

# Sequence 4.6 – IRBuilder

P. de Oliveira Castro    S. Tardieu



## Creating an IR of the program

How to build an IR representation of our program ?

- Instead of writing IR directly, we call a programmatic API, the *IR Builder*
  - Faster: IR is directly built in memory
  - Robust: The API enforces many legality rules of the IR
  - Cleaner: The IR Builder offers high-level abstractions for building the IR



## Principles of Design

- An IR Builder keeps track of an *insert point*. New instructions are added after the insert point which is then automatically moved forward.
- High level builders for complex patterns such as:
  - Calling multi-parameters functions
  - Accessing the field of a structure
  - Creating conditional branches

# Context and Function

- A Builder operates in a given *Context*
  - The *Context* captures the global data of a compilation unit
  - Whenever the builder creates a new global variable, global type, or function declaration, it is added to the *Context*
- A Builder inserts instructions in a given *BasicBlock*
  - A *BasicBlock* operates within a *Context* and belongs to a *Function*

## Inserting new instructions

How to translate  $(10+5)*2$  in IR ?

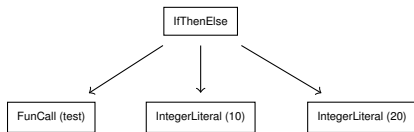
```
llvm::IRBuilder Builder(Context);
llvm::BasicBlock *const body =
    llvm::BasicBlock::Create(Context, "body", current_function);

Builder.SetInsertPoint(body);

llvm::Value * a =
    Builder.CreateAdd(Builder.getInt32(10), Builder.getInt32(5));
llvm::value * b =
    Builder.CreateMul(a, Builder.getInt32(2));
```

# How to translate Tiger AST to LLVM IR

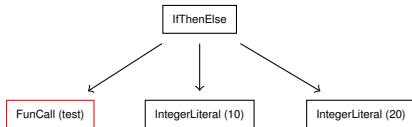
```
if test() then 10 else 20
```



Translate with a visitor that returns LLVM values !

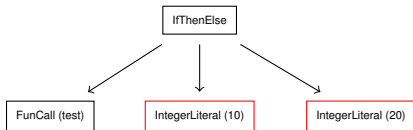
# FunCall

```
// Simplified ! (no static link + no arguments)  
llvm::Value *IRGenerator::visit(const FunCall &call) {  
    const FunDecl &decl = call.get_decl().get();  
    llvm::Function *callee =  
        Module->getFunction(decl.get_external_name().get());  
    return Builder.CreateCall(callee, {}, "call");  
}
```



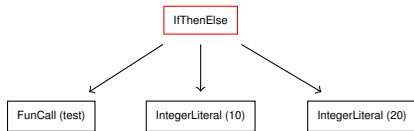
# IntegerLiteral

```
llvm::Value *IRGenerator::visit(const IntegerLiteral &literal) {  
    return Builder.getInt32(literal.value);  
}
```





# IfThenElse



IfThenElse is more complex: diverging control requires multiple basic blocks. To simplify, in the following we assume that the if always returns a value.

## IfThenElse: Prolog

```
llvm::Value *IRGenerator::visit(const IfThenElse &ite) {  
  
    // We create an allocation in the function entry block  
    // to store the if result (see lecture 4.4)  
    llvm::Value *const result =  
        alloca_in_entry(llvm_type(ite.get_type()), "if_result");  
  
    // We create three empty basic blocks  
    llvm::BasicBlock *const then_block =  
        llvm::BasicBlock::Create(Context, "if_then", current_function);  
    llvm::BasicBlock *const else_block =  
        llvm::BasicBlock::Create(Context, "if_else", current_function);  
    llvm::BasicBlock *const end_block =  
        llvm::BasicBlock::Create(Context, "if_end", current_function);
```

## IfThenElse: Condition

We branch depending on the condition,

```
Builder.CreateCondBr(  
    Builder.CreateIsNull(ite.get_condition().accept(*this)),  
    then_block,  
    else_block);
```

`ite.get_condition().accept(*this)` returns the result LLVM Value of the FunCall `test()` translation.

## IfThenElse: Then and Else bodies

```
Builder.SetInsertPoint(then_block);
llvm::Value *const then_result =
    ite.get_then_part().accept(*this);
Builder.CreateStore(then_result, result);
Builder.CreateBr(end_block);

Builder.SetInsertPoint(else_block);
llvm::Value *const else_result =
    ite.get_else_part().accept(*this);
Builder.CreateStore(else_result, result);
Builder.CreateBr(end_block);
```

## IfThenElse: Epilog

```
llvm::Value *const result =  
    alloca_in_entry(llvm_type(ite.get_type()), "if_result");  
    ...  
  
Builder.SetInsertPoint(end_block);  
return Builder.CreateLoad(result);
```