Engineers week 2010

Next Monday, Feb 15

Booths in Nedderman atrium

CSE booths are the ones nearest the elevators
object. (verb) to feel distaste for something
Method overloading

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

C c = new C();
c.m("hello");    // calls m(String)
c.m(1);          // calls m(int);
c.m(new Integer(1)); // calls m(Object)
c.m(null);       // ?
Method overloading

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

C c = new C();
c.m(“hello”); // calls m(String)
c.m(1); // calls m(int);
c.m(new Integer(1)); // calls m(Object)
c.m(null); // error! m(String) or m(Object)?
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x); // overrides C.m(String)
    void m(Object x); // overrides C.m(Object)
    void m(Integer x); // a new method
}
class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x); // overrides C.m(String)
    void m(Object x); // overrides C.m(Object)
    void m(Integer x); // a new method
}

C c = new D();
c.m(new Integer(1)); // calls what?
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);  // overrides C.m(String)
    void m(Object x);  // overrides C.m(Object)
    void m(Integer x); // a new method
}

C c = new D();
c.m(new Integer(1));  // calls D.m(Object), not D.m(Integer)
Method overloading vs. overriding

class C {
    void m(String x);
    void m(int x);
    void m(Object x);
}

class D extends C {
    void m(String x);  // overrides C.m(String)
    void m(Object x);  // overrides C.m(Object)
    void m(Integer x); // a new method
}

D d = new D();
d.m(new Integer(1));  // calls D.m(Integer)
Method lookup

If language supports overloading, method lookup is more complicated

Don’t just look for method with the same name

Find for all methods with same name whose parameter types are supertypes of the actual argument types

Then, select the *most specific method*, or report that the call is ambiguous
Most specific method

In Java (simplified):

M1 defined in C1 is *more specific* than M2 in C2 if:
- the formal params of M1 are subtypes of the formal params of M2
  - m(Integer) is more specific than m(Number)
- or, the formal param types are the same, but C1 is a subtype of C2
  - C1.m(T) is more specific than C2.m(T)

There may not be a single most specific method for a given call.
=> The call is ambiguous.
Conformance checking

Need to ensure a subclass is a *subtype* of its base class.

If a subclass overrides method m, argument types and return types must be compatible.

Need to ensure all abstract and interface methods are implemented.

Need to check access flags, exceptions also

- e.g., cannot override a public method with a private method
- e.g., method in subclass cannot throw *more* exceptions than method in superclass
Method conformance

Without method overloading:
- If a subclass declares a method M1 with the same name as a superclass method M2, M1 must also override M2
- T m(T1, ..., Tn) overrides S m(S1, ..., Sk) if:
  - n = k
  - T <: S – covariant return types
  - Si <: Ti – contravariant argument types
class C {
    Number m() { return 3.14; }
}

class D extends C {
    Integer m() { return 1; }
}

C c = new D();
Number x = c.m();  // ok!
class C {
    Integer m() { return 0; }
}

class D extends C {
    Number m() { return 2.718281828; }
}

C c = new D();
Integer x = c.m();    // uh-oh!
class C {
    void m(Integer x);
}

class D extends C {
    void m(Number x);
}

C c = new D();
c.m(1);       // ok, 1 is also a Number

C covariant argument types - wrong

```java
class C {
    void m(Number x);
}

class D extends C {
    void m(Integer x);
}

C c = new D();
c.m(1.618); // uh-oh! 1.618 is a Number, but not an Integer
```
With method overloading:

- If a subclass declares a method $M_1$ with the same signature as a superclass method $M_2$, $M_1$ must also override $M_2$
- signature = name + parameter types
- $T \ m(T_1, \ldots, T_n)$ overrides $S \ m(S_1, \ldots, S_k)$ if:
  - $n = k$
  - $T <: S$ – covariant return types
- Argument types must be the same because changing the argument types overloads the method
Exceptions

Must also check exception usage.
Often done in a separate pass than type-checking.

For each term, compute exceptions that could be thrown

\[ A \vdash e : T \]

\[ A \vdash \text{throw } e \text{ throws } \{T\} \]
Exceptions

\[
A \vdash e \text{ throws } S
\]

\[
\frac{}{A \vdash e.f \text{ throws } S \cup \{\text{NullPtrEx}\}}
\]

\[
A \vdash e_1 \text{ throws } S_1 \quad A \vdash e_2 \text{ throws } S_2
\]

\[
A \vdash e_1/e_2 \text{ throws } S_1 \cup S_2 \cup \{\text{DivBy0Ex}\}
\]

\[
A \vdash e_0 : C \quad \text{C.m declared “throws T1, ..., Tn”}
\]

\[
A \vdash e_0 \text{ throws } S_1 \quad A \vdash e_2 \text{ throws } S_2
\]

\[
A \vdash e_0.m(e_1) \text{ throws } \{T_1, ..., T_n\} \cup S_1 \cup S_2 \cup \{\text{NullPtrEx}\}
\]
Catching exceptions

Exceptions thrown by try-catch
- compute exceptions thrown in try block
- subtract exceptions caught
- add exceptions thrown in catch block

Checking catch:
- compute exceptions thrown in try block
- report error if there is a catch block for an exception not thrown

- If \{T_1, ..., T_n\} is the set of exception thrown in try block
- can catch exception type T if there is a Ti s.t. T <: Ti or Ti <: T and T <: Throwable
Summary

Semantic analysis checks that the program is semantically sound.

Covered
- type checking
- conformance checking
- exception checking

Did not discuss:
- reporting unreachable code
- reporting uninitialized variables
- reporting multiply assigned final variables
- ...
- These require more sophisticated analysis—much later in the course
Additional semantic checking

1. advanced type checking
   - records
   - functions

2. checking exceptions

2. checking for unreachable code

3. checking definite assignment
Runtime organization and objects
The address space of a process is divided into segments:

- code (aka text)
- global data
- heap
- stack

Details depend on the OS
An aside on bits

Recent operating systems (e.g., Windows 7, OS X 10.6) are 64-bit.

What are the advantages?

What are the disadvantages?
Each thread in a process maintains a call stack

*Stack frame* (aka *activation record*) contains:

- function parameters
- saved frame pointer of caller
- saved return address of caller
- local variables

Stack frame pushed when calling a function

Stack frame popped on return
Stack frame

**fp** - frame pointer
points to current frame

**sp** - stack pointer
points to current top of stack

**saved ip** - saved return address

**saved fp** - saved frame pointer of caller
aka *dynamic link*

sometimes also: *static link* - pointer to frame of
lexically enclosing function (this is not
used by most languages nowadays)

**callee frame** can access args from **caller frame**
Before call:
- caller allocates callee stack frame
- caller evaluates and stores parameters in registers or on stack
- caller stores return address in register or on stack

Prologue:
- callee stores frame pointer in stack
- callee set fp to be top of stack
- callee allocates local variables (bumping sp)
Epilogue:
- callee stores return value in register or on stack
- callee restores fp
- callee jumps to return address

After return:
- caller copies return value out
- caller deallocates callee frame
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)

<table>
<thead>
<tr>
<th>fp</th>
<th>1288</th>
<th>saved fp</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1284</td>
<td>saved ip</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>1280</td>
<td>x</td>
<td>3</td>
</tr>
<tr>
<td>sp</td>
<td>1276</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1272</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1268</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1264</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1260</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1256</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1252</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1248</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1244</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1240</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

⇒ f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
def g(y, z) = y + z

→ def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
  val a = 4
  val b = g(x+a, 2)
  b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
Call stack: in g

```python
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
```
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3
f(x, 1)
def g(y, z) = y + z

def f(x, y) = {
    val a = 4
    val b = g(x+a, 2)
    b + y
}

val x = 3

f(x, 1)
Call by reference

Implement by passing pointer as a parameter

Can pass pointer to variable on the stack

```c
int x = 0;
f(&x);    // passes x by reference

void f(int *p) { *p = 1; }    // modifies caller stack frame
```
Parameters can often be passed in registers rather than on stack
- cannot do if variable is passed by reference (since must callee needs an address)
- cannot do for multiword structures

Return value usually passed in a register (%eax in x86)
Frame pointer

Frame pointer and stack pointer usually stored in registers
- %ebp and %esp on x86, respectively

Can omit frame pointer register if stack frame size is constant
- just use fixed offset from stack pointer
- must preallocate stack frame with enough space for all parameters to pass
- frees up a register for other uses
- gcc \texttt{-fomit-frame-pointer}
Exceptions

Why?
- separate normal control flow from error-handling
- better documentation of errors that might occur
- no need to remember to check return values.

```c
int fd = open(filename, mode);
if (fd < 0)
    printf("something went wrong.");

char *p = malloc(BIG);
if (p == NULL)
    printf("out of memory");
```
Exceptions in Java

```java
try {
    FileInputStream in = new FileInputStream(file);
    int ch;
    while (true) { ch = in.read(); if (ch < 0) break; print(ch); }
}
catch (IOException e) {
    println(e.getMessage());
}
```
Local implementation

If exception thrown is caught by same method, can just jump to the handler

Need to check type of exception in handler

What if exception is not caught by same method?
Exception implementation

Maintain a stack of *handler entries*

*handler entry* = *handler map* + *frame pointer*

*handler map* = map from exception types to program counter

at start of try:
- push handler map onto stack

at end of try:
- pop

at throw:
- find topmost matching handler entry in stack
- update the frame pointer (popping the stack)
- update the instruction pointer to handler PC
Exception implementation

With this implementation, we pay for exception handlers even if no exceptions are thrown
Alternative implementation

Record exception in a global (thread-local) variable at throw

Generate code to have caller check this field
- if there is an exception, jump to handler code if any
- if no handler code, return
- if no exception, proceed normally

Again, we pay for this even if no exceptions are thrown
- Have to check exception variable at every return
Use the call stack to search for an exception handler

Three tasks:
1. map program context (i.e., the ip) to enclosing try block
2. filter the exception
3. search for the try blocks of caller methods
Map program context

Maintain a map data structure for each function that maps program counter to handler offset
Stack unwinding

Frame pointer links are a linked list of stack frames!

pc = current ip
while (frame != bottom_of_stack)
    T = find try-block covering pc
    look for handler of T
    if found
        // jump to handler
        ip = handler
    else
        pc = *(fp + 1)
        fp = *fp
Stack frames

Will go into much more detail on stack frames when we discuss IR lowering
The heap

Dynamically allocated memory is created in the heap

C, C++:
- `malloc` - finds block of appropriate size; if not found, grow the heap by asking OS for more pages of memory
- `free` - return block to the heap
- `malloc/free` manages several data structures to keep track of free space (e.g., bit vectors, free lists)

Java, C#, ...:
- `heap` is garbage collected
- process asks for block of memory
- garbage collector automatically frees memory no longer in use
- will describe GC later
Implementing objects and records
aka structs

struct S {
    int x;
    double y;
}

Records
void f() {
    struct S s; // declares a local variable of type S
    s.x = 0;
    s.y = 1.0;
}

equivalent to expansion of the struct:

void f() {
    int s$x;
    double s$y;
    s$x = 0;
    s$y = 1.0;
}
Allocating records on the heap

```cpp
void f() {
    struct S *s = new S;
    s->x = 0;
    s->y = 1.0;
    delete s;
}
```

- allocates a struct on the heap
- saves a pointer to the struct on the stack
- must dereference the pointer to access its fields

- in C++, should explicitly free the memory with delete
- in a language like Java, GC will free the memory
void f() {
    struct S *s = new S;
    s->x = 0;
    s->y = 1.0;
    delete s;
}
class C {
    int x;
    double y;
    virtual void m();
}

void f() {
    C *c = new C;
    c->x = 0;
    c->y = 1.0;
    delete c;
}
class C {
    int x;
    double y;
    void m();
}

void f() {
    C c = new C();
    c.x = 0;
    c.y = 1.0;
}
```java
class C {
    int x;
    double y;
    void m();
}

void f() {
    C c = new C();
    c.x = 0;
    c.y = 1.0;
}
```

**Note**: object can outlive the method that created it since it lives in the heap.
Objects on the stack

In C++, can also allocate objects on the stack.

We will ignore this.
Inheritance

class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new D();
}

With single inheritance, fields of the subclass are appended to the object
Inheritance

class C {
  int x;
  double y;
  void m();
}

class D extends C {
  String z;
  void n();
}

void f() {
  C c = new D();
}

Note: prefix of a D object looks exactly like a C object.
Subtyping

class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new C();
    C d = new D();
}

Note: prefix of a D object looks exactly like a C object. This allows a D to be used as a C.
Method dispatch

```java
class C {
    int x;
    double y;
    void m();
}

class D extends C {
    String z;
    void n();
}

void f() {
    C c = new C();
    C d = new D();
}
```

Method dispatch implemented using a dispatch table, accessed through the object header.
Method dispatch

\[ x.m() \]

implemented as:

\[
\begin{align*}
  t &= *x; & \text{// dereference x to get base} \\
  & & \text{// address of dispatch table} \\
  p &= *(t + m\_offset); & \text{// load the address of the} \\
  & & \text{// method’s code} \\
  (*p)(); & \text{// invoke the method}
\end{align*}
\]
To implement method override, just change address of code in dispatch table.
Method dispatch

Code generated for calling an overridden method is exactly the same.

Caller doesn’t know that the method is overridden.

Caller doesn’t even know the exact run-time class of the receiver.
Dynamic casts and instanceof

Can ask for run-time type of an object.

```java
(e instanceof C)
(C) e
```

Simple implementation:
- just add a hidden virtual method that returns a representation of the type
Multiple inheritance

Shared vs. non-shared

- given a diamond:
  - class A
  - class B extends A
  - class C extends A
  - class D extends B, C
- does D contain one copy of A (shared) or two (non-shared)?

- C++ uses virtual inheritance to make A shared
- otherwise non-shared
Non-virtual MI

class A { int w; }
class B : A { int x; }
class C : A { int y; }
class D : B, C { int z; }

Note, D has two copies of w
Also two headers.
Virtual MI

class A { int w; }
class B : virtual A { int x; }
class C : virtual A { int y; }
class D : B, C { int z; }

Note B, C, D have pointers to the “A subobject” of the object.
Interfaces

interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}

Note, there are no run-time instances of an interface.
Interfaces

```java
interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}
```

```java
I x = new D();
x.m();
J y = new C();
y.m();
```

How to do method dispatch when table entries for method m don’t line up?
Interface dispatch

One approach, used by C++ to implement MI

Multiple dispatch tables per type
- one per superclass
- space inefficient
- relies on knowing all superclasses at compile-time (not possible in Java with dynamic class loading)
Interface dispatch

Problem:

dispatch table offset for a method is determined by the class hierarchy

a given method (say m) may be at different offsets in the dispatch table because it was not introduced by a common superclass
For each interface I a class C implements, maintain $itable(I,C)$, a dispatch table for C, restricted to methods of I

To dispatch to a method m of I, first find $itable(I,C)$.

- can be located through a global hash table

Then locate the code address with a normal dispatch table lookup.
interface I {
    void n();
    void m();
}

interface J {
    void m();
    void p();
}

class C implements I, J {
    int x;
    void p() { ... }
    void n() { ... }
    void m() { ... }
}

class D implements I {
    void m() { ... }
    void n() { ... }
}

*\*p is address of dispatch table*
Garbage collection

Rationale
- simplifies interfaces
- no memory errors

Key idea: garbage if not reachable
- under approximation of real garbage (might be reachable but never used again)

Terminology: mutator

Algorithms
- mark-sweep
- copying, forwarding pointers, GCSP
- generational
- BDW
Questions?