15-411 Compiler Design, Lab 2 Checkpoint (Fall 2020)

Seth and co.

Checkpoint Due: 11:59 PM, September 29th, 2020

1 Introduction

For this checkpoint, you will implement a Dataflow Analysis Framework. Dataflow analysis is not only crucial for liveness analysis in lab 2 and future labs, but also useful for numerous optimizations, such as partial redundancy elimination, deadcode elimination, and copy propagation.

For this checkpoint, we provide you with input files containing `gen` and `kill` sets at every program point. We also provide you with reference output files containing the correct `in` and `out` sets at each program point after dataflow analysis. Your task is to extend your compiler to take in input files containing `gen` and `kill` sets, and generate output files that match the reference output files we provide you. We will test 4 types of dataflow analysis for this checkpoint: `forward-must`, `forward-may`, `backward-must`, and `backward-may`. Recall that `May` analysis uses union as the meet operator, and involves asking about whether there exists a path along which some fact is true, while `Must` analysis uses intersection as the meet operator, and asks whether some fact is true along all paths. The type of analysis to perform will be passed by the `--r2` compiler option.

2 Input and Output Format

We specify the format of the input and output files below. We will use the abstraction *dataflow facts* to denote the unit of computation within gen and kills sets. In practice, *dataflow facts* might be temps for liveness analysis, or expressions for available expressions. *Dataflow facts* will be represented as strings of non-negative integers.

Input format: Each input file has the JSON format of the form:

```
[
     ...
    {
     "Gen": ["11","12"],
     "Kill": ["13"],
     "Successors": [11, 15],
     "Is_label": false,
     "Line": 10
    },
    ...
  ]
```

The input is a JSON array of JSON objects. Each JSON object contains information necessary for dataflow analysis at a single program point. Below is breakdown of what each field in a JSON object means: Here is a breakdown of what each field means:.

- **Gen:** Array of strings. Denotes the *facts* in the `gen` set at this program point.
- **Kill:** Array of strings. Denotes the *facts* in the `kill` set at this program point.
- **Successors:** Array of integers. Denotes the line numbers of the successors of this program point. For example, a normal instruction has a single successor which is the next line, a jump instruction has a single successor which might not be the next line, a conditional jump instruction has two successors, while a return instruction does not have any successors. For this checkpoint, all program points have *at most* 2 successors.
- **Is_label:** A boolean. Denotes whether the current instruction is a label. Labels are the targets for jump and conditional jump instructions and the beginning of basic blocks. This field is not necessary for correct dataflow analysis, but might help when building a control flow graph. You can assume that if a program point is a label, its `gen` and `kill` sets are empty.
- **Line:** An integer. Denotes the line number of this program point. You can actually infer this field from the index of this program point within the `Points` array, but we include this field to aid you with debugging.

Output format: The reference output files we provide you has the JSON format of the form:

```
[
  ...
  {
    "In": ["11", "12", "14"],
    "Out": ["13", "14"],
    "Line": 10
  },
  ...
]
```

The output is a JSON array of JSON objects. Below is breakdown of what each field in a JSON object means:

- **In:** Array of strings. Denotes the *facts* in the `In` set at this program point after performing dataflow analysis.
- **Out:** Array of strings. Denotes the *facts* in the `Out` set at this program point after performing dataflow analysis.
- **Line:** An integer. Denotes the line number of this program point. Note that this field is *optional* and not checked by the verifier. However, having this field will probably aid your debugging.

The output file must contain the exact same program points as the input file in the same order. Since the result of dataflow analysis is deterministic, we provide you the reference output files, which will be compared against the output files emitted by your compiler. The verifier checks that at each program point, the `In` and `Out` sets of your output file and the reference output file contain the same *facts* (though the ordering of the *facts* within `In` and `Out` sets can be different).

Running the verifier: Your compiler is expected to recognize an option `--r2` which, when present on the command line, tells the compiler to run the L2 checkpoint. When `--r2` is present, the source file passed in will be an input file for this checkpoint. The `--r2` option takes a single argument denoting the *type* of dataflow analysis to be performed. The type can be one of `forward-may`, `forward-must`, `backward-may`, `backward-must`.

The input files for this checkpoint are generated from the L2 test files. The input file generated from the L2 test file `foo.l2` will be named `foo.l2.in`. Each input file has 4 corresponding reference output files for each of the 4 different types of dataflow analysis. All 4 reference output files are named `foo.l2.out` but are placed in different directories. For example, the reference output files for `l2-basic-checkpoint/foo.l2.in` are `l2-basic-checkpoint-forward-may/foo.l2.out`, `l2-basic-checkpoint-forward-must/foo.l2.out`, `l2-basic-checkpoint-backward-may/foo.l2.out`, and `l2-basic-checkpoint-backward-must/foo.l2.out`.

Given the `--r2` option and source file `foo.l2.in`, your compiler should generate a output file called `foo.l2.out` in the same directory as `foo.l2.in` according to the JSON output format specified above. The verifier will then read the output file generated by your compiler and compare with the reference output file we provide you.

The verifier is an executable in the `dist/verifier` directory. To run the verifier on all input files in a certain directory, such as `dist/test/l2-basic-checkpoint`, you can run the command `../runverifier l2-basic-checkpoint` in `dist/compiler`. The `runverifier` script will make your compiler using the command `make lab2`, pass in each input file to your compiler with the `--r2` option to generate an output file, and then pass the generated output file to the verifier for scoring. By default, `runverifier` runs your compiler with all 4 types of dataflow analysis on each input file. You can pass in the flags `--forward-may`, `--forward-must`, `--backward-may` , and `--backward-must` so that `runverifier` only runs the passed in types of dataflow analysis on each input file. The output files generated by your compiler would be placed in `dist/log/l2-basic-checkpoint`. For each input file, only the output file for the last run will be found in the log (so if for an input file you passed the `forward-may` and `forward-must` directions but failed `backward-may`, the log file in the log will be your output for `backward-may`).

We provide 2 different verifier executables, for mac and ubuntu. On a mac, you should pass the `--mac` flag to the `runverifier` script. If no flag is passed, `runverifier` uses the ubuntu executable by default. We don't provide a verifier executable for windows, so if you are using the windows system, your best option might be to run the verifier on the docker containers we provide you.

Alternatively, you can also test a single pair of output file and reference output file with the verifier by running `<verifier executable path> <your_output_file_path> <reference_output_file_path>`.

3 Scoring

Correctness: The verifier compares the output file generated by your compiler with the corresponding reference output file, and checks whether at each program point, the *facts* within the `In` and `Out` fields are *identical* (though the ordering of *facts* can be different). You are also welcome to add more fields other than `In`, `Out`, and `Line` to aid your debugging.

Scoring: We will run each input file on 4 types of dataflow analysis, and your compiler must perform *all* 4 types of dataflow analysis correctly for an input file to receive points for the input file. The sum of your scores on all input files is your final score.

For this checkpoint, the tests in `l2-basic-checkpoint` are worth 20 points in total while the tests in `l2-large-checkpoint` are worth 60 points in total. You will also be deducted points for tests on which your compiler times out.

4 Submission

Same as lab1.

5 Note and Hints

- You are not required to support the `--r2` option for future labs. We don't aim to create more work for you.
- You might want to adapt the JSON parsing helper code from lab1 checkpoint to parse input files of this checkpoint.
- We highly recommend implementing dataflow analysis using a control flow graph and basic blocks. Using basic blocks make dataflow analysis run much faster, which might be necessary to pass a few test cases in this checkpoint. Moreover, building a control flow graph will be necessary for SSA and other optimizations in future labs.
- It would be helpful to have an entry and exit block when building the control flow graph. The input files might have multiple exit points with no successors, so you would need to manually create an exit block and connect all exit points to it. Also remember to not include this newly created exit block in your output.
- In lab 2, to implement liveness analysis using the dataflow analysis framework you built for this checkpoint, you will probably need to generate the `gen` and `kill` sets from abstract assembly and feed them into the dataflow analysis framework, then read the `In` and/or `Out` sets at each program point to generate an interference graph like you did for lab1 checkpoint.
- Another great example of using dataflow analysis is partial redundancy elimination (PRE). One classic four-pass algorithm of performing PRE involves a `backwards-must` pass to find anticipated expressions, a `forwards-must` pass to find available expressions, another `forwards-must` pass to find postponable expressions, and finally a `backwards-may` pass to find used expressions.