CS 502 Fall 2022
Compiler Project 4 (Due October 28, 11:59pm)

This project, P4, should be submitted on data.cs.purdue.edu (or Data in short), using the turnin command described in P0 handout. You can develop your code on Data inside the **uscc** directory you created for P0, or you can develop your code on your own computer before re-uploading **uscc** (that contains your work) to Data. In both cases, make sure your links to LLVM tool are set up exactly as specified in P0.

Go to the parent directory of **uscc** on Data and type the following command to submit P4:

turnin –c cs502 –p p4 uscc

The exact command for P4 shown above should be hand typed, instead of copy-and-pasted, to avoid hidden and unwanted characters.

Make sure you **write an outline** to explain how you implement P4 in a PDF file, named **reportP4.pdf**, and place the file in the existing subdirectory named '**report**' under **uscc.**

Before submitting the **uscc** directory, make sure you go in **uscc** and run '**make clean**' to remove all binary codes. (We save your submissions through the entire semester, and we do not want to store huge files that may break the storage system.) **Failing to clean up your submitted directory might incur a penalty on the grade of the project,** depending on the severity of the nonconformance. Make sure you do not compress your **uscc** directory yourself before submission

The appendix to this handout is the detailed description (named "Programming Assignment 4) of the tasks in P4. The main ideas are already presented in the lectures and students are expected to explore the provided code skeleton and learn how to complete the parser as required.

USCC
Programming Assignment 4 – Static Single Assignment

Useful Links

In addition to the standard LLVM documentation, you will need to consult "Simple and Efficient Construction of Static Single Assignment Form" (Braun et al.), as it contains extensive pseudocode for the algorithm you will implement in this programming assignment.

Introduction

In LLVM, virtual registers must use *SSA form* – meaning that they can only be assigned once. To work around this restriction, the LLVM bitcode also supports a stack frame via the `alloca`, `load`, and `store` instructions. This was the approach taken in PA3, and is the same approach taken by Clang when it first generates LLVM IR. Clang then later converts the IR to SSA form using the *mem2reg* pass, which implements the canonical Cytron algorithm to generate SSA form.

For an example of the bitcode that is currently generated, run the following command in the tests directory:

```
$ ../bin/uscc -p ssa01.usc
```

The output for the main function should be along the lines of:

```
define i32 @main() {
entry:
  %y = alloca i32
  %x = alloca i32
  store i32 5, i32* %x
  store i32 6, i32* %y
  store i32 7, i32* %x
  store i32 10, i32* %x
  %x1 = load i32* %x
  %tobool = icmp ne i32 %x1, 0
  br i1 %tobool, label %if.then, label %if.else
if.then:                                          ; preds = %entry
  store i32 20, i32* %y
  br label %if.end
if.else:                                          ; preds = %entry
  store i32 15, i32* %y
  br label %if.end
if.end:                                           ; preds = %if.else, %if.then
  %y2 = load i32* %y
  %0 = call i32 (i8*, ...)* @printf(...)
  ret i32 0
}
```

Notice how `%x` and `%y` are pointing to allocations on the stack, and thus `store` and `load` instructions are used to write to these variables. One major issue with using the stack to store these variables is it greatly hampers optimizations – stack variables cannot be aggressively optimized, whereas virtual registers can. Therefore it is important to ensure the code is in SSA form before applying any optimizations. Furthermore, the above code has multiple useless assignments – for example, the address representing `%x` is written to three times in the entry block, but only the last write will actually see any use.

In this programming assignment, you will implement a fairly new approach to generating SSA form as outlined in the Braun paper. Braun's algorithm can directly generate SSA form from the high-level AST. This means there is no need to generate non-SSA form first. This will eliminate all use of the stack for variables (other than arrays). Conveniently, the Braun algorithm also will eliminate most useless assignments.

Braun's algorithm is simpler to implement than the Cytron algorithm because it does not require generating dominator trees. Furthermore, because USCC does not support arbitrary jumps in control flow such as `break`, `continue`, or `goto`, it means that the USCC implementation of Braun's algorithm can generate minimal SSA without the use of any of the additional passes described in Section 3 of the paper.

Most of the code for this assignment will be written in `opt/SSABuilder.cpp`. If you open `opt/SSABuilder.h`, you will see that the majority of the member functions in the `SSABuilder` class correspond to the functions discussed in the Braun paper. The member variables are two hash maps and one hash set. Both hash maps and the hash set are keyed on the pointer to a specific basic block. The `mVarDefs` map associates a basic block with the variable definitions in that basic block, while `mIncompletePhis` tracks the Phi nodes within a basic block that need to be completed at some point in the future. Finally, the `mSealedBlocks` set is used to track which blocks are *sealed* – meaning all predecessor blocks have been connected.

Helper Functions

There are two helper functions you should first implement in `opt/SSABuilder.cpp`. These functions do not correspond to functions outlined in the Braun SSA paper.

reset

The reset function should clear all of the data in the maps and set member variables. This is called every time a new function is emitted, since the SSA data is local to a specific function. Keep in mind that since both `mVarDefs` and `mIncompletePhis` have maps as values, these nested maps need to be deleted before clearing the containing map.

addBlock

The `addBlock` function is called every time a new basic block is added to the IR. This function should add an entry for the basic block to both `mVarDefs` and `mIncompletePhis`. Furthermore, the second parameter of `addBlock` states whether or not `addBlock` should immediately seal the block (by calling `sealBlock`) on it.

Local Value Numbering

You will want to implement local value numbering as outlined in Section 2.1 of the Braun paper. The two functions used for local value numbering are `writeVariable` and `readVariable`, and mostly involve accessing the appropriate maps.

Global Value Numbering

Next, you must implement global value numbering as outlined in Section 2.2 of the Braun paper. This represents the bulk of the work for this programming assignment. The three functions to implement are `readVariableRecursive`, `addPhiOperands`, and `tryRemoveTrivialPhi`.

Tips for readVariableRecursive

- When you create a Phi node, it must be added to the *beginning* of the basic block. This is a requirement enforced by LLVM. Thus, you cannot use the `IRBuilder` as used in PA3 to create these Phi nodes (as the `IRBuilder` always adds to the end of the basic block). Instead, you can use the `PhiNode::Create` factory method.
- The `BasicBlock` member function `getSinglePredecessor` can be used to determine whether a block has only one predecessor, and if so, the pointer to said block.

Tips for addPhiOperands

- You can iterate over the predecessors of a `BasicBlock` by getting a `pred_iterator` using `pred_begin` and `pred_end`.
- To add operands to a `PhiNode`, use the `addIncoming` member function.

Tips for tryRemoveTrivialPhi

- To get the number of operands attached to a `PhiNode`, use the `getNumIncomingValues` function. To access the value of a specific operand use `getIncomingValue(int)`.
- You can get an undefined value using the `UndefValue::get` static method
- You can iterate over the users of a node by getting the `use_iterator` via `use_begin` and `use_end`
- To replace a Phi node with the "same" value, you need to do two things. First use the `replaceAllUsesWith` member function. Second, you must update the variable definition map to use "same." This second step is not immediately apparent from the Braun paper, but without doing this you still reference Phi nodes that have been removed.
- Use `eraseFromParent` to delete the Phi node once it is replaced.

Sealing Blocks

The final function to implement in `opt/SSABuilder.cpp` is the `sealBlock` function as outlined in Section 2.3 of the Braun paper. This function is relatively straightforward.

Once you finish the implementation of `sealBlock`, the next step is to actually hook up the `SSABuilder` functions so they are used.

Integrating SSABuilder

There are a few different places in `parse/Symbols.cpp` and `parse/ASTEmit.cpp` that need to be edited in order to integrate the `SSABuilder` class. This section outlines what must be modified.

You may have noticed before that the `CodeContext` that's passed around everywhere during emission of the LLVM IR contains a member variable called `mSSA`. This is the instance of the `SSABuilder` that will be used throughout the code to access the SSA functionality.

Reading/Writing to Identifiers

The implementation of `Identifier::readFrom` and `Identifier::writeTo` in `parse/Symbols.cpp` can be greatly simplified now that `SSABuilder` is implemented. Their contents should be replaced with calls to `readVariable` and `writeVariable`, respectively – there is no need to have the separate checks for arrays in these functions anymore.

Initializing Variables

Previously, all local variables had their stack space allocated in `ScopeTable::emitIR`. However, with the SSA implementation, only arrays need to be allocated. This means you can eliminate the else case in this function that allocates regular variables.

One other change needs to be made for function arguments in the `ASTFunction` emission code in parse/ASTEmit.cpp. Specifically, the loop that iterates through all of the function arguments has a call to `setAddress` that needs to be replaced with a call to `writeTo`. This is noted in comments within the function.

Adding/Sealing Blocks

The last step to integrate `SSABuilder` is to ensure that whenever a basic block is created, it is added to the `SSABuilder` instance via `addBlock`. Then when a block will have no further predecessors added to it, it must be sealed via `sealBlock`. For convenience, it's also possible to add a block that's already sealed via the second parameter of the `addBlock` function.

There were only two nodes you wrote emission code for in PA3 that create basic blocks – `ASTIFStmt` and `ASTWhileStmt`. So these nodes will need to be updated to inform `SSABuilder` about the blocks. There are three other nodes that create blocks – `ASTFunction`, `ASTLogicalAnd`, and `ASTLogicalOr`. But the code for these nodes was provided for you, and so it already has the necessary calls to `addBlock` and `sealBlock`.

Testing Your Implementation

Once you've implemented all the functionality as outlined, you will want to make sure that the testEmit.py test suite still works. To execute this suite, you run the following command from the tests directory:

```
$ python testEmit.py
```

However, the test suite only checks whether the emitted code is functional. You will also want to inspect the output of a variety of test cases and ensure that a minimal number of Phi nodes is generated. For example, the `ssa01.usc` test case discussed in the introduction should have a single Phi node in the `%if.end` block.

For a more complex example, take a look at the output when you run the following command:

```
$ ../bin/uscc -p quicksort.usc
```

The `partition` function in the generated code should have three Phi nodes: two in `%while.cond` and one in `%if.end`. Notice that there still are several `load` and `store` calls – this is in order to access indices in the array. But there should be no `load` or `store` calls outside of these array accesses.

Conclusion

You now have a working implementation of static single assignment form. Since the vast majority of variables are now stored in virtual registers, this means that you can now implement a wide range of optimizations to the code. In the subsequent programming assignment, you will write a handful of optimization passes to that will be ran on the SSA code.