How Good Is Iterative Data Flow Analysis?

A single execution of a program will follow some path $b_0, b_{i_1}, b_{i_2}, ..., b_{i_n}$.

The Data Flow solution along this path is $f_{i_n}(f_{i_2}(f_{i_1}(f_0(T))))... = f(b_0, b_1, ..., b_{i_n})$

The best possible static data flow solution at some block $b$ is computed over all possible paths from $b_0$ to $b$.

Let $P_b$ = The set of all paths from $b_0$ to $b$.

$$MOP(b) = \bigwedge_{p \in P_b} f(p)$$

Any particular path $p_i$ from $b_0$ to $b$ is included in $P_b$.

Thus $MOP(b) \land f(p_i) = MOP(b) \leq f(p_i)$.

This means $MOP(b)$ is always a safe approximation to the “true” solution $f(p_i)$.

If we have the distributive property for transfer functions,

$f(a \land b) = f(a) \land f(b)$

then our iterative algorithm always computes the MOP solution, the best static solution possible.

To prove this, note that for trivial path of length 1, containing only the start block, $b_0$, the algorithm computes $f_0(T)$ which is $MOP(b_0)$ (trivially).

Now assume that the iterative algorithm for paths of length $n$ or less to block $c$ does compute $MOP(c)$.

We’ll show that for paths to block $b$ of length $n+1$, $MOP(b)$ is computed.

Let $P$ be the set of all paths to $b$ of length $n+1$ or less.

The paths in $P$ end with $b$.

$$MOP(b) = f_b(f(P_1)) \land f_b(f(P_2)) \land ...$$

where $P_1, P_2, ...$ are the prefixes (of length $n$ or less) of paths in $P$ with $b$ removed.

Using the distributive property,

$$f_b(f(P_1)) \land f_b(f(P_2)) \land ... = f_b(f(P_1) \land f(P_2) \land ...).$$

But note that $f(P_1) \land f(P_2) \land ...$ is just the input to $f_b$ in our iterative algorithm, which then applies $f_b$.

Thus $MOP(b)$ for paths of length $n+1$ is computed.
For data flow problems that aren’t distributive (like constant propagation), the iterative solution is ≤ the MOP solution. This means that the solution is a safe approximation, but perhaps not as “sharp” as we might wish.

Reading Assignment

Read “An Efficient Method of Computing Static Single Assignment Form.” (Linked from the class Web page.)

Exploiting Structure in Data Flow Analysis

So far we haven’t utilized the fact that CFGs are constructed from standard programming language constructs like IFs, Fors, and Whiles. Instead of iterating across a given CFG, we can isolate, and solve symbolically, subgraphs that correspond to “standard” programming language constructs.

We can then progressively simplify the CFG until we reach a single node, or until we reach a CFG structure that matches no standard pattern.

In the latter case, we can solve the residual graph using our iterative evaluator.

Three Program-Building Operations

1. Sequential Execution (“;”)
2. Conditional Execution (If, Switch)
3. Iterative Execution (While, For, Repeat)
Sequential Execution

We can reduce a sequential “chain” of basic blocks:

\[ b_1 \rightarrow b_2 \rightarrow \ldots \rightarrow b_n \]

into a single composite block:

\[ b_{\text{seq}} \]

The transfer function of \( b_{\text{seq}} \) is

\[ f_{\text{seq}} = f_n \circ f_{n-1} \circ \ldots \circ f_1 \]

where \( \circ \) is functional composition.

Conditional Execution

Given the basic blocks:

\[ b_p \]

\[ b_{L1} \rightarrow b_{L2} \]

we create a single composite block:

\[ b_{\text{cond}} \]

The transfer function of \( b_{\text{cond}} \) is

\[ f_{\text{cond}} = f_{L1} \circ b_p \wedge f_{L2} \circ b_p \]

Iterative Execution

Repeat Loop

Given the basic blocks:

\[ b_B \rightarrow b_C \]

we create a single composite block:

\[ b_{\text{repeat}} \]

Here \( b_B \) is the loop body, and \( b_C \) is the loop control.

If the loop iterates once, the transfer function is \( f_C \circ b_B \).

If the loop iterates twice, the transfer function is \((f_C \circ b_B) \circ (f_C \circ b_B)\).

Considering all paths, the transfer function is \((f_C \circ b_B) \wedge (f_C \circ b_B)^2 \wedge \ldots\)

Define \( \text{fix} f \equiv f \wedge f^2 \wedge f^3 \wedge \ldots \)

The transfer function of repeat is then

\[ f_{\text{repeat}} = \text{fix}(f_C \circ b_B) \]
While Loop.

Given the basic blocks:

```
  b_C
  ↓
  b_B
```

we create a single composite block:

```
  b_While
```

Here again b_B is the loop body, and b_C is the loop control.
The loop always executes b_C at least once, and always executes b_C as the last block before exiting.

The transfer function of a while is therefore

\[ f_{\text{while}} = f_C \land \text{fix}(f_C \circ f_B) \circ f_C \]

Evaluating Fixed Points

For lattices of height H, and monotone transfer functions, fix f needs to look at no more than H terms.
In practice, we can give fix f an operational definition, suitable for implementation:

Evaluate

```
(fix f)(x) {
  prev = soln = f(x);
  while (prev \neq new = f(prev)){
    prev = new;
    soln = soln \land new;
  }
  return soln;
}
```

Example—Reaching Definitions

The transfer functions are either constant-valued \( f_1=\{b_1\}, f_4=\{b_4\}, f_5=\{b_5\} \) or identity functions \( f_2=f_3=f_6=f_7=\text{Id} \).
First we isolate and reduce the conditional:
\[ f_C = f_4 \circ f_3 \land f_5 \circ f_3 = \{b4\} \circ \text{Id} \cup \{b5\} \circ \text{Id} = \{b4,b5\} \]

Substituting, we get
\[ f_6 \circ f_C = \text{Id} \circ f_C = f_C \]

We can combine \( b_C \) and \( b_6 \), to get a block equivalent to \( b_C \). That is,
\[ f_6 \circ f_C = \text{Id} \circ f_C = f_C \]

We now have
\[ f_W = f_2 \land (\text{fix}(f_2 \circ f_C)) \circ f_2 = \text{Id} \cup (\text{fix}(\text{Id} \circ f_C)) \circ \text{Id} = \]
\[ \text{Id} \cup (\text{fix}(f_C)) = \]
\[ \text{Id} \cup (f_C \land f_C^2 \land f_C^3 \land ...) = \]
\[ \text{Id} \cup \{b4,b5\} \]

We compose these three sequential blocks to get the whole solution, \( f_P \).
\[ f_P = \text{Id} \circ (\text{Id} \cup \{b4,b5\}) \circ \{b1\} = \{b1,b4,b5\}. \]
These are the definitions that reach the end of the program.
We can expand subgraphs to get the solutions at interior blocks.
Thus at the beginning of the while, the solution is \{b1\}.
At the head if the If, the solution is
(Id U (Id o f_C o Id) U
(Id o f_C o Id o f_C o Id) U ... )o({b1})
= \{b1\} U \{b4, b5\} U \{b4, b5\} U ...
= \{b1, b4, b5\}
At the head of the then part of the If, the solution is Id(\{b1, b4, b5\}) = \{b1, b4, b5\}.

Static Single Assignment Form

Many of the complexities of optimization and code generation arise from the fact that a given variable may be assigned to in many different places.
Thus reaching definition analysis gives us the set of assignments that may reach a given use of a variable.
Live range analysis must track all assignments that may reach a use of a variable and merge them into the same live range.
Available expression analysis must look at all places a variable may be assigned to and decide if any kill an already computed expression.

What If

each variable is assigned to in only one place?
(Much like a named constant).
Then for a given use, we can find a single unique definition point.
But this seems impossible for most programs—or is it?

In Static Single Assignment (SSA)
Form each assignment to a variable, \(v\), is changed into a unique assignment to new variable, \(v_i\).
If variable \(v\) has \(n\) assignments to it throughout the program, then (at least) \(n\) new variables, \(v_1\) to \(v_n\), are created to replace \(v\). All uses of \(v\) are replaced by a use of some \(v_i\).

Phi Functions

Control flow can’t be predicted in advance, so we can’t always know which definition of a variable reached a particular use.
To handle this uncertainty, we create phi functions.
As illustrated below, if \(v_i\) and \(v_j\) both reach the top of the same block, we add the assignment
\[ v_k \leftarrow \phi(v_i, v_j) \]
to the top of the block.
Within the block, all uses of \(v\) become uses of \(v_k\) (until the next assignment to \(v\)).
What does \( \phi(v_i, v_j) \) Mean?

One way to read \( \phi(v_i, v_j) \) is that if control reaches the phi function via the path on which \( v_i \) is defined, \( \phi \) “selects” \( v_i \); otherwise it “selects” \( v_j \).

Phi functions may take more than 2 arguments if more than 2 definitions might reach the same block.

Through phi functions we have simple links to all the places where \( v \) receives a value, directly or indirectly.

Example

In SSA form computing live ranges is almost trivial. For each \( x_i \) include all \( x_j \) variables involved in phi functions that define \( x_i \).

Initially, assume \( x_1 \) to \( x_6 \) (in our example) are independent. We then union into equivalence classes \( x_i \) values involved in the same phi function or assignment.

Thus \( x_1 \) to \( x_3 \) are unioned together (forming a live range). Similarly, \( x_4 \) to \( x_6 \) are unioned to form a live range.

Constant Propagation in SSA

In SSA form, constant propagation is simplified since values flow directly from assignments to uses, and phi functions represent natural “meet points” where values are combined (into a constant or \( \bot \)).

Even conditional constant propagation fits in. As long as a path is considered unreachable, it variables are set to T (and therefore ignored at phi functions, which meet values together).
We have determined that $i=6$ everywhere.

### Putting Programs into SSA Form

Assume we have the CFG for a program, which we want to put into SSA form. We must:

- Rename all definitions and uses of variables
- Decide where to add phi functions

Renaming variable definitions is trivial—each assignment is to a new, unique variable.

After phi functions are added (at the heads of selected basic blocks), only one variable definition (the most recent in the block) can reach any use. Thus renaming uses of variables is easy.

### Placing Phi Functions

Let $b$ be a block with a definition to some variable, $v$. If $b$ contains more than one definition to $v$, the last (or most recent) applies.

What is the first basic block following $b$ where some other definition to $v$ as well as $b$’s definition can reach?

In blocks dominated by $b$, $b$’s definition must have been executed, though other later definitions may have overwritten $b$’s definition.
Domination Frontiers (Again)

Recall that the Domination Frontier of a block b, is defined as

$$\text{DF}(N) = \{Z \mid M \rightarrow Z \& (N \text{ dom } M) \& \neg(N \text{ sdom } Z)\}$$

The Dominance Frontier of a basic block N, DF(N), is the set of all blocks that are immediate successors to blocks dominated by N, but which aren’t themselves strictly dominated by N.

Assume that an initial assignment to all variables occurs in $b_0$ (possibly of some special “uninitialized value.”)

We will need to place a phi function at the start of all blocks in b’s Domination Frontier.

The phi functions will join the definition to v that occurred in b (or in a block dominated by b) with definitions occurring on paths that don’t include b.

After phi functions are added to blocks in DF(b), the domination frontier of blocks with newly added phi’s will need to be computed (since phi functions imply assignment to a new $v_i$ variable).

Examples of How Domination Frontiers Guide Phi Placement

$$\text{DF}(N) = \{Z \mid M \rightarrow Z \& (N \text{ dom } M) \& \neg(N \text{ sdom } Z)\}$$

Simple Case:

Here, $(N \text{ dom } M)$ but $\neg(N \text{ sdom } Z)$, so a phi function is needed in Z.

Loop:

Here, let $M = Z = N$. $M \rightarrow Z$, $(N \text{ dom } M)$ but $\neg(N \text{ sdom } Z)$, so a phi function is needed in Z.

$$\text{DF}(N) = \{Z \mid M \rightarrow Z \& (N \text{ dom } M) \& \neg(N \text{ sdom } Z)\}$$
Sometimes Phi’s must be Placed Iteratively

Now, DF(b1) = {b3}, so we add a phi function in b3. This adds an assignment into b3. We then look at DF(b3) = {b5}, so another phi function must be added to b5.

Phi Placement Algorithm

To decide what blocks require a phi function to join a definition to a variable v in block b:

1. Compute $D_1 = DF(b)$.
   Place Phi functions at the head of all members of $D_1$.

2. Compute $D_2 = DF(D_1)$.
   Place Phi functions at the head of all members of $D_2 - D_1$.

3. Compute $D_3 = DF(D_2)$.
   Place Phi functions at the head of all members of $D_3 - D_2 - D_1$.

4. Repeat until no additional Phi functions can be added.

PlacePhi{
    For (each variable v ∈ program) {
        For (each block b ∈ CFG) {
            PhiInserted(b) = false
            Added(b) = false
        }
        List = φ
        For (each b ∈ CFG that assigns to V) {
            Added(b) = true
            List = List U {b}
        }
        While (List ≠ φ) {
            Remove any b from List
            For (each d ∈ DF(b)) {
                If (! PhiInserted(d)) {
                    Add a Phi Function to d
                    PhiInserted(d) = true
                    If (! Added(d)) {
                        Added(d) = true
                        List = List U {d}
                    }
                }
            }
        }
    }
}

Example

Initially, List={1,3,5,6}

Process 1: DF(1) = φ

Process 3: DF(3) = 4, so add 4 to List and add phi fct to 4.

Process 5: DF(5)={4,5} so add phi fct to 5.

Process 5: DF(6) = {5}

Process 4: DF(4) = {4}

We will add Phi’s into blocks 4 and 5. The arity of each phi is the number of in-arcs to its block. To find the args to a phi, follow each arc “backwards” to the sole reaching def on that path.
SSA and Value Numbering

We already know how to do available expression analysis to determine if a previous computation of an expression can be reused.

A limitation of this analysis is that it can’t recognize that two expressions that aren’t syntactically identical may actually still be equivalent.

For example, given
\[ t_1 = a + b \]
\[ c = a \]
\[ t_2 = c + b \]

Available expression analysis won’t recognize that \( t_1 \) and \( t_2 \) must be equivalent, since it doesn’t track the fact that \( a = c \) at \( t_2 \).

Value Numbering

An early expression analysis technique called value numbering worked only at the level of basic blocks. The analysis was in terms of “values” rather than variable or temporary names.

Each non-trivial (non-copy) computation is given a number, called its value number.

Two expressions, using the same operators and operands with the same value numbers, must be equivalent.

For example,
\[ t_1 = a + b \]
\[ c = a \]
\[ t_2 = c + b \]

is analyzed as
\[ v_1 = a \]
\[ v_2 = b \]
\[ t_1 = v_1 + v_2 \]
\[ c = v_1 \]
\[ t_2 = v_1 + v_2 \]

Clearly \( t_2 \) is equivalent to \( t_1 \) (and hence need not be computed).