COMP 520 - Compilers

Lecture 16 (April 2)

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: $S_A = \text{size of class A (\# fields)} = 1$
    - field x: $d_x = \text{displacement of field } x = 0$
    - method p: $d_p = \text{displacement of code for } p = ?$

- **Objects**
  
  - objects are created on the heap: `A a = new A();`
  - let $d_a$ be displacement of local var “a” in activation record

  ![mJAM runtime layout diagram](image)

  object instance in heap

  activation record on stack
mJAM: runtime support for simple classes

- mJAM code sequences

\[
\begin{align*}
\text{A } a &= \text{ new A();} \\
\text{(object creation)} &
\end{align*}
\]

\[
\begin{align*}
\text{a.x;} &
\end{align*}
\]

\[
\begin{align*}
\text{a.p();} &
\end{align*}
\]

\[
\begin{align*}
x &= x + 3;
\end{align*}
\]

\[
\begin{align*}
\text{LOADL } -1 &
\end{align*}
\]

\[
\begin{align*}
\text{LOADL } SA &
\end{align*}
\]

\[
\begin{align*}
\text{CALL newobj} &
\end{align*}
\]

\[
\begin{align*}
\text{STORE } da[LB] &
\end{align*}
\]

\[
\begin{align*}
\text{LOADL } dx &
\end{align*}
\]

\[
\begin{align*}
\text{CALL fieldref} &
\end{align*}
\]

\[
\begin{align*}
\text{LOADL } dx[OB] &
\end{align*}
\]

\[
\begin{align*}
\text{LOADL } 3 &
\end{align*}
\]

\[
\begin{align*}
\text{CALL ADD} &
\end{align*}
\]

\[
\begin{align*}
\text{STORE } dx[OB] &
\end{align*}
\]

activation record on stack

object instance in heap
Interfaces and classes

• interface
  – specifies methods (name, signature) required of an implementation
    

```
interface List {
    ...
    add(Object x);
    ...
}
```

  – is a type (can be used in type declarations)
    

```
List a = new ArrayList();
```

• class
  – implements one or more interfaces

```
class ArrayList implements List {
    ...
    add(Object x) {
        ...
    }
}
```

  – is a type
    

```
ArrayList a = b;
```

  – has a constructor
    

```
ArrayList a = new ArrayList();
```
inheritance

• a class
  – can **implement** many interfaces
  – can only **extend** (inherit) one class
    • when a class extends a superclass, it inherits an implementation
    • inherited methods can be overridden

• an interface
  – an interface can extend another interface
    • it just adds additional requirements, there is no implementation
static vs. dynamic types

- Variables and expressions have a static (compile-time) type
  - derived from declarations
  - applicability defined by scope rules
  - known at compile time, without running the program
  - does not change

- Every object has a dynamic (run-time) type
  - obtained when the object is created using new
  - dynamic type can be any subtype of the static type
  - dynamic type can depend on inputs and is undecidable, in general
run-time dispatching of overridden methods

• required for objects
  – when dynamic type specifies an overridden method

• not needed for interfaces
  – interfaces cannot be instantiated (with new)
  – so static type is always equal to dynamic type
  – and compiler can work out correct method to invoke at compile time
Runtime type safety

- Various memory errors can occur (possibly undetected) in C programs
  - Accessing an array element outside of the array bounds
  - Casting a pointer of one type as a pointer of another type
  - “segmentation fault” … attempt to access a memory location not in the process address space
  - Accessing a deallocated memory location, hence value is unpredictable
  - Failing to dispose of unused values in memory, causing memory leaks and eventually exhaustion of the process address space.

- These memory errors are precluded (or at least detected) in Java
  - All memory accesses are statically or dynamically checked for safety
    - Any value has a size and interpretation known by the compiler
    - Casts and array indexing are checked dynamically
    - Any pointer references a memory address with a known type (a class type or an array type), while null pointers are dynamically trapped
    - Objects are never deallocated explicitly, but inaccessible objects have space collected for reuse
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 (runtime lookup)
  - examples
    ```
    A a = new A();
    B b = new B();
    A c = b;
    B d = a;
    a.p();
    b.q();
    c.p();
    ```
    class A {int x; void p(){ ... }}
    class B extends A {
      int y;
      void p(){ ... }
      void q(){ ... }
    }
    ```

A

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

- mJAM runtime layout

```
  SB
  |   |   |   |
  d_A   h_p   p: d_p[A]
  |   |   |
  d_B   h_p   p: d_p[B]
  |   |   |
  h_p   h_q   q: d_q[B]

  LB
  |   |   |   |
  a:
  |   |   |
  b:

class descriptors in global segment of data memory
```

```
  d_x
  |   |   |
  d_A | S_A | x
  |   |   |
  d_x
  |   |   |
  d_B | S_B | x
  |   |   |
  d_y
  |   |   |
  S_B

```

[16] Runtime organization for OO languages
Classes with single inheritance

- mJAM code sequences (only changed sequences are shown)

```
A a = new A();  
   (object creation)
LOADL dA
LOADL SA
CALL    newobj
STORE da[LB]
a.p();    
   (dynamic invocation)
LOAD   da[LB]
CALLD   hp
```

```
A a = new A();
LOADL dA
LOADL SA
CALL    newobj
STORE da[LB]
```

```
a.p();
LOAD   da[LB]
CALLD   hp
```
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 those used by mJAM