第 6 章

过程间分析与优化

本章讨论过程间分析和和优化。首先，我们讨论调用图和过程间控制流图，它们是支撑过程间分析和优化的重要数据结构。然后，我们给出上下文相关性的概念，并给出一种上下文无关的程序分析方法，以及调用串和函数式方法等两种上下文相关的程序分析分析方法。接着，我们以过程间常量传播算法为实例，讨论过程间优化的一般方法。

6.1 引言

前文讨论的程序分析和编译器优化，只在单一函数粒度上进行，称为**过程内分析和优化**（Intraprocedural Analysis and Optimizations）。过程内分析每次考虑一个函数，因此相对高效。但过程内分析较为保守，由于缺乏其它函数的必要信息，因此过程内分析不得不假设被调用的函数可能会具有任意的行为，如改变全局变量的值，进行内存的分配释放，进行输入输出，或抛出异常等。因此过程内分析虽然较为简单，但缺乏跨过程间数据流和控制流信息等上下文相关信息，导致分析的精度有限。进而，编译器无法充分利用上下文相关信息进

行深层优化，难以达到最佳优化效果。

过程间分析和优化（Interprocedural Analysis and Optimization）是对过程内分析和优化的重要改进。它以全程序中的多个函数为粒度，尝试获取调用者与被调用者间的过程间信息，从而提升程序分析和优化的效果。与过程内程序分析和优化比较，过程间程序分析和优化具有显著的优势。它不但能获得更精确的分析结果，还能优化程序性能、提升代码质量、发现潜在的程序安全问题等。

首先，过程间分析能够更有效消除程序中的冗余计算，从而提高程序执行性能。例如，对特定函数的调用，如果实际参数总是为常量，则过程间分析可以将这些常量由调用者直接传播到被调用者中，从而发现潜在的过程间常量传播的机会。比如，对如下程序，编译器通过过程间分析，可以得知对 `bar` 函数的调用的实际参数总是常量 42。因此，编译器可以将实际参数 42 作为常量传播到被调用者 `bar` 中，进而发现被调用函数 `bar` 总是返回常量 43，最终可将函数 `foo` 中变量 x 和 y 赋值为常量 43。

```
1 int foo(){
2   int x = bar(42);
3   int y = bar(42);
4 }
5 int bar(int a){
6   return a+1;
7 }
```

与过程间分析显著不同，过程内分析由于缺乏函数间的全局信息，从而不得不假定被调用函数 `bar` 可能具有任意行为，最终变量 x 和 y 的值都是顶元 $\top$。

其次，对带有虚方法特性的面向对象编程语言，过程间

分析能够更好的优化虚方法调用。虚方法调用需要运行时方法动态绑定，会带来额外的运行时性能开销。为了减小这种开销，编译器可以通过过程间分析，推断出某些调用的实际目标对象，从而将动态绑定优化为静态调用，甚至直接内联被调用的方法。例如，如下 Java 代码的第 5 行调用了 A 类对象 obj 上的 f 虚方法，而如果过程间分析能够断定调用函数 bar 的实际参数就是 A 的对象（如第 2 行），则可以将第 5 行的虚函数调用改为对 A.f() 方法的直接调用。

```
1 int foo(){
2   bar(new A());
3 }
4 int bar(A obj){
5   obj.f();
6 }
```

这种优化对对象方法的频繁调用场景尤为关键，能够显著减少运行时的性能损耗。

此外，过程间分析可以显著提升指针和别名分析的准确度。在使用指针特性的程序中，指针间的别名关系（即是否指向同一内存位置），会影响程序分析和编译优化的效果。通过别名分析，编译器可以确定两个指针变量是否可能指向同一内存位置，这对于提高程序分析的准确性至关重要。例如，对如下代码，过程间分析能够判定指针变量 p 和 q 肯定不是别名，因此程序第 4 和 5 行的内存释放函数 free 不会导致指针的双重释放错误。而过程内分析由于无法得到函数 bar 的准确信息，只能保守假设指针 p 和 q 可能是别名，从而给出错误的信息提示。

```
1 int foo(){
2   void *p = bar();
3   void *q = bar();
```

```
4   free(p);
5   free(q);
6 }
7 void *bar(){
8   return malloc(4);
9 }
```

准确的别名信息不仅有助于实现常量传播、死代码消除等编译优化，还有助于检查程序的安全性和正确性，避免潜在的资源泄漏或数据竞争等安全漏洞。

不仅如此，过程间分析还有助于发现程序的并行化机会。通过函数间的数据依赖分析，编译器可以识别那些不依赖前后结果的操作，从而自动实现任务并行化。例如，考虑如下简化的类 MapReduce 的程序，函数 `foo` 负责将数据 `arr` 分块，并调用函数 `bar` 分块处理。如果过程间分析能够确定对函数 `bar` 的两个调用之间没有共享状态或数据依赖，就可以将这两次对函数 `bar` 的调用并行化，将其分别运行在不同的线程或核心上，提升程序在多核处理器上的运行效率。

```
1  int arr[N];
2  int foo(){
3    bar(0);
4    bar(1);
5  }
6  void bar(int index){
7    while(index < N){
8      arr[index] *= 2;
9      index += 2;
10   }
11 }
```

最后，过程间分析对于提升程序的安全性也有重要作用。例如，过程间分析可以通过追踪变量的过程间数据流动，分析 SQL 数据的特征，从而检测可能的 SQL 注入攻击。例如，

过程间通过分析传给函数 bar 的的参数 s 的具体字符串值，可以更精确的检测 SQL 注入攻击及其具体类型，并给防御提供指导。

```
int foo(){
  char *sql = bar("1';␣DELETE␣FROM␣users;␣'");
}
char *bar(char *s){
  return "SELECT␣*␣FROM␣users␣WHERE␣name␣=␣'" + s + "'";
}
```

此外，对于支持数组特性的程序，过程间分析可以通过分析作为数组下标的数据范围，确保数组访问在合法的边界内，从而有效检测并避免缓冲区溢出等危险操作。很多现代编译器支持非常强大的安全检测和增强功能，为提高软件安全性和可靠性提供了重要保障。

6.2 调用图

在进行过程间分析之前，我们需要准确表示程序中各个函数之间的调用关系。为此，我们引入一种重要的数据结构—**调用图**（Call Graph）。调用图是表达函数调用关系的一种有向图 $G = (V, E)$，其中节点集合 $V = \{v_1, \ldots, v_n\}$ 中的每个节点 v_i，$1 \leq i \leq n$，都表示程序中的一个函数（或过程）。而边集合 $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ 中的有向边 (v_i, v_j)，表示程序中存在从函数 v_i 到函数 v_j 的调用。

许多语言中（如 Fortran），函数调用是静态和直接的，即调用点的函数名和被调用的目标函数名字相同，这意味着每个函数调用点，只会有一条边指向唯一的被调用函数。在这种情况下，我们可以通过静态解析代码结构，为每个调用点

算法 6.1 调用图构建算法

输入： 整个程序 P

输出： 调用图 $G = (V, E)$

```
1: procedure constructCallGraph(P)
2:     V = {}, E = {}                    ▷ V 是函数集合，E 是函数调用边的集合
3:     for each function f ∈ P do
4:         addNode(V, f)
5:     for each function f ∈ P do
6:         for each statement s ∈ f do
7:             if s is a call statement h(...) then
8:                 addEdge(E, (f, h))
       return (V, E)
```

明确地确定要调用的目标函数，并直接生成调用图。

算法6.1给出了构建调用图的关键步骤。算法接受整个程序 P 作为输入，构建并返回其调用图 $G = (V, E)$。算法首先将所有函数 F 加入到调用图的节点集合 V 中，然后遍历程序中的每个函数 F，识别 F 中的调用指令（即形如 $H(\ldots)$），则添加代表函数 F 到 E 的调用的有向边 (F, H) 到调用图 G 的有向边集合 E 中。

由于算法需要扫描程序的每条语句一次，因此，对于有 N 条语句的程序来说，算法具有线性量级的最坏运行时间复杂度 $O(N)$。

例如，对图6.1 中给出的类 C 语言的程序，算法6.1计算得到的调用图如右侧所示。

如果程序语言支持对函数的间接调用（如函数指针、虚函数等），则构建程序的调用图变得更加困难。主要的挑战在于函数间接调用使得编译器难以静态确定调用目标。因此，在这种情况下，分析算法往往要保守静态估计所有可能的调用

```
1  int fun1(int x){
2    return (x < 10)? (x+1): x;
3  }
4  int fun2(int y){
5    return fun1(y);
6  }
7  void main(){
8    fun1(10);
9    fun2(5);
10 }
```

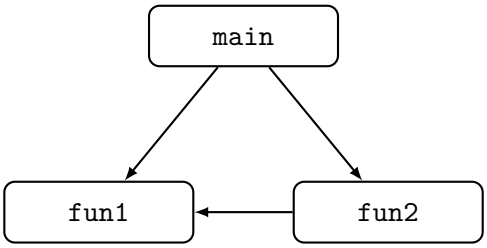


图 6.1: 示例程序及其调用图。

目标，在最坏情况下，甚至可能包括程序中全部可能的目标函数。

例如，对图6.2中的程序，在不对函数指针 `fp` 做指针分析的情况下，我们不得不保守的认为 `fp` 在每个函数调用点（即第 3、7 和 11 行）都分别可能指向任意函数，从而为其构建调用图（如上右图所示）。注意到调用图中的每个函数，都有指向其它函数（包括自身）的有向边。

为了构建更加精确的调用图，我们可以采用一些有效的策略或算法。首先，我们可以考虑函数类型信息。对于强类型语言，我们可以认为函数指针只会指向同类型的目标函数，这往往会有效减小可能的目标函数集合。例如，上面示例程序中的函数指针 `fp` 的类型为 $int \rightarrow int$，该类型与函数`fun1`和`fun2` 的类型一致，而与函数`main`的类型 $void \rightarrow void$ 不同，因此函数指针可能指向函数集合 $\{fun1, fun2\}$，而不会指向函数 `main`。基于此，算法可以构建更精确的调用图（如图6.3a所示，其中不含指向函数 `main` 的有向边）。

其次，为了构建更精确的调用图，我们可以在构建调用图

```
1  int (*fp)(int);
2  int fun1(int x){
3    return (x<10)? (*fp)(x+1): x;
4  }
5  int fun2(int y){
6    fp = &fun1;
7    return (*fp)(y);
8  }
9  void main(){
10   fp = &fun2;
11   (*fp)(5);
12 }
```

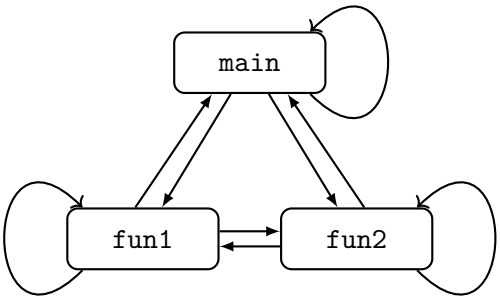
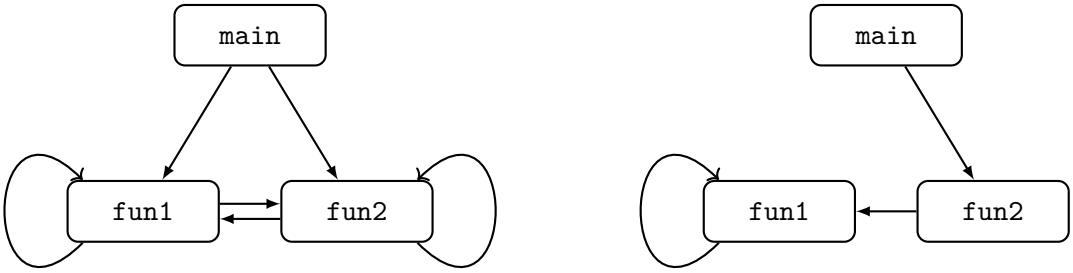


图 6.2: 示例程序及其调用图。



(a) 进行类型分析后得到的调用图。 (b) 进行指针分析后得到的调用图。

图 6.3: 对示例程序类型分析和指针分析后可以得到更精确的调用图。

之前对程序进行指针分析，以准确推断程序中指针的可能指向目标。我们将在后续章节深入讨论指针分析，这里仅针对上述示例进行自足的讨论。上述程序中的`main`函数中，函数指针`fp`在被调用之前被显式赋值为`fun2`。因此，在该调用点，函数`pf`的指向性是唯一的，即它只能指向`fun2`。类似地，在函数`fun2`中，函数指针`fp`只能指向函数`fun1`。而在函数`fun1`中，函数指针`pf`指向`fun1`自身。根据这些指针指向信息，我们可以构建更精确的调用图（如图 6.3b所示）。

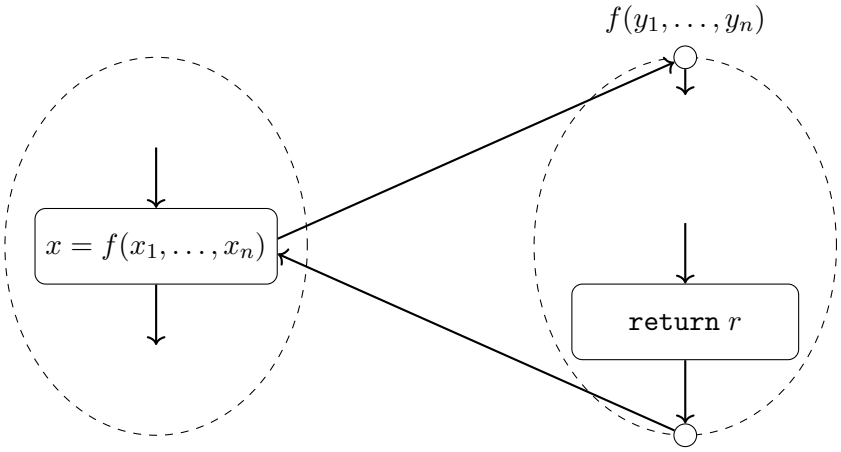


图 6.4: 过程间控制流图。

6.3 过程间控制流图

在过程内程序分析和优化中，我们用控制流图来刻画单个函数内部的控制流结构。然而，控制流图无法刻画函数间的调用/返回关系。为此，我们引入**过程间控制流图**（Interprocedural Control Flow Graph，ICFG）。过程间控制流图以控制流图为基础，并添加跨函数的调用/返回关系，从而完整地描述整个程序的控制流结构。基于过程间控制流图，我们可以将针对单个函数的分析和优化算法，扩展到整个程序，实现过程间粒度的综合分析和优化。

构建程序的过程间控制流图需要两个主要步骤。首先，我们为每个函数分别构建（过程内）控制流图。在这个过程中，我们可以对函数进行处理使其满足特定的规范，以方便程序分析和优化的实施。例如，我们可以确保每个函数的控制流图都有唯一的入口节点和出口节点。并且，为方便引用被调用函数的返回值，我们总是将该返回值命名为一个特殊的 `r` 变量，并将函数原本的返回语句 `return` y 改写为 `r` $= y$; `return r`。

构建过程间控制流图的第二步，是将上一步构建的这些

算法 6.2 过程间控制流图构建算法

输入： 整个程序 P

输出： 过程间控制流图 $G = (V, E)$

```
1:  procedure constructICFG(P)
2:      V = {}, E = {}                ▷ V 是函数集合；E 是函数间的调用/返回边
3:      g = constructCallGraph(P)
4:      c = constructCfg(P)
5:      for each function f ∈ P do
6:          addNode(V, f)
7:      for each function f ∈ P do
8:          for each statement s ∈ f do
9:              if s is a call statement h(...) then
10:                 for edge (h, t) ∈ g do
11:                     addEdge(E, (h, t))
12:                     addEdge(E, (t, h))
        return (V, E)
```

函数内的控制流图连接起来，形成过程间控制流图的整体结构。为此，我们要额外添加两种跨函数的边，分别是调用边和返回边（如图6.4所示）。调用边是从调用点 $x = f(x_1, \ldots, x_n)$ 指向被调用函数 f 入口节点的有向边，表示函数调用时的控制流转移。而返回边则从被调用函数 f 的出口节点指向调用语句 $x = f(x_1, \ldots, x_n)$ 返回点的有向边，表示函数调用返回时的控制流转移。调用边和返回边的信息则依赖于调用图所给出的函数调用结构。需要注意的是，从技术上严格来讲，返回边实际上指向调用语句 $x = f(x_1, \ldots, x_n)$ 赋值号 $=$ 左侧的部分 $x =$，即完成对变量 x 的赋值操作。在不引起混淆的情况下，我们总是简单的说返回边指向调用语句，这不影响对过程间分析和优化的相关讨论。

算法6.2给出了过程间控制流图构建的主要步骤。算法接受程序 P 作为输入，构建并输出程序的过程间控制流图。算

法首先构建程序的调用图 g 和每个函数的控制流图 c，并把所有的函数 f 都加入到节点集合 V 中。接着，算法依次扫描每个函数 f 中的语句 s，如果 s 是一条调用语句 $h(\ldots)$ 的话，则从调用图 g 中找到实际调用目标 t，并把调用边 (h,t) 和和返回边 (t,h) 加入到边集合 E 中。

除去对调用图和控制流图算的的调用外，由于算法需要扫描程序的每条语句一次，因此，对于有 N 条语句的程序来说，算法具有线性量级的最坏运行时间复杂度 $O(N)$。

6.4 过程间数据流分析

基于调用图和过程间控制流图等数据结构，我们可以将过程内的数据流分析自然地扩展到过程间，实现**过程间数据流分析**（Interprocedural data flow analysis）。在过程间分析中，我们需要在过程间控制流图上建立数据流方程，来建模信息流在函数调用和返回时的流动情况。再通过求解数据流方程，来得到每个程序点的数据流信息。

假定函数调用点

$$u : x = f(x_1, \ldots, x_n)$$

前后的数据流分别为 $In[u]$ 和 $Out[u]$（如图6.5 所示），则我们需要将调用节点 u 所确定的参数信息

$$[In[u][x_1], \ldots, In[u][x_n]],$$

沿着调用边，传递给被调用函数 f 的形参 $y_1, \ldots, y_n$。那么，对于被调用函数 $f(y_1, \ldots, y_n)$ 的入口节点 v，其数据流信息为

$$s[u] = [y_1 \mapsto In[u][x_1], \ldots, y_n \mapsto In[u][x_n]]。$$

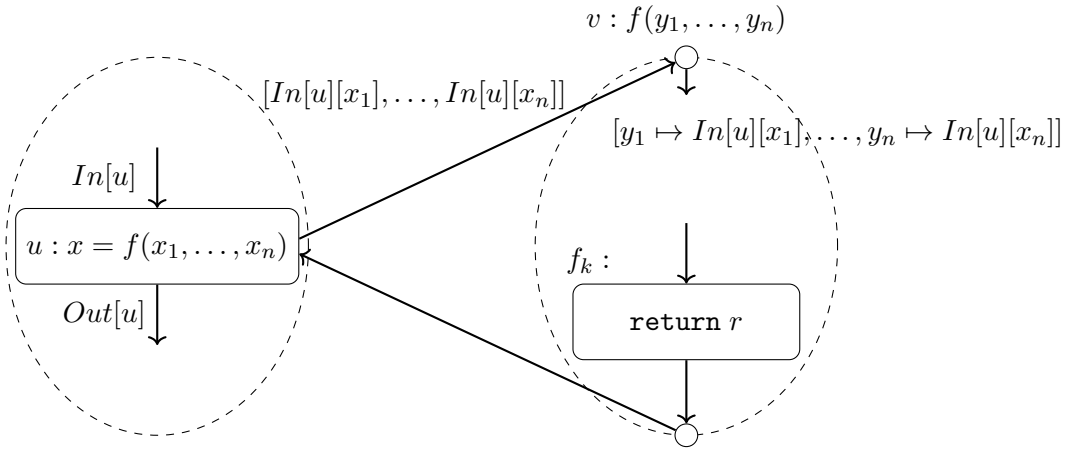


图 6.5: 数据在过程间控制流图上的流动过程。

由于函数 f 可能有多个调用点，因此，对入口节点 v 的 $In[v]$，我们有数据流方程

$$In[v] = \bigsqcup_{w \in pred(v)} s[w], \tag{6.1}$$

亦即取所有调用点 w 信息 $s[w]$ 的最小上界。

对函数调用节点 u 后的 $Out[u]$，我们有数据流方程

$$Out[u] = In[u][x \mapsto Out[f](r)], \tag{6.2}$$

其中 f_k 是被调用函数 f 的出口块（以下，在不引起混淆的情况下，我们也经常将其简写为 f）。我们请读者特别注意到数据流方程 6.2，与其过程内分析相对应方程

$$Out[u] = In[u][x \mapsto \top]$$

间的区别。

接下来，我们以图6.6 中的程序为例，通过对其进行过程间零值分析，说明如何在过程间控制流图上进行数据流分析。为清晰起见，我们分别用一对实线和虚线表示两个函数的调用和返回（亦即第 1、2 两行）。我们用 $In[i]$ 和 $Out[i], 1 \leq i \leq 6$，

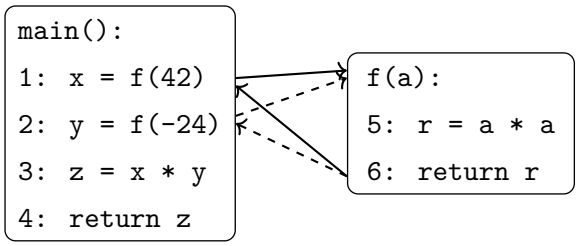


图 6.6: 对示例程序进行过程间零值分析的分析过程。

分别代表每个语句 i 前后的抽象状态，并用 $In[h]$ 和 $Out[h]$ 分别代表任意函数 h 入口和出口的抽象状态。

在程序初始化阶段，算法分别将每个函数的的参数和返回值的抽象值都初始化为底元 $\perp$，代表这些变量还为被赋值过。算法将入口函数 `main` 的参数（如果有的话），都初始化为顶元 $\top$，代表它们可能具有任意的输入值。

算法可以任意顺序分析程序中的函数。这里，我们先分析 `main`，为此，算法从抽象状态

$$\sigma = [x \mapsto \top, y \mapsto \top, z \mapsto \top]$$

开始分析函数 `main` 并将其做为第 1 条语句的 $In[1]$。由于第 1 条是函数调用语句 $x = f(42)$，则算法将实参 42 的抽象状态 $\sigma[42] = Z^{-0}$ 传播给被调用的函数 f 形参 a，将函数 f 的入口状态更新为

$$In[f] = [a \mapsto Z^{-0}]。$$

接着，算法由方程6.2计算语句 1 的输出状态

$$\begin{aligned} Out[1] &= In[1][x \mapsto Out[f][r]] \\ &= [x \mapsto \top, y \mapsto \top, z \mapsto \top][x \mapsto \perp] \\ &= [x \mapsto \perp, y \mapsto \top, z \mapsto \top]。 \end{aligned}$$

同理，算法继续分析语句 2，在传播完被调用函数 f 的参

数 a 的抽象状态 $[a \mapsto Z^{-0}]$ 后，得到语句 2 的输出状态

$$Out[2] = [x \mapsto \bot, y \mapsto \bot, z \mapsto \top]。$$

以此类推，算法最终得到函数 `main` 的出口状态为

$$Out[main] = [x \mapsto \bot, y \mapsto \bot, z \mapsto \bot]。$$

继续分析函数 `main` 会发现，其入口和出口状态都不在发生变化，到达了不动点。

算法继续分析函数 `f`。此时需要将函数 f 的入口状态

$$In[f] = [a \mapsto Z^{-0}]$$

作为初始状态依次分析所有语句，我们把具体过程留给读者作为练习。当函数的分析到达不动点时，其出口状态为

$$Out[f] = [r \mapsto Z^{-0}]。$$

至此，算法已经完成了对所有函数的第一轮分析。由于函数 f 的出口状态由 $[r \mapsto \bot]$ 变为 $[r \mapsto Z^{-0}]$，算法重新开始对程序中所有函数的新一轮分析，一直进行到所有函数的入口和出口状态都不再变化为止。我们请读者自行完成剩余的分析过程，并断定当分析到达不动点时，一定有 $z \neq 0$ 成立。

算法6.3给出了过程间零值分析的关键步骤。算法接受整个程序 P 作为输入，分析得到程序 P 中每个语法结构的抽象状态 $In[]$、$Out[]$。算法首先对每个函数 f 的入口和出口状态进行初始化。然后用迭代算法，逐个分析程序 P 中的每个函数 f，直到 f 的入口和出口状态都不再变化为止。函数 analyzeFunction 分析每个函数 h，对于其中的函数调用语句 $u : x = f(x_1, \ldots, x_n)$，算法利用数据流方程 6.1将参数数

算法 6.3 过程间零值分析算法

输入： 整个程序 P

输出： 程序中所有语句的抽象状态集合 $In[\,], Out[\,]$

```
1: procedure zeroAnalysis(P)
2:     for each function f ∈ P do                                  ▷ 初始化
3:         In[f] = Out[f] = [x_1 ↦ ⊥, ..., x_n ↦ ⊥]
4:     In[main] = [a_1 ↦ ⊤, ..., a_n ↦ ⊤]  ▷ a_i, 1 ≤ i ≤ n, 是函数 main 的形参
5:     while any function f's In[f], Out[f] changed do
6:         for each function f ∈ P do
7:             analyzeFunction(f)
8: procedure analyzeFunction(h)
9:     for each statement s ∈ h do
10:        if s is a call statement u : x = f(x_1, ..., x_n) then   ▷ 参数信息传播
11:            In[f] = In[f] ⨆ [y_1 ↦ In[u][x_1], ..., y_n ↦ In[u][x_n]]
12:            Out[u] = In[u][x ↦ Out[f][r]]
13:        else
14:            same as those in intra-procedural analyis ▷ 和过程内分析的规则相同
```

据传播到被调用函数 f，并利用数据流方程6.2计算语句 u 的 $Out[u]$。对于其它语句，算法的分析过程和过程内分析的规则相同。

6.5 上下文相关性

6.5.1 上下文相关分析

考虑数据流方程6.1，该规则直接对函数 f 不同调用点 u 的信息 $s[u]$ 进行合并 $In[f] = \bigsqcup_{u \in pred[f]} s[u]$，亦即不区分函数 f 调用的上下文，因此我们称其是**上下文无关分析**（Context-insensitive analysis）。上下文无关分析为每个函数 f 只维持一个入口状态 $In[f]$，因此相对轻量级也更高效。但是，上下文无关分析将所有调用点的信息进行合并，也因此牺牲了精确

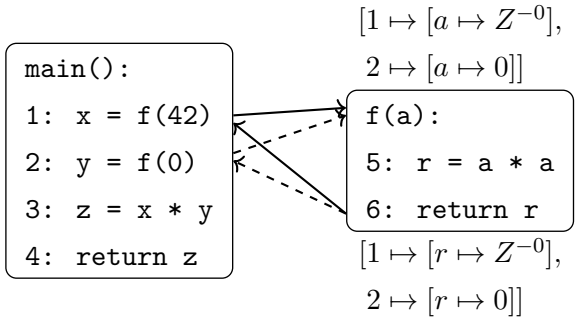


图 6.7: 对示例程序进行上下文无关的过程间零值分析会得到不精确的结果。而采用上下文相关的问题则可以得到更精确的分析结果。

性。

为了理解这一问题，我们以图6.7中的程序为例，分析上下文无关的过程间零值分析在其上的执行过程。最终可以分析得到函数 $main$ 和 f 的出口状态分别为

$$Out[main] = [x \mapsto \top, y \mapsto \top, z \mapsto \top]$$
$$Out[f] = [a \mapsto \top, r \mapsto \top]。$$

这些状态没有预期的精确，例如，不难发现变量 z 的抽象值应该为 Z^{-0}。导致不精确结果的根本原因在于第 1、2 两行函数调用的参数信息 Z^{-0} 和 0 发生了干扰。

为了能够精确的进行过程间的分析，更好的方式是将每个函数调用的上下文也考虑在内，以便将不同的函数调用区分开，这种分析方式称为**上下文相关分析**（context-sensitive analysis）。具体地，对于函数 $f(\ldots)$，我们记为其选择的上下文集合

$$C = \{c_1, \ldots, c_k\},$$

其中每个 c_i，$1 \leq i \leq k$ 都是一个上下文。且集合 C 的分别存在二元和一元运算 $\oplus : C \times C \to C$。则我们可将函数 f 的程

序分析期间的状态 σ 记为映射格

$$\begin{aligned}\sigma = [&c_1 \mapsto [x_1 \mapsto v_{11}, \ldots, x_n \mapsto v_{1n}],\\ &c_2 \mapsto [x_1 \mapsto v_{21}, \ldots, x_n \mapsto v_{2n}],\\ &\vdots\\ &c_k \mapsto [x_1 \mapsto v_{k1}, \ldots, x_n \mapsto v_{kn}]],\end{aligned}$$

其中 x_i，$1 \le i \le n$ 是函数 f 的所有变量。为简便，我们以下将映射格 $[x_1 \mapsto v_{j1}, \ldots, x_n \mapsto v_{jn}]$ 简记为 τ_j，$1 \le j \le k$。

我们将上下文无关形式的数据流方程6.1和 6.2，扩展为支持上下文相关分析的形式。对于函数调用 $u : x = f(x_1, \ldots, x_n)$，假设其上下文为 c_u，且进入的信息为

$$In[u] = [c_1 \mapsto \tau_1, \ldots, c_k \mapsto \tau_k]。$$

则需要将信息

$$\begin{aligned}&[c_u \oplus c_1 \mapsto [\tau_1[x_1], \ldots, \tau_1[x_n]],\\ &\vdots\\ &c_u \oplus c_k \mapsto [\tau_k[x_1], \ldots, \tau_k[x_n]]]\end{aligned}$$

传播到函数 $v : f(y_1, \ldots, y_n)$ 的入口节点，则 f 的入口的信息为

$$\begin{aligned}s[u] = [&c_u \oplus c_1 \mapsto [y_1 \mapsto \tau_1[x_1], \ldots, y_n \mapsto \tau_1[x_n]],\\ &\vdots\\ &c_u \oplus c_k \mapsto [y_1 \mapsto \tau_k[x_1], \ldots, y_n \mapsto \tau_k[x_n]]]。\end{aligned}$$

则我们有函数 f 的入口信息

$$In[v] = \bigsqcup_{w \in pred(v)} s[w]。 \tag{6.3}$$

同理，我们可以得到函数调用语句 u 的数据流方程为

$$\forall c.Out[u][c] = In[u][c][x \mapsto Out[f][u_c \oplus c](r)]。 \tag{6.4}$$

我们以图6.7 中的程序为实例，分析上下文相关分析的执行过程。简单起见，我们选择函数调用语句的地址作为上下文 C，则对于该程序，我们有

$$C = \{-1, 1, 2\},$$

其中元素 -1 是我们为 $main$ 函数选择的上下文（表示其调用地址未知）。我们在 C 上定义如下的运算

$$c_1 \oplus c_2 = c_1。$$

我们从入口状态

$$In[main] = [-1 \mapsto [x \mapsto \top, y \mapsto \top, z \mapsto \top]]$$

开始分析 `main` 函数。对第 1 条是函数调用语句 $x = f(42)$，则算法将实参 42 的抽象状态 $[1 \mapsto [Z^{-0}]]$ 传播给被调用的函数 f 形参 a，将函数 f 的入口信息更新为

$$In[f] = [1 \mapsto [a \mapsto Z^{-0}]]。$$

接着，算法由方程6.4计算语句 1 的输出状态

$$Out[1] = [-1 \mapsto [x \mapsto \bot, y \mapsto \top, z \mapsto \top]]。$$

同理，算法继续分析语句 2，在传播完被调用函数 f 的参数 a 的抽象状态 $[2 \mapsto [0]]$ 后，得到函数 f 的入口信息

$$[1 \mapsto [a \mapsto Z^{-0}],$$
$$2 \mapsto [a \mapsto 0]]。$$

和语句 2 的输出信息

$$Out[2] = [-1 \mapsto [x \mapsto \bot, y \mapsto \bot, z \mapsto \top]]。$$

以此类推，算法最终得到函数 `main` 的出口状态为

$$Out[main] = [-1 \mapsto [x \mapsto \bot, y \mapsto \bot, z \mapsto \bot]]。$$

继续分析函数 `main` 会发现，其入口和出口状态都不在发生变化，到达了不动点。

算法继续分析函数 `f`。此时需要将函数 f 的入口信息

$$[1 \mapsto [a \mapsto Z^{-0}],$$
$$2 \mapsto [a \mapsto 0]]。$$

作为初始状态依次分析所有语句，我们把具体过程留给读者作为练习。最终得到函数 f 的出口信息为

$$[1 \mapsto [r \mapsto Z^{-0}],$$
$$2 \mapsto [r \mapsto 0]]。$$

当最终程序的分析到达不动点时，$main$ 函数的出口状态为

$$Out[main] = [z \mapsto Z^{-0}]。$$

可以看到，和上下文无关的分析相比，上下文相关分析得到了更加精确的信息。

对于涉及到函数调用的控制流图，如何为其函数调用进行数据流建模依赖选择上下文的方式，我们将在接下来的几个小节中具体介绍。

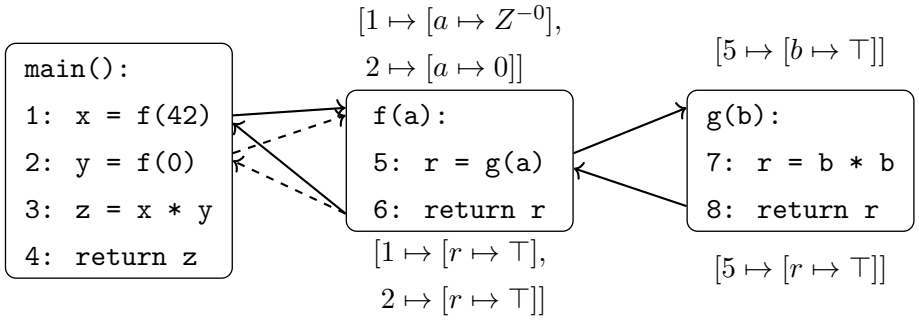


图 6.8: 函数的嵌套调用导致以调用点为上下文的过程间零值分析产生不精确的结果。

6.5.2 调用串

在上一小节，我们简单地使用函数调用的地址作为过程间分析的上下文 C，尽管这种技术比较容易实现，但它无法处理函数多层调用（包括递归调用）的情况。

对6.10中的程序，进行以调用点为上下文的过程间分析后，产生了不精确的分析结果。导致这个问题的根因在于由于只维护了一个调用点，因此函数 g 无法得到调用者的调用者（即函数 $main$）的信息，把两个不同个调用（第 1、2 行）看成了一个调用（第 5 行）。

为了解决这个问题，我们需要采用更精确的上下文。这一小节，我们将讨论一种基于**调用串**（call string）的上下文相关分析方式。调用串是由函数调用时的若干个调用点 c_i，$1 \leq i \leq n$，构成的有序序列

$$\langle c_1, c_2, \ldots, c_n \rangle,$$

其中越靠前的调用点是越近的调用。例如，对图6.10，我们可以有不同的调用串：$\langle 5 \rangle$、$\langle 5, 2 \rangle$、$\langle 5, 1 \rangle$、$\langle 1 \rangle$，等等。特别的，我们用符号 ϵ 表示空的调用串 $\langle\rangle$，它表示函数的调用点是未知的（如图6.10 中的函数 $main$）。

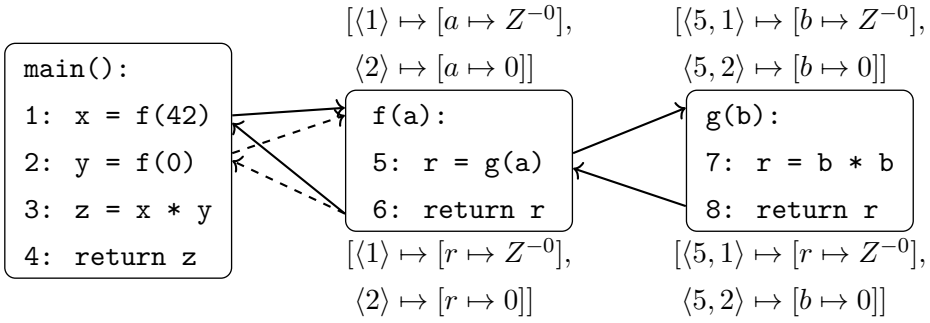


图 6.9: 以调用串为上下文的过程间零值分析。

基于调用串的上下文相关分析则是使用调用串作为上下文 C，其二元操作

$$\langle c_u\rangle \oplus \langle c_1,\ldots,c_n\rangle = \langle c_u,c_1,\ldots,c_n\rangle。 \tag{6.5}$$

我们前面给出的数据流方程6.3 和6.4，同样适用于基于调用串的过程间分析，只是将其中的二元算符换成等式 6.6给定的形式。

我们在图6.9 中给出采用调用串作为上下文对程序进行过程间分析的结果，比采用单一的调用点作为上下文更加精确。

尽管维护任意长度的调用串作为上下文会得到更加精确的结果，但也更加昂贵。因此，我们需要在分析精度和计算效率间取得一个平衡。为此，我们可以使用长度不超过 k 的调用串 $\langle c_1,\ldots,c_m\rangle$，$m \le k$ 作为上下文。对应的，我们需要将调用串的运算规则修改为

$$\langle c_u\rangle \oplus \langle c_1,\ldots,c_n\rangle = \begin{cases} \langle c_u,c_1,\ldots,c_n\rangle, & 若n < k; \\ \langle c_u,c_1,\ldots,c_{n-1}\rangle, & 若n \ge k。 \end{cases} \tag{6.6}$$

在实践中，为了满足灵活的分析需要，可以采用不同的 k 的选取策略。例如，可以选择 k 为固定的常量（如 $k=2$）；或者采用启发式算法，为不同的调用点单独选择不同的 k 值。

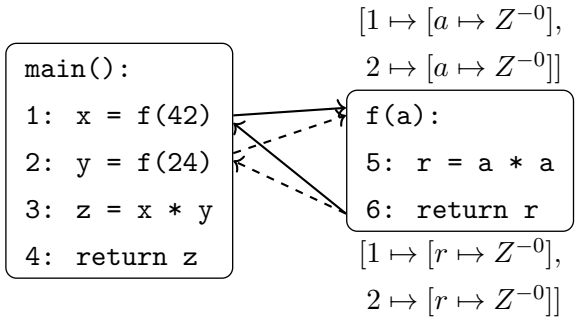


图 6.10: 对示例程序进行过程间零值分析的分析过程。

6.5.3 函数式方法

利用调用串作为上下文进行过程间分析，可以精确地的跟踪不同的函数调用。然而，由于该分析仅考虑了函数调用点的位置信息，因此对于调用未知不同但参数的抽象值相同的函数，会导致重复分析。考虑图6.10中给出的实例，尽管对函数 f 的两次调用具有相同的抽象值 Z^{-0}，但基于调用串的过程间分析仍要分析函数 f 两次。为了避免这样的重复分析，我们可以选择使用来自调用点的数据的抽象状态作为上下文信息，这便是*函数式方法*（Functional approach）的核心思想。

在函数式方法的上下文相关分析中，上下文信息 C 定义为函数调用点参数抽象值的乘积格

$C = \{V_1 \times \ldots \times V_n \mid V_i$是第$i$个参数对应的抽象值，$1 \leq i \leq n\}$。

我们在 C 上定义如下的运算

$$c_1 \oplus c_2 = c_1。 \tag{6.7}$$

我们前面给出的数据流方程6.3 和6.4，同样适用于函数式方法上下文的过程间分析，只是将其中的二元算符换成等式6.7给定的形式。

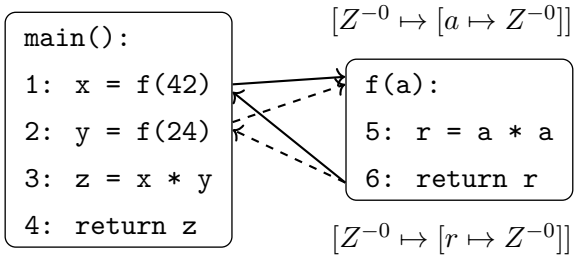


图 6.11: 基于函数式方法对示例程序进行过程间零值分析的结果。

现在，我们重新考虑图6.10中的程序实例，并给出其基于函数式方法的过程间分析结果（图6.11），该方法在减小上下文规模的情况下，得到了和调用串分析相同精度的分析结果。

基于函数式方法的上下文敏感分析在最坏情况下，需要考虑乘积格上所有元素都作为上下文 C 的情况，在格较大时，其时间复杂度较高，因此在实际应用中往往可以选择性进行使用：要么仅仅针对某些函数进行上下文敏感分析，要么只考虑部分变量的上下文信息。

6.6 过程间优化

之前章节讨论的编译器优化技术都在一个过程中进行，我们称这类优化技术为**过程内优化**（Intra-procedural optimizations）。尽管过程内优化比较容易实现，但由于其缺少必要的过程间调用信息，导致其不得不保守地分析被调用的过程，认为在其调用中有可能产生任意副作用。这种局部粒度的优化视角，使得很多编译优化难以取得最佳效果。

过程间优化（Inter-procedural optimizations）通过过程间分析，弥补了过程内分析和优化中信息的缺失，从而到达更好的优化效果。一种常见的过程间优化算法是**过程间常量传播**（Inter-procedural constant propagation）。过程间常量传播

优化追踪已知常量值随着全局变量和过程参数在调用图上的传播，这种追踪可以穿越过程体并跨越调用图的边。通过过程间常量传播，我们可以确定总是接受已知常量值作为参数的过程，或者总是返回已知常量值的过程。当发现具备这样特征的过程时，编译器便可以对与过程相关的代码进行专门的优化处理。

过程间常量传播可以划分为两个子问题：（1）发现常量的初始集合：分析程序必须在每个过程调用位置识别出具有已知常量值的参数，这个问题可以使用过程内常量传播对每个过程分别进行计算；（2）在调用图（即过程间控制流图）上传播已知常量值：给定常量的初始集合，分析程序需要跨越调用图的边过程传播常量值，这个问题可以使用过程间分析算法解决。

算法6.4给出了一个简单的过程间常量传播分析算法。该算法包含一个过程间常量传播分析函数和一个过程内常量传播分析函数，并使用一个平坦格对每一个过程的形参 f 进行建模（记为 $\mathcal{V}[f]$）。另外，为了简化算法，此处假设每个形参具有唯一的名称。

算法中的过程内常量传播分析函数首先对过程 p 进行一个通常的过程内常量传播分析，得到过程中每个变量的抽象值。随后，对于过程 p 中的每个调用点 s 处的每一个形参 f，根据为 f 进行定值的变量 v 的抽象值 $\mathcal{V}[v]$，更新该形参的抽象值 $\mathcal{V}[f]$。

算法中的过程间常量传播分析算法则包含初始化和迭代两个阶段。在初始化阶段，算法首先将程序包含的所有过程的形参映射到格的 $\perp$ 元素上。随后调用过程内常量传播分析

算法 6.4 过程间常量传播算法

输入： 待分析程序 P

输出： 形参到平坦格的映射表

```
1:  function inter-constant-prop(P)
2:      for each procedure p ∈ P do
3:          for each formal parameter f of p do
4:              V[f] = ⊥
5:      for each procedure p ∈ Program do
6:          intra-constant-prop(p)
7:      while V[ ] changed do
8:          List = { f | V[f] changed }
9:          for each f ∈ List do
10:             // suppose procedure p defines the formal parameter f
11:             intra-constant-prop(p)
12:
13: function intra-constant-prop(p)
14:     perform normal intra-procedural constant propagation analysis
15:     for each call site s ∈ p do
16:         for each formal f ∈ s do
17:             // suppose f is assigned by variable v
18:             V[f] = V[f] ∧ V[v]
```

函数对程序中包含的每一个过程进行分析。在传播阶段，算法首先记录在上一轮迭代中映射关系发生变化的形参。对于每一个被记录的形参 f，重新对定义该形参的过程 p 进行过程内常量传播分析。算法在映射表 $\mathcal{V}$ 不再改变时终止。

为了方便读者理解，此处给出算法6.4在以下示例程序

```
1  int e(int a, int b){
2    a = 2;
3    b = a + b;
4    return a;
5  }
6  int f(int i, int j){
7    int s = e(i, j);
```

```
8   int t = e(j, j);
9   return s + t;
10 }
11 void g(){
12   int x = 2;
13   int c = f(x, 1);
14 }
```

中运行的过程。在这个示例程序中，过程 g 在第 15 行调用过程 f，过程 f 在第 8 和 9 行各自调用过程 e。

算法首先进行初始化，为程序中每个形参 f 设置初值（$\mathcal{V}[f] = \bot$）。随后对程序中的每个函数进行过程内常量传播分析。过程 e 并没有调用其他过程，因此不会改变其他形参的抽象值。过程 f 调用了两次过程 e：在第 7 行的调用中，过程 e 的形参 a 由变量 i 定值，因此 $\mathcal{V}[a] = \mathcal{V}[a] \wedge \mathcal{V}[i] = \bot$；过程 e 的形参 b 由变量 j 定值，因此 $\mathcal{V}[b] = \mathcal{V}[b] \wedge \mathcal{V}[j] = \bot$。同理对第 8 行的过程调用进行分析。过程 g 在第 13 行调用了过程 f，并且过程 f 的形参 i 由变量 x 定值，因此 $\mathcal{V}[i] = \mathcal{V}[i] \wedge \mathcal{V}[x] = 2$；过程 f 的形参 j 由常量 1 定值，因此 $\mathcal{V}[j] = \mathcal{V}[j] \wedge 1 = 1$。最后经过初始化，得到如表6.1第一行所示的各形参抽象值。

初始化阶段结束后，算法进入迭代阶段。由于上一轮迭代中，形参 i 和 j 的抽象值改变，且这两个形参属于过程 f，因此需要对该过程重新进行过程内常量传播分析。在第 7 行的调用中，形参 a 的抽象 $\mathcal{V}[a] = \mathcal{V}[a] \wedge \mathcal{V}[i] = 2$，形参 a 的抽象 $\mathcal{V}[b] = \mathcal{V}[b] \wedge \mathcal{V}[j] = 1$。在第 8 行的调用中，形参 a 的抽象 $\mathcal{V}[a] = \mathcal{V}[a] \wedge \mathcal{V}[j] = \top$，形参 a 的抽象 $\mathcal{V}[b] = \mathcal{V}[b] \wedge \mathcal{V}[j] = 1$。本轮迭代后得到如表6.1第二行所示的各形参抽象值。由于形参 a 和 b 的抽象状态被改变，因此需要下一轮迭代，对过程 e 进行过程内常量传播分析。然而过程 e 不包含过程调用，因此

表 6.1: 示例程序过程间产量传播迭代过程

	$\mathcal{V}[a]$	$\mathcal{V}[b]$	$\mathcal{V}[i]$	$\mathcal{V}[j]$
1	$\bot$	$\bot$	2	1
2	$\top$	1	2	1
3	$\top$	1	2	1

不会改变其他形参的抽象值。此时所有形参的抽象值不再改变，迭代过程结束。最终各形参的抽象值如表6.1第三行所示。

值得注意的是，该算法只沿着调用图的边正向传播值为常量的实参。更进一步，我们可以定义回跳函数拓展这一算法，使其能够处理具有全局作用域的返回值和变量，该算法将作为思考题由读者自行完成。

6.7 本章小结

过程间分析与优化技术着眼于整个程序中的多个函数，突破了过程内分析和优化的局限性，从而为程序分析与优化提供了更优的效果。本章首先探讨了过程间优化的目标，并通过具体示例阐明其重要性。随后，介绍了两种关键的数据结构——调用图和过程间控制流图，以及它们各自的构建算法。接下来，讨论了如何利用上下文不敏感的过程间分析，在过程间控制流图上进行抽象状态的传播并求解不动点。为了克服上下文不敏感分析的局限性，我们进一步研究了基于调用串和函数式方法的上下文敏感的过程间分析。最后，以过程间常量传播算法为例，展示了如何应用过程间分析的结果来优化代码。

6.8　深入阅读

6.9　思考题