

[CS201]Project 2 Liveness Analysis

Qun Lou 862325101

Result

```
initialized IN[] if.end c e i6
entry IN : a b c i i2 i4 i6
entry OUT : a b c

if.then IN : a c i6
if.then OUT : c e

if.else IN : b c i4 i6
if.else OUT : c e

if.end IN : c e i6
if.end OUT :

Liveness Analysis Pass: main
initialized IN[] entry
entry IN :
```

Output: The analysis result is printed in the **standard error stream** using the format (basic-block name: variable names). First, it printed the function name 'test', then the live variables reaching the end of each basic block in every single line. The basic blocks are generated by clang and we assume each basic block in each test case will have a unique name. For example, the result of test2.ll:

```
--test2--
entry: a b c
if.then: c e
if.else: c e
if.end:
```

Note:

To compile the LLVM pass, under the 'Pass' directory:

```
mkdir build
```

to get the correct result, we should add "-fno-discard-value-names" flag when generating .ll file.

Usage:

```
opt -instnamer -enable-new-pm=0 -load ./libLLVMLivenessAnalysisPass.so -LivenessAnalysis
./test.ll
```

Design

As we learned at class, there are 2 main steps:

1. Initialize

```

IN[B]: Variables live at B's entry.
OUT[B]: Variables live at B's exit.
GEN[B]: Variables that are used in B prior to their definition in B.
KILL[B]: Variables definitely assigned value in B before any use of that variable in B.

-- Initialize sets:
for every block B
    IN[B] = GEN[B]  //(GEN[B]: Variables that are used in B prior to their definition
in B.
    OUT[B] =  $\emptyset$ 

```

2. Iterate

```

-- Iteratively solve equations:
change = true;
while change {
    change = false;
    for each B  $\neq$  Be {
        OLDIN = IN[B]
        OUT[B] =  $\bigcup_{s \in \text{succ}(B)} \text{U}(\text{IN}[S])$ 
        IN[B] = GEN[B]  $\cup$  (OUT[B] - KILL[B])
        if IN[B]  $\neq$  OLDIN then change = true
    }
}

```

Implement

For step 1, we will focus on these types of operations:

```

if(inst.getOpcode() == Instruction::Add || inst.getOpcode() == Instruction::Sub
|| inst.getOpcode() == Instruction::Mul || inst.getOpcode() == Instruction::SDiv
|| inst.getOpcode() == Instruction::PHI || inst.getOpcode() == Instruction::ICmp
|| inst.getOpcode() == Instruction::Load || inst.getOpcode() == Instruction::Store)

```

The only unique one is store whose second operands will be added to "kill" set. Otherwise, the operands will be added to "gen set".

For step2, we can use "rbegin()" to simulate the "post propagation"

```

bool change = true;
while(change){
    change = false;

```

```

for(auto it = bbs.rbegin(); it!=bbs.rend();it++){
    // ....
    //      OUT[B] =      s  $\in$  succ(B)  U(IN[S])
    //      IN[B] = GEN[B] U (OUT[B] - KILL[B])
    for(const BasicBlock *succ : llvm::successors(&bb)){
        auto succ_out = OUT[succ];
        auto succ_in = IN[succ];
        std::set_union(succ_in.begin(), succ_in.end(),
                       new_out.begin(), new_out.end(), std::inserter(new_out,
new_out.begin()));
        }// get OUT[B]
        auto out_diff_kill = std::set<const StringRef>();
        std::set_difference(new_out.begin(),new_out.end(),
                           kill.begin(),kill.end(),std::inserter(out_diff_kill,
out_diff_kill.begin()));
        std::set_union(gen.begin(),gen.end(),
                       out_diff_kill.begin(),out_diff_kill.end(),std::inserter(new_in,
new_in.begin()));
        if(new_in != old_in){
            change = true;
        }
    }
}

```