compiler design

32362 characters in 5036 words on 833 lines

Florian Moser

August 14, 2018

1 introduction

1.1 definitions

programming language
a precisely defined language

compiler
program written in a host language
translates a programming language to another one

program
sequence of expressions executed by a stateful processor

expression
reads / modifies state & determines next expression
special E_stop halts the program

execution
elaboration of expressions on some machine M
M defined by software (virtual)
M realized by hardware (physical)
translated (further compiled down) or interpreted (directly executed)

1.2 formal specifications

operational semantics
abstract machine A interprets sequences of steps
effects produced by A determines meaning

denotational semantics
mathematical construct defining outcome
steps to reach desired outcome can be chosen

axiomatic semantics
rules that describe effects of operations (assertions on programs state)

1.3 compiler

1.3.1 reasons for translation

L1 faster to develop, no hardware exists
L2 more efficient/stable, actual hardware exists

1.3.2 tasks

preservation of semantics
same results for all legal inputs
illegal inputs, non functional requirements may differ

resource management
limited registers & storage, finite representation
respect execution environment (caching mechanisms)

constraint checking
ensure program can indeed be translated
inject code to preserve constraints at runtime

1.3.3 models

ahead of execution time (AOT)
compilation prior to execution
commonly used for languages without environments

just in time (JIT)
continuous compilation at runtime (optimize frequently used methods)
commonly used for languages with VM (java, c#)

1.3.4 architecture

frontend
read input, transform, check constrains

intermediate representation (IR)
compiler internal format (tree form or more complex structures)
expresses all language constructs/concepts
may consits of multiple stages & optimizer

backend
generate code respecting machine constrains

1.3.5 design

correctness
design for correctness, tradeoff with performance
if (correct program & valid inputs) behaves same as L1
else behaves in a defined way (by L1 / L2 / machine)

2 compiler built in course

specifics for the JavaLi compiler build in course

2.1 ANTRL

ambiguity resolution by taking document order
direct left-recursion (in same rule) allowed

2.2 IR (tree based)

tree determines evaluation order

2.3 code generator

child code must be generated first

registers
list of possible registers given by architecture
minimize usage by evaluating first left or first right
handle too few registers (abort, restructure, spilling)

template
defines behaviour of IR node or collection of nodes
contains code & describes bookkeeping tasks
leaves result in register & reflects machine properties

process
traverse tree, retrieving most specific template possible
emits code (by writing to buffer / file)
does bookkeeping as specified by template

const
put value in register
if too big then require frontend to throw error
else deal with it in assembly (add to .data, assemble at runtime)

var
if left delivers an address for parent
if right delivers a value

operators
map each operator to an assembly instruction
put result of operation in one, allow reuse of other

assign
move value from right to location at left

improve code generator
special instructions with const childs (incr for $+1$, shift for $*4$)
directly use memory operand instead of movl in register first

3 frontend

3.1 grammar

provide set of rules to generate strings

elements
terminals, characters (a,...,z, empty)
non-terminals, syntactic variables (A,...,Z)
start symbol (S)
set of productions ($Z \rightarrow aZb$)

L(G)
production $\Rightarrow$ as left-hand side (LHS) $\rightarrow$ right-hand side (RHS)
derivation $\Rightarrow*$ as repeated applied productions
L(G) is set of strings w, for all w there exists S $\Rightarrow*$ w

grammar type overview
context-free (type-2) if only single non-terminal on LHS
linear if context-free and at most one NT in RHS
left-linear if linear and NT at left end of RHS
right-linear if linear and NT at right end of RHS
regular (type-3) if left- or right-linear

type-2 grammar
efficient analysis techniques known
non-terminal evaluation order does not matter
can be recognised by stack machine

type-3 grammar
generate regular languages, regex
recognised by finite deterministic automata

derivations
start at S, then find next rule to get to word w ("top-down")
left-most (always pick left-most non-terminal)
right-most (always pick right-most non-terminal)

ambiguous grammar
if more than one left-most derivation can be found
avoid ambiguity by changing language/grammar or defining precedence

3.2 lexical analysis (scanner)

transform character stream into tokens for parser

token assembly
stop if character not in token anymore (like whitespace)

maximal much strategy
put into current token as much as possible
but error reporting difficult, breaks down with complex languages

couple with parser
ask parser for expected token types (top-down vs bottom-up)

3.3 syntactical analysis (parser)

prove that word is in grammar by providing steps of derivations
can choose way to parse if multiple non-terminals

employ regular expressions
define regex for each grammar token ("top-down" approach)
because exact steps to generate symbols do not matter
can be recognised by both DFA and NFA (finite state machines)

construct parse tree
as the result of the derivation, removing unnecessary information
root is S, nodes from productions, leaves from terminals

produce IR
convert the parsed syntax tree to IR
removes intermediate steps & compresses info
like E($\rightarrow$Id) - OP($\rightarrow$+) $\rightarrow$ E($\rightarrow$Id) transformed to +($\rightarrow$Var, $\rightarrow$Var)

3.4 top-down parser

start with S, work towards retrieving w

3.4.1 simple stack machine

decision based on current input & top of stack
error/accept as output actions
reduction to change symbol on stack
match to consume from input & pop from stack
bruteforce possible as words & grammars finite

3.4.2 strategies

backtracking
deterministic bruteforce variant
start with S on stack
if (top of stack & input match) then consume input
else if (can apply production) then apply production
else if (can revert & try new production) then do it
else if (can restore input & try new production) then do it
else reject

predictive
if grammar in LL(k), the next k symbols determine action
try to construct parsing table M with (x=input, y=non-terminals)
define next production for all valid top-of-stack & k next input combinations
entries which stay empty throw a parsing error, x=$and y=$accepts
if such M can be constructed then grammar not ambiguous

3.4.3 parsing table M for LL(1)

LL-regular grammar LL(1) (left-to-right scan, left-most derivation)

compute FIRST(p)
from left to right evaluate; let current be first element of RHS
(1) if current is terminal or empty, then add to FIRST(p) and stop
(2) else add FIRST(current) - e to FIRST(p)
(2.1) if e $\in$ FIRST(current) then continue
(2.1.1) if no more following elements then add epsilon to FIRST(p)
(2.1.2) else continue at (1) with the following element
repeat for all RHS variants, start with RHS fulfilling (1)

compute FOLLOW(p)
find all occurrences of LHS in others RHS
let p' be the rule with the occurrence, let match be LHS in p'
(1) if S include $in FOLLOW(p)
(2) if match at the end add FOLLOW(p') to FOLLOW
(3) else add FIRST(next) - e to FOLLOW
(3.1) if e $\in$ FIRST(next) then continue at (2) with the following element
repeat for all LHS

construct M
write terminals to x axis, non-terminals to y axis
for each production A $\rightarrow$ B with B != e write it to the FIRST(B) cells
for each B = e, write A $\rightarrow$ e to all FOLLOW(A) cells
add extra row/column with $, writing ACCEPT in the crossing

process stack
start with stack = $S, pointer to first entry, tree = S
(1) lookup table with M[entry, S], replace with S on stack & write in tree
(2) if top of stack = entry then pop, advance pointer, goto (2)
(3) else goto (1)

3.5 bottom-up parser

start at w, tries to get back to S

3.5.1 simple stack machine

decision based on current input & top of stack
error/accept as output actions
push to shift input on stack
reduce to replace top of stack with production

3.5.2 strategies

predictive
stack as scratchpad; with symbols to be used later
delay recording production until its obvious
need to know derivations of RHS to know if its production applies
add a control-symbol between each symbol to preserve context
use parsing table M[control symbol, t] to get next action

3.5.3 parsing table M for SLR(1)

LR-regular grammar LR(1) (left-to-right scan, right-most derivation)
create LR(0) sets with FSA automaton
build simplified LR(1) parser (SLR(1)) parser

construct FSA
dot . = processing position, augment grammar with S' $\rightarrow$ .S
(1) create first state of FSA as S' $\rightarrow$ .S
(2) compute closure by following NT after dot (S $\rightarrow$ .A, A $\rightarrow$ .aA | .b)
(3) create all possible transitions (a produces A $\rightarrow$ a.A), goto (2)
give all n possible states a name I_k

create table
compute first & follow, number states & productions
states to x axis
actions (terminals+$) & gotos (non-terminals-S') to y axis
for terminal transitions, write Shift:#state to corresponding cell
for non-terminal transitions, write #state to corresponding cell
for states with S' $\rightarrow$ S. write accept to $column
for states with A $\rightarrow$ b., write Reduce:#production to FOLLOW(A) column

process stack
start with stack = <0> & set pointer to first entry
(1) tablelookup with state (right-most control symbol) & entry
(2) if shift then write entry + <shift_state> then goto (1)
(3) if reduce then record reduction X$\rightarrow$Y & remove all from stack

(3 cont.) tablelookup with state & X, then write X + <goto_state>

3.6 parser varia

generators can create LR(0,1) items, LL(1..) parsers

canonical LR (CLR)
uses LR(1) items (lookahead of one)
may resolves ambiguities, but FSA needs more states

lookahead LR (LARL)
same number of states as SLR(1) but equally powerful than CRL

limitations of context-free grammars
want to find errors with parser
L1 = { a c a | a ∈ {a, b}∗} for c separate body/definition
L1 could find undefined variables, but not context-free
L2 = {aˆn bˆn} for a definition and b calling site
L2 could check if method invocations are proper, but not context-free
use semantic analysis for that

4 semantic analysis

check properties (types & constraints)
perform transformations (casts, default values, initializers, ...)

4.1 attribute grammar

context-free grammar extended with context-sensitive info

4.1.1 attributes

attached to non-terminals
hold value
synthesised (from children) or inherited (from parent/siblings)

notation
name non-terminals of production, possibly with index (<P>, <P_0>)
can add properties functioning like global variables (<A>.N)
can specify calculations (<A_0>.N = <A_1>.N + 1)
can specify conditions (iff <A_0>.N == <A_1>.N)

4.2 symbol table

repository of program symbols
nesting may hides symbols (variable hides field)
enables checks for classes, fields, methods, variables

4.2.1 structure

global symbol table
contains class entries
contains constants such as true/false, void, ...

class entry
fields (size, location (offset from this))
methods (location)
base class

method entry
parameters (order, type)
variables (size, location (offset from frame pointer))

debugging
definition location

4.2.2 construction

walk over AST, create entries

field/method references
implicit "this" reference if not specified
explicitly specified class reference

perform checks
no duplicates (class / methods)
no undefined fields/method/classes (resolve reference)
valid inheritance (no cycles, default extend Object)
valid type (parameter passing, variable assigning)
valid type conversions (add int + float = float)
valid array access ($0 \leq i < A.length$; might needs intermediates)
defined program start (main method)
no unused variables / fields
functions return value on all execution paths
protection system respected (private/protected access)
interface/abstract correctly implemented

delayed checks
checks which can't be performed by simple analyzer
model execution checks, runtime checks, undecidable

4.3 delayed checks

model execution checks
variables initialized before usage
exceptions are caught
return statement is reachable
end of program is reachable
methods are called at least once
time/power policies respected

undecidable checks
loops/program terminates

runtime checks
array out of bounds access
implicit null checks by pointing to low & protected memory region
triggers protection error (SIGSEGV); VM catches & throws exception

5 software engineering

5.1 patterns

strategy
encapsulate set of related algorithms into class hierarchy
interface (IStrategy) passed to context (IConsumer)
ConcreteStrategy implements IStrategy

specialization
create new child class with more specific functionality
child inherits parent class, overriding or adding methods/fields

visitor
performs a double invocation (caller/callee determined at runtime)
couples data structure w/ unrelated functionality (IElements)
with access methods w/ unrelated operations (IVisitor)
IVisitor with visitMyElement implemented by MyVisitor
IElement with accept(IVisitor) implemented by MyElement
MyElement implements accept() { visitor.accept(this) }
MyVisitor has method for each IElement implementation
the visitor or the element traverses (possibly tree) structure

observer
presentation (IObserver) is notified about change in data (ISubject)
ISubject implements attach(Observer), detach(Observer), notify()
IObserver implements update()
subject calls update() upon change; then observer gets state of subject

collection extensions
provide hook for externally defined functions for easy extensibility
high order function f (named such because passed as an argument)
MyCollection provides hook(IFunction f) which applies f to all elements
MyFunction implements IFunction with map(IElement)

5.2 software quality

general approach
identify quality dimensions
set priorities, discuss tradeoffs
establish objectives (time, cost, resources, performance)
perform measurements to fulfil objectives

quality assurance
follow guidelines of software construction
pursue appropriate testing strategy
informal reviews, formal reviews, external audits

internal characteristics
primarily interests developer
flexibility (can be modified for unintended environments)
reusability (can be used in another execution environment)
testability (test if requirements are fulfilled)
maintainability (address changing needs easily)
readability (consistency, conventions, structure, used constructs)
understandability (separation of concerns)

external characteristics
primarily interests user
correctness of specification, design, implementation
robustness to handle invalid input / environment
usability (fit for purpose)
adaptability (incorporates with other systems)
integrity (access control)

traceability (access traces / logs provided)
accuracy (results as expected)
reliability (performs under assumed environment)

5.3 informal reviews

execute tests & understand source code to catch errors early
explicitly not to assign blame or evaluate staff/project

5.3.1 walk-through

devs present system

5.3.2 code reading

feedback via comments, done by co-dev

5.3.3 inspection

reviewers look at code, then meet with dev

roles
moderator (by technical person, guides discussion)
author (by author, explains decisions & gives overview)
reviewers (point out good/bad & possible problems/defects)
scribe (possibly by moderator, records discussion)
management is absent

process
planning (code sent to moderator, selects reviewers)
setup (short overview given by author)
preparation (code & documentation is read)
meeting (factual questions are discussed)
report (scribe publishes report)
follow-up (moderator assigns problems to repair)

5.4 openJDK example

review process
develop & test code
upload patch to server
inform mailing list of patch
reviewers inspect code & give feedback
push change set (code + description + reviewers)

roles
higher accepted changesets (#) unlocks new capabilities
(0) contributor (has signed contribution statement)
(2) author (has own username & listed on project webpage)
(8) committer (push-right & sponsors others change sets)
(32) reviewer (decides if code is good enough)

6 code generation

primary requirements are correctness & performance

6.1 key activities

code selection (decide assembly for IR nodes)
code scheduling (determine order of execution)
register allocation (decide which operand in register)
register assignment (decide which register used)

6.2 x86 rules

one operand may resides in memory or intermediate
6 int registers available

calling conventions
caller preserved %eax, %ecx, %edx
callee preserved %ebx, %ebp, %esp, %esi, %edi
callee must restore %ebp %esp

6.3 assembly

assume a in %eax, b in %ecx

```
movl $2, %eax # save 2 to b
movl %eax, %ecx # save a to b
addl %eax, %ecx # b = a + b
incr %eax, # a++
notl %eax, # a = - a - 1
xchg %eax, %ecx # temp = a; a = b; b = temp
imull %ecx, # a = b*a (result in edx:eax, each 32 bits)
imull %eax, %ecx # b = a*b (32 bit result)
cmp $2, %eax # set 0 flag if equal
je LABEL_1 # jump if cmp before equals
jg, jge LABEL_1 # jump if first greater (+equal)
jl, jle LABEL_1 # jump if first lesser (+equal)
jmp *(%eax) # indirect jump
jmp (%eax) # jump to address
LABEL_1:
call LABEL_1 # push address of next instruction to stack, then jump
```

6.4 sample assembly

```
push %ebp # preserve %ebp
mov %esp, %ebp  # top of the stack is now in %ebp
sub $16, %esp # make stack frame bigger
push $0xDEADBEEF  # push stuff on stack; increasing %esp+4
mov 4(%ebp), %eax  # get 1st passed parameter %ebp
mov %ebp, %esp # restore original %esp
pop %ebp # restore original %ebp
ret # returns (equivalent to pop %eax jmp *%eax)
```

6.5 operand access

constant
put in register or use directly
enforce size contains

fields
reference of object + offset from symbol table
use resulting memory location to read/write

arrays
reference of array + blocksize $*$ index
row-after-row called row-major (x[0][0], x[0][1], ...)

access
evaluate expressions & in intermediate
evaluate from left to right

6.6 assignment statement

assign to variable, field, array
handle register shortage; preserved on stack & override

6.7 conditional statement

if-then
eval/test condition
if true then continue, else jump to end

if-then-else
eval/test condition
continue if true then jump to end, else jump to else

6.8 loop statement

while expr
eval/test condition
if true then continue then jump back, else jump to end
but one unconditional branch for each expression

while expr v2
jump body
eval/test condition
if true jump back to body

6.9 method invocation

call site from caller invokes callee on target with actual arguments

calling conventions
decides what caller/callee prepares
decides where date, return address, return value are saved

call actions
identify target & starting address of callee
handle parameters (evaluate, push into stack)
save caller saved registers
find space of temporaries
determine & store return address
find space for return value
set up activation record for callee
transfer control to callee

return actions
restore registers
transfer control to caller
deliver return value
remove activation record of callee

6.10 heap

eats up, from low to high adresses
contains object instances

6.11 stack

eats down, from high to low addresses
setup by run-time system, some assembly instructions

structure
frame/base pointer fp determines start of frame (high address, low)
stack pointer sp determines end of frame (low address, high)
return value, parameters + target, return address (caller)
old fp, old sp, locals, temps (callee)

access stack elements
offsets persisted in symbol table
access parameters with fp + offset (go to caller area)
access locals with fp - offset (inside callee area)
access temps with fp - offset (can grow indefinitely)

6.12 object layout

object referenced by pointer p
p + 0 pointer to v-table
p + 4,... the corresponding fields

v-table
created completely for each class
method order in same order as base class
code location same as base class if not overridden

call method
dereference object pointer (movl (%eax), %eax)
dereference v-table at this + 0 (movl (%eax), %eax)
call code location form v-table offset (call $*4$(%eax))

7 data flow analysis

correct & accurate for all possible inputs (conservative)

7.1 applications

reason about correctness (ensure value assigned)
enable optimizations (compile-time expression evaluation)
useful for tools (find all references of symbol)
remove unused code

7.2 control flow graph (CFG)

constructed over single method
basic box is executed as a unit

construction
create blocks (started by label, ended by return/goto/condition)
reformulate if statements to "TCond# = condition; if (TCond#)"
connect blocks which are executed after one another
name blocks in order like B#
add ENTRY block (pointing to first block)
add EXIT block (successor of final blocks like returns)

analysis
local, inside basic block (like subexpression elimination)
global, over CFG (like null checks)
inter-procedural (across methods/functions)

point
after specific statement s1, before s2
after specific basic block b1, before b2
make claims at specific points

paths
for all pairs xi, xi+1 it holds one of these two
(a) xi is point before b, xi+1 is point after b
(b) xi is point after b, xi+1 is point before connected b2
infeasible to process all (possibly infinite) paths

7.3 side effects

never
local variables / parameters
field assignments (if no dereferencing needed)

in expressions
null-dereference
non-pure method evaluations
invalid array access

7.4 data flow framework generalization

D as direction of data flow (forward/backwards)
V domain of values (possible results + bottom & top)
|><| meet operator (operating on lattices)
F transfer function (process block)

termination
stop as soon as nothing has changed anymore
show that max number of transitions per element exists

7.5 constant propagation

replace constant variables with its values

process in block
set all variables to bottom
go to specific value if const assigned
else go to top

transfer
overapproximate reverse-T $\rightarrow$ exact $\rightarrow$ T

evaluation
directly place constants in code

7.6 reaching definitions

check if/where variable is defined which is used in statement
d reaches u if d is not killed on the path to u

process in block
create table, for each variable all definitions
gen by taking last definition in block of variable
kill with all other definitions of variables in gen
overapproximate gen = {}, kill = {all}

transfer
IN(B) = { d reaches B } = U OUT(predecessors B)
OUT(B) = { d leaves B } = gen_b U (IN(B) - kill_b)
start with OUT(start) = empty, overapproximate with all
iterate by determining IN for all, then OUT for all, repeat

evaluation
directly place expression of definitions in block
if d2 $\in$ IN(B) can replace d2 value at its usages

uninitialized variables
let OUT(start) = { one fake definition for each value }
if variable is used with fake definition in IN(b), then possibly undefined

7.7 live variable

check if variable is used after some assignment
dead variables need not to be stored / computed

process in block
def set with all var defined in B prior to any usage
use set with all var used in B prior to any definition
overapproximate def = {}, use = {all}

transfer
OUT(B) = { var live after B } = U IN(successors B)
IN(B) = { var live before B } = use_b U (OUT(B) - def_b)
start with IN(end) = empty, overapproximate with all
iterate by determining OUT for all; then IN for all; then repeat

evaluation
if val not in OUT(B) do not need to store/compute it

local analysis
start with OUT(B) as set of live variables
then for each statement add new live variables

use-def / def-use chains
for use of variable, persist all possible definitions
for definition of variable, persist all possible uses
do not forget loop back uses

7.8 available expressions

available if evaluated on every path and no breaking definition afterwards
breaking definition if any element is redefined
but allowed to yield different values depending on path

process in block
gen set for evaluated expression, no breaking definition
kill set for breaking definition & no following evaluation
overapproximate gen = {}, kill = {all}

transfer
IN(B) = { expr reaches B } = intersect OUT(predecessors B)
OUT(B) = { expr leaves B } = gen_b U (IN(B) - kill_b)
start with OUT(start) = empty, overapproximate with all
start with OUT(other) = all, overapproximate with none
iterate by going top to bottom, IN then OUT; then repeat

evaluation
if expr in IN(B) then do not need to recompute

8 register allocation

8.1 core tasks

which operand resides in register (map operands → virtual registers)
which register to use for operand (map virtual register → register)
reduce complexity by assigning operands first to virtual registers
in second step assign the virtual registers to physical ones

8.2 live ranges

range where virtual register is live (pending use)
can compute liveness of VR like any normal variable

calculation
instructions as list, for each row compute L and R
L liveness (not-killed variable v used in current/following rows)
R reaching (not-overridden definition d_v made in before rows)
live range where v ∈ L intersects with d_v ∈ R
multiple d_v for same v possible → multiple live ranges possible
combine liveness with reaching to decouple different definitions

granularity
starts at left of assignment, ends on right of other
or whole statements (min-length therefore 2 lines)
or whole basic block (min-length therefore 2 blocks)

interference
if ranges overlap, then interfere with each other
can model as graph; nodes are live ranges, edge iff overlap

8.3 graph coloring

k-colorable if k colors enough such that none touch
compilers try several K-colorings, then choose the best one

K coloring kempe's algorithm (simplification step)
push nodes with #neighbours < K on stack & remove from graph
until ≤ K nodes left (may not be possible → graph surgery)
color the remaining nodes (trivial bc #neighbours < K-1)
now pop from stack, append to graph & assign color
but not guaranteed to succeed

graph surgery
K coloring may still possible, but hard to find
if not found then heuristically "spill" (remove) node
repeat until
all #neighbours < K requirement reached
then continue kempe's algorithm

pick spill victim heuristics
at random, lowest cost estimate, lowest use count
estimating spill cost by estimating basic block usage
heuristics (loops execute 10 times), runtime info (if JIT)

8.4 spilled variables

variables residing in memory bc not enough registers
constraint that at most one operand can be in memory

keep spare register
spilled variables put into spare register upon usage
hence enforce a K-1 coloring (or K-2 if both must be in registers)

introduce temporary
introduced for each occurrence of spilled variable
filled before instruction with value from stack
lives only for two lines (load & use)
rewrite IR accordingly, rebuild graph and retry coloring

splitting
reduce spilling (temporary loads) by splitting live ranges
heuristics to know where to split

8.5 color improvements

coalescing
remove extra copy instruction if not needed
(last usage of v1 assigned to v2 → could possibly rename v2 to v1)
coalesce by combining nodes (but maybe K coloring then impossible)
copy property edges by marking them to attempt to use same color

pre-color
depending on instruction specific registers must be used (like imull)
per-color operands; not removed during simplification

8.6 phase coupling

code selection (imull instruction)
code scheduling (determines order of excution)
register allocation (bounds #operands in registers)

9 multiple inheritance

9.1 which base symbol to use

error if no guidance provided
add parameter which defines which base class is used
require cast to either A or B
use order of extends clause
invoke both methods / randomly select one
force symbol renaming
prohibit MI usage

9.2 diamond problem (AB extends A,B; A/B extends X)

referencing X symbols ambiguous
local changes to class may breaks behaviour at other places

9.3 object layout problems

example
AB extends A,B { a, b, method() {}}
B { b, method() {}}
A { a, method() {}}

overriding fields
offset of base classes may not matches to compiled functions
therefore can not share v-tables
choose to keep code of A (fields of A with same offset)
recompile B methods (bc fields have different offset)

casts with v-table duplication
offset of methods does not match, need to divide object layout
first part like A layout, referencing to AB v-table
second part start with reference to B' v-table, then the fields
when casting, offset this pointer (-0 if statically A/AB, -8 if B)

casts with pointer adjustments
add additional this offset entry in v-tables
no code recompile needed anymore
instead adapt this pointer as needed before invocation
but runtime performance suffers (at every method call)

9.4 C++

can extend with public virtual X
virtuals constructed only once in object hierarchy

10 attacks

10.1 control flow highjack

highjack control flow; continue at attacker specified location
possibly arbitrary machine code or reusing existing code
corrupt code pointers (return address, function pointer, vtable)

10.1.1 defences

non-executable data (NX)
data execution prevention (DEP)

10.1.2 return address

"ret" implicit "jmp return pointer"
try to corrupt return pointer (point to attacker location)
must reuse already existing code (because of NX/DEP)

return oriented programming (ROP)
use code snippets ending with ret, chain them together
put on stack address1, value, address2, value, ...

address space layout randomization (ASLR)
mapping program addresses to hardware
randomize hardware addresses (stack, heap, memory base)

10.1.3 stack canaries

adds data before %epb/return address
if data is overwritten; then buffer overflow happened
but effectiveness depends on secret & runtime slower

10.1.4 powerful attacker

no data execution / code corruption (NX/DEP)
but arbitrary data modification assumed

control flow integrity (CFI)
ensure control flow stays within CFG
bc attacker needs to change CFG to execute arbitrary code
but too conservative, needs recompilation

dynamic CFI
battles drawbacks of compile time CFI
constructs CFG at run-time, supports dynamic libraries

11 Java HotSpot VM

11.1 java execution architecture

hardware, OS, JVM, Java SE/EE, Spring, user applications

why VM
non-VM languages reimplement compiler, libraries, debugger for each OS
VM implemented for all OS/architectures, more portable code

compilation
java/scala/javascript/python to bytecode (ahead of time, using javac)
bytecode executed on abstract stack machine (by VM)

execution
template based interpreter maps bytecode to machine code sequence
JIT compilers produce machine code from bytecode
both compiled/interpreted access same stack/heap
garbage collector manages heap

JIT compilers
C1 (limited optimizations, but fast compilation & small footprint)
C2 (aggressive optimizations, but high resource demands)

compile methods
because compilation expensive specific appropriate methods to compile
interpreters collects profiling infos about methods
aggressive & optimistic compilation by C1/C2 stored in code cache
if guards assumptions proven wrong then interpreter takes over again

example optimization
inline method code; guard ensures object of the expected class

tired compilation
compile interpreter (fast start), C1 (fast compile), C2 (fast code)
collect 100 samples with interpreter, then execute C1 optimization
then collect 1000 samples with C1-compiled code, then C2 optimization
but then naive code cache inefficient (high fragmentation, bad locality)

11.2 code cache

continuous chunk of memory, stores code generated by JIT compilers

types of code
non-method code (small, immortal livetime)
C1 profiled code (medium, limited life time)
C2 non-profiled code (large, long lifetime because expensive compilation)

segment
divide code cache into non-profiled, profiled, non-methods
better cache locality (ITLB misses decreased)
sweeper (GC for code) faster because optimized for each segment

11.3 compact strings

reduce memory footprint of chars (may occupy up to 45% of data)
save as latin-1 (1 byte) instead of UTF-16 (2 bytes) (best effort)
changes GC (deduplicates strings), String (payload byte array)

11.4 ahead of time compilation

compile to native code prior to VM launch (improves startup)
compiled code shared between VMs (C1 performance)

11.5 value types

immutable, identity-less aggregates (user defined primitives)
no identity, all fields final, no subclassing

advantages
better locality (no indirection, no header)
relaxes heap pressure due to stack allocation (less GC)
new possible optimizations (like scalarization)

minimal value types (MVT)
subset of value types, meant as early access
denote class with Value Capable Class (VCC) annotation
then JVM derives value type class (DVC)
new value type bytecodes / method handles

storage formats
on heap, with header
in thread local value buffer (TLVB) with header
scalarized by JIT code without header on stack/registers)
flattened array/field with type infos stored in container header

flattening
remove references by inlining value types
for fields (only non-static)
for arrays (respect long alignment)
GC needs support to find references in inlined value types

optimizations
copy detection (reuse existing)
passing flattened value type (passing fields as arguments)
return flattened value type & class
method handles need additional handling

explorations
currently only limited set of unit tests
check performance with runtime checks (legacy code, MVP code)