Assignment 3

Implementing live variable analysis

Static Program Analysis and Constraint Solving
Master's Degree in Formal Methods in Computer Science
Year 2021/22

Submission deadline: October, 29th

Submission instructions: Students are required to submit a `.zip` file with the source code of the assignment. Feel free to use any language of your choice.

In Lesson 2 we have studied a live variable analysis that receives a program's Control Flow Graph (CFG) and yields, for every block in the CFG, the set of live variables before and after that block. Recall that a variable x is said to be *live* at some program point if there is an execution path from that point to the end of the program in which the current value of x is read.

This assignment consists in implementing this analysis. You can proceed as follows:

1. Implement an abstract data type for representing abstract syntax trees of arithmetic and boolean expressions. In this assignment we shall consider the following grammar for arithmetic expressions:

   $$e ::= n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$$

   where n denotes an integer literal and x denotes a program variable. Regarding boolean expressions, we shall use the following grammar:

   $$b ::= e_1 \leq e_2 \mid e_1 = e_2 \mid b_1 \wedge b_2$$

   Since we are not dealing with type-based analyses yet, it is recommended that you implement an abstract data type for each category, that is, one for arithmetic expressions and another one for boolean expressions (which should make use of the former).

   If you are not used to working with abstract syntax trees in programming languages, you can get some sample implementations written in Java and Haskell in *Campus Virtual*.

2. Implement a function (or method) that, given an arithmetic (resp. boolean) expression, returns the set of variables occurring in it.

3. Implement an abstract data type for representing CFGs. Each block in the CFG should be identified by a label, which is a natural number. We have three kinds of blocks: `skip` blocks, assignment blocks, and conditional blocks.

4. Define a *transfer function* (or method) that, given a block of the CFG with identifier n, and the set $LVOut_n$ of live variables at the exit of that block, returns the set $LVIn_n$ of live variables at the entrance of that block. This function should implement the formula shown in the slides:

$$LVIn_n = (LVOut_n - Kill_n) \cup Gen_n$$

   where $Kill_n$ and Gen_n depend on the contents of the block (see slides).

5. By using all of the above, implement the live variable analysis. The analysis should maintain a list/array of *LVIn* sets (one for each block), another list/array of *LVOut* sets, and apply the transfer function until a fixed point is reached.

How do I test my solution?
Ideally one would write a program that receives the source code of the program being analysed (written in *While* language). However, this would involve developing a parser, which is not within the scope of this course. If you want to test your solutions you can take one of the following approaches:

- Select some sample programs (for example, those shown in the slides) and hard-code them in the source code of your implementation (for example, in the `main` function/method).
- You can use a *While* parser available at `http://dalila.sip.ucm.es:4000/while_parser`. This parser, given a *While* program, returns a JSON object that represents its CFG. It can be accessed through an API. See `https://github.com/manuelmontenegro/while_parser_api` for more details.