CENG513 Compiler Design and Construction
LLVM Passes

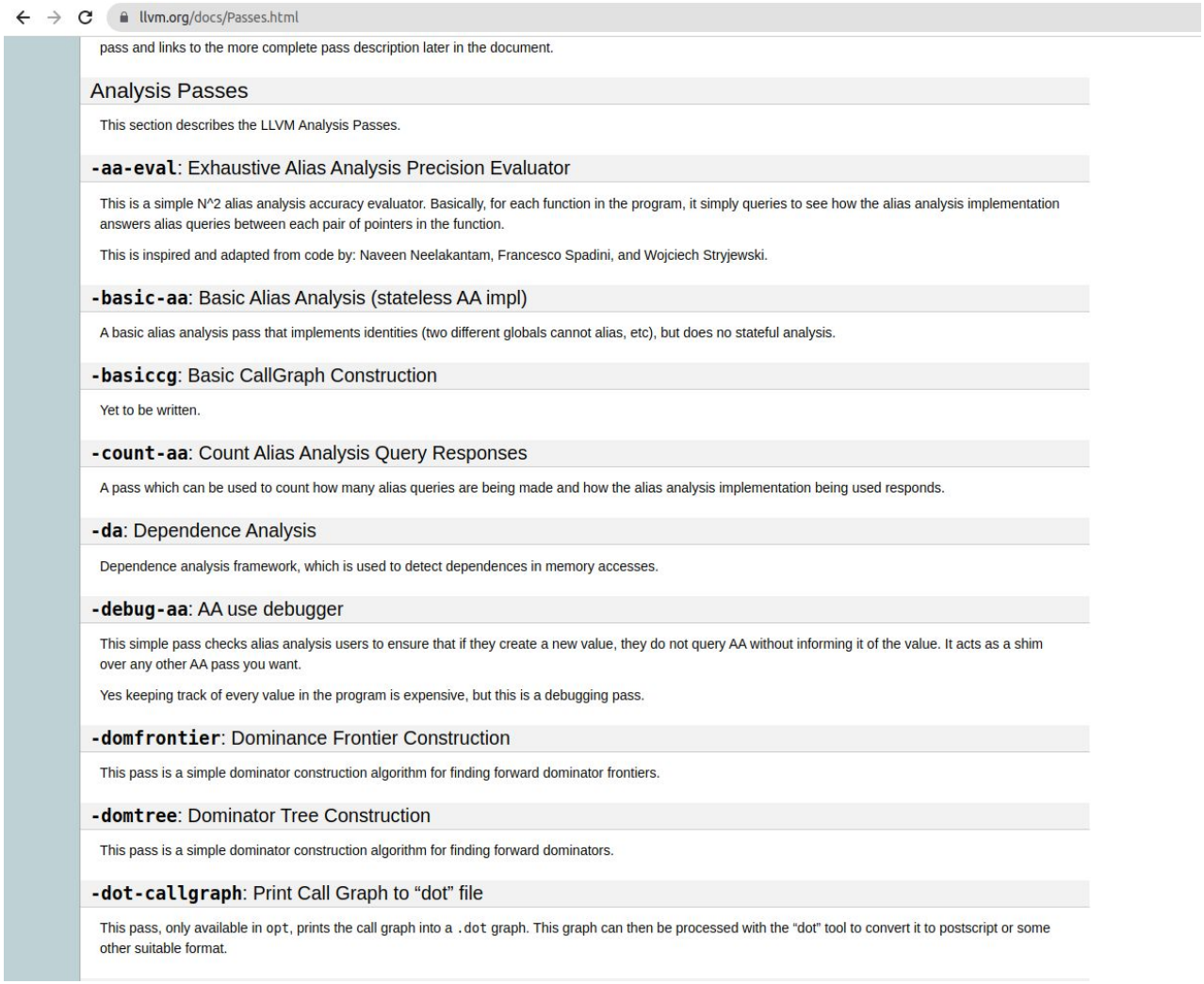
Note by Işıl ÖZ:
These slides are prepared based on LLVM Tutorial Writing an LLVM Pass: 101 given at LLVM Developers' Meeting, 2019.

LLVM Passes

Optimizations are implemented as Passes that traverse some portion of a program to either collect information or transform the program

- Analysis passes
  - Compute information that other passes can use or for debugging or program visualization purposes
  - -da: Dependence Analysis
- Transform passes
  - Mutate the program in some way
  - -licm: Loop Invariant Code Motion
- Utility passes
  - Provide some utility but don't otherwise fit categorization
  - -view-cfg: View CFG of function

LLVM Passes


llvm.org/docs/Passes.html
pass and links to the more complete pass description later in the document.
Analysis Passes
This section describes the LLVM Analysis Passes.
-aa-eval: Exhaustive Alias Analysis Precision Evaluator
This is a simple N^2 alias analysis accuracy evaluator. Basically, for each function in the program, it simply queries to see how the alias analysis implementation answers alias queries between each pair of pointers in the function.
This is inspired and adapted from code by: Naveen Neelakantam, Francesco Spadini, and Wojciech Stryjewski.
-basic-aa: Basic Alias Analysis (stateless AA impl)
A basic alias analysis pass that implements identities (two different globals cannot alias, etc), but does no stateful analysis.
-basiccg: Basic CallGraph Construction
Yet to be written.
-count-aa: Count Alias Analysis Query Responses
A pass which can be used to count how many alias queries are being made and how the alias analysis implementation being used responds.
-da: Dependence Analysis
Dependence analysis framework, which is used to detect dependences in memory accesses.
-debug-aa: AA use debugger
This simple pass checks alias analysis users to ensure that if they create a new value, they do not query AA without informing it of the value. It acts as a shim over any other AA pass you want.
Yes keeping track of every value in the program is expensive, but this is a debugging pass.
-domfrontier: Dominance Frontier Construction
This pass is a simple dominator construction algorithm for finding forward dominator frontiers.
-domtree: Dominator Tree Construction
This pass is a simple dominator construction algorithm for finding forward dominators.
-dot-callgraph: Print Call Graph to "dot" file
This pass, only available in opt, prints the call graph into a .dot graph. This graph can then be processed with the "dot" tool to convert it to postscript or some other suitable format.

-licm: Loop Invariant Code Motion

$ clang -emit-llvm -S main.c -o main.ll

$ opt -passes=licm main.ll -S -o main2.ll

llvm/include/llvm/Transforms/Scalar/LICM.h
llvm/lib/Transforms/Scalar/LICM.cpp

```
PreservedAnalyses LICMPass::run(Loop &L, LoopAnalysisManager &AM,
                                LoopStandardAnalysisResults &AR, LPMUpdater &) {
  …

LoopInvariantCodeMotion LICM(Opts.MssaOptCap, Opts.MssaNoAccForPromotionCap,
                             Opts.AllowSpeculation);
  …
if (!Changed)
    return PreservedAnalyses::all();

  auto PA = getLoopPassPreservedAnalyses();

  PA.preserve<DominatorTreeAnalysis>();
  PA.preserve<LoopAnalysis>();
  PA.preserve<MemorySSAAnalysis>();

  return PA;
}
```

LLVM Pass

- A pass operates on some unit of IR (e.g. Loop or Function)
  - Transformation pass will modify it
  - Analysis pass will generate some high-level information
- Analysis pass results are cached
  - A pass needs to request the results of the analysis pass(es) first
- Transformation pass records what's preserved

HelloWorld Pass

llvm-tutor:

```
// Main functionality provided by this pass
void visitor(Function &F) {
  errs() << "Visiting: ";
  errs() << F.getName() << " (takes ";
  errs() << F.arg_size() << " args)\n";
}

struct HelloWorld : llvm::PassInfoMixin<HelloWorld> {
  // Main entry point, takes IR unit to run the
  // pass on (&F) and the corresponding pass
  // manager (to be queried/modified if need be)
  llvm::PreservedAnalyses run(
                        Function &F,
                        FunctionAnalysisManager &) {
    visitor(F);

    // all() is a static method in PreservedAnalyses
    return llvm::PreservedAnalyses::all();
  }
};
```

HelloWorld.cpp

LLVM:

```
template <typename DerivedT> struct PassInfoMixin {
  static StringRef name() {
    // (...)
  }
};

template <typename IRUnitT,
          typename AnalysisManagerT = AnalysisManager<IRUnitT>,
          typename... ExtraArgTs>
class PassManager : public PassInfoMixin<
    PassManager<IRUnitT, AnalysisManagerT, ExtraArgTs...>> {

    PreservedAnalyses run(IRUnitT &IR, AnalysisManagerT &AM,
      ExtraArgTs... ExtraArgs) {
      // Passes is a vector of PassModel<> : PassConcept
      for (unsigned Idx = 0, Size = Passes.size(); Idx != Size; ++Idx) {
        PreservedAnalyses PassPA = P->run(IR, AM, ExtraArgs...);

        AM.invalidate(IR, PassPA);
      }
  } // end of run
} // end of PassManager
```

llvm/include/llvm/IR/PassManager.h

Transformation Pass - MBASub

a - b == (a + ~b) + 1

It replaces all instances of integer sub according to the above formula

Leverages *IRBuilder* and *ReplaceInstWithInst*

Transformation Pass - MBASub

```
PreservedAnalyses MBASub::run(llvm::Function &F,
                              llvm::FunctionAnalysisManager &) {
 bool Changed = false;

 for (auto &BB : F) {
   Changed |= runOnBasicBlock(BB);
 }
 return (Changed ? llvm::PreservedAnalyses::none()
                 : llvm::PreservedAnalyses::all());
}
```

Transformation Pass - MBASub

```
bool MBASub::runOnBasicBlock(BasicBlock &BB) {
 bool Changed = false;

 // Loop over all instructions in the block.
 for (auto Inst = BB.begin(), IE = BB.end(); Inst != IE; ++Inst) {

   // Skip non-binary (e.g. unary or compare) instruction.
   auto *BinOp = dyn_cast<BinaryOperator>(Inst);
   if (!BinOp)
     continue;

   /// Skip instructions other than integer sub.
   unsigned Opcode = BinOp->getOpcode();
   if (Opcode != Instruction::Sub || !BinOp->getType()->isIntegerTy())
     continue;

   // A uniform API for creating instructions and inserting them into basic blocks.
   IRBuilder<> Builder(BinOp);

   // Create an instruction representing (a + ~b) + 1
   Instruction *NewValue = BinaryOperator::CreateAdd(
       Builder.CreateAdd(BinOp->getOperand( 0),
                         Builder.CreateNot(BinOp->getOperand(  1))),
       ConstantInt::get(BinOp->getType(),  1));

   // The following is visible only if you pass -debug on the command line
   // *and* you have an assert build.
   LLVM_DEBUG(dbgs() << *BinOp <<  " -> " << *NewValue <<  "\n");

   // Replace `(a - b)` (original instructions) with `(a + ~b) + 1`
   ReplaceInstWithInst(&BB, Inst, NewValue);
   Changed = true;

   // Update the statistics
   ++SubstCount;
 }
 return Changed;
}
```

Build and Run MBASub Pass

```
$ cd <llvm-tutor-dir>
$ mkdir build
$ cd build
$ cmake -DLT_LLVM_INSTALL_DIR=<llvm-build-dir> ../.
$ make
>Building CXX object lib/CMakeFiles/MBASub.dir/MBASub.cpp.o
  Building CXX object lib/CMakeFiles/MBASub.dir/Ratio.cpp.o
  Linking CXX shared library libMBASub.so
  Built target MBASub
$ cd <llvm-tutor-dir>
$ clang -emit-llvm -S inputs/input_for_mba_sub.c -o mba_sub.ll
$ opt -load-pass-plugin <llvm-tutor-build-dir>/lib/libMBASub.so -passes=mba-sub -S
mba_sub.ll -o out.ll
```

```
%20 = sub i32 %6, %19 →      %20 = xor i32 %19, -1
                             %21 = add i32 %6, %20
                             %22 = add i32 %21, 1
```

References

LLVM Passes

- https://llvm.org/docs/Passes.html

Writing an LLVM Pass

- https://llvm.org/docs/WritingAnLLVMNewPMPass.html

LLVM Pass tutorial

- https://github.com/banach-space/llvm-tutor
- https://www.youtube.com/watch?v=ar7cJl2aBuU