Static Single Assignment Form

Luka Stojanović

luka@magrathea.rs
Seven Bridges Genomics

Scalability#06
23. 12. 2012.

Outline

What is SSA Form?

SSA Form

Property of compiler's IR, which says that each variable is assigned exactly once.

- Developed during 1980's and described in the paper *Efficiently computing static single assignment form and the control dependence graph* by Ron Cytron , Jeanne Ferrante , Barry K. Rosen , Mark N. Wegman , F. Kenneth Zadeck from 1991.
- Convenient for various optimization techniques.

What is SSA Form

$$
\begin{array}{l}
i := 0 \\
i := 5 \\
j := i
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
i_1 := 0 \\
i_2 := 5 \\
j_1 := i_2
\end{array}
$$

It becomes apparent that although i is used, i_1 isn't, and can be optimized away.

$$
\begin{array}{l}
\textbf{if}\ \text{condition}\ \textbf{then} \\
\quad i := 0 \\
\textbf{else} \\
\quad i := 5 \\
\textbf{end if} \\
j := f(i)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\textbf{if}\ \text{condition}\ \textbf{then} \\
\quad i_1 := 0 \\
\textbf{else} \\
\quad i_2 := 5 \\
\textbf{end if} \\
j_1 := f(i_?)
\end{array}
$$

What value of i is used in this expression?

What is SSA Form

$$
\begin{array}{l}
i := 0 \\
i := 5 \\
j := i
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
i_1 := 0 \\
i_2 := 5 \\
j_1 := i_2
\end{array}
$$

It becomes apparent that although i is used, i_1 isn't, and can be optimized away.

$$
\begin{array}{l}
\textbf{if } \text{condition } \textbf{then} \\
\quad i := 0 \\
\textbf{else} \\
\quad i := 5 \\
\textbf{end if} \\
j := f(i)
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\textbf{if } \text{condition } \textbf{then} \\
\quad i_1 := 0 \\
\textbf{else} \\
\quad i_2 := 5 \\
\textbf{end if} \\
i_3 := \phi(i_1, i_2) \\
j_1 := f(i_3)
\end{array}
$$

There is an artificial instruction ϕ inserted to designate that values from multiple branches are merged at this spot.

Optimizations

- Constant propagation
- Value range propagation
- Global value numbering
- Strength reduction
- Dead code elimination
- Register allocation

Register allocation

Graph Coloring

Graph Coloring is a problem of coloring the vertices of a graph such that no two adjacent vertices share the same color.

Register allocation can be abstracted into graph coloring problem using these abstractions:

- A variable corresponds to a node in an undirected graph.
- If two variables are alive at the same time, their nodes are connected by an edge.

Since graph coloring is NP-complete, register allocation is calculated using a heuristic.

Register allocation

If program is in SSA form, corresponding variable graph is ,,chordial", and problem of it's coloring becomes polynomial.

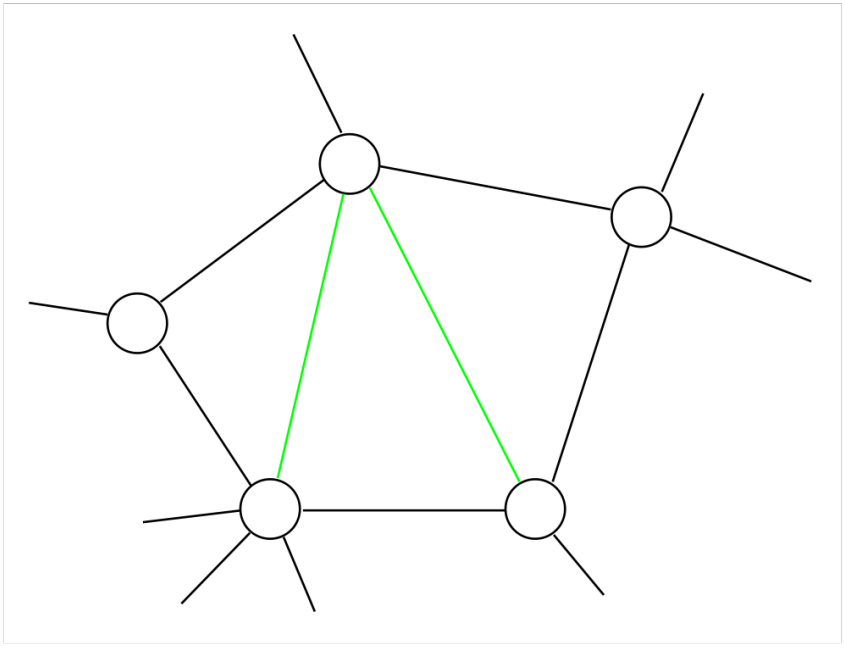


Figure: Chordial graph

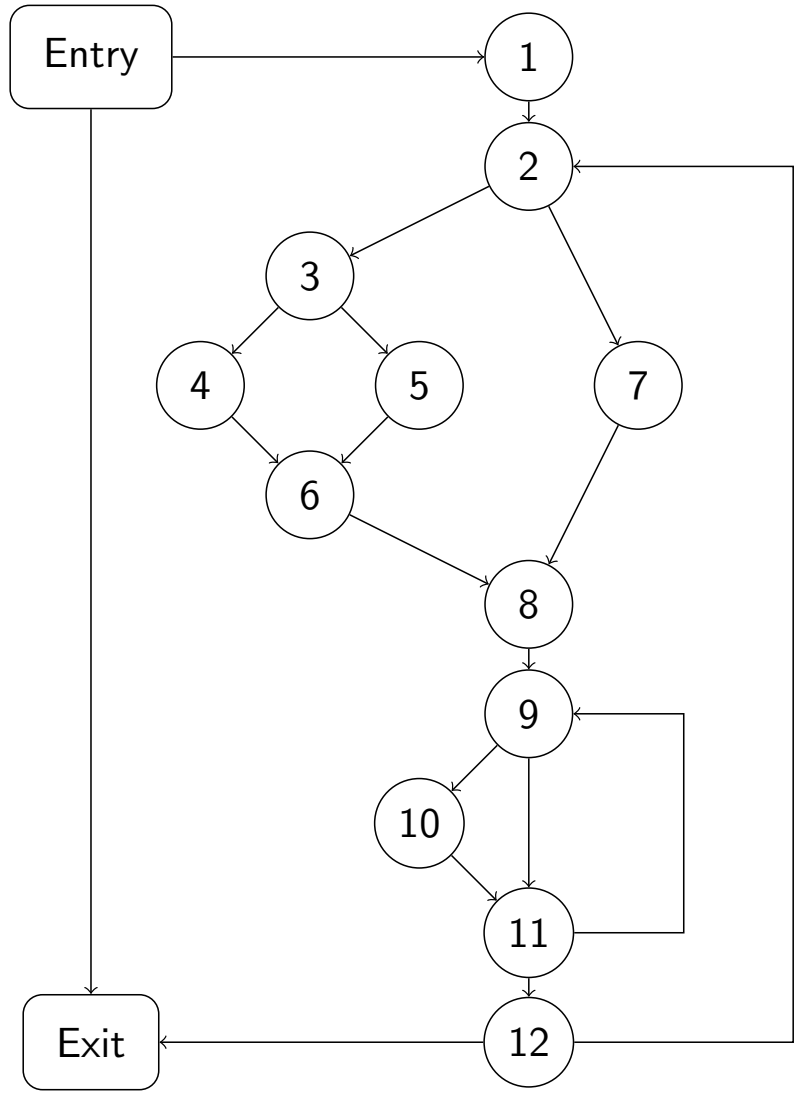
Computing SSA Form

Definition

A *control flow graph* is a directed graph whose nodes are the basic blocks of program, and two additional nodes **Entry** and **Exit**.

Control Flow Graph

```
i := 1                          (1)
j := 1                          (1)
k := 1                          (1)
l := 1                          (1)
repeat                          (2)
    if P then                   (2)
        j := i                  (3)
        if Q then               (3)
            l := 2              (4)
        else                    (5)
            l := 3              (5)
        end if                  (6)
        k := k + 1              (6)
    else                        (7)
        k := k + 2              (7)
    end if                      (8)
    print(i, j, k, l)           (8)
    repeat                      (9)
        if R then               (9)
            l := l + 4          (10)
        end if                  (11)
    until S                     (11)
    i := i + 6                  (12)
until T                         (12)
```



SSA

```
i := 1
j := 1
k := 1
l := 1
repeat
    if P then
        j := i
        if Q then
            l := 2
        else
            l := 3
        end if
        k := k + 1
    else
        k := k + 2
    end if
    print(i, j, k, l)
    repeat
        if R then
            l := l + 4
        end if
    until S
    i := i + 6
until T
```

```
i_1 := 1
j_1 := 1
k_1 := 1
l_1 := 1
repeat
    i_2 := φ(i_3, i_1)
    j_2 := φ(j_4, j_1)
    k_2 := φ(k_5, k_1)
    l_2 := φ(l_9, l_1)
    if P then
        j_3 := i_2
        if Q then
            l_3 := 2
        else
            l_4 := 3
        end if
        l_5 := φ(l_3, l_4)
        k_3 := k_2 + 1
    else
        k_4 := k_2 + 2
    end if
    j_4 := φ(j_3, j_2)
    k_5 := φ(k_3, k_4)
    l_6 := φ(l_2, l_5)
    print(i_2, j_4, k_5, l_6)
    repeat
        l_7 := φ(l_9, l_6)
        if R then
            l_8 := l_7 + 4
        end if
        l_9 := φ(l_8, l_7)
    until S
    i_3 := i_2 + 6
until T
```

Control Flow Graph

Definition

For any non-negative integer J, a *path of length* J consists of a sequence of $J+1$ nodes $(X_0, \ldots, X_J)$ and J edges $(e_1, \ldots, e_J)$.

Node or edge may repeat several times within same sequence. The *null* path, with $J = 0$ is allowed. We write $p : X_0 \xrightarrow{*} X_J$ for unrestricted path and $p : X_0 \xrightarrow{+} X_J$ for paths known to be non-null.

1. If two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at node Z, and nodes X and Y contain assignments to V, then $V := \phi(V, \ldots, V)$ is inserted in Z
2. Each mention of V is replaced with V_i
3. For every V in original program, V_i in the new program must have the same value.

Minimal SSA Form

- The *dominance frontier* mapping is constructed from CFG
- Using dominance frontiers, locations of ϕ-functions is calculated
- Variables are renamed

Definition

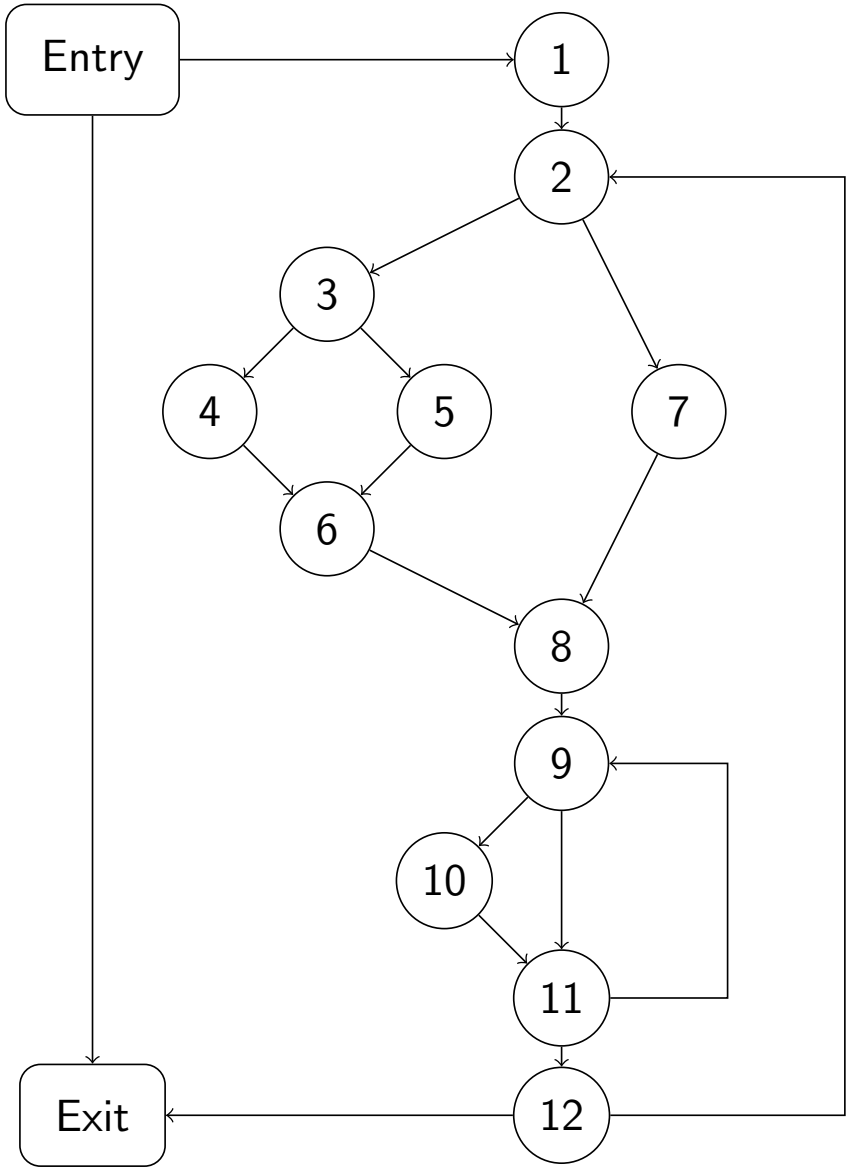
The *dominance frontier* $DF(X)$ is the set of all CFG nodes Y such that X dominates predecessor of Y but doesn't strictly dominate Y

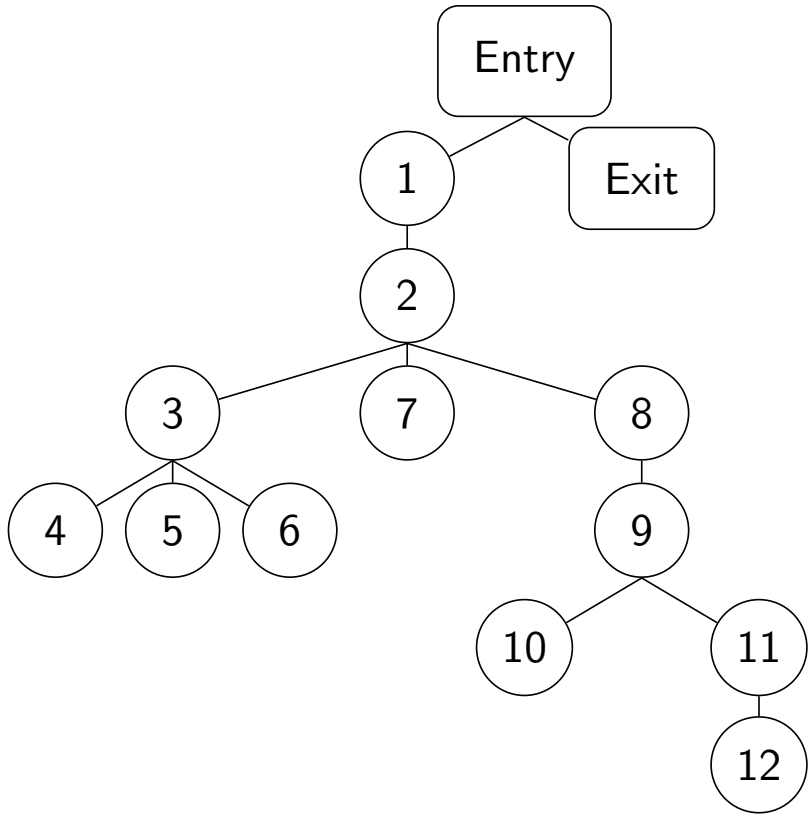
Definition

The *dominance frontier* $DF(X)$ is the set of all CFG nodes Y such that X dominates predecessor of Y but doesn't strictly dominate Y

$DF(X(= \{Y|(\exists P \in Pred(Y))(X \geqslant PandX \not\geqslant Y)\}$

Dominator Tree





Placement of ϕ functions

$IterCount := 0$
for each node X **do**
    $HasAlready[X] := 0$
    $Work[X] := 0$
end for
$W := \emptyset$
for each variable V **do**
    $IterCount := IterCount + 1$
    for each $X \in A(V)$ **do**
        $Work[X] := Work[X] + 1$
        $W \cup \{X\}$
    end for
    while $W \neq \emptyset$ **do**
        take X from W
        for each $Y \in DF(X)$ **do**
            if $HasAlready[Y] < IterCount$ **then**
                place $\langle V := \phi(V, \ldots, V) \rangle$ at Y
                $HasAlready[Y] := IterCount$
                if $Work[Y] < IterCount$ **then**
                    $Work[Y] := IterCount$
                    $W \cup \{Y\}$
                end if
            end if
        end for
    end while
end for

Where is it used

- HotSpot JVM
- Mono JIT compiler
- Dalvik
- LuaJIT
- LLVM

LLVM

,,The LLVM Project is a collection of modular and reusable compiler and toolchain technologies"

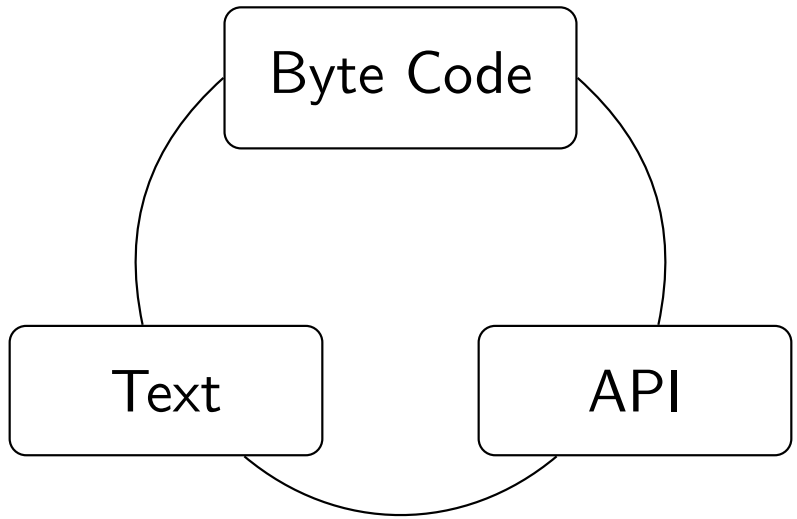


Figure: Three states of LLVM's IR

LLVM

- Three-code instructions
- Load/Store
- Infinite number of registers

SSA in LLVM

As SSA form applies only to registers, the usual way for generating code is to allocate local variables in memory and store results. Optimizer pass will remove unnecessary memory round-trips, generate ϕ instructions, etc...