Compiler Construction

Sibylle Schupp[1]

[1]Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

SoSe16

Lecture 7

STS
Software
Technology
Systems
TUHH

Outline

Motivation

Recall the notion of an abstract syntax tree (AST).

- An AST is a *representation* of a program.
  - Not the same as the original program, obviously.
  - Leaves out some parts, but also adds information.
- Other characteristics:
  - An AST is close to source code.
  - It is suitable for a source-to-source transformation or an early compiler phase.
  - It suggests tree-based algorithms.

What is an intermediate representation?

An *intermediate representation* (IR, "intermediate language") of a program is a format of the program that exhibits information relevant for other algorithms ("clients") that operate on the program.

- A compiler typically uses several intermediate representations.
- Most compilation passes operate on a particular IR and transform only programs given in that IR.
- The design of an IR is tailored towards the compilation phase and aims at efficient access to the necessary information.

Every software analysis requires one or more appropriate IRs, not just a compiler.

Kinds of intermediate representation

There are two major categories of IRs

- Tree- and graph-based IRs
  - Graphical representation
  - Algorithms: tree or graph traversals
- Linear IRs
  - Text-based representation
  - Algorithms: linear, along the text

How to design an IR?

- What is the relevant information for a compilation pass or client?
  - Program flow, dependencies, addresses, naming, instructions,...
- How to represent that information appropriately?
  - Directed edges, offset computation, variable renaming, ...
- How to get the information?
  - Some information exists implicitly and must only be made explicit. Ex.: program flow
  - Other information must be generated ("synthesized"). Ex.: names for temporaries, mangled names

For standard compilation tasks, standard IRs exist. For new software analyses, you might need to devise your own variant of an IR.

Outline

Selected IRs

- Control-flow graph; basic blocks; dominators
- Call graph
- Stack-machine code
- Three-address code (TAC)
- Static single assignment (SSA), next time

Control-flow graph

An *(intraprocedural) control-flow graph (CFG)* is a directed graph $G = (N, E)$ where N is the set of basic blocks and E the possible paths of execution.

- "Intraprocedural" means that the CFG represents a single procedure or method; the term is often omitted.
- For programs with loops, the CFG is cyclic.
- In practice, CFGs often have two extra, artificial nodes that mark a unique entry and exit of a procedure.
- Outside of compiler construction, nodes could also be single statements (not basic blocks).

CFGs have many clients: optimizations, instruction scheduling, register allocation, ...

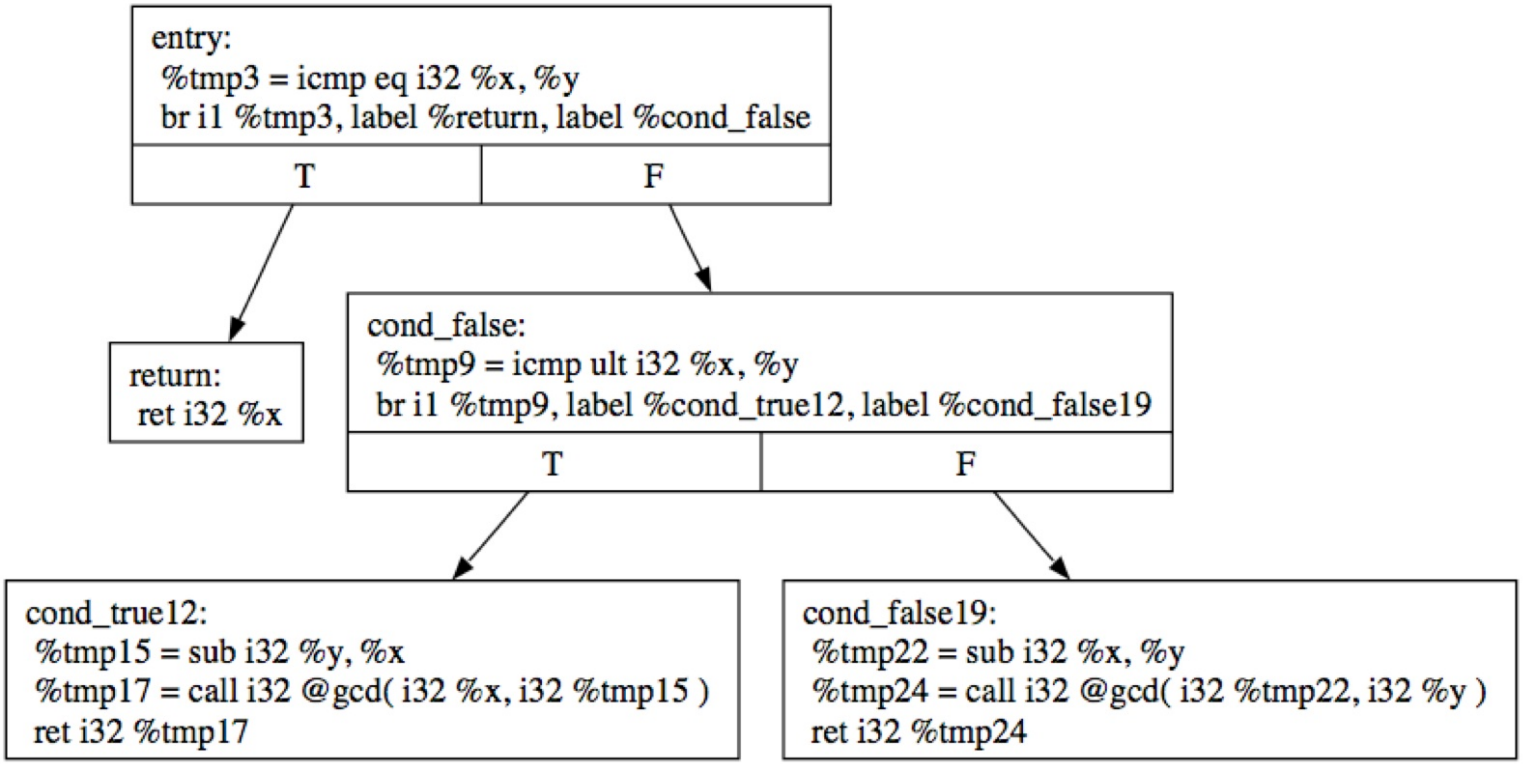
Basic blocks

A *basic block* is a sequence of statements where always either all or none of its statements are executed.

- Basic blocks (BB) can be defined for source code as well as for assembly code.
- Execution enters the basic block at the first statement and leaves it the last statement.
- The last statement of a basic block is either a test or a jump-like instruction.

Example (control-flow graph)

http://llvm.org/releases/2.6/docs/tutorial/JITTutorial2.html




```
unsigned gcd(unsigned x, unsigned y) {
  if(x == y) {
    return x;
  } else if(x < y) {
    return gcd(x, y - x);
  } else {
    return gcd(x - y, y);
  }
```

STS Software Technology Systems TUHH

History (control-flow graph)

Francis Allen, doi:10.1145/800028.808479

SIGPLAN Notices 1970 July

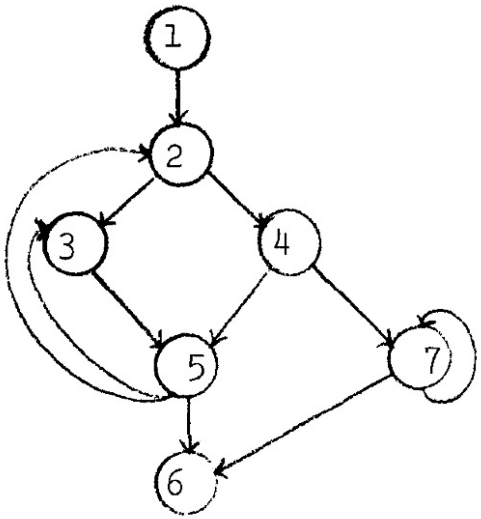
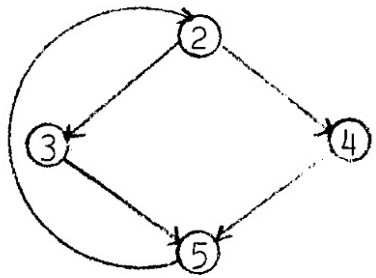


Fig. 1

One of the many subgraphs in G is G' = (B',E') in which B' = {2,3,4,5} and E' = {(2,3),(2,4),(3,5)(4,5)(5,2)}. G' can be depicted by:



A path in a directed graph is a directed subgraph, P, of ordered nodes and edges obtained by successive applications of the successor function. It is expressed as a sequence of nodes $(b_1,b_2,\ldots,b_n)$ where $b_{i+1} \epsilon \Gamma_P^1(b_i)$. The edges are implied: $(b_i,b_{i+1}) \epsilon E$. The nodes and the implied edges are not necessarily unique. A path in G, the graph in Fig. 1, is (2,3,5,3,5,2,4). It should be observed that the examples show how some of the developed notation is to be used: the nodes of the graph are arbitrarily but uniquely named and b stands for any such name.

STS Software Technology Systems TUHH

Call graphs

*Inter*procedural control flow can be represented through *call graphs*.

- A *call graph* is a directed graph $G = (N, E)$ where N denotes a set of subroutines or methods and E denotes caller-callee relation.
- Call graph construction is much more complicated than CFG construction.
  - Dynamic dispatch, polymorphism, overloading
  - Program analysis necessary ("call graph analysis") for precise call-graph construction

  Accordingly, several variations exist.

Clients of call-graph construction: devirtualization, optimizations

Dominator tree

Given a CFG $G = (N, E)$ with start node *entry*.

- A node *d dominates* a node *n* in G if every path from *entry* to *n* must go through *d*.
- Let *D* denote the set of nodes that strictly dominate *n*, i.e., $n \notin D$. The *immediate dominator* of *n*, idom(n), is the node in *D* that is closest to *n*.
  - In a connected graph, one can prove that a node has not more than one immediate dominator.
  - The immediate dominator is not necessarily the predecessor in the CFG.
- The *dominator tree* is a tree containing all nodes of the CFG and for every node *n* an edge from idom(n) to *n*.

Clients: loop optimization, SSA

Stack-machine code

Stack-machine code is a linear IR, based on one-address codes.

- It contains only instructions. Each instruction has not more than one operand.
- Arguments are kept on the stack.
- An instruction takes its arguments from the stack and pushes the result back onto the stack.
- Prominent examples: JVM (Java Virtual Machine) and .NET byte code, which is then interpreted.

Example (JVM representation of 2 + (3 *11))

```
bipush 2     ; 2
bipush 3     ; 2 3
bipush 11    ; 2 3 11
imul         ; 2 33
iadd         ; 35
```

Three-address code

An alternative linear IR is *three-address code*, which is a sequence of instructions that bind their result to a temporary:

- One operator, return value, not more than two operands
- Mostly used as abstraction from RISC code
- Requires the introduction of new identifiers (for temporaries)

Example (pseudo-LLVM representation of x+y*z)

```
%5 = mul %y %z
%7 = add %x %5
```

Outline

LLVM assembly language

http://www.llvm.org

The LLVM compiler infrastructure aims at support for all compilation phases and compilation-related tasks. It centers around an assembly language, also called LLVM.

Goals

- Universal intermediate language
  - Language- and target-independent
  - Framework for compiler and compiler tools
- Suitable for all compilation and optimization phases
  - High-level information expressible
  - Arrays, index, functions, ...
- Type-safe
- Formal semantics
  - Well-formedness
  - Internal checks by LLVM

TUHH STS Software Technology Systems

Major language parts

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

- High-level constructs
  - Modules, visibility, "structs", garbage collection, ...
- Type system
  - void, "single value", aggregate, function types
    - Bitwidth integer types: i1, i32, i64
  - Label type, opaque type, metadata type
- Low-level instructions
  - Memory instructions: alloca, load, store
  - Arithmetic, bitwise binary, comparison, conversion, ...
  - Terminator instructions

Tool support

Work flow in this course

- Generate LLVM-IR (text form)
- May use `clang`/`llvm-dis` to double-check your code generation
  - Careful, works only for common subset (obviously)

		command-line option	output
clang	front end	clang -c -emit-llvm file.c	file.bc
llvm-dis	disassembler	llvm-dis file.bc	file.ll

- Use `opt` instead of optimizing yourself

A first example

```
int tmp = 14;

int main () {
    return tmp;
}

> clang -c -emit-llvm ex1.C
> llvm-dis ex1.bc

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4        ; type of %1 is i32*
  store i32 0, i32* %1
  %2 = load i32 @tmp, align 4
  ret i32 %2
}
```

A first example (cont'd)

```
@tmp = global i32 14, align 4

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  %2 = load i32* @tmp, align 4
  ret i32 %2
}
```

- Keywords: define, alloca, store, load, ret
- Type: i32, i32*
- Identifiers: @tmp, @main, %1, %2
- Constant: 14
- Function attributes provide hints to selected optimization passes. They are not part of the function type, thus can be shared.

Identifiers in LLVM

http://www.llvm.org/docs/LangRef.html#identifiers

LLVM has two kinds of identifiers: global and local

- Global identifiers (variables, functions)
  - Represented using the at-sign
  - Global variables must be initialized (unless they are in a different compilation unit)
  - Are actually pointers, must be dereferenced before reading (`load`)
- Local identifiers (variables, registers, types)
  - Local identifiers start with a percentage sign.
  - Explicit stack allocation is done using `alloca`.
  - Unnamed temporaries are created implicitly and are represented by numbers with % prefix, e.g., %1. The numbers are assigned sequentially.
  - `load` and `store` instructions are used for reading and writing

Naming

```
@tmp = global i32 14, align 4

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %3 = alloca i32, align 4      ;;
  store i32 0, i32* %1
  %2 = load i32* @tmp, align 4
  ret i32 %2
}
> llc: ex1a.ll:9:3: error: instruction expected to be
       numbered '%1'
  %3 = alloca i32, align 4
  ^
```

- Suppose we had created the `.ll` file ourselves and named the first temporary variable differently. The backend compiler `llc` returns an error:
  - Naming of temporaries follows a fixed scheme. Within each function scope, the *n*th temporary is called `%n`.
- The first temporary, `%1`, is redundant and can be optimized away:

```
> opt -mem2reg -stats prog.bc -o progopt.bc
```

Type safety

Now consider the case where we created code with an incorrect `store` operation:

```
@tmp = global i32 14, align 4

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i16 0, i32* %1              ;;;
  %2 = load i32* @tmp, align 4
  ret i32 %2
}
> llc: ex1b.ll:10:9: error: stored value and pointer type
    do not match
  store i16 0, i32* %1
```

- The `llc` compiler type-checks the code.

What else?

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

- Types
  - Integral types: in, $1 \leq n \leq 2^{23} - 1$, e.g., `i1`
  - Pointer type, e.g., `i32 *`
  - Function type: <return type> (parameter list)
    - `i32 (i32)`
    - `double (i32, i32*)`
    - `{i32,i32} (double*)`
    - The function type is no first-class type (insns cannot generated values)
- Expressions
  - Binary (arithmetic):`add`, `fadd`, ..., `udiv`, `sdiv`
  - Memory: `alloca`, `load`, `store` ..., `getelementptr`
  - `icmp`, `fcmp`; `call`
- Function definitions (declaration)
  - Start with the keyword `define` (`declare`), the function identifier, its typed return and parameter arguments, and a list of basic blocks:
  - Basic blocks may start with a label and must end with a terminator instruction.

In-class exercise (LLVM)

Translate the following function by hand to LLVM.

In-class exercise (LLVM)

Translate the following function by hand to LLVM.

Name mangling

Consider the LLVM representation of a C++ function:

```
void printInt(int x) { }

> clang -c -emit-llvm file.cc
> llvm-dis file.bc

; Function Attrs: nounwind ssp uwtable
define void @_Z8printInti(i32 %x) #0
{
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  ret void
}
)
```

- `_Z8printInti` is a *mangled* identifier.
- C++ requires name mangling, see `http://mentorembedded.github.io/cxx-abi/abi.html#mangling`

STS Software Technology Systems

Outline

LLVM core classes

http://llvm.org/docs/ProgrammersManual.html

The *core classes* of LLVM form the API to the representation in LLVM-IR. They include

- The `Module` class
- The `Function` class
- The `IRBuilder` class
- The `Basicblock` class
- The `Value` class
- The `Const` class

The class `IRBuilder`

http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html

- The class `IRBuilder` is an auxiliary class for creating instructions and inserting them into a basic block.
- Insertions points are either the end of the block or iterator locations.
- Ex. (see Programmer's Manual, helpful hints):

```
Instruction *pi = ...;
IRBuilder<> Builder(pi);
CallInst* callOne = Builder.CreateCall(...);
CallInst* callTwo = Builder.CreateCall(...);
Value* result = Builder.CreateMul(callOne, callTwo);
```

The Kaleidoscope tutorial

http://llvm.org/docs/tutorial

The Kaleidoscope tutorial is a good start for learning the API. The setup

```
static std::unique_ptr<Module> *TheModule;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*> NamedValues;

Value *ErrorV(const char *Str) {
  Error(Str);
  return nullptr;
}
```

- All LLVM code is part of a module. `*TheModule` is the top-level structure that contains all generated code.
- The `Builder` object is used for expression generation.
- `NamedValues` is a symbol table for variables (function arguments).

Compilation schemes

Intermediate code generation is yet another example of syntax-directed translation. This time, we specify the rules informally, through pseudo code ("compilation schemes").

- We call the scheme "codegen".
- There is one scheme per syntactic category.
- We'll keep the LLVM structure in mind but also try to abstract from it.

Compilation scheme for function calls

```
codegen (f(a1,.., an)):
    function := lookup f
    if f unknown, error ``function f not known"
    for (int i = 1 .. n)
       v := codegen ai
       argstack.push_back (v)
    build callexp (function, argstack)
```

- Code generation of function calls starts by looking up the function name
- If the function has been defined, the arguments are recursively codegen'd and pushed on an argument stack.
- The final step is to build a call expression, using the function information and the argument stack.

Code generation for function calls

http://llvm.org/docs/tutorial/LangImpl3.html#expression-code-generation

```
Value *CallExprAST::codegen() {
  // Look up the name in the global module table.
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return ErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return ErrorV("Incorrect no. of arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

Compilation scheme for arithmetic expressions

```
codegen(a op b):
   first  := codegen(a);
   second := codegen(b);
   check first; check second;

   switch (op)
     case '+': build addexpr(first, second)
     case '-': build subexpr(first, second)
     case '*': build mulexpr(first, second)
     case '/': build divexpr(first, second)
     default: error ``Illegal operator''
```

- The code for the two operands is recursively generated.
- Pattern matching on the operator controls which arithmetic expression is built.
- This scheme assumes that the code type-checks.

Code generation for arithmetic expressions

http://llvm.org/docs/tutorial/LangImpl3.html#expression-code-generation

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)
    return nullptr;

  switch (Op) {
  case '+':
    return Builder.CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder.CreateFSub(L, R, "subtmp");
  case '*':
    return Builder.CreateFMul(L, R, "multtmp");
  case '<':
    L = Builder.CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder.CreateUIToFP(L,
       Type::getDoubleTy(getGlobalContext()),
                           "booltmp");
  default:
    return ErrorV("invalid binary operator");
  }
}
```

Compilation scheme for function definitions

```
codegen (t f(a1,..,an){stm}):
   function := lookup f
   if f known
      error ``function f already defined''

   for (int i=1..n)
      add (ai,typeof(ai))

   build basic block(function)
   v := codegen({stm})
   build retexp(v)
```

- If the function is already known, no code is generated.
- Otherwise, the arguments and their type are added to the symbol table.
- A basic block is created. (In LLVM, function bodies contain a basic block as top structure.)
- The code for the function body is recursively generated and inserted in the basic block. Lastly, the return expression is built and inserted.

Generating code for function definitions

```
Function *FunctionAST::codegen() {
  // First, check for a previous 'extern' declaration.
  Function *TheFunction =
      TheModule->getFunction(Proto->getName());

  if (!TheFunction)
    TheFunction = Proto->codegen();

  if (!TheFunction)
    return nullptr;

  // Create a new basic block to start insertion into.
  BasicBlock *BB = BasicBlock::Create(getGlobalContext(),
                                      "entry",
                                      TheFunction);
  Builder.SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[Arg.getName()] = &Arg;
  }
```

Generating code for function definitions (2)

```
Function *FunctionAST::codegen() {
  // ..

   if (Value *RetVal = Body->codegen()) {
    // Finish off the function.
    Builder.CreateRet(RetVal);

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

Summary

- Graph/tree-based vs. linear IRs; CFG, basic blocks; TAC
- LLVM, LLVM API
- Compilation schemes

References

- Frances E. Allen. Control flow analysis. In Proceedings of a Symposium on Compiler Optimization. ACM, New York, NY, USA, 1-19, 1970. DOI=10.1145/800028.808479 http://doi.acm.org/10.1145/800028.808479
- LLVM online resources
- IPL 6.2, 6.3