Complier Final Project Report

Group: Junyuan Wang, Yubo Cai **Project:** Register Allocation

Remark. words with orange color contain the link to the file.

1 Implementation Overview

We choose the project of **Register Allocation**. For this project, the base compiler is provided by the professor from Moodle. The base compiler can be retrieved from Reference BX compiler. Our code is completely developed under the existing reference compiler.

In this project, we implement all the features except the **Register coalescing** (Bonus part). Our project is just a simple realization of the project guidelines strictly following the algorithm from the lectures and the order of each part.

2 Utilization

The code can be retrieved from our repository on GitHub. A compressed archive will also be delivered. In the archive, the whole compiler is in the folder **bxc**. We also provide test files (including compiled executable **.exe**) as well as example files. For utilizing the compiler, one can call the **bxc.py** in the bxc folder, with the command

```
python bxc.py /path/to/bx/<filename> -t
```

The **-t** parameter is optional. If the parameter is specified, the compiler will create a **<filename>.tac** file containing the TAC representation of the whole program. However, it is **not** a JSON format.

3 Specific Implementation

3.1 SSA Generation and Minimization

For the SSA part, the features are all implemented in the file **bxc/bxlib/bxssa.py**. The first critical part is the *live-in* computation. We implement the algorithm from **Lecture 8 page 18** in function *_liveness_analysis()* by Union operations. The operations with ϕ functions (adding ϕ function definitions, updating, fill the arguments of ϕ) are implemented in *_update_phi_functions()* and *_add_phi_functions()*. The above functions basically covers the main part of **Crude SSA Generation**.

SSA Minimization are separated into 2 cases of **Removing Null Choices** and **Simplifying Renames**. The algorithm comes from **Lecture 8 page 43 and 44** and implemented with function *null_choice_elimination()* and *rename_elimination()*. Also **SSA Destruction** are implemented in *destruct()*.

3.2 Interference Graph

For the Interference Graph part, the features are implemented in the file **bxc/bxlib/bxregalloc.py**. We defined a **InterferenceGraph** class containing the data structure and methods to manipulate the graph. We followed the instructions in **section 3.1 regalloc.pdf** in the function *build()*.

3.3 Max Cardinality Search and Greedy Coloring

For the searching and coloring part, the features are also implemented in the file **bxc/bxlib/bxregalloc.py**. We use the algorithm in **page 5 regalloc.pdf** and realized in *max_cardinality_search()* and *greedy_coloring()* respectively.

3.4 Register Allocation

For the final register record and allocating part, we defined the functions in the file **bxc/bxlib/bxregalloc.py** and modified some parts of the file **bxc/bxlib/bxasmgen.py**.

The way we format the temporaries to register has changed. Also, we modified **div**, **mod**, and **shl/shr** to avoid the collision when we need certain registers for these operations.

4 Problem Encountered

When we tried to build the interference graph, we faced some problems that some interference was missing. When we closely analyzed the process of creating the interference graph, we found that the interference graph created by SSA is not always correct for the interference after SSA destruction.

For further information and a specific example, there is a very detailed explanation of our observation from here. We will try to fix in the future time as soon as possible and submit the new version by email.