中国科学技术大学
University of Science and Technology of China

代码优化

《编译原理和技术》

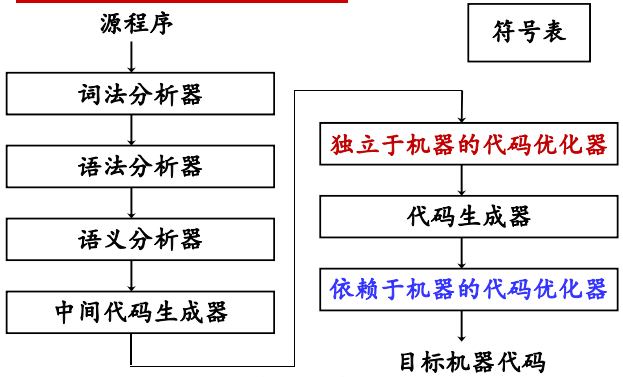
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容



本章内容

- 代码优化
  - 通过程序变换（局部变换和全局变换）来改进程序
- 代码改进变换的标准
  - 代码变换必须保程序的含义
  - 采取安全稳妥的策略
  - 变换减少程序的运行时间平均达到一个可度量的值
  - 变换所作的努力是值得的
- 本章介绍独立于机器的优化
  - 重点：数据流分析及其一般框架、循环的识别和分析

1. 优化的源头和种类

- 基本块内优化、全局优化
- 公共子表达式删除、复写传播、死代码删除
- 循环优化

优化的主要源头和主要种类

- 主要源头:程序中存在程序员无法避免的冗余计算

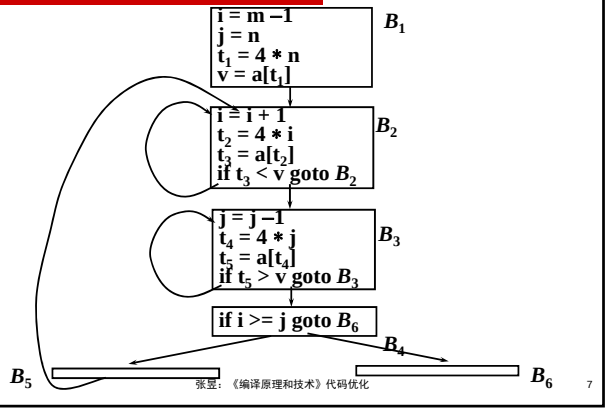
  如 A[i][j]、x.f1等数据访问操作
  - 编译后被展开成多步低级算术运算
  - 对同一数据结构的多次访问会产生许多公共低级运算
- 主要种类
  - 公共子表达式删除(common subexpression elimination)
  - 复写传播(copy propogation)
  - 死代码删除(dead code elimination)
  - 代码外提(loop hoisting, code motion)
  - ……

一个实例

```
i = m –1; j = n; v = a[n];
while (1) {
  do i = i +1; while(a[i]<v);
  do j =j –1;while (a[j]>v);
  if (i >= j) break;
  x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

```
(1) i = m –1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i +1
(6) t2 = 4*i
(7) t3= a[t2]
(8) if (t3<v) goto (5)
(9) j =j –1
(10)t4 = 4*j
(11)t5 = a[t4]
(12)…
```

一个实例：程序流图



基本块内的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

公共子表达式删除(CSE)

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

复写传播

```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = t8
a[t10] = x
goto B2
```

```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
t10 = t8
a[t8] = x
goto B2
```

复写传播变换的做法是在复写语句f = g后，尽可能用g代表f

基本块内的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

复写传播本身不是优化，但给其他优化(常量合并、DCE等)带来机会

公共子表达式删除(CSE)

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

复写传播

```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = t8
a[t10] = x
goto B2
```

```
t6 = 4 * i
x = a[t6]
t7 = t6
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
t10 = t8
a[t8] = x
goto B2
```

死代码删除(DCE)

```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```


死代码删除

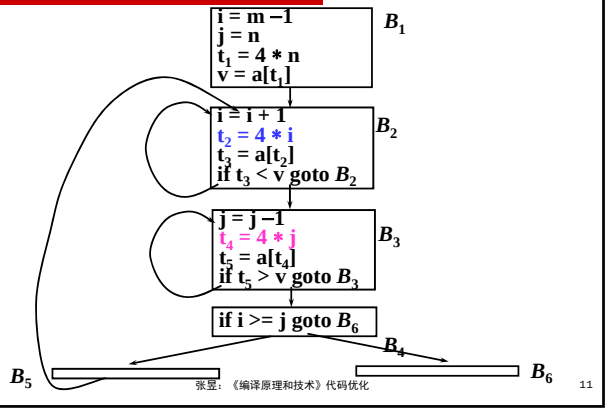
- 死代码
  - 死代码指计算的结果决不被引用的语句
  - 一些优化变换可能会引起死代码

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

debug = true;	debug = false;
. . .	. . .
if (debug) print …	if (debug) print …

靠优化来保证目标代码中没有该条件语句部分

基本块间的优化



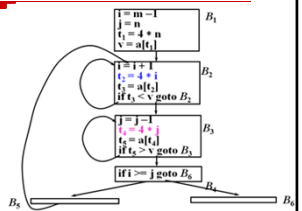
基本块间的优化

B_5 x=a[i]; a[i]=a[j]; a[j]=x;

全局CSE、复写传播、DCE

```
t6 = 4 * i
x = a[t6]
t7 = 4 * i
t8 = 4 * j
t9 = a[t8]
a[t7] = t9
t10 = 4 * j
a[t10] = x
goto B2
```

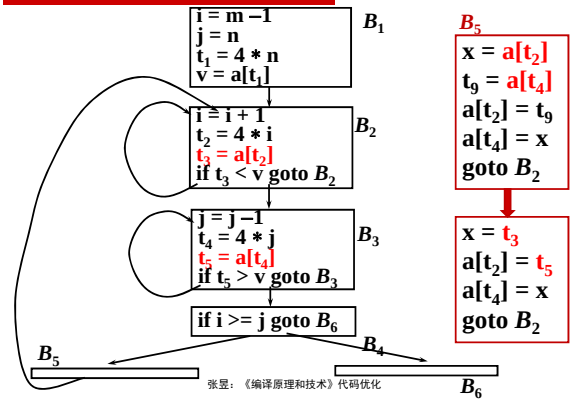
```
t6 = 4 * i
x = a[t6]
t8 = 4 * j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```



```
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
```

```
x = a[t2]
t9 = a[t4]
a[t2] = t9
a[t4] = x
goto B2
```

基本块间的优化



实例：B6的优化

B_6 x = a[i]; a[i] = a[n]; a[n] = x;

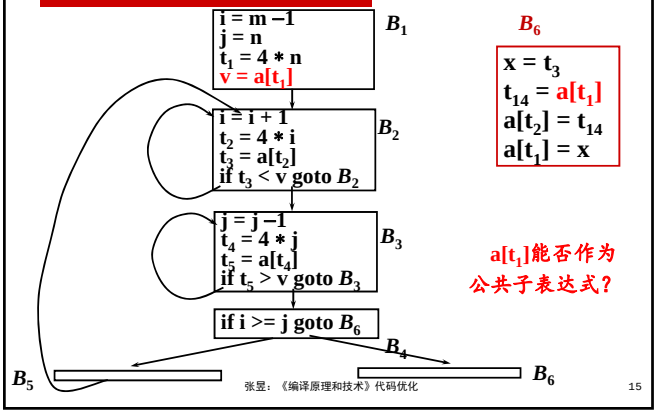
B_1 : $t_1 = 4 * n$

B_2 : $t_2 = 4 * i$, $t_3 = a[t_2]$

$t_{11} = 4 * i$
x = $a[t_{11}]$
$t_{12} = 4 * i$
$t_{13} = 4 * n$
$t_{14} = a[t_{13}]$
$a[t_{12}] = t_{14}$
$t_{15} = 4 * n$
$a[t_{15}] = x$

x = t_3
$t_{14} = a[t_1]$
$a[t_2] = t_{14}$
$a[t_1] = x$

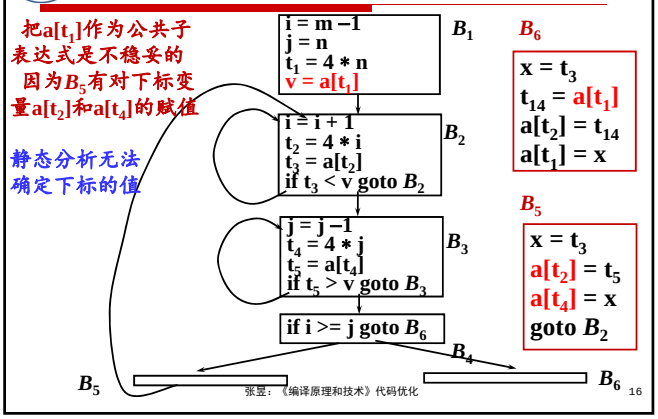
控制流对优化的影响



控制流对优化的影响

把$a[t_1]$作为公共子表达式是不稳妥的
因为B_5有对下标变量$a[t_2]$和$a[t_4]$的赋值

静态分析无法确定下标的值



循环优化

- 循环优化的主要技术
  - 归纳变量删除(induction variable elimination)
  - 强度削弱(strength reduction)：如将乘法变换成加法
  - 代码外提(loop hoisting, code motion)：将循环不变的运算外提
- 代码外提

例：while (i <= limit – 2) …

代码外提后变换成

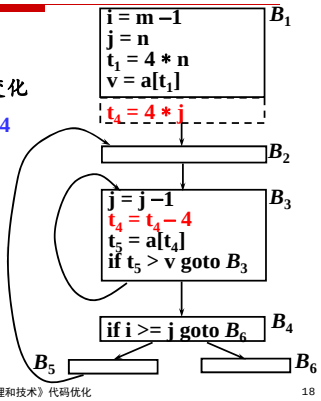
```
t = limit – 2;
while (i <= t ) …
```


循环优化-强度削弱

- 强度削弱
  - j和t_4的值步调一致地变化
  - $t_4 = 4 * j$ 变换为$t_4 = t_4 - 4$
  - 在循环前增加$t_4 = 4 * j$

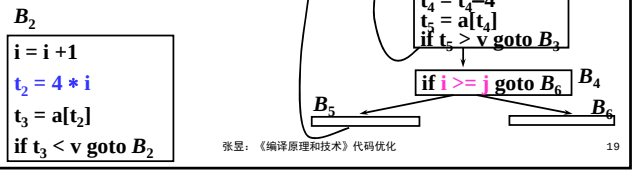
B_3

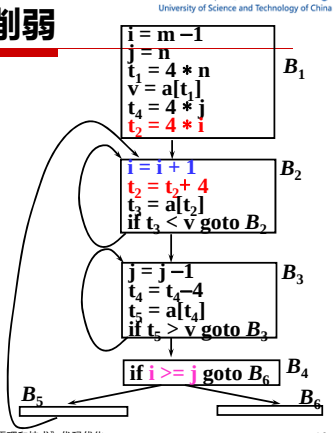
j = j –1
$t_4 = 4 * j$
$t_5 = a[t_4]$
if t_5 > v goto B_3



循环优化-强度削弱

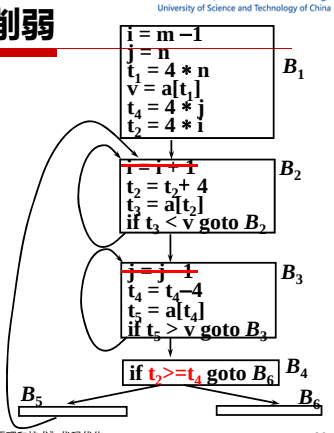
- 强度削弱
  - B_2也可以类似地变换
- 归纳变量删除
  - 循环控制条件i >= j
    可以表示为t_2 >=t_4





循环优化-强度削弱

- 强度削弱
  - B_2也可以类似地变换
- 归纳变量删除
  - 循环控制条件i >= j
    可以表示为t_2 >=t_4
  - 归纳变量 i和j 可删除



2. 程序分析

- 基本块、流图
- 控制流分析
- 数据流分析

流图上的程序点和路径

- 流图上的(程序)点
  - 基本块中，两个相邻的语句之间为程序的一个点
  - 基本块的开始点和结束点
- 流图上的路径
  - 点序列$p_1, p_2, \ldots, p_n$，对1和n - 1间的每个i，满足

(1) p_i是先于一个语句的点，p_{i+1}是同一基本块中位于该语句后的点，或者

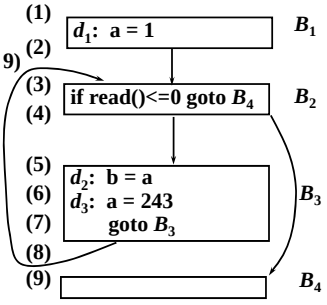
(2) p_i是某基本块的结束点，p_{i+1}是后继块的开始点

流图上的路径

举例

(1, 2, 3, 4, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)
(1, 2, 3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8, 3, 4, 9)
(1, 2, 3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8,
3, 4, 5, 6, 7, 8, …)

- 路径长度无限
- 路径数无限



控制流分析与数据流分析

- 控制流分析
  - 发现每一个过程内的控制流层次结构
  - 从构成过程的基本块开始，然后构造流图
  - 使用必经结点来找出循环
- 数据流分析
  - 确定一个过程中与数据处理有关的全局信息
  - 例如，常量传播分析是力求判定对一个特定变量的所有赋值在某个特定程序点是否总是给定相同的常数值。如果是这样，则在那一点可用一个常数来替代该变量

数据流分析举例

点(5)的所有程序状态

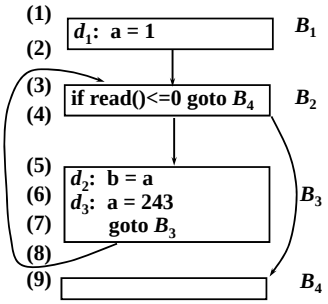
- a ∈ {1, 243}
- a由{d_1, d_3}定值

1. 到达-定值

{d_1, d_3}的定值到达点(5)

2. 常量合并

a在点(5)不是常量



3. 控制流分析

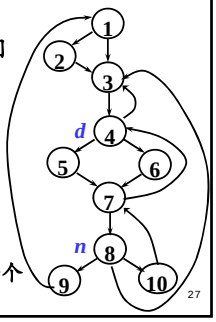
- 重点：识别循环
  支配结点、回边、流图的可归约性

循环

- 识别循环并对循环专门处理的重要性
  - 程序执行的大部分时间消耗在循环上，改进循环性能的优化会对程序执行产生显著影响
  - 循环也会影响程序分析的运行时间
- 支配结点

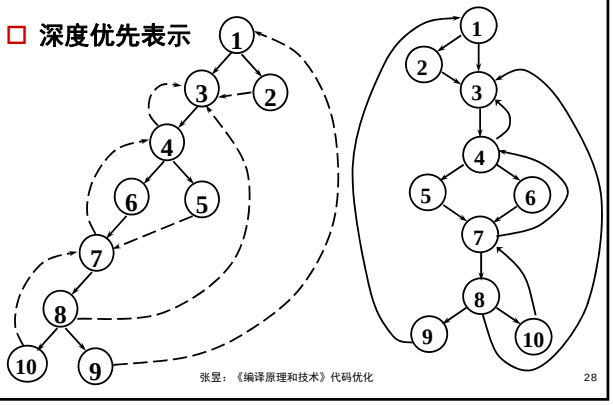
d是n的支配结点(*d dom n*): 若从初始结点起，每条到达*n*的路径都要经过*d*

  - 结点是它本身的支配结点
  - 循环的入口是循环中所有结点的支配结点
  - 支配结点集的计算可以形式化为一个数据流问题



回边和可归约性

- 深度优先表示

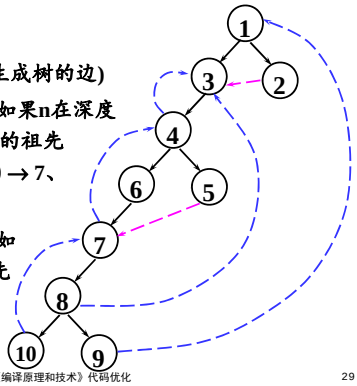


流图中的边的分类

- 深度优先表示
  - 前进边(深度优先生成树的边)
  - m→ n是后撤边，如果n在深度优先生成树上是m的祖先

    4 → 3、7 → 4、10 → 7、8 → 3和9 → 1
  - m→ n是交叉边，如果n和m在深度优先生成树上互不为对方的祖先

    2 → 3和5 → 7



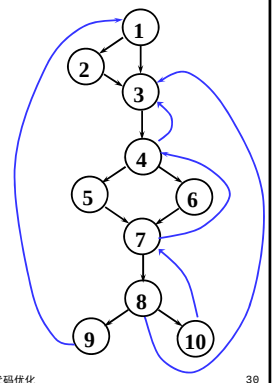
回边和可归约性

- 回边

如果有a dom b，那么边b → a叫做回边

- 可归约性

一个流图称为可归约的，如果在它任何深度优先生成树上，所有的后撤边都是回边。



回边和可归约性

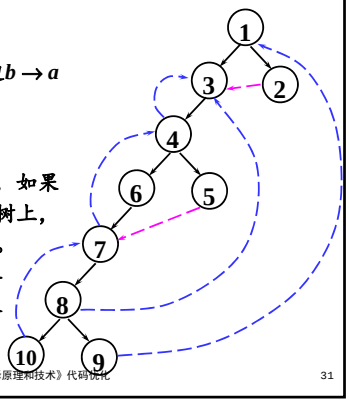
- 回边

  如果有$a\ dom\ b$，那么边$b \rightarrow a$叫做回边

- 可归约性

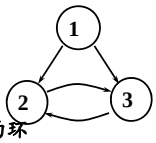
  一个流图称为可归约的，如果在它任何深度优先生成树上，所有的后撤边都是回边。

  - 如果把一个流图中所有回边删掉后，剩余的图无环



不可归约流图

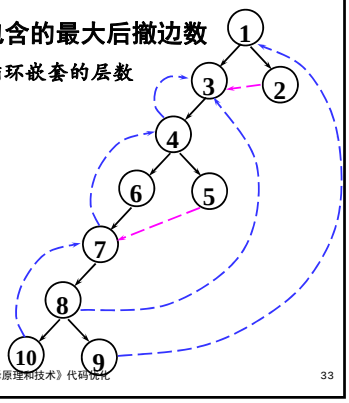
- 开始结点是1
- 2 →3 和 3 →2都不是回边
- 该图不是无环的
- 从结点2和3两处都能进入由它们构成的环



流图的深度

- 深度是无环路径中包含的最大后撤边数
  - 深度不大于流图中循环嵌套的层数
  - 该例深度为3

  $10 \rightarrow 7 \rightarrow 4 \rightarrow 3$



自然循环

- 自然循环的性质
  - 有惟一的入口结点(首结点)。首结点支配该循环中的所有结点
  - 至少存在一条回边进入该循环的首结点

  是流图的强连通分量(SCC)中的一种类型
- 回边$n \rightarrow d$确定的自然循环
  - d加上不经过 d 能到达 n 的所有结点
  - 结点d是该循环的首结点

自然循环

回边$n \rightarrow d$确定的自然循环是d加上不经过 d 能到达 n 的所有结点

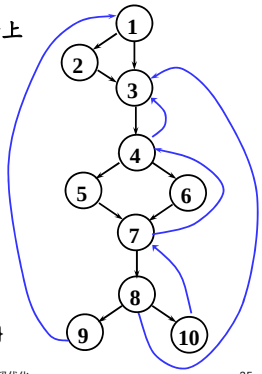
- 回边$10 \rightarrow 7$

  循环{7, 8, 10}
- 回边$7 \rightarrow 4$

  循环{4, 5, 6, 7, 8, 10}
- 回边$4 \rightarrow 3$和$8 \rightarrow 3$

  循环{3, 4, 5, 6, 7, 8, 10}
- 回边$9 \rightarrow 1$

  循环{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

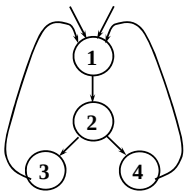


内循环

- 内循环

  若一个循环的结点集合是另一个循环的结点集合的子集

  - 两个循环有相同的首结点，但并非一个结点集是另一个的子集，则看成一个循环



中国科学技术大学 University of Science and Technology of China

4. 数据流分析举例

- 例：到达-定值分析
- 数据流分析模式
- 几种常见数据流问题
- 优化和数据流分析总结

数据流分析

- 数据流分析
  - 分析程序行为时，必须在其流图上考虑所有的执行路径（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）
  - 由于存在循环，从流图得到的程序执行路径数是无限的，且执行路径长度没有有限的上界
  - 每个程序点的不同状态数也可能无限

  程序状态：存储单元到值的映射

  不可能清楚所有执行路径上的所有程序状态

数据流抽象

- 数据流分析不打算
  - 区分到达一个程序点的不同执行路径
  - 掌握该程序点的每个完整的状态
- 数据流分析要
  - 从这些程序状态中抽取解决特定数据流分析所需信息
  - 总结出用于该分析目的的一组有限的事实，且这组事实和到达这个程序点的路径无关，即从任何路径到达该程序点都有这样的事实

    保守的：得到的信息不会误解程序的行为
  - 分析的目的不同，从程序状态提炼的信息也不同

到达-定值(reaching definitions)

- 到达一个程序点的所有定值

  可用来判断一个变量在某程序点是否为常量、是否无初值
- 别名给到达-定值的计算带来困难
  - 过程参数、数组访问、间接引用等都有可能引起别名

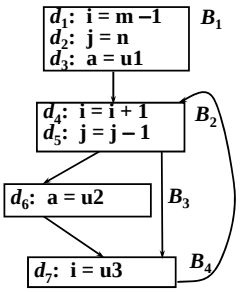
    例如：若p = = q，则p->next和q->next互为别名
  - 程序分析必须是稳妥的
  - 本章其余部分仅考虑变量无别名的情况
- 定值的注销(kill)

  在一条执行路径上，对x 的赋值会注销先前对x的所有赋值

到达-定值的计算

- 基本块向下暴露的定值
  - *gen*[*B*]：基本块*B*生成的定值
  - *kill*[*B*]：基本块*B*注销的定值

gen [B_1] = {d_1, d_2, d_3}
kill [B_1]={d_4, d_5, d_6, d_7}
gen [B_2] = {d_4, d_5}
kill [B_2]={d_1, d_2, d_7}
gen [B_3] = {d_6}
kill [B_3]={d_3}
gen [B_4] = {d_7}
kill [B_4]={d_1, d_4}



基本块的*gen* 和 *kill* 的计算

- 对三地址指令 *d*: u = v + w，它的状态迁移函数是

  $f_d(x) = gen_d \cup (x - kill_d)$，

  x为到达指令入口点的定值
- 若$f_1(x) = gen_1 \cup (x - kill_1)$，$f_2(x) = gen_2 \cup (x - kill_2)$

  则：$f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2)$

  $= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))$
- 若基本块B有n条三地址指令

  $kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$

  $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$

到达-定值的数据流方程

- 数据流方程/等式(flow equation)
  - gen_B：B中能到达B的结束点的定值
  - $kill_B$：整个程序中决不会到达B结束点的定值
  - IN[B]：能到达B的开始点的定值集合
  - OUT[B]：能到达B的结束点的定值集合

两组等式（根据gen和$kill$定义IN和OUT）

- IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[$P$]
- OUT[B] = $gen_B \cup$ (IN[B] – $kill_B$)
- OUT[ENTRY] = ∅

到达-定值方程组的迭代求解，最终到达不动点(MFP)

到达-定值的迭代计算算法

// 正向数据流分析

(1) OUT[ENTRY] = Ø;

(2) for (除了ENTRY以外的每个块B) OUT[B] = Ø;

(3) while (任何一个OUT出现变化)

(4) for (除了ENTRY以外的每个块B) {

(5) IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[$P$];

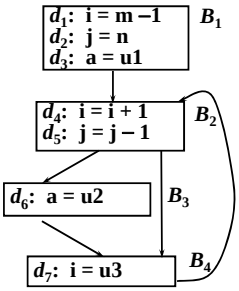
(6) OUT[B] = f_B(IN[B]);

// f_B(IN[B]) = $gen_B \cup$ (IN[B] – $kill_B$)

(7) }

到达-定值计算示例

	IN [B]	OUT [B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

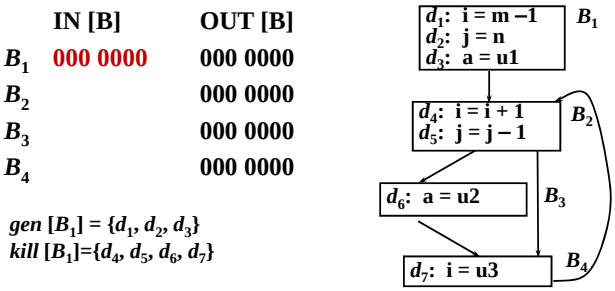
$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[$P$]
OUT[B] = $gen_B \cup$ (IN[B] – $kill_B$)

	IN [B]	OUT [B]
B_1	000 0000	000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

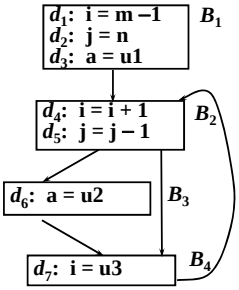
$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[$P$]
OUT[B] = $gen_B \cup$ (IN[B] – $kill_B$)

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

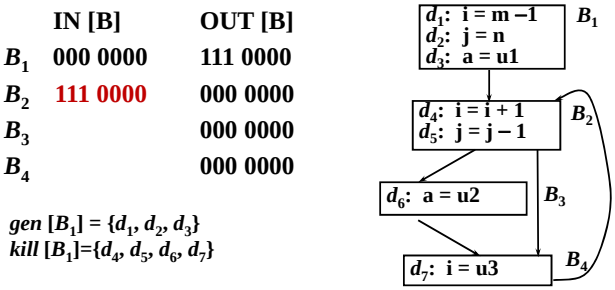
$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

IN[B] = $\cup_{P\text{是}B\text{的前驱}}$OUT[$P$]
OUT[B] = $gen_B \cup$ (IN[B] – $kill_B$)

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

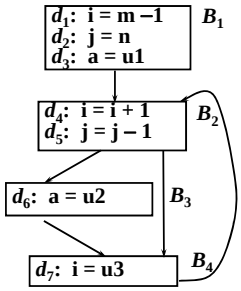
$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 0000
B_4		000 0000

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

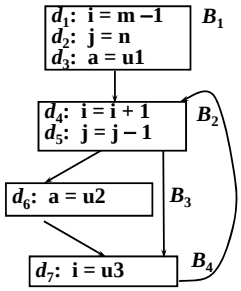
$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

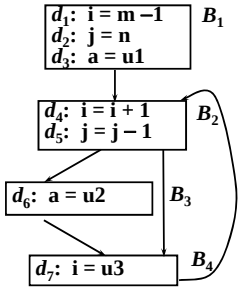
$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$ $kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$ $kill\ [B_4] = \{d_1, d_4\}$

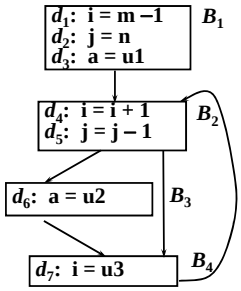
到达-定值计算示例

$IN[B] = \bigcup_{P是B的前驱} OUT[P]$

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

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1110
B_3	001 1100	000 1110
B_4	001 1110	001 0111

不再继续演示迭代计算



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$

到达-定值计算

- 到达-定值数据流等式是正向的方程
  - $OUT\ [B] = gen\ [B] \cup (IN\ [B] - kill\ [B])$
  - $IN\ [B] = \bigcup_{P是B的前驱} OUT\ [P]$

  某些数据流等式是反向的
- 到达-定值数据流等式的合流运算是求并集
  - $IN\ [B] = \bigcup_{P是B的前驱} OUT\ [P]$

  某些数据流等式的合流运算是求交集
- 对到达-定值数据流等式，迭代求它的最小解

  某些数据流方程可能需要求最大解

数据流分析模式

- 数据流值
  - 数据流分析总把程序点和数据流值联系起来
  - 数据流值代表在程序点能观测到的所有可能程序状态集合的一个抽象
  - 语句s前后两点数据流值用IN[s]和OUT[s]来表示
  - 数据流问题就是通过基于语句语义的约束（迁移函数）和基于控制流的约束来寻找所有语句s的IN[s]和OUT[s]的一个解

数据流分析模式

- 迁移函数f
  - 语句前后两点的数据流值受该语句的语义约束
  - 若沿执行路径正向传播，则OUT[s] = f_s(IN[s])
  - 若沿执行路径逆向传播，则IN[s] = f_s(OUT[s])

  若基本块B由语句$s_1, s_2, \ldots, s_n$依次组成，则
  - IN[s_i+1] = OUT[s_i], $i = 1, 2, \ldots, n-1$（逆向…）
  - $f_B = f_n \circ \ldots \circ f_2 \circ f_1$ （逆向：$f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$）
  - $OUT[B] = f_B(IN[B])$ （逆向：$IN[B] = f_B(OUT[B])$）

数据流分析模式

- 控制流约束
  - 正向传播

    $IN[B] = \bigcup_{P是B的前驱} OUT[P]$
  - 逆向传播

    $OUT[B] = \bigcup_{S是B的后继} IN[S]$
- 约束方程组的解通常不是唯一的
  - 求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解

活跃变量(live-variable)

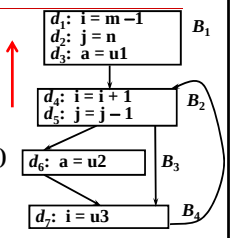
- 定义
  - 变量 x的值在p点开始的某条执行路径上被引用，则说x在p点活跃，否则称x在p点已经死亡
  - IN[B]：块B开始点的活跃变量集合
  - OUT[B]：块B结束点的活跃变量集合
  - use_B：块B中有引用且在引用前无定值的变量集
  - def_B：块B中有定值的变量集
- 应用
  - 一种重要应用就是基本块的寄存器分配

活跃变量

- 例

  $use[B_2] = \{ i, j \}$, $def[B_2] = \{ i, j \}$

- 活跃变量数据流等式
  - IN [B] = $use_B \cup$ (OUT [B] – def_B)
  - OUT[B] = $\cup_{S是B的后继}$ IN [S]
  - IN [EXIT] = ∅
- 和到达-定值等式之间的联系与区别
  - 汇合算符：都是集合并算符
  - 信息流动方向相反，IN和OUT的作用相互交换
  - *use*和*def*分别取代*gen*和*kill*
  - 仍然需要最小解



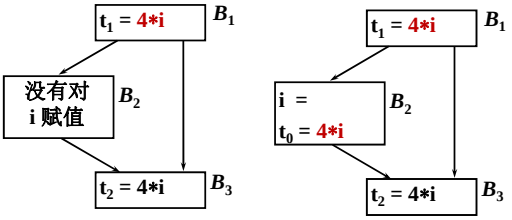
可用表达式

- 可用表达式(available expressions)

x = y + z	x = y + z	x = y + z
…	…	…
…	y = …	z = …
…	…	…
p	*p*	*p*
y + z 在*p*点可用	y + z 在*p*点不可用	y + z 在*p*点不可用

可用表达式

下面两种情况下， 4*i 在B_3的入口都可用



可用表达式

- 定义
  - 若到点*p*的每条执行路径都计算x + y，并且计算后没有对x或y赋值，那么称x + y在点*p*可用
  - e_gen_B：块*B*产生的可用表达式集合
  - e_kill_B：块*B*注销的可用表达式集合
  - IN [B]：块*B*入口的可用表达式集合
  - OUT [B]：块*B*出口的可用表达式集合
- 应用
  - 公共子表达式删除

可用表达式

- 数据流等式
  - OUT [B] = $e\_gen_B \cup$ (IN [B] – e_kill_B)
  - IN [B] = $\cap_{P是B的前驱}$ OUT [P]
  - IN [ENTRY] = ∅
- 同先前的主要区别
  - 汇合算符：使用∩而不是∪
  - 求最大解而不是最小解

数据流问题小结

- 三个数据流问题
  - 到达-定值、活跃变量、可用表达式
- 每个问题的组成
  - 数据流值的论域、数据流的方向、迁移函数、边界条件、汇合算符、数据流等式
- 见书上表9.2

代码优化总结

- 局部优化
  - 基本块内的优化
  - 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会
    - ✓ 冗余指令删除：如

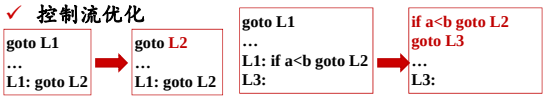
```
mov r0, a        // r0 => a
mov a, r0        // a => r0, 可删除
```

    - ✓ 删除死代码

```
goto L1
goto L2          // 语句前无标号，死代码
```


代码优化总结

- 局部优化
  - 基本块内的优化
  - 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会
    - ✓ 冗余指令删除、删除死代码
    - ✓ 控制流优化



代码优化总结

- 局部优化
  - 基本块内的优化
  - 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会
    - ✓ 冗余指令删除、删除死代码、控制流优化
    - ✓ 强度削弱、删除无用指令

```
mul $8, r0   =>  shiftleft $3, r0
add $0, r1       mul $1, r2      // 均可删除
```

    - ✓ 利用目标机指令特点

如 inc、enter(建立栈帧)、leave(清除栈帧)、龙芯的乘加指令、向量扩展指令等等

代码优化总结

- 局部优化
  - 基本块内的优化
  - 窥孔(peephole)优化：仅分析一个滑动窗口内的指令。每次转换后可能还会暴露相邻窗口间的某些优化机会
- 全局优化
  - 基本块间优化（过程内）
  - 过程间优化：程序全局优化

流敏感(flow-sensitivity)

- 流不敏感分析（flow-insensitive analysis）
  - 不考虑程序中语句执行的顺序.
  - 把程序中语句随意交换位置（即：改变控制流），如果分析结果始终不变，则该分析为流非敏感分析。

前面的数据流分析算法中(4)没有规定对基本块的操作次序

- 流敏感分析（flow-sensitive analysis）

考虑程序中过程内的控制流情况（顺序、分支、循环）

上下文敏感、域敏感、路径敏感

- 上下文不敏感分析Context-insensitive analysis
  - 在过程调用的时候忽略调用的上下文
- 上下文敏感分析Context-sensitive analysis
  - 在过程调用的时候考虑调用的上下文
- 域(不)敏感分析field-sensitive analysis
  - 分析中是否考虑结构体中的不同域、数组中的不同下标元素
- 路径(不)敏感分析path-sensitive analysis
  - 是否依据分支语句的不同谓词来计算不同的分析信息

中国科学技术大学 University of Science and Technology of China

5. 数据流分析的基础

数据流分析框架

- 数据流分析框架 $(D, V, \wedge, F)$包括
  - 数据流分析的方向D，它可以是正向或逆向
  - 数据流值的论域：半格V、汇合算子$\wedge$
  - V到V的迁移函数族F,包括适用于边界条件（ENTRY和EXIT结点）的常函数
- 半格$(V, \wedge)$
  - 是一个集合V和一个二元交运算(汇合运算)$\wedge$，满足:
  - 幂等性：对所有的x，$x \wedge x = x$
  - 交换性：对所有的x和y，$x \wedge y = y \wedge x$
  - 结合性：对所有的x, y和z，$x \wedge (y \wedge z) = (x \wedge y) \wedge z$

半格(semilattices)

- 半格有顶元$\top$（可以还有底元$\bot$）
  - 对V中的所有x，$\top \wedge x = x$
  - 对V中的所有x，$\bot \wedge x = \bot$
- 偏序关系：集合V上的关系$\preceq$
  - 自反性：对所有的x，$x \preceq x$
  - 反对称性：对所有的x和y，如果$x \preceq y$且$y \preceq x$,那么$x = y$
  - 传递性：对所有的x, y和z，如果$x \preceq y$且$y \preceq z$,那么$x \preceq z$

此外，关系$\prec$的定义

$x \prec y$当且仅当$(x \preceq y)$并且$(x \neq y)$

半格

- 半格和偏序关系之间的联系
  - 半格$(V, \wedge)$的汇合运算$\wedge$确定了半格值集V上一种偏序$\preceq$：对V中所有的x和y，$x \preceq y$当且仅当$x \wedge y = x$
  - 若$x \wedge y$等于g，则g就是x和y的最大下界

例 半格的论域V是先前全域U的幂集

- 汇合运算为集合并：$\varnothing$是顶元, U是底元, 偏序关系是$\supseteq$
- 汇合运算为集合交：U是顶元, $\varnothing$是底元，偏序关系是$\subseteq$
- 按偏序$\preceq$意义上的最大解是最精确的

(1) 到达-定值：最精确的解含最少定值

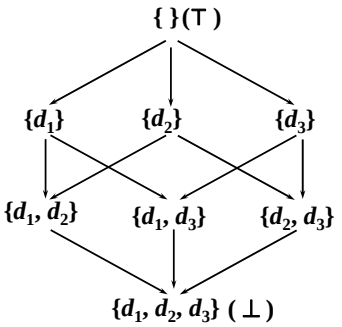
(2) 可用表达式：最精确的解含最多表达式

格图

- 格图
  - 结点是V中的元素
  - 如果$y \preceq x$, 则有从x朝下到y的有向边

右图是定值子集之间形成的格：

- 到达-定值的$\preceq$是$\supseteq$
- $x \wedge y$的最大下界是$x \cup y$



如何降低格图的规模？

- 到达-定值格图存在的问题
  - 数据流值的集合是定值集合的幂集
    =>格图结点数随变量数呈指数级增长
  - 每个变量的定值可达性独立于其他变量的定值可达性

定值半格表示为从每个变量的简单定值半格构造出的积半格

- 积半格(假定$(A, \wedge_A)$和$(B, \wedge_B)$是半格)
  - 论域是$A \times B$
  - 汇合运算$\wedge$: $(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$

数据流分析算法的收敛速度

- 半格的高度
  - 偏序集合$(V, \leq)$中的一个上升链是序列$x_1 \prec x_2 \prec \ldots \prec x_n$
  - 半格的高度就是其中最长上升链中$\prec$的个数

  例，在一个有n个定值的程序中，到达-定值的高度是n
- 数据流分析算法的收敛
  - 半格的高度有限 => 数据流分析迭代算法收敛
  - 半格的值论域有限 => 半格的高度有限
  - 半格的值论域无限 => 半格的高度可能有限

    如，常量传播算法中使用的半格

迁移函数

- 迁移函数族$F: V \to V$有下列性质
  - F包括恒等函数I，即对V中所有的x，有$I(x) = x$
  - F封闭于复合，即对F中任意两个函数f和g，$g \circ f \in F$
  - 若F中所有函数f都有单调性，即

    $x \preceq y$蕴涵$f(x) \preceq f(y)$，或 $f(x \wedge y) \preceq f(x) \wedge f(y)$

    则称框架$(D, V, \wedge, F)$是单调的
  - 框架$(D, V, \wedge, F)$的分配性

    对F中所有的f，$f(x \wedge y) = f(x) \wedge f(y)$

框架单调=> 所求得的解是数据流方程组的最大不动点

迁移函数

- 例 到达-定值分析

  若$f_1(x) = G_1 \cup (x - K_1)$，$f_2(x) = G_2 \cup (x - K_2)$
  - 若G和K是空集，则f是恒等函数
  - $f_2(f_1(x)) = G_2 \cup ((G_1 \cup (x - K_1)) - K_2)$

    $= (G_2 \cup (G_1 - K_2)) \cup (x - (K_1 \cup K_2))$

    因此f_1和f_2的复合f为$f = G \cup (x - K)$的形式
  - 分配性可以由检查下面的条件得到

    $G \cup ((y \cup z) - K) = (G \cup (y - K)) \cup (G \cup (z - K))$

    分配性：$f(y \wedge z) = f(y) \wedge f(z)$

一般框架的迭代算法

- 以正向数据流分析为例

(1) OUT[ENTRY] = v_{ENTRY};
(2) for (除了ENTRY以外的每个块B) OUT[B] = $\top$;
(3) while (任何一个OUT出现变化)
(4) for (除了ENTRY以外的每个块B) {
(5) IN[B] = $\wedge_{P\text{是}B\text{的前驱}}$ OUT[P];
(6) OUT[B] = f_B(IN[B]);
(7) }

数据流解的含义

- 结论：算法所得解是理想解的稳妥近似
- 理想解所考虑的路径
  - 执行路径集：流图上每一条路径都属于该集合

    若流图有环，则执行路径数是无限的
  - 程序可能的执行路径集：程序执行所走的路径属于该集合 — 这是理想解所考虑的路径集
  - 可能的执行路径集 $\subseteq$ 执行路径集
  - 寻找所有可能执行路径是不可判定的
- 以下讨论以正向数据流分析为例

理想解

- 理想解

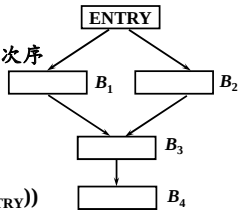
  若路径P = ENTRY $\to B_1 \to B_2 \to \ldots \to B_k$，定义
  - $f_P = f_{k-1} \circ \ldots \circ f_2 \circ f_1$
  - IDEAL[B] = $\wedge_{P\text{是从ENTRY到}B\text{的一条可能路径}} f_P(v_{\text{ENTRY}})$
- 有关理解解的结论
  - 任何大于理想解IDEAL的回答一定是不对的
  - 任何小于或等于IDEAL的值是稳妥的
  - 在稳妥的值中，越接近IDEAL的值越精确

MFP和MOP

- MFP最大不动点解
  maximal fixed point
  - 访问每个基本块（不一定按照程序执行时的次序）
  - 在每个汇合点，把汇合运算作用到当前得到的数据流值，所用的一些初值是人工引入的
- MOP执行路径上的解
  meet over paths
  - MOP[B] = $\wedge_{P\text{是从ENTRY到}B\text{的一条路径}} f_P(v_{\text{ENTRY}})$
  - MOP解汇集了所有可能路径的数据流值，包括那些不可能被执行路径的数据流值
  - 对所有的块B，MOP[B] $\preceq$ IDEAL[B]

MFP与MOP的联系

- MFP与MOP的联系
  - MFP访问基本块未必遵循执行次序
    由各块的初值和迁移函数的单调性保证结果一致
  - MFP较早地使用汇合运算
    IN[B_4] = $f_3(f_1(v_{\text{ENTRY}}) \wedge f_2(v_{\text{ENTRY}}))$
    而MOP[B_4] = $(f_3 \circ f_1)(v_{\text{ENTRY}}) \wedge (f_3 \circ f_2)(v_{\text{ENTRY}})$
    在数据流分析框架具有分配性时，二者的结果是一样的



- MFP $\preceq$ MOP $\preceq$ IDEAL

例题 1

一个C语言程序如下，右边是优化后的目标代码

```
main()                  pushl %ebp
{                           movl %esp,%ebp
   int i,j,k;               movl $1,%eax        -- j=1
   i=5;                     movl $6,%edx        -- k=6
   j=1;                L4:
   while(j<100){            addl %edx,%eax      -- j=j+6
      k=i+1;                cmpl $99,%eax
      j=j+k;                jle .L4             -- while(j≤99)
   }
}
```

完成了哪些优化？

例题 1

一个C语言程序如下，右边是优化后的目标代码

```
main()                  pushl %ebp
{                           movl %esp,%ebp
   int i,j,k;               movl $1,%eax        -- j=1
   i=5;                     movl $6,%edx        -- k=6
   j=1;                L4:
   while(j<100){            addl %edx,%eax      -- j=j+6
      k=i+1;                cmpl $99,%eax
      j=j+k;                jle .L4             -- while(j≤99)
   }
}
```

复写传播、常量合并、代码外提、删除无用赋值
对i，j和k分配内存单元也成为多余，从而被取消

例题 2

一个C语言程序

```
main()
{
   long i,j;

   while (i) {
      if (j) { i = j; }
   }
}
```

生成的汇编码见右边
为什么会有连续跳转？

```
   pushl %ebp
   movl %esp,%ebp
   subl $8,%esp
.L2:
   cmpl $0,-4(%ebp)
   jne .L4
   jmp .L3
.L4:
   cmpl $0,-8(%ebp)
   je .L5
   movl -8(%ebp),%eax
   movl %eax,-4(%ebp)
.L5:
   jmp .L2
.L3:
```

while E1 do S1

```
L2:   E1的代码
      真转 L4
      无条件转 L3
L4:   S1的代码
      JMP L2
L3:
```

if E2 then S2

```
      E2的代码
      假转 L5
      S2的代码
L5:
```


例题 2

一个C语言程序

```
main()
{
   long i,j;

   while (i) {
      if (j) { i = j; }
   }
}
```

生成的汇编码见右边
为什么会有连续跳转？

```
   pushl %ebp
   movl %esp,%ebp
   subl $8,%esp
.L2:
   cmpl $0,-4(%ebp)
   jne .L4
   jmp .L3
.L4:
   cmpl $0,-8(%ebp)
   je .L5
   movl -8(%ebp),%eax
   movl %eax,-4(%ebp)
.L5:
   jmp .L2
.L3:
```

嵌套时代码结构变成

```
L2:   E1的代码
      真转 L4
      无条件转 L3
L4:   S1的代码
      E2的代码
      假转 L5
      S2的代码
L5:   JMP L2
L3:

if E2 then S2
      E2的代码
      假转 L5
      S2的代码
L5:
```

例题 2

一个C语言程序

```
main()
{
    long i,j;

    while (i) {
        if (j) { i = j; }
    }
}
```

优化编译的汇编码见右边

```
    pushl %ebp
    movl %esp,%ebp
.L7:
    testl %eax,%eax
    je .L3
    testl %edx,%edx
    je .L7
    movl %edx,%eax
    jmp .L7
.L3:
```


例题 3 尾递归

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

- 其中的递归调用称为尾递归
- 对于尾递归，编译器应怎样产生代码，使得这种递归调用所需的时空开销大大减少？
- 计算实在参数q和p%q，存放在不同的寄存器中
- 将上述寄存器中实在参数的值存入当前活动记录中形式参数p和q的存储单元
- 转到本函数第一条语句的起始地址继续执行

例题 3 尾递归

求最大公约数的函数

```
long gcd(p,q)
long p,q;
{
    if (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

```
    movl 8(%ebp),%esi     p
    movl 12(%ebp),%ebx    q
.L4:
    movl %esi,%eax
    cltd                          扩展为64位
    idivl %ebx
    movl %edx,%ecx        p%q
    testl %ecx,%ecx       p%q
    je .L2
    movl %ebx,%esi        q⇒p
    movl %ecx,%ebx        p%q⇒q
    jmp .L4
.L2:
```


例题 4

Program → Stmt

Stmt → id := Exp | read (id) | write (Exp) |
Stmt ; Stmt |
while (Exp) do begin Stmt end |
if (Exp) then begin Stmt end
else begin Stmt end

Exp → id | lit | Exp OP Exp

定义Stmt的两个属性

- *MayDef*表示它可能定值的变量集合
- *MayUse*表示它可能引用的变量集合
- 写一个语法制导定义或翻译方案，它计算Stmt的上述*MayDef*和*MayUse*属性

例题 4

Stmt → id := Exp
{ Stmt.*MayDef* = {id.*name*} ;
Stmt.*MayUse* = Exp.*MayUse* }

Stmt → read (id)
{ Stmt.*MayUse* = ∅ ; Stmt.*MayDef* = {id.*name*}}

Stmt → write (Exp)
{ Stmt.*MayDef* = ∅ ; Stmt.*MayUse* = Exp.*MayUse* }

Stmt → $Stmt_1$; $Stmt_2$
{ Stmt.*MayUse* = $Stmt_1$.*MayUse* ∪ $Stmt_2$.*MayUse* ;
Stmt.*MayDef* = $Stmt_1$.*MayDef* ∪ $Stmt_2$.*MayDef* }

例题 4

Stmt → if (Exp) then begin $Stmt_1$ end
else begin $Stmt_2$ end
{ Stmt.*MayUse* = $Stmt_1$.*MayUse* ∪
$Stmt_2$.*MayUse* ∪ Exp.*MayUse*;
Stmt.*MayDef* = $Stmt_1$.*MayDef* ∪ $Stmt_2$.*MayDef* }

Stmt → while (Exp) do begin $Stmt_1$ end
{ Stmt.*MayUse* = $Stmt_1$.*MayUse* ∪ Exp.*MayUse*;
Stmt.*MayDef* = $Stmt_1$.*MayDef* }

Exp → id { Exp.*MayUse* = {id.*name*} }

Exp → lit { Exp.*MayUse* = ∅ }

Exp → Exp_1 OP Exp_2
{ Exp.*MayUse* = Exp_1.*MayUse* ∪Exp_2.*MayUse* }

中国科学技术大学
University of Science and Technology of China

例题 4

基于*MayDef*和*MayUse*属性，说明$Stmt_1;Stmt_2$和$Stmt_2;Stmt_1$在什么情况下有同样的语义

$Stmt_1.MayDef \cap Stmt_2.MayUse = \varnothing$ and
$Stmt_2.MayDef \cap Stmt_1.MayUse = \varnothing$ and
$Stmt_1.MayDef \cap Stmt_2.MayDef = \varnothing$