CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop

Setting up

```
static Module *TheModule;
```

This global variable contains all the generated code.

```
static LLVMContext TheContext;
```

The calls to Builder will sometimes use TheContext.

```
static 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

Types in LLVM

```
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;
}
```

Using the Variable:

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

Declaring a Function in LLVM

```
llvm::Type *returnTy = /* initalize return type */
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);
```

Promoting Types in LLVM

- What if the variable is of type i1 (boolean)
- But the function only takes i32 (int)
- We have to 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(
                llvm::getGlobalContext(),
                "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);
```

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);
```

“Backpatching” in LLVM

- Inside IfStmt->Codegen:
  - Set up a new symbol table for code locations
  - Create a new BasicBlock called iftrue
  - 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

Static Single Assignment in LLVM

- 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 */
```