Data-Flow Analysis

Approaches

- **Dynamic Analysis**
  - Assertions
  - Error seeding, mutation testing
  - Coverage criteria
  - Fault-based testing
  - Object oriented testing
  - Regression testing

- **Static Analysis**
  - Inspections
  - Dependence analysis
  - Symbolic execution
  - Software Verification
  - Data flow analysis
  - Concurrency analysis
Data Flow Analysis (DFA)

• Efficient technique for proving properties about programs
• Not as powerful as automated theorem provers, but requires less human expertise
• Uses an annotated control flow graph model of the program
  • Compute facts for each node
• Use the flow in the graph to compute facts about the whole program
  • We’ll focus on single units

Some examples of DFA techniques

• DFA used extensively in program optimization
  • e.g., determine if a definition is dead (and can be removed)
    • determine if a variable always has a constant value
    • determine if an assertion is always true and can be removed
• DFA can also be used to find anomalies in the code
  • Find def/ref anomalies [Osterweil and Fosdick]
  • Cecil/Cesar system demonstrated the ability to prove general user-specified properties [Olender and Osterweil]
  • FLAVERS demonstrated applicability to concurrent system [Dwyer and Clarke]
• Why “anomalies” and not faults?
  • May not correspond to an actual executable failure
**Data flow analysis**

- First, determine local information that is true at each node in the CFG
  - e.g., What variables are defined
  - What variables are referenced
  - Usually stored in sets
    - e.g., ref(n) is the set of variables referenced at node n
- Second, use this local information and control flow information to compute global information about the whole program
  - Done incrementally by looking at each node’s successors or predecessors
  - Use a fixed point algorithm—continue to update global information until a fixed point is reached

**Example: Reaching Definitions**

- Definition reaches a node if there is a def clear path from the definition to that node
- Definition of x at node 1 reaches nodes 2, 3, 4, 5 but not 6

---

Start from a definition, and move forward on the graph to see how far it reaches
Computing global information - reaching definitions

Definitions that **might** reach a node

```
reaching_def={x_i}
```

```
reaching_def={y_j}
```

```
reaching_def={x_i,y_j}
```

Definitions that **must** reach a node

```
reaching_def={x_i}
```

```
reaching_def={y_j}
```

```
reaching_def={x_i,y_j}
```

```
reaching_def={y_j}
```

Start from a definition, and move forward on the graph to see how far it reaches.

Reaching Definitions

- \( X_i \) means that the definition of variable \( x \) at node \( i \) **might** reach the current node.
  - Also stated as **possible**
    - \( \text{some} \)
    - \( \text{any} \)
    - \( \text{may} \)
  - **must** reach the current node.
  - Also stated as **definite**
    - \( \text{all} \)
    - \( \text{all} \)
Computing values for a node

- Keep track of the definitions into a node that have not been redefined
- Flow out of a node depends on flow in and on what happens in the node

Forward flow

Example: Possible Reaching Definitions

\[
\begin{align*}
\text{int x,y;} \\
x := \text{foo();} \\
y := x + 2; \\
\text{if } x > 0 \text{ then} \\
x := x + y; \\
\text{end if;}
\end{align*}
\]
**Example: Definite Reaching Definitions**

```plaintext
int x, y;
...
X_1 x := foo();
Y_2 y := x + 2;
if x > 0 then
  x := x + y;
  end if;
...
```

Forward flow, all path problem, def/ref sets are the initial facts associated with each node

---

**Example: Definite Reaching Definitions**

```plaintext
int x, y;
...
X_1 x := foo();
Y_2 y := x + 2;
if x > 0 then
  x := x + y;
  end if;
...
```

What happens when we add a loop?
**Example: Definite Reaching Definitions**

```
int x, y;
...
x[1] := foo();
y[2] := x + 2;
if x > 0 then
    x[4] := x + y;
end if;
...
```

Forward flow, all path problem, def/ref sets are the initial facts associated with each node.

What happens when we add a loop?

---

**Computing values for a node**

- Keep track of the definitions into a node that have not been redefined
- Flow out of a node depends on flow in and on what happens in the node

For definite reaching defs:

\[
\text{In}(n) = \bigcap_{k \in \text{pred}} \text{Out}(k)
\]

\[
\text{Out}(n) = \text{In}(n) - \text{def}(n) \cup \text{def}(n_k)
\]
**Live Variables**

- A variable, \( x \), is live at node \( p \) if there exists a def-clear path wrt \( x \) from node \( p \) to a use of \( x \).
- \( x \) is live at 2, 3, 4, but not at node 5.

**Computing global information - live variables**

- Possible live variables:
  - \( 1 \): live\( \{x, y\} \)
  - \( 2 \): live\( \{x\} \)
  - \( 3 \): live\( \{x\} \)
  - \( 4 \): live\( \{y\} \)

- Definite live variables:
  - \( 1 \): live\( \{x\} \)
  - \( 2 \): live\( \{x\} \)
  - \( 3 \): live\( \{x\} \)
  - \( 4 \): live\( \{x, y\} \)
**Live Variables: Computing values for a node**

- Keep track of the (forward) references that have not found their (backward) definition site.
- Flow out of a node depends on flow in and on what happens in the node.

```
int x, y;
...
x := foo();
y := x + 2;
if(x > 0 then
    x := x + y;
end if;
...
```

**Example: Possible Live Variables**

```
x = foo()
{ }
{ x}
{ x,y}

y = x + 2
{ x,y}
{ x, y}

if(x > 0)
{x,y}
{ x, y}

x = x + y
{ x, y}
{ x, y}
{ x}
{ x, y}
{ x, y}
```

Backward flow, any path problem, def/ref sets are the initial facts associated with each node.
Example: **Definite Live Variables**

```plaintext
int x,y;
...  
x := foo();
y := x + 2;
if x > 0 then 
x := x + y;
end if;
...
```

Backward flow, all path problem, def/ref sets are the initial facts associated with each node

```
x = foo()
{x}
y = x + 2
{x}
if(x > 0)
{ }

x = x + y
{x,y}
{ }

{ }
{ }
```

Example: **Definite Live Variables**

```plaintext
int x,y;
...  
x := foo();
y := x + 2;
if x > 0 then 
x := x + y;
end if;
...
```

• Backward flow, all path problem
• def/ref sets are the initial facts associated with each node

• \[ \text{In}(n) = \bigcap_{k \in \text{succ}(n)} \text{Out}(k) \]
• \[ \text{Out}(n) = \text{In}(n) \cap \text{def}(n) \cup \text{ref}(n) \]
### Data Flow Analysis decisions

- Backward/Forward
- Any/All
- Facts
- Equations
- Initial values

### Constant Propagation

- Some variables at a point in a program may only take on one value
- If we know this, can optimize the code when it is compiled
**Constant Propagation**

```c
int x, y;
...
x := 3;
y := x + 2;
if x > z then
  x := x + y;
end if;
...
```

**Forward flow, all paths problem**

**Facts are the computations and assignments made at each node**

**U=unknown, N= not a constant**
**Constant Propagation with a loop**

1. Forward flow, all paths problem
2. Facts are the computations and assignments made at each node

**Fixed point**

- The data flow analysis algorithm will eventually terminate
  - If there are only a finite number of possible sets that can be associated with a node
  - If the function that determines the sets that can be associated with a node is monotonic
Constant Propagation with a loop

Forward flow, all paths problem
Facts are the computations and assignments made at each node

DAVE: detects anomalous def/ref behavior

- Application independent specification of erroneous behavior
  - use of an uninitialized variable
  - redefinition of a variable that is not referenced
**Anomalous pairs of ref/def information**

**d** - defined, **r** - referenced, **u** - undefined

- **d..r**: defined variable reaches a reference
- **u..d**: undefined variable reaches a definition
- **d..d**: definition is redefined before being used
- **d..u**: definition is undefined before being used
- **u..r**: undefined variable reaches a reference

---

**Consider an unreferenced definition**

- **For a definition of** a, **want to know if that definition is not going to be referenced**
  - Is there some path where a is redefined or undefined before being used?
  - May be indicative of a problem
  - Usually just a programming convenience and not a problem

- **For a definition of** a, **want to know if on all paths is a redefined or undefined before being used**
  - May be indicative of a problem
  - Or could just be wasteful
Some versus All

For some path, a is redefined without a subsequent reference

For all paths, a is redefined without a subsequent reference

Unreferenced definition: Computing values for a node

• Keep track of the definitions into a node that have not been referenced
• Flow out of a node depends on flow in and on what happens in the node
Unreferenced definition: Computing values for a node

- Keep track of the definitions into a node that have not been referenced
- Flow out of a node depends on flow in and on what happens in the node

Forward flow equation:

\[ \text{In}(n) = \bigcap_{k \in \text{pred}(n)} \text{Out}(k) \]

\[ \text{Out}(n) = \text{In}(n) - \text{ref}(n) \cup \text{def}(n) \]

Unreferenced definitions

```c
int x,y;
...
x := foo();
y := x + 2;
if x > 0 then
  x := x + 1;
end if;
y:= ...
```

Forward flow, all paths problem
Continuing with the unreferenced def example

- A definition is redefined without being used if `def(n) ∈ In(n) - ref(n)`
- Must compute this for each node
- For this example, the last node would report a redefinition of `y` on all paths
- Finds the location where the redef occurs

Unreferenced definitions, (finds the location of the first def of the pair)

```c
int x, y;
...
    x := foo();
    y := x + 2;
    if x > 0 then
        x := x + 1;
    end if;
    y := ...
```

Backward flow, all paths problem
In values depend on direction

Data Flow Analysis Problem

• Need to determine the information that should be computed at a node
• Need to determine how that information should flow from node to node
  • Backward or Forward
  • Union or Intersection
• Often there is more than one way to solve a problem
  • Can often be solved forward or backward, but usually one way is easier than the other
• Next class look at a general framework for defining dfa problems