Introduction to LLVM

CMPT 379: Compilers
Instructor: Anoop Sarkar
anoopsarkar.github.io/compilers-class

Setting up

This global variable contains all the generated code.

```
static llvm::Module *TheModule;
```

The calls to Builder will sometimes use TheContext.

```
static llvm::LLVMContext TheContext;

static llvm::IRBuilder<> Builder(TheContext);
```

This is the method used to construct the LLVM intermediate code (IR).

Make sure your yacc actions incrementally generate instructions in the right order

To print out the LLVM output:

```
TheModule->print(llvm::errs(), nullptr);
```

Types in LLVM

decafType is not an LLVM data type. Your yacc program defines it for all the types in a Decaf program

```
llvm::Type *getLLVMType(decafType ty) {
    switch (ty) {
        case voidTy: return Builder.getVoidTy();
        case intTy: return Builder.getInt32Ty();
        case boolTy: return Builder.getInt1Ty();
        case stringTy: return Builder.getInt8PtrTy();
        default: throw runtime_error("unknown type");
    }
}
```

Constants in LLVM

```
llvm::Constant *getZeroInit(decafType ty) {
    switch (ty) {
        case intTy: return Builder.getInt32(0);
        case boolTy: return Builder.getInt1(0);
        default: throw runtime_error("unknown type");
    }
}

llvm::Value *StringConstAST::Codegen() {
    const char *s = StringConst.c_str();
    llvm::Value *GS = Builder.CreateGlobalString(s, "globalstring");
    return Builder.CreateConstGEP2_32(GS, 0, 0, "cast");
}
```

Local Variables in LLVM

```
llvm::AllocaInst *defineVariable(
                llvm::Type *llvmTy,
                string ident)
{
   llvm::AllocaInst* Alloca =
                              Builder.CreateAlloca(llvmTy, 0, ident.c_str());
   syms.enter_symtbl(ident, Alloca);
   return Alloca;
}
```

This is the symbol table you need to keep information about each identifier. We store an AllocaInst* for each variable.

Using the Variable:

```
   llvm::Value *V = syms.access_symtbl(Name);
   return Builder.CreateLoad(V, Name.c_str());
```

Assignment and checking types

```
a = b
```

We need to check if type of lvalue a is the same as type of rvalue b

```
assign: T_ID T_ASSIGN expr
```

Name

rvalue

```
llvm::AllocaInst *Alloca
= (llvm::AllocaInst *)
syms.access_symtbl(Name);
```

```
const llvm::PointerType *ptrTy =
rvalue->getType()->getPointerTo();
```

```
if (ptrTy == Alloca->getType()) {
  Builder.CreateStore(rvalue, Alloca);
}
```

Declaring a Function in LLVM

```
// initalize return type
llvm::Type *returnTy = getLLVMType(ReturnType);
std::vector<llvm::Type *> args;
// args := initialize the vector of argument types
 llvm::Function *func = llvm::Function::Create(
       llvm::FunctionType::get(returnTy, args, false),
       llvm::Function::ExternalLinkage,
       Name,
       TheModule
   );
 syms.enter_symtbl(Name, func);
```

This is the symbol table you need to keep information about each identifier

Promoting Types in LLVM

- What if the variable is of type i1 (boolean)
- But the function only takes i32 (int)
- We must promote the type i1 to i32
- LLVM can do that for you using the ZExt instruction

```
llvm::Value *promo =
        Builder.CreateZExt(*i, Builder.getInt32Ty(), "zexttmp");
```

Basic Blocks in LLVM

```
// Create a new basic block which contains a sequence of LLVM instructions
llvm::BasicBlock *BB =
        llvm::BasicBlock::Create(
                TheContext,
                "entry",
                func);
// insert into symbol table
syms.enter_symtbl(string("entry"), BB);
// All subsequent calls to IRBuilder will place instructions in this location
Builder.SetInsertPoint(BB);
```

Function parameters

When you generate code for a method declaration do the following:

1. Create a new symbol table for local variables
2. Create a `BasicBlock`, let's say BB
3. Set insertion point for instructions `Builder.SetInsertPoint(BB)`
4. Add the arguments to the function as allocated on the stack (next slide)
5. You can check if a function has a `return` statement by checking if the value of `BB->getTerminator()` is `NULL`.

Function parameters

```
func foo(x int) int {
        x = 1;
}
```

For `Function*` `func` iterate through the function arguments and allocate them into the stack.

```
for (auto &Arg : func->args()) {
  llvm::AllocaInst *Alloca =
       CreateEntryBlockAlloca(func, Arg.getName());
  // Store the initial value into the alloca
  Builder.CreateStore(&Arg, Alloca);
  // Add to symbol table
  syms.enter_symtbl(Arg.getName(), Alloca);
}
```

Useful Tricks in LLVM

- Finding the current function you are in: `llvm::Function *func = Builder.GetInsertBlock()->getParent();`
- External function

```
llvm::Function::Create(
        llvm::FunctionType::get(returnTy, args, false),
        llvm::Function::ExternalLinkage,
        Name,
        TheModule);
```

Modulus in LLVM

- Use `CreateSRem()` for signed operators in Decaf
- LLVM uses the C/C++ style modulus
- So `-4%3 == -1`

Backward Function Declarations

Backwards Declarations

```
extern func print_int(int) void;
package Test {
  func main() int {
    test(10, 13);
  }
  func test(a int, b int) void {
    print_int(a);
    print_int(b);
  }
}
```

Iterate through the list of function signatures and insert them into the symbol table.

Control Flow in LLVM

“Backpatching” in LLVM

- Inside IfStmt->Codegen:
  - Set up a new symbol table for code locations
  - Create a new BasicBlock called iftrue (see slide 9)
  - Create a new BasicBlock called iffalse
  - Create a new BasicBlock called end
  - Subsequent code generation anywhere else can insert code into these code locations
  - Can be used for break, continue, short-circuits, etc.

"Backpatching" in LLVM

Setting up the branching between Basic Blocks:

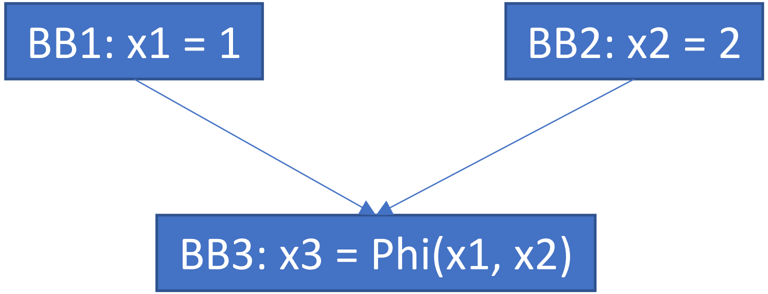
```
// val contains the Expr value for the conditional
Builder.CreateCondBr(val, IfTrueBB, EndBB);
Builder.SetInsertPoint(IfTrueBB);
IfTrueBlock->Codegen();
```

After the IfStmt we continue with the end Basic Block:

```
Builder.CreateBr(EndBB);
// pop the symbol table after IfStmt Codegen is done
Builder.SetInsertPoint(EndBB);
```

Static Single Assignment in LLVM

- For normal control flow using CreateBr and CreateCondBr no need for Phi functions
- LLVM produces the Phi functions automatically using algorithms we will study in class



Short-circuit of boolean expressions

- For short circuit of boolean expressions you have to write the PHI function yourself

```
llvm::PHINode *val =
      Builder.CreatePHI(type, 2, "phival");
// type is an LLVM::Type
val->addIncoming(L, CurBB);
val->addIncoming(opval, OpValBB);
// CurBB and OpValBB are the two basic blocks that are
incoming blocks for the PHI function
```

Short-circuit of boolean expressions

```
package sckt {
        func main() int {
                var a, b, c bool;
                a = true; b = false;
                c = a || b;
        }
}
```

Short-circuit of boolean expressions

```
; ModuleID = 'sckt'
source_filename = "DecafComp"
define i32 @main() {
func:
  ; removed all variable init code
  store i1 true, i1* %a
  store i1 false, i1* %b
  %a1 = load i1, i1* %a
  br i1 %a1, label %skctend, label %noskct
```

```
c = a || b;
```

if a is True then
do not evaluate b

```
noskct: ; preds = %func
  %b2 = load i1, i1* %b
  %ortmp = or i1 %a1, %b2
  br label %skctend
```

```
skctend: ; preds = %noskct, %func
  %phival =
    phi i1 [ %a1, %func ], [ %ortmp, %noskct ]
  store i1 %phival, i1* %c
  ret i32 0
}
```

Q: Why is CreatePHI using OpValBB not NoSkctBB aka nosckt: ?

Short-circuit of boolean expressions

```
; ModuleID = 'sckt'
source_filename = "DecafComp"
define i32 @main() {
func:
   ; removed all variable init code
   store i1 true, i1* %a
   store i1 false, i1* %b
   %a1 = load i1, i1* %a
   br i1 %a1, label %skctend, label %noskct

noskct: ; preds = %func
  %b2 = load i1, i1* %b
  %ortmp = or i1 %a1, %b2
  br label %skctend
```

```
llvm::Value *opval = Builder.CreateOr(L, R, "ortmp");
llvm::BasicBlock *OpValBB = Builder.GetInsertBlock();
```

```
llvm::Value *L = LHS->Codegen();
```

```
c = a || b;
```

if a is True then
do not evaluate b

```
llvm::PHINode *val =
   Builder.CreatePHI(L->getType(), 2, "phival");
val->addIncoming(L, CurBB); / * func */
val->addIncoming(opval, OpValBB); / * noskct */
```

```
skctend: ; preds = %noskct, %func
   %phival =
     phi i1 [ %a1, %func ], [ %ortmp, %noskct ]
   store i1 %phival, i1* %c
   ret i32 0
```