

第9章 机器无关优化

如果我们简单地把每个高级语言结构独立地翻译成为机器代码，那么会带来相当大的运行时刻的开销。本章讨论如何消除这样的低效率因素。在目标代码中消除不必要的指令，或者把一个指令序列替换为一个完成同样功能的较快的指令序列，通常被称为“代码改进”或者“代码优化”。

局部代码优化(在一个基本块内改进代码)的相关知识已经在 8.5 节介绍过了。本章将处理全局代码优化问题。在全局优化中，代码的改进将考虑在多个基本块内发生的事情。我们将在 9.1 节中讨论一些主要的代码改进机会。

大部分全局优化是基于数据流分析(data-flow analyse)技术实现的。数据流分析技术是一组用以收集程序相关信息的算法。所有数据流分析的结果都具有相同的形式：对于程序中的每个指令，它们描述了该指令每次执行时必然成立的一些性质。不同性质的分析方法各不相同。比如，对于常量传播分析而言，要判断在程序的每个点上，程序使用的各个变量是否在该点上具有唯一的常量值。比如，这个信息可以用于把变量引用替换为常量值。另一个例子是，活跃性分析确定在程序的每个点上，在某个变量中存放的值是否一定会在被读取之前被覆盖掉。如果是，我们就不需要在寄存器或内存位置上保留这个值。

我们将在 9.2 节介绍数据流分析技术。其中还包括几个重要的例子，说明我们如何使用在全局范围内收集到的信息来改进代码。9.3 节将介绍一个数据流框架的总体思想，9.2 节中的数据流分析技术是这个框架的特例。我们实际上可以使用同一个算法来解决这些数据流分析的实例。我们还能够度量这些算法的性能，并且证明它们对所有分析技术的实例而言都是正确的。9.4 节是总体框架的一个例子，它的分析功能比前面的例子更强大。然后，我们将在 9.5 节中考虑一个被称为“部分冗余消除”的功能强大的技术。这个技术可用于优化程序中各个表达式求值的位置。这个问题的解决方案由不同的数据流分析问题的解决方案通过组合而得到。

在 9.6 节，我们将讨论程序中循环的发现和分析。对循环的识别引出了另一个用来解决数据流问题的算法族。这些算法基于一个结构良好的(即可归约的)程序中的循环的层次结构。这个处理数据流分析的方法将在 9.7 节中讨论。最后，在 9.8 节中将使用层次化分析来消除归纳变量(归纳变量本质上就是用来对循环的迭代次数进行计数的变量)。这种代码改进是我们能够对那些由常用程序设计语言书写的程序所做的最重要的改进之一。

9.1 优化的主要来源

编译器的优化必须保持源程序的语义。除了一些非常特殊的场合之外，一旦程序员选择并实现了某种算法，编译器不可能完全理解这个程序并把它替换为一个全然不同且更加高效的等价算法。编译器只知道如何应用一些相对低层的语义转换。在进行转换时，编译器用到一些常见的性质，比如像 $i + 0 = i$ 这样的代数恒等式或使用一些程序语义(如在同样的值上进行同样的运算必然得到同样的结果)。

9.1.1 冗余的原因

在一个典型的程序中会存在很多冗余的运算。有时，在源代码中会用到冗余。比如，程序员可能发现重新计算某些结果会更为直接和方便，而让编译器去发现实际上只需要进行一次这样的计算。但更多的时候，冗余性是使用高级程序设计语言编程的副产品。在大部分程序设计语

言(不包含 C 或者 C ++, 它们允许对指针进行算术运算)中, 程序员别无选择, 只能使用类似于 $A[i][j]$ 或 $X \rightarrow f1$ 的方式来访问一个数组的元素或一个结构的字段。

当一个程序被编译后, 每一个这样的高层数据结构访问都会被扩展成为多个低层次的算术运算, 比如计算一个矩阵 A 的第 (i, j) 个元素的位置的运算。对同一个数据结构的访问通常共享了很多公共的低层运算。程序员不知道这些低层运算, 因此不能自己去消除这些冗余。实际上, 从软件工程的角度看, 程序员只通过数据元素的高层名字来访问它们是比较好的做法。这样, 程序容易书写, 并且更重要的是, 程序更容易理解和演化。通过让一个编译器来消除这些冗余, 我们在两个方面都得到了最好的结果: 程序不仅高效而且易于维护。

9.1.2 一个贯穿本章的例子: 快速排序

接下来, 我们将使用被称为快速排序(quicksort)的排序程序的片断来说明几个重要的可以改进代码的转换。在图 9-1 中的 C 程序是从 Sedgewick[⊖]那里拿来的, 它讨论了如何对这样一个程序进行手工优化。我们将不会在这里讨论这个程序在算法方面的所有精妙细节, 比如, $a[0]$ 必然存放着已经排好序的元素的最小者, 而 $a[max]$ 则存放最大的元素。

```
void quicksort(int m, int n)
    /* 递归地对 a[m] 和 a[n] 之间的元素排序 */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* 片断由此开始 */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* 对换 a[i] 和 a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* 对换 a[i] 和 a[n] */
    /* 片断在此结束 */
    quicksort(m, j); quicksort(i+1, n);
}
```

图 9-1 快速排序算法的 C 代码

在我们可以优化掉地址计算中的冗余之前, 程序中的地址运算首先必须被分解成为低层次的算术运算, 这样才能暴露出冗余之处。在本章的其余部分, 我们假设中间表示形式由三地址语句组成, 其中所有的中间表达式的结果都由临时变量来存放。在图 9-1 中标记出的程序片断的中间代码显示在图 9-2 中。

在这个例子中, 我们假设整数占用 4 个字节。赋值运算 $x = a[i]$ 按照 6.4.4 节中的方法被翻译成为图 9-2 中(14)、(15)步所示的两个三地址语句, 即

```
t6 = 4*i
x = a[t6]
```

类似地, $a[j] = x$ 变成了第(20)和(21)步, 即

```
t10 = 4*j
a[t10] = x
```

请注意, 在原程序中的每个数组访问都被翻译成为一对语句, 其中包含一个乘法和一个数组下标

[⊖] R. Sedgewick, "Implementing Quicksort Programs", *Comm. ACM*, 21, 1978, pp. 847-857.

运算。结果，这个短短的程序片断被翻译成为一个相当长的三地址运算序列。

(1)	i = m-1	(16)	t7 = 4*i
(2)	j = n	(17)	t8 = 4*j
(3)	t1 = 4*n	(18)	t9 = a[t8]
(4)	v = a[t1]	(19)	a[t7] = t9
(5)	i = i+1	(20)	t10 = 4*j
(6)	t2 = 4*i	(21)	a[t10] = x
(7)	t3 = a[t2]	(22)	goto (5)
(8)	if t3 < v goto (5)	(23)	t11 = 4*i
(9)	j = j-1	(24)	x = a[t11]
(10)	t4 = 4*j	(25)	t12 = 4*i
(11)	t5 = a[t4]	(26)	t13 = 4*n
(12)	if t5 > v goto (9)	(27)	t14 = a[t13]
(13)	if i >= j goto (23)	(28)	a[t12] = t14
(14)	t6 = 4*i	(29)	t15 = 4*n
(15)	x = a[t6]	(30)	a[t15] = x

图 9-2 图 9-1 中程序片断的三地址代码

图 9-3 是图 9-2 中的程序的流图。基本块 B_1 是其入口结点。8.4 节介绍过，图 9-2 中所有的条件和无条件跳转语句的目标在图 9-3 中都被替换为以它们的目标语句为首语句的基本块。在图 9-3 中有三个循环。基本块 B_2 和 B_3 本身就是循环。基本块 B_2 、 B_3 、 B_4 、 B_5 一起组成了一个循环，其中 B_2 是唯一的入口结点。

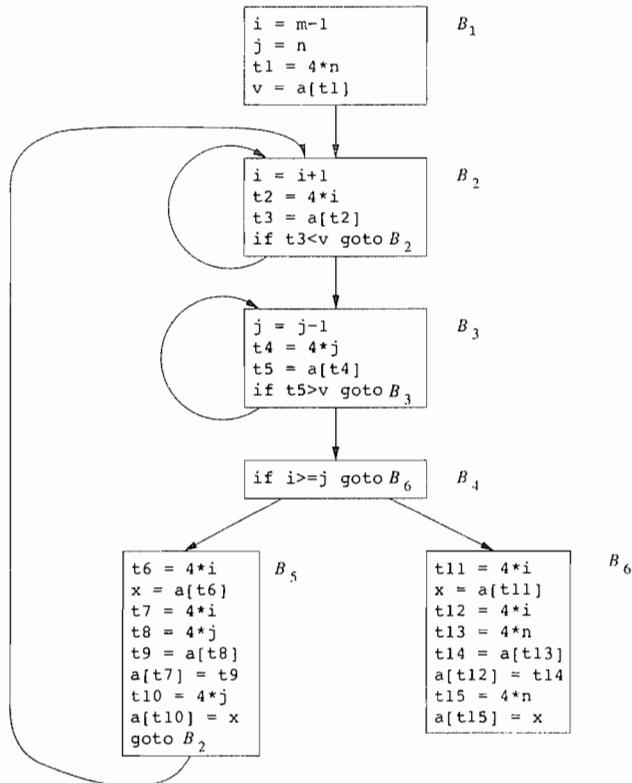


图 9-3 快速排序代码片断的流图

9.1.3 保持语义不变的转换

编译器可以使用很多种方法改进一个程序，但不改变程序所计算的函数。公共子表达式消除、复制传播、死代码消除和常量折叠都是这样的函数不变（或者说语义不变）转换的常见例子。我们将逐一介绍这些方法。

一个程序中经常包含对同一个值的多次计算，比如计算数组中的偏移量。9.1.2节提到过，某些这样的重复计算不可能由程序员来避免，因为这些计算过程处于可在源语言中处理的细节的更下层。比如，在图9-4a中显示的基本块 B_5 中对 $4*i$ 和 $4*j$ 进行了重复计算，尽管这些计算全都不是程序员显式要求的。

9.1.4 全局公共子表达式

如果表达式 E 在某次出现之前已经被计算过，并且 E 中变量的值从那次计算之后就一直没被改变，那么 E 的该次出现就称为一个公共子表达式（common subexpression）。如果将 E 的上一次计算结果赋予变量 x ，且 x 的值在中间没有被改变[⊖]，那么我们就可以使用前面计算得到的值，从而避免重新计算 E 。

例9.1 在图9-4a中对 $t7$ 和 $t10$ 的赋值分别计算了公共子表达式 $4*i$ 和 $4*j$ 。这些步骤已经在图9-4b中被消除了。消除后的代码使用 $t6$ 来替代 $t7$ ，使用 $t8$ 来替代 $t10$ 。

例9.2 图9-5显示了从图9-3中流图的基本块 B_5 和 B_6 中消除全局和局部公共子表达式之后的结果。我们首先讨论对 B_5 的转换，然后再讨论一些和数组相关的精妙之处。

如图9-4b所示，在消除局部公共子表达式之后， B_5 仍然对 $4*i$ 和 $4*j$ 进行求值。它们都是公共子表达式。更明确地讲，使用在 B_3 中计算得到的 $t4$ 的值， B_5 中的三个语句

```
t8 = 4*j
t9 = a[t8]
a[t8] = x
```

可以替换为

```
t9 = a[t4]
a[t4] = x
```

观察一下图9-5，我们会发现当控制流从 B_3 中计算 $4*j$ 的点传递到 B_5 中时， j 和 $t4$ 的值都没有改变。因此，当需要 $4*j$ 时可以使用 $t4$ 来替代。

在用 $t4$ 替换 $t8$ 之后， B_5 中的另一个公共子表达式就显露出来了。新的子表达式是 $a[t4]$ ，对应于源代码层次上的值 $a[j]$ 。当控制流离开 B_3 进入 B_5 时，不仅仅 j 保留了它的值， $a[j]$ 也保留了原来的值。这个值在计算出来之后保存到临时变量 $t5$ 中。因为中间没有对数组 a 中元素的赋值，因此 $a[j]$ 的值不变。 B_5 中的语句

```
t9 = a[t4]
a[t6] = t9
```

可以被替换为

```
a[t6] = t5
```

类似地，可以看出图9-4b的基本块 B_5 中赋给 x 的值和 B_2 中赋给 $t3$ 的值相同。图9-5中的

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B_2
```

a) 消除之前

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B_2
```

b) 消除之后

图9-4 局部公共子表达式消除

⊖ 即使 x 被改变，如果我们把 E 的计算结果同时赋值给变量 x 和另一个新的变量 y ，我们仍然可以用 y 来替代对 E 的计算，从而复用该计算过程。

B_5 是从图 9-4b 的 B_5 中消除了与源代码级表达式 $a[i]$ 和 $a[j]$ 值对应的公共子表达式之后的结果。对于图 9-5 中的 B_6 也进行了一系列类似的转换。

图 9-5 的 B_1 和 B_6 中的表达式 $a[t1]$ 不被认为是公共子表达式，虽然在这两个地方都可以使用 $t1$ 。在控制流离开 B_1 到达 B_6 之前，它还可能经过 B_5 ，而 B_5 中存在对 a 的赋值。因此， $a[t1]$ 到达 B_6 时的值可能和它离开 B_1 时的值有所不同。把 $a[t1]$ 作为一个公共子表达式是不安全的。

□

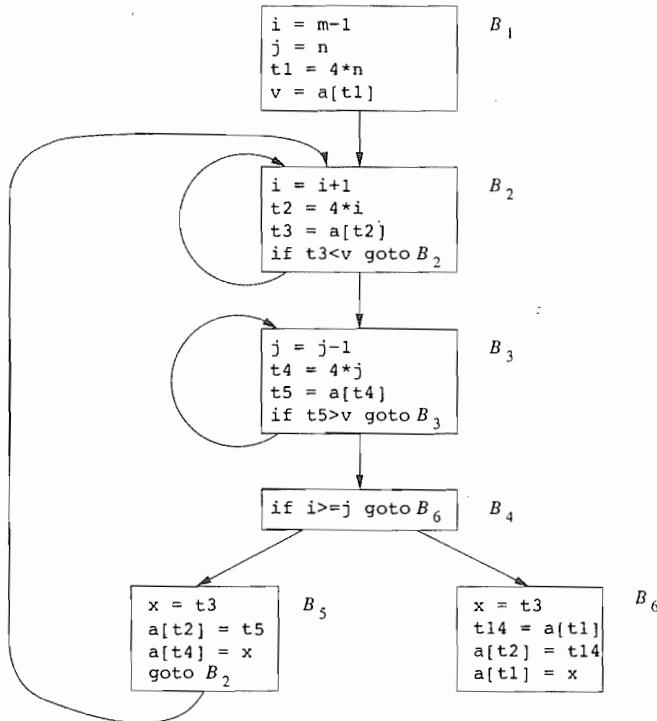


图 9-5 经过公共子表达式消除之后的 B_5 和 B_6

9.1.5 复制传播

图 9-5 中的基本块 B_5 可以通过使用两个新转换来消除 x ，从而得到进一步改进。其中的一个转换考虑形如 $u = v$ 的赋值表达式，这种表达式被称为复制语句 (copy statement)，或者简称复制。只要我们更加细致地考虑例 9.2，很快就会发现一些复制语句。因为常用的公共子表达式消除算法会引入这些复制语句，其他一些优化算法也会引入这样的语句。

例 9.3 为了消除图 9-6a 中的公共子表达式语句 $c = d + e$ ，我们必须使用新的变量 t 来存放 $d + e$ 的值。在图 9-6b 中，赋给变量 c 的是变量 t 的值，而不是表达式 $d + e$ 的值。因为控制流可能经过对 a 的赋值到达语句 $c = b + e$ 处，也可能经过对 b 的赋值到达这里，因此把 $c = d + e$ 替换为 $c = a$ 或 $c = b$ 都是不正确的。

隐藏在复制传播转换之后的基本思想是在复制语句 $u = v$ 之后尽可能地用 v 来替代 u 。比如，图 9-5 的基本块 B_5 中的赋值语句 $x = t3$

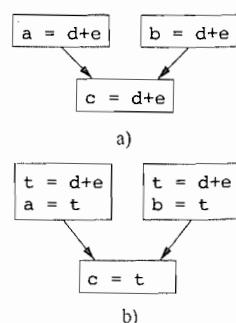


图 9-6 在公共子表达式消除过程中引入的复制语句

是一个复制语句。把复制传播应用于 B_5 会生成图 9-7 中的代码。这个改变看起来可能不像是一个改进，但是，正如我们将在 9.1.6 节看到的，它给了我们消除对 x 赋值的语句的机会。

9.1.6 死代码消除

如果一个变量在某一程序点上的值可能会在以后被使用，那么我们就说这个变量在该点上活跃(live)。否则，它在该点上就是死的(dead)。与此相关的一个想法就是死(或者说无用)代码。所谓死代码就是其计算结果永远不会被使用的语句。程序员不大可能有意引入死代码，死代码多半是因为前面执行过的某些转换而造成的。

例 9.4 假设变量 `debug` 在程序的不同点上被设置为 `TRUE` 或者 `FALSE`，并在如下的语句中使用：

```
if (debug) print ...
```

编译器可能能够推导出这样的结果：每次程序运行到这个语句时，`debug` 的值都是 `FALSE`。通常，出现这种情况的原因是不管程序实际上沿着什么分支运行，在测试 `debug` 的取值之前的最后一个对 `debug` 赋值的语句总是：

```
debug = FALSE
```

如果复制传播把 `debug` 替换为 `FALSE`，那么因为 `print` 语句不可能被运行到，所以它就成为死代码。我们可以把这个测试和 `print` 语句从目标代码中全部消除。更加一般地讲，如果在编译时刻推导出一个表达式的值是常量，就可以使用该常量来替代这个表达式。这个技术被称为常量折叠。□

复制传播的好处之一就是它经常把一些复制语句变成死代码。比如，先进行复制传播再进行死代码消除就可以去掉图 9-7 的代码中对 x 的赋值，并将其转换成为

```
a[t2] = t5
a[t4] = t3
goto B2
```

这个代码是对图 9-5 中的基本块 B_5 的进一步改进。

9.1.7 代码移动

对于优化工作而言，循环(尤其内部循环)是一个重要的地方。因为程序往往将它们的大部分运行时间花费在循环上。如果我们减少一个内部循环中的指令个数，即使因此增加了该循环外的代码，程序的运行时间也可以减少。

减少循环内部代码数量的一个重要改动是代码移动(code motion)。这个转换处理的是那些不管循环执行多少次都得到相同结果的表达式(即循环不变计算)，在进入循环之前就对它们求值。请注意，“在循环之前”的说法假设了存在一个循环入口。所谓循环入口就是一个基本块，所有循环外部到循环的跳转指令都以它为目标(见 8.4.5 节)。

例 9.5 在下面的 `while` 语句中，对 $limit - 2$ 的求值是一个循环不变计算：

```
while (i <= limit-2) /* 不改变 limit 值的语句 */
```

进行代码移动之后将得到如下的等价代码：

```
t = limit-2
while (i <= t) /* 不改变 limit 或 t 值的语句 */
```

现在， $limit - 2$ 的计算只在进入循环之前被执行一次。之前，如果我们重复循环体 n 次，就会对 $limit - 2$ 计算 $n + 1$ 次。□

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

图 9-7 进行复制传播转换后的基本块 B_5

9.1.8 归纳变量和强度消减

另一个重要的优化是在循环中找到归纳变量并优化它们的计算。对于一个变量 x , 如果存在一个正的或负的常数 c 使得每次 x 被赋值时它的值总是增加 c , 那么 x 就称为“归纳变量”。比如, 在图 9-5 中, i 和 $t2$ 都是 B_2 组成的循环中的归纳变量。归纳变量可以通过每次迭代进行一次简单的增量运算(加法或减法)来计算。把一个高代价的运算(比如乘法)替换为一个代价较低的运算(比如加法)的转换被称为强度消减(strength reduction)。但是归纳变量不仅允许我们在适当的时候进行强度消减优化;在我们沿着循环运行时,如果有一组归纳变量的值的变化保持步调一致,我们常常可以将这组变量删剩一个。

在处理循环时,按照“从里到外”的方式进行工作是很有用的。也就是说,我们应该从内部循环开始,然后逐步处理较大的外围循环。这样,我们将看到这个优化是如何从最内层的循环之一(即 B_3)开始被应用到我们的快速排序例子中的。请注意, j 和 $t4$ 的值的步调保持一致;因为 $4 * j$ 被赋给 $t4$, 每次 j 的值减少 1 时 $t4$ 的值就减少 4。变量 j 和 $t4$ 就形成了一个很好的归纳变量对的例子。

当一个循环中存在两个或更多的归纳变量时,有可能只留下一个而删除其他的变量。对于图 9-5 中的内层循环 B_3 , 我们不能把 j 或 $t4$ 完全删除。 $t4$ 在 B_3 中使用, 而 j 在 B_4 中使用。但是, 我们可以用这个例子来说明强度消减优化以及归纳变量消除的部分过程。当考虑由 B_2 、 B_3 、 B_4 、 B_5 组成的外层循环时, j 最终会被消除。

例 9.6 在图 9-5 中, 关系 $t4 = 4 * j$ 在对 $t4$ 赋值之后一定成立, 并且 $t4$ 没有在内层循环 B_3 中的其他地方被改变, 这意味着关系 $t4 = 4 * j + 4$ 在紧跟语句 $j = j - 1$ 之后必然成立。因此我们可以用 $t4 = t4 - 4$ 来替代赋值语句 $t4 = 4 * j$ 。唯一的问题是在我们第一次进入基本块 B_3 时, $t4$ 还没有值。

因为我们必须在进入基本块 B_3 的时候保证关系 $t4 = 4 * j$ 成立, 所以在初始化 j 本身的基本块的尾部放置了一个对 $t4$ 的初始化语句。这个语句在图 9-8 中以附加在基本块 B_1 上的虚线框表示。

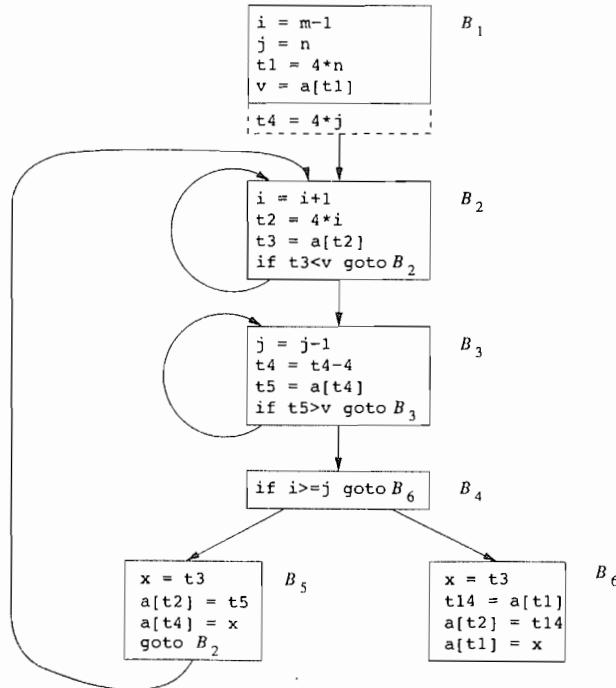


图 9-8 对基本块 B_3 中的 $4 * j$ 应用强度消减优化

虽然我们增加了一个指令，但是它只会在基本块 B_1 中执行一次。只要乘法运算比加法或者减法需要更多的时间，那么把一个乘法运算替换为减法运算就能加快目标代码的执行速度。而这个结论在很多机器上都成立。□

我们用另一个归纳变量消除的例子来结束本节。在这个例子中，我们将在包含了 B_2 、 B_3 、 B_4 和 B_5 的外层循环中处理 i 和 j 。

例 9.7 在强度消减优化被应用到分别环绕 B_2 、 B_3 的两个内部循环之后， i 和 j 的唯一用途是计算基本块 B_4 中的测试的结果。我们知道 i 和 $t2$ 的值满足关系 $t2 = 4 * i$ ，而 j 和 $t4$ 的值满足关系 $t4 = 4 * j$ 。因此，测试 $i \geq j$ 可以被替换为 $t2 \geq t4$ 。一旦进行这个替换， B_2 中的 i 和 B_3 中的 j 就变成了死变量，而在这些基本块中对它们的赋值就变成了可以删除的死代码。最后得到的流图如图 9-9 所示。□

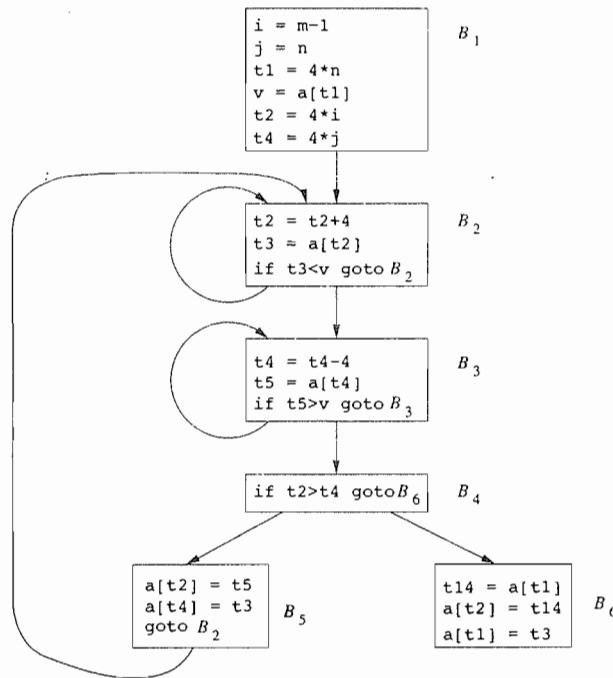


图 9-9 归纳变量消除之后的流图

我们已经讨论的代码改进转换都是很有效的。和图 9-3 中原来的流图相比，图 9-9 中基本块 B_2 和 B_3 中的指令数目由 4 条减少为 3 条。 B_5 中的指令数目由 9 条减少到 3 条，而 B_6 中的指令数目由 8 条减少到 3 条。确实， B_1 中的指令从 4 条指令增长为 6 条指令，但是在这个代码片断中 B_1 只被执行一次，因此总的运行时间几乎不会受到 B_1 的大小的影响。

9.1.9 9.1 节的练习

练习 9.1.1：对于图 9-10 中的流图：

- 1) 找出流图中的循环。
- 2) B_1 中的语句(1)和(2)都是复制语句。其中 a 和 b 都被赋予了常量值。我们可以对 a 和 b 的哪些使用进行复制传播，并把对它们的使用替换为对一个常量的使用？在所有可能的地方进行这种替换。
- 3) 对每个循环，找出所有的全局公共子表达式。

4) 寻找每个循环中的归纳变量。同时要考虑在(2)中引入的所有常量。

5) 寻找每个循环的全部循环不变计算。

练习 9.1.2: 把本节中的转换技术应用到图 8-9 中的流图上。

练习 9.1.3: 把本节中的转换应用到练习 8.4.1 和练习 8.4.2 中得到的流图中去。

练习 9.1.4: 图 9-11 中是用来计算两个向量 A 和 B 的点积的中间代码。尽你所能, 通过下列方式优化这个代码: 消除公共子表达式, 对归纳变量进行强度消减, 消除归纳变量。

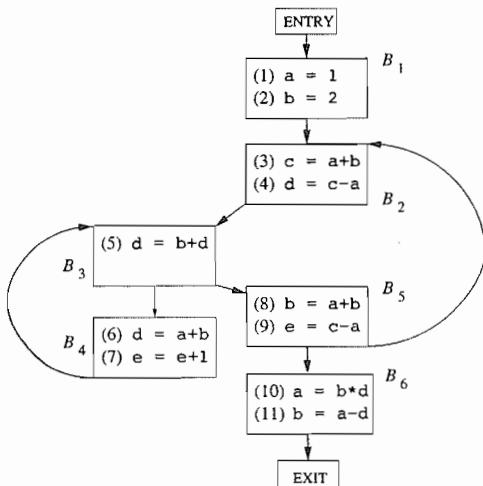


图 9-10 练习 9.1.1 的流图

```

dp = 0.
i = 0
L: t1 = i*8
t2 = A[t1]
t3 = i*8
t4 = B[t3]
t5 = t2*t4
dp = dp+t5
i = i+1
if i < n goto L
    
```

图 9-11 计算点积的中间代码

9.2 数据流分析简介

在 9.1 节中介绍的所有优化都依赖于数据流分析。“数据流分析”指的是一组用来获取有关数据如何沿着程序执行路径流动的相关信息的技术。比如, 实现全局公共子表达式消除的方法之一要求我们确定在程序的任何可能执行路径上, 两个在文字上相同的表达式是否给出相同的值。另一个例子是, 如果某一个赋值语句的结果在任何后续的执行路径中都没有被使用, 那么我们可以把这个赋值语句当作死代码消除。这些以及很多其他重要问题, 都可以通过数据流分析来回答。

9.2.1 数据流抽象

从 1.6.2 节中可知, 程序的执行可以看作是对程序状态的一系列转换。程序状态由程序中的所有变量的值组成, 同时包括运行时刻栈的栈顶之下各个栈帧的相关值。一个中间代码语句的每次执行都会把一个输入状态转换成一个新的输出状态。这个输入状态和处于该语句之前的程序点相关联, 而输出状态和该语句之后的程序点相关联。

当我们分析一个程序的行为时, 我们必须考虑程序执行时可能采取的各种通过程序的流图的程序点序列(“路径”)。然后我们从各个程序点上可能的程序状态中抽取出需要的信息, 用以解决特定数据流分析问题。在更加复杂的分析中, 我们必须考虑调用和返回执行时会形成在不同过程的流图之间跳转的路径。但是, 在我们刚开始研究的时候, 我们将关注穿越单个过程的单个流图的路径。

让我们看一下流图会给出哪些关于可能执行路径的信息。

- 在一个基本块内部, 一个语句之后的程序点和它的下一个语句之前的程序点相同。

- 如果有一个从基本块 B_1 到基本块 B_2 的边，那么 B_2 的第一个语句之前的程序点可能紧跟在 B_1 的最后一个语句后的程序点之后。

这样，我们可以把从点 p_1 到点 p_n 的一个执行路径 (excution path，简称路径) 定义为满足下列条件的点的序列 p_1, p_2, \dots, p_n ：对于每个 $i = 1, 2, \dots, n - 1$ ：

- 1) 要么 p_i 是紧靠在一个语句前面的点，且 p_{i+1} 是紧跟在该语句后面的点。
- 2) 要么 p_i 是某个基本块的结尾，且 p_{i+1} 是该基本块的一个后继基本块的开头。

一般来说，一个程序有无穷多条可能的执行路径，执行路径的长度并没有上界。程序分析把可能出现在某个程序点上的所有程序状态总结为有穷的特性集合。不同的分析技术可以选择抽象掉不同的信息，并且一般来说，没有哪个分析会给出状态的完全表示。

例 9.8 即使是图 9-12 中的简单程序也描述了无限多个执行路径。最短的完全执行路径由程序点 $(1, 2, 3, 4, 9)$ 组成，它不进入任何循环。次短的路径执行一次循环，它由程序点 $(1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)$ 组成。在这个例子中，我们知道在第一次执行程序点 (5) 时，因为 d_1 的定值， a 的值必然是 1。我们说 d_1 在第一次迭代的时候到达了点 (5)。在其后的迭代中， d_3 到达了点 (5)， a 的值是 243。 \square

一般来说，跟踪所有路径上的所有程序状态是不可能的。在数据流分析中，我们并不区分到达一个程序点的路径之间的差异。此外，我们并不跟踪整个状态，而是抽象掉某些细节，只保留进行分析所需要的数据。下面的两个例子将说明一个程序点上的同一个状态可以导出不同的抽象信息。

1) 为了帮助用户调试他们的程序，我们可能希望找出在某个程序点上一个变量可能有哪些值，以及这些值可能在哪里定值。比如，我们可能对在程序点 (5) 上的所有程序状态进行如下总结： a 的值总是 $\{1, 243\}$ 中的一个，而它由 $\{d_1, d_2\}$ 中的一个定值。可能沿着某条路径到达某个程序点的定值称为到达定值 (reaching definition)。

2) 假设我们感兴趣的不是到达定值，而是常量折叠的实现。如果对变量 x 的某次使用只有一个定值可以到达，并且该定值把一个常量赋给 x ，那么我们可以简单地把 x 替换为该常量。另一方面，如果有多个对 x 的定值可以到达某一个程序点，我们就不能对 x 进行常量折叠转换。因此，为了进行常量折叠，我们希望找到这样的定值：对于某个给定的程序点，不管执行哪条路径，它们都是唯一到达该点的对相应变量的定值。对于图 9-12 中的点 (5)，没有哪个定值是到达该点的对 a 的唯一定值，因此对于点 (5) 上的 a 来说，这个集合是空的。即使一个变量在某个点上被唯一定值，该定值必须把一个常量值赋给该变量，才可能进行常量折叠转换。这样，我们可以简单地把某些变量描述成“非常量”，而不是记录它们所有可能的取值，或者所有可能的定值。

因此，我们看到，根据分析的目的，同样的信息可以通过不同的方式进行概括。 \square

9.2.2 数据流分析模式

在所有的数据流分析应用中，我们都会把每个程序点和一个数据流值 (data-flow value) 关联起来。这个值是在该点可能观察到的所有程序状态的集合的抽象表示。所有可能的数据流值的集合称为这个数据流应用的域 (domain)。比如，到达定值的数据流值的域是程序的定值集合的

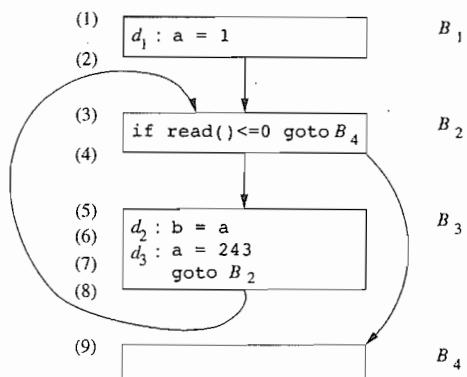


图 9-12 说明数据流抽象的例子程序

所有子集的集合。某个数据流值是一个定值的集合，而我们希望把程序中的每个点和可能到达该点的定值的精确集合关联起来。如上面讨论的，对于抽象方式的选择依赖于分析的目标。考虑到效率问题，我们只跟踪相关的信息。

我们把每个语句 s 之前和之后的数据流值分别记为 $IN[s]$ 和 $OUT[s]$ 。数据流问题 (data-flow problem) 就是要对一组约束求解。这组约束对所有的语句 s 限定了 $IN[s]$ 和 $OUT[s]$ 之间的关系。约束分为两种：基于语句语义 (传递函数) 的约束和基于控制流的约束。

传递函数

在一个语句之前和之后的数据流值受该语句的语义的约束。比如，假设我们的数据流分析涉及确定各个程序点上各变量的常量值。如果变量 a 在执行语句 $b = a$ 之前的值为 v ，那么在该语句之后 a 和 b 的值都是 v 。一个赋值语句之前和之后的数据流值的关系被称为传递函数 (transfer function)。

传递函数有两种风格：信息可能沿着执行路径向前传播，或者沿着执行路径逆向流动。在一个前向数据流问题中，一个语句 s 的传递函数 (通常被记为 f_s) 以语句前的数据流值作为输入，并产生语句之后的新数据流值。也就是

$$OUT[s] = f_s(IN[s])$$

反过来，在一个逆向流问题中，语句 s 的传递函数 f_s 把一个语句之后的数据流值转变成为语句之前的新数据流值。也就是：

$$IN[s] = f_s(OUT[s])$$

控制流约束

第二组关于数据流值的约束是从控制流中得到的。基本块中的控制流很简单。如果一个基本块 B 由语句 s_1, s_2, \dots, s_n 顺序组成，那么 s_i 输出的控制流值[⊖]和输入 s_{i+1} 的控制流值相同。也就是说

$$IN[s_{i+1}] = OUT[s_i] \quad i = 1, 2, \dots, n-1$$

基本块之间的控制流边会生成一个基本块的最后一个语句和后继基本块的第一个语句之间的约束，这些约束更加复杂。比如，如果对可能到达一个程序点的所有定值感兴趣，那么到达一个基本块的首语句的定值的集合就是到达它的各个前驱基本块的最后一个语句之后的定值集合的并集。下一节将给出基本块之间数据流的细节。

9.2.3 基本块上的数据流模式

从技术上讲，数据流模式涉及程序中每个点上的数据流值。但是如果我们认识到基本块内部的数据流处理通常很简单，就可以节约数据流分析所需的时间和空间。控制流从基本块的开始流动到结尾，中间没有中断或者分支。这样，我们就可以用进入和离开基本块的数据流值的方式来重新描述这个模式。对于每个基本块 B ，我们把紧靠其前和紧随其后的数据流值分别记为 $IN[B]$ 和 $OUT[B]$ 。关于 $IN[B]$ 和 $OUT[B]$ 的约束可以按照下面的方法，根据关于 B 中的各个语句 s 的 $IN[s]$ 和 $OUT[s]$ 的约束得到。

假设基本块由语句 s_1, s_2, \dots, s_n 顺序组成。如果 s_1 是基本块 B 的第一个语句，那么 $IN[B] = IN[s]$ 。类似地，如果 s_n 是基本块 B 的最后一个语句，那么 $OUT[B] = OUT[s_n]$ 。基本块 B 的传递函数记为 f_B ，它可以通过将该基本块中各语句的传递函数组合起来获得该传递函数。也就是说，设 f_s 是语句 s_i 的传递函数，那么 $f_B = f_{s_n} \circ \dots \circ f_{s_2} \circ f_{s_1}$ 。该基本块的开头和结尾处的数据流值的关系是

[⊖] 原文如此，但是似乎应该是“数据流值”。——译者注

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

因基本块之间的控制流而产生的约束可以很容易地通过重写得到，把原来约束中的 $\text{IN}[s_1]$ 和 $\text{OUT}[s_n]$ 分别替换为 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 即可。比如，如果一个数据流值表示的是可能被赋予某个变量的常量集合，那么我们就得到一个前向流问题，其中

$$\text{IN}[B] = \bigcup_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$$

我们很快就会在处理活跃变量分析时看到逆向数据流问题。逆向数据流问题的方程是类似的，但是 IN 和 OUT 值的角色被调换了。也就是说：

$$\text{IN}[B] = f_B(\text{OUT}[B])$$

$$\text{OUT}[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S]$$

和线性算术方程不同，数据流方程通常没有唯一解。我们的目标是寻找一个最“精确的”满足这两组约束（即控制流和传递的约束）的解。也就是说，我们需要一个解，它能够支持有效的代码改进，但是又不会导致不安全的转换。这些不安全的转换改变了程序计算的内容。在后面数据流分析中的“保守主义”部分对这个问题进行了简短的讨论，在 9.3.4 节中给出了更加深入的讨论。在下面的小节中，我们将讨论可通过数据流分析解决的问题的某些最重要的例子。

9.2.4 到达定值

“到达定值”是最常见和有用的数据流模式之一。只要知道当控制到达程序中每个点的时候，每个变量 x 可能在程序中的哪些地方被定值，我们就可以确定很多有关 x 的性质。下面仅仅给出两个例子：一个编译器能够根据到达定值信息知道 x 在点 p 上的值是否为常量，而如果 x 在点 p 上被使用，则调试器可以指出 x 是否未经定值就被使用。

如果存在一条从紧随在定值 d 后面的程序点到达某一个程序点 p 的路径，并且在这条路径上 d 没有被“杀死”，我们就说定值 d 到达程序点 p 。如果在这条路径上有对变量 x 的其他定值，我们就说变量 x 的这个定值被“杀死”了[⊖]。直观地讲，如果某个变量 x 的一个定值 d 到达点 p ，在点 p 处使用的 x 的值可能就是由 d 最后定值的。

探测未定值先使用

下面介绍我们如何使用到达定值问题的解来探测未定值先使用的情况。其窍门是在流图的人口处对每个变量 x 引入一个哑定值。如果 x 的哑定值到达了一个可能使用 x 的程序点 p ，那么 x 就可能在定值之前被使用。请注意，我们永远不能绝对肯定这个程序包含一个错误。因为有可能存在某种原因使得到达 p 点而没有真正对 x 赋值的路径实际上并不存在。这个原因可能涉及复杂的逻辑问题。

变量 x 的一个定值是（可能）将一个值赋给 x 的语句。过程参数、数组访问和间接引用都可以有别名，因此指出一个语句是否向特定程序变量 x 赋值并不是件容易的事情。程序分析必须是保守的。如果我们不知道一个语句是否给 x 赋了一个值，我们必须假设它可能对 x 赋值。也就是说，在语句 s 之后，变量 x 的值可能还是 s 执行之前的原值，但也可能变成了 s 所产生的新值。为简单起见，在本章的其余部分我们假设仅仅处理没有别名的程序变量。这类变量包括大多数语言中的局部标量变量。在处理 C 或者 C++ 语言时，有些局部变量的地址会被计算出来，这种局部变量不属于这类变量。

[⊖] 注意，路径中可能包含循环，因此我们可能沿着这条路径到达 d 的另一次出现。这种情况下， d 没有被“杀死”。

例 9.9 图 9-13 中显示的是一个具有 7 个定值的流图。让我们注意观察所有到达基本块 B_2 的定值。所有在 B_1 中的定值都到达了基本块 B_2 的开头。因为在转回基本块 B_2 的循环中找不到其他的对 j 的定值，基本块 B_2 中的定值 $d_5: j = j - 1$ 也可以到达基本块 B_2 的开头。但是，这个定值杀死了定值 $d_2: j = n$ ，使得 d_2 不能到达 B_3 和 B_4 。 B_2 中的语句 $d_4: i = i + 1$ 却不能到达 B_2 的开头，这是因为变量 i 总是被 $d_7: i = u3$ 重新定值。最后，定值 $d_6: a = u2$ 也能够到达 B_2 的开头。

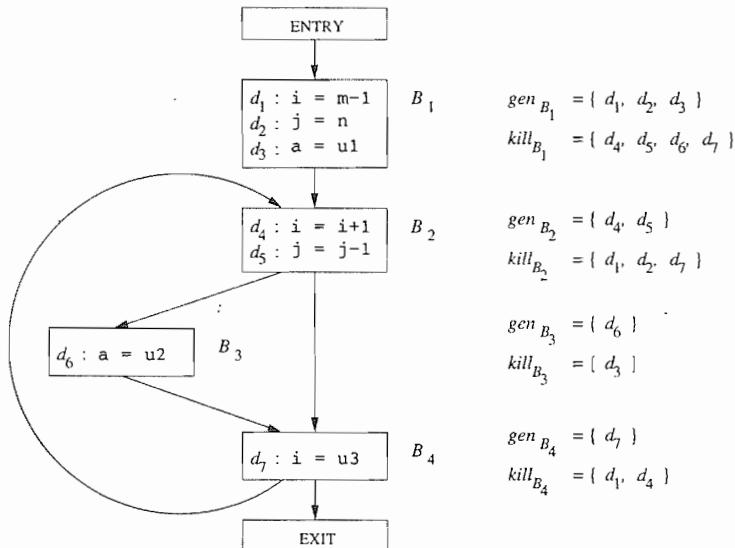


图 9-13 演示到达定值的流图

我们在前面定义到达定值时，有时允许一定的不精确性。但是它们都是在“安全”或者说“保守”的方向上不精确。比如，请注意我们假设一个流图的所有边都可以通过。在实践中这个假设可能是不正确的。再比如，在下面的程序片断中，没有哪个 a 和 b 的取值可以使得控制流真的能够到达 statement 2：

```
if (a == b) statement 1; else if (a == b) statement 2;
```

在一般情况下，决定一个流图的每条路径是否都可以被执行是一个不可判定问题。因此，我们简单地假设流图中的每条路径都可能在程序的某次执行时通过。在大部分到达定值的应用中，在一个定值不可能到达某点的情况下假设其能够到达是保守的。因此，我们可以允许那些在程序实际执行中根本不会被遍历的路径，我们也可以安全地允许定值穿越某个对同一变量的不确定定值。

数据流分析中的保守主义

实际数据流值是通过程序的所有可能执行路径来定义的。所有的数据流模式计算得到的都是对实际数据流值的估算。我们必须保证所有的估算误差都在“安全”的方向上。如果一个策略性决定不允许我们改变程序计算出的内容，它就被认为是“安全的”（或者说“保守的”）。遗憾的是，安全的策略会让我们错失一些能够保持程序含义的代码改进机会。但实际上对所有的代码优化技术而言，没有哪个安全的策略可以不错失任何机会。使用不安全策略就是以改变程序含义的代价来加快代码速度。一般来说，这是不可接受的。

因此在设计一个数据流模式的时候,我们必须知道这些信息将如何被使用,并保证我们做出的任何估算都是在“保守”或者说“安全”的方向上。每个模式和应用都要单独考虑。比如,如果我们把到达定值信息用于常量折叠,那么把一个实际不可到达的定值当作可到达就是安全的(我们可能在 x 实际是一个常量且可以被折叠的情况下认为 x 不是一个常量),但是把一个实际可到达的定值当作不可到达就是不安全的(我们可能把 x 替换为一个常量,但是实际上程序有时会赋予 x 一个不同于该常量的值)。

到达定值的传递方程

现在我们为到达定值问题设置约束。我们首先检查单个语句的细节。考虑一个定值

$$d: u = v + w$$

在这里, + 号代表了一个一般性的二元运算符。以后我们经常会这么做。

这个语句“生成”了一个变量 u 的定值 d , 并“杀死”了程序中其他对 u 的定值, 而进入这个语句的其他定值都没有受到影响。因此, 定值 d 的传递函数可以被表示为

$$f_d(x) = \text{gen}_d \cup (x - \text{kill}_d) \quad (9.1)$$

其中 $\text{gen}_d = \{d\}$, 即由这个语句生成的定值的集合, 而 kill_d 是程序中所有其他对 u 的定值。

我们在 9.22 节讨论过, 一个基本块的传递函数可以通过把它包含的所有语句的传递函数组合起来而构造得到。下面我们会看到, 形如(9.1)的函数的组合仍然是这种形式。我们把这种形式称为“生成 - 杀死形式”。假设有两个函数 $f_1(x) = \text{gen}_1 \cup (x - \text{kill}_1)$ 和 $f_2(x) = \text{gen}_2 \cup (x - \text{kill}_2)$ 。那么

$$\begin{aligned} f_2(f_1(x)) &= \text{gen}_2 \cup (\text{gen}_1 \cup (x - \text{kill}_1) - \text{kill}_2) \\ &= (\text{gen}_2 \cup (\text{gen}_1 - \text{kill}_2)) \cup (x - (\text{kill}_1 \cup \text{kill}_2)) \end{aligned}$$

这个规则可以扩展到由任意多个语句组成的基本块。假设基本块 B 有 n 个语句, 而第 i 个语句的传递函数为 $f_i(x) = \text{gen}_i \cup (x - \text{kill}_i)$, $i = 1, 2, \dots, n$, 那么基本块 B 的传递函数可以写成:

$$f_B(x) = \text{gen}_B \cup (x - \text{kill}_B)$$

其中

$$\text{kill}_B = \text{kill}_1 \cup \text{kill}_2 \cup \dots \cup \text{kill}_n$$

而

$$\begin{aligned} \text{gen}_B &= \text{gen}_n \cup (\text{gen}_{n-1} - \text{kill}_n) \cup (\text{gen}_{n-2} - \text{kill}_{n-1} - \text{kill}_n) \cup \\ &\dots \cup (\text{gen}_1 - \text{kill}_2 - \text{kill}_3 - \dots - \text{kill}_n) \end{aligned}$$

因此, 和单个语句一样, 一个基本块也会生成一个定值集合并杀死一个定值集合。集合 gen 中包含了所有在紧靠基本块之后的点上“可见”的该基本块中的定值——我们把它们称为“向下可见”(downwards exposed)的。在一个基本块中, 一个定值是向下可见的, 仅当它没有被同一个基本块中较后的对同一变量的定值“杀死”。一个基本块的 kill 集就是所有被块中各个语句杀死的定值的集合。请注意, 一个定值可能同时出现在基本块的 gen 集和 kill 集中。在这种情况下, 该定值会被这个基本块生成, 即优先考虑该定值是否在 gen 集中。这是因为在 gen-kill 形式中, kill 集会在 gen 集之前被使用。

例 9.10 基本块

$$\begin{aligned} d_1: \quad a &= 3 \\ d_2: \quad a &= 4 \end{aligned}$$

的 gen 集是 $\{d_2\}$, 因为 d_1 不是向下可见的。基本块的 kill 集包括了 d_1 和 d_2 , 因为 d_1 杀死了 d_2 , d_2 杀死了 d_1 。虽然如此, 因为减去 kill 集的运算先于和 gen 集的并集运算, 这个基本块的传递函

数的结果中总是包含定值 d_2 。 □

控制流方程

下面我们考虑根据基本块之间的控制流得到的约束集合。因为只有一个定值能够沿着至少一条路径到达某个程序点，那么这个定值就到达该程序点，所以只要从 P 到 B 有一条控制流边， $\text{OUT}[P] \subseteq \text{IN}[B]$ 就成立。然而，一个定值到达某个程序点的必要条件是它能够沿着某条路径到达这个程序点，因此 $\text{IN}[B]$ 不应该大于 B 的所有前驱基本块出口点的到达定值的并集。也就是说，可以安全地假设如下的方程式成立：

$$\text{IN}[B] = \bigcup_{P \text{ 是 } B \text{ 的一个前驱基本块}} \text{OUT}[P]$$

我们把并集运算称为到达定值的交汇运算 (meet operator)。在任何数据流模式中，我们用交运算来汇总各条路径会合点上不同路径所作的贡献。

到达定值的迭代算法

我们假设每个控制流图都有两个空基本块，包括代表了这个图的开始点的 ENTRY 结点以及 EXIT 结点，所有离开这个图的控制流都流向它。因为没有定值到达这个图的开始，所以基本块 ENTRY 的传递函数是一个简单的返回空集 \emptyset 的常函数，即 $\text{OUT}[\text{ENTRY}] = \emptyset$ 。

到达定值问题使用下面的方程定义：

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

且对于所有的不等于 ENTRY 的基本块 B ，有

$$\text{OUT}[B] = \text{gen}_B \cup (\text{IN}[B] - \text{kill}_B)$$

$$\text{IN}[B] = \bigcup_{P \text{ 是 } B \text{ 的一个前驱基本块}} \text{OUT}[P]$$

可以使用下面的算法来求这个方程组的解。这个算法的结果是这个方程组的最小不动点 (least fixedpoint)，即对于各个 IN 和 OUT，这个解给出的值总是此方程组的其他解所给出的值的子集。下面这个算法的结果是可接受的，因为在某个 IN 或 OUT 集中的定值确实可以到达该 IN 或 OUT 所描述的程序点。这个解也是我们所期望的，因为它没有包含任何我们确定不会到达的定值。

算法 9.11 到达定值。

输入：一个流图，其中每个基本块 B 的 kill_B 集和 gen_B 集都已经计算出来。

输出：到达流图中各个基本块 B 的入口点和出口点的定值的集合，即 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 。

方法：我们使用迭代的方法来求解。一开始，我们“估计”对于所有基本块 B 都有 $\text{OUT}[B] = \emptyset$ ，并逐步逼近想要的 IN 和 OUT 的值。因为我们必须不停迭代直到各个 IN 值（因此各个 OUT 值也）收敛，所以我们可以用一个布尔变量 $change$ 来记录每次扫描各基本块时是否有 OUT 值发生改变。但是，在此算法及以后描述的类似算法中，我们假设用来跟踪变更情况的确切机制是可理解的，因此我们删除了这些细节。

图 9-14 中粗略地给出了这个算法。前两行对某些数据流值进行了初始化[⊖]。从第(3)行开始是一个循环。在循环中我们不停地迭代直到各个值收敛。第(4)行到第(6)行组成的内层循环对人口结点之外的所有基本块应用数据流方程。 □

直观地讲，算法 9.11 尽量向前传播各个定值，直到该定值被杀死，这样做模拟了程序的所有可能的执行情况。算法 9.11 最终必然会终止，因为对于每个 B ， $\text{OUT}[B]$ 绝对不会变小。一旦某

[⊖] 细心的读者可能会注意到，可以很容易把(1)、(2)两行合并。但是，在类似的数据流算法中，初始化人口结点或出口结点时用的方法可能和初始化其他结点的方法不同。因此我们依照所有的迭代算法的模式，即像行(1)那样应用“边界条件”的动作，与行(2)中的初始化动作分开进行。

一个定值被加入到 OUT 值中，它会一直待在那里。（见练习 9.2.6。）因为所有定值的集合是有限的，最终必然有一趟 while 循环的执行没有向任何 OUT 加入任何内容。此时算法就终止了。在此时终止迭代是安全的，因为如果各个 OUT 值没有改变，下一趟中各个 IN 值也不会改变。而如果各个 IN 值没有改变，OUT 值也不会改变，如此下去，所有后续的迭代都不会改变 IN 和 OUT 的值。

流图中的结点个数是 while 循环的迭代次数的上界。其理由是如果一个定值能够到达某个程序点，它必然可以通过无环的路径到达该点，而一个流图中的结点个数是无环路径中结点数的上界。在 while 循环的每次迭代中，每个定值至少沿着问题中的路径前进一个结点。而且，根据各个结点在内层循环中被访问的顺序，它经常一次前进多个结点。

实际上，如果我们适当地安排第(4)行中 for 循环访问基本块的顺序，经验表明 while 循环的平均迭代次数小于 5（见 9.6.7 节）。因为定值的集合可以使用位向量表示，而这些集合的运算可以使用位向量上的逻辑运算来实现，算法 9.11 在实际应用中出奇地高效。

例 9.12 我们将使用位向量来表示图 9-13 中的七个定值 d_1, d_2, \dots, d_7 。其中左起第 i 个位表示 d_i 。集合的并运算通过相应的位向量的逻辑 OR 运算实现。两个集合的差 $S-T$ 的计算方法是首先计算 T 的位向量的补，然后再将这个补和 S 的位向量进行逻辑 AND 运算。

图 9-15 中显示的是算法 9.11 中的 IN 和 OUT 集的取值。其初始值用上标 0 表示，如 $\text{OUT}[B]^0$ 。它们由图 9-14 中的第(2)行的循环赋值。它们都是空集，用比特向量 000 0000 表示。算法的后续迭代中的取值也使用上标表示，第一趟迭代的值标记为 $\text{IN}[B]^1$ 和 $\text{OUT}[B]^1$ ，第二趟迭代的值标记为 $\text{IN}[B]^2$ 和 $\text{OUT}[B]^2$ 。

假设第(4)行到第(6)行的 for 循环在执行时， B 依次取值

$B_1, B_2, B_3, B_4, \text{EXIT}$

当 $B = B_1$ 时，因为 $\text{OUT}[\text{ENTRY}] = \emptyset$ ，所以 $\text{IN}[B_1]^1$ 是空集，而 $\text{OUT}[B_1]^1$ 等于 gen_{B_1} 。这个值和前面的值 $\text{OUT}[B_1]^0$ 不同，因此我们知道在第一轮中有些值发生了变化（因此会继续进行第二次循环）

然后我们考虑 $B = B_2$ ，并计算

$$\begin{aligned}\text{IN}[B_2]^1 &= \text{OUT}[B_1]^1 \cup \text{OUT}[B_4]^0 \\ &= 111\ 000 + 000\ 0000 = 111\ 0000 \\ \text{OUT}[B_2]^1 &= \text{gen}_{B_2} \cup (\text{IN}[B_2]^1 - \text{kill}_{B_2}) \\ &= 000\ 1100 + (111\ 0000 - 110\ 0001) = 001\ 1100\end{aligned}$$

这个计算过程在图 9-15 中做了概括。比如，在第一趟循环的最后， $\text{OUT}[B_2]^1 = 001\ 1100$ ，反映了 d_4 和 d_5 在 B_2 中生成的事实，而 d_3 到达了 B_2 的开头但是没有在 B_2 中被杀死。

请注意，在第二轮之后， $\text{OUT}[B_2]$ 的值有所改变，反映了 d_6 也到达 B_2 的开头且没有被 B_2 死亡。在第一趟中我们没有了解到这个事实，因为从 d_6 到 B_2 结尾的路径（即 $B_3 \rightarrow B_4 \rightarrow B_2$ ）没有在一趟中被顺序经过。也就是说，当我们知道 d_6 到达 B_4 的结尾时，我们已经在第一趟中计算了 $\text{IN}[B_2]$ 和 $\text{OUT}[B_2]$ 。

在第二趟之后，OUT 集合中的所有值都没有改变。因此，算法在第三趟之后终止。此时，各个 IN 和 OUT 的值如图 9-15 中最后两列所示。□

```

1) OUT[ENTRY] = ∅;
2) for (除 ENTRY 之外的每个基本块 B) OUT[B] = ∅;
3) while (某个 OUT 值发生了改变)
4)   for (除 ENTRY 之外的每个基本块 B) {
5)     IN[B] = ∪P 是 B 的一个前驱 OUT[P];
6)     OUT[B] = gen_B ∪ (IN[B] - kill_B);
}

```

图 9-14 计算到达定值的迭代算法

Block B	$\text{OUT}[B]^0$	$\text{IN}[B]^1$	$\text{OUT}[B]^1$	$\text{IN}[B]^2$	$\text{OUT}[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

图 9-15 IN 和 OUT 的计算过程

9.2.5 活跃变量分析

有些代码改进转换所依赖的信息是按照程序控制流的相反方向进行计算的，我们现在将要研究这样的一个例子。在活跃变量分析 (live-variable analysis) 中，我们希望知道对于变量 x 和程序点 p ， x 在点 p 上的值是否会在流图中的某条从点 p 出发的路径中使用。如果是，我们就说 x 在 p 上活跃；否则就说 x 在 p 上是死的。

活跃变量信息的重要用途之一是为基本块进行寄存器分配。在 8.6 节和 8.8 节中已经介绍了这个问题的某些方面。在一个值被计算并保存到一个寄存器中后，它很可能在基本块中使用。如果它在基本块的结尾处是死的，就不必在结尾处保存这个值。另外，在所有寄存器都被占用时，如果我们还需要申请一个寄存器的话，那么应该考虑使用一个存放了已死亡的值的寄存器，因为这个值不需要保存到内存。

这里我们直接以 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 的方式定义数据流方程。 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 分别表示在紧靠基本块 B 之前和紧随 B 之后的点上的活跃变量集合。这些方程可以通过以下的方法得到：首先定义各个语句的传递函数，然后再把它们组合起来得到一个基本块的传递函数。我们给出下面的定义：

1) def_B 是指如下变量的集合，这些变量在 B 中的定值（即被明确地赋值）先于任何对它们的使用。

2) use_B 是指如下变量的集合，它们的值可能在 B 中先于任何对它们的定值被使用。

例 9.13 比如，图 9-13 中的基本块 B_2 一定使用了 i 。除非 i 和 j 互为对方的别名，否则会在对 j 的任何重新定值之前使用 j 。假设图 9-13 中的变量之间没有别名关系，那么 $\text{use}_{B_2} = \{i, j\}$ 。另外， B_2 显然对 i 和 j 定值。假设没有别名问题，因为 B_2 在定值之前使用了 i 和 j ，所以 $\text{def}_{B_2} = \{\}$ 。□

根据这些定义， use_B 中的任何变量都必然被认为在基本块 B 的入口处活跃，而 def_B 中的变量在 B 的开头一定是死的。实际上， def_B 中的成员“杀死”了某个变量可能因从 B 开始的某条路径而成为活跃变量的任何机会。

这样，把 def 和 use 与未知的 IN 和 OUT 值联系起来的方程定义如下：

$$\text{IN}[\text{EXIT}] = \emptyset$$

且对于所有的不等于 EXIT 的基本块 B 来说：

$$\text{IN}[B] = \text{use}_B \cup (\text{OUT}[B] - \text{def}_B)$$

$$\text{OUT}[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S]$$

第一个方程描述了边界条件，即在程序的出口处没有变量是活跃的。第二个方程说明一个变量要在进入一个基本块时活跃，必须满足下面两个条件中的一个：要么它在基本块中被重新定值之前就被使用；要么它在离开基本块时活跃且在基本块中没有对它重新定值。第三个方程说一个变量在离开一个基本块时活跃且当仅当它在进入该基本块的某个后继时活跃。

应该注意一下活跃性方程和到达定值方程之间的关系：

- 两组方程都以并集运算作为交汇运算。其原因是在各个数据流模式中，我们都沿着路径传播信息，并且我们只关心是否存在任何路径具有我们想要的性质，而不是关心某些结

论是否在所有的路径上都成立。

- 但是，活跃性的信息流逆向遍历，这和控制流的方向相反。其中的原因是在这个问题中，我们试图保证在一个程序点 p 上对变量 x 的使用可以被传递到在某个执行路径中 p 之前的所有程序点，这样我们才知道在前面的这些点上 x 的值会被使用。

为了解决一个逆向传播的数据流问题，我们对 $\text{IN}[\text{EXIT}]$ （而不是 $\text{OUT}[\text{ENTRY}]$ ）进行初始化。 IN 和 OUT 集合的角色相互对调了， use 和 def 分别替代了 gen 和 kill 。和到达定值问题一样，活跃性方程的解不必是唯一的，且我们希望得到具有最小活跃变量集合的解。解方程时使用的算法本质上是算法 9.11 的逆向传播版本。

算法 9.14 活跃变量分析。

输入：一个流图，其中每个基本块的 use 和 def 已经计算出来。

输出：该流图的各个基本块 B 的入口和出口处的活跃变量集合，即 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 。

方法：执行图 9-16 中的程序。 □

```

 $\text{IN}[\text{EXIT}] = \emptyset;$ 
for (除 EXIT 之外的每个基本块  $B$ )  $\text{IN}[B] = \emptyset;$ 
while (某个 IN 值发生了改变)
    for (除 EXIT 之外的每个基本块  $B$ ) {
         $\text{OUT}[B] = \bigcup_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S];$ 
         $\text{IN}[B] = \text{use}_B \cup (\text{OUT}[B] - \text{def}_B);$ 
    }
}

```

图 9-16 计算活跃变量的迭代算法

9.2.6 可用表达式

如果从流图入口结点到达程序点 p 的每条路径都对表达式 $x + y$ 求值，且从最后一个这样的求值之后到 p 点的路径上没有再次对 x 或 y 赋值[⊖]，那么 $x + y$ 在点 p 上可用 (available)。对于可用表达式数据流模式而言，如果一个基本块对 x 或 y 赋值 (或可能对它们赋值)，并且之后没有再重新计算 $x + y$ ，我们就说该基本块“杀死”了表达式 $x + y$ 。如果一个基本块一定对 $x + y$ 求值，并且之后没有再对 x 或 y 定值，那么这个基本块生成表达式 $x + y$ 。

请注意，“杀死”或“生成”一个可用表达式的概念和达到定值中的概念并不完全相同。尽管如此，这些“杀死”或“生成”的概念在行为上和到达定值中的相应概念在本质上是一致的。

可用表达式信息的主要用途是寻找全局公共子表达式。比如，在图 9-17a 中，如果 $4 * i$ 在基本块 B_3 的入口点可用，那么基本块 B_3 中的表达式 $4 * i$ 就是一个公共子表达式。它在该处可用的条件是 i 在基本块 B_2 中没有被赋予一个新值，或者像图 9-17b 所示的那样在 B_2 中对 i 赋值后又重新计算了 $4 * i$ 。

我们可以从头到尾地处理基本块内的各个语句，计算一个基本块内各个点上生成的表达式的集合。在基本块前面的点上没有任何生成的表达式。如果在点 p 处可用表达式的集合是 S ，而 q 是 p 之后的点，且它们之间是语句 $x = y + z$ ，那么通过下面的两个步骤可得到点 q 上的可用表达式集合。

1) 把表达式 $y + z$ 添加到 S 中。

2) 从 S 中删除任何涉及变量 x 的表达式。

请注意，因为 x 可能和 y 或 z 相同，所以上面的步骤必须按照正确的顺序执行。在我们到达基本块的结尾处时， S 就是该基本

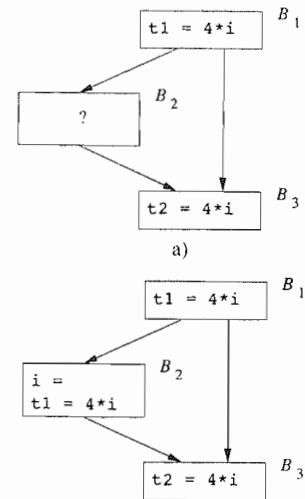


图 9-17 跨越多个基本块的潜在的公共子表达式

[⊖] 请注意，如在本章中通常使用的，我们使用运算符 $+$ 来代表一个一般的运算符，不是一定指加法运算。

块生成的表达式集合。而被杀死的表达式的集合就是所有类似于 $y + z$ 的表达式，其中 y 或 z 在基本块中被定值，并且这个基本块没有生成 $y + z$ 。

例 9.15 考虑图 9-18 中的四个语句。在第一个语句之后 $b + c$ 可用。在第二个语句之后 $a - d$ 变得可用，但是因为 b 被重新定值， $b + c$ 变得不再可用。第三个语句并没有使 $b + c$ 可用，因为 c 的值立刻就被改变了。在最后一个语句之后，因为 d 的值已经改变， $a - d$ 不再可用。因此这个基本块没有生成任何可用表达式，所有涉及 a 、 b 、 c 、 d 的表达式都被杀死了。□

我们可以用类似于计算到达定值的方法来寻找可用表达式。假设 U 是所有出现在程序中一个或多个语句的右部的表达式的全集。对于每个基本块 B ，令 $\text{IN}[B]$ 表示在 B 的开始处可用的 U 中的表达式的集合。令 $\text{OUT}[B]$ 表示在 B 的结尾处可用的表达式集合。定义 e_{gen}_B 为 B 生成的表达式的集合，而 e_{kill}_B 为被 B 杀死的 U 中的表达式的集合。请注意， IN 、 OUT 、 e_{gen} 和 e_{kill} 都可以使用位向量表示。下面的方程给出了未知的 IN 和 OUT 值之间，以及它们和已知量 e_{gen} 与 e_{kill} 之间的关系：

$$\text{OUT}[\text{ENTRY}] = \emptyset$$

并且对于除 ENTRY 之外的所有基本块 B ，有

$$\text{OUT}[B] = e_{\text{gen}}_B \cup (\text{IN}[B] - e_{\text{kill}}_B)$$

$$\text{IN}[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$$

上面的方程和到达定值方程组看起来几乎一样。和到达定值类似，这个方程组的边界条件也是 $\text{OUT}[\text{ENTRY}] = \emptyset$ ，这是因为在 ENTRY 的出口处没有任何可用表达式。其中最重要的不同之处在于这个方程组的交汇运算是交集运算，而不是并集运算。因为只有当一个表达式在一个基本块的所有前驱的结尾处都可用，它才会在该基本块的开头可用，因此使用交集运算是正确的。相反，只要一个定值到达了一个基本块的任何一个前驱的结尾处，它就到达了该基本块的开头，所以在到达定值方程组中使用并集运算作为交汇运算。

使用 \cap 而不是 \cup 使得可用表达式方程组的表现和到达定值方程组的表现不同。虽然两组方程都没有唯一解，但到达定值方程组的解是符合“到达”的定义的最小集合。在求解到达定值方程的过程中，我们首先假设任何地方都没有定值到达，然后逐渐增大到达定值的集合，最终构建得到该解。在这个方法里，除非找到一条能把某个定值 d 传播到某个点 p 的实际路径，否则我们从来不假设 d 能够到达 p 。相反，对于可用表达式方程组，我们希望得到具有最大可用表达式集合的解。因此，我们首先给出较大的近似值，然后逐步消减。

首先，我们假设“在除了人口基本块结尾处之外的所有地方，所有表达式（即集合 U ）都是可用的”。只有当我们发现有一条路径使得某个表达式不可用时，我们才删除这个表达式。这种方法看起来不是那么显而易见，但是我们可以得到一个真正的可用表达式的集合。在处理可用表达式时，生成一个可用表达式的精确集合的子集是保守的。之所以说使用子集是保守的，是因为我们将把这个信息用于把一个可用表达式的计算替换为之前计算得到的值。不知道一个表达式是可用的只会使我们失去改进代码的机会，而把一个不可用的表达式认为可用则会使我们改变程序的计算结果。

例 9.16 我们将把注意力集中在图 9-19 中的基本块 B_2 上，说明 $\text{OUT}[B_2]$ 的初始近似值对 $\text{IN}[B_2]$ 的影响。令 G 和 K 分别为 $e_{\text{gen}}_{B_2}$ 和 $e_{\text{kill}}_{B_2}$ 的缩写。 B_2 的数据流方程为

$$\text{IN}[B_2] = \text{OUT}[B_1] \cap \text{OUT}[B_2]$$

语句	可用表达式
$a = b + c$	\emptyset
$b = a - d$	$\{b + c\}$
$c = b + c$	$\{a - d\}$
$d = a - d$	$\{a - d\}$
	\emptyset

图 9-18 可用表达式的计算

$$\text{OUT}[B_2] = G \cup (\text{IN}[B_2] - K)$$

令 I^j 和 O^j 分别表示 $\text{IN}[B_2]$ 和 $\text{OUT}[B_2]$ 的第 j 次循环计算得到的近似值，这些方程式可以被写成下列的迭代计算式：

$$I^{j+1} = \text{OUT}[B_1] \cap O^j$$

$$O^{j+1} = G \cup (I^{j+1} - K)$$

从 $O^0 = \emptyset$ 开始，我们得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \emptyset$ 。但是，如果我们从 $O^0 = U$ 开始，那么我们得到 $I^1 = \text{OUT}[B_1] \cap O^0 = \text{OUT}[B_1]$ ，而这才是我们应该得到的值。直观地讲，以 $O^0 = U$ 作为初始值得到的解更符合我们的期望，因为这个解正确地反映了下面的事实：如果 $\text{OUT}[B_1]$ 中的某个表达式没有被 B_2 杀死，那么它在 B_2 的结尾处可用。

□

算法 9.17 可用表达式。

输入：一个流图，对其中的每个基本块 B , e_kill_B 和 e_gen_B 的值已经计算得到。流图的初始基本块是 B_1 。

输出：在流图的各个基本块 B 的入口处和出口处的可用表达式集合，即 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 。

方法：执行图 9-20 中的算法。图 9-20 中各个步骤的解释类似于图 9-14 的算法中的解释。

□

```

 $\text{OUT}[\text{ENTRY}] = \emptyset;$ 
 $\text{for}$  (除 ENTRY 之外的每个基本块  $B$ )  $\text{OUT}[B] = U$ ;
 $\text{while}$  (某个 OUT 值发生了改变)
     $\text{for}$  (除 ENTRY 之外的每个基本块  $B$ ) {
         $\text{IN}[B] = \bigcap_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P];$ 
         $\text{OUT}[B] = e\_gen_B \cup (\text{IN}[B] - e\_kill_B);$ 
    }
}

```

图 9-20 计算可用表达式的迭代算法

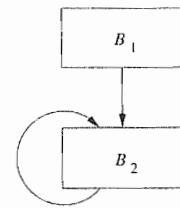
9.2.7 小结

在本节中，我们讨论了数据流问题的三个实例：到达定值、活跃变量和可用表达式。如图 9-21 中所总结的，每个问题的定义都是通过数据流值的域、数据流的方向、传递函数族、边界条件和交汇运算来定义的。我们一般用 \wedge 表示交汇运算。

图 9-21 的最后一列显示了迭代算法中使用的初始值。我们选择这些值的目的是使得迭代算法可以找到方程组的最精确解。严格地讲，这个选择并不是数据流问题的定义的一部分，因为它是为满足迭代算法的需要而人工给出的产品。还有其他途径可以解决数据流问题。比如，我们已经看到了如何把一个基本块中各个语句的传递函数组合起来得到该基本块的传递函数。我们可以用类似的组合方法来计算整个过程的传递函数，或者计算从过程的入口处到各个程序点的传递函数。我们将在 9.7 节中讨论这类方法的其中一种。

	到达定值	活跃变量	可用表达式
域	定值的集合	变量的集合	表达式的集合
方向	前向	后向	前向
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e_gen_B \cup (x - e_kill_B)$
边界条件	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
交汇运算(\wedge)	\cup	\cup	\cap
方程组	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P, \text{pred}(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S, \text{succ}(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P, \text{pred}(B)} \text{OUT}[P]$
初始值	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

图 9-21 三个数据流问题的总结

图 9-19 将 OUT 集合初始化为 \emptyset 局限性太大

9.2.8 9.2 节的练习

练习 9.2.1：对图 9-10 中的流图（见 9.1 节的练习），计算下列值：

- 1) 每个基本块的 gen 和 $kill$ 集合。
- 2) 每个基本块的 IN 和 OUT 集合。

练习 9.2.2：对图 9-10 的流图，计算可用表达式问题中的 e_gen 、 e_kill 、IN 和 OUT 集合。

练习 9.2.3：对图 9-10 的流图，计算活跃变量分析中的 def 、 use 、IN 和 OUT 集合。

! 练习 9.2.4[⊖]：假设 V 是复数的集合。下面的哪个运算可以被用作 V 上的一个半格结构的交汇运算？

- 1) 加法： $(a + ib) \wedge (c + id) = (a + c) + i(b + d)$
- 2) 乘法： $(a + ib) \wedge (c + id) = (ac - bd) + i(ad + bc)$
- 3) 按分量求最小： $(a + ib) \wedge (c + id) = \min(a, c) + i \min(b, d)$
- 4) 按分量求最大： $(a + ib) \wedge (c + id) = \max(a, c) + i \max(b, d)$

! 练习 9.2.5：我们曾经说过，如果一个基本块 B 由 n 个语句组成，并且第 i 个语句的 gen 集合和 $kill$ 集合分别是 gen_i 和 $kill_i$ ，那么基本块 B 的传递函数的 gen 集合 gen_B 和 $kill$ 集合 $kill_B$ 可以由下面的公式给出：

$$\begin{aligned} kill_B &= kill_1 \cup kill_2 \cup \dots \cup kill_n \\ gen_B &= gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ &\quad \dots \cup (gen_1 - kill_2 - kill_3 - \dots - kill_n) \end{aligned}$$

请通过对 n 的归纳来证明这个说法。

! 练习 9.2.6：请通过对算法 9.11 中第(4)到第(6)行的 for 循环的迭代次数的归纳，证明 IN 和 OUT 的值都不会缩小。也就是说，一但某个定值在某次循环的时候被放到其中的一个集中，它决不会在以后的某次循环中消失。

! 练习 9.2.7：证明算法 9.11 的正确性，也就是证明：

- 1) 如果定值 d 被放到 $IN[B]$ 或 $OUT[B]$ 中，那么相应地必然有一条从 d 到基本块 B 的开始处或结尾处的路径。在这条路径中，由 d 定值的变量不会被重新定值。
- 2) 如果定值 d 最后没有被放到 $IN[B]$ 或 $OUT[B]$ 中，那么相应地必然没有从 d 到基本块 B 的开始处或结尾处的路径。在这条路径中，由 d 定值的变量不会被重新定值。

! 练习 9.2.8：证明有关算法 9.14 的下列性质：

- 1) 各个 IN 和 OUT 的值不会缩小。
- 2) 如果变量 x 被放到 $IN[B]$ 或 $OUT[B]$ 中，那么相应地有一条从基本块 B 的开始处或结尾处出发的路径，在这条路径上 x 可能被使用。
- 3) 如果变量 x 没有被放到 $IN[B]$ 或 $OUT[B]$ 中，那么相应地没有从基本块 B 的开始处或结尾处出发的路径，使得 x 在这条路径上被使用。

为什么可用表达式算法是正确的

我们需要解释一下为什么下面的结论成立，即在一开始的时候把人口基本块之外的所有基本块的 OUT 值都设置为 U（即所有表达式的集合），最终仍可以得到这些数据流方程的保守解。也就是说，找到的可用表达式确实都是可用的。第一，因为在这个数据流模式中的

[⊖] 本练习在 9.3 节之后完成。

交汇运算是交集运算，任何发现 $x + y$ 在某个程序点上不可用的理由都会在流图中沿着所有可能的路径向前传播，直到 $x + y$ 被重新计算并再次变得可用为止。第二，只有两个理由可能会使 $x + y$ 变成不可用的。

1) 因为 x 或 y 在基本块 B 中被定值且其后没有计算 $x + y$ ，因此 $x + y$ 被杀死。在这种情况下，我们第一次应用传递函数 f_B 的时候， $x + y$ 就会从 $\text{OUT}[B]$ 中被删除。

2) 在某些路径中， $x + y$ 一直没有被计算。因为 $x + y$ 肯定不会在 $\text{OUT}[\text{ENTRY}]$ 中，并且它也不会在上面说的那条路径中被生成。我们通过对路径长度的归纳来证明 $x + y$ 最终会从这条路径的所有基本块的 IN 和 OUT 值中删除。

因此，当各个 IN 和 OUT 值不再改变的时候，图 9-20 中提到的迭代算法给出的解将只包含真正的可用表达式。

! 练习 9.2.9：证明有关算法 9.17 的下列特性：

- 1) 各个 IN 和 OUT 的值决不会增长。也就是说，这些集合在后来的取值总是它们前面取值的子集(不一定是真子集)。
- 2) 如果表达式 e 从 $\text{IN}[B]$ 或 $\text{OUT}[B]$ 中被删除，那么必然相应地存在一条从流图入口到达 B 的开始处或结尾处的路径，要么 e 在这条路径上从没有被计算过，要么在最后一次对 e 计算之后， e 的某个参数被重新定值了。
- 3) 如果表达式最终保留在 $\text{IN}[B]$ 或 $\text{OUT}[B]$ 中，那么相应地从流图入口到基本块 B 开始处或结尾处的所有路径中， e 都被计算，且在最后一次计算 e 之后， e 的参数都没有被重新定值。

! 练习 9.2.10：细心的读者可能注意到在算法 9.11 中，我们可以把各个基本块 B 的 gen_B 初始化为 $\text{OUT}[B]$ ，这样可以减少一些运行时间。类似地，我们还可以在算法 9.14 中把 use_B 初始化为 $\text{IN}[B]$ 。我们没有这么做的原因是用了统一的方法来处理这个主题。我们将在算法 9.25 中再次看到这一点。但是，可以在算法 9.17 中把 e_{gen}_B 初始化为 $\text{OUT}[B]$ 吗？为什么可以或不可以？

! 练习 9.2.11：至今为止，我们的数据流分析没有利用条件跳转的语义。假设我们在一个基本块的结尾处找到一个如下的测试：

```
if (x < 10) goto ...
```

我们如何利用对测试表达式 $x < 10$ 的理解来改进有关到达定值的知识？请记住，在这里“改进”意味着我们要消除某些实际上永远不可能达到某个程序点的到达定值。

9.3 数据流分析基础

我们已经给出了几个数据流抽象的有用的例子，现在我们以整体的方式抽象地研究数据流模式族。我们将正式回答下列有关数据流算法的基本问题：

- 1) 数据流分析中用到的迭代算法在什么情况下是正确的？
- 2) 通过迭代算法得到的解有多精确？
- 3) 迭代算法收敛吗？
- 4) 这些方程组的解的含义是什么？

在 9.2 节中我们描述到达定值问题的时候已经非正式地回答了上面的问题。对于后来的几个数据流问题，我们并没有从头回答同样的提问，我们依靠新问题和已讨论的问题之间的相似之处来解释新问题。本节中我们试图做到一劳永逸。针对一大类的数据流问题，我们给出一个一般性的方法来严格地回答这些问题。我们首先确定数据流模式的预期特性，并证明这些特性所

蕴含的信息，包括正确性、精确性、数据流算法的收敛性，以及方程组解的含义。这样，在理解老算法或者写新算法的时候，我们只需要给出相应的数据流问题定义所具有的特性，就可以立刻得到对上面各个问题的回答。

对一类模式给出一个统一的理论框架也有实践意义。这个框架有助于我们在软件设计中确定求解算法的可复用组件。因为不需要对类似的细节进行多次重复编码，所以不仅编码的工作量降低了，编程错误也会减少。

一个数据流分析框架(D, V, \wedge, F)由下列元素组成

- 1) 一个数据流方向 D ，它的取值包括 FORWARD(前向)或 BACKWARD(逆向)。
- 2) 一个半格(定义请见 9.3.1 节)，它包括值集 V 和一个交汇运算 \wedge 。
- 3) 一个从 V 到 V 的传递函数族 F 。这个传递函数族中必须包括可用于刻划边界条件的函数，即作用于任何数据流图中的特殊结点 ENTRY 和 EXIT 的常值传递函数。

9.3.1 半格

半格(semilattice)是满足下列条件的一个集合 V 和一个二元交汇运算 \wedge 。对于 V 中的所有 x 、 y 和 z ：

- 1) $x \wedge x = x$ (交汇运算是等幂的)。
- 2) $x \wedge y = y \wedge x$ (交汇运算是可交换的)。
- 3) $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (交汇运算是符合结合律的)。

半格有一个顶元素，表示为 \top ，使得对于 V 中的所有 x ， $\top \wedge x = x$ 。

半格可能还有一个底元素，表示为 \perp ，使得对于 V 中的所有 x ， $\perp \wedge x = \perp$ 。

偏序

正如我们将看到的，一个半格的交汇运算定义了值域上的一个偏序。假设 \leq 为 V 上的一个关系，如果对于 V 上的所有 x 、 y 和 z 都有：

- 1) $x \leq x$ (该偏序是自反的)。
- 2) 如果 $x \leq y$ 且 $y \leq x$ ，那么 $x = y$ (该偏序是反对称的)。
- 3) 如果 $x \leq y$ 且 $y \leq z$ ，那么 $x \leq z$ (该偏序是传递的)

那么 \leq 就是一个偏序(partial order)。

二元组(V, \leq)被称为偏序集(partially ordered set, poset)。对于一个偏序集，定义如下的关系 $<$ 会带来一些方便：

$$x < y \text{ 当且仅当 } (x \leq y) \text{ 且 } x \neq y$$

半格的偏序

为半格(V, \wedge)定义一个如下的偏序 \leq 会有所帮助。对于 V 中的所有 x 和 y ，我们定义

$$x \leq y \text{ 当且仅当 } x \wedge y = x$$

因为交汇运算 \wedge 是等幂的、可交换的且满足结合律，上面定义的序 \leq 就是自反的、反对称的和传递的。下面来说明其中的原因：

- 自反性：即对于所有的 x ， $x \leq x$ 。因为交汇运算是等幂的，因此 $x \wedge x = x$ 。
- 反对称性：即如果 $x \leq y$ 且 $y \leq x$ ，那么 $x = y$ 。在证明中， $x \leq y$ 意味着 $x \wedge y = x$ ，而 $y \leq x$ 意味着 $y \wedge x = y$ 。根据 \wedge 的可交换性， $x = (x \wedge y) = (y \wedge x) = y$ 。
- 传递性：即如果 $x \leq y$ 且 $y \leq z$ ，那么 $x \leq z$ 。证明如下： $x \leq y$ 且 $y \leq z$ 意味着 $x \wedge y = x$ 且 $y \wedge z = y$ 。那么使用交汇运算的结合律得到 $(x \wedge z) = ((x \wedge y) \wedge z) = (x \wedge (y \wedge z)) = (x \wedge y) = x$ 。因为已经证明了 $x \wedge z = x$ ，我们有 $x \leq z$ ，从而证明了传递性。

例 9.18 在 9.2 节的例子中使用的交汇运算是集合的并集或交集运算。它们都是等幂的，可交换的和可结合的。对于集合的并运算，顶元素是 \emptyset ，而底元素是全集 U 。这是因为对于 U 的任何子集 x 都有 $\emptyset \cup x = x$ 且 $U \cup x = U$ 。对于集合的交汇运算， \top 是 U 而 \perp 是 \emptyset 。半格的值域 V 就是全集 U 的所有子集的集合。这个集合有时被称为 U 的幂集 (power set)，并用 2^U 表示。

对于 V 中的所有 x 和 y ， $x \cup y = x$ 意味着 $x \supseteq y$ 。因此，并集运算确定的偏序为 \supseteq ，即集合的包含关系。相应地，集合的交集运算确定的偏序是 \subseteq ，即集合的被包含关系。也就是说，对于由交集运算所确定的偏序而言，元素较少的集合被认为是较小的值；但是对于由并集运算确定的偏序而言，元素较多的集合却被认为是较小的。一个较大的集合在偏序中反而较小是违反直觉的。但是根据前面的定义，这种情况是不可避免的[⊖]。

9.2 节中讨论过，一个数据流方程组通常有多个解，而(根据偏序关系 \leq 而言)最大的解是最精确的。比如，在到达定值问题中，所有的数据流方程的解中最精确的解是具有最少定值的解。这个解对应于由此问题的交汇运算(即并集运算)所定义的偏序中的最大元素。在可用表达式中，最精确的解是具有最多表达式的解。同样，它是相对于由交集运算(即此问题的交汇运算)定义的偏序的最大解。□

最大下界

在交汇运算和它确定的偏序之间还有一个有用的关系。假设 (V, \wedge) 是一个半格。域元素 x 和 y 的最大下界 (greatest lower bound, glb) 是一个满足下列条件的元素 g ：

- 1) $g \leq x$
- 2) $g \leq y$ ，且
- 3) 如果 z 是使得 $z \leq x$ 且 $z \leq y$ 成立的元素，那么 $z \leq g$ 。

我们的结论是， x 和 y 的交汇运算值就是它们的唯一最大下界。为了说明其中的原因，令 $g = x \wedge y$ 。可以观察到下列性质：

- 因为 $(x \wedge y) \wedge x = x \wedge y$ ，所以 $g \leq x$ 。这个结论的证明只涉及结合性、可交换性和等幂性质。也就是，

$$g \wedge x = ((x \wedge y) \wedge x) = (x \wedge (y \wedge x)) = (x \wedge (x \wedge y)) = ((x \wedge x) \wedge y) = (x \wedge y) = g$$
- 通过类似的论证可以得到 $g \leq y$ 。
- 假设 z 是任意的满足 $z \leq x$ 和 $z \leq y$ 的元素。已知 $z \leq g$ ，因此除非 z 就是 g ，否则它不是 x 和 y 的一个最大下界。证明如下： $(z \wedge g) = (z \wedge (x \wedge y)) = ((z \wedge x) \wedge y)$ 。因为 $z \leq x$ ，我们知道 $(z \wedge x) = z$ ，因此 $(z \wedge g) = (z \wedge y)$ 。因为 $z \leq y$ ，我们知道 $z \wedge y = z$ ，因此 $z \wedge g = z$ 。我们已经证明了 $z \leq g$ ，并且得出结论 $g = x \wedge y$ 是 x 和 y 的唯一最大下界。

并函数、最小上界和格

和一个偏序集合中的元素的最大下界操作对应，我们可以把元素 x 和 y 的最小上界 (least upper bound, lub) 定义为满足下列条件的元素 b : $x \leq b$ 且 $y \leq b$ ，并且对于任何满足 $x \leq z$ 和 $y \leq z$ 的元素 z 都有 $b \leq z$ 。可以证明，如果存在最小上界，那么最多只有一个最小上界。

在一个真的格中有两个域元素上的运算：我们已经看到的交汇运算 \wedge ，以及记为 \vee 的并函数。并 (join) 函数给出了两个元素的最小上界。因此格中的元素总是存在最小上界。至今

[⊖] 并且，如果我们把偏序定义为 \geq 而不是 \leq ，对于并集而言就不会产生这样的问题，但是对于交集而言还是会这样的问题。

为止我们一直讨论的是“半个”格，即只存在交汇运算和并函数之一。也就是说，我们的半格是一个交半格（meet semilattice）。人们也可以讨论并半格（join semilattice），即只有并函数的半格。实际上有些程序分析的文献就使用并半格的概念。因为传统的数据流文献讲的是交半格，所以在本书中我们也使用交半格。

格图

把域 V 画成一个格图对我们会有所帮助。格图的结点是 V 的元素，而它的边是向下的，即如果 $y \leqslant x$ ，那么从 x 到 y 有一个边。比如，图 9-22 给出了一个到达定值数据流模式的集合 V 。其中有三个定值： d_1 、 d_2 和 d_3 。因为半格中的偏序关系 \leqslant 是 \supseteq ，从这三个定值的集合的子集到其所有超集有一个向下的边。因为 \leqslant 是传递的，如果图中有一条从 x 到 y 的路径，我们可以按照惯例省略从 x 到 y 的边。因此，虽然在这个例子中 $\{d_1, d_2, d_3\} \leqslant \{d_1\}$ ，我们并没有画出这条边，因为这个边可以用经过 $\{d_1, d_2\}$ 的路径来表示。

有一点也很有用，即我们可以从这样的图中读出交汇值。因为 $x \wedge y$ 就是它们的最大下界，因此这个值总是最高的、从 x 和 y 都有向下的路径到达的元素 z 。比如，如果 x 是 $\{d_1\}$ 而 y 是 $\{d_2\}$ ，那么图 9-22 中的 z 就是 $\{d_1, d_2\}$ 。这是正确的，因为这里的交汇运算是并集运算。顶元素将出现在格图的顶部，也就是说，从 \top 到图中的每个元素都有一条向下的路径。类似地，底元素将出现在图的底部，从每个元素都有一条边到达 \perp 。

乘积格

图 9-22 中只涉及了三个定值，而一个典型程序的格图可能相当大。数据流值的集合是定值的幂集。因此如果一个程序中有 n 个定值，则该程序的数据流值集合包含 2^n 个元素。但是，一个定值是否到达某个程序点和其他定值的可达性无关。我们因此可以用“乘积格”的方式来表示定值的格^②。这个乘积格由各个定值对应的简单格构造得到。也就是说，如果程序中只有一个定值 d ，那么相应的格将只包括两个元素：空集 $\{\}$ （它是顶元素）以及 $\{d\}$ （它是底元素）。

严格地讲，我们按照下面的方式构造乘积格。假设 $\{A, \wedge_A\}$ 和 $\{B, \wedge_B\}$ 是两个（半）格。这两个格的乘积格定义如下：

1) 乘积格的域是 $A \times B$ 。

2) 乘积格的交汇运算 \wedge 定义如下：如果 (a, b) 和 (a', b') 是乘积格域中的元素，那么

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b') \quad (9.19)$$

乘积格的偏序可以很简单地用 A 的偏序 \leqslant_A 和 B 的偏序 \leqslant_B 来表示：

$$(a, b) \leqslant (a', b') \text{ 当且仅当 } a \leqslant_A a' \text{ 且 } b \leqslant_B b' \quad (9.20)$$

为了看出为什么从式(9.19)可以推出式(9.20)，请注意下面的性质：

$$(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$$

我们可能会问在什么情况下 $(a \wedge_A a', b \wedge_B b') = (a, b)$ ？当且仅当 $a \wedge_A a' = a$ 且 $b \wedge_B b' = b$ 的时候这个等式成立。而这两个条件和 $a \leqslant_A a'$ 和 $b \leqslant_B b'$ 是一回事。

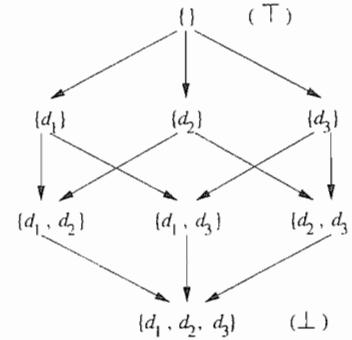


图 9-22 定值的子集的格

^② 在这里及以后的讨论中，我们常常会把“半格”中的“半”字去掉，因为像我们现在讨论的那些格都有一个并（或者说 lub）运算符，虽然我们不会使用这个运算符。

格的乘积是一个满足结合律的运算，因此我们可以证明规则(9.19)和(9.20)可以被扩展到任意多个格。也就是说，如果我们有格 (A_i, \wedge_i) ($i=1, 2, \dots, k$)，那么这 k 个格按照这个顺序的乘积的域为 $A_1 \times A_2 \times \dots \times A_k$ ，其交汇运算定义为：

$$(a_1, a_2, \dots, a_k) \wedge (b_1, b_2, \dots, b_k) = (a_1 \wedge_1 b_1, a_2 \wedge_2 b_2, \dots, a_k \wedge_k b_k)$$

而偏序定义为

$$(a_1, a_2, \dots, a_k) \leq (b_1, b_2, \dots, b_k) \text{ 当且仅当对于所有的 } i, a_i \leq b_i。$$

半格的高度

通过研究一个数据流问题中的半格的“高度”，我们可以知道一些关于数据流分析算法收敛速度的信息。偏序集 (V, \leq) 的一个上升链(ascending chain)是一个满足 $x_1 < x_2 < \dots < x_n$ 的序列。一个半格的高度(height)是所有上升链中的 $<$ 关系个数的最大值。也就是说，高度比链中的元素个数少一。比如，一个有 n 个定值的程序的到达定值半格的高度是 n 。

如果一个半格具有有穷的高度，就可以比较容易地证明相应的迭代数据流算法的收敛性。显然，一个由有穷值集组成的格具有有穷的高度；一个具有无穷多个值的格也可能具有有穷的高度。在常量传播算法中使用的格就是一个这样的例子，我们将在 9.4 节中详细地说明这个例子。

9.3.2 传递函数

一个数据流框架中的传递函数族 $F: V \rightarrow V$ 具有下列性质：

- 1) F 有一个单元函数 I ，使得对于 V 中的所有 x , $I(x) = x$ 。
- 2) F 对函数组合运算封闭。也就是说，对于 F 中的任意函数 f 和 g ，定义为 $h(x) = g(f(x))$ 的函数 h 也在 F 中。

例 9.21 在到达定值中， F 有单元函数，即 gen 和 $kill$ 都是空集的传递函数。对函数组合的封闭性实际上已经在 9.2.4 节中得到证明，我们在这里简单地重复一下证明过程。假设我们具有两个函数

$$f_1(x) = G_1 \cup (x - K_1) \text{ 和 } f_2(x) = G_2 \cup (x - K_2)$$

那么

$$f_2(f_1(x)) = G_2 \cup ((G_1 \cup (x - K_1)) - K_2)$$

根据代数规则，上式的右部和下式等价：

$$(G_2 \cup (G_1 - K_2)) \cup (x - (K_1 \cup K_2))$$

如果我们令 $K = K_1 \cup K_2$, $G = G_2 \cup (G_1 - K_2)$ ，我们就证明了 f_1 和 f_2 的组合 $f(x) = G \cup (x - K)$ 的形式表明它是 F 的成员。如果我们考虑可用表达式的问题，上面用于到达定值的证明也同样可以证明 F 具有单元函数并且对函数组合运算封闭。□

单调的框架

要使得数据流分析问题的迭代算法能够完成任务，我们还要求数据流框架再满足一个条件。对于一个框架，如果框架中的所有传递函数都是单调的，那么我们就说这个框架是单调的。 F 中的传递函数 f 是单调函数的条件是对于域 V 中的任意两个元素，如果第一个元素大于第二个元素，那么 f 作用于第一个元素的结果也大于它作用于第二元素所得到的结果。

正式的定义如下，一个数据流框架 (D, F, V, \wedge) 是单调的(monotone)，如果

$$\text{对于所有的 } V \text{ 中的 } x \text{ 和 } y \text{ 以及 } F \text{ 中的 } f, x \leq y \text{ 蕴含 } f(x) \leq f(y) \quad (9.22)$$

单调性可以被等价地定义为

$$\text{对于所有的 } V \text{ 中的 } x \text{ 和 } y \text{ 以及 } F \text{ 中的 } f, f(x \wedge y) \leq f(x) \wedge f(y) \quad (9.23)$$

式(9.23)说明，如果我们对两个值应用交汇运算再应用函数 f ，那么得到的结果绝对不会大

于首先将 f 分别应用于两个值，然后再对结果应用交汇运算而得到的值。这两个关于单调的定义看起来很不相同，它们各有各的用处。我们会发现这两个定义分别适用于不同的环境。稍后我们将给出一个简略的证明，表明它们确实是等价的。

我们将首先假设式(9.22)成立并证明式(9.23)成立。因为 $x \wedge y$ 是 x 和 y 的最大下界，我们知道

$$x \wedge y \leq x \text{ 且 } x \wedge y \leq y$$

因此由式(9.22)可知：

$$f(x \wedge y) \leq f(x) \text{ 且 } f(x \wedge y) \leq f(y)$$

因为 $f(x) \wedge f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界，我们证明了(9.23)。

反过来，我们假设式(9.23)成立并证明式(9.22)。我们假设 $x \leq y$ 并使用式(9.23)来得到 $f(x) \leq f(y)$ 的结论，从而证明式(9.22)。式(9.23)告诉我们

$$f(x \wedge y) \leq f(x) \wedge f(y)$$

但是因为我们已经假设了 $x \leq y$ ，根据定义有 $x \wedge y = x$ 。因此，式(9.23)表明

$$f(x) \leq f(x) \wedge f(y)$$

因为 $f(x) \wedge f(y)$ 是 $f(x)$ 和 $f(y)$ 的最大下界，我们得到 $f(x) \wedge f(y) \leq f(y)$ 。这样

$$f(x) \leq f(x) \wedge f(y) \leq f(y)$$

因此式(9.23)蕴含式(9.22)。

可分配的框架

数据流分析框架经常会遵守一个比式(9.23)更强的条件，我们把这个条件称为可分配条件 (distributivity condition)，即对于 V 中的所有 x 和 y 以及 F 中的所有 f ，有

$$f(x \wedge y) = f(x) \wedge f(y)$$

当然，如果 $a = b$ ，那么根据等幂性有 $a \wedge b = a$ ，因此 $a \leq b$ 。这样，可分配性蕴含了单调性，但是反过来并不成立。

例 9.24 令 y 和 z 为到达定值框架下的定值集合。令 f 是一个定义为 $f(x) = G \cup (x - K)$ 的函数，其中 G 和 K 为某个定值的集合。通过检验下面的等式

$$G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$$

我们就可以证明到达定值的框架满足可分配性条件。

虽然上面的等式看起来很难，但我们可以首先考虑在 G 中的那些定值。这些定值一定都在上面等式的左部和右部所定义的两个集合中。因此我们只需要考虑不在 G 中的定值的集合。在这种情况下，我们可以把 G 从所有的地方删除，并验证等式

$$(y \cup z) - K = (y - K) \cup (z - K)$$

通过 Venn 图就可以很容易地验证这个等式。 □

9.3.3 通用框架的迭代算法

我们可以对算法 9.11 进行推广，使之能够处理各种数据流问题。

算法 9.25 通用数据流框架的迭代解法。

输入：一个由下列部分组成的数据流框架：

- 1) 一个数据流图，它有两个被特别标记为 ENTRY 和 EXIT 的结点。
- 2) 数据流的方向 D 。
- 3) 一个值集 V 。
- 4) 一个交汇运算 \wedge 。
- 5) 一个函数的集合 F ，其中 f_B 表示基本块 B 的传递函数。

6) V 中的一个常量值 v_{ENTRY} 或者 v_{EXIT} 。它们分别表示前向和逆向框架的边界条件。

输出：上述数据流图中各个基本块 B 的 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 的值。这些值在 V 中。

方法：解决前向和逆向数据流问题的算法分别显示在图 9-23a 和图 9-23b 中。和 9.2 节中的各个数据流迭代算法类似，我们通过不断近似逼近的方式来计算各个基本块的 IN 值和 OUT 值。□

```

1)  $\text{OUT}[\text{ENTRY}] = v_{\text{ENTRY}}$ ;
2) for (除 ENTRY 之外的每个基本块  $B$ )  $\text{OUT}[B] = \top$ ;
3) while (某个 OUT 值发生了改变)
4)   for (除 ENTRY 之外的每个基本块  $B$ ) {
5)      $\text{IN}[B] = \bigwedge_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$ ;
6)      $\text{OUT}[B] = f_B(\text{IN}[B])$ ;
}

```

a) 前向数据流问题的迭代算法

```

1)  $\text{IN}[\text{EXIT}] = v_{\text{EXIT}}$ ;
2) for (除 EXIT 之外的每个基本块  $B$ )  $\text{IN}[B] = \top$ ;
3) while (某个 IN 值发生了改变)
4)   for (除 EXIT 之外的每个基本块  $B$ ) {
5)      $\text{OUT}[B] = \bigwedge_{S \text{ 是 } B \text{ 的一个后继}} \text{IN}[S]$ ;
6)      $\text{IN}[B] = f_B(\text{OUT}[B])$ ;
}

```

b) 逆向数据流问题的迭代算法

图 9-23 数据流问题迭代算法的前向和逆向的版本

也可以改写算法 9.25，使得它把实现交汇运算的函数作为一个参数，同时也把实现各基本块的传递函数的函数作为参数。流图本身和边界值也都作为参数。使用这种方法，编译器的实现者就可以避免为编译器优化阶段所使用的每个数据流框架都从头编写基本迭代算法的代码。

我们可以使用至今为止讨论的抽象框架来证明该迭代算法的一组有用的性质：

1) 如果算法 9.25 收敛，其结果就是数据流方程组的一个解。

2) 如果框架是单调的，那么找到的解就是数据流方程组的最大不动点 (Maximum FixedPoint, MFP)。一个最大不动点是一个具有下面性质的解：在任何其他解中， $\text{IN}[B]$ 和 $\text{OUT}[B]$ 的值和 MFP 中对应的值之间具有 \leq 关系。

3) 如果框架的半格是单调的，且高度有穷，那么这个迭代算法必定收敛。

论证这些论点时，我们首先假设框架是前向的。对于逆向框架的论证实质上是一样的。第一个性质很容易证明。如果在 while 循环结束的时候方程组没有被满足，那么各个 OUT 值（对前向框架）或 IN 值（对逆向框架）中至少有一个值改变了，我们必须再次运行该循环。

为了证明第二个性质，我们首先证明，在运行算法迭代时任意的基本块 B 的 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 所取的值只能（相对于格中的 \leq 关系而言）下降。这个性质可以通过归纳方法证明。

归纳基础：归纳的基础步骤是证明 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 的值在第一个迭代之后不大于初始值。这个论断的正确性是显而易见的，因为所有不等于 ENTRY 的基本块 B 的 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 都被初始化为 \top 。

归纳步骤：假设经过 k 次迭代之后，那些值都不大于第 $(k-1)$ 次迭代后的值，我们要证明第 $k+1$ 次迭代和第 k 次迭代相比同样如此。图 9-23a 的第 5 行是：

$$\text{IN}[B] = \bigwedge_{P \text{ 是 } B \text{ 的一个前驱}} \text{OUT}[P]$$

我们用 $\text{IN}[B]^i$ 和 $\text{OUT}[B]^i$ 标记 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 在第 i 次迭代之后的值。假设 $\text{OUT}[P]^k \leq \text{OUT}[P]^{k-1}$ ，由交汇运算的性质可知 $\text{IN}[B]^{k+1} \leq \text{IN}[B]^k$ 。接下来，第(6)行说

$$\text{OUT}[B] = f_B(\text{IN}[B])$$

因为 $\text{IN}[B]^{k+1} \leq \text{IN}[B]^k$ ，由单调性可知 $\text{OUT}[B]^{k+1} \leq \text{OUT}[B]^k$ 。

请注意，每一个 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 值的改变都必须满足上述等式。交汇运算返回的是其输入的最大下界，且传递函数返回的值是和基本块本身及它的给定输入一致的唯一解。因此，如果该迭代算法终止，其结果值至少和任何其他解的相应值一样大。也就是说，算法 9.25 的结果是数

据流方程式的最大不动点。

最后考虑第三点，即数据流框架具有有穷高度的情况。因为每个 $\text{IN}[B]$ 和 $\text{OUT}[B]$ 的值在每次被改变时都会减小，而程序在某一轮循环中没有值改变时就会停止，因此算法的迭代次数不会大于框架高度和流图结点个数的乘积，因此算法必然终止。

9.3.4 数据流解的含义

现在我们知道使用前面的迭代算法得到的解是最大不动点，但从程序语义的角度来看，这个结果又代表了什么呢？为了理解一个数据流框架 (D, F, V, \wedge) 的解，我们首先描述一下一个框架的理想解应该是什么样子。我们将给出下面的性质，即一般情况下不能得到理想解，但是算法 9.25 保守地给出了理想解的近似值。

理想解

不失一般性，我们假设现在感兴趣的数据流框架是一个前向的数据流问题。考虑一个基本块 B 的入口点。求理想解的第一步是要找到从程序入口到达 B 的开头的所有可能的执行路径。只有当程序的某次执行能够准确地沿着某条路径进行，这条路径才被称为“可能的”。然后，理想的求解方法将计算每个可能路径尾端的数据流值，并对这些数据流值应用交汇运算得到它们的最大下界。那么，程序的任何执行都不可能在该程序点上产生一个更小的数据流值。另外，这个界限还是紧致的：根据流图中到达 B 的所有可能路径计算得到的数据流值的集合的最大下界不可能变得更大。

我们现在更为正式地定义理想解。对于一个流图中的每个基本块 B ，令 f_B 是 B 的传递函数。考虑任意从初始结点 ENTRY 到某个基本块 B_k 中的路径

$$P = \text{ENTRY} \rightarrow B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_{k-1} \rightarrow B_k$$

程序的路径可能包含环，因此一个基本块可能在路径 P 中多次出现。定义 P 的传递函数 f_P 为 $f_{B_1}, f_{B_2}, \dots, f_{B_{k-1}}$ 的函数组合的结果。请注意， f_{B_k} 没有参与组合运算，这表明这条路径只到达 B_k 的开头，而不是其结尾。执行这条路径而创建的数据流值就是 $f_P(v_{\text{ENTRY}})$ ，其中 v_{ENTRY} 是代表初始结点 ENTRY 的常值传递函数的结果。因此，基本块 B 的理想结果是

$$\text{IDEAL}[B] = \bigwedge_{P \text{是从 } \text{ENTRY} \text{ 到 } B \text{ 的一个可能路径}} f_P(v_{\text{ENTRY}})$$

按照问题中数据流框架的格理论偏序关系 \leq ，我们有下面的结论：

- 任何比 IDEAL 更大的答案都是错误的。
- 任何小于或者等于这个理想值的值都是保守的，即安全的。

直观地讲，越接近理想值的值就越精确[⊖]。下面说明为什么方程的解和理想值之间必须具有 \leq 关系。请注意，对于任何基本块，只要忽略程序可能执行的某些路径就可能得到该基本块的大于 IDEAL 的解。但是，如果我们基于这样的较大解来改进代码，就不能保证这些被忽略的路径中一定不会有某些执行效果使得我们的代码改进不正确。反过来，任何小于 IDEAL 的值都可以被看作是包含了某些不必要的路径，它们可能是流图中不存在的路径，也可能流图中存在此路径但程序却不会按这条路径执行。这些较小的解将只允许进行对程序的所有可能执行都正确的转换，但是它们会禁止 IDEAL 值原本允许的某些转换。

基于路径交汇运算的解

但是正如 9.1 节中所讨论的，寻找所有可能的执行路径是一个不可判定问题。因此，我们必须

[⊖] 请注意，在一个前向的问题中，我们希望 $\text{IN}[B]$ 的值等于 $\text{IDEAL}[B]$ 的值。我们没有在这里讨论逆向的问题。在逆向的问题中，我们把 $\text{IDEAL}[B]$ 定义为 $\text{OUT}[B]$ 的理想值。

须使用近似方法。在数据流抽象中，假设流图中的每条路径都可能被执行。因此，我们可以用如下方法定义 B 的基于路径交汇运算的解。

$$\text{MOP}[B] = \bigwedge_{P \text{ 是从 ENTRY 到 } B \text{ 的一个流图路径}} f_P(v_{\text{ENTRY}})$$

请注意，和前面讨论 IDEAL 时一样，MOP[B] 解给出的是前向数据流框架中 IN[B] 的值。如果我们要考虑反向数据流框架，那么我们会把 MOP[B] 当作 OUT[B] 的值。

在 MOP 解中考虑的路径是所有可能被执行路径的超集。因此，MOP 解中交汇运算的输入不仅包括所有可执行路径的数据流值，还包括了一些和不可能执行路径相关的数据流值。把理想解和一些其他的值进行交汇运算不可能创造出一个大于理想值的解。因此，对所有的 B ，我们有 $\text{MOP}[B] \leq \text{IDEAL}[B]$ 。我们简单地说 $\text{MOP} \leq \text{IDEAL}$ 。

最大不动点和 MOP 解

请注意，如果流图包含环，那么在 MOP 解中需要考虑的路径数量仍然是无界的。因此，不能直接由 MOP 的定义得到算法。当然，迭代算法也不是先找到所有到达一个基本块的路径，然后再应用交汇运算的，而是采用如下方法：

- 1) 这个迭代算法访问各个基本块，其访问的顺序并不一定是执行的顺序。
- 2) 在每个路径交汇点，算法对当前已经得到的数据流值应用交汇运算。其中一部分被使用的值可能是在初始化过程中人为加入的，并不表示从程序开始的执行结果。

那么，MOP 解和算法 9.25 产生的 MFP 解之间有何关系呢？

我们首先讨论一下访问结点的顺序。在一次迭代中，我们可能在访问一个结点的前驱之前就访问这个结点。如果其前驱为 ENTRY 结点，OUT[ENTRY] 已被初始化为正确的常量值。其他结点的 OUT 值被初始化为顶元素 \top ，这个值不小于最后的结果。由单调性可知，使用 \top 作为输入得到的结果不小于期望解。从某种意义上说，我们把 \top 当作表示不包含任何信息的值。

提前应用交汇运算的效果是什么呢？考虑图 9-24 中的简单例子，并假设我们对 IN[B_4] 的值感兴趣。根据 MOP 的定义：

$$\text{MOP}[B_4] = ((f_{B_3} \circ f_{B_1}) \wedge (f_{B_3} \circ f_{B_2})) (v_{\text{ENTRY}})$$

在迭代算法中，如果我们按照 B_1, B_2, B_3, B_4 的顺序访问结点，那么

$$\text{IN}[B_4] = f_{B_3}((f_{B_1}(v_{\text{ENTRY}}) \wedge f_{B_2}(v_{\text{ENTRY}})))$$

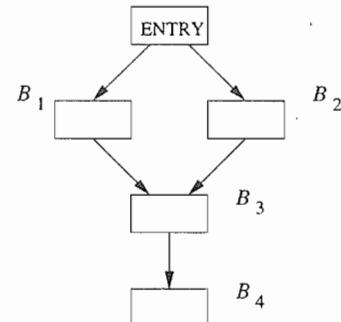


图 9-24 说明提前应用路径交汇运算的效果的流图

在 MOP 的定义中最后才应用交汇运算，而迭代法则提早使用这个函数。只有当数据流框架为可分配时得到的解才是相同的。如果一个数据流框架单调但不可分配，我们仍然有 $\text{IN}[B_4] \leq \text{MOP}[B_4]$ 。回忆一下，总的来说，如果对所有的基本块 B 都有 $\text{IN}[B] \leq \text{IDEAL}[B]$ ，那么这个解就是安全的（保守的）。这个解当然是安全的，因为 $\text{MOP}[B] \leq \text{IDEAL}[B]$ 。

我们现在简略说明一下为什么迭代算法提供的 MFP 解总是安全的。对 i 进行简单的归纳就可以表明在第 i 次迭代之后得到的值小于或等于对所有长度小于或等于 i 的路径进行交汇运算而得到的值。但是当迭代算法终止的时候，它得到的值和再进行任意多次迭代所得到的值相同。因此其结果不会大于 MOP 解。因为 $\text{MOP} \leq \text{IDEAL}$ 且 $\text{MFP} \leq \text{MOP}$ ，我们知道 $\text{MFP} \leq \text{IDEAL}$ ，因此由迭代算法提供的 MFP 解是安全的。

9.3.5 9.3节的练习

练习 9.3.1：构造一个三个格的乘积的格图。其中的每个格都是基于单一定值 $d_i (i=1, 2, 3)$ 。得到的格图和图 9-22 中的格图有什么关系？

! 练习 9.3.2：在 9.3.3 节中，我们说如果框架具有有限的高度，那么迭代算法收敛。这里给出一个框架没有有限高度且迭代算法也不收敛的例子。令值集 V 是非负实数，令交汇运算为取最小值运算。有三个传递函数：

- 1) 单元函数 $f_I(x) = x$ 。
- 2) “半”函数，即函数 $f_H(x) = x/2$ 。
- 3) “一”函数，即函数 $f_O(x) = 1$ 。

传递函数的集合 F 是这三个函数以及它们按照各种可能方式组合得到的函数。

- 1) 描述函数集 F 。
- 2) 这个框架的 \leq 关系是什么？
- 3) 给出一个流图并在流图的各个结点上赋予传递函数，使得算法 9.25 对这个流图不收敛。
- 4) 这个框架是单调的吗？它是可分配的吗？

! 练习 9.3.3：我们说如果框架单调且具有有限高度，那么算法 9.25 收敛。这里给出一个框架的例子。它说明单调性是很重要，有穷高度不足以保证算法收敛。这个框架的域 V 是 $\{1, 2\}$ ，交汇运算是 \min ，而函数集 F 只有单元函数(f_I)和“替换”函数($f_S(x) = 3 - x$)，它的功能是使得值在 1 和 2 之间互换。

- 1) 说明这个框架具有有限高度，但是不单调。
- 2) 给出一个流图的例子，并给每个结点赋予一个传递函数，使得算法 9.25 对这个流图不收敛。

! 练习 9.3.4：令 $MOP_i[B]$ 为所有从程序入口结点到达基本块 B 的长度不大于 i 的路径的交汇运算结果值。证明在算法 9.25 迭代 i 次之后， $IN[B] \leq MOP_i[B]$ 。同时证明，作为上面结论的推论，如果算法 9.25 收敛，它必然收敛于某个和 MOP 解具有 \leq 关系的值。

! 练习 9.3.5：假设一个框架的传递函数集合 F 具有 gen-kill 形式。也就是说，域 V 是某个集合的幂集，而 $f(x) = G \cup (x - K)$ ，其中 G 和 K 是两个集合。证明如果交汇运算是并集运算或交集运算，框架都是可分配的。

9.4 常量传播

在 9.2 节中讨论的所有数据流模式实际上都是具有有限高度的可分配框架的简单例子。这样，迭代算法 9.25 的前向或逆向版本可以用来解决这些问题，并求出每个问题的 MOP 解。在本节中，我们将深入研究一个具有更多有趣性质的有用的数据流框架。

回忆一下常量传播（或者说“常量折叠”），即把那些在每次运行时总是得到相同常量值的表达式替换为该常量值。下面描述的常量传播框架和至今已经讨论的数据流问题都有所不同。不同之处在于：

- 1) 它的可能数据流值的集合是无界的。即使对于一个确定的流图也是如此。
- 2) 它不是可分配的。

常量传播是一个前向数据流问题。表示此问题数据流值的半格和问题的传递函数族在下面给出。

9.4.1 常量传播框架的数据流值

这个问题的数据流值的集合是一个乘积格，其中每个分量对应于程序中的一个变量。单个

变量的格由下列元素组成：

- 1) 所有符合该变量类型的常量值。
- 2) 值 NAC, 即 not-a-constant, 表示非常量值。当确定一个变量的值不是常量值时, 该变量就被映射到值 NAC。这个变量被映射到 NAC 值的原因可能是它被赋予了一个输入变量的值, 或者它从一个不具常量值的变量中获得值, 也可能是在到达同一程序点的不同路径上被赋予不同的常量值。
- 3) 值 UNDEF, 代表未定义。如果还不能确定任何有关这个变量的信息, 就把它映射到这个值上。原因很可能是还没有发现有哪个对这个变量的定值能够到达问题中的程序点。

请注意, NAC 和 UNDEF 是不同的, 实质上它们是对立的。NAC 说我们已经知道一个变量有多种定值方式, 因此我们知道它不是常量; UNDEF 是说有关这个变量我们知道得非常少, 以至于我们根本不能确定任何事情。

一个典型的整数类型变量的半格如图 9-25 所示。这里, 顶元素是 UNDEF, 底元素是 NAC。也就是说, 半格的偏序的最大值是 UNDEF, 最小值是 NAC。其他的常量值是无序的, 但是它们都比 UNDEF 小而比 NAC 大。如 9.3.1 节所讨论的, 两个值的交是它们的最大下界。因此, 对于所有的值 v , 有

$$\text{UNDEF} \wedge v = v \text{ 且 } \text{NAC} \wedge v = \text{NAC}$$

对于任意的常量 c , 有

$$c \wedge c = c$$

且给定两个不同的常量 c_1 和 c_2 , 有

$$c_1 \wedge c_2 = \text{NAC}$$

这个框架中的一个数据流值是从程序中的各个变量到上面的常量半格中的某个值的映射。变量 v 在一个映射 m 中的值记为 $m(v)$ 。

9.4.2 常量传播框架的交汇运算

这个数据流问题的数据流值的半格就是图 9-25 中所示半格的乘积, 对于每个变量有一个图 9-25 中所示的半格。因此, $m \leq m'$ 当且仅当对于所有的变量 v 都有 $m(v) \leq m'(v)$ 。换句话说, $m \wedge m' = m''$ 当且仅当对于所有的变量 v , $m(v) \wedge m'(v) = m''(v)$ 。

9.4.3 常量传播框架的传递函数

下面我们假设一个基本块只包含一个语句。包含多个语句的基本块的传递函数可以通过将各个语句对应的传递函数组合起来而构造得到。函数集合 F 由一组传递函数组成, 这些传递函数接受的输入是一个从程序变量到常量格中元素的映射, 而其返回值则是另一个这样的映射。

F 包含一个单元函数, 它接受一个映射作为输入并返回相同的映射。 F 也包含了对应于 ENTRY 结点的常值传递函数。这个传递函数对于任意的输入映射都返回映射 m_0 , 而对于所有的变量 v , $m_0(v) = \text{UNDEF}$ 。因为在执行任何程序语句之前任何变量都没有定义, 因此这个边界条件是合理的。

一般来说, 令 f_s 为语句 s 的传递函数, 并令 m 和 m' 表示满足 $m' = f_s(m)$ 的两个数据流值。我们将用 m 和 m' 之间的关系来描述 f_s 。

- 1) 如果 s 不是一个赋值语句, 那么 f_s 就是单元函数。
- 2) 如果 s 是一个对变量 x 的赋值, 那么对于所有变量 $v \neq x$, $m'(v) = m(v)$; 其中 $m'(x)$ 的定

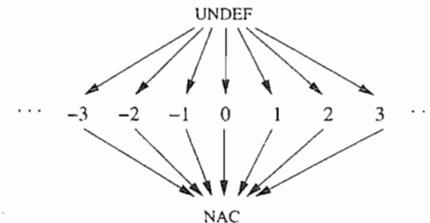


图 9-25 表示了一个整数类型变量的所有可能“取值”的半格

义如下：

(a) 如果语句 s 的右部(RHS)是一个常量 c , 那么 $m'(x) = c$ 。

(b) 如果 RHS 形如 $y + z^\ominus$, 那么

$$m'(x) = \begin{cases} m(y) + m(z) & \text{如果 } m(y) \text{ 和 } m(z) \text{ 都是常量值} \\ \text{NAC} & \text{如果 } m(y) \text{ 或者 } m(z) \text{ 是 NAC} \\ \text{UNDEF} & \text{否则} \end{cases}$$

(c) 如果 RHS 是其他表达式(比如一个函数调用, 或者使用指针的赋值), 那么 $m'(x) = \text{NAC}$ 。

9.4.4 常量传递框架的单调性

现在我们来证明常量传递框架是单调的。首先, 我们可以考虑一个函数 f_s 对于单个变量的影响。除了情况 2(b)之外, f_s 要么没有改变 $m(x)$ 的值, 要么把 x 的映射值改成一个常量或者 NAC。在这些情况下, f_s 无疑是单调的。

对于情况 2(b), f_s 的影响如图 9-26 所示。第一列和第二列代表 y 及 z 的可能输入值, 最后一列表示 x 的输出值。每列(或者每个子列)中的值按照从大到小的方式排列。为了说明函数的单调的, 我们将检验下面的性质, 即对于 y 的每个可能的输入值, x 的值不会在 z 值变小的时候变大。比如, 在 y 具有常量值 c_1 的情况下, 当 z 的值从 UNDEF 变为 c_2 、再变为 NAC 时, x 的取值相应地从 UNDEF 变为 $c_1 + c_2$ 、再到 NAC。我们可以对 y 的所有可能取值重复这个检验过程。因为对称性, 我们甚至不需要对第二个运算分量重复这个过程就可以得出结论: 当输入变小的时候输出不会变大。

9.4.5 常量传播框架的不可分配性

上面定义的常量传播框架是单调的, 但不是可分配的。也就是说, 迭代解 MFP 是安全的, 但是可能比 MOP 解小。可以用一个例子来证明这个框架不是可分配的。

例 9.26 在图 9-27 的程序中, x 和 y 在基本块 B_1 中被分别设置为 2 和 3, 而在基本块 B_2 中被分别设置为 3 和 2。我们知道, 不管按照哪条路径执行, 在基本块 B_3 的结尾处 z 的值都是 5。但是, 上面的迭代算法没有发现这个事实。相反地, 它在 B_3 的入口处应用交汇运算, 并把 x 和 y 的值都设置为 NAC。因为两个 NAC 相加的结果还是 NAC, 算法 9.25 在程序的出口处产生的输出是 $z = \text{NAC}$ 。这个结果是安全的, 但是不够精确。算法 9.25 不够精确的原因是它没有跟踪 x 和 y 之间的相关性: 当 x 是 2 时 y 必然是 3, 而当 x 是 3 时 y 必然是 2。可以使用一个更加复杂的框架来跟踪包含程序变量的表达式之间的相等关系, 但是这个方法的代价要高得多。这个方法将在练习 9.4.2 中讨论。

从理论上讲, 我们可以把精确度的丧失归因于常量传播框架的不可分配性。令 f_1, f_2, f_3 分别是代表基本块 B_1, B_2, B_3 的传递函数。如图 9-28 所示,

$$f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$$

体现了这个框架的不可分配性。□

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC

图 9-26 $x = y + z$ 的常量传播函数

⊕ 和往常一样, + 表示一个一般性的运算符号, 而不是只表示加法。

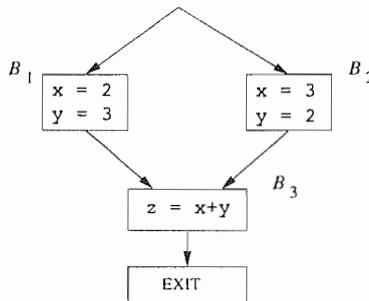


图 9-27 一个说明常量传播框架不可分配的例子

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

图 9-28 不可分配的传递函数的例子

9.4.6 对算法结果的解释

在迭代算法中使用值 UNDEF 有两个目的：初始化 ENTRY 结点，以及在迭代之前对程序内部的点进行初始化。UNDEF 在这两种情况下的含义略有不同。第一种情况是说变量在程序开始执行的时候是没有定值的；第二种情况是表示因为在迭代过程开始的时候缺乏信息，因此我们把解近似估算为顶元素 UNDEF。在迭代过程结束后，在 ENTRY 结点的出口处各个变量的值仍然是 UNDEF，因为 OUT[ENTRY] 不会改变。

UNDEF 值也可能出现在某些其他的程序点上。它们的出现意味着在到达该程序点的所有路径中尚未发现任何对该变量的定值。请注意，根据我们定义交汇运算的方式，只要有一个对该变量定值的路径到达该程序点，变量的值就不是 UNDEF 了。如果到达一个程序点的所有定值都有同样的常量值，那么即使该变量可能在某些路径上没有被定值，它仍然会被当作是常量。

如果假设被分析的程序是正确的，我们的算法就可以发现比不做这个假设时更多的常量。也就是说，我们的算法会为可能未定值的变量选择适当的值，以便程序能够更加高效地执行。在大多数程序设计语言中，这种改变是合法的，因为在这些语言中未定值的变量可以取任何值。如果语言的语义要求所有未定值的变量取某个特定的值，那么我们就必须相应地改变在这个数据流问题中使用的公式。如果我们对寻找程序中可能未定值的变量感兴趣，就可以用公式刻画出一个不同的数据流分析问题，以提供相应的结果（见练习 9.4.1）。

例 9.27 在图 9-29 中，变量 x 在基本块 B_2 和 B_3 的出口处的值分别为 10 和 UNDEF。因为 $\text{UNDEF} \wedge 10 = 10$ ， x 在基本块 B_4 的入口点的值是 10。因此可以在使用 x 的基本块 B_5 中把 x 替换为常量 10，从而对 B_5 进行优化。如果被执行的路径是 $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$ ，那么到达基本块 B_5 时 x 的值尚未定值。因此把对 x 的使用替换为 10 看起来是不正确的。

但是如果断言 Q' 为真时断言 Q 不可能为假，那么这个执行路径实际上不可能出现。虽然程序员可能知道这个事实，但判定这个事实已经超出了任何数据流分析技术的能力。因此，如果我们假设程序是正确的，并且所有变量在被使用之前都已经定值，那么 x 在基本块 B_5 开始处的值只能是 10。如果程序一开始就是不正确的，那么选择 10 作为 x 的值不可能比允许 x 取随机值的效果更糟。□

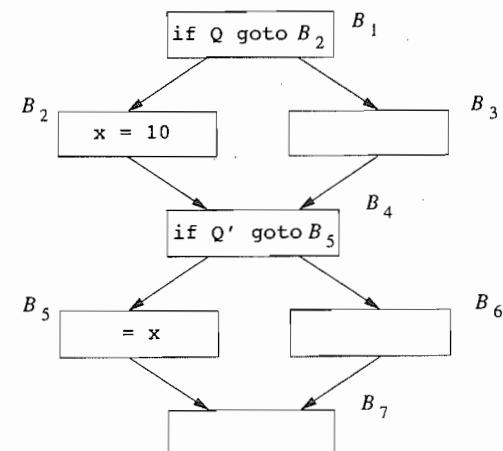


图 9-29 UNDEF 和一个常量值的交汇运算值

9.4.7 9.4节的练习

! 练习 9.4.1：假设我们希望检测一个变量是否有可能在尚未初始化的情况下到达某个使用它的程序点。你将如何修改本节中的框架来检测这种情况？

!! 练习 9.4.2：在一个有趣且功能强大的数据流分析框架中，值域 V 是对表达式的所有可能的分划。两个表达式在此分划的同一个等价类中当且仅当沿着任何路径到达问题中的程序点时它们一定具有相同的值。为了避免列出无穷多个表达式，我们可以只列出最少的等值表达式对来表示 V 。比如，如果我们执行语句

```
a = b
c = a + d
```

那么最小的等值表达式对的集合是 $\{a \equiv b, c \equiv a - d\}$ 。从这些表达式对可以推出其他的等值关系，比如 $c \equiv b + d$ 和 $a + e \equiv b + e$ 等，但是没有必要明确地把这些表达式都列出来。

- 1) 适用于这个框架的交汇运算是什么？
- 2) 给出一个数据结构来表示域中的值，并给出一个算法来实现交汇运算。
- 3) 适用于各个语句的传递函数是什么？解释一下 $a = b + c$ 这样的语句对于一个表达式分划（即 V 中的一个值）的影响。
- 4) 这个框架是单调的吗？是可分配的吗？

9.5 部分冗余消除

在本节中，我们详细考虑如何尽量减少表达式求值的次数。也就是说，我们希望考虑一个流图中所有可能的执行顺序并检查 $x + y$ 这样的表达式被求值的次数。通过移动各个对 $x + y$ 求值的位置，并在必要时把求值结果保存在临时变量中，我们常常可以在很多执行路径中减少这个表达式被求值的次数，并保证不增加任何路径中的求值次数。请注意，在流图中 $x + y$ 被求值的位置可能增多，但是相对来说这一点并不重要，只要对表达式 $x + y$ 求值的次数被减少就行了。

应用本节开发的代码转换可以提高所生成的代码的性能。这是因为，正如我们即将看到的，在改进后的代码中，只有在绝对必要时才会进行一次运算。每个优化编译器都或多或少地实现了本节中描述的转换，虽然有些编译器使用的算法没有本节中的算法那么“激进”。但是，还有另一个动机促使我们来讨论这个问题。在流图中寻找（一个或多个）适当的位置来对各个表达式求值需要进行四种不同的数据流分析。因此，对“部分冗余消除”（即尽量减少表达式求值次数的技术）的研究可以帮助我们理解数据流分析技术在编译器中所扮演的角色。

程序中的冗余以多种形式存在。如 9.1.4 节所讨论的，它可能以公共子表达式的形式存在，即对表达式的多次求值产生同样的结果。它也可能以循环不变表达式的方式存在，这个表达式在循环的每次迭代中都得到相同的值。冗余也可能是部分性的，即只能在部分路径而不是全部路径中找到这个冗余。公共子表达式和循环不变表达式可以被看作是部分冗余的特例。因此可以设计一个部分冗余消除算法来消除不同种类的冗余。

接下来，我们首先讨论冗余的不同形式，以便对这个问题有直观的理解。然后再描述一般性的冗余消除问题，并在最后给出解决问题的算法。这个算法很有意思，因为它涉及多种数据流问题的求解。这些问题中既有前向问题，也有逆向问题。

9.5.1 冗余的来源

图 9-30 演示了冗余的三种形式：公共子表达式、循环不变表达式和部分冗余表达式。该图中给出了优化之前和之后的代码。

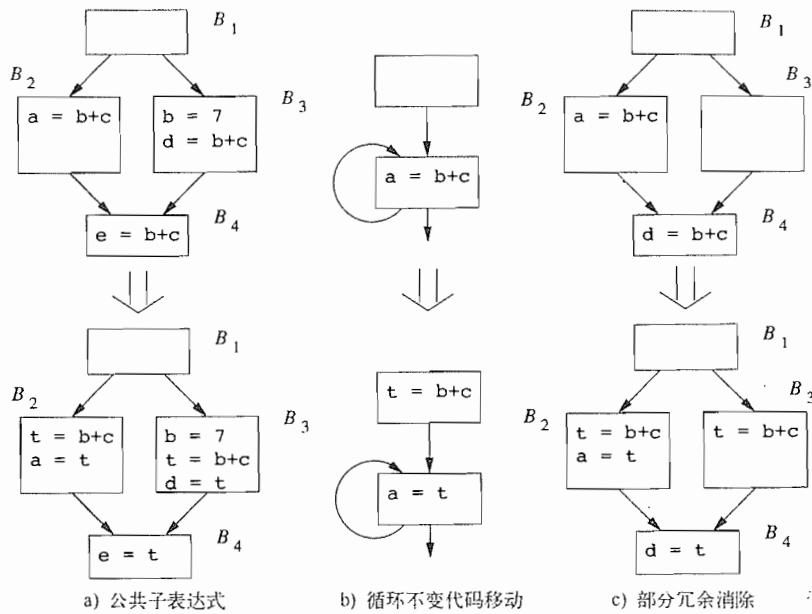


图 9-30 各种冗余的例子

全局公共子表达式

在图 9-30a 中, 基本块 B_4 中计算的表达式 $b + c$ 是冗余的。不管按照哪条路径, 当控制流到达 B_4 时这个表达式在之前已经被求过值了。正如我们在这个例子中观察到的, 在不同的路径上表达式可能取不同的值。我们可以按照下面的方法优化代码。在基本块 B_2 和 B_3 中把 $b + c$ 的计算结果存放到同一个临时变量(比如说 t)中。然后在基本块 B_4 中不再重新计算这个表达式, 而是直接把 t 的值赋给 e 。如果在对 $b + c$ 的最后一次求值之后以及基本块 B_4 之前对 b 或 c 有一个赋值运算, 那么 B_4 中的这个表达式就不再是冗余的。

正式地讲, 如果按照 9.2.6 节的说法, 一个表达式 $b + c$ 在程序点 p 上是一个可用表达式, 我们就说该表达式在该点上是(完全)冗余的。也就是说, 在所有到达 p 的路径中, 表达式 $b + c$ 已经被求过值, 并且在最后一次求值之后变量 b 和 c 没有被重新定值。后一个条件是必须的, 因为虽然从字面上看表达式 $b + c$ 在到达点 p 时已经执行过了, 但在 p 点计算得到的 $b + c$ 的值可能是不同的, 因为运算分量可能已经改变了。

寻找“深层”公共子表达式

使用可用表达式分析来寻找冗余表达式时只能够找出字面上相同的可用表达式。比如, 如果两个代码片断

$t1 = b + c; a = t1 + d;$

和

$t2 = b + c; e = t2 + d;$

之间 b 和 c 没有被重新定值, 那么应用公共子表达式消除可以发现在第一个代码片断中的 $t1$ 和第二个代码片断中的 $t2$ 具有相同的值。但是它无法发现 a 和 e 的值也相同。如果要找出这一类“深层”的公共子表达式, 我们可以重复应用公共子表达式消除技术, 直到某一次应用时找不到新的公共子表达式为止。另一种可能的方法是使用练习 9.4.2 中的框架来找出“深层”公共子表达式。

循环不变表达式

图 9-30b 给出了一个循环不变表达式的例子。假设变量 b 和 c 在循环中没有被重新定值，那么 $b + c$ 就是循环不变的。我们可以把循环中的所有重复执行替换为循环外的单次计算，从而优化程序。我们把计算的结果赋予一个临时变量，比如说 t ，然后把循环中的表达式替换为 t 。当进行与此类似的“代码移动”优化时，我们还需要考虑另一个问题。我们不应该执行任何在未优化时不执行的指令。比如，如果有可能在不执行这个循环不变指令时就离开循环，那么我们就应该把该指令移动到循环之外。这样做有两个原因。

- 1) 如果该指令会引发一个异常，那么执行此指令可能会抛出一个原程序中本来不会发生的异常。

- 2) 如果循环提早退出，“优化”过的程序需要的执行时间比原程序更多。

为了保证 while 循环中的循环不变表达式可以被优化，编译器通常把语句

```
while c {
    S;
}
```

表示成为下面的等价语句

```
if c {
    repeat
        S;
    until not c;
}
```

通过这种方法，各个循环不变表达式可以直接放置在 repeat-until 结构之前。

在公共子表达式消除中，一个冗余的表达式计算被直接丢弃。循环不变表达式消除和公共子表达式消除不同，它要求把循环内的一个表达式移动到循环之外。因此，这个优化通常叫做“循环不变代码移动”。循环不变代码移动可能需要重复进行，因为一旦一个变量被确定具有循环不变的值，使用这个变量的某些表达式也可能成为循环不变的。

部分冗余表达式

一个部分冗余表达式的例子如图 9-30c 所示。基本块 B_4 中的表达式 $b + c$ 在路径 $B_1 \rightarrow B_2 \rightarrow B_4$ 上冗余，但是在路径 $B_1 \rightarrow B_3 \rightarrow B_4$ 上不冗余。我们可以在基本块 B_3 上放一个计算 $b + c$ 的指令，从而消除前一条路径上的冗余。所有 $b + c$ 的计算结果都被写进临时变量 t ，并且 B_4 中对 $b + c$ 的计算用 t 替代。因此，和循环不变代码移动一样，部分冗余消除需要放置一些新的表达式计算指令。

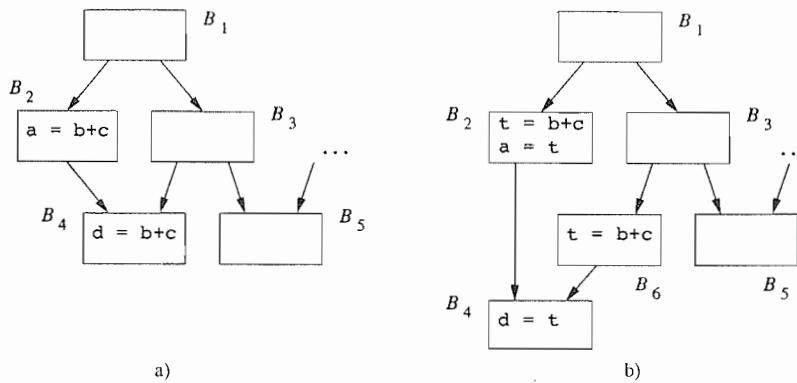
9.5.2 可能消除所有冗余吗

可能消除各条路径上的所有冗余计算吗？除非我们能够通过创建新的基本块来改变流图，否则答案是“不能”。

例 9.28 在图 9-31a 显示的例子中，如果程序的执行路径是 $B_1 \rightarrow B_2 \rightarrow B_4$ ，则表达式 $b + c$ 在基本块 B_4 中冗余地计算。但是，我们不能简单地把 $b + c$ 的计算指令移到 B_3 ，因为这么做会在执行路径为 $B_1 \rightarrow B_3 \rightarrow B_5$ 时多计算一次 $b + c$ 。

我们想做的是在基本块 B_3 和 B_4 之间的边上插入 $b + c$ 的计算指令。为了插入这个指令，我们可以创建一个新的基本块，比如 B_6 ，把该指令放到 B_6 中，并使得从 B_3 开始的控制流首先经过 B_6 再到达 B_4 。这个转换显示在图 9-31b 中。□

我们把所有从一个具有多个后继的结点到达另一个具有多个前驱的结点的边定义为流图的关键边 (critical edge)。通过在关键边上引入新的基本块，我们总是可以找到一个基本块作为放置表达式的适当位置。比如在图 9-31b 中从 B_3 到 B_4 的边就是关键边，因为 B_3 具有两个后继而 B_4 有两个前驱。

图 9-31 $B_3 \rightarrow B_4$ 是一条关键边

仅靠增加基本块可能不足以消除所有的冗余计算。如例 9.29 所示，我们要复制代码，以便把找到的具有冗余特性的路径隔离开来。

例 9.29 在图 9-32a 所示的例子中，表达式 $b+c$ 沿着路径 $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$ 被冗余地计算。我们可能愿意从这条路径的基本块 B_6 中删除 $b+c$ 的冗余计算，并只在路径 $B_1 \rightarrow B_3 \rightarrow B_4 \rightarrow B_6$ 上计算这个表达式。但是，源程序中没有哪个程序点或边唯一地对应于第二条路径。为了创建这样一个程序点，我们可以复制一对基本块 B_4 和 B_6 ，其中的一对经过 B_2 到达，而另一对经过 B_3 到达，如图 9-32b 所示。基本块 B_2 中的表达式 $b+c$ 的结果存放在 t 中，并在 B'_6 中被移动到变量 d 中。 B'_6 是从 B_2 到达的 B_6 的拷贝。

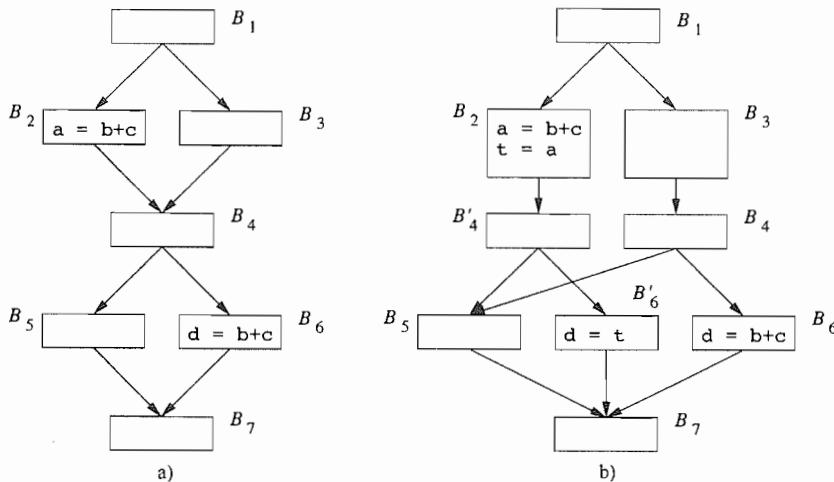


图 9-32 为消除冗余所做的代码复制

因为路径数目和程序中条件分支的数目之间具有指数关系，所以消除所有的冗余表达式可能会大大增加优化后代码的大小。因此我们对所讨论的冗余消除技术做了一些限制，即只允许引入新的基本块，但是不允许复制控制流图的任何部分。

9.5.3 懒惰代码移动问题

我们期望使用部分冗余消除算法进行优化而得到的程序能够具有下列性质：

- 1) 所有不复制代码就可以消除的表达式冗余计算都被消除掉了。
- 2) 优化后的程序不会执行原来的程序中不执行的任何计算。

3) 表达式的计算时刻应该尽量靠后。

最后一个性质是很重要的，因为找到的冗余表达式的值通常会在被使用之前一直存放在寄存器中。尽量靠后地计算一个值可以尽可能地降低该值的生命周期，即从该值被定值的时刻到它最后被使用的时刻之间的时间间隔。缩短生命周期也就尽可能降低了它使用寄存器的时间。我们把以尽可能延迟计算为目标的部分冗余消除优化称为懒惰代码移动。

为了形成对于这个问题的直观理解，我们首先讨论如何推导单条路径上的某个表达式是否具有部分冗余性。为方便起见，我们在下面的讨论中假设每个语句都是由它自己组成的单语句基本块。

完全冗余

如果在到达基本块 B 的所有路径中，一个表达式 e 已经被求过值且 e 的运算分量在其后没有被重新定值，那么 B 中的 e 就是冗余的。令 S 是那些使得基本块 B 中的 e 变得冗余，并且包含表达式 e 的基本块的集合。所有的从 S 中的某个基本块离开的边必然形成一个割集(cut set)。如果把这些边删除，那么基本块 B 必然和程序的入口点分离。而且，在从 S 中的基本块到 B 的路径中， e 的所有运算分量都没有被重新定值。

部分冗余

如果基本块 B 中的一个表达式 e 只是部分冗余，那么懒惰代码移动算法将在该流图中放置这个表达式的附加拷贝，试图使得 B 中的 e 成为完全冗余的。如果该尝试成功，那么经过优化后的流图也会有一个基本块的集合 S ，其中的每个基本块都包含表达式 e ，并且离开它们的边成为程序入口和 B 的割集。和完全冗余的情况一样， e 的所有运算分量都不会在从 S 中的基本块到 B 的路径上被重新定值。

9.5.4 表达式的预期执行

对于被插入的表达式还有一个约束，即保证优化后的程序不会执行额外的运算。一个表达式的各个拷贝所放置的程序点必须预期执行(anticipated)此表达式。如果从程序点 p 出发的所有路径最终都会计算表达式 $b + c$ 的值，并且 b 和 c 在那时的值就是它们在点 p 上的值，那么我们说一个表达式 $b + c$ 在程序点 p 上被预期执行。

现在让我们研究一下在一个无环路径 $B_1 \rightarrow B_2 \rightarrow \dots \rightarrow B_n$ 中消除部分冗余时应该做些什么。假设表达式 e 仅仅在 B_1 和 B_n 中求值，并且 e 的运算分量没有在这条路径的基本块中被重新定值。假设有一些边和上面的路径交汇，也有一些边从这条路径离开。我们看到， e 在基本块 B_i 的入口处没有被预期执行当且仅当存在一个从 B_j ($i \leq j < n$) 发出的边，它通向一条不使用 e^\ominus 的值的执行路径。因此，对执行的预期限制了一个表达式最前可以被插入到哪里。

我们就可以创建一个包含了边 $B_{i-1} \rightarrow B_i$ 的满足下面条件的割集。如果 e 在 B_i 的入口处可用或者被预期执行，这个割集就使得 e 在 B_n 中冗余。如果 e 在 B_i 的入口处被预期执行却不可用，我们必须在这条进入 B_i 的边上放一个 e 的拷贝。

我们可以选择放置表达式拷贝的位置，因为流图中通常有多个割集满足所有要求。在上面的讨论中，表达式的计算在进入我们所关心的路径的边上引入。这样做可以在不引入冗余计算的情况下使得表达式的计算尽量靠近对表达式值的使用。请注意，这些被引入的运算本身可能因为程序中同一个表达式的其他实例而体现出部分冗余性。这种部分冗余性可以通过进一步上移计算过程而消除。

\ominus 请注意，对表达式值的使用和对变量值的使用不同，它实际上是说该表达式出现在某个语句的右部。——译者注

总结一下，对表达式的预期执行限制了一个表达式可以被放置得有多靠前。不能把它放置得太靠前，以至于放置它的地方还没有预期执行它。一个表达式被放置得越靠前，能够删除的冗余性就越多。在能够消除同样的冗余性的各个解中，最后一个计算该表达式的位置可以使存放该表达式的寄存器的生命周期最小化。

9.5.5 懒惰代码移动算法

至此，这里的讨论给出了一个包含四个步骤的算法。第一步使用执行预期来确定表达式可以被放在哪里；第二步寻找最前的能够在不复制代码且不引入不必要计算的情况下消除最多的冗余运算的割集。这个步骤把表达式的计算放置在最前的预期执行这些表达式的程序点上。第三步把割集尽量向后推，直到继续后推会改变程序语义或引入新的冗余为止。最后的第四步很简单，它删除那些给只使用一次的对临时变量赋值的语句，达到清洗代码的目的。每一个步骤都伴随一个数据流分析过程：第一个和第四个是逆向数据流问题，而第二个和第三个是前向的数据流问题。

算法概览

- 1) 使用一个逆向数据流分析过程找到各个程序点上预期执行的所有表达式。
- 2) 第二步把对表达式的计算放置在满足下面条件的程序点上。对于每个这样的点，总存在某条路径使得这些点是此路径中第一个预期执行这个表达式的点。我们在一个表达式最先被预期执行的地方放置了该表达式的拷贝之后，假设有一个这样的程序点 p ，所有到达它的原有路径中该表达式都被预期执行，那么现在该表达式在程序点 p 上可用。可用性可以用一个前向的数据流分析过程完成。如果我们希望把这个表达式放置在尽量靠前的地方，我们只要找出满足下面条件的程序点就可以了：在这些点上此表达式被预期执行但是不可用。
- 3) 在一个表达式最早被预期执行的地方对表达式求值可能会使得表达式的值在被使用之前很久就被生成了。一个表达式可被后延到某个程序点的条件如下：在到达这个点的所有路径上，这个表达式在这个程序点之前已经被预期执行，但是还没有使用这个值。可后延表达式通过使用一个前向的数据流分析过程找到。我们把表达式放置在不能再后延的程序点上。
- 4) 使用一个简单的逆向数据流分析过程来删除那些给程序中只使用一次的临时变量赋值的语句。

预处理步骤

我们现在给出完整的懒惰代码移动算法。为了使算法简单一些，我们假设开始时每个语句自己组成一个基本块，而且我们只在基本块的开头引入新的表达式计算指令。为了保证这个简化不会降低这个技术的有效性，如果一个边的目标结点有多个前驱，我们就在这个边的源结点和目标结点之间插入一个新的基本块。这么做显然也考虑了对程序中所有的关键边。

我们把每个基本块 B 的语义抽象为两个集合： e_use_B 表示 B 中计算的表达式，而 e_kill_B 表示被 B 杀死的表达式，即某个运算分量在 B 中定值的表达式的集合。在对懒惰代码移动技术中的四个数据流分析模式进行讨论时，将会一直使用例 9.30。这四个数据流分析模式在图 9-34 中进行了简单的定义。

例 9.30 在图 9-33a 的流图中，表达式 $b + c$ 出现三次。因为基本块 B_9 是一个循环的一部分，块中的表达式 $b + c$ 可能被执行很多次。在 B_9 中对此表达式的计算不仅是循环不变的，它还是一个冗余表达式，因为表达式的值已经在基本块 B_7 中使用。对于这个例子来说，我们只需要计算 $b + c$ 两次：一次在基本块 B_5 中计算，而另一次在 B_2 之后到 B_7 之前的路径上计算。本节讨论的懒惰代码移动算法将把表达式计算放置在基本块 B_4 和 B_5 的开头。□

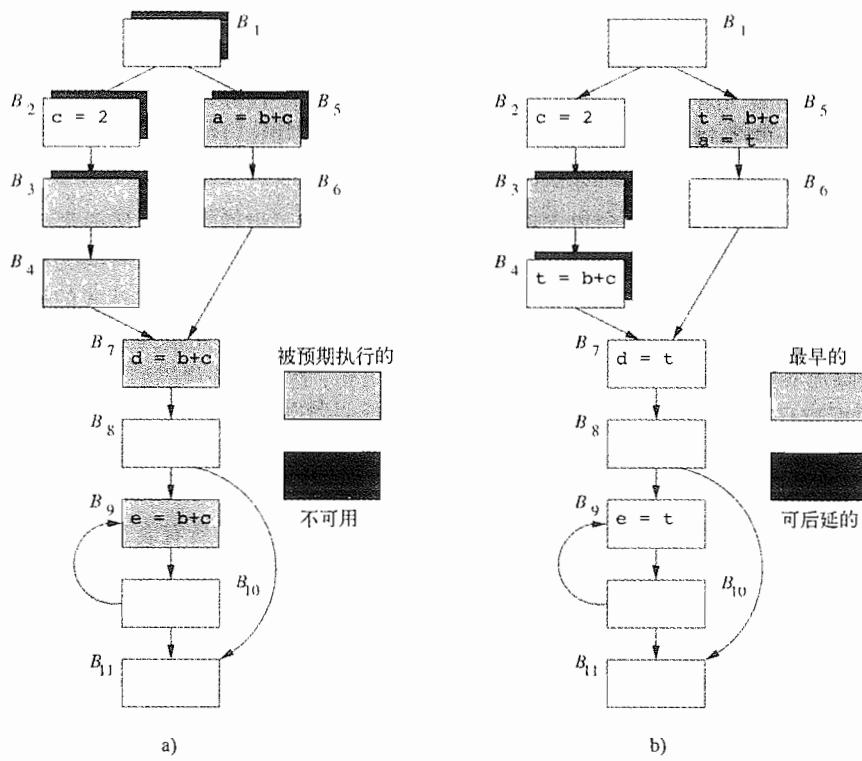


图 9-33 例 9.30 的流图

预期执行的(anticipated)表达式

回顾一下预期执行的定义。如果从程序点 p 出发的所有路径最终都会计算表达式 $b + c$ 的值，并且计算时 b 和 c 的值就是它们在点 p 上的值，那么我们说表达式 $b + c$ 在程序点 p 上被预期执行。

在图 9-33a 中，所有在其入口处预期执行表达式 $b + c$ 的基本块都用浅灰色方块表示。表达式 $b + c$ 在基本块 B_3 、 B_4 、 B_5 、 B_6 、 B_7 和 B_9 中被预期执行。它在 B_2 的入口处没有被预期执行，这是因为 c 的值在该基本块内被重新计算，因此假如在 B_2 的开始处计算 $b + c$ 的值，这个计算结果不会在任何路径上被使用。在 B_1 的入口处也没有预期执行 $b + c$ ，因为在从 B_1 到 B_2 的分支上这个计算是不必要的（虽然在路径 $B_1 \rightarrow B_5 \rightarrow B_6$ 上使用了这个计算）。类似地，因为有 B_8 到 B_{11} 的分支，该表达式也没有在 B_8 的开头被预期执行。一条路径上的各个结点是否预期执行一个表达式可能会不断交替变化，如路径 $B_7 \rightarrow B_8 \rightarrow B_9$ 所示。

预期执行表达式问题的数据流方程组如图 9-34b 所示。问题的分析过程是逆向的。只有当一个表达式在基本块 B 的出口处被预期执行，且它不在 e_kill_B 集合中，那么它在基本块的入口处也被预期执行。基本块 B 同时也生成一个表达式集合 e_use_B ，表示基本块 B 新使用了其中的表达式。在一个程序的出口处没有表达式被预期执行。我们关心的是在所有后继路径中都被预期执行的表达式，因此交汇运算是交集运算。因此，和我们在 9.2.6 节中讨论可用表达式时类似，内部程序点的初始值是表达式的全集 U 。

可用(avaiable)表达式

第二步之后，一个表达式的多个拷贝会被分别放置到该表达式首次被预期执行的程序点上。

这么做之后，如果原来的程序中所有到达程序点 p 的路径都预期执行这个表达式，那么现在这个表达式就在点 p 上可用。这个问题和 9.2.6 节中描述的可用表达式问题类似。但是这里使用的传递函数略有不同。一个表达式在一个基本块的出口处可用的条件有两个：

	a) 被预期执行的表达式	b) 可用表达式
域	表达式集合	表达式集合
方向	逆向	前向
传递函数	$f_B(x) = e_use_B \cup (x - e_kill_B)$	$f_B(x) = (anticipated[B].in \cup x) - e_kill_B$
边界条件	$IN[EXIT] = \emptyset$	$OUT[ENTRY] = \emptyset$
交汇运算(\wedge)	\cap	\cap
方程组	$IN[B] = f_B(OUT[B])$ $OUT[B] = \bigwedge_{S,succ(B)} IN[S]$	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P,pred(B)} OUT[P]$
初始化	$IN[B] = U$	$OUT[B] = U$
	c) 可后延表达式	d) 被使用的表达式
域	表达式集合	表达式集合
方向	前向	逆向
传递函数	$f_B(x) = (earliest[B] \cup x) - e_use_B$	$f_B(x) = (e_use_B \cup x) - latest[B]$
边界条件	$OUT[ENTRY] = \emptyset$	$IN[EXIT] = \emptyset$
交汇运算(\wedge)	\cap	\cup
方程组	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P,pred(B)} OUT[P]$	$IN[B] = f_B(OUT[B])$ $OUT[B] = \bigwedge_{S,succ(B)} IN[S]$
初始化	$OUT[B] = U$	$IN[B] = \emptyset$

$$\begin{aligned} earliest[B] &= anticipated[B].in - available[B].in \\ latest[B] &= (earliest[B] \cup postponable[B].in) \cap \\ &\quad (e_use_B \cup \neg(\bigcap_{S,succ(B)} (earliest[S] \cup postponable[S].in))) \end{aligned}$$

图 9-34 部分冗余消除中的四个数据流分析过程

1) 下列条件之一成立

① 在入口处可用。

② 在基本块的入口处所预期执行的表达式集合中(即如果我们选择在入口处计算这个表达式，它就会在入口处变得可用)。

2) 没有被这个基本块杀死。

用于可用表达式的数据流方程组如图 9-34b 所示。为了避免混淆 IN 的含义，我们在数据流分析问题的名字后加上“ $[B].in$ ”，以这种方式来表示某次分析所得到的结果。

依据最前放置的策略而在一个基本块 B 上放置的表达式的集合(即 $earliest[B]$)被定义为被预期执行但不可用的表达式集合，即

$$earliest[B] = anticipated[B].in - available[B].in.$$

例 9.31 图 9-35 的流图中表达式 $b + c$ 在基本块 B_3 的入口点没

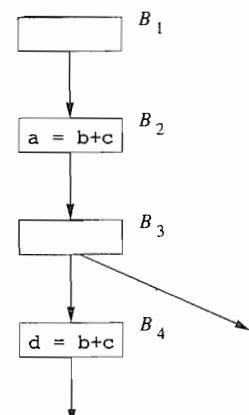


图 9-35 例 9.31 的流图，用以说明可用表达式的使用

有被预期执行，但是在基本块 B_4 的入口处被预期执行。然而，没有必要在基本块 B_4 中计算表达式 $b + c$ ，因为 B_2 使得表达式 $b + c$ 在此处变得可用。 \square

例 9.32 图 9-33a 中带有黑色阴影的各个基本块上的表达式 $b + c$ 不可用，这些基本块是 B_1 、 B_2 、 B_3 和 B_5 。该表达式的靠前放置的位置使用带有黑色阴影的灰色方块表示，它们是 B_3 和 B_5 。请注意， $b + c$ 被认为在 B_4 的入口处可用，因为在一条路径 $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$ 中， $b + c$ 至少被预期执行一次——在这个例子里是 B_3 ——并且从 B_3 的入口点开始， b 和 c 都没有被重新计算。 \square

2 × 2 方块的补全

被预期执行的表达式（其他文献中也称之为“很忙的表达式”）是一类我们之前没有看到的数据流分析。虽然我们已经看到了活跃变量分析（见 9.2.5 节）这样的逆向框架，且我们看到了可用表达式分析（9.2.6 节）那样使用交集运算作为交汇运算的框架。这是第一个具有这两个特点的有用分析技术的例子。几乎我们使用的所有分析技术都可以放到四个分组中的某一个中。这四个组按照下面的方法进行刻划：它们是前向的还是逆向的，它们是使用并集运算还是交集运算作为交汇运算（可以按照这两个特性的不同取值把各个数据流分析模式分别放到一个 2×2 方块中的某个空格中，而本节的分析技术填补了方阵中的一个空格，译者注）。同时请注意，使用并集的分析总是涉及是否存在一条路径使得某件事情为真，而使用交集的分析考虑的是某些事情是否对于所有的路径都为真。

可后延 (postponable) 表达式

算法的第三步在保持原程序语义并将冗余最小化的情况下把表达式的计算尽量地延后。例 9.33 说明了这个步骤的重要性。

例 9.33 在图 9-36 所示的流图中，表达式 $b + c$ 在路径 $B_1 \rightarrow B_5 \rightarrow B_6 \rightarrow B_7$ 中被计算两次。表达式 $b + c$ 甚至在基本块 B_1 的开头就被预期执行了。如果我们在表达式被预期执行的时候立刻计算它的值，那么我们就要在 B_1 中计算 $b + c$ 的值。计算结果将在一开始就被保存起来，经过由基本块 B_2 、 B_3 组成的循环的执行，最后由基本块 B_7 使用。在另一种方法中，我们可以把表达式 $b + c$ 的计算推迟到 B_5 的开始以及控制流即将从 B_4 到达 B_7 的时候。 \square

正式地讲，一个表达式 $x + y$ 可后延到程序点 p 的前提如下：在所有从程序入口结点到达 p 的路径中都会碰到一个位置较前的 $x + y$ ，并且在最后一个这样的位置到 p 之间没有对 $x + y$ 的使用。

例 9.34 让我们再次考虑图 9-33 中的表达式 $b + c$ 。其中可放置 $b + c$ 的两个最前的点是 B_3 和 B_5 。请注意，这两个基本块在图 9-33a 中都被表示为带有黑色阴影的灰色方块，这表示在且只在这两个基本块上 $b + c$ 被预期执行但不可用。我们不能把 $b + c$ 从 B_5 后延到 B_6 ，因为 $b + c$ 在 B_5 中被使用了。但是我们可以把它从 B_3 后延到 B_4 。

但是，我们不能把 $b + c$ 从 B_4 后延到 B_7 。原因是虽然 $b + c$ 在 B_4 中没有使用，把它放到 B_7 中而不是 B_4 中会引起路径 $B_5 \rightarrow B_6 \rightarrow B_7$ 上的冗余计算。我们将看到， B_4 是我们能够计算 $b + c$ 的最后位置之一。 \square

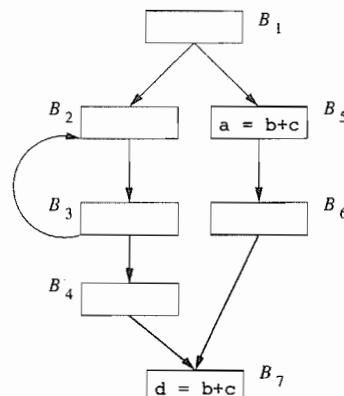


图 9-36 例 9.33 的流图，用以说明后延一个表达式的需求

可后延表达式问题的数据流方程组如图 9-34c 所示。这个分析过程是前向的。我们不能把一个表达式“后延”到程序的入口处，因此 $\text{OUT}[\text{ENTRY}] = \emptyset$ 。如果一个表达式在 B 中没有使用，且它以后延到 B 的入口处，或者它在 $\text{earliest}[B]$ 中，那么它就可以被后延到 B 的出口处。除非一个基本块的所有前驱结点出口处的可后延集合中都包含某个表达式，否则该表达式不能被后延到这个基本块的入口处。因此，这个数据流分析的交汇运算是交集运算，并且各个内部程序点必须被初始化为相应半格的顶元素——全集。

粗略地说，一个表达式将被放置在边界上，即一个表达式从可后延转变成为不可后延的地方。更加明确地说，表达式 e 可以被放置在基本块 B 的开始处的前提条件是该表达式在 B 入口处的 earliest 集合或可后延集合中。另外，当下列条件之一成立时， B 在 e 的后延边界中：

- 1) e 不在集合 $\text{postponable}[B].out$ 中。换句话说， e 在 e_use_B 中。
- 2) e 不能被后延到 B 的某个后继基本块。换句话说，存在一个 B 的后继基本块使得 e 不在该后继入口处的 earliest 集合和可后延集合中。

因为在算法的预处理阶段引入了新的基本块，所以在上述两种情形中，表达式 e 可以放在基本块 B 的前面。

例 9.35 图 9-33b 显示了上述分析的结果。其中的灰色方块表示了相应 earliest 集合中包含 $b+c$ 的基本块，而黑色阴影的方块表示了相应可后延集合中包含 $b+c$ 的基本块。因此，表达式 $b+c$ 的最后放置位置在基本块 B_4 和 B_5 的入口处，这是因为

- 1) $b+c$ 在 B_4 的可后延集合中，但是不在 B_7 的可后延集中，并且
- 2) B_5 的 earliest 集合包含了 $b+c$ ，并且它使用了 $b+c$ 。

如图所示，该表达式的值在基本块 B_4 和 B_5 中被存放到临时变量 t 中，在任何其他地方的 $b+c$ 都被替换为 t 。□

被使用的(used)表达式

最后，用一个逆向分析过程来确定一个被引入的临时变量是否在它所在基本块之外的其他地方使用。如果从程序点 p 出发的一条路径在表达式被重新求值之前使用了该表达式，那么我们说该表达式在点 p 上被使用。这个分析实质上是活跃性分析(是对表达式而言，而不是对变量而言)。

被使用的表达式问题的数据流方程组如图 9-34d 所示。这个分析过程是逆向的。如果一个在基本块 B 的出口点被使用的表达式不在 B 的最后放置(latest)集合中，那么它也是一个在 B 的入口点处被使用的表达式。一个基本块生成了 e_use_B 集合中的全部表达式，就是说新近使用了这些表达式。在程序的出口处没有表达式被使用。因为我们关心的是找出被任何后续路径所使用的表达式，因此这个问题的交汇运算是并集运算。因此，各个内部点必须被初始化为相应的半格的顶元素——空集。

综合全部步骤

本算法的各个步骤在算法 9.36 中进行了汇总。

算法 9.36 懒惰代码移动。

输入：一个流图，其中每个基本块 B 的 e_use_B 和 e_kill_B 已经计算得到了。

输出：一个经过修改且满足 9.5.3 节所描述的懒惰代码移动的四个条件的数据流图。

方法：

- 1) 在每条进入某个具有多个前驱的基本块的边上插入一个空基本块。
- 2) 按照 9-34a 中的定义，计算出所有基本块 B 的 $\text{anticipated}[B].in$ 的值。

3) 按照图 9-34b 中的定义, 计算出所有基本块 B 的 $\text{available}[B]. \text{in}$ 的值。

4) 为每个基本块 B 计算它的最早放置位置;

$$\text{earliest}[B] = \text{anticipated}[B]. \text{in} - \text{available}[B]. \text{in}$$

5) 按照图 9-34c 的定义, 计算出所有基本块 B 的 $\text{postponable}[B]. \text{in}$ 的值。

6) 计算所有基本块 B 的最后放置集合:

$$\text{latest}[B] = (\text{earliest}[B] \cup \text{postponable}[B]. \text{in}) \cap$$

$$(e_{\text{use}}[B] \cap \neg (\bigcup_{S \text{ 在 } \text{succ}[B] \text{ 中}} (\text{earliest}[S] \cup \text{postponable}[S]. \text{in})))$$

请注意, 其中的 \neg 表示的是以程序中所计算的全部表达式的集合作为全集的补集运算。

7) 按照图 9-34d 中的定义, 找到所有基本块 B 的 $\text{used}[B]. \text{out}$ 值。

8) 对于程序计算的每个表达式, 比如 $x + y$, 做下列处理:

① 为 $x + y$ 创建一个新的临时变量, 比如说 t 。

② 对于所有基本块 B , 如果 $x + y$ 在 $\text{latest}[B] \cap \text{used}[B]. \text{out}$ 中, 就把 $t = x + y$ 加入到 B 的开头。

③ 对于所有基本块 B , 如果 $x + y$ 在集合 $e_{\text{use}}[B] \cap (\neg \text{latest}[B] \cup \text{used}. \text{out}[B])$ 中, 就用 t 来替换原来的每个 $x + y$ 。 \square

总结

部分冗余消除技术用统一的算法归纳出不同类型的冗余计算。这个算法说明了如何使用多个数据流问题来寻找最优的表达式位置。

1) 有关位置的约束由预期执行表达式分析提供。预期执行表达式分析是一个逆向的数据流分析, 并使用交集运算作为交汇运算。因为它确定的是对于各个程序点, 一个表达式是否在该点之后的所有路径中被使用。

2) 一个表达式的最前放置位置就是该表达式在其上被预期执行但又不可用的程序点。可用表达式是通过一个前向数据流分析找到的, 它使用交集运算作为交汇运算。对各个程序点, 这个数据流分析技术计算了一个表达式是否在所有路径中都在该点之前被预期执行。

3) 一个表达式的最后放置位置就是该表达式在其上不可再后延的程序点。如果到达一个程序点的所有路径都没有碰到某个表达式, 那么该表达式在此程序点上可以后延。可后延表达式是通过一个前向的数据流分析技术找到的, 这个分析技术使用交集运算作为交汇运算。

4) 除非一个临时赋值语句被其后的某条路径使用, 否则该赋值语句可以被消除。我们通过一个逆向的数据流分析来发现被使用的表达式, 它使用并集运算作为交汇运算。

9.5.6 9.5 节的练习

练习 9.5.1: 对于图 9-37 中的流图:

1) 计算各个基本块的开头和结尾的预期执行的 (anticipated) 表达式集合。

2) 计算各个基本块的开头和结尾的可用 (available) 表达式集合。

3) 计算各个基本块的 earliest 集合。

4) 计算各个基本块的开头和结尾的可后延 (postponable) 表达式集合。

5) 计算各个基本块的开头和结尾的被使用的 (used) 表达式集合。

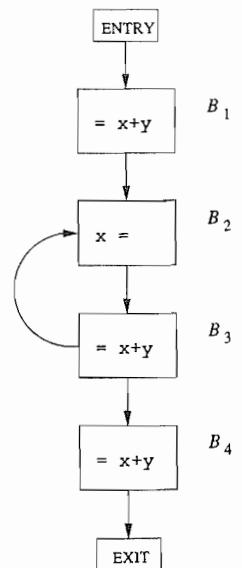


图 9-37 练习 9.5.1 的流图

6) 计算各个基本块的 *latest* 集合。

7) 引入临时变量 t , 指出它在什么地方被计算, 并在什么地方被使用。

练习 9.5.2: 对于图 9-10 中的流图(见 9.1 节的练习)重复练习 9.5.1。你可以只分析表达式 $a + b$ 、 $c - a$ 和 $b * d$ 。

!! 练习 9.5.3: 在本节中讨论的概念也可以应用到部分死亡代码的消除。如果一个变量的定值仅仅对于部分路径活跃, 但对于其他路径是死亡的, 那么这个定值就是部分死亡的 (partially dead)。我们可以只在该变量活跃的路径上执行这个定值, 从而优化这个程序的执行效率。在消除部分冗余时, 表达式被移动到原来的表达式之前; 和消除部分冗余相反, 部分死亡代码消除中新的定值被放在原来的定值之后。设计一个算法来删除部分死亡代码, 使得表达式只在一定会被使用时才进行求值。

9.6 流图中的循环

在至今为止的讨论中, 循环并没有被区别对待, 对它们的处理方式和其他类型的控制流没有什么不同。但是, 循环的重要性在于程序花费大部分时间来执行循环, 改进循环效率的优化有很大的影响。因此, 识别循环并有针对性地处理它们是很重要的。

循环也会影响程序分析所需的时间。如果一个程序不包含任何循环, 我们只需要对程序进行一趟扫描就可以得到数据流问题的答案。比如, 一个前向数据问题只需要按照拓扑次序对所有的结点进行一次访问就可以解决。

在这一节中, 我们将介绍下列概念: 支配结点、深度优先排序、回边、图的深度和可归约性。我们在后面进行的对寻找循环及迭代式数据流分析的收敛速度的讨论中需要用到这些概念。

9.6.1 支配结点

如果每一条从流图的入口结点到结点 n 的路径都经过结点 d , 我们就说 d 支配 (dominate) n , 记为 $d \text{ dom } n$ 。请注意, 在这个定义下每个结点支配它自己。

例 9.37 考虑图 9-38 中的以结点 1 作为入口结点的流图。

入口结点支配所有结点(这个结论对所有的流图都成立)。结点 2 只能支配它自己, 因为控制流可以通过以 $1 \rightarrow 3$ 开头的路径到达所有其他结点, 所以结点 3 支配除 1、2 之外的所有结点。结点 4 支配除 1、2、3 之外的所有其他结点, 因为所有从 1 开始的路径的开头要么是 $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, 要么是 $1 \rightarrow 3 \rightarrow 4$ 。结点 5 和 6 都只支配它们自身, 因为控制流可以选择从它们中的某一个结点通过, 从而绕过另一个结点。最后, 结点 7 支配结点 7、8、9、10; 结点 8 支配结点 8、9、10; 9 和 10 只支配它们自身。□

一种有用的表示支配结点信息的方法是用所谓的支配结点树(dominator tree)来表示。在树中, 入口结点就是根结点, 并且每个结点 d 只支配它在树中的后代结点。比如, 图 9-39 显示了图 9-38 中流图的支配结点树。

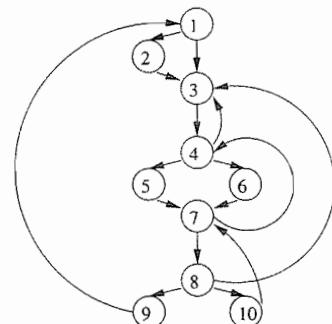


图 9-38 一个流图

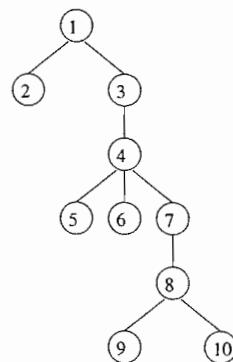


图 9-39 图 9-38
中流图的支配结点树

支配结点的一个性质决定了一定存在支配结点树：每个结点 n 具有唯一的直接支配结点（immediate dominator） m 。在从入口结点到达结点 n 的任何路径中，它是 n 的最后一个支配结点。用 dom 关系来表示， n 的直接支配结点 m 具有以下性质：如果 $d \neq n$ 且 $d dom n$ ，那么 $d dom m$ 。

我们将给出一个简单的算法来计算流图中各个结点 n 的所有支配结点。这个算法基于如下原理：如果 p_1, p_2, \dots, p_k 是 n 的所有前驱并且 $d \neq n$ ，那么 $d dom n$ 当且仅当对于每个 i ， $d dom p_i$ 。这个问题可以写成一个前向数据流分析问题。数据流的值域是基本块的集合。一个结点的支配结点集合（它自己除外）是它的所有前驱的支配结点的交集；因此这个问题的交汇运算是交集运算。基本块 B 的传递函数直接把 B 自身加入到输入结点集合中。问题的边界条件是 ENTRY 结点支配它自身。最后，内部结点的初始值是全集，也就是所有结点的集合。

算法 9.38 寻找支配结点。

输入：一个流图 G , G 的结点集是 N , 边集是 E , 而入口结点是 ENTRY。

输出：对于 N 中的各个结点 n , 给出 $D(n)$, 即支配 n 的所有结点的集合。

方法：求出由图 9-40 给定参数的数据流问题的解。输入流图的基本块就是结点。对于 N 中的所有结点 n , $D(n) = OUT[n]$ 。□

使用这个数据流算法来寻找支配结点很高效。我们将在 9.6.7 节看到，只要对流图中的结点进行几次访问就可以得到问题的解。

	支配结点
域	N 的幂集
方向	前向
传递函数	$f_B(x) = x \cup \{B\}$
边界条件	$OUT[ENTRY] = \{ENTRY\}$
交汇运算(\wedge)	\cap
方程式	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P, pred(B)} OUT[P]$
初始化设置	$OUT[B] = N$

图 9-40 一个计算支配结点的数据流算法

关系 dom 的性质

有关支配结点的一个关键性质是如果我们从入口结点沿着一个无环路径到达结点 n ，那么 n 的所有支配结点都出现在这条路径中，并且它们总是以相同顺序出现在所有这样的路径中。为了说明原因，假设在一个到达 n 的无环路径 P_1 中支配结点 a 和 b 的顺序为先 a 后 b ，而在另一条路径 P_2 中 b 在 a 之前。那么我们可以沿着 P_1 到达 a 然后再沿着 P_2 到达 n ，从而避开了 b 。因此， b 实际上不支配 n 。

通过这个推理过程，我们可以证明 dom 是传递的：如果 $a dom b$ 并且 $b dom c$ ，那么 $a dom c$ 。关系 dom 也是反对称的：如果 $a \neq b$ ，那么 $a dom b$ 和 $b dom a$ 不可能同时成立。而且，如果 a 和 b 是 n 的两个支配结点，那么 $a dom b$ 或 $b dom a$ 中必然有一个成立。最后可以推出除了入口结点之外的每个结点 n 必然有一个唯一的直接支配结点，即在从入口结点到 n 的任何无环路径中出现的离 n 最近的支配结点。

例 9.39 让我们回顾一下图 9-38 中的流图，并假设图 9-23 中第(4)到(6)行的 for 循环依照数字顺序访问其结点。令 $D(n)$ 为 $OUT[n]$ 中的结点的集合。因为 1 是入口结点，算法的第一行首先把 $\{1\}$ 赋给 $D(1)$ 。结点 2 的前驱只有 1，因此 $D(2) = \{2\} \cup D(1)$ 。这样 $D(2)$ 就被设置为 $\{1, 2\}$ 。然后考虑结点 3，它的前驱是 1、2、4 和 8。因为所有内部结点的值都被初始化为结点的全集 N ，

$$D(3) = \{3\} \cup (\{1\} \cap \{1, 2\} \cap \{1, 2, \dots, 10\} \cap \{1, 2, \dots, 10\}) = \{1, 3\}$$

其余的计算过程如图 9-41 所示。因为在图 9-23a 中，第(3)到(6)行的外层循环的第二次迭代中这些值不再改变，它们就是这个支配结点问题的最终答案。□

$$\begin{aligned}
 D(4) &= \{4\} \cup (D(3) \cap D(7)) = \{4\} \cup (\{1, 3\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4\} \\
 D(5) &= \{5\} \cup D(4) = \{5\} \cup \{1, 3, 4\} = \{1, 3, 4, 5\} \\
 D(6) &= \{6\} \cup D(4) = \{6\} \cup \{1, 3, 4\} = \{1, 3, 4, 6\} \\
 D(7) &= \{7\} \cup (D(5) \cap D(6) \cap D(10)) \\
 &= \{7\} \cup (\{1, 3, 4, 5\} \cap \{1, 3, 4, 6\} \cap \{1, 2, \dots, 10\}) = \{1, 3, 4, 7\} \\
 D(8) &= \{8\} \cup D(7) = \{8\} \cup \{1, 3, 4, 7\} = \{1, 3, 4, 7, 8\} \\
 D(9) &= \{9\} \cup D(8) = \{9\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 9\} \\
 D(10) &= \{10\} \cup D(8) = \{10\} \cup \{1, 3, 4, 7, 8\} = \{1, 3, 4, 7, 8, 10\}
 \end{aligned}$$

图 9-41 例 9.39 中支配结点计算的最终结果

9.6.2 深度优先排序

如 2.3.4 节中所介绍的，对一个流图的深度优先搜索 (depth-first search) 逐一访问图的所有结点。搜索过程从入口结点开始，并首先访问离入口结点最远的结点。一个深度优先过程中的搜索路线形成了一个深度优先生生成树 (Depth-First Spanning Tree, DFST)。2.3.4 节介绍过，一个先序遍历过程首先访问一个结点，然后从左到右递归地访问该结点的子结点。另外，一个后序遍历过程首先递归地从左到右访问一个结点的子结点，然后访问该结点本身。

还有一种排序方式对于流图分析很重要：深度优先排序 (depth-first ordering)。它的顺序正好和后序遍历的顺序相反。也就是说，在深度优先排序中，我们首先访问一个结点，然后遍历该结点的最右子结点，再遍历这个子结点左边的子结点，依此类推。但是在我们为流图构造生成树之前，我们可以选择把一个结点的哪个后继作为它在树中的最右子结点，再选择哪个后继是下一个子结点，等等。在我们给出深度优先排序的算法之前，首先考虑一个例子。

例 9.40 图 9-38 中流图的一个可能的深度优先表示法如图 9-42 所示。实线边形成了这棵树，虚线边是流图中其他的边。这棵树的深度优先遍历是 $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10$ ，然后回到 8，再到 9。我们再一次回到 8，再回到 7、6 和 4，然后前进到 5。我们从 5 回到 4，然后回到 3 和 1。我们从 1 前进到 2，然后从 2 回到 1。这样我们就遍历了整棵树。

因此，这次遍历的前序序列是：

1, 3, 4, 6, 7, 8, 10, 9, 5, 2

图 9-42 中树的后序遍历顺序是：

10, 9, 8, 7, 6, 5, 4, 3, 2, 1

深度优先排序的顺序和后序遍历序列相反，即

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

□

现在我们给出一个算法来寻找一个流图的深度优先生生成树和相应的深度优先排序。正是这个算法从图 9-38 的流图中找到了图 9-42 中的 DFST。

算法 9.41 深度优先生生成树和深度优先排序。

输入：一个流图 G 。

输出： G 的一个 DFST 树 T 和 G 中结点的一个深度优先排序。

方法：我们使用图 9-43 的递归过程 $search(n)$ 。这个算法首先把 G 的所有结点初始化为“unvisited”，然后调用 $search(n_0)$ ，其中 n_0 是入口结点。当它调用 $search(n)$ 的时候，首先把 n 标记为“visited”，以免把 n 再次加入到树中。它使用 c 作为计数器，从 G 的结点总数一直倒计数到 1。在算法执行的时候把 c 的值赋给结点 n 的深度优先编号 $dfn[n]$ 。边的集合 T 形成了 G 的深度优先生生成树。

□

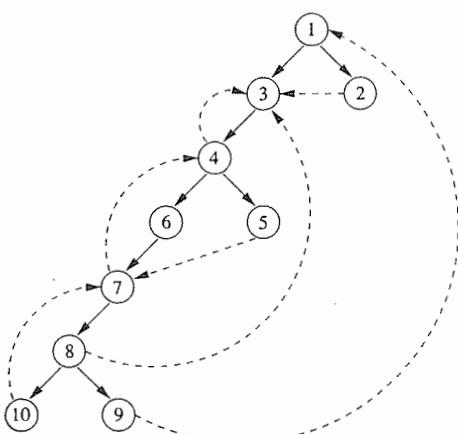


图 9-42 图 9-38 中流图的一个深度优化表示

```

void search(n) {
    将 n 标记为“visited”;
    for (n 的各个后继 s)
        if (s 标记为“unvisited”)
            将边  $n \rightarrow s$  加入到 T 中;
            search(s);
    }
    dfn[n] = c;
    c = c - 1;
}

main() {
    T = ∅; /* 边集 */
    for (G 的各个结点 n)
        把 n 标记为 “unvisited”;
    c = G 的结点个数;
    search( $n_0$ );
}

```

图 9-43 深度优先搜索算法

例 9.42 对于图 9-42 中的流图，算法 9.41 把 c 设置为 10，并调用 $\text{search}(1)$ 开始搜索。其余的执行序列显示在图 9-44 中。 \square

调用 $\text{search}(1)$	结点 1 有两个后继。假设首先考虑 $s = 3$ ， 把边 $1 \rightarrow 3$ 加入到 T 中。
调用 $\text{search}(3)$	把边 $3 \rightarrow 4$ 加入到 T 中。
调用 $\text{search}(4)$	结点 4 有两个后继，4 和 6。假设首先考虑 $s = 6$ ， 把边 $4 \rightarrow 6$ 加入到 T 中。
调用 $\text{search}(6)$	把边 $6 \rightarrow 7$ 加入到 T 中。
调用 $\text{search}(7)$	结点 7 有两个后继结点 4 和 8。但是 4 已经被 $\text{search}(4)$ 标记为“visited”，因此当 $s = 4$ 时不做任何处理。 对于 $s = 8$ ，把边 $7 \rightarrow 8$ 加入到 T 中。
调用 $\text{search}(8)$	结点 8 有两个后继，9 和 10。假设首先考虑 $s = 10$ ， 把边 $8 \rightarrow 10$ 加入到 T 中。
调用 $\text{search}(10)$	10 有后继 7，但是 7 已经被标记为“visited”。 因此 $\text{search}(10)$ 设置 $\text{dfn}[10] = 10$, $c = 9$ 并结束。
回到 $\text{search}(8)$	把 s 设置为 9，并把边 $8 \rightarrow 9$ 加入到 T 中。
调用 $\text{search}(9)$	9 的唯一后继 1 已经被设置为“visited”， 因此设置 $\text{dfn}[9] = 9$, $c = 9$ 。
回到 $\text{search}(8)$	8 的最后一个后继 3 已经是“visited”，因此 不处理 $s = 3$ 的情况。到此为止，8 的所有后继 都已经处理过了，因此设置 $\text{dfn}[8] = 8$, $c = 7$ 。
回到 $\text{search}(7)$	7 的所有后继都已经被处理过了，因此设置 $\text{dfn}[7] = 7$, $c = 6$ 。
回到 $\text{search}(6)$	6 的所有后继都已经被处理过了，因此设置 $\text{dfn}[6] = 6$, $c = 5$ 。
回到 $\text{search}(4)$	4 的后继 3 已经是“visited”，但是 5 还没有， 因此把边 $4 \rightarrow 5$ 加入到树中。
调用 $\text{search}(5)$	5 的后继 7 已经是“visited”，因此设置 $\text{dfn}[5] = 5$ ， $c = 4$ 。
回到 $\text{search}(4)$	4 的所有后继都已经被处理过了，因此设置 $\text{dfn}[4] = 4$ ， $c = 3$ 。
回到 $\text{search}(3)$	设置 $\text{dfn}[3] = 3$, $c = 2$ 。
回到 $\text{search}(1)$	2 尚未被处理，因此把边 $1 \rightarrow 2$ 加入到 T 中。
调用 $\text{search}(2)$	设置 $\text{dfn}[2] = 2$, $c = 1$ 。
回到 $\text{search}(1)$	设置 $\text{dfn}[1] = 1$, $c = 0$ 。

图 9-44 算法 9.41 在图 9-42 的流图上执行的过程

9.6.3 深度优先生成树中的边

当我们为一个流图构造 DFST 时，流图的边可以被分为三大类：

- 1) 前进边 (advancing edge)，即那些从一个结点 m 到达 m 在树中的一个真后代结点的边。DFST 中的所有边本身都是前进边。在图 9-42 中没有其他的前进边。但是，假如有一条边 $4 \rightarrow 8$ ，那么这条边就是前进边。
- 2) 有些边从一个结点 m 到达 m 在树中的某个祖先 (包括 m 自身)，我们将把这些边称为后退边 (retreating edge)。比如，图 9-42 中的 $4 \rightarrow 3, 7 \rightarrow 4, 8 \rightarrow 3, 10 \rightarrow 7$ 和 $9 \rightarrow 1$ 都是后退边。
- 3) 对于有些边 $m \rightarrow n$ ，在 DFST 中 m 和 n 都不是对方的祖先。边 $2 \rightarrow 3$ 和 $5 \rightarrow 7$ 是图 9-42 中这种边的例子。我们把这种边称为交叉边 (cross edge)。交叉边的一个重要性质是：如果我们把一个结点的子结点按照它们被加入到树中的顺序从左到右排列，那么所有的交叉边都是从右到左的。

应该注意，边 $m \rightarrow n$ 是一个后退边当且仅当 $dfn[m] \geq dfn[n]$ 。为了说明原因，请注意如果 m 是 n 在 DFST 中的一个后代，那么 $search(m)$ 在 $search(n)$ 之前运行结束，因此 $dfn[m] \geq dfn[n]$ 。反过来，如果 $dfn[m] \geq dfn[n]$ ，那么要么 $search(m)$ 在 $search(n)$ 之前结束，要么 $m = n$ 。但是如果有一条边 $m \rightarrow n$ ，那么 $search(n)$ 必须在 $search(m)$ 之前开始，否则 n 是 m 的后继的事实将使得 m 成为 n 在 DFST 中的一个后代。因此， $search(m)$ 运行的时间是 $search(n)$ 运行时间中的一个区间，由此我们可以知道 n 是 m 在 DFST 中的一个祖先。

9.6.4 回边和可归约性

回边是指一条边 $a \rightarrow b$ ，它的头 b 支配了它的尾 a 。对于任何流图，每条回边都是后退边，但并不是所有的后退边都是回边。如果一个流图的任何深度优先生成树中所有后退边都是回边，那么该流图被称为可归约的 (reducible)。换句话说，如果一个流图是可归约的，那么它的所有 DFST 的后退边的集合都是相同的，并且就是流图的回边集合。但如果流图是不可归约的 (即不是可归约的)，那么所有的回边在任何 DFST 中都是后退边，但是每个 DFST 中都可能另有一些后退边不是回边。这样的后退边集合在不同的 DFST 中有所不同。因此，如果我们删除流图中所有回边后得到的流图带有环，那么该图就是不可归约的。反过来也成立。

为什么回边是后退边

假设 $a \rightarrow b$ 是一条回边，即它的头支配它的尾。当图 9-43 中的 $search$ 函数到达 a 时，对 $search$ 的调用序列必然是流图中的一条路径。当然，这条路径必然包含 a 的所有支配结点。由此可知，当 $search(a)$ 被调用时，对 $search(b)$ 的调用必然已经开始但尚未结束。因此，当 a 被加入到树中时 b 已经在树中，并且 a 是作为 b 的一个后代被加入的。因此 $a \rightarrow b$ 必然是一条后退边。

在实践中出现的流图几乎都是可归约的。如果只使用诸如 if-then-else、while-do、continue 和 break 语句这样的结构化控制流语句，那么得到的程序的流图总是可归约的。即使使用了 goto 语句，程序也经常是可归约的，因为程序员在逻辑上会使用循环和分支的方式思考问题。

例 9.43 图 9-38 的流图是可归约的。图中的所有后退边都是回边。

也就是说，这些边的头支配各自边的尾。



例 9.44 考虑图 9-45 中的流图，它的初始结点是 1。结点 1 支配结点

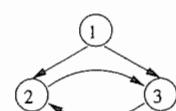


图 9-45 不可归约流图的规范形式

2和3，但是2不支配3，3也不支配2。因此，这个流图没有回边，因为没有哪条边的头支配其尾结点。根据我们选择从 $\text{search}(1)$ 首先调用 $\text{search}(2)$ 还是 $\text{search}(3)$ ，可以得到两个可能的深度优先生成树。在第一种情况下，边 $3 \rightarrow 2$ 是一个后退边但不是回边；在第二种情况下， $2 \rightarrow 3$ 是一个后退边但不是回边。直观地讲，使得这个流图不可归约的原因是环 $2-3$ 可以由两个不同的地方进入：结点2和结点3。□

9.6.5 流图的深度

给定一个流图的深度优先生成树，该流图的深度(depth)是各条无环路径上的后退边数目中的最大值。我们可以证明这个深度永远不会大于直观上所说的流图中循环嵌套的深度。如果一个流图是可归约的，那么我们可以用“回边”来替换上面的“深度”定义中的“后退边”，因为任何DFST中的后退边集合就是回边集合。深度的定义因此独立于实际所选的DFST，我们确实可以说“一个流图的深度”，而不是流图的特定于某个深度优先生成树的深度。

例 9.45 图 9-42 中流图的深度是 3，因为有一条具有三条后退边的路径

$$10 \rightarrow 7 \rightarrow 4 \rightarrow 3$$

但是没有包含四个或更多后退边的无环路径。这里的最“深”的路径恰巧只包含了后退路径，这只是一个巧合。一般来说，在一个最深路径中可以包含后退边、前进边和交叉边。□

9.6.6 自然循环

在一个源程序中，循环可以有很多种描述方法：它们可以被写成 for 循环、while 循环或 repeat 循环；它们甚至还可以用标号和 goto 语句来定义。从程序分析的角度来看，循环在源代码中以什么形式出现并不重要，重要的是它们是否具有易于被优化的性质。我们特别关心的是一个循环是否只有一个唯一的入口结点。如果是这样，编译器的分析可以假设某些初始条件在循环的每次迭代的开头成立。这种优化机会引发了定义“自然循环”的需求。

自然循环(natural loop)通过两个重要的性质来定义。

- 1) 它必须具有一个唯一的入口结点，称为循环头(header)。这个入口结点支配了循环中的所有结点，否则它就不会成为循环的唯一入口。
- 2) 必然存在一条进入循环头的回边，否则控制流就不可能从“循环”中直接回到循环头，也就是说实际上并没有循环。

给定一个回边 $n \rightarrow d$ ，我们定义该边的自然循环(natural loop of the edge)是 d 加上那些不经过 d 就能够到达 n 的结点的集合。结点 d 是这个循环的循环头。

算法 9.46 构造一条回边的自然循环。

输入：一个流图 G 和一条回边 $n \rightarrow d$ 。

输出：由回边 $n \rightarrow d$ 的自然循环中的所有结点组成的集合 $loop$ 。

方法：令 $loop$ 等于 $\{n, d\}$ 。把 d 标记为“visited”，以便搜索过程不至于越过结点 d 。从结点 n 开始对输入的反向控制流图进行深度优先的搜索。把所有访问到的结点都加入 $loop$ 。这个过程可以找到所有不经过 d 就可以到达 n 的结点。□

例 9.47 在图 9-38 中有五条回边，这些边的头结点支配了它们的尾结点。它们是： $10 \rightarrow 7$, $7 \rightarrow 4$, $4 \rightarrow 3$, $8 \rightarrow 3$ 和 $9 \rightarrow 1$ 。请注意，这些边恰好就是所有的被认为在流图中形成循环的边。

回边 $10 \rightarrow 7$ 有自然循环 $\{7, 8, 10\}$ ，因为 8 和 10 是不经过 7 就能到达 10 的结点。回边 $7 \rightarrow 4$ 的自然循环由 $\{4, 5, 6, 7, 8, 10\}$ 组成，因此包含了回边 $10 \rightarrow 7$ 的循环。因此，我们假设后者是包含在前者中的一个内部循环。

回边 $4 \rightarrow 3$ 和 $8 \rightarrow 3$ 的自然循环具有同样的头，即结点 3；它们恰巧具有同样的结点集合： $\{3, 4, 5, 6, 7, 8, 10\}$ 。因此，我们将把这两个循环合并成为一个。这个循环包含了前面找到的两个较小的循环。

最后，回边 $9 \rightarrow 1$ 的自然循环是整个流图，因此是最外层的循环。在这个例子中，四个循环是逐层嵌套的。然而，通常会有两个互不包含的循环。□

因为一个可归约的流图中的所有后退边都是回边，我们可以把每条后退边和一个自然循环关联起来。这个结论对于不可归约流图不成立。比如，图 9-45 中的不可归约流图中有一个由结点 2 和 3 组成的环。环中的边都不是回边，因此这个环不满足自然循环的定义。我们并不把这个环当作自然循环，因此也不会优化它。这种情况是可接受的，因为假设所有循环都有唯一的人口点可以使循环分析变得更加简单。而且不管怎么说，不可归约的程序在实践中很少见到。

如果我们只把自然循环当作“循环”，那么可以得到下面的有用性质，即除非两个循环具有同样的循环头，否则它们要么是分离的，要么一个嵌套在另一个中。这样我们就很自然地得到了最内层循环 (innermost loop) 的定义，即不包含其他循环的循环。

当两个自然循环像图 9-46 中那样具有相同的循环头时，很难说谁是内层的循环。因此，如果两个自然循环具有相同的循环头且没有哪一个循环真正包含在另一个循环中，它们将被合并在一起，当作一个循环处理。

例 9.48 在图 9-46 中的回边 $3 \rightarrow 1$ 和 $4 \rightarrow 1$ 的自然循环分别是 $\{1, 2, 3\}$ 和 $\{1, 2, 4\}$ 。我们将把它们合并成一个循环 $\{1, 2, 3, 4\}$ 。

然而，假如图 9-46 中有另一个回边 $2 \rightarrow 1$ ，它的循环是 $\{1, 2\}$ 。这个循环将是第三个以 1 为循环头的自然循环。这个循环真包含于循环 $\{1, 2, 3, 4\}$ ，因此它不会和其他两个自然循环合并，而是作为包含在 $\{1, 2, 3, 4\}$ 中的内层循环进行处理。□

9.6.7 迭代数据流算法的收敛速度

我们现在可以讨论迭代算法的收敛速度了。如 9.3.3 节中所讨论的，算法的最大迭代次数可能是格的高度和流图结点数的乘积。对于很多数据流分析而言，我们可以对求值过程进行适当排序，使算法经过很少的迭代就能收敛。我们感兴趣的性质是是否所有影响一个结点的重要事件都可以通过一个无环的路径到达该点。在至今已经讨论过的数据流分析问题中，到达定值、可用表达式和活跃变量问题具有这个性质，而常量传递则不具有这个性质。更加明确地说：

- 如果一个定值 d 在 $\text{IN}[B]$ 中，那么必然有一条从包含 d 的基本块到达 B 的无环路径使得 d 在该路径上的所有 IN 和 OUT 值中。
- 如果表达式 $x + y$ 在基本块 B 的入口处不可用，那么必然有一条具有下列性质的无环路径：要么该路径从程序的入口结点出发并且不包含任何杀死或产生 $x + y$ 的语句；要么该路径从一个杀死了 $x + y$ 的基本块出发，并且从此之后该路径中没有产生表达式 $x + y$ 。
- 如果 x 在基本块 B 的出口处活跃，那么必然有一个从 B 开始到达对 x 的某次使用的无环路径，在此路径上没有对 x 的定值。

我们可以检验出在上述各个情况中，带有环的路径不会增加任何内容。比如，如果可以通过一个带环的路径从基本块 B 的结尾到达 x 的使用点，那么我们可以消除这个环，得到一个更短的路径。沿着这个较短路径依然可以从 B 到达 x 的这个使用点。

反过来，常量传播就没有这个性质。考虑如下一个简单程序，它仅包含一个由单个基本块组

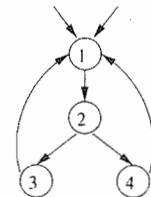


图 9-46 具有相同循环头的两个循环

成的循环，基本块中的代码为

```
L:    a = b
      b = c
      c = 1
      goto L
```

当第一次访问这个基本块时，我们发现 c 具有常量值 1，但是 a 和 b 没有定值。第二次访问该基本块时，我们发现 b 和 c 都有常量值 1。经过对该基本块的三次访问之后，赋给 c 的常量值 1 才到达 a 。

如果所有有用的信息都通过无环路径传播，我们就有可能调整迭代数据流算法中访问结点的顺序，以便经过几轮结点访问就可以保证这些信息已经沿着所有的无环路径传递完毕。

回顾一下 9.6.3 节中说过，如果 $a \rightarrow b$ 是一条边，那么只有当该边是后退边的时候 b 的深度优先编号才会小于 a 的编号。对于前向的数据流问题，按照深度优先顺序来访问结点是很合适的。明确地说，我们对图 9-23a 中的算法进行修改，把算法中访问流图中各个基本块的第(4)行代码替换为：

```
for (按照深度优先顺序，对所有不同于 ENTRY 的各个基本块 B) {
```

例 9.49 假设一个定值 d 在如下路径上传播，

3→5→19→35→16→23→45→4→10→17

其中的整数表示该路径上的各个基本块的深度优先编号。那么，图 9-23a 中算法的第(4)到(6)行的循环第一次运行时， d 将从 $\text{OUT}[3]$ 传播到 $\text{IN}[5]$ 再传播到 $\text{OUT}[5]$ ，…，最后到达 $\text{OUT}[35]$ 。因为 16 排在 35 之前， d 不会在这一轮中到达 $\text{IN}[16]$ 。在 d 被放进 $\text{OUT}[35]$ 的时候，我们已经计算了 $\text{IN}[16]$ 。但是下次我们运行第(4)到(6)行的循环时，因为此时 d 已经在 $\text{OUT}[35]$ 中，它将在计算 $\text{IN}[16]$ 的时候被加入进去。定值 d 同时会被传播到 $\text{OUT}[16]$, $\text{IN}[23]$, …, 最后到达 $\text{OUT}[45]$ 。它必须在这里等待下一轮计算，因为这一轮中已经计算过 $\text{IN}[4]$ 了。在第三轮中， d 将传播到 $\text{IN}[4]$, $\text{OUT}[4]$, $\text{IN}[10]$ 和 $\text{IN}[17]$ 。因此在三轮之后我们使得定值 d 到达了基本块 17。□

从这个例子中不难抽取出一般规律。如果我们在图 9-23a 中使用深度优先排序，那么把任何到达定值沿着一条无环路径传播所需要的迭代轮次不会大于路径中从高编号基本块到低编号基本块的边的个数加一。这些边恰好就是后退边，因此，所需轮次就是流图的深度加一。当然，算法 9.11 还需要再做一次不改变任何值的迭代，才能检测出所有定值都已经被传播到了所有它能够到达的地方。因此，使用了深度优先基本块排序的这个算法所执行的迭代轮次的上限实际上是深度加二。一项研究[⊖]表明，常见流图的平均深度大约是 2.75。因此这个算法的收敛速度很快。

产生不可归约数据流图的一个原因

在一种情况下我们通常不能指望一个流图是可归约的。如果像我们在算法 9.46 中寻找自然循环所做的那样，把一个程序的流图的边反向，那么我们不大可能得到一个可归约流图。直观的理由是，虽然典型程序的循环只有一个入口，但这些循环有时会有几个出口。当我们把流图的边反向时，这些出口就变成了入口。

[⊖] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience* 1: 2(1971), pp. 105-133.

在类似于活跃变量这样的逆向数据流问题中，我们以深度优先排序的逆序来访问结点。这样，我们可以沿着路径

$$3 \rightarrow 5 \rightarrow 19 \rightarrow 35 \rightarrow 16 \rightarrow 23 \rightarrow 45 \rightarrow 4 \rightarrow 10 \rightarrow 17$$

经过一个轮次把基本块 17 中对某个变量的一次使用逆向传播到 IN[4]。然后在这里等待下一次迭代，以便把它传播到 OUT[45]。在第二轮迭代中，它到达 IN[16]，在第三轮中它从 OUT[35] 到达 OUT[3]。

总的来说，深度加一次迭代足以把一个变量的使用沿着任何无环路径逆向传递完毕。但是，我们必须在每次迭代中按照深度优先排序的逆序来访问各个结点，因为这样才能在一次迭代中把变量的使用沿着任意长的下降结点序列传递。

在一些数据流分析问题中，环形路径不会给分析增加任何信息。至今为止讨论的界限是所有此类问题的上界。在一些特殊的问题中，比如对于支配结点问题，迭代算法的收敛速度更快。在输入流图是可归约的情况下，如果以深度优先顺序访问各个结点，那么数据流算法的第一轮迭代就可以得到各个结点的支配结点集合。如果我们之前不知道输入流图是可归约的，那么我们需要一次额外的迭代来确定算法已经收敛了。

9.6.8 9.6 节的练习

练习 9.6.1: 对于图 9-10 中的流图(见 9.1 节的练习)：

- 1) 计算支配关系。
- 2) 寻找每个结点的直接支配结点。
- 3) 构造支配结点树。
- 4) 找出该流图的一个深度优先排序。
- 5) 根据问题 4 的答案，指明其中的前进、后退和交叉边以及树的边。
- 6) 这个流图是可归约的吗？
- 7) 计算这个流图的深度。
- 8) 找出这个流图的自然循环。

练习 9.6.2: 对于下列流图重复练习 9.6.1。

- 1) 图 9-3。
- 2) 图 8.9。
- 3) 从练习 8.4.1 得到的流图。
- 4) 从练习 8.4.2 得到的流图。

! 练习 9.6.3: 证明下列有关 dom 关系的性质。

- 1) 如果 $a dom b$ 且 $b dom c$ ，那么 $a dom c$ (传递性)。
- 2) 如果 $a \neq b$ ，那么 $a dom b$ 和 $b dom a$ 不可能同时成立(反对称性)。
- 3) 如果 a 和 b 是 n 的两个支配结点，那么 $a dom b$ 和 $b dom a$ 之一必然成立。

4) 除了人口结点，每个结点 n 都有一个唯一的直接支配结点——在任何从人口结点到达 n 的无环路径中，这个支配结点是离 n 最近的支配结点。

! 练习 9.6.4: 图 9-42 是图 9-38 中流图的一个深度优先表示。这个流图有多少个其他的深度优先表示？请记住，不同的子结点顺序表示不同的深度优先表示。

!! 练习 9.6.5: 证明一个流图是可归约的当且仅当我们删除所有回边(即头结点支配尾结点的边)后得到的流图是无环的。

! 练习 9.6.6: 一个具有 n 个结点的完全流图在任意两个结点 i 和 j 之间(在两个方向上)都有边 $i \rightarrow j$ 。 n 取什么值的时候这个完全流图是可归约的？

! 练习 9.6.7: 一个在 n 个结点 $1, 2, \dots, n$ 上的无环完全流图对于所有的结点 i 和 j ($i < j$) 都有边 $i \rightarrow j$ 。其中结点 1 是人口结点。

1) n 取什么值的时候这个图是可归约的?

2) 如果给所有的结点 i 都加上自循环边 $i \rightarrow i$, 是否会改变对问题 a 的答案?

! 练习 9.6.8: 一个回边 $n \rightarrow h$ 的自然循环被定义为 h 加上所有能够不经过 h 而直接到达 n 的结点的集合。说明 h 支配 $n \rightarrow h$ 的自然循环中的所有结点。

!! 练习 9.6.9: 我们说过, 图 9-45 的流图是不可归约的。如果图中的那些边被替换为不同的路径(当然结束点除外), 且各条路径的结点集合两两不相交, 那么得到的流图还是不可归约的。实际上, 结点 1 不一定要是入口结点, 它可以是任何能够从入口结点沿着某条路径到达的结点, 只要该条路径的所有中间结点都不是上面明确给出的四条路径中的一部分。证明上面的论述反过来也成立: 每个不可归约流图都有一个如下的子图。这个子图和图 9-45 中的流图类似, 只是该流图的边可以被替换为结点互不相交的路径, 而结点 1 可以是任意能够从入口结点经过某条不和其他四条路径相交的路径到达的结点。

!! 练习 9.6.10: 说明每个不可归约流图的每个深度优先表示都有一条不是回边的后退边。

!! 练习 9.6.11: 说明如果条件

$$f(a) \wedge g(a) \wedge a \leq f(g(a))$$

对于所有的函数 f, g 和值 a 成立, 那么通用迭代算法, 即算法 9.25, 在按照深度优先排序执行每次迭代时, 经过深度加二次迭代之后必然收敛。

! 练习 9.6.12: 找到一个具有两棵不同深度的 DFST 的不可归约流图。

! 练习 9.6.13: 证明下列结论:

1) 如果一个定值 d 在 $\text{IN}[B]$ 中, 那么存在某条从包含 d 的基本块到达 B 的无环路径, 使得 d 在该路径上的所有 IN 和 OUT 值中。

2) 如果一个表达式 $x + y$ 在基本块 B 的入口处不可用, 那么必然存在某条到达 B 的无环路径满足下面的条件: 要么该路径从程序入口结点开始并且不包含任何杀死或生成 $x + y$ 的语句; 要么该路径从一个杀死了 $x + y$ 的基本块开始, 并且路径中不包含任何生成 $x + y$ 的语句。

3) 如果 x 在基本块 B 的出口处活跃, 那么必然有一条从 B 到 x 的某个使用点的路径, 在该路径上没有对 x 的定值。

9.7 基于区域的分析

至今为止我们讨论的迭代数据流分析算法只是解决数据流问题的方法之一。接下来我们讨论另一种被称为基于区域的分析 (region-based analysis) 的方法。回顾一下, 在迭代分析方法中, 我们为各个基本块创建传递函数, 然后通过在基本块上进行反复扫描来寻找不动点解。一个基于区域的分析技术并不仅仅为各个基本块创建传递函数, 它为越来越大的程序区域构造用以描述该区域运行情况的传递函数。最终构造出整个过程的传递函数, 并用这个传递函数直接得到想要的数据流值。

一个使用迭代算法的数据流框架通过一个数据流值的半格和一组对函数组合运算封闭的传递函数来描述, 而基于区域的分析还需要更多的元素。一个基于区域的框架包括一个数据流值的半格和一个传递函数的半格。后一种半格必须包括一个交汇运算、一个组合运算符和一个闭包运算符。我们将在 9.7.4 节看到所有这些元素的作用。

基于区域的分析技术特别适用于那些包含环的路径可能改变数据流值的数据流问题。它的闭包运算符允许我们概括描述一个循环的运行效果, 这种方法比迭代分析中的方法更加有效。

这个技术对过程间分析也是有用的。在进行过程间分析时，和一次过程调用关联的传递函数可以和那些与基本块相关联的传递函数一样处理。

为简单起见，我们在这一节中将只考虑前向数据流问题。我们将首先通过大家熟知的到达定值的例子来说明基于区域的分析技术的工作原理。在 9.8 节中，当我们研究归纳变量的时候，我们将给出一个有关这个技术的更有说服力的例子。

9.7.1 区域

在基于区域的分析中，程序被看作是一个区域（region）的层次结构。区域（粗略地讲）就是一个流图中只具有单个人口结点的部分。我们会发现，把代码看作区域层次结构的概念是很直观的，因为一个基于块结构的过程很自然地被组织成一个区域层次结构。在一个块结构的程序中，每个语句就是一个区域，因为控制流只能从一个语句的开头进入。每个语句嵌套层次对应于区域层次结构中的一层。

正式地讲，流图的一个区域是满足如下条件的一个结点集 N 和边集 E ：

- 1) 在 N 中有一个支配 N 中所有结点的头结点 h 。
- 2) 如果某个结点 m 能够不经过 h 到达 N 中的 n ，那么 m 也在 N 中。
- 3) E 是所有位于 N 中的任意两个结点 n_1 和 n_2 之间的控制流边的集合。有些进入 h 的边（可能）不在其中。

例 9.50 显然，一个自然循环就是一个区域，但是一个区域不一定包含一条回边，也不需要包含任何环。比如，图 9-47 中的结点 B_1 和 B_2 以及边 $B_1 \rightarrow B_2$ 形成了一个区域；结点 B_1 、 B_2 、 B_3 以及边 $B_1 \rightarrow B_2$ 、 $B_2 \rightarrow B_3$ 和 $B_1 \rightarrow B_3$ 也形成一个区域。

但是由结点 B_2 、 B_3 以及边 $B_2 \rightarrow B_3$ 组成的子图不是一个区域，因为控制流可能从结点 B_2 或 B_3 进入这个子图。更准确地说， B_2 和 B_3 都不支配另一个结点，因此违反了区域定义中的条件(1)。即使我们“选择”某个结点（比如 B_2 ）作为“头结点”，我们还是会违反条件(2)，因为我们可以不经过 B_2 就从 B_1 到达 B_3 ，而 B_1 不在这个“区域”中。□

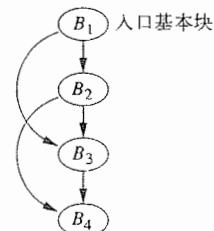


图 9-47 区域的例子

9.7.2 可归约流图的区域层次结构

在接下来的内容中，我们将假设流图是可归约的。如果我们偶尔必须处理不可归约流图的话，我们可以使用一种被称为“结点分割”的技术。该技术将在 9.7.6 节中讨论。

为了构造区域的层次结构，我们需要找出自然循环。回顾一下 9.6.6 节，在一个可归约流图中，任何两个自然循环要么不相交，要么一个循环嵌套在另一个循环里。在对一个流图进行“分析”并得到它的区域层次结构的过程一开始的时候把每个基本块本身看作一个区域。我们把这些区域称为叶子区域（leaf region）。然后，我们把自然循环从内到外（即从最内层的循环开始）排序。处理一个循环时，我们用两个步骤把整个循环替换为一个结点：

- 1) 首先，循环 L 的循环体（所有的结点以及除了到达循环头的回边之外的所有边）被替换为一个结点，该结点代表一个区域 R 。原先到达 L 的循环头的边现在进入代表 R 的结点。从 L 的任意出口结点出发的边被替换为从代表 R 的结点到达同一个目标结点的边。但是，如果该边是一个回边，那么它变成了 R 上的一个圈。我们把 R 称为循环体区域（body region）。
- 2) 然后，我们构造一个代表整个自然循环 L 的区域 R' ， R' 称为一个循环区域（loop region）。 R 和 R' 之间的唯一区别在于后者包含了到达 L 的循环头的回边。换句话说，在流图中用 R' 替换 R 时，我们要做的是删除从 R 到其自身的边。

我们按照这个方法不断进行处理，把越来越大的循环替换成为单个结点。在处理时先把循环替换成为带有圈的结点，再替换成为不带圈的结点。因为可归约流图中的循环要么相互嵌套，要么相互不相交，所以在按照这个归约过程构造得到的一系列流图中，循环区域结点可以表示对应的自然循环的所有结点。

各个自然循环最终都会归约成为单一的结点。此时，要么这个流图被归约为单个结点；要么还有多个结点但是不包含循环，即归约得到的流图是一个包含多个结点的无环图。在前一种情况下，我们已经完成了对区域层次结构的构造；而在后一种情况下，我们需要为整个流图再构造一个循环体区域。

例 9.51 考虑图 9-48a 中的控制流图。在这个流图中有一条从 B_4 到 B_2 的回边。相应的区域层次结构在图 9-48b 中显示，图中显示的边是区域流图的边。图中总共有 8 个区域：

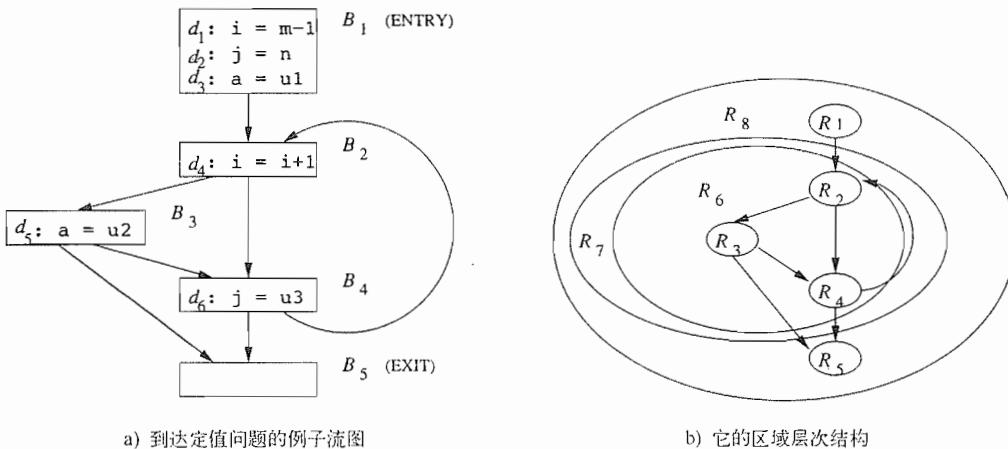


图 9-48 例 9.51 的控制流图

1) 区域 R_1, \dots, R_5 是叶子区域，分别代表了基本块 B_1 到 B_5 。每个基本块也是它的区域的出口基本块。

2) 循环体区域 R_6 表示流图中唯一循环的循环体。它由区域 R_2, R_3 和 R_4 组成，并包括了三个区域之间的边： $B_2 \rightarrow B_3$ 、 $B_2 \rightarrow B_4$ 和 $B_3 \rightarrow B_4$ 。这个区域有两个基本块： B_3 和 B_4 ，因为它们有不包含在区域内的、离开它们的边。图 9-49a 显示了把 R_6 归约为一个结点之后得到的流图。请注意，虽然边 $R_3 \rightarrow R_5$ 和 $R_4 \rightarrow R_5$ 都被替换为边 $R_6 \rightarrow R_5$ ，记住 $R_6 \rightarrow R_5$ 实际上代表了两条原来的边是很重要的。因为我们最终要沿着这条边传播传递函数，所以需要记住从 B_3 或 B_4 出发的传递函数将到达 R_5 的头结点。

3) 循环区域 R_7 代表整个自然循环。它包含了一个子区域 R_6 和一条回边 $B_4 \rightarrow B_2$ 。它也有两个出口结点，仍然是 B_3 和 B_4 。图 9-49b 中显示了把整个自然循环归约为 R_7 之后得到的流图。

4) 最后，区域 R_8 是顶层区域。它包含三个区域： R_1, R_7 和 R_5 ，以及三条区域之间的边 $B_1 \rightarrow B_2, B_3 \rightarrow B_5$ 和 $B_4 \rightarrow B_5$ 。当我们把流图归约为 R_8 的时候，它就变成单个结点。因为没有回边到达它的头结点 B_1 ，因此不需要再执行一个最后的步骤，把这个区域 R_8 归约成为一个循环区域。□

我们用下面的算法来概括按照层次结构分解一个可归约流图的过程。

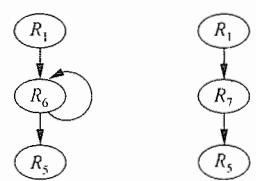


图 9-49 把图 9-48 的流图

归约为单个区域的步骤

算法 9.52 构造一个可归约流图的自底向上的区域序列。

输入：一个可归约流图 G 。

输出： G 的区域的列表，该列表可用于基于区域的数据流问题。

方法：

1) 一开始，列表中以某种顺序包含了由 G 的各个基本块组成的所有叶子区域。

2) 不断选择满足如下条件的自然循环 L ：如果 L 中包含了其他的自然循环，那么这些被包含的循环对应的循环体区域和循环区域已经被加入到列表中。首先把由 L 的循环体（即循环 L 中除了到达 L 的循环头的各条回边之外的其余部分）组成的区域加入到列表中，然后再加入 L 的循环区域。

3) 如果整个流图本身不是一个自然循环，在列表的最后加入由整个流图组成的区域。 \square

为什么叫做“可归约的”

现在我们可以知道可归约流图得名的原因了。虽然我们不会证明下面的性质，但本书中用涉及流图的回边来定义的“可归约流图”和其他几个按照能否把一个流图机械地归约成一个结点的定义方式实际上是等价的。在 9.7.2 节中描述的把自然循环塌缩成为一个结点的过程是上述机械化归约过程的一种。可归约流图的另一个有趣的定义是可以按照下列方法被归约成为一个结点的所有流图：

T_1 ：删除从一个结点到达自身的边。

T_2 ：如果结点 n 有唯一的前驱 m ，并且 n 不是流图的入口结点，那么把 m 和 n 合并。

9.7.3 基于区域的分析技术概述

对于每个区域 R 以及每个 R 中的子区域 R' ，我们计算一个传递函数 $f_{R, \text{IN}[R']}$ 来概括在 R 内部的从 R 的入口到 R' 的入口的全部可能路径的执行效果。如果存在一条从区域 R 中的基本块 B 到达 R 之外的基本块的边，我们就说 B 是 R 的一个出口基本块（exit block）。我们也为 R 中的每个出口基本块 B 计算一个传递函数，记为 $f_{R, \text{OUT}[B]}$ 。这个传递函数概括了所有在 R 中从 R 的入口基本块到达 B 的出口处的所有可能路径的执行效果。

然后我们沿着这个区域层次结构逐步向上，为越来越大的区域计算传递函数。我们从由单个基本块组成的区域开始。对于任何一个这样的区域 B ， $f_{B, \text{IN}[B]}$ 就是一个单元函数，而 $f_{B, \text{OUT}[B]}$ 则是基本块 B 自身的传递函数。当我们沿着层次结构向上时，

- 如果 R 是一个体区域，那么属于 R 的边构成了 R 的子区域的一个无环图。我们可以按照子区域的拓扑排序计算传递函数。
- 如果 R 是一个循环区域，那么我们只需要考虑到达 R 的头结点的回边的效果。

最终我们必然会到达层次结构的顶部，并计算得到对应于整个流图的区域 R_n 的传递函数。在算法 9.53 中描述了计算过程。

下一步是计算各个基本块的入口和出口处的数据流值。我们按照相反的方向，从区域 R_n 开始沿着层次结构向下处理各个区域。对于每个区域，我们计算其入口处的数据流值。对于区域 R_n ，我们应用 $f_{R_n, \text{IN}[R]}(\text{IN}[\text{ENTRY}])$ 来计算 R_n 的子区域 R 的入口处的数据流值。我们重复这个过程，直到到达位于区域层次结构中的叶子上的基本块为止。

9.7.4 有关传递函数的必要假设

为了使得基于区域的分析能够解决问题，我们要对框架中的传递函数集合的性质作出某些

假设。明确地说，我们需要作用于传递函数之上的三个基本原子运算：组合、交汇运算和闭包运算。使用迭代算法的数据流框架只需要其中的第一个运算。

组合

一个结点序列的传递函数可以把表示各个结点的传递函数组合起来得到。令 f_1 和 f_2 是结点 n_1 和 n_2 的传递函数。执行 n_1 再执行 n_2 的效果可以用函数 $f_2 \circ f_1$ 来表示。函数组合已经在 9.2.2 节中讨论过，并且在 9.2.4 节中用到达定值给出了一个例子。为了回顾一下，令 gen_i 和 $kill_i$ 是 f_i 的 gen 和 $kill$ 集合。那么

$$\begin{aligned}f_2 \circ f_1(x) &= gen_2 \cup ((gen_1 \cup (x - kill_1)) - kill_2) \\&= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))\end{aligned}$$

因此， $f_2 \circ f_1$ 的 gen 和 $kill$ 集合分别是 $(gen_2 \cup (gen_1 - kill_2))$ 和 $(kill_1 \cup kill_2)$ 。对于所有具有 $gen-kill$ 形式的传递函数，这个想法都是可行的。其他形式的传递函数可能也对组合运算封闭，但是我们必须单独考虑各种情况。

交汇运算

这里，传递函数集合本身就是一个具有交汇运算 \wedge_f 的半格的值域。两个传递函数 f_1 和 f_2 的交，即 $f_1 \wedge_f f_2$ ，被定义为 $(f_1 \wedge_f f_2)(x) = f_1(x) \wedge f_2(x)$ ，其中 \wedge 是数据流值的交汇运算。传递函数上的交汇运算用来把具有相同结尾点的不同执行路径的执行效果组合起来。从现在开始，在不会引起歧义时我们把传递函数的交汇运算也写成 \wedge 。对于到达定值框架，我们有

$$\begin{aligned}(f_1 \wedge f_2)(x) &= f_1(x) \wedge f_2(x) \\&= (gen_1 \cup (x - kill_1)) \cup (gen_2 \cup (x - kill_2)) \\&= (gen_1 \cup gen_2) \cup (x - (kill_1 \cap kill_2))\end{aligned}$$

也就是说， $f_1 \wedge f_2$ 的 gen 和 $kill$ 集合分别是 $gen_1 \cup gen_2$ 和 $kill_1 \cap kill_2$ 。仍然和处理组合运算一样，对于任何 $gen-kill$ 形式的传递函数集合都可以进行同样的处理。

闭包

如果 f 表示一个环的传递函数，那么 f^n 表示沿着这个环执行 n 次的效果。当不知道迭代次数的时候，我们必须假设这个环将被执行 0 到多次。我们用 f^* ，即 f 的闭包来表示这样的循环的传递函数。闭包 f^* 的定义如下

$$f^* = \bigwedge_{n \geq 0} f^n$$

请注意， f^0 必须是单元传递函数，因为它代表把这个循环执行 0 次的效果，即从入口处开始但不运行的效果。如果令 I 表示单元传递函数，那么可以把上式写成

$$f^* = I \wedge (\bigwedge_{n > 0} f^n)$$

假设在一个到达定值框架中的传递函数 f 有一个 gen 集和一个 $kill$ 集。那么，

$$\begin{aligned}f^2(x) &= f(f(x)) \\&= gen \cup ((gen \cup (x - kill)) - kill) \\&= gen \cup (x - kill) \\f^3(x) &= f(f^2(x)) \\&= gen \cup (x - kill)\end{aligned}$$

以此类推，即所有的 f^n 都是 $gen \cup (x - kill)$ 。也就是说，如果传递函数具有 $gen-kill$ 形式，那么沿着一个循环执行的次数对该函数没有影响。因此，

$$\begin{aligned}f^*(x) &= I \wedge f^1(x) \wedge f^2(x) \wedge \dots \\&= x \cup (gen \cup (x - kill)) \\&= gen \cup x\end{aligned}$$

也就是说，传递函数 f^* 的 gen 和 $kill$ 集合分别是 gen 和 \emptyset 。直观地讲，因为我们可能根本不沿着这个循环执行， x 中的任何元素都可以到达这个循环的入口处。在此后的所有迭代中，到达定值中总是包括所有在 gen 集合中的元素。

9.7.5 一个基于区域的分析算法

下面的算法根据某个满足9.7.4节中假设的框架，解决了一个可归约流图上的前向数据流分析问题。回顾一下， $f_{R, \text{IN}[R']}$ 和 $f_{R, \text{OUT}[B]}$ 是两个传递函数，它们把区域 R 的入口点上的数据流值分别转换为在子区域 R' 入口处和出口基本块 B 的数据流值。

算法 9.53 基于区域的分析。

输入：一个具有9.7.4节中所列性质的数据流框架和一个可归约流图 G 。

输出： G 中的每个基本块 B 的数据流值 $\text{IN}[B]$ 。

方法：

1) 使用算法9.52来构造 G 的自底向上的区域序列，假设它们是 R_1, R_2, \dots, R_n ，其中 R_n 是最顶层的区域。

2) 进行自底向上的分析，计算概括了每个区域的执行效果的传递函数。对于按照自底向上顺序排列的每个区域 R_1, R_2, \dots, R_n ，进行下列计算：

④ 如果 R 是一个对应于基本块 B 的叶子区域，令 $f_{R, \text{IN}[B]} = I, f_{R, \text{OUT}[B]} = f_B$ 。其中， f_B 是基本块 B 的传递函数。

⑤ 如果 R 是一个循环体区域，执行图9-50a中的计算。

⑥ 如果 R 是一个循环区域，执行图9-50b中的计算。

3) 进行自顶向下的扫描，找出各个区域开始处的数据流值。

⑦ $\text{IN}[R_n] = \text{IN}[\text{ENTRY}]$ 。

⑧ 按照自顶向下的顺序，对 $\{R_1, R_2, \dots, R_{n-1}\}$ 中的每个区域 R 计算

$$\text{IN}[R] = f_{R', \text{IN}[R]}(\text{IN}[R'])$$

其中 R' 是直接包含区域 R 的区域。

```

1) for (按照拓扑排序，对于每个直接包含于 $R$ 
   的子区域 $S$ ) {
2)    $f_{R, \text{IN}[S]} = \bigwedge_S$  的头结点在 $R$ 中的前驱  $B f_{R, \text{OUT}[B]}$ ;
    /* 如果 $S$ 是区域 $R$ 的头，那么 $f_{R, \text{IN}[S]}$ 就是
       对空集应用交汇运算的结果，也就是单元函数 */
3)   for ( $S$ 中的每个出口基本块 $B$ )
4)      $f_{R, \text{OUT}[B]} = f_{S, \text{OUT}[B]} \circ f_{R, \text{IN}[S]}$ ;
}

```

a) 构造一个循环体区域 R 的传递函数

```

1) 令 $S$ 为直接包含于 $R$ 的循环体区域；就是说 $S$ 就是从 $R$ 中
   删除了到达 $R$ 的头结点的回边后得到的区域
2)  $f_{R, \text{IN}[S]} = (\bigwedge_S$  的头结点在 $R$ 中的前驱  $B f_{S, \text{OUT}[B]})^*$ ;
3) for ( $R$ 中的每个出口基本块 $B$ )
4)    $f_{R, \text{OUT}[B]} = f_{S, \text{OUT}[B]} \circ f_{R, \text{IN}[S]}$ ;
}

```

b) 为一个循环区域 R' 构造传递函数

图9-50 基于区域的数据流计算的细节

让我们首先看一下算法中自底向上分析过程的工作细节。在图 9-50a 的第(1)行中，我们按照某个拓扑排序访问一个循环体区域的各个子区域。第(2)行中计算的传递函数代表了所有从 R 的头结点到 S 的头结点的可能路径的执行效果；第(3)和(4)行中计算的传递函数代表了所有从 R 的头结点到 R 的出口点（即所有的某个后继在 S 之外的基本块的出口点）的可能路径的执行效果。请注意，按照第(1)行所构造的拓扑排序， R 的所有前驱 B' 必然在 S 之前的区域中。这样， $f_{R, \text{OUT}[B']}$ 一定已经在算法的外层循环的前面某次迭代中由第(4)行计算完毕。

对于循环区域，我们执行图 9-50b 中第(1)到(4)行的各个步骤。其中，第(2)行计算了沿着循环体区域 S 重复执行零次或多次的效果。第(3)和第(4)行计算了进行一次或多次迭代之后在循环出口处的效果。

在算法的自顶向下扫描中，步骤 3(a)首先把问题的边界条件作为最顶层区域的输入。然后，如果 R 直接包含于 R' ，我们只需要将传递函数 $f_{R', \text{IN}[R]}$ 应用到 $\text{IN}[R']$ 就可以计算得到 $\text{IN}[R]$ 。□

例 9.54 让我们应用算法 9.53 来寻找图 9-48a 中流图的到达定值。第一步构造出自底向上访问各个区域的顺序；这个顺序将作为各个区域的下标，比如 R_1, R_2, \dots, R_n 。

五个基本块的 gen 集和 $kill$ 集的值概括如下：

B	B_1	B_2	B_3	B_4	B_5
gen_B	$\{d_1, d_2, d_3\}$	$\{d_4\}$	$\{d_5\}$	$\{d_6\}$	\emptyset
$kill_B$	$\{d_4, d_5, d_6\}$	$\{d_1\}$	$\{d_3\}$	$\{d_2\}$	\emptyset

请回忆一下 9.7.4 节中对 $gen-kill$ 形式的传递函数的简化后的规则：

- 要计算传递函数的交汇，只要计算 gen 集合的并集和 $kill$ 集合的交集。
- 组合传递函数时，计算两个函数的 gen 集的并集和 $kill$ 集的并集。但是这个规则有一个例外，当一个表达式被第一个函数生成且没有被第二个函数生成，同时又被第二个函数杀死的时候，这个表达式不在最后的 gen 中。
- 在计算一个传递函数的闭包时，保持原来的 gen 集合，但是用 \emptyset 替代原来的 $kill$ 集合。

前面的五个区域 R_1, \dots, R_5 分别是基本块 B_1, \dots, B_5 。对于 $1 \leq i \leq 5$ ， $f_{R_i, \text{IN}[B_i]}$ 都是单元函数， $f_{R_i, \text{OUT}[B_i]}$ 是 B_i 的传递函数：

$$f_{B_i, \text{OUT}[B_i]}(x) = (x - kill_{B_i}) \cup gen_{B_i}$$

算法 9.53 的第二步构造的其他传递函数如图 9-51 中。区域 R_6 由区域 R_2, R_3 和 R_4 组成， R_6 代表该循环的循环体，因此不包含回边 $B_4 \rightarrow B_2$ 。对这些区域的处理顺序就是它们的唯一的拓扑排序： R_2, R_3, R_4 。首先请记住边 $B_4 \rightarrow B_2$ 到达了 R_6 之外， R_2 在 R_6 中没有前驱。因此 $f_{R_6, \text{IN}[B_2]}$ 是单元函数[⊕]，而 $f_{R_6, \text{OUT}[B_2]}$ 是 B_2 本身的传递函数。

区域 B_3 的头结点有一个 R_6 中的前驱，即 R_2 。到达它的入口处的传递函数就是到达 B_2 出口处的传递函数 $f_{R_6, \text{OUTN}[B_2]}$ 。这个函数已经被计算出来。我们把这个函数和 B_3 的传递函数组合起来，计算出到达 B_3 出口处的传递函数。 B_3 就在由它自身组成的区域中。

最后，因为 B_2 和 B_3 都是 R_4 的头结点 B_4 的前驱，对于到达 R_4 入口处的传递函数，我们必须计算

⊕ 严格地讲，我们说的是 $f_{R_6, \text{IN}[R_2]}$ ，但是对类似于 R_2 这样的单基本块区域，如果我们在这个上下文环境下使用基本块名字而不是区域名字的话，文字表述通常会更清楚。

		传递函数	gen	kill
R_6	$f_{R_6, \text{IN}[R_2]} = I$	\emptyset	\emptyset	
	$f_{R_6, \text{OUT}[B_2]} = f_{R_2, \text{OUT}[B_2]} \circ f_{R_6, \text{IN}[R_2]}$	$\{d_4\}$	$\{d_1\}$	
	$f_{R_6, \text{IN}[R_3]} = f_{R_6, \text{OUT}[B_2]}$	$\{d_4\}$	$\{d_1\}$	
	$f_{R_6, \text{OUT}[B_3]} = f_{R_3, \text{OUT}[B_3]} \circ f_{R_6, \text{IN}[R_3]}$	$\{d_4, d_5\}$	$\{d_1, d_3\}$	
	$f_{R_6, \text{IN}[R_4]} = f_{R_6, \text{OUT}[B_2]} \wedge f_{R_6, \text{OUT}[B_3]}$	$\{d_4, d_5\}$	$\{d_1\}$	
	$f_{R_6, \text{OUT}[B_4]} = f_{R_4, \text{OUT}[B_4]} \circ f_{R_6, \text{IN}[R_4]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$	
R_7	$f_{R_7, \text{IN}[R_6]} = f_{R_6, \text{OUT}[B_4]}$	$\{d_4, d_5, d_6\}$	\emptyset	
	$f_{R_7, \text{OUT}[B_3]} = f_{R_6, \text{OUT}[B_3]} \circ f_{R_7, \text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_3\}$	
	$f_{R_7, \text{OUT}[B_4]} = f_{R_6, \text{OUT}[B_4]} \circ f_{R_7, \text{IN}[R_6]}$	$\{d_4, d_5, d_6\}$	$\{d_1, d_2\}$	
R_8	$f_{R_8, \text{IN}[R_1]} = I$	\emptyset	\emptyset	
	$f_{R_8, \text{OUT}[B_1]} = f_{R_1, \text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	
	$f_{R_8, \text{IN}[R_7]} = f_{R_8, \text{OUT}[B_1]}$	$\{d_1, d_2, d_3\}$	$\{d_4, d_5, d_6\}$	
	$f_{R_8, \text{OUT}[B_3]} = f_{R_7, \text{OUT}[B_3]} \circ f_{R_8, \text{IN}[R_7]}$	$\{d_2, d_4, d_5, d_6\}$	$\{d_1, d_3\}$	
	$f_{R_8, \text{OUT}[B_4]} = f_{R_7, \text{OUT}[B_4]} \circ f_{R_8, \text{IN}[R_7]}$	$\{d_3, d_4, d_5, d_6\}$	$\{d_1, d_2\}$	
	$f_{R_8, \text{IN}[R_5]} = f_{R_8, \text{OUT}[B_3]} \wedge f_{R_8, \text{OUT}[B_4]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$	
	$f_{R_8, \text{OUT}[B_5]} = f_{R_5, \text{OUT}[B_5]} \circ f_{R_8, \text{IN}[R_5]}$	$\{d_2, d_3, d_4, d_5, d_6\}$	$\{d_1\}$	

图 9-51 使用基于区域的流分析计算图 9-48a 中流图的传递函数

$$f_{R_6, \text{OUT}[B_2]} \wedge f_{R_6, \text{OUT}[B_3]}$$

这个传递函数和传递函数 $f_{R_4, \text{OUT}[B_4]}$ 组合，得到我们想要的函数 $f_{R_6, \text{OUT}[B_4]}$ 。请注意，比如， d_3 没有在这个函数中被杀死，因为路径 $B_2 \rightarrow B_4$ 没有对变量 a 重新定值。

现在考虑循环区域 R_7 。它只包含一个表示它的循环体的区域 R_6 。因为只有一个到达 R_6 的头结点的回边 $B_4 \rightarrow B_2$ ，代表这个循环体执行 0 次或者多次的传递函数就是 $f_{R_6, \text{OUT}[B_4]}^*$ ：这个函数的 gen 集合是 $\{d_4, d_5, d_6\}$ ，而 kill 集合是 \emptyset 。区域 R_7 有两个出口，即基本块 B_3 和 B_4 。因此，这个传递函数和 R_6 的各个传递函数相组合，得到对应于 R_7 的传递函数。请注意某些定值，比如 d_6 ，是怎样因为路径 $B_2 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$ ，甚至路径 $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2 \rightarrow B_3$ 的原因而被加入到函数 $f_{R_7, \text{OUT}[B_3]}$ 的 gen 集合中去的。

最后考虑区域 R_8 ，即整个流图。它的子区域是 R_1 、 R_7 和 R_5 。我们将按照这个拓扑顺序考虑这些子区域。和前面一样，传递函数 $f_{R_8, \text{IN}[B_1]}$ 是一个单元函数，而传递函数 $f_{R_8, \text{OUT}[B_1]}$ 是 $f_{R_1, \text{OUT}[B_1]}$ ，也就是 f_{B_1} 。

R_7 的头结点 B_2 只有一个前驱 B_1 ，因此到达它的入口的传递函数就是 R_8 中从 B_1 离开的传递函数。我们把 $f_{R_8, \text{OUT}[B_1]}$ 和 R_7 中到达 B_3 和 B_4 出口处的传递函数相组合，得到它们在 R_8 中相应的传递函数。最后我们考虑 R_5 ，它的头结点 B_5 在 R_8 中有两个前驱，即 B_3 和 B_4 。因此，我们计算 $f_{R_8, \text{OUT}[B_3]} \wedge f_{R_8, \text{OUT}[B_4]}$ 可以得到 $f_{R_8, \text{IN}[B_5]}$ 。因为基本块 B_5 的传递函数是单元函数，因此 $f_{R_8, \text{OUT}[B_5]} = f_{R_8, \text{IN}[B_5]}$ 。

第三步根据传递函数计算实际的到达定值。在步骤 3(a)， $\text{IN}[R_8] = \emptyset$ ，因为在程序的开头没有到达定值。图 9-52 显示了步骤 3(b) 是如何计算其余的数据流值的。这个步骤从 R_8 的各个子区域开始。因为从 R_8 的开始处到它的各个子区域开始处的传递函数已经计算出来了，通过简单地应用这些传递函数就可以找到各个子区域的开始处的数据流值。我们重复这个步骤，直到得到各个叶子区域的数据流值为止。这些叶子区域就是各个基本块。请注意，图 9-52 中显示的数据流值和我们对相同流图应用迭代数据流分析技术而得到的值是完全一致的。当然，这两组值必须一致。□

$\text{IN}[R_8]$	$= \emptyset$
$\text{IN}[R_1]$	$= f_{R_8, \text{IN}[R_1]}(\text{IN}[R_8]) = \emptyset$
$\text{IN}[R_7]$	$= f_{R_8, \text{IN}[R_7]}(\text{IN}[R_8]) = \{d_1, d_2, d_3\}$
$\text{IN}[R_5]$	$= f_{R_8, \text{IN}[R_5]}(\text{IN}[R_8]) = \{d_2, d_3, d_4, d_5, d_6\}$
$\text{IN}[R_6]$	$= f_{R_8, \text{IN}[R_6]}(\text{IN}[R_8]) = \{d_1, d_2, d_3, d_4, d_5, d_6\}$
$\text{IN}[R_4]$	$= f_{R_8, \text{IN}[R_4]}(\text{IN}[R_8]) = \{d_2, d_3, d_4, d_5, d_6\}$
$\text{IN}[R_3]$	$= f_{R_8, \text{IN}[R_3]}(\text{IN}[R_8]) = \{d_2, d_3, d_4, d_5, d_6\}$
$\text{IN}[R_2]$	$= f_{R_8, \text{IN}[R_2]}(\text{IN}[R_8]) = \{d_1, d_2, d_3, d_4, d_5, d_6\}$

图 9-52 基于区域的流分析的最后一步

9.7.6 处理不可归约流图

如果预计到需要用编译器或其他程序处理软件进行处理的程序中经常会有不可归约流图，那么我们建议使用迭代算法，而不是基于层次结构的方法来解决数据流分析问题。但是，如果我们只准备偶尔处理一下不可归约流图，那么使用下面的“结点分割”技术就足够了。

首先尽可能依据自然循环构造区域。如果流图是不可归约的，我们会发现得到的流图包含环，但是没有回边，因此我们不能进一步对这个流图进行分析。在图 9-53a 中显示了一种典型的情形。这个流图和图 9-45 中的不可归约流图具有同样的结构，但是就像图 9-53 中的结点内的小结点所显示的，这个流图的结点可能实际上是一个复杂的区域。

我们选取一个具有多个前驱，且不是整个流图的头结点的区域 R 。如果 R 有 k 个前驱，那么建立 k 个对应于 R 的整个流图的拷贝，并将 R 的头结点的 k 个前驱分别连接到不同的拷贝。请记住，只有一个区域的头才可能具有区域之外的前驱。我们只是给出（而不准备证明）下面的结论：这样的结点分割的结果是，在寻找新的回边并构造出这些回边的区域之后，区域的个数至少减少了一。这样得到的流图可能还是不可归约的，但是我们可以不断交替进行两个步骤：结点分割步骤；寻找新自然循环并将其塌缩为单个结点的步骤。最终我们会得到单个区域，即流图已经被完全归约。

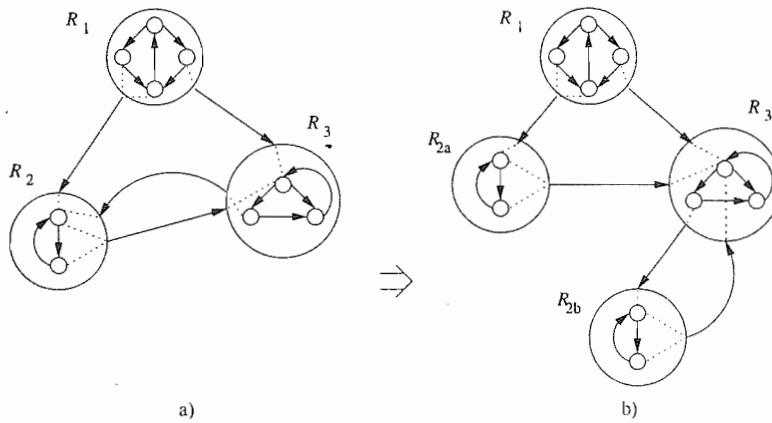


图 9-53 复制一个区域使得一个不可归约流图变成可归约的

例 9.55： 图 9-53b 中显示的结点分割把边 $R_{2b} \rightarrow R_3$ 变成了一个回边，因为现在 R_3 支配 R_{2b} 。因此这两个区域可以合二为一。得到的三个区域—— R_1 、 R_{2a} 及新产生的区域——组成了一个新的无环的流图，因此可能被组合到单个循环体区域中。这样我们就把整个流图归约为单个区域。一般来讲，可能还需要更多的分割处理，在最坏的情况下，最后得到的基本块的个数可能和原流图中基本块的个数成指数关系。□

我们想要的是针对原流图的答案。因此我们还必须考虑对分割所得流图的数据流分析结果和这个答案之间的关系。我们可以考虑两个方法：

1) 分割区域可能有益于优化过程，我们可以简单地修订这个流图使得只有某些基本块被复制。到达每个基本块副本的路径可能只是到达原基本块的路径的一个子集。因此相对于原基本块上的数据流值而言，在这些基本块副本上的数据流值可能包含更加精确的信息。比如，到达每个基本块副本的定值要少于到达原基本块的定值。

2) 如果我们希望不是真的分割而是想保留原来的流图，那么在分析完分割所得的流图之后，我们可以查看每个被分割基本块 B ，以及它对应的拷贝集合 B_1, B_2, \dots, B_k 。我们可以计算 $\text{IN}[B] = \text{IN}[B_1] \wedge \text{IN}[B_2] \wedge \dots \wedge \text{IN}[B_k]$ ，并且类似地计算 OUT 值。

9.7.7 9.7 节的练习

练习 9.7.1: 对于图 9-10 的流图(见 9.1 节中的练习)：

- 1) 寻找所有可能的区域。但是你可以忽略区域列表中那些只有一个结点且没有边的区域。
- 2) 给出算法 9.52 所构造的嵌套区域的集合。
- 3) 按照 9.7.2 节中“为什么叫做可归约的”部分中所描述的方法，给出该流图的一个 $T_1 - T_2$ 归约。

练习 9.7.2: 在下列流图上重复练习 9.7.1：

- 1) 图 9-3。
- 2) 图 8-9。
- 3) 你在练习 8.4.1 中得到的流图。
- 4) 你在练习 8.4.2 中得到的流图。

练习 9.7.3: 证明每个自然循环都是一个区域。

!! 练习 9.7.4: 说明一个流图是可归约的当且仅当它可以按照下列方式被转化成为单一结点：

- 1) 9.7.2 节的“为什么叫做可归约的”部分中描述的 T_1 和 T_2 运算。
- 2) 9.7.2 节中引入的区域定义。

练习 9.7.5: 说明如果你对一个不可归约流图应用结点分割技术，然后对分割后得到的流图进行 $T_1 - T_2$ 归约，你最后得到的流图的结点一定严格少于原流图的结点数目。

! 练习 9.7.6: 如果你交替地使用结点分割技术和 $T_1 - T_2$ 归约来归约一个具有 n 个结点的完全有向图，会发生什么情况？

9.8 符号分析

在本节中，我们将使用符号分析来说明基于区域的分析技术的使用。在这个分析中，我们用符号表示的方式跟踪程序中的变量的值，把这些变量的值表示为关于输入变量及其他变量的表达式。我们把这些变量称为参考变量。用同一组参考变量来表示变量的值可以描绘出这些变量之间的关系。符号分析可以被用于多种目的，比如优化、并行化和用于程序理解的分析。

例 9.56 考虑图 9-54 中的简单程序的例子。这里我们使用 x 作为唯一的参考变量。符号分析会发现在第 2 行和第 3 行中分别对 y 和 z 赋值的语句执行之后， y 和 z 的值分别是 $x - 1$ 和 $x - 2$ 。这个信息是很用的。比如，可以用来确定在第 4 行和第 5 行中的赋值语句将会在不

```

1) x = input();
2) y = x-1;
3) z = y-1;
4) A[x] = 10;
5) A[y] = 11;
6) if (z > x)
7)     z = x;

```

图 9-54 说明符号分析
动机的一个例子程序

同的内存位置上进行写运算，因而是可以并行执行的。并且，我们还可以指出条件 $z > x$ 永远不可能为真，从而允许优化程序把第 6 行和第 7 行的条件语句全部删除。□

9.8.1 参考变量的仿射表达式

因为我们不可能为所有计算得到的值创建一种简洁而又封闭的符号表达式，所以选择了一个抽象域，并且使用域中的最精确的表达式来近似表达计算结果。在此之前我们已经看到了这个策略的一个例子：常量传播。在常量传播中，我们的抽象域由所有常量值和特殊符号 UNDEF 及 NAC 组成。其中，UNDEF 表示我们尚未决定该值是否为常量，而当已经发现一个变量不是常量的时候使用 NAC。

我们在这里给出的符号化分析技术尽可能地把值表示成为参考变量的仿射表达式。如果一个关于变量 v_1, v_2, \dots, v_n 的表达式可以被表示为 $c_0 + c_1 v_1 + \dots + c_n v_n$ ，那么这个表达式就是仿射的，其中 c_0, c_1, \dots, c_n 都是常量。这样的表达式也被非正式地称为线性表达式。严格地讲，只有当 $c_0 = 0$ 的时候，仿射表达式才是线性的。我们对仿射表达式感兴趣的原因是循环中的数组下标经常可以表示成仿射表达式——这些信息可用于优化和并行化处理。在第 11 章中，我们将更详细地讨论这个主题。

归纳变量

一个仿射表达式不一定只能使用程序变量作为参考变量，它也可以使用一个循环的迭代次数作为参考变量。如果一个变量在某个程序点上的值能够被表示为 $c_1 i + c_0$ ，其中 i 是包含该程序点的最内层循环的迭代次数，那么这个变量称为归纳变量 (induction variable)。

例 9.57 考虑代码片断

```
for (m = 10; m < 20; m++)
{ x = m*3; A[x] = 0; }
```

假设我们为该循环引入一个变量 i 来表示已执行的迭代次数。在循环第一次迭代时， i 的值是 0，第二次迭代的时候 i 的值是 1，以此类推。我们可以把变量 m 表示成为 i 的一个仿射表达式，也就是 $m = i + 10$ 。变量 x ，也就是 $3m$ ，在循环的连续迭代中的取值是 30, 33, …, 57。因此 x 具有仿射表达式 $x = 30 + 3i$ 。我们说 m 和 x 都是这个循环的归纳变量。□

把变量表示成为循环次数的仿射表达式使得我们可以直接计算该变量在各次迭代中的值，而且能够实现多种代码转换。一个归纳变量在循环的各次迭代中所取的值可以通过加法运算，而不是乘法运算计算得到。这个转换称为“强度消减”。它已经在 8.7 节和 9.1 节中介绍过了。比如，我们可以从例 9.57 的循环中消除乘法运算 $x = m * 3$ ，只要把程序改写为：

```
x = 27;
for (m = 10; m < 20; m++)
{ x = x+3; A[x] = 0; }
```

另外，请注意在该循环中被赋予 0 值的内存位置，即 $\&A + 30, \&A + 33, \dots, \&A + 57$ ，也都是循环迭代次数的仿射表达式。实际上，这些整数值是该循环中唯一需要进行计算的值；我们只需要保留 m 或 x 中的一个。上面的代码可以直接替换为下面的代码：

```
for (x = &A+30; x <= &A+57; x = x+3)
*x = 0;
```

除了加快计算速度，符号化分析对于实现并行化也是有用的。当循环中的数组下标是循环迭代次数的仿射表达式时，我们可以考虑不同迭代中的数据访问的关系。比如，我们可以指出在每次迭代中被写入的内存位置是不同的，因此循环的全部迭代可以在不同的处理器上并行执行。这样的信息在第 10 章和第 11 章中被用来从顺序程序中抽取并行性。

其他参考变量

如果一个变量不是我们已选取的参考变量的线性函数，我们还可以选择把它的值当作将来

进行的运算的参考变量。比如，考虑下面的代码片段：

```
a = f();
b = a + 10;
c = a + 11;
```

虽然在上面的函数调用之后 a 的值本身不能被表示成任何参考变量的线性函数，但它仍可以被用作后继语句的参考变量。比如，使用 a 作为参考变量，我们就可以发现在程序的结尾处 c 比 b 大 1。

例 9.58 本节中多次使用的例子是基于图 9-55 中显示的源代码的。其中的内层循环和外层循环是很容易理解的，因为除了在 for 循环的头部， f 和 g 的值都没有被改变。因此有可能把 f 和 g 替换为分别对外层和内层循环的迭代次数进行计数的参考变量 i 和 j 。也就是说，我们可以令 $f = i + 99$ 和 $g = j + 9$ ，然后把 f 和 g 完全替换掉。在翻译成中间代码时，我们可以利用每个循环至少会迭代一次的信息，把对 $i \leq 100$ 和 $j \leq 10$ 的测试推迟到循环的尾部进行。在图 9-55 的代码中引入 i 和 j ，并把 for 循环当作 repeat 循环处理之后，就可以得到如图 9-56 中显示的流图。

```
1) a = 0;
2) for (f = 100; f < 200; f++) {
3)     a = a + 1;
4)     b = 10 * a;
5)     c = 0;
6)     for (g = 10; g < 20; g++) {
7)         d = b + c;
8)         c = c + 1;
}
}
```

图 9-55 例 9.58 的源代码

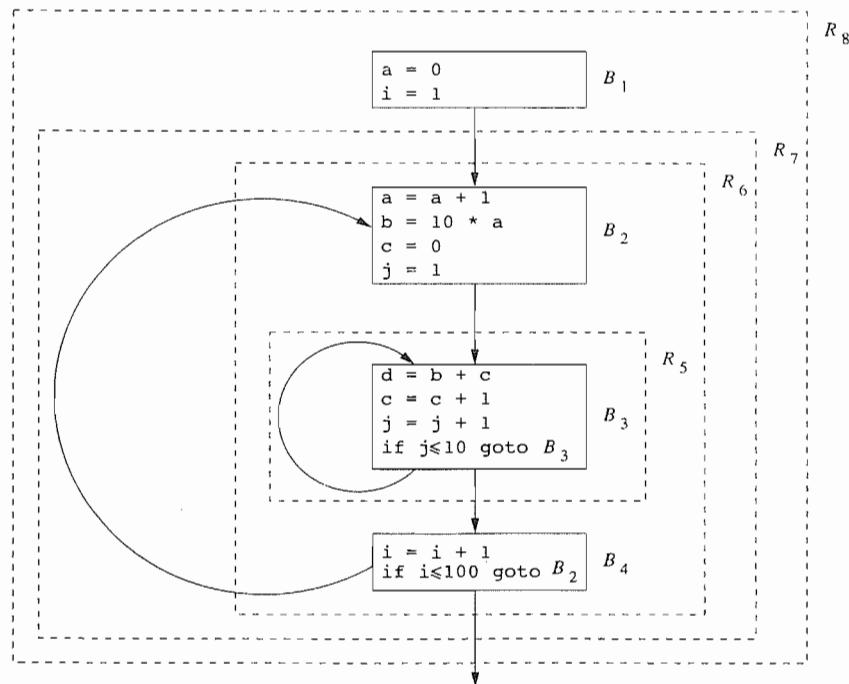


图 9-56 例 9.58 的流图和它的区域层次结构

可以发现， a 、 b 、 c 和 d 都是归纳变量。在代码的每一行上赋给这些变量的值的序列被显示在图 9-57 中。我们将看到，我们可以找出用参考变量 i 和 j 表示的这些变量的仿射表达式。它们是，在第 4 行的 $a = i$ ，第 7 行的 $d = 10i + j - 1$ 和第 8 行的 $c = j$ 。□

行	变量	$i = 1$ $j = 1, \dots, 10$	$i = 2$ $j = 1, \dots, 10$	$1 \leq i \leq 100$ $j = 1, \dots, 10$	$i = 100$ $j = 1, \dots, 10$
3	a	1	2	i	100
4	b	10	20	$10i$	1000
7	d	$10, \dots, 19$	$20, \dots, 29$	$10i, \dots, 10i + 9$	$1000, \dots, 1009$
8	c	$1, \dots, 10$	$1, \dots, 10$	$1, \dots, 10$	$1, \dots, 10$

图 9-57 例 9.58 中的各个程序点上看到的值的序列

9.8.2 数据流问题的公式化

这个分析寻找关于某些参考变量的仿射表达式。这些参考变量包括(1)用于对各个循环所执行的迭代进行计数的参考变量, (2)在必要时存放区域入口处的值的参考变量。这个分析也可以找到归纳变量、循环不变表达式以及常量。这里常量可以看作是仿射表达式的退化情况。请注意, 这个分析不能够找到所有的常量, 因为它只跟踪参考变量的表达式。

数据流值: 符号化映射

这个分析使用的数据流值的域是符号化映射, 它是将程序中的变量映射到值的函数。这个值可以是一个参考值的仿射函数或者表示非仿射表达式的特殊符号 NAA。如果只有一个变量, 那么相应半格的底元素值就是一个把该变量映射为 NAA 的映射。 n 个变量的半格就是各个变量的半格的积。我们使用 m_{NAA} 来表示这个半格的底元素, 它把所有变量都映射为 NAA。就像在常量传播中所做的那样, 我们可以把顶层数据流值定义为把所有变量都映射为一个未知值的符号化映射。但是, 在基于区域的分析中我们不需要顶元素的值。

例 9.59 图 9-58 显示了例 9.58 的代码中和各个基本块关联的符号化映射。我们将在稍后看到如何发现这些映射, 它们是在图 9-56 的流图上进行基于区域的数据流分析的结果。

和程序人口相关联的符号化映射是 m_{NAA} 。在 B_1 的出口处, a 的值被设置为 0。在 B_2 的入口处, 在第一次迭代时 a 的值是 0, 然后在每一次外层循环的迭代中都增加一。因此, 在进入第 i 次迭代时其值为 $i - 1$, 在迭代结束时为 i 。因为变量 b 、 c 、 d 在外层循环入口处的值未知, 所以在 B_2 入口处的符号化映射把 b 、 c 、 d 映射到 NAA。到现在为止, 它们的值依赖于外层循环的迭代次数。在 B_2 出口处的符号化映射反映了该基本块中对 a 、 b 和 c 赋值的语句的运行效果。其他的符号化映射可以用类似的方法推导得到。一旦我们确认图 9-58 中的映射是有效的, 就可以把图 9-55 中对 a 、 b 、 c 和 d 的赋值替换为适当的仿射表达式。也就是说, 我们可以把图 9-55 替换为图 9-59 中的代码。

单个语句的传递函数

这个数据流问题中的传递函数根据符号化映射计算得到新的符号化映射。为了计算一个赋值语句的传递函数, 我们解释该语句的语义, 并决定被赋值的变量能否被表示为赋值语句右边的

m	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$\text{IN}[B_1]$	NAA	NAA	NAA	NAA
$\text{OUT}[B_1]$	0	NAA	NAA	NAA
$\text{IN}[B_2]$	$i - 1$	NAA	NAA	NAA
$\text{OUT}[B_2]$	i	$10i$	0	NAA
$\text{IN}[B_3]$	i	$10i$	$j - 1$	NAA
$\text{OUT}[B_3]$	i	$10i$	j	$10i + j - 1$
$\text{IN}[B_4]$	i	$10i$	j	$10i + j - 1$
$\text{OUT}[B_4]$	$i - 1$	$10i - 10$	j	$10i + j - 11$

图 9-58 例 9.58 中的程序的符号化映射

```

1) a = 0;
2) for (i = 1; i <= 100; i++) {
3)   a = i;
4)   b = 10*i;
5)   c = 0;
6)   for (j = 1; j <= 10; j++) {
7)     d = 10*i + j - 1;
8)     c = j;
}
}

```

图 9-59 将图 9-55 的代码中的赋值语句替换为关于参考变量 i 和 j 的仿射表达式之后的代码

□

值的仿射表达式。所有其他变量的值保持不变。

一个语句 s 的传递函数记为 f_s , 其定义如下:

- 1) 如果 s 不是一个赋值语句, 那么 f_s 就是一个单元函数。
- 2) 如果 s 是一个对 x 赋值的语句, 那么

$$f_s(m)(x) = \begin{cases} m(v) & \text{对于所有的变量 } v \neq x \\ c_0 + c_1 m(y) + c_2 m(z) & \text{如果 } x \text{ 被赋值为 } c_0 + c_1 y + c_2 z, \\ & (c_1 = 0, \text{ 或者 } m(y) \neq \text{NAA}), \text{ 并且} \\ & (c_2 = 0, \text{ 或者 } m(z) \neq \text{NAA}) \\ \text{NAA} & \text{否则} \end{cases}$$

其中表达式 $c_0 + c_1 m(y) + c_2 m(z)$ 用来表示所有可能出现在对 x 赋值的语句的右部、关于变量 y 和 z 的各种形式的表达式。这些表达式向 x 赋予的值是变量之前的值的一次仿射变换的结果。这些表达式是 c_0 , $c_0 + y$, $c_0 - y$, $y + z$, $x - y$, $c_1 * y$ 和 $y/(1/c_1)$ 。请注意, 在很多情况下, c_0 、 c_1 和 c_2 中的一个或者多个的值为 0。

例 9.60 如果该赋值语句是 $x = y + z$, 那么 $c_0 = 0$ 而 $c_1 = c_2 = 1$ 。如果该赋值表达式是 $x = y/5$, 那么 $c_0 = c_2 = 0$ 而 $c_1 = 1/5$ 。 \square

关于值映射上的传递函数的注意事项

我们定义符号化映射上的传递函数的方法中有一个微妙之处, 即可以选择不同的方式来表示一个计算的效果。如果 m 是一个传递函数的输入映射, 那么 $m(x)$ 实际上表示的是“变量 x 在入口处可能具有的任何值”。我们努力尝试把该传递函数的结果表示为输入映射中用到的参考变量的仿射表达式。

你应该知道对 $f(m)(x)$ 这样的表达式的正确解释, 其中 f 是一个传递函数, m 是一个映射, 而 x 是一个变量。按照数学上的约定, 我们从左边开始应用函数, 也就是说我们首先计算 $f(m)$, 结果是一个映射。因为映射也是函数, 随后我们可以把它应用于一个变量 x 并得到一个值。

传递函数的组合

令 f_1 和 f_2 是两个以其输入映射 m 来定义的传递函数。为了计算 $f_2 \circ f_1$, 我们把 f_2 的定义中的 $m(v_i)$ 的值替换为 $f_1(m)(v_i)$ 的定义。我们把所有对 NAA 的运算都替换为 NAA。也就是:

- 1) 如果 $f_2(m)(v) = \text{NAA}$, 那么 $(f_2 \circ f_1)(m)(v) = \text{NAA}$ 。
- 2) 如果 $f_2(m)(v) = c_0 + \sum_i c_i m(v_i)$, 那么

$$(f_2 \circ f_1)(m)(v) = \begin{cases} \text{NAA} & \text{如果对于某个 } i \neq 0, c_i \neq 0 \text{ 且 } f_1(m)(v_i) = \text{NAA} \\ c_0 + \sum_i c_i f_1(m)(v_i) & \text{否则} \end{cases}$$

例 9.61 例 9.58 中的各个基本块的传递函数可以通过把组成它们的语句的传递函数组合起来计算得到。这些传递函数在图 9-60 中定义。 \square

数据流问题的解决方法

我们使用 $\text{IN}_{i,j}[B_3]$ 和 $\text{OUT}_{i,j}[B_3]$ 来表示在内层循环的第 j 次迭代和外层循环的第 i 次迭代时基本块 B_3 的输入和输出数据流值。对于其他的基本块, 我们使用 $\text{IN}_i[B_k]$ 和 $\text{OUT}_i[B_k]$ 来表示在外层循环的第 i 次迭代时的相应数据流值。我们还可以看到, 图 9-58 中显示的符号化映射满足传递函数给出的约束。这些约束在图 9-61 中列出。

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
f_{B_1}	0	$m(b)$	$m(c)$	$m(d)$
f_{B_2}	$m(a) + 1$	$10m(a) + 10$	0	$m(d)$
f_{B_3}	$m(a)$	$m(b)$	$m(c) + 1$	$m(b) + m(c)$
f_{B_4}	$m(a)$	$m(b)$	$m(c)$	$m(d)$

图 9-60 例 9.58 的传递函数

$$\begin{aligned}
 \text{OUT}[B_k] &= f_H(\text{IN}[B_k]), \text{ 对所有的 } B_k \\
 \text{OUT}[B_1] &\geq \text{IN}_1[B_2] \\
 \text{OUT}_i[B_2] &\geq \text{IN}_{i,1}[B_3], \quad 1 \leq i \leq 10 \\
 \text{OUT}_{i,j-1}[B_3] &\geq \text{IN}_{i,j}[B_3], \quad 1 \leq i \leq 100, 2 \leq j \leq 10 \\
 \text{OUT}_{i,10}[B_4] &\geq \text{IN}_i[B_4], \quad 2 \leq i \leq 100 \\
 \text{OUT}_{i-1}[B_4] &\geq \text{IN}_i[B_2], \quad 1 \leq i \leq 100
 \end{aligned}$$

图 9-61 嵌套循环的每次迭代上满足的约束

第一个约束说明，一个基本块的输出映射是通过把基本块的传递函数应用到输入映射上而得到的。其余的约束说明，在程序执行的时候，一个基本块的输出映射必须大于或等于后继基本块的输入映射。

请注意，我们的迭代数据流算法不能给出上面的解，因为它无法用已执行的迭代次数来表达数据流值。正如我们将在下一节看到的，可以用基于区域的分析来找出这样的解。

9.8.3 基于区域的符号化分析

我们可以把 9.7 节中描述的基于区域的分析技术进行扩展，用以寻找一个循环的第 i 次迭代中各个变量的表达式。和其他基于区域的算法一样，一个基于区域的符号化分析也有一个自底向上的处理过程和一个自顶向下的处理过程。这个自底向上的处理过程用一个传递函数来概括一个区域的执行效果。这个传递函数把入口处的符号化映射转变为出口处的输出符号化映射。在自顶向下的处理过程中，符号化映射的值被向下传播到内层区域。

不同之处在于我们处理循环的方法。在 9.7 节，循环的效果是用闭包运算来概括的。给定一个其循环体传递函数为 f 的循环， f 的闭包 f^* 被定义为在任意多次应用 f 可能产生的所有效果之上无穷多次应用交汇运算而得到的结果。但是，为了找到一个归纳变量，我们需要确定一个变量的值是否为至今已执行的迭代次数的仿射函数。相应的符号化映射必须把正在执行的迭代的序号作为参数。不仅如此，只要我们知道一个循环执行迭代的总次数，就可以使用这个数字来找到循环之后归纳变量的值。比如，在例 9.58 中我们断定在执行了第 i 次迭代之后， a 的值是 i 。因为循环共有 100 次迭代，在循环结束的时候 a 的值一定是 100。

接下来，我们首先定义基本运算符：用于符号化分析的传递函数的交汇运算和组合运算。然后说明如何使用它们进行基于区域的归纳变量分析。

传递函数的交汇运算

当计算两个函数的交时，除非两个函数把一个变量映射成为同一个不是 NAA 的值，这个变量的值就是 NAA。因此

$$(f_1 \wedge f_2)(m)(v) = \begin{cases} f_1(m)(v) & \text{如果 } f_1(m)(v) = f_2(m)(v) \\ \text{NAA} & \text{否则} \end{cases}$$

带参数的函数组合

为了把一个变量表示成为一个关于循环下标的仿射函数，我们要计算出将某个函数组合给定多次后的效果。如果一次迭代的效果可以用一个传递函数 f 概括，那么对某个 $i \geq 0$ ，执行 i 次迭代的效果记为 f^i 。请注意，当 $i=0$ 时， $f^i=f^0=I$ 是一个单元函数。

程序中的变量可以分成四种类型：

1) 如果 $f(m)(x) = m(x) + c$ ，其中 c 是一个常数，那么对于所有的 $i \geq 0$ ， $f^i(m)(x) = m(x) + ci$ 。如果一个循环的循环体可以用传递函数 f 表示，我们说 x 是这个循环的一个基本归纳变量 (basic induction variable)。

2) 如果 $f(m)(x) = m(x)$, 那么对于所有的 $i \geq 0$, $f^i(m)(x) = m(x)$ 。变量 x 没有被改变。如果循环的循环体具有传递函数 f , 那么 x 的值在循环的任意多次迭代结束之后依然保持不变。我们说 x 是该循环的符号化常量 (symbolic constant)。

3) 如果 $f(m)(x) = c_0 + c_1 m(x_1) + \cdots + c_n m(x_n)$, 其中每个 x_k 要么是基本归纳变量, 要么是符号化常量, 那么对于 $i > 0$, 有

$$f^i(m)(x) = c_0 + c_1 f^i(m)(x_1) + \cdots + c_n f^i(m)(x_n)$$

我们说 x 虽然不是基本归纳变量, 但它依然是一个归纳变量。请注意, 上述公式对于 $i=0$ 不成立。

4) 在其他情况下, $f^i(m)(x) = \text{NAA}$ 。

要得到执行固定多次迭代的效果, 我们只需要把上面的 i 替换成为该迭代次数即可。当迭代次数未知时, 在最后一次迭代开始时变量的值由 f^* 给出。在这种情况下, 其值仍然可以用仿射函数表示的变量只有那些循环不变变量。

$$f^*(m)(v) = \begin{cases} m(v) & \text{如果 } f(m)(v) = m(v) \\ \text{NAA} & \text{否则} \end{cases}$$

例 9.62 对于例 9.58 的最内层循环, 执行 $i (i > 0)$ 次迭代的效果由传递函数 $f_{B_3}^i$ 描述。根据 f_{B_3} 的定义, 我们看到 a 和 b 是符号化常量。因为 c 在每次迭代中增加一, 所以它是一个基本归纳变量。因为 d 是符号化常量 b 和基本归纳变量 c 的仿射函数, 所以它是一个归纳变量。由此可得:

$$f_{B_3}^i(m)(v) = \begin{cases} m(a) & \text{如果 } v = a \\ m(b) & \text{如果 } v = b \\ m(c) + i & \text{如果 } v = c \\ m(b) + m(c) + i & \text{如果 } v = d \end{cases}$$

如果我们不能指出基本块 B_3 的循环迭代了多少次, 那么就不能使用 f^i , 而必须使用 f^* 来表示在循环结束时的条件。此时我们有

$$f_{B_3}^*(m)(v) = \begin{cases} m(a) & \text{如果 } v = a \\ m(b) & \text{如果 } v = b \\ \text{NAA} & \text{如果 } v = c \\ \text{NAA} & \text{如果 } v = d \end{cases}$$

□

一个基于区域的算法

算法 9.63 基于区域的符号化分析。

输入: 一个可归约的流图 G 。

输出: G 的每个基本块 B 的符号化映射 $\text{IN}[B]$ 。

方法: 我们对算法 9.53 做出如下的修改。

1) 我们改变了为一个循环区域构造传递函数的方法。在原来的算法中, 我们使用传递函数 $f_{R, \text{IN}[S]}$ 来把循环区域 R 入口处的符号化映射变换为经过未知多次迭代之后位于循环体 S 的入口处的符号化映射。如图 9-50b 所示, 这个函数被定义为所有回到循环入口处的路径的传递函数的闭包。在这里, 我们定义 $f_{R, i, \text{IN}[S]}$ 来表示从循环区域入口处开始直到第 i 次迭代的入口处的执行效果。因此,

$$f_{R, i, \text{IN}[S]} = (\bigwedge_{\text{头结点 } S \text{ 的在 } R \text{ 中的前驱 } B} f_{S, \text{OUT}[B]})^{i-1}$$

2) 如果一个区域的迭代次数已知, 该区域的执行效果的描述是把上面定义中的 i 替换为实际迭代次数。

3) 在算法的自顶向下处理过程中, 我们计算 $f_{R_i, i, \text{IN}[S]}$ 就可以找出与一个循环的第 i 次迭代的人口处相关的符号化映射。

4) 如果一个变量的输入值 $m(v)$ 被区域 R 中的某个符号化映射的右部使用, 并且在该区域的人口处 $m(v) = \text{NAA}$, 则我们引入一个新的参考变量 t , 在区域 R 的开始处加上赋值语句 $t = v$, 并且所有对 $m(v)$ 的引用都被替换为 t 。如果我们不在这个点上引入一个参考变量, 那么 v 的取值 NAA 将被传递到内层循环。 \square

例 9.64 对于例 9.58, 我们在图 9-62 中显示了该程序的传递函数是如何在算法的自底向上处理过程中被计算出来的。区域 R_5 是内层循环, 它的循环体是 B_5 。表示从区域 R_5 的人口处到达第 j ($j \geq 1$) 次迭代开始处的路径的传递函数是 $f_{B_3}^{j-1}$; 表示到达第 j 次迭代结尾处的路径的传递函数是 $f_{B_3}^j$ 。

区域 R_6 由基本块 B_2 和 B_4 以及它们之间的循环区域 R_5 组成。从 B_2 和 R_5 的人口处开始的传递函数可以用原算法中的同样方法来计算。因为 f_{B_4} 是一个单元函数, 所以传递函数 $f_{R_6, \text{OUT}[B_3]}$ 表示了基本块 B_2 和整个内层循环的执行效果的组合。因为已知内层循环将迭代 10 次, 所以我们可以把 j 替换为 10 来精确描述内层循环的执行效果。其余的传递函数可以用类似的方式计算得到。计算得到的实际传递函数显示在图 9-63 中。

$$\begin{aligned} f_{R_5, j, \text{IN}[B_3]} &= f_{B_3}^{j-1} \\ f_{R_5, j, \text{OUT}[B_3]} &= f_{B_3}^j \\ f_{R_6, \text{IN}[B_2]} &= I \\ f_{R_6, \text{IN}[R_5]} &= f_{B_2} \\ f_{R_6, \text{OUT}[B_4]} &= I \circ f_{R_5, 10, \text{OUT}[B_3]} \circ f_{B_2} \\ f_{R_7, i, \text{IN}[R_6]} &= f_{R_6, \text{OUT}[B_4]}^{i-1} \\ f_{R_7, i, \text{OUT}[B_4]} &= f_{R_6, \text{OUT}[B_4]}^i \\ f_{R_8, \text{IN}[B_1]} &= I \\ f_{R_8, \text{IN}[R_7]} &= f_{B_1} \\ f_{R_8, \text{OUT}[B_4]} &= f_{R_7, 100, \text{OUT}[B_4]} \circ f_{B_1} \end{aligned}$$

图 9-62 例 9.58 的自底向上处理
过程中的传递函数的关系

f	$f(m)(a)$	$f(m)(b)$	$f(m)(c)$	$f(m)(d)$
$f_{R_5, j, \text{IN}[B_3]}$	$m(a)$	$m(b)$	$m(c) + j - 1$	NAA
$f_{R_5, j, \text{OUT}[B_3]}$	$m(a)$	$m(b)$	$m(c) + j$	$m(b) + m(c) + j - 1$
$f_{R_6, \text{IN}[B_2]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_6, \text{IN}[R_5]}$	$m(a) + 1$	$10m(a) + 10$	0	$m(d)$
$f_{R_6, \text{OUT}[B_4]}$	$m(a) + 1$	$10m(a) + 10$	10	$10m(a) + 9$
$f_{R_7, i, \text{IN}[R_6]}$	$m(a) + i - 1$	NAA	NAA	NAA
$f_{R_7, i, \text{OUT}[B_4]}$	$m(a) + i$	$10m(a) + 10i$	10	$10m(a) + 10i + 9$
$f_{R_8, \text{IN}[B_1]}$	$m(a)$	$m(b)$	$m(c)$	$m(d)$
$f_{R_8, \text{IN}[R_7]}$	0	$m(b)$	$m(c)$	$m(d)$
$f_{R_8, \text{OUT}[B_4]}$	100	1000	10	1009

图 9-63 在例 9.58 的自底向上处理过程中计算得到的传递函数

在程序入口处的符号化映射就是 m_{NAA} 。我们使用自顶向下处理过程来计算到达逐层嵌套的区域的人口处的符号化映射, 直到我们得到了所有基本块的符号化映射为止。一开始的时候我们首先计算区域 R_8 中的基本块 B_1 的数据流值:

$$\begin{aligned} \text{IN}[B_1] &= m_{\text{NAA}} \\ \text{OUT}[B_1] &= f_{B_1}(\text{IN}[B_1]) \end{aligned}$$

再向下到达区域 R_7 和 R_6 , 我们得到

$$\begin{aligned} \text{IN}_i[B_2] &= f_{R_7, i, \text{IN}[R_6]}(\text{OUT}[B_1]) \\ \text{OUT}_i[B_2] &= f_{B_2}(\text{IN}_i[B_2]) \end{aligned}$$

最后, 在区域 R_5 中我们得到

$$\begin{aligned} \text{IN}_{i,j}[B_3] &= f_{R_5, j, \text{IN}[B_3]}(\text{OUT}_i[B_2]) \\ \text{OUT}_{i,j}[B_3] &= f_{B_3}(\text{IN}_{i,j}[B_3]) \end{aligned}$$

毫不奇怪，这些等式产生的就是我们在图 9-58 中显示的结果。□

例 9.58 显示了一个简单的程序，其中的各个符号化映射中的每个变量都有一个仿射表达式。我们使用例 9.65 来说明为什么以及如何在算法 9.63 中引入参考变量。

例 9.65 考虑图 9-64a 中的简单例子。令 f_j 为描述内层循环迭代 j 次的执行效果的传递函数。即使 a 的值可能在该循环的执行中上下变动，我们看到 b 是一个基于 a 在此循环的入口处的取值的归纳变量。也就是说， $f_j(m)(b) = m(a) - 1 + j$ 。因为 a 被赋予了一个输入值，所以在内层循环入口处的符号化映射把 a 映射为 NAA。我们在该入口处引入一个新的参考变量 t 来保存 a 的值，并像图 9-64b 中那样进行替换。□

9.8.4 9.8 节的练习

练习 9.8.1：对于图 9-10 中的流图（见 9.1 节的练习），给出下列基本块的传递函数。

- 1) 基本块 B_2 。
- 2) 基本块 B_4 。
- 3) 基本块 B_5 。

练习 9.8.2：考虑图 9-10 中由基本块 B_3 和 B_4 组成的内层循环。如果 i 表示了该循环的迭代执行次数，而 f 是从该循环的入口（即 B_3 的开始处）到 B_4 的出口处的循环体（即不包含 B_4 到 B_3 的边）的传递函数，那么 f^i 是什么？请记住， f 把一个映射 m 作为参数，而 m 给变量 a, b, d, e 中的每一个赋予一个值。虽然我们不知道这些变量的值，但是我们用 $m(a)$ 等来表示它们。

练习 9.8.3：现在考虑图 9-10 中由 B_2, B_3, B_4, B_5 组成的外层循环。令 g 为循环的入口处 B_2 到它的出口处 B_5 的循环体的传递函数。令 i 表示由 B_3 和 B_4 组成的内层循环的迭代次数（我们无法知道迭代的具体次数），并令 j 表示外层循环的迭代次数（我们还是无法知道迭代的具体次数）。那么 g^j 是什么？

9.9 第 9 章总结

- 全局公共子表达式：一个重要的优化方法是寻找同一个表达式在两个不同基本块中的计算过程。如果一个在另一个前面，我们可以把第一次计算该表达式时得到的结果存放起来，并在再次计算该表达式时使用这个结果。
- 复制传播：一个复制语句 $u = v$ 把一个变量 v 赋值给另一个变量 u 。在有些情况下，我们可以把所有对 u 的使用替换为对 v 的使用，从而消除这个赋值语句以及变量 u 。
- 代码移动：另一种优化方法是把一个计算过程移动到它所在的循环之外。只有当循环的每次迭代中这个计算过程都生成同样的值，这种改变才是正确的。
- 归纳变量：很多循环都有归纳变量。这些变量在循环执行时的不同迭代中的取值是一个线性序列。有些归纳变量仅仅用于对迭代进行计数，它们经常可以被消除，从而降低了

```

1) for (i = 1; i < n; i++) {
2)   a = input();
3)   for (j = 1; j < 10; j++) {
4)     a = a - 1;
5)     b = j + a;
6)     a = a + 1;
}

```

a) 变量 a 在其中上下变动的一个循环

```

for (i = 1; i < n; i++) {
  a = input();
  t = a;
  for (j = 1; j < 10; j++) {
    a = t - 1;
    b = t - 1 + j;
    a = t;
  }
}

```

b) 参考变量 t 使得 b 成为一个归纳变量

图 9-64 引入参考变量的需求

循环的一次迭代所需要的时间。

- **数据流分析：**一个数据流分析模式在程序的每个点上都定义了一个值。程序的各个语句都有相关联的传递函数。这些函数给出了一个语句之前和之后的数据流值之间的关系。具有多个前驱的语句的值是它的各个前驱的值的组合。这个组合通过交汇(或者说汇流)函数计算得到。
- **基本块的数据流分析：**因为数据流值在一个基本块内的传播过程通常很简单，所以数据流方程通常给每个基本块设置两个值，称为 IN 值和 OUT 值。这两个值分别表示该基本块在开始处和结尾处的数据流值。把基本块中各个语句的传递函数组合起来就可以得到代表整个基本块的传递函数。
- **到达定值：**到达定值数据流框架的数据流值是程序中的语句的集合。这些语句给一个或者多个变量定值。如果一个变量肯定在一个基本块内被重新定值，那么该基本块的传递函数杀死了对这个变量的定值，同时它还加入(“生成”)了在该模块中发生的对变量的定值。只要一个定值到达某个点的任意一个前驱，它就到达了该点，因此交汇运算是并集运算。
- **活跃变量：**另一个重要的数据流框架计算了在各个程序点上活跃的(将在重新定值之前被使用的)变量。这个框架和到达定值框架类似，但是传递函数是逆向传递数据流值的。一个变量在某个基本块的开始处活跃的条件是，要么在该基本块中它在定值之前就被使用，要么该基本块中没有对它重新定值且它在该基本块结尾处活跃。
- **可用表达式：**为了寻找全局公共子表达式，我们要确定各个程序点上的可用表达式。所谓可用表达式就是之前已经计算过，且在最后一次计算之后它的运算分量都没有被重新定值的表达式。这个问题的数据流框架和到达定值框架类似，但是其交汇运算是交集运算，而不是并集运算。
- **数据流问题的抽象：**常见的数据流问题，比如前面提到过的那些，都可以用一个通用的数学结构表达。数据流值是一个半格的成员，这个半格的交汇运算就是数据流问题的交汇(汇流)函数。传递函数把半格元素映射到半格元素。要求传递函数的集合必须对于组合运算封闭，并且包含单元函数。
- **单调框架：**每个半格都有一个 \leq 关系 $a \leq b$ 当且仅当 $a \wedge b = a$ 。单调框架具有以下性质：每个传递函数都保持了 \leq 关系。也就是说，对于任意的格元素 a 和 b 以及传递函数 f ， $a \leq b$ 蕴含了 $f(a) \leq f(b)$ 。
- **可分配框架：**这种框架满足下面的条件：对于所有的格元素 a 和 b 以及传递函数 f ， $f(a \wedge b) = f(a) \wedge f(b)$ 。可以证明可分配框架的条件蕴含了单调框架的条件。
- **抽象框架的迭代解法：**所有的单调数据流框架可以通过一个迭代算法来解决。在这个解法中，首先(按照不同的框架)适当地初始化各个基本块的 IN 和 OUT 值，然后应用传递函数和交汇运算不断地计算这些变量的新值。这个解法总是安全的(即按照它的解对程序进行优化不会改变程序所做的计算)。但是只有当框架是可分配的时，这个解才一定是可能的解中最好的。
- **常量传播框架：**虽然诸如到达定值这类的基本框架都是可分配的，但存在一些单调但不可分配的框架。这类框架中的一个例子是关于常量传播的。在常量传播框架使用的半格中，格元素是从程序变量到常量以及两个特殊值的映射。这两个特殊值分别代表“无信息”和“一定不是常量”。
- **部分冗余消除：**很多有用的优化，比如代码移动和全局公共子表达式消除，可以被扩展为同一个问题。该问题称为部分冗余消除。如果在某个点上需要计算一个表达式，但是这个表

达式只在到达这个点的部分路径上可用，那么我们可以只在该表达式不可用的路径上进行计算。正确地应用这个想法要求解决四个不同的数据流问题，并做一些其他的操作。

- 支配结点：如果在一个流图中所有到达某结点的路径都必须经过另一个结点，那么后一个结点就支配前一个结点。一个真支配结点是不同于被支配结点的支配结点。除了入口结点，每个结点都有一个直接支配结点——被该结点的所有其他真支配结点所支配的真支配结点。
- 流图的深度优先排序：如果我们从一个流图的入口结点开始对它进行深度优先搜索，我们会得到一个深度优先生成树。结点的深度优先排序是这棵树的后序遍历次序的逆序。
- 边的分类：当我们构造一个深度优先生成树之后，相应流图的全部边可以分成三大类：前进边（即从祖先结点到真后代结点的边）、后退边（即从后代结点到祖先结点的边）和交叉边（其他）。生成树的一个重要性质是所有的交叉边都是从树的右边到达左边。另一个重要性质是在这些边中，如果按照深度优先排序（即后序次序的逆序），只有后退边的头比它的尾的排序更靠前。
- 回边：回边就是其头结点支配尾结点的边。不管选择流图的哪一棵深度优先生成树，每条回边都是一条后退边。
- 可归约流图：如果不管选择哪个深度优先生成树，该树的每个后退边都是一条回边，那么这个流图就是可归约的。绝大部分流图都是可归约的，控制流语句都是通常的循环和分支语句的程序的流图一定是可归约的。
- 自然循环：一个自然循环是一个结点的集合。集合中有一个头结点，它支配了该集合中的所有其他结点，并且至少有一条回边进入这个头结点。给定任意的回边，我们可以构造出它的自然循环。循环中包括回边的头结点，以及所有不经过头结点就能够到达此回边的尾结点的其他结点。两个具有不同头结点的自然循环要么互不相交，要么一个循环完全包含在另一个循环里面。这个性质使得我们可以讨论嵌套循环的层次结构，前提是“循环”指的是自然循环。
- 深度优先排序提高了迭代算法的效率：如果沿着无环路径传播信息足以得到正确结果，即环路不会增加信息，那么相应的迭代算法只需要很少几次迭代就可以得到正确结果。如果我们按照深度优先顺序访问结点，那么任何向前传递信息的数据流框架（比如到达定值）都可以在确定次数内收敛。收敛次数不大于所有无环路径中的后退边的最大个数加上2。如果我们用深度优先顺序的逆序（即后序次序）访问结点，上面的结论对于逆向传播的框架（比如活跃变量）也成立。
- 区域：区域是一个结点和边的集合。区域中有一个头结点 h 支配了其中的所有结点。除了 h 之外，区域中所有结点的前驱必须也在此区域中。区域的边集包含了区域中的任意两个结点之间的边，但是可能不包含某些或所有到达头结点的边。
- 区域和可归约流图：可归约流图可以被扫描分析成为一个由区域组成的层次结构。这些区域要么是循环区域，要么是循环体区域。循环区域包含了所有进入头结点的边，而循环体区域不包含到达头结点的边。
- 基于区域的数据流分析：不同于迭代方法的另一种数据流分析方法是沿着区域层次结构向上然后再向下扫描，计算从各个区域的头到达该区域中各个结点的传递函数。
- 基于区域的归纳变量检测：基于区域的分析技术的重要应用之一是用以寻找归纳变量的数据流框架。该框架试图找出循环区域中每个满足下面条件的变量的公式。这些变量的值可表示为循环迭代次数的仿射（线性）函数。

9.10 第 9 章参考文献

两个对代码作充分优化的早期编译器是 Alpha[7] 和 Fortran H[16]。关于循环优化技术(比如代码移动)的基础性论文是[1]，虽然论文中的某些思想的早期版本出现在[8]中。一本非正式发行的书[4]在传播代码优化思想方面很有影响。

对数据流分析的迭代算法的第一个描述来自于 Vyssotsky 和 Wegner 的未发表的技术报告[20]。对于数据流分析的科学研究被认为是从 Allen[2] 和 Cocke[3] 的两篇文章开始的。

本节描述的基于格理论的抽象是基于 Kildall[13] 的文章。文中假设这些框架具有可分配性，但是很多框架不满足这个性质。在很多这样的框架出现后，论文[5]和[11]把单调性条件加入到模型中去。

部分冗余消除是[17]首先提出的。而本章中描述的懒惰代码移动算法是基于[14]。

支配结点的概念由[13]中描述的编译器首先使用。但是这个思想最早出现在[18]中。

可归约流图的概念来自于[2]。像本章中表示的这些流图的结构来自于[9]和[10]。[12]和[15]首先把流图的可归约性与常见的嵌套式控制流结构联系起来。这种联系解释了为什么这一类流图是如此的常见。

通过 $T_1 - T_2$ 归约来定义流图可归约性的思想来自于[19]。在基于区域的分析技术中使用了这个定义。基于区域的方法首先被[21]中所描述的编译器使用。

在 6.2.4 节中介绍的静态单赋值(Static Single-Assignment, SSA)中间表示形式把数据流和控制流都合并到其表示方法中。SSA 表示法支持了同一个公共框架中的很多种优化转换的实现[6]。

1. Allen, F. E., "Program optimization," *Annual Review in Automatic Programming* 5 (1969), pp. 239-307.
2. Allen, F. E., "Control flow analysis," *ACM Sigplan Notices* 5:7 (1970), pp. 1-19.
3. Cocke, J., "Global common subexpression elimination," *ACM SIGPLAN Notices* 5:7 (1970), pp. 20-24.
4. Cocke, J. and J. T. Schwartz, *Programming Languages and Their Compilers: Preliminary Notes*, Courant Institute of Mathematical Sciences, New York Univ., New York, 1970.
5. Cousot, P. and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," *Fourth ACM Symposium on Principles of Programming Languages* (1977), pp. 238-252.
6. Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems* 13:4 (1991), pp. 451-490.
7. Ershov, A. P., "Alpha — an automatic programming system of high efficiency," *J. ACM* 13:1 (1966), pp. 17-24.
8. Gear, C. W., "High speed compilation of efficient object code," *Comm. ACM* 8:8 (1965), pp. 483-488.

9. Hecht, M. S. and J. D. Ullman, "Flow graph reducibility," *SIAM J. Computing* 1 (1972), pp. 188-202.
10. Hecht, M. S. and J. D. Ullman, "Characterizations of reducible flow graphs," *J. ACM* 21 (1974), pp. 367-375.
11. Kam, J. B. and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica* 7:3 (1977), pp. 305-318.
12. Kasami, T., W. W. Peterson, and N. Tokura, "On the capabilities of while, repeat, and exit statements," *Comm. ACM* 16:8 (1973), pp. 503-512.
13. Kildall, G., "A unified approach to global program optimization," *ACM Symposium on Principles of Programming Languages* (1973), pp. 194-206.
14. Knoop, J., "Lazy code motion," *Proc. ACM SIGPLAN 1992 conference on Programming Language Design and Implementation*, pp. 224-234.
15. Kosaraju, S. R., "Analysis of structured programs," *J. Computer and System Sciences* 9:3 (1974), pp. 232-255.
16. Lowry, E. S. and C. W. Medlock, "Object code optimization," *Comm. ACM* 12:1 (1969), pp. 13-22.
17. Morel, E. and C. Renvoise, "Global optimization by suppression of partial redundancies," *Comm. ACM* 22 (1979), pp. 96-103.
18. Prosser, R. T., "Application of boolean matrices to the analysis of flow diagrams," *AFIPS Eastern Joint Computer Conference* (1959), Spartan Books, Baltimore MD, pp. 133-138.
19. Ullman, J. D., "Fast algorithms for the elimination of common subexpressions," *Acta Informatica* 2 (1973), pp. 191-213.
20. Vyssotsky, V. and P. Wegner, "A graph theoretical Fortran source language analyzer," unpublished technical report, Bell Laboratories, Murray Hill NJ, 1963.
21. Wulf, W. A., R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, New York, 1975.