \cup J(Defs(S)) = J(S)$，因此可以得出结论为：

在$J(Defs(v))$中对v设置 phi 函数，就能在生成满足单一到达定义的程序的同时插入较少的 phi 函数。

计算$J(Defs(v))$非常麻烦，不过可以通过支配边界和迭代支配边界进行计算。在一个基本块x中对变量a进行定义，在不考虑路径中对变量进行重新定义的情况下，所有被x支配的基本块中，a的值一定是x中所定义的值。而对于x的支配边界中的基本块，情况则有所不同：它们的控制流不一定来自于x，a的值只是有可能是x中所定义的值。支配边界所支配的基本块中，当然也无法确定a的值。支配边界是恰好不能确定a是否取x中所定义的值的分界线。

为了便于后续说明，采用n_x符号表示 X 号基本块。例如图 1-11 中，n_4支配n_7的前驱节点n_5，但是n_4不支配n_7，所以n_4的支配边界是$\{n_7\}$。若变量a在n_2被声明，在n_3和n_4中分别被定义为 0 和 1，这样将不能确定在程序执行到n_7时a的值是 0 还是 1。

计算 CFG 中每个节点的支配边界的算法如图 1-10。

```
//计算 CFG 中每个 node 的支配边界
For (a, b) in CFG edges do:
    x <- a;
    while x dose not strictly dominate b do:
        DF(x) = (DF(x) ∪ b);
        x = immediate dominator(x);
```

图 1-10 支配边界算法

以图 1-11，展示图 1-10 支配边界算法的执行过程：

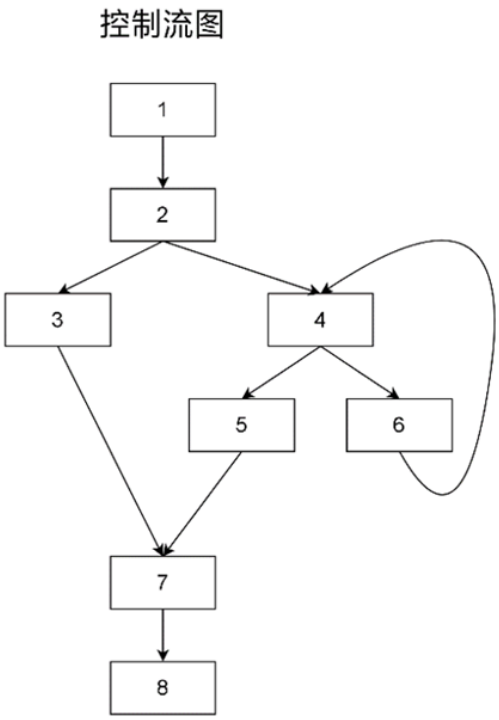


图 1-11 一个简单的控制流图

1）初始化所有基本块的支配边界为空，$DF(*) = \{\}$；

2）首先遍历$n_1 -> n_2$，由于n_1严格支配n_2，所以无任何操作；

3）遍历$n_2 -> n_3$，由于n_2严格支配n_3，所以无任何操作；

4）遍历$n_2 -> n_4$，由于n_2严格支配n_4，所以无任何操作；

5）遍历$n_3 -> n_7$，由于n_3不严格支配n_7，所以n_3的支配边界中含有n_7，$DF(n_3) = \{\} \cup \{n_7\}$；继续寻找$n_3$的直接支配者$n_2$，发现$n_2$严格支配$n_7$，所以跳转到下一条边；

6）遍历$n_4 -> n_5$，由于n_4严格支配n_5，所以无任何操作；

7）遍历$n_4 -> n_6$，由于n_4严格支配n_6，所以无任何操作；

8）遍历$n_5 -> n_7$，由于n_5不严格支配n_7，所以n_5的支配边界中含有n_7，$DF(n_5) = \{\} \cup \{n_7\}$；继续寻找$n_5$的直接支配者$n_4$，发现$n_4$也不严格支配$n_7$，$DF(n_4) = \{\} \cup \{n_7\}$；继续寻找$n_4$的直接支配者$n_2$，发现$n_2$严格支配$n_7$，所以跳转到下一条边；

9）遍历$n_6 -> n_4$，由于n_6不严格支配n_4，所以n_6的支配边界中含有n_4，$DF(n_6) = \{\} \cup \{n_4\}$；继续寻找$n_6$的直接支配者$n_4$，发现$n_4$严格支配自己，所以跳转到下一条边；

10）遍历$n_7 -> n_8$，由于n_7严格支配n_8，所以无任何操作；

11）最终得到所有节点的支配边界：

$$DF(n_3) = DF(n_4) = DF(n_5) = \{n_7\};$$

$$DF(n_6) = \{n_4\};$$

$$DF(n_1) = DF(n_2) = DF(n_7) = DF(n_8) = \{\};$$

为了简单起见，定义一个基本块集合的支配边界是集合中所有基本块的支配边界的并集，即有$DF(A \cup B) = DF(A) \cup DF(B)$，从而定义迭代支配边界。

迭代支配边界（iterated dominance frontier）：节点集S的迭代支配边界$DF^+(S)$是通过迭代地计算支配边界，直到到达一个不动点得到的。

迭代方式为：$DF^+(S) = DF_{i \to \infty}(S)$，$DF_1(S) = DF(S)$，$DF_{i+1}(S) = DF(S \cup DF_i(S))$。

利用迭代支配边界，有$DF^+(S) = J(S \cup \{entry\})$，其中$entry$为程序的入口基本块，即

基本块集合 S 的迭代支配边界为$S \cup \{entry\}$的汇合点。如果变量在$entry$中被定义，有$Defs(v) \cup entry = Defs(v)$，而如果变量在$entry$中没有被定义，这个 phi 函数也可以正常构建，只不过控制流来自$entry$时值为未定义，导致程序可能出现未定义行为。算法默认所有的变量都在入口基本块中被定义一次，因此不会出现未定义行为。回到计算$J(Defs(v))$，即有$J\big(Defs(v)\big) = J(Defs(v) \cup \{entry\}) = DF^{+}(Defs(v))$。所以只需要计算出$Defs(v)$的迭代支配边界，就能够知道要在哪些基本块中插入$v$对应的 phi 函数，而其迭代支配边界可在算法执行过程中迭代计算出来。现在，假设编译器已经有了每个基本块的支配边界，插入 phi 函数的算法如图 1-12。

```
//插入 phi 函数
for v: variable name in original program do:
    F <- {}                    // 已经插入关于变量 v 的 phi 函数的基本块集合
    W <- {}                    // 所有定义了变量 v 的基本块集合
    for d in Defs(v) do
      let B be the basic block containing d   // v 的一个定义 d 所在的基本块
      W <- W ∪ {B}                            // 初始化 W 为 Defs(v)

    while W != {} do
        remove a basic block X from W         // 从 W 中获得一个基本块, X
        for Y : basic block in DF(X) do       // 在 X 的支配边界插入 phi 函数
            if Y not in F then
                add v <- phi(...) at entry of Y   // 插入 phi 函数
                F <- F ∪ {Y}
                if Y not in Defs(v) then //  插入的 phi 函数也是对 v 的定义，
                    W <- W ∪ {Y}          // 这些定义也要插入相应的 phi 函数，
                                          // 因而将 Y 插入到 W 中去。
```

图 1-12 插入 phi 函数算法

以上是一般的插入 phi 函数的过程。算法执行过程潜在的执行了计算迭代支配边界的过程。既有 $DF_{i+1}(W \cup F) = DF(W \cup F \cup DF_i(X))$，其中$X$是$W$中的元素。编译器还可以通过剪枝进一步缩小需插入的 phi 函数的变量的集合。考虑只在单个基本块中活跃的内存变量，其定义和使用只影响当前基本块，不涉及跳转等操作，所以对其无需插入 phi 函数。针对这一现象，编译器可以先计算跨多个基本块的活跃变量的集合，该集合被称为全局名字(global name)集合。随后，编译器只对该集合中的变量执行图 1-12 插入 phi 函数算法，而忽略不在该集合中的内存变量。该方法叫做半剪枝的 phi 函数插入。

在程序中插入 phi 函数后，程序中每个变量依然会有多个定义，但程序中的每个定义都可以确定地到达对它的使用，在程序的任意一个位置，一定能够唯一地找到一个已定义变量的定义点，不会出现“这个变量可能在这里定义，也可能在那里定义”的情况，因为这种情况变成了 phi 函数定义了该变量，kill 了原有的多个定义。

下面，结合实际例子说明插入 phi 函数图 1-12 执行过程。对图 1-9 的 IR 插入 phi 函数算法，算法步骤如下。

1）识别 global name：该 IR 中有两个内存变量分别是%1 = alloca i32 和%2 = alloca i32；进一步发现，只有%2 出现在多个基本块中，因而 global name = {%2}，所以算法只需对%2 插入 phi 函数。下面对%2 插入 phi 函数。

2）初始化 F, W：对%2 的定义即 store 仅出现在基本块 5 和 8 中，所以有 Defs(%2) = {5, 8}。初始化 F = {}, W = {5, 8}。

3） 循环迭代，插入 phi 函数，当 W 不为空时候：

A）从 W 中取出 5，此时 X = 5， W = {8}；在 X 的支配边界 DF(X) = DF(5) = {6} 中插入 phi 函数。

遍历 DF(X)：首先 Y = 6。再 Y 中插入%2 对应的 phi 函数，并用全局唯一的寄存器名字为其命名。此时，该 phi 函数相关参数此时无法填写，在后续重命名过程中进行填写。然后，编译器将 Y 添加到 F 中去，F = {6}；又 Y 不在 W 中，所以此时也将 Y 添加到 W 中去，W = {8, 6}。因为此时 Y 中插入的 phi 函数也是对%2 的定义，也需要插入其对应的 phi 函数。最后，DF(X)所有元素都遍历过，所以遍历结束，此时 IR 如图 1-13 所示，基本块 6 中多出%10 = phi [??, ??], [??, ??]。

```
define dso_local i32 @main() {
    %1 = alloca i32
    %2 = alloca i32
    store i32 1, i32* %1
    %3 = load i32, i32* %1
    %4 = icmp sgt i32 %3, 0
    br i1 %4, label %5, label %8

5:
    store i32 1, i32* %2
    br label %6

6:
    %10 = phi [??, ??], [??, ??]
    %7 = load i32, i32* %2
    ret i32 %7

8:
    %9 = sub i32 0, 1
    store i32 %9, i32* %2
    br label %6
}
```

图 1-13 对基本块 6 插入关于%2 的 phi 函数

b）从 W 中取出 8，此时 X = 8, W = {6}；在 X 的支配边界 DF(X) = DF(8) = {6}中插入 phi 函数。由于 DF(X)中所有基本块已经在 F 中，即已经插入过%2 的 phi 函数，所以本次操作结束。

c）从 W 中取出 6，此时 X = 6, W = {}；在 X 的支配边界 DF(X) = DF(6) = {}中插入 phi 函数。由于 DF(X)为空，所以本次操作结束。

4）当 W 为空，程序执行结束，此时已插入所有内存变量%2 所有必要的 phi 函数。

5）插入 phi 函数执行结束，最终结果即如图 1-13 所示。

1.2.2 变量重命名

为了使程序变为 SSA 形式，还需要进行变量重命名。在该过程中，不仅对变量进行重命名，还利用重命名后变量填写 phi 函数的参数。需要注意的是，在本章节介绍的 Mem2Reg 场景下，仅有内存变量不符合 SSA 形式，所有寄存器变量都符合 SSA 形式（包括新插入的

phi 函数）。利用这一点，编译器可以简化 Mem2Reg 的变量重命名过程，仅对涉及到 alloca/store/load 指令的变量进行重命名。

变量重命名的时候使用栈结构来存储内存变量的值。当对程序进行扫描的时候，碰到 store 指令或者内存变量对应的 phi 函数，则将对应的值推入栈中，即当前的定义杀死的该内存变量之前的定义。当碰到 load 指令时，则将栈顶元素赋予对应的寄存器，因为该栈顶元素是当前唯一可达的定义。需要额外注意的是，对 phi 函数也执行相应的入栈操作。

该过程如图 1-14 所示，算法对 CFG 进行 DFS 遍历。在每个基本块中，算法遍历基本块中的所有指令，对 store 指令和 phi 函数执行入栈操作；对 load 指令使用栈顶元素替换相关寄存器变量的所有使用。当扫描完所有指令之后，填写当前基本块后继的 phi 函数参数。然后，对后继节点递归调用变量重命名函数。最后，当变量重命名函数退出该基本块的时候，恢复内存变量的栈空间状态到函数开始前的初始状态。

```
//变量重命名
foreach Memory Variables i do:   // 为每个内存变量分配一个空的栈空间；
    stack[i] <- Ø;
Rename(entry)                    // entry 为 IR 的入口；
```

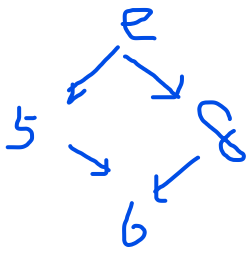
```
def Rename(B: Basic Block):
    foreach instruction in B do:
        // 对于 phi，推入值到相应栈中，杀死之前的定义；
        if instruction is “val <- phi(…, …)” do:
            find the phi function to which Memory Variables i belongs;
            push val onto stack[i];

        // 对于 store，推入值到相应栈中，杀死之前的定义；
        if instruction is “store val, i” do:
            push val onto stack[i];

        // 对于内存变量的 load，直接使用栈顶元素进行替换
        if instruction is “j = load i” do:
            replace all uses of j with top(stack[i])

    // 填写后继基本块中的 phi 函数参数；
    foreach S in successor of B in the CFG do:
        fill in memory variables i’s phi-function parameters with top(stack[i])
    // 递归遍历基本块
    foreach S in successor of B in the dominator tree do:
        Rename(S)
    // 恢复栈空间状态，退出本次调用；
    foreach “store val, i” in B and each i’phi function in B do:
        pop(stack[i])
```

图 1-14 变量重命名算法

对于执行完插入 phi 函数的图 1-13，编译器对其执行变量重命名的过程如下：

识别所有内存变量，并为其创建栈空间：根据 alloca 指令，发现两个内存变量%1 和%2。为其创建对应的栈空间。

开始 DFS 遍历 CFG，遍历顺序为{entry -> 5 -> 6 -> 8}



1）对基本块 entry 执行 Rename

扫描指令，扫描到 store i32 1, i32* %1，执行入栈操作，将 1 推入%1 的栈中；

扫描指令，扫描到%3 = load i32, i32* %1，使用%1 的栈顶元素替换所有%3 的使用，即使用常量 1 替换所有%3 的使用。此时栈中内容如图 1-15。后续没有 load/store/phi 函数，扫描结束。

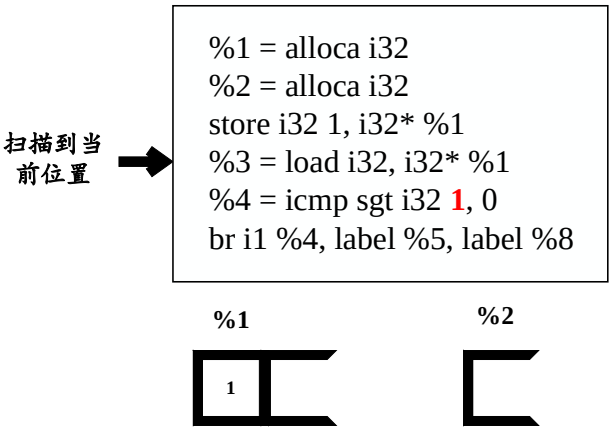


图 1-15 变量重命名运行实例截图 1

填写后继的 phi 函数参数，后继为基本块 5 和基本块 8，它们都没有 phi 函数，跳过；

对后继进行 Rename，此时先对基本块 5 进行 Rename。

2）对基本块 5 执行 Rename：

扫描指令，扫描到 store i32 1, i32* %2，执行入栈操作，将 1 推入%2 的栈中；

扫描指令，后续没有 load/store/phi 函数，扫描结束；

填写后继的 phi 函数，后继为 6，其中存在%2 的 phi 函数%10 = phi [??, ??], [??, ??]。此时使用%2 的栈顶元素对其回填一部分参数，即当从基本块 5 跳转到基本块 6 的时候，寄存器变量%10 取 1。得到结果如图 1-16。

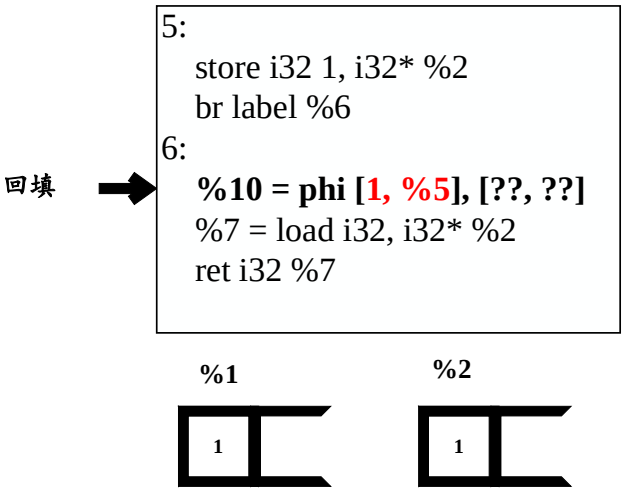


图 1-16 变量重命名运行实例截图 2

对基本块 5 的后继基本块 6 执行 Rename。

3）对基本块 6 执行 Rename。

扫描指令，扫描到%10 = phi [1, %5], [??, ??]，该 phi 函数是%2 对应的 phi 函数，执行入栈操作，将%10 推入%2 的栈中；

扫描指令，扫描到%7 = load i32, i32* %2，所以用%2 的栈顶元素（即%10）替换所有%7 的使用。得到结果如图 1-17，后续没有 load/store/phi 函数，扫描结束。

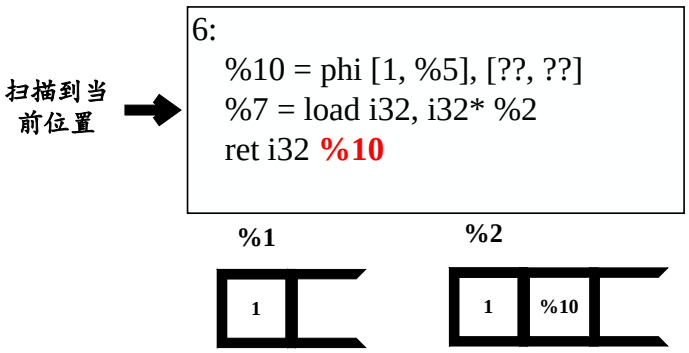


图 1-17 变量重命名运行实例截图 3

基本块 6 中没有任何 load/store/phi 函数，且无后继，所以结束对基本块 6 的 Rename 执行，同时结束基本块 5 的 Rename 执行，恢复内存变量的栈空间为进入基本块 5 时的状态。

4）对基本块 8 执行 Rename。注意此时，递归调用函数是从基本块 entry 进入基本块 8，栈空间如图 1-18。

图 1-18 变量重命名运行实例截图 4

扫描指令，扫描到 store i32 %9, i32* %2，执行入栈操作，将%9 推入%2 的栈中；后续没有 load/store/phi 函数，扫描结束；

填写后继的 phi 函数，后继为基本块 6，其中存在%2 的 phi 函数%10 = phi [1, %5], [??, ??]。此时使用%2 的栈顶元素对其回填一部分参数，即当从基本块 8 跳转到基本块 6 的时候，寄存器变量%10 取%9。得到结果如图 1-19。

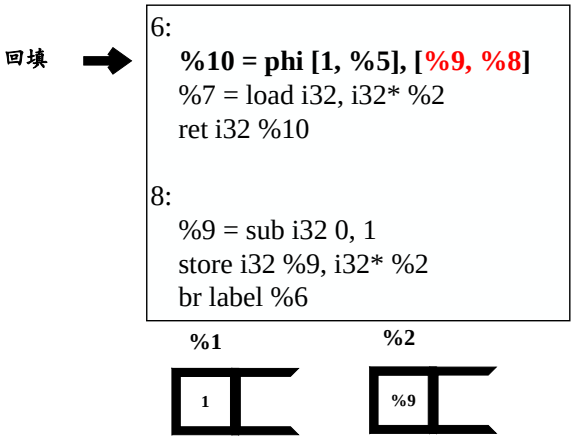


图 1-19 变量重命名运行实例截图 5

对基本块 5 的后继基本块 6 执行 Rename；不过基本块 6 已经执行过 Rename，所以跳过。

5）遍历结束，完成变量重命名。

执行完插入 phi 函数和变量重命名之后，编译器已经完成了 Mem2Reg。此时所有涉及内存变量的 alloca/load/store 指令都可以安全删除。最后，编译器移除所有冗余的指令。对于图 1-9 的 IR，执行完 Mem2Reg 后，其最终代码如图 1-20 所示。

```
define dso_local i32 @main() {
    %4 = icmp sgt i32 1, 0
```

```
    br i1 %4, label %5, label %8

5:
    br label %6

6:
    %10 = phi i32 [ 1, %5 ], [ %9, %8 ]
    ret i32 %10

8:
    %9 = sub i32 0, 1
    br label %6
}
```

图 1-20 Mem2Reg 后 SSA 形式 IR

至此，本小节介绍完了 Mem2Reg 的完整流程。这里做简单总结 Mem2Reg 的三个过程：

1）插入 phi 函数：对涉及多个基本块的内存变量插入 phi 函数，利用迭代支配边界寻找需要插入 phi 函数的基本块。

2）变量重命名：仅有内存变量不符合 SSA 形式，因而对涉及内存变量的 phi 函数以及 load/store 指令和相关寄存器变量进行重命名，确保所有变量都符合 SSA 形式。

3）删除冗余指令：当完成插入 phi 函数和变量重命名之后，所有内存变量已提升为寄存器变量，此时可安全删除涉及内存变量的所有 alloca/load/store 指令。