LLVM and IR Construction

Fabian Ritter

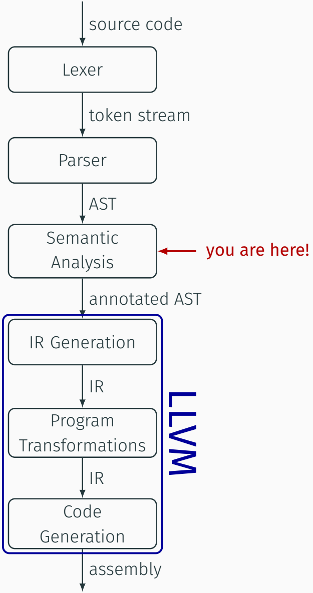
based on slides by Christoph Mallon and Johannes Doerfert

http://compilers.cs.uni-saarland.de

Compiler Design Lab
Saarland University

SIC Saarland Informatics Campus

Project Progress



source code

Lexer

token stream

Parser

AST

Semantic Analysis

you are here!

annotated AST

IR Generation

IR

Program Transformations

IR

Code Generation

LLVM

assembly

LLVM

- open source
- large, active research community
- used in industry:

  Apple, Google, Intel, NVIDIA, Sony, ...

  knowing LLVM might be helpful on your CV!
- front-ends for many languages:

  C/C++, Fortran, Rust, Swift, Julia, Haskell, ...
- back-ends for many architectures:

  X86(-64), ARM/AArch64, MIPS, WebAssembly, ...
- it's HUGE

Getting LLVM

We use **LLVM 5.0.0**.

- Build it yourself: `./build_llvm.sh`

  pros: same as on the test server, RTTI enabled, debug build
  cons: requires time and a strong system:
  > 4 GB RAM, ~15 GB HDD (including clang)

- Build it with a modified build script:

  e.g. replace `Debug` build type with `Release`, add `clang`

- Get binaries from the website:

  `http://releases.llvm.org/download.html#5.0.0`

  (and add its `bin` folder to the `PATH` environment variable)

- From package manager/pre-installed: **not recommended!**

  cons: possibly wrong version, vendor modified, no RTTI...

LLVM Intermediate Representation

- SSA-based representation of control flow graphs
- dumpable in human-readable, assembly-like form (`*.ll`)
- dumpable as compact bitcode (`*.bc`)

Instructions

```
%sum = add i32 4, %var ; Binary operations
%cmp = icmp sge i32 %a, %b

%value = load i32 , i32* %location ; Memory operations
store i32 %value, i32* %location
%ptr = alloca i32

br label %next-block ; Terminator Instructions
br i1 %cmp, label %then-block , label %else-block
ret i32 %a

%X = trunc i32 257 to i8 ; Cast Instructions
%Y = sext i32 %V to i64
%Z = bitcast i8* %x to i32*

%ret = call i32 @foo(i8* %fmt, i32 %val) ; Other Instructions
%phi = phi i32 [ %value-a, %block-a ], [ %value-b, %block-b ]
%l-th-element-addr = getelementptr i32 , i32* %p, i64 %l
```

- create using `IRBuilder<>::Create...(...)`
- consider the instruction reference for details[1,2]

[1] https://llvm.org/docs/LangRef.html#instruction-reference

[2] https://llvm.org/docs/GetElementPtr.html

Types

- machine integer type: `i8`, `i32`,..., `i<N>`

  sign agnostic, interpretation depends on instructions

  (nuw/nsw, udiv/sdiv,...)

  create using `IntegerType::get(...)` (if necessary)
- pointer types: `<Ty>*`

  void pointers do not exist, use `i8*` instead

  create using `PointerType::getUnqual(...)`
- structure types: `{ <Ty1>, <Ty2>, <...> }`

  members don't have names, only indices

  create using `StructType::Create(...)`
- function types: `<Ty> (<Ty1>, <Ty2>, <...>)`

  create using `FunctionType::Create(...)`

Basic Blocks

- contain a list of instructions:
  - 0 or more PHINodes
  - 0 or more non-terminator, non-phi instructions
  - exactly 1 terminator instruction
- know their predecessors and successors
- create using `BasicBlock::Create(...)`

```
while-header:
  %.01 = phi i32 [ %n, %entry ], [ %1, %while-body ]
  %.0 = phi i32 [ 1, %entry ], [ %0, %while-body ]
  %while-condition = icmp ne i32 %.01, 0
  br i1 %while-condition, label %while-body, label %while-end
```

Functions

- have parameters and a return type
- contain a list of basic blocks
- declarations are functions without basic blocks
- create using `Function::Create(...)`

```
define i32 @fac(i32 %n) {
   ...
}
```

Global Variables

- constant pointers to modifiable memory locations
- accessed only via load/store
- create using its constructor

```
@fortytwo = global i32 42
```

Modules

- correspond to translation units
- contain function definitions/declarations, globals, struct types
- create using its constructor with an `LLVMContext`

LLVM Intermediate Representation — Example

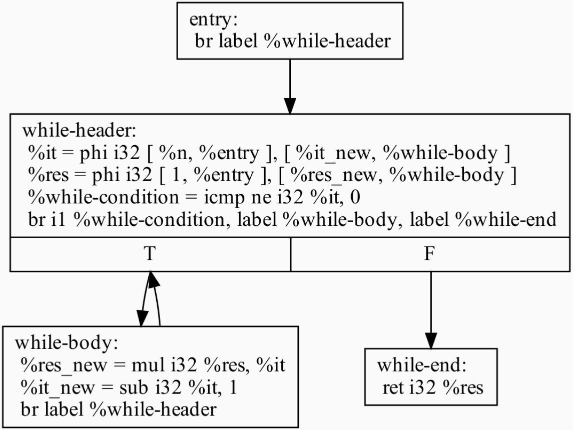
```
define i32 @fac(i32 %n) {
entry:
  br label %while-header

while-header:
  %it = phi i32 [ %n, %entry ], [ %it_new, %while-body ]
  %res = phi i32 [ 1, %entry ], [ %res_new, %while-body ]
  %while-condition = icmp ne i32 %it, 0
  br i1 %while-condition, label %while-body, label %while-end

while-body:
  %res_new = mul i32 %res, %it
  %it_new = sub i32 %it, 1
  br label %while-header

while-end:
  ret i32 %res
}
```

LLVM Intermediate Representation — Example



```
entry:
 br label %while-header
```

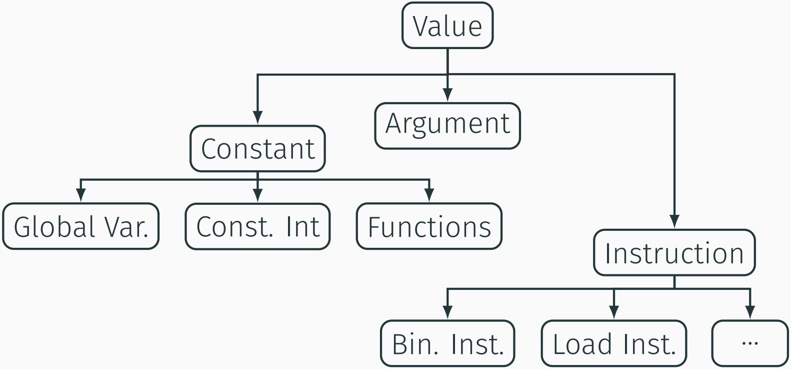
```
while-header:
 %it = phi i32 [ %n, %entry ], [ %it_new, %while-body ]
 %res = phi i32 [ 1, %entry ], [ %res_new, %while-body ]
 %while-condition = icmp ne i32 %it, 0
 br i1 %while-condition, label %while-body, label %while-end
```

```
while-body:
 %res_new = mul i32 %res, %it
 %it_new = sub i32 %it, 1
 br label %while-header
```

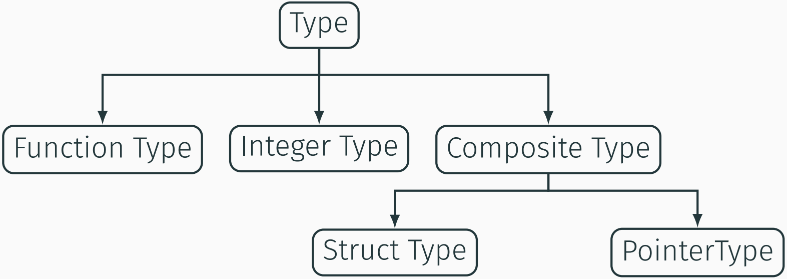
```
while-end:
 ret i32 %res
```

CFG for 'fac' function

LLVM API — Inheritance Diagrams



LLVM API — Inheritance Diagrams



LLVM IR and SSA Form

How to directly generate IR in SSA form?

Don’t! :)

Only `Value`s (“virtual registers”/“variables”) are in SSA form.

Use `alloca`s in the entry basic block to get stack slots for variables and load/store them as required.

Later, use LLVM’s `mem2reg` pass to promote these variables to registers.

Useful Commands

- Generate (human readable) LLVM-IR from C/C++ input:
  ```
  clang -emit-llvm -c -S -o OUT.ll IN.c
  ```
  requires `clang`
- Draw CFG of function `foo` from dumped LLVM-IR module:
  ```
  opt -dot-cfg IN.ll; dot -Tpdf cfg.foo.dot > OUT.pdf
  ```
  requires `dot`/`graphviz`
- Execute dumped LLVM-IR module:
  ```
  lli IN.ll <argv arguments>
  ```
- Create binary from dumped LLVM-IR module:
  ```
  clang -o OUT IN.ll
  ```
  requires `clang`
- Create architecture specific assembly:
  ```
  llc -o OUT.s IN.ll
  ```
- Create binary from architecture specific assembly:
  ```
  cc -o OUT IN.s
  ```
- Get more help:
  ```
  <TOOL> --help
  ```

Getting Help

- General language reference manual:

  `http://llvm.org/docs/LangRef.html`

- Doxygen code documentation:

  (well accessible via Google/Bing/DuckDuckGo/...)

  `http://llvm.org/doxygen/index.html`

- Full command line tools guide:

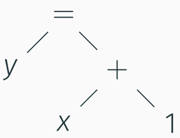
  `http://llvm.org/docs/CommandGuide/`

- Ask in our forum!

IR Construction

Examples

Code Generation for Expressions



- Do not evaluate expression
- Create code, which, when run, evaluates the expression
- IR construction is code generation, just for a virtual machine
- Recursively create code for expressions
- Create code for operands, then create code for current node
- Same order as evaluating, but generating code instead

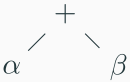
Code Generation for a Constant

1

```
virtual Value* Expression::makeRValue();
```

```
virtual Value* Constant::makeRValue() {
  return createConstantNode(value);
}
```

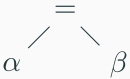
Code Generation for $+$



- Generate code for operands
- Then generate code for $+$

```
virtual Value* Addition::makeRValue() {
  l = left->makeRValue();
  r = right->makeRValue();
  return createAddNode(l, r);
}
```

Code Generation for =



- L-value: address of the object denoted by an expression
- R-value: value of an expression
- L and R stand for left and right hand side (of assignment)
- Assignment happens as side effect of the expression

```
virtual Value* Assignment::makeRValue() {
  address = left->makeLValue();
  value   = right->makeRValue();
  createStoreNode(address, value);
  return value;
}
```

Code Generation for $*$ (Indirection)

- R-value of $*\alpha$ is the value loaded from the address denoted by the R-value of α
- Address of the object denoted by $*\alpha$ is the value of α: L-value of $*\alpha$ is the R-value of α

```
virtual Value* Indirection::makeRValue() {
  address = operand->makeRValue();
  return createLoadNode(address);
}

virtual Value* Indirection::makeLValue() {
  return operand->makeRValue();
}
```

Code Generation for & (Address)

- Value of $\&\alpha$ is the address of the object denoted by α: R-value of $\&\alpha$ is the L-value of α
- $\&\alpha$ does not denote an object: $\&\alpha$ is not an L-value

```
virtual Value* Address::makeRValue() {
  return operand->makeLValue();
}

virtual Value* Address::makeLValue() {
  PANIC("invalid L-value");
}
```

Connection between L-Value and R-Value

- R-value is just loading from L-value
- Unfortunately most expressions are not an L-value, i.e. do not denote an object

```
virtual Value* Expression::makeRValue() {
  address = makeLValue();
  return createLoadNode(address);
}

virtual Value* Expression::makeLValue() {
  PANIC("invalid L-value");
}
```

Different Code Generation in Different Contexts

```
expr = ...  /* L-value */
...  = expr /* R-value */
if (expr)   /* Control flow */
```

- Code generated depends on context, where the expression appears
- L-value: address of the object denoted by an expression
- R-value: value of an expression
- Control Flow: Branch depending on result of an expression
- Different contexts call each other recursively for operands

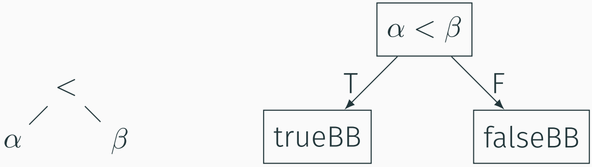
Control-Flow Code Generation for Condition

```
if (C) S1 else S2
```

- If C evaluates to $\neq 0$ continue at S1
- Otherwise continue at S2
- Label/Basic block of S1 and S2 are input for code generation

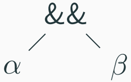
```
virtual void Expression::makeCF(trueBB, falseBB);
```

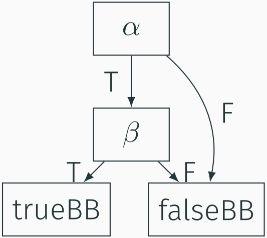
Control-Flow Code Generation for $<$



```
virtual void LessThan::makeCF(trueBB, falseBB) {
  l    = left->makeRValue();
  r    = right->makeRValue();
  cond = createCmpLessThanNode(l, r);
  createBranch(trueBB, falseBB, cond);
}
```

Control-Flow Code Generation for &&

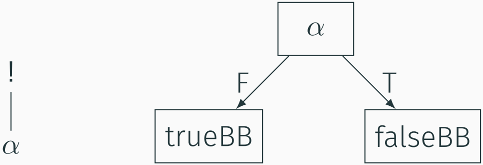




- Lazy evaluation: β might have side effects
- Stop evaluation if value of left hand side determines result

```
virtual void LogicalAnd::makeCF(trueBB, falseBB) {
  extraBB = createBasicBlock();
  left->makeCF(extraBB, falseBB);
  setCurrentBB(extraBB);
  right->makeCF(trueBB, falseBB);
}
```

Control-Flow Code Generation for !

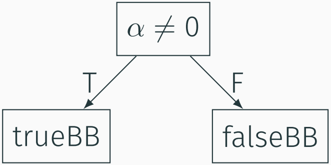


- To negate the condition, just swap the targets

```
virtual void LogicalNegation::makeCF(trueBB, falseBB) {
  operand->makeCF(falseBB, trueBB);
}
```

Control-Flow Code Generation for Arbitrary Expression

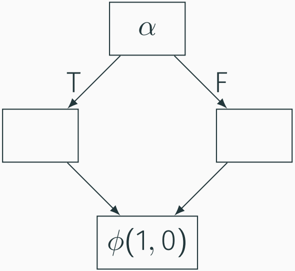
α



- Test R-value $\neq 0$

```
virtual void Expression::makeCF(trueBB, falseBB) {
  PANIC("implement this");
}
```

R-value Code Generation for Control Flow Expression

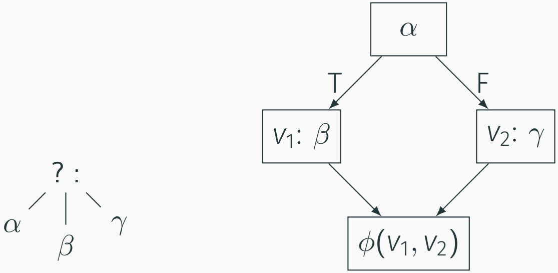


α

- Control flow operators produce 1 or 0
- Select the value depending on whether the true or false basic block was reached

```
virtual Value* ControlFlowExpression::makeRValue() {
  PANIC("implement this");
}
```

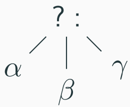
R-value Code Generation for Conditional Expression

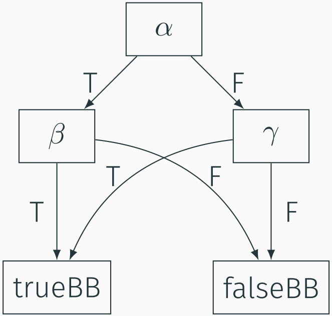


- First evaluate condition α to control flow
- Then either evaluate consequence β or alternative γ
- Pick result using a ϕ

```
virtual Value* ConditionalExpression::makeRValue() {
  PANIC("implement this");
}
```

Control-Flow Code Generation for Conditional Expression





- First evaluate condition α to control flow
- Then either evaluate consequence β or alternative γ to control flow

```
virtual void ConditionalExpression::makeCF(trueBB, falseBB) {
  PANIC("implement this");
}
```

Keep it simple!