Compiler construction: Set 3

Static binding
= compile time
Dynamic binding
= runtime

Method resolution is dynamic in resolving
which class's methods are needed.
Searching up the class hierarchy as required.

Method resolution is static in resolving
overload of methods based on signature.

Symbol table questions:
1) How store the members of a class?
   - variables
   - methods
   Are a hash table specific to each class
2) How to store overloaded methods within a class?
   \[ a \cdot m(3, 4, 7, 12, 8) \] ; hashing id + signature
   Fails: too many possibilities
hash them to a list of possibilities

ASTs: goals for design
1) ability to unparses to a program equivalent to original program.
2) hide implementation details: use getters to access data within classes.
3) API changes for different compilation phases.
   new methods: `unparse()`, `SemanticCheck()`, `generateCode()`

ST modifications: type, kind

variables + constants + literals

- R-value
- L-value

\[ a = b + 10 \]

- R-value
  - L-value
  - address

C: derive the L-value of a variable by the operator & (referencing)

derive an R-value from an L-value by the operator * (dereferencing)

for code generation:

- L-value is stored in ST
- R-value requires generated code to access, might add to AST a dereferencing node.
Hard example:
\[ a[b] = e \]

Control stack:
- R value
- L value (static)
- L value (dynamic)
- Base
- Offset (Static)

\[ m() \rightarrow j() \rightarrow g() \]

Java Virtual Machine
- bytecode
- Jasmin assembler

```
CSX-J9D
```

Compiler → assembler code → Jasmin → class file → java

bytecode:
- compact: 0-address
- implicit stack
- safe: stack size errors
- type errors

Disassembler:
- java: runs program (class file)
- javac: compiles source code → class
- javap -c: disassembles a class file
Class file has several sections

code

Constant pool

values and ands to fields in classes
associated type

16-bit indexed: each value.ptr uses 1 index

types are represented by strings

byte B  short S  char C
int I  void V
float F  array of int [I
bool Z  array of type [C
double D  array of type [D
long J  reference to type [C

+ arithmetic: all stack entries are 32-bit quantities

* add: pops 2 ints, adds, pushes result

* add: adds 2 longs, each is stored in
  2 consecutive stack cells

* mul: the arguments
  and the result

registers: 32 bits

as many as you need

first ones: parameters 0, 1, 2, ...

static method: no instance

dynamic method: "this" is in register 0.

next ones: general purpose for local variables

64-bit quantity, use 2 adjacent registers

even, odd numbered registers.
load: register \rightarrow stack
store: stack \rightarrow register

load: i\text{load} \text{ reg0}

store: i\text{store} \text{ reg0}

fields (global variables in C++, fields in Java)

static get static name \text{ type} \rightarrow L\text{ java/io/PrintStream}

dynamic

java/lang/System/out

\text{from field to stack}

put static name \text{ type}

\text{from stack to field}

get field (instance \rightarrow stack) name \text{ type}

Put field (stack \rightarrow instance) name \text{ type}

reference to the instance is popped from the stack

getfield before ref:to object after value \rightarrow 1 \text{ or 2 calls depending on type}

putfield \text{ value} \rightarrow

to place value of instance field into a local variable: getfield, then store.
type conversion: pop old value, push converted value

i2f : integer to float

Calling a method

static :

*invokeStatic* 

java/lang/Math/pow (D D) D

dynamic :

*invokeVirtual* 

ref to object popped from stack

java/lang/System/Out#println (I)

call foo (1, 2, 3, 4)

return value

1 4

2 3

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

type of string:

L java/lang/String;

Java arrays are instances of an Array class.

special instructions:

iaload : load an integer from an int array

index

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

iasstore : place an integer into an int array

index

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

related : f, d, l, b, s, for basic types

Example:

\[ a[4] = 3 \]

local variable, in reg. 7

[load 4]
[ldc 4]
[ldc 3]
[iasstore]
array of class, such as array of String,

```
aload
astore
ref
array

index
value: ref to the item in the array
ptr to the array
```

new array type

```
length
```

e-ref to new array

Branching: 3 byte: opcode distance

```
5 byte: 1B 2B 4B
```

goto (loc)

conditional pop an integer

```
ifgt (loc)
ifeq
iflt
ifge
ifle
ifne
```

conditional based on 2 values, both popped.

```
ifeqcmpgt (loc)
```

Symbol tables (SST)

Static scoping: scope of an identifier is known to compiler. nested scopes with inheritance, with hiding to organize SST

1) list of STs for currently active scopes.
2) unified ST with a field for level.

b) elsewhere: to close a scope.

Specific concerns of the source language:

- records (structs, classes)
  - can be nested
    - the type has components
      1) record
      2) ST of its fields
      3) length (in bytes)

- explicit named types entered into ST
  - name, type, kind = type
  - variables of the named type:
    - ST entry just points to the type entry in ST.

Pascal:

```pascal
type r = record a: int; b: char end;
var v: r;
```

ST:

```
| a   |
| b   |
| c   |
```

Hash table:

```
| a: int, offset 0 |
| b: char, offset 4 |
```
- Overloading: multiple competing declarations of the same identifier within a scope can be handled by building a list of all overloaded meanings.
- Altered search rules
  - Pascal with statement
    
    \[\text{with } n \text{ do }\]
    \[\begin{align*}
    & n.a := 7 \\
    & a := 8 \\
    \end{align*}\]
    
    end;
  - Push a new scope onto the ST list
    - Scope of the record's type.
    - Place a temporary value for "this" in ST.
- Forward references
  - 2-pass compilation (2 passes through AST in checking semantics)
    - Check as a scope is closed that all promises are fulfilled.

---

**Type descriptors**

- Easy: int, long
- Others: arrays (element type, range, index type)
- Records (see above)
- Pointer (base type)
- Difficult: dynamic-sized arrays.
  - Type descriptor must be available at runtime.
Semantics checking during checking, ST can be updated, AST can be modified.

Other semantic checks
1) type consistency
2) reachability
3) exception handling
Reachability

\[ S_s \]

isReachable \((i)\) terminates Normally \((t)\)

if either branch has \(t\).

\[ S \rightarrow S \]

null statement

return

break

continue

\[ S \]

if true/false

while

while true

while false

int \(v\); \(v = 10\)

error!
Exceptions

```java
try {
    // throws e1, e2
    3 catch (ExceptionType e1) { 
        throws e1
    } catch (e2) {
        throws e2
    } catch (ExceptionType e2) {
        method(...) throws ExceptionType
    }
}
```

1. must declare all exceptions a method can throw.
2. all exceptions declared as thrown must be possible.
3. a catch may only mention a possible exception.

**Implementation:**
- keep track of all exceptions S can throw,
- at end of method, verify that the exceptions thrown inside the method are covered by the declaration.
- a catch block removes exceptions from the set, can add to the set.
for B ∈ S

code: l2: B
    if false goto l1
    S
    goto l2

l1:

goto l1;
l1: S
l1: B
    if true goto l2

Switch statement

- Few cases: pretend it is if
- Dense cases: use jump table
- Otherwise: search method (binary search, tree, hash table)

Runtime organization:

- Each procedure invocation gets a new stack frame on the stack.

Heap

base

local variables

main

A

B

B