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:

\[ b_1 \rightarrow b_2 \rightarrow \cdots \rightarrow b_n \]

into a single composite block:

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

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

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

where \( \circ \) is functional composition.
Conditional Execution

Given the basic blocks:

\[ b_p \]

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

we create a single composite block:

\[ b_{cond} \]

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

\[ f_{cond} = f_{L1} \circ f_p \land f_{L2} \circ f_p \]
Iterative Execution

Repeat Loop
Given the basic blocks:

\[
\begin{align*}
&b_B \\
&\downarrow \\
&b_C
\end{align*}
\]

we create a single composite block:

\[
\begin{align*}
&b_{\text{repeat}}
\end{align*}
\]

Here $b_B$ is the loop body, and $b_C$ is the loop control.
If the loop iterates once, the transfer function is \( f_C \circ f_B \).

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

Considering all paths, the transfer function is \((f_C \circ f_B) \land (f_C \circ f_B)^2 \land \ldots\).

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

The transfer function of repeat is then

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

Given the basic blocks:

![Diagram of While Loop]

we create a single composite block:

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, $\text{fix } f$ needs to look at no more than $H$ terms.

In practice, we can give $\text{fix } f$ an operational definition, suitable for implementation:

Evaluate

$$(\text{fix } f)(x) \{$$

$\quad \text{prev} = \text{soln} = f(x);$$

$\quad \text{while } (\text{prev} \neq \text{new} = f(\text{prev})) \{$$

$\quad \quad \text{prev} = \text{new};$$

$\quad \quad \text{soln} = \text{soln} \land \text{new};$$

$\quad \}$$

$\quad \text{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 ($f_2=f_3=f_6=f_7=Id$).
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

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

We isolate and reduce the while loop formed by \( b_2 \) and \( b_C \), creating \( b_W \).

The transfer function is

\[
f_W = f_2 \land (\text{fix}(f_2 \circ f_C) \circ f_2) =
\]

\[
\text{Id} \cup (\text{fix}(\text{Id} \circ f_C) \circ \text{Id}) =
\]

\[
\text{Id} \cup (\text{fix}(f_C)) =
\]

\[
\text{Id} \cup (f_C \land f_C^2 \land f_C^3 \land ...) =
\]

\[
\text{Id} \cup \{b4,b5\}
\]
We now have

$$f_P = Id \circ (Id \cup \{f_4,f_5\}) \circ \{b_1\} = \{b_1,b_4,b_5\}.$$ 

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

\[(\text{Id} \cup (\text{Id} \circ f_C \circ \text{Id}) \cup (\text{Id} \circ f_C \circ \text{Id} \circ f_C \circ \text{Id}) \cup \ldots )({\{b1}\}) = \{b1\} \cup \{b4,b5\} \cup \{b4,b5\} \cup \ldots = \{b1,b4,b5\}\]

At the head of the then part of the If, the solution is \(\text{Id}(\{b1,b4,b5\}) = \{b1,b4,b5\}\).
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

Original CFG

```
x=1
a=x
b=x
x=1
x==10
c=x
x++
print x
```

CFG in SSA Form

```
x_1=1
a=x_1
b=x_3
x_3=\phi(x_1,x_2)
x_4=1
x_5=\phi(x_4,x_6)
x_5==10
c=x_5
x_6=x_5+1
print x_5
```
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 \( \perp \)).

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).
Example

\[
\begin{align*}
    i &= 6 \\
    j &= 1 \\
    k &= 1 \\
\end{align*}
\]

repeat
    if \(i == 6\)
        \(k = 0\)
    else
        \(i = i + 1\)
    \(i = i + k\)
    \(j = j + 1\)
until \(i == j\)

\[
\begin{align*}
    i_1 &= 6 \\
    j_1 &= 1 \\
    k_1 &= 1 \\
\end{align*}
\]

repeat
    \(i_2 = \phi(i_1, i_5)\)
    \(j_2 = \phi(j_1, j_3)\)
    \(k_2 = \phi(k_1, k_4)\)
    if \(i_2 == 6\)
        \(k_3 = 0\)
    else
        \(i_3 = i_2 + 1\)
        \(i_4 = \phi(i_2, i_3)\)
        \(k_4 = \phi(k_3, k_2)\)
        \(i_5 = i_4 + k_4\)
        \(j_3 = j_2 + 1\)
until \(i_5 == j_3\)

We have determined that \(i = 6\) everywhere.