COMP 520 - Compilers

Lecture 16 (March 30, 2017)

Runtime organization of object oriented languages

• Reading for today
  – PLPJ Chapter 6: secn 6.7
  – Also need to know: code generation, chapter 7
Today’s topics

- Review of miniJava classes without inheritance
  - mJAM representation of objects
  - location in memory

- mJAM virtual machine
  - execution of binary object code (example)
  - debugging capabilities

- mJAM support for classes with single inheritance
  - representation
  - mJAM support

- Related issues
mJAM memory organization

- Two separate memories
  - Code store
    - compiler-generated program is loaded into code segment
    - predefined runtime functions are located in the primitive segment
    - TAM can not write into code store
  - Data store
    - static constants and variables are loaded into global segment
    - procedure invocation and expression evaluation uses execution stack
      - expands downwards
    - dynamically allocated values are allocated on the heap
      - expands upwards
      - memory for deleted values can be reused
- ABI defines fixed addresses and usage conventions
  - various locations in memories are accessed relative to machine registers (CB, SB, HT, etc.)
miniJava: simple classes, no inheritance

• Classes
  ```java
class A { int x; void p() { x = 3; } }
  ```
  – runtime entity descriptions in AST
  • class A: $S_A =$ size of class A (# fields)
  • field x: $d_x =$ displacement of field x
  • method p: $d_p =$ displacement of code for p

• Objects
  – objects are created on the heap: $A \ a = \text{new} \ A()$
  – let $d_a$ be displacement of local var “a” in activation record

mJAM runtime layout

 activation record on stack

object instance in heap

reserved

$d_x \rightarrow x$

$L_B \rightarrow d_a$

$S_A \rightarrow$
mJAM: runtime support for simple classes

- mJAM code sequences

A a = new A();  // object creation
a. x;            // qualified reference
a. p();          // method invocation
x = x + 3;       // field upd within p()

LOADL -1
LOADL SA
CALL newobj
STORE da[LB]

a.x;
LOAD da[LB]
LOADL dx
CALL fieldref

a.p();
LOAD da[LB]
CALLI dp[CB]

x = x + 3;
LOAD dx[OB]
LOADL 3
CALL ADD
STORE dx[OB]

reserved
-1 | SA

OB
| call instance
| [ within p() ]

LB

d_a

object instance in heap

activation record on stack

"this"

miniJava runtime organization
Example: mJAM execution of a miniJava Program

• See pdf
  – 16a-Counter-Trace-mJAM
Classes with single inheritance (Java)

- **Class hierarchy**
  
  ```java
  class A { int x; void p() { ... } }
  class B extends A { int y; void p() { ... } void q() { ... } }
  ```

  - inheritance hierarchy
    - “class B extends class A”, or “B is a subtype of A”

  - **fields**
    - fields of B extend the fields of A
    - runtime layout of fields in A is a prefix of the runtime layout of fields in B

  - **methods**
    - methods of B extend the methods of A
    - methods of B can redefine (override) methods of A
Static and dynamic type with single inheritance

- **Object type**
  - **static type (declared type)**
    - used by compiler for type checking
      - determines accessible fields and available methods on objects
      - type rules for assignments
        » assignment: type of RHS must be a subtype of type of LHS
        » method call: type of arg \( i \) must be a subtype of type of parameter \( i \)
  - **dynamic type (run-time type)**
    - generally only known at runtime
      - part of the representation of an object
        » initialized at time of creation from object constructor
      - dynamic type is always a subtype of the static type (guaranteed by type system)
      - dynamic type determines which method is invoked (method override)

- **examples**
  ```java
  class A { int x; void p(){ ... } }   class B extends A {
    int y;
    void p(){ ... }
    void q(){ ... }
  }
  A a = new A();  B b = new B();
  A c = b;   B d = a;
  a. p();   class B extends A {
  b. q();
  c. p();
  ```
mJAM representation of single inheritance

```java
class A { int x; void p() { ... } }
class B extends A {
    int y;
    void p() { ... }
    void q() { ... }
}
```

- runtime entity descriptions in AST
  - class A: $S_A =$ size of class A
  - class A: $d_A =$ displacement of class descriptor for A
  - class B: $S_B =$ size of class B (including size of class A)
  - class B: $d_B =$ displacement of class descriptor for B
  - field x: $d_x =$ displacement of field x in A and B
  - field y: $d_y =$ displacement of field y in B
  - method p: $h_p =$ index of method p in A and B
  - method q: $h_q =$ index of method q in B
  - method p in A: $d_{p[A]} =$ displacement of code for p() in A
  - method p in B: $d_{p[B]} =$ displacement of code for p() in B
  - method q in B: $d_{q[B]} =$ displacement of code for q() in B
Classes with single inheritance

• mJAM runtime layout

```
SB
\text{d}_A \uparrow^h_p \downarrow^h_q \quad \text{class descriptors in global segment of data memory}
\text{p: } d_p[A]
\text{p: } d_p[B]
\text{q: } d_q[B]
LB
\text{a: }
\text{b: }
\text{d}_B \uparrow^h_p \downarrow^h_q \quad \text{d}_A \mid S_A
\text{d}_B \mid S_B
x
y
```

Runtime organization for OO languages
Classes with single inheritance

- mJAM code sequences (only changed sequences are shown)

A a = new A();
  (object creation)
  LOADL d_{A}
  LOADL S_{A}
  CALL newobj
  STORE d_{a}[LB]

a. p();
  (dynamic invocation)
  LOAD d_{a}[LB]
  CALLD h_{p}

```plaintext
p: d_{p}[A]
p: d_{p}[B]
q: d_{q}[B]
```

```plaintext
a:

b:
```

```plaintext
d_{A} | S_{A}
```

```plaintext
d_{B} | S_{B}
```

```plaintext
d_{x} -> x
```

```plaintext
d_{y} -> y
```
Related issues

• single inheritance
  – type operations
    • instanceof
    • casting
  – super() superclass constructor invocation

• multiple inheritance
  – we lose the prefix property of runtime layout!

• optimization
  – dynamic method dispatch has high cost
  – converting dynamic to static calls

• dynamically loaded classes
  – Java loads classes on demand, hence cannot use simple representations such as used by mJAM