Interoperability of programming languages: object oriented vs. functional

Fabricio Chalub

1 de outubro de 2003

Outline

Functional $\rightsquigarrow$ Object-Oriented

Abstract Syntax: a concrete example

Interoperating Java with SML

Functional $\rightsquigarrow$ Object-Oriented

From THORUP, Lars and TOFTE, Mads. **Object-Oriented Programming and Standard ML.**

object: set of encapsulated **instance variables** and a set of **methods**.

messages: (**methods**) are allowed to access and update the instance variables.

class: **generator**, which can generate objects containing methods that share code but operate on separate, mutable instance variables.

Functional $\rightsquigarrow$ Object-Oriented: objects and classes

Three approaches:

1. objects as closures (Uday S. Reddy)
2. objects as structures, classes as meta-objects (MOP)
3. objects as structures, classes as modules (Thorup & Tofte, Ierusalimschy)
4. objects as structures, classes as prototype objects (cloning)

Functional $\rightsquigarrow$ Object-Oriented: method invocation

Methods are messages sent to objects.

Two approaches:

1. functional: the method does not change the object, but returns a modified copy (Pierce)
2. imperative: the method can change the state of the object

Abstract syntax: a concrete example

```
public class factorial
{   public factorial () { }
    public int return_one () { return 1; }
    public int compute(int i)
    {   int f = 1;  while (i > 0)  { f = f * i; i = i - 1; }
        return f;  }
}
```

```
seq (while (app-seq (>, tuple-seq (val (i), 0)),
       stm (seq (stm (store-seq (f,
                        app-seq (*, tuple (val(f), val(i))))),
                     store-seq (i,
                     app-seq (-, tuple (val (i), 1))))))),
    null-val)
```

Objects using Abstract Syntax

```
f = { return_one = closure (NULL, 1)
      compute = closure (x, ABSTRACT-COMPUTE)   }
```

Method invokation

```
compute (f, 20) => (#compute f) (20)

f.compute(20) => (#compute f) (20)
```

Metaobjects: metaclasses

```
Factorial = { class = ‘‘factorial’’,
              variables = { x1 = ref 0,
                            x2 = ref 1 },
              methods = { return_one = closure (NULL, 1),
                          compute = closure (x, COMPUTE) }
   }
```

Objects: constructors

```
let val f = Factorial.new() in
    f.compute(10)
end

let val f = new (Factorial) in
    compute(f, 10)
end
```