CS412/413

Introduction to Compilers

Radu Rugina

Lecture 36: DU Chains and SSA Form

03 May 04

Outline

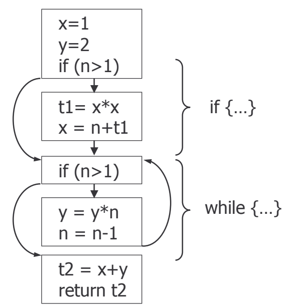
- Program representations:
  - DU chains
  - UD chains
  - Static Single Assignment
- Analysis using DU/UD chains, SSA

CFG Representation

- Accurate analysis: need a representation which captures program control flow
- Dataflow analysis uses CFG representation
  - Graph edges characterize control flow
- Issue: use control flow to compute data flow
- Consequences: analysis of a CFG subgraph may modify only a small fraction of the dataflow information
- Expensive to propagate all dataflow information along control flow when most of it remains unchanged
- … can't we explicitly compute data flow?

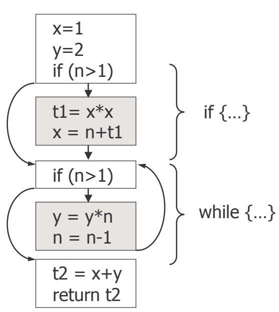
Example

```
int foo(int n) {
   int x=2, y=3;
   if (n>1) {
        x =n+x*x;
   }
   while (n>1) {
        y= y*n;
        n = n-1;
   }
   return x+y;
}
```



Example

- If statement:
  - modifies x, n, t1
  - doesn't use/define y, t2
- While statement:
  - modifies y, n
  - doesn't use/define x, t1, t2

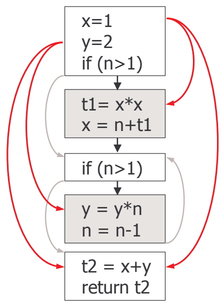


Definitions and Uses

- How can we avoid propagating the information through all CFG subgraphs?
- Solution: for each definition of a variable, identify all possible uses of that variable
  - Directly propagate the information from the definitions to the uses
  - Skip CFG subgraphs that don't define/use the variable

Definitions and Uses

- Uses of x = 1
  - t1=x*x, t2=x+y
  - no uses in while loop
- Uses of y = 2
  - y=y*n, t2=x+y
  - no uses in if statement



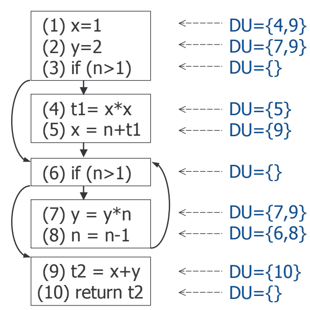
Def-Use Chains

- Use a list structure = def-use (DU) chain
  - For each definition d compute a chain (list) of uses that d may reach
  - Is a sparse representation of data flow
  - Compute information only at the program points where it is actually used!
- Once we compute DU chains, we don't need the CFG program representation to perform analysis
  - No need to compute information at each program point
  - Must re-formulate analysis algorithms using DU chains

Analysis Using DU Chains

- Can use a worklist algorithm to implement analysis
- Initialization: worklist = all instructions
- At each step:
  - Remove an instruction from the worklist
  - Compute effect of the instruction (transfer function)
  - Propagate information directly to all the uses (use the meet operator to merge information)
  - Add all the uses to the worklist
- Terminate when the worklist is empty

Example: DU Chains



DU and UD Chains

- UD chains: for each use compute the set of all definitions which may reach that use
- UD, DU chains:
  - Same info, encoded differently:

    UD[I] = { I′ | I ∈ DU[I′] }
  - Sparse representation of reaching definitions:

    DU[I] = { I′ | I ∈ RD before I′ and

    ∃ x . x ∈ def[I] ∩ use[I′]}

Static Single Assignment

- Idea: rewrite program to explicitly express the DU/UD relation in the code
- SSA form:
  - Each variable defined only once
  - Use ϕ-functions at control-flow join points
- UD relation: for each use of a variable, there is a unique definition of that variable
- DU relation: for each definition of a variable, there may be multiple uses of that definition
- Results in an implicit representation of DU/UD relation!

Example

Program

```
x = 0
y = 1

if (n>0)
   x = x+y
else
   y = y-x

n = x*y
```

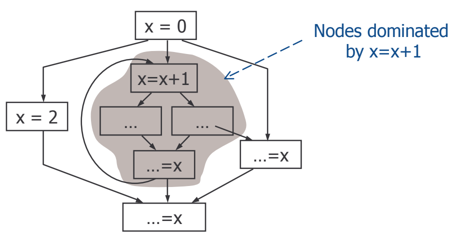
SSA Form

```
x1 = 0
y1 = 1

if (n1>0)
   x2 = x1+y1
else
   y2 = y1-x1

x3= φ(x1,x2)
y3= φ(y1,y2)
n2 = x3*y3
```

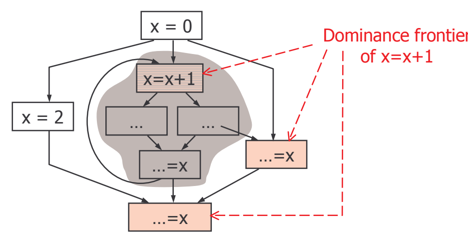

Placing ϕ Functions

- Placing ϕ-functions at each join point is inefficient
- Use dominator relation
- Dominance frontier of n = nodes w such that n dominates a predecessor of w, but does not strictly dominate w
- Rule: if node n defines variable x, then place a ϕ-function for x at each of the nodes in the dominance frontier of n
- Intuition:
  - if a definition x=… dominates node n then any path to n goes through that definition - no need to place any ϕ-function
  - place ϕ-functions at the nodes adjacent to the region of nodes dominated by x=…

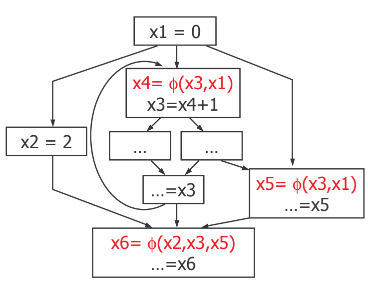
Dominator Relation



Dominance Frontier



Placing ϕ Functions



Space Requirements

- SSA representation requires less space than DU chains
- Consider N definitions of x which may reach M uses of x
- Space required for DU chain: N*M
- Space required for SSA form: usually linear in the program size (N+M)
- Example:
  if (…) x=1; if (…) x=2; …; if (…) x=10;
  if (…) y=x+1; if (…) y=x+2; …; if (…) y=x+20;

Analysis Using SSA Form

- Similar to analysis using DU chains
- If we want to compute some information for each variable (e.g. constant folding): keep a single set of values valid at all program points
- Flow of values explicitly represented ϕ-functions
  - Transfer function of ϕ-function is meet operation of arguments

Example

- Functions for x,y,n
- Variables after renaming: x1,x2,x3; y1,y2,y3; n1,n2,n3
- Constant folding:

  Iteratively compute constant values for x1-x3, y1-y3, n1-n3

```
x = 1
y = 2
n = 0
while (n<10) {
  x = y*y;
  y = x-y;
  n =n+1;
}
```


Aliasing and SSA

- Load and store instructions are problematic
  - Load: don't know which variable is actually used
  - Store: don't know which variable is actually defined
- Conservative approximation:
  - Load: insert a function which merges all variables
  - Store: insert a ϕ-function for each variable
- With pointer aliasing information:
  - Load: merge only the possible targets of the load
  - Store: insert ϕ-functions only for variables that may be modified
- Need to perform pointer analysis before translation to SSA
  - Alias analysis = fundamental analysis

Summary

- DU chains: sparse representation of data flow
  - Allow efficient implementation: information flows from definitions directly to the uses
  - Must compute DU chains first
- SSA: better representation
  - Smaller size than DU chains
  - Must efficiently place ϕ-functions
- Aliasing information required for either representation