Tutorial 9: LLVM IR Optimization API

Assignment 6

- Memory to Register Promotion (`alloca2reg`) (11 marks out of 16)
- Open-ended IR optimization design (5 marks out of 16)

Agenda

- Assignment 6 Part 1 Description
- Data Structure & LLVM IR Optimization API
- General Steps to Implement Part 1

Assignment 6 Description

Perform a function pass: `alloca2reg` for Minic. You will work on the `src/Alloca2Reg.cpp` source file. Command `opt` will do a function pass based on `Alloca2Reg` library object on the IR bitcode. Please review **"Safely Promote Local Variable to Registers"** section from the lecture (under Files/Slides/Optimizations.pdf).

```
# After generating your bitstream,
# Run Alloca2Reg pass
opt -O0 -load src/liballoca2reg.so --alloca2reg output.bc -o output_opt.bc
# Generate IR assembly code (Important for debugging!)
llvm-dis output.bc -o output.ll
llvm-dis output_opt.bc -o output_opt.ll
```

- Writing an LLVM Pass https://llvm.org/docs/WritingAnLLVMPass.html
- LLVM Language Reference Manual: https://llvm.org/docs/LangRef.html
- llvm Namespace Reference: https://llvm.org/doxygen/namespacellvm.html

LLVM IR Optimization Useful APIs in Assignment 6 (Copied from a6 handout)

Name	Description
struct Alloca2RegPass	Derived struct from llvm::FunctionPass.
Alloca2RegPass::ID = 0	Declares pass identifier used by LLVM to identify pass. This allows LLVM to avoid using expensive C++ runtime information. LLVM uses ID's address to identify a pass, so the initialization value is not important.
Alloca2RegPass::TargetAllocas & collectTargetAllocas()	Helps to collect all of the alloca instructions which can be removed. Please look at step 1 from the lecture.
Alloca2RegPass::Post & Pre	As mentioned in lecture, in step 2, they hold the representative value of the variable at the end of the basic block BB.
Alloca2RegPass::runOnFunction()	Overrides an abstract virtual method inherited from llvm::FunctionPass.
Other variables	To run an LLVM Pass automatically with clang. You don't need to worry about it. You could check out the link here for more details.

Table 1: Data Structure Description

Name under namespace llvm	Description
AllocaInst, StoreInst, LoadInst	alloca, store, load instructions
PointerType* AllocaInst::getType()	Overload to return most specific pointer type.
Type* PointerType::getElementType()	Return the type of the element which PointerType points to.
bool Type::isIntegerTy()	Return true if the type is integer type.
Value* StoreInst::, LoadInst::getPointerOperand()	Get the pointer operand.
Value* StoreInst::getValueOperand()	Get the value operand.
Instruction::eraseFromParent()	This method unlinks 'this' from the containing basic block and deletes it.
BasicBlock::begin(), end()	Instruction iterator methods
Function::begin(), end()	Basic Block iterator methods
PHINode member functions	You can search it for your implementation.

Table 2: LLVM API Description

General Steps to Implement Part 1

1. Detect all alloca inst. Note we will not touch array and load/store non-pointer operands.
2. Iterate all BBs.
   (a) In the same block, as for the variable using alloca inst, record its value of store inst and replace the following load inst's value. (Do some research how to replace all of the uses)
   (b) If no store inst, create PHI node, replace the value of load inst with the PHI node and record the PHI in `pre`. However, it will be `UndefValue` if it is entry block of a function.
   (c) In the meantime, record its last value into `post`. Note that the recorded value is either stored value or PHI node value.
3. Fill and complete all incoming edges of PHI nodes in `pre` you created in the last step by reading `post`.
4. Remove unused PHI, all corresponding load, store, allca inst. PHI nodes will perform read/write functionality.