Objects and Aspects: Row Polymorphism

Neel Krishnaswami

Department of Computer Science
Carnegie Mellon University
neelk@cs.cmu.edu

Overview

1. Records and Objects

2. Row Polymorphism

3. Using Rows

Encoding Objects in Type Theory

- Question: What is the type of an object?
- Answer: It is the set of messages it can send and receive.
- Question: How can we say this mathematically?
- Answer: Using recursive types, existential types, and records.

Encoding Objects in Type Theory

- Question: What is the type of an object?
- Answer: It is the set of messages it can send and receive.
- Question: How can we say this mathematically?
- Answer: Using recursive types and records.

Encoding Objects in Type Theory

Here's an interface definition.

```
interface Point {
  pos: int;
  move: int -> Point
}
```

Here's what the type-theoretic version looks like:

$$Point = \mu P.Rec\{pos : int; move : int \to P\}$$

What this means

The μ is a "fixed point" operator, which introduces a recursive type.

$$\begin{aligned} Point &= \mu P.Rec\{pos : int; move : int \to P\} \\ &= \mu P.Rec\{pos : int; \\ &\qquad move : int \to \mu P.Rec\{pos : int; move : int \to P\}\} \end{aligned}$$

This lets us model the idea that an object can return another object of with the same interface it supports. It seems like we can now model OO, if we also add subtyping on records. (That is, if $Rec\{a : \tau, b : \sigma, c : \rho\} \leq Rec\{a : \tau, b : \sigma\}$.)

The Loss of Information Problem

Consider the following function:

$$foo :: Rec\{a : int\} \rightarrow int \times Rec\{a : int\}$$
$$foo = \lambda x.\ (x.a, x)$$

`foo` has the type $Rec\{a : int\} \rightarrow int \times Rec\{a : int\}$. This seems innocuous, but consider what happens when you say:

```
let (n, r) = foo({a:5, b:true})
```

The variable `r` has the less-precise type $Rec\{a : int\}$ – we've forgotten the existence of the `b` field. This is the famous "loss of information" problem.

Row Polymorphism

Row polymorphism was invented to address the loss of information problem. The core idea is to add a polymorphic type variable to remember all of the extra fields:

$$foo :: \forall r : \mathsf{row}.(r/a) \Rightarrow \; Rec\{a : int|r\} \rightarrow int \times Rec\{a : int|r\}$$
$$foo = \lambda x.\ (x.a, x)$$

The way to read this type is: "for all rows r which lack a field a, we have a function type which *takes* records with a field a of type `int` and some other fields described by r, and *returns* an `int` and another record of the same type."

Basic Operations on Records

- Field Selection

$$select_l :: \forall \alpha : \mathsf{type}.\forall r : \mathsf{row}.(r/l) \Rightarrow Rec\{l : \alpha | r\} \rightarrow \alpha$$

- Field Removal

$$remove_l :: \forall \alpha : \mathsf{type}.\forall r : \mathsf{row}.(r/l) \Rightarrow Rec\{l : \alpha | r\} \rightarrow Rec\{r\}$$

- Record Extension

$$add_l :: \forall \alpha : \mathsf{type}.\forall r : \mathsf{row}.(r/l) \Rightarrow \alpha \rightarrow Rec\{r\} \rightarrow Rec\{l : \alpha | r\}$$

Uses of Row Polymorphism

Row polymorphism admits type inference, and this means that an OO language that uses row polymorphism rather than subtyping on the records also has type inference.

```
let o = object
  val lst = []
  method add(x) =  {< lst = (x :: lst) >}
  method length = List.length(lst)
end

val o : <add : 'b -> 'a; length : int> as 'a
```

(This is an example from Ocaml 3.08.)

Salient Points, Pt. 0

Consider the type of `o`.

```
val o : <add : 'b -> 'a; length : int> as 'a
```

Observe the similarity of this type to the type

$\mu\alpha.\forall\beta.\ Rec\{add : \beta \rightarrow \alpha; length : int\}$

Loss of Information Revisited

Note that Ocaml does not have a loss of information problem:

```
let foo(o) = (o#length, o)

val foo : (<length : 'b; ..> as 'a) -> ('b * 'a)
```

The row ("..") guarantees that the second component of the return value has all the methods as the original object. (Also note that the absence predicate is implicit – the row variable is assumed to lack a `length` field.)

Structural Typing

```
let o = object
  val lst = []
  method add(x) =  {< lst = (x :: lst) >}
  method length = List.length(lst)
end
```

Note that this is a *literal* object, created as an instance of no class. This works because row polymorphism is a structural type discipline, rather than a nominal discipline. This lets the compiler statically typecheck prototype objects.

(Ocaml has classes, but classes and inheritance exist for code reuse purposes, rather than to induce subtyping relationships.)

Mixins

Rows can also let you write mixins as ordinary functions, and infer their type (this is not legal Ocaml!):

```
fun print(o) =
  {< method printlen =
       Printf.printf ``%d elements'' self#length | o >}
```

Which could potentially be given the type:

```
(<length : int; ..> as 'a)  ->
  (<length: int; printlen: unit; ..> as 'a)
```

With explicit absence predicates, this is:

$$\forall r : \mathsf{row}.(r/length) \Rightarrow (r/printlen) \Rightarrow$$
$$\mu\alpha.\, Rec\{length : int|r\} \rightarrow \mu\alpha.\, Rec\{length : int; printlen : unit|r\}$$

Upcasts

```
let o = object(self) method happy = "Joy!"
                     method print = Printf.printf "%s" self#happy
        end

let p = object(self) method misery = "Woe!"
                     method print = Printf.printf "%s" self#misery
        end

type printable = <print:unit>

let list = [ (o :> printable); (p :> printable) ]

val o : < happy : string; print : unit >
val p : < misery : string; print : unit >
val list : printable list
```

Upcasts must be explicit.

Error Messages

```
# [o; p];;
Characters 4-5:
  [o; p];;
      ^
This expression has type < misery : string; print : unit >
but is here used with type < happy : string; print : unit >
Only the first object type has a method misery
```

Rows are formally quite simple, which makes generating reasonably high quality error messages relatively easy. (Question to the audience: how localize are error messages in Scala, Cecil or Java 1.5 like?)

Downcasts

Downcasts not supported by Ocaml, because the designers don't like runtime type tests and refuse to implement them. However, there's no fundamental theoretical obstacle to it; you might write:

```
typecase o with
  | <print:unit; ..> -> o#print
  | otherwise -> Printf.printf ``default''
```

Discussion

- Row polymorphism is structural, rather than nominal. I think this is a virtue, but Jonathan and many other people (sometimes) disagree!
- This permits typing mixins as ordinary first-class functions. This lets you select and compose mixins at runtime, rather than compile time. (Can you do this in Scala?)
- Type errors with rows are simpler than with bounded quantification. This advantage is reduced somewhat if you want to take advantage of type inference.
- Upcasts are always explicit. How serious a limitation is this?
- Polymorphic types are predicative rather than impredicative. This is weaker than bounded quantification. Is this a problem at all?

Bonus Slides!

This contains extra slides cut from the talk, but which might be useful for discussion purposes.

The Type Theory of Rows: Kinds and Types

Kinds classify types, in the same way that types classify terms. We have *two* basic kinds of types – regular types and row types.

$$\begin{array}{rcl} \kappa & ::= & \text{type} \\ & | & \text{row} \\ & | & \kappa \rightarrow \kappa \end{array}$$

τ represents expressions of kind type.

Type Constants

For each kind κ, we can generate the legal type expressions C of that kind:

$$\begin{array}{rcll} C^{\kappa} & ::= & \chi^{\kappa} & constants \\ & | & \alpha & variables \\ & | & C^{\kappa' \to \kappa}\ C^{\kappa'} & applications \end{array}$$

Purpose	Constant	Kind	Example
Integers	int	type	int
Function Types	$\to$	type $\to$ type $\to$ type	$int \to int$
Empty Row	$\{\}$	row	$\{\}$
Row Extension	$\{l : -\vert -\}$	type $\to$ row $\to$ row	$\{l : int \vert r\}$
Record Constructor	Rec	row $\to$ type	$Rec\{a : int\}$

Predicates and Type Schemes

The syntax for row types permits writing row expressions like $\{l : \tau; l : \tau'\}$, so we need a mechanism for prohibiting such row expressions. We have seen these predicates before; they are the constraints of the form r/l.

In the ML tradition, we introduce polymorphism by adding *type schemes* to the language:

$$\sigma ::= \forall \alpha : \kappa.\, \sigma \mid \rho$$
$$\rho ::= \tau \mid \pi \Rightarrow \rho$$

π is any row absence predicate.

Type Checking

The typing judgement is of the form:

$$P|\Gamma \vdash e : \sigma$$

This reads, "Given row constraints P and variable types Γ, the type of the expression e is σ."

(As an aside $\Gamma ::= \bullet \mid \Gamma, x : \sigma$).

The Most Important Rules

The two typing rules most relevant for row types are the introduction and elimination rules for the constraints:

$$\frac{P|A \vdash e : \pi \Rightarrow \rho \quad P \models \pi}{P|A \vdash e : \rho} \{\Rightarrow E\}$$

$$\frac{P \cup \{\pi\} \mid A \vdash e : \rho}{P|A \vdash e : \pi \Rightarrow \rho} \{\Rightarrow I\}$$

This is how the type system tracks the information in the row constraints – you can add a constraint to a type if it is in the set P, and you can strip it off a type if you remember to put it in the context subsequently.