COMP 520 - Compilers

Lecture 16 (April 19, 2022)

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
  – layout of mJAM memory

• 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
    - mJAM can not write into code store
  - **Data store**
    - static constants and variables are loaded into static segment
    - method invocation creates a frame
    - expression evaluation occurs at stack top
      - expands downwards
    - object instances are dynamically allocated on the heap
      - expands upwards
      - (no garbage collection)

- **ABI defines fixed addresses and usage conventions**
  - various locations in memories are accessed relative to machine registers (CB, SB, LB, ST, etc.)
miniJava: simple classes, no inheritance

- **Classes**

  ```java
class A { int x; void p(){x = 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: $A \ a = \text{new} \ A();$
  - let $d_a$ be displacement of local var “a” in activation record

  ![mJAM runtime layout diagram]

  - LB
  - mJAM runtime layout
  - activation record on stack
  - object instance in heap
  - $S_A$
  - $d_x$
  - $d_a$
  - reserved
  - $x$
  - activation record on stack
  - object instance in heap
mJAM – adapted from TAM (text appx C)

- Instructions

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Instruction mnemonic</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD (n) d[r]</td>
<td>Fetch an n-word object from the data address (d + register r), and push it on to the stack.</td>
</tr>
<tr>
<td>1</td>
<td>LOADA d[r]</td>
<td>Push the data address (d + register r) on to the stack.</td>
</tr>
<tr>
<td>2</td>
<td>LOADI (n)</td>
<td>Pop a data address from the stack, fetch an n-word object from that address, and push it on to the stack.</td>
</tr>
<tr>
<td>3</td>
<td>LOADL d</td>
<td>Push the 1-word literal value d on to the stack.</td>
</tr>
<tr>
<td>4</td>
<td>STORE (n) d[r]</td>
<td>Pop an n-word object from the stack, and store it at the data address (d + register r).</td>
</tr>
<tr>
<td>5</td>
<td>STOREI (n)</td>
<td>Pop an address from the stack, then pop an n-word object from the stack and store it at that address.</td>
</tr>
<tr>
<td>6</td>
<td>CALL (n) d[r]</td>
<td>Call the routine at code address (d + register r), using the address in register r as the static link.</td>
</tr>
<tr>
<td>7</td>
<td>CALLI</td>
<td>Pop a closure (static link and code address) from the stack, then call the routine at that code address.</td>
</tr>
<tr>
<td>8</td>
<td>RETURN (n) d</td>
<td>Return from the current routine: pop an n-word result from the stack, then pop the topmost frame, then pop d words of arguments, then push the result back on to the stack.</td>
</tr>
<tr>
<td>9</td>
<td>–</td>
<td>(unused)</td>
</tr>
<tr>
<td>10</td>
<td>PUSH d</td>
<td>Push d words (uninitialized) on to the stack.</td>
</tr>
<tr>
<td>11</td>
<td>POP (n) d</td>
<td>Pop an n-word result from the stack, then pop d more words, then push the result back on to the stack.</td>
</tr>
<tr>
<td>12</td>
<td>JUMP d[r]</td>
<td>Jump to code address (d + register r).</td>
</tr>
<tr>
<td>13</td>
<td>JUMPT</td>
<td>Pop a code address from the stack, then jump to that address.</td>
</tr>
<tr>
<td>14</td>
<td>JUMPIF (n) d[r]</td>
<td>Pop a 1-word value from the stack, then jump to code address (d + register r) if and only if that value equals n.</td>
</tr>
<tr>
<td>15</td>
<td>HALT</td>
<td>Stop execution of the program.</td>
</tr>
</tbody>
</table>

- a denotes a data address
- c denotes a character
- i denotes an integer
- n denotes a non-negative integer
- t denotes a truth value (0 for false or 1 for true)
- v denotes a value of any type
- w denotes any 1-word value

call static method at d[CB]
call instance method at d[CB], instance code addr at stacktop
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 S_A
CALL newobj
STORE da[LB]

LOAD da[LB]
LOADL dx
CALL fieldref

LOAD d_a[LB]
CALLI d_p[CB]

LOAD d_x[OB]
CALL ADD
STORE d_x[OB]
```

```
LOAD d_a[LB]
LOADL d_x
CALL fieldref

LOAD d_a[LB]
CALLI d_p[CB]

LOAD d_x[OB]
CALL ADD
STORE d_x[OB]
```

**Diagram:***

- **激活记录（Activation Record）**
  - `a` 在堆上的实例地址
  - `d_a` 在堆上的实例地址
  - `S_A` 堆上的保留区域

- **栈（Stack）**
  - `a` 实例的堆分配
  - `d_a` 实例的堆分配

- **方法调用（Method Call）**
  - `a.x()` 方法调用
  - `a.p()` 方法调用
  - `x = x + 3;` 字段更新

- **数据结构（Data Structures）**
  - `d_x` 字段
  - `x` 字段
  - `d_a` 对象实例
  - `S_A` 对象实例

**图注：**
- `d_a` 实例地址
- `S_A` 堆上的保留区域
- `x` 字段更新
Another example

```java
class T {
    public int x;

    public T get_this() {
        x = 3;
        return this;
    }
}

class Mainclass {
    public static void main(String [] args) {
        T t = new T();
        T s = t.get_this();
        System.out.println(s.x);
    }
}
```
PA4 / PA5

- PA4 functionality is the goal for your miniJava compiler project
  - PA4 due Thu 4/21
  - test results will be available a few days later
  - You can make changes and resubmit a final version

- PA5 adds *optional* extensions
  - PA5 will be distributed 4/21 and due Tue 4/26 (last class)
  - list of additional options and point values will be described
  - You can choose to implement one or more option(s) or simply go with PA4 functionality.
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**

```java
A a = new A();
B b = new B();
A c = b;
B d = a;
a.p();
b.q();
c.p();
```

```java
class A {int x; void p(){ ... }}

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p:</td>
<td>d_p[A]</td>
<td></td>
</tr>
<tr>
<td>q:</td>
<td>d_q[B]</td>
<td></td>
</tr>
</tbody>
</table>

LB

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Class descriptors in global segment of data memory
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 da[Lb]
a.p();
    (dynamic invocation)
LOAD d_a[Lb]
CALLD h_p
```

Runtime organization for OO languages
Related issues

- **single inheritance**
  - type operations
    - instanceof
    - casting
    - `super()` superclass constructor invocation

- **multiple inheritance**
  - we lose the prefix property of runtime layout!
  - not supported as such in Java, instead provides “interfaces”

- **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
Interfaces and classes

• interface
  – specifies methods (name, signature) required of an implementation
    ```java
    interface List {
        ...
        add(Object x);
        ...
    }
    ```
  – is a type (can be used in type declarations)
    ```java
    List a = new ArrayList();
    ```

• class
  – implements one or more interfaces
  – provides method bodies
    ```java
    class ArrayList implements List {
        ...
        add(Object x) { ... }
        ...
    }
    ```
  – is a type
    ```java
    ArrayList a = b;
    ```
  – has a constructor
    ```java
    ArrayList a = new ArrayList();
    ```
interface vs inheritance

• inheritance
  – extends a single super-class
    • fields and methods are extended or overridden
  – requires compile time and run-time support

• interface
  – an interface can extend one or more interfaces
    • it just adds additional requirements, there is no implementation
  – requires only compile-time support

• a class
  – can implement many interfaces
  – can only extend (inherit) one other class
    • when a class extends a superclass, it inherits an implementation
    • inherited methods can be overridden
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
The PA4 checkpoint (4/21)

• your pa4 directory should have
  – miniJava package
    • Compiler.java
    • SyntacticAnalyzer
    • AbstractSyntaxTrees
    • ContextualAnalyzer
    • CodeGenerator (new subpackage)
  – mJAM package (supplied on our web page)
    • Interpreter.java
    • Disassembler.java
    • Instruction.java
    • Machine.java
    • ObjectFile.java

• mJAM is needed only to check everything is working
  – pa4 testing will not copy your mJAM, it uses the mJAM as distributed

• pa4 readiness check will be available: /check/pa4.pl
Compiling and running miniJava programs (Linux)

- **Compiling test.java**
  - `java miniJava/Compiler test.java`
    - Use `mJAM.ObjectFile` to write `test.mJAM` (note spelling!), be sure that it is written in the same directory as `test.java`
    - Do not run the generated program as part of compilation!

- **Disassembling test.mJAM**
  - `java mJAM/Disassembler test.mJAM`
    - Should write `test.asm` in same directory as `test.mJAM`

- **Executing test.mJAM**
  - `java mJAM/Interpreter test.mJAM`
    - `System.out.println` results from `test.java` will appear on stdout prefixed by “>>> “

- **Debugging test.mJAM**
  - `java mJAM/Interpreter test.mJAM test.asm`
    - Show machine data store and state, show code, set/remove breakpoints, single instruction execution
    - Type “?” for help
Check results

• To compare miniJava and java semantics of program `foo.java`

1. Run as miniJava program
   
   ```
   java miniJava/Compiler foo.java
   java mJAM/Interpreter foo.mJAM
   ```

2. Run as java program
   
   ```
   javac foo.java
   java foo.class
   ```

• Note that mJAM println prefixes output with “>>> “
The PA4-example (lec 15) is available on our web page
- generates code for the Counter.java example (lec 16)
  • illustrates the Machine interface to generate mJAM instructions

- .. then executes the generated code using mJAM
  • the Interpreter is started in debug mode so you can trace execution of the example code
  • to simplify the testing of your code generator you can install a similar shortcut to automatically execute generated code (e.g. in your compiler driver)
    - If you do so, be sure to restore standard functionality before submitting PA4