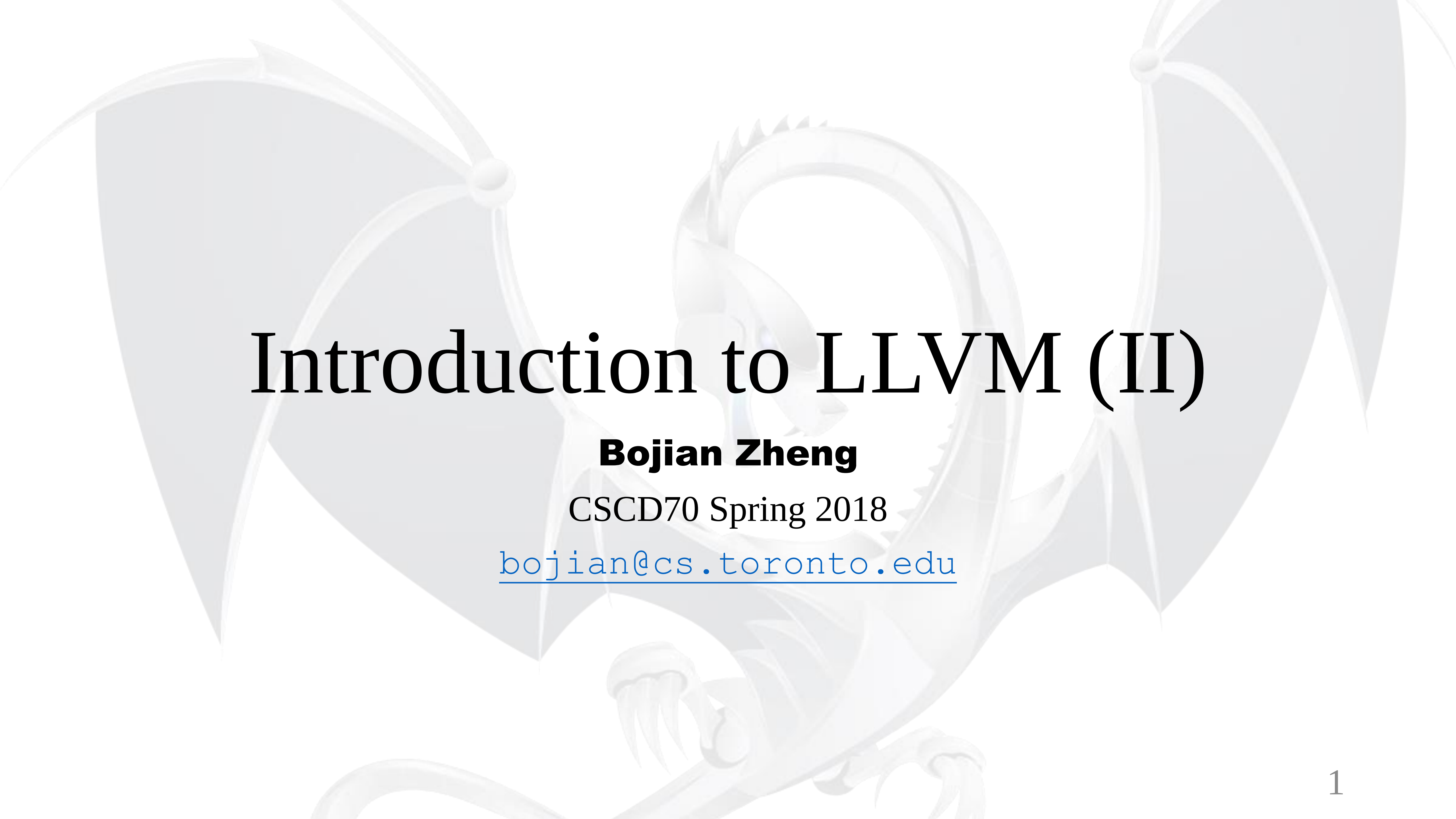


Introduction to LLVM (II)

Bojian Zheng

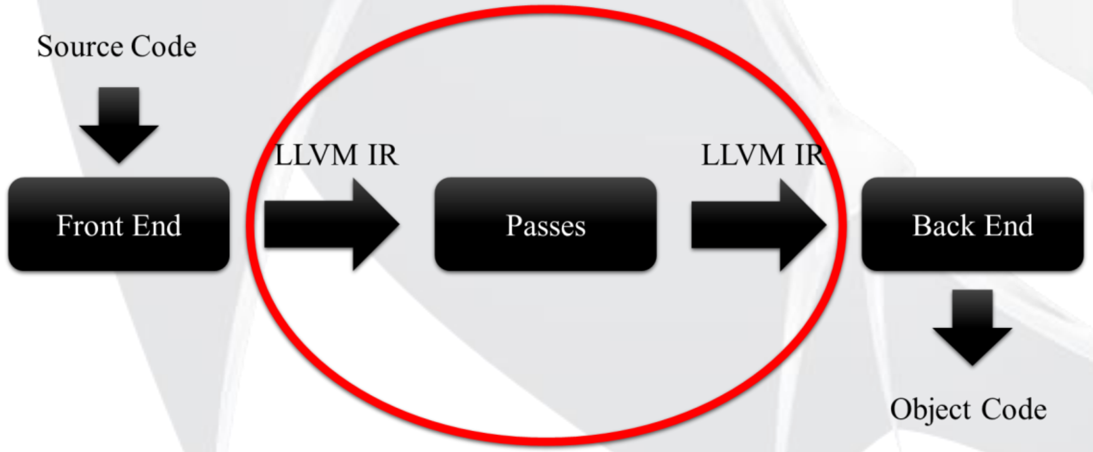
CSCD70 Spring 2018

bojian@cs.toronto.edu

Makefile Error: `Optimize.mk`

- In the `Optimize.mk` file provided in the first assignment, you might need to add **`./`** in front of the optimizer target `FunctionInfo.so`.
- Thanks a lot to **Chengyu (Tyrone) Xiong** for pointing this out.

Review



- Keywords:
  - Intermediate Representation (IR)
  - Optimization Pass
  - Analysis & Transformation

Review

- Keywords:
  - Program Structure
  - Iterators
  - Downcasting
  - LLVM Pass Interface

Transformations

Insert/Remove/Move/Replace Instructions

- Three Options
  - **`Instruction`** class methods.
  - Ask parent (**`BasicBlock`**) to do this.
  - Make use of **`BasicBlockUtils`**.

Attention! **Iterator Hazard**

- As you do transformations, **iterators might be invalidated**.
  - → Demo on `std::vector < unsigned > ::iterator`
- Thanks a lot to **Qiongsi Wu** for bringing this up.

Attention! **Reference Updates**

Original Code

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

Transformed Code

~~%2 = add %1, 0~~

%3 = mul **???**, 2

Questions?

- Keywords:
  - Iterator Hazard
  - References Update (More Later On)

LLVM Instruction: The User-Use-Usee Design Pattern

LLVM Class Hierarchies

$$\begin{tikzcd}\text{Value} \arrow[r] & \text{User} \arrow[r] & \text{Instruction}\end{tikzcd}$$

Value (Usee)

- The **Value** class is the most important base class in LLVM.
  - It has a type (integer, floating point, …): `getType()`
  - It might or might not have a name: `hasName()`, `getName()`
  - Keeps track of a list of **User**s that are using **this Value**.

Instruction (User)

- An **<u>User</u>** keeps track of a list of **<u>Values</u>** that it is using as **<u>Operands</u>**:

```
User user = …

for (auto iter  = user.op_begin();
          iter != user.op_end(); ++iter)
{Value * operand = *iter; …}
```

- An **<u>Instruction</u>** is a **<u>User</u>**.

But wait, …

- Is **Instruction (User)** a **Value (Usee)**?

$$\%2 = \mathrm{add}\ \%1, 10$$

- DO NOT interpret this statement as "the result of Instruction $\mathrm{add}\ \%1, 10$ is assigned to $\%2$", instead, think this way – "$\%2$ is the **Value Representation of Instruction** $\mathrm{add}\ \%1, 10$".
- Therefore, whenever we use the Value $\%2$, we mean to use the Instruction $\mathrm{add}\ \%1, 10$.

To Conclude

- Suppose we have an instruction: `Instruction inst = …`
  - What is this instruction using?

```
for (auto iter  = inst.op_begin();
          iter != inst.op_end(); ++iter)
{…}
```

  - What is using this instruction?

```
for (auto iter  = inst.user_begin();
          iter != inst.user_end(); ++iter)
{…}
```

Questions?

- Keywords:
  - User-Use-Usee Design Pattern

Optimizer Manager

Optimizer Manager

- What is this doing?

```
void Analysis::getAnalysisUsage(AnalysisUsage & AU)
const
{
   AU.setPreservesAll();
}
```

- Very frequently, when writing a pass, we want the followings:
  - What information does this pass require?
  - Will this information still be preserved after this pass?

Questions

- Keywords:
  - Require
  - Preserve

Code Download Links

- https://github.com/ArmageddonKnight/CSCD70-Tutorial-Demo
- **Visitor** Design Pattern
  - serves as an alternative to Dynamic Casting.
  - You can find an example on this in the repository.