第 4 章

格与单调框架

在本章中，我们将讨论格理论及其在数据流分析中的应用。首先，我们通过讨论一个称作零值分析的具体实例，引入以格理论研究数据流分析的动机。接下来，我们给出格理论的严格定义，并讨论在程序分析中非常重要的几类具体格，包括幂集格、平坦格、乘积格和映射格。接着，我们以格为工具，分析数据流方程解的单调性以及用于求解的高效不动点算法。随后，我们讨论单调框架，将其作为求解数据流分析问题的一般性框架，并讨论如何通过将问题对应的格和转移函数作为参数传入单调框架，以实现对多种数据流方程的统一求解。最后，我们将基于单调框架，重新刻画数据流分析问题，给出包括活跃分析、可用表达式、忙碌表达式、以及到达定值分析等几类重要数据流分析的单调框架描述，这为刻画其它的数据流分析问题奠定了统一基础。

4.1 零值分析

为说明引入格进行数据流分析的研究动机，我们考虑一种特定的程序分析—*零值分析* （zero analysis）。零值分析尝

试分析程序中变量（以及表达式）是否可能取值为零，分析的结果在很多程序分析和优化场景中都具有重要作用，例如，我们可以利用该分析，来判定作为除数的表达式是否有可能为零值，从而静态检测程序中可能的“除零错”。

作为示例，考虑如下程序 不难静态分析判断：无论程序

```
1  int f(int n){
2    int x, y, z, k;
3    x = 3;
4    y = 5;
5    if(n==42)
6      z = x;
7    else
8      z = y;
9    k = 1/z;
10   return k;
11 }
```

图 4.1: 进行零值分析的示例程序

第 5 行中的条件判断 $n == 42$ 无论是否成立，程序第 9 行除法 $1/z$ 表达式中的除数 z 肯定不为零，因此零值分析算法可以静态判定上述程序不会在运行时触发除零错。

为了进行类似这样的零值分析，由于变量具有无穷的取值（如变量 $z \in (-\infty, +\infty)$），因此静态穷举变量的所有可能取值一般是不现实的。一个自然的想法是定义一个抽象论域，把程序中出现的常量映射到该抽象论域上，从而把在实际论域上的程序分析转换为在该抽象论域上的程序分析问题。例如，对于上述零值分析，我们为任意程序变量 x 定义如下的抽象论域

$$L = \left\{\bot, 0, Z^{-0}, \top\right\},$$

其中符号 $\bot$ 表示变量 x 还未被赋值过或者被赋值了非整型数

据（如除零的结果），符号 0 表示变量 x 具有零值（注意，我们复用了整数集合中的元素 0，请读者根据上下文进行区分），符号 Z^{-0} 表示变量 x 具有除 0 之外所有可能整数值（亦即 Z^{-0} 可视为集合 $Z - \{0\}$），符号 $\top$ 表示变量 x 可能具有任意的整型值。

直观上，任一变量 x 可能具有的抽象值 L，给定了我们关于该变量 x 具体值的“证据量”。例如，抽象值 $\bot$ 表示还没有任何证据，表明该变量 x 具有整型值；而抽象值 0 表示有证据表明变量 x 一定取零值；抽象值 $\top$ 表示发现了不同的证据，表明 x 可以取零或非零的值。因此，按照抽象值信息量的大小，抽象值集合 L 可以组成如下结构：

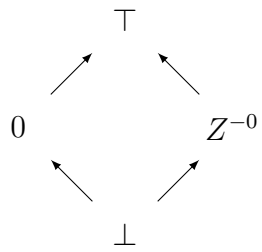

图 4.2: 抽象值 $L = \{\bot, 0, Z^{-0}, \top\}$ 构成的有向结构。

图中的有向边“$\rightarrow$”表示对变量 x 具有的抽象值证据量的增加。非形式的，任意给定两个抽象值 $l_1, l_2 \in L$，我们称在它们上面且离它们最近的值为其“最近公共祖先”。例如，值 $\bot$ 和 0 的最近公共祖先为 0，而值 0 和 Z^{-0} 的最近公共祖先为 $\top$。在第4.2小节，我们将对这里的非形式化讨论以更严格的形式刻画。

基于抽象论域，程序分析关注程序中变量抽象值的流动和变化过程。例如，考虑图 4.1中的示例程序，第 3、4 两行分别表明变量 x 和 y 具有抽象值 Z^{-0}，而无论程序执行条件语句的哪个分支（即第 6 行或第 8 行），在执行第 9 行前，我们

都可以判定变量 z 的抽象值是 Z^{-0}，因此，程序在运行时不会触发除零错。而假设我们将第 3 行的赋值语句改为 $x = 0$，则变量 x 的抽象值是 0，这也使得第 6 行赋值语句将变量 z 的抽象值赋值为 0。当分析进行到第 9 行时，分析算法将看到变量 z 的两个可能抽象值，即 0 和 Z^{-0}，则分析算法需要将这两个抽象值进行合并，以便进行后续语句的分析，而根据图4.2，自然的选择是这两个抽象值的最近公共祖先抽象值 $\top$。最终，程序分析会根据第 9 行代码中变量 z 的抽象值 $\top$，给出一个“潜在除零错”的警告或提示信息。

为了记录零值分析过程中，任意程序变量 x 的抽象值 L 及其变化过程，我们引入一个抽象存储

$$\sigma : x \mapsto L, \tag{4.1}$$

将程序中的每个变量 x，都映射到其对应的抽象值 L。我们用记号 $\sigma(x)$ 表示读变量 x 在抽象存储 σ 对应的抽象值，而用记号 $\sigma[x \mapsto v]$ 表示将抽象存储 σ 中变量 x 的抽象值更新为 v。

给定语句 n，我们记 n 前后的抽象存储 σ 分别为 $In[n]$ 和 $Out[n]$，则我们可以用数据流方程

$$In[n] = \bigsqcup_{p \in pred[n]} Out[p] \tag{4.2}$$

$$Out[n] = \mathcal{V}(n, In[n]) \tag{4.3}$$

刻画零值分析问题。其中式4.2给出了前向数据流分析的典型计算过程，即将 n 的所有前驱节点 p 的信息 $Out[p]$，按照函数 $\bigsqcup$ 进行合并。函数 $\bigsqcup(\sigma_1, \ldots, \sigma_n)$ 接受 n 个抽象存储 σ_i，$1 \leq i \leq n$，计算得到一个结果抽象存储

$$\sigma = \{x \mapsto \sigma_1(x) \sqcup \ldots \sqcup \sigma_n(x)\},$$

其中运算规则 $\sqcup$ 由表4.1给出。结合图4.2，不难验证表中的计算规则 $\sqcup$ 计算了图4.2 中任意两个节点的最近公共祖先。

表 4.1: $\sqcup$ 的运算规则。

$\sqcup$	$\bot$	0	Z^{-0}	$\top$
$\bot$	$\bot$	0	Z^{-0}	$\top$
0	0	0	$\top$	$\top$
Z^{-0}	Z^{-0}	$\top$	Z^{-0}	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

式4.3中的数据流方程 $\mathcal{V}(n, In[n])$，接受语句 n 和抽象存储 $In[n]$（即 σ）作为输入，计算得到语句后新的抽象存储作为结果，亦即该函数刻画了对抽象存储的改变。该函数 $\mathcal{V}$ 基于对语句 n 语法形式的归纳进行定义：

$$
\begin{aligned}
\mathcal{V}(x = 0, \sigma) &= \sigma[x \mapsto 0] \\
\mathcal{V}(x = c, \sigma) &= \sigma[x \mapsto Z^{-0}] \quad \text{其中整数常量} c \neq 0 \\
\mathcal{V}(x = y, \sigma) &= \sigma[x \mapsto \sigma(y)] \\
\mathcal{V}(x = y \oplus z, \sigma) &= \sigma[x \mapsto (\sigma(y) \widehat{\oplus} \sigma(z))] \\
\mathcal{V}(x = f(y), \sigma) &= \sigma[x \mapsto \top]
\end{aligned}
$$

其中，对于整型常量 0，函数将变量 x 的抽象值映射为 0（再次说明：0 处于抽象论域 L）；对于非零的整型常量 c，函数将变量 x 的抽象值映射为 Z^{-0}；对于变量 y，函数将变量 x 的抽象值映射为 y 的抽象值 $\sigma(y)$；对于函数调用 $f(y)$，函数将变量 x 的抽象值映射为抽象值 $\top$，表明函数的返回值可能等于任意的（零或非零）整型值。

对于二元运算赋值语句 $x = y \oplus z$，函数首先计算表达式 $y \oplus z$ 在抽象论域 L 上的抽象值 $\sigma(y) \widehat{\oplus} \sigma(z)$，并将变量 x 映射

算法 4.1 进行零值程序分析的迭代算法

输入： 控制流图 G

输出： 每个节点的抽象存储 σ

```
1: function zeroAnalysis(G)
2:     for each statement s ∈ G do
3:         In[s] = Out[s] = {x_1 ↦ ⊥, ..., x_n ↦ ⊥}
4:     In[s_0] = {x_1 ↦ ⊤, ..., x_n ↦ ⊤}
5:     while some In[] or Out[] still changed do
6:         for each s ∈ G do
7:             In[s] = ⨆_{p∈pred[s]} Out[p]
8:             Out[s] = 𝒱(s, In[s])
```

到该抽象值。抽象运算 $\widehat{\oplus}$ 按二元运算 $\oplus$ 的语法形式归纳定义。例如，对于整数加法 $+$，其对应的抽象加法 $\widehat{+}$ 定义在表4.2 中给出。表中计算规则的定义可以通过其涉及的抽象值的含义进行理解，例如，计算规则 $Z^{-0} \,\widehat{\oplus}\, Z^{-0} = \top$ 表明两个非零值相加可能得到零或非零值（亦即 $\top$）。

表 4.2: 抽象加法 $\widehat{+}$ 的运算规则

$\widehat{+}$	$\bot$	0	Z^{-0}	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	Z^{-0}	$\top$
Z^{-0}	$\bot$	Z^{-0}	$\top$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

减法、乘除法等其它二元运算相应的抽象定义规则与此类似，我们将其作为练习留给读者完成。

我们可根据上述数据流方程，实现静态分析算法。算法4.1 中的分析函数 `zeroAnalysis` 给出了基于迭代的分析算法。该算法首先将每个程序点的抽象存储都初始化为

$$\{x_1 \mapsto \bot, \ldots, x_n \mapsto \bot\}\,,$$

即程序中的每个变量 x_i，$1 \leq i \leq n$，都映射到抽象值 $\perp$，代表还没有任何证据表明变量被赋值过。算法将函数入口语句 s_0 的 $In[s_0]$ 初始化为

$$\{x_1 \mapsto \top, \ldots, x_n \mapsto \top\},$$

该抽象存储代表 s_0 一定被执行，且此时变量具有任意可能的整型值 $\top$（具体地，函数局部变量未被初始化、而函数可能具有任意实参值）。

然后，算法开始进行迭代，分别计算每个程序点 i，$1 \leq i \leq n$，的抽象存储 $In[i]$ 和 $Out[i]$，直到所有程序点上的抽象存储都不再变化为止。

把算法4.1应用到示例 4.1，在第一轮循环结束后，每个语句的 $In[]$、$Out[]$ 值如表4.3 所示。例如，对于语句 $x = 3$ 来说

$$In[x = 3] = \{x \mapsto \top, y \mapsto \top, z \mapsto \top, k \mapsto \top\},$$

执行赋值操作之后，x 的值为非零值 3，将变量 x 映射到抽象值 Z^{-0}，其他变量的抽象值保持不变，因此

$$Out[x = 3] = \left\{x \mapsto Z^{-0}, y \mapsto \top, z \mapsto \top, k \mapsto \top\right\}。$$

算法继续执行，所有的抽象存储 σ 不再改变，算法执行终止。

在结束本节前，我们讨论该算法的一个重要性质—终止性。为此，我们考察数据流方程中的两个等式4.2和 4.3，尝试证明其单调性，即它们都使得等式左侧的项单调增大。这可以通过证明两个子结论完成：（1）这两个方程都分别使得其左侧的赋值结果（即抽象存储 σ）单调递增；（2）赋值结果都有上界。

在本小节，我们先给出直观的证明思路，而把对严格证明的讨论留到下一个小节。首先，我们先考察等式4.2，从表4.1中

表 4.3: 对示例程序进行零值分析的第一轮结果

s	语句 s	$In[s]$ $Out[s]$
1	$x = 3;$	$\{n \mapsto \top, x \mapsto \top, y \mapsto \top, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto \top, z \mapsto \top, k \mapsto \top\}$
2	$y = 5;$	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto \top, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$
3	`if`$(\ldots)$	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$
4	$z = x;$	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto Z^{-0}, k \mapsto \top\}$
5	`else`	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$
6	$z = y;$	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto \top, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto Z^{-0}, k \mapsto \top\}$
7	$k = 1/z;$	$\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto Z^{-0}, k \mapsto \top\}$ $\{n \mapsto \top, x \mapsto Z^{-0}, y \mapsto Z^{-0}, z \mapsto Z^{-0}, k \mapsto \top\}$

的计算规则可以看到，运算 $\bigsqcup$ 总是使得计算结果单调增大，这就证明了结论（1）。（我们把对该结论的详细分析，作为练习留给读者完成。）其次，注意到抽象存储 σ 有平凡的上界

$$\{x_1 \mapsto \top, \ldots, x_n \mapsto \top\}。$$

这就说明了等式4.2的计算能够终止。

同理，我们可以考察等式4.3，并证明其单调性和有界性（我们把详细过程作为练习，留给读者完成）。综上，由这两个等式4.2和 4.3的单调性和有界性，算法能够执行终止。

4.2 格

4.2.1 格的定义

我们可以用格理论统一表示类似上述零值分析中引入的抽象论域 L，为此，我们首先给出

定义 4.1 (偏序) 集合 S 上的偏序 $\sqsubseteq$ 是 $S \times S$ 上的一个二元关系，满足三个条件：

1. 自反性，即对于所有元素 $x \in S$，都有 $x \sqsubseteq x$；
2. 传递性，对于所有元素 $x, y, z \in S$，若 $x \sqsubseteq y$ 且 $y \sqsubseteq z$，则 $x \sqsubseteq z$；
3. 反对称性，对于所有元素 $x, y \in S$，若 $x \sqsubseteq y$ 且 $y \sqsubseteq x$，则 $x = y$。

通常，我们将偏序关系表示为二元组 $(S, \sqsubseteq)$，但有时也简单的直接称具有偏序关系的集合 S 为偏序集合 S。

偏序关系自然刻画了很多实际问题中的二元关系。

例 4.2 集合的包含关系 $\subseteq$ 是偏序关系。

首先，对于任何集合 A，不难验证都有 $A \subseteq A$（自反性）。其次，对于任何集合 A, B, C，如果 $A \subseteq B$ 且 $B \subseteq C$，那么 $A \subseteq C$（传递性）。最后，对于任何集合 A 和 B，如果 $A \subseteq B$ 和且 $B \subseteq A$，那么 $A = B$（反对称性）。

例 4.3 整数集 Z 上的 $\leq$ 关系是偏序关系。

首先，对于任何整数 z，都有 $z \leq z$（自反性）。其次，对于任何三个整数 z_1, z_2, z_3，如果 $z_1 \leq z_2$ 且 $z_2 \leq z_3$，那么 $z_1 \leq z_3$

(传递性)。最后，对于任何两个整数 z_1 和 z_2，如果 $z_1 \leq z_2$ 且 $z_2 \leq z_1$，那么 $z_1 = z_2$（反对称性)。

例 4.4 零值分析中图4.2中抽象值集合 L 上的 $\rightarrow$ 是偏序关系。

我们请读者自行验证该结论。

定义 4.5 (上界与下界) 给定偏序集 $(L, \sqsubseteq)$，且集合 $S \subseteq L$，则元素 $x \in L$ 是 S 的一个上界，当且仅当对所有 $y \in S$，都有 $y \sqsubseteq x$。同理，元素 $x \in L$ 是 S 的一个下界，当且仅当对所有 $y \in S$，都有 $x \sqsubseteq y$。

需要注意的是，上面的定义不要求集合 S 的上界（或下界）在集合 S 本身中。

定义 4.6 (最小上界与最大下界) 我们称元素 x 是集合 S 的最小上界（least upper bound，lub)，如果 x 是 S 的上界，并且对于 S 的所有其他上界 y，都有 $x \sqsubseteq y$。我们可将集合 S 的最小上界记为 $\bigsqcup S$，也称为 S 的并。

我们称元素 x 是集合 S 的最大下界 (greatest lower bound，glb)，如果 x 是 S 的下界，并且对于 S 的所有其他下界 y，都有 $y \sqsubseteq x$。我们可将集合 S 的最大下界记为 $\bigsqcap S$，也称为 S 的交。

特别的，如果集合 S 只包含两个元素，即 $S = \{s_1, s_2\}$，则我们经常将最小上界或最大下界简写为中缀算符。例如，对于最小上界，我们可将 $\bigsqcup S$ 写作 $s_1 \sqcup s_2$，表示两个元素 s_1 和 s_2 的最小上界。

定义 4.7 (格) 一个偏序集 $(L, \sqsubseteq)$ 是一个格，当且仅当对于任意两个元素 $x, y \in L$，$x \sqcup y$ 和 $x \sqcap y$ 都存在。

根据格的定义，我们图4.3中给出了几个格的示例；而在

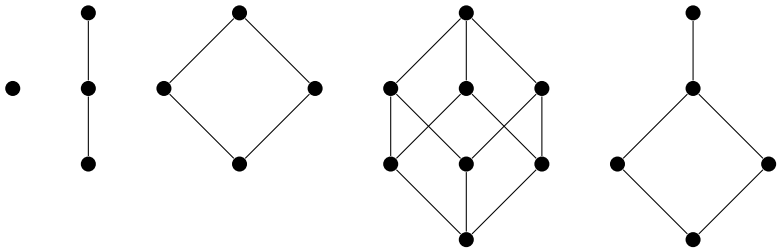

图 4.3: 格的示例图

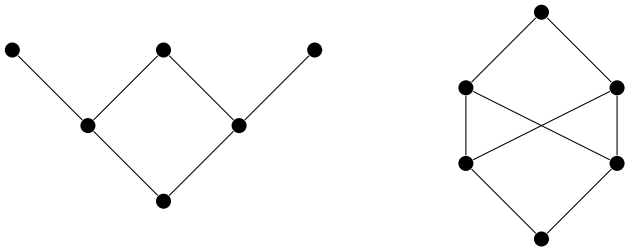

图 4.4: 格的反例图

图4.4中给出了几个格的反例。

定义 4.8 (完全格) 一个偏序集 $(L, \sqsubseteq)$ 是一个完全格，当且仅当，对于 L 的每个子集 $S \subseteq L$，$\bigsqcup S$ 和 $\bigsqcap S$ 都存在。

特别注意，每个完全格一定是一个格。但由于完全格要求每个子集（包括无限子集）都存在最小上界和最大下界，因此格未必是完全格。

例 4.9 偏序集 $(Z, \leq)$ 在整数的小于等于关系 $\leq$ 下是一个格，因为对应任意两个元素 $x, y \in Z$，我们都有

$$x \sqcup y = \max(x, y),$$
$$x \sqcap y = \min(x, y)。$$

然而，偏序集 $(Z, \leq)$ 不是完全格，因为如果我们取自然数集 $N \subseteq Z$，显然 N 不存在最小上界。

给定完全格 $(L, \sqsubseteq)$，显然 $\bigsqcup L$ 和 $\bigsqcap L$ 都存在，我们称 $\bigsqcup L$ 为顶元（top），记为 $\top$，称 $\bigsqcap L$ 为底元（bottom），记为 $\bot$。

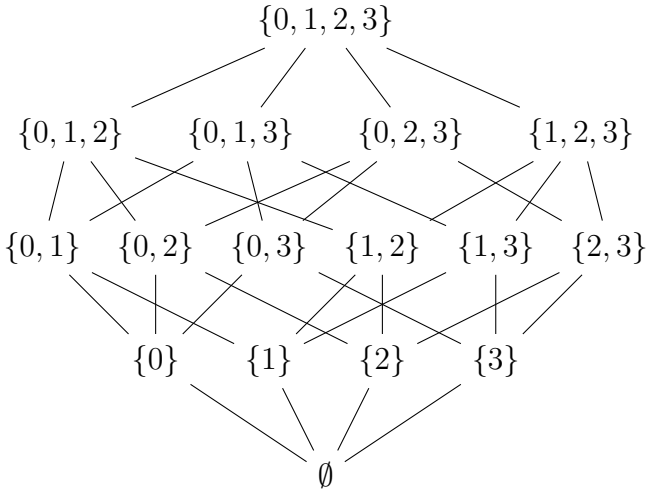

图 4.5: 集合 $\{0, 1, 2, 3\}$ 构成的幂集格示意图

格的高度 $height(L)$ 是指从底元 $\bot$ 到顶元 $\top$ 的最长路径的长度。例如，图 4.3中各个完全格的高度分别为 0、2、2、3 和 3。

接下来我们会讨论在程序分析中常用的几种格结构，包括幂集格、平坦格、乘积格和映射格等。

4.2.2 幂集格

幂集格（powerset lattice）是由一个有限集 A 的所有子集所构成的完全格，记作 $(\mathcal{P}(A), \subseteq)$，其中 $\mathcal{P}(A)$ 表示集合 A 的幂集，即 A 所有子集构成的集合。在幂集格中，底元 $\bot$ 是空集 $\emptyset$，顶元 $\top$ 是全集 A，两个元素的最小上界是它们的并集，即 $x \sqcup y = x \cup y$，而最大下界是它们的交集，即 $x \sqcap y = x \cap y$。幂集格的高度等于集合 A 的元素个数，即 $height(\mathcal{P}(A)) = |A|$。

例 4.10 给定集合 $A = \{0, 1, 2, 3\}$，其幂集格如图4.5所示。该幂集格中包括集合 A 的所有子集，高度为 4。

类似地，我们还可以定义反幂集格（reverse powerset lat-

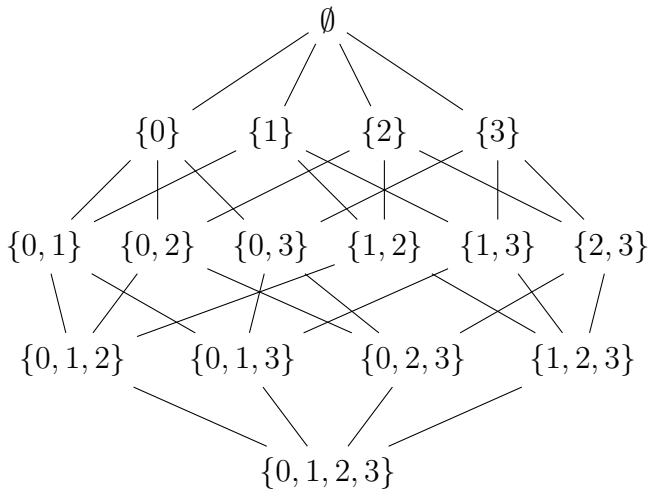

图 4.6: 集合 $\{0,1,2,3\}$ 构成的反幂集格示意图

tice），表示为 $(\mathcal{P}(A),\supseteq)$，即将偏序关系反转为 $\supseteq$。在反幂集格中，格的底元 $\perp$ 是全集 A，而顶元 $\top$ 是空集 $\emptyset$。两个元素的最小上界是它们的交集，即 $x \sqcup y = x \cap y$，而最大下界是它们的并集，即 $x \sqcap y = x \cup y$。幂集格的高度等于集合 A 的元素个数，即 $height(\mathcal{P}(A)) = |A|$。

例 4.11　给定集合 $A = \{0,1,2,3\}$，其反幂集格如图4.6所示。该幂集格中包括集合 A 的所有子集，高度为 4。

4.2.3 平坦格

平坦格（flat lattice）是一类高度为 2 的完全格，其底元为 $\perp$，顶元为 $\top$，中间为集合中若干互相不存在偏序关系的元素。一般的，对于集合 $A = \{a_1, a_2, \ldots, a_n\}$，它的平坦格如图4.7所示。

例 4.12　零值分析中格 $L = \left\{\perp, 0, Z^{-0}, \top\right\}$（图4.2）构成了一个平坦格。

例 4.13　整数集合 Z 可构成了一个如图4.8 所示的无限平坦格。常量传播优化经常使用该格，来分析始终具有常量值的

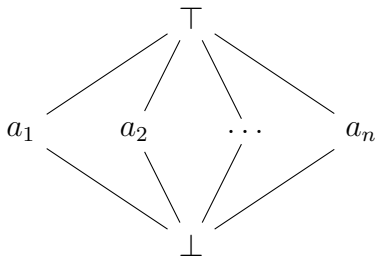

图 4.7: 集合 $A = \{a_1, \ldots, a_n\}$ 构成的平坦格示意图

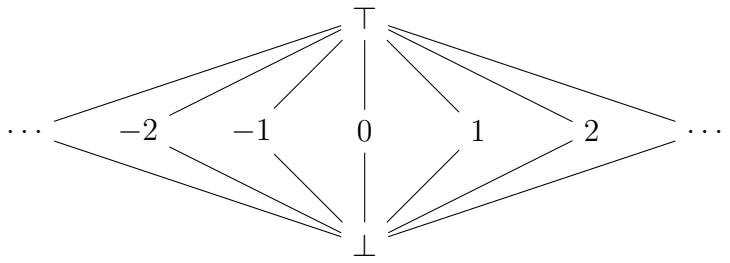

图 4.8: 整数集合 Z 构成的平坦格示意图

变量。

4.2.4 乘积格

给定 n 个完全格 $L_1, L_2, \ldots, L_n$，则它们的乘积（笛卡尔积）

$$L_1 \times L_2 \times \ldots \times L_n$$

也是一个完全格，称为**乘积格**（product lattice）。乘积格中的每个元素是由各个格中元素组成的 n 元组，即

$$L_1 \times L_2 \times \ldots \times L_n = \{(x_1, x_2, \ldots, x_n) \mid x_i \in L_i\},$$

n 元组 $(x_1, x_2, \ldots, x_n)$ 之间的偏序关系 $\sqsubseteq$，由元组各分量 x_i 的偏序关系 $\sqsubseteq_i$，$1 \leq i \leq n$，决定，即

$$(x_1, x_2, \ldots, x_n) \sqsubseteq (x'_1, x'_2, \ldots, x'_n) \iff x_i \sqsubseteq_i x'_i,\ 1 \leq i \leq n。$$

特别的，相同的 n 个完全格 L 的乘积可以简洁地表示为

$$L^n = \underbrace{L \times L \times \ldots \times L}_{n}。$$

乘积格的高度取决于参与乘积的各个格的高度，且总高度是各个格的高度之和，即

$$height(L_1 \times \ldots \times L_n) = \sum_{i=1}^{n} height(L_i)。$$

4.2.5 映射格

给定集合 $A = \{a_1, a_2, \ldots\}$ 和一个完全格 $L = \{x_1, x_2, \ldots\}$，映射格（map lattice）由如下映射

$$A \to L = \{[a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots]\}$$

构成，即映射格 $A \to L$ 中的每个元素是从集合 A 到完全格 L 的一个函数。映射格上的偏序关系 $\sqsubseteq_{A \to L}$ 由集合 A 中任意元素 x 的函数值 $(A \to L)(x)$ 在完全格 L 上的偏序 $\sqsubseteq_L$ 确定，即

$$f \sqsubseteq_{A \to L} g \iff \forall a \in A : f(a) \sqsubseteq_L g(a) \quad 其中 \quad f, g \in A \to L。$$

映射格的高度取决于集合 A 的大小和完全格 L 的高度，且 $height(A \to L) = |A| \times height(L)$。

例 4.14 零值分析中使用的抽象存储 σ（见定义4.1）实际上是映射格 $X \to L$，它将程序中的变量集合 X 映射到抽象值构成的格 L。

4.3 单调性与不动点

在数据流分析一章中，我们使用各种数据流方程和基于迭代的不动点算法，来计算控制流图中各个节点的数据流信

息，值得注意的是，不同的数据流分析尽管计算的具体数据流信息不同，但它们具有类似的结构。在本节以及之后的几个小节中，我们将利用格理论给它们以统一的刻画，并进一步给出不动点算法作为计算数据流信息的更高效的一般性方法。

作为一个启发性的例子，我们来对下面一个简单的程序进行零值分析：

```
1 a = 10;
2 b = a + input();
3 c = a - b;
```

类似于4.1节，我们为每一条语句 n 的入口和出口引入一个抽象存储 σ，分别记为 $In[n]$ 和 $Out[n]$，并使用一组等式描述每个抽象状态的计算过程：

$$In[1] = [a \mapsto \top, b \mapsto \top, c \mapsto \top] \tag{4.4}$$

$$Out[1] = In[1][a \mapsto Z^{-0}] \tag{4.5}$$

$$In[2] = Out[1] \tag{4.6}$$

$$Out[2] = In[2][b \mapsto In[2](a) \widehat{+} \top] \tag{4.7}$$

$$In[3] = Out[2] \tag{4.8}$$

$$Out[3] = In[3][c \mapsto In[3](a) \widehat{-} In[3](b)] \tag{4.9}$$

在进行数据流分析的过程中,每个抽象存储 $In[i]$、$Out[n]$,$1 \leq i \leq 3$，都可以看成一个变量，则上面 6 个等式则组成了一个方程组，方程组的解便是满足这些约束的数据流值。这就为我们进行数据流分析提供了一种新的思路：利用抽象状态和方程组来为每个程序点进行建模，通过解方程组得到数据流值。

一般的，求解一个方程组可能存在无解、有唯一解或有

多个解等几种情况，而在数据流分析时，我们通常希望能够得到尽可能精确的数据流结果，这意味着如果有多个解的话，我们希望在这些解中找到“最精确”的一个。为此，我们接下来将讨论如何利用格抽象，来给出对类似上述的数据流方程的统一的求解方法，从而得到最精确数据流值。

定义 4.15 (单调性) 设 L_1 和 L_2 都是格，我们称函数 $f : L_1 \to L_2$ 是单调的（monotone），如果对任意 $x, y \in L_1$, 若 $x \sqsubseteq y$ 成立，则 $f(x) \sqsubseteq f(y)$ 成立。

例 4.16 设格 L 是如图4.5所示的幂集格，函数

$$f(x) = x \cup \top$$

是该格 L 到自身上的一个函数（注意，$\top = \{0, 1, 2, 3\}$）。则根据单调性的定义，显然函数 f 是单调的。不难验证，另一个函数

$$g(x) = \top - x$$

不是单调的。

在数据流分析中，格的偏序可理解为表示信息的精度，所以直观上，函数的单调性意味着函数会对更精确的输入产生更精确的输出，亦即函数会始终将函数值朝格的“顶元”推动。

单调性的定义可以自然推广到具有多个参数的函数。例如，给定有两个参数的函数 $f : L_1 \times L_2 \to L_3$，其中 L_1、L_2 和 L_3 都是格。如果对任意的 $x_1, x_2 \in L_1$ 和 $y_1, y_2 \in L_2$，每当 $(x_1, y_1) \sqsubseteq (x_2, y_2)$，都有 $f(x_1, y_1) \sqsubseteq f(x_2, y_2)$，则函数 f 是单调的。

定义 4.17 (不动点与最小不动点) 设 $x \in L$, 如果 $f(x) = x$，那么我们说 x 是函数 f 的一个不动点 (fixed point)。特别的，如果对 f 的其他不动点 y，都满足 $x \sqsubseteq y$，那么称 x 为函数 f 的

最小不动点（least fixed point）。

例 4.18 设 f 是图4.5所示幂集格上的一个函数

$$f(x) = x \cup \top,$$

则 $\top$ 是函数 f 的一个不动点，且是其最小不动点。

设 L 是一个完全格，L 上的一个方程组具有以下形式：

$$\begin{aligned} x_1 &= f_1(x_1, x_2, \ldots, x_n) \\ x_2 &= f_2(x_1, x_2, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, x_2, \ldots, x_n) \end{aligned}$$

其中变量 $x_1, \ldots, x_n \in L$，且函数 $f_1, \ldots, f_n : L^n \to L$ 被称为约束函数（constraint function）。一个方程的解提供了一个来自格 L 的值，使得所有的方程都成立。

我们可以将 n 个函数组合成一个函数 $f : L^n \to L^n$：

$$f(x_1, \ldots, x_n) = (f_1(x_1, \ldots, x_n), \ldots, f_n(x_1, \ldots, x_n))。$$

这时原方程组可以写成

$$x = f(x),$$

其中 $x \in L^n$。这表明，方程组的解就是它所用约束函数的不动点。

例 4.19 考虑本节开头我们建立的方程组

$$In[1] = [a \mapsto \top, b \mapsto \top, c \mapsto \top] \tag{4.10}$$

$$Out[1] = In[1][a \mapsto Z^{-0}] \tag{4.11}$$

$$In[2] = Out[1] \tag{4.12}$$

$$Out[2] = In[2][b \mapsto In[2](a) \widehat{+} \top] \tag{4.13}$$

$$In[3] = Out[2] \tag{4.14}$$

$$Out[3] = In[3][c \mapsto In[3](a) \widehat{-} In[3](b)] \tag{4.15}$$

其中，$In[1]$，…，$Out[3]$ 是映射格 $L' = X \to L$ 上的变量。通过引入合适的函数记号 f_1，…，f_6：$L'^6 \to L'^6$（例如，$f_2(In[1], \ldots, Out[3]) = In[1][a \mapsto Z^{-0}]$），我们可将上述等式组写成：

$$In[1] = f_1(In[1], \ldots, Out[3]) \tag{4.16}$$

$$Out[1] = f_2(In[1], \ldots, Out[3]) \tag{4.17}$$

$$In[2] = f_3(In[1], \ldots, Out[3]) \tag{4.18}$$

$$Out[2] = f_4(In[1], \ldots, Out[3]) \tag{4.19}$$

$$In[3] = f_5(In[1], \ldots, Out[3]) \tag{4.20}$$

$$Out[3] = f_6(In[1], \ldots, Out[3]) \tag{4.21}$$

于是我们可以通过求解函数 $f = (f_1, ..., f_6)$ 的一个不动点，来得到方程组的一个解。

通常情况下，我们需要的是方程组的一个最精确的解，这时我们需要求方程组的最小不动点。

定理 4.20 (不动点（fixed-point）) *在一个高度有限的完全格 L 中，每一个单调函数 $f : L \to L$ 都有唯一一个最小不动点 $lfp(f)$：*

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot),$$

其中 $f^i(\bot) = f(f^{i-1}(\bot)) = f(f(f^{i-2}(\bot))) = ...$。

证明. 证明分为三步。(1) 先证明函数 f 不动点的存在性。由于 $\bot$ 是格的底元，显然有 $\bot \sqsubseteq f(\bot)$。又因为函数 f 是单调

的，所以可以得到 $f(\bot) \sqsubseteq f(f(\bot)) = f^2(\bot)$，通过归纳可得对于任意 $i \geq 0$，有 $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$，于是我们可以得到一个递增链

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \sqsubseteq f^i(\bot) \sqsubseteq f^{i+1}(\bot) \sqsubseteq \ldots$$

又因为格 L 的高度是有限的，所以必然存在某个 k，使得 $f^k(\bot) = f^{k+1}(\bot)$，即 $f^k(\bot)$ 是函数 f 的一个不动点。进一步，由于 $f^k(\bot) = f^i(\bot) \sqcup f^k(\bot)$，$0 \leq i \leq k-1$，则可知 $lfp(f) = f^k(\bot)$。

（2）再证明 $f^k(\bot)$ 是函数 f 的最小不动点。设 x 为函数 f 的任意一个不动点，显然有 $\bot \sqsubseteq x$，根据 f 的单调性，于是有 $f(\bot) \sqsubseteq f(x) = x$，通过归纳可得 $f^k(\bot) \sqsubseteq x$，所以 $f^k(\bot)$ 是 f 的最小不动点。

（3）最后证明最小不动点的唯一性。根据偏序 $\sqsubseteq$ 的反对称性，最小不动点也是唯一的。 证毕

不动点定理给程序分析问题提供了一个统一的结论：只要保证完全格的高度是有限的，并且约束函数是单调的，那么构建在其上的方程组总是有解，并且解总是最小的。

从计算的视角，不动点定理的证明过程也提供了一种计算最小不动点的方法：只需要简单的计算递增链 $\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$，直到到达最小不动点 $f^k(\bot)$，这种计算方法称为**朴素的不动点算法**（native fixed-point algorithm）。

算法4.2给出了朴素不动点算法的实现。算法首先将变量 x 初始化为 $\bot$，然后使用 `while` 循环不断迭代计算递增链，直到 x 与函数值 $f(x)$ 相等为止，即达到了函数 f 的最小不动点。如图 4.9所示，这个过程可以形象的描述为：变量 x 从底元素

算法 4.2 朴素的不动点算法

输入： 方程组的约束函数 f

输出： f 的最小不动点，同时也是方程组最精确的解

```
1: function naiveFixedPoint(f)
2:     x = ⊥
3:     while x ≠ f(x) do
4:         x = f(x)
5:     return x
```

$\perp$ 开始，沿着格不断向上走，直到达到最小不动点 $f^k(\perp)$ 为止。

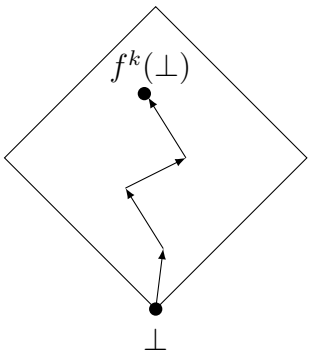

图 4.9: 最小不动点计算过程的形象化描述

朴素不动点算法的时间复杂度主要取决于格 L 的高度和约束函数 f，其中，格 L 的高度决定了 `while` 循环的执行次数，而约束函数决定了每次迭代和条件判断的开销。朴素不动点算法并不考虑格的特殊结构，也不考虑控制流图的前驱和后继等结构信息，只是不断地进行迭代直到不动点为止。因此，在实际的计算中，朴素不动点算法未必高效。在下一节，我们将讨论更加高效的不动点算法。

最小不动点是方程组最精确的可能解，但由于方程组仅仅是实际程序行为的保守近似（回想一下我们在??节中对 May-Must 的分析），这意味着语义上最精确的可能解通常低于格中的最小不动点。例如，对于使用幂集格的数据流分析而言，

如果约束函数 f 使用的是集合并运算 $\cup$（其底元 $\perp = \emptyset$），则其最小不动点是实际数据流结果的超集；而如果约束函数 f 使用的是集合交运算 $\cap$（其底元 $\perp = U$），则其最小不动点是实际数据流结果的子集。

4.4 单调框架

基于前面讨论的格理论，本节我们给出求解数据流分析问题的一般性刻画——单调框架，并给出一种高效的基于工作表的不动点算法的实现。

4.4.1 定义

典型的数据流分析通常接受控制流图和一个高度有限的完全格作为输入，并且，对于需要分析的信息，我们可以定义一个约束函数，将控制流图中的程序信息与格通过描述数据流约束的方程组联系起来。

形式化的，我们给出

定义 4.21 (单调框架) 一个单调框架（monotone framework）$\mathcal{F} = (C, L, f)$，由程序的控制流图 C、有限高度的完全格 L、和单调的约束函数 f 组成。

对于特定的数据流分析，可以通过实例化单调框架 $\mathcal{F}$ 中的完全格 L 和约束函数 f 来完成。

例 4.22 考虑4.3节零值分析的例子，我们可以给定 $\mathcal{F} = (C, L, f)$，其中 C 是带分析的程序，而 L 是映射格，将程序变量 X 映射

为抽象值。分析函数 f

$$In[n] = \bigsqcup_{p \in pred[n]} Out[p]$$
$$Out[n] = \mathcal{V}(n, In[n])$$

将程序点集合 $\{In[1], Out[1], ..., In[3], Out[3]\}$ 中元素，映射为对应的映射格元素 L。不难证明函数 f 是单调的，因此 $\mathcal{F} = (C, L, f)$ 构成了针对零值分析的一个单调框架。

4.4.2 基于工作表的不动点算法

尽管朴素不动点算法可以正确地工作，但由于它可能包括冗余计算，因此执行效率不高。首先，如果约束函数 f 所依赖的输入值 x 没有发生变化，则再次计算 $f(x)$ 会得到相同的结果。例如，在到达定值分析中，如果在某轮迭代中，到达某个基本块 B 的定值集合，和上一轮相比没有变化，则不必对该基本块进行重新计算。其次，所有约束函数 f 类似于并行计算，只会依赖上一次迭代的结果。事实上，如果在函数 f 的计算过程中，考虑信息的依赖性并优先计算被依赖的信息，则会使得不动点收敛得更快。例如，在活跃分析中，由于每个基本块的 $Out[]$ 集合，都依赖于其后继基本块的 $In[]$ 集合，因此，按照控制流图的逆拓扑序，从后继块到当前块计算活跃变量集合，则能使得算法尽快终止。

为了解决上述两类冗余计算的问题，我们给如如下两个技术。首先，为了解决计算顺序的问题，我们显式刻画计算过程中节点间的依赖关系。为此，我们引入依赖函数

$$dep : Node \to \mathcal{P}(Node)$$

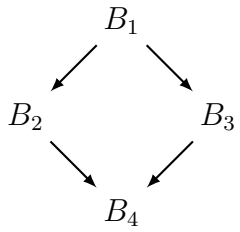

图 4.10: 示例控制流图

来表示依赖于某个节点 n 的节点集合。同理，我们也可以引入 dep 的逆:

$$dep^{-1}(n) = \{v \mid n \in dep(v)\} ,$$

表示节点 n 依赖的节点集合。需要特别注意的是，节点的依赖关系取决于待分析的信息流向，而不是节点在控制流图中的拓扑关系。例如，对于图4.10 所示的控制流图，如果我们对其进行达到定值分析，则有

$$dep(B_1) = \{B_2, B_3\}$$
$$dep^{-1}(B_4) = \{B_2, B_3\} 。$$

反之，如果我们对其进行活跃分析，则有

$$dep(B_4) = \{B_2, B_3\}$$
$$dep^{-1}(B_1) = \{B_2, B_3\} 。$$

其次，为了解决对约束函数 f 的重复计算问题，我们引入显式的工作表，对节点的计算过程做记录。

基于这两个技术，我们在算法4.3中，给出求解最小不动点的一个基于工作表实现。算法接受单调框架 $\mathcal{F} = (C, L, f)$ 作为输入，输出最小不动点。算法首先将抽象值 x_i，$1 \leq i \leq n$，都初始化为平凡的底元 $\perp$，并将所有的控制流图 G 中的节点 v_i，$1 \leq i \leq n$，都加入到工作表 W 中，这意味着控制流图中

算法 4.3 基于工作表的不动点算法

输入： 约束函数 $f_1, \ldots, f_n$

输出： 约束函数的最小不动点

```
1: function simpleWorkList(G, L, f_1, ..., f_n)
2:     (x_1, ..., x_n) = (⊥, ..., ⊥)
3:     W = {v_1, ..., v_n}
4:     while W ≠ ∅ do
5:         v_i = remove(W)
6:         y = f_i(x_1, ..., x_n)
7:         if y ≠ x_i then
8:             x_i = y
9:             for each v_j ∈ dep(v_i) do
10:                if v_j ∉ W then
11:                    add(W, v_j)
12:    return (x_1, ..., x_n)
```

所有节点上的约束函数 f_i 都需要被计算一次。算法每次迭代都从工作表 W 中取出一个节点 v_i，并利用该节点对应的约束函数 f_i 计算得到结果 y，如果计算结果 y 相对旧的值 x_i 发生了改变，便更新节点的约束变量 x_i 为新的值 y。此时，所有依赖节点 x_i 的节点信息也需要被重新计算，因此，算法会将 $dep(v_i)$ 中不在工作表 W 的节点 v_j 加入到工作表 W 中。

工作表算法一定会终止，因为在每次迭代中，要么抽象值 x_i 沿着格向上移动，要么删除工作表 W 中的一个元素。假设 $|dep(v)|$ 和 $|dep^{-1}(v)|$ 对所有节点都是一个常数（这与控制流图的结构有关，通常是一个很小的常数 $c \leq 2$），设程序控制流图中节点的数量为 n，完全格的高度为 h，且每个约束函数 f 的复杂度为 $O(k)$，由于工作表算法需要为每个节点进行计算，每个节点的抽象状态至多沿着格向上移动 h 次，即至多需要计算 h 次约束函数，所以工作表算法的时间复杂度为 $O(n \times h \times k)$。

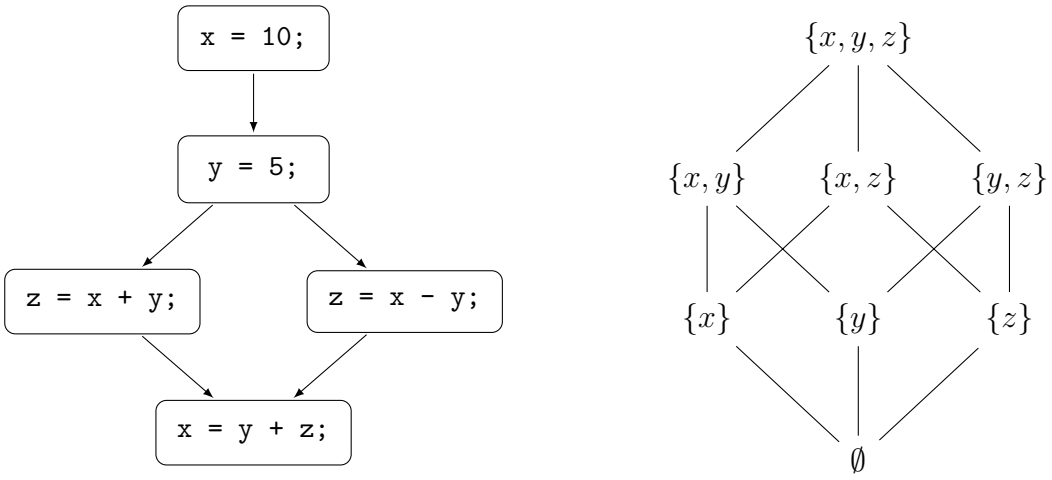

图 4.11: 活跃分析幂集格示例

4.5 基于单调框架的数据流分析

在本节中，我们将使用单调框架的技术，对第??章讨论的典型的数据流分析问题给出统一描述。

4.5.1 活跃分析

在活跃分析中，节点的状态为在当前节点处活跃的变量集合，并且，在交汇节点处，我们用集合并来实现约束函数。因此，我们可以使用程序变量构成的幂集格 L，来描述活跃分析中节点的状态。图4.11给出了一个包含三个变量的程序，及其对应的幂集格。

我们可以将活跃分析所使用的数据流方程

$$Out[n] = \bigcup_{s \in succ[n]} In[s]$$

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

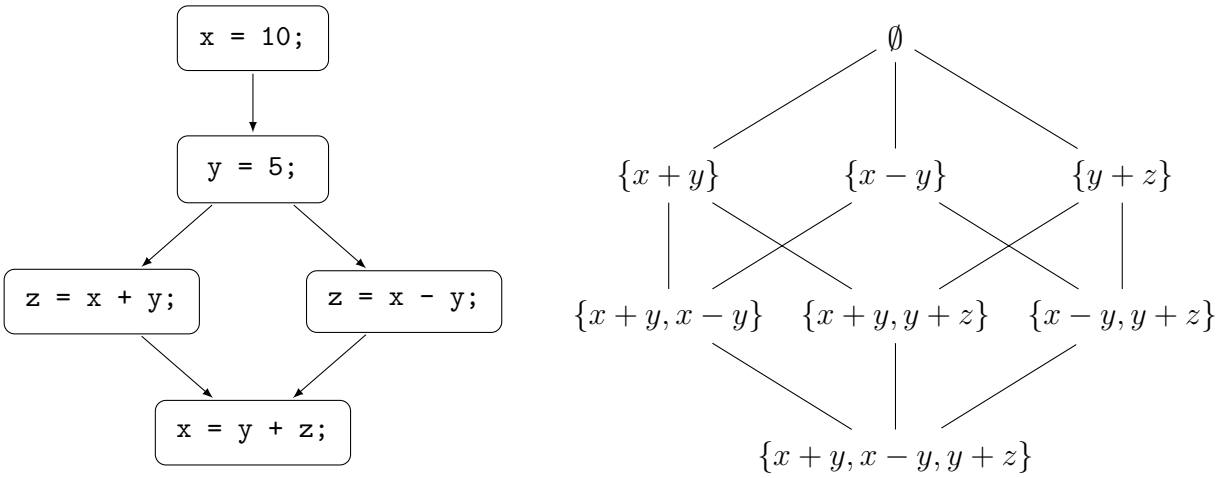

图 4.12: 可用表达式分析反幂集格示例

分别抽象为对应的转移函数

$$f_1(x) = \bigcup_{s \in succ[x]} In[s]$$

$$f_2(x) = Gen \cup (Out[x] - Kill)。$$

我们需要验证函数 f_1 和 f_2 都是单调的。首先，函数 f_1 是单调递增的，因为任何后继节点的 $In[]$ 集合增大，都不可能导致集合求并结果减小。类似的，函数 f_2 也是单调的。综上，活跃分析可以通过单调框架实现。

4.5.2 可用表达式分析

在可用表达式分析中，节点的状态为在当前节点处可用的表达式，并且在节点交汇处，我们用集合交来实现约束函数。因此，我们可以使用程序中由表达式构成的反幂集格 L，来描述可用表达式分析中节点的状态。图4.12给出了一个包含三条表达式的程序，及其对应的可用表达式分析反幂集格。

可用表达式数据流方程可以使用转移函数

$$f_1(x) = \bigcap_{p \in pred[x]} Out[p]$$
$$f_2(x) = Gen \cup (In[x] - Kill)$$

表示。同理,不难验证函数 f_1, f_2 都是单调的。因此 $(C, L, (f_1, f_2))$ 构成了一个单调框架。

4.5.3 忙碌表达式分析

在忙碌表达式分析中，节点的状态为在当前节点处忙碌的表达式，并且在节点交汇处，我们用集合交来实现约束函数。因此，我们可以使用程序中由表达式构成的反幂集格 L，来描述可用表达式分析中节点的状态。图4.12给出了一个包含三条表达式的程序，及其对应的忙碌表达式分析反幂集格。

忙碌表达式数据流方程可以使用转移函数

$$f_1(x) = \bigcap_{s \in succ[x]} In[s]$$
$$f_2(x) = Gen \cup (Out[x] - Kill)$$

进行表示。同理，不难验证函数 f_1，f_2 都是单调的。因此 $(C, L, (f_1, f_2))$ 构成了一个单调框架。

4.5.4 达到定值分析

在到达定值分析中，节点的状态为能够到达该节点处的定值语句，并且，在交汇节点处，需要对集合取并集。因此，我们可以使用程序中包含语句的幂集格 L，来描述到达定值

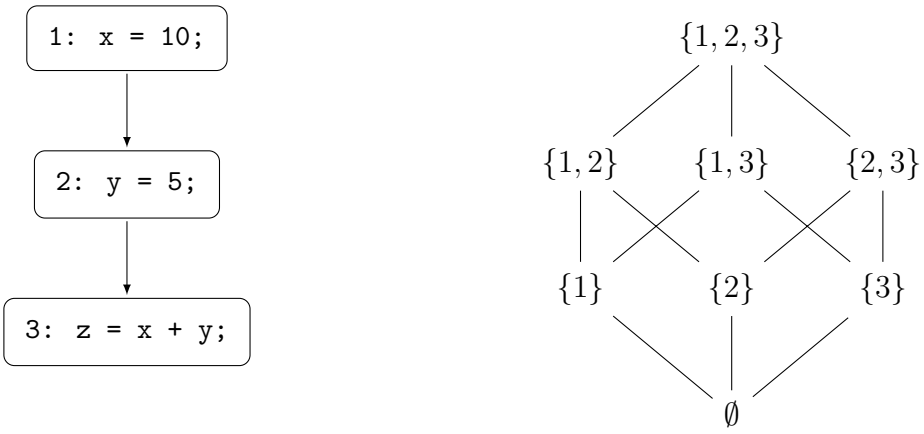

图 4.13: 到达定值分析幂集格示例

分析中节点的状态。图4.13展示了一个包含三条语句的程序，及其对应的到达定值分析幂集格。

到达定值分析数据流方程可以使用转移函数

$$f_1(x) = \bigcup_{p \in pred[p]} Out[p]$$
$$f_2(x) = Gen \cup (In[x] - Kill)$$

表示。同理，不难验证转移函数 f_1 和 f_2 均是单调增大的。因此 $(C, L, (f_1, f_2))$ 构成了一个单调框架。

4.6 深入阅读

格理论的起源可以追溯到 George Boole[3] 在 1854 年提出的布尔代数，将布尔代数作为格的一种特殊类型。之后，Dedekind[1, 5] 在群论研究中引入了类似的“格”概念。现代格理论的形成则始于 1930 年代，Garrett Birkhoff[2] 对格理论进行了系统的定义和推广，使其成为数学和计算机科学中的重要工具。

Kildall、Kam 和 Ullman[9, 8] 的开创性工作中确立了格

与程序分析之间的联系，而 Kam 和 Ullman[8] 给出了单调框架的概念。Graham 和 Wegman[6] 也讨论了流函数的单调性。Hecht[7] 对这些概念进行了详细的论述。

不动点定理可以追溯到上世纪 50 年代，Kleene[10] 原始的不动点定理中，要求函数 f 必须是连续的，格只需满足完全偏序而并不必是完全格。Tarski[12] 的不动点定理构建在完全格之上，并给出了不动点定理的诸多应用。Cousot[4] 给出了 Tarski 不动点定理的构造性证明。Møller 等 [11] 根据节点间的关系，进一步给出了基于工作表的不动点算法。

4.7 本章小结

在本章中，我们首先通过零值分析的具体实例，给出了引入格理论的动机。进而，我们讨论了格理论中的基本概念，并给出了在程序分析中常用的几种重要的格。随后，我们深入讨论了数据流方程的单调性及其求解的朴素不动点算法。在此基础上，我们引入了单调框架的概念，将其作为求解数据流分析问题的一般性方法，并给出了基于工作表的不动点算法，以提高分析的效率。最后，我们应用单调框架对前面讨论过的几种重要的数据流分析，包括活跃分析、可用表达式分析、忙碌表达式分析、以及到达定值分析进行了重构，为其给出了统一描述。

4.8 思考题

1. 在零值分析中，我们给出了抽象二元算符 $\widehat{+}$ 的定义（见表 4.2）。请给出减法 $\widehat{-}$、乘法 $\widehat{*}$、除法 $\widehat{/}$ 等其它抽象二元算符的定义。

2. 图4.3给出了算法 4.1 在示例程序上的第一轮迭代执行的结果，请给出剩余迭代执行的过程和结果。

3. 分析说明表4.1中给定的计算规则具有单调递增性质。回想一下，任意给定的函数 f 具有单调性，当 $x_1 \sqsubseteq x_2$ 时，都有 $f(x_1) \sqsubseteq f(x_2)$。

4. 请分析说明等式4.3的单调性。

5. 试证明，如果 $(S, \sqsubseteq)$ 是一个非空有限格，则 $(S, \sqsubseteq)$ 也是一个完全格。

6. 试证明，如果 A 是有限集且 L 具有有限高度，则从 A 到 L 的映射格的高度为 $height(A \to L) = |A| \cdot height(L)$。

7. 试证明，$x \sqsupseteq y \Leftrightarrow x \sqcup y = x$ 和 $x \sqsubseteq y \Leftrightarrow x \sqcap y = x$，并给出 $\sqsubseteq$ 不等式方程组的等价等式方程组。

8. 选择一种你熟悉的编程语言，根据单调框架的形式化定义，思考如何利用合适的数据结构来实现一个单调框架。并基于你的单调框架实现数据流分析算法。

9. 在??节我们讨论了将基本块进行伪拓扑排序来提高数据流计算的效率，分析比较它和工作表不动点算法有哪些相识之处。你认为那个方法更加高效?

10. 给定图4.14所示的控制流图，使用本章介绍的四种基于单调框架的数据流分析算法对它进行分析。

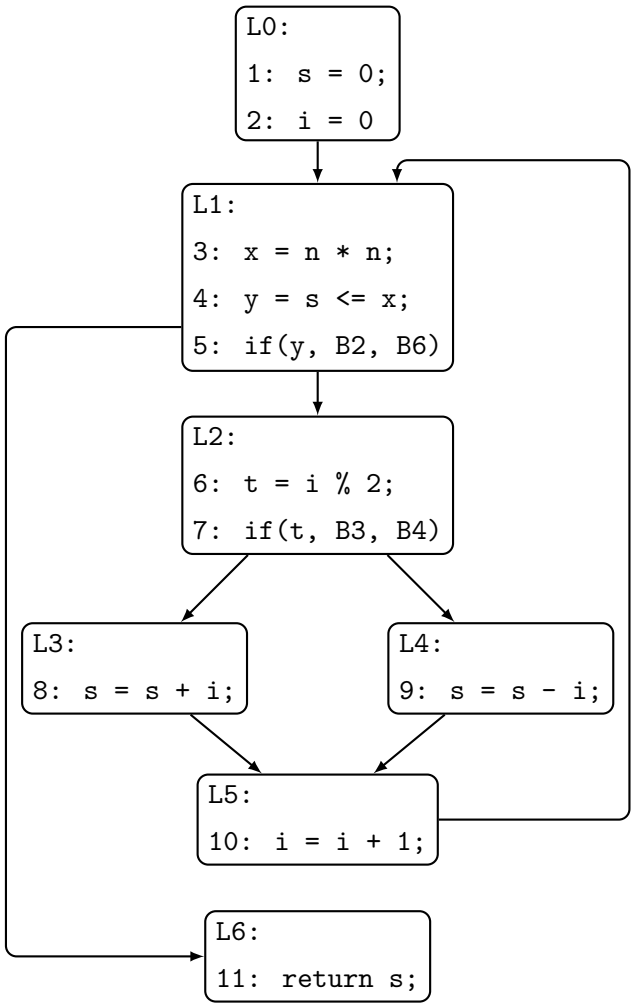
L0:
1: s = 0;
2: i = 0
L1:
3: x = n * n;
4: y = s <= x;
5: if(y, B2, B6)
L2:
6: t = i % 2;
7: if(t, B3, B4)
L3:
8: s = s + i;
L4:
9: s = s - i;
L5:
10: i = i + 1;
L6:
11: return s;

图 4.14: 控制流图

参考文献

[1] Heinr Beckurts and R Dedekind. Über zerlegungen von zahlen durch ihre grössten gemeinsamen theiler. pages 1 – 40.

[2] Garrett Birkhoff. Lattice Theory, volume 25. American Mathematical Soc.

[3] George Boole. An Investigation of the Laws of Thought: On Which Are Founded the Mathematical Theories of Logic and Probabilities, volume 2. Walton and Maberly.

[4] Patrick Cousot and Radhia Cousot. Constructive versions of tarski's fixed point theorems. 82(1):43 – 57.

[5] R. Dedekind. Ueber die von drei moduln erzeugte dualgruppe. 53:371 – 403.

[6] Susan L Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. 23(1):172 – 202.

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

[8] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. 7(3):305 – 317.

[9] Gary A Kildall. A unified approach to global program optimization. In Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 194 – 206.

[10] Stephen Cole Kleene. Introduction to metamathematics.

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

[12] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications.