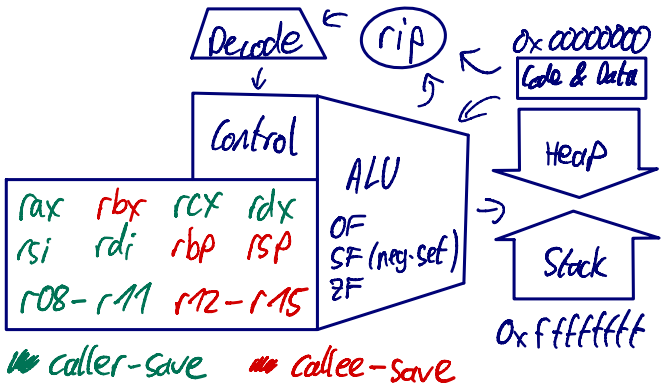
x86 Lite & System V ABI



function arguments

1st	2nd	3rd	4th	5th	6th	6<n th
rdi	rsi	rdx	rcx	r8	r9	((n-7)+2)·8+rbp

return value in rax

stack frame setup

pushq %rbp ; movq %rsp, %rbp ; subq $c, %rsp

teardown ~ alternatively use 128B red zone

addq $c, %rsp ; popq %rbp ; retq

Mid-Level IRs & LLVM Lite

triples	quadruples	SSA	stack-based
OP a b	a = b OP c	static single assignment	push / pop

find a balance between expressiveness and ease of translation / interpretation.

LLVM Lite IR

SSA with types, alloca/load/store, phi-nodes

```
define i1 @myfunc (i64 %0 ) {
    %1 = alloca i1
    %2 = icmp ne i64 %0 42
    store i1 %2 i1* %1
    br i1 %2 , label %yes , label %no
yes:  %3 = ...
no:   %4 = ...
end:
    %5 = phi i1 [%3, %yes] [%4, %no]
    ret %5
}
```

every function defines a CFG of basic blocks with branching or returning terminators

custom types and arrays are accessed via GEP:

```
struct Point { long x; long y; };
<~~> %Point = type {i64, i64}
ps[n].y <~~>
getelementptr %Point* %ps, i64 %n, i32 1
```

Lexing & Parsing

Chomsky Hierarchy

Regular ⊂ Context-Free ⊂ -Sensitive ⊂ Recursively Enumerable

lexing

Transform a stream of characters into a stream of tokens (keywords, identifiers, operators, ...). Often disambiguated by "longest match"

Context-Free Grammars

$S \mapsto (S)\ S$ (production)

terminals '(' and ')'
non terminal S
start symbol S

parse tree describes structure of (leftmost, rightmost, and any derivation if unambiguous.
LHS of production is a single terminal.

LL(1) Grammars

left-to-right scanning, leftmost derivation, one lookahead symbol.

eliminate left recursion:

$S \mapsto S\alpha_1 | \ldots | S\alpha_n | \beta_1 | \ldots | \beta_m$

becomes $S \mapsto \beta_1 S' | \ldots | \beta_m S'$

$S' \mapsto \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon$

top-down parsing using a table:

For all productions $A \mapsto \gamma$

(1) Add $\mapsto \gamma$ to entries (A, token) for tokens appearing first in strings derived from γ.

(2) If γ can derive ε, add $\mapsto \gamma$ to entries (A, token) for tokens following A in the grammar.

≥2 productions for entry => grammar NOT LL(1)

LR(0)

left-to-right scanning, rightmost derivation, no lookahead symbol.
works with any kind of recursion.
bottom-up (shift/reduce) parsing using a table
indicate top of stack using '.'
consume nonterminals using shift
consume terminals using goto
reduce when '.' reaches end of production
store state with symbol on stack: ε, t_3

Other LR

LR(1):
State is set of LR(1) items, i.e. LR(0) item and look-ahead symbol.
LALR(1), SLR(1), GLR

λ-Calculus & Type Inference

Untyped λ-calculus

exp ::= x | fun x -> exp | exp exp | (exp)
val ::= fun x -> exp
substitution e.g. $x\{v/x\} = v$
free / bound variables e.g. fun x -> x y (x: bound, y: free)
α-equivalence
Turing-complete
operational semantics:

$$\frac{}{v \Downarrow v} \quad \text{and} \quad \frac{exp_1 \Downarrow (fun\ x \to exp_3) \quad exp_2 \Downarrow v \quad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

Y-Combinator

$Y = \lambda f.\ (\lambda x.\ f(x\ x))\ (\lambda x.\ f(x\ x))$
satisfies $Yf = f(Yf)$, so it is used as the fix-point operator

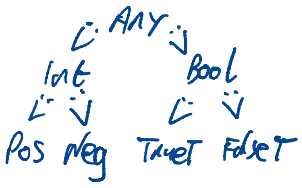
Simply Typed λ-Calculus with Integers

well-typed, guaranteed to terminate

Type-Safety

If $\vdash p : t$ is a well-typed program, then it 1) terminates in a well-defined way or 2) computes forever.

Subtyping



Least upper bound of two types is the lowest common type wrt. the subtyping hierarchy.

reflexive: $T <: T$
transitive: $T_1 <: T_2$ and $T_2 <: T_3 \Rightarrow T_1 <: T_3$
antisymmetric: $T_1 <: T_2$ and $T_2 <: T_1 \Rightarrow T_1 = T_2$
For functions:

$$\frac{S_1 <: T_1 \quad T_2 <: S_2}{(T_1 \to T_2) <: (S_1 \to S_2)}$$

For records: width subtyping (by par. count)
depth subtyping (by par. types)

! MUTABLE INVARIANCE ! $T_1\ ref <: T_2\ ref \Rightarrow T_1 = T_2$

Typing Techniques

Single Inheritance

Dispatch vectors with identical offsets for common functions (width subtyping).

Multiple Inheritance

(1) Multiple dispatch vector tables based on static types.
⊕ efficient
⊖ cast has runtime cost
(2) search with cache / hash tables
⊕ reasonably efficient
⊖ wasted space in DV
⊖ slow dispatch if conflict
(3) sparse D.V.s or binary search trees
⊕ benefits from branch prediction
⊖ need to know entire class hierarchy

Optimizations

constant folding, algebraic simplification (caution! Floats!), constant propagation, copy propagation, dead code elimination, inlining, code specialization, common subexpression elimination, loop-invariant code motion, strength reduction, loop unrolling

Liveness & Data-Flow

a variable v is live at program point L
$\Leftrightarrow$ v is defined before and used after L

precisely: v is live on edge e
$\Leftrightarrow$ $\exists$ node n in CFG with $v \in use[n]$
s.t. path from e to n
has no statement s' with $v \in def[s']$

$in[n] = use[n] \cup (out[n] - def[n])$
$out[n] = \bigcup_{n' \in succ[n]} in[n']$

Register Allocation

greedy linear-scan

Compute liveness
for each instruction defining some x:
  used = {r | reg r = uid_loc(y) s.t. $y \in live(x)$}
  available = pal − used
  if available = $\emptyset$
    uid_loc(x) := slot n ; n += 1
  else
    uid_loc(x) := reg r

Kempe Graph coloring

if $\exists v.\ deg(v) < K$
  kempe-color G removing v
  Add v back and color using a compatible col
else
  kempe-color G removing some u
  Add u back, try to color or spill

Precolor and Coalesce

Necessary allocations (to satisfy the ISA) can be enforced by precoloring and setting infinite degrees.
To eliminate moves, source and destination can be coalesced into a single vertex with more neighbors.

Loop Optimization

Dominators

A dominates B $\Leftrightarrow$ the only way of reaching B is via A

$in[n] = \bigcap_{n' \in pred[n]} out[n']$

$out[n] = in[n] \cup \{n\}$

Dominance Frontier

$DF[X] = \{y \mid X \text{ dominates a predecessor of } y \text{ and } X \text{ does not strictly dominate } y\}$
$= \{y \mid \exists M \to y \wedge (X\ dom\ M) \wedge \neg(X\ sdom\ y)\}$

for all nodes B
  if #pred[B] $\geq$ 2
    for $p \in pred[B]$
      runner := p
      while runner $\neq$ doms[B]
        DF[runner] := DF[runner] $\cup$ {B}
        runner := doms[runner]

preferred location for ϕ-nodes. Alternative: place everywhere and eliminate iteratively.

Garbage Collection

Allocate space as needed for new objects. Once memory runs out, compute what might be used again, then free remainder.

Mark and Sweep

per-object mark bit, set by elimination from to-do list (initially roots, filled recursively). Sweep by scanning over heap.
Problem: mark needs unbounded storage for to-do list. Solve through pointer reversal.
⊕ no objects moved => stable refs, performance
⊖ memory fragmentation

Stop and Copy

partition heap in two halves, sequentially copying and fixing pointers.
⊕ cheap allocation and collection
⊖ instable references => language support!

Reference Counting

store counter per object, increment on assignment, decrement on reassignment. Free when counter hits 0
⊕ incremental GC without large pauses
⊖ high overhead
⊖ must be careful with circular structures

Analyzing Dataflow

The monotone framework allows for iterative analysis of changing properties in programs. It does fix-point calculations with the goal of finding least upper bounds $\bigsqcup$ (may) or greatest lower bounds $\bigsqcap$ (must). Forwards analysis takes into account past behavior, backwards analysis looks at future behavior. Note: $\sqcap$ must distribute!

Initially: $out[n] = in[n] \in \{\emptyset, \mathcal{L}\}$
repeat until fix point
  for all n
    Compute $out[n]$ and $in[n]$

Variable Liveness (backward, may)
initially: $out[n] = in[n] = \emptyset \subset VARIABLES$
$in[n] = use[n] \cup (out[n] \setminus def[n])$
$out[n] = \bigcup_{n' \in succ[n]} in[n']$

Very Busy Expressions (backward, must)
Initially: $out[n] = in[n] = \emptyset \subset EXPRESSIONS$
$in[n] = gen[n] \cup (out[n] \setminus kill[n])$
$out[n] = \bigcap_{n' \in succ[n]} in[n']$

with gen $\hat{=}$ expression computed before any of their ops
  kill $\hat{=}$ expression where ops are changed before

Reaching Definitions (forward, may)
initially: $out[n] = in[n] = \emptyset \subset ASSIGNMENTS$
$out[n] = gen[n] \cup (in[n] \setminus kill[n])$
$in[n] = \bigcup_{n' \in pred[n]} out[n']$

with gen $\hat{=}$ last assignment to variable in block
  kill $\hat{=}$ not last assignment to variable in block

Available Expressions (forward, must)
initially: $out[n] = in[n] = \emptyset \subset EXPRESSIONS \times LOCS$
$out[n] = gen[n] \cup (in[n] \setminus kill[n])$
$in[n] = \bigcap_{n' \in pred[n]} out[n']$

with gen $\hat{=}$ last storage of expression in loc
  kill $\hat{=}$ overwritten locs.