

Code Generation for Data Processing

Lecture 4: LLVM and IR Design

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2022/23

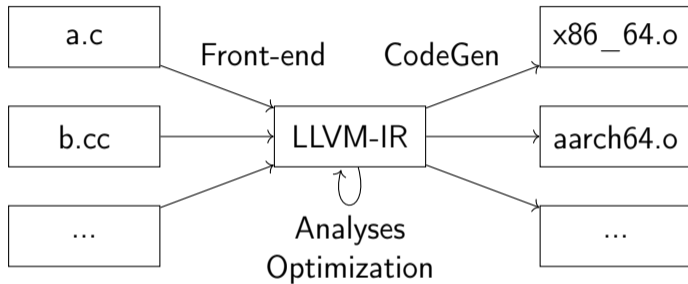
LLVM “Core” Library

- ▶ Optimizer and compiler back-end
- ▶ “Set of compiler components”
 - ▶ IRs: LLVM-IR, SelDag, MIR
 - ▶ Analyses and Optimizations
 - ▶ Code generation back-ends
- ▶ Started from Chris Lattner’s master’s thesis
- ▶ Used for C, C++, Swift, D, Julia, Rust, Haskell, ...

LLVM Project

- ▶ Umbrella for several projects related to compilers/toolchain
 - ▶ LLVM Core
 - ▶ Clang: C/C++ front-end for LLVM
 - ▶ libc++, compiler-rt: runtime support
 - ▶ LLDB: debugger
 - ▶ LLD: linker
 - ▶ MLIR: experimental IR framework

LLVM: Overview



- ▶ Independent front-end derives LLVM-IR, LLVM does opt. and code gen.
- ▶ LTO: dump LLVM-IR into object file, optimize at link-time

LLVM-IR: Overview

- ▶ SSA-based IR, representations textual, bitcode, in-memory
- ▶ Hierarchical structure
 - ▶ Module
 - ▶ Functions, global variables
 - ▶ Basic blocks
 - ▶ Instructions
- ▶ Strongly/strictly typed

```
define dso_local i32 @foo(i32 %0) {  
    %2 = icmp eq i32 %0, 0  
    br i1 %2, label %10, label %3  
  
3: ; preds = %1, %3  
    %4 = phi i32 [ %7, %3 ], [ 1, %1 ]  
    %5 = phi i32 [ %8, %3 ], [ %0, %1 ]  
    %6 = mul nsw i32 %5, %5  
    %7 = mul nsw i32 %6, %4  
    %8 = add nsw i32 %5, -1  
    %9 = icmp eq i32 %8, 0  
    br i1 %9, label %10, label %3  
  
10: ; preds = %3, %1  
    %11 = phi i32 [ 1, %1 ], [ %7, %3 ]  
    ret i32 %11  
}
```

LLVM-IR: Data types

- ▶ First class types:
 - ▶ `i<N>` – arbitrary bit width integer, e.g. `i1`, `i25`, `i1942652`
 - ▶ `ptr/ptr addrspace(1)` – pointer with optional address space
 - ▶ `float/double/half/bfloat/fp128/...`
 - ▶ `<N x ty>` – vector type, e.g. `<4 x i32>`
- ▶ Aggregate types:
 - ▶ `[N x ty]` – constant-size array type, e.g. `[32 x float]`
 - ▶ `{ ty, ... }` – struct (can be packed/opaque), e.g. `{i32, float}`
- ▶ Other types:
 - ▶ `ty (ty, ...)` – function type, e.g. `{i32, i32} (ptr, ...)`
 - ▶ `void`
 - ▶ `label/token/metadata`

LLVM-IR: Modules

- ▶ Top-level entity, one compilation unit – akin to C/C++
- ▶ Contains global values, specified with linkage type

- ▶ Global variable declarations/definitions

```
@externInt = external global i32, align 4
```

```
@globVar = global i32 4, align 4
```

```
@staticPtr = internal global ptr null, align 8
```

- ▶ Function declarations/definitions

```
declare i32 @readPtr(ptr)
```

```
define i32 @return1() {
```

```
    ret i32 1
```

```
}
```

- ▶ Global named metadata (discarded during compilation)

LLVM-IR: Functions

- ▶ Functions definitions contain all code, not nestable
- ▶ Single return type (or `void`), multiple parameters, list of basic blocks
 - ▶ No basic blocks \Rightarrow function declaration
- ▶ Specifiers for `callconv`, section name, other attributes
 - ▶ E.g.: `noinline/alwaysinline, noreturn, readonly`
- ▶ Parameter and return can also have attributes
 - ▶ E.g.: `noalias, nonnull, sret(<ty>)`

LLVM-IR: Basic Block

- ▶ Sequence of instructions
 - ▶ ϕ nodes come first
 - ▶ Regular instructions come next
 - ▶ Must end with a terminator
- ▶ First block in function is entry block
Entry block cannot be branch target

LLVM-IR: Instructions – Control Flow and Terminators

- ▶ Terminators end a block/modify control flow
- ▶ `ret <ty> <val>/ret void`
- ▶ `br label <dest>/br i1 <cond>, label <then>, label <else>`
- ▶ `switch/indirectbr`
- ▶ `unreachable`
- ▶ Few others for exception handling

- ▶ Not a terminator: `call`

LLVM-IR: Instructions – Arithmetic-Logical

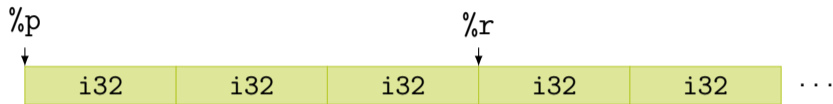
- ▶ add/sub/mul/udiv/sdiv/urem/srem
 - ▶ Arithmetic uses two's complement
 - ▶ Division corner cases are *undefined behavior*
- ▶ fneg/fadd/fsub/fmul/fdiv/frem
- ▶ shl/lshr/ashr/and/or/xor
 - ▶ Out-of-range shifts have an undefined result
- ▶ icmp <pred>/fcmp <pred>/select <cond>, <then>, <else>
- ▶ trunc/zext/sext/fptrunc/fpext/fptoui/fptosi/uitofp/sitofp
- ▶ bitcast
 - ▶ Cast between equi-sized datatypes by reinterpreting bits

LLVM-IR: Instructions – Memory and Pointer

- ▶ `alloca <ty>` – allocate addressable stack slot
- ▶ `load <ty>, ptr <ptr>/store <ty> <val>, ptr <ptr>`
 - ▶ May be volatile (e.g., MMIO) and/or atomic
- ▶ `cmpxchg/atomicrmw` – similar to hardware operations
- ▶ `ptrtoint/inttoptr`
 - ▶ Changes provenance! `inttoptr(ptrtoint(%x))` is not equal to `%x`
- ▶ `getelementptr` – address computation on `ptr/structs/arrays`

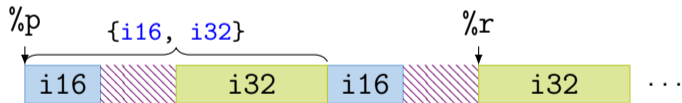
LLVM-IR: getelementptr Examples

- ▶ `%r = getelementptr i32, ptr %p, i64 3`



Equivalent in C: `&((int*) p)[3]`

- ▶ `%r = getelementptr {i16, i32}, ptr %p, i64 1, i32 1`



Equivalent in C: `&((struct {short _0; int _1;}*) p)[1]._1`

- ▶ Also works with nested structs and arrays

LLVM-IR: undef and poison

- ▶ `undef` – unspecified value, compiler may choose any value
 - ▶ `%b = add i32 %a, i32 undef → i32 undef`
 - ▶ `%c = and i32 %a, i32 undef → i32 %a`
 - ▶ `%d = xor i32 %b, i32 %b → i32 undef`
 - ▶ `br i1 undef, label %p, label %q → undefined behavior`
- ▶ `poison` – result of erroneous operations
 - ▶ Delay *undefined behavior* on illegal operation until actually relevant
 - ▶ Allows to speculatively “execute” instructions in IR
 - ▶ `%d = shl i32 %b, i32 34 → i32 poison`

LLVM-IR: Intrinsic

- ▶ Not all operations provided as instructions
- ▶ Intrinsic functions: special functions with defined semantics
 - ▶ Replaced during compilation, e.g., with instruction or lib call
- ▶ Benefit: no changes needed for parser/bitcode/... on addition

- ▶ Examples:
 - ▶ `declare iN @llvm.ctpop.iN(iN <src>)`
 - ▶ `declare {iN, i1} @llvm.sadd.with.overflow.iN(iN %a, iN %b)`
 - ▶ `memcpy, memset, sqrt, returnaddress, ...`

LLVM-IR: Tools

- ▶ clang can emit LLVM-IR bitcode

```
clang -O -emit-llvm -c test.c -o test.bc
```

- ▶ llvm-dis disassembles bitcode to textual LLVM-IR

```
clang -O -emit-llvm -c test.c -o - | llvm-dis
```

- ▶ llc compiles LLVM-IR (textual or bitcode) to assembly

```
clang -O -emit-llvm -c test.c -o - | llc
```

```
clang -O -emit-llvm -c test.c -o - | llvm-dis | llc
```

Example Listings omitted – they would span several slides

LLVM-IR: Example

```
define dso_local <4 x float> @foo2(<4 x float> %0, <4 x float> %1) {  
    %3 = alloca <4 x float>, align 16  
    %4 = alloca <4 x float>, align 16  
    store <4 x float> %0, ptr %3, align 16  
    store <4 x float> %1, ptr %4, align 16  
    %5 = load <4 x float>, ptr %3, align 16  
    %6 = load <4 x float>, ptr %4, align 16  
    %7 = fadd <4 x float> %5, %6  
    ret <4 x float> %7  
}
```


LLVM-IR: Example

```
define dso_local i32 @foo3(i32 %0, i32 %1) {  
    %3 = tail call { i32, i1 } @llvm.smul.with.overflow.i32(i32 %0, i32 %1)  
    %4 = extractvalue { i32, i1 } %3, 1  
    %5 = extractvalue { i32, i1 } %3, 0  
    %6 = select i1 %4, i32 -2147483648, i32 %5  
    ret i32 %6  
}
```

LLVM-IR: Example

```
define dso_local i32 @sw(i32 %0) {  
    switch i32 %0, label %4 [  
        i32 4, label %5  
        i32 5, label %2  
        i32 8, label %3  
        i32 100, label %5  
    ]  
2: ; preds = %1  
    br label %5  
3: ; preds = %1  
    br label %5  
4: ; preds = %1  
    br label %5  
5: ; preds = %1, %1, %4, %3, %2  
    %6 = phi i32 [ %0, %4 ], [ 9, %3 ], [ 32, %2 ], [ 12, %1 ], [ 12, %1 ]  
    ret i32 %6  
}
```

LLVM-IR: Example

```
@switch.table.sw = private unnamed_addr constant [7 x i32] [i32 12, i32 32, i32 12,
                                                    i32 12, i32 9, i32 12, i32 12], align 4

define dso_local i32 @sw(i32 %0) {
    %2 = add i32 %0, -4
    %3 = icmp ult i32 %2, 7
    br i1 %3, label %4, label %13

4: ; preds = %1
    %5 = trunc i32 %2 to i8
    %6 = lshr i8 83, %5
    %7 = and i8 %6, 1
    %8 = icmp eq i8 %7, 0
    br i1 %8, label %13, label %9

9: ; preds = %4
    %10 = sext i32 %2 to i64
    %11 = getelementptr inbounds [7 x i32], ptr @switch.table.sw, i64 0, i64 %10
    %12 = load i32, ptr %11, align 4
    br label %13

13: ; preds = %1, %4, %9
    %14 = phi i32 [ %12, %9 ], [ %0, %4 ], [ %0, %1 ]
    ret i32 %14
}
```

LLVM-IR API

- ▶ LLVM offers two APIs: C++ and C
 - ▶ C++ is the full API, exposing nearly all internals
 - ▶ C API is more limited, but more stable
- ▶ Nearly all major versions have breaking changes
- ▶ Some support for multi-threading:
 - ▶ All modules/types/... associated with an LLVMContext
 - ▶ Different contexts may be used in different threads

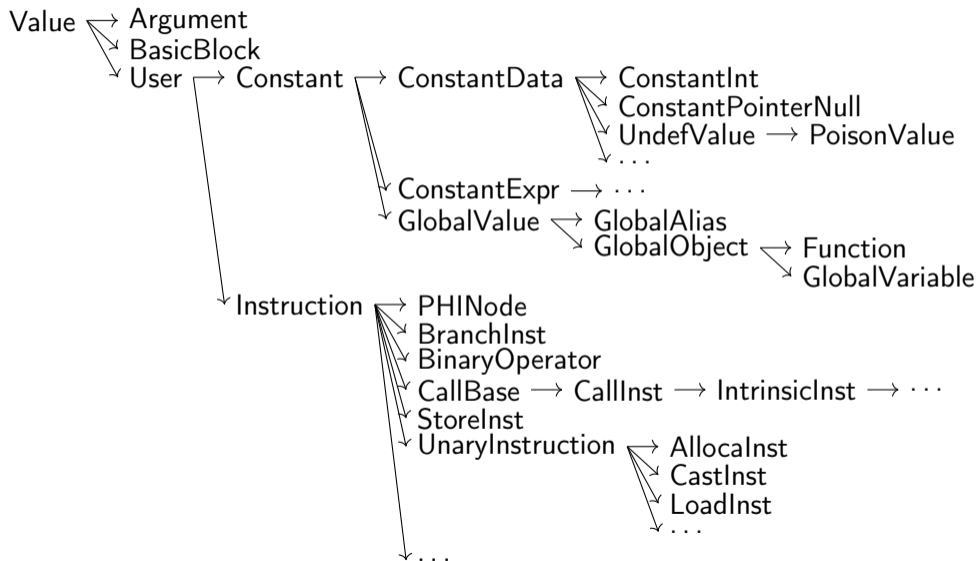
LLVM-IR C++ API: Basic Example

```
#include <llvm/IR/IRBuilder.h>
int main(void) {
    llvm::LLVMContext ctx;
    auto modUP = std::make_unique<llvm::Module>("mod", ctx);

    llvm::Type* i64 = llvm::Type::getInt64Ty(ctx);
    llvm::FunctionType* fnTy = llvm::FunctionType::get(i64, {i64}, false);
    llvm::Function* fn = llvm::Function::Create(fnTy,
        llvm::GlobalValue::ExternalLinkage, "addOne", modUP.get());
    llvm::BasicBlock* entryBB = llvm::BasicBlock::Create(ctx, "entry", fn);

    llvm::IRBuilder<> irb(entryBB);
    llvm::Value* add = irb.CreateAdd(fn->getArg(0), irb.getInt64(1));
    irb.CreateRet(add);
    modUP->print(llvm::outs(), nullptr);
    return 0;
}
```

LLVM-IR API: Almost Everything is a Value... (excerpt)



LLVM-IR API: Programming Environment

- ▶ LLVM implements custom RTTI
 - ▶ `isa<>`, `cast<>`, `dyn_cast<>`
- ▶ LLVM implements a multitude of specialized data structures
 - ▶ E.g.: `SmallVector<T, N>` to keep N elements stack-allocated
 - ▶ Custom vectors, sets, maps; see manual⁸
- ▶ Preferably uses `ArrayRef`, `StringRef`, `Twine` for references
- ▶ LLVM implements custom streams instead of std streams
 - ▶ `outs()`, `errs()`, `dbgs()`

⁸<https://www.llvm.org/docs/ProgrammersManual.html>

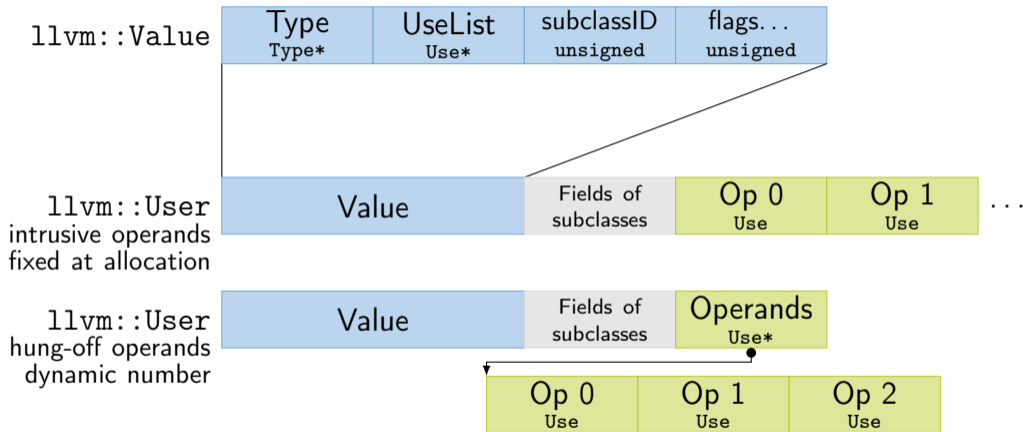
LLVM-IR API: Use Tracking

- ▶ Values track their users

```
llvm::Value* v = /* ... */;  
for (llvm::User* u : v->users())  
    if (auto i = llvm::dyn_cast<llvm::Instruction>(u))  
        // ...
```

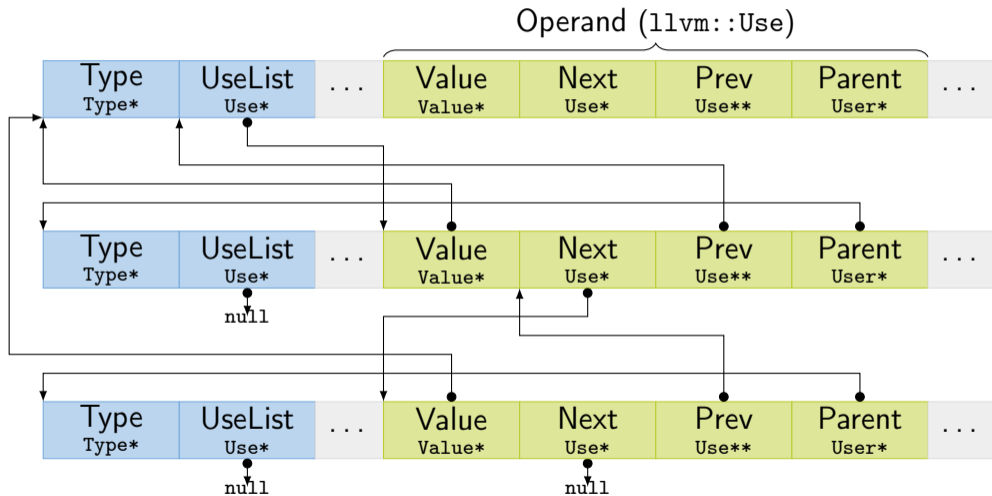
- ▶ Simplifies implementation of analyses
- ▶ Allows for easy replacement:
 - ▶ `inst->replaceAllUsesWith(replVal);`

LLVM IR Implementation: Value/User



PHINode additionally stores n BasicBlock* after the operands, but aren't users of blocks.

LLVM IR Implementation: Use



LLVM IR Implementation: Instructions/Blocks

- ▶ `Instruction` and `BasicBlock` have pointers to parent and next/prev
 - ▶ Linked list updated on changes and used for iteration
 - ▶ Instructions have cached *order* (integer) for fast “comes before”
- ▶ `BasicBlock` successors: blocks used by terminator
- ▶ `BasicBlock` predecessors:
 - ▶ Iterate over users of block – these are terminators (and blockaddress)
 - ▶ Ignore non-terminators, parent of using terminator is predecessor
 - ▶ Same predecessor might be duplicated (\rightsquigarrow `getUniquePredecessor()`)
- ▶ Finding first non- ϕ requires iterating over ϕ -nodes

LLVM and IR Design

- ▶ LLVM provides a decent general-purpose IR for compilers
- ▶ But: not ideal for all purposes
 - ▶ High-level optimizations difficult, e.g. due to lost semantics
 - ▶ Several low-level operations only exposed as intrinsics
 - ▶ IR rather complex, high code complexity
 - ▶ High compilation times
- ▶ Thus: heavy trend towards custom IRs

IR Design: High-level Considerations

- ▶ Define purpose!
- ▶ Structure: SSA vs. something else; control flow
 - ▶ Control flow: basic blocks/CFG vs. structured control flow
 - ▶ Remember: SSA can be considered as a DAG, too
 - ▶ SSA is easy to analyse, but non-trivial to construct/leave
- ▶ Broader integration: keep multiple stages in single IR?
 - ▶ Example: create IR with high-level operations, then incrementally lower
 - ▶ Model machine instructions in same IR?
 - ▶ Can avoid costly transformations, but adds complexity

IR Design: Operations

- ▶ Data types
 - ▶ Simple type structure vs. complex/aggregate types?
 - ▶ Keep relation to high-level types vs. low-level only?
 - ▶ Virtual data types, e.g. for flags/memory?
- ▶ Instruction format
 - ▶ Single vs. multiple results?
 - ▶ Strongly typed vs. more generic result/operand types?
 - ▶ Operand number – fixed vs. dynamic?

IR Design: Operations

- ▶ Allow instruction side effects?
 - ▶ E.g.: memory, floating-point arithmetic, implicit control flow
- ▶ Operation complexity and abstraction
 - ▶ E.g.: `CheckBounds`, `GetStackPtr`, `HashInt128`
 - ▶ E.g.: `load` vs. `MOVQconstidx4`
- ▶ Extensibility for new operations (e.g., new targets, high-level ops)

IR Design: Implementation

- ▶ Maintain user lists?
 - ▶ Simplifies optimizations, but adds considerable overhead
 - ▶ Replacement can use copy and lazy canonicalization
 - ▶ User *count* might be sufficient alternative
- ▶ Storage layout: operation size and locations
 - ▶ For performance: reduce heap allocations, small data structures
- ▶ Special handling for arguments vs. all-instructions?
- ▶ Metadata for source location, register allocation, etc.
- ▶ SSA: ϕ nodes vs. block arguments?

IR Example: Go SSA

- ▶ Strongly typed
 - ▶ Structured types decomposed
- ▶ Explicit memory side-effects
- ▶ Also High-level operations
 - ▶ IsInBounds, VarDef
- ▶ Only one type of value/instruction
 - ▶ Const64, Arg, Phi
- ▶ No user list, but user count
- ▶ Also used for arch-specific repr.

```
env GOSSAFUNC=fac go build test.go
```

```
b1:  
  v1 (?) = InitMem <mem>  
  v2 (?) = SP <uintptr>  
  v5 (?) = LocalAddr <*int> {~r1} v2 v1  
  v6 (7) = Arg <int> {n} (n[int])  
  v8 (?) = Const64 <int> [1] (res[int])  
  v9 (?) = Const64 <int> [2] (i[int])  
Plain -> b2 (+9)  
b2: <- b1 b4  
  v10 (9) = Phi <int> v9 v17 (i[int])  
  v23 (12) = Phi <int> v8 v15 (res[int])  
  v12 (+9) = Less64 <bool> v10 v6  
If v12 -> b4 b5 (likely) (9)  
b4: <- b2  
  v15 (+10) = Mul64 <int> v23 v10 (res[int])  
  v17 (+9) = Add64 <int> v10 v8 (i[int])  
Plain -> b2 (9)  
b5: <- b2  
  v20 (12) = VarDef <mem> {~r1} v1  
  v21 (+12) = Store <mem> {int} v5 v23 v20  
Ret v21 (+12)
```

LLVM and IR Design – Summary

- ▶ LLVM is a modular compiler framework
- ▶ Extremely popular and high-quality compiler back-end
- ▶ Primarily provides optimizations and a code generator
- ▶ Main interface is the SSA-based LLVM-IR
 - ▶ Easy to generate, friendly for writing front-ends/optimizations
- ▶ IR design depends on purpose and integration constraints
- ▶ Structurally similar IRs can strongly differ in capabilities

LLVM and IR Design – Questions

- ▶ What is the structure of an LLVM-IR module/function?
- ▶ Which LLVM-IR data types exist?
How do they relate to the target architecture?
- ▶ How do semantically invalid operations in LLVM-IR behave?
- ▶ What is special about intrinsic functions?
- ▶ How to derive LLVM-IR from C code using Clang?
- ▶ How does LLVM's `replaceAllUsesWith` work?
How could this work without building/maintaining user lists?
- ▶ How can an SSA-based IR make side effects explicit?
- ▶ How would you design an IR for optimizing Brainfuck?