LLVM Programming

2023.03.23

SWPP Practice Session

Seunghyeon Nam

What Exactly Is LLVM?

- LLVM is an intermediate representation (IR)
- LLVM is an IR-based compiler framework

Intermediate Representation (IR)

- It’s a language by itself!
- Bridges over high-level languages and assemblies

LLVM IR

- IR used in LLVM framework
- Static single assignment (SSA) form
- Strongly typed with no implicit conversions
- Has notion of function, control flow, block and variable
  - Some even say it’s a ‘C with vectors’

LLVM IR

```
define dso_local noundef i64 @_Z16llvm_ir_functionil(i32 noundef %arg0, i64 noundef %arg1) local_unnamed_addr #0 {
entry:
  %conv = sext i32 %arg0 to i64
  %add = add nsw i64 %conv, 42
  %cmp = icmp sgt i32 %arg0, 8
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds
  %mul = mul nsw i64 %add, %arg1
  br label %if.end

if.else:                                          ; preds
  %div = sdiv i64 %add, %arg1
  br label %if.end

if.end:                                           ; preds
  %v2.0 = phi i64 [ %mul, %if.then ], [ %div, %if.else ]
  ret i64 %v2.0
}
```

```cpp
#include <cstdint>

int64_t llvm_ir_function(int32_t arg0, int64_t arg1) {
    constexpr int64_t c0 = 42;
    const int64_t v1 = arg0 + c0;
    int64_t v2;
    if (v1 > 50) {
        v2 = v1 * arg1;
    } else {
        v2 = v1 / arg1;
    }
    return v2;
}
```

```
clang++ -S -emit-llvm -O2 main.cpp
```

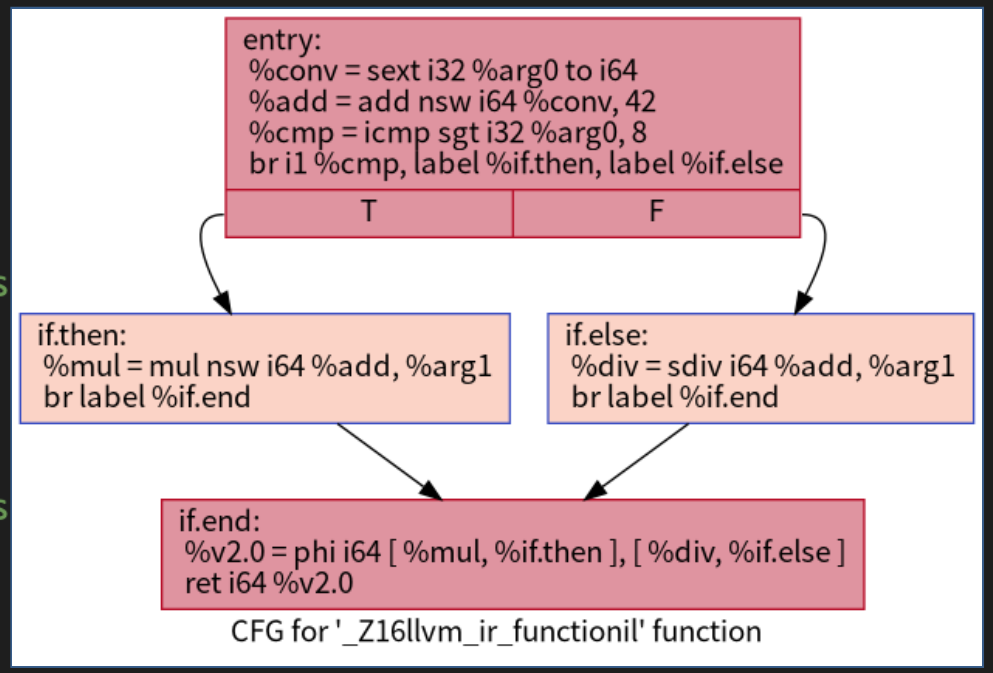
LLVM IR

```
define dso_local noundef i64 @_Z16llvm_ir_functionil(i32 noundef %arg0, i64 noundef %arg1) local_unnamed_addr #0 {
entry:
  %conv = sext i32 %arg0 to i64
  %add = add nsw i64 %conv, 42
  %cmp = icmp sgt i32 %arg0, 8
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds
  %mul = mul nsw i64 %add, %arg1
  br label %if.end

if.else:                                          ; preds
  %div = sdiv i64 %add, %arg1
  br label %if.end

if.end:                                           ; preds
  %v2.0 = phi i64 [ %mul, %if.then ], [ %div, %if.else ]
  ret i64 %v2.0
}
```



Look at 04.Debugging.pdf

LLVM IR

```
define dso_local noundef i64 @_Z9factorialm(i64 noundef %n) local_unnamed_addr #0 {
entry:
  %cmp.not4 = icmp ult i64 %n, 2
  br i1 %cmp.not4, label %for.cond.cleanup, label %for.body

for.cond.cleanup:                                 ; preds = %for.
  %ret.0.lcssa = phi i64 [ 1, %entry ], [ %mul, %for.body ]
  ret i64 %ret.0.lcssa

for.body:                                         ; preds = %entr
  %i.06 = phi i64 [ %inc, %for.body ], [ 2, %entry ]
  %ret.05 = phi i64 [ %mul, %for.body ], [ 1, %entry ]
  %mul = mul i64 %i.06, %ret.05
  %inc = add nuw i64 %i.06, 1
  %exitcond = icmp eq i64 %i.06, %n
  br i1 %exitcond, label %for.cond.cleanup, label %for.body, !llvm.loop !5
}
```

```
uint64_t factorial(uint64_t n) {
    uint64_t ret = 1;
    for (uint64_t i = 2; i <= n; i++) {
        ret *= i;
    }
    return ret;
}
```

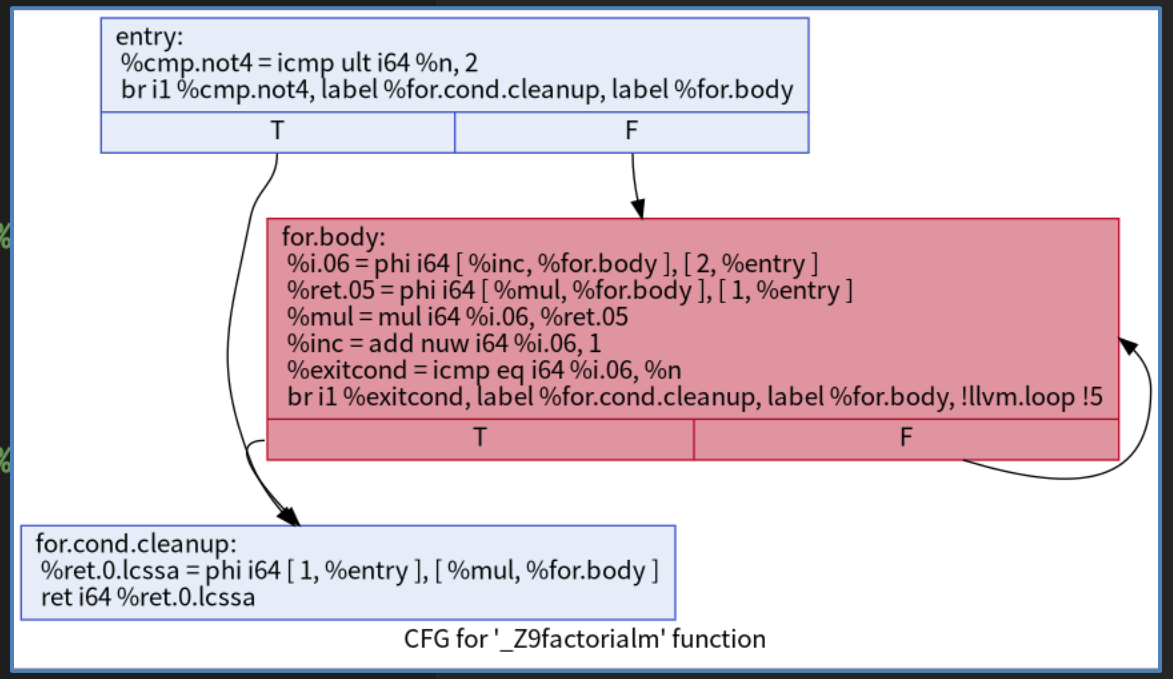
```
clang++ -S -emit-llvm –O2 main.cpp
```

LLVM IR

```
define dso_local noundef i64 @_Z9factorialm(i64 noundef %n) local_unnamed_addr #0 {
entry:
  %cmp.not4 = icmp ult i64 %n, 2
  br i1 %cmp.not4, label %for.cond.cleanup, label %for.body

for.cond.cleanup:                                 ; preds = %
  %ret.0.lcssa = phi i64 [ 1, %entry ], [ %mul, %for.body ]
  ret i64 %ret.0.lcssa

for.body:                                         ; preds = %
  %i.06 = phi i64 [ %inc, %for.body ], [ 2, %entry ]
  %ret.05 = phi i64 [ %mul, %for.body ], [ 1, %entry ]
  %mul = mul i64 %i.06, %ret.05
  %inc = add nuw i64 %i.06, 1
  %exitcond = icmp eq i64 %i.06, %n
  br i1 %exitcond, label %for.cond.cleanup, label %for.body, !llvm.loop !5
}
```



CFG for '_Z9factorialm' function

Look at 04.Debugging.pdf

Function

- Just an ordinary function, nothing new…
- Can be `call`ed & ends upon `ret`
- Composed of 0+ argument(s) and 1+ basic block(s)

Basic Block

- A unit of control flow
- Functions as ’`br`(anch) target’
  - AKA jump label
- Must end with terminator
  - Terminator instructions: `ret`, `br`, `switch`, `unreachable`, …
- Composed of 1+ instruction(s)

Instruction

- Single operation
- Takes & returns variable(s)
- Various effects (arithmetic, control flow, etc)
- List of syntax are available here

Arithmetic Instruction

- `add, sub, mul, udiv, sdiv, shl, ashr, and, xor, …`
- `<result> = add <ty> <op1>, <op2>`

Conversion Instruction

- `zext, trunc, inttoptr, …`
- Necessary as LLVM does not have implicit type casting
- `<result> = trunc <ty> <value> to <ty2>`

Comparison Instruction

- `icmp, fcmp, …`
- `<result> = icmp <cond> <ty> <op1>, <op2>`
  - `<cond>` is the type of comparison (eq, ge, lt, …)
  - Returns `i1` (bool)

Control Flow Instruction

- `br, switch, call, ret, phi`
- `br` : (Un)conditional jump to a basic block (1-2 dests)
- `switch` : Match and jump to a basic block (1+ dests)
- `call` : Call a function and use its return value if there's one
- `ret` : End function and optionally return a value

Control Flow Instruction

- `br, switch, call, ret, phi`
- `phi` : Fetch value from one of the basic block's predecessor
  - Used to express value that is dependent to the control flow
  - Necessary as LLVM uses SSA (conditional assignment?)
  - `<result> = phi <ty> [<val0>, <label0>], ...`
  - If the block before was `<labelN>`, `<result> = <valN>`

Memory Access Instruction

- `load, store, …`
- `<result> = load <ty>, ptr <pointer>[, align <alignment>]`
- `store <ty> <value>, ptr <pointer>[, align <alignment>]`
- `<pointer>` should be aligned with the `<alignment>`

Memory Management Instruction

- `alloca, ~~malloc~~`
- `<result> = alloca <type> [, <ty> <NumElements>]`
- Allocates memory space on the runtime stack
  - Automatically freed at the end of the function
- Can only appear at the beginning of the function
  - AKA ‘entry block’

Types in LLVM IR

- Integer types (`u8`, `i32`, `u64`, `u4096`, …)
- Floating-point types (`f32`, `f64`, …)
- Pointer types (`*i64`, `*u16`, …)
- Vector types (`<4 x i32>`, `<16 x f64>`, …)

LLVM Framework

- Collection of middle-end & back-ends
  that can be reused among various languages and hardwares
- Middle-end is responsible for code analysis and optimization
  - IR to IR transformation
- Back-end is responsible for target hardware
  - IR to assembly transformation

LLVM Framework

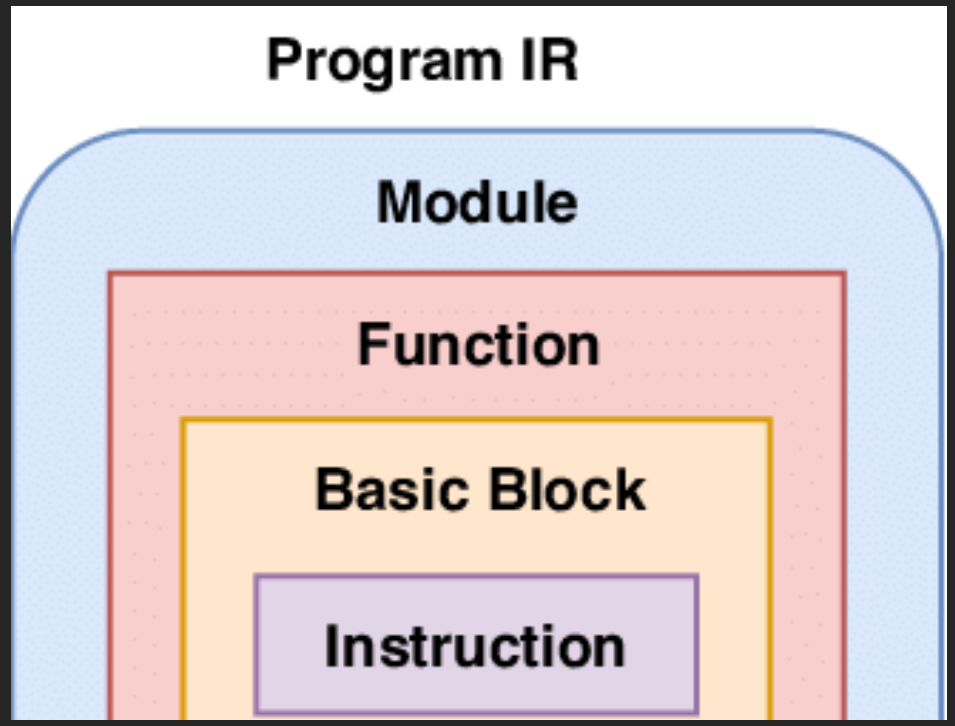
- Front-end is responsible for converting source language into LLVM IR
  - Significantly less workload than implementing full toolchain
  - Enjoy well-implemented and tested optimizations for free!
  - And support multiple architectures for free as well!

LLVM API

- C++ API to manipulate LLVM IR programs
- Manipulate means a lot of things
  - Iterate through basic blocks / instructions / etc
  - Find and replace certain pattern in an IR program
  - Find uses and definition of some value
  - Insert & use attributes

LLVM IR & API

- Each IR component has a corresponding type in the API




```
llvm::Module
llvm::Function
llvm::BasicBlock
llvm::Instruction
```

LLVM IR & API

- Each instruction is also a type in the API
- They all inherit the same base class, `llvm::Instruction`
- See here for the full hierarchy!

LLVM IR & API

- Almost everything is an (inherits) `llvm::Value` in the API
  - This includes `Function`, `BasicBlock`, `Instruction` as well!
- Use `llvm::dyn_cast<T>()` to cast a type into another
  - Nonnull if the cast succeeds, `nullptr` if it fails