Objectifying

V. Zaytsev @ NOOL @ SPLASH 2017

Rascal intro

- made at CWI, Amsterdam [same team as ASF+SDF]
- one-stop shop by Klint, Vinju and van der Storm
- see [SCAM'09] [GTTSE'09] [SLE'11] [SCP 2015]
- Java-like concrete syntax
- Haskell-like expressions
- GLL-like parsing
- grammars and ADTs
- patterns for metaprogramming
- no OO at all (originally planned, decided otherwise)

Language Design & Composition

- the two were never friends
- For design, wait for SPLASH-I (or the posters)
  - cf. DYOL @ MoDELS'17, http://slebok.github.io/dyol
- Largely unsolved questions:
  - how to add OO to FP?
  - is it easier to make an imperative language functional or procedural?
  - how to add queries to a 3GL?
- This exercise:
  - add OO to a metaprogramming language

Rascal expressions

```
1 + 2

sqrt(3*3 + 4*4)

[size(s) | list[int] s <- ss]

{n*n | n <- [-10..10]}
```

Rascal grammars

```
syntax CSV = {Value ","}+;

lexical Value = [a-zA-Z]+ !>> [a-zA-Z];

layout L = [\ \n \r \t]* !>> [\ \n \r \t];
```

Rascal functions

```
void push(int x, list[int] xs)
{
    xs += x;
}

int peek(list[int] xs)
    = xs[size(xs)-1];
```

Rascal algebraic data types

```
data Bool
    = tt()
    | ff()
    | conj(Bool L, Bool R)
    | disj(Bool L, Bool R)
;

conj(tt(), disj(tt(), ff()))
```

Rascal patterns

```
visit(T)
{
    case tt(): counter += 1;
}

visit(T)
{
    case tt() => ff();
}
```

Do you have a moment to talk about parsing?

Parsing is a binding

- parsing is not a grammar
- parsing is a process
- parsing is a mapping
- grammar is a double spec

OO is rich

- clusters
- classes
- objects
- prototypes
- traits
- mixins
- enumerations
- records
- variants

BOOL bindings

BOOL

```
Foo := plus[word] ~ list[str]

syntax CFoo = BoolWord+ boolword;

alias AFoo = list[str];

AFoo implodeFoo(CFoo T)
    = ["<element>" | BoolWord element ← T.boolword];

AFoo bindFoo(str input)
    = implodeFoo(parse(#CFoo, input));
```

BOOL classes

BOOL

```
Point    := seq[int x, comma, int y]
          ~ class[int x, int y, method add, method sub]
Point.add := fun[Point l, Point r]
             ~ Point[x:=l.x+r.x; y:=l.y+r.y]
Point.sub := fun[Point l, Point r]
             ~ Point[x:=l.x-r.x; y:=l.y-r.y]
```

```
data APoint = newPoint(int x, int y, APoint(APoint l, APoint r)
add, APoint(APoint l, APoint r) sub);
```

```
APoint addPoint(APoint l, APoint r) = newPoint(l.x+y.x, l.y+r.y,
l.add, l.sub);
```

BOOL records

BOOL

```
Pair := seq[int x, comma, int y] ~ record[int x, int y]

syntax CPair = BoolInt x "," BoolInt y;

alias APair = tuple[int x, int y];

APair newPair(int x, int y) = < x, y >;

alias IPair = tuple[CPair(str) parse, APair(str) bind,
                    APair(CPair) implode, APair(int, int) new];
IPair Pair = ...
```

BOOL clusters

BOOL

```
Point := Pair ~ cluster[method add, method sub]
Point.add :=
    fun[Pair l, Pair r] ~ Pair[x:=l.x+r.x; y:=l.y+r.y]
Point.sub :=
    fun[Pair l, Pair r] ~ Pair[x:=l.x-r.x; y:=l.y-r.y]

alias IPoint
    = tuple[APair(APair, APair) add, APair(APair, APair) sub];
public IPoint Point
 = < APair (APair l, APair r) { return newPair(l.x+r.x, l.y+r.y);},
    APair (APair l, APair r) { return newPair(l.x-r.x, l.y-r.y);} >;
```

Using BOOL

BOOL

```
Pair  := seq[int x, comma, int y] ~ record[int x, int y]
Point := Pair ~ cluster[method add, method sub]
```

```
CPair a = Pair.parse("2,3");
APair b = Pair.new(2,3);
APair c = Pair.bind("2,3");
c.x = 42;
APair d = Point.add(b,c);
```

BOOL

Concluding remarks

- Tuple-based records & clusters work better than ADT-based classes
- A decent alternative is to use closures
- True bidirectionality is hard in Rascal
- Bindings can form a solid algebra/category
  - ongoing work
- OO checklist
  - inheritance: between clusters
  - encapsulation: none; no easy way to hide private parts
  - polymorphism: lost; existed in Rascal on function level