软件分析

静态单赋值

熊英飞
北京大学
2016

术语-流敏感(flow-sensitivity)

- 流非敏感分析（flow-insensitive analysis）：如果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。
- 流敏感分析（flow-sensitive analysis）：其他情况

- 数据流分析通常为流敏感的

另一种视角：方程求解

- 数据流分析的传递函数和⊓操作定义了一组方程
  - $\mathrm{D}_{\mathrm{v}_1} = \mathrm{F}_{\mathrm{v}_1}\left(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \dots, \mathrm{D}_{\mathrm{v}_\mathrm{n}}\right)$
  - $\mathrm{D}_{\mathrm{v}_2} = \mathrm{F}_{\mathrm{v}_2}\left(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \dots, \mathrm{D}_{\mathrm{v}_\mathrm{n}}\right)$
  - …
  - $\mathrm{D}_{\mathrm{v}_\mathrm{n}} = \mathrm{F}_{\mathrm{v}_\mathrm{n}}\left(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \dots, \mathrm{D}_{\mathrm{v}_\mathrm{n}}\right)$
- 其中
  - $F_{v_1}\left(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \dots, \mathrm{D}_{\mathrm{v}_\mathrm{n}}\right) = f_{v_1}(I)$
  - $F_{v_i}\left(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \dots, \mathrm{D}_{\mathrm{v}_\mathrm{n}}\right) = f_{v_i}\left(\sqcap_{j \in pred(i)} D_{v_j}\right)$
- 数据流分析即为求解该方程的最大解
  - 传递函数和⊓操作表达了该分析的安全性条件，所以该方程的解都是安全的
  - 最大解是最有用的解

方程组求解算法

- 在数理逻辑学中，该类算法称为Unification算法
  - 参考：http://en.wikipedia.org/wiki/Unification_(computer_science)
- 对于单调函数和有限格，标准的Unification算法就是我们学到的数据流分析算法
  - 从$(I, \top, \top, \ldots, \top)$开始反复应用$F_{v_1}$到$F_{v_n}$，直到达到不动点
  - 增量优化：每次只执行受到影响的F_{v_i}

流非敏感分析

- 转换成同样的方程组，并用不动点算法求解

```
a=100;
if(a>0)
  a=a+1;
b=a+1;
```

流非敏感符号分析

$$a = 正 \sqcap a + 正$$
$$b = a + 正$$

不考虑位置，用所有赋值语句更新所有变量

流非敏感活跃变量分析

$$\mathrm{DATA} = \mathrm{DATA} \cup \{a\}$$

对于整个程序产生一个集合，只要程序中有读取变量v的语句，就将其加入集合

时间空间复杂度

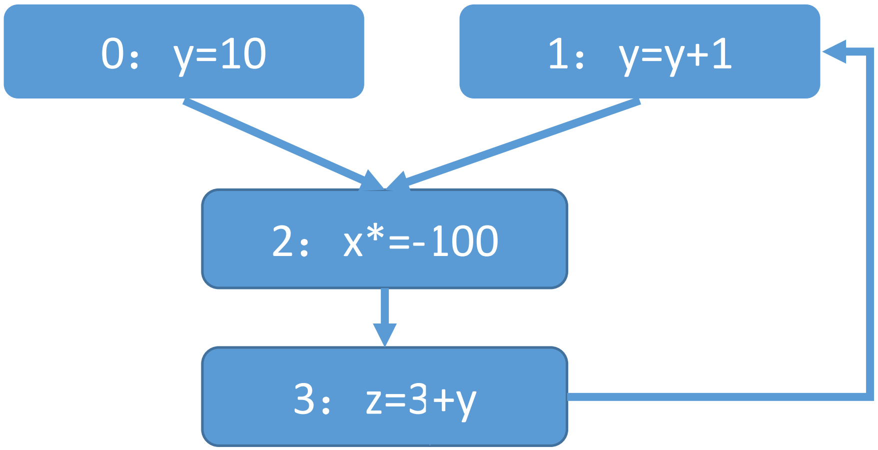
- 活跃变量分析：语句数为n，程序中变量个数为m，使用bitvector表示集合
- 流非敏感的活跃变量：每条语句的操作时间为O(m)，因此时间复杂度上界为O(nm)，空间复杂度上界为O(m)
- 流敏感的活跃变量分析：格的高度为O(m)，转移函数、交汇运算和比较运算都是O(m)，时间复杂度上界为O(nm^2)，空间复杂度上界为O(nm)

- 对于特定分析，流非敏感分析能到达很快的处理速度和可接受的精度（如基于SSA的指针分析）

关于变量中保存值的分析

- 大量分析是关于变量中保存了什么值的
  - 符号分析
  - 区间分析
  - 常量传播

数据流分析的问题

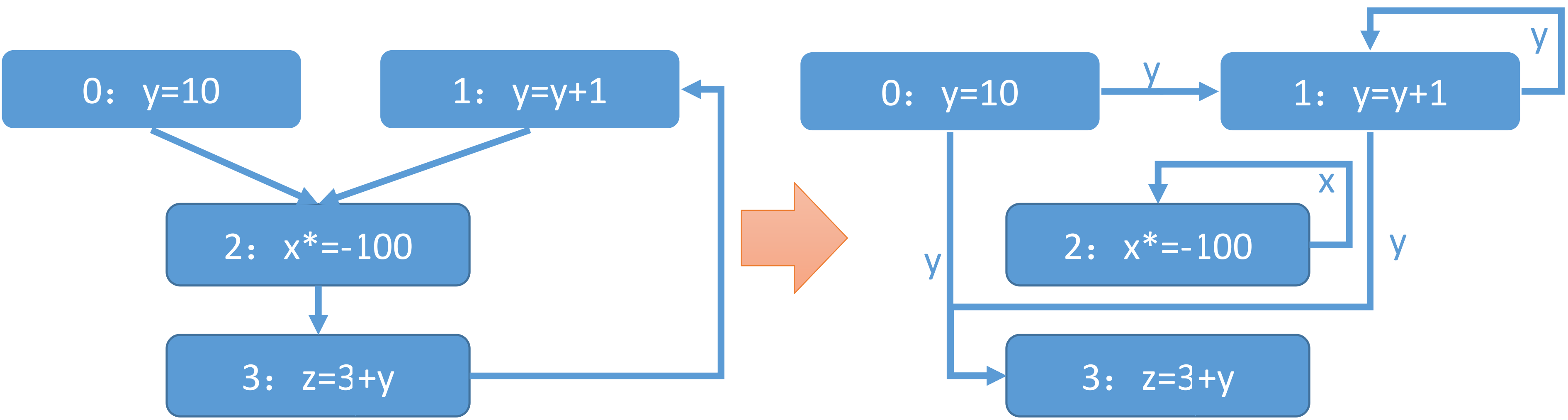


- 问题1： 每个结点都要保存一份关于x,y,z的值
  - 即使结点2和y没有关系
- 问题2： 当1的转换函数更新y的时候， 该更新只和3有关， 但我们不可避免的要通过2才能到达3

Def-Use关系

- 给定变量x，如果结点A可能改变x的值，结点B可能使用结点A改变后的x的值，则结点A和结点B存在Def-Use关系

基于Def-Use的数据流分析



- 每个结点只保存自己定义的变量的抽象值
- 只沿着**Def-Use**边传递抽象值
- 通常图上的边数大幅减少，图变得稀疏（sparse）
- 分析速度大大高于数据流分析

$$y_0 = f_0(\quad)$$
$$y_1 = f_1(y_0 \sqcap y_1)$$
$$x_2 = f_2(x_2 \sqcap x_0)$$
$$z_3 = f_3(y_0 \sqcap y_1)$$

相关性质

- 健壮性Soundness: 用原数据流算法求出来的每一个结果新算法都会求出来
- 准确性Precision: 用新算法求出来的每一个结果原算法都会求出来

基于Def-Use的数据流分析：问题1

- 如何获取Def-Use关系
  - 可以通过Reaching Definition获取Def-Use关系

- 如何还原原始数据流分析的结果
  - 通过Reaching Definition获取使用变量以外的其他变量的定义

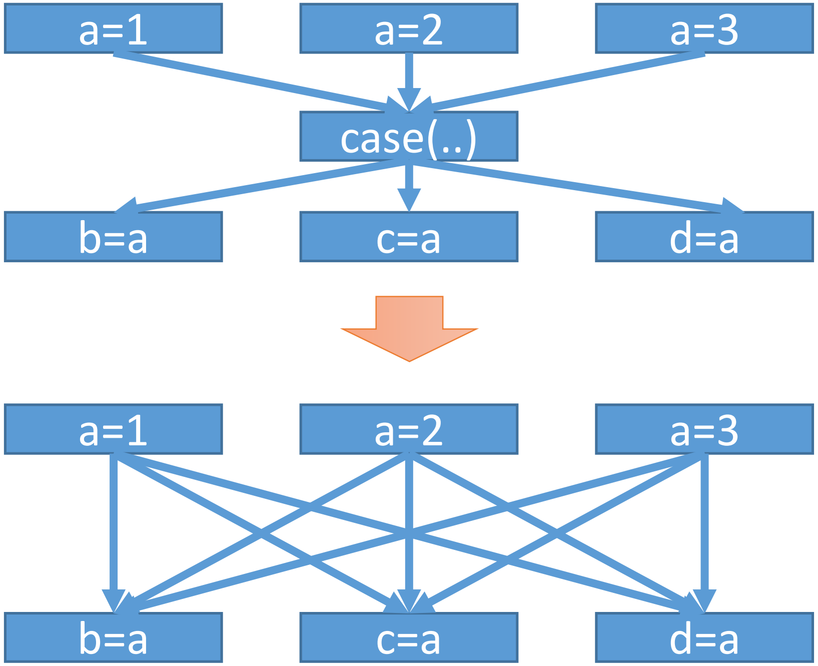
- Reaching Definition的复杂度
  - 程序中赋值语句个数为m，控制流图结点为n
  - 更新单个节点的时间为O(nm)计算O(n)次并集操作，每次O(m)（假设采用良好的hash函数实现集合）
  - 总共需要更新O(mn)次
  - 总时间$O(n^2m^2)$

- Reaching Definition本身就不够快

基于Def-Use的数据流分析：问题2

- 如果可能的定义较多，程序中的边会大幅增长，分析速度反而变慢

```
case (...) of
    0: a := 1;
    1: a := 2;
    2: a := 3;
end
case (...) of
    0: b := a;
    1: c := a;
    2: d := a;
end
```



静态单赋值形式
Single Static Assignment

- 每个变量都只被赋值一次

x=10;	x0=10;
y=y+1;	y0=y0+1;
x=y+x;	x1=y0+x0;
y=y+1;	y1=y0+1;
z=y;	z0=y1;

练习：把以下程序转成静态单赋值形式

```
x=10;
x+=y;
if (x>10)
  z=10;
else
  z=20;
x+=z;
```

```
x0=10;
x1=x0+y;
if (x1>10)
  z0=10;
else
  z1=20;
x2=x1+z?;
```

引入函数ϕ

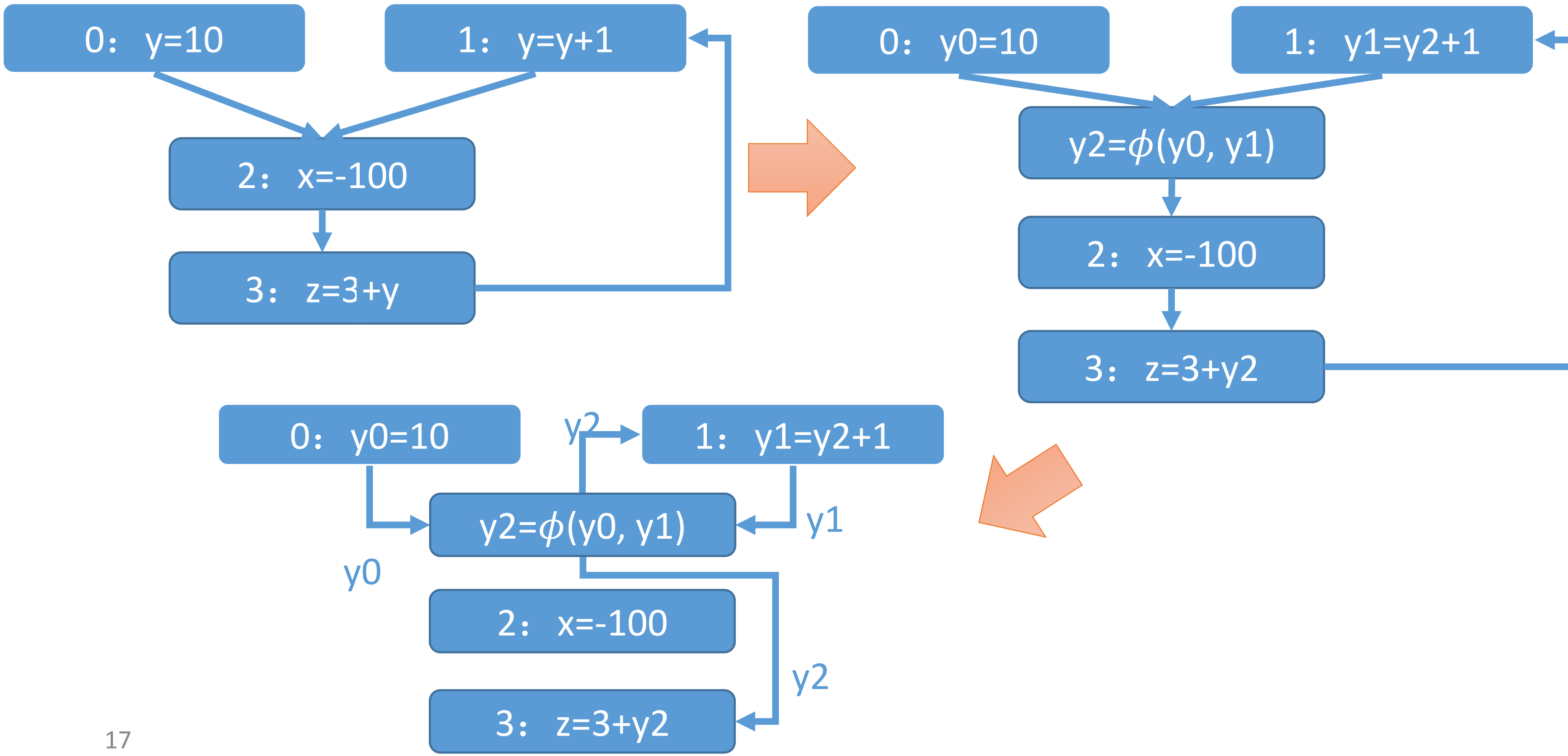
```
x=10;
x+=y;
if (x>10)
  z=10;
else
  z=20;
x+=z;
```

```
x0=10;
x1=x0+y;
if (x1>10)
  z0=10;
else
  z1=20;
z2=ϕ(z0, z1);
x2=x1+z2;
```

函数ϕ代表根据不同的控制流选择不同的值

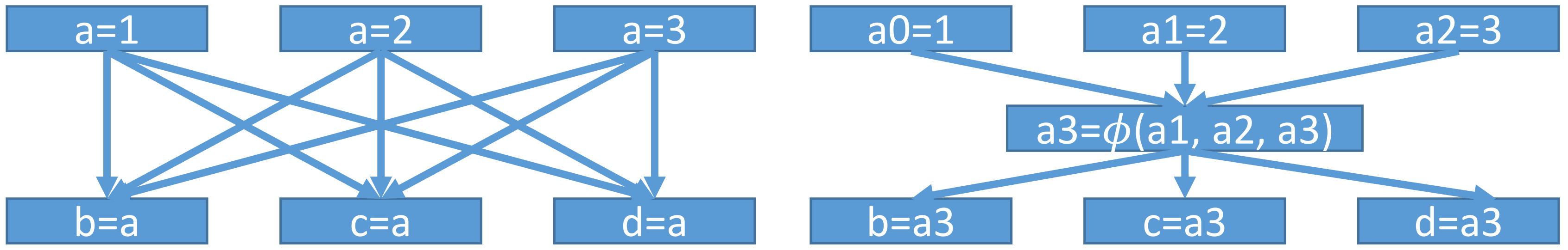
静态单赋值与数据流分析

- 静态单赋值直接提供了Def-Use链



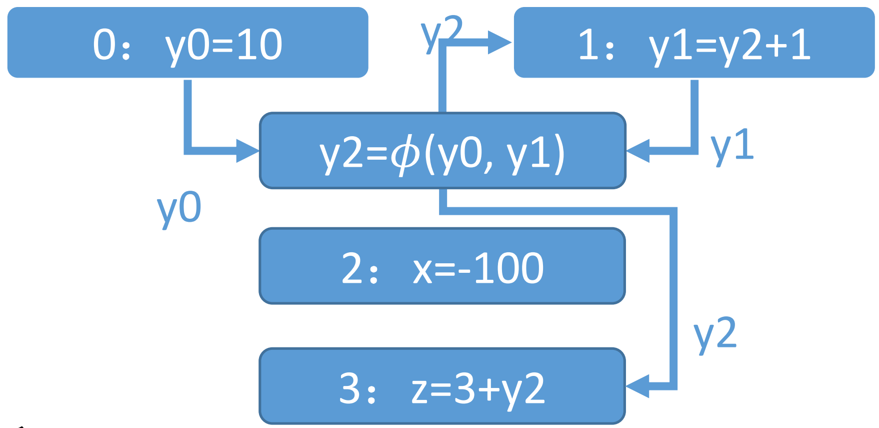
静态单赋值的好处

- 静态单赋值存在高效算法
  - 复杂度为$O(E\alpha(E,N))$或者$O(N^2)$
  - E为边数，N为结点数，α为Ackerman函数的逆
- 静态单赋值中的边不会平方增长



静态单赋值vs流非敏感分析

- 静态单赋值形式上的流非敏感分析与流敏感分析等价

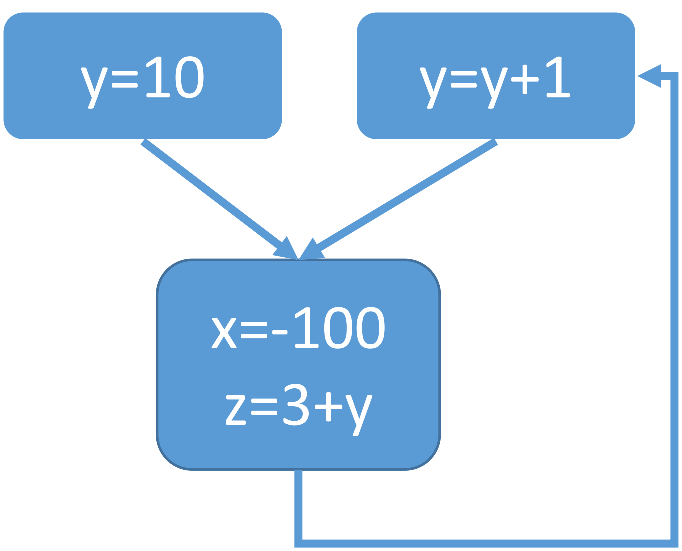


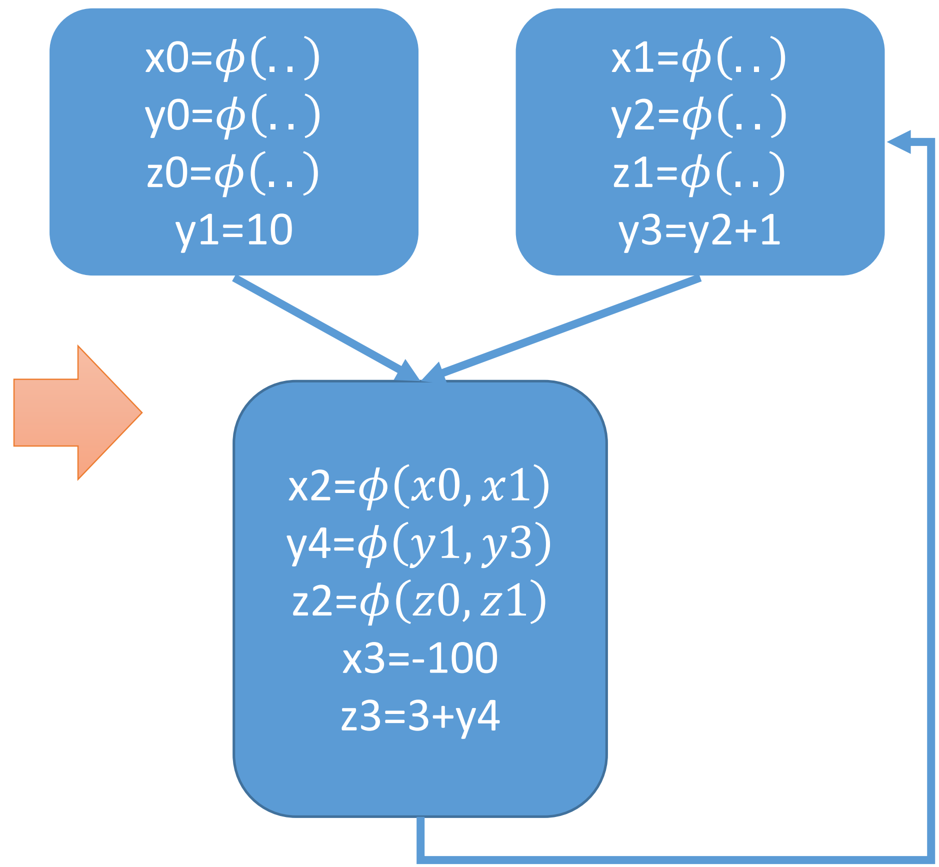
基于静态单赋值形式的分析通常又称为稀疏分析Sparse Analysis

- 流非敏感分析：
  - 每次有值变化时，挑选受影响的转换函数重新执行
  - 全局只保存一份抽象数据
- 流敏感分析：
  - 每次有值变化时，沿着边寻找后继节点的转换函数重新执行
  - 每个结点保存一份抽象数据

转换到静态单赋值形式

- 简单算法
  - 每个基本块的头部对所有变量添加ϕ函数
  - 替换对应变量的值





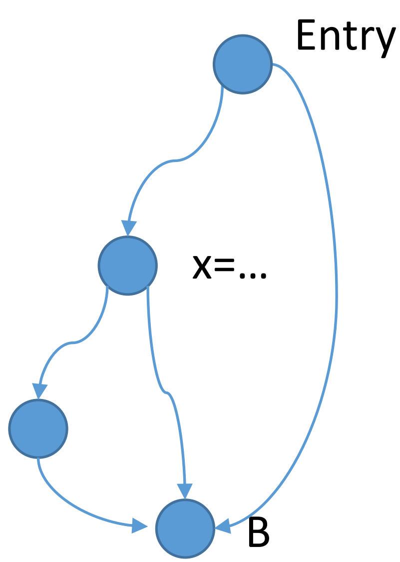
简单算法的问题

- 简单算法引入大量额外ϕ函数
  - 控制流图的每个结点会保存所有变量的值
  - 每条控制流图的边都会对每个变量产生Def-Use关系
  - 实际图并没有变得稀疏，反而可能更加稠密

- 希望能尽量减少引入的ϕ函数，即产生ϕ函数尽量少的静态单赋值形式

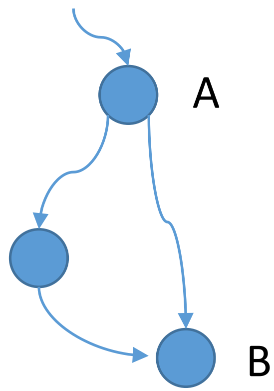
加入ϕ函数的条件

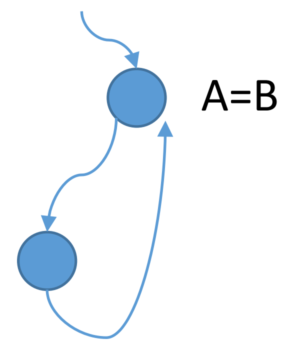
- 至少两条路径在B处汇合
- 其中一条经过了某个赋值语句
- 另外一条没有经过



支配关系

- 结点A支配（dominate）结点B：所有从Entry到B的路径都要通过A

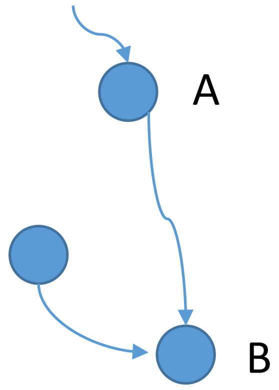


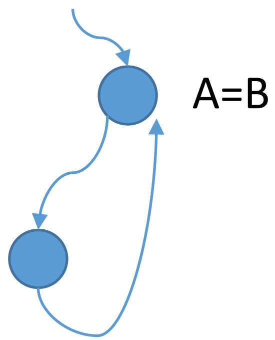


- 结点A严格支配（Strictly dominate）结点B：A支配B并且A和B不是一个结点
  - A不严格支配B => 至少存在一条路径，在到达B之前不经过A

支配边界Dominance Frontier

- 结点A的支配边界中包括B，当且仅当
  - A支配B的某一个前驱结点—至少有一条路径经过A
  - A不严格支配B—至少有一条路径没有经过A，且两条在B处汇合





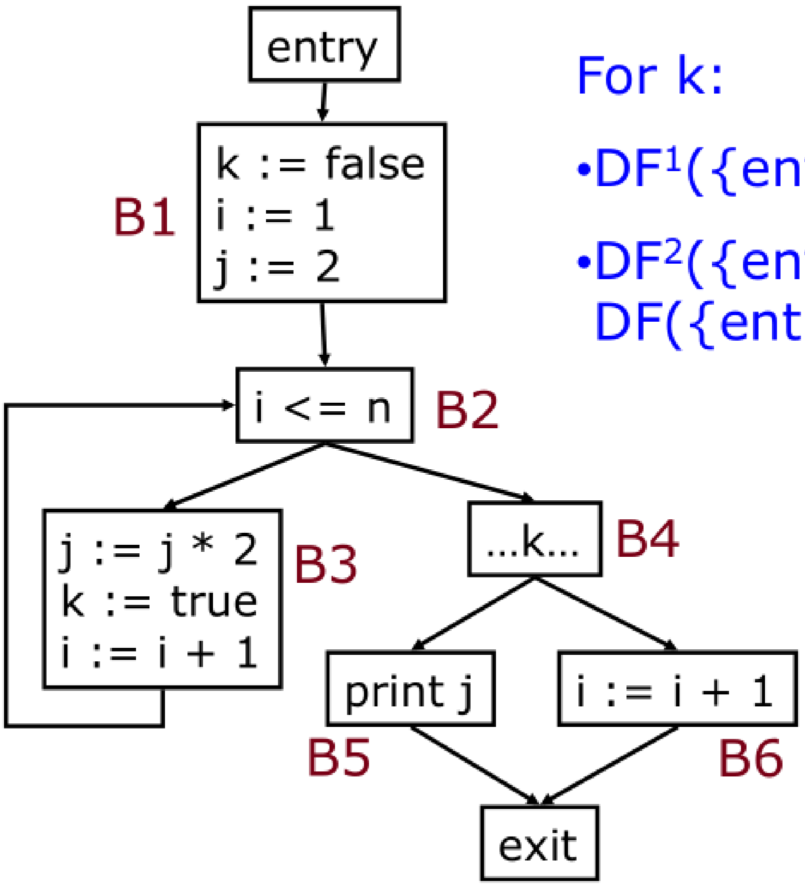
- 对任意赋值语句x=...所在的结点A，所有A的支配边界需要插入ϕ函数计算x的值

转换到静态单赋值形式

- 令DF(a)为a的支配边界集合
- 定义
  - $\mathrm{DF(A)} = \bigcup_{\{\mathrm{a}\in\mathrm{A}\}} \mathrm{DF(a)}$
  - $\mathrm{DF^+(A)} = \lim_{i\to\infty} DF^i(A)$
  - $DF^1(A) = DF(A)$
  - $DF^{i+1}(A) = DF(DF^i(A))$

- 对任意变量i，令A为所有对i赋值的结点，$\mathrm{DF^+(A)}$就是所有需要插入ϕ函数的结点

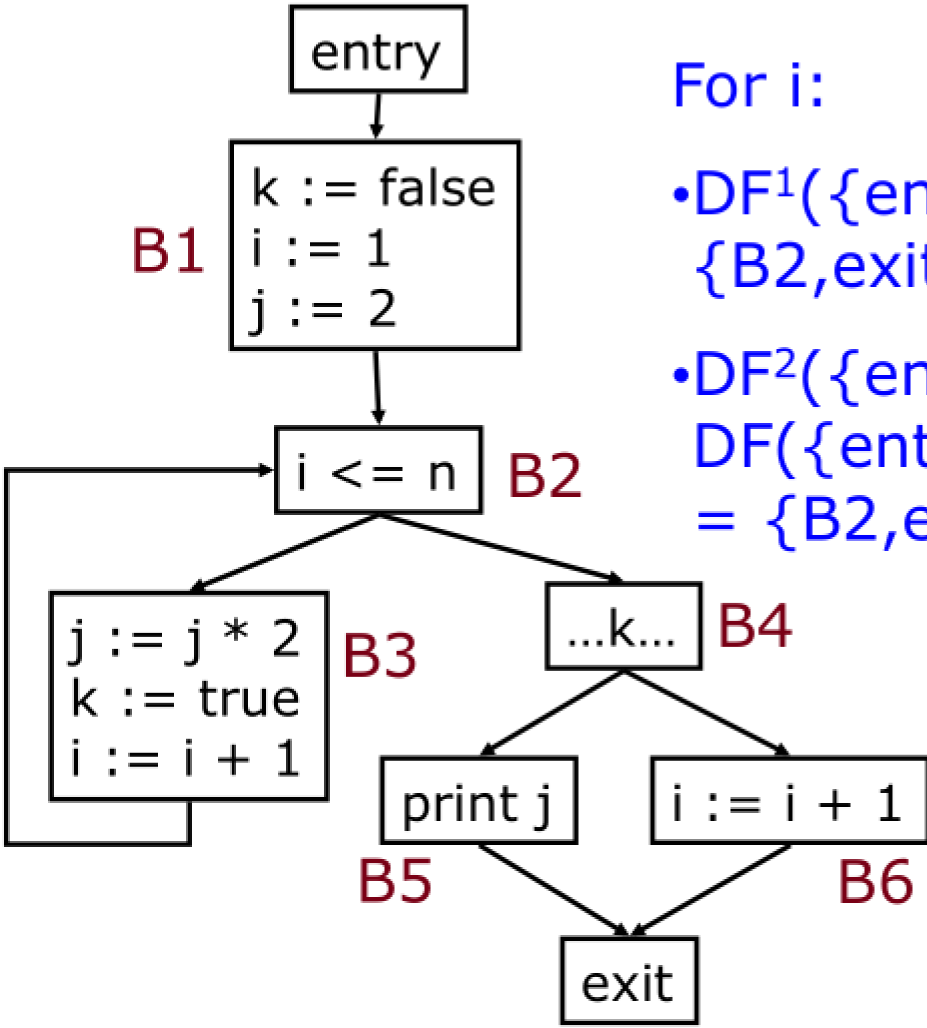
转换示例



For k:

- DF^1({entry,B1,B3}) = {B2}
- DF^2({entry,B1,B3}) = DF({entry,B1,B2,B3}) = {B2}

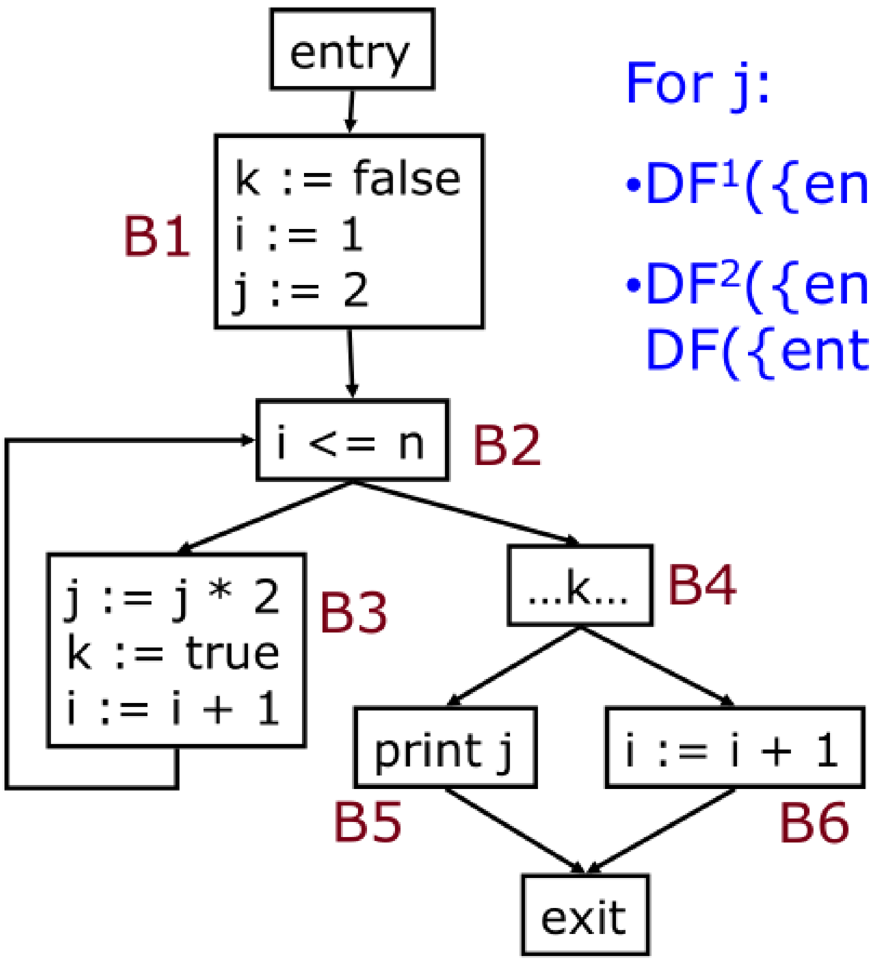
转换示例



For i:

- $DF^1(\{entry,B1,B3,B6\}) = \{B2,exit\}$
- $DF^2(\{entry,B1,B3,B6\}) = DF(\{entry,B1,B2,B3,B6,exit\}) = \{B2,exit\}$

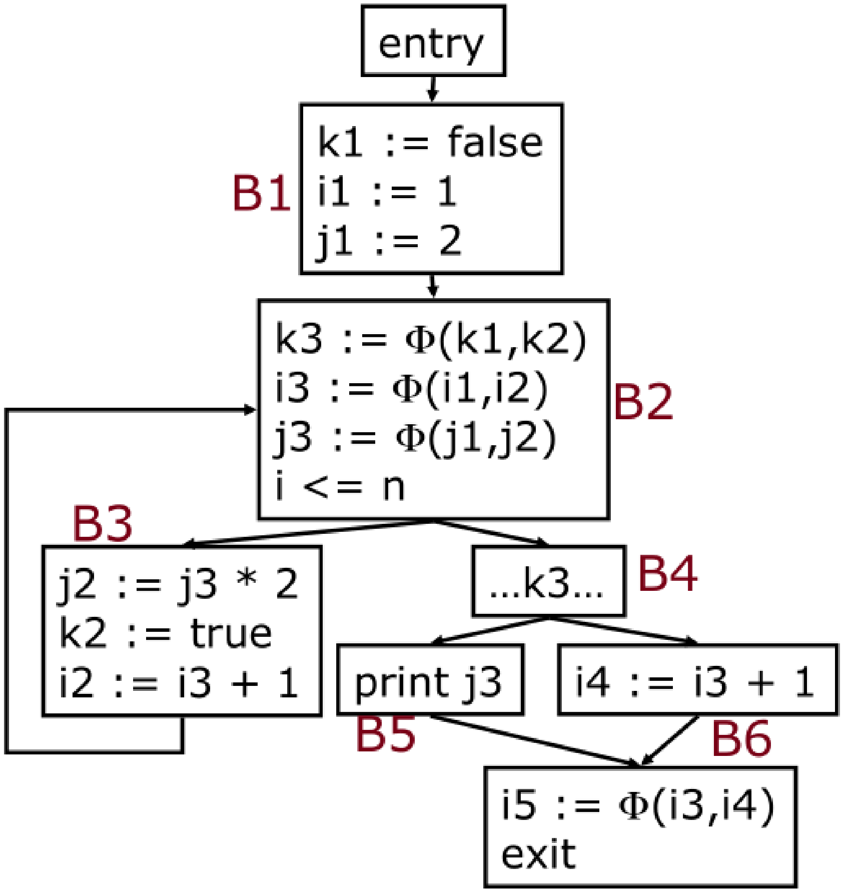
转换示例



For j:

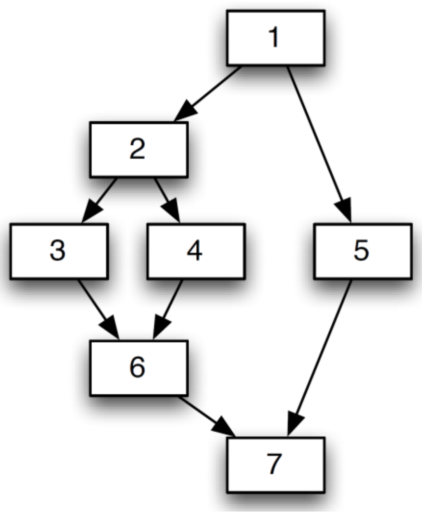
- $DF^1(\{entry, B1, B3\}) = \{B2\}$
- $DF^2(\{entry, B1, B3\}) = DF(\{entry, B1, B2, B3\}) = \{B2\}$

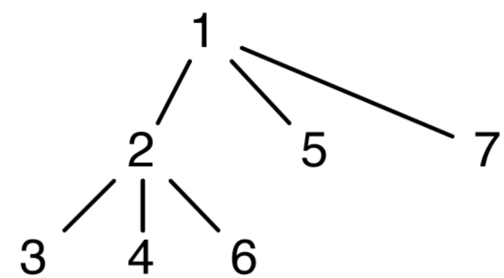
转换结果


entry
B1
k1 := false
i1 := 1
j1 := 2
B2
k3 := Φ(k1,k2)
i3 := Φ(i1,i2)
j3 := Φ(j1,j2)
i <= n
B3
j2 := j3 * 2
k2 := true
i2 := i3 + 1
B4
...k3...
print j3
B5
i4 := i3 + 1
B6
i5 := Φ(i3,i4)
exit

计算支配边界—直接支配者

- 直接支配者immediate dominator：如果a严格支配b，并且不存在c，a严格支配c且c严格支配b，则a是b的直接支配者，记为idom(b)
- 直接支配关系构成一颗树，称为支配树





计算支配树的算法

- Lengauer and Tarjan算法
  - 复杂度为O(Eα(E,N))
  - E为边数，N为结点数，α为Ackerman函数的逆
  - Ackerman函数基本可以认为是常数
- Cooper, Harvey, Kennedy算法，2001年
  - 复杂度为$O(N^2)$
  - 实验证明在实践中比Lenauer and Tarjan算法要快

计算支配边界

for(每个结点b)

  if b的前驱结点数 ≥ 2

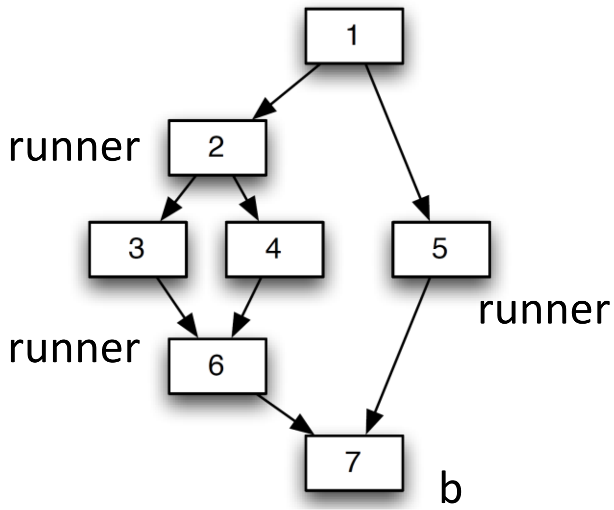
    for(每个b的前驱p)

      runner := p

      while runner ≠ idom(b)

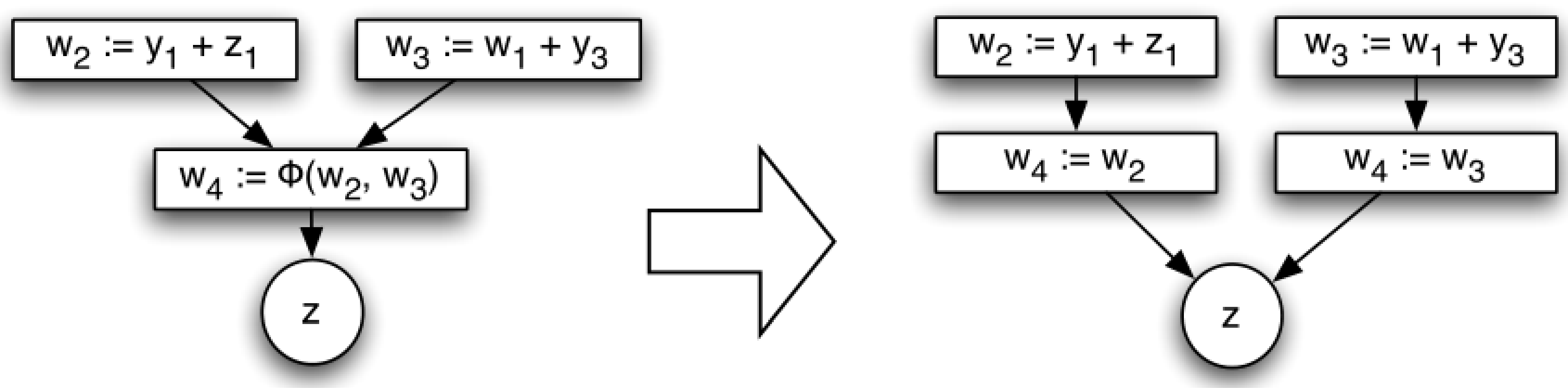
        将b加入runner的支配边界

        runner := idom(runner)



转换回标准型

- 有些分析任务中我们需要再从静态单赋值转换回标准型
  - 程序优化
- 转换过程就是删除掉静态单赋值中的ϕ函数



实践中的静态单赋值形式

- 基于静态单赋值优化数据流分析的条件：
  - 需要分析的每一个内存位置一旦赋值都不会发生改变。
- 这个条件总能成立吗？

C:

```
a=10;
i=&a;
*i=10;
```

Java:

```
a.f=10;
y=a.f;
a.f=20;
y=a.f;
```

解决方案：部分SSA

- 把内存位置分成两组，转换SSA的时候只转换能转换的组，并只对转换的组做优化
- Java的情况：栈上的变量为优化组，堆上的变量为不优化组
- C的情况：把变量分成address-taken和top-level的两组
  - address-taken: 曾经被&取过地址的变量
  - top-level：从没被&取过地址的变量

C的情况的例子

```
int a, b, *c, *d;

int*  w = &a;
int*  x = &b;
int** y = &c;
int** z =  y;
      c =  0;
     *y =  w;
     *z =  x;
      y = &d;
      z =  y;
     *y =  w;
     *z =  x;
```

a-d均为address-taken变量

$w_1 = \mathrm{ALLOC}_a$
$x_1 = \mathrm{ALLOC}_b$
$y_1 = \mathrm{ALLOC}_c$
$z_1 = y_1$
$\mathrm{STORE}\ 0\ y_1$
$\mathrm{STORE}\ w_1\ y_1$
$\mathrm{STORE}\ x_1\ z_1$
$y_2 = \mathrm{ALLOC}_d$
$z_2 = y_2$
$\mathrm{STORE}\ w_1\ y_2$
$\mathrm{STORE}\ x_1\ z_2$

LLVM IR所采用的SSA形式