C++ Programming

Exercise Sheet 12
Software engineering, EIM-I
Philipp Schubert
January 19, 2018

Solutions to this sheet are due on 26.01.2018 til 14:00. Please hand in a digital version of your answers via e-mail. The e-mails subject has to contain cppp. Do zip-compress your solutions. For questions please send mail or speak to me during the exercises.

Note: If you copy text elements / code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited. All of your files that belong to your solution have to be contained in a single .zip file that is named according to the following naming scheme: <name>_<surname>_solution<XX>.zip. Replace <name> and <surname> with your actual name and replace <XX> with the sheet number the solutions belong to. You can look up your results using this link [https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CCh1v-LwulBIA3RvKAI7SH2M/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CCh1v-LwulBIA3RvKAI7SH2M/edit?usp=sharing).

In this exercise sheet you get the chance to make yourself familiar with some of the very basics of static code analysis. You can achieve 16 points in total.

**Exercise 1.**
Draw the control-flow graphs for the following functions:

```
int foo(string s) {
    int i = -1;
    try {
        i = getSensorValue();
    } catch (runtime_error("bad!")) {
        cout << "could not read sensor value";
    }
    return i;
}
```

(3 P.)

```
int bar(int s, int e) {
    int i = s;
    for (int j = 1; j <= e; ++j) {
        i += j;
    }
    return i;
}
```

(3 P.)
**Important:** In the following you are free to choose whether you work on exercise 2 or on exercise 3. However, you are allowed to solve both exercises of course. (Solving both exercises might be especially interesting for those of you who would like to get a better idea of what we are doing in the Software Engineering Group.)

**Exercise 2.**
Performing a sign analysis, that is an analysis determining the sign of a variable which can be positive, negative or zero. We define this analysis to use tuples $(x, S)$ as data-flow facts, where $x$ is the identifier of a variable and $S \in \mathcal{L}$ is a set contained in the powerset lattice over the domain $D = \{\bot, +, -, 0, \top\}$:

\[
\begin{array}{c}
\top \\
\{+, -\} \\
\{+, 0\} \\
\{-, 0\} \\
\{+\} \\
\{-\} \\
\{0\} \\
\bot
\end{array}
\]

The special symbol $\bot = \emptyset$ denotes 'no information' whereas the special symbol $\top = \{+, -, 0\}$ denotes the most imprecise element in the lattice. The direction of this analysis is forward and the merge operator is $\cup$ (set union). An example, say at some program node $n$ you are left with the set $n_{out} = \{\langle a, \bot \rangle, \langle b, \{+\} \rangle, \langle c, \top \rangle, \langle d, \{+, -\} \rangle\}$, that means that the four data-flow facts in $n_{out}$ hold at statement $n$ stating there is a variable ...

- $a$ for which you have no information about its sign
- $b$ which is positive at this statement
- $c$ for which you cannot tell anything useful because it can have any sign
- $d$ which may be positive or negative

Observe the following code:

```cpp
void foo(bool b) {
    int a = 5;
    int c = 2;
    if (b) {
        a = a - 7;
    } else {
        a = a + 3;
    }
    cout << a;
}
```
a) Draw the control-flow graph for `foo()`. (3 P.)

b) Annotate each edge in the control-flow graph with the set of data-flow facts that hold after a certain node. If two different control-flow edges lead to the same node, their data-flow information have to be merged before they enter the next node. In this analysis set union must be used, meaning we go upwards in \( \mathcal{L} \) (and therefore become more imprecise). What tells you the analysis about the signs of \( a \) and \( c \) at the very end of `foo()`? (7 P.)

Exercise 3.
Observe the LLVM intermediate representation (IR) of a small function performing some arithmetic computations.

```c
define i32 @Z11my_functionj(i32) #4 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32 * %2, align 4
  store i32 1, i32 * %3, align 4
  store i32 1, i32 * %4, align 4
  br label %5

  ; <label>:5 ; preds = %13, %1
  %6 = load i32, i32 * %4
  %7 = load i32, i32 * %2
  %8 = icmp ule i32 %6, %7
  br i1 %8, label %9, label %16

  ; <label>:9 ; preds = %5
  %10 = load i32, i32 * %4
  %11 = load i32, i32 * %3
  %12 = mul i32 %11, %10
  store i32 %12, i32 * %3, align 4
  br label %13

  ; <label>:13 ; preds = %9
  %14 = load i32, i32 * %4
  %15 = add i32 %14, 1
  store i32 %15, i32 * %3, align 4
  br label %5

  ; <label>:16 ; preds = %5
  %17 = load i32, i32 * %3
  ret i32 %17
}
```

The LLVM language reference can be found on [llvm.org/docs/LangRef.html](https://llvm.org/docs/LangRef.html). Here is a short summary of the most relevant parts:

- The \( \%X \), where \( X \) is a number are variables
- \( \%0 \) is the first and only formal parameter that this function receives
- The \( i32 \) denotes a 32-bit integer type
- The `alloca` instruction allocates a variable on the stack (creates a local variable)
- The `store` stores a variable to a memory location
• The \textit{br} branches to the specified label (or labels depending on the first parameter)
• The \textit{load} instruction loads a variable from a memory location
• The \textit{icmp ule} instruction performs an unsigned less or equal integer comparison
• The \textit{ret} instruction returns a variable from a function

a) What does this function compute and what would be a better function name? (7 P.)
b) Translate the LLVM IR into a semantically equivalent piece of C++ code! (3 P.)