Compiler Design

by dcamenisch

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 $n > 6$, $(n-7)+2)*8+$ `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)

Storage Types: local variable `%uid`, global variable `@gid`, abstract locations (stack-allocated with `alloca`), heap-allocated structures (`malloc`).

Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph (SSA).

The entry block of the CFG does not have to be labeled, the last instruction of a block is called the terminator.

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 - analog of LEA.

```
getelementptr <ty>* <ptrval> {, <ty> <idx>}*
```

GEP never dereferences the address it is calculating.

Lexing

Lexing is the process of taking the source code as an input and producing a token stream as output. The problem is to precisely define tokens and matching tokens simultaneously.

One way of implementing a lexer is, using regular expressions. RegEx rules precisely describe a sets of strings. But RegEx alone can be ambiguous if we have multiple matching rules. Most languages therefore choose the longest match or have another specified order.

RegEx can be implemented forming an nondeterministic automata and then transforming it to a deterministic finite automata.

Parsing

In this part we take the token stream and generate an abstract syntax tree (AST).

- Chomsky Hierarchy:
  - Regular - Productions have at most one nonterminal and it is at the start or end of the word
  - Context-Free (CFG) - LHS of productions only have a single nonterminal
  - Context-Sensitive -
  - Recursively Enumerable
- A CFG consists of a set of terminals, a set of nonterminals, a start symbol and a set of productions.
- Derivation Orders - Productions 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

  All orders will lead to the same parse tree.
- A grammar is **ambiguous** if there are multiple derivation trees for the same word.
- In CFGs ambiguity can (often) be removed by encoding precedence and associativity in the grammar.

LL & LR Parsing

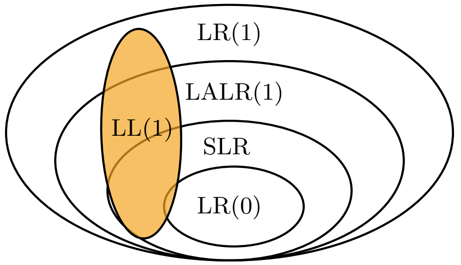
- Top-down vs. Bottom-up
- There is a problem: Want to decide which production to apply based on the look-ahead symbol $\rightarrow$ LL(1) Grammars: not all grammars can be parsed top-down with a single lookahead.
- LL(1) means **L**eft-to-right scanning, **L**eft-most derivation, **1** lookahead symbol.
- Left-factoring a grammar can make it LL(1): If there is a common prefix we can add a new non-terminal at the decision point.
- We also need to eliminate left-recursion:
  - $S \rightarrow S\ \alpha_1 \mid \cdots \mid S\ \alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$
    Rewrite as
    $S \ \rightarrow \beta_1\ S' \mid \cdots \mid \beta_m\ S'$
    $S' \rightarrow \alpha_1\ S' \mid \cdots \mid \alpha_n\ S' \mid \epsilon$
- 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*
- The **first set** of a nonterminal contains all terminals that begin strings derivable from the nonterminal.
- The **follow set** of a nonterminal contains all terminals that can appear immediately to the right of the given nonterminal.
- Constructing the parsing table: Consider a given production $A \rightarrow \gamma$
  (Case 1) Construct the first set - Add the production $\rightarrow \gamma$ to the entry for each token in the set.
  (Case 2) If γ can derive ϵ, then we construct the follow set - Add the production $\rightarrow \gamma$ to the entry for each token in the set.
  If there are two different productions for a given entry, the grammar is not LL(1).

- Bottom-up Parsing, LR(k) Parser: **L**eft-to-right scanning, **R**ightmost derivation, **k** lookahead symbols

  LR grammars are more expressive than LL grammars. They can handle left-recursive and right-recursive grammars.

  Technique: "Shift-Reduce" parsers: Work bottom up and construct right-most derivation of a program in the grammar. Better error detection/recovery, but 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) state: items to track progress on possible upcoming reductions.
- LR(0) item: a production with an extra separator "." in the RHS.
- The idea is that the stuff before the "." is already on the stack and the rest is what might be seen next.
- Constructing the DFA:
  * Add new production: $S' \mapsto S\$$
  * Start state of the DFA = empty stack, $S' \mapsto .S\$$
  * Closure:
    1. Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the ".".
    2. The added items have the "." at the beginning
    3. Keep iterating until a fixed point is reached.
- Run parser state though a DFA. DFA can be represented as a table of shape state × (terminals + nonterminals). Two types of actions: shift and go to state n, reduce using reduction $X \mapsto \gamma$
- An LR(0) machine only works if states with reduce actions have a single reduce action else we will encounter shift/reduce or reduce/reduce conflicts (use LR(1) grammar).
- For LR(1) we have a shift-reduce conflict if the shifted token is contained in the follow set of the reduction.



Lambda Calculus

- Lambda calculus has variables, functions and function application. Instead of `(fun x -> e)` we write: $\lambda x.e$
- The only values are closed functions
- To substitute value v for variable x in expression e we replace all free occurrences of x in e by v: $e\{v/x\}$
- Function application is interpreted by substitution.
- In `fun y -> x + y`, `x` is said to be free and `y` is bound by `fun y`.
- A term without free variables is closed, else it is open.
- Two terms that differ only by consistent renaming of bound variables are alpha equivalent.
- To avoid accidently capturing a free variable by a substitution $e_1\{e_2/x\}$, we first pick an alpha equivalent version of e_1 such that the bound variables do not mention the free variables of e_2.
- Operational Semantics is a way to give meaning to a program (interpreter) using inference rules. $exp \Downarrow v$ means exp evaluates to v.
- With inference rules we can build up derivation or proof trees. Leaves of the tree are axioms.

Typing

Applying a set of inference rules allows for type checking of a program.

- A well-typed program either terminates in a well-defined way, or it continues computing forever.
- If we view types as sets of values, there is a natural inclusion relation Pos ⊆ Int. This gives rise to a subtype relation P <: Int and to a subtyping hierarchy.
- The LUB (least upper bound) is defined for two types $T_1 \vee T_2$.
- Soundness of a subtypinig rule = Matches subset relation of value set
- Argument type is contravariant (it is okay if a function takes more arguments), output type is covariant (it is okay if a function returns less arguments).

$$\frac{S_1 <: T_1 \qquad T_2 <: S_2}{(T_1 -> T_2) <: (S_1 -> S_2)}$$

- Mutable structure are invariant: covariant / contravariant reference types are unsound.
- Structural vs. Nominal Typing - is type equality / subsumption defined by the structure or the name.

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?`
- Subtyping `R` <: `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

There are different kinds of optimization: Power, Space, Time.

- Constant Folding: If operands are statically known, compute value at compile-time. More general algebraic simplification: Use mathematical identities.
- Constant Propagation: If x is a constant replace its uses by the constant.
- 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 to local variables).
- Code Specialization: Create Specialized versions of a function that is called form different places with different arguments.
- Common Subexpression Elimination: 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 (hoist outside)
  - Strength Reduction (replace expensive ops by cheap ones by creating a dependent induction variable)
  - Loop Unrolling

Liveness Analysis

- `uid1` and `uid2` can be assigned to the same register if their values are not needed at the same time (more fine grained than scope).
- A variable v is live at a point L, if v is defined before L and used after L.
- For a statement (node) s we define:
  - use$[s]$: set of variables used by s
  - def$[s]$: set of variables define by s
  - in$[s]$: set of variables live on entry to n
  - out$[s]$: set of variables live on exit from n
- It holds: in$[n] \supseteq$ use$[n]$, in$[n] \supseteq$ out$[n] \setminus$ def$[n]$ and out$[n] \supseteq$ in$[n']$ if $n' \in$ succ$[n]$.
- 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 Analysis: Compute information for all variables simultaneously. Solve the equations by iteratively converging on a solution.

Algorithm

```
for all n, in[n] = [], out[n] = []
repeat until no change in 'in' and 'out'
  for all n
    out[n] = union of in[n'] for all n' in succ[n]
    in[n] = use[n] union (out[n] \ def[n])
```

Register Allocation

- Linear-Scan Register Allocation: Compute liveness information and then scan through the program, for each instruction try to find an available register, else spill it on the stack.
- Graph Coloring: Compute liveness information for each temp, create an inference graph (nodes are temps and there is an edge if they are alive at the same time), 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 a node.
- Improve by adding `move` related edges (temps used in a move should have the same color).
- Precolored Nodes: Certain variables must be pre-assigned to registers (`call`, `imul`, caller-save registers)

Generalizing Dataflow Analysis

This type of iterative analysis for liveness also applies to other analyses.

- Reaching Definitions: What variable def. reach a particular use of the variable? Used for constant / copy propagation.
- Available Expressions: Want to perform subexpression elimination.

Very Busy Expressions

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

Loops

A loop is a set of nodes in the CFG, with a distinguished entry, the header, and exit nodes.

- A loop is a strongly connected component (SSC), the head is reachable from each node and vice versa.
- Concept of dominators: A dominates B (A dom 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

Single Static Assignment (SSA)

Each LLVM IR `%uid` can be assigned only once.

- ϕ function chooses version of a variable by the path how control enters the ϕ node.
- The dominance frontier of a node B is the set of all CFG nodes y such that B dominates a predecessor of y, but does not strictly dominate y. Intuitively: starting at B, there is a path to y, but there is another route that does not go through B.
- Location of the ϕ function can be computed by dominance frontier:

```
for all nodes B
  if num of pred[B] >= 2
    for each p in pred[B]
      runner = p
      while runner != doms[B]
        DF[runner] = DF[runner] union {B}
        runner = doms[runner]
```

- ϕ nodes can be placed at dominator tree join nodes.
- eliminate ϕ nodes after optmization

Garbage Collection

- 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)

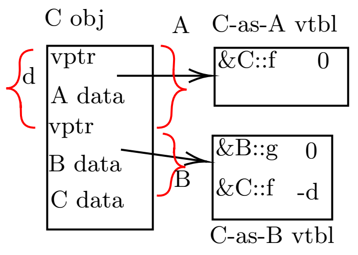
  Pointer reversal can be used to allow auxiliary data to be stored in the objects.
- Stop and Copy

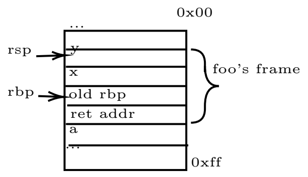
  Memory is organized into two areas: Old space (used for allocation), new space (use as a reserve for GC)

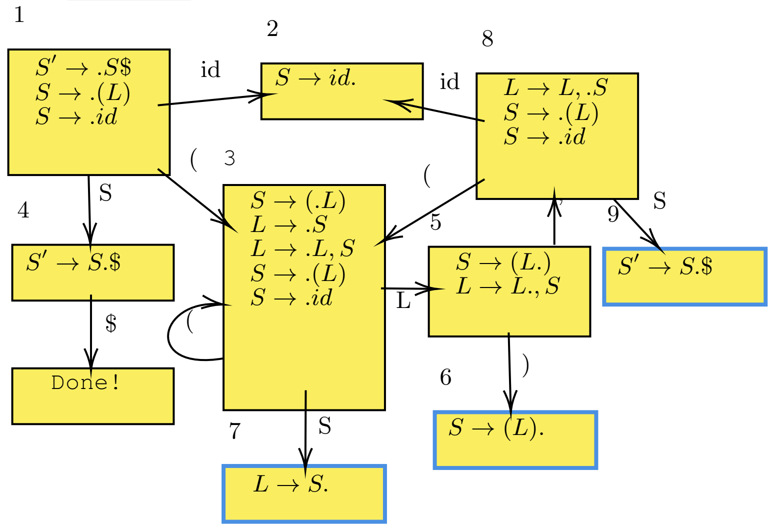
  When old space is full all reachable objects are moved, old and new are swapped. The new space is partitioned into copied and scanned, copied and empty regions.

  Copied and scanned contain objects whose pointer fields were followed and fixed, while the copied region contains the objects that where copied but whose pointer filed were not (yet) fixed.
- Reference Counting

  Store number of references in the object itself, assignments modify that number. If the reference count is zero, remove the object. Cannot collect circular structures.




0x00
...
rsp
y
x
rbp
old rbp
ret addr
a
...
0xff
foo's frame


1
$S' \to .S\$$
$S \to .(L)$
$S \to .id$
id
2
$S \to id.$
id
8
$L \to L,.S$
$S \to .(L)$
$S \to .id$
(
3
(
$S \to (.L)$
$L \to .S$
$L \to .L,S$
$S \to .(L)$
$S \to .id$
S
4
$S' \to S.\$$
$
Done!
(
L
5
$S \to (L.)$
$L \to L.,S$
S
9
$S' \to S.\$$
)
6
$S \to (L).$
S
7
$L \to S.$