## Data Flow Analysis and Optimizations

**Last Time**

- Optimizations with SSA

**Today**

- Data Flow Analysis
- Data Flow Frameworks
- Constant Propagation
- Reaching Definitions

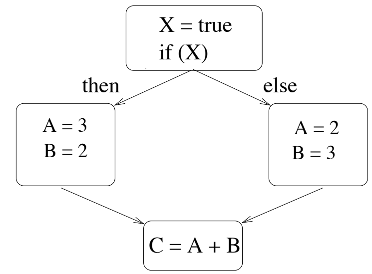


## Data Flow Analysis

Data flow analysis tells us things we want to know about programs. For example:

- Is this computation loop invariant?
- Which definition reaches this use?
- Is this value a constant?

**Example:**






## Data Flow Analysis

Systems of equations that compute information (e.g., uses, definitions, values) about variables at program points.

A **Monotone Data Flow Framework**

- *point* - start and/or end of a basic block
- Information for a forward problem

$$\begin{aligned} \text{INFO}_{in}(v) &= \text{merge}_{p \in PRED(v)}(\text{INFO}_{out}(p)) \\ \text{INFO}_{out}(v) &= \text{transfer}(\text{INFO}_{in}(v)) \end{aligned}$$

- Transfer functions:

  $t_v$ is transfer function for $v$, how information is changed by $v$.

  $T_Q$ is transfer function for a path from *Entry* (information carried on path $q$)

Given $Q$: *Entry* $\rightarrow^{+} x$, such that $q = q_o \rightarrow q_1 \rightarrow \ldots q_n$, the **transfer function** is:

$$t_{q_n-1}(t_{q_n-2}(\ldots(t_2(t_1(t_0(\top)))\ldots)$$

**Meet Over All Paths Solution**

$$MOP(v) = \sqcap_{q \in Paths(v)} T_q(\top)$$



## Data Flow Framework

1. A *semilattice* $\mathcal{L}$ with a binary meet operation $\sqcap$, such that $a, b, c \in \mathcal{L}$ :
   - $a \sqcap a = a$ (*idempotent*)
   - $a \sqcap b = b \sqcap a$ (*commutative*)
   - $a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$ (*associative*)
2. $\sqcap$ imposes an order on $\mathcal{L} : \forall\ a, b \in \mathcal{L}$
   - $a \succeq b \Leftrightarrow a \sqcap b = b$
   - $a \succ b \Leftrightarrow a \succeq b$ and $a \neq b$
3. A semilattice has a unique *bottom* element $\perp : \forall a \in \mathcal{L}$
   - $a \sqcap \perp = \perp$.
   - $a \succeq \perp$
4. It has a unique *top* or *identity* element $\top : \forall a \in \mathcal{L}$
   - $a \sqcap \top = a$
   - $\top \succeq a$

## Problem Representation

- Choose a semilattice $\mathcal{L}$ to represent facts
- Attach to each $a \in \mathcal{L}$ a *meaning*
  each $a \in \mathcal{L}$ is a distinct a set of known facts
- With each node $n$, associate a function $f_n : \mathcal{L} \rightarrow \mathcal{L}$
  $f_n$ models behavior of code corresponding to $n$
- Let $\mathcal{F}$ be the set of all functions the code generates



## Constant Propagation — Example Framework

Constant propagation lattice:

$$\begin{array}{ccccccc} & & & \top & & & \\ \ldots & \text{-2} & \text{-1} & 0 & 1 & 2 & \ldots \\ & & & \bot & & & \end{array}$$

1. meet rules
   - $a \sqcap \top = a$
   - $a \sqcap \bot = \bot$
   - $a \sqcap a = a \iff$ constant $a$ and $a = a$
   - $a \sqcap b = \bot$otherwise
2. meet properties impose a partial order on $\mathcal{L}$
   - $3 \sqcap 3 = 3$
   - $3 \sqcap 2 = 2 \sqcap 3$
   - $3 \sqcap (2 \sqcap 4) = (3 \sqcap 2) \sqcap 4$
3. bottom
   - $a \sqcap \bot = \bot$ for every $a \in \mathcal{L}$.
   - $\forall a \in \mathcal{L}, a \succeq \bot$
4. top
   - $a \sqcap \top = a$ for every $a \in \mathcal{L}$
   - $\forall a \in \mathcal{L}, \top \succeq a$



## Data Flow Framework

A *descending chain* in $\mathcal{L}$ is a sequence $x_1, x_2, \cdots, x_n$:

1. $x_i \in \mathcal{L}, 1 \leq i \leq n$, and
2. $x_i \succeq x_{i+1}, 1 \leq i < n$

If $\forall a \in \mathcal{L}, \exists$ constant $b_a$ such that any chain beginning with $a$ has length $\leq b_a$, we say that $\mathcal{L}$ is bounded.

Any bounded semilattice has *finite descending chains*.



## Admissible Function Spaces — [Kam & Ullman]

For a bounded semilattice $\mathcal{L}, \mathcal{F} : \mathcal{L} \rightarrow \mathcal{L}$ is an *admissible function space* $\iff$

1. Monotonic:
   $$\forall f \in \mathcal{F}, \forall x, y \in \mathcal{L},\ x \preceq y \Rightarrow f(x) \preceq f(y)$$
2. Identity operation:
   $$\exists f_I \in \mathcal{F} : \forall x \in \mathcal{L}, f_I(x) = x$$
3. Closed under composition:
   $$f, g \in \mathcal{F} \Rightarrow f \circ g \in \mathcal{F}$$
   where $\forall x \in \mathcal{L}, [f \circ g](x) = f(g(x))$
4. $\bot$ exists to any $x \in \mathcal{L}$
   $$\forall x \in \mathcal{L}, \exists \text{ a finite subset } H \subseteq \mathcal{F}\ \ni x = \sqcap_{f \in H} f(\bot)$$

## Monotone Data Flow Framework

is a triple $\langle \mathcal{L}, \sqcap, \mathcal{F} \rangle$ where

- $\sqcap$ is the meet operation, or *confluence* operator
- $\langle \mathcal{L}, \sqcap \rangle$ is a semilattice of finite length with bottom $\perp$
- $\mathcal{F}$ is a *monotone function space* on $\mathcal{L}$:
  a set of unary functions such that each operation $f \in \mathcal{F}$ is monotonic:

$$\forall f \in \mathcal{F}, \forall x, y \in \mathcal{L},\ x \preceq y \Rightarrow f(x) \preceq f(y)$$

A monotone data flow framework $\langle \mathcal{L}, \sqcap \rangle$ is **distributive** $\iff$

$$\forall f \in \mathcal{F}, \forall x, y \in \mathcal{L},\ f(x \sqcap y) = f(x) \sqcap f(y)$$

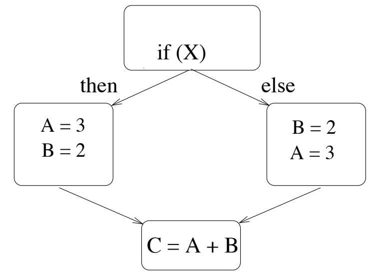
### Meet Over All Paths Solution

$$MOP(v) = \sqcap_{q \in Paths(v)} T_q(\top)$$



## Constant Propagation Example Framework

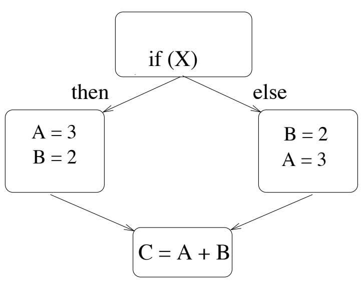
1. Is CP monotonic?
2. Is CP distributive?
3. Is every solution a meet over all paths solution?






## Constant Propagation Example Framework

1. Is CP monotonic?
   Yes
2. Is CP distributive?
   No
   $f(A+B) = 5$
   $f(A) + f(B) = 2 \sqcap 3 + 3 \sqcap 2 = \perp$
3. Is every solution a meet over all paths solution?
   No






## Reaching Definitions

For each vertex, find the set of variable definitions that might reach that vertex.

$GEN(v)$ variable $v$ *may* be defined or assigned to

$KILL(v)$ variable $v$ is defined, overwriting other definitions

```
1 read N
2 call check(N)
3 i ← 1
4 while i < N do
5         a[i] ← a[i] + i
6         i ← i + 1
7 end
8 print a[n]
```

| | *GEN* | *KILL* | *PRED* | *SUCC* |
|---|---|---|---|---|
| 1 : | $N_1$ | $N$ | | 2 |
| 2 : | $N_2$ | | 1 | 3 |
| 3 : | $i_3$ | $i$ | 2 | 4 |
| 4 : | | | 3, 7 | 5, 8 |
| 5 : | $a_5$ | | 4 | 6 |
| 6 : | $i_6$ | $i$ | 5 | 7 |
| 7 : | | | 6 | 4 |
| 8 : | | | 4 | |

## Reaching Definitions: Transfer Function

$IN(v)$ the set of definitions that reach statement $v$

$$IN(v) = \bigcup_{p \in PRED(v)} OUT(p)$$

$OUT(v)$ the set of reaching definitions just after statement $v$

$$OUT(v) = GEN(v) \cup (IN(v) - KILL(v))$$

$IN$ is an *inherited* attribute

$OUT$ is a *synthesized* attribute

$GEN$ and $KILL$ are *basic* attributes

**Forward** data flow problems propagate from predecessors of $v$ to $v$

**Backward** data flow problems propagate from successors of $v$ to $v$



## Reaching Definitions

Monotone Data Flow Framework

- A = set of *generations*,
  generation = (statement, variable)
- Lattice: $\mathcal{L} = \langle 2^A, \cup \rangle$
  $2^A \equiv$ the set of all subsets of $A$
  What does this look like?
- initial value = $\{\}$
- transfer function $t_v(x) = GEN(v) \cup (x - KILL(v))$
- monotone: $x \subseteq y \Rightarrow t_v(x) \subseteq t_v(y)$
- distributive: $t_v(x \cup y) = t_v(x) \cup t_v(y)$



## Work List Iterative Algorithm

initialize *ReachingDefinitions*$(n)$
*worklist* $\leftarrow$ the set of all nodes
**while** *worklist* $\neq \{\}$
    take $n$ from *worklist*
    recompute *ReachingDefinitions*$(n)$
    **if** *ReachingDefinitions*$(n)$ changed
      *worklist* $\leftarrow$ *worklist* $\cup SUCC(n)$
    **end**
**end**

initialization
$IN(v) \leftarrow \{\}$
$OUT(v) \leftarrow GEN(v)$

computation
$IN(v) =$
$OUT(v) =$



## Reaching Definitions Algorithm

**foreach** $v \in V$
  $IN(v) \leftarrow \{\}$
  $OUT(v) \leftarrow GEN(v)$
**end**
*worklist* $\leftarrow V$
**while** *worklist* $\neq \{\}$
    take $v$ from *worklist*
    $IN(v) \leftarrow \bigcup_{p \in PRED(v)} OUT(p)$
    $OUT(v) \leftarrow GEN(v) \bigcup (IN(v) - KILL(v))$
    **if** $OUT(v)$ changed
      *worklist* $\leftarrow$ *worklist* $\cup SUCC(v)$
    **end**
**end**

## Reaching Definitions

For each vertex, find the set of variable definitions that might reach that vertex.

$GEN(v)$ variable $v$ *may* be defined or assigned to

$KILL(v)$ variable $v$ is defined, overwriting other definitions

```
1 read N
2 call check(N)
3 i ← 1
4 while i < N do
5         a[i] ← a[i] + i
6         i ← i + 1
7 end
8 print a[n]
```

| | GEN | KILL | PRED | SUCC |
|---|---|---|---|---|
| 1 : | $N_1$ | $N$ | | 2 |
| 2 : | $N_2$ | | 1 | 3 |
| 3 : | $i_3$ | $i$ | 2 | 4 |
| 4 : | | | 3, 7 | 5, 8 |
| 5 : | $a_5$ | | 4 | 6 |
| 6 : | $i_6$ | $i$ | 5 | 7 |
| 7 : | | | 6 | 4 |
| 8 : | | | 4 | |



## Reaching Definitions Example

| | Initial value | | iteration 1 | | iteration 2 | | iteration 3 | |
|---|---|---|---|---|---|---|---|---|
| | IN | OUT | IN | OUT | IN | OUT | IN | OUT |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | | | | | | | | |
| 8 | | | | | | | | |



## Questions

- Does this always terminate?
- What answer does it compute?
- How fast (or slow) is it?



## Reaching Definitions Algorithm

**foreach** $v \in V$
  $IN(v) \leftarrow \{\}$
  $OUT(v) \leftarrow GEN(v)$
**end**
*worklist* $\leftarrow V$
**while** *worklist* $\neq \{\}$
    take $v$ from *worklist*
    $IN(v) \leftarrow \bigcup_{p \in PRED(v)} OUT(p)$
    $OUT(v) \leftarrow GEN(v) \bigcup (IN(v) - KILL(v))$
    **if** $OUT(v)$changed
      *worklist* $\leftarrow$ *worklist* $\cup\, SUCC(v)$
    **end**
**end**

## Work List Iterative Algorithm

**Questions**

- Does this always terminate?
- How fast (or slow) is it?
- What answer does it compute?
- How fast can we make it?



## Termination

*Why does the iterative data flow algorithm terminate?*

*Sketch of proof for reaching definitions*:

1. each node is initialized to $\{\}$
2. a definition has only one statement that generates it
3. $\mathcal{F}$ is associative $\Rightarrow$ $\mathcal{F}$ is monotone $\Rightarrow$ each $x \in$ *reaching definitions* can be added once
4. $N * (E+1)$ trips to take a definition to every node

*Consequence of finite descending chain property*

Question: How do we generalize this proof?



## Correctness and Quality of Solution

*Does it compute the answer we want?*

**Definition:** For each basic block $b$

$$MOP(b) = \sqcap_{q \in Paths(b)} T_q(\top)$$

- Paths that reach a block are reachable in the control flow graph, which may be conservative
- Perfect Solution $=$ meet over *real* paths taken during program execution
- $MOP \leq$ Perfect Solution
- In some sense, $MOP$ is best feasible solution
- Not guaranteed to achieve $MOP$ solution unless transfer functions distributive



## Quality of Solution

**Maximal Fixed Point ($MFP$)**

- Any iterative data-flow problem that satisfies admissible function requirements when it converges to a solution and terminates, will have reached a Maximal Fixed Point solution
- $MFP$ is unique, regardless of order of propagation
- If distributive, $MFP = MOP$
- Otherwise, $MFP \leq MOP$
- So, $MFP \leq MOP \leq$ Perfect Solution

## How fast can we make the iterative algorithm?

**Execution time of iterative framework**

**For each basic block:** # successors (predecessors) + constant bit vector operations

**Number of visits to basic block:** length of longest acyclic path

**What is the complexity equation?** $O(n^2)$

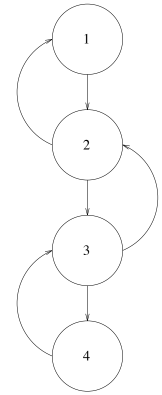
**Where is unnecessary work being performed?**

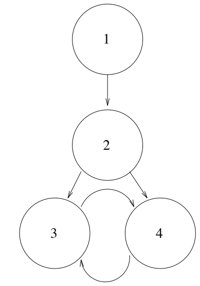
- Iteration over every node on each pass
- Testing for altered sets on each pass
- Extra pass to detect stabilization

**Problem**: Nodes may be visited in any order



## Examples






## How fast can we make the iterative algorithm?

**To avoid unnecessary work:**

- Bound number of visits by visiting a node roughly *after* all its predecessorsn
  (reverse PostOrder for forward data-flow problem;
  conceptually, PostOrder for backward problem)
- Change to algorithm:

  *changed* $\leftarrow$ *false*
  **do**
      **foreach** $v \in V$ in rPostOrder **do**
          solve for $b$
          **if** *old* $\neq$ *new*
              *changed* $\leftarrow$ *true*
          **end**
      **end**
  **while** *changed*
- How does this improve performance?



## PostOrder and Reverse PostOrder

**Step1: PostOrder**

**proc** *main*() $\equiv$
    *count* $\leftarrow 1$
    *Visit*$(Entry)$
**end**

**proc** *Visit*$(v)$ $\equiv$
    mark $v$ as visited
    **foreach** successor $s$ of $v$ not yet visited
        *Visit*$(s)$
    **end**
    *PostOrder*$(v) \leftarrow$ *count* $++$
**end**

**Step 2: rPostOrder**

**foreach** $v \in V$ **do**
    *rPostOrder*$(v) \leftarrow |V| -$ *PostOrder*$(v)$
**end**

Depth-first search $\approx$ rPostOrder

## Analysis of Data-flow Frameworks

*Key things to look for in a data-flow framework*

- the domain and its size
- size of a single fact
- forward or backward problem
- model of characteristic function

**Representation**

- Sets represented by *bit vector*
- **Size of each bit vector:**
  **Available Expressions:** # distinct expressions in program
  **Reaching Definitions:** # definitions in program
  **Live Variable Analysis:** # variables in program

**Complexity**

- distinguish bit-vector steps from logical steps
- watch out for complex mappings ($GEN \rightarrow KILL$)



## Summary

- Iterative data-flow framework used to solve global data-flow problems
- Use semi-lattice to represent facts
- Analysis on semi-lattice with finite descending chains and monotone data-flow framework guarantees termination
- Monotonic data-flow framework guarantees $MFP$ solution reached
- Distributive to guarantee $MOP$ solution reached
- rPostOrder (or PostOrder) for “rapid” data-flow problems guarantees bound of $O(n(d+2))$ complexity, where $d$ is maximum number of retreating edges on any acyclic path in the CFG (loop “interconnectiveness”)