Advances in Programming Languages
APL4: Row variables in OCaml — Structural typing for objects

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 22 January 2009
Semester 2 Week 2
Outline

1. Quadtrees and octrees
2. Statically checking subtypes in Java
3. Row variables: structural subtypes 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.

\[\text{type quadtree} = \begin{cases} \text{Clear} \\ | \quad \text{Black} | \quad \text{White} | \quad \text{Red} | \quad \text{Green} | \quad \text{Blue} \\ | \quad \text{Tree of quadtree} \ast \text{quadtree} \ast \text{quadtree} \ast \text{quadtree} \end{cases}\]

\[\text{type picture} = \{ \ \text{title} : \text{string}; \ \text{image} : \text{quadtree} \ \} \]
Quadtree example?

Video by Handkor
http://handkor.googlepages.com/
Early test sequence for “Hippocrates’s Dilemma”
Octree example

Video by Handkor
http://handkor.googlepages.com/
Collision test — 256 marbles & gravity
Outline

1. Quadtrees and octrees
2. Statically checking subtypes in Java
3. Row variables: structural subtypes for objects
Subtyping arrays in Java

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

Not all is well with Java types

```java
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.toUpperCase()); // This isn't going to be pretty
```

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

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
Typing in OO languages

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

(a) usable, and
Typing in OO languages

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

(a) usable, and

(b) correct.
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.
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:
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:

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

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

```
public static <T>max(Collection<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.
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
public static Object max(Collection coll)
```

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

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

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

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

and it might still throw a ClassCastException.

(Java 1.2, 1998)

(Java 6, 2006)
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
public static Object max(Collection coll)
```

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

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

and it might still throw a `ClassCastException`.

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

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

dpair1 a = new pair1();  // Create one new pair object
dpair2 b = a;  // Assign it to another
```

This is by design:
- it can help with safe programming; and
- it certainly helps the compiler with typechecking.
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?
1 Quadtrees and octrees

2 Statically checking subtypes in Java

3 Row variables: structural subtypes for objects
OCaml provides strongly-typed *records*:

```ocaml
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:

```ocaml
# 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:

\[ \tau ::= \alpha \mid \tau \times \tau \mid \tau \to \tau \]

\[ \sigma ::= \forall \vec{\alpha}.\tau \]

Here \( \tau \) is a type, \( \alpha \) is a type variable and \( \sigma \) 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:

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

Ian Stark APL4 2008-01-22
Add types for records, where \( m_1 \ldots m_k \) are labels and \( \rho \) is a row variable:

\[
\begin{align*}
\tau & ::= \alpha \\
& \quad \tau \times \tau \\
& \quad \tau \rightarrow \tau \\
& \quad \langle m_1 : \tau_1, \ldots, m_k : \tau_k | \rho \rangle \\
\sigma & ::= \forall \vec{\alpha} \vec{\rho}. \tau
\end{align*}
\]

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

\[
\lambda x. (x \# m) : \forall \alpha \forall \rho. \langle m : \alpha | \rho \rangle \rightarrow \alpha
\]

This is row polymorphism.

OCaml automatically infers polymorphic row types where possible:

\[
\begin{align*}
\text{let } \text{getfield } p = p \# m \\
\text{val getfield : } < m : 'a; .. > \rightarrow 'a = <\text{fun}> \\
\text{let } \text{double } p = p \# \text{height} \times 2;\\n\text{val double : } < \text{height} : \text{int}; .. > \rightarrow \text{int} = <\text{fun}>
\end{align*}
\]
OCaml uses row types to represent an object as a record of methods.

```ocaml
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.

```ocaml
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.

```ocaml
let boost x = x#credit 20;;
```

```ocaml
val boost : < credit : int \rightarrow \ 'a; .. > \rightarrow \ 'a = <fun>
```

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

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

```ocaml
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:

```ocaml
max : (< enquire : int; .. > as a’) list \rightarrow \ 'a
```
Homework

- Test both the bank account objects in OCaml.
- Find out what you can about the F# language. This builds on a basis very similar to OCaml, but is also driven by the nature of the .NET platform. What are the differences between the F# and OCaml types for objects?
- Subtyping is also possible for records, without row variables. Find out what the difference is between width and depth subtyping. If you find a good source for this, then post a link on the newsgroup.
Static typing for object-oriented programming is tricky.

Co- and contra-variance is important in checking subtypes.

Where Java uses nominal typing, OCaml uses structural typing.

Row variables, and row polymorphism, allow structural typing of objects.

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