Contents

1 Assesment 3 - Static Analysis and Constraint Solving | UCM 22-23

Tree view of the project:

```
.
|-- app
|   |-- Main.hs
|-- assesment1.cabal
|-- CHANGELOG.md
|-- LICENSE
|-- README.md
|-- src
    |-- Lib.hs
```

- *Main.hs* contains the `main` function of the program. This `main` function calls `mainLibFn` function of *Lib.hs*.
- *Lib.hs* contains the implemented **Live Variable Analysis**. Further details at the end of this file.
- *assesment1.cabal* contains the needed configuration for building and running the project.
- Other files such as LICENSE and CHANGELOG were generated automathically by doing `cabal init` and thus can be ignored.

1.1 Building & Compiling

As Cabal has been used the next steps should be followed for running the program (from the root folder of the project):

1.1.1 Building

-> Builds the project

```
cabal build
```

1.1.2 Running

-> Runs the project

```
cabal run
```

- Note: calls `cabal build` if project is not built on its last version

1.1.3 Cleaning

-> Cleans the build of the project

```
cabal clean
```

1.2 Testing the program

Two **while** program samples have been hardcoded and can be used to test the program. Both can be checked in the code and are shown below.

The first one is the corresponding example shown in the slides:

```
x := 1;
if y > 0 then
    x := x - 1;
end
x := 2;
```

This **while** program correspondant hardcoded version can be found under `nodes1` inside *src/Lib.hs*.

The other program is the one shown in the handout exercise:

```
x := 10;
if x > 0 then
    x := x - 1;
    y : y + x;
end
z := 2;
z := y * z;
```

This **while** program correspondant hardcoded version can be found under `nodes2` inside *src/Lib.hs*.

To switch between two provided programs and test both of them you must change the argument to `f` function in function `mainLibFn` (*src/Lib.hs*):

```
357  mainLibFn :: IO()
358  mainLibFn = do
359      let result = f nodes2 -- HERE
```

1.3 Some comments about the implementation

This part of the README tries to provide a quick view about the structure followed while designing the 'language', by defining the data structures required to satisfy the expressions shown in the statement of the exercise, although code tries to be aclarative through comments.

1. Expressions

Can be of two types: AExp (arithmetic) & BExp (boolean)

```
data Exp = AStmt AExp
    | BStmt BExp
```

1. Arithmetic expressions

```
data AExp = Const Int  -- Constan value
    | Var Char  -- Var (only of one single character)
    | AOp ABinOp AExp AExp  -- Arithmetic binary expression
```

Since arithmetic operators have no real significance in this implementation (they just represent **types** of operations but the result is never given), they've been included in a special data structure and represented with the type `ABinOp` as they always take two values.

```
data ABinOp = AAdd  -- (+) operator
    | ASub  -- (-) operator
    | AMul  -- (*) operator
```

3. Boolean expressions

```
data BExp = BLEq AExp AExp  -- less or equal than
    | BGr AExp AExp  -- greater than
    | BEq AExp AExp  -- equal as
    | BAnd BExp BExp  -- (&&) operator
```

Although the exercise only required less or equal for comparing values greater than is the same as negating previous statement, so for a more intuitive implementation of the harcoded expressions this operator has been added.

4. Blocks & Nodes

```
data CFGBlock = AssignBlock AExp AExp  -- Assignment of a value to a var
    | CondBlock BExp  -- conditional block type
```

This structure characterizes the expressions as block type: they can be just of two types, either assignments (so the left part e.g. the first AExp arg can come only of type (Var Char) and others been discarded then; right part can be any AExp) or conditional expressions e.g. boolean expressions (BExp type). This represents the idea of block inside each node of the correspondant graph built.

```
data CFG = Block {
    block :: CFGBlock,  -- block of the node of the graph (expression type)
    label :: Int,  -- label of the node
    succs :: [Int]  -- list of labels of each of the successors of the correspondant node of the grap
}
```

This structure characterizes the nodes of the graph, made up of a block (see previous structure), a label (the identifier) and a list of integers that represents the list of labels of its successors. This list should be implemented with a Set so no repetition can occur, but as implementation is hardcoded in the program it won't occur (but can be easily changed by just doing Set.fromList & Set.toList stuff...).

5. Analysis

By having the references of each successor then the `F` function can be coded by just having a map of (Int, CFG) type that maps an integer (the label of the correspondant node) to that node of the graph. Check the `F` function implemented in the code: *src/Lib.hs* -> `f :: [CFG]-> (LVMap, LVMap)`. `LVMap` is an alias for `Map Int VarSet` and `VarSet` is an alias for `Set Char`. In the case of `LVMap` maps an integer (the label of the correspondant node/block) to the values of either LVOut or LVIn of that block, represented by a set of chars a.k.a. the set of variables.