Machine-independent Optimization

主要考虑全局代码优化的问题

大部分全局优化是基于数据流分析（data-flow analysis）的技术实现的

优化的主要来源

编译器的优化必须保持源程序的语义，且只能进行一些相对低层次的转换，常用的优化方式有

- 在目标代码中消除不必要的指令
- 将一个指令序列替换为更高效的完成相同功能的指令序列

全局公共子表达式

如果表达式 E 在某次出现前已经被计算过，且其中变量的值在那次计算后没有改变，则 E 的出现就被称为公共子表达式（common subexpression），如果之前计算时 E 的结果被赋给 x 且 x 没有被修改过，则可以用 x 代替 E 的计算

复制传播

复制语句，形如 `u = v`，使得在语句后的程序中 u 的值等于 v 的值

如果某个位置上 u 的值一定等于 v 的值，则可以用 v 代替 u，从而可能彻底消除对 u 的使用

死代码消除

如果一个变量在某个程序点上的值会在之后被使用，则称这个变量在这个程序点上是活跃的（live），否则则说这个变量是死的（dead）

对死的变量的赋值就是没有用的死代码

代码移动

即减少循环内部的代码数量，如将循环不变表达式移出循环，在循环开始前就完成计算

e. g.

```
while(i < limit - 2)
...
```

limit - 2 如果在循环中不变，则可以移至循环开始处计算

归纳变量和强度消减

循环中的归纳变量即指对于变量 x，存在常数 c 满足每次对 x 的赋值都使 x 增加 c，则称 x 为**归纳变量**

对于归纳变量，可以将赋值操作改为增量操作，且对于两个步调一致的归纳变量，可以删去其中一个

如循环中的语句

```
i = i - 1
t = 4 * i
```

可以将第二句替换为 `t = t - 4`

强度消减即用低代价的操作（加减）替换高代价的操作（乘除）。归纳变量可以进行强度消减

数据流分析

数据流分析是用来获得数据如何沿着程序执行路径流动的信息的技术，是全局优化的基础

数据流抽象

程序点：指三地址语句之前或之后的位置

- 在基本块中，一个语句之后的程序点等于下一个语句之前的程序点
- 如果流图中有从基本块 B_1 到 B_2 的边，则 B_2 第一条语句的点可能紧跟 B_1 最后一条语句之后的点执行

程序状态：在某个运行时刻，PC 指向某个程序点的时候，各个变量和动态内存中的值

对于同一个程序点，可能对应多个程序状态，数据流分析将可能出现在某个程序点上的**程序状态集合**总结为一些特性

不同的需求对应了不同的性质集合和算法，要求得到的性质集合是一个安全的估计，即根据这些性质优化不会改变语义

数据流分析模式

数据流分析中，将每个程序点和一个**数据流值**（data-flow value）关联起来，这个值是在该点能观察到的所有程序状态集合的抽象

所有可能的数据流值的集合称为**域**（domain）

将语句 s 之前和之后的数据流值记为 $IN[s]$ 和 $OUT[s]$，数据流问题就是对一组约束求解。约束限定了对于所有语句 s 的 $IN[s]$ 和 $OUT[s]$ 之间的关系。

约束可分为基于传递函数和基于控制流的约束

传递函数

在一条语句之前和之后的数据流值受该语句的语义的约束。一条语句之前和之后的数据流值的关系称为**传递函数**（transfer function）

信息可能沿着执行路径前向或逆向传播，在前向数据流问题中，传递函数的形式为

$$OUT[s] = f(IN[s])$$

在逆向数据流问题中，传递函数的性质为

$$IN[s] = f(OUT[s])$$

控制流约束

在基本块中的控制流很简单，即

$$IN[s_{i+1}] = OUT[s_i], i = 1, 2, \dots n-1$$

基本块之间会生成更复杂的约束

基本块上的数据流模式

基本块中的控制流很简单，没有分支和中断，故基本块可以看作是一系列语句的复合

对于基本块 B，可以用 $IN[B]$ 和 $OUT[B]$ 描述进入和离开基本块时的数据流值

- $IN[B]$ 即第一条语句的 IN 值
- 将各个语句的传递函数复合起来得到 $f_B = f_n \circ f_{n-1} \circ \cdots \circ f_1$，则 $OUT[B] = f_B(IN[B])$

对于逆向数据流同理

则基本块之间的约束可分为

- 前向时，$IN[B]$ 与 B 的各个前驱 B_i 的 $OUT[B_i]$ 间有约束关系

- 逆向时，$OUT[B]$ 与 B 的各个后继 B_i 的 $IN[B_i]$ 间有约束关系

数据流方程不同于线性方程，没有唯一解。通常都是寻找一个满足约束的最精确的解

数据流问题

到达定值

到达定值问题即当控制流到达某个程序点时，每个变量可能在程序的哪些地方被定值

到达定值的信息可以有很多应用

- 确定 x 在某个点上是否为常量
- 确定 x 是否未经定值就使用

变量 x 的定值是对 x 的赋值语句，如果不能确定一个语句是否给 x 赋值（过程调用，数组访问，指针间接引用），则为了安全性的考虑，必须假设其**可能**对 x 定值

如果存在一条从对 x 的定值 d 到程序点 p 的路径，且该路径上 d 没有被杀死，则称定值 d 到达程序点 p。如果路径上有对 x 的其他定值，则称 d 被杀死了

如果有定值 d 到达 p，则 p 处使用的 x 有可能是 d 最后定值的。

计算出的到达定值允许不精确，但一定是安全的，即可能有的定值因为程序运行的原因永远不会实际出现，但不能允许可能到达的定值不出现在计算结果中

考虑一个定值 `d: u = v + w`，则这个语句生成了 u 的定值，并且杀死了其他对 u 的定值，则其传递函数可以表示为

$$f_d(x) = gen_d \cup (x - kill_d)$$

$gen_d = \{d\}$，即该语句生成的定值的集合，而 $kill_d$ 是所有其他对 u 的定值

则扩展到基本块情况，其传递函数可写为

$$f_B(x) = gen_B \cup (x - kill_B)$$

其中

$$kill_B = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$
$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup \cdots \cup (gen_1 - kill_2 - \cdots - kill_n)$$

即 $kill_B$ 是各个语句杀死的定值的并集，而 gen_B 是每个语句 i 生成且没有被之后语句杀死的定值的集合

设 B 的前驱的集合为 P，则控制流方程为

$$IN[B] = \bigcup_{B_i \in P} OUT[B_i]$$

即一个定值能沿**某条**路径到达程序点，这个定值就是到达定值

将并集运算称为到达定值的**交汇运算**（meet operator）

假设流图中有两个空的基本块，分别为入口节点 ENTRY 和出口节点 EXIT，则有

$$OUT[ENTRY] = \varnothing$$

则解数据流方程只需将所有基本块的 OUT 设为空，然后求出其 gen 和 $kill$ 集，根据上述的传递函数和控制流方程，迭代求解出最小不动点（least fixpoint），即给出的解是其他所有解的子集

这个算法能结束，因为 OUT 集不会减小，一个定值被加入 OUT 就会一直在集合中，而最终所有定值的集合都是有限的，故最终会到达一个不会变的情况。最大迭代次数是流图的节点个数，即遍历了所有节点。算法结束时，所有的数据流值都满足方程

活跃变量分析

活跃变量分析（live-variable analysis）中，希望知道对于变量 x 和程序点 p，x 在点 p 的值是否会在某条从 p 出发的路径中使用，如果是则称 x 在 p 上活跃

活跃变量可用于寄存器分配

活跃变量是一个逆向数据流问题，从 OUT 值计算出 IN 值。定义

- use_B 是基本块 B 中值先于定值被使用的变量的集合
- def_B 是基本块 B 中定值先于任何使用的变量的集合

则计算其的方法为

$$def_B = def_1 \cup def_2 \cup \cdots \cup def_n$$
$$use_B = use_1 \cup (use_2 - def_1) \cup \cdots \cup (use_n - def_1 - def_2 - \cdots - def_{n-1})$$

对于语句 `s:x = y + z` 有 $def_s = \{x\}, use_s = \{y, z\}$

则易得 use_B 中的变量在基本块开头都活跃，而 def_B 的变量在基本块开头都是死的

其传递函数为

$$IN[B] = use_B \cup (OUT[B] - def_B)$$

其控制流方程为（设 S 为基本块 B 后继的集合）

$$OUT[B] = \bigcup_{B_i \in S} IN[B_i]$$

初始化 $IN[EXIT] = \varnothing$，同时令所有 $IN[B] = \varnothing$，迭代求解即可

同到达定值一样，交汇运算都是并集，且 $IN[B]$ 集合越来越大，故迭代一定会终结

可用表达式分析

如果从流图入口到程序点 p 的每条路径都对 x + y 求值，且从最后一次求值到 p 的路径上没有对 x 或 y 再次赋值，则称 x + y 在 p 可用（available）

可用表达式的用途在于寻找全局公共子表达式

可用表达式的分析同样可用生成-杀死模式

- 基本块对 x 或 y 赋值，且没有重新计算 x + y，则称该基本块杀死了 x + y
- 基本块求值 x + y，且没有对 x 或 y 赋值，则称该基本块生成了 x + y

对于基本块生成的表达式，可以顺序扫描处理，令初始集合 $S = \varnothing$，则对于每条语句 `x = y + z`

- 将 y + z 加入 S
- 从 S 中删除所有涉及 x 的表达式

基本块杀死的表达式就是分量被定值后没有被再次生成的表达式

其传递函数为

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

其控制流方程为（设 P 为基本块 B 前驱的集合）

$$IN[B] = \bigcap_{B_i \in P} OUT[B_i]$$

不同于上述数据流问题，可用表达式分析的交汇运算是交集

初始化 $OUT[ENTRY] = \varnothing$，且令所有 $OUT[B]$ 为全集，这样可以求得更有用的解

总结上述三个数据流问题

	到达定值	活跃变量	可用表达式
域	Sets of definitions	Sets of variables	Sets of expressions
方向	Forwards	Backwards	Forwards
传递函数	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
边界条件	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
交汇运算($\wedge$)	$\cup$	$\cup$	$\cap$
方程组	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
初始值	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

可以得到解数据流问题的通用框架

前向：

```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (除 ENTRY 之外的每个基本块 B) OUT[B] = ⊤;
3)  while (某个 OUT 值发生了改变)
4)      for (除 ENTRY 之外的每个基本块 B) {
5)          IN[B] = ⋀_{P是B的一个前驱} OUT[P];
6)          OUT[B] = f_B(IN[B]);
        }
```

逆向：

```
1)  IN[EXIT] = v_EXIT;
2)  for (除 EXIT 之外的每个基本块 B) IN[B] = ⊤;
3)  while (某个 IN 值发生了改变)
4)      for (除 EXIT 之外的每个基本块 B) {
5)          OUT[B] = ⋀_{S是B的一个后继} IN[S];
6)          IN[B] = f_B(OUT[B]);
        }
```

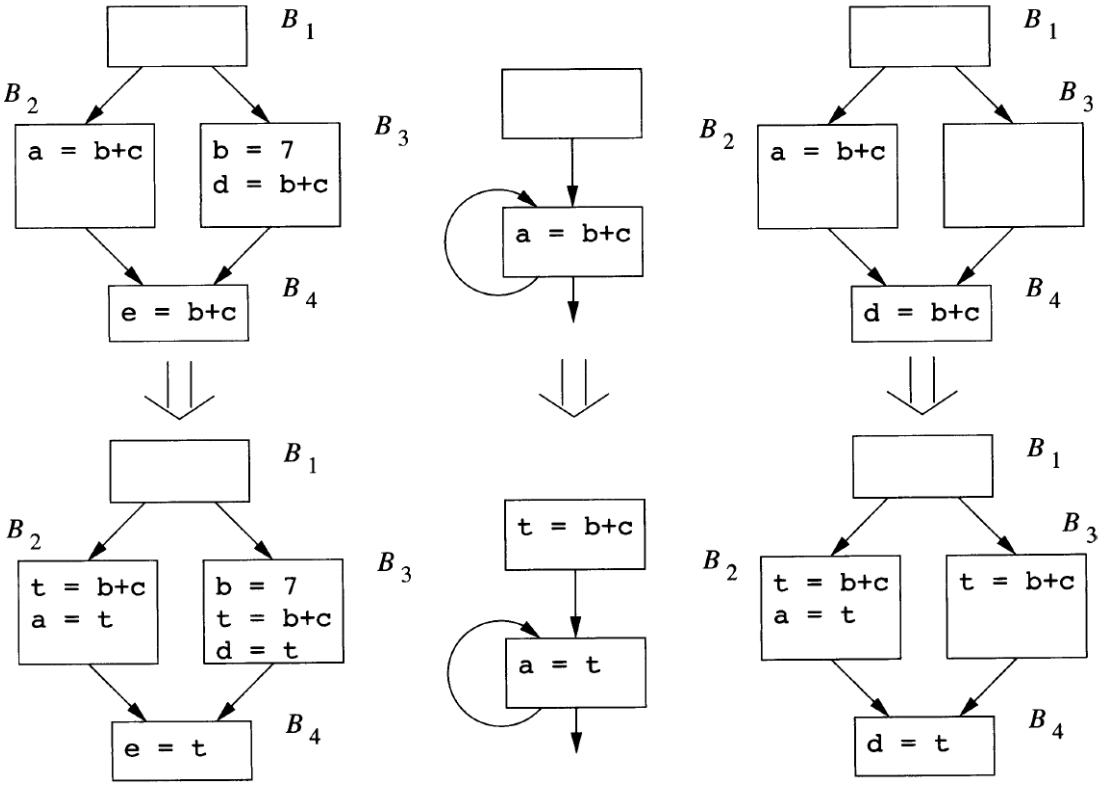
部分冗余消除

优化目标是尽可能减少对表达式的求值，为此可以移动表达式的位置或是将表达式的值存储在临时变量中

冗余有三种形式

- 公共子表达式：如果对 x + y 求值的程序点上 x + y 可用，则不需要再次求值
- 循环不变表达式：循环中 x + y 的值不变，则只需要计算一次
- 部分冗余表达式：程序按某些路径到达时 x + y 已被计算，但是按另一些路径到达时 x + y 未被计算

如下图



a) 公共子表达式　　b) 循环不变代码移动　　c) 部分冗余消除

流图中的循环

循环的优化对于程序的优化有很大的影响，因为程序花费大部分时间执行循环

支配节点

如果每条从入口到节点 n 的路径都经过节点 d，则称 d 支配（dominate）n，记为 d dom n

每个节点都支配自身，且入口节点支配所有节点

则可用使用支配节点树来表示支配节点信息，树的根节点为入口节点，且每个节点支配其在树中的后代

称从入口到节点 n 的任何路径中最后一个支配 n 的节点为 n 的**直接支配节点**

则求解支配节点是一个前向数据流问题

其传递函数为

$$OUT[B] = IN[B] \cup \{B\}$$

控制流方程为

$$IN[B] = \bigcap_{B_i \in P} OUT[B_i]$$

初始化令 $OUT[ENTRY] = \{ENTRY\}$，且令所有 $OUT[B]$ 为全集

深度优先排序

对流图的 DFS 从入口节点开始，访问一个节点，然后访问其最右子节点

搜索路线形成深度优先生成树（DFST），可以将流图中的边分类为

- 前进边：指向真后代
- 后退边：指向祖先
- 交叉边：边的两个节点都不是对方的祖先

回边和可规约性

一条边 $a \to b$ 为回边说明 b 支配 a

对于流图，任何回边都是后退边，但是不是所有后退边都是回边。如果一个流图的 DFST 中所有后退边都是回边，则称该流图是可归约的（reducible）

实践中出现的流图基本都是可归约的（编程思路为顺序，分支，循环）

给定一个流图的 DFST，其**深度**（depth）的定义为各条无环路径上后退边数量的最大值。对于可归约的流图，其深度与具体的 DFST 无关

自然循环

自然循环（natural loop）有重要的性质

- 有唯一的入口节点，支配其余节点
- 存在进入循环头的回边

给定一条回边 $n \to d$ 可以定义一个自然循环，其包含的节点是 d 加上不经过 d 就能到达 n 的节点，这个循环的头是 d

则对于可归约的流图，可以将每条后退边和一个自然循环关联

对于自然循环，除非两个循环有同样的循环头，否则

- 这两个循环互不相交
- 一个嵌套于另一个

则定义最内层循环为不包含其他循环的循环。通常最内层循环是最需要优化的