

# Row Polymorphism

John Skaller

November 1, 2017

## 1 Records

Felix has structurally typed records as illustrated below:

```
var r : (a:int,b:string) = (a=1,b="Hello");
println$ r._strr;
```

The order of the fields with different names is irrelevant.

The generic `_strr` can be used to convert a record to a string, provided values of the field types can also be converted, either with `_strr` or `str`.

### 1.1 Named value projections

The names of the fields can be used as projections:

```
var i = a r;
var s = r.b;
```

Stand alone value projections can be specified like:

```
var p = a of (a:int,b:string);
var i = p r;
```

### 1.2 Pointer projections

Pointer projections are also supported:

```
var pi : &int = &r.a;
var ps : &string = r&.b;
var i = *pi;
var s = *ps;
var p = a of &(a:int,b:string);
var i2 = *(p &r);
```

### 1.3 Repeated field names

Record field names may be repeated:

```
var rr : (a:int,a:double,a:int,b:string)
    = (a=1,a=2.3,a=4L,b="Hello")
;
```

In this case projections refer to the leftmost field of the given name. The order of fields with the same name matters.

### 1.4 Blank field names

One can also use blank field names. The grammar allows the name `n` to be used. Alternatively, the name can be omitted, or both the name and `=` sign can be omitted, except for the first field:

```
var x = (a=1,=2,n""=3);
println$ x._strr;
```

### 1.5 Tuples

If all the field names are blank, you can also omit the `=` sign from the first entry, and, if precedence allows, the parentheses:

```
var x : int * int * string = 1,2,"hello";
println$ x._strr;
```

in which case the record called a tuple. Note the distinct type syntax using n-fix operator `*`.

### 1.6 Tuple value projections

Plain decimal integer literals can be used for tuple projections:

```
var x : int * int * string = 1,2,"hello";
println$ x.1;
```

Standalone tuples projections are denoted like:

```
var x : int * int * string = 1,2,"hello";
var prj1 = proj 1 of (int * int * string);
println$ prj1 x;
```

## 1.7 Tuple pointer projections

Since tuples are a special case of records we also have pointer projections.

```
var x = 1,2,"Hello";
var px = &x;
var p2 = px . 1;
var two = *ps;
```

## 1.8 Arrays

If the types of a tuple are all the same, it is called an array:

```
var x : int ^ 3 = 1,2,3;
println$ x._strr;
```

## 1.9 Array Value projections

In this case the projections can be either an integer expression, or an expression of the type of the array index:

```
var x : int ^ 3 = 1,2,3;
var one = 0 x;
var two = x.1;
var three = x.(case 2 of 3);
```

If the projection is an integer it is bounds checked at run time. If it is a compact linear type which is the type of the array index, no bounds check is required.

## 1.10 Array pointer projections

And of course arrays have pointer projections:

```
var x : int ^ 3 = 1,2,3;
var px = &x;
var one = 1;
var p2 = px . one;
var v2 = *p2;
```

## 2 Ties

A tie is a natural transformation which can be applied to any data functor. Given a functor  $F : \text{TYPE} \rightarrow \text{TYPE}$ , there is an associated functor

$$(\&F)T = F(\&T)$$

which for each type  $T$  is the same data structure of values of type pointer to  $T$ .

The generic operator `_tie` maps a pointer to a product value to a product of pointers. The argument must be a pointer so the components are addressable.

## 2.1 Record tie

For records we have

```
// record tie
var rec : (a:int, a:int, b:int) = (a=1,a=2,b=3);
var prec : &(a:int, a:int, b:int) = &rec;
var tierrec : (a:&int, a:&int, b:&int) = _tie prec;
println$ *(tierrec.a), *(tierrec.b);
```

which maps a pointer to a record object to a record of pointers to its components.

## 2.2 Tuple tie

For tuples we have

```
// tuple tie
var tup : int * int * string = (1,2,"Hello");
var ptup : &(int * int * string) = &tup;
var tietup : &int * &int * &string = _tie ptup;
println$ *(tietup.0), *(tietup.1);
```

which maps a pointer to a tuple to a tuples of pointers to its components,

## 2.3 Array tie

For small arrays, length less than 20, we have

```
// array tie
var arr = (1,2,3);
var parr = &arr;
var tiearr = _tie parr;
println$ *(tiearr.0), *(tiearr.1);
```

which maps a pointer to an array to an array of pointers to its components.

Ties obey the rule:

$$(\text{\_tie } p).\pi = p.\pi$$

where  $p$  is a pointer to a product type and  $\pi$  is a projection.

### 3 Record pattern match

A record can be pattern matched using a record pattern consisting of some of the fields of the record:

```
var x = (a=1,b=2,c=3);
match x with
| (a=va, c=vc) => println$ va, vc;
endmatch;
```

Only the first of a duplicated field can be accessed. If a field name is repeated, both associated variables refer to the same leftmost field.

### 4 Supertype coercion

A record can be coerced to a record with less fields:

```
var x = (a=1,a=2,b=3,c=4);
typedef ab = (a:int,b:int);
var y = x :>> ab;
// (a=1,b=3)
```

A supertype coercion works by copying the required fields as encountered in a left to right scan of the sorted fields: if a field is repeated the leading fields are retained and the trailing fields lost.

### 5 Field removal

A record can be constructed from another by removing selected fields:

```
var x = (a=1,a=2,b=3,c=4);
var y = (x without a a c);
```

In a record with duplicated fields, the left most field is removed for each field name in the removal list, so two fields of the same name are removed in succession by repetition in the removal list.

### 6 Functional update

A record can be constructed from another with some field values being replaced:

```
var x = (a=1,a=77,a=66,b=2,c=3);
var y = (x with a=99,a=44);
println$ y._strrr;
```

If the record has duplicated fields, repeated replacements replace the values of successive duplicates.

The type of the replacement value is checked and must be the same as in the record. Consequently the updated record will have the same type as the original record.

## 7 Polyrecords

A list of fields may be pushed onto the left of any type with a polyrecord expression:

```
var x = (a=1,b=2);  
var y = (a=3,d=2 | x); // (a=3,a=1,b=2,d=2)
```

The type of the result is a record if the RHS term is a record, including tuples or unit tuple.

## 8 Row polymorphic function parameters

A polyrecord type can be used as or in a function parameter with a type variable in the poly slot.

```
fun move[T] (p:(x:int,y:int | T)) =>  
  (x=p.x+1,y=p.y+1 | (p without x y))  
;
```

Now we can move a circle:

```
var circle = (x=0,y=0,r=1);  
var c = move circle;  
println$ c.x,c.y,c.r; // r not lost
```

or a square:

```
var square = (x=0,y=0,w=1,h=1);  
var s = move square;  
println$ c.x,c.y,c.r; // r not lost
```

This is row polymorphism!

## 9 Dynamic Objects

Felix provides a Java like API for dynamic construction of objects. An interface definition is just an alias for a record:

```

interface X_t = {
  get: unit -> int;
  set: int -> 0;
}
// typedef X_t = (
//   get: unit -> int,
//   set: int -> 0
// );

```

An object definition is just a function which returns a record of function closures:

```

object X (var x:int) implements X_t =
{
  method fun get () => x;
  method proc set (a:int) => x = a;
}

```

Here, the `implements` clause specifies the return type of the function, and the `object` statement is sugar for a function which returns a record consisting of closures of the functions marked `method`.

This provides the object system with the full power of structurally typed record manipulations, whilst also providing complete encapsulation via functional abstraction.

The return type specification is optional.

Interfaces can be derived from others by extension:

```

interface XPos { get_x: unit -> int; }
interface YPos { get_y: int -> int; }
interface Square extends XPos, YPos {
  get_w: unit -> int;
  get_h: unit -> int;
}

```

An extension expression can also be written:

```

var r =
  extend
    (a=1,a=2,b=3),
    (a=99,a=88)
  with
    (a=11,a=22,c=4)
  end
  // (a=11,b=3,c=4)
;

```

Extensions aggregates the first field of a name in each record, for each record, replacing any duplicates in a left to right scan.