Mini-Assignment - 2

Vikas Patnala (CS20BTECH11037)

February 8, 2024

Q1

(a)

Some examples of how Lowering of Data Structures from C/CPP to LLVM IR is done

1. Array : The array in the C and C++ languages are lowered into the LLVM IR as contiguous blocks of memory and the elements are accessed using load and store instructions and also using the pointer arithmetic. Below two lines are the format of the datatype in LLVM IR and the example taken from the generated array.ll file

   ```
   [<# elements> x <elementtype>]
   [10 x i32]
   ```

2. Struct : The struct is translated to LLVM IR as LLVM struct types where each member corresponds to a field within the LLVM struct type and for accessing the struct member we can use *Get Element Pointer(GEP)* instruction. Below two lines are the format of the datatype in LLVM IR and the example taken from the generated array.ll file

   ```
   %st = type { <type list> }
   %struct.st = type { i32, i32, i32 }
   ```

The LLVM IR codes are for the mentioned data types are generated in the directory `Mini-Asgn-2_CS20BTECH11037/data_structs/out/`.

(b)

In LLVM IR the CPP Array is declared using the Array Type of LLVM IR where as CPP Vector is declared using the Vector Type of LLVM IR. The syntax of the array type of LLVM is showed in the part (a) while the syntax and an example of vector type is

```
< <# elements> x <elementtype> >
<16 x i32>
```

The vector type data structure represents the collection of elements of same type that can be operated in parallel using SIMD instructions due to this property the vector type is more powerful than the array type i.e. using vector type operations can be parallelised which can improve the performance and efficiency of the program that have data parallelism (such tasks are image processing and large computations).

Another difference between the these types is vector is not a aggregate type where as array type is. An aggregate type is a type that can be decomposed into multiple elements of other types (such as a structure type is an aggregate type which can be decomposed into multiple element of different types. An array type is an aggregate type which can be decomposed into multiple elements of same type). A vector type is not an aggregate type because it cann't decomposed into multiple elements of other types which is vector type has some limitations in LLVM IR such as not being able to use the *getelementptr* instruction to get the indexed value instead it uses *extractelement* and *insertelement* istructions to manipulate the vector elements.

(c)

LLVM IR doesn't support the *union* datatype instead they are lowered from C and CPP to LLVM IR as a *struct* with one element consists of largest size datatype of union elements.

```
union un{
    char a;
    int b;
    double c;
};
```

The LLVM IR for the above union data structure is

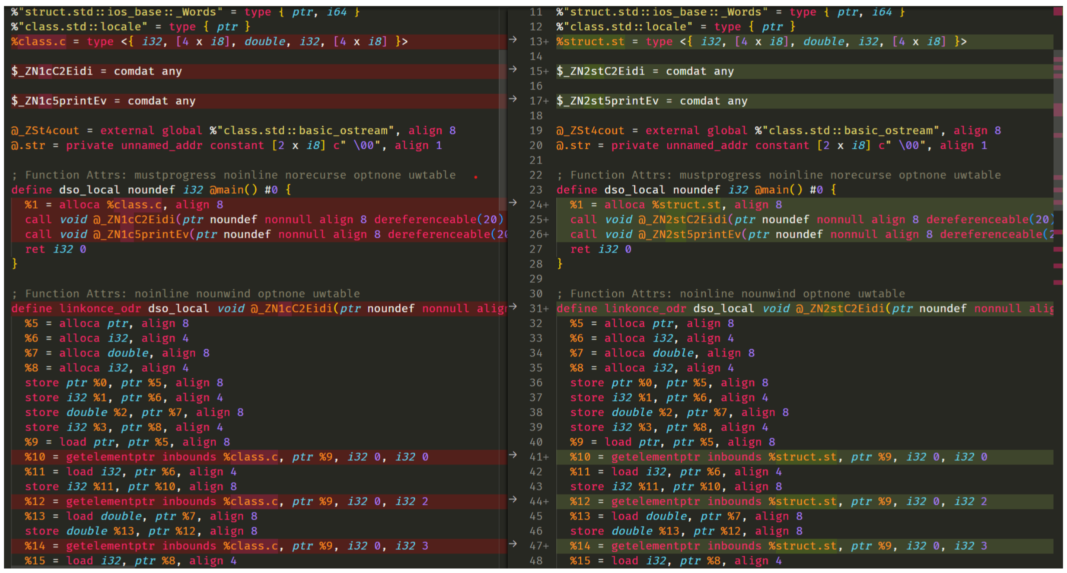
```
%union.un = type { double }
```

If I want to access the other datatypes of the union then we need to cast the struct to whatever type the frontend want. If we try to assign the value of the datatype which is not recently written will lead to undefined behaviour or accessing any two variables at the same time the new storing datatype will override the previous datatypes values. Hence unions are not type safe and the type safety should have to take by the frontend compiler to ensure the proper usage of union.

(d)

As we know that the *struct* and *class* in cpp (everything a class can do can also potentially done by a struct such as inheritance, polymorphism, encapulation etc) are almost equivalent except for some default accessibility (when inheriting from class the default mode is private where as for struct is public).

Similarly when they are lowered down to the LLVM IR the *class* is represented using a struct means there is no data struture for the class in the IR where as it uses the struct and other metadata to keep track of the class functionality.



```
%"struct.std::ios_base::_Words" = type { ptr, i64 }
%"class.std::locale" = type { ptr }
%class.c = type <{ i32, [4 x i8], double, i32, [4 x i8] }>

$_ZN1cC2Eidi = comdat any

$_ZN1c5printEv = comdat any

@_ZSt4cout = external global %"class.std::basic_ostream", align 8
@.str = private unnamed_addr constant [2 x i8] c" \00", align 1

; Function Attrs: mustprogress noinline norecurse optnone uwtable
define dso_local noundef i32 @main() #0 {
  %1 = alloca %class.c, align 8
  call void @_ZN1cC2Eidi(ptr noundef nonnull align 8 dereferenceable(20)
  call void @_ZN1c5printEv(ptr noundef nonnull align 8 dereferenceable(2
  ret i32 0
}

; Function Attrs: noinline nounwind optnone uwtable
define linkonce_odr dso_local void @_ZN1cC2Eidi(ptr noundef nonnull alig
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  %7 = alloca double, align 8
  %8 = alloca i32, align 4
  store ptr %0, ptr %5, align 8
  store i32 %1, ptr %6, align 4
  store double %2, ptr %7, align 8
  store i32 %3, ptr %8, align 4
  %9 = load ptr, ptr %5, align 8
  %10 = getelementptr inbounds %class.c, ptr %9, i32 0, i32 0
  %11 = load i32, ptr %6, align 4
  store i32 %11, ptr %10, align 8
  %12 = getelementptr inbounds %class.c, ptr %9, i32 0, i32 2
  %13 = load double, ptr %7, align 8
  store double %13, ptr %12, align 8
  %14 = getelementptr inbounds %class.c, ptr %9, i32 0, i32 3
  %15 = load i32, ptr %8, align 4
```

```
%"struct.std::ios_base::_Words" = type { ptr, i64 }
%"class.std::locale" = type { ptr }
%struct.st = type <{ i32, [4 x i8], double, i32, [4 x i8] }>

$_ZN2stC2Eidi = comdat any

$_ZN2st5printEv = comdat any

@_ZSt4cout = external global %"class.std::basic_ostream", align 8
@.str = private unnamed_addr constant [2 x i8] c" \00", align 1

; Function Attrs: mustprogress noinline norecurse optnone uwtable
define dso_local noundef i32 @main() #0 {
  %1 = alloca %struct.st, align 8
  call void @_ZN2stC2Eidi(ptr noundef nonnull align 8 dereferenceable(20)
  call void @_ZN2st5printEv(ptr noundef nonnull align 8 dereferenceable(2
  ret i32 0
}

; Function Attrs: noinline nounwind optnone uwtable
define linkonce_odr dso_local void @_ZN2stC2Eidi(ptr noundef nonnull alig
  %5 = alloca ptr, align 8
  %6 = alloca i32, align 4
  %7 = alloca double, align 8
  %8 = alloca i32, align 4
  store ptr %0, ptr %5, align 8
  store i32 %1, ptr %6, align 4
  store double %2, ptr %7, align 8
  store i32 %3, ptr %8, align 4
  %9 = load ptr, ptr %5, align 8
  %10 = getelementptr inbounds %struct.st, ptr %9, i32 0, i32 0
  %11 = load i32, ptr %6, align 4
  store i32 %11, ptr %10, align 8
  %12 = getelementptr inbounds %struct.st, ptr %9, i32 0, i32 2
  %13 = load double, ptr %7, align 8
  store double %13, ptr %12, align 8
  %14 = getelementptr inbounds %struct.st, ptr %9, i32 0, i32 3
  %15 = load i32, ptr %8, align 4
```

Figure 1: Difference between the IR of class and struct files with a naive example

Q2

Representation

LLVM IR uses the *IEEE 754* standard representation for representing the floating point numbers. LLVM supports the following floating point data types

1. *float* - single precision

2. *double* - double precision

3. *fp128* - extended precision

there are also other architecture specific types such as $x86\_fp80$. IEEE 754 standard specifies the format using a sign bit, exponent and mantissa.

Operations Supported

LLVM IR supports multiple operations for floating point numbers such as addition(*fadd*), subtraction(*fsub*), multiplication(*fmul*), division(*fdiv*), convertion from and to to integer types(*sitofp*, *fptosi*, *fptoui*) and comparision operators(*fcmp*).

Optimizations

There are several optimization that can be listing a few

- Constant Folding : This optimization evaluates constant expressions at compile time and replace them with direct floating point constants such as replacing *fadd float 3.2, 2.6* with *float 5.8*
- Algebraic Simplification : In this optimization we Apply algebraic identities and rules to simplify expression for example *fmul float %f, 1.0* can be simplified to *%f*
- Strength Reduction : This optimization try to replace to expensive operations with cheaper ones which doesn't effect the correctness of code for example *fdiv float %f, 5.0* can be replaced with *fmul float %f, 0.2.*

These operations are not always safe(doesn't ensure correctness), as they may introduce some errors(such as numerical precision before and after the optimization, NaN can be introduced) and violate the IEEE 754 standard. And these optimization are usually controlled by *fast-math flags.*

Type lowering is a concept that is transforming types that are not supported by the target architecture into a type that that architecture should support and one such example is *fp128* is not supported by most of architectures and it is lowered to pair of *i64* values to represent the sign, exponent and mantissa.

Q3

Expressibility of a programming language is defined in terms of it's ease of writing and understanding code by both the compiler and humans.

The Expressibility of LLVM IR grammar is high because LLVM IR is easy to understand by the compiler as well as by the humans because LLVM IR is close to assembly while also maintaining the high level programming languages semantics.

LLVM supports multiple features such as control flow (branches, loops), functions. This is also designed to facilitate analysis and transformations on IR. LLVM IR is well known for its code design because of which many developers can easily extend the current features to domain specific optimizations and analysis. LLVM IR is made simple by including designs such as Single Static Assignment (where every local variable is assigned and accessed only once).

We can also see that grammars for the languages such as C, CPP are very complex and they span over million of lines whereas when we see the grammar of the LLVM it is very clear and striaght forward for understanding and grammar for LLVM is around 5300 lines from which we can say that the LLVM is easy for compiler to understand

There are two way of lowering the ternary operator ? : from CPP to LLVM IR

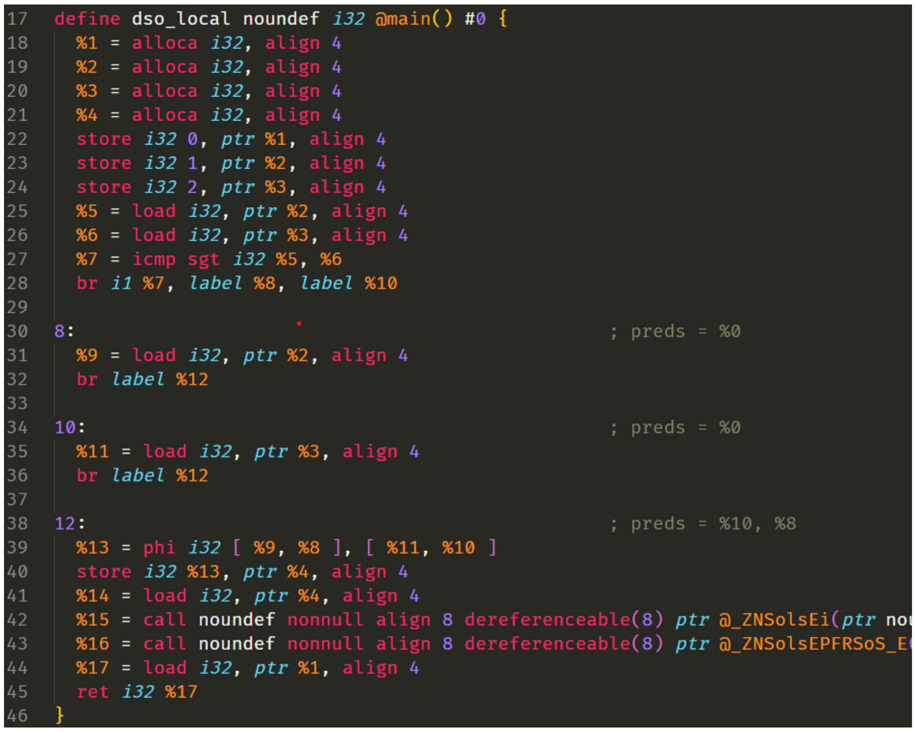
1. select Instruction : This instruction takes a boolen condition and two values and returns one of values depending on the condition. The ternary operation $a?b : c$ is reduced to LLVM IR as

```
// parsing grammar for select
 ::= 'select' TypeAndValue ',' TypeAndValue ','
   TypeAndValue
// ex
%cond = icmp ne i32 %x 0
%result = select i1 %cond, i32 %b, i32 %c
```

2. phi Instruction : This instruction takes a list of pairs of values and basic blocks and returns the value corresponding to the basic block that was executed before. The ternary operation $a?b : c$ is reduced to LLVM IR as

```
// parsing grammar for phi instruction
```

```
::= ’phi’ FastMathFlag* Type ’[’ Value ’,’ Value ’]’ (’,’
    ’[’ Value ’,’ Value ’]’)*
// ex
br i1 %cond, label %true, label %false
true:
    br label %merge
false:
    br label %merge
merge:
    %result = phi i32 [%b, %true], [%c, %false]
```



```
define dso_local noundef i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  store i32 1, ptr %2, align 4
  store i32 2, ptr %3, align 4
  %5 = load i32, ptr %2, align 4
  %6 = load i32, ptr %3, align 4
  %7 = icmp sgt i32 %5, %6
  br i1 %7, label %8, label %10

8:                                                ; preds = %0
  %9 = load i32, ptr %2, align 4
  br label %12

10:                                               ; preds = %0
  %11 = load i32, ptr %3, align 4
  br label %12

12:                                               ; preds = %10, %8
  %13 = phi i32 [ %9, %8 ], [ %11, %10 ]
  store i32 %13, ptr %4, align 4
  %14 = load i32, ptr %4, align 4
  %15 = call noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEi(ptr no
  %16 = call noundef nonnull align 8 dereferenceable(8) ptr @_ZNSolsEPFRSoS_E(
  %17 = load i32, ptr %1, align 4
  ret i32 %17
}
```

Figure 2: LLVM IR of the ternary operator

The example code which I have submitted *ternaryOpp.cpp* is reduced to IR using the second *phi* instruction only as it is more general and can support more than two branches where as *select* supports only two branches.

1 References

- LLVM language reference manual
- Floating-Point Arithmetic Paper
- Official LLVM IR grammar
- A reference manual for how LLVM does Mapping of high level languages