C++ Programming

Exercise sheet 12
Software engineering, EIM-I
Philipp Schubert

July 17, 2020

Solutions to this sheet are due on 24.07.2020 til 16:00. Please hand in a digital version of your answers via e-mail. The e-mail’s subject has to contain cppp20. Do zip-compress your solutions.

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_<X>.zip. Replace <name> and <surname> with your actual name and replace <X> with the number of the exercise sheet. You can look up your results using this link: [https://docs.google.com/spreadsheets/d/1LtRF02JZ2kX3o1vG4UJK87jaQoW2J3hp19dK2J3k/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1LtRF02JZ2kX3o1vG4UJK87jaQoW2J3hp19dK2J3k/edit?usp=sharing)

This exercise sheet will help you to familiarize yourself with the very basics of static program analysis. You can achieve 16 points in total.

Exercise 1.

Draw the control-flow graphs for the following functions:

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

(3 P.)

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

(3 P.)
Important: In the following, you are free to choose whether you would like to work on exercise 2 or exercise 3. You are also 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 \( \langle x, S \rangle \) as data-flow facts, where \( x \) is the identifier of a variable and \( S \in \mathcal{L} \) is a set drawn from the power set lattice over the domain \( D = \{ \perp, +, -, 0, \top \} \):

![Power set lattice diagram]

The special symbol \( \perp = \{ \} \) 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). For example, say at some program node \( n \) you are left with the set \( n_{out} = \{ \langle a, \perp \rangle, \langle b, \{ + \} \rangle, \langle c, \top \rangle, \langle d, \{ +, - \} \rangle \} \), 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 very statement
- \( c \) for which you cannot tell anything of use 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's value: " << a << \\
};
```
a) Draw foo()’s control-flow graph. (3 P.)

b) Annotate each edge in the control-flow graph with the set of data-flow facts that holds after its respective outgoing node. If two different control-flow edges lead to a common successor node, their data-flow information has to be merged before you can continue propagating the data-flow information. In this analysis, the merge operator is set union, meaning we go upwards in \( \mathcal{L} \) (and therefore gain imprecision). Report on the analysis’s results at the very end of foo()? (7 P.)

Exercise 3.
Observe the LLVM intermediate representation (IR) of the following function that performs some arithmetic computation.

```llvm
define dso_local i32 @_Z11my_functionii(i32 %x, i32 %y) #4 {
  entry:
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  %result = alloca i32, align 4
  %i = alloca i32, align 4
  store i32 %x, i32 *%x.addr, align 4
  store i32 %y, i32 *%y.addr, align 4
  %0 = load i32, i32 *%x.addr, align 4
  store i32 1, i32 *%i, align 4
  br label %for.cond

for.cond: ; preds = %for.inc, %entry
  %1 = load i32, i32 *%i, align 4
  %2 = load i32, i32 *%y.addr, align 4
  %cmp = icmp slt i32 %1, %2
  br i1 %cmp, label %for.body, label %for.end

for.body: ; preds = %for.cond
  %3 = load i32, i32 *%x.addr, align 4
  %4 = load i32, i32 *%result, align 4
  %mul = mul nsw i32 %4, %3
  store i32 %mul, i32 *%result, align 4
  br label %for.inc

for.inc: ; preds = %for.body
  %5 = load i32, i32 *%i, align 4
  %inc = add nsw i32 %5, 1
  store i32 %inc, i32 *%i, align 4
  br label %for.cond

for.end: ; preds = %for.cond
  %6 = load i32, i32 *%result, align 4
  ret i32 %6
}
```

The LLVM language reference can be found on [llvm.org/docs/LangRef.html](http://llvm.org/docs/LangRef.html).

Here is a short summary of the most relevant parts:

- %X, is a local variables
- i32 denotes a 32-bit integer type
• The `alloca` instruction allocates a variable on the stack (creates a local variable)
• The `store` instruction stores a value to a memory location (variable)
• The `br` instruction branches to the specified label (or labels depending on the first parameter)
• The `load` instruction loads a value from a memory location (variable)
• The `icmp slt` instruction performs a signed less than integer comparison
• The `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 program in LLVM IR into a semantically equivalent piece of C++ code! (3 P.)