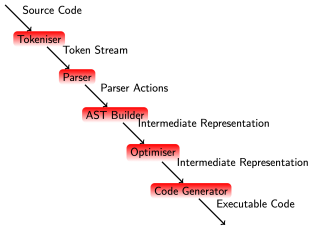
CS 131 Compilers: Discussion 6: Introduction to Modern Intermediate Representation

杨易为 季杨彪 尤存翰
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 4 月 5 日

1 Reusable IR

1. Modern compilers are made from loosely coupled components.
2. Front ends produce IR, AST as well. Thus they have to maintain all the information on Basic Block.
3. Middle 'ends' transform IR (optimisation / analysis / instrumentation)
4. Back ends generate native code (object code or assembly)



As with any other piece of software using libraries simplifies development.

1.1 Optimisation Passes

1. Modular, transform IR (Analysis passes just inspect IR)
2. Can be run multiple times, in di erent orders
3. May not always produce improvements when run in the wrong order!
4. Some intentionally pessimise code to make later passes work better

1.2 Register vs Stack IR

1. Stack makes interpreting, naive compilation easier • Register makes various optimisations easier
2. Which ones?

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b
r2 = load c
r3 = r1 + r2
r4 = load b
r5 = load c
r6 = r4 + r5
r7 = r3 * r6
store a r6
```

Source language:

```
a = (b+c) * (b+c);
```

```
load b
load c
add
load b
load c
add
mul
store a
```

1.2.1 Problems with CSE and Stack IR

1. Entire operation must happen at once (no incremental algorithm)
2. Finding identical subtrees is possible, reusing results is harder • If the operations were not adjacent, must spill to temporary

1.2.2 Hierarchical vs Flat IR

1. Source code is hierarchical (contains structured flow control, scoped values)
2. Assembly is flat (all flow control is by jumps)
3. Intermediate representations are supposed to be somewhere between the two
4. Think about the possible ways that a for loop, while loop, and if statement with a backwards goto might be represented.

1.2.3 Hierarchical IR

1. Easy to express high-level constructs
2. Preserves program semantics
3. Preserves high-level semantics (variable lifetime, exceptions) clearly
4. Example: Ark Compiler, Flamming my compiler

1.3 Flat IR

1. Easy to map to the back end
2. Simple for optimisations to process
3. Must carry scope information in ad-hoc ways (e.g. LLVM IR has intrinsics to explicitly manage lifetimes for stack allocations)
4. Examples: LLVM IR, CGIR, PTX

2 What Is LLVM IR?

1. Unlimited Single-Assignment Register machine instruction set • Strongly typed
2. Three common representations:
3. Human-readable LLVM assembly (.ll files)
4. Dense ‘bitcode’ binary representation (.bc files) • C++ classes

2.1 Unlimited Register Machine?

1. Real CPUs have a fixed number of registers
2. LLVM IR has an infinite number
3. New registers are created to hold the result of every instruction
4. CodeGen’ s register allocator determines the mapping from LLVM registers to physical registers
5. Type legalisation maps LLVM types to machine types and so on (e.g. 128-element float vector to 32 SSE vectors or 16 AVX vectors, 1-bit integers to 32-bit values)

2.2 Static Single Assignment

1. Registers may be assigned to only once
2. Most (imperative) languages allow variables to be... variable
3. This requires some e ort to support in LLVM IR: SSA registers are not variables
4. SSA form makes dataflow explicit: All consumers of the result of an instruction read the output register(s)

Example:

```
int a = someFunction();
a++;
```

- One variable, assigned to twice.

```
%a = call i32 @someFunction()
%a = add i32 %a, 1
```

```
error: multiple definition of local value named 'a'
  %a = add i32 %a, 1
    ^
```

```
%a = call i32 @someFunction()
%a2 = add i32 %a, 1
```

- Front end must keep track of which register holds the current value of a at any point in the code
- How do we track the new values?

2.3 Translating to LLVM IR The Easy Way

```
; int a
%a = alloca i32, align 4
; a = someFunction
%0 = call i32 @someFunction()
store i32 %0, i32* %a
; a++
%1 = load i32* %a
%2 = add i32 %1, 1
store i32 %2, i32* %a
```

1. Numbered register are allocated automatically
2. Each expression in the source is translated without worrying about data flow
3. Memory is not SSA in LLVM

2.3.1 Isn' t That Slow?

- Lots of redundant memory operations
- Stores followed immediately by loads
- The Scalar Replacement of Aggregates (SROA) or mem2reg pass cleans it up for us

```
%0 = call i32 @someFunction()
%1 = add i32 %0, 1
```

Important: SROA only works if the `alloca` is declared in the entry block to the function!

2.4 Sequences of Instructions

1. A sequence of instructions that execute in order is a basic block
2. Basic blocks must end with a terminator
3. Terminators are intraprocedural flow control instructions.

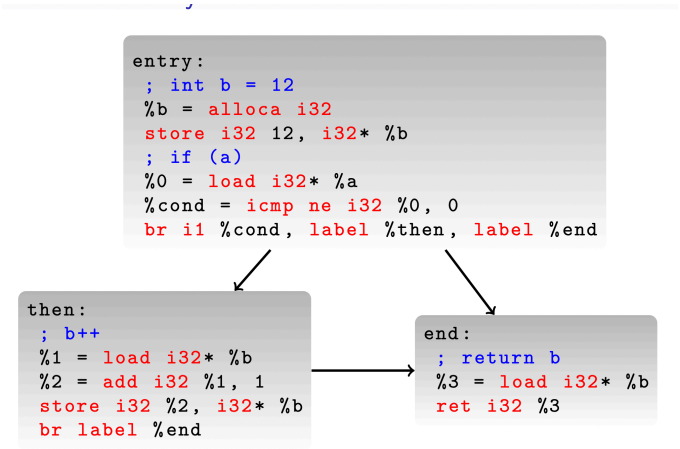
4. call is not a terminator because execution resumes at the same place after the call
5. invoke is a terminator because flow either continues or branches to an exception cleanup handler
6. This means that even "zero-cost" exceptions can have a cost: they complicate the control-flow graph (CFG) within a function and make optimisation harder.

2.5 Intraprocedural Flow Control

1. Assembly languages typically manage flow control via jumps / branches (often the same instructions for inter- and intraprocedural flow)
2. LLVM IR has conditional and unconditional branches
3. Branch instructions are terminators (they go at the end of a basic block)
4. Basic blocks are branch targets
5. You can't jump into the middle of a basic block (by the definition of a basic block)

2.6 'Phi, my lord, phi!' - Lady Macbeth, Compiler Developer

1. ϕ nodes are special instructions used in SSA construction
2. Their value is determined by the preceding basic block
3. ϕ nodes must come before any non- ϕ instructions in a basic block
4. In code generation, ϕ nodes become a requirement for one basic block to leave a value in a specific register. Alternate representation: named parameters to basic blocks



2.7 Why Select?

x86:

```
testl %edi, %edi
setne %al
movzbl %al, %eax
orl $12, %eax
ret
```

ARM:

```
mov r1, r0
mov r0, #12
cmp r1, #0
movne r0, #13
mov pc, lr
```

PowerPC:

```
cmplwi 0, 3, 0
beq 0, .LBB0_2
li 3, 13
blr
.LBB0_2:
li 3, 12
blr
```

2.8 Functions

- LLVM functions contain at least one basic block
- Arguments are registers and are explicitly typed
- Registers are valid only within a function scope

```
@hello = private constant [13 x i8] c"Hello
    world!\00"

define i32 @main(i32 %argc, i8** %argv) {
entry:
  %0 = getelementptr [13 x i8]* @hello, i32 0,
      i32 0
  call i32 @puts(i8* %0)
  ret i32 0
}
```

2.9 The Most Important LLVM Classes

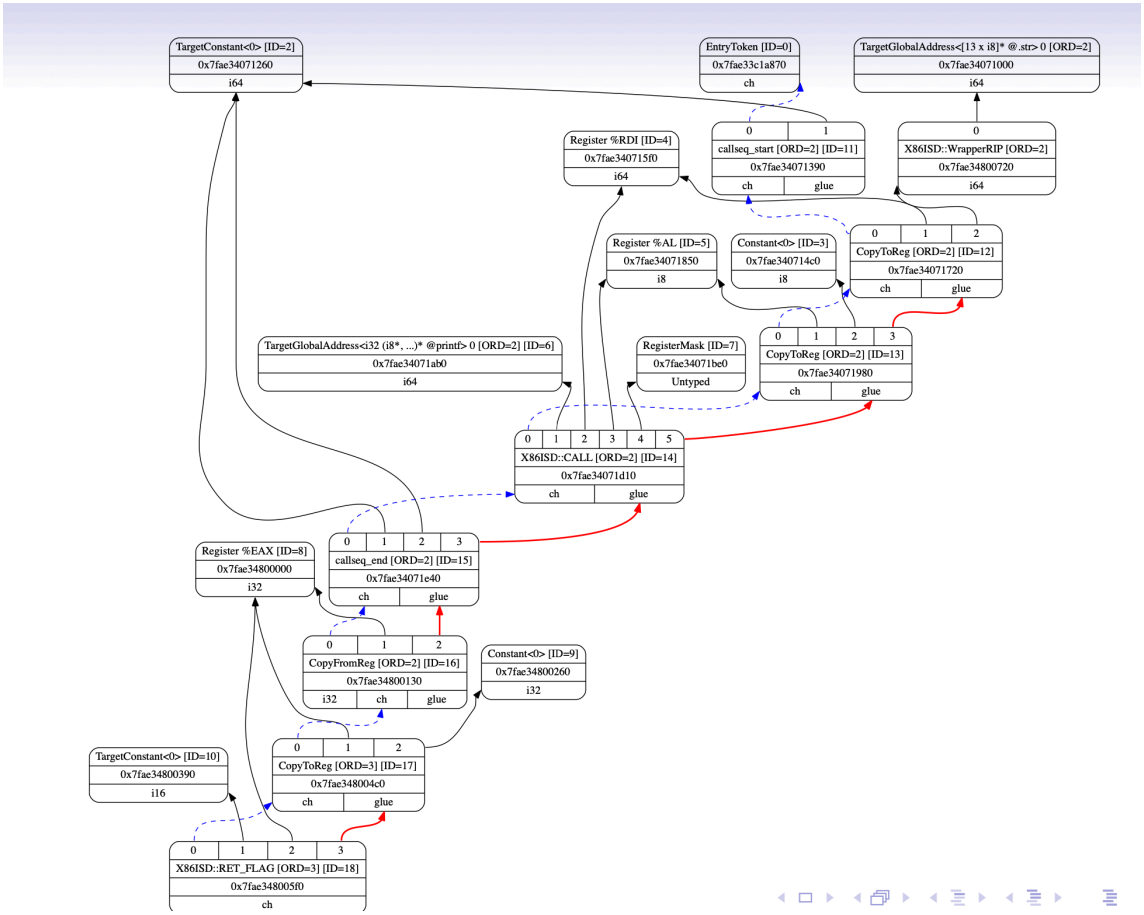
1. Module - A compilation unit.
2. Function - Can you guess?
3. BasicBlock - a basic block
4. GlobalVariable (I hope it' s obvious)
5. IRBuilder - a helper for creating IR
6. Type - superclass for all LLVM concrete types
7. ConstantExpr - superclass for all constant expressions
8. PassManagerBuilder - Constructs optimisation pass sequences to run
9. ExecutionEngine - Interface to the JIT compiler

2.10 Writing A Simple Pass

1. Memoise an expensive library call
2. Call maps a string to an integer (e.g. string intern function) • Mapping can be expensive.
3. Always returns the same result.

2.11 Selection DAG

1. DAG defining operations and dependencies
2. Legalisation phase lowers IR types to target types
   (a) Arbitrary-sized vectors to fixed-size
   (b) Float to integer and softfloat library calls
3. DAG-to-DAG transforms simplify structure
4. Code is still (more or less) architecture independent at this point
5. Some peephole optimisations happen here


TargetConstant<0> [ID=2]
0x7fae34071260
i64
EntryToken [ID=0]
0x7fae33c1a870
ch
TargetGlobalAddress<[13 x i8]* @.str> 0 [ORD=2]
0x7fae34071000
i64
0 1
callseq_start [ORD=2] [ID=11]
0x7fae34071390
ch glue
0
X86ISD::WrapperRIP [ORD=2]
0x7fae34800720
i64
Register %RDI [ID=4]
0x7fae340715f0
i64
Register %AL [ID=5]
0x7fae34071850
i8
Constant<0> [ID=3]
0x7fae340714c0
i8
0 1 2
CopyToReg [ORD=2] [ID=12]
0x7fae34071720
ch glue
TargetGlobalAddress<i32 (i8*, ...)* @printf> 0 [ORD=2] [ID=6]
0x7fae34071ab0
i64
RegisterMask [ID=7]
0x7fae34071be0
Untyped
0 1 2 3
CopyToReg [ORD=2] [ID=13]
0x7fae34071980
ch glue
0 1 2 3 4 5
X86ISD::CALL [ORD=2] [ID=14]
0x7fae34071d10
ch glue
Register %EAX [ID=8]
0x7fae34800000
i32
0 1 2 3
callseq_end [ORD=2] [ID=15]
0x7fae34071e40
ch glue
0 1 2
CopyFromReg [ORD=2] [ID=16]
0x7fae34800130
i32 ch glue
Constant<0> [ID=9]
0x7fae34800260
i32
TargetConstant<0> [ID=10]
0x7fae34800390
i16
0 1 2
CopyToReg [ORD=3] [ID=17]
0x7fae348004c0
ch glue
0 1 2 3
X86ISD::RET_FLAG [ORD=3] [ID=18]
0x7fae348005f0
ch