

CSC2/455 Software Analysis and Improvement

Dominators and SSA Form - II

Sreepathi Pai

February 6, 2022

URCS

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

Dominators

- A node n in the CFG dominates a node m iff:
 - n is on all paths from entry to m
 - by definition, a node n always dominates itself
 - if $n \neq m$, then n *strictly* dominates m
- Computed using a dataflow-style analysis
 - Each node annotated with a set of its dominators

Static Single Assignment Form

- Simple algorithm to generate SSA form
 - Introduce ϕ functions
 - Rename variables using Reaching Definitions
- Algorithm can generate excessive ϕ functions
 - TODAY: Use *dominance frontiers* to place the minimal number of ϕ functions
- Also today: Removing ϕ functions
 - Machines don't support ϕ functions, so we must emulate them

Maximal SSA Form

- Insert ϕ nodes for each definition at every join node
- Rename LHS
- Rename RHS using reaching definitions

Reducing the number of ϕ nodes

- Why insert ϕ nodes at only join nodes?
- Can we skip inserting ϕ nodes for a definition at some join node?

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

Dominance Frontiers

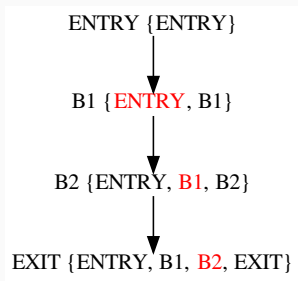
- The dominance frontier of a node n ($DF(n)$) is a *set* of nodes
- A node $m \in DF(n)$ iff:
 - n does not strictly dominate m
 - n dominates q where $q \in \text{pred}(m)$
- Note that dominance frontiers only contain *join* nodes
 - I.e. nodes with multiple predecessors
- Computing the dominance frontier of each node:
 - Iterative Data-flow analysis?

Dominance Frontiers: Direct algorithm

Direct calculation of dominance frontiers using *dominator trees*.

Immediate Dominators

- The *immediate* dominator of a node m ($IDOM(m)$) is the node n :
 - such that n strictly dominates m , and
 - n does not strictly dominate o where $o \in (DOM(m) - \{m\})$
 - in some sense, n is the “closest” dominator in the CFG to m .
- By definition, ENTRY has no immediate dominator



Not Strictly Dominates

- n strictly dominates m
 - $SDOM(n, m) = n \in DOM(m) \wedge n \neq m$
- n does not strictly dominate m
 - $\neg SDOM(n, m) = n \notin DOM(m) \vee n = m$

Dominator Tree

- Note that each node in the CFG can have only one immediate dominator
 - Can you see why?
- Create a graph $G = (V, E)$, where:
 - V is the set of basic blocks
 - There is an edge (n, m) in E if n is the immediate dominator of m (i.e. $IDOM(m) = n$)

ENTRY



B1

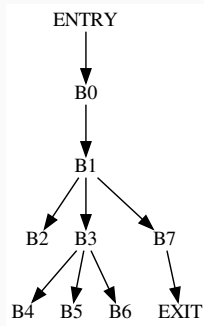
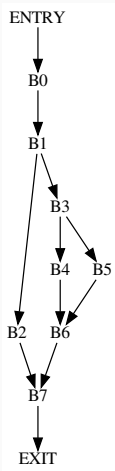


B2



EXIT

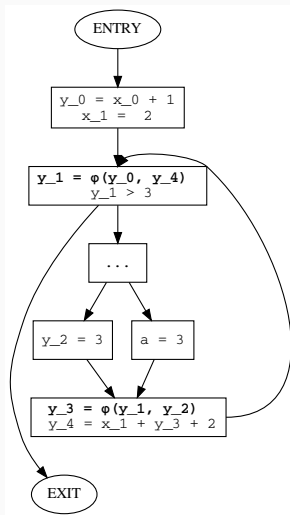
Example: CFG and its dominator tree



Computing the dominance frontier

- Find all join nodes in CFG, e.g. j
- For all nodes n that dominate predecessors of j (in the CFG)
 - If n does not strictly dominate j , add j to $DF(n)$
- This last step can be operationalized over all predecessors p of j in the CFG:
 - Start traversing the dominator tree at p
 - If p is $IDOM(j)$, stop. Otherwise add j to $DF(p)$
 - Repeat by moving up the dominator tree until you reach $IDOM(j)$

Example: Non-redundant ϕ functions



Placing ϕ functions

- For each definition d in basic block n :
 - Place a ϕ function for d in all nodes m where $m \in DF(n)$
 - Note that each ϕ function is also a definition!
 - Repeat, until no more ϕ functions need to be inserted
- This is the minimal number of ϕ functions for a definition d structurally
 - Can we further reduce the overall number of ϕ functions?
- (Figure 9.9 in Cooper and Turczon)

Other optimizations

- Dead definitions
 - Definitions that are not read (i.e. overwritten) do not need ϕ functions
- Two forms:
 - *Semi-pruned SSA* form, using “globals” names (those variables that are live in to a block)
 - *Pruned SSA* form, using `LIVEOUT` information

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

Renaming variables

- SSA form introduced “subscripts” for each variable
- Should we drop them when generating code?

```
a_0 = x_0 + y_0  
b_0 = a_0  
a_1 = 17  
c_0 = a_0
```

Problem with dropping subscripts

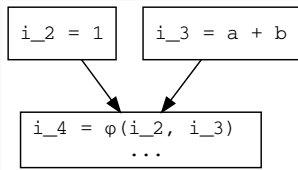
```
a = x + y
b = a
a = 17
c = a    # WRONG!
```

Handling subscripts

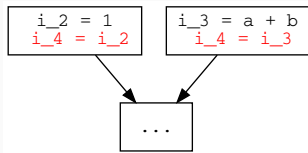
- Each definition becomes a new variable
 - I.e. Do NOT drop subscripts
- Preserves data dependences
 - Esp. important when we aggressively move code from basic blocks (e.g. very busy expressions, loop invariant code motion, etc.)

Code for ϕ functions

- Introduce copies along each incoming edge to a join node



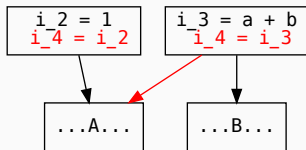
Inserting appropriate copies along incoming edges



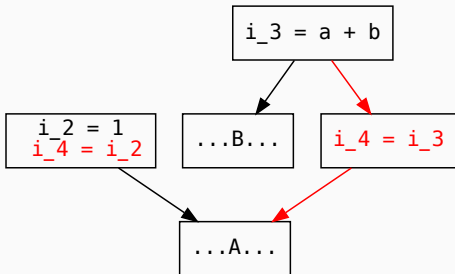
Critical edges

- Executing ϕ functions by inserting copies into predecessor blocks is not always correct
- If such a predecessor block has multiple successors, then the ϕ function may execute when it shouldn't
 - This *may* be harmless, but not always
- Edges connecting such predecessors to the block containing the ϕ function are called *critical* edges

Critical Edges: Example



Splitting critical edges



- Such edges need to be *split* by inserting a block on that edge
- See the discussion in Cooper and Turczon for more details and an example

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

Purely Functional Programs

- Everything is a value
- No “assignment”, just binding values to names
- No control flow such as jumps
 - Must be emulated using functions

Example: Factorial

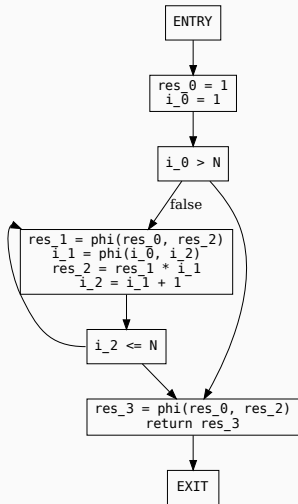
```
def fact(N):  
    res = 1  
    for i in range(1, N+1):  
        res *= i  
  
    return res  
  
def fac(N):  
    return 1 if N <= 1 else N * fac(N - 1)
```

Factorial: 3 Address Code

```
def fact(N):  
    res = 1  
    i = 1  
    if i > N goto loop_end  
  
    loop_head:  
        res = res * i  
        i = i + 1  
        if i <= N goto loop_head  
  
    loop_end:  
        return res
```

Factorial: SSA form

```
def fact(N):  
    res_0 = 1  
    i_0 = 1  
    if i_0 > N goto loop_end  
  
    loop_head:  
        res_1 = phi(res_0, res_2)  
        i_1 = phi(i_0, i_2)  
  
        res_2 = res_1 * i_1  
        i_2 = i_1 + 1  
        if i_2 <= N goto loop_head  
  
    loop_end:  
        res_3 = phi(res_0, res_2)  
        return res_3
```



Factorial: Function Conversion

```
def fact(N):
    res_0 = 1
    i_0 = 1

    def loop_head(res_1, i_1):
        res_2 = res_1 * i_1
        i_2 = i_1 + 1
        return loop_head(res_2, i_2) if i_2 <= N else loop_end(res_2)

    def loop_end(res_3):
        return res_3

    return loop_end(res_0) if i_0 > N else loop_head(res_0, i_0)
```

- Each basic block is converted to a function
- Parameters to this function are the LHS of the ϕ functions in that BB
- Arguments picked from arguments of ϕ function depending on the path the BB was on.

Outline

Review

Dominance Frontiers and Dominator Trees

Emitting code for SSA form

The SSA Form and Functional Programming

Postscript

References

- Chapter 9 of Cooper and Turczon
 - Section 9.2.1
 - Section 9.3
- Optional:
 - Various authors, The SSA book
 - Andrew W. Appel, SSA is functional programming