如何构建SSA形式的CFG

孙子平

最近更新于 2022年8月12日 · 14 分钟阅读时长 · 静态分析

这篇文章是关于如何从非SSA（静态单一赋值）形式的CFG（控制流图）构造出SSA形式的控制流图。这主要涉及到图论中的Dominator理论。难点在于phi函数的插入。

1 简介

SSA中的每个变量仅被定义一次。SSA形式的代码极大地降低了定义使用链的可能数目。在传统的非SSA形式的代码中，如果有D处定义和U处使用，就可能有$D \times U$种可能的组合。因而SSA形式的代码有利于程序的优化和分析。

顺序执行的代码SSA形式较为简单。但程序会有分支和合并，通过在合并处插入ϕ函数，就能解决带分支代码的SSA形式。ϕ函数表示从进来的分支中选取某一个值作为新的值。如下面的代码：

```
if (p)
  v = 1;
else
  v = 2;
return v;
```

就会被转化成：

```
if (p)
  v1 = 1;
else
  v2 = 2;
v3 = phi(v1, v2);
return v3;
```

使用SSA形式中的一个分析例子是常量传播分析。常量传播分析是指分析哪些变量是常量，对于非SSA形式的分析，这较为困难。对于SSA形式，我们可以将那些使用常量定义的变量，将其所有出现的地方替换成常量，不断迭代直到到达不动点即可。

1.1 何处安放ϕ函数

假设V在程序中只有一处赋值。那么V的值要么是程序开始处的V_0，要么是被赋值后的V_1（注：这里可能在原作者[1]眼中所有的变量都是在程序入口处有定义的，见控制流图（CFG））。假设X是给V赋值的基本块，那么对于X严格支配的基本块Y，它见到的值一定是V_1。如果控制流跑到了Z，而Z不被X严格支配，且Z是这个路径中的第一个，那么Z即可能从X看到V_1又可能从程序开始处看到V_2。Z被称为X的支配边界（dominance frontier），需要添加ϕ函数。**因此总的来说，我们可以寻找到给V赋值的基本块的所有支配边界，它们就是需要插入ϕ函数的地方。**

使用支配边界进行SSA计算的这个想法也适用于计算**控制依赖**。**控制依赖**可以确定语句执行的条件。

2 控制流图（CFG）

程序的语句可以被组织成基本块，控制流从基本块的第一个语句进入，到最后一条语句流出。CFG是一个有向图，其节点除了基本块外，还有Entry和Exit节点。Entry到程序的任何入口基本块会有一条边，程序的任何出口到基本块会有一条边。此外还有一条从Entry到Exit的边，原因之后解释。其他的边代表执行流的跳转。一个拥有多个后继的节点称为**分支**，一个拥有多个前驱的节点称为**合并**。每个节点在Entry节点都会有一个赋值，代表程序进入时它的值，这个赋值与其他赋值同等对待。

我们使用$p : X_0 \xrightarrow{*} X_J$代表一般的路径（可空，长度$J$的路径包含$J+1$个节点和$J$个边），使用$p : X_0 \xrightarrow{+} X_J$代表非空路径。

对于两个非空路径$p : X_0 \xrightarrow{+} X_J$和$q : Y_0 \xrightarrow{+} Y_K$，我们说它们**交汇**于节点$Z$如果：

$$\begin{cases} X_0 \neq Y_0 \\ X_J = Z = Y_K \\ (X_j = Y_k) \Rightarrow (j = J \vee k = K) \end{cases}$$

直觉来说，就是p和q从不同的节点出发，然后没有在中间交于相同的节点，只是在最后交于Z，然后其中有的边可能包含循环$Z \xrightarrow{+} Z$的路径。

3 静态单一赋值形式（SSA）

一个赋值语句A形如$LHS(A) \leftarrow RHS(A)$。其中LHS是一个互异的目标变量元组，而RHS是一个表达式元组，两个元组长度相等。语义上，RHS中的每一个表达式都赋值给了对应的LHS的目标变量。

将程序转换成SSA形式分为两步，首先，一些平凡的ϕ函数被插入，形如$V \leftarrow \phi(V, V, \ldots)$。第二步则是替换所有的$V$为新的变量$V_i$，这里被替换的$V$包括分支语句中出现的和赋值中出现的。因而，本文中，一个赋值可能是个普通赋值或者ϕ赋值。

先前提到的ϕ赋值有如下的形式$V \leftarrow \phi(R, S, \ldots)$，其中$V, R, S, \ldots$是变量，$\phi$赋值位于基本块的开始。右手边变量的个数应当与进入基本块的前驱数目相等，这里要求基本块的前驱以某种形式排序。如果控制流从第j个前驱流入，那么V的取值就是右手边第j个变量。

SSA形式可以被看作一个程序的性质，或者一个从不具备该性质的程序到具备的变换。作为变换，它要求新程序满足以下的性质，对于每一个原始程序的变量V：

1. 如果$X \xrightarrow{+} Z$和$Y \xrightarrow{+} Z$交汇于节点Z，且X和Y包含了对V的赋值（原始程序中的），ϕ赋值应当被插入到Z中（新程序）；
2. 每一个对V的使用（包括ϕ函数）都被替换为V_i，使程序成为静态单一赋值；
3. 沿着任何控制流路径，源程序中的V的取值于新程序中的V_i的取值必须一样。

最小SSA形式是指插入的ϕ函数尽可能少。一些没有必要的ϕ函数可能会影响程序的优化。另一种是**修剪的SSA形式**，它是指如果变量没有在交汇点Z中及之后使用，就删掉ϕ语句。不过有时我们会需要在所有交汇的地方放置ϕ函数，但本文的算法经过微小的改动就可以得到修剪的SSA形式。

3.1 其他的程序结构

对于数组，将数组元素视为变量会很不方便，因而可以引入两个函数$Access(A, i)$表示访问数组A的第i个元素，其返回值就是A的第i个元素的值，$Update(A, j, V)$表示修改数组A的第j个元素，将其值置为V，并返回新的数组A。所以对A某个元素的赋值相当于对整个数组A赋值。

结构体大体上可以看成是数组。

除此之外，可能存在到变量的隐式引用，比如全局变量被子过程的使用和改变、变量别名、解引用指针等等。对于语句S，3中类型的引用影响到了到SSA形式的转换：

- $MustMod(S)$：一定被S修改的变量集合；
- $MayMod(S)$：可能被S修改的变量集合；
- $MayUse(s)$：在S执行之前的值可能被S用到的变量集合。

将 S 转 化 为 赋 值 语 句 A 时 ， $MayMod(S)$ 中 的 所 有 变 量 应 当 出 现 在 $LHS(A)$ 中 ， $MayUse(S) \cup (MayMod(S) - MustMod(S))$的所有变量出现在$RHS(A)$中（这部分我不太理解）。

对于堆内存的访问，将堆内存视为一整个变量对于大多优化算法足够。如果我们不能获取到函数体，那就要假定所有的全局变量及参数引用的对象会被改变，而调用者的局部变量不应假定为会改变。当然更细致的堆内存模型和别名分析是很有帮助的。更细致的分析可以减少副作用以及减少LHS和RHS元组的长度。

3.2 SSA算法概览

1. 从CFG中构造出支配边界映射；
2. 使用支配边界插入ϕ函数；

3. 重命名变量。

4 支配

4.1 支配者树

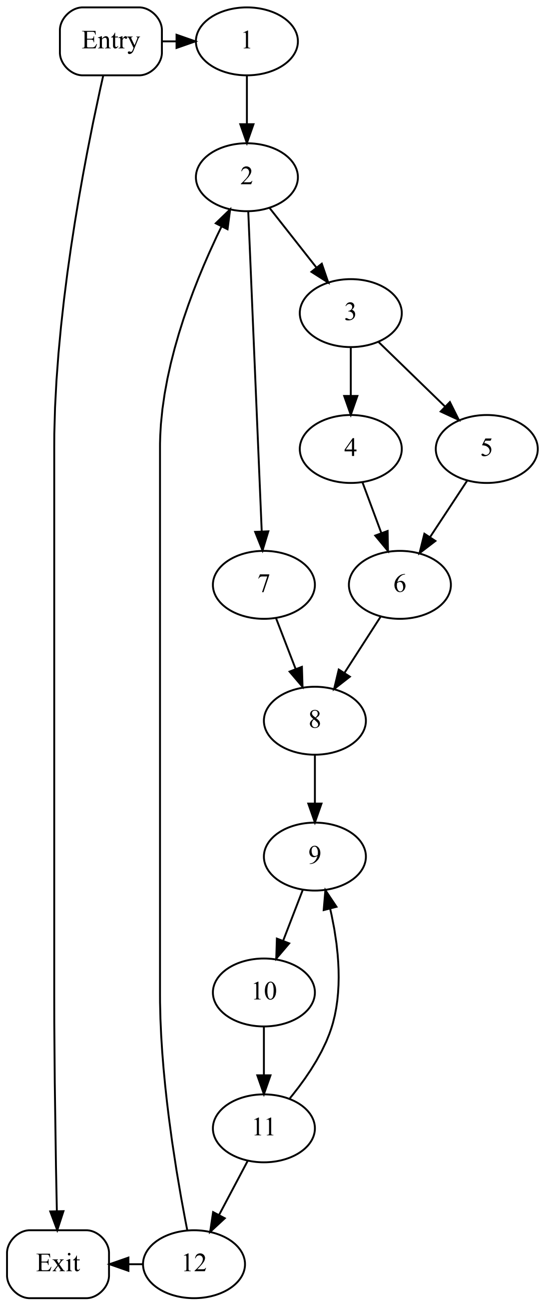
图论中的相关概念：

- **支配**：x支配$y \Leftrightarrow$ 从起始节点到y的每条路径都经过了x，记为$x \geqslant y$；从定义来说$\forall x, x$支配x；这是一个偏序关系（满足自反、传递）。
- **严格支配**：x严格支配$y \Leftrightarrow x$支配$y \wedge x \neq y$，记为$x \gg y$；如果x不严格支配y，则记为$x \not\gg y$。
- **支配边界**：$y \in x$的支配边界$\Leftrightarrow x$支配了y的前驱节点，但d没有严格支配y；从定义来说x的支配边界可能包含x自己；直观理解支配边界就是支配从有到无的界线。
- **立即支配者**：x是y的立即支配者$\Leftrightarrow x$严格支配y且$\forall z$严格支配y，x不严格支配z；我们会用idom来表示立即支配者；直观理解y的idom就是离y最接近的严格支配y的节点；一个节点的idom是唯一的。
- **支配者树**：每个节点的立即支配者组成了一棵树（支配的偏序确保是有向无环的，idom的唯一进而确保是棵树）。

注意支配的概念是对于一个有**起始节点**的有向图的。

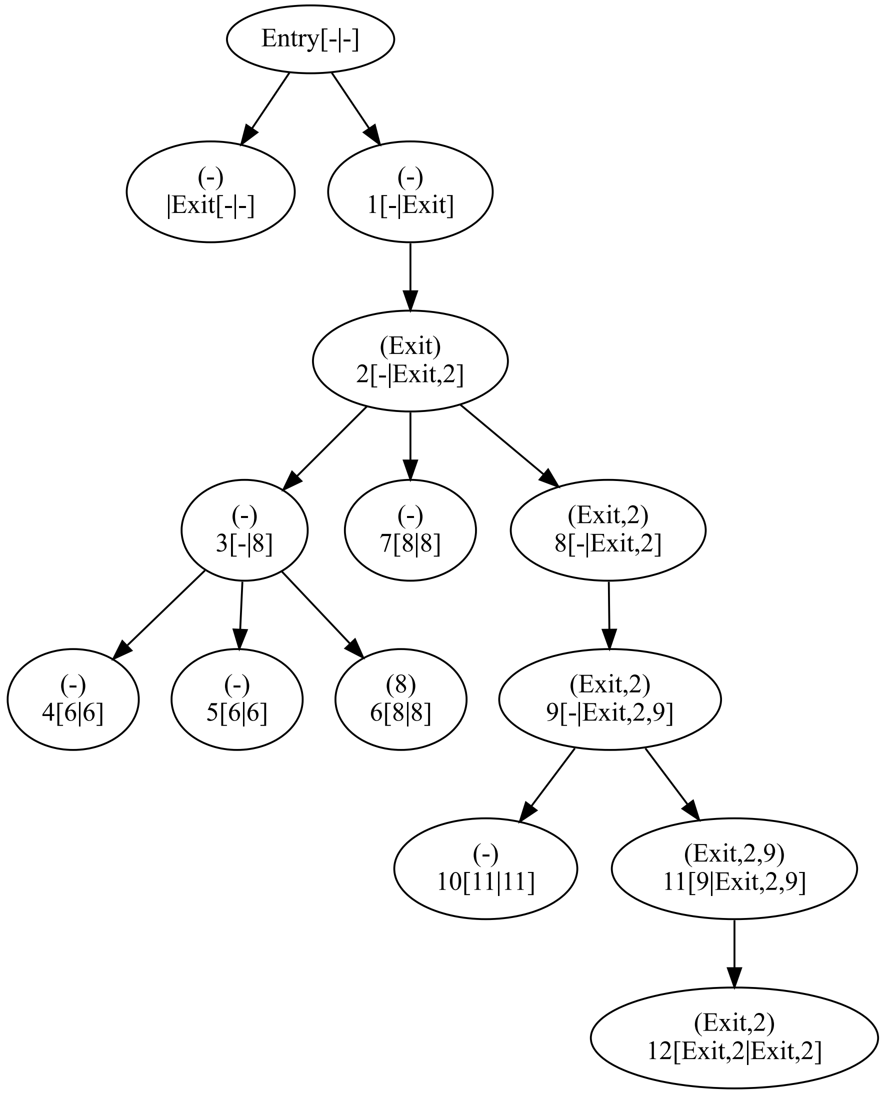
在CFG中，支配者树的根是Entry，除了Entry外，其他节点都有idom。支配者树可以在$O(E\alpha(E, N))$的时间内给出，甚至可以用更复杂的算法在$O(E)$时间内给出。由于$\alpha(E, N)$很小，我们假定支配者树是线性时间内求解的。

考虑下面的图：



其支配者树如下，其中节点X的标签为：

$$
\begin{gathered}
(DF_{up}(X)) \\
X[DF_{local}(X)|DF(X)]
\end{gathered}
$$



下文中，前驱$Pred$、后继$Succ$和路径这些名词是CFG上的，而父亲$Parent$、孩子$Children$、祖先、子孙这些名词是指支配者树的。关于支配者树的计算我将在稍后给出。

4.2 支配边界

首先我们给出支配边界$DF(X)$的形式化定义：

$$DF(X) = \{Y | \exists P \in Pred(Y)(X \geqq P \wedge X \not\gg Y)\}$$

直接依据定义计算支配边界会具有很高的复杂度（二次复杂度）。为了以线性于$\sum_X |DF(X)|$的速度计算支配边界，我们对每个节点定义两个中间的集合DF_{local}和DF_{up}，使得：

$$DF(X) = DF_{local}(X) \cup \bigcup_{Z \in Children(X)} DF_{up}(Z)$$

对于任意节点X，一些X的后继可能会对$DF(X)$有贡献，这种贡献$DF_{local}(X)$被定义为：

$$DF_{local}(X) \stackrel{\text{def}}{=} \{Y \in Succ(X) | X \not\gg Y\}$$

对于任意非Entry的节点Z，$DF(Z)$中的一些节点或许会对$DF(idom(Z))$，这种贡献$DF_{up} = (Z)$被定义为：

$$DF_{up}(Z) \stackrel{\text{def}}{=} \{Y \in DF(Z) | idom(Z) \not\gg Y\}$$

引理1： 公式1是正确的。

引理1证明： 由于支配关系是自反的，所以公式2中，X支配自己，故而$DF_{local}(X) \subseteq DF(X)$。由于支配关系是传递的，所以公式3中的$Z$严格支配$Y$的前驱而$X = idom(Z)$，故而$DF_{up}(X) \subseteq DF(X)$。类似的，我们可以证明$X$的支配边界在其前驱为$X$的情况下在$DF_{local}(X)$中，否则在$DF_{up}(X)$中。

引理2： 对于任意节点X，

$$DF_{local}(X) = \{Y \in Succ(X) | idom(Y) \neq X\}$$

引理3： 对于任意节点X和它的任意孩子Z（支配树上的），

$$DF_{up}(Z) = \{Y \in DF(Z) | idom(Y) \neq X\}$$

引理3证明： 推导公式4可以推导出公式3较为复杂。使用反证法。

于是就有了下方计算$DF(X)$的算法：

1. 自底向上遍历支配者树上的每个节点X：
    1. $DF(X) \leftarrow \emptyset$
    2. 对于每个$Y \in Succ(X)$：
        1. 如果$idom(Y) \neq X$，则$DF(X) \leftarrow DF(X) \cup \{Y\}$（计算$DF_{local}(X)$）
    3. 对于每个$Z \in Children(X)$：
        1. 对于每个$Y \in DF(Z)$：
            1. 如果$idom(Y) \neq X$，则$DF(X) \leftarrow DF(X) \cup \{Y\}$（计算$DF_{up}(X)$）

$DF_{local}(X)$总的计算时间为$O(|E|)$（E为CFG的边集），$DF_{up}(X)$总的计算时间正比于所有DF的大小和，最坏情况为$O(|N|^2)$（N为CFG的顶点集），但通常而言，$DF_{up}(X)$的计算时间是线性的。

4.3 支配边界与合并的关系

对于CFG上的节点集合S，$J(S)$是它们的合并节点Z，也就是存在两个非空的CFG路径，从S中不同的两点出发，交汇在Z。而$J^+(S)$被定义为下列序列的极限（其实是闭包）：

$$\begin{cases} J_1 = J(S) \\ J_{i+1} = J(S \cup J_i) \end{cases}$$

特别的，如果S是某变量V的赋值节点集合，$J^+(S)$是V的ϕ函数节点集合。

同时，我们定义节点集合上的DF：

$$DF(S) = \bigcup_{x \in S} DF(X)$$

同样就可以定义$DF^+(S)$为下列序列的极限：

$$\begin{cases} DF_1 = DF(S) \\ DF_{i+1} = DF(S \cup DF_i) \end{cases}$$

这里只是给出一个定义，并不是最快的计算方法。

如果S是某变量V的赋值节点集合，我们会证明（这个定理依赖于V的定义在Entry）：

$$J^+(S) = DF^+(S)$$

引理4： 对任意CFG中非空路径$p : X \xrightarrow{+} Z$，存在路径上的一个节点$X' \in \{X\} \cup DF^+(\{X\})$支配$Z$。除非$X$支配$p$的每个节点，$X' \in DF^+(X)$。

引理5： 对CFG中两个不同的节点X, Y，若有非空路径$p : X \xrightarrow{+} Z$和$q : Y \xrightarrow{+} Z$交汇于Z。那么$Z \in DF^+(\{X\}) \cup DF^+(\{Y\})$。

引理5证明： 假设X'和Y'分别是引理4中p和q存在的节点。X'在q上时，即$X' = Z$，只需要考虑$Z = X$的情况，此时$Z \in DF(X) \subseteq DF^+(X)$。同理$Y'$在$p$上也成立。如果$X'$不在$q$上且$Y'$不在$p$上，则可以推导出$X'$和$Y'$支配$Z$，进而推导出$X'$支配$Y'$或$Y'$支配$X'$，与交汇定义矛盾（存在中间交点）。

引理6： 对于任意CFG节点集合S，$J(S) \subseteq DF^+(S)$。

引理7： 对于任意包含$Entry$的CFG节点集合S，$DF(S) \subseteq J(S)$。

定理： 对于任意包含$Entry$的CFG节点集合S，$DF^+(S) = J^+(S)$。

5 构造最小SSA形式

5.1 使用支配边界寻找ϕ函数需要的地方

接下来给出放置平凡ϕ函数的算法，它需要用到以下3个数据结构：

- $Worklist : Queue\langle CFGNode\rangle$
- $Visited : Map\langle CFGNode, bool\rangle$
- $Placed : Map\langle CFGNode, bool\rangle$

算法如下：

1. 对于每个变量V：
    1. $Worklist \leftarrow V$的所有赋值节点
    2. $Visited \leftarrow$全$false$
    3. $Placed \leftarrow$全$false$
    4. 如果$Worklist.empty()$为假
        1. $X \leftarrow Worklist.pop()$
        2. 对每个$Y \in DF(X)$，如果$Placed[Y]$为假
            1. $Placed[Y] \leftarrow true$
            2. 在Y处放置$\langle V \leftarrow \phi(V, \ldots, V)\rangle$
            3. 如果$Visited[Y]$为假
                1. $Visited[Y] \leftarrow true$
                2. $Worklist.push(Y)$

这个算法的复杂度为$O(\sum_X (A_{tot}(X) \times |DF(X)|))$。这里$A_{tot}$是总的赋值数目（包括$\phi$），一般情况下，这个算法线性于$A_{tot}(X)$。

5.2 重命名

我们给出一个递归函数$Search(X : CFGNode)$，它有唯一的参数X是一个CFG节点。此外还有以下的“全局变量”作为上下文：

- $Stacks : Map\langle Variable, Stack\langle Integer\rangle\rangle$
- $Counters : Map\langle Variable, Integer\rangle$

首先：

1. $Stacks \leftarrow$全为空栈
2. $Counters \leftarrow$全为0
3. 调用$Search(Entry)$

$Search(X : CFGNode)$实现如下：

1. 对于每个$A : Statement \in X$：
    1. 如果A是个普通赋值：
        1. 对每个$V : Variable \in RHS(A)$
            1. 使用V_i替换V，其中i是$Stacks[V].top()$
    2. 对每个$V : Variable \in LHS(A)$：
        1. 使用V_i替换V，其中i是$Counters[V]$
        2. $Stacks[V].push(Counters[V])$
        3. $Counters[V] += 1$
2. 对于每个$Y : CFGNode \in Succ(X)$：
    1. $j \leftarrow$基本块X到Y的边的序号
    2. 对每个$F : \phi$函数$\in Y$：
        1. 使用V_i替换$RHS(F)$的第j个操作数，其中i是$Stacks[V].top()$
3. 对于每个$Y : CFGNode \in Children(X)$：
    1. 调用$Search(Y)$
4. 对于每个$A : StatementinX$
    1. 对每个$V : Variable \in oldLHS(A)$：
        1. $Stacks[V].pop()$

6 控制依赖的构建

控制依赖是反向控制流图的支配边界。类似地我们定义反向支配和立即反向支配者等概念。

一个CFG节点Y被认为是**控制依赖**于X，如果满足下面两条：

1. 存在一个非空路径$p : X \stackrel{+}{\to} Y$，使得$Y$反向支配$X$之后的$p$上所有节点。
2. Y没有严格反向支配X。

这等价于X的某条出边使得Y一定被执行，但也存在一些从X出发的路径Y不被执行。

X, Y 是 CFG 节 点 ， 那 么 Y 控 制 依 赖 于 X ， 当 且 仅 当 在 RCFG 中 $X \in DF(Y)$ 。 因 而 计 算 $CD : MultiMap\langle CFGNode, CFGNode\rangle$控制依赖算法如下：

1. 对每个CFG节点Y：
    1. 对每个$X \in RDF(Y)$：
        1. $CD.put(X, Y)$

通过添加$Entry \to Exit$的边，控制依赖的根将成为$Entry$。

1. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4), 451-490. ↩

编辑本页

静态分析

孙子平

静态分析方向研究生

cv

下一页

从CFG直接构建GSA的算法

上一页

数值分析基础第1章：引论

相关

- 编程语言入坑指南
- Tarjan的支配树算法
- 构建数据依赖的实现
- Packrat记忆化Parser与Recoverable Parser
- 区间分析