第12章　过程间分析

在这一章中，我们讨论一些不能使用过程内分析技术解决的优化问题，由此引出了过程间分析的重要性。我们将首先描述过程间分析的常见形式，并解释实现它们的难点。然后将描述过程间分析的应用。对于诸如C和Java这样广泛使用的程序设计语言，指针别名分析是所有过程间分析技术的关键之处。因此本章将用大量篇幅讨论获取程序中的指针别名信息所需要的技术。我们先给出Datalog的描述，这种表示方法极大地隐藏了一个高效指针分析技术的复杂性。然后我们描述一个用于指针别名分析的算法，并说明如何使用二分决策图(Binary Decision Diagram，BDD)来高效地实现这个算法。

大部分编译器优化技术，包括那些在第9、10、11章中描述的技术，都是每次在一个过程中执行的。我们把这样的分析称为过程内分析。这些分析保守地假设被调用的过程有可能改变过程可见的所有变量的状态，并且它们还可能产生某种副作用，比如改变此过程可见的任何变量的值，或产生导致调用栈释放的异常。因此，过程内分析虽然不精确，但是却相对简单。有些优化不需要过程间分析，而有些优化不借助过程间分析几乎不会产生有用的信息。

一个过程间分析处理的是整个程序，它将信息从调用者传送到被调用者，或者反向传送。一个相对简单但有用的技术是过程内联(inline)，就是把一个过程调用替换为被调用过程的过程体。在替换时需要考虑参数传递和返回值，因此需要进行适当修改。只有当我们知道这个过程调用的目标后才可以应用这个方法。

如果过程是通过一个指针或面向对象编程中常见的过程分发机制间接调用的，那么对程序指针或引用的分析有时可以确定这个间接调用的目标。如果目标是唯一的，那么就可以应用过程内联方法。

即使确定了每个过程调用只有一个调用目标，仍然必须谨慎使用内联转换。一般来说，不可能直接内联递归的过程，并且即使没有递归，内联转换也可能指数级地增加代码的大小。

12.1　基本概念

在本节中，我们将介绍调用图，就是告诉我们哪个过程调用了哪个过程的图。我们也会介绍“上下文相关”的思想，即进行数据流分析时需要认识到过程调用的序列是什么。也就是说，当上下文相关分析在区分程序中的不同“位置”时，它不仅考虑当前的程序点，还考虑当前栈中的活动记录的序列(或其大纲)。

12.1.1　调用图

一个程序的调用图(call graph)是一个结点和边的集合，并满足

1) 对程序中的每个过程都有一个结点。

2) 对于每个调用点(call site)都有一个结点。所谓调用点就是程序中调用某个过程的一个位置。

3) 如果调用点 c 调用了过程 p，就存在一条从 c 的结点到 p 的结点的边。

很多用诸如C或Fortran语言编写的程序直接进行过程调用，因此每个调用的调用目标可以静态地确定。在这种情况下，调用图中的每个调用点都恰好有一条边指向一个过程。但是，如果程序使用了过程参数或函数指针，一般来说，需要到程序运行时刻才能知道调用目标，而且实际

上可能各次调用的目标都有所不同。那么，一个调用点可能连接到调用图中的多个甚至所有的过程。

对于面向对象程序设计语言来说，间接调用是标准的调用方式。特别地，当存在子类对方法进行重载的情况时，对方法 m 的使用可能指向多个不同方法中的任意一个，这要取决于该调用所作用的接收对象的子类。使用这样的虚(virtual)方法调用意味着我们需要知道接收者的类型之后才可以确定调用了哪个方法。

例 12.1 图 12-1 显示了一个 C 程序。该程序声明 pf 是一个指向类型为"整数到整数"的函数的全局指针。有两个函数 fun1 和 fun2 是这个类型。此外，main 函数不是 pf 所指向的类型。图中显示了三个调用点，标记为 c1、c2 和 c3，这些标号不是程序的一部分。

```
        int (*pf)(int);

        int fun1(int x) {
            if (x < 10)
c1:             return (*pf)(x+1);
            else
                return x;
        }

        int fun2(int y) {
            pf = &fun1;
c2:         return (*pf)(y);
        }

        void main() {
            pf = &fun2;
c3:         (*pf)(5);
        }
```

图 12-1 一个具有函数指针的程序

最简单的对 pf 可能指向哪个函数的分析只查看函数的类型。函数 fun1 及 fun2 和 pf 所指向的对象具有相同的类型，而 main 则不同。因此，一个保守的调用图如图 12-2a 所示。对这个程序进行更深入的分析，就可以观察到 pf 在 main 中指向 fun2，而在 fun2 中指向 fun1。但是没有其他的对任何指针的赋值，因此 pf 不可能指向 main 函数。这个推理过程产生的调用图和图 12-2a 中的相同。

一个更加精确的分析将指出 pf 在 c3 上只可能指向 fun2，因为紧靠这个调用之前的赋值语句将 fun2 赋给 pf。类似地，pf 在 c2 处只可能指向 fun1。分析的结果是，对 fun1 的第一次调用必然是 fun2 做出的，且 fun1 不会改变 pf 的值，因此当我们在 fun1 中时，pf 就指向 fun1。特别地，我们可以确信 pf 在 c1 处指向 fun1。因此，图 12-2b 是一个更加精确、正确的调用图。 □

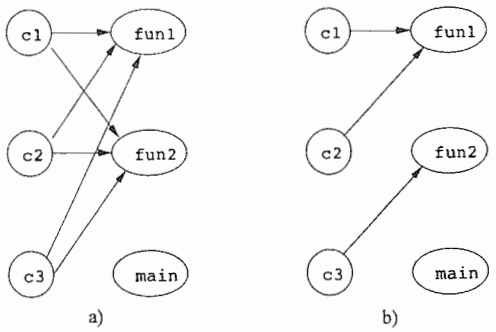


图 12-2 从图 12-1 得到的调用图

一般来说，当出现了对函数或方法的引用或指针时，要求我们对所有过程参数、指针、接收对象类型等的可能取值进行静态估计。要得到一个精确的估计值就必须进行过程间分析。这个分析从可以静态观察到的目标开始，迭代地进行。当发现一个新的调用目标时，分析过程就会把一条新边加入到调用图中，并不断寻找更多的目标，直到收敛。

12.1.2 上下文相关

过程间分析很具有挑战性，因为各个过程的行为和它被调用时所在的上下文相关。例 12.2 通过一个小程序上的过程间常量传播问题说明了上下文的重要性。

例 12.2 考虑图 12-3 中的程序片段。函数 f 在三个调用点 c1、c2 和 c3 上被调用。在循环的每次迭代中，常量 0 在 c1 上被作为实在参数传递，而常量 243 在 c2 和 c3 上被传递，这些调用分别返回常量 1 和 244。因此，在各个上下文中函数 f 的实在参数都是常量，但是常量的具体值要根据上下文而定。

如我们将看到的，除非我们能够知道在上下文 c1 中调用 f 时返回 1，且在其他两个上下文中调用 f 时返回 244，否则不可能指出 t1、t2 和 t3 都被赋予了一个常量值（因此 $X[i]$ 也被赋予了常量值）。通过一个简单的分析就可以知道对 f 的各次调用的返回值可能是 1 或 244。 □

```
        for (i = 0; i < n; i++) {
c1:             t1 = f(0);
c2:             t2 = f(243);
c3:             t3 = f(243);
                X[i] = t1+t2+t3;
        }

        int f (int v) {
                return (v+1);
        }
```

图 12-3 用来说明上下文相关分析的需求的一个程序片断

一种非常简单但是极端不精确的过程间分析方法称为上下文无关分析（context-insensitive analysis）。它把每个调用和返回语句看作一个“goto”操作。我们创建一个超级控制流图。图中除了一般的过程内控制流边外还有一些附加的边。这些边包括

1）从每个调用点到它所调用的过程的开始处的边。

2）从返回语句回到调用点的边[⊖]。

另外还增加了一些赋值语句，它们把实在参数赋给相应的形式参数，并把返回值赋给接收返回结果的变量。然后，我们就可以对这个超级控制流图应用那些为分析单个过程而设计的标准分析技术，找出上下文无关的过程间分析结果。这个模型虽然简单，但它抽象掉了过程调用中输入值和输出值之间的重要关系，使得分析结果不够精确。

例 12.3 图 12-3 中的程序的超级控制流图显示在图 12-4 中。块 B_6 就是函数 f。块 B_3 包含了调用点 c1，它把形式参数 v 设置为 0，然后跳转到 f 的开始处。类似地，B_4 和 B_5 分别表示调用点 c2 和 c3。B_4 可以从 f（基本块 B_6）的结尾处到达。我们在 B_4 中把 f 的返回值赋给 t1。然后把形式参数 v 设置成 243 并通过跳转到 B_6 再次调用 f。请注意，没有从 B_3 到达 B_4 的边。在从 B_3 到达 B_4 的路上，控制流必须穿越 f。

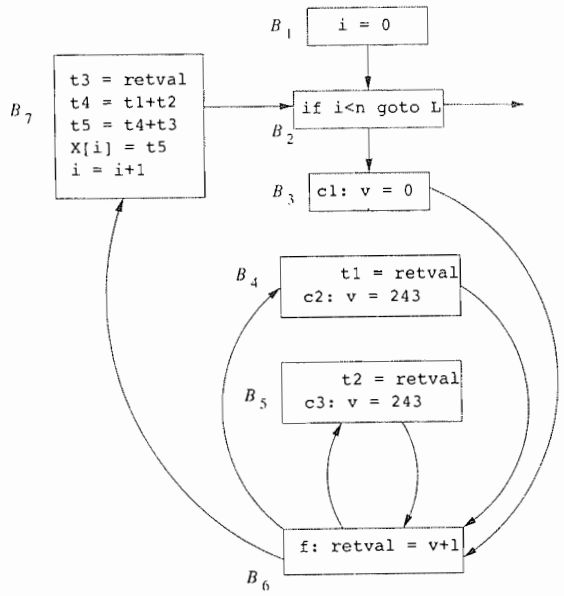


图 12-4 图 12-3 的控制流图，它把函数调用当作控制流处理

⊖ 实际上是从返回语句到跟在调用点之后的指令的边。

B_5 和 B_4 类似。它接收来自 f 的返回值并把返回值赋给 t2，并初始化对 f 的第三次调用。块 B_7 表示了第三次调用的返回和对 $X[i]$ 的赋值。

如果我们把图 12-4 当作单个过程的流图，那么可以断定当控制流进入 B_6 时，v 的值可以是 0 或 243。因此，我们最多能够断定 retval 的值是 1 或 244，而不会是其他值。类似地，关于 t1、t2 和 t3，我们只能断定它们的值是 1 或 244。因此，看起来 $X[i]$ 的值为 3、246、489 或 732 之一。反过来，一个上下文相关分析可以区分每次调用上下文的结果，并产生例 12.2 中描述的直觉结果：t1 总是 1，t2 和 t3 的值总是 244，而 $X[i]$ 的值为 489。 □

12.1.3 调用串

在例 12.2 中，我们只需要知道调用过程 f 的调用点就可以区分不同的上下文。一般情况下，一个调用上下文是通过整个调用栈中的内容来定义的。我们把栈中各个调用点组成的串称为调用串(call string)。

例 12.4 图 12-5 是对图 12-3 进行细微的修改后得到的。这里我们把对 f 的调用替换成对 g 的调用。函数 g 随后用同样的参数调用 f。函数 g 调用 f 的地点是 c4，这是一个新增的调用点。

有三个对应于 f 的调用串：(c1, c4)、(c2, c4)和(c3, c4)。如我们在这个例子中见到的，函数 f 中 v 的值并不由调用串中的直接(或者说最后)调用点 c4 决定。这些常量值实际上是由每个调用串中的第一个元素决定的。 □

例 12.4 表明，与分析相关的信息可能在调用链的早期就被引入。事实上，如例 12.5 所示，为了得到最精确的答案，有时甚至需要考虑计算整个调用串。

例 12.5 这个例子说明了对不限长度的调用串的分析能力是如何产生出更加精确的结果的。在图 12-6 中，我们看到如果用一个正数值 c 来调用 g，g 将会被递归地调用 c 次。每次 g 被调用的时候，它的参数 v 的值减一。因此，在调用串为 $c2(c4)^n$ 的上下文中，g 的参数 v 的值是 $243-n$。因此，g 的功能就是把 0 或任何负参数加一，并对任何大于等于 1 的参数返回 2。

```
        for (i = 0; i < n; i++) {
c1:         t1 = g(0);
c2:         t2 = g(243);
c3:         t3 = g(243);
            X[i] = t1+t2+t3;
        }

        int g (int v) {
c4:         return f(v);
        }

        int f (int v) {
            return (v+1);
        }
```

图 12-5 演示调用串的程序片段

```
        for (i = 0; i < n; i++) {
c1:         t1 = g(0);
c2:         t2 = g(243);
c3:         t3 = g(243);
            X[i] =  t1+t2+t3;
        }

        int g (int v) {
            if (v > 1) {
c4:             return g(v-1);
            } else {
c5:             return f(v);
        }

        int f (int v) {
            return (v+1);
        }
```

图 12-6 需要分析整个调用串的递归程序

函数 f 有三个可能的调用串。如果我们从 c1 处的调用开始，那么 g 可以立刻调用 f，因此(c1, c5)就是这样的一个串。如果我们从 c2 或 c3 开始，那么我们共调用 g 243 次，然后再调用 f。这些调用串是(c2, c4, c4, …, c5)和(c3, c4, c4, …, c5)，在这两种情况下的序列中都有 242 个 c4。在这些上下文中，在第一个上下文中 f 的参数 v 的值是 0，而在另外两个中的参数值为 1。 □

在设计一个上下文相关分析的时候，我们可以选择不同的精确度。比如，我们可以选择只使用调用串中最直接的 k 个调用点来区分上下文，而不是使用整个调用串来提高分析结果的质量。这个技术被称为 k-界限上下文分析技术。上下文无关分析就是 k - 界限上下文分析技术在 $k=0$ 时的特例。我们可以使用 1-界限分析技术找出例 12.2 中的所有常量，用 2-界限分析技术找出例 12.4 中的所有常量。但是，只要例 12.5 中的常量 243 被替换成为不同的任意大小的常量值，没有任何 k-界限分析可以找出该例中的所有常量。

如果不选定一个固定的 k 值，另一种可行方法是对所有无环调用串进行完全的上下文相关分析。所谓无环调用串就是不包含递归环的调用串。对于所有带有递归的调用串，我们可以把所有的递归环都塌缩成一个点，以便限定需要分析的不同上下文的数目。在例 12.5 中，从调用点 c2 开始的调用可以用调用串(c2，c4 *，c5)近似地表示。请注意，使用这种方案时，即使对于不带递归的程序，不同调用上下文的数目和程序中的过程数目呈指数关系。

12.1.4 基于克隆的上下文相关分析

上下文相关分析的另一个方法是在概念上克隆被调用过程，对于每个感兴趣的上下文都进行一次克隆。然后我们就可以对克隆过的调用图应用上下文无关分析。例 12.6 和 12.7 分别给出了和例 12.4 和 12.5 等价的克隆版本。在实际分析时，我们不需要真的克隆代码，而是可以直接使用一个高效的内部表示来跟踪各个克隆部分的分析结果。

例 12.6 图 12-5 的克隆版本显示在图 12-7 中。因为每个调用上下文指向一个不同的克隆，因此不存在混淆的情况。比如，g1 的输入为 0，产生输出 1；g2 和 g3 接受输入 243 并产生输出 244。 □

```
        for (i = 0; i < n; i++) {
c1:         t1 = g1(0);
c2:         t2 = g2(243);
c3:         t3 = g3(243);
            X[i] =  t1+t2+t3;
        }
      int g1 (int v) {
c4.1:       return f1(v);
      }
      int g2 (int v) {
c4.2:       return f2(v);
      }
      int g3 (int v) {
c4.3:       return f3(v);
      }

      int f1 (int v) {
          return (v+1);
      }
      int f2 (int v) {
          return (v+1);
      }
      int f3 (int v) {
          return (v+1);
      }
```

图 12-7 图 12-5 的克隆版本

例 12.7 例 12.5 的克隆版本显示在图 12-8 中。我们创建了过程 g 的一个克隆版本，它代表所有首先在 c1、c2 和 c3 处调用的 g 的实例。在这种情况下，如果一个分析技术能够从 $v=0$ 推导

出 $v>1$ 不成立，那么该分析将会确定在调用点 c1 处的调用将会返回 1。但是，这个分析不能很好地处理递归，不能分析得到调用点 c2 和 c3 处的常量值。 □

```
        for (i = 0; i < n; i++) {
c1:           t1 = g1(0);
c2:           t2 = g2(243);
c3:           t3 = g3(243);
              X[i] =  t1+t2+t3;
        }

       int g1 (int v) {
           if (v > 1) {
c4.1:           return g1(v-1);
           } else {
c5.1:           return f1(v);}
       }

       int g2 (int v) {
           if (v > 1) {
c4.2:           return g2(v-1);
           } else {
c5.2:           return f2(v);}
       }

       int g3 (int v) {
           if (v > 1) {
c4.3:           return g3(v-1);
           } else {
c5.3:           return f3(v);}
       }

       int f1 (int v) {
           return (v+1);
       }
       int f2 (int v) {
           return (v+1);
       }
       int f3 (int v) {
           return (v+1);
       }
```

图 12-8 图 12-6 的克隆版本

12.1.5 基于摘要的上下文相关分析

基于摘要的过程间分析是基于区域的分析技术的扩展。基本上，在一个基于摘要的分析中，每个过程使用一个简洁的描述(摘要)来刻划。这个描述包含这个过程的某些可观察行为。摘要的主要目的是避免在每个可能调用某过程的调用点上都重复分析该过程的过程体。

让我们首先考虑没有递归的情况。每个过程被建模为只有一个入口点的区域。每一对调用者－被调用者之间具有类似于外层区域－内层区域的关系。和过程内分析的唯一不同在于，在过程间分析时，一个过程区域可能嵌套在多个不同的外层区域中。

这个分析由两部分组成：

1）一个自底向上的阶段，它为每个过程计算出一个总结该过程的效果的传递函数。

2）一个自顶向下的阶段，它传播和调用者有关的信息，计算出被调用者的结果。

为了得到完全上下文相关的结果，来自不同调用上下文的信息必须被单独传递到被调用者。如果希望计算过程更高效，但是允许相对的不精确性，那么也可以使用一个交函数合并来自各个

调用者的信息，然后再向下传播到被调用者。

例 12.8 对于常量传播，每个过程都使用一个传递函数作为其摘要，该传递函数描述了过程是如何通过它的过程体传播常量的。在例子 12.2 中，我们可以把 f 总结为如下的函数：如果把一个常量 c 作为 v 的实在参数，那么该函数返回常量 $c+1$。基于这个信息，这个分析过程将确定 t1、t2 和 t3 分别具有常量值 1、244 和 244。请注意，这个分析过程并没有因为不可实现的调用串而产生不精确的结果。

回忆一下，例子 12.4 扩展了例子 12.2，增加了一个函数 g 来调用 f。因此我们可以得出结论，g 的传递函数和 f 的传递函数是相同的。我们仍然可以确定 t1、t2 和 t3 分别具有常量值 1、244 和 244。

现在让我们考虑例子 12.2 中函数 f 内的参数 v 的值是什么。最初考虑时，我们可以把所有调用上下文的结果组合在一起。因为 v 的值可以是 0 或者 243，所以可以简单地确定 v 不是一个常量。这个结论是合理的，因为没有哪个常量可以替换代码中的 v。

如果我们希望得到更加精确的结果，那么可以为感兴趣的上下文计算特定值。必须把信息从我们感兴趣的上下文向下传递，以确定和这个上下文相关的答案。这个步骤和基于区域的分析中的自顶向下过程类似。比如，v 在调用点 c1 处的值为 0，而它在调用点 c2 和 c3 处的值为 243。为了利用 f 内部的常量传播性质，我们需要创建两个克隆来表示这两者的不同，第一个克隆是针对输入值 0 的特例，而后一个克隆是针对输入值 243 的特例，如图 12-9 所示。 □

```
        for (i = 0; i < n; i++) {
c1:             t1 = f0(0);
c2:             t2 = f243(243);
c3:             t3 = f243(243);
                X[i] = t1+t2+t3;
        }

        int f0 (int v) {
            return (1);
        }

        int f243 (int v) {
            return (244);
        }
```

图 12-9 将所有可能的常量参数传递给函数 f 后的分析结果

通过例子 12.8，最后我们看到如果希望在不同的上下文中以不同的方式编译代码，仍然需要克隆代码。本方法和基于克隆的方法的不同之处在于后者在分析之前就需要根据调用串进行克隆。在基于摘要的方法中，克隆是在分析之后，以分析结果为基础进行克隆。即使没有进行克隆，在基于摘要的方法中关于一个被调用过程的运行效果的推理结果也是精确的，不会出现不可实现路径的问题。

除了克隆一个函数，我们也可以对代码进行内联处理。内联的另一个效果是消除了过程调用的开销。

我们可以用计算不动点解的方法来处理递归。当出现递归时，我们首先找出调用图中的强连通分量。在自底向上阶段，只有当一个强连通分量的所有后继都已经被访问之后，我们才访问这个分量。对于一个非平凡的强连通分量，我们迭代地为该分量中的每个过程计算传递函数，直到重复过程收敛为止。也就是说，我们迭代地更新这些传递函数，直到它们不再发生改变为止。

```
int (*p)(int);
int (*q)(int);

int f(int i) {
    if (i < 10)
        {p = &g; return (*q)(i);}
    else
        {p = &f; return (*p)(i);}
}

int g(int j) {
    if (j < 10)
        {q = &f; return (*p)(j);}
    else
        {q = &g; return (*q)(j);}
}

void main() {
    p = &f;
    q = &g;
    (*p)((*q)(N));
}
```

图 12-10 练习 12.1.1 的程序

12.1.6 12.1 节的练习

练习 12.1.1： 图 12-10 中是一个带有两个函数指针 p 和 q 的 C 程序。N 是常量，它可能比 10 小也可能比 10 大。请注意，这个程序会产生无穷的过程调用序列，但是这和

我们当前考虑这个问题的目的无关。

1）找出本程序中的所有调用点。

2）对于每个调用点，p 可能指向哪些函数？q 可能指向哪些函数？

3）画出这个程序的调用图。

4）描述 f 和 g 的所有调用串。

练习 12.1.2：图 12-11 中有一个函数 id，这个函数是一个“单位函数”。它的返回值就是传递给它的参数值。图中还有一个代码片段，该片段包含一个分支语句，后面跟随一个计算 $x+y$ 的和的赋值语句。

```
int id(int x) { return x;}

    ...
if (a == 1) { x = id(2); y = id(3); }
else        { x = id(3); y = id(2); }
z = x+y;
    ...
```

图 12-11 练习 12.1.2 的代码片断

1）检查这个代码，关于 z 在结尾处的值，我们可以有哪些结论？

2）把对 id 的调用当作控制流处理，为这个代码片段构造流图。

3）如果我们对问题 2 中得到的流图应用 9.4 节中描述的常量传播分析，可以确定哪些常量值？

4）图 12-11 中的全部调用点是哪些？

5）共有哪些调用了 id 的上下文？

6）改写图 12-11 中的代码，为每一个调用 id 的上下文克隆一个函数 id 的新版本。

7）把对 id 的调用作为控制流处理，构造你在问题 6 中得到的代码的流图。

8）对于在问题 7 中得到的流图进行常量传播分析。现在可以确定哪些常量值？

12.2 为什么需要过程间分析

我们已经说明了过程间分析有多么困难，现在让我们来解决一个重要的问题：我们为什么以及希望在什么时候使用过程间分析。虽然我们使用常量传播的例子来演示过程间分析，但这个过程间优化技术并不是容易使用的，而且进行这个分析也没有什么特别的好处。仅仅通过过程内分析和把最频繁执行的代码段中的过程调用进行内联处理，就可以获得常量传播的大部分好处。

但是，有很多理由可以说明为什么过程间调用是非常重要的。下面我们描述过程间分析的几个重要应用。

12.2.1 虚方法调用

上面提到过，面向对象程序有很多小的方法。如果我们每次只对一个方法进行优化，那么只能找到很少的优化机会。对方法调用进行解析就可以促成更多的优化。像 Java 这样的程序设计语言动态地载入它的类。结果，我们在编译时刻不知道在 $x.m(\ )$ 这样的调用中对 m 的某次使用到底指向（可能的）多个名为 m 的方法中的哪一个。

很多 Java 语言的实现使用了一个即时编译器，在运行时刻对它的字节码进行编译。一个常见的优化技术是找出程序执行的剖面图，并确定接收对象通常是什么类型。然后，我们可以把调用最频繁的方法内联到调用代码中。相应的代码包含了对这个类型的动态检查，如果运行时刻的接收对象具有预期的类型就执行内联的方法。

只要所有的源代码都在编译时刻可用，我们就可以使用另一种方法来解析对方法名字 m 的使用。然后，可以进行过程间分析，确定对象类型。如果变量 x 的类型是唯一的，那么 $x.m()$ 的使用就可以被解析。我们明确地知道在这个上下文中 m 指向哪个方法。在这种情况下，我们可

以内联这个 m 的代码，编译器甚至不需要在生成的代码中加入对 x 的类型检查代码。

12.2.2 指针别名分析

即使我们不想执行诸如到达定值这样的常见数据流分析的过程间分析版本，这些分析实际上也可以从过程间指针分析中获益。第 9 章给出的所有分析只能应用于没有别名的局部标量变量。然而，指针的使用是很常见的，在像 C 这样的语言中尤其如此。如果知道多个指针是否可能*互为别名*(即可能指向同一个位置)，我们就可以提高第 9 章中介绍的分析技术的精确度。

例 12.9 考虑下面的三个语句组成的序列，它们可能组成了一个基本块：

```
*p = 1;
*q = 2;
 x = *p;
```

如果不知道 p 和 q 是否可能指向同一个位置，也就是说，它们是否可能互为别名，那么就不能确定 x 在基本块的结尾处等于 1。 □

12.2.3 并行化

如第 11 章中所讨论的，将一个应用并行化的最有效方法是寻找最粗粒度的并行性，例如在一个程序的最外层循环中找到的并行性。要完成这个任务，过程间分析技术是非常重要的。标量优化(即基于简单变量的值的优化，比如第 9 章中讨论的技术)和并行化之间有很大的不同。在并行化中，一个可疑的数据依赖关系就可能使得整个循环不可并行化，从而大大降低优化的有效性。这种对不精确性的放大在标量优化中是看不到的。在标量优化中，我们只需要找出大部分优化机会即可。错过一两个机会并不会引起很大的不同。

12.2.4 软件错误和漏洞的检测

过程间分析不仅仅对优化代码很重要。同样的技术也可以用于分析已有软件，寻找各种编码错误。这些错误可能会使得软件变得不可靠，黑客可以利用这些错误来控制或毁坏一个计算机系统。计算机系统的代码错误可能引起严重的安全漏洞。

静态分析可以用于检测是否存在常见的多种错误模式。比如，一个数据项必须用一个锁来保护。另一个例子是，在操作系统中屏蔽一个中断后必须随后重新启用这个中断。这类错误的一个重要源头是跨越过程边界的代码之间的不一致性，因此过程间分析极为重要。PREfix 和 Metal 是两个实用的工具，它们有效地使用过程间分析技术在大型程序中寻找多种程序错误。这类工具可以静态地找到错误，从而大大提高软件可靠性。但是，这些工具既不完全也不健全。从这个意义上说，它们不能找到所有的错误，并且不是报告的所有警告都是错误。遗憾的是，它们使用的过程间分析技术相当地不精确，如果让这些工具报告所有可能的错误，大量的假报警会使得工具无法使用。无论如何，虽然这些工具不是完美的，但它们的系统化使用已经表明它们能够大大提高软件的可靠性。

当考虑安全缺陷时，我们非常期望找到一个程序中所有可能的错误。在 2006 年，黑客使用的两个“最流行”的威胁系统完全的入侵形式是：

1) Web 应用中输入确认机制的缺失：SQL 注入是这种攻击最流行的形式之一。黑客们利用这个弱点，通过操控被 Web 应用接收的输入来获取对数据库的控制。

2) C 和 C++ 程序中的缓冲区溢出。因为 C 和 C++ 不对数组访问进行边界检查，黑客就可以写出一个精心构造的字符串，使得它从缓冲区中延伸到未预料到的区域，从而控制这个程序的运行。

在下一节中，我们将讨论如何使用过程间分析技术来保护程序不受这样的攻击。

12.2.5 SQL 注入

SQL 注入是一种黑客攻击方法。黑客可以通过操纵一个 Web 应用的用户输入，从而获得对

数据库的未授权访问。比如，银行可能希望只要它的用户能够提供正确的口令，他就可以在线完成业务处理。这类系统的一个常用体系结构是让用户在一个 Web 表单中输入字符串，然后把这些字符串组成某个用 SQL 语言编写的数据库查询的一部分。如果系统开发者不小心，用户提供的字符串可能以不可预料的方式改变这个 SQL 语句的含义。

例 12.10 假设一个银行向它的客户提供了对一个关系

```
AcctData(name, password, balance)
```

的访问。也就是说，这个关系是一个由多个三元组组成的表，每个三元组包含一个客户的名字、账户口令和该账户的余额。系统的本意是使得客户只有在提供了他们的名字和正确口令之后才能够看到账户余额。让一个黑客看到账户余额并不是可能发生的最糟糕的事情，但是这个简单例子是更复杂情况的典型代表，更复杂的情况是黑客可以使用那个账户付账。

系统可以按照如下方式实现一次余额查询：

1）用户调用一个 Web 表单，在表单中输入他们的名字和口令。

2）名字被拷贝到一个变量 n，口令被拷贝到另一个变量 p。

3）然后，可能在某些其他过程中，执行下列 SQL 查询：

```
SELECT balance FROM AcctData
WHERE name = ':n' and password = ':p'
```

我们向不熟悉 SQL 的读者解释一下这个查询含义。该语句的含义是："在表 AcctData 中找出一行，要求第一个分量(名字)等于变量 n 中的当前字符串，而第二个分量(口令)等于变量 p 中的当前字符串；然后打印这一行的第三个分量(余额)。"请注意，这个 SQL 语句使用了单引号而不是双引号来分割符号串，n 和 p 之前的冒号表明它们是外围语言的变量。

假设一个黑客想找到 Charles Dickens 的账户余额，他向 n 和 p 提供了下面的值：

```
n = Charles Dickens' --    p = who cares
```

这个奇怪的字符串的作用是把上面的查询转变成

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens' --' and password = 'who cares'
```

在很多数据库系统中，--是一个注释引导符号，其作用是把该行中跟在其后的所有内容看作一个注释。结果，现在这个查询语句要求数据库系统打印出每个名字为'Charles Dickens'的个人的账户余额，而不考虑在 name-password-balance 三元组中和该名字一起出现的口令。也就是说，删除注释之后，这个查询变成了： □

```
SELECT balance FROM AcctData
WHERE name = 'Charles Dickens'
```

在例子 12.10 中，这个"坏"字符串被保存在两个变量中，它们可能在过程之间传递。但是，在更加真实的情况中，这些字符串可能被多次复制，或者和其他字符串组成完整的查询语句。如果我们不对整个程序进行全面的过程间分析，就不能指望能够检测到导致 SQL 注入攻击的代码错误。

12.2.6 缓冲区溢出

当一个由用户提供的精心制作的数据被写到了预想的缓冲区之外并操纵程序的执行时，就发生了*缓冲区溢出攻击*(buffer overflow attack)。比如，一个 C 程序可能从用户那里读取一个字符串 s，然后使用函数调用

```
strcpy(b,s);
```

把它拷贝到一个缓冲区 b 中。如果字符串 s 实际上比缓冲区 b 长，那么在缓冲区 b 之外的某些内存位置上的值将会被改变。这个情况本身可能会使程序产生故障，或者至少产生错误的答案，因为程序使用的某些数据可能已经被改变了。

但是实际情况会更糟糕，选择字符串 s 的黑客可以选择一个特别的值，使得它的作用不仅仅是引起一个错误。比如，如果该缓冲区位于一个运行时刻栈中，那么它可能离存放该函数的返回地址的位置很近。一个经过精心选择的恶意的 s 值可以覆盖掉这个地址，当函数返回时，它跳转到黑客选择的地方。如果黑客熟悉操作系统和硬件，那么他们就能够执行一个命令，让系统赋予他们控制这台计算机的能力。在有些情况下，他们甚至可以有能力让那个假的返回地址把控制传递到作为字符串 s 的一部分的代码中，这样就能将任何种类的程序插入到正在执行的代码中。

为了防止缓冲区溢出，我们要么必须通过静态的方法证明每个数组写运算都处于边界之内，要么必须进行适当的动态数组边界检查。因为在 C 和 C++ 程序中必须手工插入这些边界检查，程序员很容易忘记插入测试代码，或者插入错误的测试代码。人们已经开发了启发式工具来检查是否在调用一个 `strcpy` 之前至少进行了某些测试，虽然这些测试不一定是正确的。

动态边界检查是不可避免的，因为不可能静态地确定用户输入的大小。静态分析可以做的所有事情就是保证正确地插入了动态检查代码。因此，一个可行的策略是让编译器在每个写操作上插入动态边界检查，并以静态分析为手段尽可能优化掉动态检查代码。这样就不再需要去捕捉每个可能违背边界条件的情况。而且，我们只需要优化那些频繁执行的代码区域。

即使我们不在乎运行开销，在 C 程序中插入边界检查也不是容易的事情。一个指针可能指向某个数组的中间，而且我们还不知道这个数组的大小。可以使用已有的技术来动态跟踪各个指针指向的缓冲区的大小。这个信息允许编译器为所有的访问都插入数组边界测试。有意思的是，我们并不建议一检测到缓冲区溢出就停止执行程序。实际上，实践中确实会发生缓冲区溢出，如果我们不允许所有的缓冲区溢出，一个程序就很容易出错。解决的方法是动态扩展缓冲区的大小以应对缓冲区溢出。

可以利用过程间分析技术来提高动态的数组边界检查的速度。比如，假设我们只关注和用户输入字符串有关的缓冲区溢出，那么可以使用静态分析技术来决定哪个变量可能存放了用户提供的内容。和 SQL 注入一样，如果我们能够跟踪一个输入值在过程间传递复制的过程，就有利于消除不必要的边界检查。

12.3 数据流的一种逻辑表示方式

可以说，到现在为止，我们对数据流问题和解答的表示方法是基于集合理论的。也就是说，我们把信息表示成集合，并通过交、并这样的运算来计算结果。比如，当我们在 9.2.4 节中介绍到达定值问题时，我们为一个基本块 B 计算 IN[B]和 OUT[B]，并把它们描述为定值的集合。我们用基本块 B 的 *gen* 和 *kill* 集合来表示这个基本块的内容。

为了应对过程间分析的复杂性，我们引入一个更加通用且更加明确的基于逻辑的表示方法。我们不再说诸如“定值 D 在 IN[B]中”这样的断言，而是使用类似于 $in(B, D)$ 这样的表示方法来表示同样的意思。这么做使我们把那些用以推断程序性质的简明的“规则”表示出来。它也使我们能高效地实现这些规则，实现方法是对集合运算的位向量方法进行推广。最后，逻辑方法使我们能把几个看起来不一样的分析合并成为一个一体化的算法。比如，在 9.5 节中，我们用四个数据流分析组成的序列及两个中间步骤描述了部分冗余消除方法。在逻辑表示方法中，这些步骤可以被合并成为一组逻辑规则。我们可以同时求解这些规则。

12.3.1 Datalog 简介

Datalog 是一个使用类 Prolog 表示方法的语言，但是它的语义要比 Prolog 简单得多。首先，Datalog 的元素是形如 $p(X_1, X_2, \cdots, X_n)$ 的原子(atom)，其中：

1) p 是一个断言——一个表示了一类语句的符号，比如“一个定值到达了一个基本块的

开始处”。

2) X_1、X_2、…、X_n 是变量或常量的项。我们也可以把一些简单表达式当作一个断言的参数。[⊖]

一个*基础原子*(ground atom)是一个其参数都是常量的断言。每个基础原子表明了一个特定的事实，它的值要么是真要么是假。把一个断言表示为一个关系(或者说令该断言取真值的基础原子的表)通常比较方便。每个基础原子表示成关系的一行，或者说一个元组。这个关系的列以属性命名，对于每个属性，每个元组都有一个对应的分量。这些属性对应于用关系方式表示的基础原子的分量。在该关系中的所有基础原子的值都是真，不在此关系中的基础原子的值都为假。

例 12.11 我们假设断言 $in(B, D)$ 表示“定值 D 到达了基本块 B 的开始处”，并假设对于特定的流图 $in(b_1, d_1)$ 为真且 $in(b_2, d_1)$ 和 $in(b_2, d_2)$ 也为真。我们也可以假设对于这个流图而言，所有其他关于 in 的描述都是假的。那么图 12-12 中的关系就表示了对应于这个流图的此断言的取值。

B	D
b_1	d_1
b_2	d_1
b_2	d_2

图 12-12 使用一个关系来表示一个断言的值

这个关系的属性为 B 和 D。这个关系有三个元组，分别是 (b_1, d_1)、(b_2, d_1) 和 (b_2, d_2)。□

有时我们也会看到一个实际上是变量及常量之间的比较运算的原子。比如 $X \neq Y$ 或者 $X = 10$。在这些例子中，断言实际上是比较运算符。也就是说，我们可以把 $X = 10$ 看作它的断言形式：*equals*$(X, 10)$。但是比较断言和其他断言有一个最大的不同之处。一个比较断言有它的标准解释，而像 *in* 这样的普通断言的含义是由一个 Datalog 程序(将在下面描述)定义的。

字面值(literal)是一个原子或其否定形式。我们在一个原子前加 NOT 来表示否定。因此，NOT $in(B, D)$ 是一个断言，表示定值 D 不能到达基本块 B 的开始处。

12.3.2 Datalog 规则

规则是表示逻辑推理关系的一种方法。在 Datalog 中，规则也说明了如何完成对正确的事实的计算。一个规则的形式为：

$$H :- B_1 \& B_2 \& \cdots \& B_n$$

其中的组成部分如下：

- H 和 B_1，B_2，…、B_n 是字面值，即原子或原子的否定形式。但 H 不能是否定形式。
- H 是规则的*头*，B_1、B_2、…、B_n 组成了规则的*体*。
- 每个 B_i 有时被称为规则的*子目标*(subgoal)。

我们应该把符号 :- 读作“如果”。一个规则的含义是“如果规则体为真，那么规则头也为真”。更精确地说，我们按照下面的方法把规则*应用*到一组给定的基础原子集合上。考虑所有可能把规则中的变量替代为常量的替换方法。如果某个替换方法使得规则体的每个子目标都为真(假设所有且只有给定的基础原子为真)，那么我们可以推断：按照这个替换方法把规则头中的变量替换为常量之后得到的断言为真。不能使所有子目标都为真的替换方法没有给我们任何信息，替换后的规则头可能为真也可能为假。

一个 Datalog *程序*是一组规则的集合。这个程序被应用于一组“数据”，即某些断言的基础原子集合。这个程序的结果也是一组基础原子的集合，这个集合通过应用程序中的规则推断得到。程序将不断应用其中的规则，直到不能推断出新的基础原子为止。

⊖ 严格地讲，这样的项是从函数符号构造而来的。它们大大增加了 Datalog 实现的复杂性。但是，只有在不把事情复杂化的情况下我们才会使用少量运算符，比如常量的加法和减法。

例 12.12 一个 Datalog 程序的简单例子是给定一个图的(有向)边，计算这个图的路径。也就是说，断言 $edge(X, Y)$ 表示“有一条从结点 X 到 Y 的边”；断言 $path(X, Y)$ 表示从 X 到 Y 有一条路径。定义路径的规则是：

1) $path(X, Y)$:- $edge(X, Y)$

2) $path(X, Y)$:- $path(X, Z)$ & $path(Z, Y)$

第一个规则是说一条边就是一条路径。也就是说，只要我们把变量 X 替换为一个常量 a 且把变量 Y 替换为一个常量 b，并且 $edge(a, b)$ 为真(即有一条从结点 a 到结点 b 的边)，那么 $path(a, b)$ 也成立(即有一条从 a 到 b 的路径)。第二个规则是说如果有一条从某个结点 X 到某个结点 Z 的路径，并且还有一条路径从 Z 到结点 Y，那么存在一条从 X 到 Y 的路径。这个规则表示“传递封闭性”。请注意，任何路径都可以通过选取路径上的边并不断应用传递封闭性规则得到。

比如，假设下列事实(基础原子)为真：$edge(1, 2)$、$edge(2, 3)$ 和 $edge(3, 4)$。那么，我们可以使用第一个规则进行三次不同的替换，推断出 $path(1, 2)$、$path(2, 3)$ 和 $path(3, 4)$。例如，按照 $X=1$ 和 $Y=2$ 进行替换可以得到第一个规则的实例 $path(1, 2)$:- $edge(1, 2)$。因为 $edge(1, 2)$ 为真，所以可以推导出 $path(1, 2)$。

根据这三个关于 $path$ 的事实，我们可以多次使用第二个规则。如果按照 $X=1$、$Z=2$ 和 $Y=3$ 进行替换，我们可以得到这个规则的实例 $path(1, 3)$:- $path(1, 2)$&$path(2, 3)$。因为规则体中的两个子目标都已经推导出来，已知它们为真，所以可以推出规则头：$path(1, 3)$。然后，使用替换方法 $X=1$、$Y=2$ 和 $Z=4$ 推出规则头 $path(1, 4)$。也就是说，从结点 1 到结点 4 有一条路径。 □

Datalog 的编码规则

我们将在 Datalog 程序中使用如下编码规则：

1) 变量以大写字符开头。

2) 所有的其他元素以小写字符或其他符号(比如数字)开头。这些元素包括断言和用作断言参数的常量。

12.3.3 内涵断言和外延断言

按照 Datalog 程序的惯例，我们把断言分成两类：

1) EDB 断言，或者说*外延数据库*(extensional database)断言，是事先定义的断言。也就是说，它们的真值事实要么通过一个关系或表给出，要么根据断言的含义给出(比如一个比较断言的情况)。

2) IDB 断言，或者说*内涵数据库*(intensional database)断言，只能通过规则定义。

一个断言要么是 IDB，要么是 EDB，且只能是其中之一。这个规定的结果是，任何出现在一个或多个规则头中的断言必然是一个 IDB 断言。出现在规则体中的断言可以是 IDB，也可以是 EDB。比如，在例子 12.12 中，*edge* 是一个 *EDB* 断言，*path* 是一个 IDB 断言。回忆一下，我们给出了一些关于 *edge* 的事实，比如 $edge(1, 2)$，但是所有的 *path* 事实都是通过规则推导出来的。

当 Datalog 程序用于表示数据流算法时，其中的 EDB 断言是根据流图本身计算得到的。IDB 断言被表示成规则，而数据流问题的解决方法就是根据这些规则和给定的 EDB 事实中推导出所有可能的 IDB 事实。

例 12.13 让我们考虑可以如何在 Datalog 中表示到达定值问题。首先，在语句层次上(而不是在基本块层次上)考虑问题是有道理的。也就是说，从一个基本块构造它的 *gen* 和 *kill* 集合的计算过程将会和到达定值本身的计算集成在一起。因此，图 12-13 中给出的基本块 b_1 是很典型的。

请注意，如果一个基本块内有 n 个语句，那么我们用编号 1，2，…，n 来标记块内的程序点。第 i 个定值在第 i 点上出现，而在 0 点上没有定值出现。

程序中的一个点可以表示为一个二元组 (b, n)，其中 b 是一个基本块，而 n 是 0 到基本块 b 内的语句数量之间的一个整数。我们的表示方法需要两个 EDB 断言：

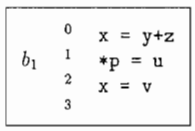


图 12-13 一个语句中包含指针的基本块

1）$def(B, N, X)$ 为真当且仅当基本块 B 中的第 N 个语句可以对变量 X 定值。比如，在图 12-13 中，$def(b_1, 1, x)$ 为真，$def(b_1, 3, x)$ 为真且 $def(b_1, 2, Y)$ 对所有可能在这个点上被指针 p 指向的变量 Y 都为真。现在我们将假设 Y 可以是任何具有 p 所指类型的变量。

2）$succ(B, N, C)$ 为真当且仅当在流图中基本块 C 是基本块 B 的后继，且 B 具有 N 个语句。也就是说，控制流可以从 B 的点 N 到达 C 的点 0。比如，假设 b_2 是图 12-13 中基本块 b_1 的前驱，且 b_2 具有 5 个语句，那么 $succ(b_2, 5, b_1)$ 为真。

这个 Datalog 程序有一个 IDB 断言 $rd(B, N, C, M, X)$。这个断言为真当且仅当在基本块 C 上的第 M 个语句中对变量 X 的定值到达了基本块 B 的点 N。定义断言 rd 的规则在图 12-14 中显示。

规则 1 说明，如果基本块 B 的第 N 个语句对 X 定值，那么 X 的这个定值到达 B 的第 N 个点（即紧跟在这个语句之后的点）。我们前面给出了到达定值问题的集合理论表示方法，而这个规则对应于该表示方法中的概念 gen。

规则 2 表示除非一个定值被某个语句杀死，否则它可以穿越这个语句。而杀死一个定值的唯一方法是 100% 肯定地对其中的变量重新定值。详细地说，规则 2 说明来自基本块 C 中的第 M 个语句的对变量 X 的定值到达基本块 B 中的点 N 的条件是

1）它到达了前一个结点，即 B 中的点 $N-1$。

2）同时至少有一个不同于 X 的变量 Y 可能在 B 的第 N 个语句中定值。

最后，规则 3 表示了流图的控制流。它说基本块 C 中第 M 个语句中对 X 的定值到达基本块 B 的第 0 点的条件是存在某个具有 N 个语句的基本块 D，使得这个对 X 的定值到达 D 的结尾处，并且 B 是 D 的一个后继。 □

例 12.13 中的 EDB 断言 *succ* 显然可以从流图中获得。如果我们保守地估计一个指针可能指向任何地方，那么可以从流图中得到 *def* 断言。如果我们希望把一个指针所指向的范围限定在具有适当类型的变量中，那么我们可以从符号表中获取类型信息，从而使用一个较小的关系 *def*。另一种可选的方法是把 *def* 变成一个 *IDB* 断言，并通过规则来定义它。这些规则将使用更基本 EDB 断言，而这些断言本身可以从流图和符号表中获得。

例 12.14 假设我们引入两个新的 EDB 断言：

1）$assign(B, N, X)$ 为真当且仅当基本块 B 的第 N 个语句的左部为 X。请注意，X 可以是一个变量，也可以是一个具有左值的简单表达式，比如 $*p$。

2）如果 X 的类型为 T，那么 $type(X, T)$ 为真。同样，X 可以是具有左值的任意表达式，而 T 可以是任何合法的类型表达式。

1) $rd(B,N,B,N,X)$:- $def(B,N,X)$

2) $rd(B,N,C,M,X)$:- $rd(B,N-1,C,M,X)$ & $def(B,N,Y)$ & $X \neq Y$

3) $rd(B,0,C,M,X)$:- $rd(D,N,C,M,X)$ & $succ(D,N,B)$

图 12-14 断言 *rd* 的规则集合

然后，我们就可以写出 *def* 的规则，使得 *def* 成为一个 IDB 断言。图 12-15 是对图 12-14 的一

个扩展，它增加了两个 *def* 的可能规则。规则 4 说明，如果基本块 B 的第 N 个语句对 X 赋值，那么这个语句就对 X 定值。规则 5 说明，如果基本块 B 的第 N 个语句对 $*P$ 赋值，且 X 是具有 P 所指类型的任何变量，那么这个语句也可能对 X 定值。其他类型的赋值语句需要其他的 *def* 规则。

现在举例说明如何使用图 12-15 中的规则进行推导。让我们重新考虑图 12-13 中的基本块 b_1。第一个语句把一个值赋给变量 x，因此事实 $assign(b_1, 1, x)$ 出现在 EDB 中。第三个语句也对 x 赋值，因此 $assign(b_1, 3, x)$ 也是一个 EDB 事实。第二个语句通过 p 间接赋值，因此第三个 EDB 事实是 $assign(b_1, 2, *p)$。规则 4 允许我们推导出 $def(b_1, 1, x)$ 和 $def(b_1, 3, x)$。

假设 p 的类型是指向整数的指针（$*int$），且 x 和 y 都是整数。那么我们可以使用规则 5，令 $B = b_1$，$N = 2$，$P = p$，$T = \text{int}$，且 X 等于 x 或 y，推导得到 $def(b_1, 2, x)$ 和 $def(b_1, 2, y)$。类似地，我们可以对其他类型为整数或可转变为整数的变量推导出同样的结果。 □

1) $rd(B,N,B,N,X)$:- $def(B,N,X)$

2) $rd(B,N,C,M,X)$:- $rd(B,N-1,C,M,X)$ & $def(B,N,Y)$ & $X \neq Y$

3) $rd(B,0,C,M,X)$:- $rd(D,N,C,M,X)$ & $succ(D,N,B)$

4) $def(B,N,X)$:- $assign(B,N,X)$

5) $def(B,N,X)$:- $assign(B,N,*P)$ & $type(X,T)$ & $type(P,*T)$

图 12-15 断言 *rd* 和 *def* 的规则

12.3.4 Datalog 程序的执行

每一组 Datalog 规则都定义了它的 IDB 断言的关系。这些关系是程序中的 EDB 断言关系表的函数。开始时假设 IDB 关系为空（即对于所有可能的参数，各个 IDB 断言为假）。然后重复应用这些规则，根据这些规则不断推导出新的事实。当推导过程收敛时，就完成了程序的运行。运行得到的 IDB 关系就形成了程序的输出。这个过程将在下面的算法中正式给出。这个算法和第 9 章中讨论的迭代算法类似。

算法 12.15 Datalog 程序的简单求值。

输入：一个 Datalog 程序和各个 EDB 断言的事实集合。

输出：每个 IDB 断言的事实集合。

方法：对于程序中的每个断言 p，令 R_p 为使该断言为真的事实关系。如果 p 是一个 EDB 断言，那么 R_p 就是该断言给出的所有事实。如果 p 是一个 IDB 断言，我们将计算 R_p。执行图 12-16 中的算法。 □

```
for (p的每个断言IDB)
    R_p = ∅;
while (改变了任何 R_p 的值) {
    考虑所有可能的对各个规则中的变量进行常量
      替换的方法;
    对于每个替换方法，使用当前的 R_p 来确定EDB 和 IDB断言
      的真假值，确定是否某个规则体的所有子目标都为真;
    if (某个替换方法使得一个规则的规则体为真)
        设规则的头断言为q，将替换后的头加入到 R_q 中。
}
```

图 12-16 Datalog 程序的求值

例 12.16 例 12.12 中的程序计算一个图中的路径。应用算法 12.15 时，最初 EDB 断言 *edge* 保存了该图的所有边，而 *path* 的关系为空。第一轮的时候，规则 2 没有产生任何结果，因为此时还没有 *path* 的事实。但是规则 1 使得所有的 *edge* 事实都变成了 *path* 事实。也就是说，在第一轮过后，我们知道 $path(a, b)$ 成立当且仅当有一条从 a 到 b 的边。

在第二轮中，规则 1 没有生成新的 *path* 事实，因为 EDB 关系 *edge* 没有改变。但是，现在规则 2 令我们把两个长度为 1 的路径连接到一起生成一个长度为 2 的路径。也就是说，在第二轮之后，$path(a, b)$ 为真当且仅当从 a 到 b 有一条长度为 1 或 2 的路径。类似地，在第三轮中，我们可

以把长度不大于 2 的路径连接起来找到所有长度不大于 4 的路径。在第四轮，我们发现最大长度为 8 的路径，并且一般来说，在第 i 轮之后，$path(a, b)$ 为真当且仅当有一个从 a 到 b 且长度不大于 2^{i-1} 的路径。 □

12.3.5 Datalog 程序的增量计算

有一个可行的方法可以提高算法 12.15 的效率。请注意，一个新的 IDB 事实只能在第 i 轮被发现的条件如下：它是对某一个规则进行常量替换后的结果，并且其中至少有一个子目标经过变换变成刚刚在第 $i-1$ 轮发现的新事实。这个论断的证明如下：如果子目标中的所有事实在第 $i-2$ 轮的时候都是已知的，那么这个“新”的事实应该在第 $i-1$ 轮进行同样的常量替换时就已经被发现了。

为了利用这个性质，我们为每个 IDB 断言 p 引入一个断言 $newP$，该断言只对上一轮中新发现的 p 事实成立。每一个在其子目标中包含了一个或多个 IDB 的规则都被替换为一组规则。这组规则中的每一个都是通过把原来规则体中的某一个 IDB 断言 q 替换为 $newQ$ 而得到的。最后，对于所有的规则，我们把规则头的断言 h 替换为 $newH$。得到的这些规则被称为具有增量式形式（incremental form）。

像算法 12.15 那样，对应于每个 IDB 断言 p 的关系累积了所有的 p 的事实。在每一轮中，我们

1）应用新的规则对 $newP$ 断言求值。

2）然后从 $newP$ 中减去 p，保证 $newP$ 中的事实确实是新的。

3）把这些 $newP$ 的事实加入到 p 中。

4）把所有 $newX$ 关系表设置为空，准备进行下一轮计算。

这个想法将在算法 12.18 中正式描述。在此之前，我们将先给出一个例子。

1)	$newPath(X,Y)$	:-	$edge(X,Y)$
2a)	$newPath(X,Y)$	:-	$path(X,Z)$ & $newPath(Z,Y)$
2b)	$newPath(X,Y)$	:-	$newPath(X,Z)$ & $path(Z,Y)$

图 12-17 Datalog 程序 path 的增量式规则

例 12.17 再次考虑例子 12.12 中的 Datalog 程序。该程序的规则的增量形式在图 12-17 中给出。规则 1 的规则体中没有 IDB 子目标，因此除了规则头之外没有任何改变。但是规则 2 中有两个 IDB 子目标，因此它变成了两个不同的规则。在每个规则中，$path$ 在规则体中的某次出现被替换为 $newPath$。这两个规则合起来保证了上面描述的思想得以实施，即根据规则连接起来的两条路径中至少有一条是在上一轮中发现的。 □

算法 12.18 Datalog 程序的增量求值。

输入：一个 Datalog 程序和各个 EDB 断言的事实集合。

输出：各个 IDB 断言的事实集合。

方法：对于程序中的每个断言 p，令 R_p 表示使此断言为真的事实的关系。如果 p 是一个 EDB 断言，那么 R_p 就是该断言对应的事实集合。如果 p 是一个 IDB 断言，我们将计算得到 R_p。另外，对于每个 IDB 断言 p，令 R_{newP} 为对应于断言 p 的“新”事实的关系。

1）把程序的规则修改为上面描述的增量形式。

```
for (每个IDB断言 p) {
    R_p = φ;
    R_newP = φ;
}
repeat {
    考虑对所有规则中的变量的所有常量替换方案;
    对每个替换方案，利用 R_p 和 R_newP 来决定各个
    EDB 和 IDB 断言的真假，从而确定是否有某个
    规则体的所有子目标都为真;
    if (某个替换方案使得一个规则的规则体为真)
        把替换后的该规则的头加入到 R_newH 中，其中
            h 是该规则的头的断言;
    for (每个断言 p) {
        R_newP = R_newP − R_p;
        R_p = R_p ∪ R_newP;
    }
} until (所有 R_newP 都为空);
```

图 12-18 Datalog 程序的求值

2）执行图 12-18 中的算法。 □

集合表示法的增量求值

以增量的方式来解决基于集合理论的数据流问题也是可行的。比如，在到达定值问题中，只有当一个定值刚被发现在基本块 B 的前驱 p 的 OUT[P]中时，这个定值才能够在本次计算中出现在 IN[B]中。我们之所以没有尝试以增量的方式来解决这样的数据流问题，因为位向量的实现方式已经非常高效了。一般来说，直接计算整个向量要比决定一个事实是否为新事实更加容易。

12.3.6 有问题的 Datalog 规则

有些 Datalog 规则，或者说程序，在技术上没有任何意义，因此不应该使用。两种最严重的风险是：

1）*不安全规则*：这些规则的头中有一个变量没有以适当的方法出现在规则体中。正确的方法必须限定这个变量只能取那些出现在 EDB 中的值。

2）*不可分层的程序*：一组规则之间存在涉及否定形式的循环定义。

我们将详细讨论这两个风险。

安全规则

出现在某个规则头的任何变量都必须出现在规则体中。不仅如此，这个变量所在的子目标必须是一个普通 IDB 或 EDB 原子。我们不能接受一个变量只出现在一个否定原子中或比较运算符中的情况。制定这个策略是为了避免那些可能使我们推导出无穷多个事实的规则。

例 12.19 规则

$$p(X, Y) \text{ :- } q(Z) \ \&\ \text{NOT } r(X) \ \&\ X \neq Y$$

是不安全的。原因有两个：变量 X 只出现在否定的子目标 $r(X)$ 和比较表达式 $X \neq Y$ 中；Y 只出现在比较式中。结果是只要 $r(X)$ 为假且 Y 不同于 X，p 对于无穷多个二元组(X, Y)为真。 □

可分层的 Datalog 程序

为了让一个程序有意义，递归定义和否定形式必须分开。正式要求如下。我们必须能够把 IDB 断言分割成为多个*层次*(strata)，使得如果存在一个规则，其头断言为 p 且有一个形如 NOT $q(\cdots)$的子目标，那么 q 要么是一个 EDB，要么是一个层次低于 p 的 IDB 断言。只要满足这个规则，我们就可以用算法 12.15 或算法 12.18 从低到高地对各个层次求值。首先处理处理较低层次的 IDB，在处理较高层次时把低层次上的 IDB 当作 EDB。但是，如果我们违反了这个规则，那么如下面的例子所示，迭代算法可能无法收敛。

例 12.20 考虑下面的由单个规则构成的 Datalog 程序：

$$p(X) \text{ :- } e(X) \ \&\ \text{NOT } p(X)$$

假设 e 是一个 EDB 断言，并且只有 $e(1)$为真。那么 $p(1)$为真吗？

这个程序是不可分层的。不管我们把 p 放在哪一层，它的规则中有一个子目标是某个 IDB（即 p 本身）的否定形式，且这个 IDB（即 p）所在的层次当然不会比 p 的层次更低。

如果我们应用上面的迭代算法，我们从 $R_p = \emptyset$ 开始，因此开始时的答案是“不；$p(1)$不为真。”但是，因为 $e(1)$和 NOT $p(1)$都为真，所以第一次迭代时推导出 $p(1)$。但是，之后的第二次迭代告诉我们 $p(1)$为假。也就是说，在这个规则中，把 X 替换为 1 不会令我们推导出 $p(1)$，因为子目标 NOT $p(1)$为假。类似地，第三次迭代说 $p(1)$为真，第四次迭代说它是假，如此往复。

我们断定这个不可分层的程序是无意义的，也不能把它看作一个正确的程序。 □

12.3.7 12.3 节的练习

！ **练习 12.3.1**：在这个问题中，我们将考虑一个比例子 12.13 简单的到达定值数据流分析问题。假设每个语句本身就是一个基本块，并且一开始的时候假设每个语句对且只对一个变量定值。EDB 断言 *pred*(*I*, *J*)表示语句 *I* 是语句 *J* 的一个前驱。EDB 断言 *defines*(*I*, *X*) 表示语句 *I* 所定值的变量为 *X*。我们将使用 IDB 断言 *in*(*I*, *D*)和 *out*(*I*, *D*) 分别表示定值 *D* 到达语句 *I* 的开头和结尾。请注意，一个定值实际上是一个语句的编号。图 12-19 是一个 Datalog 程序，它表示计算到达定值的常用算法。

请注意，规则 1 是说明一个语句杀死了它自己，但是规则 2 保证一个语句总是在它自己的输出集合中。规则 3 是普通的传递函数。因为 *I* 可以有多个前驱，所以规则 4 表示了交汇运算的情况。

你要解决的问题是修改这些规则来处理常见的二义性定义的情况，比如通过一个指针进行赋值运算。在这种情况下，*defines*(*I*, *X*)对多个不同的 *X* 和一个 *I* 成立。一个定值最好表示为一个二元组(*D*, *X*)，其中 *D* 是一个语句，*X* 是一个可能被 *D* 定值的变量。这样做的结果是，*in* 和 *out* 变成了带有三个参数的断言。例如，*in*(*I*, *D*, *X*)表示在语句 *D* 上对 *X* 的(可能的)定值到达了语句 *I* 的开始处。

练习 12.3.2：编写一个和图 12-19 类似的 Datalog 程序来计算可用表达式。除了断言 *defines* 之外，再加上一个断言 *eval*(*I*, *X*, *O*, *Y*) 。这个断言说明语句 *I* 使得表达式 *X O Y* 被求值。这里 *O* 是表达式中的运算符，例如 + 。

练习 12.3.3：编写一个和图 12-19 类似的 Datalog 程序来计算活跃变量。除了断言 *defines* 之外，假设一个断言 *use*(*I*, *X*)表示语句 *I* 使用了变量 *X*。

```
1)  kill(I,D)   :-  defines(I,X) & defines(D,X)

2)  out(I,I)    :-  defines(I,X)
3)  out(I,D)    :-  in(I,D) & NOT kill(I,D)

4)  in(J,D)     :-  out(J,D) & pred(J,I)
```

图 12-19 一个简单的到达定义分析的 Datalog 程序

练习 12.3.4：在 9.5 节中，我们定义了一个涉及六个概念的数据流计算，这些概念包括：预期执行的、可用的、最早的(earlist)、可后延的、最后的(latest)和被使用的。假设我们已经编写了一个 Datalog 程序。程序中包含了一些以 EDB 断言方式定义的可以从程序中获得的概念(例如 gen 和 kill 信息)，并且使用这些 EDB 断言和这六个概念中的其他概念定义了每个概念。这六个概念中的哪些概念依赖于哪些其他概念？这些依赖关系中哪些是否定形式的？得到的 Datalog 程序是可分层的吗？

练习 12.3.5：假设 EDB 断言 *edge*(*X*, *Y*)包含下面的事实：

edge(1, 2) *edge*(2, 3) *edge*(3, 4)

edge(4, 1) *edge*(4, 5) *edge*(5, 6)

1) 使用算法 12.15 中的简单求值策略，在这个数据上模拟运行例子 12.12 中的 Datalog 程序。给出每一轮中找到的 *path* 事实。

2) 在这个数据上模拟执行图 12-17 中的 Datalog 程序。该程序实现了算法 12.18 中的增量式求值策略。给出每一轮中找出的 *path* 的事实。

练习 12.3.6：下面的规则

$$p(X, Y) \text{ :- } q(X, Z) \ \&\ r(Z, W) \ \&\ \text{NOT } p(W, Y)$$

是一个较大的 Datalog 程序 *P* 的一部分。

1) 指出这个规则的头、规则体和各个子目标。

2）哪些断言一定是程序 P 的 IDB 断言？

3）哪些断言一定是 P 的 EDB 断言？

4）这个规则安全吗？

5）P 是可分层的吗？

练习 12.3.7：把图 12-14 中的规则转换成为增量形式。

12.4 一个简单的指针分析算法

在本节中，我们开始讨论一个非常简单的控制流无关的指针别名分析技术。这个技术假设被分析程序中没有过程调用。我们将在以后各节中说明如何处理过程，首先给出上下文无关的处理方法，然后再给出上下文相关的方法。控制流相关会增加很多复杂性，并且对于 Java 这样的语言来说，因为方法常常很小，所以控制流相关性和上下文相关性相比就不是那么重要。

在指针别名分析中，我们希望了解的基本问题是一对给定的指针是否可能互为别名。回答这个提问的方法之一是对每个指针计算下面问题的答案："这个指针可能指向哪些对象？"如果两个指针可能指向同一个对象，那么它们可能互为别名。

12.4.1 为什么指针分析有难度

对 C 语言程序进行指针别名分析特别困难，因为 C 程序可以对指针进行任何运算。实际上，程序可以读入一个整数并把它赋给一个指针，这么做会使得这个指针成为程序中所有其他指针变量的别名。Java 中的指针称为引用，对它们的分析要简单得多。它不支持算术运算，并且指针只能指向一个对象的开头。

指针别名分析必须是过程间分析。没有过程间分析，我们就必须假设任何方法调用都可能改变所有可被它访问的指针变量所指向的内容，造成所有过程内的指针别名分析非常低效。

支持间接函数调用的语言对指针别名分析提出了另一个挑战。在 C 语言中，人们可以通过调用一个解引用的函数指针来实现函数的间接调用。在分析被调用函数之前，我们需要知道这个函数指针指向哪里。显然，在分析被调用的函数之后，我们会发现这个函数指针可能指向更多的函数，因此这个过程需要迭代进行。

虽然 C 语言中的大部分函数是被直接调用的，但是 Java 中的虚方法使得很多调用成为间接调用。给定一个 Java 程序中的调用 x.m()，对象 x 可能属于很多个类，这些类都具有名为 m 的方法。我们对 x 的实际类型了解得越精确，我们的调用图也就越精确。在理想情况下，我们可以在编译时刻准确地确定 x 的类，从而准确知道 m 指向哪个方法。

例 12.21 考虑下面的 Java 语句序列：

```
Object o;
o = new String();
n = o.hashCode();
```

这里 o 被声明为一个 `Object`。如果不分析 o 指向什么，我们必须把在各个类中声明的名为"hashCode"的所有方法都当作可能的调用目标。知道 o 指向一个 `String` 对象将会使过程间分析把范围精确地缩小到在 `String` 中声明的方法。 □

也可以使用近似的方法来减少目标的数量。比如，我们可以静态地确定被创建的对象的类型，然后把分析范围限定在这些类型中。但是，如果我们可以在做指针指向分析的同时，利用指针指向信息动态地构造调用图，就可以得到更加精确的结果。更加精确的调用图不仅仅可以获得更加精确的结果，也可以大大减少分析所需时间。

指针指向分析是很复杂的。它不是"简单的"数据流问题，在这类问题中我们只需要模拟单

次执行一个语句循环的效果。在指针分析中，当我们发现一个新的指针目标时，我们必须重新分析所有把这个指针所指向的内容赋给其他指针的语句。

为简单起见，我们将主要关注 Java。我们将从控制流无关和上下文无关的分析开始。当前我们假设程序中没有方法调用。然后，我们描述如何在计算指针指向分析结果的同时动态地构造调用图。最后我们将描述一个处理上下文相关性的方法。

12.4.2 一个指针和引用的模型

假设我们的语言可以用下列方式来表示和操作引用：

1）某些程序变量的类型为“指向 T 的指针”或“指向 T 的引用”，其中 T 是一个类型。这些变量可以是静态的，也可能位于运行时刻栈中。我们简单地称它们为变量。

2）有一个对象的堆。所有变量都指向堆中的对象，不指向其他变量。这些对象称为堆对象（heap object）。

3）一个堆对象可以有多个字段（field），一个字段的值可以是指向一个堆对象的引用（但是不能指向一个变量）。

可以使用这个结构很好地对 Java 建模，我们将在例子中使用 Java 的语法。请注意，在对 C 语言建模时这个结构的效果就不太好，因为在 C 语言中指针变量可以指向其他指针变量。而且，从原则上讲，任何 C 语言的值都可以被强制转化成为一个指针。

因为我们进行的是上下文无关的分析，所以只需要断定一个给定的变量 v 能够指向一个给定的堆对象 h，不需要指出在程序中的什么地方 v 可能指向 h，或者在什么样的上下文中 v 可以指向 h。请注意，变量可以通过它的全名来命名。在 Java 中，这个全名包括了模块、类、方法和方法中的块以及变量名本身。因此，我们可以区分标识符相同的多个变量。

堆对象没有名字。因为可能动态创建出无限多个对象，所以人们经常使用近似方式给对象命名。一个惯例是使用创建对象的语句来指定对象。因为一个语句可以被执行多次，每次都可以创建一个新的对象，因此一个形如“v 可以指向 h”的断言实际上是说“v 可以指向标号为 h 的语句创建的一个或者多个对象。”

分析的目标是确定各个变量以及每个堆对象的各字段可能指向哪些对象。我们把这个分析称为指针指向分析（points-to analysis）。如果两个指针的指向集合相交，那么它们互为别名。这里我们描述的是一个基于包含（inclusion-based）的分析技术。也就是说，一个形如 v = w 的语句使得变量 v 指向 w 所指向的所有对象，但是反过来不成立。虽然这个方法看起来显而易见，但我们还可以使用其他的方法来定义指向分析。比如，我们可以定义一个基于等价关系（equivalence-based）的分析，使得形如 v = w 的语句把变量 v 和 w 转变成一个等价类。等价类中的变量指向同样的对象。虽然这种表示法不能很好地估算别名问题，但它仍然为哪些变量指向同一类对象的问题提供了一个快速的求解方法，而且效果通常很好。

12.4.3 控制流无关性

我们首先给出一个很简单的例子，说明在指针指向分析中忽略控制流带来的影响。

例 12.22 图 12-20 中创建了三个对象 h、i 和 j，并分别赋给变量 a、b 和 c。显然到第 3 行结束的时候，a 指向 h，b 指向 i，c 指向 j。

如果接着分析语句 4 ~ 6，会发现在第 4 行之后，a 只指向 i。第 5 行之后，b 只指向 j。第 6 行之后 c 只指向 i。 □

```
1)  h:  a = new Object();
2)  i:  b = new Object();
3)  j:  c = new Object();
4)      a = b;
5)      b = c;
6)      c = a;
```

图 12-20 例 12.22 的 Java 代码

上面的分析是控制流相关的，因为我们沿着控制流计算了在每个语句之后各个变量会指向哪个对

象。换句话说，除了考虑各个语句生成哪些指向信息之外，我们也考虑了每个语句“杀死了”哪些指向信息。比如，语句 b = c; 杀死了之前的事实“b 指向 i”并生成了新的关系“b 指向 c 所指向的东西”。

一个控制流无关分析忽略了控制流。这么做实质上就是假设被分析程序中的各个语句可以按照任何顺序执行。它只计算一个全局性的指向映射，这个映射指明了每个变量在程序执行的各点上可能指向哪些对象。如果一个变量在程序中两个不同语句的执行之后指向两个不同的对象，我们只记录它可能指向这两个对象。换句话说，在控制流无关分析中，任何赋值都不会“杀死”任何指向关系，而是只能“生成”更多的指向关系。为了计算控制流无关分析的结果，我们不断向指针指向关系中加入各个语句的效果，直到无法找到新的关系为止。显然，缺乏控制流相关性大大弱化了分析的结果，但是这么做通常可以降低为表示分析结果而使用的数据的大小，并使得算法更快地收敛。

例 12.23 回到例 12.22，第 1 行到第 3 行仍然告诉我们 a 可以指向 h；b 可以指向 i；c 可以指向 j。根据第 4 行和第 5 行，a 可以指向 h 和 i；b 可以指向 i 和 j。根据第 6 行，c 可以指向 h、i 和 j。这个信息又影响了第 5 行，接着又影响了第 4 行。最后，我们只得到一个没有用的结论，即任何指针可能指向任何对象。 □

12.4.4 在 Datalog 中的表示方法

现在我们基于上面的讨论把一个控制流无关的指针别名分析正式表示出来。现在忽略过程调用，并将关注四种可能影响指针的语句：

1）对象创建：h: $T\ v = new\ T()$;　这个语句创建了一个新的堆对象，并且变量 v 可以指向它。

2）复制语句：v = w;　这里 v 和 w 是两个变量。这个语句使得 v 指向 w 当前所指的堆对象，即 w 被复制到 v 中。

3）字段保存：v.f = w;　v 所指向的对象类型必须有一个字段 f，并且这个字段必须是某一种引用类型。令 v 指向堆对象 h，并令 w 指向 g。这个语句使得 h 中的字段 f 现在指向 g。请注意，变量 v 的值没有改变。

4）字段读取：v = w.f;　这里 w 是一个指向某个具有字段 f 的堆对象的变量，而 f 指向某个堆对象 h。这个语句使得变量 v 指向 h。

请注意，源代码中的复合字段访问，比如 v = w.f.g，可以被分解为两个基本的字段读取语句：

```
v1 = w.f;
v  = v1.g;
```

现在，我们把这个分析用 Datalog 规则正式表示出来。首先，只需要计算两个 IDB 断言：

1）$pts(V, H)$ 表示变量 V 可能指向一个堆对象 H。

2）$hpts(H, F, G)$ 表示堆对象 H 的字段 F 可能指向堆对象 G。

EDB 关系根据程序本身构造得到。因为在控制流无关的分析中，程序中语句的位置和分析无关，所以只需要在 EDB 中确定程序中是否存在某种形式的语句。在接下来的内容中，我们将做一个方便的简化。我们没有定义专门的 EDB 关系来存放从程序中获取的信息，而是使用带引号的语句的方式来指明一个或者多个 EDB 关系。这些关系表示程序中存在这样的语句。比如，“H:$T\ V = new\ T()$”是一个 EDB 事实，它表示在语句 H 中有一个赋值使得变量 V 指向一个新的类型为 T 的对象。在实践中，我们假设有一个对应的 EDB 关系，其中包含的基础原子和程序中这种形式的语句一一对应。

根据这种约定，我们在编写 Datalog 程序时要做的全部工作就是为上面的四种语句中的每一种写出一个规则。相应的程序在图 12-21 中给出。规则 1 说明如果语句 H 是把一个新对象赋给 V 的赋值语句，变量 V 就可能指向堆对象 H。规则 2 说明如果存在一个复制语句 V = W，并且 W 可以指向 H，那么 V 也可以指向 H。

1)	$pts(V,H)$	:-	"H : T V = new T()"
2)	$pts(V,H)$	:-	"$V = W$" & $pts(W,H)$
3)	$hpts(H,F,G)$	:-	"$V.F = W$" & $pts(W,G)$ & $pts(V,H)$
4)	$pts(V,H)$	:-	"$V = W.F$" & $pts(W,G)$ & $hpts(G,F,H)$

图 12-21 控制流无关指针分析的 Datalog 程序

规则 3 说明，如果存在一个形如 V. F = W 的语句，W 可以指向 G，并且 V 可以指向 H，那么 H 的字段 F 可以指向 G。最后，规则 4 说明，如果存在一个形如 V = W. F 的语句，W 可以指向 G，并且 G 的字段 F 可以指向 H，那么 V 可以指向 H。请注意，pts 和 $hpts$ 是相互递归的，但是这个 Datalog 程序可以用任何一个在 12. 3. 4 节中讨论的迭代算法进行求值。

12. 4. 5 使用类型信息

因为 Java 是类型安全的，变量只能指向和它的声明类型相兼容的类型。比如，把一个属于一个变量的声明类型的超类的对象赋给这个变量将引发一个运行时刻异常。考虑图 12-22 中的简单例子，其中 S 是 T 的子类。如果 p 为真，这个程序将引发一个运行时刻异常，因为 a 不能被赋予一个类型为 T 的对象。这样，因为类型约束的原因，我们可以静态地断定 a 只能指向 h 而不能指向 g。

```
     S a;
     T b;
     if (p) {
g:     b = new T();
     } else
h:     b = new S();
     }
     a = b;
```

图 12-22 具有类型错误的 Java 程序

因此，我们在分析技术中引入三个 EDB 断言。这些断言反映了被分析代码中的重要类型信息。我们将使用下面的断言：

1）$vType(V, T)$表示变量 V 被声明为类型 T。

2）$hType(H, T)$表示堆对象 H 在分配时具有类型 T。如果一个被创建的对象是由某个核心代码例程返回的，那么有可能不能精确决定它的类型。此时，被创建对象的类型可以被保守地设定为所有可能的类型。

3）$assignable(T, S)$表示类型为 S 的对象可以被赋值给一个类型为 T 的变量。这个信息通常是从程序中子类型的声明中收集的，同时也包含了关于语言中预定义类的信息。$assignable(T, T)$总是取真值。

我们可以修改图 12-21 中的规则，使得只有当被赋值变量被赋予的堆对象的类型是可赋值类型时才可以进行推理。新的规则在图 12-23 中显示。

我们首先修改规则 2。新规则的最后三个子目标表示我们可以断定 V 可以指向 H 的条件是 V 和堆对象 H 的类型分别是 T 和 S，并且类型为 S 的对象可以被赋给指向类型 T 的变量。规则 4 中也增加了一个类似的附加约束。请注意，在规则 3 中没有附加约束，因为所有的保存运算必须通过变量进

1)	$pts(V,H)$	:-	"H : T V = new T()"
2)	$pts(V,H)$	:-	"$V = W$" & $pts(W,H)$ & $vType(V,T)$ & $hType(H,S)$ & $assignable(T,S)$
3)	$hpts(H,F,G)$	:-	"$V.F = W$" & $pts(W,G)$ & $pts(V,H)$
4)	$pts(V,H)$	:-	"$V = W.F$" & $pts(W,G)$ & $hpts(G,F,H)$ & $vType(V,T)$ & $hType(H,S)$ & $assignable(T,S)$

图 12-23 向控制流无关指针分析增加类型约束

行，而这些变量的类型已经被检查过了。在其中加入的任何类型约束只能多处理一种情况，即基对象为 null 常量的情况。

12.4.6 12.4 节的练习

练习 12.4.1：在图 12-24 中，h 和 g 用于表示新创建对象的标号，它们不是代码的一部分。你可以假设类型为 T 的对象有一个字段 f。使用本节中的 Datalog 规则来推导出所有可能的 *pts* 和 *hpts* 事实。

```
h: T a = new T();
g: T b = new T();
   T c = a;
   a.f = b;
   b.f = c;
   T d = c.f;
```

图 12-24 练习 12.4.1 的代码

！练习 12.4.2：对下列代码

```
g: T a = new T();
h:   a = new T();
   T c = a;
```

应用本节中的算法将可以推导出 a 和 b 都可能指向 h 和 g。如果代码写成

```
g: T a = new T();
h: T b = new T();
   T c = b;
```

我们就能够精确地推导出 a 可能指向 h，且 b 和 c 可能指向 g。请给出一个可以避免这种不精确情况的过程内数据流分析技术。

！练习 12.4.3：如果我们用图 12-21 中的规则 2 所描述的复制运算来模拟函数调用和返回操作，就可以把本节中的分析技术扩展为过程间分析技术。也就是说，一个调用把实在参数复制到相应的形式参数中，而函数返回运算把存储返回值的变量复制到被赋予调用结果的变量中。考虑图 12-25 的程序。

```
t p(t x) {
    h: T a = new T();
       a.f = x;
       return a;
}

void main() {
    g: T b = new T();
       b = p(b);
       b = b.f;
}
```

图 12-25 指针指向分析的示例代码

1）对这个代码进行控制流无关的分析。

2）某些在问题 1 中做出的推导实际上是“假冒的”，也就是说它们并不表示任何可能在运行时刻产生的事件。这个问题的源头可以追溯到对变量 b 的多次赋值。改写图 12-25 中的代码，使得没有变量被多次赋值。对修改后的代码再次分析，说明每个推导得到的 *pts* 和 *hpts* 的事实都会在运行时刻发生。

12.5 上下文无关的过程间分析

现在我们考虑方法调用。我们首先解释如何使用指针指向分析技术来获得精确的调用图，而调用图又可以用于计算精确的指针指向分析结果。然后，我们正式描述如何动态地生成调用图，并说明如何用 Datalog 简洁地描述这个分析过程。

12.5.1 一个方法调用的效果

在 Java 中，一个形如 `x = y.n(z)` 的方法调用对指针指向关系的影响可以计算如下：

1）确定接收对象的类型，也就是 y 所指向对象的类型。假设它的类型是 t。令 m 是最低的具有名为 n 的例程的 t 的超类中的那个名为 n 的例程。请注意，一般情况下只能动态确定被调用的方法。

2）方法 m 的形式参数被赋予了实在参数所指向的对象。实在参数不仅仅包括直接传递的参数，也包括接收对象本身。每个方法调用把接收对象赋给 `this` 变量[⊖]。我们把 `this` 变量当作各个方法的第 0 个形式参数。在 `x = y.n(z)` 中有两个形式参数：y 所指向的对象被赋给变量 `this`，而 z 所指向的对象被赋给 m 中声明的第一个形式参数。

⊖ 请记住，变量是通过它们所属的方法进行区分的，因此有很多个名字为 `this` 的变量，程序中的每个方法有一个 `this` 变量。

3）方法 m 的返回对象被赋给这个赋值语句的左部变量。

在上下文无关分析中，参数和返回值都由复制语句建模。尚待解决的一个有意思的问题是如何确定接收对象的类型。我们可以根据这个变量的声明保守地确定它的类型。比如，被声明变量的类型为 t，那么只有 t 的某个子类型中名字为 n 的方法会被调用。遗憾的是，如果被声明变量的类型为 `Object`，那么所有名为 n 的方法都是可能的调用目标。在密集使用对象层次结构和包含了大型类库的实际程序中，这个方法可能会得到很多虚假的调用目标，使得分析过程既缓慢又不精确。

我们需要知道被分析的变量可能指向什么样的对象，以便计算出调用目标。但是，除非我们知道了调用目标，否则无法找出所有这些变量会指向什么样的对象。这个递归关系要求我们在计算指针指向集合的同时找出调用目标。这个分析需要不断进行，直到找不到新的调用目标和新的指针指向关系为止。

例 12.24 在图 12-26 的代码中，r 是 s 的一个子类，而 s 本身又是 t 的一个子类。如果只使用声明类型的信息进行分析，`a.n()`可以调用在上述代码中声明的三个名为 n 的方法中的任何一个，因为 s 和 r 都是 a 的声明类型 t 的子类型。不仅如此，看起来在第 5 行之后 a 可以指向对象 g、h 和 i。

```
        class t {
1) g:     t n() { return new r(); }
        }
        class s extends t {
2) h:     t n() { return new s(); }
        }
        class r extends s {
3) i:     t n() { return new r(); }
        }

        main () {
4) j:     t a = new t();
5)          a = a.n();
        }
```

图 12-26 一个虚拟方法调用

通过分析程序中指针指向关系，我们首先确定 a 可以指向 j，而 j 是一个类型为 t 的对象。因此，第 1 行中声明的方法是一个调用目标。分析第 1 行，我们确定 a 也可能指向 g，而 g 是一个类型为 r 的对象。因此，第 3 行中声明的方法也可能是一个调用目标，且现在 a 可能也指向 i，而 i 是另一个类型为 r 的对象。因为没有发现更多的新调用目标，这个分析过程终止了。它既没有分析第 2 行中声明的方法，也没有断定 a 可能指向 h。 □

12.5.2 在 Datalog 中发现调用图

为了写出上下文无关的过程间分析的 Datalog 规则，我们引入三个 EDB 断言，每个断言都能够从源代码中轻易获得：

1）$actual(S, I, V)$ 表示 V 是调用点 S 上的第 I 个实在参数。

2）$formal(M, I, V)$ 表示 V 是方法 M 中声明的第 I 个形式参数。

3）$cha(T, N, M)$ 表示在一个类型为 T 的接收对象上调用 N 时实际调用的方法是 M（cha 是 class hierarchy analysis（类层次结构分析）的缩写）。

调用图的每个边都被表示为一个 IDB 断言 *invokes*。当我们找到的调用图的边越来越多时，根据参数被传入及返回值被传出的情况，我们会得到越来越多的指针指向关系。这个效果被总结为图 12-27 中的规则。

1) $invokes(S, M)$:- "$S: V.N(...)$" & $pts(V, H)$ & $hType(H, T)$ & $cha(T, N, M)$

2) $pts(V, H)$:- $invokes(S, M)$ & $formal(M, I, V)$ & $actual(S, I, W)$ & $pts(W, H)$

图 12-27 发现调用图的 Datalog 程序

第一个规则计算了调用点的调用目标。也就是说，"S: $V.N(\ldots)$"表明存在一个标号为 S 的调用点，它调用了由 V 指向的接收对象的名为 N 的方法。这些子目标表示，如果 V 可以指向实际类型为 T 的堆对象 H，并且 M 是在调用 T 类型的对象时所使用的方法，

那么调用点 S 可能调用方法 M。

第二个规则说明，如果调用点 S 可以调用方法 M，那么 M 的每个形式参数都可能指向本次调用中相应的实在参数所指向的任何对象。处理返回值的规则留作练习请读者自行完成。

把这两个规则和 12.4 节中解释的规则组合起来，我们就建立了一个使用调用图的上下文无关的指针指向分析方法。其中的调用图是在分析的同时动态生成的。这个分析的副作用是使用上下文无关和控制流无关的指针指向分析生成了一个调用图。相对于那个只根据类型声明和语法分析得到的调用图，这个调用图要精确得多。

12.5.3 动态加载和反射

Java 这样的语言支持类的动态加载。因此分析一个程序执行的所有代码是不可能的，也就不可能静态地给出任何对调用图和指针别名的保守的估算。静态分析只能基于被分析代码给出一个近似。请记住，这里描述的所有分析都可以在 Java 字节码的层次上应用，因此不需要检查它们的源代码。这个选项非常重要，因为 Java 程序常常使用很多的类库。

即使假设已经分析了所有被执行的代码，还有一个更加复杂的机制使得我们不可能进行保守分析：反射机制。反射机制允许程序动态地决定将要创建的对象的类型、被调用的方法的名字以及被访问的字段名。这些类型、方法和字段名可以通过计算获得，也可以根据用户输入获得，因此一般情况下唯一可能的近似估算就是假设什么都有可能。

例 12.25 下面的例子给出了反射机制的常见用法：

```
1) String className = ...;
2) Class c = Class.forName(className);
3) Object o = c.newInstance();
4) T t = (T) o;
```

其中的 `Class` 库中的方法 `forName` 的输入是一个包含了类名的字符串，它返回这个类。方法 `newInstance` 返回该类的一个实例。对象 o 的类型被强制转换成为所有预期类的超类 T，而不是直接把 `Object` 当作 o 的类型。 □

虽然很多大的 Java 应用使用反射机制，但它们通常使用一些常见的习惯用法，比如例子 12.25 中给出的用法。只要这个应用没有重新定义类加载器，我们就可以在知道 `className` 的值时指出这个对象所属的类。如果 `className` 的值是在程序中定义的，因为 Java 中的字符串是不可变的，那么知道 `className` 指向什么值就可以知道这个类的名字。这个技术是指针指向分析的另一个应用。如果 `className` 的值是基于用户输入的，那么指针指向分析可以帮助确定这个值是在哪里输入的，而开发者就可以限定这个值的取值范围。

类似地，我们可以利用类型强制转换语句，即例子 12.25 中的第 4 行，来估算动态创建的对象的类型。假设没有重新定义强制类型转换的异常处理程序，那么这个对象必然属于类型 T 的某一个子类。

12.5.4 12.5 节的练习

练习 12.5.1：对于图 12-26 中的代码，

1）构造 EDB 关系 *actual*、*formal* 和 *cha*。

2）推导出所有可能的 *pts* 和 *htps* 事实。

！**练习 12.5.2**：你将如何向 12.5.2 节中的 EDB 断言和规则中加入附加的断言和规则来处理下面的情况：如果一个方法调用返回了一个对象，那么被赋值为这个调用结果的变量可能指向任何用以存放返回值的变量所指向的任何对象。

12.6 上下文相关指针分析

12.1.2 节中讨论过，上下文相关性可以大大提高过程间分析的精确性。我们讨论了两种过程间分析的方法，一种基于克隆的方法（见 12.1.4 节），另一种是基于摘要的方法（见 12.1.5 节）。那么我们应该使用哪一个方法呢？

在计算指针指向信息的摘要时有几个难点。首先，这些摘要很大。每个方法的摘要必须包括这个函数和所有被调用者可能做出的所有更新所产生的影响。这些影响需要用输入参数来表示。也就是说，一个方法可能改变的指向集合包括：所有可通过静态变量及输入参数到达的所有数据的指向集合，以及由该方法及被调用方法所创建的全部对象的指向集合。虽然人们已经给出了复杂的解决方案，但是现在还没有解决方法可以被应用到大型程序中。即使摘要可以通过自底向上的方式计算得到，但如何在一个典型的自顶向下处理过程中计算所有上下文环境下的指针指向集合是一个更大的问题。因为上下文环境的数量可能按照指数级增长。这样的信息对于一些全局性查询是必须的，比如在代码中找出指向某个特定对象的所有指针。

在本节中，我们将讨论基于克隆的上下文相关分析技术。基于克隆的分析直接为每个感兴趣的上下文都给出一个对应方法的克隆。然后，我们对克隆得到的调用图进行上下文无关分析。虽然这个方法看起来简单，但最大的难点在于如何处理大量克隆的细节。有多少个上下文？即使我们像 12.1.3 中讨论的那样把所有递归调用环塌缩为一个点，在一个 Java 应用中找到 10^{14} 个上下文的情况也并不少见。把这么多上下文的分析结果用某种方式表示出来是我们所面临的挑战。

我们把对上下文相关性的讨论分成两个部分：

1）如何在逻辑上处理上下文相关性？这个部分较为简单，因为可以直接对克隆得到的调用图应用上下文无关的分析算法。

2）如何表示指数量级的上下文？方法之一是把这个信息表示为一个二分决策图（BDD）。这是一个经过高度优化的数据结构，曾经用于很多其他的应用。

这个处理上下文相关性的方法很好地说明了抽象方面的重要性。我们将说明如何应用人们多年来在 BDD 抽象方面所做的工作来消除算法的复杂性。我们可以用很少几行 Datalog 程序来表示一个上下文相关的指针指向分析。而这个程序利用了已有的几千行用于 BDD 数据操作的代码。这个方法具有多个重要的优势。首先，它使得人们能够比较容易地表示那些利用指针指向分析结果进行深度分析的技术。无论如何，指针指向分析结果本身并不令人感兴趣。第二，它使得正确写出这个分析方法的任务变得容易得多，因为它利用了很多行经过充分调试的代码。

12.6.1 上下文和调用串

下面描述的上下文相关的指针指向分析假设我们已经计算得到了一个调用图。这个假设有助于我们使用紧凑的方式来表示多个调用上下文。为了得到调用图，我们首先运行一次上下文无关的指针指向分析过程。12.5 节讨论过，这个分析过程同时生成了调用图。现在我们描述如何创建克隆的调用图。

调用串形成了活跃的函数调用的历史，而一个上下文就是一个调用串的表示形式。另一个看待上下文的方法是把它看作一个调用序列的摘要。这些调用的活动记录当前位于运行时刻栈中。如果栈中没有递归函数，那么这个调用串（即调用了栈中函数的位置的序列）是一个完全表示。同时它也是一个可接受的表示方式，因为只有有限多个不同的上下文。虽然上下文的个数可能是程序中函数数量的指数级。

但如果程序中存在递归函数，那么可能的调用串的数目是无穷的，我们不能用所有可能的调用串来表示不同的上下文。可以使用多个方法来限制不同的上下文的数目。比如，我们可以编

写一个描述了所有可能调用串的正则表达式，然后使用3.7节中的方法把这个表达式转化成为一个确定的有穷状态自动机。之后，各个上下文就可以使用这个自动机的状态来标识。

这里，我们将采用一个更简单的方案，它包含非递归调用的全部历史，但是把递归调用当作“难以分拆”的内容。我们首先找出程序中相互递归调用的函数的集合。这个过程很简单，因此这里不再详细讨论。考虑一个以程序中各个函数为结点的图。如果函数 p 调用了函数 q，那么图中就存在一条从结点 p 到 q 的边。这个图的强连通分量(SCC)就是相互递归调用函数的集合。下面的这个特例很常见。一个函数 p 调用了它自身，但是它不在包含了其他函数的 SCC 中，那么函数 p 本身是一个 SCC，而所有的非递归函数本身也是 SCC。如果一个 SCC 具有多个成员(即相互递归调用的情况)，或者它包含唯一一个递归成员，我们就说这个 SCC 是*非平凡的*(nontrivial)。单个非递归函数组成的 SCC 是*平凡* SCC。

前面有一个规则说任何调用串都是一个上下文，我们对这个规则做如下修改。给定一个调用串，如果下面情况成立就删除一个调用点 s 的出现：

1) s 在一个函数 p 中。

2) 函数 q 在调用点 s 处被调用(有可能 $q=p$)。

3) p 和 q 位于同一个强连通分量中(即 p 和 q 相互递归调用，或者 $p=q$ 且 p 是递归函数)。

这么做的结果是，当一个非平凡 SCC 的成员 S 被调用时，这个调用的调用点变成了上下文的一部分，但是在 S 中对同一 SCC 中其他函数的调用都不在这个上下文中。最后，当一个 S 之外的调用发生时，我们把该调用点记录为这个上下文的一部分。

例 12.26 图 12-28 中给出了五个函数的略图，图中给出了一些调用点和这些函数中的调用。检查一下这些调用就会发现，q 和 r 是相互递归的。但是 p、s 和 t 根本不会递归调用。因此，我们的上下文将是除了 s3 和 s5 之外的所有调用点的列表。函数 q 和 r 之间的递归调用就发生在 s3 和s5 处。

```
void p() {
          h: a   = new T();
         s1: T b = q(a);
         s2:       s(b);
     }

T    q(T w) {
         s3: c   = r(w);
          i: T d = new T();
         s4:       t(d);
             return d;
     }

T    r(T x) {
         s5: T e = q(x);
         s6:       s(e);
             return e;
     }

void s(T y) {
         s7: T f = t(y);
         s8:   f = t(f);
     }

T    t(T z) {
          j: T g = new T();
             return g;
     }
```

图 12-28 与一个运行实例对应的函数和调用点

让我们考虑从 p 到 t 的所有路径，也就是所有调用了 t 的上下文：

1) p 可以在 s2 处调用 s，然后 s 可以在 s7 或者 s8 处调用 t。因此，两个可能的调用串是(s2, s7)和(s2, s8)。

2) p 可以在 s1 处调用 q。然后，q 和 r 可以多次递归地调用对方。我们把这个环打开：

① 在 s4 处，t 直接被 q 调用。这个选择可以得到唯一的上下文(s1, s4)。

② 在 s6 处，r 调用 s。这里，我们可以通过在 s7 处或 s8 处的调用到达 t。这么做给出了两个新的上下文(s1, s6, s7)和(s1, s6, s8)。

因此，总共有五个不同的上下文调用了 t。请注意，所有这些上下文都省略了递归调用点 s3 和 s5。比如，上下文(s1, s4)实际上表示了对应于调用串(s1, s3, (s5, s3)^n, s4)的无穷集合，其中 $n \geqslant 0$。 □

现在我们描述一下如何得到克隆调用图。每一个被克隆的方法都使用程序中的方法 M 和一个上下文 C 来标识。在原调用图的边上加上相应的上下文就可以得到克隆后的调用图的边。请注意，在原调

用图中有一条连接调用点 S 和方法 M 的边的条件是断言 *invokes*(S, M)为真。为了增加上下文以标识克隆调用图中的例程，我们可以定义一个相应的断言 *CSinvokes*，*CSinvokes*(S, C, M, D)为真的条件是上下文 C 中的调用点 S 调用了方法 M 的上下文 D。

12.6.2 在 Datalog 规则中加入上下文信息

为了找出上下文相关的指针指向关系，我们可以直接把相同的上下文无关指针指向分析技术应用到克隆的调用图上。因为在这个克隆的调用图中的方法是用原方法和它的上下文来表示的，我们相应地修正了所有的 Datalog 规则。为简单起见，下面的规则不包括类型约束，且符号"_"表示了任何新的变量。

IDB 断言 *pts* 中必须增加一个表示上下文的参数。断言 *pts*(V, C, H)表示上下文 C 中的变量 V 可以指向堆对象 H。所有这些规则都是不解自明的，但规则 5 是一个例外。规则 5 表示，如果上下文 C 中的调用点 S 调用了上下文 D 的方法 M，那么上下文 D 的方法 M 的形式参数可能指向由上下文 C 中的相应实在参数指向的对象。

1) $pts(V,C,H)$:- "$H: T\ V\ =$ new $T()$" & $CSinvokes(H,C,\_,\_)$

2) $pts(V,C,H)$:- "$V = W$" & $pts(W,C,H)$

3) $hpts(H,F,G)$:- "$V.F = W$" & $pts(W,C,G)$ & $pts(V,C,H)$

4) $pts(V,C,H)$:- "$V = W.F$" & $pts(W,C,G)$ & $hpts(G,F,H)$

5) $pts(V,D,H)$:- $CSinvokes(S,C,M,D)$ & $formal(M,I,V)$ & $actual(S,I,W)$ & $pts(W,C,H)$

图 12-29 上下文相关的指针指向分析的 Datalog 图

12.6.3 关于相关性的更多讨论

上面我们描述的是一个上下文相关性的公式化表示。这个方法已经体现出实用性。使用下一节将要描述的一些技巧，它就能够处理很多真实的大型 Java 程序。虽然如此，这个算法还是不能处理最大型的 Java 应用。

在这个表示方法中，堆对象是通过它们的调用点来命名的，但是却不具有上下文相关性。这个简化处理可能引起一些问题。考虑一下对象工厂设计设计模式，这个设计模式中同一类型的所有对象都由同一个例程分配。当前的表示方案会使得那个类的所有对象都共享同一个名字。应对这一情况的比较容易的方法是把相应的对象创建代码进行实质上的内联处理。在处理更具一般性的情况时，我们期望提高对象命名的上下文相关性。虽然很容易向 Datalog 规则中加入对象的上下文相关信息，但是要使得相应的分析方法能够被用于大规模程序则是另一个问题了。

另一个相关性的重要形式是对象相关性。一个对象相关的技术可以区分在不同的接收对象上调用的方法。我们考虑一下这样的场景：在某个调用点所处的上下文中有一个变量可能指向同一个类的两个不同的接收对象。这两个不同的接收对象的字段可能指向不同的对象。如果不区分接收对象，在由 this 对象引用的字段之间的复制语句将产生虚假的指向关系，除非我们对不同的接收对象分别进行分析。在有些分析中，对象相关性要比上下文相关性更加有用。

```
void p() {
          h: T a = new T();
          i: T b = new T();
         c1: T c = q(a,b);
}

T    q(T x, T y) {
          j: T d = new T();
         c2:   d = q(x,d);
         c3:   d = q(d,y);
         c4:   d = r(d);
               return d;
}

T    r(T z) {
               return z;
}
```

图 12-30 练习 12.6.1 和练习 12.6.2 的代码

12.6.4 12.6 节的练习

练习 12.6.1：如果我们把本节中的方法应用到图 12-30 中的代码上，我们能够区分的上下文有哪些？

！练习 12.6.2：对图 12-30 中的代码进行上下文相关性分析。

！练习 12.6.3：按照 12.5 节中的方法，扩展本节中的 Datalog 规则，使之包含类型和子类型信息。

12.7 使用 BDD 的 Datalog 的实现

二分决策图(Binary Decision Diagram，BDD)是一个用图来表示布尔函数的方法。因为对 n 个变量有 2^{2^n} 个布尔函数，没有哪种表示方法能够很简洁地表示所有的布尔函数。但是，在实践中出现的布尔函数常常具有很多规律。因此，人们常常可以找到一个 BDD 来简洁地表示他们想要表示的布尔函数。

我们为了分析程序而开发了一些 Datalog 程序。事实表明，用这些 Datalog 程序描述的布尔函数也不例外，也可以使用 BDD 简洁地表示。BDD 方法在实践中是相当成功的，虽然我们需要通过商业 BDD 操作程序包中的一些启发式规则或技术才可以找到用以表示程序信息的简洁的 BDD。值得一提的是，它比使用传统数据库管理系统的方法具有更好的性能，因为传统数据库管理系统是为了在典型商业数据中出现的更加不规则的数据模式而设计的。

讨论经过多年开发才得到的所有 BDD 技术已经超出了本书的范围。我们将在这里介绍 BDD 的表示方法。然后，指出如何把一个关系数据表示成为 BDD。在用诸如算法 12.18 的算法来执行 Datalog 程序时需要进行某些运算。我们也指出了如何操作 BDD 来完成这些运算。最后，我们描述了如何在 BDD 中表示指数量级的上下文。这种表示法是在上下文相关性分析中成功应用 BDD 的关键。

12.7.1 二分决策图

一个 BDD 把一个布尔函数表示成为一个带根的 DAG 图。这个 DAG 的每个内部结点都用被表示函数的一个变量作为标号。在图的底部是两个叶子，一个标号为 0，另一个标号为 1。每个内部结点有两条指向子结点的边，这两条边分别称为“低边”和“高边”。低边对应于该结点对应变量取值为 0 时的情况，而高边对应于相应变量取值为 1 时的情况。

给定这些变量的一个真假赋值，我们可以从 DAG 的根开始确定函数的取值。在每个结点上，比如说标号为 x 的结点上，分别根据 x 的真假值为 0 或 1 来决定沿着相应的低边或高边前进。如果我们最后到达标号为 1 的叶结点，那么被表示的函数对于这个真假赋值取真值，否则该函数取假值。

例 12.27 在图 12-31 中，我们看到一个 BDD。稍后会看到它所表示的函数。请注意，我们已经把所有的“低”边标记为 0，所有的“高”边标记为 1。考虑对于变量 $wxyz$ 的真假赋值：$w=x=y=0$，$z=1$。从根结点开始，因为 $w=0$，我们选取低边，从而走到最左的标号为 x 的结点。因为 $x=0$，我们还是从这个结点沿着低边到达最左的标号为 y 的结点。因为 $y=0$，我们下一步移动到最左的标号为 z 的结点。现在，因为 $z=1$，我们将选择高边并最后到达标号为 1 的叶子结点。我们的结论是，这个函数相对于这个真假赋值取真值。

现在考虑真假赋值 $wxyz=0101$，也就是说 $w=y=0$，$x=z=1$。我们还是从根结点开始。因为 $w=0$，我们还是移动到最左边的标号为 x 的结点。但这一次因为 $x=1$，我们沿着高边直接跳到叶子结点 0。也就是说，我们不仅知道真假赋值 0101 使得这个函数为假，而且因为不需要查看 y 或者 z 的值，任何形如 $01yz$ 的真假赋值都会使得这个函数取值为 0。这个“短路”能力是 BDD 成为布尔函数的简洁表示方法的理由之一。 □

图 12-31 中内部结点分为多个层次——每个层中的结点都使用同一个特定的变量作为标号。虽然这并不是一个绝对的要求，但把我们的讨论范围限制在*排序 BDD* 之内会带来方便。在一个排序 BDD 中，相应的变量有一个排序 $x_1, x_2, \cdots, x_n$，并且不论何时有一条从标号为 x_i 的父结点到标号为 x_j 的子结点的边就意味着 $i<j$。我们将看到，操作排序 BDD 相对容易，并且从现在开

始我们假设所有的 BDD 都是排序的。

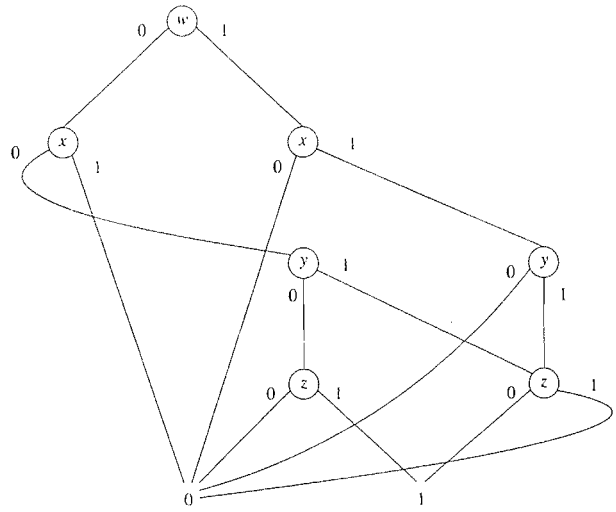


图 12-31 一个二分决策图

还需要注意的是，BDD 是有向无环图(DAG)，而不是树。不仅仅叶子结点 0 和 1 通常有很多父结点，内部结点也可能具有多个父结点。比如，在图 12-31 中最右的标号为 z 的结点有两个父结点。把得到同样结果的多个结点合并起来也是 BDD 通常比较简洁的理由之一。

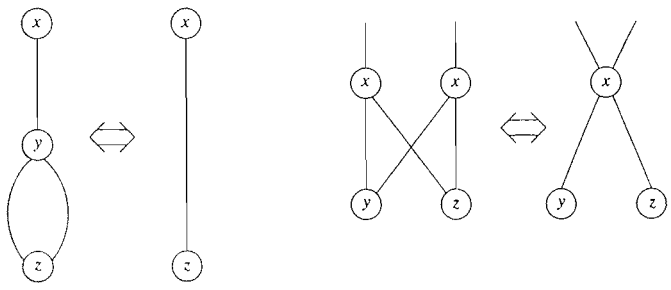
12.7.2 对 BDD 的转换

在上面的讨论中，我们提到了两个简化 BDD 的方法，它们可以使得 BDD 更加简洁：

1）*短路*：如果一个结点 N 的低边和高边都到达同一个结点 M，那么我们可以消除 N。原来进入 N 的边直接进入 M。

2）*结点合并*：如果两个结点 N 和 M 的两条低边都到达同一个结点，并且两条高边也到达同一个结点，那么我们可以把 N 和 M 合并。原来进入 N 或者 M 的边都进入合并后的结点。

也可以在相反的方向上进行这两个转换。特别地，我们可以在从 N 到 M 的边上引入一个结点。从引入结点流出的高边和低边都到达结点 M，而原来从 N 到 M 的边现在到达这个刚被引入的结点。但是请注意，新结点的标记变量必须是按照排序处于 N 和 M 之间的某一个变量。图 12-32给出了这两个转换的图示。



a) 短路　　b) 结点合并

图 12-32 BDD 的转换

12.7.3 用 BDD 表示关系

我们至今为止处理的关系都具有从“域”中取值的分量。一个关系的某个分量的域是该关系的各个元组的相应分量的可能取值的集合。比如，关系 $pts(V, H)$ 的第一个分量的域为所有程序变量，而第二个分量的域为所有对象创建语句。如果一个域具有多于 2^{n-1} 个可能取值且不多于 2^n 个可能值，那么它需要 n 个二进制位(或者说布尔变量)来表示这个域中的值。

因此，我们可以用一组布尔变量来表示关系元组的各个分量的域中的值。关系的一个元组可以被看作是这组布尔变量的真假赋值。我们可以把关系看作是这组布尔变量上的一个布尔函数。该函数对于某个真假赋值返回真值，当且仅当这个赋值表示了此关系中的一个元组。下面的例子可以说明这个想法。

例 12.28 考虑一个关系 $r(A, B)$，其中 A 和 B 的域都是 $\{a, b, c, d\}$。我们将把二进制位 00 作为 a 的编码，01 对应于 b，10 对应于 c 以及 11 对应于 d。令关系 r 的元组为：

A	B
a	b
a	c
d	c

我们使用布尔变量 wx 来对第一个分量(A)进行编码，使用变量 yz 为第二个分量(B)进行编码。那么关系 r 就变成了：

w	x	y	z
0	0	0	1
0	0	1	0
1	1	1	0

也就是说，关系 r 被转换成为一个对三个真假赋值 $wxyz = 0001$、0010 和 1110 取真值的布尔函数。请注意，这三个二进制序列恰巧就是图 12-31 中从根结点到达叶子结点 1 的路径上的标号。也就是说，如果使用上述编码方法，在那个图中的 BDD 表示了这个关系 r。 □

12.7.4 用 BDD 操作实现关系运算

现在，我们看到了如何把关系表示成 BDD。但是，要实现像算法 12.18(Datalog 程序的增量式求值)那样的算法，我们还需要能够操作 BDD 以反映相应关系上的运算。下面给出了我们要完成的主要的关系运算：

1) *初始化*：我们需要创建一个 BDD 来表示一个关系的单个元组。我们将通过合并运算把这些表示单个元组的 BDD 集成到表示大型关系的 BDD 中去。

2) *合并*：为了表示关系的合并，我们使用布尔函数的逻辑 OR 运算来表示得到的关系。这个运算不仅用来构造初始关系，也用于把具有相同头断言的多个规则的结果合并起来，还会用于把新的断言事实合并到老事实的集合中去。算法 12.18 要求实现这些运算。

3) *投影*：当我们对一个规则体求值的时候，我们需要构造出由那些使得规则体取真值的元组所蕴含的头断言的关系。从表示这个关系的 BDD 的角度来说，我们需要消除其中的一些结点，这些结点的布尔变量标号没有用来表示头关系中的分量。我们可能还需要对 BDD 中的某些变量重新命名，以使得它们和头关系分量的布尔变量相对应。

4) *连接*：为了找出令一个规则体为真的变量的赋值组合，我们需要把对应于各个子目标的关系“连接”起来。比如，假设我们有两个子目标 $r(A, B)\&s(B, C)$。这些子目标的关系的连接是满足下列

条件的三元组(a, b, c)的集合：(a, b)是r的关系中的一个元组，且(b, c)是s的关系的一个元组。我们将看到，在对BDD中的布尔变量重新命名，使得对应于两个B分量的变量同名之后，这个BDD操作和逻辑AND运算类似，而逻辑AND运算和在BDD上实现关系合并的逻辑OR运算类似。

单一元组的BDD

为了初始化一个关系，我们需要使用一种方法来为那些只对单个真假赋值取真值的函数构造BDD。假设布尔变量为x_1，x_2，…，x_n，并且这个唯一的真假赋值为$a_1a_2\cdots a_n$，其中每个a_i是0或1。相应的BDD对于每个x_i有一个结点N_i。如果$a_i=0$，那么N_i的高边直接到达叶子结点0，而低边到达结点N_{i+1}，或在$i=n$时到达叶子1。如果$a_i=1$，我们进行同样的处理，只是高边和低边顺序相反。

这个策略给出了一个BDD，它能够检查每个$x_i(i=1, 2, \cdots, n)$是否具有正确的值。一旦找到不正确的值，我们就直接跳转到叶子结点0。只有当所有变量的取值都正确时，我们才会在最后到达叶子结点1处。

作为例子，可以回到前面的图12-33b。这个BDD表示了一个当且仅当$x=y=0$(即真假赋值为00时)才取真值的函数。

合并

我们将详细地给出一个算法来计算BDD的逻辑OR，也就是这两个BDD所表示的关系的合并。

算法12.29 BDD的合并。

输入：两个排序的BDD，它们的变量集合相同，且排序也相同。

输出：一个BDD，它表示的函数是两个输入BDD所表示的布尔函数的逻辑OR。

方法：我们将描述一个合并两个BDD的递归过程。这个过程按照BDD中出现的变量集合的大小进行归纳。

归纳基础：零个变量。这两个BDD必然都是叶子结点，其标号是1或0。如果两个输入中有一个是1，那么输出就是标号为1的叶子结点；如果两个输入都是0，那么输出叶子结点的标号是0。

归纳步骤：假设两个BDD中总共出现了k个变量y_1，y_2，…，y_k。执行下列步骤：

1）如果必要，使用反向的短路转换加入一个新的根，使得两个BDD的根的标号都是y_1。

2）设两个BDD的根为N和M，令它们的低边子结点分别为N_0和M_0，它们的高边子结点分别为N_1和M_1。对分别以N_0和M_0为根的两个BDD递归地应用这个算法。同时也对分别以N_1和M_1为根的两个BDD应用这个算法。在得到的两个BDD中，第一个BDD表示的函数取真值的条件是：相应的真假赋值中$y_1=0$，并且它使得两个输入BDD中的一个或全部取真值。第二个BDD表示同样的函数，不过其中的$y_1=1$。

3）创建一个新的标号为y_1的根结点。它的低边子结点是通过递归构造得到的第一个BDD的根结点，而它的高边子结点是第二个BDD的根结点。

4）在刚刚通过合并得到的BDD中把两个标号为0的叶子结点合并，同时把两个标号为1的叶子结点合并。

5）在可能的时候应用合并和短路转换，简化得到的BDD。 □

例12.30 在图12-33a和图12-33b中有两个简单的BDD。第一个BDD表示函数x OR y，而第二个BDD表示函数

NOT x AND NOT y

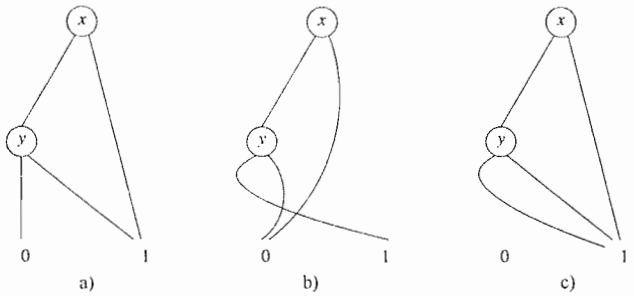


图 12-33 为逻辑 OR 构造 BDD

请注意，它们的逻辑 OR 的结果是常量函数 1，即永真函数。对这两个 BDD 应用算法 12.29 时，我们考虑两个根的低边子结点和它们的高边子结点。我们先考虑后者。

在图 12-33a 中，根的高边子结点是 1，而在图 12-33b 中的相应子结点是 0。因为这两个子结点都在叶子层次上，所以不需要在每条边上插入标号为 y 的结点，尽管我们这么做会得到同样的结果。结点 0 和 1 的合并是算法中归纳基础的情况，合并后生成一个标号为 1 的叶子结点。这个叶子结点将成为新的根结点的高边结点。

图 12-33a 和图 12-33b 中的根的低边结点的标号都是 y，因此我们递归地计算它们的合并 BDD。这两个结点的低边子结点的标号分别为 0 和 1，因此它们的低边子结点的合并是标号为 1 的叶子结点。当我们加入新的根结点 x 后，我们得到图 12-33c 中的 BDD。

我们还没有完成，因为图 12-33c 还可以进一步简化。标号为 y 的结点的两个子结点都是结点 1，因此我们可以把结点 y 删除，并把 1 当作根结点的低边子结点。现在，根结点的两个子结点都是叶子结点 1，因此我们可以消除根结点。也就是说，表示这个合并操作结果的最简单的 BDD 就是叶子 1 本身。 □

12.7.5 在指针指向分析中使用 BDD

要使上下文无关的指针指向分析能够正确工作已经很不容易了。BDD 变量的排序可以极大地影响表示的大小。要得到一个能够使得分析很快结束的 BDD 变量排序，需要各种各样的考虑，也包括尝试和犯错。

使得上下文相关的指针指向分析能够有效执行是一件更加困难的事情，因为程序中有指数量级的上下文。特别是，如果我们随意使用编号来表示一个调用图中的上下文，那么我们甚至不能处理很小的 Java 程序。按照适当的方式对上下文进行编号是很重要的，它可以使指针指向分析中的编码变得非常紧凑。同一方法的调用路径相似的两个上下文之间有很多共同点，因此对一个方法的 n 个上下文连续编码是比较合适的。类似地，因为同一个调用点上的调用者 - 被调用者对之间具有很多相似之处，所以我们希望对上下文进行编码的方式可以使得一个调用点上的每个调用者 - 被调用者对之间的编码的数值总是相差一个常数。

即使有了一个很合理的对调用上下文编码的方案，但高效地分析大型 Java 程序仍然困难重重。人们发现，主动机器学习有助于获取较好的变量排序，使得算法能够高效地处理大型应用。

12.7.6 12.7 节的练习

练习 12.7.1：使用例子 12.28 中的符号编码方式，生成一个 BDD 来表示由元组 (b, b)、(c, a) 和 (b, a) 组成的关系。你可以用任意方式对布尔变量进行排序，以获取最简洁的 BDD。

！**练习 12.7.2**：如果用最简洁的 BDD 来表示 n 个变量上的异或函数，那么这个 BDD 中有多少个结点？把它表示成为一个关于 n 的函数。n 个变量上的异或函数是说如果这 n 个变量中有奇

数个变量为真，那么这个函数就为真；如果有偶数个变量为真，那么函数值为假。

练习 12.7.3：修改算法 12.29，使之能够生成两个 BDD 的交集(即逻辑 AND)。

!! **练习 12.7.4**：找出在表示关系的排序 BDD 之上的进行下列关系运算的算法：

1）通过投影消除某些布尔变量。也就是说，运算得到的 BDD 所表示的函数如下：给定一个被保留变量的真假赋值 α，如果存在被消除变量的任何一个真假赋值，它和 α 一起使得原函数取真值，那么结果函数的取值也是真。

2）把两个关系 r 和 s 连接起来，只要一个来自 r 的元组和一个来自 s 的元组在 r 和 s 的共同属性上具有相同的值，这两个元组就组合起来成为新关系的一个元组。实际上，只需要考虑下面的情况就足够了：这两个关系都只有两个分量，且它们有一个公共分量。也就是说，这两个关系是 $r(A, B)$ 和 $s(B, C)$。

12.8 第 12 章总结

- *过程间分析*：对跨越过程边界的信息进行跟踪的数据流分析称为过程间分析。很多分析技术，比如指针指向分析，只有当它是过程间分析的时候才可以完成有意义的分析工作。
- *调用点*：程序中调用其他过程的程序点称为*调用点*。在一个调用点上被调用的过程可能是显然的。但是，如果这个调用是通过指针间接进行的，或者它调用的是具有多个实现的虚方法，那么被调用的过程也可能是不明确的。
- *调用图*：一个程序的调用图是一个二分图，图的结点分为对应于调用点的结点和对应于过程的结点。如果一个过程在一个调用点上被调用，那么就有一条从这个调用点结点到这个过程结点的边。
- *内联*：只要一个程序中没有递归，原则上我们可以把所有的过程调用替换为过程代码的拷贝，并对得到的程序使用过程内分析技术。从效果上看，这个分析是过程间分析。
- *控制流相关性和上下文相关性*：如果一个数据流分析得到的事实和程序中的位置相关，那么它就是控制流相关的。如果一个数据流分析得到的事实和过程调用的历史相关，那么它就是上下文相关的。一个数据流分析可以是控制流相关的、上下文相关的、两者都相关或者都不相关。
- *基于克隆的上下文相关分析*：从原则上讲，一旦我们建立了过程调用的不同上下文，就可以想象对于每个上下文都有一个该过程的克隆。按照这种方法，一个上下文无关分析技术可以用来进行上下文相关分析。
- *基于摘要的上下文相关分析*：另一个过程间分析的方法，扩展原来为过程内分析而设计的基于区域的分析技术。每个过程有一个传递函数，并且在每一个调用该过程的地方它都被当作一个区域处理。
- *过程间分析技术的应用*：需要过程间分析技术的重要应用之一是检测软件的安全漏洞。这些漏洞的常见特性是一个过程从某个不可信的输入源读取数据，而另一个过程以可能被利用的方式使用这个输入。
- Datalog：Datalog 语言是 if-then 规则的简单表示方式，它可以用于在高层次上描述数据流分析。一组 Datalog 规则(或者说 Datalog 程序)可以使用多个标准算法中的任意一个算法进行求值。
- Datalog *规则*：一个 Datalog 规则由一个规则体(前提)和一个规则头(结果)组成。规则体是一个或多个原子，而规则头则是一个原子。原子就是作用于一组参数的断言，这些参数的值可以是变量或常量。规则体的多个原子通过逻辑 AND 连接，而规则体中的原子可

能是断言的否定形式。

- IDB 和 EDB 断言：一个 Datalog 程序中的 EDB 断言的真值事实在事先给出。在一个数据流分析中，这些断言对应于那些可以从被分析代码中获取的事实。IDB 断言本身是通过规则定义的。在一个数据流分析中，它们对应于我们想从被分析代码中抽取的信息。
- Datalog 程序的求值：我们应用规则的方法是把规则中的变量替换为一些能够使该规则体取真值的常量。当我们做了这样的替换后，就可以推断将规则头中的变量进行相同替换后得到的断言也为真。这个操作不断重复，直到不能推导出更多的事实为止。
- Datalog 程序的增量求值：通过增量求值的方法可以改进 Datalog 程序的求值效率。我们将进行多轮求值。在每一轮中，我们只考虑如下的变量到常量的替换方法：它使得规则体中至少有一个原子是刚刚在上一轮中被发现的事实。
- Java 指针分析：我们可以用一个框架对 Java 中的指针分析建模。在这个框架中，有一些指向堆对象的引用变量，而这些堆对象中又有一些字段可以指向其他堆对象。可以用一个 Datalog 程序写出一个上下文无关的指针分析方法。这个分析可以推导出两种事实：一个变量可能指向一个堆对象，以及一个堆对象的字段可能指向另一个堆对象。
- 使用类型信息改进指针分析：引用变量所指向的堆对象的类型要么和变量类型相同，要么是变量类型的子类型。如果我们能够利用这个事实，我们就可以得到更加精确的指针分析结果。
- 过程间指针分析：为了进行过程间分析，我们必须增加一些规则来反映参数是如何传递的，返回值是如何被赋给变量的。这些规则实质上和把一个引用变量复制到另一个引用变量的规则相同。
- 寻找调用图：因为 Java 具有虚方法，过程间分析要求我们首先界定有哪些过程可能在一个给定调用点上被调用。找出哪里可以调用哪些程序的限制的基本方法是分析对象的类型，并利用下面的事实：一个虚方法调用所指向的实际方法必须属于适当的类。
- 上下文相关分析：当过程具有递归特性时，我们必须把调用串中所包含的信息浓缩到有限多个上下文中。做这件事的有效方法之一是从调用串中删除某个过程调用与之相互递归调用的另一个过程(可能是调用者本身)的调用点。使用这样的表示方式，我们可以修改过程内指针分析的规则，使断言中包含上下文信息。这个方法模拟了基于克隆的分析。
- 二分决策图：BDD 是一种使用带根的 DAG 表示布尔函数的简洁方法。内部结点对应于布尔变量，并且有两个子结点，即低子结点(表示 0 值)和高子结点(表示 1 值)。图中有标号分别为 0 和 1 的两个叶子结点。一个真假赋值使得被表示函数取真值当且仅当从图的根结点有一条如下的路径到达叶子结点 1。这条路径从根结点开始，如果一个结点上的变量取值为 0，那么我们就走到低子结点，否则走到高子结点。
- BDD 和关系：一个 BDD 可以作为 Datalog 程序中的断言的简洁表示方法。常量被编码为一组布尔变量的真假赋值，BDD 表示的函数为真当且仅当它的布尔变量表示了使这个断言取真值的事实。
- 使用 BDD 实现数据流分析：任何可以被表示为 Datalog 规则的数据流分析都可以使用 BDD 上的操作来实现。这些 BDD 表示了规则所涉及的断言。这个表示方法经常会得到一个比其他已知方法更加高效的实现。

12.9 第 12 章参考文献

过程间分析的一些基本概念可以在[1, 6, 7, 21]中找到。Callahan 等[11]描述了一个过程间常量传播算法。

Steensgaard[22]发布了第一个可伸缩的指针别名分析技术。这个技术是上下文无关、控制流无关的且基于等价关系。基于包含的指针指向分析的一个上下文无关版本首先由 Andersen[2]提出。之后，Heintze 和 Tardieu[15]描述了实现这个分析的高效算法。Fähndrich、Rehof 和 Das[14]给出了一个上下文相关、控制流无关且基于等价的分析技术，这个技术可以处理很大规模的程序，比如 gcc。在之前针对上下文相关的、基于包含关系的指针指向分析的各种尝试中，值得一提的是 Emami、Ghiya 和 Hendren[13]的工作。他们的工作是一个基于克隆的上下文相关、控制流相关、基于包含的指针指向分析算法。

二分决策图(BDD)首先出现在 Bryant[9]的工作中。它们第一次用于数据流分析是由 Berndl[4]等人提出的。DBB 在无关性指针分析中的应用由 Zhu[25]和 Berndl 等人[8]报告。Whaley 和 Lam[24]描述了第一个上下文相关、控制流无关、基于包含关系的算法。这个算法是第一个被证明可以用于实际应用的算法。这篇文章描述了一个称为 bddbddb 的工具，它可以把使用 Datalog 描述的分析自动地转化成为 BDD 代码。对象相关性首先由 Milanova、Rountev 和 Ryder[18]提出。

对 Datalog 的讨论见 Ullman 和 Widom[23]。在 Lam 等人的工作[16]中也可以看到有关把数据流分析和 Datalog 联系起来的讨论。

Metal 代码检查工具在 Engler 等的著作[12]中描述，而 PREfix 检查程序由 Bush、Pincus 和 Sielaff[10]创建。Ball 和 Rajamani[4]开发了一个名为 SLAM 的程序分析引擎，它使用了模型检查和符号执行技术来模拟一个系统的所有可能行为。Ball 等[5]已经基于 SLAM 建立了一个被称为 SDV 的静态分析工具，并通过把 BDD 应用到模型检查中，在 C 语言的设备驱动程序中寻找 API 的使用错误。

Livshits 和 Lam[17]描述了如何使用上下文相关的指针指向分析来寻找 Java Web 应用中的 SQL 漏洞。Ruwase 和 Lam[20]描述了如何跟踪数组长度并自动加入动态边界检查代码。Rinard 等[19]描述了如何动态扩展数组大小来应对溢出缓冲区的内容。Avots 等[3]把上下文相关的 Java 指针指向分析扩展到 C 语言中，并说明了如何使用它来降低动态检测缓冲区溢出的开销。

1. Allen, F. E., "Interprocedural data flow analysis," *Proc. IFIP Congress 1974*, pp. 398–402, North Holland, Amsterdam, 1974.

2. Andersen, L., *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, Univ. of Copenhagen, Denmark, 1994.

3. Avots, D., M. Dalton, V. B. Livshits, and M. S. Lam, "Improving software security with a C pointer analysis," *ICSE 2005: Proc. 27th International Conference on Software Engineering*, pp. 332–341.

4. Ball, T. and S. K. Rajamani, "A symbolic model checker for boolean programs," *Proc. SPIN 2000 Workshop on Model Checking of Software*, pp. 113–130.

5. Ball, T., E. Bounimova, B. Cook, V. Levin, J. Lichtenber, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *EuroSys* (2006), pp. 73–85.

6. Banning, J. P., "An efficient way to find the side effects of procedural calls and the aliases of variables," *Proc. Sixth Annual Symposium on Principles of Programming Languages* (1979), pp. 29–41.

7. Barth, J. M., "A practical interprocedural data flow analysis algorithm," *Comm. ACM* **21**:9 (1978), pp. 724–736.

8. Berndl, M., O. Lohtak, F. Qian, L. Hendren, and N. Umanee, "Points-

to analysis using BDD's," *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pp. 103–114.

9. Bryant, R. E., "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Computers* **C-35**:8 (1986), pp. 677–691.

10. Bush, W. R., J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software–Practice and Experience*, **30**:7 (2000), pp. 775–802.

11. Callahan, D., K. D. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," *Proc. SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices*, **21**:7 (1986), pp. 152–161.

12. Engler, D., B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," *Proc. Sixth USENIX Conference on Operating Systems Design and Implementation* (2000). pp. 1–16.

13. Emami, M., R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 224–256.

14. Fähndrich, M., J. Rehof, and M. Das, "Scalable context-sensitive flow analysis using instantiation constraints," *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (2000), pp. 253–263.

15. Heintze, N. and O. Tardieu, ""Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (2001), pp. 254–263.

16. Lam, M. S., J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," *Proc. 2005 ACM Symposium on Principles of Database Systems*, pp. 1–12.

17. Livshits, V. B. and M. S. Lam, "Finding security vulnerabilities in Java applications using static analysis" *Proc. 14th USENIX Security Symposium* (2005), pp. 271–286.

18. Milanova, A., A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java" *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1–11.

19. Rinard, M., C. Cadar, D. Dumitran, D. Roy, and T. Leu, "A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors)," *Proc. 2004 Annual Computer Security Applications Conference*, pp. 82–90.

20. Ruwase, O. and M. S. Lam, "A practical dynamic buffer overflow detector," *Proc. 11th Annual Network and Distributed System Security Symposium* (2004), pp. 159–169.

21. Sharir, M. and A. Pnueli, "Two approaches to interprocedural data flow analysis," in S. Muchnick and N. Jones (eds.) *Program Flow Analysis: Theory and Applications*, Chapter 7, pp. 189–234. Prentice-Hall, Upper Saddle River NJ, 1981.

22. Steensgaard, B., "Points-to analysis in linear time," *Twenty-Third ACM Symposium on Principles of Programming Languages* (1996).

23. Ullman, J. D. and J. Widom, *A First Course in Database Systems*, Prentice-Hall, Upper Saddle River NJ, 2002.

24. Whaley, J. and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 131–144.

25. Zhu, J., "Symbolic Pointer Analysis," *"Proc. International Conference in Computer-Aided Design* (2002), pp. 150–157.