50.054 Static Single Assignment

ISTD, SUTD

Learning Outcomes

1. Explain the characteristics of Static Single Assignment Form
2. Define Dominance Frontier and Iteratative Dominance Frontiers
3. Implement unstructured SSA construction algorithm

Recall Name Analysis

To analysis

- Variable's scope
- Variable's definition
- Variable's use

Static Single Assignment

It's an intermediate representation.

- Variables are defined/assigned (syntactically) once only
  - Immutablility (if *alpha* renaming is included)! Like FP
  - Re-assignment of the same variable must be renamed.
- Multiple re-definition of the same (original) variable are merged via ϕ assignments
- Make use-def explicit
- Optimization for
  - Register Allocation
  - Model Checking
- Improve accuracy for
  - Type analysis
  - Flow-insensitive analysis
- Compiling FP to SIMP and vice versa
- Code Obfuscation

Example

```
// PA1
1: x <- input
2: s <- 0
3: c <- 0
4: t <- c < x
5: ifn t goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: rret <- s
10: ret
```

```
// SSA_PA1
1: x0 <- input
2: s0 <- 0
3: c0 <- 0
4: s1 <- phi(3:s0, 8:s2)
   c1 <- phi(3:c0, 8:c2)
   t0 <- c1 < x0
5: ifn t0 goto 9
6: s2 <- c1 + s1
7: c2 <- c1 + 1
8: goto 4
9: rret <- s1
10: ret
```

Unstructured SSA Syntax

$$
\begin{array}{rccl}
(\texttt{Labeled Instruction}) & \mathit{li} & ::= & l : \overline{\phi}\ i \\
(\texttt{Instruction}) & i & ::= & d \leftarrow s \mid d \leftarrow s\ \mathit{op}\ s \mid \mathit{ret} \mid \mathit{ifn}\ s\ \mathit{goto}\ l \mid \mathit{goto}\ l \\
(\texttt{PhiAssignment}) & \phi & ::= & d \leftarrow \mathit{phi}(\overline{l : s}) \\
(\texttt{Labeled Instructions}) & \mathit{lis} & ::= & \mathit{li} \mid \mathit{li}\ \mathit{lis} \\
(\texttt{Operand}) & d, s & ::= & r \mid c \mid t \\
(\texttt{Temp Var}) & t & ::= & x \mid y \mid \ldots \\
(\texttt{Label}) & l & ::= & 1 \mid 2 \mid \ldots \\
(\texttt{Operator}) & \mathit{op} & ::= & + \mid - \mid < \mid == \mid \ldots \\
(\texttt{Constant}) & c & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
(\texttt{Register}) & r & ::= & r_{ret} \mid r_1 \mid r_2 \mid \ldots
\end{array}
$$

Unstructured SSA PA Operational Semantics

- Rules are of shape.

$$P \vdash (L, li, p) \longrightarrow (L', li', p')$$

- New rules

$$(\texttt{pPhi1}) \qquad (L, l : [] \; i, p) \longrightarrow (L, l : i, p)$$

$$(\texttt{pPhi2}) \qquad \frac{l_i = p \quad c_i = L(s_i) \qquad j \in [1, i-1] : l_j \neq p}{(L, l : d \leftarrow phi(l_1 : s_1, .., l_n : s_n); \overline{\phi} \; i, p) \longrightarrow (L \oplus (d, c_i), l : \overline{\phi} \; i, p)}$$

- All other rules are nearly identical to those for PA modulo the treatment of the ϕ assignments

Minimality

```
// SSA_PA2
1: x0 <- input
2: s0 <- 0
3: c0 <- 0
4: s1 <- phi(3:s0, 8:s2)
   c1 <- phi(3:c0, 8:c2)
   t0 <- c1 < x0
5: ifn t0 goto 9
6: s2 <- c1 + s1
7: c2 <- c1 + 1
8: goto 4
9: s3 <- phi(5:s1) // another phi assignment
   rret <- s3
10: ret
```

This above is not minimal. If the phi assignment at instruction 9 is removed with `s3` replaced by `s1`. We don't change the result of the program.

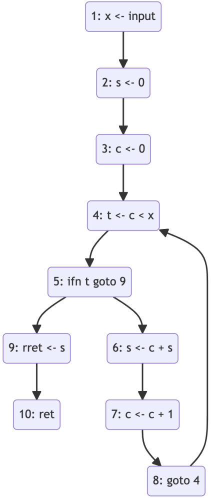
Constructing Minimal SSA

- Constructing a non-minimal SSA is easy. Just introduce phi-assignments at every instruction!
- Constructing a minimal SSA is not trivial.
- Cytron's algorithm.
  1. Turn the program into a CFG
  2. Derive the dominance frontiers from the graph (CFG)
  3. Insert phi assignments to the dominance frontiers of v, given a variable is assigned at vertex v.
  4. Rename the variables being re-assigned.

Constructing CFG

- It's easy for PA. Just following the label, the next labels and the goto'ed labels.

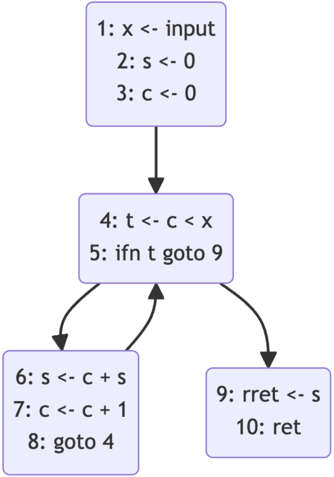
```
// PA1
1: x <- input
2: s <- 0
3: c <- 0
4: t <- c < x
5: ifn t goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: rret <- s
10: ret
```



Constructing CFG

- Better result, compact graph. But we stick to the simple (naive) approach.

```
// PA1
1: x <- input
2: s <- 0
3: c <- 0
4: t <- c < x
5: ifn t goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: rret <- s
10: ret
```



Finding the right places to insert ϕ assignments

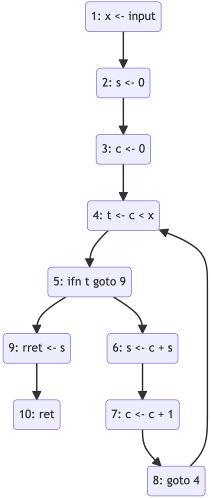
- We need some graph machinery.

Graph Theory Recap

- $G = (V, E)$ is a (directed) graph, V is a set of vertices and E is the set of edges.
- Graph on the right

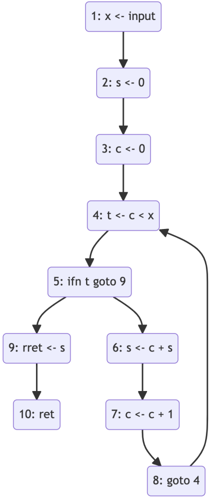
$$\begin{aligned} G &= (V, E) \\ V &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} \\ E &= \left\{ \begin{array}{c} (1,2), (2,3), (3,4), \\ (4,5), (5,6), (6,7), \\ (7,8), (8,4), (5,9), (9,10) \end{array} \right\} \end{aligned}$$

- Short hand notations
  - $v \in G$ iff $v \in V \wedge G = (V, E)$.
  - $(v, v') \in G$ iff $(v, v') \in E \wedge G = (V, E)$.



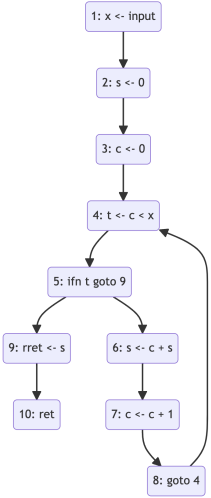
Graph Theory Recap

- Let $G = (V, E)$, a path from v_1 to v_2, written as $path(v_1, v_2)$, exists iff
  1. $v_1 = v_2$ or
  2. $\{(v_1, u_1), (u_1, u_2), ..., (u_n, v_2)\} \subseteq E$.
- v_1 and v_2 are connected, $connect(v_1, v_2)$ iff
  1. $path(v_1, v_2)$ or $path(v_2, v_1)$ exist, or
  2. $\exists v_3$ such that $connect(v_1, v_3)$ and $connect(v_2, v_3)$
- G is connected iff $\forall v_1, v_2 \in G$, $connect(v_1, v_2)$.
- All CFGs are connected.



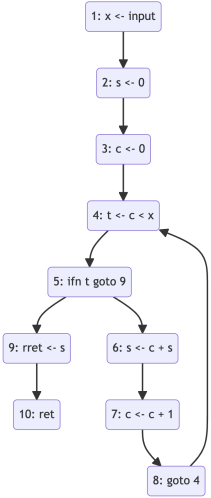
Graph Theory Recap

- Let v_1 be the source vertex of a graph (i.e. the starting vertex)
  - $u \preceq v$ iff for all path $path(v_1, v) = (v_1, ..., v)$ we find a prefix such that $(v_1, ..., u)$.
- e.g. $1 \preceq 1$, $1 \preceq 2$, $2 \preceq 5$, etc.



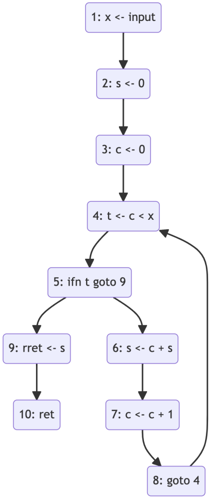
Graph Theory Recap

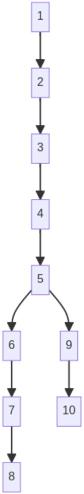
- Dominance is reflexive. $v \preceq v$.
- Dominance is anti-symmetric. $v_1 \preceq v_2$ and $v_2 \preceq v_1$ imply $v_1 = v_2$
- Dominance is transitive. $v_1 \preceq v_2$ and $v_2 \preceq v_3$ implies $v_1 \preceq v_3$.
- $v_1 \prec v_2$ iff $v_1 \preceq v_2$ and $v_1 \neq v_2$.



Graph Theory Recap

- $v_1 = idom(v_2)$ iff $v_1 \prec v_2$ and $\neg\exists v_3 . v_3 \prec v_2$ and $v_1 \prec v_3$.
- $idom(v)$ is unique if it exists.
- A Dominator Tree can be constructed by $idom(v)$ function, i.e. v_2 is a child of v_1 if $idom(v_2) = v_1$.



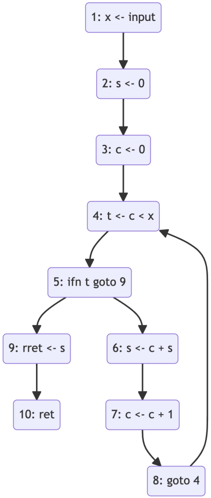


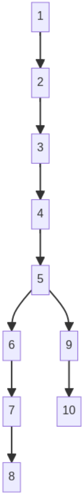
Dominance Frontiers = Places for ϕ assignments

$$df(v, G) = \{v_2 \mid (v_1, v_2) \in G \wedge v \preceq v_1 \wedge \neg(v \prec v_2)\}$$

$df(v, G)$ is the set of vertices that are not strictly dominated by v but their predecesors are (dominated by v).
e.g. `df(6) = {4}` because

- vertex 8 is one of the predecesors of the vertex 4 and
- vertex 8 is dominated by vertex 6, but not the vertex 4 is not domainated by vertex 6.
- what about `df(5)` and `df(4)`?





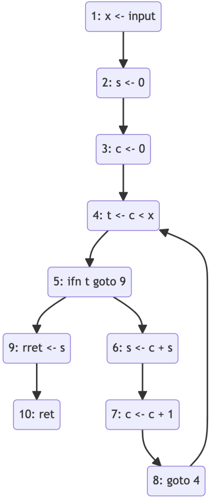
Dominance Frontier - A recursive definition

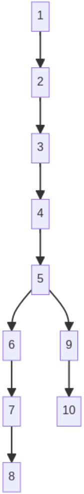
Let T be the dominator tree, G be the CFG.

$$df(v,G) = df_{local}(v,G) \cup \bigcup_{u \in child(v,T)} df_{up}(u,G) \ (E1)$$
$$df_{local}(v,G) = \{w \mid (v,w) \in G \wedge \neg(v \prec w)\} \ (E2)$$
$$df_{up}(v,G) = \{w \mid w \in df(v,G) \wedge \neg(idom(v) \prec w)\} \ (E3)$$

- $(E1)$ says the dominance frontier of v consists of the local set $(E2)$ and the children upward set $(E3)$.
- Example

$$\begin{array}{rcl}
df(6) & = & df_{local}(6) \cup df_{up}(7) \\
df_{local}(6) & = & \{\} \\
df_{up}(7) & = & \{w \mid w \in df(7) \wedge \neg(idom(7) \prec w)\} \\
df(7) & = & df_{local}(7) \cup df_{up}(8) \\
df_{local}(7) & = & \{\} \\
df_{up}(8) & = & \{w \mid w \in df(8) \wedge \neg(idom(8) \prec w)\} \\
df(8) & = & df_{local}(8) \\
df_{local}(8) & = & \{4\}
\end{array}$$



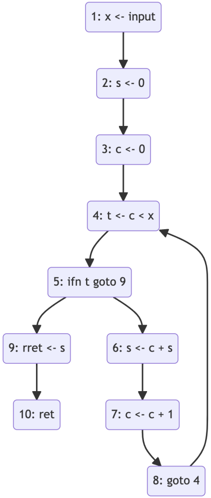


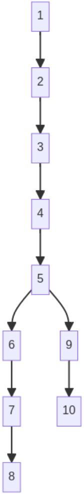
Dominance Frontier Algorithm

For each vertex v by traversing the dominator tree bottom up:

1. compute $df_{local}(v, G)$
2. compute $\bigcup_{u \in child(v,T)} df_{up}(u, G)$, which can be looked up from the a memoization table.
3. save $df(v, G) = df_{local}(v, G) \cup \bigcup_{u \in child(v,T)} df_{up}(u, G)$ in the memoization table.

vertex	succs	children	idom	df_{local}	df_{up}	df
10	{}	{}	9	{}	{}	{}
9	{10}	{10}	5	{}	{}	{}
8	{4}	{}	7	{4}	{4}	{4}
7	{8}	{8}	6	{}	{4}	{4}
6	{7}	{7}	5	{}	{4}	{4}
5	{6,9}	{6,9}	4	{}	{4}	{4}
4	{5}	{5}	3	{}	{}	{4}
3	{4}	{4}	2	{}	{}	{}
2	{3}	{3}	1	{}	{}	{}
1	{2}	{2}		{}	{}	{}





Iterative Dominance Frontier

A more general problem.

1. A variable `x` is assigned at location l.
2. According to dominance table, $df(l) = \{l'\}$.
   Now a ϕ assignment of `x` must be inserted at l',
3. We repeat step 1 with l'.

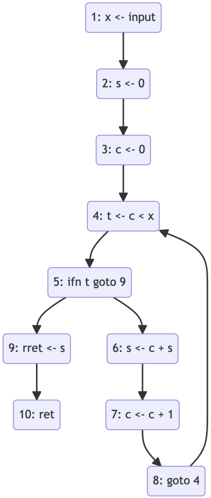
Let S denote a set of vertices of a graph G. We define

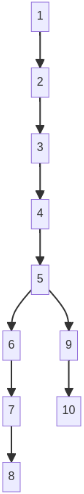
$$df(S, G) = \bigcup_{v \in S} df(v, G)$$

We define the iterative dominance frontier recursively as follows

$$\begin{array}{l} df_1(S, G) = df(S, G) \\ df_n(S, G) = df(S \cup df_{n-1}(S, G), G) \\ \exists k \geq 1. df_k(S, G) = df_{k+1}(S, G) \end{array}$$

Let's call it $df^{+}(S, G)$. For example `df+(6) = { 4 }`





Constructing Minimal SSA

- Constructing a non-minimal SSA is easy. Just introduce phi-assignments at every instruction!
- Constructing a minimal SSA is not trivial.
- Cytron's algorithm.
  1. Turn the program into a CFG
  2. Derive the dominance frontiers from the graph (CFG)
  3. **Insert phi assignments to the (iterative) dominance frontiers of v, given a variable is assigned at vertex v.**
  4. Rename the variables being re-assigned.

Inserting Phi Assignments

- From Dominance Table we build a dictionary E mapping labels to sets of variable.
- $(l, S) \in E$ implies that $x \in S$ having phi-assignments to be inserted at l.
- Labels `1,2,3,9,10` has no DF. Labels `4,5,6,7,8` all having DF at `4`.
- `s` and `c` are modified at `6,7`

```
E = Map(
    4 -> Set("s","c") )
```

```
// PA1
1: x <- input    // x
2: s <- 0        // s
3: c <- 0        // c
4: t <- c < x    // t
5: ifn t goto 9
6: s <- c + s    // s
7: c <- c + 1    // c
8: goto 4
9: rret <- s
10: ret
```

Inserting Phi Assignments

- Input: the original program `P`, a list of labeled instructions, and E.
- Output: the modified program `Q`. a list of labeled instructions.

Let `Q = []` for each `l:i` in `P`

- case `E.get(l)` of
  - `None`
    1. add `l:i` to `Q`
  - `Some(xs)`
    1. `phis = map (\x -> x <- phi( k:x | (k,l) in G)) xs`
    2. add `l:phis i` to `Q`

```
// PRE_SSA_PA1
1: x <- input
2: s <- 0
3: c <- 0
4: s <- phi(3:s, 8:s)
   c <- phi(3:c, 8:c)
   t <- c < x
5: ifn t goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: rret <- s
10: ret
```

Constructing Minimal SSA

- Constructing a non-minimal SSA is easy. Just introduce phi-assignments at every instruction!
- Constructing a minimal SSA is not trivial.
- Cytron's algorithm.
    1. Turn the program into a CFG
    2. Derive the dominance frontiers from the graph (CFG)
    3. Insert phi assignments to the (iterative) dominance frontiers of v, given a variable is assigned at vertex v.
    4. **Rename the variables being re-assigned**.

Renaming variables

Inputs:

- a dictionary of stacks `K`, where the keys are the variables. e.g. `K(x)` returns the stack for variable `x` from the PA program.
- the input program in with phi assignment but owing the variable renaming, We view the program as a **dictionary mapping labels to labeled instructions**.

1. For each variable `x` in the program, initialize `K(x) = Stack()`.
2. Let label `l` be the root of the dominator tree T.
3. Let `vars` be an empty list
4. Rename the variables in `l`,
   4.1 for any variable appearing on the LHS, `x`, append to `vars`
   4.2 whenver we generate a new name for `x` say `x_i`, push `x_i` to `K(x)`.
5. For each successor `k` of `l` in the CFG G
   5.1 update the phi assignment operands in `k` w.r.t to `l`, since we are going from `l` to ‘k
6. Recursively apply step 3 to the children of `l` in the T.
7. For each `x` in `vars`, `K(x).pop()`

Extra Technical Details

1. The Renaming variable steps can be simplified when we implement it in FP.
   - No stack is reqiured, since we use recursive functions
2. Algorithm to convert SSA PA back to PA exists. (refer to the notes)

Structured SSA

SSA can be constructed directly on SIMP.

```
x = input;
s = 0;
c = 0;
while c < x {
    s = c + s;
    c = c + 1;
}
return s;
```

```
x1 = input;
s1 = 0;
c1 = 0;
join { s2 = phi(s1,s3); c2 = phi(c1,c3); }
while c2 < x1 {
    s3 = c2 + s2;
    c3 = c2 + 1;
}
return s2;
```

Further Readings

- https://dl.acm.org/doi/10.1145/2955811.2955813
- https://dl.acm.org/doi/abs/10.1145/3605156.3606457
- https://dl.acm.org/doi/10.1145/202530.202532