Compilation 2024

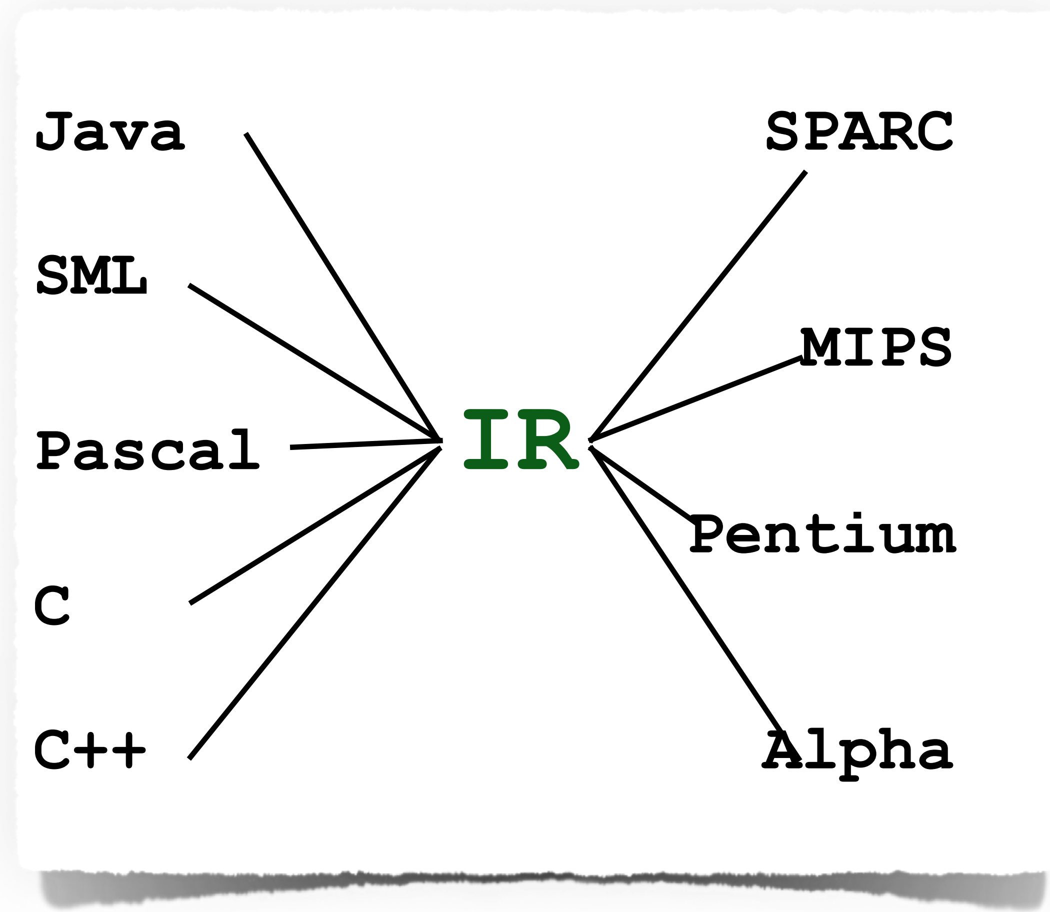
LLVM Intermediate Representation

Amin Timany
timany@cs.au.dk

Partly based on Aslan Askarov’s slides

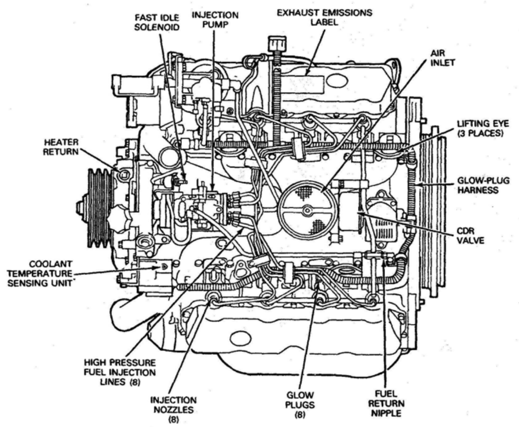
Intermediate Representation

- Translation source/target uses IR as a bridge
- Simplification: n+m combinations, not n×m



Role of IR in Translation

- Compiler frontend:
  Translate source to IR, enforce wellformedness
- Compiler backend:
  Translate IR to assembly, often optimizing
- IR should be “near” both, still independent of source language *and* of machine details
- Source, Target: Incongruent complexity
  IR: clear, simple!



The challenge of designing a (good) IR

- A good IR hides many architecture specific aspects that often vary between architectures but highlights low-level aspects that are common

  - So, what low-level aspects should we then have in the IR?

LLVM Intermediate representation

- Convenient abstraction over many architecture-specific issues that we do not want to deal with yet at this point in the compilation process
  - infinite registers (don’t yet care about the exact number and purpose of registers)
  - direction of the stack growth
  - calling convention
  - exact instruction set

- Exposes low-level aspects important for the code generation
  - notion of a function (and implicitly of the call stack)
  - allocation on stack during function execution (yet agnostic to the actual direction of stack growth)
  - storing and loading into pointers (for both stack and heap-based locations)
  - instructions for arithmetics
  - instructions for jumps, conditional jumps

LLVM Compiler Infrastructure

- LLVM originally stands for Low-Level-Virtual-Machine
- Origins: academic research @UIUC for analysis and optimization of pointer intensive-programs (e.g., C/C++)
  - Needed a compiler and program representation that would make it possible and easy to implement the analysis/optimizations
    - Chris Lattner's PhD thesis (2005)
- LLVM today:
  - not just an IR, but a major compiler infrastructure that continues to evolve
    - Lattner worked for Apple during 2005-2017, where LLVM was brought to production quality; there Lattner also designed Swift; now at Google
  - popular target for many compilers, not just ahead of time compilers, but also JITs

LLVM in our project

- Advantages of using LLVM in our project:
  - Getting a working compiler *early*
    - *fast* compiler thanks to industrial-strength backends
  - Relevance: if you develop a new compiler today, you will be likely targeting LLVM and not a toy IR
  - Arguably, a nicer IR design than in textbook

- We will use a strict subset of LLVM that we call LLVM--
  - based on LLVMLite by S. Zdancewic@UPenn
  - We will first write LLVM programs by hand in this subset
  - Later, we will translate Dolphin programs into LLVM--

LLVM features

- Intermediate assembly-like language
- Infinite number of registers
  - A register can be assigned to only once
    - aka *single-static assignment form,* (SSA)
- Types
  - First-class types:
    - Single-value types:
      - **integer** • float • x86_mmx • **pointer** • vector • label
    - Aggregate types:
      - **arrays** • **structures**
  - **Function** type and **void** type

Identifiers

- Global identifiers: prefixed with @
- Local identifiers: prefixed with %
  - Full LLVM allows for *unnamed* identifiers represented as unsigned numbers %1
  - LLVM-- only allows named identifiers

LLVM memory model

stack frame during function invocation

implicitly allocated stack memory (via use of `%x,%y…` registers)

explicitly allocated stack memory (via `alloca` instruction)

Structure of an LLVM program

- Program:
  - Global declarations – a list of globals, (e.g., string literals)
  - Types: a list of named types
  - Function declarations
- Function declaration
  - Header (params, ret ty)
  - Control Flow Graph
- Control Flow Graph
  - Entry basic block + labeled basic blocks
- Basic block: list of instructions + terminator
- Terminator: return, branch/conditional branch instruction, or unreachable

LLVM-- Instructions: Basics

- Binary operations: a = 10 - (y + z)
  - `%x = add i64 %y, %z            ; x=y+z`
  - `%a = sub i64 10, %x            ; a=10-x`
  - Also: **`mul`**, **`sdiv`**, **`shl`**, **`lshr`**, etc.

- Comparison of integers or pointers
  - `%x = icmp ne i64 %y, 0       ; x=y<>0`
  - `%a = icmp ult i64 %b, c      ; a=b<c`
  - Also: **`eq`**, **`slt`**, etc.

LLVM-- Instructions: Call and Memory

Note @ sign in function name

- Function call
  - `%x = call i8* @foo (i8* null, i64 %y)`
- Allocation on the stack – returns a pointer
  - `%ptr = alloca i64  ; type of ptr is i64*`
- Store to the memory location given by a pointer
  - `store i64 0, i64* %ptr`
- Load from the memory location given by a pointer
  - `%x = load i64, i64* %ptr ; type of x is i64`
- Note that pointers appearing in **`load/store`** can be obtained via **`alloca`** but also via heap allocation through runtime

LLVM-- Instructions: terminators

- Conditional branch:
  - **`br`** `i1 %x, label %L1, label %L2`
- Unconditional branch
  - **`br`** `label %L3`
- Return
  - **`ret`** `i64 5`
  - **`ret`** `void`
- End of basic block is unreachable
  - **`unreachable`**

Types in LLVM--

- Integer types: `i1, i8, i64`
  - `i1` is used for branching instructions
  - `i8` is used for arbitrary pointers, i.e., `i8*`
- Pointer type: `<type name>*`
- Structures, fixed-sized arrays, named types
  - We will talk about this when we translate from Dolphin
- Type conversion:
  - **`bitcast/ptrtoint`**

LLVM basic blocks

```c
int factorial(int X) {
  if (X == 0) return 1;
  return X*factorial(X-1);
}
```

C source

basic blocks

```llvm
define i32 @factorial(i32 %X) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %X, i32* %2, align 4
  %3 = load i32* %2, align 4
  %4 = icmp eq i32 %3, 0
  br i1 %4, label %5, label %6

; <label>:5                                       ; preds = %0
  store i32 1, i32* %1
  br label %12

; <label>:6                                       ; preds = %0
  %7 = load i32* %2, align 4
  %8 = load i32* %2, align 4
  %9 = sub nsw i32 %8, 1
  %10 = call i32 @factorial(i32 %9)
  %11 = mul nsw i32 %7, %10
  store i32 %11, i32* %1
  br label %12

; <label>:12                                      ; preds = %6, %5
  %13 = load i32* %1
  ret i32 %13
}
```