

9.3 静态单赋值形式

随着时间的推移，许多不同的数据流问题已经得以阐明。如果每种变换都使用其自身特异性的分析，那么在分析这一趟处理上花费的实现、调试及维护方面的时间和工作可能变得过多。为限制编译器编写者必须实现和编译器必须运行的分析的数目，使用单趟分析来支持多种变换是可取的。

实现此类“通用”分析的一种策略涉及建立程序的一种变体形式，将数据流和控制流均直接编码到IR中。5.4.2节和8.5.1节引入的静态单赋值形式具有这种性质。它可以充当很大一组变换的基础。只要有一种实现能够将代码转换为静态单赋值形式，编译器即可进行许多经典的标量优化。

如图9-7a所示代码片断，请考虑其中对变量x的各处使用。灰线显示了哪些定义可以到达x的每个使用处。图9-7b给出了改写过的同一代码片断，其中已经将x转换为静态单赋值形式。x的各处定义已经用下标重命名，以确保每个定义都有唯一的静态单赋值形式名。为简单起见，我们保留对其他变量的引用原样不动。

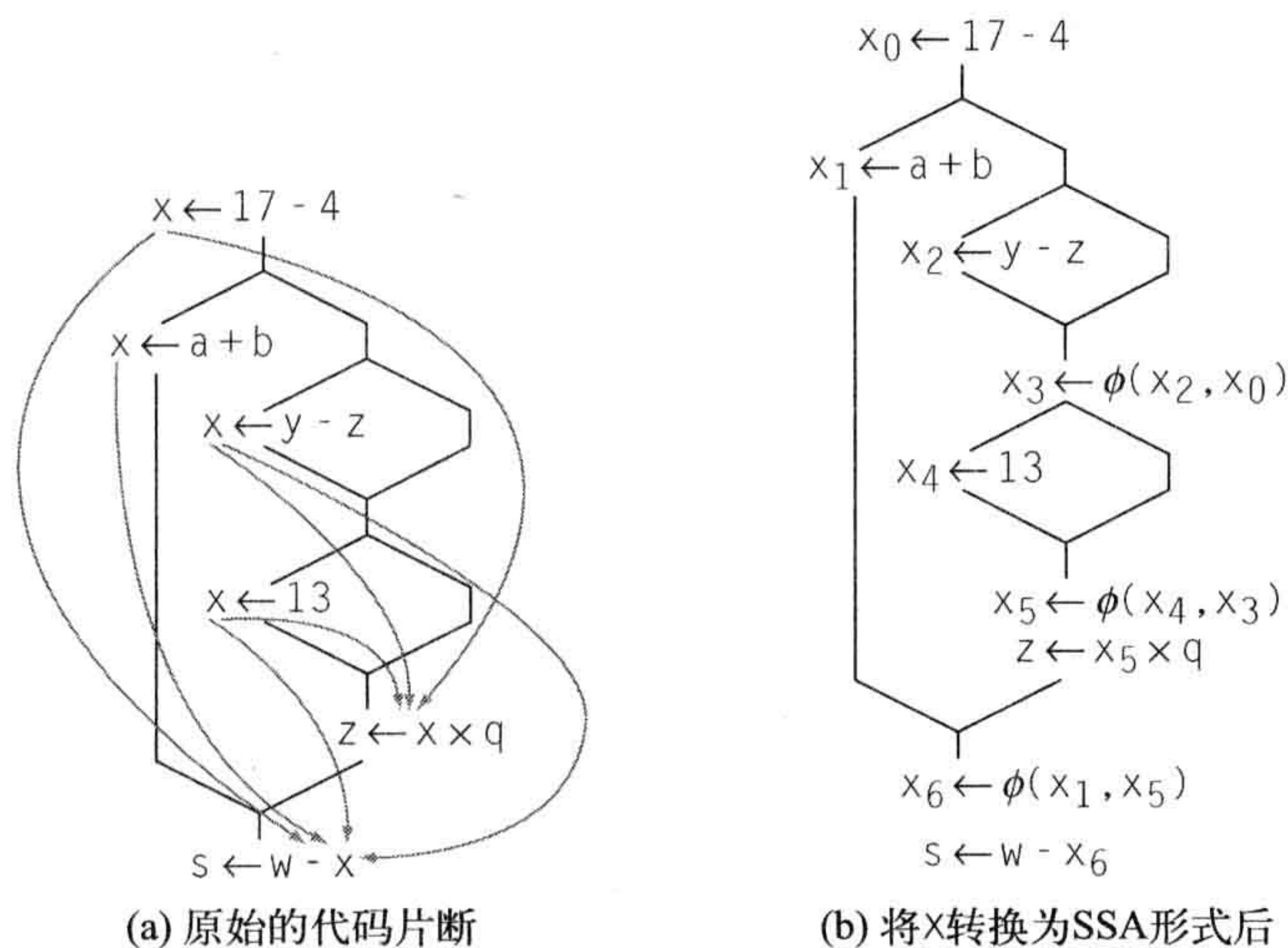


图9-7 静态单赋值形式：将控制流编码到数据流中

该代码的静态单赋值形式包括了新的赋值（到 x_3 、 x_5 和 x_6 ），这些使得x的各个的静态单赋值形式名与x的各处使用（在对s和z的赋值中）协调起来。这些赋值确保了沿CFG中的每条边，x的当前值都分配了一个唯一的名字，名字的分配与控制流沿哪条代码路径转移到该边无关。这些赋值操作的右侧包含了一个特殊函数—— ϕ 函数，其中合并了来自不同边的值。

ϕ 函数的参数是与进入基本程序块的各条边相关联的值的静态单赋值形式名。在控制流进入一个基本程序块时，该程序块中的所有 ϕ 函数都将并发执行。当前控制流进入基本程序块时经由的CFG边对应的参数即为 ϕ 函数的值。在记号表示方面，我们是从左到右书写各个参数，则对应于进入程序块各条边从左到右的次序。在印刷纸面上，这一点很容易。但在编译器实现中需要做一些簿记工作。

构造静态单赋值形式的过程会在CFG中每个汇合点之后插入 ϕ 函数，汇合点即为CFG中多条代码路径汇合之处。在汇合点处，不同的静态单赋值形式名必须调和为一个名字。在整个过程已经转换为静态单赋值形式之后，有两条规则必定成立：(1)过程中的每个定义都创建了一个唯一的名字；(2)每个

使用处都引用了一个定义。

为将一个过程转换为静态单赋值形式，编译器必须为每个变量向代码插入适当的 ϕ 函数，且必须用下标重命名变量以符合上述两个规则。这个简单的两步计划就产生了构造静态单赋值形式的基本算法。

9.3.1 构造静态单赋值形式的简单方法

为构造程序的静态单赋值形式，编译器必须向CFG中的汇合点处插入 ϕ 函数，且必须重命名变量和临时值，使之符合支配静态单赋值形式名字空间的规则。该算法概述如下。

(1) 插入 ϕ 函数 在具有多个前趋的每个程序块起始处，为当前过程中定义或使用的每个名字 y ，插入一个 ϕ 函数，如 $y \leftarrow \phi(y, y)$ 。对于CFG中的每一个前趋块， ϕ 函数都应该有一个参数与之对应。这一规则在需要插入 ϕ 函数之处均插入一个 ϕ 函数。当然它也插入了许多非必要的 ϕ 函数。

此算法可以按任意次序插入 ϕ 函数。 ϕ 函数的定义要求位于程序块顶部的所有 ϕ 函数并发执行，即它们同时读取其输入参数，然后同时写出其输出值。这使算法能够避免次序可能引入的许多次要细节。

(2) 重命名 在插入 ϕ 函数之后，编译器可以计算可达定义（参见9.2.4节）。由于插入的 ϕ 函数也是定义，它们确保了对任一使用处都只有一个定义能够到达。接下来，编译器可以重命名每个使用处的变量和临时值，以反映到达该处的定义。

编译器必须对到达每个 ϕ 函数的定义进行归类，使之与到达 ϕ 函数所在程序块的代码路径相对应。虽然在概念上颇为简单，但这项任务是需要一些簿记工作的。

该算法为程序构造出了一个正确的静态单赋值形式。每个变量都刚好定义一次，而每个引用都使用了某个不同定义的名字。但该算法产生的静态单赋值形式可能具有很多不必要的 ϕ 函数。这些额外的 ϕ 函数可能会带来问题。它们会降低在静态单赋值形式上执行的某些种类分析的精确度。它们会占用空间，使得编译器浪费内存来表示冗余（即形如 $x_j \leftarrow \phi(x_i, x_i)$ ）或不活动的 ϕ 函数。它们同样会增加使用由此产生的静态单赋值形式的任何算法的代价，因为相关的算法必须遍历所有不必要的 ϕ 函数。

我们将此版本的静态单赋值形式称为最大静态单赋值形式（maximal SSA form）。构建具有较少 ϕ 函数的静态单赋值形式需要更多的工作，特别地，编译器必须分析代码来确定不同的值在CFG中何处会聚。这种计算依赖于9.2.1节描述的支配性信息。

接下来的三个小节将详细阐述一种构建半剪枝静态单赋值形式（semipruned SSA form）的算法，这种版本的静态单赋值形式具有较少的 ϕ 函数。9.3.2节说明了如何使用9.2.1节引入的支配性信息来计算支配边界（dominance frontier），以指引 ϕ 函数的插入。9.3.3节给出了一种用于插入 ϕ 函数的算法，9.3.4节说明了如何重写变量名以完成静态单赋值形式的构建，9.3.5节讨论了在将代码转换回可执行形式的过程中可能出现的困难。

9.3.2 支配边界

最大静态单赋值形式的主要问题是它包含了过多的 ϕ 函数。为减少 ϕ 函数的数目，编译器必须更谨慎地判断何处真正需要 ϕ 函数。放置 ϕ 函数的关键在于，理解在每个汇合点处究竟哪个变量需

要 ϕ 函数。为高效并有效地解决这个问题，编译器可以改变问题的表述方式。对每个程序块 i ，编译器可以判断对程序块 i 中某些定义需要插入 ϕ 函数的程序块的集合。在这种计算中，支配性发挥了关键作用。

考虑CFG的结点 n 中的一个定义。该值到达某个结点 m 时，如果 $n \in \text{Dom}(m)$ ，则该值不需要 ϕ 函数，因为到达 m 的每条代码路径都必然经由 n 。该值无法到达 m 的唯一可能是有另一个同名定义的干扰，即在 n 和 m 之间的某个结点 p 中，出现了与该值同名的另一个定义。在这种情况下，在 n 中的定义无需 ϕ 函数，而 p 中的重新定义则需要。

结点 n 中的定义，仅在CFG中 n 支配区域以外的汇合点，才需要插入相应的 ϕ 函数。更正式地说，结点 n 中的定义，仅在满足下述两个条件的汇合点才需要插入对应的 ϕ 函数：(1) n 支配 m 的一个前趋 ($q \in \text{preds}(m)$ 且 $n \in \text{Dom}(q)$)；(2) n 并不严格支配 m 。（使用严格支配性而非支配性，使得可以在单个基本程序块构成的循环起始处插入一个 ϕ 函数。在这种情况下， $n = m$ ，且 $m \notin \text{Dom}(n) - \{n\}$ 。）我们将相对于 n 具有这种性质的结点 m 的集合称为 n 的支配边界，记作 $\text{DF}(n)$ 。

严格支配性
当且仅当 $a \in \text{DOM}(b) - \{b\}$ 时， a 严格支配 b 。

非正式地， $\text{DF}(n)$ 包含：在离开 n 的每条CFG路径上，从结点 n 可达但不支配的第一个结点。在我们一直使用的例子的CFG中， B_5 支配 B_6 、 B_7 和 B_8 ，但并不支配 B_3 。在每条离开 B_5 的路径上， B_3 都是 B_5 不支配的第一个结点，因而， $\text{DF}(B_5) = \{B_3\}$ 。

1. 支配者树

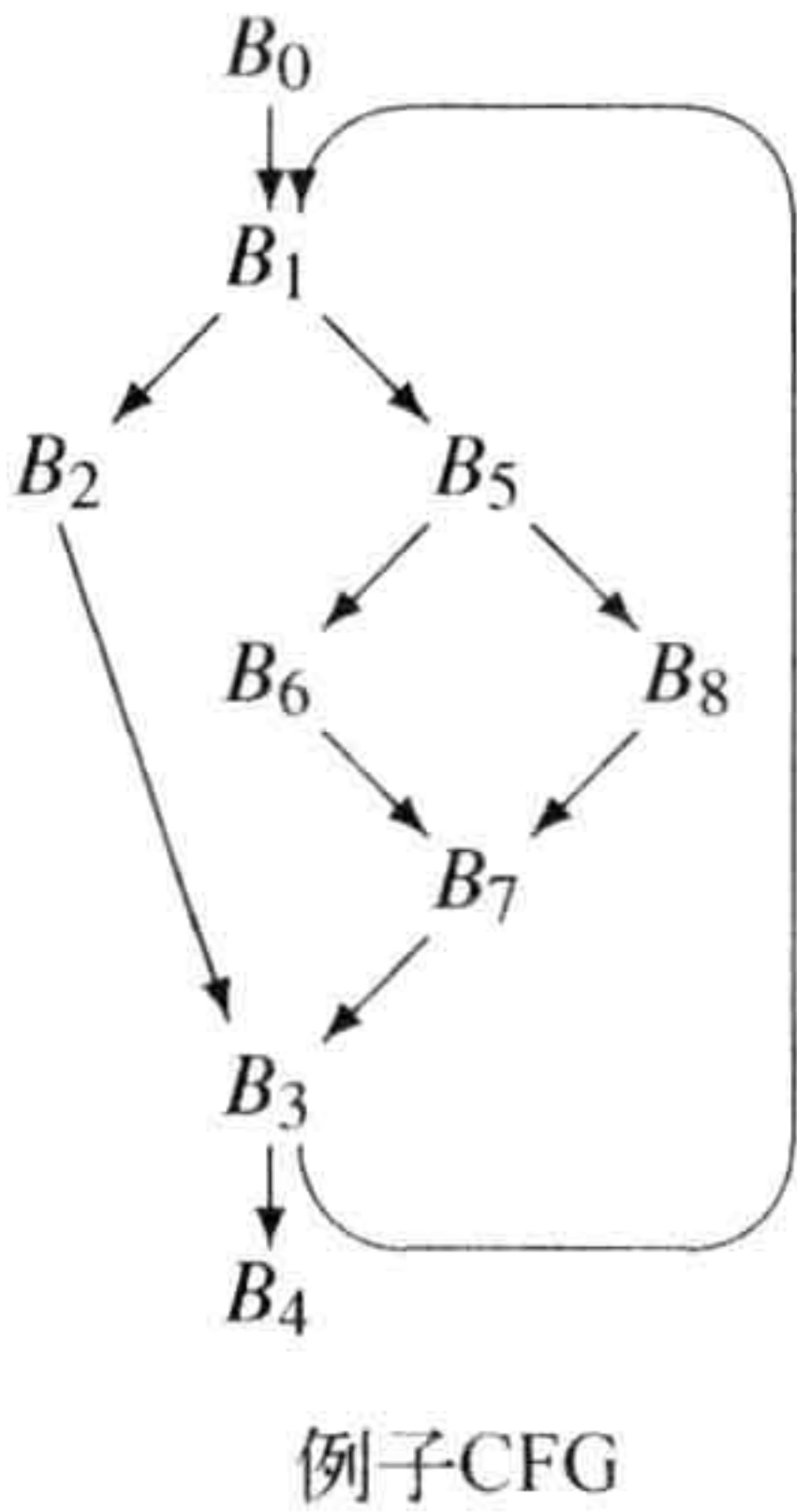
在给出计算支配边界的算法之前，我们必须引入一个更进一步的概念，即支配者树 (dominator tree)。给出流图中的一个结点 n ，严格支配 n 的结点集是 $\text{Dom}(n) - n$ 。该集合中与 n 最接近的结点称为 n 的直接支配结点，记作 $\text{IDom}(n)$ 。流图的入口结点没有直接支配结点。

支配者树
编码了流图支配性信息的树。

流图的支配者树包含流图中的每个结点。该树的边用一种简单的方法编码了 IDom 集合。如果 m 为 $\text{IDom}(n)$ ，那么支配者树中有一条边从 m 指向 n 。我们的例子CFG的支配者树在右边给出。请注意， B_6 、 B_7 和 B_8 都是 B_5 的子结点，尽管 B_7 在CFG中并不是 B_5 的直接后继结点。

支配者树简洁地编码了每个结点的 IDom 信息及其完整的 Dom 集合。给出支配者树中的一个结点 n ， $\text{IDom}(n)$ 只是其在树中的父结点。 $\text{Dom}(n)$ 中的各个结点，就是从支配者树的根结点到 n 之间的路径上的那些结点 (含根结点和 n)。从支配者树，我们可以读取下列集合。

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
Dom	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,3\}$	$\{0,1,3,4\}$	$\{0,1,5\}$	$\{0,1,5,6\}$	$\{0,1,5,7\}$	$\{0,1,5,8\}$
IDom	—	0	1	1	3	1	5	5	5

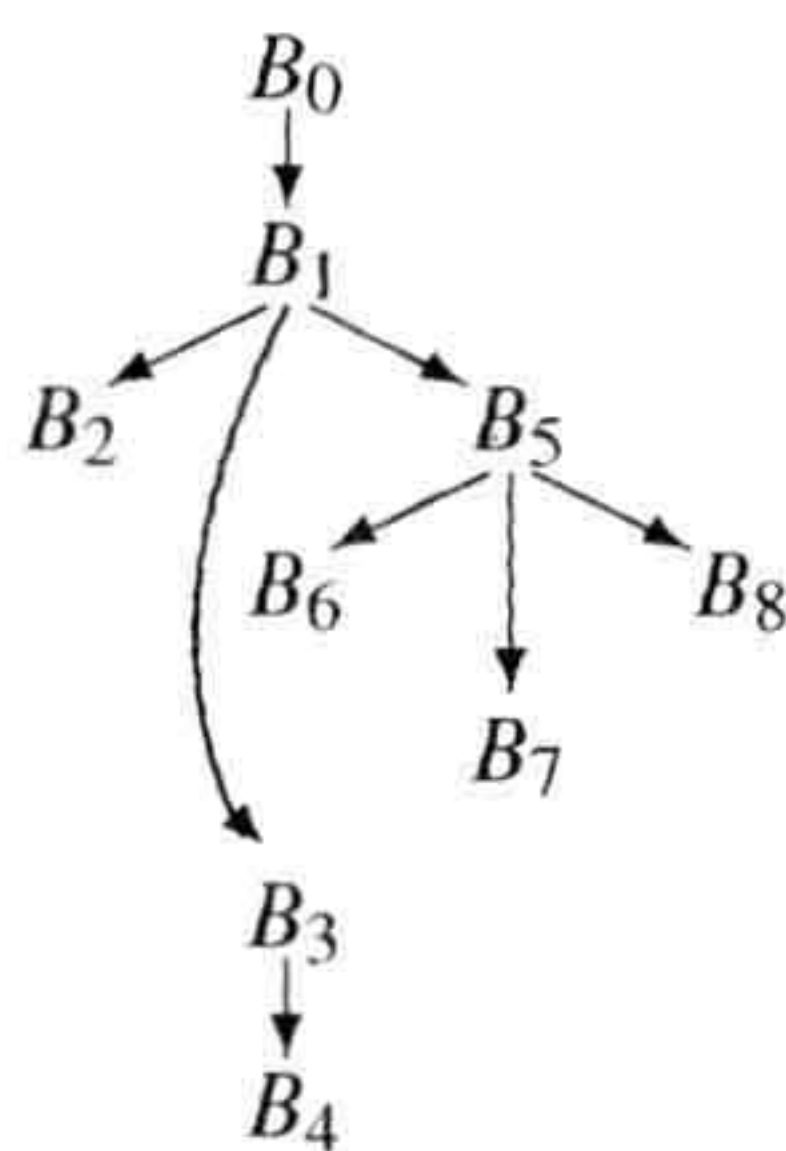


这些Dom集合与早先的计算结果是匹配的，“—”表示未定义值。

2. 计算支配边界

为高效地插入 ϕ 函数，我们需要为流图中的每个结点计算支配边界。我们可以将为图中每个结点 n 计算 $DF(n)$ 的任务表述为一个数据流问题。同时使用支配者树和CFG，我们可以表示出一个简单且直接的算法，如图9-8所示。因为CFG中只有汇合点才是支配边界的成员，我们首先识别出图中的所有汇合点。对于一个汇合点 j ，我们考察其在CFG中的每个前趋结点。

该算法基于三个见解。第一， DF 集合中的结点必定是图中的汇合点。第二，对于一个汇合点 j ， j 的每个前趋结点 k 必定有 $j \in DF(k)$ ，因为如果 j 具有多个前趋结点，则 k 是无法支配 j 的。第三，如果对 j 的某些前趋结点 k ， $j \in DF(k)$ ，那么对每个结点 $l \in Dom(k)$ ，必定有 $j \in DF(l)$ ，除非 $l \in Dom(j)$ 。



其支配者树

```

for all nodes,  $n$ , in the CFG
   $DF(n) \leftarrow \emptyset$ 
for all nodes,  $n$ , in the CFG
  if  $n$  has multiple predecessors then
    for each predecessor  $p$  of  $n$ 
      runner  $\leftarrow p$ 
      while runner  $\neq IDom(n)$ 
         $DF(runner) \leftarrow DF(runner) \cup \{n\}$ 
        runner  $\leftarrow IDom(runner)$ 
  
```

图9-8 计算支配边界的算法

该算法遵循这些见解执行。它会定位CFG中的各个汇合点 j 。接下来，对 j 的每个前趋结点 p ，它会从 p 开始沿支配者树向上走，直至找到支配 j 的一个结点。根据前段的第二和第三个见解，在算法对支配者树的遍历中，除了遍历到的最后一个结点（支配 j ）之外，对其余每个结点 l 都有 j 属于 $DF(l)$ 。这里需要少量簿记工作，以确保任何结点 n 都只添加到某个结点的支配边界一次。

要理解这个算法的工作方式，请再次考虑例子CFG及其支配者树。分析程序会按某种次序考察各个结点，寻找具有多个前趋结点的结点。假定算法按名字的顺序处理各个结点，那么它将按 B_1 、 B_3 、 B_7 的次序找到各个汇合点。

(1) B_1 对于其在CFG中的前趋结点 B_0 ，算法发现 B_0 是 $IDom(B_1)$ ，因此它不会进入while循环。对于其在CFG中的前趋结点 B_3 ，算法将 B_1 添加到 $DF(B_3)$ ，接下来前进到 B_1 。算法将 B_1 添加到 $DF(B_1)$ ，接下来前进到 B_0 ，并停止于 B_0 。

(2) B_3 对于其在CFG中的前趋结点 B_2 ，算法将 B_3 添加到 $DF(B_2)$ ，然后前进到 B_1 ，而 B_1 是 $IDom(B_3)$ ，故停止。对于其在CFG中的前趋结点 B_7 ，算法将 B_3 添加到 $DF(B_7)$ 并前进到 B_5 。然后算法将 B_3 添加到 $DF(B_5)$ 并前进到 B_1 ，停止于此。

(3) B_7 对于其在CFG中的前趋结点 B_6 ，算法将 B_7 添加到 $DF(B_6)$ ，前进到 B_5 ， B_5 为 $IDom(B_7)$ ，故停止。对于其在CFG中的前趋结点 B_8 ，算法将 B_7 添加到 $DF(B_8)$ 并前进到 B_5 ，停止于此。

累积这些结果，我们得到以下支配边界。

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF	$\emptyset$	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	$\emptyset$	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

9.3.3 放置 ϕ 函数

朴素的算法会在每个汇合结点起始处为每个变量放置一个 ϕ 函数。有了支配边界之后，编译器可以更精确地判断何处可能需要 ϕ 函数。其基本思想很简单。基本程序块 b 中对 x 的定义，则要求在 $DF(b)$ 集合包含的每个结点起始处都放置一个对应的 ϕ 函数。因为 ϕ 函数是对 x 的一个新的定义，此处插入的 ϕ 函数进而可能导致插入额外的 ϕ 函数。

编译器可以进一步缩小插入的 ϕ 函数集合。只在单个基本程序块中活动的变量，绝不会出现与之相应的活动 ϕ 函数。为应用这一见解，编译器可以计算跨多个程序块的活动变量名的集合，该集合被称为全局名字（global name）集合。它可以对该集合中的名字插入 ϕ 函数，而忽略不在该集合中的名字。（正是这一约束将半剪枝静态单赋值形式与其他各种静态单赋值形式区分开来。）

“全局”这个词用在此处，意味着我们关注的是跨越整个过程的某些性质。

编译器可以用很小的代价找到全局名字集合。在每个基本程序块中，编译器可以查找具有向上展现用法的名字，即活动变量计算中的UEVar集合。任何出现在一个或多个LiveOut集合中的名字，都必然出现在某个基本程序块中的UEVar集合中。将所有UEVar集合取并集，编译器即可得到在一个或多个基本程序块入口处活动的名字的集合，显然此即活动于多个程序块中的名字的集合。

如图9-9a所示的算法衍生自计算UEVar的显然算法。该算法构造了单个集合Globals，而LiveOut的计算则必须对每个基本程序块分别计算一个不同的集合。随着算法构建Globals集合，它同时也为每个名字构造了一个列表，包含了所有定义该名字的基本程序块。这些程序块列表充当了一个初始化的WorkList，供插入 ϕ 函数的算法使用。

<pre> Globals $\leftarrow \emptyset$ Initialize all the Blocks sets to $\emptyset$ for each block b VARKILL $\leftarrow \emptyset$ for each operation i in b, in order assume that op_i is "$x \leftarrow y \text{ op } z$" if $y \notin \text{VARKILL}$ then Globals $\leftarrow \text{Globals} \cup \{y\}$ if $z \notin \text{VARKILL}$ then Globals $\leftarrow \text{Globals} \cup \{z\}$ VARKILL $\leftarrow \text{VARKILL} \cup \{x\}$ Blocks(x) $\leftarrow \text{Blocks}(x) \cup \{b\}$ (a) 找到全局名字集合 </pre>	<pre> for each name $x \in \text{Globals}$ WorkList $\leftarrow \text{Blocks}(x)$ for each block $b \in \text{WorkList}$ for each block d in $DF(b)$ if d has no ϕ-function for x then insert a ϕ-function for x in d WorkList $\leftarrow \text{WorkList} \cup \{d\}$ (b) 重写代码 </pre>
--	--

图9-9 插入 ϕ 函数

插入 ϕ 函数的算法如图9-9b所示。对于每个全局名字 x ，算法将WorkList初始化为Blocks(x)。对于

WorkList上的每个基本程序块 b ，算法在 b 的支配边界中每个程序块 d 的起始处插入 ϕ 函数。因为根据定义，一个基本程序块中的所有 ϕ 函数都是并发执行的，所以算法可以按任何次序在 d 的起始处插入这些 ϕ 函数。在向 d 添加对应于 x 的 ϕ 函数之后，算法将 d 添加到WorkList，以反映 d 中对 x 的新赋值操作。

1. 示例

图9-10概括了我们一直使用的例子。图9-10a给出了代码，图9-10b显示了CFG，图9-10c给出了每个基本程序块的支配边界，图9-10e给出了根据CFG构建的支配者树。

ϕ 函数插入算法中的第一步是找到全局名字集合并为每个名字计算Blocks集合。对于图9-10a中的代码，全局名字集合是 $\{a, b, c, d, i\}$ 。图9-10d给出了Blocks集合。请注意，算法为 y 和 z 创建了Blocks集合，虽然二者并不在Globals中。将Globals和Blocks的计算分离开来，可以避免实例化这些额外的集合，代价是需要增加一趟对代码的处理。

```

B0: i ← 1
    → B1
B1: a ← ...
    c ← ...
    (a < c) → B2, B5
B2: b ← ...
    c ← ...
    d ← ...
    → B3
B3: y ← a + b
    z ← c + d
    i ← i + 1
    (i ≤ 100) → B1, B4

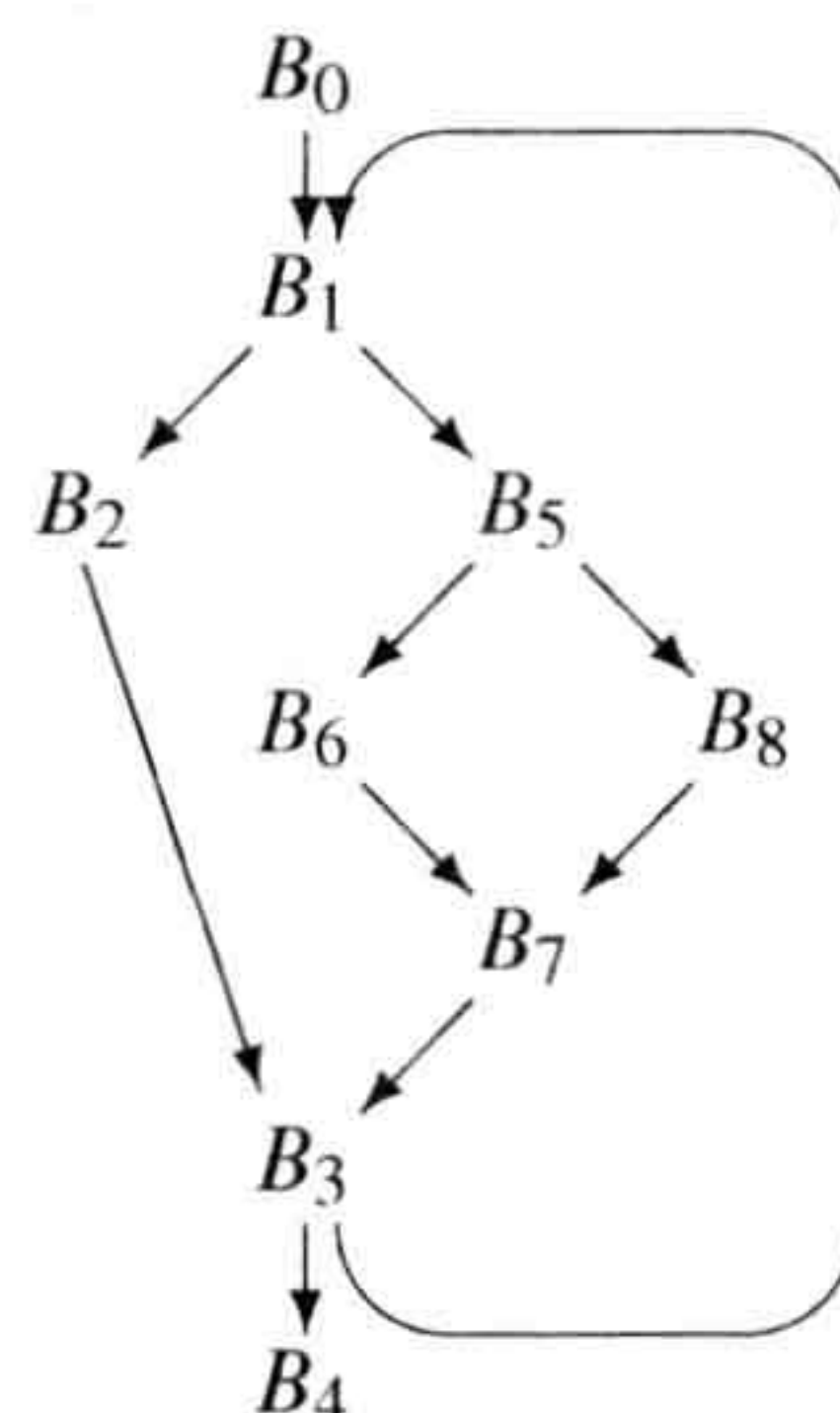
```

```

B4: return
B5: a ← ...
    d ← ...
    (a ≤ d) → B6, B8
B6: d ← ...
    → B7
B7: b ← ...
    → B3
B8: c ← ...
    → B7

```

(a) 各个基本程序块的代码



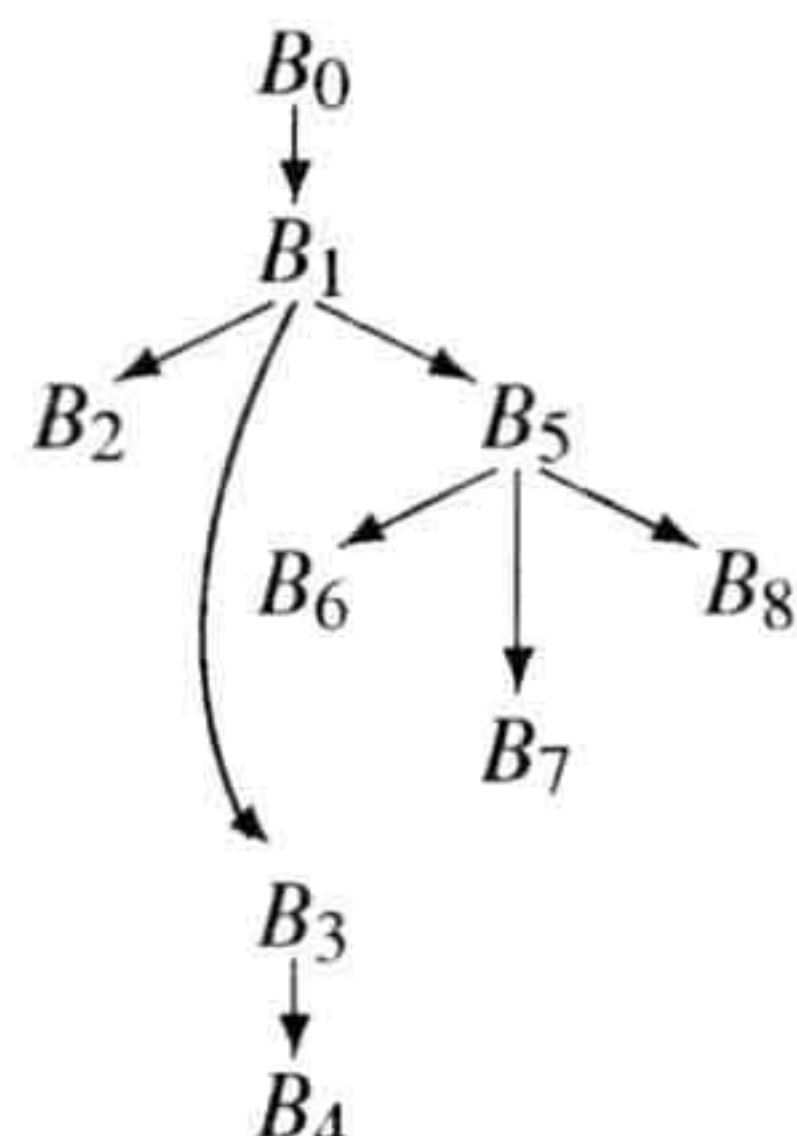
(b) 控制流图

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF	$\emptyset$	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	$\emptyset$	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

(c) CFG中的支配边界

	a	b	c	d	i	y	z
Blocks	$\{1,5\}$	$\{2,7\}$	$\{1,2,8\}$	$\{2,5,6\}$	$\{0,3\}$	$\{3\}$	$\{3\}$

(d) 每个名字的Blocks集合



(e) 支配者树

图9-10 用于插入 ϕ 函数的例子静态单赋值形式

ϕ 函数重写算法逐个处理各个名字。考虑其对例子中变量 a 的操作。它会将WorkList初始化为 $\text{Blocks}(a)$ ，其中包含 B_1 和 B_5 。 B_1 中的定义使得算法在 $\text{DF}(B_1) = \{B_1\}$ 中的各个基本程序块起始处插入一个 ϕ 函数。该操作也将 B_1 加入到WorkList中。接下来，它从WorkList删除 B_5 ，并在 $\text{DF}(B_5) = \{B_3\}$ 中的每个基本程序块起始处插入一个 ϕ 函数。在程序块 B_3 处的插入操作也会将 B_3 置于WorkList上。在算法将 B_3 从WorkList移除时，它试图将一个 ϕ 函数添加到 B_1 起始处，因为 $B_1 \in \text{DF}(B_3)$ 。算法注意到 B_1 已经有了相应的 ϕ 函数，因此不会执行插入。因而，对 a 的处理停止下来，此时WorkList为空。算法对Globals中的每个名字遵循相同的逻辑，如此会产生以下插入操作。

	a	b	c	d	i
ϕ 函数	$\{B_1, B_3\}$	$\{B_1, B_3\}$	$\{B_1, B_3, B_7\}$	$\{B_1, B_3, B_7\}$	$\{B_1\}$

由此生成的代码如图9-11所示。

```

B0: i ← 1
      → B1
B1: a ←  $\phi(a, a)$ 
      b ←  $\phi(b, b)$ 
      c ←  $\phi(c, c)$ 
      d ←  $\phi(d, d)$ 
      i ←  $\phi(i, i)$ 
      a ← ...
      c ← ...
      (a < c) → B2, B5
B2: b ← ...
      c ← ...
      d ← ...
      → B3
B3: a ←  $\phi(a, a)$ 
      b ←  $\phi(b, b)$ 
      c ←  $\phi(c, c)$ 
      d ←  $\phi(d, d)$ 
      y ← a + b
      z ← c + d
      i ← i + 1
      (i ≤ 100) → B1, B4
B4: return
B5: a ← ...
      d ← ...
      (a ≤ d) → B6, B8
B6: d ← ...
      → B7
B7: c ←  $\phi(c, c)$ 
      d ←  $\phi(d, d)$ 
      b ← ...
      → B3
B8: c ← ...
      → B7

```

图9-11 加入了 ϕ 函数、尚未进行重命名的示例代码

将算法的处理限于全局名字集合，使其避免了对基本程序块 B_1 中的 x 和 y 插入“死亡”的 ϕ 函数。（ $B_1 \in DF(B_3)$ ， B_3 包含了对 x 和 y 的定义。）但局部名字和全局名字之间的区分不足以避免所有的“死亡” ϕ 函数。例如， B_1 中 b 的 ϕ 函数是不活动的，因为在使用 b 值之前重新定义了 b 。为避免插入这些 ϕ 函数，编译器可以构建LiveOut集合，并在插入 ϕ 函数之算法的内层循环中增加一个对变量活动性的条件判断。这一改动使得算法可以产生剪枝静态单赋值形式。

不同风格的静态单赋值形式

在文献中已经提议了几种不同风格的静态单赋值形式。这些风格的差别在于其插入 ϕ 函数的条件。对于一个给定的程序，这些不同风格的算法可以生成不同的 ϕ 函数集合。

最小静态单赋值形式 (minimal SSA) 在任何汇合点处插入一个 ϕ 函数，只要对应于同一原始名字的两个不同定义会合。这将插入符合静态单赋值形式定义、数目最少的 ϕ 函数。但其中一些 ϕ 函数可能是死亡的，定义并没有规定值在会合时一定是活动的。

剪枝静态单赋值形式 (pruned SSA) 向 ϕ 函数插入算法添加一个活动性判断，以避免添加死亡的 ϕ 函数。构造过程必须计算LiveOut集合，因此构建剪枝静态单赋值形式的代价高于构建最小静态单赋值形式。

半剪枝静态单赋值形式 (semipruned SSA) 是最小静态单赋值形式和剪枝静态单赋值形式之间的一种折中。在插入 ϕ 函数之前，算法先删除非跨越基本程序块边界活动的任何名字。这可以缩减名字空间并减少 ϕ 函数的数目，而又没有计算LiveOut集合的开销。这就是图9-9中给出的算法。

当然， ϕ 函数的数目取决于转换为静态单赋值形式的具体程序。对于一些程序来说，半剪枝静态单赋值形式和剪枝静态单赋值形式减少的 ϕ 函数数目很可观。缩减静态单赋值形式的规模可以使

编译过程更为快速，因为使用静态单赋值形式的那些趟处理过程可以运作在包含的操作数目较少、 ϕ 函数也较少的程序之上。

2. 效率改进

为提高效率，编译器应该避免两种类型的复制。首先，对每个全局名字算法都应该避免将任何基本程序块多次放置到WorkList上。它可以维护一个已经处理的基本程序块的清单。由于算法必须对每个全局名字重置该清单，实现应该使用一种稀疏集或类似的结构（参见B.2.3节）。

其次，一个给定的基本程序块可能出现在WorkList上多个结点的支配边界中。如图9-11所示，算法必须查找这样的基本程序块，以寻找此前已存在的 ϕ 函数。为避免这种查找，对指定变量（例如 x ）来说，编译器可以维护一个基本程序块的清单，列出已包含针对该变量 ϕ 函数的基本程序块。这需要采用一个稀疏集，针对需要处理的每个全局名字，与WorkList一同重新初始化。

9.3.4 重命名

在最大静态单赋值形式的描述中，我们指出重命名变量在概念上很简单。但仍然需要对其中的细节作一些解释。

在最终的静态单赋值形式中，每个全局名字都变为一个基本名，而对该基本名的各个定义则通过添加数字下标来区分。对于对应到源语言变量的名字，比如说 x ，算法使用 x 作为基本名。因而，重命名算法遇到的对 x 的第一个定义将被命名为 x_0 ，第二个将被命名为 x_1 。对于编译器产生的临时值，算法必须产生一个不同的基本名。

该算法如图9-12所示，对过程的支配者树进行了先根次序遍历，其中对定义和使用都进行了重命名。在每个基本程序块中，算法首先重命名由程序块顶部的 ϕ 函数定义的值，然后按序访问程序块中的各个操作。算法会用当前的静态单赋值形式名重写各个操作数，接下来为操作的结果创建一个新的静态单赋值形式名。算法的后一步使得新名字成为当前的名字。在程序块中所有的操作都已经重写之后，算法将使用当前的静态单赋值形式名重写程序块在CFG中各后继结点中的适当 ϕ 函数参数。最后，算法对当前程序块在支配者树中的子结点进行递归处理。当算法从这些递归调用返回时，它会将当前静态单赋值形式名的集合恢复到访问当前程序块之前的状态。

为管理这一处理过程，算法对每个全局名字使用一个计数器和一个栈。全局名字的栈包含了该名字当前静态单赋值形式的下标。在每个定义处，算法通过将目标名字的当前计数器值压栈来产生新的下标，并将计数器加1。因而，名字 n 栈顶的值总是 n 当前静态单赋值形式名的下标。作为处理程序块的最后一步，算法会将该程序块中产生的所有名字从栈中弹出，以恢复在该程序块的直接支配结点末尾处的当前静态单赋值形式名字集合。处理当前程序块在支配者树中余下的兄弟结点，可能需要这些名字。

栈和计数器服务于不同且分离的目的。当算法中的控制流在支配者树中上下移动时，栈模拟了当前程序块中最新定义的生命周期。而在另一方面，计数器则是单调递增的，以确保各个连续的定义都能分配一个唯一的静态单赋值形式名。

图9-12总结了该算法。该算法初始化了栈和计数器，然后对支配者树的根结点（CFG的入口结点）调用Rename。Rename会重写该程序块，并下降到其在支配者树的各个后继结点上递归处理。为完成对该程序块的处理，Rename会弹出处理该程序块期间压栈的任何名字。函数NewName会操纵计数器和栈，

以按需创建新的静态单赋值形式名。

<pre> for each global name i counter[i] ← 0 stack[i] ← ∅ Rename(n_0) NewName(n) $i \leftarrow \text{counter}[n]$ $\text{counter}[n] \leftarrow \text{counter}[n] + 1$ push i onto $\text{stack}[n]$ return "n_i" </pre>	<pre> Rename(b) for each ϕ-function in b, "$x \leftarrow \phi(\dots)$" rewrite x as $\text{NewName}(x)$ for each operation "$x \leftarrow y \text{ op } z$" in b rewrite y with subscript $\text{top}(\text{stack}[y])$ rewrite z with subscript $\text{top}(\text{stack}[z])$ rewrite x as $\text{NewName}(x)$ for each successor of b in the CFG fill in ϕ-function parameters for each successor s of b in the dominator tree Rename(s) for each operation "$x \leftarrow y \text{ op } z$" in b and each ϕ-function "$x \leftarrow \phi(\dots)$" pop($\text{stack}[x]$) </pre>
---	---

图9-12 插入 ϕ 函数之后的重命名

还有最后一个细节。在程序块 b 末尾处，Rename 必须重写 b 在 CFG 中的各个后继结点中 ϕ 函数的参数。编译器必须在这些 ϕ 函数中为 b 按序分配一个参数槽位。在绘制静态单赋值形式时，我们总是假定从左到右的次序，以便匹配从左到右绘制边的次序。但在内部，编译器可以按任何一致的方式对边和参数槽位编号，以产生所需的结果。这要求构建静态单赋值形式的代码与构建 CFG 的代码之间的协作。（例如，如果 CFG 实现使用边的列表表示离开每个程序块的各条边，那么该列表本身蕴涵的次序就决定了这种映射关系。）

1. 示例

为完成对前述例子的处理，我们将重命名算法应用到图9-11中的代码中。假定 a_0 、 b_0 、 c_0 和 d_0 是在进入 B_0 时定义的。图9-13给出了全局名字集合中各个名字的计数器和栈在重命名处理期间各个时间点上的状态。

算法对支配者树进行了一趟先根次序遍历，这对应于按名字的递增次序访问各个结点，从 B_0 到 B_8 。各个栈和计数器的初始配置如图9-13a所示。随着算法逐步处理各个程序块，它需要进行下列操作。

- 程序块 B_0 该程序块只包含一个操作。Rename 会将 i 重写为 i_0 ，将计数器加1，并将 i_0 压入 i 的栈中。接下来，算法将访问 B_0 在 CFG 中的后继结点 B_1 ，并将与 B_0 对应的 ϕ 函数参数重写为其当前名字： a_0 、 b_0 、 c_0 、 d_0 和 i_0 。接下来，算法递归到 B_0 在支配者树中的子结点 B_1 。处理完 B_1 之后，算法将弹出 i 的栈并返回。
- 程序块 B_1 Rename 进入 B_1 时的状态如图9-13b所示。算法会将 ϕ 函数的目标重写为新的名字： a_1 、 b_1 、 c_1 、 d_1 和 i_1 。接下来，算法为 a 和 c 的定义创建新名字，并重写它们。它还会重写比较操作中使用的 a 和 c 。 B_1 在 CFG 的两个后继结点都没有 ϕ 函数，因此算法将递归到 B_1 在支配者树中的子结点 B_2 、 B_3 和 B_5 。处理完这些子结点，算法会弹出处理 B_1 期间压栈的数据并返回。
- 程序块 B_2 Rename 进入 B_2 时的状态如图9-13c所示。该程序块没有需要重写的 ϕ 函数。Rename 重写了 b 、 c 和 d 的定义，并为其分别创建了一个新的静态单赋值形式名。算法接下来重写 B_2 在 CFG 中后继结点 B_3 中的 ϕ 函数的参数。图9-13d给出了在弹栈之前的栈和计数器。最后，算法

会弹栈并返回。

- 程序块 B_3 Rename 进入 B_3 时的状态如图 9-13e 所示。请注意，此时栈已经恢复到 Rename 进入 B_2 时的状态，但计数器仍然反映出算法曾经在 B_2 内部创建的新名字。在 B_3 中，Rename 会重写 ϕ 函数的目标，并为这些目标分别创建新的静态单赋值形式名。接着，算法重写程序块中的各个赋值操作，将使用替换为当前的静态单赋值形式名，将定义替换为创建的新静态单赋值形式名。（因为 y 和 z 不是全局名字，算法留下二者原封不动。）

	a	b	c	d	i
计数器	1	1	1	1	0
栈	a ₀	b ₀	c ₀	d ₀	

(a) 初始条件，在进入 B_0 时

	a	b	c	d	i
计数器	3	2	3	2	2
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂		

(c) 在进入 B_2 时

	a	b	c	d	i
计数器	3	3	4	3	2
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂		

(e) 在进入 B_3 时

	a	b	c	d	i
计数器	4	4	5	4	3
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂		

(g) 在进入 B_5 时

	a	b	c	d	i
计数器	5	4	5	6	3
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂	d ₄	
	a ₄				

(i) 在进入 B_7 时

	a	b	c	d	i
计数器	1	1	1	1	1
栈	a ₀	b ₀	c ₀	d ₀	i ₀

(b) 在进入 B_1 时

	a	b	c	d	i
计数器	3	3	4	3	2
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂	b ₂	c ₂	d ₂	
			c ₃		

(d) 在 B_2 结束处

	a	b	c	d	i
计数器	4	4	5	4	3
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂	b ₃	c ₂	d ₃	i ₂
	a ₃		c ₄		

(f) 在 B_3 结束处

	a	b	c	d	i
计数器	5	4	5	5	3
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂	d ₄	
	a ₄				

(h) 在进入 B_6 时

	a	b	c	d	i
计数器	5	5	6	7	32
栈	a ₀	b ₀	c ₀	d ₀	i ₀
	a ₁	b ₁	c ₁	d ₁	i ₁
	a ₂		c ₂	d ₄	
	a ₄				

(j) 在进入 B_8 时

图 9-13 重命名例子的各个状态

B_3 在CFG中有两个后继结点 B_1 和 B_4 。在 B_1 中,算法使用如图9-13f所示的栈和计数器,重写了与来自 B_3 的边对应的 ϕ 函数参数。 B_4 没有 ϕ 函数。接下来, Rename递归到 B_3 在支配者树中的子结点 B_4 。在该调用返回时, Rename将弹栈并返回。

- 程序块 B_4 该程序块只包含一条返回语句。其中没有 ϕ 函数、定义、使用,而其在CFG或支配者树中也没有后继结点。因而, Rename不执行任何操作,不会改变栈和计数器。
- 程序块 B_5 在处理 B_4 之后, Rename将弹出 B_3 期间的压栈数据,返回到 B_1 末尾处的状态。此时的栈如图9-13g所示,算法将递归到 B_1 在支配者树中的最后一个子结点 B_5 。 B_5 没有 ϕ 函数。Rename会重写两个赋值语句和条件语句中的表达式,并按需创建新的静态单赋值形式名。 B_5 在CFG中的两个后继结点都没有 ϕ 函数。Rename接下来递归到 B_5 在支配者树中的子结点 B_6 、 B_7 和 B_8 。最后,算法会弹栈并返回。
- 程序块 B_6 Rename进入 B_6 时的状态如图9-13h所示。 B_6 没有 ϕ 函数。Rename将重写对 d 的赋值,产生新的静态单赋值形式名 d_5 。接着,算法访问 B_6 在CFG中的后继结点 B_7 中的 ϕ 函数。它会将对应于来自 B_6 的代码路径的 ϕ 函数参数重写为当前的名字 c_2 和 d_5 。由于 B_6 在支配者树中没有子结点,它将对 d 弹栈并返回。
- 程序块 B_7 Rename进入 B_7 时的状态如图9-13i所示。算法首先用新的静态单赋值形式名 c_5 和 d_6 重命名 ϕ 函数的目标。接下来,它用新的静态单赋值形式名 b_4 重写对 b 的赋值操作。然后,算法用当前静态单赋值形式名的集合,来重写 B_7 在CFG中后继结点 B_3 中 ϕ 函数的参数。由于 B_7 在支配者树中没有子结点,算法将弹栈并返回。
- 程序块 B_8 Rename进入 B_8 时的状态如图9-13j所示。 B_8 没有 ϕ 函数。Rename将用新的静态单赋值形式名 c_6 重写对 c 的赋值操作。算法会考察 B_8 在CFG中的后继结点 B_7 ,并将对应的 ϕ 函数参数重写为其当前的静态单赋值形式名 c_6 和 d_4 。由于 B_8 在支配者树中没有子结点,算法将弹栈并返回。

图9-14给出了Rename停止之后的示例代码。

<pre> B0: i0 ← 1 → B1 B1: a1 ← φ(a0, a3) b1 ← φ(b0, b3) c1 ← φ(c0, c4) d1 ← φ(d0, d3) i1 ← φ(i0, i2) a2 ← ... c2 ← ... (a2 < c2) → B2, B5 B2: b2 ← ... c3 ← ... d2 ← ... → B3 </pre>	<pre> B3: a3 ← φ(a2, a4) b3 ← φ(b2, b4) c4 ← φ(c3, c5) d3 ← φ(d2, d6) y ← a3 + b3 z ← c4 + d3 i2 ← i1 + 1 (i2 ≤ 100) → B1, B4 B4: return B5: a4 ← ... d4 ← ... (a4 ≤ d4) → B6, B8 </pre>	<pre> B6: d5 ← ... → B7 B7: c5 ← φ(c2, c6) d6 ← φ(d5, d4) b4 ← ... → B3 B8: c6 ← ... → B7 </pre>
---	--	--

图9-14 重命名之后的示例代码

2. 最终改进

对NewName的精巧实现可以减少栈操作花费的时间和空间。此中对栈的主要使用是在从一个基本程序

块退出时重置名字空间。如果一个基本程序块重新定义了同一基本名几次, *newName* 只需维护最新的名字, 对于例子中程序块 B_1 中的 a 和 c 就是如此。 *newName* 可能会在单个程序块内部多次重写栈中的同一槽位。

这使得栈的最大长度变得可预测, 栈不可能比支配者树的深度更长。这样做降低了整体空间需求, 避免了在每次压栈时判断溢出, 也减少了压栈和弹栈操作的次数。它需要另一种机制以判断从一个基本程序块退出时对哪些栈进行弹栈操作。 *newName* 可以将一个基本程序块涉及的各个栈的入口串联起来。 *Rename* 可以使用这一串联信息来弹出适当的栈。

9.3.5 从静态单赋值形式到其他形式的转换

因为现代处理器没有实现 ϕ 函数, 编译器需要将静态单赋值形式转换回可执行代码。如果依据现存的各个例子, 很容易让读者误认为: 编译器只需去掉静态单赋值形式名字的下标、恢复基本名并删除 ϕ 函数, 即可完成这样的反向转换。如果编译器只是构建静态单赋值形式然后将其转回可执行代码, 这种方法是可行的。但如果代码已经被重排或值已经重命名过, 这种方法可能会产生不正确的代码。

举例来说, 我们在8.4.1节看到: 使用静态单赋值形式名使局部值编号算法 (LVN) 能够发现并删除更多的冗余。

执行LVN算法前	执行LVN算法后	执行LVN算法前	执行LVN算法后
$a \leftarrow x + y$	$a \leftarrow x + y$	$a_0 \leftarrow x_0 + y_0$	$a_0 \leftarrow x_0 + y_0$
$b \leftarrow x + y$	$b \leftarrow a$	$b_0 \leftarrow x_0 + y_0$	$b_0 \leftarrow a_0$
$a \leftarrow 17$	$a \leftarrow 17$	$a_1 \leftarrow 17$	$a_1 \leftarrow 17$
$c \leftarrow x + y$	$c \leftarrow x + y$	$c_0 \leftarrow x_0 + y_0$	$c_0 \leftarrow a_0$

原始名字空间

SSA名字空间

左表给出了一个包含4个操作的基本程序块, 以及使用代码自身的名字空间时LVN算法产生的结果。右表给出的是同一个例子, 但使用了SSA名字空间。因为SSA名字空间给 a_0 赋予了一个不同于 a_1 的名字, LVN算法可以将最后一个操作中对 $x_0 + y_0$ 的求值替换为对 a_0 的引用。

但请注意, 只是简单地去掉变量名的下标将产生不正确的代码, 因为这将使得 c 被赋值为17 (与原来的语义不同)。更激进的变换, 如代码移动和复制折叠 (copy folding), 它们重写静态单赋值形式的方法可能会引入更多微妙的问题。

为避免这样的问题, 编译器可以保持SSA名字空间原样不动, 将每个 ϕ 函数替换为一组复制操作 (每个复制操作对应于一条进入当前程序块的边)。对于 ϕ 函数 $x_i \leftarrow \phi(x_j, x_k)$, 编译器应该沿传入 x_j 的边插入 $x_i \leftarrow x_j$, 沿传入 x_k 的边插入 $x_i \leftarrow x_k$ 。

图9-15给出将示例代码中的 ϕ 函数替换为复制操作之后的情形。 B_3 中的4个 ϕ 函数已经被替换为 B_2 和 B_7 中各自添加的一组四个的复制操作。类似地, B_7 中的两个 ϕ 函数也使得 B_6 和 B_8 中分别添加了两个复制操作。在上述两种情况下, 编译器可以将复制操作插入到前趋程序块中。

B_1 中的 ϕ 函数则展现了一种更复杂的情形。编译器可以在其前趋结点 B_0 中直接插入复制操作, 却不能对其前趋结点 B_3 这样做。因为 B_3 有多个后继结点, 在 B_3 末尾处对 B_1 中的 ϕ 函数插入复制操作, 将导致这些操作也会在 B_3 到 B_4 的代码路径上执行, 而这些操作在这种情况下是不必要的且可能导致不正确的结果。为弥补这种问题, 编译器可以拆分边 (B_3, B_1) , 在 B_3 和 B_1 之间插入一个新程序块, 将复制操