Recall the MOP Solution

**Goal**
- For a forward problem, consider all possible paths from the entry to a given program point, compute the flow values at the end of each path, and then meet these values together
- **Meet-over-all-paths (MOP) solution at each program point**
- $\cap_{\text{all paths } n_1, n_2, \ldots, n_i} (f_{n_1}(\ldots f_{n_2}(f_{n_i}(v_{entry})))))$

---

Legality

“Is $v_{MFP}$ legal?” = “Is $v_{MFP} \subseteq v_{MOP}$?”

**Look at Merges**
- $v_{MOP} = F_r(v_{p1}) \cap F_r(v_{p2})$
- $v_{MFP} = F_r(v_{p1} \cap v_{p2})$
- $v_{MFP} \subseteq v_{MOP} = F_r(v_{p1} \cap v_{p2}) \subseteq F_r(v_{p1}) \cap F_r(v_{p2})$

**Observation**
\[ \forall x, y \in V \quad f(x \cap y) \subseteq f(x) \cap f(y) \quad \iff \quad x \subseteq y \Rightarrow f(x) \subseteq f(y) \]

\[ \therefore \quad v_{MFP} \text{ legal when } F_r \text{ (the flow functions) are monotonic} \]
Monotonicity

Monotonicity: $(\forall x, y \in V)(x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y))$

- If the flow function $f$ is applied to two members of $V$, the result of applying $f$ to the “lesser” of the two members will be under the result of applying $f$ to the “greater” of the two.
- Giving a flow function more conservative inputs leads to more conservative outputs (never more optimistic outputs).

Why else is monotonicity important?

For monotonic $F$ over domain $V$

- The maximum number of times $F$ can be applied to self w/o reaching a fixed point is $\text{height}(V) - 1$.
- IDFA is guaranteed to terminate if the flow functions are monotonic and the lattice has finite height.

Efficiency

Parameters

- $n$: Number of nodes in the CFG
- $k$: Height of lattice
- $t$: Time to execute one flow function

Complexity

- $O(nkt)$

Example

- Reaching definitions?
### Accuracy

**Distributivity**
- \( f(u \sqcap v) = f(u) \sqcap f(v) \)
- \( V_{\text{MFP}} \sqsubseteq V_{\text{MOP}} \equiv F_{r}(v_{p1} \sqcap v_{p2}) \sqsubseteq F_{r}(v_{p1}) \sqcap F_{r}(v_{p2}) \)
- If the flow functions are distributive, MFP = MOP

**Examples**
- Liveness?
- Reaching constants?

\[
\begin{align*}
    f(u \sqcap v) &= f(\{x=2,y=3\} \sqcap \{x=3,y=2\}) \\
                 &= f(\emptyset) = \emptyset \\
    f(u) \sqcap f(v) &= f(\{(x=2,y=3\}) \sqcap f(\{x=3,y=2\}) \\
                     &= \{ \{x=2,y=3,w=5\} \sqcap \{x=3,y=2,w=5\} \} \\
                     &= \{w=5\}
\end{align*}
\]

\(\Rightarrow MFP \neq MOP\)

---

### Concepts

**Lattices**
- Conservative approximation
- Optimistic (initial guess)
- Data-flow analysis frameworks
- Tuples of lattices

**Data-flow analysis**
- Fixed point
- Meet-over-all-paths (MOP)
- Maximum fixed point (MFP)
- Legal/safe (monotonic)
- Efficient
- Accurate (distributive)
Static Single Assignment Form

Last Time
- Lattice theoretic framework for data-flow analysis

Today
- Program representations
- Static single assignment (SSA) form
  - Program representation for sparse data-flow
  - Conversion to and from SSA

Next Time
- Reuse optimizations

Data Dependence

Definition
- Data dependences are constraints on the order in which statements may be executed

Types of dependences
- Flow dependence: \( s_1 \) writes memory that \( s_2 \) later reads (RAW)
  \[
  s_1: \ x = 17 \\
  s_2: \ \text{print}(x)
  \]
- Anti-dependence: \( s_1 \) reads memory that \( s_2 \) later writes (WAR)
  \[
  s_1: \ \text{print}(x) \\
  s_2: \ x = 18
  \]
- Output dependences: \( s_1 \) writes memory that \( s_2 \) later writes (WAW)
  \[
  s_1: \ x = 19 \\
  s_2: \ x = 20
  \]
Data Dependence (cont)

True dependences
– Flow dependences represent actual flow of data

False dependences
– Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated

\[ s_1: \text{print} (x) \quad s_1: \text{print} (x_1) \]
\[ s_2: x = 18 \quad s_2: x_2 = 18 \]

Other dependences
– Input dependences: \( s_1 \) reads memory that \( s_2 \) later reads (RAR)

\[ s_1: y = x + 1 \]
\[ s_2: \text{print} (x) \]

Example

Identify the dependences

\[ s_1: a = b; \]
\[ s_2: b = c + d; \]
\[ s_3: e = a + d; \]
\[ s_4: b = 3; \]
\[ s_5: f = b * 2; \]
Representing Data Dependences

**Implicitly**
- Use variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

**Def-use chains (du chains)**
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, consumes space

---

**DU Chains**

**Definition**
- du chains link each def to its uses

**Example**

```plaintext
s_1  a = b;
s_2  b = c + d;
s_3  e = a + d;
s_4  b = 3;
s_5  f = b * 2;
```

---
UD Chains

Definition
- ud chains link each use to its defs

Example

\[
\begin{align*}
    s_1 & : \quad a = b; \\
    s_2 & : \quad b = c + d; \\
    s_3 & : \quad e = a + d; \\
    s_4 & : \quad b = 3; \\
    s_5 & : \quad f = b * 2;
\end{align*}
\]

Representing Data Dependences (cont)

Implicitly
- Use variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

Def-use chains (du chains)
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, consumes space

Alternate representations
- e.g., Static single assignment form (SSA), dependence flow graphs (DFG), value dependence graphs (VDG)
Static Single Assignment Form

Idea
– Each variable has only one static definition
– Makes it easier to reason about values instead of variables
– Similar to the notion of functional programming

Transformation to SSA
– Rename each definition
– Rename all uses reached by that definition

Example
\[
\begin{align*}
\mathbf{v} & := \ldots \\
\ldots & := \ldots \mathbf{v} \ldots \\
\mathbf{v} & := \ldots \\
\ldots & := \ldots \mathbf{v} \ldots \\
\mathbf{v} & := \ldots \\
\ldots & := \ldots \mathbf{v} \ldots \\
\mathbf{v}_0 & := \ldots \\
\ldots & := \ldots \mathbf{v}_0 \ldots \\
\mathbf{v}_1 & := \ldots \\
\ldots & := \ldots \mathbf{v}_1 \ldots
\end{align*}
\]

What do we do when there's control flow?
SSA and Control Flow

Problem
- A use may be reached by several definitions

SSA and Control Flow (cont)

Merging Definitions
- $\phi$-functions merge multiple reaching definitions

Example
Exercise

Q: How do we transform the following code to SSA form?

\[
\begin{align*}
1 & \quad v := 1 \\
2 & \quad v := v + 1
\end{align*}
\]

\[
\begin{align*}
1 & \quad v_0 := 1 \\
2 & \quad v_1 := v_0 + v_2 \\
2 & \quad v_2 := v_1 + 1
\end{align*}
\]

SSA vs. ud/du Chains

SSA form is more constrained

Advantages of SSA
- More compact
- Some analyses become simpler when each use has only one def
- Value merging is explicit
- Easier to update and manipulate?

Furthermore
- Eliminates false dependences (simplifying context)

```plaintext
for (i=0; i<n; i++)
    A[i] = i;
for (i=0; i<n; i++)
    print(foo(i));
```

Unrelated uses of \(i\) are given different variable names
SSA vs. ud/du Chains (cont)

Worst case du-chains?

```
switch (c1) {
    case 1:   x = 1; break;
    case 2:   x = 2; break;
    case 3:   x = 3; break;
}
x_4 = \phi(x_1, x_2, x_3)
```

```
switch (c2) {
    case 1:   y1 = x; break;
    case 2:   y2 = x; break;
    case 3:   y3 = x; break;
    case 4:   y4 = x; break;
}
```

$m$defs and $n$uses leads to $m \times n$ du chains

Transformation to SSA Form

Two steps

- Insert $\phi$-functions
- Rename variables
Where Do We Place $\phi$-Functions?

**Basic Rule**
- If two distinct (non-null) paths $x \rightarrow z$ and $y \rightarrow z$ converge at node $z$, and nodes $x$ and $y$ contain definitions of variable $v$, then we insert a $\phi$-function for $v$ at $z$

$$
\begin{align*}
  v_1 := & \ldots \\
  v_2 := & \ldots \\
  \phi(v_1, v_2) := & \ldots v_3 \ldots \\
  v_1 := & \ldots \\
  v_2 := & \ldots
\end{align*}
$$

Approaches to Placing $\phi$-Functions

**Minimal**
- As few as possible subject to the basic rule
- How is this sub-optimal?

**Briggs-Minimal**
- Same as minimal, except $v$ must be live across some edge of the CFG

$$
\begin{align*}
  v = & v \\
  v = & v
\end{align*}
$$

Briggs Minimal will not place a $\phi$ function in this case because $v$ is not live across any CFG edge. Exploits the short lifetimes of many temporary variables

Can we do better than Briggs-Minimal?
**Approaches to Placing \( \phi \)-Functions (cont)**

**Pruned**
- Same as minimal, except does not insert dead \( \phi \)-functions
- What's the difference between Pruned and Briggs-Minimal?

\[
\begin{align*}
\text{Briggs Minimal will add a } & \phi \text{ function because } v \text{ is live across the blue edge, but Pruned SSA will not because the } \\
\text{\( \phi \) function is dead (assuming that this is the entire CFG)}\end{align*}
\]

**Why would we ever use Briggs Minimal instead of Pruned SSA?**

---

**Machinery for Placing \( \phi \)-Functions**

**Recall Dominators**
- \( d \text{ dom } i \) if all paths from entry to node \( i \) include \( d \)
- \( d \text{ sdom } i \) if \( d \text{ dom } i \) and \( d \neq i \)

**Dominance Frontiers**
- The **dominance frontier** of a node \( d \) is the set of nodes that are “just barely” not dominated by \( d \); i.e., the set of nodes \( n \), such that
  - \( d \) dominates a predecessor \( p \) of \( n \), and
  - \( d \) does **not** strictly dominate \( n \)
- \( \text{DF}(d) = \{ n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \neq \text{sdom } n \} \)

**Notational Convenience**
- \( \text{DF}(S) = \bigcup_{s \in S} \text{DF}(s) \)

**What is the significance of the dominance frontier?**
Dominance Frontier Example

\[
\text{DF}(d) = \{n \mid \exists p \in \text{pred}(n), \ d \text{ dom } p \text{ and } d \not\text{sdom } n\}
\]

\[
\text{Dom}(5) = \{5, 6, 7, 8\}
\]

\[
\text{DF}(5) = \{4, 5, 12, 13\}
\]

Where shall we place \(\phi\) functions?

In SSA form, definitions must dominate uses

---

Dominance Frontier Example II

\[
\text{DF}(d) = \{n \mid \exists p \in \text{pred}(n), \ d \text{ dom } p \text{ and } d \not\text{sdom } n\}
\]

\[
\text{Dom}(5) = \{5, 6, 7, 8\}
\]

\[
\text{DF}(5) = \{4, 5, 13\}
\]

Node 4 is the first point of convergence between the entry and node 5, so do we need a \(\phi\) function at node 13?
**SSA Exercise**

DF(8) = \{10\}
DF(9) = \{10\}
DF(2) = \{6\}
DF({8,9}) = \{10\}
DF(10) = \{6\}
DF({2,8,9,10}) = \{6,10\}

**Dominance Frontiers Revisited**

Suppose that node 3 defines variable x

DF(3) = \{5\}

Do we need to insert a \(\phi\)-function for \(x\) anywhere else?
Yes. At node 6. Why?
Dominance Frontiers and SSA

Let
- \( DF_1(S) = DF(S) \)
- \( DF_{i+1}(S) = DF(S \cup DF_i(S)) \)

Iterated Dominance Frontier
- \( DF_\infty(S) \)

Theorem
- If \( S \) is the set of CFG nodes that define variable \( v \), then \( DF_\infty(S) \) is the set of nodes that require \( \phi \)-functions for \( v \)

Algorithm for Inserting \( \phi \)-Functions

for each variable \( v \)
    WorkList \( \leftarrow \emptyset \)
    EverOnWorkList \( \leftarrow \emptyset \)
    AlreadyHasPhiFunc \( \leftarrow \emptyset \)
    for each node \( n \) containing an assignment to \( v \)
        WorkList \( \leftarrow \text{WorkList} \cup \{n\} \)
        EverOnWorkList \( \leftarrow \text{WorkList} \)
    while WorkList \( \neq \emptyset \)
        Remove some node \( n \) from WorkList
        for each \( d \in DF(n) \)
            if \( d \notin \text{AlreadyHasPhiFunc} \)
                Insert \( \phi \)-function for \( v \) at \( d \)
                AlreadyHasPhiFunc \( \leftarrow \text{AlreadyHasPhiFunc} \cup \{d\} \)
            if \( d \notin \text{EverOnWorkList} \)
                WorkList \( \leftarrow \text{WorkList} \cup \{d\} \)
                EverOnWorkList \( \leftarrow \text{EverOnWorkList} \cup \{d\} \)
                Process each node at most once
                Insert at most one \( \phi \) function per node

Next Time

Lecture
- Will start at 2:15pm
- Data-flow analysis and SSA

Reading
- Csmith paper due Sunday February 15\textsuperscript{th} at 5:00pm