CSE 231 Midterm

Due in class on Wednesday May 13th, 2015

Name:

Student ID:

Instructions: This is a open-book midterm on which you will work individually. "open-book" means that you have access to all the materials from the CSE 231 class web page.

May-Have-Sign Analysis

In this problem, your task will be to design a may-have-sign analysis. At each program point, the analysis computes elements of the set $\{+, -, 0\}$ for each variable X. The meaning of $+$ for variable X is that X may be positive (where "positive" means $X > 0$). Likewise, $-$ means that X may be negative (where "negative" means $X < 0$), and finally 0 means that X may be zero.

To start, we will consider a simple language without pointers, and then we will add pointers. In particular, consider the language where statements are taken from the following grammar:

$$\begin{array}{lll} S & ::= & X := C \mid X := Y \mid X := Y \; op \; Z \\ X, Y, Z & ::= & \text{Variable names} \\ C & ::= & \text{Integer constant literals} \\ op & ::= & + \mid - \mid * \end{array}$$

In addition to the above statements, the language also supports regular C-style control constructs, like `if` and `while`, which in our CFG will be represented using:

- branch statements of the form $\texttt{if}(X == C)$. Branch statements have one incoming edge and two outgoing edges.
- `merge` nodes that have two incoming edges and one outgoing edge.

To narrow things down, we will make the domain D be $Powerset(Vars \times \{+, -, 0\})$ where $Vars$ is the set of variables in the program. For example, an element of this lattice would look like this:

$$\{(\texttt{a}, +), (\texttt{a}, 0), (\texttt{b}, -), (\texttt{b}, 0), (\texttt{c}, 0)\}$$

If this set were returned by the analysis for a given edge in the CFG, it would mean that `a` may be positive or zero, `b` may be negative or zero, and `c` may be zero.

To give you a sense for how the analysis should work, consider the following code snippet, in which we assume the entry dataflow fact is $\{(\texttt{d}, +), (\texttt{d}, -), (\texttt{d}, 0)\}$. The computed information is shown for every program point as an inline comment.

```
// {(d,+),(d,-),(d,0)}
b := 0
// {(d,+),(d,-),(d,0),(b,0)}
c := 1
// {(d,+),(d,-),(d,0),(b,0),(c,+)}
if (d == 0)
   // {(d,0),(b,0),(c,+)}
   a := 30
   // {(d,0),(b,0),(c,+),(a,+)}
else
   // {(d,+),(d,-),(b,0),(c,+)}
   a := -30
   // {(d,+),(d,-),(b,0),(c,+),(a,-)}
// {(d,0),(d,+),(d,-),(b,0),(c,+),(a,+),(a,-)}
if (a == 10)
   // {(d,0),(d,+),(d,-),(b,0),(c,+),(a,+)}
   a := b + a
   // {(d,0),(d,+),(d,-),(b,0),(c,+),(a,+)}
else
   // {(d,0),(d,+),(d,-),(b,0),(c,+),(a,+),(a,-)}
   a := b - c
   // {(d,0),(d,+),(d,-),(b,0),(c,+),(a,-)}
// {(d,0),(d,+),(d,-),(b,0),(c,+),(a,+),(a,-)}
c := a * b
// {(d,0),(d,+),(d,-),(b,0),(c,0),(a,+),(a,-)}
```

As a sanity check, make sure your analysis computes the above information correctly!

The information computed by this analysis is useful for performing dead code elimination. For example, consider what would happen if the condition for the second branch were `a == 0` instead of `a == 10`. Since the dataflow fact immediately prior to the branch tells us `a` may not be 0, we know that the true side of the branch is unreachable and can be removed from the program safely.

a) Define the lattice over which the dataflow analysis will run. The set D has already been defined above; you should define $\bot$ and $\top$, and then go on to define $\sqcup, \sqcap, \sqsubseteq$.

Solution

$$\bot = \emptyset$$
$$\top = \mathit{Vars} \times \{+, -, 0\}$$
$$\sqcup = \cup$$
$$\sqcap = \cap$$
$$\sqsubseteq = \subseteq$$

b) How large is your lattice (ie: what is the size of the D set), and what is the height of your lattice?

Solution

$$|D| = 2^{3|\mathit{Vars}|}$$
$$\mathit{Height} = 3|\mathit{Vars}|$$

c) Now you will write the flow functions for all the statement forms in the language (including branches and merge statements).

As a hint, here is the definition for the flow function for $X := Y$, along with a helper function *kill* that you will find useful and you should use in your own flow functions:

$$\begin{aligned} F_{X:=Y}(S) &= S - \mathit{kill}(X) \cup \{(X, s) \mid (X, s) \in S\} \\ \mathit{kill}(X) &= \bigcup_{s \in \{+,-,0\}} \{(X, s)\} \end{aligned}$$

Now fill in the definitions for the remaining flow functions:

$X := C$:

$X := Y \ op \ Z$:

Hint: Define an "abstract" version of op called $\widehat{op}$, and use it in your flow function. For example, the normal addition operation + can be abstracted to an operation $\widehat{+}$ that applies to signs rather than integers, like so: $- \ \widehat{+} \ + = \{+, -, 0\}$. In English, this says "the sum of a negative and positive number may be positive, negative, or zero." Finish defining the abstract operators $\widehat{+}$, $\widehat{-}$, and $\widehat{*}$, then use them to define $\widehat{op}$.

`merge` :

`branch` with condition $X == C$:

Hint: The flow function should have form $F_{\texttt{branch}(X==C)} = (out_T, out_F)$, where out_T and out_F are the facts computed for the outgoing true and false sides of the branch, respectively.

Solution

$$
\begin{array}{rcl}
kill(x) & = & \bigcup_{s\in\{+,-,0\}}\{(x,s)\} \\
sign(c) & = & \begin{cases} + & \text{if } c > 0 \\ - & \text{if } c < 0 \\ 0 & \text{if } c = 0 \end{cases} \\
F_{x:=c}(S) & = & S - kill(x) \cup \{(x, sign(c)\} \\
F_{x:=y}(S) & = & S - kill(x) \cup \{(x,s) \mid (y,s) \in S\} \\
F_{x:=y\ op\ z}(S) & = & S - kill(x) \cup \{(x,s) \mid (y,a) \in S \wedge (z,b) \in S \wedge s = a\ \widehat{op}\ b\}
\end{array}
$$

where

$$
\begin{array}{lllll}
+ & \widehat{+} & + & = & \{+\} \\
+ & \widehat{+} & - & = & \{+,-,0\} \\
+ & \widehat{+} & 0 & = & \{+\} \\
- & \widehat{+} & - & = & \{-\} \\
- & \widehat{+} & 0 & = & \{-\} \\
0 & \widehat{+} & 0 & = & \{0\}
\end{array}
\qquad
\begin{array}{lllll}
+ & \widehat{-} & + & = & \{+,-,0\} \\
+ & \widehat{-} & - & = & \{+\} \\
+ & \widehat{-} & 0 & = & \{+\} \\
- & \widehat{-} & + & = & \{-\} \\
- & \widehat{-} & - & = & \{+,-,0\} \\
- & \widehat{-} & 0 & = & \{-\} \\
0 & \widehat{-} & + & = & \{-\} \\
0 & \widehat{-} & - & = & \{+\} \\
0 & \widehat{-} & 0 & = & \{0\}
\end{array}
\qquad
\begin{array}{lllll}
+ & \widehat{*} & + & = & \{+\} \\
+ & \widehat{*} & - & = & \{-\} \\
+ & \widehat{*} & 0 & = & \{0\} \\
- & \widehat{*} & - & = & \{+\} \\
- & \widehat{*} & 0 & = & \{0\} \\
0 & \widehat{*} & 0 & = & \{0\}
\end{array}
$$

$\widehat{+}$ and $\widehat{*}$ are commutative, so we omit their symmetric cases.

$F_{\texttt{branch}(x == c)}(S) = (out_T, out_F)$ where

$$
\begin{aligned}
in(x) &= \{(x,s) \mid (x,s) \in S\} \\
out_T &= \begin{cases} S - kill(x) \cup (\{(x,+)\} \cap in(x)) & \text{if } c > 0 \\ S - kill(x) \cup (\{(x,0)\} \cap in(x)) & \text{if } c = 0 \\ S - kill(x) \cup (\{(x,-)\} \cap in(x)) & \text{if } c < 0 \end{cases} \\
out_F &= \begin{cases} S - \{(x,0)\} & \text{if } c = 0 \\ S & \text{otherwise} \end{cases}
\end{aligned}
$$

d) Simulate your analysis on the following code snippet. Assume the entry dataflow fact is {}.

```
y := 10;
z := -1;
if (x == 0) {
  x := 1;
  do {
    if (x == 42) {
      z := y + x;
    }
    else {
      x := x + z;
    }
    w := y - x;
  } while (x == -100)
}
else {
    w := 0;
}
t := w - z;
```

Draw the control flow graph on the next page, and write down the dataflow facts that are computed, as we did in class. If there is iteration, write the dataflow facts for each iteration.

e) We will now add pointers to the original language by extending its grammar as follows:

$$S \quad ::= \quad \ldots \mid X := \&Y \mid X := *Y \mid *X := Y$$

Assume that the sign of a pointer is unknown (i.e., may be positive, negative, or zero). Thus, the flow function for address assignment statements $X := \&Y$ is the following:

$$F_{X:=\&Y}(S) = S \cup \{(X, +), (X, -), (X, 0)\}$$

Furthermore, assume that you are given the result of a prior may-point-to analysis. That is, for each program point, you are provided a map *mayPointTo* that maps each variable to the set of variables to which it may point. For example, if *mayPointTo*(`a`) = {`a,b`} then `a` may point to `a` or `b`.

Write flow functions for these new statements.

$X := *Y$:

$*X := Y$:

Solution

$$
\begin{aligned}
F_{x:=\&y}(S) &= S \cup \{(x,+),(x,-),(x,0)\} \\
F_{x:=*y}(S) &= S - kill(x) \cup \{(x,s) \mid z \in mayPointTo(y) \wedge (z,s) \in S\} \\
F_{*x:=y}(S) &= S \cup \{(z,s) \mid z \in mayPointTo(x) \wedge (y,s) \in S\}
\end{aligned}
$$