The idea of static analysis

You need to know some *facts* about the run-time behavior of the program

- local facts at a specific statement, e.g., is a local variable live at a procedure call site?
- all the places where a fact is true, e.g., statements that are not executable (i.e., dead code)
- global facts, e.g., no uninitialized variables, deadlock freedom

The kind of facts you are interested in and the precision with which you need to know those facts determine the analysis needed.

Move calculation to compile time

basic approach: all calculation happens at run time

Questions

- Are the operand values for an operator known statically?
- What are the operand values?

Move calculation to compile time

`x := 3 + y`   `x := 3 + c`

The value of `y` has the same constant value, `c`, *on all executions* reaching this point.

Maximize usage of registers

basic approach: variables get registers for entire program run

Questions

- What is the lifetime of a variable? (scope rules)
- For what portion of a variables lifetime is it actually used?

Maximize usage of registers

```
[1]  x := 3 + y                *
[2]  if (c) then             *
[3]    z := x;               *
     else                        +
[4]    z := 1;                   +
... no more uses of x ...    +
```

The register for `x` can be reassigned at a point where the value of `x` is used not subsequently on *any program execution*.

Eliminate redundant computation

basic approach: recalculate all expressions in the program as needed

Questions

- Does an equivalent expression occur before the expression of interest?
- Can we save the value of the previous expression and reuse it?

Eliminate redundant computation

```
[1]  x := 3 + y;
     ... no definitions of y ...
[2]  z := y + 3;
```

The expression has been computed *on all executions* reaching this point and none of the sub-terms have changed values since that computation.

Minimize test set size

One form of coverage criteria in software testing is *data flow* coverage, where tests exercise paths that connect generators of values to subsequent consumers of those values.

```
[1]  x := 3 + y;
[2]  if (c) then
[3]    y := x;
[4]  else
[5]    x := 1;
[6]    y := x + 3;
[7]  z := x + y
```

Minimize test set size

Considering variable `x`, do we need test cases for

- value generated at `[1]` and consumed at `[3]`
- value generated at `[1]` and consumed at `[6]`
- value generated at `[1]` and consumed at `[7]`
- value generated at `[5]` and consumed at `[3]`
- value generated at `[5]` and consumed at `[6]`
- value generated at `[5]` and consumed at `[7]`

Minimize test set size

Questions

- What pairs of generators/consumers can occur at run-time?
- Can we detect impossible pairs?

A pair (d,u) is impossible, if there is *no program execution* on which the value assigned to `x` at `[d]` is the value referenced in `x` at `[u]`.

Approximation in analyses

Fundamental decidability results tell us that we can’t have accurate answers to arbitrary questions about programs (e.g., halting).

Two options:

- restrict the questions we ask (e.g., type checking)
- approximate the answers

Analyses should be designed to produce all of the *facts* that are true of the program’s dynamic behavior.

Approximation in analyses

There is a tension between the degree of approximation (precision of the results of an analysis) and the efficiency of the analysis.

This will show up later in the course in many forms

- flow sensitivity
- context sensitivity
- qualified/combined analyses

In general, the less precise the faster the analysis

Your application determines how you make this tradeoff

Approximation terminology

In the readings in this course you will encounter lots of different terms for the same concept

- conservativeness
- pessimism
- safety
- soundness

These are all refering to the idea that an analysis should produce at least the set of facts that are true of the program's dynamic behavior.

Start very simple

Simple non-procedural imperative language

A classic analysis problem

Formulating the flow analysis

Example program

```
y := x;
z := 1;
while y>1 do
(
  z := z*y;
  y := y-1
);
y := 0
```

computes `x` factorial into `z`

Syntax of *While*

Syntactic categories

$$\begin{array}{rcll} a & \in & \textsf{\textbf{AExp}} & \text{arithmetic expressions} \\ b & \in & \textsf{\textbf{BExp}} & \text{boolean expressions} \\ S & \in & \textsf{\textbf{Stmt}} & \text{statememts} \end{array}$$

Syntax of *While*

We will ignore lexical issues and assume that variables, scalars, labels, and operators are given

$$
\begin{array}{rclll}
x, y & \in & \textbf{Var} & \text{variables} \\
n & \in & \textbf{Num} & \text{scalar literals} \\
l & \in & \textbf{Lab} & \text{labels} \\
op_a & \in & \textbf{Op}_a & \text{arithmetic operators} \\
op_b & \in & \textbf{Op}_b & \text{boolean operators} \\
op_r & \in & \textbf{Op}_r & \text{relational operators}
\end{array}
$$

Syntax of *While*

We will ignore syntactic issues and assume that a concrete syntax is given that can produce programs for our abstract syntax

$$
\begin{array}{rcl}
a & ::= & x \mid n \mid a_1 \; op_a \; a_2 \\
b & ::= & \texttt{true} \mid \texttt{false} \mid \\
 & & \texttt{not} \;\; b \mid \\
 & & b_1 \; op_b \; b_1 \mid a_1 \; op_r \; a_2 \\
S & ::= & [x := a]^l \mid \\
 & & [\texttt{skip}]^l \mid \\
 & & S_1 ; S_2 \mid \\
 & & \texttt{if} \;\; [b]^l \;\; \texttt{then} \;\; S_1 \;\; \texttt{then} \;\; S_2 \mid \\
 & & \texttt{while} \;\; [b]^l \;\; \texttt{do} \;\; (\;\; S_1 \;\;)
\end{array}
$$

Example in *While* Language

$$
\begin{array}{l}
\texttt{[y := x]}^1\texttt{;} \\
\texttt{[z := 1]}^2\texttt{;} \\
\texttt{while [y>1]}^3\texttt{ do} \\
\texttt{(} \\
\quad \texttt{[z := z*y]}^4\texttt{;} \\
\quad \texttt{[y := y-1]}^5 \\
\texttt{);} \\
\texttt{[y := 0]}^6
\end{array}
$$

Labels let us distinguish like statements