SSA Form

SSA Form － Static Single Assignment Form（ 静态单一赋值格式 ），它可以看成是一种中间（语言）表示，可以由其它中间表示（如普通三地址语句）转换得到。相比较，它具有如下基本特点：

1) ***SSA 中每个变量，如 v，对它的定值（赋值）仅由唯一的定值语句来完成***。就是说，原来的中间表示（或流图）中如果有多个变量 v 的定值语句，那么对这些定值语句中的左值变量 v 要重新命名（如加下标，v_i 或者 v_j , i <> j, 且它们被看成不同”的变量)。见图 1。

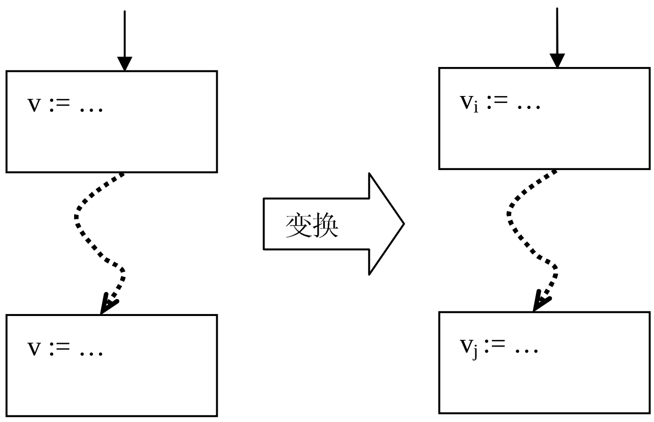

图 1

2) ***SSA 中引入函数∅***。该函数用来处理原中间表示（和流图）中某变量多处定值到达同一引用点（或同一基本块入口），如∅（v_i,v_j,v_k,…），其中参数个数等于到达的定值个数（各个参数已重新命名)；并且又引入一新变量，如 v_m， 令 v_m := ∅（v_i,v_j,v_k,…）; ∅函数的引入目的解决单一定值格式，它的作用可理解为从其参数列表中“任意选择”其一赋给 v_m。

3) *SSA 中变量（如 v）的引用点，所引用的值仅来自于同名变量的值*。这点可以在前两点的基础上，通过分析到达引用点的定值而将引用变量名重命名来获得。见图 2。

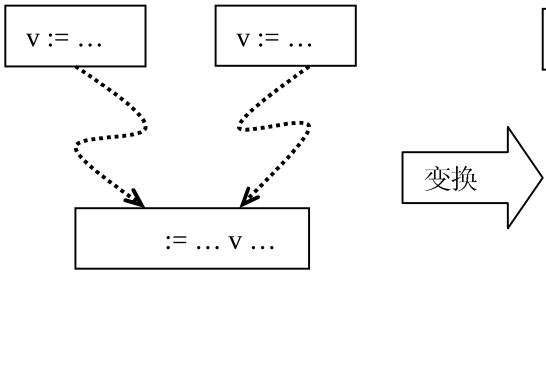

图 2

将一般的中间表示转换成 SSA 格式，有很多好处。由 SSA 基本特点可以较容易确定 DU 链（解决变量在某定值点的引用情况的数据流问题）。此外，对于一些优化措施，如常量传播、值编号（value numbering）、不变代码外提、强度消弱等均可以简单而有效地实现。图 3 和图 4 分别是一个程序（流图）及其相应的 SSA 格式。

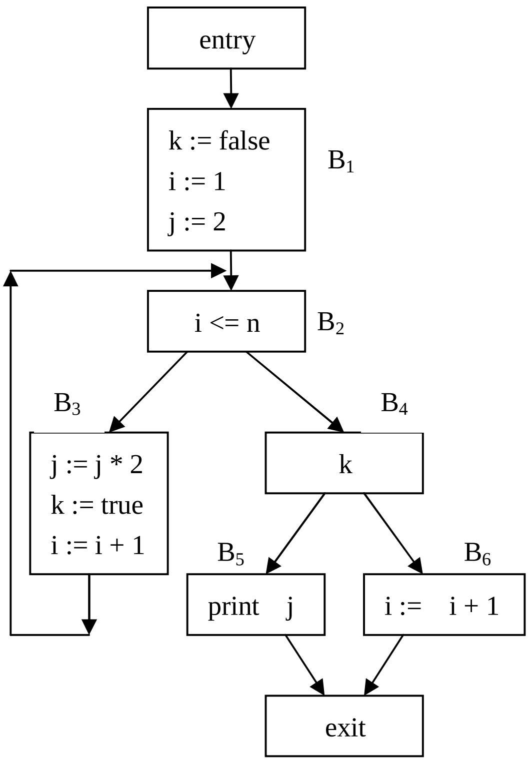

图 3

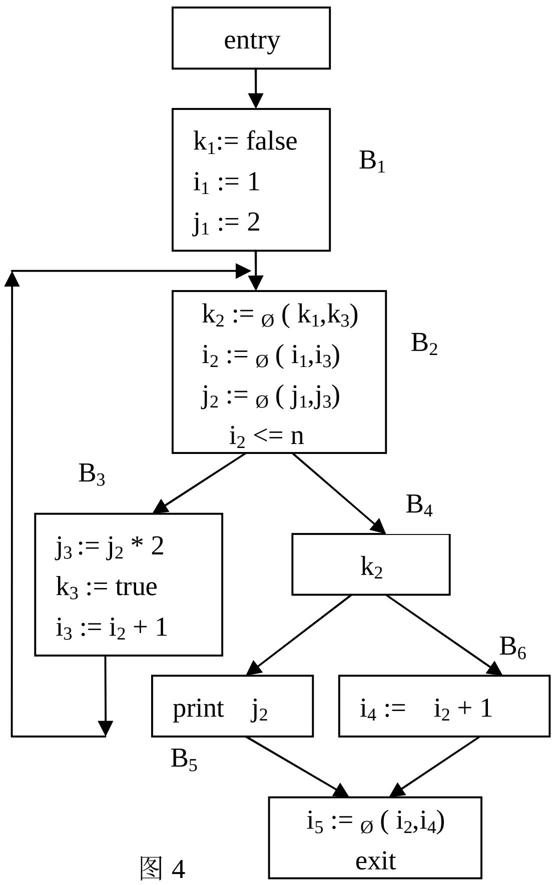

图 4

从图 3 和图 4 中，变量 i 分成了五个，i_1, i_2, i_3, i_4 和 i_5, 而 j, k 也各分成了三个。
图 4 中每个（下标化）变量仅被定值（或赋值）一次。此外，在基本块 B_2 和结束块（exit）中添加了函数∅（...）。
由上述例子，可以看到，要获得 SSA 格式的中间表示，首先得计算出在哪些汇流点（join point，如图 4 中的 B_2）需要插入∅函数；其次在确定需要∅函数的地点后，进行∅函数的插入工作并重新命名有关变量。
最小 SSA 格式－－即含有最少函数∅的 SSA 格式。那如何获得此种 SSA 呢？我们可以从控制（必经）前沿集合(dominance frontier)－－DF 集入手，通过 DF 的正闭包 DF^+ 计算，来求得需要函数∅的汇流点。

定义 1 ：$DF(x) = \{ y \mid \exists z \in pred(y),\ x\ dom\ z$ 且 $x\ !sdom\ y \}$

其中，x 为流图节点，sdom 表示严格必经关系，如 x sdom y ➔ x dom y 且 x <> y, !sdom 则反之。

定义 2 ：$DF(S) = \bigcup_{x \in S} DF(x)$ ，S 为节点集合。

定义 3 ：$DF^+(S) = \lim_{i \to \infty} DF^{(i)}(S)$ ，其中，

$$DF^1(S) = DF(S); \qquad DF^{i+1}(S) = DF(S \cup DF^i(S))$$

如果 S 是含有变量 x 的赋值语句的节点集合，则 $DF^+(S \cup \{entry\})$ 是最终需要有关 x 的∅函数的节点（集合）。由于直接计算 DF 集合较困难，我们可以利用以下集合来简化计算：

定义 4 ：$DF_{local}(x) = \{ y \mid y \in succ(x),\ y \notin IDOM(x) \}$，其中

IDOM(x) = { y | x 是 y 的最近的必经节点, y<>x }，即在必经节点树上，x 的所有子节点集合。

定义 5 ：$DF_{up}(x,z) = \{ y \in DF(z) \mid z \in IDOM(x)$ 且 $y \notin IDOM(x) \}$， 则

定义 6 ：$DF(x) = DF_{local}(x) \cup \bigcup_{z \in IDOM(x)} DF_{up}(x,z)$

图 5 和图 6 给出了计算 DF(x)、DF^+(S) 和 DF(S) 的伪代码,

计算流图中各节点的 DF()

输入：流图 G（N,E）及其必经节点（树）信息，P ：必经节点树的后序遍历序列

输出：DF(x), x ∈ N

```
procedure Dominance_Frontier()
{
 for   i := 1 to |P|   do

     DF( P(i) ) := ∅ ;

     //  计算  DF_local()

     for each y ∈   succ( p(i) ) do

        if y ∉ IDOM(x)   then

           DF( P(i) )    ∪= { y } ;
        endif
     endfor

     //计算  DF_up()

     for each z ∈ IDOM(x) do

        for each y ∈ DF(z) do

           if y ∉ IDOM(x)   then

              DF( P(i) )    ∪= { y } ;
           endif
        endfor
     endfor
 endfor
end
```

图 5

```
function   DF_Plus( S : SET of Node )
return   SET of Node

var change : boolean;
   D, DFP : SET of Node;
begin
     DFP := DF_Set( S ) ;

     repeat
        change := false;
        D := DF_Set( S  ∪  DFP) ;
        if   D <> DFP   then
             DFP := D ;
             change := TRUE;
        endif
     until   ( not change )
     return DFP
end

function DF_Set( S : SET of Node)
return   SET of Node

var
   x : Node;
   D : SET of Node;
begin
   D := ∅;

   for each x ∈ S do

      D  ∪= DF( x ) ;
   endfor
   return D;
end
```

图 6

例 1．构造 SSA 的过程

下面以图 3 来说明构造 SSA 的过程。图 7 是图 3 流图的必经节点树。

对该必经节点树进行后序遍历，可以得到节点次序：

B_3，B_5，B_6，exit，B_4，B_2，B_1，entry

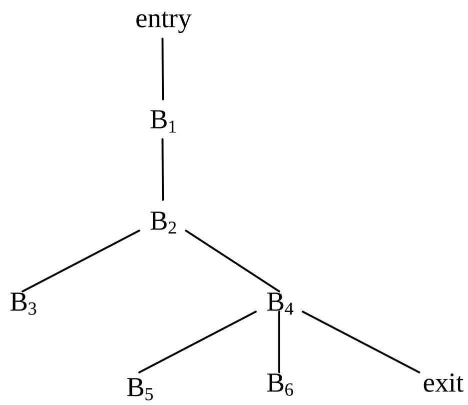

图 7

a) 我们首先按上述次序依次计算各基本块的 DF 集合。

DF（B_3）= { B_2 }
- local = { B_2 }
  - B_2 是 B_3的后继，
  - 且 $B_2 \notin$ IDOM(B_3)，因为
  - B3 在图 7 中是叶子节点，
  - IDOM(B3)= {} //空集合
- UP = {}
  - IDOM(B_3)= {}

DF(B_5) = { exit }
- local = { exit }
  - exit 是 B_5的后继,且由图 7 知，exit $\notin$ IDOM(B_5)
- UP = {}
  - IDOM(B_5)= {}

DF(B_6) = { exit }
- local = { exit }
  - exit 是 B_6的后继,且由图 7 知，exit $\notin$ IDOM(B_6)
- UP = {}
  - IDOM(B_6)= {}

DF(exit) = {}
- local = {}
  - exit 没有后继节点
- UP = {}
  - IDOM(exit)= {}，exit 在图 7 中是叶子节点

DF(B_4) = {}
- local = {}
  - B_5和 B_6 是 B_4的后继,且由图 7 知，二者均 ∈ IDOM(B_4)
- UP = {}
  - B_5 、B_6和 exit 均 ∈ IDOM(B_4),
  - 但是由于 exit ∈ DF（B_5）和 DF（B_6）且 exit∈IDOM(B_4),因此 exit ∉ UP
  - DF(exit) = {}

DF(B_2) = { B_2 }
- local = {}
  - B_3和 B_4 是 B_2的后继,且由图 7 知，二者均 ∈ IDOM(B_2)，因此二者 ∉ local
- UP = { B_2 }
  - B_3 、B_4均 ∈ IDOM(B_2),
  - B_2 ∈ DF（B_3） 且 B_2 ∉ IDOM（B_2） ，因此 B_2 ∈ UP
  - DF(B_4) = {}

DF(B_1) = {}
- local = {}
  - B_2 是 B_1的后继,且由图 7 知，B_2 ∈ IDOM(B_1)
- UP = {}
  - IDOM(B_1)= { B_2 }, DF(B_2) = { B_2 }, 而 B_2 ∈ IDOM(B_1)
  - 因此，B_2 ∉ UP

DF(entry) = {}
- local = {}
  - B_1 是 entry 的后继,且由图 7 知，B_1 ∈ IDOM(entry)
- UP = {}
  - IDOM(entry)= { B_1 }, DF(B_1) = {}

b）然后，我们计算出需要函数∅的基本块（集合）

含变量 k 定值的基本块有：B_1 和 B_3

— $DF^1(\{entry, B_1, B_3\}) = \{B_2\}$

— $DF^2(\{entry, B_1, B_3\}) = DF(\{entry, B_1, B_2, B_3\}) = \{B_2\}$

含变量 i 定值的基本块有：B_1 、B_3 和 B_6

— $DF^1(\{entry, B_1, B_3, B_6\}) = \{B_2, exit\}$

— $DF^2(\{entry, B_1, B_3, B_6\}) = DF(\{entry, B_1, B_2, B_3, B_6, exit\})$
$= \{B_2, exit\}$

含变量 j 定值的基本块有：B_1 和 B_3

— $DF^1(\{entry, B_1, B_3\}) = \{B_2\}$

— $DF^2(\{entry, B_1, B_3\}) = DF(\{entry, B_1, B_2, B_3\}) = \{B_2\}$

c）最后我们重新命名原来的被定值的变量以及函数∅所引入的新变量，即可得到图 4 中 SSA 格式。

例 2. SSA 下常量传播

原程序	SSA 格式
i := 6;	i_1 := 6;
j := 1;	j_1 := 1;
k := 1;	k_1 := 1;
repeat	repeat
	$i_2 \leftarrow \varnothing(i_1 , i_5)$
	$j_2 \leftarrow \varnothing(j_1 , j_3)$
	$k_2 \leftarrow \varnothing(k_1 , k_4)$
if (i = 6) then	if (i_2 = 6) then
k := 0	k_3 := 0
else	else
i := i + 1;	$i_3 := i_2 + 1$;
	$i_4 \leftarrow \varnothing(i_2 , i_3)$
	$k_4 \leftarrow \varnothing(k_3 , k_2)$
i := i + k;	$i_5 := i_4 + k_4$;
j := j + 1;	$j_3 := j_2 + 1$;
until (i = j);	until ($i_5 = j_3$);

图 8

图 8 是某程序片段及其相应的 SSA 格式。这里我们直接在程序中给出了 SSA 格式。

根据实际需要，我们引入数据流框架形式规范定义中的特殊值 TOP（$\top$）和 BOTTOM（$\bot$），二者可以分别理解为“最佳求解值”和“最差求解值”，并约定任意的“值”a, b，a≠b, 对于“运算∧”有：$a \wedge a = a$ ，$a \wedge \top = a$, $a \wedge \bot = \bot$, $a \wedge b = \bot$ 。 并约定每个变量初始化为值$\top$, 常量

传播仅沿标记为“可执行”控制流边进行。值⊤可以穿越“不可执行”的控制流边传播。

图 9 和图 10 显示了两趟常量传播的过程。第二次传播后，结果不在变化。

SSA Form	Pass 1
i_1 := 6	i_1 := 6;
j_1 := 1;	j_1 := 1;
k_1 := 1	k_1 := 1;
repeat	repeat
$i_2 \leftarrow \varnothing(i_1, i_5)$	$i_2 \leftarrow \varnothing(i_1, i_5) \Rightarrow (6 \wedge \top) = 6$
$j_2 \leftarrow \varnothing(j_1, j_3)$	$j_2 \leftarrow \varnothing(j_1, j_3) \Rightarrow (1 \wedge \top) = 1$
$k_2 \leftarrow \varnothing(k_1, k_4)$	$k_2 \leftarrow \varnothing(k_1, k_4) \Rightarrow (1 \wedge \top) = 1$
if (i_2 = 6) then	if (i_2 = 6) then $\Rightarrow$ (6=6) = TRUE
k_3 := 0	k_3 := 0
else	else
$i_3 := i_2 + 1$;	/* $i_3 := i_2 + 1$; 没有执行；i_3 的值仍然为⊤ */
$i_4 \leftarrow \varnothing(i_2, i_3)$	$i_4 \leftarrow \varnothing(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
$k_4 \leftarrow \varnothing(k_3, k_2)$	$k_4 \leftarrow \varnothing(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
	/* k_2 的“值”1 不能穿越不可执行代码到达 */
$i_5 := i_4 + k_4$;	$i_5 := i_4 + k_4$; $\Rightarrow (6 + 0) = 6$
$j_3 := j_2 + 1$;	$j_3 := j_2 + 1$; $\Rightarrow (1 + 1) = 2$
until ($i_5 = j_3$);	until ($i_5 = j_3$) $\Rightarrow$ (6 = 2) = FALSE;

图 9

Pass 1	Pass 2
i_1 := 6	i_1 := 6;
j_1 := 1;	j_1 := 1;
k_1 := 1	k_1 := 1;
repeat	repeat
$i_2 \leftarrow \varnothing(i_1, i_5) \Rightarrow (6 \wedge \top) = 6$	$i_2 \leftarrow \varnothing(i_1, i_5) \Rightarrow (6 \wedge 6) = 6$
$j_2 \leftarrow \varnothing(j_1, j_3) \Rightarrow (1 \wedge \top) = 1$	$j_2 \leftarrow \varnothing(j_1, j_3) \Rightarrow (1 \wedge 2) = \perp$
$k_2 \leftarrow \varnothing(k_1, k_4) \Rightarrow (1 \wedge \top) = 1$	$k_2 \leftarrow \varnothing(k_1, k_4) \Rightarrow (1 \wedge 0) = \perp$
if (i_2 = 6) then $\Rightarrow$ (6=6) = TRUE	if (i_2 = 6) then $\Rightarrow$ (6=6) = TRUE
k_3 := 0	k_3 := 0
else	else
/* 没有执行；i_3的值仍然为$\top$ */;	/* 没有执行；i_3的值仍然为$\top$ */
$i_4 \leftarrow \varnothing(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$	$i_4 \leftarrow \varnothing(i_2, i_3) \Rightarrow (6 \wedge \top) = 6$
$k_4 \leftarrow \varnothing(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$	$k_4 \leftarrow \varnothing(k_3, k_2) \Rightarrow (0 \wedge \top) = 0$
	/* k_2的“值”$\perp$不能穿越不可执行代码到达 k_4*/
$i_5 := i_4 + k_4$; $\Rightarrow (6 + 0) = 6$	$i_5 := i_4 + k_4$; $\Rightarrow (6 + 0) = 6$
$j_3 := j_2 + 1$; $\Rightarrow (1 + 1) = 2$	$j_3 := j_2 + 1$; $\Rightarrow (\perp + 1) = \perp$
until ($i_5 = j_3$); $\Rightarrow$ (6 = 2) = FALSE	until ($i_5 = j_3$); $\Rightarrow (6 = \perp) = \perp$

图 10

我们可以看到经过常量传播后带来的新的程序优化机会。我们从 Pass 2 的 SSA 中，可以了解到 i_5处的定值是常量 6，而 i_2处的引用值也是常量 6。在这种情况下，结合其它多种优化技术，程序可以得到很好的优化结果。见图 11。

```
j := 1;
/*
k := 0; //  若不活跃，可以删除
i := 6; //  若不活跃，可以删除
*/
repeat
    j := j + 1;
until ( 6=j );
```

图 10