Row Polymorphism

John Skaller

August 2, 2016

1 Records

Felix has structurally typed records as illustrated below:

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

The order of the fields fields with different names is irrelevant.

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 $F : \mathtt{TYPE} \to \mathtt{TYPE}$, there is an associated functor

$$\&FT = F\&T$$

which for each type T builds a data structure of pointers to T.

The operator *_tie* maps each F to $\&F$. 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 tierec : (a:&int, a:&int, b:&int) = _tie prec;
println$ *(tierec.a), *(tierec.b);
```

which maps a pointer to a record object to a record of pointers to its components. 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, and finally, for small arrays 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:

$$(\_tiep).\pi = p.\pi$$

where p is a pointer to a product type and π is a projection.

2.1 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.

3 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.

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

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!