Advances in Programming Languages

APL3: Row variables in OCaml

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 17 January 2008
Semester 2 Week 2

Outline

Outline

1. OCaml overview: types, expressions
2. OCaml example: region quadtrees
3. Row variables: structural typing for objects

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;
- 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

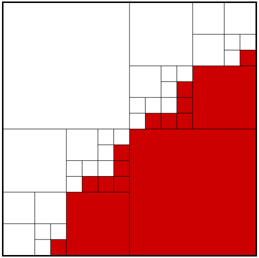
type shape = Circle **of** int | Rectangle **of** int*int

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

Outline

1. OCaml overview: types, expressions
2. OCaml example: region quadtrees
3. Row variables: structural typing for objects

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 }
```

Example: Quadtrees


```
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

```
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)
```

Outline

Subtyping arrays in Java

Java has subtyping: a value of one type may be used at any more general type. So String $<$ Object, and every String is an Object.

Not all is well with Java types

```
String[] a = { "Hello" };      // A small string array
Object[] b = a;                // Now a and b are the same array
b[0] = Boolean.FALSE;          // Drop in a Boolean object
String s = a[0];               // Oh, dear
System.out.println(s);         // This isn't going to be pretty
```

This compiles without error or warning: in Java, if S $<$ T then S[] $<$ T[].

Except that it isn't. So every array assignment gets a runtime check.

Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance; (really, it's not)

Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance; (really, it's not)
- it's also extremely hard to get right.

How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from: (Java 1.2, 1998)

How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from: (Java 1.2, 1998)

public static Object max(Collection coll)

which always returns an Object, whatever is stored in the collection, to:

How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from: (Java 1.2, 1998)

public static Object max(Collection coll)

which always returns an Object, whatever is stored in the collection, to:

public static <T **extends** Object & Comparable<? **super** T>> T
max(Collection<? **extends** T> coll)

How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from: (Java 1.2, 1998)

```
public static Object max(Collection coll)
```

which always returns an Object, whatever is stored in the collection, to:

```
public static <T extends Object & Comparable<? super T>> T
  max(Collection<? extends T> coll)
```

and it might *still* throw a ClassCastException. (Java 6, 2006)

How hard?

Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from: (Java 1.2, 1998)

public static Object max(Collection coll)

which always returns an Object, whatever is stored in the collection, to:

public static <T **extends** Object & Comparable<? **super** T>> T
max(Collection<? **extends** T> coll)

and it might *still* throw a ClassCastException. (Java 6, 2006)

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.

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 *)
```

If object typing is tough to sort out nominally, then how do we attempt to do it structurally?

Records and record types

OCaml provides strongly-typed *records*:

```
type picture = { title : string; image : quadtree }
let p = { title = "Look at me"; image = i }

# p.title;;
- : string = "Look at me"
```

This could be the basis for an object system; records can even have *mutable* fields to serve as instance variables.

However, field names are strictly tied to their record:

```
# fun x -> x.title;;
- : picture -> string = <fun>
```

Objects need more flexibility. Subtyping is one possibility, but there is another mechanism already available...

Parametric polymorphism

A simple type system:

$$\tau ::= \alpha \quad | \quad \tau \times \tau \quad | \quad \tau \to \tau$$
$$\sigma ::= \forall \vec{\alpha}.\tau$$

Here τ is a type, α is a *type variable* and σ is a *type scheme*.

Type schemes characterise functions that carry out the same action at a range of types, for example:

$$\lambda x.x : \forall \alpha.\alpha \to \alpha$$

This is *parametric polymorphism*, implemented in Java/C# as *generics*.

OCaml automatically infers polymorphic types where possible:

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

Row variables

Add types for records, where $m_1 \dots m_k$ are labels and ρ is a *row variable*:

$$\tau ::= \alpha \quad | \quad \tau \times \tau \quad | \quad \tau \to \tau \quad | \quad \langle m_1 : \tau_1, \dots, m_k : \tau_k \mid \rho \rangle$$
$$\sigma ::= \forall \vec{\alpha} \vec{\rho}. \tau$$

We can now type functions that carry out the same action at a range of different record types. For example, using # for field selection:

$$\lambda x.(x\#m) : \forall \alpha \forall \rho. \langle m{:}\alpha | \rho \rangle \to \alpha$$

This is *row polymorphism*.

OCaml automatically infers polymorphic row types where possible:

```
let getfield p = p#m
val getfield : < m : 'a; .. > -> 'a = <fun>

let double p = p#height * 2;;
val double : < height : int; .. > -> int = <fun>
```

OCaml uses row types to represent an object as a record of methods.

```
let a =                              (* Saving account *)
  object
    val mutable balance = 0
    method credit n = balance <- balance + n
    method enquire = balance
  end;;
val a : < credit : int -> unit; enquire : int > = <obj>
```

Automatic type inference gives the most general type for an object.

(OCaml does also have classes for objects that share method suites.)

Different object types can share methods with the same name.

```
let b =                              (* Spending account *)
  object
    val mutable balance = 0
    method credit n = balance <- balance + n
    method debit n = balance <- balance - n
    method enquire = balance
  end;;
val b : < credit : int -> unit; debit : int -> unit; enquire : int >
    = <obj>
```

Account b has all the methods of a, and more.

(We could also use inheritance to generate one class from another.)

Define a function to add credit to an account.

```
let boost x = x#credit 20;;
val boost : < credit : int -> 'a; .. > -> 'a = <fun>
```

OCaml infers a very general type, so we can apply this to both existing accounts:

```
boost a; a#enquire;;
- : int = 20

boost b; b#debit 5; b#enquire;;
- : int = 15
```

It is even possible to infer a type for the function that takes a list of any type of accounts and selects the one of greatest value:

```
max : (< enquire : int; .. > as a') list -> 'a
```

Exercises

- What is an octree, and why would you use one in Microsoft's XNA game development toolkit?
- Copy and paste the quadtree code and run it in OCaml.
- Do the same for the bank account objects, and test them.
- Write a function to compute the nonblank area of a quadtree.
- Write a function to display a quadtree using the OCaml graphics library.

Summary

- OCaml is a functional programming language with a rich static type system.
- We saw some example OCaml code for manipulating quadtrees, a structure for variable-resolution 2-dimensional spatial data.
- Static typing for object-oriented programming is tricky.
- Row variables allow structural typing of objects.

Summary

- OCaml is a functional programming language with a rich static type system.
- We saw some example OCaml code for manipulating quadtrees, a structure for variable-resolution 2-dimensional spatial data.
- Static typing for object-oriented programming is tricky.
- Row variables allow structural typing of objects.