HOT Programming in Java

COS 441
Princeton University
Fall 2004

What’s HOT
Term coined by Bob Harper
Higher-Order and Typed
Some HOT languages
SML, Haskell, Java, C#

Features of HOT Languages
• Type-Safe
• Lexical Closures
• Garbage Collected

A HOT Programming Style
SML is arguably “HOTer” than most language
– Immutable by default
– Supports inductive datatypes and pattern matching
– Advanced module system
– Second-Class Polymorphism with Type Inference
How does Java stack up?

Object Orientated vs HOT Programming
Mitchell asked the question (10.2.5)
“Can you program in an Object Oriented way in ML?”
Today’s question is
“Can you program in a HOT way in and Object Oriented language?”
Alternatively
How can I use what I learned about programming in SML and use it in a “real language”

A Simple Example

datatype nat = Zero | Succ of nat

fun add(m,Zero) = m
| add(m,Succ(n)) = Succ(add(m,n))

fun fib(Zero) = Succ(Zero)
| fib(Succ(Zero)) = Succ(Zero)
| fib(Succ(Succ(n))) = add(fib(Succ(n)), fib(n))
Encoding a Datatype

```java
datatype nat = Zero | Succ of nat

abstract class Nat {
}

class Zero extends Nat {
    Zero() { /* nop */ }
}

class Succ extends Nat {
    final Nat n;
    Nat(Nat n) { this.n = n; }
}
```

Creating a Value

```java
val one = Succ(Zero)

import static Nat.*;

final Nat one = new Succ (new Zero());
```

Creating a Value

```java
val one = Succ(Zero)

import static Nat.*;

final Nat one = Succ (Zero);
```

The General Pattern

When encoding a type of the form

\[ t = A + B + C + D \]

Create an abstract class of type “t” and have each case inherit from the abstract base class

Encoding a Function

```java
fun add(m,Zero)       = m
    | add(m,Succ(n))   = Succ(add(m,n))
```

Several ways to do this
- Will try obvious pure OO way first
- Later approach using type casts
- Finally Visitor Pattern

Encoding a Function

```java
fun add(Zero,m)       = m
    | add(Succ(n),m)   = Succ(add(n,m))
```

Swap order of arguments so that add is inductively defined on its first argument
Now think of \( add(n,m) \) as \( n.add(m) \) where \( n \) is either a Zero or Succ object
Define methods appropriately
Encoding a Function

```java
fun add(Zero, m) = m
| add(Succ(n), m) = Succ(add(n, m))

abstract class Nat {
  abstract Nat add(Nat n);
}
class Zero extends Nat {
  Nat add(Nat m) { return m;
  }
}
class Succ extends Nat {
  Nat add(Nat m) {
    return new Succ(this.n.add(m));
  }
}
```

A General Pattern?

Technique we described was not a general solution relies on having simple inductive definition of first argument

How do we handle binary methods like “eq”

```java
fun eq(Zero, Zero) = true
| eq(Succ(n), Succ(m)) = eq(n, m)
| eq(_, _) = false
```

Double Dispatch

```java
fun eq(Zero, Zero) = true
| eq(Succ(n), Succ(m)) = eq(n, m)
| eq(_, _) = false
```

class Zero extends Nat {
  abstract boolean eq(Nat m);
  abstract boolean zeroEq();
  abstract boolean succEq(Nat n);
}

Double Dispatch

```java
fun eq(Succ(n), Succ(m)) = eq(n, m)
| eq(_, _) = false
```

class Succ extends Nat {
  boolean eq(Nat m) {
    return m.succEq(this.n);
  }
  boolean zeroEq() { return false; }
  boolean succEq(Nat n) {
    return n.eq(this.n);
  }
}

Double Dispatch

```java
fun eq(Succ(n), Succ(m)) = eq(n, m)
| eq(_, _) = false
```

class Succ extends Nat {
  boolean eq(Nat m) {
    return m.succEq(this.n);
  }
  boolean zeroEq()      { return true; }
  boolean succEq(Nat n) { return false; }
}

What about Fib?

Encoding Fib not easy to encode for other reasons besides double dispatch!

```java
fun fib(Zero) = Succ(Zero)
| fib(Succ(Zero)) = Succ(Zero)
| fib(Succ(Succ(n))) = add(fib(Succ(n), fib(n))
What about Fib?

```
fun fib(Zero) = Succ(Zero)
| fib(Succ(Zero)) = Succ(Zero)
| fib(Succ(Succ(n))) = add(fib(Succ(n)), fib(n))
```

Nat fib(Nat n) {
  if(n instanceof Zero) {
    return new Succ(new Zero());
  }
  Succ m = (Succ) n;
  ...
}

InstanceOf

Java provides `instanceof` operator to test for class at runtime and dynamic type casts to cast appropriately.

Requires runtime type info (not available in C++)

Error Prone and Hard to read

Reducing to Primitive Matching

```
fun fib(Zero) = Succ(Zero)
| fib(Succ(Zero)) = Succ(Zero)
| fib(Succ(Succ(n))) = add(fib(Succ(n), fib(n))
```

```
fun fib(Zero) = Succ(Zero)
| fib(Succ(m)) = fibSucc(m)
and fibSucc(Zero) = Succ(Zero)
| fibSucc(Suc(n)) = add(fib(Succ(n), fib(n))
```

Visitor Design Pattern

```
abstract class Nat {
  abstract Nat accept(NatVisitor v);
}
```

```
interface NatVisitor {
  Nat vZero();
  Nat vSucc(Nat n);
}
```

Visitor Design Pattern

```
class Zero extends Nat {
  Nat accept(NatVisitor v) {
    return v.vZero();
  }
}
class Succ extends Nat {
  final Nat n;
  Nat accept(NatVisitor v) {
    return v.vSucc(this.n);
  }
}
```

Visitor for Computing Fib

```
class Fib implements NatVisitor {
  static Nat fib(Nat n) {
    return n.accept(new Fib());
  }
  public vZero() {
    return new Succ(new Zero());
  }
  public vSucc(Nat n) {
    return FibSucc.fibSucc(n);
  }
}
```
class FibSucc implements NatVisitor {
    static Nat fibSucc(Nat n) {
        return n.accept(new FibSucc());
    }
    public vZero() {
        return new Succ(new Zero());
    }
    public vSucc(Nat n) {
        return fib.fib(new Succ(n)).add(Fib.fib(n));
    }
}

Reducing to Primitive Matching

fun fib(Zero) = Succ(Zero)
| fib(Succ(Zero)) = Succ(Zero)
| fib(Succ(Succ(n))) = add(fib(Succ(n), fib(n))

Alternate way that avoids extra function

Using An Anonymous Inner Class

class Fib implements NatVisitor {
    public vSucc(final Nat n) {
        return n.accept(new NatVisitor() {
            public vZero { return ...;}
            public vSucc(Nat m) {
                return fib(n,m);
            }
        });
    }
}

A Generic Visitor

abstract class Nat {
    abstract <T> accept(Visitor<T> v);
}

interface Visitor<T> {
    T vZero();
    T vSucc(Nat n);
}

How Painful Is This?

<table>
<thead>
<tr>
<th></th>
<th>Lines of SML</th>
<th>Lines of Java</th>
<th>Java/ SML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name Library</td>
<td>33</td>
<td>26</td>
<td>0.79</td>
</tr>
<tr>
<td>MinML Syntax</td>
<td>30</td>
<td>188</td>
<td>6.27</td>
</tr>
<tr>
<td>Static Semantics</td>
<td>48</td>
<td>86</td>
<td>1.79</td>
</tr>
<tr>
<td>Dynamic Semantics</td>
<td>36</td>
<td>84</td>
<td>2.33</td>
</tr>
<tr>
<td>Shape Library</td>
<td>62</td>
<td>109</td>
<td>1.76</td>
</tr>
</tbody>
</table>

Ratio is ~2.5 Lines of Java for Every Line of SML

How Fast Is This?

<table>
<thead>
<tr>
<th></th>
<th>SML</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 Loops of fib(25)</td>
<td>33 sec</td>
<td>60 sec</td>
</tr>
<tr>
<td>Native</td>
<td>0.25 sec</td>
<td>0.12 sec</td>
</tr>
</tbody>
</table>

Java is about 2x slower but our scheme is 4x more expensive when compare to SML/NJ
Dimensions of Feature Extensions

<table>
<thead>
<tr>
<th>Fixed set of Objects</th>
<th>Extensible number of functions or attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero, Succ</td>
<td>Add, Mul, Fact, Fib, ...</td>
</tr>
</tbody>
</table>

I Want Both

We have been extending MinML interpreter in both dimensions!
Different operational semantics
M-Machine, C-Machine, E-Machine, ...
Different expressions and types
Pairs, Sums, Recursive, Polymorphic, ...

Language Design Challenge

- A good language should support extensions in both dimensions
- ML clunky in the object extension direction
- Java is clunky in the functionality extension dimension
- New languages being designed as we speak to support both well
  - Nice, Scala, MultiJava, ...

An Old Saying

A FORTRAN programmer can program FORTRAN in any language!
i.e. Program in any language as if it were FORTRAN!

Summary

A HOT programmer can program HOTly in any language!
i.e. Never let your programming language stop you from being too clever! 😊