Compiler Design

by dcamenisch

1 Introduction

This document is a summary of the 2022 edition of the lecture *Compiler Design* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at TODO: INSERT LINK.

The simplified compiler has the following structure:

- Lexical Analysis
- Parsing
- Intermediate Code Generation
- Code Generation

x86lite

- x86lite memory consists of 2^{64} bytes numbered `0x00000000` through `0x0xffffffff`, split into 8-byte quadwords (has to be quadword-aligned)
- The stack grows from high addresses to low addresses, `rsp` points to the top of the stack, `rbp` points to the bottom of the current stack frame
- Registers: `rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, rip, r08 - r15`
- Flags: `OF` overflow, `SF` sign (1 = negative), `ZF` zero
- Instructions: `SRC` before `DEST` (AT&T syntax), prefix register with % and immediate values with $
- Operands:
  - `Imm`: 64-bit literal signed integer
  - `Lbl`: label representing a machine address
  - `Reg`: one of the registers, the value is its content
  - `Ind`: machine address [base:`reg`][index:`reg`][disp:`int32`] (base + index*8 + disp)
- x86 assembly is organized into labeled blocks, indicating code locations used by jumps, etc. Program begins execution at designated label (`main`)
- Calling Conventions (SYSTEM V AMD64 ABI):
  - Setup Stack Frame:
    ```
    pushq %rbp
    movq %rsp, %rbp
    ```
  - Caller Save - freely usable by the called code
  - Callee Save - must be restored by the called code (`rbp, rsp, rbx, r12-15`)
  - Arguments:
    1..6: `rdi, rsi, rdx, rcx, r08, r09`
    7+: on the stack (in right-to-left order)
    For $n > 6$, $(n-7)+2) * 8 + \texttt{rbp}$
  - Return value in `rax`
  - 128 byte "red zone" - scratch pad for the callee (beyond `rsp`)

Intermediate Representations

Direct translation is bad as it is hard to optimize the resulting assembly code. The representation is too concrete, as it already committed to using certain registers etc. Further retargeting the compiler to a new architecture is hard. Finally control-flow is not structured, arbitrary jumps from one code block to another. Implicit fall-through makes sequences non-modular.

- IR's: allows machine independent code generation and optimization
- Multiple IR's: get program closer to machine code without losing the information needed to do analysis and optimizations (high / mid / low level IR)
- Good IR: Easy translation target, easy to translate, narrow interface (fewer constructs means simpler phases / optimizations)
- Basic Blocks are a sequence of instructions that are always executed from the first to last instruction. They start with a label and end with a control-flow instruction (no other control-low instruction or label)
- Basic blocks can be arranged into a control-flow graph: Nodes are basic blocks - directed edges represent potential jumps

LLVM (Low Level Virtual Machine)

- LLVM provides the `getelementptr` instruction to compute pointer values: Given a pointer and a path through the structured data pointed to by that pointer, GEP computes an address - abstract analog of LEA.

Parsing

- Derivation Orders
  - Productions of the grammar can be applied in any order. There are two standard orders:
    - Leftmost derivation: Find the left-most nonterminal and apply a production to it
    - Rightmost derivation: Find the right-most nonterminal and apply a production there
- LL & LR Parsing
  - Top-down vs. Bottom-up
  - There is a problem: Want to decide which production to apply based on the look-ahead symbol → LL(1) Grammars: not all grammars can be parsed top-down with a single lookahead.
  - LL(1) means **L**eft-to-right scanning, **L**eft-most derivation, **1** lookahead symbol.
  - Left-factoring a grammar can make it LL(1): If there is a common prefix we can add a new nonterminal at the decision point.
  - We also need to eliminate left-recursion:
    - $S \to S\ \alpha_1 \mid \cdots \mid S\ \alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$
      Rewrite as
      $S \to \beta_1\ S' \mid \cdots \mid \beta_m\ S'$
      $S' \to \alpha_1\ S' \mid \cdots \mid \alpha_n\ S' \mid \epsilon$
  - Predictive Parsing: Given an LL(1) grammar: For a given nonterminal, the lookahead symbol uniquely determines the production to apply - top-down parsing = predictive parsing - driven by a predictive parsing table: *nonterminal x input token → production*

- First / Follow

  Consider a given production $A \to \gamma$

  (Case 1) Construct the set of all input tokens that may appear **first** in strings that can be derived from γ - Add the production $\to \gamma$ to the entry for each such token

  (Case 2) If γ can derive ϵ, then we construct the set of all input tokens that may **follow** the nonterminal A in the grammar - Add the production $\to \gamma$ to the entry for each such token

- Bottom-up Parsing (LR Parsers)

  LR(k) parser: **L**eft-to-right scanning, **R**ightmost derivation, **k** lookahead symbols

  Technique: "Shift-Reduce" parsers: Work bottom up instead of top down. Construct rightmost derivation of a program in the grammar. Better error detection/recovery (poor error reporting)

  Parser state: Stack of terminals and nonterminals. Unconsumed input is a string of terminals. Current derivation step is stack + input

- Shift: Move look-ahead token to the stack
- Reduce: Replace symbols γ at the top of stack with nonterminal X s.t. $X \to \gamma$ is a production. pop γ, push X.
- Action Selection Problem:
  - Given a stack σ and a lookahead symbol b, should the parser **shift** b onto the stack (new stack is σb) , or **reduce** a production $X \mapsto \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack if αX)?
  - Sometimes the parser can reduce, but should not, sometimes the stack can be reduced in different ways
  - Main idea: Decide based on a prefix α of the stack plus look-ahead
- $LR(0)$ states: $LR(0)$ state: items to track progress on possible upcoming reductions. $LR(0)$ item: a production with an extra separator "." in the rhs.
- Run parser state though a DFA. DFA can be represented as a table of shape state $\times$ (terminals + nonterminals). two types of actions: shift and go to state n, reduce using reduction $X \mapsto \gamma$

Types

- LUB = Least upper bound
- Soundness of a subtypinig rule = Matches subset relation of value set
- argument type is contravariant, output type is covariant
  - It's okay if a function takes more arguments
  - It's okay if a function returns less arguments
- Mutable structure are invariant: covariant reference types are unsound, contravariant reference types are also unsound
- Mutable structures are invariant
- Structural vs. Nominal Typing

OAT Type System

- Primitive (non-reference) types: int, bool
- Definitely non-null reference types: R (named) mutable structs with width subtyping, strings, arrays - Possibly-null reference types: R?

Compiling Objects

- Objects contain a pointer to a dispatch vector (also called vtable) with pointers to method code.
- DV layout of new method is appended to the class which is being extended
- Multiple Inheritance approaches: Allow multiple DV tables (C++) Choose which DV to use based on static types, casting requires runtime operations; Use a level of indirection: Map method identifiers to code pointers using a hash table, search up through the class hierarchy; Give up separate compilation: Use sparse dispatch vectors or binary decision trees.
- Multiple Dispatch Vectors:Objects may have multiple entry points with individual DVs, casts change entry point of a variable

Optimizations

- Problem: many optimizations trade space for time (e.g. Loop unrolling)
- Constant Folding: If operands are statically known, compute value at compile-time (has to preformed at every stage of optimization - constant expressions can be created by translation or earlier optimizations / can enable further optimizations). Also: Algebraic Simplification (Use mathematical identities)
- Copy Propagation: For x = y replace uses of x with y
- Dead Code Elimination: If side-effect free code can never be observed, safe to eliminate it
- Inlining: Replace a function call with the body of the function (arguments are rewritten)
- Code Specialization: Create Specialized versions of a function that is called form different places with different arguments.
- Common Subexpression Elimination: In some sense, it is the opposite of inlining: fold redundant computations together
- Loop Optimizations
  - Hot spots often occur in loops (esp. inner loops)
  - Loop Invariant Code Motion
  - Strength Reduction (replace expensive ops by cheap ones by creating a dependent induction variable)
  - Loop unrolling

Code Analysis

- Liveness
  - Observation: uid1 and uid2 can be assigned to the same register if their values will not be needed at the same time. Liveness property is more fine grained than scope.
  - Liveness analysis is one example of dataflow analysis: A variable v is live on edge e if there is a node n in the CFG such that use[n] contains v and a directed path from e to n such that for every statement s' on the path def[s'] does not contain v

  - Dataflow: Compute information for all variables simultaneously. Solve the equations by iteratively converging on a solution: Start with a rough approximation to the answer, refine the answer at each iteration, keep going until nor more refinement is possible.

Liveness: backward, may (out = union in, in = gen union (out \ kill))
Reaching: forward, may (in = union out, out = gen union (in \ kill))
Very busy: background, must (out = intersection in, in = gen union (out \ kill))
Available: forward, must (in = intersection out, out = gen union (in \ kill))
Key idea: Iterative solution of a system of equations over a lattice. Iteration terminates if flow functions are monotonic, equivalent to the MOP answer if flow functions distribute over meet

Register Allocation

- Register Allocation: Compute liveness information for each temp, create an infereence graph, try to color the graph.
- Kempe: 1. Find a node with degree ¡ k and cut it out of the graph, 2. Recursively k-color the remaining subgraph, 3. When remaining graph is colored, there must be at least one free color available for the deleted node. If the graph cannot be colored we spill that node.
- Precolored nodes: Certain variables must be preassigned to registers (call, imul, caller-save registers)

Very busy Expressions

- Expression e is very busy at location p of every path from p must evaluate e before any variable in e is redefined - backward, must

Loops

- A loop is a strongly connected component (head reachable from each node)
- Concept of dominators: A dominates B = if the only way to reach B from start node is via A. A loop contains at least 1 back edge. (back edge = target dominates the source)
- dom is transitive and anti-symmetric, can be computed as a forward dataflow analysis

SSA

- Where to Place phi functions: Compute dominance frontier:
- for all nodes B:
- if #(pred $[B]$) ≥ 2
  - for each $p \in$ pred $[B]$
    - runner := p
    - while (runner $\neq$ doms$[B]$)
      - DF [runner] := DF [runner] $\cup \{B\}$
      - runner := doms [runner]
- phi nodes can be placed at dominator tree join nodes
- eliminate phi nodes after optmization

GC

- Garbage: An object x is reachable iff a register contains a pointer to x or another reachable object y contains a pointer to x
- reachable objects can be found by starting from registers and following all pointers
- Mark and Sweep

  When memory runs out, GC executes two phases: mark phase: trace reachable objects; sweep phase: collects garbage objects (extra bit reserved for memory management)

  pointer reversal can be used to allow auxiliary data to be stored in the objects.
- Stop and Copy

  Memory is organized into two areas: Old space (used for allocation), new space (use as a reserve for GC)

  When old space is full all reachable objects are moved, old and new are swapped.
- Reference Counting

  Store number of references in the object itself, assignments modify that number. Cannot collect circular structures.