Interp & Virtual Machines   Interp (1).

intro : interpreters
- AST ~~vs~~ SM vs 3AC
- interp vs JIT

translate : every time? (Perl)
Java → class (JIT | interp)

example source interprete

interp.script
```
#!interp -foo -bar
yada
  yada
    yada.
```
spaces.

interp.c → interp

interp : argv[0] = "interp"
argv[1] = "-foo"
argv[2] = "interp.script"
argv[3] = "bar"
argv[4] = "baz

interp bar baz

{ present only if given
{ only one word ~~permitted~~

→ if not present
argv[1] is script name.

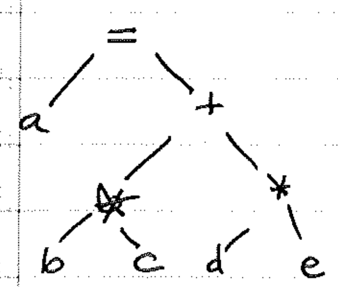
Interpreters & Virtual Machines

compilers → IL. → native code
→ interp

IL → HIR (AST)
MIR (medium = SM or 3AC)
LIR (close to native machine).

awk (Aho, Weinberger, Kernighan)
- pattern lang
- obsoleted by Perl

- construct AST
- direct interp of AST
- standard postorder traversal.




```
abstract class ast {
    abstract Value eval();
}

class mul extends ast {
    Value eval() {
        lv = l.eval()
        rv = r.eval()
        ~~return~~
        av = lv * rv
        return av
    }
    ast l
    ast r
}
```

Interp (3)

<u>non-oo lang</u>

```
eval (t)
    switch (t -> opcode) {
        case MUL :  eval_mul (t); break
        ⋮
    }
}
```

prob: awk slower than perl
((nawk / gawk is not))

while ⇝ while (e.eval() is true)
s.eval()

```
  while
  /   \
 e     S
```

---

alternate : gen SMC (stk m/c)
gen TAC (3 adr code).

which is better?
SMC – Java, .NET, ocaml, mosml
3AC – Dis, Parrot

2 ans "better" → interp
JIT.

"better" — code size
— code speed

~~Oversimplified~~ Interp (4)

SMC ~~3AC~~ : add, mul, ld, st, jeq ...

$a = b*c + d*e$

```
.. ld  b
.. ld  c
:  mul
.. ld  d
:.. ld  e
.  mul
.  add
.. st  a
```

$\boxed{\text{13 bytes}}$

~~interp~~
~~for(;;){~~
~~switch (*ip++){~~
~~case ld: *++sp~~

machine code: 8 RISC insns
(3 CISC insns?)
VAX
(7 Pent mous)

interp 1:

```
for(;;){
switch (*ip++){
  case ld: *++sp = loc[*ip++]; break
  case st : loc[*ip++] = *sp--; break
  case add: --sp; sp[0] += sp[1] ; break
  case mul : --sp ; sp[0] *= sp[1] ; break
  :
}}.
```

```
sw:
cmp  ip, 255
bgtu  default
t1 = *ip
t1 = t1 << 2
ip += 1
t2 = &switch
t3 = t1 + t2
goto *t3
```

```
ld:   sp += 4
      t1 = ip << 2
      ip += 1
      t2 = [loc + t1]
      *sp = t2
      goto od
```

~~break = goto od~~

od: goto sw.

```
add:  sp -= 4
      t1 = *sp
      t2 = [sp+4]
      add
      *sp = t3
      goto od.
```

ld = 6
st = 6
add = 6
mul = 6.

overhead = 9

9 × 14 = 126 insns

~~goto~~ Interp (5)

optim: cache tos

```
case ld: *++sp = tos; tos = loc[*ip++]; break
     st: loc[*ip++] = tos; tos = *sp-- break
     add: tos += *sp-- ; break
     mul: tos *= *sp-- ; break.
```

```
ld:  sp += 4                        add: t1 = *sp
     *sp = tos  <- t1 = *ip              sp -= 4
     t1 = ip << 2                        tos += t1
     ip += 4                             goto od
     tos = [loc + t1]                    (4)
     goto od
     (6) 7
```

new count: $5 \times (8 + \cancel{6}\,7) + 3 \times (8 + 4) = 111$ insns

~~cache~~ cache tos; to2?
no — too much copying

switch — why ≤ 255 check
on uchar ?!?!

```
swith (*ip++) {
```

gcc goto


```
static void *insn [] = {
    &&_ld, &&_st, &&_add, &&_mul };
int : register sw = insn;  // reg avoids repeated
                                        sethi/or
      goto *sw[*ip++];
```

don't count init

```
_ld: *++sp = tos; tos = loc[*ip++]; goto *sw[*ip++]
_st: loc[*ip++] = tos; tos = *sp++; goto *sw[*ip++]
_add: tos += *sp--; goto *sw[*ip++]
_mul: tos *= *sp--; goto *sw[*ip++]
```

use macros.

```
ld:  sp += 4                 add: t1 = *sp
     *sp = tos                    sp -= 4
     t1 = *ip                     tos += t1
     t1 = t1 << 2                 t2 = *ip
     tos = [loc + t1]             ip += 1
     t2 = [ip + 1]                t3 = [sw + t2]
     ip += 2                      goto *t3
     t3 = [sw + t2]                  7
     goto *t3
        9
```

$5 \times 9 + 3 \times 7 = 66$ insns

Superoperators

combine common seqs.

op | ldop | opst

3x as many opers

ld b	→	9
ldmul c	→	10
ld d	→	9
ldmul e	→	10
addst a	→	10
		48 insns

cache tos, gnu goto

interp CISC faster than interp RISC because of dispatch overhead

again:
- 8 RISC
- 3 VAX
- 7 Pentium.

superops

ldldmul b c	→	11 insns
ldldmul c d	→	11
addst a	→	10
		33 insns

other insns

so far: all vars are [fp + x], $x \in 0..255$

movevars: offset = [fp + *ip]
offset2 = [fp + ip[0] + ip[1]]

opcode → constopnd → i.e. (*ip++ << 8) + *ip++

t1 = *ip
t1 = t1 << 2
val = [fp+t1]
ip+=1

(4)

vs

t1 = *ip
t2 = ip[1]
t3 = t1 << 10 ← t2 = t2 << 2
t4 = t3 + t2
ip += 2
val = [fp+t4]

(7)

32 bit opnd

ip |= 3
opnd = ip[1]
ip += 5

3 more insns — only fns > 256 locals

Proliferation of superopcodes
256 enough? — no.

=> use 65536 opcodes — ½ words instead

=> use expanding opcodes

case 256: goto *sp2[*ip++]
case 255: goto *sp3[*ip++]

OR

Huffman compressed

Threaded code — must be created @ load time ⇒ machine addresses

Interp (9)

- no byte codes
- use direct address of interp

foo : { foo code } goto *ip++

byte code: $t_1 = *ip$
$ip += 1$
$t_2 = t_1 \ll 2$
$t_3 = sw[t_2]$
goto $*t_3$

5

threaded: $t_1 = *ip$
$ip += 4$
goto $*t_1$

3

elim 2 insns per interp

gcc goto: 66 insns → 50
superops: 48 → 38

superops more important
- waste 4 bytes per insn!!
- but can have up to 4,294,967,296 ops

note 38/8 = 4.75
slowdown

notes: prim ops used
- array index & throw exn coded binary
- throw exn
- string ops: strcmp = one dispatch then call C-code
- all builtin (intrinsic) fns
  - call C sub as one opcode

super
primitives
- Perl regex
- sort is one opcode

Back to bytecodes:

load const : ld ∅
ld 1
ld -1 } one byte in Java

ldconst n ($n \in 0..255$)

~~superops~~ : - add a const for each

~~const pool~~
what about > 255 or < -256.
2 byte? - ldcon2
const pool
- special one byte : ld∅, ld1, ldn1, ld2
- load one byte
- load const pool.
list all consts in a fn.

call : OOcall
- by index into dispatch table.

static fn.
- many fns, indexed
256 fns in a class?
- each fn : indexed callout
from ~~co~~ const pool

one fn = 256 ~~consts~~?
either abs# or fnaddr

TAC

Interp (11)

$a = b * c + d * e$

```
t1 = b * c
t2 = d * e
a = t1 + t2
```

12 bytes (1 byte win? really?).

prob: all opnds local vars 0..255.

```
fetch:  ir = *ip++
        goto *sw[ir>>24]

add:    loc[ir>>16 & 0xFF]
        = loc[ir >> 8 & 0xFF]
        + loc[ir & 0xFF]
        goto fetch
```

elim 1 insn if put dispatch in tail

```
fetch:  ir = *ip
        ip += 4
        t1 = ir >> 22
        t2 = t1 & 0x3FC        (6)
        t3 = [sw + t2]
        goto *t3
          :

add:    a1 = ir >> 14
        a2 = a1 & 0x3Fc
        l1 = ir >> 6
        l2 = a & 0x3FC
        l3 = [loc + l2]
        r1 = ir << 2           (11)
        r2 = r1 & 0x3Fc
        r3 = [loc + r2]
        a3 = l3 + r3
        [loc + a2] = a3
        goto fetch
```

$6 + 11 = 17$ per insn
$\times 3$

---

51 for expr

clulorp$(11\frac{1}{2})$

TAC vs SM (super)

~~$t_1 = a * b$~~

~~t_2~~

*	t_1	b	c
*	t_2	c	d
+	a	t_1	t_2

ld b
* c
ld d
* e
+sT a

~~cls~~

*: l = loc[ir>>6&3FE]
r = loc[ir<<2&3FC]
loc[ir >>14& 3FC] = l*r
goto *sw [ir>>22&3FC]

$$\begin{array}{r} 14 \\ \times 3 \\ \hline 42 \end{array}$$

$\frac{42}{8} = \cancel{6}$ 5.25

a = b*c + d * e

*: r = loc[ip[0]]
tos = tos * r
t = ip[1]
ip += 2
goto *sw [t]

$9 \times 5 = 45$

$\frac{45}{8} = 5.625$

TAC modes

- what about non-reg opnds?

$\boxed{op}$ $\boxed{mode}$ opnds.....

→ imm
loc
con
⋮

- not designed for interp

```
  op = *ip++
mode = *ip++
  goto *op1 [mode &>>
op1:              goto *op2
```