Programming Languages and Techniques
(CIS120e)

Lecture 19
Oct 25, 2010

Transition to Java

Recap: The Functional Style

• Core ideas:
  – immutable (persistent / declarative) data structures
  – recursion
  – functions as data

• Good for:
  – simple, elegant descriptions of complex algorithms and/or data
  – parallelism, concurrency, and distribution
  – “symbol processing” programs (compilers, theorem provers, etc.)

OCaml: What’s Left

OCaml is not a very large language — we’ve actually seen most of its important features. But we’ve omitted a few...

• Module system
  – One of OCaml’s most interesting features is its excellent support for large-scale programming
  – We saw just the tip of the iceberg: structures and signatures
  – Key feature: functors (functions from structures to structures)

• Object system
  – OCaml actually includes a powerful system of classes and objects
  – We left them out to avoid confusion with Java’s way of doing things

• Miscellaneous handy type-system features
  – e.g. “polymorphic variants” (used, for example, to support parameter passing by name instead of by position)
Language Support for FP

- “Functional languages” (OCaml, Standard ML, F#, Haskell, Scheme) promote this style as a default and work hard to implement it efficiently
- “Hybrid languages” (Scala, Python) offer it as one possibility among others
- Mainstream OO languages (Java, C#, C++) favor a different style by default
  - But many common OO idioms and design patterns have a functional flavor
  - And most of them are gradually adding features (like anonymous functions) that make functional-style programming more convenient

Critique of Hand-Rolled Objects

- Using the features of OCaml we have seen so far, we can “roll our own” objects out of records, functions, and references
- But...
  - minor: syntax is clunky (too many parens, etc.)
  - major: object types are too rigid
    - need to introduce “controller objects”

The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - dynamic dispatch (“receiver” of method call determines behavior)
  - classes (“templates” for objects)
  - subtyping (grouping object types by common functionality)
- Good for:
  - GUIs!
    - and other complex software systems that include many different implementations of the same “interface” (set of operations) with different behaviors (cf. widgets)
  - Design methodologies that emphasize correspondences between “objects” in the computer and things in the real world

Java for OCaml Programmers
The Java Programming Language

• Designed in the early 90s, first popularized for web programming
• Key design principles:
  – Platform independence
  – Security
  – Stability

Type System Organization

<table>
<thead>
<tr>
<th>Type</th>
<th>OCaml</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive types</td>
<td>int, float, char, bool, ...</td>
<td>int, float, char, boolean, ...</td>
</tr>
<tr>
<td>structured types</td>
<td>tuples, datatypes, records, functions (objects encoded as records of functions)</td>
<td>objects (records are a special case of objects; tuples, datatypes, and “free-standing” function types are omitted)</td>
</tr>
<tr>
<td>generics</td>
<td>‘a list</td>
<td>List&lt;A&gt;</td>
</tr>
</tbody>
</table>

Just-in-time details

• Some aspects of Java involve quite a bit of detail
• We expect you to use various online and print resources to fill in the details (and you can ask when in doubt)
• But don’t worry about details until you need them
• The best way to learn details is to use them in solving a problem

Objects in OCaml and Java

```ocaml
type counter = {inc: unit->int; dec: unit->int}
let newcounter () : counter =
let r = ref 0 in
{inc = (fun () ->
    r := !r + 1; !r);
    dec = (fun () ->
    r := !r - 1; !r);
}
let c1 = newcounter();
;; print_int (c1.inc());
;; print_newline()
;; print_int (c1.dec());
;; print_newline()
```

```java
class Counter {
    private int r = 0;
    public int inc () {
        r = r + 1;
        return r;
    }
    public int dec () {
        r = r - 1;
        return r;
    }
}
class Main {
    public static void main (String[] args) {
        Counter c = new Counter();
        System.out.println(c.inc());
        System.out.println(c.dec());
    }
}
```
Information Hiding

- OCaml uses structures and signatures
  - hide functionality by not mentioning it in the interface
- Java uses interfaces for the same thing
  - just like signatures, except that they can’t mention fields

```ocaml
interface CounterInterface {
  int inc();
}
```

```java
class CounterImpl implements CounterInterface {
  public int r = 0;
  public int inc () {
    r = r + 1;
    return r;
  }
  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Information Hiding: Private Members

- Another mechanism in Java: marking fields and methods “private”

```java
class Counter {
  private int r = 0;
  public int inc () {
    r = r + 1;
    return r;
  }
  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Null Pointers

- Every Java variable is mutable
- A Java variable of reference type T can contain either the special value “null” or a pointer to a T in the heap
  - i.e., a Java variable of reference type T is comparable to an OCaml; variable of type T option ref
  - The dereferencing of the pointer and the check for “null” are implicitly performed every time a variable is used

```java
let foo (co : counter option ref) : int =
  begin match !co with
  | None -> failwith "NullPointerException"
  | Some c -> c.inc()
  end
```

```java
class Foo {
  public int foo (Counter c) {
    return c.inc();
  }
}
```

Expressions vs. Statements

- OCaml is an expression language
  - Every program phrase is an expression (and returns a value)
  - The special value () of type unit is used as the result of expressions that are evaluated only for their side effects
  - Semicolon is an operator that combines two expressions (where the left-hand one returns type unit)
- Java is a statement language
  - Some program phrases are expressions (and return values)
  - Others are statements (terminated by semicolons)