CS241 Advanced Compiler Half-way Report

Anbang Xu(35086995), Jianfeng Jia(62487298)

Overview

By the time of this week, we have done the step 1, which is generated the SSA intermediate result and the corresponding CFG graph. The missing part is array implementations. Below is the brief description about our design.

1. A simple recursive-descent parser

This parser is to check the grammar recursively. Each EBNF form is one function. It will check if the input is following the given grammar. If the grammar has some errors, we will throw "SyntaxError" exceptions which contain the line # to warn the user. Without generating the intermediate result, all the test programs are passed the grammar checking.

2. SSA design

The SSA variable in our code is a (varName, version) pair. If the variable is not Array type, we will use the SSA form to store it in the intermediate instructions. The version field will keep updated once the *assignment* is called. And the Phi function will update the later SSA versions.

3. Basic Block

Each Block records an array of the SSA instructions. Besides that, it has its own LocalVariable which records the latest version of the SSA variable within this block. It also has two branch out pointers, one is pointing to the "next" normal Block, anther is pointing to the "negBranch" block, which could be null if the block is not the "if" or "while" statement. All the blocks are connected by these two links. And we just keep the "root" block inside the parser to print the whole graph.

4. CFG data structure

The Basic Block itself is enough to describe the CFG result. However, it need to loop through the whole link list to get the tail of the subgraph. Then we defined a CFG structure to record the "head" and "tail" of one subgraph. All the "statement" function will return the CFG result directly. And it will be easier to connect the different statements.

5. Intermediate code generator and Phi function manager

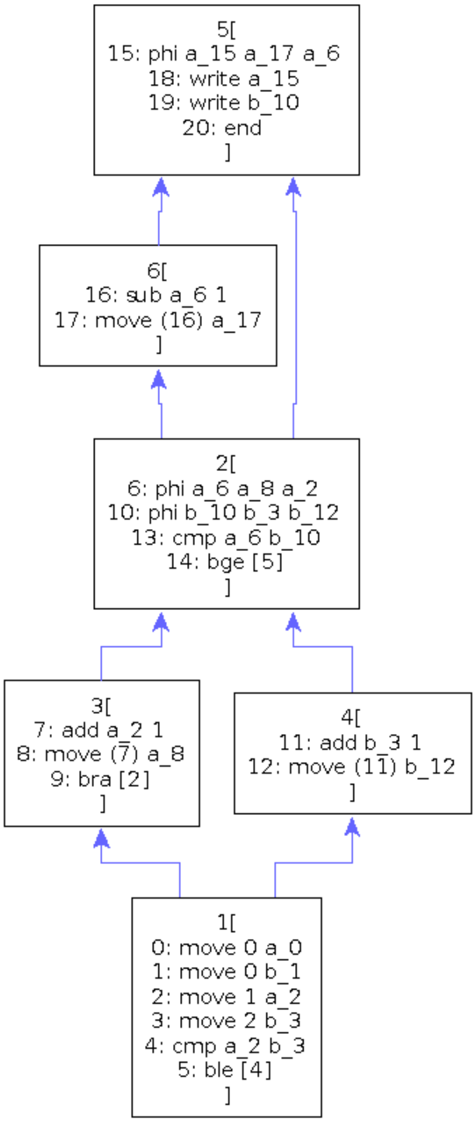
The intermediate code generator converts every single operation to the intermediate instruction and stores in the corresponding Basic Block of CFG. The phi function manager is used to generate a phi function, update a phi function and change the the corresponding variable references.

6. Def-use-chain

We build the data dependence information(def-use-chain) for all constants and variables. This is currently used to update all corresponding values to be results of phi functions.

7. VCG result

Here is two VCG graph generated by our program. The visualization tool is using:(http://pp.ipd.kit.edu/firm/yComp). The graph is bottom-up view: the left is for if statement for test007.txt , the right one is generated by while statement for test008.txt(The two red points is connected together, the layout of that VCG tool is not so good).



```
5[
15: phi a_15 a_17 a_6
18: write a_15
19: write b_10
20: end
]
```

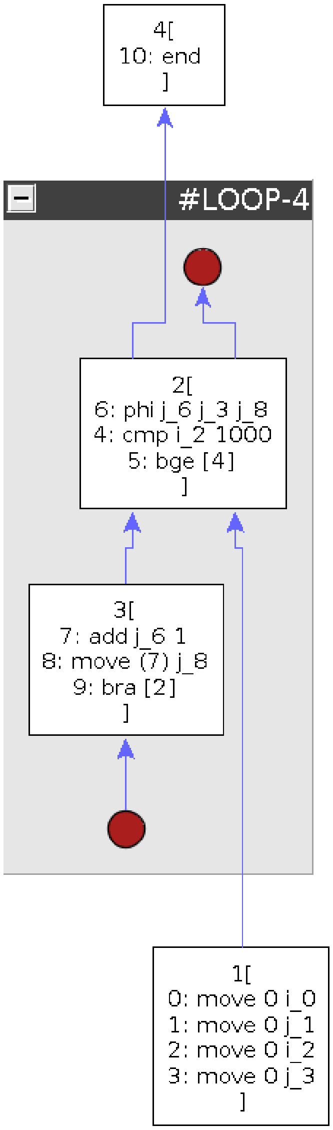
```
6[
16: sub a_6 1
17: move (16) a_17
]
```

```
2[
6: phi a_6 a_8 a_2
10: phi b_10 b_3 b_12
13: cmp a_6 b_10
14: bge [5]
]
```

```
3[
7: add a_2 1
8: move (7) a_8
9: bra [2]
]
```

```
4[
11: add b_3 1
12: move (11) b_12
]
```

```
1[
0: move 0 a_0
1: move 0 b_1
2: move 1 a_2
3: move 2 b_3
4: cmp a_2 b_3
5: ble [4]
]
```



```
4[
10: end
]
```

```
2[
6: phi j_6 j_3 j_8
4: cmp i_2 1000
5: bge [4]
]
```

```
3[
7: add j_6 1
8: move (7) j_8
9: bra [2]
]
```

```
1[
0: move 0 i_0
1: move 0 j_1
2: move 0 i_2
3: move 0 j_3
]
```

TODO:

1. Make sure SSA works correctly;
2. Generate and Visualize the dominator tree;
3. Optimization
4. Global Register Allocation