Advances in Programming Languages

APL3: A little OCaml

Ian Stark

School of Informatics
The University of Edinburgh

Monday 19 January 2009
Semester 2 Week 2

Outline

Outline

Objective Caml

Objective Caml (OCaml) is:

- A strongly-typed functional language, a version of ML; with
- high-performance native-code compilers for many processors;
- as well as a portable bytecode compiler;
- and an interactive execution environment.

Features include:

- First-class higher-order functions;
- Imperative actions, arrays, mutable state;
- Objects, classes, multiple inheritance;
- Parametric polymorphism, exceptions;
- Records, variants, and general algebraic datatypes.

Simple statements

```
# let x = 3 in x+x;;
- : int = 6

# let square x = x*x;;
val square : int -> int = <fun>

# let rec factorial n = if n < 1 then 1 else n*(factorial(n-1));;
val factorial : int -> int = <fun>

# factorial (square 3);;
- : int = 362880
```

Type constructions

("Thursday", 9, 10) : string ∗ int ∗ int

[2. ; 2.5 ; 3.] : float list

[| 'a'; 'b' |] : char array

fun x y −> (x+y)/2 : int −> int −> int

type day = { month:string; date:int }
{ month = "Jan"; date = 17 } : day

type shape = Circle **of** int | Rectangle **of** int∗int

type 'a tree = Node **of** 'a ∗ 'a tree ∗ 'a tree | Leaf

Outline

Nominal vs. Structural

Java uses predominantly *nominative* or *nominal typing*: the only relations between types are those stated explicitly by the programmer.

```
class pair1 { int x; int y; }      // Pair of integers
class pair2 { int x; int y; }      // Also a pair of integers

pair1 a = new pair1();             // Create one new pair object
pair2 b = a;                       // Assign it to another
                                   // Get an "incompatible types" error
```

This is by design:

- it can help with safe programming; and
- it certainly helps the compiler with typechecking.

Nominal vs. Structural

In contrast, OCaml uses *structural typing*: the properties of types can be deduced from their structure.

```
type pair1 = int * int          (* Type abbreviation *)
type pair2 = int * int          (* An identical one *)

let a : pair1 = (5,6)           (* Create a new pair *)
let b : pair2 = a               (* Copy it to another *)
                                (* No error *)
```

This is also by design. However, if we want nominal typing, then we can enforce it with datatype wrapping:

```
type pair1 = Pair1 of int * int
type pair2 = Pair2 of int * int
```

Polymorphism: Parametric and OO

Many OCaml functions can be used at several types: they are *polymorphic*.

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.filter ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Even as the types change, the action of the function is essentially the same. This is *parametric polymorphism*, and is heavily used in functional programming languages like Haskell and ML.

OCaml automatically infers polymorphic types where possible:

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

```
# let diag z = (z,z);;
val diag : 'a -> 'a * 'a = <fun>
```

Polymorphism: Parametric and OO

Parametric polymorphism was added in Java 5 as *generics*, through types like List<String> and methods with a type parameter:

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
}
```

The same feature arrived in C# 2.0, and generics are now extensively used in the standard libraries of both languages.

Note that C++ *templates* can achieve a similar effect (and many others), but at the cost of duplicating code during compilation. The ideal for parametric polymorphism is that because the action is the same, the executing code should be the same too.

Polymorphism: Parametric and OO

Parametric polymorphism was added in Java 5 as *generics*, through types like List<String> and methods with a type parameter:

```
public interface Iterator<E> {
  E next();
  boolean hasNext();
  void remove();
}
```

The design of generics in Java evolved from Haskell, via the research languages *Pizza* and *GJ*.

Haskell Generics are something else again...

Maurice Naftalin, Phil Wadler.
Java Generics and Collections.
O'Reilly, 2006.

Polymorphism: Parametric and OO

Object-oriented code is *polymorphic* when it can be used with objects from different classes:

```
Shape[] shapeArray;
...
for (Shape s : shapeArray)    // For every shape in the array ...
{ s.draw(); }                 // ... invoke its "draw" method.
```

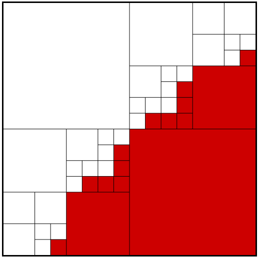
Each Shape s may actually be a Square, Circle or other implementation of Shape, each with its own implementation of draw.

In Java, a class-based language, this kind of polymorphism is closely tied to subtypes and inheritance. In object-based or dynamically-typed languages, it need not be.

With *parametric* polymorphism, the same must happen at every type. Here, with *ad-hoc* polymorphism, a different thing may happen at each type.

Outline

A *region quadtree* represents two-dimensional spatial data, such as images, with variable resolution. Where information density is nonuniform it is more efficient than a simple two-dimensional array.



```
type quadtree = Clear
              | Black | White | Red | Green | Blue
              | Tree of quadtree * quadtree * quadtree * quadtree

type picture = { title : string; image: quadtree }
```

```
let rec isclear : quadtree -> bool
        = fun qt ->
            match qt with
              Clear -> true
            | Tree (a,b,c,d) -> isclear a && isclear b
                              && isclear c && isclear d
            | _ -> false

(* nonblank : picture -> bool *)
let nonblank pic = not (isclear pic.image)
```

Example: Quadtrees (3/3)

```
let rec chop : int -> quadtree -> quadtree
  = fun n qt ->
      if n <= 0 then Clear
      else
        match qt with
          Tree (a,b,c,d) -> Tree  (chop (n-1) a, chop (n-1) b,
                                   chop (n-1) c, chop (n-1) d)
        | colour -> colour

(* thumbnail : picture -> picture *)
let thumbnail { title = t; image = i } = { title = t; image = chop 8 i }

(* summary : picture list -> picture list *)
let summary pics = List.map thumbnail (List.filter nonblank pics)
```

Homework

- Find out what an octree is. (Bonus: Why would you use one in Microsoft's XNA game development toolkit?)
- Copy and paste the quadtree code and run it in OCaml.
- Write a function to compute the nonblank area of a quadtree.
- Write a function to display a quadtree: either by converting it to a list of strings, or (better) using the OCaml graphics library.

Summary

- OCaml is a functional programming language with a rich static type system.
- Where Java uses *nominal* typing, OCaml uses *structural* typing.
- Type polymorphism may be *parametric* (OO "generic") or *ad-hoc* (classic OO).
- We saw some example OCaml code for manipulating quadtrees, a structure for variable-resolution 2-dimensional spatial data.

Benjamin C. Pierce.
Types and Programming Languages.
MIT Press, 2002