第 2 章

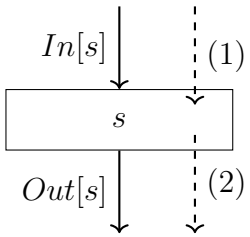
数据流分析

数据流分析是一种重要的程序分析技术，它静态分析程序中值的使用和传递情况，并指导后续的程序优化。在本章，我们将讨论数据流分析的基本概念并阐述其在编译器中的作用；然后，我们将讨论多种数据流分析技术，包括支配节点、活跃分析、可用表达式、忙碌表达式和到达定值；最后，我们将讨论数据流分析方法的改进与优化，以及对不同分析算法进行比较，为研究数据流分析技术的一般理论奠定基础。

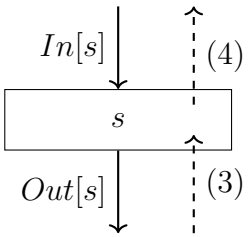
2.1 引言

数据流分析是一种重要的静态程序分析技术，它通过对程序中变量和表达式的数据及其流动进行系统分析，得到程序关于数据定义和使用的精确信息，为程序优化和其它程序分析提供支持。

数据流分析在编译器、软件安全、软件工程等相关领域都发挥着重要作用。首先，在编译器领域，数据流分析常用于代码优化。例如，数据流分析能够识别并优化常量表达式、冗余计算、死代码和进行其他改进，从而提升程序的执行效率。



(a) 从语句 s 前的数据流信息 $In[s]$，顺着控制流的方向，计算得到语句 s 后的的数据流信息 $Out[s]$



(b) 从语句 s 后的数据流信息 $Out[s]$，逆着控制流的方向，计算得到语句 s 前的的数据流信息 $In[s]$

图 2.1: 数据流信息的前向和后向的计算与传播过程。其中，实线代表程序的控制流执行方向，虚线代表数据流信息的计算方向。符号 $l[s]$、$In[s]$ 和 $Out[s]$ 分别代表语句 s 本身的数据流信息、以及语句前后的数据流信息

其次，在软件安全领域，数据流分析能够有效提升软件安全性。例如，数据流分析能够分析指针使用信息，从而发现空指针引用或指针“释放后使用”等安全问题，避免产生安全漏洞。最后，数据流分析能够给软件工程提供支撑。例如，在软件开发过程中，数据流分析能够提供变量的“定义-使用”信息、函数的调用点信息等，支持更有效的软件开发过程。

数据流分析的核心过程分为三步。首先，数据流分析严格刻画控制流图节点（或边）上的数据流信息。其次，通过数据流方程，数据流分析由节点或边上的数据流信息，得到局部数据流信息。最后，数据流分析通过迭代过程，把局部数据流信息，传播到整个控制流图上，从而得到控制流图上所有程序点的全局数据流信息。图2.1给出了数据流信息在控制流图上进行计算和传播的过程。首先，如图2.1a所示，对给定的语句 s，我们假设 s 的数据流信息为 $l[s]$，一般的，该信息的具体内容取决于正在进行的具体数据流分析，并且，该信息可以从正在被分析的程序代码中得到。

其次，我们假设语句 s 前后的数据流信息分别为 $In[s]$ 和 $Out[s]$，则我们可以按方程

$$Out[s] = f(In[s], l[s]) \tag{2.1}$$

沿着控制流执行的方向，从语句 s 前的数据流信息 $In[s]$ 和语句信息 $l[s]$，计算得到语句 s 后的数据流信息 $Out[s]$，亦即图2.1a 中的虚线 (1) → (2)。我们可称这种方向的数据流信息计算为**前向传播**（Forward propagation），其中方程2.1称为数据流方程，方程中的函数 f 称为**转移函数**（Transfer function），它表达了从语句 s 的数据流输入信息 $In[s]$ 和语句信息 $l[s]$，计算输出信息 $Out[s]$ 的具体规则。需要注意的是，语句信息 $l[s]$、数据流方程、和转移函数 f 等，都和具体的数据流分析算法相关。

最后，我们需要把每个语句的局部数据流信息 $In[s]$ 和 $Out[s]$ 传播到所有的程序点上。为此，我们需要考虑语句 s 在控制流图中的前驱和后继节点。在图2.1a中，假设语句 s 的前驱节点是 $p_1, \ldots, p_n$，则我们可以按如下方程

$$In[s] = \bigvee (Out[p_1], \ldots, Out[p_n]) \tag{2.2}$$

计算语句 s 前的数据流信息，其中 $\bigvee$ 是和具体数据流分析算法相关的函数，它接受 n 个输入 $Out[p_1], \ldots, Out[p_n]$，并将计算得到的结果赋值给左侧的 $In[s]$。

同理，图2.1b给出了数据流分析**后向传播**（Backward propagation）的过程（其中的虚线 (3) → (4)）。该过程按照方程

$$In[s] = f(Out[s], l[s])$$

从语句 s 后的数据流信息 $Out[s]$ 和语句局部信息 $l[s]$，计算得到语句 s 前的数据流信息 $In[s]$。对该方程的详细分析与前向传播的过程类似，我们将其作为练习，留给读者完成。

我们假设语句 s 的后继为 $s_1, \ldots, s_n$，则我们可按方程

$$Out[s] = \bigvee (In[s_1], \ldots, In[s_n]) \tag{2.3}$$

由后继 $s_1, \ldots, s_n$ 的 $In[]$ 计算得到语句 s 的 $Out[]$ 信息，最终计算得到全局数据流信息。

本章将围绕上述数据流分析的一般过程，具体讨论几类重要的数据流分析算法，从而为后续章节讨论数据流分析的一般理论和框架奠定基础。本章的讨论按照不同的分析粒度展开，首先介绍支配节点分析（基本块粒度）和活跃分析（变量粒度），随后介绍可用表达式和忙碌表达式分析（表达式粒度），以及达定值分析（语句粒度）。最后，本章还将讨论对数据流分析的重要改进与优化方法，并对不同数据流分析技术进行比较。

2.2 支配节点

复杂控制流图包括循环等控制结构，使得理解程序的结构和进行数据流分析变得困难。为此，我们引入**支配节点**（Dominator）的概念。在控制流图中，我们称 d 是 n 的支配节点（或称节点 d 支配节点 n），如果从控制流图的入口块 s_0 出发，任何到达 n 的路径都必须经过 d。显然，每个节点平凡的支配它自己。我们用 $Dom[n]$ 来表示节点 n 的所有支配节点的集合。

作为示例，我们考虑图2.2中的控制流图。节点 B_6 的支配节点包括 B_0、B_1、B_5 和 B_6，因为从入口节点 B_0 到 B_6 的所有路径都必须经过这些节点。因此，B_6 的支配节点集合是

$$Dom[B_6] = \{B_0, B_1, B_5, B_6\}。$$

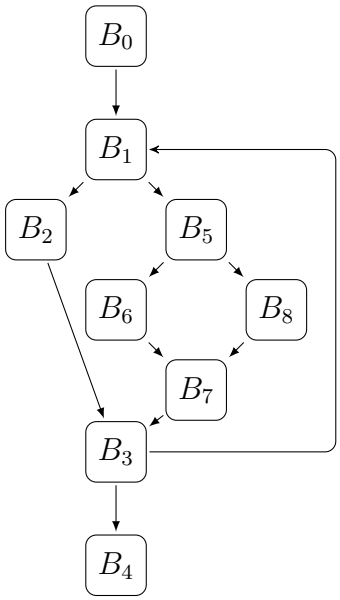


图 2.2: 控制流图示例

表 2.1: 图2.2中各个节点的支配节点集合

\	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
$Dom[n]$	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

类似的，我们可以计算图2.2中其它节点的支配节点集合，最终的结果在表2.1中列出。

假设有一个节点 n，它的前驱节点是 $p_1, \ldots, p_k$。而节点 d（且 $d \neq n$）支配 n 的每一个前驱节点 p_i，$1 \leq i \leq p$，那么 d 一定也支配 n。这是因为从起始节点 s_0 到达 n 的所有路径都必须经过其中一个前驱节点 p_i，而从 s_0 到任意 p_i 的路径都需要经过 d。反过来，如果 d 支配 n，那么 d 也必须支配所有的前驱节点 p_i。否则，就会存在一条从 n_0 到 n 的路径，它通过一个没有被 d 支配的前驱节点，这与 d 支配 n 的假设相矛盾。因此，我们可以使用方程

$$Dom[n] = \{n\} \bigcup \left(\bigcap_{m \in pred[n]} Dom[m] \right) \tag{2.4}$$

算法 2.1 支配节点迭代计算算法

输入： 程序控制流图 G

输出： 每个节点的支配节点集合

```
1: function ComputeDominators(G)
2:     for each node n ∈ G do
3:         Dom[n] = N
4:     Dom[s_0] = {s_0}
5:     while any Dom[n] changed do
6:         for each node n ∈ G do
7:             Dom[n] = {n} ∪ (⋂_{i∈pred[n]} Dom[i])
```

来计算支配节点集合 $Dom[n]$。其中，$pred[n]$ 是节点 n 的前驱节点集合。

算法2.1给出了通过迭代，计算控制流图中每个节点支配节点集合的过程。首先，算法将每个节点 n 的支配节点集合 $Dom[n]$ 初始化为 N，其中 N 是控制流图中的所有节点的全集。算法将入口节点 s_0 的支配节点集合 $Dom[s_0]$ 初始化为集合 $\{s_0\}$，即控制流图入口节点 s_0 的支配节点集合只有自身（且不会在计算中改变）。接着，算法利用方程2.4，不断更新每个节点 n 的支配节点集合 $Dom[n]$，直到所有支配节点集合都不再变化为止。

为分析算法的最坏时间复杂度，我们首先考虑算法的最内层循环（即第 6 到 7 行）。设图中的每个节点 n 平均有 m 个前驱节点，即 $|pred[n] = m|$，且假设两个集合的交集运算 $\cap$ 的运行时间和集合元素个数 N 成线性关系，即时间复杂度为 $O(N)$，则执行 m 次交集计算的时间复杂度为 $O(mN)$（第 7 行），那么，第 6 到 7 行的循环总的执行时间复杂度为 $O(mN^2)$。再考虑算法最外层的循环（第 5 行），由于在算法迭代结束前，至少有一个支配节点集合 $Dom[n]$ 发生了变化，而每个支配

表 2.2: 支配节点集合迭代计算的过程

\	$Dom[n]$								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
–	0	N	N	N	N	N	N	N	N
1	0	0,1	0,1,2	0,1,2,3	0,1,2,3,4	0,1,5	0,1,5,6	0,1,5,6,7	0,1,5,8
2	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
3	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

节点集合最多变化 N 次，则循环次数最多可能为 N^2 次。综上，算法的最坏运行时间复杂度为 $O(m * N^4)$。在典型的控制流图中，节点的前驱节点数量 m 一般是个不依赖于节点总数 N 的小常数（典型的，$m \leq 3$），则整个算法总时间复杂度为 $O(N^4)$。

作为示例，对图2.2中的控制流图，应用迭代算法计算其支配节点集合的过程如表2.2所示。在初始化后，除了 B_0 节点外，其它节点的支配节点集合 $Dom[n]$ 都被初始化为全集 N。计算按照从 B_0 到 B_8 的顺序依次进行，亦即表的每一行都按照从左到右的顺序进行计算。以节点 $B[3]$ 为例，在第一轮迭代中，我们有

$$\begin{aligned} Dom[B_3] &= \{B_3\} \cup (Dom[B_2] \cap Dom[B_7]) \\ &= \{B_3\} \cup (\{B_0, B_1, B_2\} \cap N) \\ &= \{B_3\} \cup \{B_0, B_1, B_2\} \\ &= \{B_0, B_1, B_2, B_3\} \end{aligned}$$

同理，可以继续该行剩余的 $Dom[B_4]$ 等集合的值。最终，第三轮迭代的结果和第二轮迭代的结果一致，则算法已经达到了一个不动点。不难发现，表中最终的 $Dom[]$ 集合与表2.1相同。另外，尽管从理论上看，外层循环的的执行次数为 N^2（算法第 5 行），但表2.2中的计算说明，如果选择合理的计算顺序，

表 2.3: 数据流方程的一般形式在支配节点集合计算的对应

\	一般数据流计算	支配节点集合的计算
$In[n], Out[n]$	语句 n 前后的任意数据流信息	节点构成的集合
$l[n]$	语句 n 的任意局部信息	节点 n 本身构成的集合 $\{n\}$
f	任意转移函数	集合的并 $\cup$
传播方向	任意前向或后向	前向
$\bigvee$	数据流的传播	集合的交 $\cap$

每个支配节点集合在一次循环中都会改变，因此，外层循环的执行次数为 N。

在结束本小节的讨论前，我们将支配节点的数据流及其计算过程，对应到我们在第 2.1小节讨论的一般数据流分析的过程，如表 2.3所示。首先，在支配节点集合计算中，我们令 $In[n]$ 和 $Out[n]$ 都是由控制流图节点构成的集合。并且，我们定义语句 n 的局部信息 $l[n] = \{n\}$，且令节点 n 的支配节点集合 $Dom[n] = Out[n]$。接着，我们利用方程

$$Out[n] = f(In[n], l[n]),$$

从语句 n 前的数据流信息 $In[n]$ 和局部信息 $l[n]$，计算语句后的数据流信息 $Out[n]$。其中，转移函数定义为集合的并，即

$$f \triangleq \cup。$$

最后，我们将数据流信息在控制流图上进行前向传播

$$In[n] = \bigvee_{p \in pred[n]} Out[p],$$

由前驱节点的支配节点集合 $Dom[\,]$，计算当前节点 n 的集合 $In[n]$，最终得到全局数据流信息。注意到，我们将传播计算 $\bigvee$ 定义为集合的交运算 $\cap$。

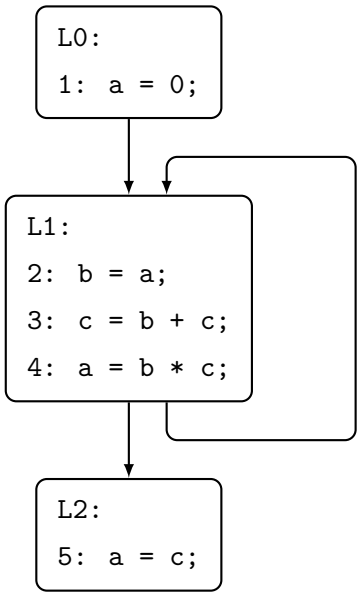


图 2.3: 进行活跃分析示例程序的控制流图

2.3 活跃分析

编译器经常需要分析程序中变量的使用情况，以指导对程序的优化。例如，如果编译器能够确定两个变量 x 和 y 在程序中不会同时使用，则编译器可以通过复用寄存器，把这两个变量分配到同一个物理寄存器 r 中，以降低寄存器的使用压力。

一般的，如果程序中的某个变量 x 在某个程序点 n 后面还要使用，则称这个变量在该程序点 n 是**活跃的**（live），分析变量的活跃性的过程便被称为**活跃分析**（liveness analysis）。

作为示例，考虑图??中给出的控制流图。为方便引用，控制流图中每条语句左侧标记了唯一的序号。先考虑变量 a，第 1 条语句赋值的 a 的值，将被第 2 条语句读取，因此，变量 a 在语句 1 后是活跃的。同理，第 4 条语句赋值的变量 a，将被第 2 条语句读取，因此，变量 a 在第 4 条语句后也是活跃的。类似的，变量 b 被第 2 条语句赋值后，将被第 3 和 4 两条语句

表 2.4: 活跃分析的生成集合和杀死集合

语句 s	生成集 $Gen[s]$	杀死集 $Kill[s]$
$t = x_1 \oplus x_2$	$\{x_1, x_2\}$	$\{t\}$
$t = x$	$\{x\}$	$\{t\}$
`if`(x, L_1, L_2)	$\{x\}$	$\{\}$
`jmp` L	$\{\}$	$\{\}$
`ret` x	$\{x\}$	$\{\}$
$t = f(x)$	$\{x\}$	$\{t\}$

分别读取，因此，变量 b 在第 3 和 4 两条语句后活跃。但是，不难分析，变量 b 在语句 1 和 4 后面都不活跃。这说明变量 a 和 b 不会同时活跃，因此它们可以放置在同一个物理寄存器中。同理，可以分析变量 c 与变量 a 以及 b 的活跃范围均有重叠，因此需要单独的物理寄存器存放 c。

变量活跃性可以通过数据流方程进行建模和求解。一般的，在数据流分析中，语句对变量的读取称为对变量的**使用**（use），而对变量的赋值称为对变量的**定值**（define）。我们用符号 $Gen[n]$ 和 $Kill[n]$，分别代表一个语句 n 对变量的使用集合和定值集合。直观上，集合的名字 Gen 代表了变量使用将在语句前将成为活跃的，而名字 $Kill$ 代表了变量的定值则会杀死该语句后变量的活跃性。我们在表2.4中，给出了部分语句的生成集 Gen 和杀死集 $Kill$。

我们用符号 $In[n]$ 和 $Out[n]$，分别代表在语句 n 前后的活跃变量集合，则它们可由数据流方程

$$
\begin{aligned}
In[n] &= Gen[n] \cup (Out[n] - Kill[n]) \\
Out[n] &= \bigcup_{s \in succ[n]} In[s]
\end{aligned}
$$

描述。首先，如果一个变量属于集合 $Gen[n]$，或者属于集合

算法 2.2 活跃分析数据流方程迭代算法

输入： 一个控制流图 G，其中每条语句的 Gen 集和 $Kill$ 集都已计算得到

输出： 控制流图中每条语句的 $In[]$ 集合和 $Out[]$ 集合

```
1: function livenessAnalysis(G)
2:     for each n ∈ G do
3:         In[n] = Out[n] = ∅
4:     while any Out[n] or In[n] changed do
5:         for each n ∈ G do
6:             Out[n] = ⋃_{s∈succ[n]} In[s]
7:             In[n] = Gen[n] ∪ (Out[n] − Kill[n])
```

$Out[n]$ 但不属于集合 $Kill[n]$，则该变量属于集合 $In[n]$。其次，如果一个变量属于节点 n 的任何后继 s 的 $In[s]$，则该变量属于 $Out[n]$。

根据上述数据流方程，我们在算法2.2中给出了求解活跃分析的过程。首先，算法将所有节点 n 的 $In[n]$ 和 $Out[n]$ 集合都初始化为空集 $\emptyset$（第 2-3 行）。接着，算法不断迭代，利用数据流方程的定义，计算每个节点的 In 和 Out 集合，直到所有集合都不再变化为止，此时到达了一个不动点。

为了分析算法的最坏时间复杂度，我们假设程序中共计包含 N 条语句和 V 个变量。则算法的内层循环体（第 6-7 行）的执行时间为为 $O(sV)$，其中 s 为节点的最大后继数量。算法的 `for` 循环对 N 个节点迭代，其执行时间为 $O(sVN)$。最后，`while` 循环每次迭代只会使得每个 In 和 Out 集合不变或增大，但 In 和 Out 集合元素数量变化次数至多为 $2VN$。因此，该算法在最坏情况下时间复杂度为 $O(sV^2N^2)$。

作为示例，表2.5展示了迭代算法2.2 在图2.3程序上的迭代计算过程。在每轮迭代过程中，算法按照从节点 5 到 1 的顺序进行迭代。注意到，迭代算法在第三轮迭代后，得到了不动

表 2.5: 活跃分析数据流方程求解过程

			迭代 1		迭代 2		迭代 3	
n	$Gen[n]$	$Kill[n]$	$Out[n]$	$In[n]$	$Out[n]$	$In[n]$	$Out[n]$	$In[n]$
5	c	a		c		c		c
4	b, c	a	c	b, c	a, c	b, c	a, c	b, c
3	b, c	c	b, c	b, c	b, c	b, c	b, c	b, c
2	a	b	b, c	a, c	b, c	a, c	b, c	a, c
1		a	a, c	c	a, c	c	a, c	c

点并终止。

2.4 可用表达式

一个表达式 e 可能会在程序中被多次计算。如果某个程序点 p 处的一个表达式 e 在该程序点前 q 处被计算过，且 p 处的计算结果与程序点 q 处表达式的结算结果相同，我们便能将当前程序点 p 处的表达式 e 直接替换成 q 处的计算结果，从而消除冗余计算，以提高程序性能。

考虑图2.4中的示例控制流图。表达式 $e + f$ 出现在基本块 $\{L_1, L_2, L_3\}$ 中，并且，无论控制流如何到达基本块 L_3 的第 4 条语句，表达式 $e + f$ 分别会在第 1 条或第 3 条语句中计算过，且计算结果与第 4 条语句处的计算结果相同，我们称第 4 条语句中的表达式计算 $e + f$ 是完全冗余的。因此，我们可将第 4 条语句中的冗余表达式 $e + f$，替换为先前的计算结果。具体的，我们可将第 1 条语句改写为 $t = e + f$ 和 $a = t$ 两条，并将第 3 行语句也改写为 $t = e + f$ 和 $b = t$ 两条。然后将第 4 条语句改写为 $c = t$。

与此相反，虽然表达式 $e - f$ 也出现在基本块 L_3 中，但

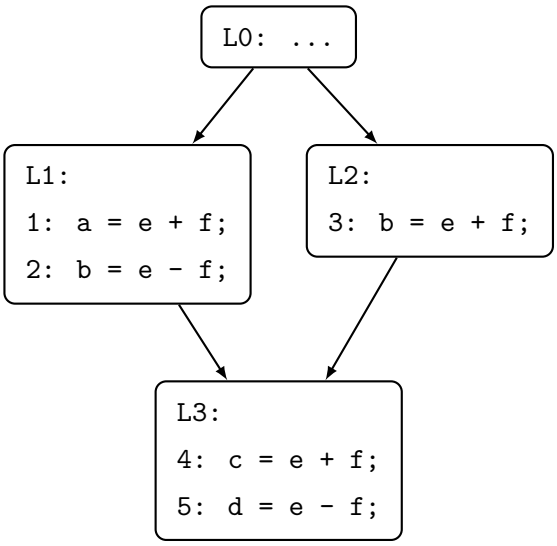


图 2.4: 可用表达式示例程序控制流图

它的值只在基本块 L_1 中计算过但未在 L_2 中计算过，因此，表达式 $e-f$ 在第 5 条语句处不是完全冗余的，也自然无法将 L_3 中表达式 $e-f$ 替换为 L_1 中的相同表达式。

为了严格刻画完全冗余表达式，我们需要引入**可用表达式**（available expression）的概念。表达式 $x \oplus y$ 在节点 n **可用**，指的是从控制流图的入口节点 s_0 到节点 n 的**每条**路径上，表达式 $x \oplus y$ 都至少计算过一次，并且在每条路径上进行最近一次对该表达式计算后，没有再对 x 和 y 的定值。例如，对图2.4中的控制流图，表达式 $e+f$ 在第 4 条语句处是可用的，但表达式 $e-f$ 在第 5 条语句处不是可用的。

对于语句 n，我们用符号 $Gen[n]$ 和 $Kill[n]$，分别表示语句 n **生成**和**杀死**的表达式集合，其计算规则在表2.6中给出。赋值语句 $t = x_1 \oplus x_2$ 的规则，生成了表达式 $\{x_1 \oplus x_2\}$，$-Kill[s]$，并且杀死了所有包含变量 t 的表达式。需要注意，赋值语句的生成集 $Gen[s]$ 中包含减法，这是因为如果该赋值左侧定值变量 t 是 x_1 或 x_2 的话，则该定值会杀死表达式 $x_1 \oplus x_2$ 自身。赋值语句 $t = x$ 或函数调用语句 $t = f(x)$，同样会杀死所有包

表 2.6: 可用表达式的生成集合和杀死集合

语句 s	生成集 $Gen[s]$	杀死集 $Kill[s]$
$t = x_1 \oplus x_2$	$\{x_1 \oplus x_2\} - Kill[s]$	包含 t 的表达式
$t = x$	{}	包含 t 的表达式
`if`(x, L_1, L_2)	{}	{}
`jmp` L	{}	{}
`ret` x	{}	{}
$t = f(x)$	{}	包含 t 的表达式

算法 2.3 可用表达式数据流方程迭代算法

输入： 一个控制流图 G，其中每条语句的 Gen 集和 $Kill$ 集都已完成计算

输出： 控制流图中每条语句的 In 集和 Out 集

```
1: function AvailableExpression(G)
2:     for each n ∈ G do
3:         In[n] = Out[n] = N
4:     In[s_0] = ∅
5:     while any In[n] or Out[n] changed do
6:         for each n ∈ G do
7:             In[n] = ⋂_{p∈pred[n]} Out[p]
8:             Out[n] = Gen[n] ∪ (In[n] − Kill[n])
```

含变量 t 的表达式。

利用生成集 Gen 和杀死集 $Kill$，我们可以用数据流方程

$$In[n] = \bigcap_{p \in pred[n]} Out[p]$$

$$Out[n] = Gen[n] \cup (In[n] - Kill[n])$$

刻画每个节点 n 入口和出口的可用表达式集合 $In[n]$ 和 $Out[n]$。

基于可用表达式的数据流方程，算法2.3给出了对其进行迭代求解的过程。算法首先将所有节点 n 的 $In[n]$ 和 $Out[n]$ 集合初始化为全集 N，并将入口节点 s_0 的 In 集合初始化为空集 $\emptyset$。接着，算法对每个节点的 In 和 Out 集合进行迭代计

表 2.7: 可用表达式数据流方程求解过程

			迭代 1		迭代 2	
n	$Gen[n]$	$Kill[n]$	$In[n]$	$Out[n]$	$In[n]$	$Out[n]$
1	$e+f$			$e+f$		$e+f$
2	$e-f$		$e+f$	$e+f, e-f$	$e+f$	$e+f, e-f$
3	$e+f$			$e+f$		$e+f$
4	$e+f$		$e+f$	$e+f$	$e+f$	$e+f$
5	$e-f$		$e+f$	$e+f, e-f$	$e+f$	$e+f, e-f$

算，直到到达一个不动点为止。

接下来我们分析算法2.3的时间复杂度。我们假设程序中共计包含 N 条语句和 E 个表达式。则算法的内层循环体（第 7-8 行）的执行时间为为 $O(pE)$，其中 p 为节点的最大前驱数量。算法的 `for` 循环对 N 个节点迭代，其总执行时间为 $O(pEN)$。最后，`while` 循环每次迭代只会使得每个 In 和 Out 集合不变或减小，但 In 和 Out 集合元素数量变化次数至多为 $2EN$。因此，该算法在最坏情况下时间复杂度为 $O(pE^2N^2)$。

表2.7给出了迭代算法2.3，在图2.4中程序上的迭代计算过程。算法迭代按照从节点 1 到 5 的顺序进行，并在两轮迭代后终止。从最后的计算结果可以看到，表达式 $e+f$ 在节点 2、4 和 5 处分别是可用的。

2.5 忙碌表达式

给定程序点 n，如果从 n 出发到程序控制流图出口的每条路径上，都有对表达式 $x+y$ 的计算，并且在 n 到达每个表达式计算点上，都没有对变量 x 或 y 的定值，则我们能够将所有这些路径上对表达式 $x+y$ 的计算提前到节点 n 处，从而

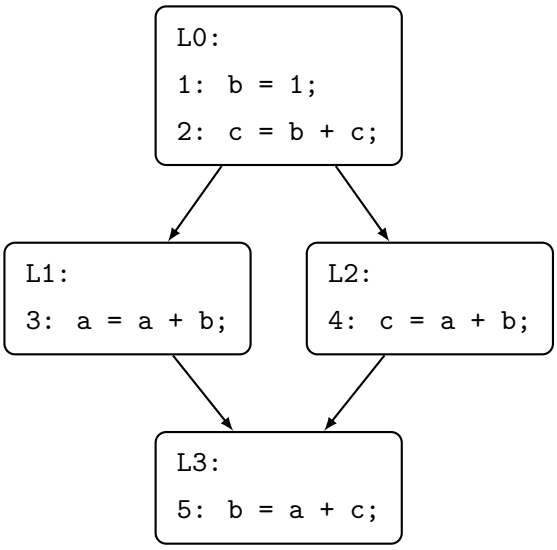


图 2.5: 忙碌表达式示例程序控制流图

消除这些程序点上的冗余计算 $x + y$。

以图2.5中的程序为例，假设其出口为基本块 L_3。则我们从第 2 条语句出发，不管沿着路径 $L_0 \to L_1 \to L_3$ 或 $L_0 \to L_2 \to L_3$，到达出口块 L_3 时，都会经过对表达式 $a + b$ 的计算（分别在第 3 条和第 4 条语句）。并且，在从第 2 条语句到达第 3 条或第 4 条语句前，都没有对变量 a 和 b 进行定值。则我们可以将第 3 条和 4 条语句中的表达式计算 $a + b$，都提升到第 2 条语句后。具体的，我们在第 2 条语句后新增加语句 $t = a + b$, 并将第 3 条和第 4 条语句分别改写为 $a = t$ 和 $c = t$。这样，我们消除了在第 3 条和第 4 条语句中的冗余计算 $a + b$。

一般的，我们将具有以上性质的表达式称为**忙碌表达式**（busy expression），或**预期执行表达式**（anticipated expression）。为了定义忙碌表达式的数据流方程，表2.8 给出了语句 s 的忙碌表达式的生成集合 $Gen[s]$ 和杀死集合 $Kill[s]$。赋值语句 $t = x_1 \oplus x_2$，生成了忙碌表达式 $x_1 \oplus x_2$，并且杀死了包含变量 t 的表达式。需要注意的是，与可用表达式中赋值语句的生成集合不同，忙碌表达式中赋值语句 s 的生成集合 $Gen[s]$ 不需

表 2.8: 忙碌表达式的生成集合和杀死集合

语句 s	生成集 $Gen[s]$	杀死集 $Kill[s]$
$t = x_1 \oplus x_2$	$\{x_1 \oplus x_2\}$	包含 t 的表达式
$t = x$	{}	包含 t 的表达式
`if`(x, L_1, L_2)	{}	{}
`jmp` L	{}	{}
`ret` x	{}	{}
$t = f(x)$	{}	包含 t 的表达式

要减去包含 t 的表达式，这本质上是因为忙碌表达式分析采用由后向前的顺序进行计算，表达式是向上暴露的。赋值语句 $t = x$ 和函数调用语句 $t = f(x)$，会杀死所有包括变量 t 的表达式。

利用生成集 Gen 和杀死集 $Kill$，我们可以用数据流方程

$$
\begin{aligned}
In[n] &= Gen[n] \cup (Out[n] - Kill[n]) \\
Out[n] &= \bigcap_{s \in succ[n]} In[s]
\end{aligned}
$$

来给出每个节点 n 前后的忙碌表达式集合 $In[n]$ 和 $Out[n]$。

基于上述数据流方程，算法2.4给出了基于迭代求解数据流方程的过程。算法首先将所有节点 n 的 $In[n]$ 和 $Out[n]$ 集合初始化为全集 N，并将出口节点 $EXIT$ 的 $Out[EXIT]$ 初始化为空集 $\emptyset$（第 2-4 行）。接着，算法通过迭代，利用数据流方程计算每个节点 n 的 $In[n]$ 和 $Out[n]$ 集合，直到到达一个不动点（第 5-8 行）。

接下来我们分析算法2.4的时间复杂度。我们假设程序中共计包含 N 条语句和 E 个表达式。则算法的内层循环体（第 7-8 行）的执行时间为为 $O(sE)$，其中 s 为节点的最大后继数量。算法的 `for` 循环对 N 个节点迭代，其总执行时间为

算法 2.4 忙碌表达式数据流方程迭代算法

输入： 一个控制流图 G，其中每条语句的 Gen 集和 $Kill$ 集都已完成计算

输出： 控制流图中每条语句的 In 集和 Out 集

```
1: function BusyExpression(G)
2:     for each n ∈ G do
3:         In[n] = Out[n] = N
4:     Out[EXIT] = ∅
5:     while any In[n] or Out[n] changed do
6:         for each n ∈ G do
7:             Out[n] = ⋂_{s∈succ[n]} In[s]
8:             In[n] = Gen[n] ∪ (Out[n] − Kill[n])
```

表 2.9: 忙碌表达式数据流方程求解过程

			迭代 1		迭代 2	
n	$Gen[n]$	$Kill[n]$	$Out[n]$	$In[n]$	$Out[n]$	$In[n]$
5	$a+c$	$a+b, b+c$		$a+c$		$a+c$
4	$a+b$	$a+c, b+c$	$a+c$	$a+b$	$a+c$	$a+b$
3	$a+b$	$a+b, a+c$	$a+c$	$a+b$	$a+c$	$a+b$
2	$b+c$	$a+c, b+c$	$a+b$	$a+b, b+c$	$a+b$	$a+b, b+c$
1		$a+b, b+c$	$a+b, b+c$		$a+b, b+c$	

$O(sEN)$。最后，`while` 循环每次迭代只会使得每个 In 和 Out 集合不变或减小，但 In 和 Out 集合元素数量变化次数至多为 $2EN$。因此，该算法在最坏情况下时间复杂度为 $O(sE^2N^2)$。

作为示例，表2.9展示了迭代算法2.4 在图2.5示例程序上的迭代计算过程。算法按照语句从 5 到 1 的顺序进行迭代计算，并在进行两轮迭代后到达不动点而终止。从计算结果可以看到，表达式 $a+b \in Out[2]$，这意味着从第 2 条语句往后到程序出口，表达式 $a+b$ 都忙碌，因此将表达式 $a+b$ 的计算向前放置到语句 2 后，可以消除 $a+b$ 的冗余计算。

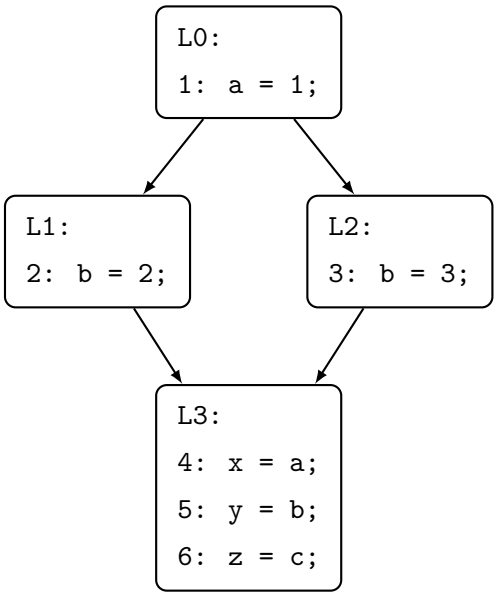


图 2.6: 示例控制流图

2.6 到达定值

在许多程序分析和优化问题中，对于变量 x 的特定使用，如果能够确定哪些对 x 的定值，能够到达该使用，我们就能确定关于变量 x 的额外性质，从而给程序分析和优化提供指导。

考虑图2.6中的示例，可以确定在基本块 B_3 中，能到达第 4 条语句对变量 a 使用的定值点是第 1 条语句，由于这是唯一能到达该使用的定值，因此可以把对该变量 x 定值常量 1 传播到第 4 条语句处，从而该语句被改写为 $x = 1$。同理，可以分析确定第 2 和第 3 这两条对变量 b 的定值，能够到达第 5 行对 b 的使用，而由于第 2 条和第 3 条对变量 b 的定值并不相同（分别为常量 2 和 3），因此我们无法将其这些常量传播到第 5 条对变量 b 的使用。最后，考虑对第 6 条语句对变量 c 的使用，由于没有任何对 c 的定值能够到达该语句，这实际上意味着第 6 条语句将读取 c 的一个未初始化的值，高质量的编译

表 2.10: 到达定值的生成集和杀死集

语句 s	生成集 $Gen[s]$	杀死集 $Kill[s]$
$d: t = x_1 \oplus x_2$	$\{d\}$	$defs(t) - \{d\}$
$d: t = x$	$\{d\}$	$defs(t) - \{d\}$
`if`(x, L_1, L_2)	$\{\}$	$\{\}$
`jmp` L	$\{\}$	$\{\}$
`ret` x	$\{\}$	$\{\}$
$d: t = f(x)$	$\{d\}$	$defs(t) - \{d\}$

器往往会产生一条“变量未初始化”的警报或错误信息。

我们将这种针对变量的使用，分析有哪些定值能够到达的过程，称为**到达定值**（reaching definition）分析。一般的，对于一个对变量 x 的定值 d，如果存在一条从 d 到达某一个程序点 p 的路径，并且在这条路径上不存在其他对变量 x 的定值，我们就说定值 d **到达**了程序点 p。

为了定义到达定值的数据流方程，表2.10 给出了语句 s 的到达定值的生成集合 $Gen[s]$ 和杀死集合 $Kill[s]$，其中每个集合中的元素都是定值的语句编号，并且 $defs(t)$ 是对变量 t 所有定值的集合。赋值语句 $d: t = x_1 \oplus x_2$，生成了定值集合 $\{d\}$，杀死了定值集合 $defs(t) - \{d\}$，即杀死了除 d 本身之外的对变量 t 的所有定值。赋值语句 $d: t = x$ 和函数调用语句 $d: t = f(x)$ 的规则类似，我们将其分析过程作为练习，留给读者完成。

基于生成集 Gen 和杀死集 $Kill$，我们可以用数据流方程

$$In[n] = \bigcup_{p \in pred[n]} Out[p]$$

$$Out[n] = Gen[n] \cup (In[n] - Kill[n])$$

来计算程序中到达每个语句 n 前后的定值集合 $In[n]$ 以及

算法 2.5 到达定值

输入： 一个控制流图 G，其中每条语句的 Gen 集和 $Kill$ 集都已完成计算

输出： 控制流图中每条语句的 In 集和 Out 集

```
1: function ReachDefinition(G)
2:     for each n ∈ G do
3:         In[n] = Out[n] = ∅
4:     while any In[n] or Out[n] changed do
5:         for each n ∈ G do
6:             In[n] = ⋃_{p∈pred[n]} Out[p]
7:             Out[n] = Gen[n] ∪ (In[n] − Kill[n])
```

$Out[n]$。

基于上述数据流方程，算法2.5给出了计算到达定值的过程。算法首先将所有语句 n 的 $In[n]$ 和 $Out[n]$ 集合初始化为空集 $\emptyset$ （第 2-3 行）。接着，算法利用迭代，基于数据流方程对语句 n 的 $In[n]$ 和 $Out[n]$ 集合进行计算，直到达到不动点为止。

接下来我们分析算法2.5的时间复杂度。我们假设程序中共计包含 N 条语句，其中包括 D 条定值。则算法的内层循环体（第 7-8 行）的执行时间为为 $O(pD)$，其中 p 为节点的最大前驱数量。算法的 `for` 循环对 N 个节点迭代，其总执行时间为 $O(pDN)$。最后，`while` 循环每次迭代只会使得每个 In 和 Out 集合不变或减小，但 In 和 Out 集合元素数量变化次数至多为 $2DN$。因此，该算法在最坏情况下时间复杂度为 $O(pD^2N^2)$。

作为示例,考虑图2.7中的控制流图,假设算法按 L_0, L_1, L_2, L_3 的顺序对基本块进行处理。表2.11给出了算法对该流图每次迭代的结果。在第 3 次迭代后，$In[]$ 和 $Out[]$ 集合不再发生变化，算法运行结束。

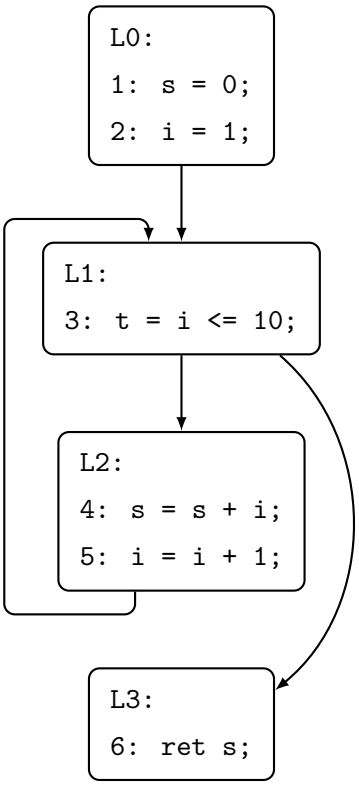


图 2.7: 示例控制流图

2.7 改进与优化

本章中，我们直接基于数据流方程，给出了数据流分析算法的实现。尽管这种实现方式简单直接，但没有考虑到具体数据流方程的特点，因而时间复杂度较高。在本小节，我们讨论对这些算法实现的一些改进与优化，以进一步降低算法的运行时间复杂度。

2.7.1 支配节点的高效计算

给定控制流图 G 中的节点 n，其支配节点集合为 $Dom[n]$，我们记

$$Sdom[n] = Dom[n] - \{n\} ,$$

表 2.11: 算法的迭代执行过程

			迭代 1		迭代 2		迭代 3	
n	$Gen[n]$	$Kill[n]$	$In[n]$	$Out[n]$	$In[n]$	$Out[n]$	$In[n]$	$Out[n]$
1	1	4		1		1		1
2	2	5	1	1,2	1	1,2	1	1,2
3	3		1,2	1,2,3	1,2,3,4,5	1,2,3,4,5	1,2,3,4,5	1,2,3,4,5
4	4	1	1,2,3	2,3,4	1,2,3,4,5	2,3,4,5	1,2,3,4,5	2,3,4,5
5	5	2	2,3,4	3,4,5	2,3,4,5	4,4,5	2,3,4,5	3,4,5
6			1,2,3	1,2,3	1,2,3,4,5	1,2,3,4,5	1,2,3,4,5	1,2,3,4,5

称 $Sdom[n]$ 为节点 n 的**严格支配节点**。可以证明：若 $Sdom[n] \neq \emptyset$，则存在唯一的节点 m，使得对任意的节点 $k \in Sdom[n]$，节点 k 都支配 m。（我们把对该结论的证明，作为练习留给读者完成。）直观上，节点 m 是支配节点 n 的所有节点中“最小”的一个，或离节点 n“最近”的一个。我们称节点 m 是节点 n 的**直接支配节点**（Immediate dominator），记为 $Idom[n]$，显然 $|Idom[n]| \leq 1$。

控制流图的节点及其直接支配节点的支配关系，构成了**支配树**（Dominator tree）。支配树的根是控制流图的入口节点 s_0，且支配树的边从节点 n 的直接支配节点 $Idom[n]$ 指向节点 n。则不难得到，节点 n 的所有支配节点 $Dom[n]$，即是从节点 n 到树根节点 s_0 经过路径上的所有节点，即支配节点集合 $Dom[n]$ 可以表示为：

$$Dom[n] = \{n\} \cup Idom[n] \cup Idom[Idom[n]] \cup \ldots \cup \{s_0\}。$$

支配树的概念，启发了计算支配节点的新的算法。为此，我们首先定义数组 $doms$ 来表示支配树，其中每个数组元素 $doms[n]$ 都存储了节点 n 的直接支配节点 $Idom[n]$（即支配树中节点 n 的双亲节点）。则通过从 n 开始遍历 $doms$ 数组，我们就可以得到 n 的支配节点集合。

算法 2.6 支配节点高效计算

输入： 程序控制流图 G，其节点按后序遍历顺序排序

输出： 每个节点的直接支配节点

```
1:  doms[]
2:  function calculateDom(G)
3:      for each node n do
4:          doms[n] = Undefined
5:      doms[s_0] = s_0
6:      while doms changed do
7:          for each node i, in reverse postorder (except start node) do
8:              semi_idom = first (processed) predecessor of i
9:              for all other predecessors p of i do
10:                 if doms[p] ≠ Undefined then
11:                     semi_idom = intersect(p, semi_idom)
12:             if doms[i] ≠ semi_idom then
13:                 doms[i] = semi_idom
14: function intersect(b_1, b_2)
15:     while b_1 ≠ b_2 do
16:         while b_1 < b_2 do
17:             b_1 = doms[b_1]
18:         while b_2 < b_1 do
19:             b_2 = doms[b_2]
20:     return b_1
```

数组 $doms$ 可以通过算法 2.6计算得到。算法接受控制流图 G 作为输入，计算其支配树。其中，图 G 中的所有节点，已经按照后序遍历顺序编号，该编号将用作该节点在数组 $doms$ 中的下标。首先，算法将 $doms$ 数组元素都初始化为 $Undefined$，并将起始节点 s_0 对应的数组元素 $doms[s_0]$ 值设为自身（第 3 到 5 行）。然后，算法按照*逆后序遍历*（reverse postorder）的顺序遍历所有节点 n （除起始节点 s_0 外），以确保在处理节点 n 时，n 的前驱节点尽量已经处理过。对每个节点 i，算法先选取节点 i 已处理过的前驱节点中的一个，作为节点 i 的近

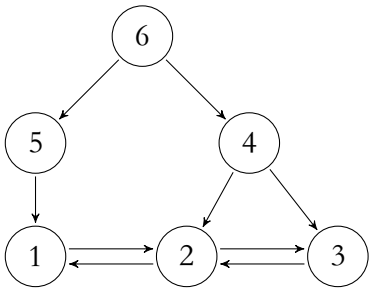


图 2.8: 计算支配树的控制流图示例

似直接支配节点 $semi\_idom$。接着，算法遍历节点 i 的其它剩余前驱节点 p，如果该节点已经计算过直接支配节点的话（亦即其值不等于 $Undefined$），则通过调用函数 $intersect$ 来更新近似直接支配节点 $semi\_idom$ 的值。最后，如果 $dom[i]$ 中原有的值和新计算的 $semi\_idom$ 值不同，则将 $dom[i]$ 中值更新为新计算的值 $semi\_idom$。算法重复上述过程，直到 $doms$ 数组不再发生变化为止，此时数组 $doms$ 中元素即为对应节点的直接支配节点，进而可以得出每个节点的支配节点集合以及支配树。

函数 $intersect$ 接受两个节点 b_1 和 b_2 作为输入，计算并返回二者在（近似）支配树中的共同最小祖先节点。由于支配树按照图的后续遍历进行编号，祖先节点的编号比孩子节点更大，所以算法在计算祖先节点时，总是更新较小的节点编号（第 16 和 18 行）。

作为示例，我们应用算法2.6，计算图2.8中的控制流图的支配树，具体过程在表2.12和图 2.9中给出。表2.12的列给出了 $doms[]$ 数组的值，而表的行给出了数组 $doms[]$ 在每一轮计算后的状态。算法首先将数组 $doms$ 初始化为表第一行的状态，注意到除了入口节点 6 外，其它节点的直接支配节点均被置为 u（$Undefined$）。由于算法需要按逆后续遍历顺序处理节点，所以算法在每轮迭代中，都按照从 5 到 1 的顺序对节点

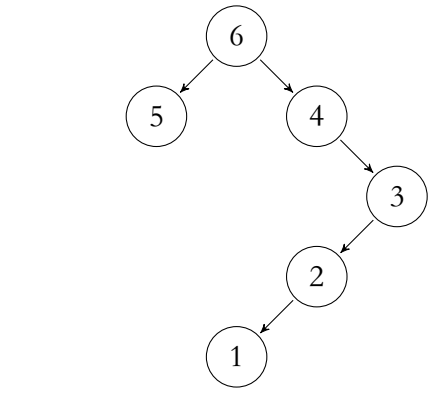
表 2.12: 支配节点数组 $doms$ 迭代过程

	$doms[n]$					
	1	2	3	4	5	6
初始化	u	u	u	u	u	6
第 1 轮	6	4	4	6	6	6
第 2 轮	6	6	4	6	6	6
第 3 轮	6	6	6	6	6	6
第 4 轮	6	6	6	6	6	6

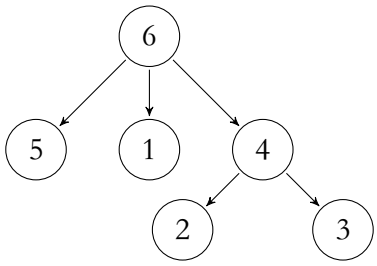
进行遍历（见图2.9a给出的深度优先遍历生成树以及逆后序遍历序）。以第 1 轮迭代为例，算法首先处理节点 5，将 $doms[5]$ 的值置为其前驱 6。同理，节点 4 的直接支配节点 $doms[4]$ 被置为其前驱 6，而节点 3 的直接支配节点 $doms[3]$ 被置为其前驱 4。对于节点 2，其已经有两个前驱节点 3 和 4 的直接支配节点（分别为 $doms[3]$ 和 $doms[4]$）已经被计算过，因此算法首先可将近似直接支配节点 $semi\_idom$ 置为节点 3，并调用函数 $intersect$ 计算节点 3 与其另外一个前驱节点 4 的最小公共祖先节点，返回节点 4。同理，可计算得到节点 1 的直接支配节点为 1。

第一轮迭代结束后，我们在图2.9b中，给出了算法生成的近似支配树。近似性体现在图中的节点的直接支配节点，未必是其双亲节点，而可能是非双亲的祖先节点。例如，考虑节点 2，其图中的双亲节点是 4，而通过考察图2.8可知，节点 4 并不是 2 的支配节点，当然更不可能是直接支配节点。

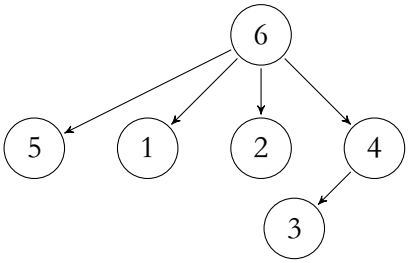
算法继续按上述步骤进行迭代，分别得到表 2.12中第二轮和第三轮的计算结果，其分别对应的图2.9c和 2.9d中的近似支配树。最终，算法在执行完四轮后，确认数组 $doms$ 不再变化，算法执行终止。



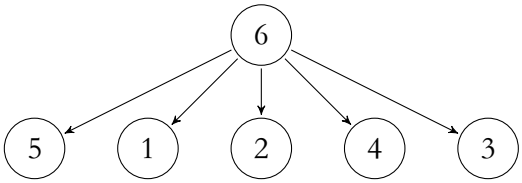
(a) 深度优先遍历生成树，及其给出的后续遍历序列



(b) 算法进行完第一轮迭代后得到的近似支配树



(c) 算法进行完第二轮迭代后得到的近似支配树



(d) 算法进行完第三轮迭代后得到的近似支配树，这也是算法最终生成的支配树

图 2.9: 算法执行的迭代过程，以及产生的近似支配树

算法的时间复杂度由几部分组成。首先，后序遍历控制流图的时间复杂度为 $O(V)$，其中 V 是图中节点的数量。接下来，算法的内层循环在遍历图时（即算法 2.6第 7 行），每次迭代会访问每个节点一次，并且在每个节点上执行集合的交集运算。注意到集合交集运算本质上是在近似支配树上找最近公共祖先，因此每次迭代的总时间复杂度为 $O(V)$，而内层循环的总时间复杂度为 $O(V^2)$。

此外，算法的外层迭代（即算法 2.6第 6 行），在最坏情况下，每个 $doms$ 元素都可以改变 V 次，因为外层循环将执行 $O(V^2)$ 次。但这只是理论的上界，可以证明，外层迭代算法的最多迭代次数为 $d(G)+3$，其中 $d(G)$ 是图的循环连通度。结合以上分析，算法的总体时间复杂度为 $O(V^2(d(G)+3))$。

表 2.13: 位向量的一些常用操作

	并集	交集	补集	差集	添加元素 e_i	删除元素 e_i
集合表示	$S \cup T$	$S \cap T$	$U - S$	$S - T$	$S \cup \{e_i\}$	$S - \{e_i\}$
位向量表示	$S \mid T$	$S \,\&\, T$	$\neg S$	$S \,\&\, \neg T$	$S \mid (1 << i)$	$S \,\&\, \neg(1 << i)$

2.7.2 位向量

对于一个有限域，如果域中的 N 个元素可以使用介于 0 到 $N-1$ 之间数字进行索引（例如将所有元素放在一个数组中，通过数组下标进行索引），那么该有限域上的集合 S 可以使用一个只包含 0 或 1 的**位向量**（bit vector）来表示，其中 S 中的第 i 位为 1 表示第 i 个元素在集合中存在，否则元素不存在。

基于位向量表示，集合操作可以通过位操作来实现。例如，计算两个集合的并集可以通过对位向量按位或实现；两个集合的交集可以通过按位与实现。表 2.13总结了一些集合操作所对应的位向量操作。

相比于使用集合数据结构，位向量表示无论是在时间上还是空间上都更具有优势。事实上，假设计算机的字长为 W，位向量表示可以同时操作 W 位。两个长度为 N 的位向量的操作通常可以使用 N/W 条指令或迭代完成计算。对第 i 个元素的添加和删除操作也只需要常数的时间。此外，集合中的一个元素只占用一比特位，更加节省内存空间。

然而，位向量表示也不总是有益的。由于有限域中的每一个元素都要在位向量中使用一位进行表示，位向量的长度固定为有限域的大小。对于那些域空间过大而集合中元素又相对较少的场景，位向量非常稀疏，即大部分都是零比特位，

此时使用其他数据结构表示可能会更加高效。

2.7.3 基本块粒度的局部信息

前面数据流分析算法的介绍中，我们都以语句为粒度进行计算，算法每次迭代需要对所有的语句进行一趟遍历。事实上，我们同样可以以基本块为粒度进行计算，从而减少每次迭代需要遍历的节点的数量，提高数据流分析计算的速度。

数据流分析能够以基本块为粒度进行计算基于以下事实：由于基本块内部的语句顺序执行，因此，一个基本块的数据流方程可以通过把它包含的所有语句的数据流方程组合起来得到。

我们以前向数据流分析为例来推导该事实，后向数据流分析与此类似。考虑同一个基本块中的一条语句 s 和其前驱语句 p，集合 $In[s]$ 等于 $Out[p]$，那么有

$$\begin{aligned} Out[s] &= Gen[s] \cup (In[s] - Kill[s]) \\ &= Gen[s] \cup ((Gen[p] \cup (In[p] - Kill[p])) - Kill[s]) \end{aligned}$$

考虑到集合的分配律 $(A \cup B) - C = (A - C) \cup (B - C)$ 和结合律 $(A - B) - C = A - (B \cup C)$，于是有

$$\begin{aligned} Out[s] = (&Gen[s] \cup (Gen[p] - Kill[s])) \\ &\cup (In[p] - (Kill[p] \cup Kill[s])) \end{aligned}$$

若我们将连续的两个语句 ps 看成整体，且令其生成集和杀死集分别为

$$\begin{aligned} Gen[ps] &= Gen[s] \cup (Gen[p] - Kill[s]) \\ Kill[ps] &= Kill[p] \cup Kill[s] \end{aligned}$$

则方程

$$Out[ps] = Gen[ps] \cup (In[ps] - Kill[ps])$$

给出了在 ps 整体上的计算规则。

这个规则可以推广到包含任意多个语句的基本块。假设基本块 B 包含 n 条语句，第 i 条语句的生成集和杀死集分别为 $Gen[i]$ 和 $Kill[i]$，其中 $1 \leq i \leq n$，则

$$Out[B] = Gen[B] \cup (In[B] - Kill[B])$$

其中

$$\begin{aligned} Kill[B] &= Kill[1] \cup Kill[2] \cup \ldots \cup Kill[n] \\ Gen[B] &= Gen[n] \cup (Gen[n-1] - Kill[n]) \cup \ldots \\ &\quad \cup (Gen[1] - Kill[2] - \ldots - Kill[n]) \end{aligned}$$

这样，在进行数据流分析时，我们可以先按上述公式，计算基本块块的生成集和杀死集，并继续以基本块为单位进行整个控制流图的计算。

通过以基本块为粒度进行计算，可以大大加速数据流分析的计算速度。假设数据流中共有 K 个基本块，共包含 N 条语句。计算得到每个基本块的 In 和 Out 的时间复杂度为 $O(K^3)$。此外，为了计算得到 $Kill[B]$ 和 $Gen[B]$，可以对流图中每个基本块中的所有语句进行一遍遍历计算得到，时间复杂度为 $O(N)$。最后，根据基本块的 In 和 Out，对基本块内包含的语句进行一遍遍历即可得到每条语句的 In 和 Out，时间复杂度同样为 $O(N)$。因此，以基本块为粒度进行数据流分析的总的时间复杂度为 $O(K^3 + 2N)$。

作为示例，表2.14展示了对图2.6中的控制流图，按照基本块为粒度计算到达定值时每次迭代的结果。算法按照 $L_0 \rightarrow$

表 2.14: 基本块为粒度的到达定值计算过程

			迭代 1		迭代 2		迭代 3	
B	$Gen[B]$	$Kill[B]$	$In[B]$	$Out[B]$	$In[B]$	$Out[B]$	$In[B]$	$Out[B]$
0	1,2	6,7		1,2		1,2		1,2
1	4		1,2	1,2,4	1,2,4,6,7	1,2,4,6,7	1,2,4,6,7	1,2,4,6,7
2	6,7	1,2	1,2,4	4,6,7	1,2,4,6,7	4,6,7	1,2,4,6,7	4,6,7
3			1,2,4	1,2,4	1,2,4,6,7	1,2,4,6,7	1,2,4,6,7	1,2,4,6,7

$L_1 \rightarrow L_2 \rightarrow L_3$ 的顺序遍历各个基本块。从表中可以看到，以基本块为粒度进行计算，显著减少了每次迭代需要遍历节点的数量。

2.7.4 基本块排序

在前向数据流分析中，前驱基本块计算输出的信息将作为后继基本块的输入。因此，如果我们能够重新组织基本块的排序，保证在某个块计算之前，它所有的前驱都尽量完成了计算，那么数据流分析可以更快的终止。同理，在后向数据流分析中，优先计算后继节点，将有助于分析算法更快终止。

为此，我们可以在继续数据流分析前，对控制流图进行排序，将节点尽量按照信息流动的顺序进行排序。具体的，在前向数据流分析中，我们可以按照伪拓扑序对控制流图中的节点进行排序。由于在伪拓扑序中，多数基本块都出现在其后继节点之前，因此在每次遍历计算过程中，每个节点所需要的前驱节点信息基本都已经计算完成。同理，对于后向数据流分析，可以使用逆伪拓扑排序来对节点进行排序，以便在计算过程中优先计算后继节点的数据流信息。

表 2.15: 数据流分析的总结和比较

	支配节点	活跃分析	可用表达式	忙碌表达式	到达定值
域	基本块集合	变量集合	表达式集合	表达式集合	定值集合
方向	前向	后向	前向	后向	前向
交汇运算	$\cap$	$\cup$	$\cap$	$\cap$	$\cup$
数据流方程	$Dom = \{n\} \cup \left(\bigcap_{pred} Dom\right)$	$Out = \bigcup_{succ} In$ $In = Gen \cup (Out - Kill)$	$In = \bigcap_{pred} Out$ $Out = Gen \cup (In - Kill)$	$Out = \bigcap_{succ} In$ $In = Gen \cup (Out - Kill)$	$In = \bigcup_{pred} Out$ $Out = Gen \cup (In - Kill)$
初始值	$Dom[B] = U$	$In[B] = \emptyset$	$Out[B] = U$	$In[B] = U$	$Out[B] = \emptyset$
边界条件	$Dom[ENTRY] = \{ENTRY\}$	$Out[EXIT] = \emptyset$	$In[ENTRY] = \emptyset$	$Out[EXIT] = \emptyset$	$In[ENTRY] = \emptyset$
May-Must	Must	May	Must	Must	May

2.8 总结与比较

在本章中，我们一共讨论了数据流分析的五个实例。尽管这些数据流分析讨论的具体问题不同，但它们在分析方向、交汇运算、初始值以及边界条件等信息方面也表现了共性。我们在表2.15中，对这些具体的数据流分析进行了总结和比较。

首先，数据流分析作用在不同的论域上。这些论域指的是待分析的数据流信息，包括但不限于基本块集合、变量集合、表达式集合、或定值集合等。在后续章节，我们还会讨论用于数据流分析的其它可能论域。

其次，按照数据流方程分析的方向和控制流方向的关系，数据流分析可以分为前向和后向分析。前向分析按照控制流的方向，从 In 集合计算 Out 集合，而后续分析的方向正好相反。这个事实意味着，在进行数据流方程的计算时，如果按照该方向进行节点的排序，则算法可更快的收敛。

再次，数据流信息在控制流图的边上计算和传递时，需要使用不同的交汇运算。例如，对于支配节点分析，当节点 n 前驱节点的数据流信息传递到 n 进行交汇时，节点 n 需要进行集合的交运算 $\cap$。相反，对于活跃分析，当节点 n 后继节点的数据流信息传递到 n 进行交汇时，节点 n 需要进行集合的

并运算 $\cup$。我们将在后续的章节将指出：交汇运算最终取决于待分析的数据流信息的语义。

第四，数据流分析使用不同的初始值，且该值与其所使用的交汇运算有关。例如，对于支配节点分析，我们使用全集 U 初始化每个节点，支配节点分析使用的集合交运算 $\cap$，将使得每个集合减小。相反，对于活跃分析，我们使用空集 $\emptyset$ 初始化每个节点，活跃分析使用的集合并运算 $\cup$，将使得每个集合增大。

最后，按照数据流方程计算得到的静态信息，和程序动态运行期间的动态信息的关系，数据流方程计算的信息可分为确定的（Must）和不确定的（May）。一般的，确定信息指的是程序动态运行肯定会满足数据流计算的静态信息。例如，在支配节点分析中，如果某个节点 n 的支配节点为 $Dom[n]$，则在程序的任意一次执行到达 n 时，我们可以确定该执行必须经过 $Dom[n]$ 中的所有节点，而注意到该次执行可能也经过了除 $Dom[n]$ 外的节点，所以 $Dom[n]$ 代表了程序执行到节点 n 时经过的节点的一个子集。同理，在可用表达式分析中，如果某个节点 n 的可用表达式集合为 A，则在程序的任意一次执行到达 n 时，我们可以确定该执行必须计算了 A 中的所有表达式，而注意到该次执行可能也计算了除 A 外的其它表达式，所以 A 代表了程序执行到节点 n 时计算的实际表达式的一个子集。与此相反，在活跃分析中，如果某个节点 n 的活跃变量集合为 V，则当程序的任意一次执行经过 n 结束时，未必执行了 V 中某个变量 x 的使用，所以 V 代表节点 n 后实际活跃变量的一个超集。类似的，在到达定值分析中，如果某个节点 n 的到达定值集合为 D，则当程序的任意一次执行到达 n 时，未必执行了 D 中的某个定值 d，所以 D 代表节点 n

实际定值的一个超集。

在结束本小节前，值得指出的是，由于数据流分析本质上是静态分析，所以无论它们计算得到的信息是确定还是不确定，以及计算得到的是子集还是超集，数据流信息必须都是保守的。保守性指的是讲数据流分析得到的信息用于程序分析和优化时，不能改变程序的语义。例如，可用表达式分析可被用于删除类似 $t = x_1 \oplus x_2$ 中的冗余表达式计算 $x_1 \oplus x_2$，因此，可用表达式分析必须保证当前带删除的表达式在每个到达该程序点的路径上都计算过，否则将得到错误的值。同理，到达定值分析可被用于将具有常量的定值 $x = n$，传播到 x 的使用点 p，但到达定值分析必须保证程序点 p 包含能到达该点的所有对 x 的定值，否则将可能传播错误的值。

2.9 本章小结

数据流分析静态分析数据的使用信息，在编译器中起着至关重要的作用。本章首先介绍了数据流分析的基本概念，并简述了其基本流程。在此基础上，本章深入探讨了不同粒度的数据流分析技术，包括面向基本块的支配节点分析、面向变量的活跃分析、面向表达式的可用表达式和忙碌表达式分析、以及面向语句的到达定值分析。最后，本章还讨论了对上述数据流分析方法的改进和优化，并对它们进行了总结和比较。

2.10 深入阅读

数据流分析作为重要的程序分析技术之一，起源于 20 世纪 60 年代初 Vyssotsky 在贝尔实验室进行的早期工作 [25]。编

译器相关书籍 [1, 8, 13]，对后续发展出的许多数据流分析算法，都有详尽的描述。

支配节点的概念在编译器优化中扮演着重要角色，该概念最早由 Prosser 在 1959 年提出，但并未提供计算算法 [22]。Lowry 和 Medlock 在其编译器中实现了一种时间复杂度为 $O(N^2)$ 的支配节点计算算法 [16]。随后，有大量的相关研究致力于发展支配节点的高效计算算法 [23, 15, 2, 7]，这些工作使得计算大规模有向图的支配节点成为可能。

Kennedy[12, 11] 最早提出了活跃分析，Cocke[6] 和 Ullman[24] 提出了可用表达式，Morel 和 Renvoise[17] 提出了忙碌表达式（预期执行表达式），Allen [2, 9] 提出了到达定值分析。如今这些分析已经作为数据流分析的经典算法，在许多编译原理教材中 [1, 3, 5] 都有讨论。

Khedker 和 Dhamdhere[14] 给出了数据流分析的位向量框架的第一个形式化定义，后续文献 [1, 3, 18, 10, 19, 21, 13] 进行了更详细的讨论。文献 [1, 3] 给出了关于基本块粒度的数据流分析，讨论正确性证明。

Bodik 等 [4] 讨论了数据流分析的 May-Must，并将它们用于完全的冗余消除。文献 [14, 20] 对 May-Must 进行了更进一步的讨论。

2.11 思考题

1. 请结合图2.1，解释后向数据流分析的执行过程。

2. 使用算法2.1，计算出图2.10 中各节点的支配节点集合。

3. 在支配节点计算的迭代算法中，如何通过优化策略（如逆

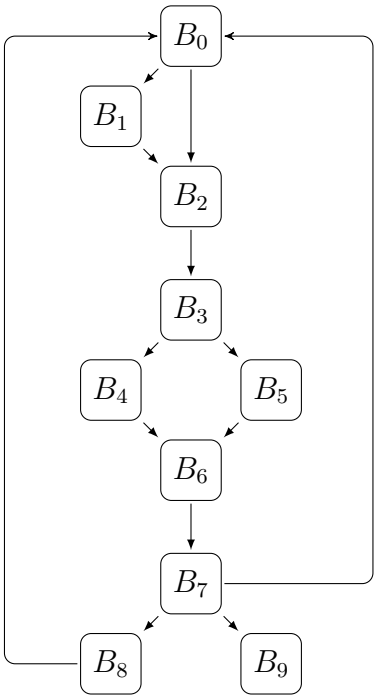


图 2.10: 示例控制流图

后序遍历等）提高算法的效率？探讨这些优化策略的理论基础及其在实际编译器中的应用效果。

4. 请证明对于控制流图中的节点 n，若其严格支配节点集合 $Sdom[n] \neq \emptyset$, 则存在唯一的节点 $m \in Sdom[n]$, 使得 $Sdom[n]$ 中所有的节点都支配节点 m。

5. 给定图2.11所示的控制流图，使用本章介绍的五种数据流分析算法对它进行分析，给出每次迭代后的结果。

6. 使用位向量并以基本块为粒度，对图2.11中的控制流图再次进行本章介绍的五种数据流分析，并给出每次迭代后的结果。对比算法执行效率的变化。

7. 使用拓扑排序，对图2.11中的控制流图再次进行本章介绍的五种数据流分析，并给出每次迭代后的结果。对比算法执行效率的变化。

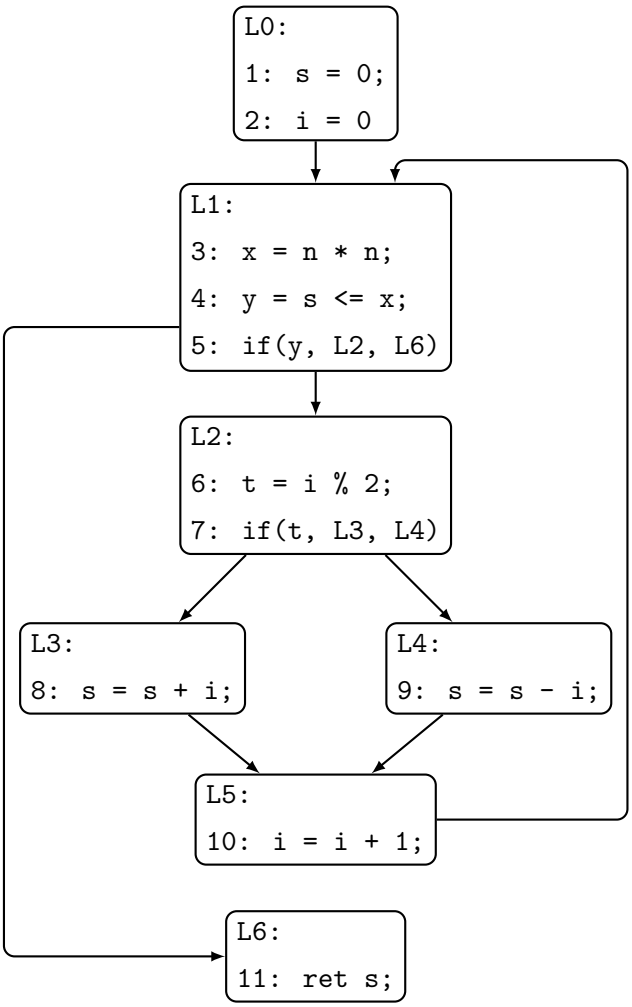

L0:
1: s = 0;
2: i = 0
L1:
3: x = n * n;
4: y = s <= x;
5: if(y, L2, L6)
L2:
6: t = i % 2;
7: if(t, L3, L4)
L3:
8: s = s + i;
L4:
9: s = s - i;
L5:
10: i = i + 1;
L6:
11: ret s;

图 2.11: 示例控制流图

参考文献

[1] Alfred V. Aho and Alfred V. Aho, editors. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley, 2nd ed edition.

[2] Frances E Allen. Control flow analysis. 5(7):1 – 19.

[3] Andrew W Appel. Modern Compiler Implementation in ML. Cambridge university press.

[4] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. 33(5):1 – 14.

[5] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. 17(2):181 – 196.

[6] John Cocke. Global common subexpression elimination. 5(7):20 – 24.

[7] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm.

[8] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Elsevier/Morgan Kaufmann, 2nd ed edition.

[9] given-i=FrancesE family=Allen, given=FrancesE. A basis for program optimization.

[10] Matthew S. Hecht. Flow Analysis of Computer Programs. Elsevier Science Inc.

[11] K. W. Kennedy. Node listings applied to data flow analysis. In Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '75, pages 10 – 21. ACM Press.

[12] Ken Kennedy. A global flow analysis algorithm. 3(1-4):5 – 15.

[13] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. Data Flow Analysis: Theory and Practice. CRC Press.

[14] Uday P Khedker and Dhananjay M Dhamdhere. A generalized theory of bit vector data flow analysis. 16(5):1472 – 1511.

[15] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. 1(1):121 – 141.

[16] Edward S Lowry and Cleburne W Medlock. Object code optimization. 12(1):13 – 22.

[17] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. 22(2):96 – 103.

[18] Steven Muchnick. Advanced Compiler Design Implementation. Morgan kaufmann.

[19] Steven S Muchnick and Neil D Jones. Program Flow Analysis: Theory and Applications, volume 196. Prentice-Hall Englewood Cliffs.

[20] Anders Møller and Michael I Schwartzbach. Static program analysis.

[21] Flemming Nielson, Hanne R Nielson, and Chris Hankin. Principles of Program Analysis. springer.

[22] Reese T Prosser. Applications of boolean matrices to the analysis of flow diagrams. In Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference, pages 133 – 138.

[23] Robert Tarjan. Testing flow graph reducibility. In Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, pages 96 – 107.

[24] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. 2(3):191 – 213.

[25] Victor Vyssotsky and Peter Wegner. A graph theoretical fortran source language analyzer.