CSCD70 Compiler Optimization

Tutorial #2 Introduction to LLVM (ii)

Bojian Zheng
bojian@cs.toronto.edu

Department of Computer Science, University of Toronto

Acknowledgement: Thanks to Professor Gennady Pekhimenko, Professor Nandita Vijaykumar and students from previous offerings of CSCD70.

Review

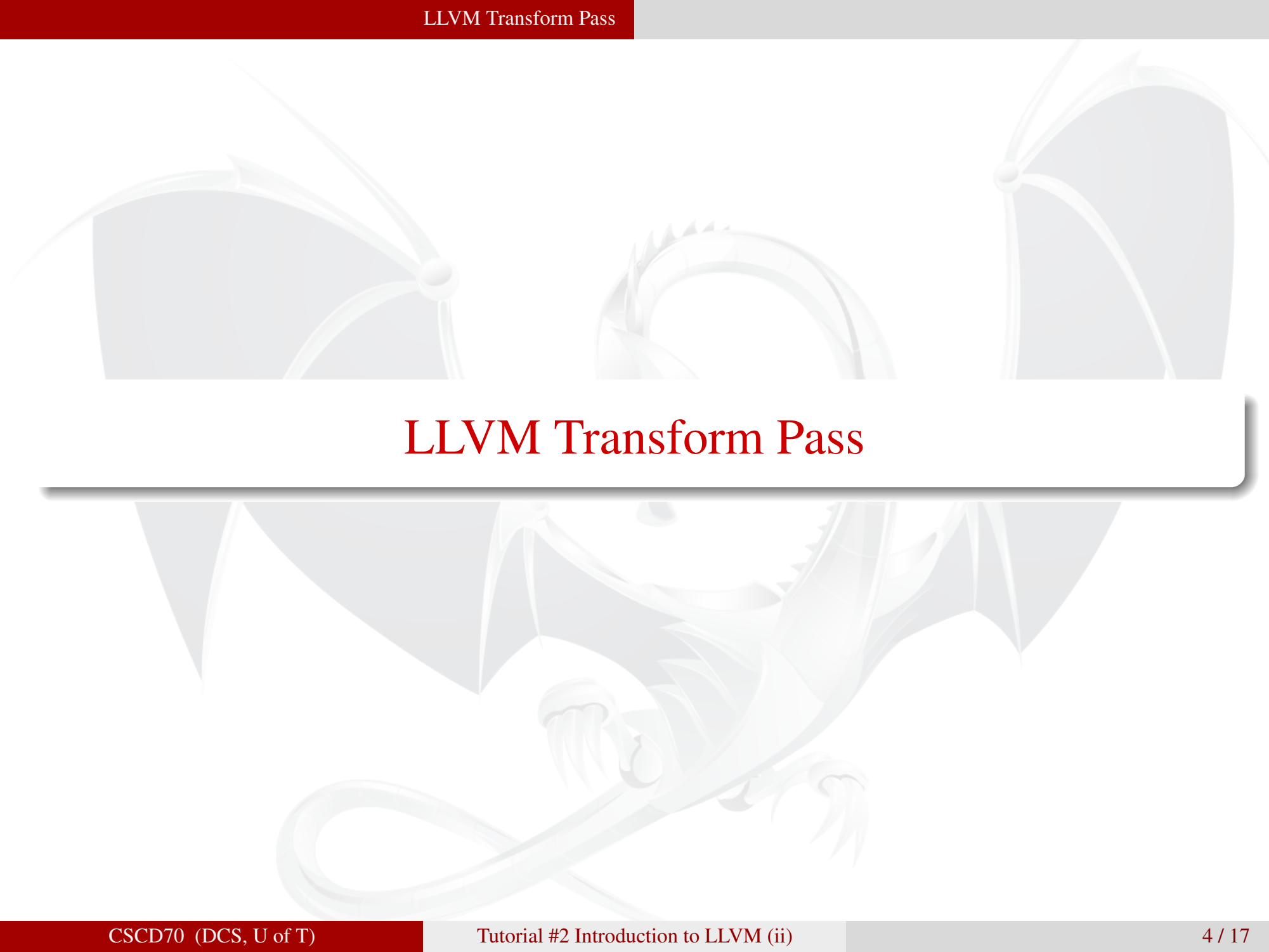
In the previous tutorial, we have discussed the following topics:

- C++ (P?)Review
  - Pass by Reference
  - Public Inheritance
  - Standard Template Library (STL)
- How to write an LLVM Analysis Pass?
  - Intermediate Representation (IR) and Optimization Passes
  - Analysis vs. Transform
  - LLVM Module, Iterators, Downcasting, LLVM Pass Interfaces

Abstract

In this tutorial, we will be further discussing the following topics:

- C++ (P?)Review
- How to write an LLVM Analysis pass?
- How to write an LLVM Transform pass?
- How to connect between passes?



LLVM Transform Pass

Basic Instruction Manipulation

- There are many ways of manipulating instructions: `Instruction`, `BasicBlock`, `BasicBlockUtils`

☞ `Example1-Transform_Pass_Sample`

User-Use-Value

Suppose that we have the following code to optimize:

```
%2 = add %1, 0 ; Algebraic Identity
%3 = mul %2, 2
```

- Program **crashes** because **we did not update the references properly**.
- How to make sure that all references (i.e. **uses**) are updated properly?
  $\Rightarrow$ LLVM **User-Use-Value** Relationship

User-Use-Value

LLVM Class Hierarchy

Goal

We are going to show how the `Instruction` plays the role *both* as an **User** *and* as an **Usee** (`Value`).

`Value`

- The `Value` class is the most important *base* class in LLVM, as almost all object types inherit from it.
- A `Value` has a *type* (e.g., integer, floating point): `getType`.
- A `Value` might or might not have a *name*: `hasName`, `getName`.
- Most importantly, a `Value` has a list of **Users** that are using itself.

Instruction as an User

An `Instruction` is an `User`.

- Each `User` (`Instruction`) has a list of values it is using. Those values are known as the **Operands**, of type `Value`.

```
User &Inst = ...
for (auto Iter  = Inst.op_begin();
          Iter != Inst.op_end(); ++Iter)
{ Value *Operand = *Iter; }
// %2 = add %1, 0  →  operand %1 & 0
```

Instruction as an Usee

- Why is an `Instruction` is an **Usee**?
- The answer lies in our interpretation of LLVM `Instruction`:

```
%2 = add %1, 0
```

  - ✘ The result of instruction `add %1, 0` is assigned to %2.
  - ✔ %2 **is the** `Value` **representation of instruction** `add %1, 0`.

- Therefore, wherever in later text we use the value %2, we mean to use the instruction `add %1, 0`.

Summary

Let us now return back to the problem we have at the very beginning:

```
%2 = add %1, 0 ; Algebraic Identity
%3 = mul %2, 2
```

- Let `Inst` be a reference to instruction `%2 = add %1, 0`

```
for (auto Iter  = Inst.op_begin();
          Iter != Inst.op_end(); ++Iter)
{ Value *Operand = *Iter; }
```

⇒ Operand `%1`, `0`

```
for (auto Iter  = Inst.user_begin();
          Iter != Inst.user_end(); ++Iter)
{ User *InstUser = *Iter; }
```

⇒ Instruction `mul %2, 2` (alternatively, Value `%3`)

Homework Assignments

In addition to going through `Example1-Transform_Pass_Sample`, we strongly suggest that you think about the following problems:

1. Assume that we have the C code below:

```
y = p + 1;
y = q * 2;
z = y + 3;
```

   What shall be returned as the users of the instruction `y = p + 1`?

2. Think about an alternative way of counting the number of explicit call sites in Assignment 1, by making use of the User-Use-Value relationship.
3. Check the documentation for `llvm::Value`, and see whether there is any method that you benefit from to update all use references at once.

LLVM Pass Manager

LLVM Pass Manager

Recall

Why such isolation exists?

- Better **Readability**
- Very frequently, multiple passes might require the **same** information.
  $\Rightarrow$ The isolation **avoids redundant analysis** (more later).

Now that we have the isolation, how should we build up the connection?

- LLVM **Pass Manager**
- Where is it?

```
virtual void getAnalysisUsage(
    AnalysisUsage &AU) const override {
  AU.setPreservesAll();
}
```

LLVM Pass Manager

What does the Pass Manager do?

- **Requires** & **Preserves** Information
  - If a pass A **requires** pass B, by the time A initiates, LLVM will automatically run B if B has not been run before.
  - If a pass A **preserves** pass B, by the time pass A terminates, LLVM will preserve B, and B will NOT be rerun unless subsequent passes invalidate it.
- By default, a pass requires and preserves _____ other passes.
- The documentation here has more details on the Pass Manager.

LLVM Pass Manager

In this course, we will be mostly using the function calls below:

- `setPreservesAll`: Preserves all previous passes.
- `setPreservesCFG`: Preserves the CFG.
- `addPreserved<A>`: Preserves pass A after `this` pass.
- `addRequired<A>`: Requires pass A to run before `this` pass. `getAnalysis<A>` gets the information from the required pass A.

☞ `Example2-Pass_Manager`

Review

In this tutorial, we have discussed about the followings:

- C++ (P?)Review
- How to write an LLVM Analysis pass?
- How to write an LLVM Transform pass?
  - User-Use-Value Relationship
- How to connect between passes?
  - LLVM Pass Manager: Require & Preserve

☞ **Homework Assignment**: `LocalOpt`