静态程序分析

Anders Møller和Michael I. Schwartzbach

2017年1月23日

计算机科学系
奥胡斯大学，丹麦
{amoeller,mis}@cs.au.dk

目录

前言

静态程序分析是在不实际运行程序的情况下推理计算机程序行为的艺术。 这不仅对于优化编译器生成高效代码有用，还对于自动错误检测和其他可以帮助程序员的工具有用。 静态程序分析器是一种推理其他程序行为的程序。 对于任何对编程感兴趣的人来说，有什么比编写分析程序更有趣的呢？

正如图灵和赖斯所知，用常见编程语言编写的程序的所有有趣属性在数学上是不可判定的。 这意味着软件的自动推理通常必须涉及近似。 众所周知，测试可以揭示错误，但通常无法证明其不存在。 相比之下，静态程序分析可以 - 在适当的近似条件下 - 检查程序的所有可能执行，并提供关于其属性的保证。 在开发这种分析时面临的关键挑战之一是如何确保高精度和高效性以实际应用。

这些笔记介绍了静态程序分析的原理和应用。 我们涵盖了基本类型分析、格理论、控制流图、数据流分析、固定点算法、缩小和扩大、路径敏感性、过程间分析和上下文敏感性、控制流分析以及指针分析。 一个带有堆指针和函数指针的微小命令式编程语言被用于展示多种不同的静态分析技术。

我们强调基于约束的静态分析方法，在适当的约束系统下，将分析任务概念上划分为前端生成程序代码约束和后端解决约束以产生分析结果的过程。 这种方法使得分析规范与算法性能相关的方面分离开来，从而确定其精确性。 在实际实现分析时，我们经常即时解决生成的约束，而不需要显式地表示它们。

我们专注于完全自动的分析（即不涉及程序员指导，例如循环不变式的形式）和保守的分析（通常意味着完备但不完全），并且我们只考虑图灵完备的语言（像大多数用于普通软件开发的编程语言一样）。

我们涵盖的分析使用不同类型的约束系统来表达，每种系统都有自己的约束求解器：

- 术语一致性约束，使用几乎线性的并查集算法，
- 条件子集约束，使用立方算法，以及
- 在格上的单调约束，使用变种的不动点求解器.

演示的风格旨在精确但不过于正式。读者应该熟悉高级编程语言的概念和编译器构造的基础知识。

我们将看到执行程序的静态分析所需的基本工具。 现实生活中的应用总是回归到我们所涵盖的技术，尽管通常需要许多变化和扩展。

两个主要领域将完全不涉及。分析的质量只能相对于一套预期的应用程序来衡量。 很少有竞争性分析可以进行正式比较，因此这个领域的大部分工作都是通过实验来确定所提出分析的精确性和效率。 分析的正确性需要对底层编程语言进行形式化语义的定义。 完全形式化的正确性证明非常费时，大部分仍然是学术练习。 即便如此，通常可以提供令人信服的非正式正确性论证。

这些笔记附带了一个网站，提供讲义幻灯片，我们所涵盖的大部分算法的Scala实现，以及额外的练习：

http://cs.au.dk/~amoeller/spa/

第一章

介绍

关于给定程序有许多有趣的问题，例如：

- 程序在每个输入上都会终止吗？
- 执行过程中堆可以变得多大？
- 是否存在导致空指针解引用、除零或算术溢出的输入？

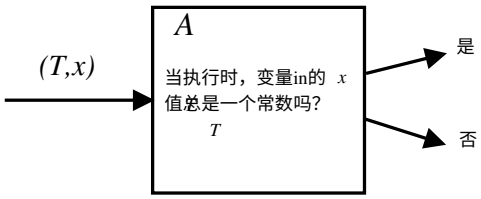
- 所有变量在使用之前都被初始化了吗？
- 数组是否总是在其边界内访问？
- 所有断言都能保证成功吗？
- 程序中是否包含死代码，或者更具体地说，函数 `f`是否从 `main`可达？

- 变量 `x`的值是否取决于程序输入？
- 是否可能在将来读取变量 `x`的值？
- 在堆中， `p`和 `q`是否指向不相交的结构？
- 是否可能存在悬空引用，例如指向已被释放的内存的指针？

- 在程序终止之前，是否所有资源都被正确释放？

当推理程序的正确性和优化程序以提高性能时，会出现这样的问题。 关于正确性，程序员通常使用测试来增加对程序按预期工作的信心，但正如Dijkstra所说：“程序测试可以用来显示错误的存在，但永远不能用来显示错误的不存在。”理想情况下，我们希望对我们的程序在所有可能的输入情况下可能做出的行为有保证，并且我们希望这些保证是自动提供的，也就是由程序提供的。 程序分析器是这样一种程序，它以其他程序作为输入，并决定它们是否具有给定的属性。

Rice定理是[1953]年的一个普遍结果，非正式地说明了关于程序行为的所有有趣问题（在图灵完备的编程语言中编写的程序）都是不可判定的。 对于任何特殊情况，这很容易看出。 例如，假设存在一个分析器，可以判断程序中的变量是否具有常量值。 换句话说，分析器是一个程序A，它以程序T和T的一个变量x作为输入，并决定当T被执行时x是否具有常量值。



然后，我们可以利用这个分析器来决定停机问题，使用以下程序作为输入，其中 TM(j)模拟了第 j个图灵机的空输入：

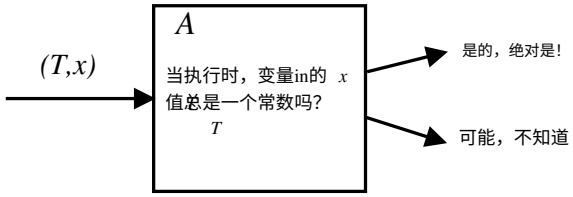
```
x = 17; if (TM(j)) x = 18;
```

如果且仅当第 j个图灵机在空输入上不停机时， x具有一个常数值。 如果假设的常值分析器 A存在，则我们有一个决策过程来解决停机问题，而这是不可能的。

这似乎是一个令人沮丧的结果。 然而，我们真正的目标不是决定这样的属性，而是解决实际问题，如使程序运行更快或占用更少的空间，或者在程序中找到错误。 解决方案是接受近似答案，这些答案仍然足够精确以满足我们的应用需求。

通常，这种近似是保守的（或者安全的），意味着所有的错误都倾向于同一方向，这由我们的预期应用程序决定。

再次考虑确定变量是否具有常量值的问题。
如果我们的预期应用程序是执行常量传播优化，那么分析只能在变量确实是常量时回答“是”，并且在变量可能是或可能不是常量时必须回答“可能”。 当然，最简单的解决方案是始终回答“可能”，因此我们面临的工程挑战是尽可能频繁地回答“是”，同时获得合理的分析性能。



[1]从这一点开始，我们只考虑图灵完备的语言。

另一个例子是问题：指针 p可能指向哪些变量？ 如果我们的预期应用程序是用 x替换 *p以节省解引用操作，那么分析只能在 p肯定指向 x时回答“&x”，并且在这是错误的或者无法确定答案时回答“？ ”。如果这是错误的或者无法确定答案，那么回答“？ ”。 如果我们的预期应用程序是确定 *p的最大大小，那么分析必须回复一个可能过大的集合 {&x,&y,&z,...}，保证包含所有目标。

一般来说，所有的优化应用都需要保守的近似。
如果我们得到了错误的信息，那么优化就是不可靠的，并且改变了程序的语义。 相反，如果我们得到了琐碎的信息，那么优化就无法产生任何效果。

近似答案也可以用于在程序中查找错误，这可以看作是一种弱形式的程序验证。 以C语言中的指针编程为例。 这充满了危险，比如空指针引用、悬空指针、内存泄漏和意外的别名。 普通的编译器技术对指针错误提供很少的保护。 考虑下面这个小程序，如果用精确地42个参数执行，会出现各种错误：

```
int main(int argc, char *argv[]) {
  if (argc == 42) {
    char *p,*q;
    p = NULL;
    printf("%s",p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    free(p);
    p = (char *)malloc(100);
    p = (char *)malloc(100);
    q = p;
    strcat(p,q);
  }
}
```

标准工具如gcc -Wall和 lint检测不到错误。 通过测试找到错误可能会错过错误，除非我们恰好有一个测试用例以恰好运行程序42个参数。 然而，如果我们对于关于空值和指针目标的问题有近似答案，那么许多上述错误可以在静态情况下捕获，而无需实际运行程序。

练习**1.1**：描述上述程序中的所有错误。

1.1 静态程序分析的不可判定性

（本节需要熟悉通用图灵机的概念；这不是以下章节的先决条件。）

上述从停机问题的约简表明，一些静态分析问题是不可判定的，特别是决定一个给定变量在程序中是否具有常量值的问题。 然而，终止通常是程序员关心的最不重要的问题之一，即他们的程序是否正确工作。 如果我们允许假设我们的程序总是终止，那么决定一个变量是否具有常量值的问题是否仍然是不可判定的？ 那么其他期望的程序属性呢？

使用对角化论证结合经典的Rice定理证明，我们可以一劳永逸地展示一个非常强大的结果：即程序行为的每个非平凡的[2]属性都是不可判定的，即使我们只考虑在每个输入上终止的程序。

换句话说，每一次尝试构建一个完全自动的程序分析器，即一个能够检查其他程序的输入-输出行为的非平凡正确性属性的程序，都注定会失败，即使程序分析器被允许假设给定的程序总是终止。 近似是不可避免的。

如果我们将程序建模为确定性图灵机，那么将“正确性”定义为特殊的失败状态的不可达性似乎是合理的。也就是说，在给定的输入上，图灵机最终会在其接受状态（直观地返回“是”）、拒绝状态（直观地返回“否”）、失败状态（意味着正确性条件已被违反）或机器发散（即永不终止）中终止。

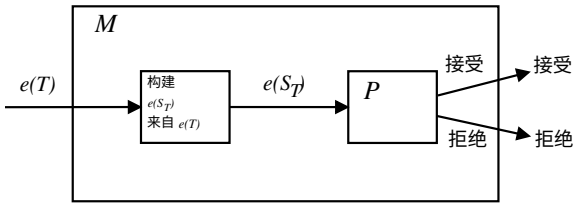
我们可以通过矛盾证明来展示这个不可判定性结果。 假设 P是一个能够判断给定总图灵机中是否可达失败状态的程序。 （如果输入不是一个总图灵机，P的输出是未指定的 - 我们只要求它正确分析总图灵机。）假设 P在其接受状态中停机，当且仅当给定图灵机的失败状态是不可达的，否则在拒绝状态中停机。 也就是说，直观上， P表示如果给定的图灵机工作正确，则回答“是”，否则回答“否”。 我们的目标是展示 P不存在。

如果 P存在，那么我们也可以构建另一个图灵机，让我们称之为 M，它以图灵机T的编码 $e(T)$作为输入，然后构建另一个图灵机S_T的编码 $e(S_T)$，其行为如下：S_T本质上是一个专门模拟 T的通用图灵机。

[2]如果存在至少一个程序满足该属性且至少一个程序不满足，则该属性是非平凡的。相反，平凡的属性可以在不查看输入程序的情况下决定，并且在实践中我们关心的所有属性都是非平凡的。[3]通过“程序的行为”，我们指的是它们的输入输出语义，与语法属性相对。

[4]从技术上讲，我们在这里仅限于安全属性；活性属性可以使用其他可计算模型类似地处理。

让 w表示 ST的输入。 现在构造 S_T，使其在最多 $|w|$步骤中模拟$_T$在输入 $e(\ T)$上的运行。 如果模拟在T的接受状态结束，则 $S\ T$进入其失败状态。 很明显可以以这种方式创建 S_T，使其只能通过这种方式达到失败状态。 如果模拟没有在$_T$的接受状态结束（即已经进行了 $|w|$步骤，或者模拟达到 T的拒绝或失败状态），则 $S\ T$进入其接受状态或拒绝状态（我们选择哪个状态并不重要）。 这完成了对 S_T相对于$_T$和 w的工作原理的解释。 注意 $S\ T$永远不会发散，并且它在最多 $|w|$步骤后仅在$_T$接受输入 $e(\ T)$的情况下达到其失败状态。 在构建ST之后， M将其传递给我们的假设程序分析器 P。 假设 P按照承诺工作，如果 S_T是“正确”的，则它以接受结束，此时我们也让 M在其接受状态下停止；否则以拒绝结束，此时 M同样在其拒绝状态下停止。



我们现在问：M是否接受输入M？也就是说，如果我们运行M并将T设置为M，会发生什么？如果M接受输入M，那么必须是PT接受输入ST，这意味着ST是“正确”的，因此它的失败状态是不可达的。换句话说，对于任何输入w，无论其长度如何，ST都不会达到其失败状态。这又意味着T不接受输入T。然而，我们有T=M，这就与我们假设M接受输入M相矛盾。相反，如果M拒绝输入M，则P拒绝输入ST，因此ST的失败状态对于某些输入v是可达的。这意味着必须存在某个w，使得ST在输入v上的$|w|$步骤中达到失败状态，因此T必须接受输入T，这又导致了矛盾。根据构造，M在任何输入上都会停机，但对于输入M来说，接受或拒绝都是不可能的。总之，理想的程序分析器P是不存在的。

练习 **1.2**：在上面的证明中，假设的程序分析器 P只需要正确分析总是终止的程序。 换句话说，分析器可以假设给定的程序总是终止。 如果我们想要证明以下较弱的属性，该证明如何简化？ 不存在图灵机 P可以决定给定图灵机中是否可达到失败状态。 （注意，给定的图灵机现在不被假设为总体的。）

第二章

一个微型编程语言

我们在接下来的章节中使用一个微型命令式编程语言，称为 *TIP*。 它的设计目标是具有最小的语法，同时包含使静态分析有趣和具有挑战性的所有结构。

2.1 TIP的语法

在本节中，我们基于上下文无关文法介绍了TIP语言的形式语法。

表达式

所有基本表达式都表示整数值：

$$
\begin{aligned}
E \rightarrow\ & intconst \\
& \mid \text{标识符} \\
& \mid E + E \mid E - E \mid E * E \mid E / E \mid E > E \mid E == E \\
& \mid (\ E\ ) \\
& \mid \text{输入}
\end{aligned}
$$

输入表达式从输入流中读取一个整数。 比较运算符对于假返回0，对于真返回1。 指针表达式将在后面添加。

语句

简单语句很熟悉：

$$
\begin{array}{rl}
S \rightarrow & \text{标识符} = E\,; \\
\mid & \text{输出} \quad E\,; \\
\mid & S\ S \\
\mid & \\
\mid & \texttt{if (}E\texttt{) \{ }S\texttt{ \}} \left[\texttt{ else \{ }S\texttt{ \}}\right]^{?} \\
\mid & \texttt{while (}E\texttt{) \{ }S\texttt{ \}}
\end{array}
$$

我们使用符号 $[\ldots]^{?}$ 表示可选部分。在条件中，我们将0解释为假，所有其他值解释为真。输出语句将一个整数值写入输出流。

函数

函数可以接受任意数量的参数并返回一个单一的值：

$$F \rightarrow id\ \texttt{(}\ id, \ldots, id\ \texttt{)\ \{}\ \left[\texttt{ var }id, \ldots, id\texttt{;}\ \right]^{?}\ S\ \texttt{return}\ E\texttt{; \}}$$

var块声明了一组局部变量。 函数调用是一种额外的表达式：

$$E \rightarrow id\ \texttt{(}\ E, \ldots, E\ \texttt{)}$$

指针

最后，为了允许动态内存，我们将指针引入堆：

$$
\begin{array}{rl}
E \rightarrow & \&id \\
\mid & \texttt{malloc} \\
\mid & \texttt{*}E \\
\mid & \texttt{null}
\end{array}
$$

第一个表达式创建一个指向变量的指针，第二个表达式在堆中分配一个新的单元，第三个表达式解引用一个指针值。 为了给堆单元赋值，我们允许另一种形式的赋值：

$$S \rightarrow \texttt{*}id = E\texttt{;}$$

请注意，指针和整数是不同的值，因此不允许指针算术运算。 当然， malloc 只分配一个堆单元，但这足以说明指针所带来的挑战。

我们还允许用函数名表示函数指针。 为了使用它们，我们将函数调用进行了泛化：

$$E \rightarrow \texttt{(}E\texttt{)(}\ E, \ldots, E\ \texttt{)}$$

函数指针可以作为对象或高阶函数的简单模型。

程序

程序只是一组函数的集合：

$$P \to F \ldots F$$

最后一个函数是主函数，它启动执行。 它的参数按顺序从输入流的开头提供，并且它返回的值被附加到输出流中。 我们做出了一个简化的假设，即程序中所有声明的标识符都是唯一的，即没有两个不同的程序点引入相同的标识符名称。

练习**2.1**：证明任何程序都可以被规范化，使得所有声明的标识符都是唯一的。

TIP缺少许多常用编程语言的特性，例如类型注解、全局变量、记录、对象、嵌套函数和指针算术。 我们将在后面的章节中考虑其中一些特性的练习。

为了简洁起见，我们故意没有详细说明TIP语言的所有细节，无论是语法还是语义。

练习**2.2**：识别TIP语言中的不完全规定部分，并提出有意义的选择以使其更加明确。

2.2 示例程序

以下TIP程序都计算给定整数的阶乘。 第一个是迭代的：

```
ite(n) {
  var f;
  f = 1;
  while (n>0) {
    f = f*n;
    n = n-1;
  }
  return f;
}
```

第二个程序是递归的：

```
rec(n) {
  var f;
  if (n==0) { f=1; }
  else { f=n*rec(n-1); }
  return f;
}
```

第三个程序过于复杂：

```
foo(p,x) {
  var f,q;
  if (*p==0) { f=1; }
  else {
    q = malloc;
    *q = (*p)-1;
    f=(*p)*((x)(q,x));
  }
  return f;
}
```

```
main() {
   var n;
   n = input;
   return foo(&n,foo);
}
```

2.3 控制流图

为了进行分析，通常将程序视为控制流图，这是程序源代码的另一种表示方式。

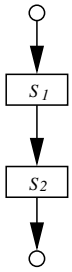
目前，我们只考虑TIP语言的子集，该子集由一个不带指针的函数体组成。控制流图（CFG）是一个有向图，其中节点对应语句，边表示可能的控制流。为了方便起见，CFG总是有一个入口点，表示为entry，和一个出口点，表示为exit。 我们可以将它们视为无操作语句。

如果 v是CFG中的一个节点，则 $pred(v)$表示前驱节点的集合，$succ(v)$ 表示后继节点的集合。

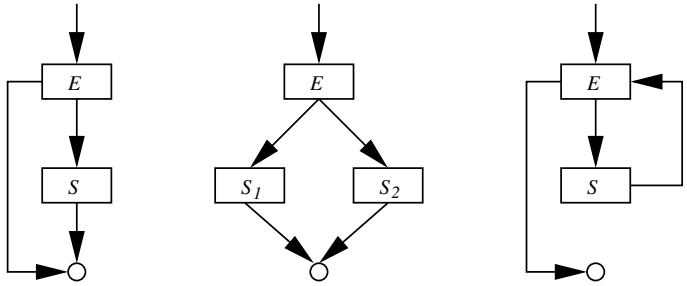
语句的控制流图

目前，我们只考虑简单语句，可以通过归纳的方式构建CFG。 赋值语句、输出语句、返回语句和声明语句的CFG如下所示：

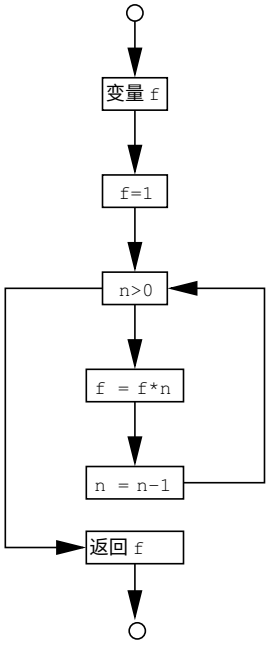
对于序列 S_1 S_2，我们消除 S_1的出口节点和S_2的入口节点，并将语句连接起来：



类似地，其他控制结构可以通过归纳图构造来建模：



使用这种系统化的方法，迭代阶乘函数的CFG如下所示：



我们在第7章讨论了由多个函数组成的整个程序的控制流图。

第三章

类型分析

我们的编程语言没有明确的类型，但是当然各种操作只能应用于特定的参数。具体来说，以下限制似乎是合理的：

- 算术运算和比较只适用于整数；
- 只有整数可以作为主函数的输入和输出；
- 控制结构中的条件必须是整数；
- 只能调用函数；
- 一元 *运算符只适用于指针。

我们假设违反这些要求会导致运行时错误。 因此，对于给定的程序，我们希望在执行过程中这些要求得到满足。
由于这是一个有趣的问题，我们立即知道它是不可判定的。

我们采用一种保守的近似方法：类型可判定。 如果一个程序满足从给定程序的语法树系统地推导出的一系列类型约束，则该程序是可判定的。 这个条件意味着上述要求在执行过程中得到了保证，但反过来则不成立。 因此，我们的类型检查器将是保守的，并拒绝一些实际上不会违反任何要求的程序。

3.1 类型

我们首先定义一种描述可能值的类型语言：

$$
\begin{aligned}
\tau \rightarrow\ & \texttt{int} \\
\mid\ & \&\tau \\
\mid\ & (\tau,\ldots,\tau)\texttt{->}\tau
\end{aligned}
$$

类型术语分别描述整数、指针和函数指针。
语法通常会生成有限类型，但对于递归函数和数据结构，我们需要正则类型。这些类型被定义为在上述构造函数上定义的正则树。 请记住，如果可能的树只包含有限数量的不同子树，则它是正则的。

> 练习 **3.1**：展示如何通过有限自动机表示正则类型，以便两个类型在它们的自动机接受相同语言时相等。

3.2 类型约束

对于给定的程序，我们生成一个约束系统，并在约束可解时定义程序为可类型化的。 在我们的情况下，我们只需要考虑带有变量的正则类型术语上的相等约束。 这类约束可以使用一致化算法高效地解决。

对于每个标识符 id我们引入一个类型变量 $[\![id]\!]$，并且对于每个非标识符表达式 E引入一个类型变量 $[\![E]\!]$。 在这里， E指的是语法树中的一个具体节点，而不是它所对应的语法。这使得我们的表示略微模糊，但比严谨正确的方法更简单。 （为了避免歧义，例如，可以使用表示 $[\![E]\!]_v$其中 v是语法树节点的唯一标识符。）假设所有声明的标识符都是唯一的（参见练习2.1），则不需要为相同标识符的不同出现使用不同的类型变量。

约束条件是针对我们语言中的每个构造系统地定义的：

intconst:	$[\![\mathit{intconst}]\!] = \texttt{int}$
E_1 op E_2:	$[\![E_1]\!] = [\![E_2]\!] = [\![E_1 \text{ op } E_2]\!] = \texttt{int}$
E_1==E_2:	$[\![E_1]\!] = [\![E_2]\!] \wedge [\![E_1 == E_2]\!] = \texttt{int}$
`input`:	$[\![\texttt{input}]\!] = \texttt{int}$
$id = E$:	$[\![id]\!] = [\![E]\!]$
`output` E:	$[\![E]\!] \models \texttt{int}$
`if (`E`)` S:	$[\![E]\!] \models \texttt{int}$
`if (`E`)` S_1 `else` S_2:	$[\![E]\!] \models \texttt{int}$
`while (`E`)` S:	$[\![E]\!] \models \texttt{int}$
$id(id_1,\ldots,id_n)\{\ldots\texttt{return}\ E;\ \}$:	$[\![id]\!] = ([\![id_1]\!],\ldots,[\![id_n]\!]) \to [\![E]\!]$
$id(E_1,\ldots,E_n)$:	$[\![id]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!]) \to [\![id(E_1,\ldots,E_n)]\!]$
$(E)(E_1,\ldots,E_n)$:	$[\![E]\!] = ([\![E_1]\!],\ldots,[\![E_n]\!]) \to [\![(E)(E_1,\ldots,E_n)]\!]$
&*id*:	$[\![\&id]\!] \models \&[\![id]\!]$
`malloc`:	$[\![\texttt{malloc}]\!] \models \&\alpha$
`null`:	$[\![\texttt{null}]\!] \models \&\alpha$
*E:	$[\![E]\!] \models \&[\![*E]\!]$
*id=E:	$[\![id]\!] \models \&[\![E]\!]$

在上述规则中，每个 α的出现表示一个新的类型变量。 注意变量引用和声明不产生任何约束，并且在抽象语法中不存在括号表达式。

此外，所有的术语构造器都满足一般的术语相等公理：

$$c(t_1, \ldots, t_n) = c'(t'_1, \ldots, t'n) \;\Rightarrow\; t_i = t'_i \text{对于每个 } i$$

其中 c是术语构造器之一，例如 &。

因此，给定的程序会产生一组关于带有变量的类型术语的等式约束。

> 练习 **3.2**：解释上述每个类型约束。

一个解决方案为每个类型变量分配一个类型，以满足所有等式约束。 该算法的正确性声明是存在一个解决方案意味着在执行过程中不会发生指定的运行时错误。

3.3 解决约束

如果存在解，则可以使用正则项统一算法在几乎线性时间内计算出解。 由于约束条件也可以在线性时间内提取出来，整个类型分析非常高效。

复杂的阶乘程序生成了以下约束条件，其中不显示重复项：[[foo]] = ([[p]],[[x]])->[[f]]

[[*p]] = int
[[1]] = int
[[p]] = &[[*p]]
[[malloc]] = &α
[[q]] = &[[*q]]
[[f]] = [[(*p)*((x)(q,x))]]
[[(x)(q,x)]] = int
[[input]] = int
[[n]] = [[input]]
[[foo]] = ([[&n]],[[foo]])->[[foo(&n,foo)]]
[[*p==0]] = int
[[f]] = [[1]]
[[0]] = int
[[q]] = [[malloc]]
[[q]] = &[[(*p)-1]]
[[*p]] = int
[[(*p)*((x)(q,x))]] = int
[[x]] = ([[q]],[[x]])->[[(x)(q,x)]]
[[main]] = ()->[[foo(&n,foo)]]
[[&n]] = &[[n]]
[[*p]] = [[0]]

这些约束有一个解，其中大多数变量被赋值为 int，除了：[[p]] = &int[[q]] = &int[[malloc]] = &int[[x]] = ϕ[[foo]] = ϕ[[&n]] = &int[[main]] = ()->int

其中 ϕ是对应于无限展开的常规类型

ϕ = (&int,ϕ)->int

练习3.3：画出 ϕ的展开图。

由于存在这个解，我们得出结论，我们的程序类型是正确的。 递归类型也适用于数据结构。例如程序：

```
var p;
p = malloc;
*p = p;
```

创建了约束条件：

[[p]] = &α
[[p]] = &[[p]]

其解为 [[p]] = ψ其中 ψ = &ψ。 有些约束条件有无限多个解。 例如，函数：

```
poly(x) {
  return *x;
}
```

具有类型 &α->α对于任何类型 α，这对应于它展示的多态行为。

3.4 松弛和限制

类型分析当然只是近似的，这意味着某些程序将被不公平地拒绝。一个简单的例子是：

```
bar(g,x) {
  var r;
  if (x==0) { r=g; } else { r=bar(2,0); }
  return r+1;
}

main() {
  return bar(null,1);
}
```

这永远不会导致错误，但由于它生成了等价于以下约束条件，因此无法进行类型推断：

int = [[r]] = [[g]] = &α

这些明显是无法解决的。

> 练习 **3.4**：解释这个程序的行为。

可以使用更强大的多态类型分析来接受上述程序，但许多其他示例将不可避免地被拒绝。

另一个问题是，这种类型系统忽略了其他几个运行时错误，例如对空指针的解引用，读取未初始化的变量，除以零，以及这个程序演示的更微妙的逃逸堆栈单元：

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

问题在于 *p表示一个已经从baz函数中逃逸的堆栈单元。 正如我们将看到的，这些问题可以通过更有雄心的静态分析来解决。

第四章

格论

我们将研究的静态分析技术基于数学中的格理论，我们在本章中简要回顾一下。

4.1 示例：符号分析

作为一个激励的例子，假设我们希望设计一个分析，可以找出给定程序中变量和表达式的可能符号。 在具体的执行中，值可以是任意整数。 相反，我们的分析将整数值抽象成三个类别或抽象值：正数（+），负数（-）和零（0）。 与我们在第三章考虑的分析类似，我们通过引入近似来规避不可判定性。 也就是说，分析必须准备处理不确定的信息，例如在某些表达式的符号不明确的情况下，我们添加了一个特殊的抽象值（?）表示“不知道”。 我们还必须决定在某些表达式的符号在某些执行中是正的但在其他执行中不是正的情况下，我们对什么信息感兴趣。 对于这个例子，让我们假设我们只对确定的信息感兴趣，也就是说，分析应该只报告

+对于给定的表达式，如果确定该表达式在每次执行中都会评估为正数，则为真，否则为假。 此外，引入一个抽象值 ⊥对于那些值不是数字（而是指针）或在任何执行中没有值的表达式，这也是有益的，因为它们无法从程序入口点访问。

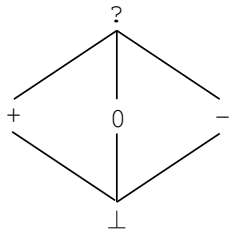
考虑这个程序：

```
var a,b,c;
a = 42;
b = 87;
```

```
if (input) {
  c = a + b;
} else {
  c = a - b;
}
```

在这里，分析可以得出结论，在程序结束时， a和 b在所有可能的执行中都是正数。 变量 c的符号根据具体的执行情况可能是正数或负数，因此分析必须报告 ?。

总的来说，我们有一个由五个抽象值组成的抽象域。 $\{+, -, 0, ?, \bot\}$，我们可以按照以下方式组织，最不精确的信息在顶部，最精确的信息在底部：



排序反映了 ⊥表示整数值的空集合和 ?表示所有整数值的集合。

这个抽象域是一个格。 我们在5.2节继续开发符号分析，但首先需要建立数学基础。

4.2 格

偏序是一个集合 S配备了一个二元关系 $\sqsubseteq$，满足以下条件：

- 自反性： $\forall x \in S$： $x \sqsubseteq x$
- 传递性： $\forall x, y, z \in S$： $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- 反对称性: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

当 $x \sqsubseteq y$时，我们说 y是 x的一个安全近似，或者说 x至少和 y一样精确。

我们说 $y \in S$是 X的一个上界，记作 $X \sqsubseteq y$，如果我们有 $\forall x \in X : x \sqsubseteq y$。 类似地， $y \in S$是 X的一个下界，记作 $y \sqsubseteq X$，如果$\forall x \in X : y \sqsubseteq x$。 最小上界，记作 $\sqcup X$，定义如下:

$$X \sqsubseteq \sqcup X \wedge \forall y \in S : X \sqsubseteq y \Rightarrow \sqcup X \sqsubseteq y$$

双重，一个最大下界，写作 $\sqcap X$，被定义为：

$$\sqcap X \sqsubseteq X \wedge \forall y \in S : y \sqsubseteq X \Rightarrow y \sqsubseteq \sqcap X$$

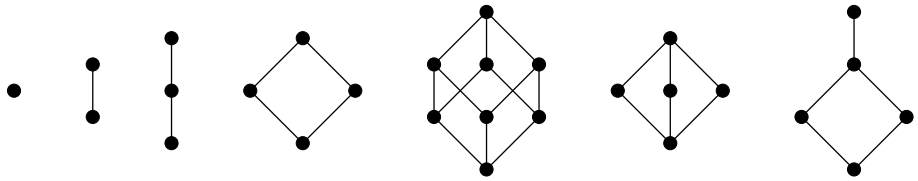
格是一个偏序，其中 $\sqcup X$和 $\sqcap X$对于所有 $X \subseteq S$都存在。（文献通常称之为完全格。）格必须有一个唯一的最大元素表示为 $\top$和一个唯一的最小元素表示为 $\bot$。

练习**4.1**：证明 $\sqcup S$和 $\sqcap S$分别是 S中的唯一最大元素和唯一最小元素。换句话说，我们有 $\top = \sqcup S$和 $\bot = \sqcap S$。

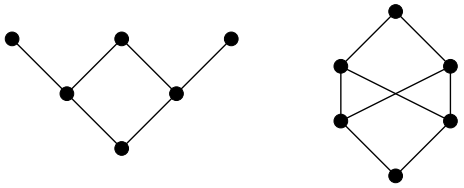
练习4.2：证明 $\sqcup S = \sqcap \emptyset$和 $\sqcap S = \sqcup \emptyset$。

我们经常会看到有限格。对于这些格的要求可以简化为观察到 $\bot$和 $\top$存在，并且每对元素 x和 y都有一个最小上界写作 $x \sqcup y$和一个最大下界写作 $x \sqcap y$。一个有限偏序可以通过哈斯图来表示，其中元素是节点，顺序关系是从较低节

点到较高节点的边的传递闭包。使用这种符号，以下所有的偏序也是格：

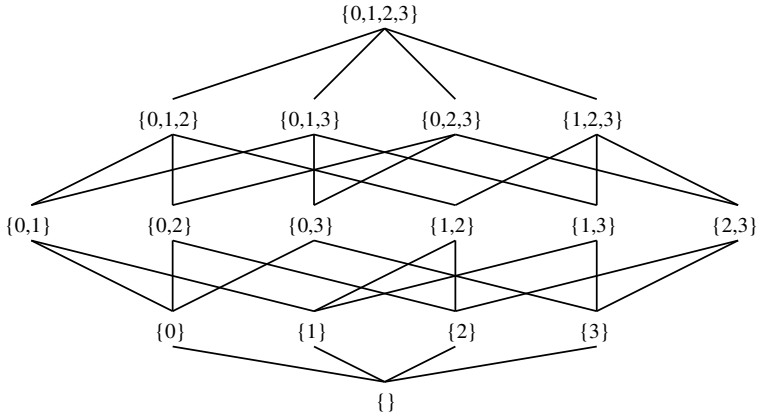


而这些偏序不是格：



练习**4.3**：为什么这两个图不定义格?

每个有限集合 A定义了一个格 $(2^A, \subseteq)$，其中 $\bot = \emptyset$，$\top = A$，$x \sqcup y = x \cup y$，并且 $x \sqcap y = x \cap y$。我们称之为幂集格对于 A。对于一个有四个元素的集合，幂集格看起来像这样：



格的高度被定义为从 $\perp$到
$\top$的最长路径的长度。 例如，上述幂集格的高度为4。 一般来说，格$(2^A, \subseteq)$的高度为 $|A|$。

4.3 构建格

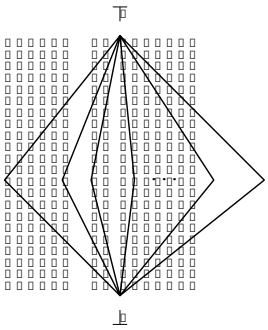
如果 $L_1, L_2, \ldots, L_n$是具有有限高度的格，那么它们的乘积也是具有有限高度的:

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

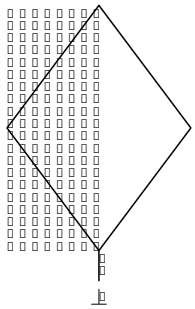
其中 $\sqsubseteq$被逐点定义。 注意 $\sqcup$和 $\sqcap$可以逐点计算，并且高度$(L_1 \times \ldots \times L_n) =$高度$(L_1) + \ldots +$高度$(L_n)$。还有一个求和运算符:

$$+L_n = \{(i, x_i) \mid x_i \in L_i \backslash \{\perp, \top\}\} \cup \{\perp, \top\}$$

其中 $\perp$和 $\top$如预期所示，且 $(i, x) \sqsubseteq (j, y)$if and only if $i = j$ and $x \sqsubseteq y$.注意$(L_1 + \ldots + L_n)$ 的高度 $= max\{\ height(L_i)\}$. 求和运算符可以如下所示：

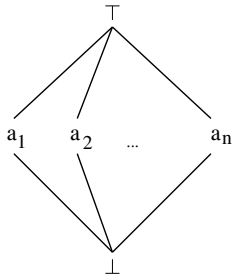


如果 L是一个格，那么 $lift(L)$也是一个格，可以通过以下方式说明：



并且$lift(L)$ 的高度 $= height(L) + 1$如果 Lhas finite height.

如果 A是一个集合（不一定是格），那么 $flat(A)$可以表示为



是一个高度为2的格。

练习**4.4**：证明 $flat(A)$可以看作是一种和格的求和。

最后，如果 A是一个集合， L是一个格，那么我们可以得到一个映射格：

$$A \to L = \{[a_1 \to x_1, \ldots, a_n \to x_n] \mid x_i \in L\}$$

按点排序： $f \sqsubseteq g \Leftrightarrow \forall a_i : f(a_i) \sqsubseteq g(a_i)$。 如果 A是有限的， L的高度也是有限的，则 $height(A \to L) = |A| \cdot height(L)$。

练习**4.5**：证明每个乘积格是同构于一个映射格。

练习**4.6**：验证上述有关构造的格的高度的说法。

集合 $Sign = \{+, -, \mathbb{0}, ?, \bot\}$按照第4.1节中描述的顺序形成一个格，我们用它来描述符号分析中的抽象值。 映射格的一个例子是StateSigns $= V$ars $\to Sign$ 其中 $Vars$是我们希望分析的程序中出现的变量名的集合。 这个格的元素描述了为所有变量提供抽象值的抽象状态。

乘积格的一个例子是$ProgramSigns = StateSigns^n$其中 n是程序的控制流图中的节点数。 这个格描述了程序中所有节点的抽象状态。

4.4 方程和不动点

从第4.1节继续进行符号分析，以下程序每行的变量符号是什么？

```
var a,b;        // 1
a = 42;         // 2
b = a + input; // 3
a = a - b;      // 4
```

我们可以从程序中为每个程序变量和行号推导出一个约束变量的方程系统：

$$
\begin{aligned}
a_1 &= \mathsf{?} \\
b_1 &= \mathsf{?} \\
a_2 &= \mathsf{+} \\
b_2 &= b_1 \\
a_3 &= a_2 \\
b_3 &= a_2 \mathbin{\mathsf{+}} \mathsf{?} \\
a_4 &= a_3 \mathbin{\mathsf{-}} b_3 \\
b_4 &= b_3
\end{aligned}
$$

这里的运算符 +和 -作用于抽象值，我们将在第5.2节中讨论。在这个约束系统中，约束变量的值来自于抽象值格 $Sign$。 我们还可以推导出以下等价的方程系统，其中每个约束变量的值来自于第4.3节的抽象状态格 $StateSigns$：

$$
\begin{aligned}
x_1 &= [\mathsf{a} \to \mathsf{?}, \mathsf{b} \to \mathsf{?}] \\
x_2 &= x_1[\mathsf{a} \to \mathsf{+}] \\
x_3 &= x_2[\mathsf{b} \to x_2(\mathsf{a}) \mathbin{\mathsf{+}} \mathsf{?}] \\
x_4 &= x_3[\mathsf{a} \to x_3(\mathsf{a}) \mathbin{\mathsf{-}} x_3(\mathsf{b})]
\end{aligned}
$$

注意，在这个例子的程序中，每个方程只依赖于前面的方程，因此在这种情况下，可以通过简单的替换找到解决方案。 然而，互相递归的方程可能会出现，例如，对于包含循环的程序。 我们现在展示如何在一般情况下解决这样的方程系统。

当且仅当对于所有的$x, y \in L$：$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$时，函数 $f : L \to L$被称
请注意，这个性质并不意味着 f是广义的，也就是说，对于所有的$x \in L$：$x \sqsubseteq f(x)$；例如，将所有输入映射到 $\bot$的函数是单调的，但对于具有多个元素的格，它不是广义的。

请注意，单调函数的组合仍然是单调的。 此外，单调性的定义可以自然地推广到具有多个参数的函数。 将其视为函数时，$\sqcup$和$\sqcap$是单调的。

我们说对于f，如果$f(x) = x$，则$x \in L$是f的不动点。 对于f，最小不动点x是f的一个不动点，其中对于f的每个不动点y，$x \leqslant y$。让L是一个有有限
高度的格。 方程系统的形式如下：

$$
\begin{aligned}
x_1 &= F_1(x_1, \ldots, x_n) \\
x_2 &= F_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &= Fx_n(x_1, \ldots, x_n)
\end{aligned}
$$

其中 x_i是变量，而 $F_i : L^n \to L$是一组函数的集合。 如果所有的函数都是单调的，那么系统就有一个唯一的最小解，这个解可以通过函数 $F : L^n \to L^n$的最小不动点来获得，定义如下：

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

我们需要的核心结果是不动点定理。 在一个有限高度的格 L中，每个单调函数 f都有一个唯一的最小不动点，定义如下：

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

这个定理的证明非常简单。 观察到$\bot \sqsubseteq f(\bot)$因为 $\bot$是最小元素。 由于 f是单调的，我们可以得出 $f(\bot) \sqsubseteq f^2(\bot)$，并且通过归纳法可以得出 $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$。 因此，我们有一个递增的链：

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

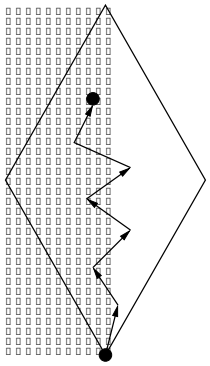
由于 L被假定具有有限高度，我们必须对于某个 k有 $f^k(\bot) = f^{k+1}(\bot)$。 我们定义 $fix(f) = f^k(\bot)$并且由于 $f(fix(f)) = f^{k+1}(\bot) = f^k(\bot) = fix(f)$，我们知道 $fix(f)$是一个不动点。 现在假设 x是另一个不动点。 由于$\bot \sqsubseteq x$，根据单调性我们得到 $f(\bot) \sqsubseteq f(x) = x$，通过归纳我们得到 $fix(f) = f^k(\bot) \sqsubseteq x$。 因此， $fix(f)$是最小的不动点。
通过反对称性，它也是唯一的。

细心的读者可能已经注意到，这是一个构造性的不动点定理的证明。 我们将在第5.1节中讨论从证明中可以推断出的算法。

计算不动点的时间复杂度取决于三个因素：

- 格的高度，因为这提供了 k的上界；
- 计算 f的成本；
- 测试相等性的成本。

计算不动点可以被看作是从 $\bot$开始向上走的格子之旅：



我们可以类似地解决形式为　　不等式　　of的系统：

$$
\begin{aligned}
x_1 &\sqsubseteq F_1(x_1, \ldots, x_n) \\
x_2 &\sqsubseteq F_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &\sqsubseteq F_n(x_1, \ldots, x_n)
\end{aligned}
$$

通过观察，关系 $x \sqsubseteq y$等价于 $x = x \sqcap y$。 因此，通过将系统重写为以下形式，解得保持不变：

$$
\begin{aligned}
x_1 &= x_1 \sqcap F_1(x_1, \ldots, x_n) \\
x_2 &= x_2 \sqcap F_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &= x_n \sqcap F_n(x_1, \ldots, x_n)
\end{aligned}
$$

这只是一个具有单调函数的方程系统，与之前一样。 相反，形式为的不等式：

$$
\begin{aligned}
x_1 &\sqsupseteq F_1(x_1, \ldots, x_n) \\
x_2 &\sqsupseteq F_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &\sqsupseteq F_n(x_1, \ldots, x_n)
\end{aligned}
$$

可以重写为：

$$
\begin{aligned}
x_1 &= x_1 \sqcup F_1(x_1, \ldots, x_n) \\
x_2 &= x_2 \sqcup F_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &= x_n \sqcup F_n(x_1, \ldots, x_n)
\end{aligned}
$$

通过观察，关系 $x \sqsupseteq y$等价于 $x = x \sqcup y$。

练习**4.7**：证明 $x \sqsubseteq y$等价于 $x = x \sqcap y$。

第5章

使用单调框架进行数据流分析

经典的数据流分析，也称为单调框架，从一个CFG和一个具有有限高度的格L开始。 这个格可以对所有程序固定，也可以根据给定的程序进行参数化。

对于CFG中的每个节点 v，我们分配一个变量 $[[v]]$，其范围为 L的元素。 对于编程语言中的每个构造，我们定义一个数据流约束，它将对应节点的变量值与其他节点的变量值（通常是邻居节点）相关联。

至于类型推断，我们将模糊地使用符号 $[[S]]$表示 $[[v]]$ if$S$是与 v相关联的语法。 从上下文中，意义总是清晰的。

我们可以系统地提取给定CFG中变量的一系列约束。如果所有约束恰好是方程或不等式，并且右侧是单调的，那么我们可以使用不动点算法计算出唯一的最小解。

如果所有解都对应于关于程序的正确信息，则数据流约束是正确的。 分析是保守的，因为解可能更或少不精确，但计算最小解将提供最高程度的精确度。

5.1 不动点算法

如果CFG有节点 $V = \{v_1, v_2, \ldots, v_n\}$, 那么我们在格 L^n中工作。 假设节点 v_i 生成数据流方程 $[[v_i]] = F_i([[v_1]], \ldots, [[v_n]])$,我们构建组合函数 $F : L^n \to L^n$如前所述：

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n))$$

紧接着从不动点定理得到的朴素算法如下：

```
x = (⊥, ..., ⊥);
执行 {
 t = x;
 x = F(x);
} while ( x = t);
```

计算不动点 x的算法。 另一种算法，称为混沌迭代，利用我们的格 L^n的结构来计算不动点$(x_1, \ldots, x_n)$：

```
x_n = ⊥;
当 (     ∃i : x_i = F_i(x_1, ..., x_n)) {
 x_i = F_i(x_1, ..., x_n);
}
```

术语"混沌"来自于 i被非确定性地选择。

练习**5.1**: 证明混沌迭代计算 F的最小不动点。

练习**5.2**: 假设我们有一种有效确定循环条件是否成立的方法，为什么混沌迭代比朴素算法更好?

为了在混沌迭代算法中获得一种有效确定循环条件是否成立的方法，我们进一步研究了各个约束的结构。

在一般情况下，每个变量 $[[v_i]]$依赖于所有其他变量。 然而，实际上， F_i的一个实例通常只读取少数其他变量的值。 我们将这些信息表示为一个映射：

$$dep : V \to 2^V$$

对于每个节点 v，它告诉我们在其数据流方程的右侧以非平凡方式出现的其他节点的子集 $[[v]]$。 也就是说，de$p(v)$是可能依赖于 v的信息的节点集合。 有了这些信息，我们可以提出工作列表算法来计算固定点 $(x_1, \ldots, x_n)$：

```
x_1 = ⊥; ...x_n = ⊥;
W = {1, ..., n};
while (W = ∅) {
 i = W.removeNext();
 y = F_i(x_1, ..., x_n);
 if (y = x_i) {
  for (v_j ∈ dep(v_i)) W.add(j);
```

```
    x_i = y;
   }
  }
```

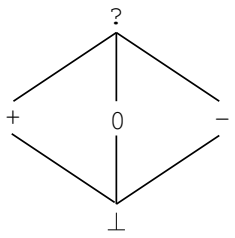
这里将集合 W称为工作列表，具有 `add`和 `removeNext`操作，用于添加和（非确定性地）移除项目。

> 练习 **5.3**：给出一个足够强大的不变量来证明工作列表算法的正确性。

还有可能进行进一步的改进。 可能有益于分别处理由 dep映射引起的图的强连通分量，并且工作列表集合可以改为优先队列，以便利用特定数据流问题的领域特定知识。

5.2 示例：符号分析，再探讨

继续来自第4.1节的例子，我们想要确定所有表达式的符号（`+`，`0`，`-`）。 我们从描述抽象值的小型格 $Sign$开始：



我们分析的完全格是映射格：

$$\text{变量} \rightarrow \text{符号}$$

其中，变量集$Vars$是给定程序中出现的变量集合。 这个格的每个元素可以看作是一个抽象状态。 对于每个控制流图节点v，我们分配一个变量$[[v]]$，它表示节点之前程序点的所有变量的抽象状态的符号值。

数据流约束模拟了抽象环境的影响。
对于变量声明，我们相应地更新：

$$[[v]] = JOIN(v)\ [id_1 \rightarrow \texttt{?}, \ldots, id_n \rightarrow \texttt{?}]$$

对于赋值语句，我们使用约束：

$$[[v]] = JOIN(v)\ [id \rightarrow eval(JOIN(v), E)]$$

对于其他所有节点，约束为：

$$[[v]] = JOIN(v)$$

在哪里：

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

而且 $eval$执行表达式的抽象评估：

$$eval(\sigma, id) = \sigma(id)$$
$$eval(\sigma, intconst) = sign(intconst)$$
$$eval(\sigma,\ E_1 \text{ op } E_2) = \overline{\text{op}}(eval(\sigma,\ E_1), eval(\sigma,\ E_2))$$

其中 σ是当前环境， $sign$给出整数常量的符号而 op是给定运算符的抽象评估，由以下表格定义：

+	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	?
-	⊥	-	-	?	?
+	⊥	+	?	+	?
?	⊥	?	?	?	?

-	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	?
-	⊥	-	?	-	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

*	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0	0
-	⊥	0	+	-	?
+	⊥	0	-	+	?
?	⊥	0	?	?	?

/	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	?	0	0	?
-	⊥	?	?	?	?
+	⊥	?	?	?	?
?	⊥	?	?	?	?

>	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	0	?
-	⊥	0	?	0	?
+	⊥	+	+	?	?
?	⊥	?	?	?	?

==	⊥	0	-	+	?
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	+	0	0	?
-	⊥	0	?	0	?
+	⊥	0	0	?	?
?	⊥	?	?	?	?

很明显，我们的约束的右侧与单调函数对应并不明显。 然而，操作符和映射更新显然是单调的，所以一切都归结为符号格上的抽象运算符的单调性。 这最好通过繁琐的手动检查来验证。 请注意，对于一个具有 n个元素的格， $n \times n$表的单调性可以在时间复杂度为$O(n^3)$内自动验证。

练习**5.4**：描述一个 $O(n^3)$算法来检查一个
由一个 $n \times n$表给出的运算符的单调性。

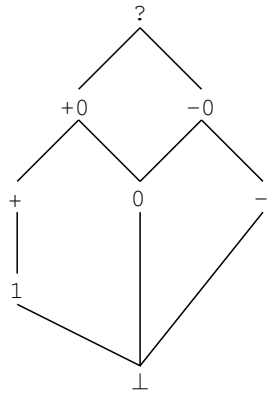
练习**5.5**：检查上述表确实定义了单调运算符
在符号格上。

练习**5.6**：论证这些表是在
符号格中最精确的，因为必须保持正确性。

练习**5.7**：为第4.1节中的示例程序生成方程系统，然后使用其中一个固定点
算法解决它。

练习**5.8**：编写一个小程序，导致具有
相互递归约束的方程系统。

在上述分析中，我们丢失了一些信息，因为例如表达式 `(2>0)==1`被分析为?，这似乎过于粗糙。此外， `+/+`的结果是? 而不是 `+`，因为例如 `1/2`被舍入为零。 为了更精确地处理这些情况，我们可以用元素 `1`（常数1）、 `+0`（正数或零）和 `-0`（负数或零）来丰富符号格，以跟踪更精确的抽象值：



并且通过 8×8表格来描述抽象运算符。

练习**5.9**：通过 8×8表格来定义扩展符号格上的六个运算符。 检查它们是否适当单调。

理论上，符号分析的结果可以用来消除分母表达式具有符号为零的错误，从而拒绝这些程序。
`0`或 `?`。 然而，除非我们添加路径敏感性等技术（见第6章），否则得到的分析可能会不公平地拒绝太多程序，从而不实用。

5.3 示例：活跃变量

如果一个变量的当前值在程序点可能在程序的剩余执行期间被读取，则该变量在该程序点是活跃的。 这个属性显然是不可判定的，但可以通过一种称为活跃性分析（或活跃变量分析）的静态分析来近似。

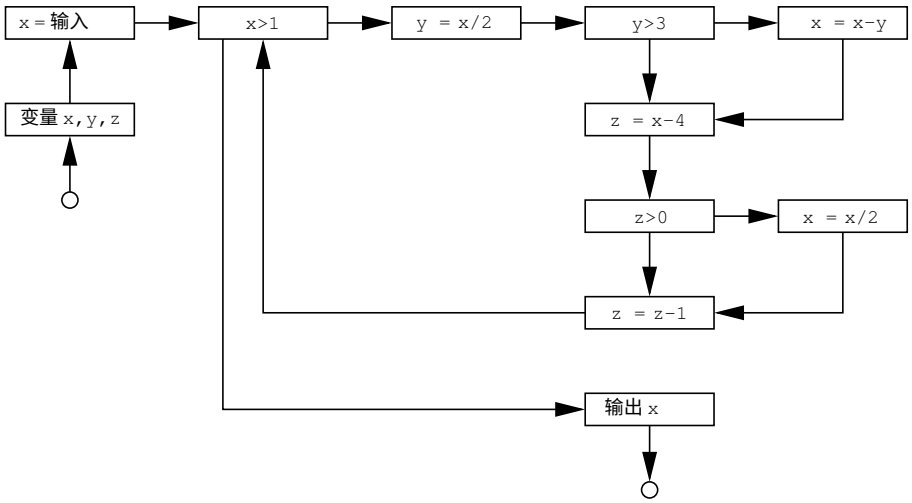
我们使用一个幂集格，其中的元素是给定程序中出现的变量。 这是一个参数化格的示例，也就是说，它取决于被分析的具体程序。对于这个示例程序：

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

格点如下:

$$L = (2^{\{\mathtt{x},\mathtt{y},\mathtt{z}\}}, \subseteq)$$

相应的CFG如下所示:



对于每个CFG节点 v我们引入一个约束变量 $[[v]]$表示在该节点之前的程序点上活跃的程序变量的子集。 分析将是保守的，因为计算得到的集合可能过大。我们使用

辅助定义:

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

对于退出节点，约束条件为:

$$[\![exit]\!] = \{\}$$

对于条件和输出语句，约束条件为:

$$[\![v]\!] = JOIN(v) \cup vars(E)$$

对于赋值语句，约束条件为:

$$[\![v]\!] = JOIN(v) \setminus \{id\} \cup vars(E)$$

对于变量声明，约束条件为:

$$[\![v]\!] = JOIN(v) \setminus \{id_1, \ldots, id_n\}$$

最后，对于其他所有节点，约束条件为:

$$[\![v]\!] = JOIN(v)$$

在这里， $vars(E)$表示出现在 E中的变量集合。 这些约束条件明显具有单调递增的右侧。

> 练习**5.10**：证明约束条件的右侧定义了单调递增函数。

直觉是，如果一个变量在当前节点中被读取，或者在某个未来节点中被读取，除非它在当前节点中被写入，那么它是活跃的。 我们的示例程序产生了以下约束：

$$
\begin{aligned}
&[\![\texttt{变量 x,y,z}]\!] = [\![\texttt{x=输入}]\!] \setminus \{\texttt{x}, \texttt{y}, \texttt{z}\}\\
&[\![\texttt{x=输入 }]\!] = [\![\texttt{x>1}]\!] \setminus \{\texttt{x}\}\\
&[\![\texttt{x>1}]\!] = ([\![\texttt{y=x/2}]\!] \cup [\![\texttt{输出 x }]\!]) \cup \{\texttt{x}\}\\
&[\![\texttt{y=x/2}]\!] = ([\![\texttt{y>3}]\!] \setminus \{\texttt{y}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{y>3}]\!] = [\![\texttt{x=x-y}]\!] \cup [\![\texttt{z=x-4}]\!] \cup \{\texttt{y}\}\\
&[\![\texttt{x=x-y}]\!] = ([\![\texttt{z=x-4}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x,y}\}\\
&[\![\texttt{z=x-4}]\!] = ([\![\texttt{z>0}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{z>0}]\!] = [\![\texttt{x=x/2}]\!] \cup [\![\texttt{z=z-1}]\!] \cup \{\texttt{z}\}\\
&[\![\texttt{x=x/2}]\!] = ([\![\texttt{z=z-1}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{z=z-1}]\!] = ([\![\texttt{x>1}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{z}\}\\
&[\![\texttt{输出 x}]\!] = [\![\texttt{退出}]\!] \cup \{\texttt{x}\}\\
&[\![\texttt{退出}]\!] = \{\}
\end{aligned}
$$

其最小解为：

[[入口]] = {}
[[变量 x,y,z]] = {}
[[x=输入]] = {}
[[x>1]] = {x}
[[y=x/2]] = {x}
[[y>3]] = {x, y}
[[x=x-y]] = {x, y}
[[z=x-4]] = {x}
[[z>0]] = {x, z}
[[x=x/2]] = {x, z}
[[z=z-1]] = {x, z}
[[输出 x]] = {x}
[[退出]] = {}

根据这些信息，一个聪明的编译器可以推断出 y和 z永远不会同时存在，而且在赋值 z=z-1中写入的值从未被读取。 因此，程序可以安全地进行优化为：

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

这样可以节省一个赋值的成本，并且可能导致更好的寄存器分配。

我们可以估计这个分析的最坏情况复杂度，例如使用第5.1节中的朴素算法。我们首先观察到，如果程序有n个CFG节点和k个变量，那么lattice($^{2^V}$ars)n的高度为$k{\cdot}n$，这限制了我们可以执行的迭代次数。 每个lattice元素可以表示为长度为$k{\cdot}n$的位向量。 对于每次迭代，我们需要对大小为k的集合执行$O(n)$的交集、差集或相等操作，总共需要时间$O(k{\cdot}n)$。 因此，总的时间复杂度为$O(k^{\wedge 2}{\cdot}\ n^{\wedge 2})$。

练习 **5.11**：如果使用工作列表算法，活跃性分析的最坏情况复杂度是多少？

5.4 示例：可用表达式

在程序中，如果一个非平凡的表达式在程序点可用，那么它的当前值在执行过程中已经被计算过了。 对于所有程序点的可用表达式集合可以使用数据流进行近似计算。

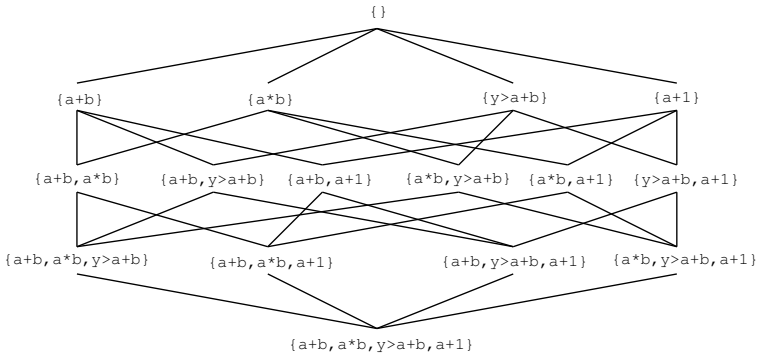
分析。 我们使用的格子的元素是程序中出现的所有表达式，并按照逆子集包含进行排序。 对于这个具体的程序：

```
变量 x, y, z, a, b;
z = a+b;
y = a*b;
while (y > a+b) {
  a = a+1;
  x = a+b;
}
```

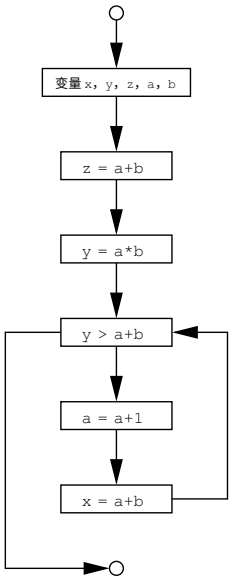
我们有4个不同的非平凡表达式，所以我们的格子是：

$$L = (2^{\{\texttt{a+b},\texttt{a*b},\texttt{y>a+b},\texttt{a+1}\}}, \supseteq)$$

看起来像：



我们格子中最大的元素是 $\emptyset$，它对应于平凡的信息。 与上述程序对应的流程图是：



对于每个CFG节点 v，我们引入一个约束变量 $[[v]]$，范围在 L内。 我们的意图是它应该包含在该节点之后程序点上始终可用的表达式子集。 例如，在循环中的条件中，表达式 a+b是可用的，但在循环中的最后一个赋值语句中不可用。 我们的分析是保守的，因为计算得到的集合可能太小。 数据流约束定义如下，这次我们定义如下：

$$JOIN(v) = \bigcap_{w \in pred(v)} [[w]]$$

对于入口节点，我们有以下约束条件：

$$[[\text{入口}]] = \{\}$$

如果v包含条件E或语句输出E，则约束条件为：

$$[[v]] = JOIN(v) \cup exps(E)$$

如果v包含形如$id=E$的赋值语句，则约束条件为：

$$[[v]] = (JOIN(v) \cup exps(E)) \downarrow id$$

对于其他类型的节点，约束条件为：

$$[[v]] = JOIN(v)$$

这里的函数$\downarrow id$会移除所有包含对变量 id的引用的表达式，而 $exps$函数定义如下：

$exps(intconst) = \emptyset$
$exps(id) = \emptyset$
$exps(input) = \emptyset$
$exps(E_1 \text{op} E_2) = \{E_1 \text{op} E_2\} \cup exps(E_1) \cup exps(E_2)$

其中 op是任何二进制运算符。 直觉是，如果一个表达式可以从所有进入的边获得或者在节点v中计算得到，那么它就在节点v中可用，除非它的值被赋值语句销毁。 同样，约束的右侧是单调函数。 对于示例程序，我们生成以下具体约束：

$[[\text{入口}]] = \{\}$
$[[\text{变量 x,y,z,a,b}]] = [[\text{入口}]]$
$[[\text{z=a+b}]] = exps(\text{a+b}) \downarrow \text{z}$
$[[\text{y=a*b}]] = ([[\text{z=a+b}]] \cup exps(\text{a*b})) \downarrow \text{y}$
$[[\text{y>a+b}]] = ([[\text{y=a*b}]] \cap [[\text{x=a+b}]]) \cup exps(\text{y>a+b})$
$[[\text{a=a+1}]] = ([[\text{y>a+b}]] \cup exps(\text{a+1})) \downarrow \text{a}$
$[[\text{x=a+b}]] = ([[\text{a=a+1}]] \cup exps(\text{a+b})) \downarrow \text{x}$
$[[\text{退出}]] = [[\text{y>a+b}]]$

使用固定点算法，我们得到最小解：

$[[\text{入口}]] = \{\}$
$[[\text{变量 x,y,z,a,b}]] = \{\}$
$[[\text{z=a+b}]] = \{\text{a+b}\}$
$[[\text{y=a*b}]] = \{\text{a+b}, \text{a*b}\}$
$[[\text{y>a+b}]] = \{\text{a+b}, \text{y>a+b}\}$
$[[\text{a=a+1}]] = \{\}$
$[[\text{x=a+b}]] = \{\text{a+b}\}$
$[[\text{退出}]] = \{\text{a+b}, \text{y>a+b}\}$

这证实了我们关于 a+b的假设。 观察到在节点 v之前的程序点上可用的表达式可以计算为 $JOIN(v)$。 有了这个知识，优化编译器可以系统地将程序转换为（稍微）更高效的版本：

```
变量 x,y,z,a,b,aplusb;
aplusb = a+b;
z = aplusb;
y = a*b;
while (y > aplusb) {
  a = a+1;
  aplusb = a+b;
  x = aplusb;
}
```

同时保证保留语义。

我们可以再次估计分析的最坏情况复杂度。 我们首先观察到，如果程序有n个CFG节点和k个非平凡表达式，那么

格有k个高度$\cdot n$，这限制了我们执行的迭代次数。
每个格元素可以表示为长度为k的位向量。 对于每次迭代，我们必须执行$O(n)$次交集、并集或相等操作，总共需要时间$O(kn)$。 因此，总的时间复杂度为$O(k^{\wedge}2n^{\wedge}2)$。

5.5 示例：非常繁忙的表达式

如果一个表达式在其值改变之前肯定会再次被计算，则称其为非常繁忙。 为了近似这个属性，我们需要与可用表达式相同的格和辅助函数。 对于每个CFG节点v，变量$[[v]]$表示在节点之前的程序点上肯定繁忙的表达式集合。我们定义：

$$JOIN(v) = \bigcap_{w \in succ(v)} [[w]]$$

退出节点的约束是：

$$[[exit]] = \{\}$$

对于条件和 输出 语句，我们有：

$$[[v]] = JOIN(v) \cup exps(E)$$

对于赋值语句，约束是：

$$[[v]] = JOIN(v) \downarrow id \cup exps(E)$$

对于其他所有节点，我们有约束：

$$[[v]] = JOIN(v)$$

直观上，如果一个表达式在当前节点中被计算，或者在所有未来的执行中都会被计算，除非赋值改变了它的值，那么它就是非常繁忙的。在示例程序中：

```
var x,a,b;
x = input;
a = x-1;
b = x-2;
while (x>0) {
  output a*b-x;
  x = x-1;
}
output a*b;
```

分析显示，在循环内部 a*b非常繁忙。 编译器可以执行代码提升，并将计算移动到最早的程序点，其中它非常繁忙。 这将把程序转换为更高效的版本：

```
var x,a,b,atimesb;
x = input;
a = x-1;
b = x-2;
atimesb = a*b;
while (x>0) {
  output atimesb-x;
  x = x-1;
}
output atimesb;
```

5.6 示例：到达定义

给定程序点的到达定义是可能定义变量当前值的赋值。 对于这个分析，我们需要一个包含程序中所有赋值（实际上是CFG节点）的幂集格。

对于之前的示例程序：

```
var x,y,z;
x = input;
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

格变为：

$$L = (2^{\{\texttt{x=input},\texttt{y=x/2},\texttt{x=x-y},\texttt{z=x-4},\texttt{x=x/2},\texttt{z=z-1}\}}, \subseteq)$$

对于每个CFG节点v，变量 $[[\,v]]$表示可能在节点后的程序点上定义变量的赋值集合。我们定义

$$JOIN(v) = \bigcup_{w \in pred(v)} [[w]]$$

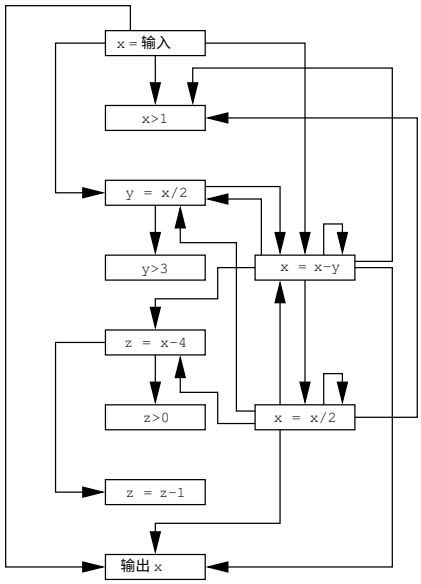
对于赋值语句，约束是：

$$[[v]] = JOIN(v)\downarrow id \ \cup \{v\}$$

对于所有其他节点，它只是：

$$[[v]] = JOIN(v)$$

这次 $\downarrow id$函数会删除所有对变量 *id*的赋值。 这种分析可以用来构建一个def-use图，它类似于CFG，但边从定义指向可能的使用。 对于示例程序，def-use图如下：



def-use图是程序的进一步抽象，是优化如死代码消除和代码移动的基础。

练习**5.12**：证明d**e**f-**use**图始终是CFG的传递闭包的子图。

5.7 正向，反向，可能和必须

我们迄今为止看到的四种经典分析可以以各种方式进行分类。 它们都只是一般单调框架的实例，但它们的约束具有特定的结构。

前向分析是一种对每个程序点计算关于过去行为的信息的分析。 这些的例子是可用表达式和到达定义。 它们可以通过约束的右侧只依赖于CFG节点的前驱来进行描述。 因此，分析从入口节点开始，在CFG中向前移动。

反向分析是指对于每个程序点计算关于未来行为的信息。 这种分析的例子有活跃变量和非常繁忙表达式。 它们可以通过约束的右侧只依赖于CFG节点的后继来进行描述。 因此，分析从退出节点开始，在CFG中向后移动。

可能性分析是指描述可能为真的信息，并计算一个上界近似。 这种分析的例子有活跃变量和到达定义。 它们可以通过使用并集运算符组合信息来描述约束的右侧。

必然性分析是指描述必然为真的信息，并计算一个下界近似。 这种分析的例子有可用表达式和非常繁忙表达式。 它们可以通过使用交集运算符组合信息来描述约束的右侧。

因此，我们的四个例子展示了每种可能的组合，如下图所示：

	向前	向后
五月	到达定义	活跃性
必须	可用表达式	非常繁忙的表达式

这些分类大多是植物性质的，但了解它们可能会为构建新的分析提供灵感。

5.8 示例：初始化变量

让我们尝试定义一种分析，确保变量在读取之前被初始化。 这可以通过计算每个程序点上保证被初始化的变量集来解决，因此我们的格是给定程序中出现的变量的反向幂集。 初始化是过去的属性，所以我们需要进行向前分析。 此外，我们需要明确的信息，这意味着必须进行分析。 这意味着我们的约束是以前任和交集的形式表达的。 基于此，它们或多或少是自给自足的。 对于入口节点，我们有以下约束：

$$[\![\text{入口}]\!] = \{\}$$

对于作业，我们有以下约束条件：

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!] \cup \{id\}$$

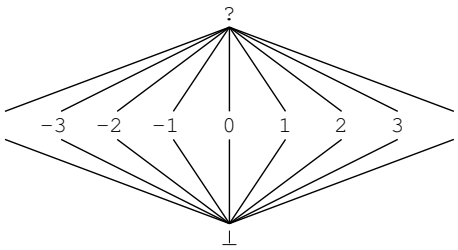
对于其他所有节点，约束为：

$$[\![v]\!] = \bigcap_{w \in pred(v)} [\![w]\!]$$

编译器现在可以检查每个变量的使用情况，确保它包含在计算得到的初始化变量集合中。

5.9 示例：常量传播

与符号分析相关的一种分析是常量传播，我们希望确定每个程序点上具有常量值的变量。 这种分析的结构与符号分析完全相同，只是基本格子被替换为：



并且运算符以以下方式进行抽象，例如加法：

$\lambda n \lambda m.$`if (`$n = \bot \vee m = \bot$`) {`$\bot$`} else if (`$n =$ `?`$\vee m =$ `?) {?} else {`$n+m$`}`

基于这种分析，优化编译器可以转换程序：

```
var x,y,z;
x = 27;
y = input;
z = 2*x+y;
if (x < 0) { y = z-3; } else { y = 12; }
输出 y;
```

转换为:

```
var x,y,z;
x = 27;
y = 输入;
z = 54+y;
if (0) { y = z-3; } else { y = 12; }
输出 y;
```

经过可达定义分析和死代码消除后，可以简化为:

```
var y;
y = 输入;
输出 12;
```

5.10 示例：区间分析

区间分析计算每个整数变量的下界和上界，这是一种有趣的分析结果，因为 区间分析可以用于优化，例如数组边界检查、数值溢出和高效的整数表示。

可以使用无限高度的格子进行优化，我们必须使用一种特殊的技术，称为扩展，以确保向固定点的收敛。

通过使用一种称为缩小的补充技术，可以获得更高的精度。 增加的精度可以通过使用一种称为缩小的补充技术获得。

描述单个变量的格子被定义为：

$$\text{区间} = \text{提升}(\{[l,\, h] \mid l,\, h \in N \ \ \wedge \ \ l \leq h\})$$

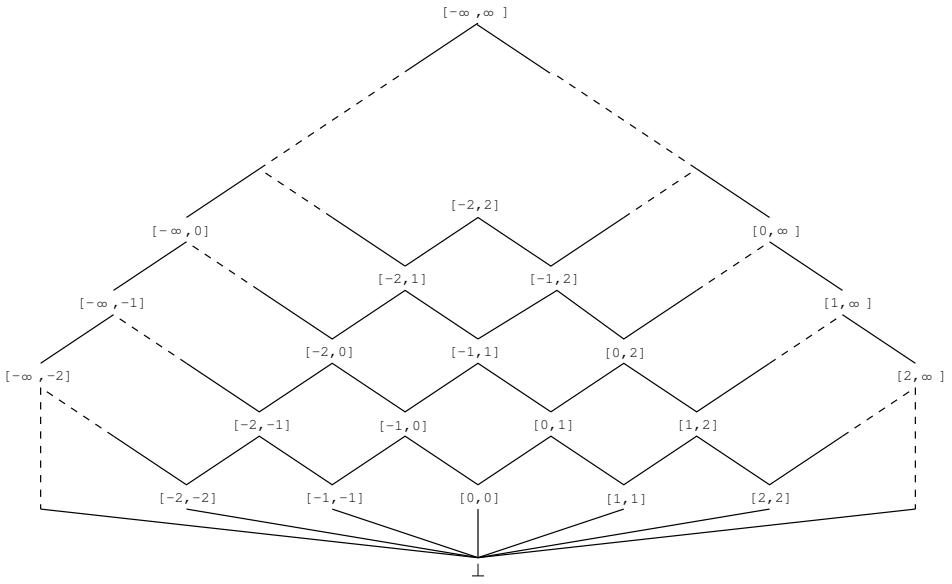
在哪里：

$$N = \{-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty\}$$

是在无限端点上扩展的整数集合，区间的顺序是：

$$[l_1, h_1] \sqsubseteq [l_2, h_2] \ \Leftrightarrow \ l_2 \leq l_1 \wedge h_1 \leq h_2$$

对应于点的包含关系。 这个格子看起来如下：



很明显，我们没有一个有限高度的格子，因为它包含了无限链，例如：

$$[0,0] \sqsubseteq [0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \sqsubseteq [0,4] \sqsubseteq [0,5] \ldots$$

这一点延伸到了我们最终使用的格子，即：

$$L = \mathit{Vars} \to \mathit{Interval}$$

对于入口节点，我们使用返回 $\bot$元素的常数函数：

$$[\![\mathit{entry}]\!] = \lambda x.\bot$$

对于赋值操作，约束条件为：

$$[\![v]\!] = JOIN(v)\ [id \to eval(JOIN(v),\ E)]$$

对于其他所有节点，约束为：

$$[\![v]\!] = JOIN(v)$$

在哪里：

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

而且 $eval$执行表达式的抽象评估：

$$\begin{array}{l} eval(\sigma, id) = \sigma(id) \\ eval(\sigma, intconst) = [intconst, intconst] \\ eval(\sigma,\ E_1\ \mathsf{op}\ E_2) = \overline{\mathsf{op}}(eval(\sigma,\ E_1), eval(\sigma,\ E_2)) \end{array}$$

其中，所有的抽象算术运算符都由以下方式定义：

$$\overline{\mathsf{op}}([\text{制表符}_1,\ h_1], [\text{制表符}_2,\ h_2]) = [\min_{x \in [\text{制表符}_1, h_1], y \in [\text{制表符}_2, h_2]} x \text{操作} y,\ \max_{x \in [\text{制表符}_1, h_1], y \in [\text{制表符}_2, h_2]} x \text{操作} y]$$

例如，$\bar{+}([1, 10], [-5, 7]) = [1 - 5, 10 + 7] = [-4, 17].$

练习 **5.13**: 论证这些定义在区间格上产生单调算子。

练习 **5.14**: 展示如何使抽象比较运算符更加精确。

格子的高度是无限的，因此我们无法使用单调框架，因为固定点算法可能永远不会终止。 这意味着对于格子 L^n的近似序列:

$$F^i(\bot, \ldots, \bot)$$

永远不需要收敛。 解决这类问题的一种强大技术将在下一节中介绍。

5.11 扩大

为了获得在第5.10节中介绍的区间分析的收敛性，我们将使用一种称为 $widening$的技术，该技术引入了一个函数 $w : L^n \to L^n$，使得序列变为：

$$(w \circ F)^i(\bot, \ldots, \bot)$$

现在收敛于一个大于每个 $F^i(\bot, \ldots, \bot)$的固定点，并因此表示程序的正确信息。扩展函数 w

直观地说，它将足够粗糙的信息以确保终止。 对于我们的区间分析， w被定义为点对点到单个区间。 它相对于一个必须包含 $-\infty$和 ∞的固定有限子集 $B \subset N$进行操作。
通常情况下，B可以用给定程序中出现的所有整数常量进行初始化，但也可以使用其他启发式方法。 在一个单一的区间上，我们有：

$$w([l,h]) = [max\{i \in B \mid i \leq l\}, min\{i \in B \mid h \leq i\}]$$

它找到了允许的区间中最适合的区间。

练习**5.15**：证明由于w是一个广义单调函数，$w(I\text{nterva}l)$是一个有限格，所以$w$idening技术保证可以正确工作。

5.12 缩小

$w$idening通常会超过目标，但随后的一种称为narro$w$ing的技术可以改善结果。如果我们定义：

$$\text{fi}x = \bigsqcup F^i(\bot, \ldots, \bot) \qquad \text{fi}xw = \bigsqcup (w \circ F)^i(\bot, \ldots, \bot)$$

那么我们有fi$x \sqsubset$fi$x$w。 然而，我们还有fi$x \sqsubseteq F($fi$x$w$) \sqsubset$fi$x$w，这意味着对F的后续应用可能会改进我们的结果并产生可靠的信息。 事实上，这种称为narrowing的技术可以任意多次迭代。

练习 5.16 ：证明 $\forall i : fix \sqsubseteq F^{i+1}(fixw) \sqsubseteq F^i(fixw) \sqsubseteq fixw.$

一个例子将展示这些技术的好处。考虑这个
程序：

```
y = 0; x = 7; x = x+1;
while (input) {
  x = 7;
  x = x+1;
  y = y+1;
}
```

如果不进行扩展，分析将在循环后产生以下发散的近似序列：

$$\begin{array}{l}
[\mathtt{x} \to \bot, \mathtt{y} \to \bot] \\
[\mathtt{x} \to [8,8], \mathtt{y} \to [0,1]] \\
[\mathtt{x} \to [8,8], \mathtt{y} \to [0,2]] \\
[\mathtt{x} \to [8,8], \mathtt{y} \to [0,3]] \\
\vdots
\end{array}$$

如果我们应用扩大操作，基于集合 $B = \{-\infty, 0, 1, 7, \infty\}$ 种子中的程序中出现的常量，那么我们将得到一个收敛的序列：

$[\mathtt{x} \to \bot, \mathtt{y} \to \bot]$
$[\mathtt{x} \to [7, \infty], \mathtt{y} \to [0, 1]]$
$[\mathtt{x} \to [7, \infty], \mathtt{y} \to [0, 7]]$
$[\mathtt{x} \to [7, \infty], \mathtt{y} \to [0, \infty]]$

然而，对于 x的结果令人沮丧。 幸运的是，一些缩小的迭代将结果改进为：

$[\mathtt{x} \to [8, 8], \mathtt{y} \to [0, \infty]]$

这实际上是我们所能期望的最好结果。相应地，进一步的缩小没有效果。 请注意，这是一个递减的序列：

$$\mathit{fixw} \sqsupseteq F(\mathit{fixw}) \sqsupseteq F^2(\mathit{fixw}) \sqsupseteq F^3(\mathit{fixw}) \ldots$$

不能保证收敛，因此启发式算法必须确定应用缩小的次数。

第六章

路径敏感性

到目前为止，我们通过简单地将 if-和while-语句视为两个分支之间的非确定性选择来忽略条件的值。 这被称为路径不敏感分析，因为它不区分导致给定程序点的不同路径。 这种技术未能包含一些可能在静态分析中使用的信息。 例如，考虑以下程序：

```
x = 输入;
y = 0;
z = 0;
while (x > 0) {
  z = z+x;
  if (17 > y) { y = y+1; }
  x = x-1;
}
```

先前的区间分析（带有扩大）将得出以下结论：在 while-循环之后，变量 x在区间 $[-\infty, \infty]$内，变量 y在区间 $[0, \infty]$内，变量 z在区间 $[-\infty, \infty]$内。 然而，考虑到使用了条件语句，这个结果过于悲观。

6.1 断言

为了利用可用的信息，我们将扩展语言，添加一个人工语句 assert(E)，其中 E是一个布尔表达式。 这个语句将在运行时如果 E为假则中止执行，否则没有任何效果，然而，我们只会在 E保证为真的地方插入它。 在区间分析中，这些新语句的约束条件将通过利用条件语句中的信息来缩小各个变量的区间。

在示例程序中，条件的含义可以通过以下程序转换进行编码：

```
x = 输入;
y = 0;
z = 0;
当 (x > 0) 时 {
  断言(x > 0);
  z = z+x;
  如果 (17 > y) { 断言(17 > y); y = y+1; }
  x = x-1;
}
断言(!(x > 0));
```

对于带有断言语句的节点 v 的约束可以简单地给出为：

$$[\![v]\!] = JOIN(v)$$

在这种情况下，不会获得额外的精度。事实上，需要深入了解具体的静态分析才能为这些结构定义非平凡且正确的约束。

对于区间分析，提取与格元素相关的一般条件或谓词（如 $E_1 > E_2$ 或 $E_1 == E_2$）所携带的信息是复杂的，本身就是一个需要大量研究的领域。为了简单起见，我们只考虑两种类型的条件：$\mathrm{id} > E$ 和 $E > \mathrm{id}$。前一种断言可以通过约束来处理

$$[\![v]\!] = JOIN(v)[id \to gt(JOIN(v)(id), eval(JOIN(v), E))]$$

其中

$$gt([l_1, h_1], [l_2, h_2]) = [l_1, h_1] \sqcap [l_2, \infty]$$

> 练习 **6.1**：证明这个对于 `assert`的约束是正确且单调的。

否定条件以类似的方式处理，而其他条件
被给予平凡但正确的恒等约束。

通过这个细化，上述示例的区间分析将得出结论：在 `while`循环之后，变量 `x`在区间 $[-\infty..0]$内， `y`在区间 $[0, 17]$内， `z`在区间 $[0, \infty]$内。

> 练习**6.2**：讨论如何给出更多的条件来给**assert**提供非平凡的约束，以进一步提高分析精度。

6.2 分支相关性

在条件分支处使用assert语句提供了一种简单的路径敏感性，称为控制敏感性，但对于推理程序中分支的相关性来说是不足够的。这里是一个典型的例子：

```
 if (条件) {
  打开();
  标志 = 1;
} else {
  标志 = 0;
}
...
if (flag) {
   关闭();
}
```

我们在这里假设 open和 close是用于打开和关闭特定文件的内置函数。 文件最初是关闭的，而"..." "由不调用 open或 close或修改 flag的语句组成。 我们希望设计一个分析，可以检查只有在文件当前打开时才调用 close。

作为起点，我们使用这个格子来模拟文件的打开/关闭状态：

$$L = (2^{\{\text{打开,关闭}\}}, \subseteq)$$

对于每个CFG节点 v变量 $[[v]]$表示节点后程序点处文件的可能状态。 对于打开和关闭语句，约束条件为：

$$[[\texttt{open()}]] = \{\text{打开}\}$$

$$[[\texttt{close()}]] = \{\text{关闭 }\}$$

对于入口节点，我们定义：

$$[[entry]] = \{\text{关闭 }\}$$

对于其他不修改文件状态的节点，约束条件简单地为：

$$[[v]] = JOIN(v)$$

其中 $JOIN$按照正常的向前、可能性分析定义：

$$JOIN(v) = \bigcup_{w \in pred(v)} [[w]]$$

在示例程序中， close函数只有在open被调用时才会被调用，但是当前分析无法发现这一点。

> 练习**6.3**：编写示例程序产生的约束条件，并展示 [[flag]]（最后一个 if条件的节点）的解为
> {打开,关闭}.

争论程序是否具有所需属性显然涉及到
标志变量，而上面的格子忽略了它。 因此，我们可以尝试稍微

更复杂的格子 - 一个乘积格子，它同时跟踪文件的状态和标志的值:

$$L' = (2^{\{\text{打开},\text{关闭}\}}, \subseteq) \times (2^{\{\text{标志}^0,\text{标志}^{0}\}}, \subseteq)$$

此外，我们插入 assert来确保条件不被忽略：

```
if (条件) {
  assert(条件);
  打开();
  标志 = 1;
} else {
  assert(!条件);
  标志 = 0;
}
...
if (标志) {
  assert(标志);
  关闭();
} else {
  assert(!标志);
}
```

然而，这仍然不够。 在第一个 `if-else`
语句之后的程序点，分析只知道可能调用了打开并且标志
可能为0。

> 练习**6.4**：指定与 L'格子相匹配的约束条件。 然后展示分析在第一个 `if-else`语句之后的第一个节点产生格子元素 $(2^{\{\text{开},\text{闭}\}}, 2^{\{\text{标志}^{=0},\text{标志}^{=0}\}})$。

目前的分析也被称为独立属性分析，因为文件的抽象值独立于布尔标志的抽象值。
我们需要的是一种关系分析，可以跟踪变量之间的关系。 这可以通过将分析推广到每个程序点维护多个抽象状态来实现。 如果 L是上述定义的原始格子，我们用以下方式替换它

$$L'' = P \to L$$

其中 P是有限集合的路径上下文。 这里的路径上下文是关于程序状态的谓词。（例如，TIP中的条件表达式定义了这样的谓词。）一般来说，每个语句都在 $|P|$不同的路径上下文中进行分析，每个路径上下文描述了通向该语句的一组路径。 对于上面的例子，我们可以使用 $P = \{\text{标志}= 0,\text{标志}= 0\}$。

对于打开、关闭和输入的约束条件是：

$$[\![\texttt{open()}]\!] = \lambda p.\{\text{open}\}$$

$$[\![\texttt{close()}]\!] = \lambda p.\{\text{closed}\}$$

$$[\![entry]\!] = \lambda p.\{\text{closed}\}$$

对于赋值语句的约束条件确保 flag得到特殊处理：

$$[\![\texttt{flag = 0}\ ]\!] = [\texttt{flag} = 0 \to \bigcup_{p \in P} JOIN(v)(p),\ \texttt{flag} = 0 \to \emptyset]$$

$$[\![\texttt{flag =}\ \ n]\!] = [\texttt{flag} = 0 \to \bigcup_{p \in p} JOIN(v)(p),\ \texttt{flag} = 0 \to \emptyset]$$

$$[\![\texttt{flag =}\ \ E]\!] = \lambda q. \bigcup_{p \in P} JOIN(v)(p)$$

在这里， n是一个非零的整数常量， E是一个非整数常量表达式， $JOIN$ 是逐点定义的：

$$JOIN(v)(p) = \bigcup_{w \in pred(v)} [\![w]\!](p)$$

情况 $[\![v]\!](p) = \emptyset$对应于 v处于不可行路径上下文的情况。
对于 assert，我们还对 flag给予特殊处理：

$$[\![\texttt{assert(flag)}]\!] = [\texttt{flag} = 0 \to JOIN(v)(\texttt{flag} = 0),\ \texttt{flag} = 0 \to \emptyset]$$

请注意与 flag
= 1情况的约束条件之间的细微但重要的差异。 与之前一样，否定表达式的情况类似。

> 练习**6.5**：为assert(!flag)给出适当的约束条件。

最后，对于任何其他节点 v，包括其他 assert节点，约束条件保持不同路径上下文的数据流信息分开，但否则只是传播和合并信息：

$$[\![v]\!] = \lambda p.JOIN(v)(p)$$

虽然这是正确的，但我们可以通过识别其他符合格子给定的抽象的模式来制定更精确的约束条件 assert节点。

对于我们的具体程序，生成了以下约束条件：

$[\![$入口$]\!] = \lambda p.\{$闭合$\}$
$[\![$条件 $]\!] = [\![$入口 $]\!]$
$[\![$assert(条件) $]\!] = [\![$条件 $]\!]$
$[\![$打开()$]\!] = \lambda$p.{打开}
$[\![\texttt{flag = 1}\ ]\!] = \big[\texttt{flag} = 0 \to \bigcup_{p \in P} [\![$打开()$]\!](p),\ \texttt{flag} = 0 \to \emptyset\big]$
$[\![$assert(!条件) $]\!] = [\![$条件 $]\!]$
$[\![\texttt{flag = 0}\ ]\!] = \big[\texttt{flag} = 0 \to \bigcup_{p \in P} [\![$assert(!条件) $]\!](p),\ \texttt{flag} = 0 \to \emptyset\big]$
$[\![\texttt{...}]\!] = \lambda p.\big([\![\texttt{flag = 1}\ ]\!](p) \cup [\![\texttt{flag = 0}\ ]\!](p)\big)$
$[\![\texttt{flag}]\!] = [\![\texttt{...}]\!]$

$[\![\texttt{assert(flag)}]\!] = [\texttt{flag} = 0 \to [\![\texttt{flag}]\!](\texttt{flag} = 0),\ \texttt{flag} = 0 \to \emptyset]$
$[\![\texttt{close()}]\!] = \lambda p.\{\text{closed}\}$
$[\![\texttt{assert(!flag)}]\!] = [\texttt{flag} = 0 \to [\![\texttt{flag}]\!](\texttt{flag} = 0),\ \texttt{flag} = 0 \to \emptyset]$
$[\![\textit{exit}]\!] = \lambda p.\big([\![\texttt{close()}]\!](p) \cup [\![\texttt{assert(!flag)}]\!](p)\big)$

最小解是，对于每个 $[\![v]\!](p)$：

	$\texttt{flag} = 0$	$\texttt{flag} = 0$
$[\![\textit{entry}]\!]$	{关闭 }	{关闭 }
[条件]]	{关闭 }	{关闭 }
[断言(条件)]]	{关闭 }	{关闭 }
[[打开()]]	{打开}	{打开}
[flag = 1]]	$\emptyset$	{打开}
[断言(!条件)]]	{关闭 }	{关闭 }
[flag = 0]]	{关闭 }	$\emptyset$
[[...]]	{关闭 }	{打开}
[flag]]	{关闭 }	{打开}
[断言(flag)]]	$\emptyset$	{打开}
[关闭()]]	{关闭 }	{关闭 }
[断言(!flag)]]	{关闭 }	$\emptyset$
[[退出]]	{关闭 }	{关闭 }

分析产生的格元素为 $[\texttt{flag}= 0 \to \{\text{关闭}\}, \texttt{flag}= 0 \to \{\text{打开}\}]$对于第一个 if-else语句之后的程序点。对于assert(flag)语句的约束将消除文件在该点关闭的可能性。这确保了只有在文件打开时才调用 close，如期望的那样。

练习**6.6**：对于当前示例，基本格 L被定义为有限集合 A的幂集。 证明$P \to 2^A$与 $2^{P \times A}$同构。 （这解释了为什么这样的分析被称为关系型。）

练习**6.7**：描述一个改进了的示例程序的变体，在与常量传播相结合时，当前分析将得到改进。

一般来说，程序分析设计师可以选择 P。 Of-ten P由出现在程序条件语句中的谓词的组合组成。 这很快导致指数级增长：对于 k个谓词，每个语句可能需要在 2^k个不同的路径上下文中进行分析。 然而，在实践中，这些分析步骤通常存在很多冗余。 因此，除了在格元素相对于 assertpredicates进行推理的挑战之外，在路径敏感分析中避免过多的冗余计算也需要相当大的努力。 一种方法是迭代细化，其中 P最初是一个单一的通用路径上下文，然后通过添加相关谓词进行迭代细化，直到可以建立或证明所需的属性，或者分析无法选择相关谓词。

因此放弃了。

练习**6.8**：假设我们将 open的规则更改为

$$[[\texttt{open()}]] = \lambda p.\{\texttt{open}\}$$

到

$[[\texttt{open()}]] = \lambda p.$如果 $JOIN(v)(p) = \emptyset$那么 $\emptyset$否则 $\{\texttt{open}\}$论证这是正确的，并且对于某些程序比原始规则更精确。

练习**6.9**：以下是先前示例程序的变体：

```
if (条件) {
  flag = 1;
} else {
  flag = 0;
}
...
if (条件) {
  open();
}
...
if (flag) {
  关闭();
}
```

展示一种路径敏感分析如何证明对于这个变体，只有在 open被调用时才会调用 close。

练习**6.10**：构建另一个 open/close示例程序的变体，其中只有选择 P才能建立所需的属性，该属性在程序源代码中不作为条件表达式出现。（这样的程序可能难以使用迭代细化技术处理。）

第七章

程序间分析

到目前为止，我们只分析了单个函数的主体部分，这被称为函数内部分析。 现在我们考虑整个程序的函数间分析，包括多个函数和函数调用。

7.1 过程间控制流图

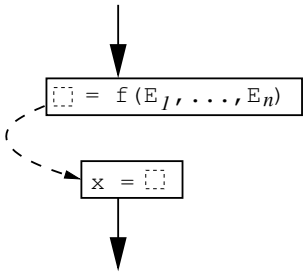
我们使用包含函数的TIP语言的子集，但仍然忽略指针。 正如我们将看到的，整个程序的控制流图非常简单易得。 当添加函数指针时，情况变得更加复杂，我们将在后面的章节中讨论。

首先，我们像往常一样构建所有单个函数主体的控制流图。然后，我们只需将它们粘合在一起，以正确反映函数调用。 我们需要注意参数传递、返回值和跨函数调用的局部变量值。 为了简单起见，我们假设所有函数调用都是在赋值的情况下进行的：

$$\texttt{x = f(E}_1\texttt{,...,E}_n\texttt{);}$$

> 练习 **7.1**：通过引入新的临时变量，展示任何程序可以重写为这种形式。

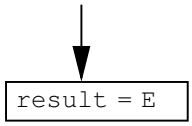
在CFG中，我们使用两个节点来表示每个函数调用语句：一个表示从调用者到f的入口的调用节点，一个表示从f的出口返回到调用者的调用后节点：



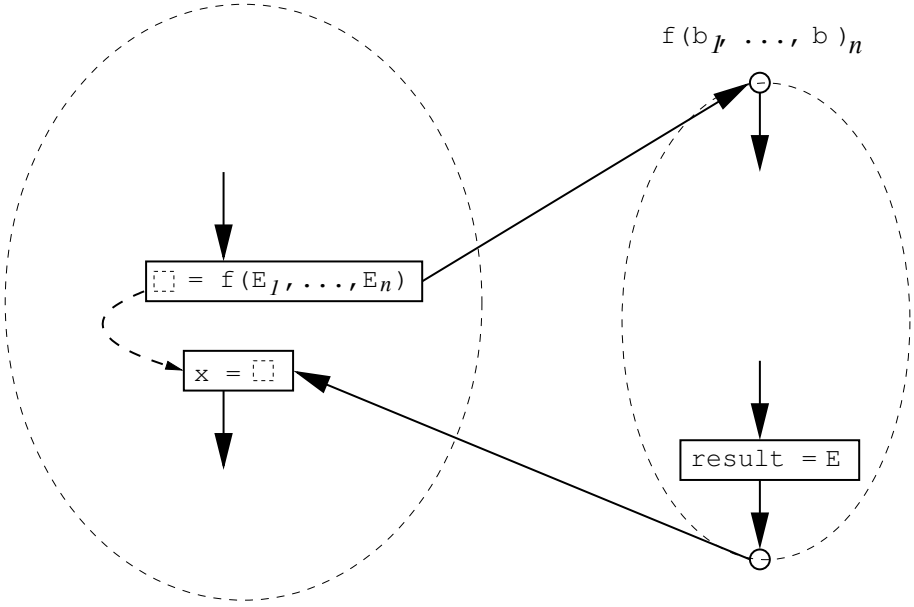
接下来，我们表示每个返回语句

返回 E;

作为一个使用特殊变量命名为 result的赋值：



我们现在可以将调用者和被调用者如下连接在一起：



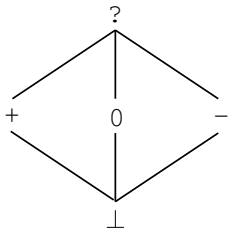
调用节点和其调用后节点之间的连接由一个特殊边表示，我们需要用它来传播调用者的局部变量的抽象值。

有了这个过程间控制流图，我们可以应用单调框架。 示例在以下章节中给出。

练习 7.2：在一个有
n个CFG节点的程序中，过程间CFG可能包含多少条边?

7.2 示例：过程间符号分析

回顾第4.1节和第5.2节的过程内符号分析。该分析使用格 $Sign$来模拟值：



抽象状态通过映射格 $Vars \to Sign$来表示。为了使分析过程间化，我们为
函数入口和出口定义约束条件。 对于函数 f(b$_1$, ..., b$_n$)的入口节点 v，我们考虑所有调用者的抽象状态 $pred(v)$，并模拟参数的传递E$_1$, ..., E$_n$：

$$[\![v]\!] = \bigsqcup_{w \in pred(v)} \text{对于 } [\![w]\!] \text{，有 } \{\mathtt{b}_1 \to eval([\![w]\!], \mathtt{E}_1), \ldots, \mathtt{b}_n \to eval([\![w]\!], \mathtt{E}_n)\} \bot$$

对于一个存储返回值在变量 x中的调用后节点 v，其中v'是相应的调用节点，数据流可以通过以下约束建模：

$$[\![v]\!] = [\![v']\!][\mathtt{x} \to [\![w]\!](\mathtt{result})] \quad \text{其中} \quad w \in pred(v)$$

我们利用一个事实，即调用后节点 v恰好有一个前驱 $w \in pred(v)$。 该约束从调用节点 v'获取局部变量的抽象值，并从 w获取 result的抽象值。

在向后分析中，我们将考虑调用节点和函数退出节点，而不是函数入口节点和调用后节点。 还要注意，我们利用了TIP语言的变体在这里没有全局变量、堆、嵌套函数或高阶函数的事实。

第八章

控制流分析

如果我们将高阶函数、对象或函数指针引入编程语言中，那么控制流和数据流就会突然交织在一起。 在每个调用点，很难看出正在调用哪段代码。 控制流分析的任务是以保守的方式近似计算这类语言的过程间控制流图。

8.1 λ-演算的闭包分析

纯粹的控制流分析最好通过经典的λ-演算来说明：

$$
\begin{array}{l}
E \rightarrow \lambda id.E \\
\quad \mid \text{标识符} \\
\quad \mid E\ E
\end{array}
$$

稍后我们将将这种技术推广到完整的TIP语言。 为了简化起见，我们假设所有的λ-绑定变量都是不同的。 为了构建这个演算中一个项的控制流图，我们需要计算每个表达式 E可能求值为的闭包集合。 闭包可以由形如 λid的符号来模拟，它标识了一个具体的λ-抽象。 这个问题被称为闭包分析，可以使用变种的单调框架来解决。 然而，由于这种语言中的过程内控制流是平凡的，我们可以直接在AST上执行分析。

我们使用的格是给定术语中出现的闭包的幂集按子集包含排序。 对于每个语法树节点 v我们引入一个约束变量 $[[v]]$表示结果闭包的集合。 对于一个抽象 $\lambda id.E$我们有约束条件：

$$\lambda id \in [[\lambda id.E]]$$

（函数可能会评估为自身），对于一个应用 $E_1 E_2$条件约束为：

$$\lambda id \in [[E_1]] \Rightarrow \left([[E_2]] \subseteq [[id]] \ \wedge \ [[E]] \subseteq [[E_1 E_2]]\right)$$

对于每个闭包 $\lambda id.E$（实际参数可能流入形式参数，并且函数体的值是函数调用的可能结果之一）。

> 练习**8.1**：展示如何将得到的约束条件转化为标准的单调不等式，并通过固定点计算求解。

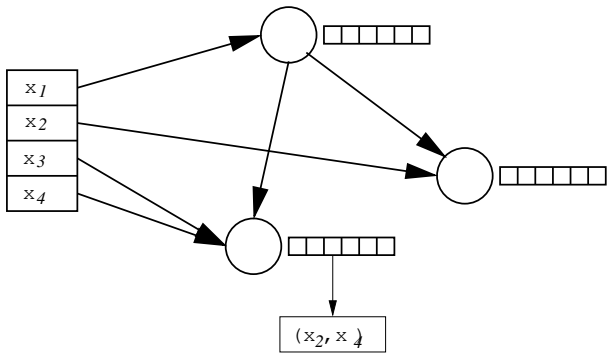
8.2 立方算法

闭包分析的约束条件是一个可以在立方时间内求解的一般类的实例。 许多问题属于这个类别，因此我们将更详细地研究算法。

我们有一个有限的令牌集合$\{t_1, \ldots, t_k\}$和一个有限的变量集合$x_1, \ldots, x_n$其值是令牌的集合。 我们的任务是读取一组形式为$t \in x$或$t \in x \Rightarrow y \subseteq z$的约束条件，并生成最小解。

> 练习**8.2**：展示唯一的最小解存在，因为解在交集下是封闭的。

该算法基于一个简单的数据结构。 每个变量都映射到一个有向无环图（DAG）中的节点。 每个节点都有一个关联的位向量，属于$\{0,1\}^k$，最初定义为全0。 每个位都有一个关联的变量对列表，用于建模条件约束。 DAG中的边反映包含关系约束。 位向量将始终直接表示最小解。 一个示例图可能如下所示：



约束条件逐个添加。 形如 $t \in x$的约束条件通过查找与 x相关联的节点并将相应的位设置为1来处理。
如果其配对列表不为空，则在相应的节点之间添加一条边。

对于每一对 (y, z)，都会为to y和 z之间添加一条边，并清空列表。 形如 $t \in x \Rightarrow y \subseteq z$的约束条件首先测试与 x对应的节点中与 t对应的位是否为1。 如果是这样，则在与 y和 z对应的节点之间添加一条边。 否则，将 (y, z)添加到该位的列表中。

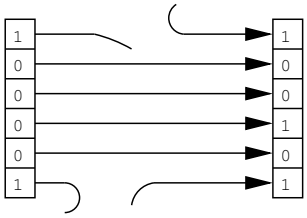
如果新添加的边形成一个循环，则将该循环上的所有节点合并为一个单独的节点，这意味着它们的位向量被合并在一起，并且它们的配对列表被连接起来。 变量到节点的映射相应地更新。 无论如何，为了重新建立所有包含关系，我们必须沿着图中的所有边传播每个新设置的位的值。

为了分析该算法的时间复杂度，我们假设令牌和变量的数量都是 $O(n)$。 在程序的闭包分析中，这显然是成立的，程序的大小为 n。

在循环中合并DAG节点的操作最多可以执行 $O(n)$次。 每次合并涉及的节点最多为 $O(n)$个，并且它们的位向量的并集计算时间最多为 $O(n^2)$。 这部分的总计算量为 $O(n^3)$。

新边最多插入 $O(n^2)$次。 常量集合最多包含 $O(n^2)$次，每个约束条件 $t \in x$一次。

最后，为了限制沿边传播位的成本，我们想象每条边上对应的位对通过一个微小的位线连接。
每当源位被设置为1时，该值沿着位线传播
然后被中断：



由于我们最多有 n^3位线，传播的总成本为 $O(n^3)$。 总结一下，算法的总成本也是 $O(n^3)$。 这似乎也是一个下界，被称为立方时间瓶颈。

该算法所涵盖的约束类型是更一般的集合约束的简单情况，允许对有限项集合进行更丰富的约束。 一般的集合约束也可以在时间 $O(2^{2^n})$内解决。

8.3 函数指针的控制流图

现在考虑我们允许函数指针的小语言。 对于一个计算的函数调用：

$$E \to (E)(E_1, \ldots, E_n)$$

从语法上看，我们无法确定可能被调用的函数。 通过假设具有正确的任何函数

参数的数量可以被调用。 然而，通过进行控制流分析，我们可以做得更好。 请注意，函数调用 $id(E_1,\ldots,E_n)$可以被视为通用符号 $(id)(E_1,\ldots,E_n)$的语法糖。我们的格是包含每个函数名 id的令牌集合的幂集，按子集包含关系排
序。 对于每个语法树节点 v，我们引入一个约束变量 $[[v]]$表示 v可能求值为的函数或函数指针的集合。 对于一个常量函数名 id，我们有约束条件：

$$id \in [[id]]$$

对于赋值 $id=E$，我们有约束条件：

$$[[E]] \subseteq [[id]]$$

最后，对于计算的函数调用，对于每个带有参数 $a_1,\ldots$ 和返回表达式 E'的函数 f的定义，我们有约束条件：

$$f \in [[E]] \Rightarrow \left([[E_i]] \subseteq [[a_i]] \wedge [[E']] \subseteq [[(E)(E_1,\ldots,E_n)]]\right)$$

如果我们限制自己只对可类型化的程序进行分析，并且只为那些函数 f生成类型正确的调用的约束，那么我们可以得到更精确的分析结果。

根据推断的信息，我们像以前一样构建CFG，但是根据控制流分析，在调用点和所有可能的目标函数之间建立边。 考虑以下示例程序：

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

控制流分析生成以下约束条件：

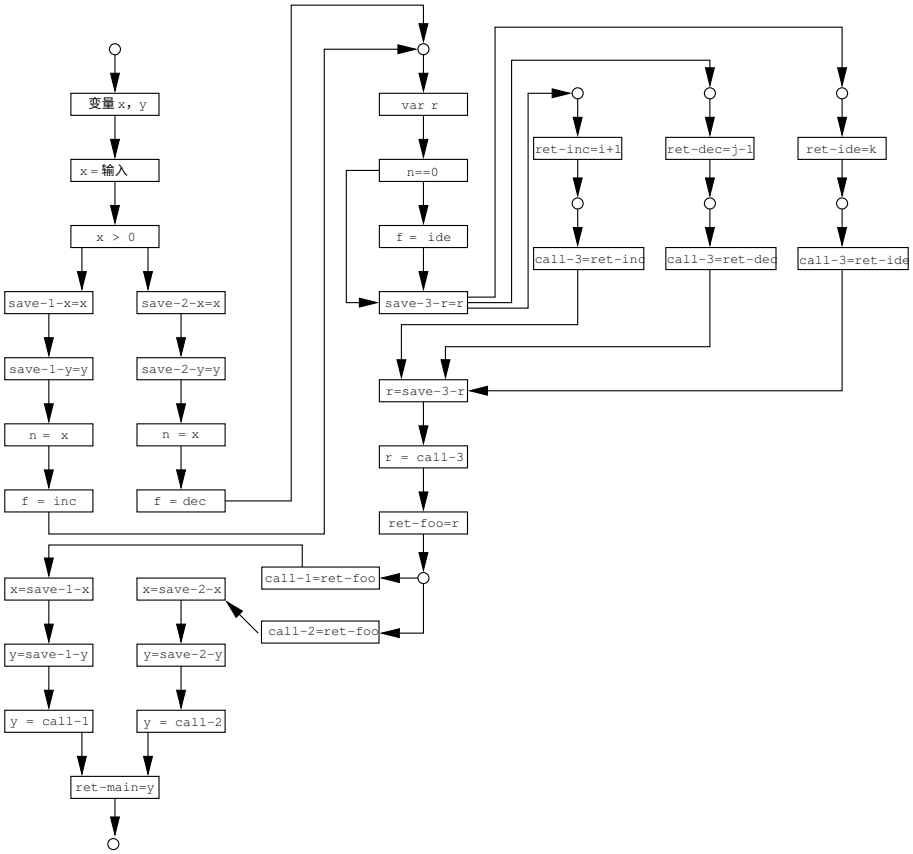
$$\texttt{inc} \in [[\texttt{inc}]]$$
$$\texttt{dec} \in [[\texttt{dec}]]$$

$$
\begin{aligned}
&\texttt{ide} \in [\![\texttt{ide}]\!] \\
&[\![\texttt{ide}]\!] \subseteq [\![\texttt{f}]\!] \\
&[\![\texttt{(f)(n)}]\!] \subseteq [\![\texttt{r}]\!] \\
&\texttt{inc} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{i}]\!] \ \wedge \ [\![\texttt{i+1}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
&\texttt{dec} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{j}]\!] \ \wedge \ [\![\texttt{j-1}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
&\texttt{ide} \in [\![\texttt{f}]\!] \Rightarrow [\![\texttt{n}]\!] \subseteq [\![\texttt{k}]\!] \ \wedge \ [\![\texttt{k}]\!] \subseteq [\![\texttt{(f)(n)}]\!] \\
&[\![\texttt{input}]\!] \subseteq [\![\texttt{x}]\!] \\
&[\![\texttt{foo(x,inc)}]\!] \subseteq [\![\texttt{y}]\!] \\
&[\![\texttt{foo(x,dec)}]\!] \subseteq [\![\texttt{y}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \ \wedge \ [\![\texttt{inc}]\!] \subseteq [\![\texttt{f}]\!] \ \wedge \ [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,inc)}]\!] \\
&\texttt{foo} \in [\![\texttt{foo}]\!] \Rightarrow [\![\texttt{x}]\!] \subseteq [\![\texttt{n}]\!] \ \wedge \ [\![\texttt{dec}]\!] \subseteq [\![\texttt{f}]\!] \ \wedge \ [\![\texttt{r}]\!] \subseteq [\![\texttt{foo(x,dec)}]\!]
\end{aligned}
$$

最小解的非空值为：

$$
\begin{aligned}
&[\![\texttt{inc}]\!] = \{\texttt{inc}\} \\
&[\![\texttt{dec}]\!] = \{\texttt{dec}\} \\
&[\![\texttt{ide}]\!] = \{\texttt{ide}\} \\
&[\![\texttt{f}]\!] = \{\texttt{inc}, \texttt{dec}, \texttt{ide}\} \\
&[\![\texttt{foo}]\!] = \{\texttt{foo}\}
\end{aligned}
$$

基于此，我们可以构建以下程序的单变量跨过程控制流图：



然后可以作为后续过程间静态分析的基础。

8.4 面向对象语言中的控制流

具有函数指针或高阶函数的语言必须使用前面描述的控制流分析类型来获得合理精确的CFG。对于常见的面向对象语言，如Java或C#，它也很有用，但类层次结构和类型系统提供的附加结构允许一些更简单的替代方案。 在面向对象的环境中，问题是在给定的方法调用点上可能执行哪些方法实现：

```
x.m(a,b,c)
```

最简单的解决方案是扫描类库并选择任何名为m的方法，其签名接受实际参数的类型。 更好的选择，称为类层次分析（*CHA*），是仅考虑由x的声明类型跨越的类层次结构的部分。 进一步的改进，称为

快速类型分析（*RTA*）是进一步限制为实际分配对象的类。另一种技术，称为变量类型分析（*VTA*），在进行保守假设的同时进行程序内部的控制流分析。

这些技术当然比完全的控制流分析快得多，并且对于实际程序来说，它们通常足够精确。

第9章

指针分析

TIP语言的最终扩展引入了简单指针和动态内存。 由于我们的玩具版本的malloc只分配一个单元，所以我们无法在堆中构建任意结构。 然而，指针的主要问题在我们考虑的语言片段中得到了充分的体现。

9.1 指向分析

必须获取的最重要信息是指针可能指向的可能单元的集合。在执行过程中，可能有任意多个可能的位置，因此我们必须选择一些有限的代表。 规范选择是为每个命名为id的变量引入一个抽象单元id和一个抽象单元malloc-i，其中i是一个唯一的索引，用于程序中每次malloc操作的出现。 我们使用*C*ell来表示给定程序的这些单元的集合，并使用*Loc*来表示单元的抽象位置的集合，对于每个*c*∈*C*ell，写作& *c*∈*Loc*。

我们将要学习的第一个指向分析是流不敏感的。 这种分析的最终结果是一个函数 pt，对于每个指针变量 p返回它可能指向的单元的集合 $pt(p)$。当然，我们必须进行保守分析，所以这些集合通常会很大。

有了这些信息，可以近似得出许多其他事实。 如果我们想知道指针变量 p和 q是否可能是别名，那么通过检查 $pt(p) \cap pt(q)$是否非空，可以得到一个安全的答案。

一个几乎微不足道的分析，称为*address taken*，是使用所有可能的单元，除了如果给定程序中出现表达式 & *id*，则只包括*id*。
这只适用于非常简单的应用程序，所以通常更倾向于采用更有雄心的方法。 如果我们限制自己只对可类型化的程序进行分析，那么任何指向分析都可以通过删除那些类型与指针变量不相等的位置来改进。

9.2 安德森算法

指向分析的一种方法与控制流分析非常相似。 对于每个单元格 c我们引入一个约束变量 $[[c]]$范围在位置集合上。

分析假设程序已经被规范化，以便每个指针操作都属于以下六种类型之一：

1) id = malloc
2) id_1 = &id_2
3) id_1 = id_2
4) id_1 = * id_2
5) *id_1 = id_2
6) id = null

> 练习 **9.1**：展示如何通过引入新的临时变量来系统地执行此规范化。

对于每个指针操作，我们生成约束：

$$
\begin{array}{rl}
id \texttt{ = malloc}: & \texttt{\&malloc-i} \in [[id]] \\
id_1 \texttt{ = \&}id_2\texttt{:} & \texttt{\&标识符}_2 \in [[\texttt{标识符}_1]] \\
\texttt{]标识符}_1 \texttt{ = 标识符}_2 & [[\texttt{标识符}_2]] \subseteq [[\texttt{标识符}_1 \\
\texttt{]标识符}_1 \texttt{ = *标识符}_2\texttt{:} & \texttt{\&}\alpha \in [[\texttt{标识符}_2]] \Rightarrow [[\alpha]] \subseteq [[\texttt{标识符}_1]] \\
\texttt{*标识符}_1 \texttt{ = 标识符}_2\texttt{:} & \texttt{\&}\alpha \in [[\texttt{标识符}_1]] \Rightarrow [[\texttt{标识符}_2]] \subseteq [[\alpha]]
\end{array}
$$

对于每个位置 $\&\alpha \in Loc$，生成了最后两个约束条件，但实际上我们只需要考虑那些在给定程序中实际上被引用的地址。 忽略了空赋值，因为它对应于平凡的约束条件$\emptyset \subseteq$ [[标识符]]。 由于这些约束条件符合立方算法的要求，可以在时间 $O(\mathsf{N}^3)$内解决。 得到的指向函数定义如下：

$$pt(p) = \{\alpha \in Cell \mid \&\alpha \in [[p]]\}$$

考虑以下示例程序：

```
var p,q,x,y,z;
p = malloc;
x = y;
x = z;
*p = z;
p = q;
q = &y;
x = *p;
p = &z;
```

Andersen算法生成这些约束条件：

$\&\texttt{malloc-1} \in [\![\texttt{p}]\!]$
$[\![\texttt{y}]\!] \subseteq [\![\texttt{x}]\!]$
$[\![\texttt{z}]\!] \subseteq [\![\texttt{x}]\!]$
$\&\alpha \in [\![\texttt{p}]\!] \Rightarrow [\![\texttt{z}]\!] \subseteq [\![\alpha]\!]$
$[\![\texttt{q}]\!] \subseteq [\![\texttt{p}]\!]$
$\&\texttt{y} \in [\![\texttt{q}]\!]$
$\&\alpha \in [\![\texttt{p}]\!] \Rightarrow [\![\alpha]\!] \subseteq [\![\texttt{x}]\!]$
$\&\texttt{z} \in [\![\texttt{p}]\!]$

最小解中的非空值为：

$pt(\texttt{p}) = \{\texttt{malloc-1}, \texttt{y}, \texttt{z}\}$
$pt(\texttt{q}) = \{\texttt{y}\}$

这给出了一个相当精确的结果。 请注意，虽然这个算法是流不敏感的，但约束的方向性意味着数据流仍然以一定的准确性建模。

9.3 斯廷斯加德算法

一种流行的替代算法是Steensgaard算法，它通过将赋值视为双向来执行更粗粒度的分析。该分析可以通过术语统一性来优雅地表示。 我们为每个单元格c使用一个术语变量$[\![c]\!]$，并使用术语构造器$\&t$表示指向t的指针。 （与第9.2节相比，注意符号的变化：这里，$[\![c]\!]$是一个术语变量，并且不直接表示一组位置。）

$$
\begin{array}{rl}
id = \texttt{malloc}: & [\![id]\!] = \&[\![\texttt{malloc-i}]\!] \\
id_1 = \&id_2: & [\![id_1]\!] = \&[\![id_2]\!] \\
id_1 = id_2: & [\![id_1]\!] = [\![id_2]\!] \\
id_1 = *id_2: & [\![id_2]\!] = \&\alpha \ \wedge \ [\![id_1]\!] = \alpha \\
*id_1 = id_2: & [\![id_1]\!] = \&\alpha \ \wedge \ [\![id_2]\!] = \alpha
\end{array}
$$

每个α表示一个新的项变量。

通常情况下，项构造函数满足一般的项相等公理：

$$\&\alpha_1 = \&\alpha_2 \ \Rightarrow \ \alpha_1 = \alpha_2$$

得到的指向函数定义如下：

$$pt(p) = \{t \in Cell \mid [\![p]\!] = \&[\![t]\!]\}$$

对于之前的示例程序，Steensgaard算法生成以下约束条件:

$[\![\texttt{p}]\!] = \&[\![\texttt{malloc-1}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{y}]\!]$
$[\![\texttt{x}]\!] = [\![\texttt{z}]\!]$

$[\![\mathtt{p}]\!] = \&\alpha_1 \qquad [\![\mathtt{z}]\!] = \alpha_1$
$[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$
$[\![\mathtt{q}]\!] = \&[\![\mathtt{y}]\!]$
$[\![\mathtt{x}]\!] = \alpha_2 \qquad [\![\mathtt{p}]\!] = \&\alpha_2$
$[\![\mathtt{p}]\!] = \&[\![\mathtt{z}]\!]$

这反过来意味着:

$$pt(\mathtt{p}) = pt(\mathtt{q}) = \{\mathtt{malloc\text{-}1}, \mathtt{y}, \mathtt{z}\}$$

这比Andersen算法更不精确，但使用更快的算法。

9.4 过程间指向分析

如果函数指针与其他指针有所区别，那么我们可以通过首先计算一个函数间CFG，然后运行Andersen算法或Steensgaard算法来执行函数间指向分析，如前面所述。 然而，如果函数指针也可能有间接引用，那么我们需要同时执行控制流分析和指向分析，以解决例如函数调用:

```
(***x)(1,2,3);
```

为了表达组合算法，我们进行了语法简化，所有函数调用的形式如下：

id_1 = (*id*$_2$)($a_1, \ldots, a_n$);

其中 id_i和 a_i是变量。 同样，所有返回表达式都假设只是变量。

> 练习 **9.2**：展示如何以系统化的方式进行这些简化。

Andersen算法已经类似于控制流分析，只需通过适当的约束进行扩展。 对常量函数 *f*的引用生成约束条件：

$$f \in [\![f]\!]$$

计算得到的函数调用生成约束条件

$$f \in [\![id_2]\!] \Rightarrow \left([\![a_1]\!] \subseteq [\![x_1]\!] \wedge \ \ldots \ \wedge \ [\![a_n]\!] \subseteq [\![x_n]\!] \wedge \ [\![id]\!] \subseteq [\![id_1]\!]\right)$$

对于每个函数定义的出现

f($x_1, \ldots, x_n$) { ...返回 *id*; }

这将保持控制流分析的精度。

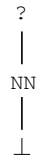
9.5 示例：空指针分析

我们现在还能够定义一个检测null解引用的分析。 具体来说，我们希望确保只有在*id*不是 null时才执行 * *id*。 让我们考虑函数内部分析，这样我们可以忽略函数调用。

与之前一样，我们假设程序已经规范化，所以所有指针操作都是这些类型的：

1) id = malloc
2) id_1 = &id_2
3) id_1 = id_2
4) id_1 = *id_2
5) *id_1 = id_2
6) id = null

我们使用的基本格子称为 $Null$。



其中 NN表示绝对不是 null，而 ?表示可能是 null的值。然后我们形成以下映射格子：Cell→Null

对于每个CFG节点 n，我们引入一个约束变量 $[\![n]\!]$ 以表示映射格子中的一个元素。

我们将使用每个约束变量来描述节点之后的程序点的抽象状态。 我们将使用每个约束变量来描述节点之后的程序点的抽象状态。

对于不涉及指针操作的所有节点，我们有以下约束条件：

$$[\![v]\!] = JOIN(v)$$

其中

$$JOIN(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

对于堆加载操作 id_1 = *id_2，我们需要对程序变量 id_1的变化进行建模。 我们的抽象有一个单独的抽象单元用于 id_1。 在假设进行函数内部分析的情况下，该抽象单元表示一个具体单元。 （在进行函数间分析时，我们需要考虑每个运行时的堆栈帧都有该变量的实例。）对于表达式 *id_2，我们可以向指向分析询问可能的单元 $pt(id_2)$。

基于这些观察，我们可以给出堆加载操作的约束条件：

$$id_1 \text{ = * } id_2\text{:} \quad [\![v]\!] = load(JOIN(v), id_1, id_2)$$

其中

$$load(\sigma, id_1, id_2) = \sigma[id_1 \rightarrow \bigsqcup_{\alpha \in pt(id_2)} \sigma(\alpha)]$$

类似的推理给出了对指针变量的其他操作的约束条件:

$$\begin{array}{rl} id \texttt{ = malloc}: & [\![v]\!] = JOIN(v)[id \rightarrow \texttt{NN}, \texttt{malloc-i} \rightarrow \texttt{?}] \\ id_1 \texttt{ = \&}id_2\texttt{:} & [\![v]\!] = JOIN(v)[id_1 \rightarrow \texttt{NN}] \\ id_1 \texttt{ = } id_2\texttt{:} & [\![v]\!] = JOIN(v)[id_1 \rightarrow JOIN(v)(id_2)] \\ id \texttt{ = null}: & [\![v]\!] = JOIN(v)[id \rightarrow \texttt{?}] \end{array}$$

练习**9.3**: 解释为什么上述四个约束是单调的和正确的.

对于堆存储操作 $*id_1 = id_2$，我们需要对 id_1指向的内容的变化进行建模。这可能是多个抽象单元，即t$pt(id_1)$。 此外，每个抽象堆单元 `malloc-i`可能描述多个具体单元。 在堆存储操作的约束条件中，我们必须将新的抽象值与 $pt(id_1)$中的每个受影响单元的现有值进行合并:

$$*\text{标识符}_1 \texttt{ = } \text{标识符}_2\texttt{:} \quad [\![\text{变量}]\!] = \text{存储}(\text{连接}(\text{变量}), \text{标识符}_1, \text{标识符}_2)$$

其中

$$\text{存储}(\sigma, \text{标识符}_1, \text{标识符}_2) = \sigma \quad [\alpha \rightarrow \sigma(\alpha) \sqcup \sigma(\text{标识符}_2)\ ]$$
$$\alpha \in pt(\text{标识符}_1)$$

在堆存储操作中，我们模拟将新的抽象值合并到现有值中的情况被称为弱更新。 相比之下，在强更新中，新的抽象值覆盖了现有值，在修改程序变量的所有操作中都可以看到这一点。 强更新通常比弱更新更精确，但可能需要更复杂的分析抽象来检测可以安全应用强更新的情况。

在对给定程序进行空指针分析之后，如果在程序点 v处进行指针解引用 $*id$，则可以保证安全性

$$JOIN(v)(id) = \texttt{NN}$$

这种分析的精度当然取决于底层指向分析的质量。

考虑以下有错误的示例程序：

```
p = malloc;
q = &p;
n = null;
*q = n;
*p = n;
```

Andersen算法计算出以下指向集合：

$$pt(\mathtt{p}) = \{\mathtt{malloc\text{-}1}\}$$
$$pt(\mathtt{q}) = \{\mathtt{p}\}$$
$$pt(\mathtt{n}) = \emptyset$$

基于这些信息，空指针分析生成以下约束条件：

$$[[\mathtt{p{=}malloc}]] = \bot[\mathtt{p} \to \mathtt{NN}, \mathtt{malloc\text{-}1} \to ?]$$
$$[[\mathtt{q{=}\&p}]] = [[\mathtt{p{=}malloc}]][\mathtt{q} \to \mathtt{NN}]$$
$$[[\mathtt{n{=}null}]] = [[\mathtt{q{=}\&p}]][\mathtt{n} \to ?]$$
$$[[\mathtt{*q{=}n}]] = [[\mathtt{n{=}null}]][\mathtt{p} \to [[\mathtt{n{=}null}]](\mathtt{p}) \sqcup [[\mathtt{n{=}null}]](\mathtt{n})]$$
$$[[\mathtt{*p{=}n}]] = [[\mathtt{*q{=}n}]][\mathtt{malloc\text{-}1} \to [[\mathtt{*q{=}n}]](\mathtt{malloc\text{-}1}) \sqcup [[\mathtt{*q{=}n}]](\mathtt{n})]$$

其中最小解为：

$$[[\mathtt{p{=}malloc}]] = [\mathtt{p} \to \mathtt{NN}, \mathtt{q} \to \bot, \mathtt{n} \to \bot, \mathtt{malloc\text{-}1} \to ?]$$
$$[[\mathtt{q{=}\&p}]] = [\mathtt{p} \to \mathtt{NN}, \mathtt{q} \to \mathtt{NN}, \mathtt{n} \to \bot, \mathtt{malloc\text{-}1} \to ?]$$
$$[[\mathtt{n{=}null}]] = [\mathtt{p} \to \mathtt{NN}, \mathtt{q} \to \mathtt{NN}, \mathtt{n} \to ?, \mathtt{malloc\text{-}1} \to ?]$$
$$[[\mathtt{*q{=}n}]] = [\mathtt{p} \to ?, \mathtt{q} \to \mathtt{NN}, \mathtt{n} \to ?, \mathtt{malloc\text{-}1} \to ?]$$
$$[[\mathtt{*p{=}n}]] = [\mathtt{p} \to ?, \mathtt{q} \to \mathtt{NN}, \mathtt{n} \to ?, \mathtt{malloc\text{-}1} \to ?]$$

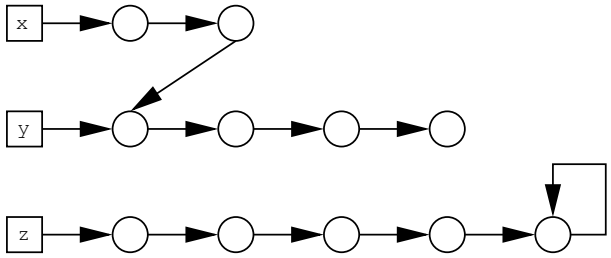
通过检查这些信息，分析可以静态地检测到当*q=n被评估时，即在n=null之后立即评估，变量q肯定是非空的。另一方面，当评估 *p=n时，我们不能排除可能性，即 p可能包含null。

练习**9.4**：使用弱更新展示堆加载操作的替代约束条件，以及使用修改后的分析方法给出比上述分析结果更不精确的示例程序。

练习**9.5**：使用强更新展示堆存储操作的（不安全的）替代约束条件，以及使用修改后的分析方法给出错误结果的示例程序。

9.6 示例：形状分析

到目前为止，我们将堆视为一个无定形的结构，并且只回答关于基于堆栈的变量的问题。可以使用形状分析对堆进行更详细的分析。请注意，即使 malloc操作只分配一个堆单元，我们仍然可以生成有趣的堆。一个非平凡堆的例子是：



其中 x、 y和 z是程序变量。 我们将寻求回答关于程序变量中包含的结构的不相交性的问题。 在上面的例子中， x和 y不是不相交的，而 y和 z是不相交的。

形状分析需要一个更有雄心的形状图的格子，这些图是有向图，其中节点是给定程序的抽象单元，边对应可能的指针。 形状图按照它们的边集的包含关系进行排序。 因此， $\bot$是没有边的图， $\top$是完全连接的图。 形式上，我们的格子是

$$2^{\text{单元} \times \text{单元}}$$

按照通常的子集包含进行排序。 对于每个CFG节点 v，我们引入一个约束变量 $[[v]]$，表示描述该程序点之后所有可能存储的形状图。 对于对应于各种指针操作的节点，我们有以下约束：

$$\begin{array}{rll}
id \texttt{ = malloc}: & [[v]] = JOIN(v)\downarrow id \cup \{(id, \texttt{malloc-i})\} \\
id_1 \texttt{ = \&}id_2\texttt{:} & [[v]] = JOIN(v)\downarrow id_1 \cup \{(id_1, id_2)\} \\
id_1 \texttt{ = } id_2\texttt{:} & [[v]] = assign(JOIN(v), id_1, id_2) \\
id_1 \texttt{ = *}id_2\texttt{:} & [[v]] = load(JOIN(v), id_1, id_2) \\
\texttt{*}id_1 \texttt{ = } id_2\texttt{:} & [[v]] = store(JOIN(v), id_1, id_2) \\
id \texttt{ = null}: & [[v]] = JOIN(v)\downarrow id
\end{array}$$

对于其他所有节点，约束为：

$$[[v]] = JOIN(v)$$

其中

$$JOIN(v) = \bigcup_{w \in pred(v)} [[w]]$$

$$\sigma\downarrow x = \{(s, t) \in \sigma \mid s = x\}$$

$$assign(\sigma, x, y) = \sigma\downarrow x \cup \{(x, t) \mid (y, t) \in \sigma\}$$

$$load(\sigma, x, y) = \sigma\downarrow x \cup \{(x, t) \mid (y, s) \in \sigma, (s, t) \in \sigma\}$$

$$store(\sigma, x, y) = \sigma \cup \{(s, t) \mid (x, s) \in \sigma, (y, t) \in \sigma\}$$

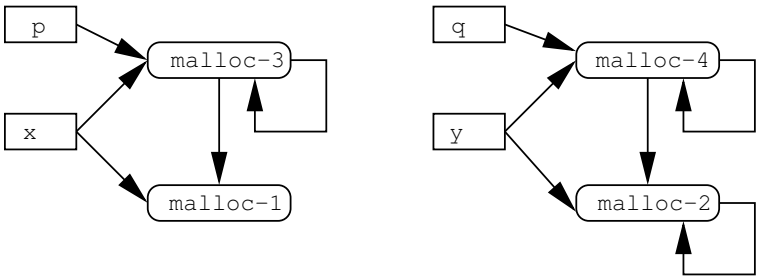
注意堆存储操作的约束使用弱更新。

练习**9.6**：解释上述约束。

现在考虑以下程序：

```
var x,y,n,p,q;
x = malloc; y = malloc;
*x = null; *y = y;
n = input;
while (n>0) {
  p = malloc; q = malloc;
  *p = x; *q = y;
  x = p; y = q;
  n = n-1;
}
```

循环结束后，分析产生以下形状图：



从这个结果我们可以安全地得出结论， x和 y将始终不相交。

请注意，我们的形状分析还计算了一个流敏感的指向映射对于每个程序点 v由以下方式定义：

$$pt(p) = \{t \mid (p, t) \in [\![v]\!]\}$$

这种分析比Andersen算法更精确，但显然也更昂贵。 以一个例子来说明，考虑以下程序：

```
x = &y;
x = &z;
```

在这些语句之后，Andersen算法会预测 $pt(\mathtt{x}) = \{\mathtt{y}, \mathtt{z}\}$而形状分析计算 $pt(\mathtt{x}) = \{\mathtt{z}\}$对于最终程序点。这种流敏感的指针信息可以用来提升 null pointer分析。 然而，仍然需要进行初始的流不敏感的指针分析来构建使用函数指针的程序的CFG。 相反地，如果我们有另一个指针分析，那么它可以通过限制 $load$和 $store$函数中考虑的位置来提升形状分析的精度。 或者，我们可以通过适当地增加格子来在数据流分析期间执行即时的指针和控制流分析。

9.7 示例：逃逸分析

我们之前对程序显示的逃逸堆栈单元错误感到遗憾：

```
baz() {
  var x;
  return &x;
}

main() {
  var p;
  p=baz(); *p=1;
  return *p;
}
```

这超出了类型系统的范围。 在执行简单的形状分析之后，我们可以轻松地进行逃逸分析以捕捉此类错误。
我们只需要检查形状图中返回表达式的可能单元是否无法访问函数本身中定义的参数或变量，因为所有其他位置必须位于调用栈上的较早帧中。