Getting Started With `While`

You can download the source files for the exercise from the course website:

https://strec.wp.mines-telecom.fr

Once you downloaded the `While` source package, you should have the following files in your working directory:

```
./src/WhileRun.cc
./src/WhileInterpreter.cc
./src/WhileAnalysis.cc
./src/While.g4
./src/WhileDeadCodeAnalysis.cc
./src/WhileCFG.cc
./src/WhileConstantRegisterAnalysis.cc
./src/WhileInterproceduralPipelineAnalysis.cc
./CMakeLists.txt
./include/WhileInterpreter.h
./include/WhileAnalysis.h
./include/WhileColor.h
./include/WhileLang.h
./include/WhileCFG.h
./test/sort.whl
./test/swap.whl
./test/fib.whl
./test/string.whl
./test/min.whl
./test/max.whl
./COPYING
```

Make a new directory `build` and execute the following commands in order to build the code:

```
cmake ..
make -j12
```

This should build two executable files `while-analysis` and `while-run`. The latter is an interpreter of the language, which allows you to execute `While` programs. For instance, the following command will execute a simple insertion sort on a table of 5 integers:

```
./while-run ../test/sort.whl
```

You objective is to study simple analyses which are part of the program `while-analysis`.

1 Dead Code Analysis (20mn minutes)

Aims: *Understand the operation of a static analysis on simple programs.*

A static analysis in the `While` framework is always derived from the class `WhileDataFlowAnalysis`, which is defined in the file `./include/WhileAnalysis.h`. Here is an code excerpt of the relevant member functions:

```
template<typename D>
struct WhileDataFlowAnalysis : public WhileAnalysisInterface<D>
{
  // Functions to override inherited from WhileAnalysisInterface:
  virtual D transfer(const WhileInstr &i, const D input) = 0;
  virtual D join(std::list<D> inputs) = 0;

  virtual std::ostream &dump_first(std::ostream &s, const D &value) = 0;
  virtual std::ostream &dump_pre(std::ostream &s, const D &value) = 0;
  virtual std::ostream &dump_post(std::ostream &s, const D &value) = 0;
};
```

The abstract domain is modeled as a template parameter `D`, which can be defined freely. An analysis implementation in addition has to provide code for the member function `transfer` – modeling the transfer function of the analysis – and the member function `join` – modeling the join operator seen in the lecture.

The various *dump* functions also have to be implemented, they are used to display the analysis information alongside the analyzed program.

- Open the files `./include/WhileAnalysis.h` and `./src/WhileDeadCodeAnalysis.cc` and have a look at the code that defines a *Dead Code Analysis* implemented by the class `WhileDeadCode`.
- The abstract domain of this analysis is a simple enum:

  ```
  enum WhileReachability
  {
    REACHABLE,
    DEAD
  };
  ```

  Code that is marked with `REACHABLE` might be executed, while code that is marked with `DEAD` can never be executed.
- Run the analysis on the example program `./test/max.whl` as follows:

  ```
  ./while-analysis WDCA ../test/max.whl
  ```

- The analysis prints the control-flow graph (CFG) of the analyzed program. Code highlighted in green is `REACHABLE`, while code in red is `DEAD`. For the considered example only the last instruction, a `WRETURN` instruction is dead.
- Have a look at the implementation of the `join` function, shown below:

  ```
  WhileReachability join(std::list<WhileReachability> inputs) override
  {
  ```

```
  if (inputs.empty())
    return REACHABLE;

  for(WhileReachability r : inputs)
  {
    if (r == REACHABLE)
      return REACHABLE;
  }

  return DEAD;
}
```

The function takes a list of `WhileReachability` values as input, each corresponding to the analysis information at the end of a predecessor basic block. If the end of any of the predecessors is reachable, the code of the current basic block is considered reachable too. In addition a corner case is considered, all basic blocks that do not have any predecessors are considered reachable as well. Basic blocks where the analysis information for all predecessors is `DEAD` are in-turn considered dead.

- Finally, lets have a look at the `transfer` function:

```
WhileReachability transfer(const WhileInstr &i, const WhileReachability input) over
{
  WhileReachability result = input;
  switch(i.Opc)
  {
    case WBRANCH:
    case WRETURN:
      // code after those instructions is definitely dead
      result = DEAD;
      break;

    case WCALL:
    case WLOAD:
    case WSTORE:
    case WPLUS:
    case WMINUS:
    case WMULT:
    case WDIV:
    case WEQUAL:
    case WUNEQUAL:
    case WLESS:
    case WLESSEQUAL:
    case WBRANCHZ:
      // do not render code dead
      break;
  };

  return result;
}
```

The function contains a `switch` covering all possible instruction types of the `While` program representation. Only two kinds of instructions have an actual impact on the analysis: `WBRANCH` and `WRETURN` instructions. Any instruction that immediately follows one of these two kinds of instructions is definitely not reachable anymore by any other instruction. For all other instruction kinds the analysis simply preserves the input value, i.e., `input` is copied into `result`.

2 Constant Analysis (90mn minutes)

Aims: *Modify the code of a partial static analysis on simple programs.*

You task is now to complete the code of a *Constant Value Analysis*, as presented in the lecture. An initial skeleton of the analysis is provided in the file `./src/WhileConstantRegisterAnalysis.cc`.

- Lets first have a look at the analysis domain, which is a bit more complex than before:

  ```
  enum WhileConstantKind
  {
    TOP,
    BOTTOM,
    CONSTANT
  };

  struct WhileConstantValue
  {
    WhileConstantKind Kind;
    int Value;

    WhileConstantValue() : Kind(TOP), Value(0)
    {
    }

    WhileConstantValue(int value) : Kind(CONSTANT), Value(value)
    {
    }

    WhileConstantValue(WhileConstantKind kind) : Kind(kind), Value(0)
    {
    }
  };

  typedef std::map<int, WhileConstantValue> WhileConstantDomain;
  ```

  The abstract domain is a map (last line), which associates a register of the program representation (represented by an integer number) with a `WhileConstantValue`. Constant values may be in three distinct states, indicated by the member `Kind` of type `WhileConstantKind`:

  1. `TOP`:
     This state indicates that no decision has been made yet, i.e., the analysis has not determined yet whether the register's value is constant.
  2. `BOTTOM`:
     This state indicates that the analysis found a *contradiction*, i.e., the register may contain different values.

3. `CONSTANT`:
   This state indicates that the analysis was able to determine that the register always holds the same constant value. The value of the constant is stored in the member variable `Value`.

- Implement the `join` function combining two values of type `WhileConstantValue`. The current implementation of the function always returns `BOTTOM` and is shown below:

```
static WhileConstantValue join(const WhileConstantValue &a,
                               const WhileConstantValue &b)
{
  // TODO: Implement the join function (replacing BOTTOM).
  return BOTTOM;
}
```

  Your code should return a more sensible `WhileConstantValue` considering the rules explained in the lecture:

  1. If the two input values represent the same constant, return that constant.
  2. If one of the two input values is `TOP`, return the respective other value.
  3. In all other cases return `BOTTOM`.

  Compare your code with the explanations in the lecture.

- Implement the `transfer` function for which only the prototype is shown here for brevity:

```
WhileConstantDomain transfer(const WhileInstr &instr,
                             const WhileConstantDomain input) override;
```

  This function takes a `While` instruction and a `WhileConstantDomain`, i.e., a `map`, as input. The function is then supposed to model the effect of the instruction on the abstract value.

  The code already comes with two helper functions `readDataOperand` and `updateRegisterOperand`. The former obtains the abstract value associated with a register operand of an instruction, while the latter replaced the abstract value associated with a register operand by a new value. The use of these two functions is illustrated for some instructions in the code.

  You task is now the complete the code in order to obtain a working analysis:

  – Lets focus first on the `WCALL` instruction. The current code calls `updateRegisterOperand` and always provides the value `0`. This clearly is not right. Correct the code such that some reasonable value is used instead.

    Note that the analysis is intra-procedural, which means that you don't have any information on what other functions are doing.

  – Next complete the code of the `WLOAD` instruction. You have to add a call to `updateRegisterOperand`. Adapt the solution from the `WCALL` instruction. However, contrary to the `WCALL` instruction this kind of instructions has its destination register operand at another index position (`idx` argument) as illustrated by the comments in the code.

  – Finally implement the transfer function for all binary operators (`WPLUS` through `WLESSEQUAL`). Use the values obtained for the two input operands using `readDataOperand` and compute the result value. Then use `updateRegisterOperand` in order to update the abstract value of the destination register.

Also handle the case when one of the two operands is not a constant. Use `updateRegisterOperand` in order to update the abstract value of the destination register in this case.

- At this point you should have a working constant analysis. You can try it by running:

  ```
  /while-analysis WCRA ../test/max.whl
  ```

  Test your code with your own `While` code and make sure that it works correctly.

- Now extend the analysis to handle more cases, e.g., by exploiting basic mathematical properties of certain operations such as multiplication.