15-411 Compiler Design, Lab 1 Checkpoint

15-411 staff

1 Introduction

Doing register allocation correctly and efficiently is a non-trivial and essential part of developing your L1 compiler. This checkpoint is designed to get you off to a good start. For the checkpoint we provide you with input files containing liveness information generated from L1 test files, as well as a verifier that checks whether your register allocation is correct and scores the result based on how many registers are used. The purpose of this checkpoint is to help you start working on the back-end early and help test the correctness and quality of your register allocation. Note that you should read the lab1 handout before reading this handout.

Your task is to extend the starter code of your compiler to parse input files containing liveness information, perform register allocation based on that information, and finally to output your register allocation using the output format specified below.

2 Input and Output Format

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

```
//target 8
[
 ...
 {
  "Uses": [ "%t9", "%t10" ],
  "Defines": [ "%t11" ],
  "Live_out": [ "%t11" ],
  "Move": false,
  "Line": 30,
 },
 {
  "Uses": [ "%t11" ],
  "Defines": [ "%eax" ],
  "Live_out": [],
  "Move": true,
  "Line": 31,
 },
 ...
]
```

The first line of the input file is a directive `//target k`, meaning that your compiler should use at most k registers in register allocation to receive points for this input. The rest of the input file is a JSON list of JSON objects. The ith object in the list represents the liveness information of the ith abstract assembly line. For example, the two objects above correspond to the abstract assembly lines below:

```
%t11 <-- %t9 * %t10
%eax <--%t11
```

Here is a breakdown of what the fields in each object means:

- **Uses:** Array of strings. Denotes the temps used on this line. Note that all temps used must have been defined some point earlier in the abstract assembly.
- **Defines:** Array of strings. Denotes the temp or register defined on this line. Note that there can be at most one temp or register defined on a line.
- **Live_out:** Array of strings. Denotes live-out temps on this line. See slides for definition of live-out. You should be able to derive the exact same live-out sets from the given used and defined sets.
- **Move:** A boolean, true if the instruction is a move instruction and false otherwise. This field is only relevant if you want to implement register coalescing.
- **Line:** Integer. Denotes the line number of this line within the abstract assembly program. The purpose of this field is to aid you with debugging, since error messages from the verifier contain line numbers.
- **Temps and Registers:** Temps are represented by `%t[1-9][0-9]*`. For this checkpoint, the only registers that are present in the inputs files are `edx` (to support div and mod) and `eax` (to support return). Note that registers only appear in `Defines` and never in `Uses` or `Live_out`.

Output format: The output files generated by your compiler must adhere to the JSON format of the form:

```
[
{ "%t1": "%edx" },
{ "%t2": "%edx" },
{ "%t3": "%eax" },
{},
 ...
]
```

The output is also a JSON list of JSON objects. The ith object in the list must correspond to the ith abstract assembly line. There is at most one temp defined on an abstract assembly line, and the corresponding JSON object in your output should map this defined temp to the register allocated for this temp. If no temp is defined on an abstract assembly line, the corresponding JSON object

should be an empty JSON object, as shown above.

The reason we don't ask you to output a single mapping from each temp to the register allocated for that temp is we wanted to be flexible. If there are multiple definitions of the same temp, some register allocator implementations might map each definition to a different register. That being said, the reference compiler used to generate the input files implements SSA, so all temps have a unique definition within the input files we provide you.

Running the verifier: Your compiler is expected to recognize a flag `-r` which, when present on the command line, tells the compiler to run the L1 checkpoint. When `-r` is present, the source file passed in will be the input file for the checkpoint. The input file generated from the L1 test file `foo.l1` will be named `foo.l1.in`, and is guaranteed to be well-formed. Given the `-r` flag and source file `foo.l1.in`, your compiler should generate a output file called `foo.l1.out` in the same directory as `foo.l1.in` according to the JSON output format specified above. The verifier will then read in both the input and output files to evaluate the quality and correctness of your register allocation.

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/l1-basic-checkpoint`, you can run the command `../runverifier l1-basic-checkpoint` in `dist/compiler`. The `runverifier` script will make your compiler, pass in each input file to your compiler with the `-r` flag to generate an output file, and then pass both the input and output files to the verifier for scoring. The output files generated by your compiler would be placed in `dist/log/l1-basic-checkpoint`.

We provide 3 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 either run the verifier on the docker containers we provide you or use an ubuntu virtual machine.

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

Correctness: The verifier checks 2 conditions in the output:

1. That temps defined on each line in the input file are allocated to some register in the output.
2. That no conflicts occur, i.e., temps that interfere should not assigned to the same register.

3 Note and Hints

- You are not required to support the `-r` flag for future labs. The purpose of this checkpoint is to encourage you to write a good register allocator early which would help you immensely for future labs. We don't aim to create more work for you.
- JSON parsing helper code is available for the following languages - OCaml, C++, and Rust.

- We have set the target number for each input file leniently. You should be able to beat the target if you follow the coloring algorithm from lecture without any extra optimizations.
- The input files are generated from the L1 test files that compile (all L1 test files except those starting with `//test error`). Each input file is named after the L1 test file used to generate it. For example, *return06.l1.in* is generated from the test file *return06.l1*.
- The reference compiler we use to generate input files uses SSA, constant propagation, and copy propagation. These optimizations are not required for lab 1. In fact, we recommend that you do NOT implement these optimizations until you have passed all the L1 tests. Implementing these optimizations can hide a variety of ills that will come back to bite you later on.
- The register names in your output can be any string ("blue", "kitten", ...) as long as all constraints are satisfied. The verifier will count the number of different register strings in your output when scoring. However, we strongly recommend using actual x86 register names for at least the first 16 register names. After the first 16 you can name them whatever you want, but "spillXX" or some such might be useful for understanding what is happening with your code.