GAVIN KING — RED HAT

JARGON FILE FOR MODERN LANGUAGES

“A language that doesn't affect the way you think about programming, is not worth knowing.”

–ALAN PERLIS

“You can measure a programmer's perspective by noting his attitude on the continuing vitality of FORTRAN.”

–ALAN PERLIS

Existential types

Type classes

Higher-kinded types

Unions and intersections

Flow typing

Structural typing

Pattern matching

Rank-N polymorphism

Type constructors

Monads
Functors

Algebraic data types

OMGWTF??

¿¿Que chingaos??

Why are static type systems so complicated?

- Static typing allow the creation of tools that reason about the code
- No static type checker can verify everything that a human could prove about the program
- But, for what it can prove, it is *much* faster and doesn't make mistakes

It's also much faster than tests!

And static typing can *guarantee* that a refactoring didn't break anything

- A static type system that's too simple robs us of a certain kind of expressivity — it limits what we can abstract
- A more advanced type system gives us more tools for abstraction, at the cost of complexity

UNION AND INTERSECTION TYPES

TIPOS UNION Y INTERSECCIÓN

A UNION TYPE IS A CHOICE BETWEEN TYPES

```
Null | String arg
        = process.arguments[0];
value len = arg.size; //error!
```

Note: this has nothing to do with "unions" in C

UNION TYPES MAKE GENERIC TYPE ARGUMENT INFERENCE TRACTABLE

```
File file = …;
Url url = …;
value arg = ArrayList(file, url);
```

`arg` has the inferred type `ArrayList<File|Url>`

AN INTERSECTION TYPE IS A COMBINATION OF TYPES

```
Stream<Object&T> coalesce<T>
                 (Stream<T> stream) => …;
```

```
Stream<String|Null> stream = …;
Stream<String> strings
               = coalesce(stream);
```

THE COMPILER HAS TO KNOW HOW TO REASON ABOUT UNIONS AND INTERSECTIONS

```
Object & <String|Null>
```

```
⇒ Object & String | Object & Null
```

```
⇒ String | Nothing
```

```
⇒ String
```

ELEGANT IDENTITIES FOR REASONING ABOUT SUBTYPING

$$\texttt{List<String>} \mid \texttt{List<Null>}$$

$$\subset \texttt{List<String|Null>}$$

$$\texttt{Consumer<Float>} \mid \texttt{Consumer<Integer>}$$

$$\subset \texttt{Consumer<Float\&Integer>}$$

FLOW-SENSITIVE TYPING

TIPADO SENSITIVO AL FLUJO

FLOW-SENSITIVE TYPING ADAPTS THE TYPES ACCORDING TO ASSERTIONS THAT OCCUR IN THE CODE

```
Null | String arg
        = process.arguments[0];
if (is Object arg) {
    //must be a String
    value len = arg.size;
}
```

SUM TYPES

TIPOS SUMA

Or, alternatively,

"Algebraic data types"

"Tagged union types"

A SUM TYPE IS A CHOICE BETWEEN DISJOINT CASES

```
//case 1
class Leaf(String content)
        extends Tree() {}
//case 2
class Branch(Tree left, Tree right)
        extends Tree() {}

//a sum type
class Tree() of Leaf | Branch {}
```

WE HANDLE THE CASES OF A SUM TYPE USING A SWITCH

```
Tree tree = …;

switch (tree)
  case (is Leaf) {
      print(tree.content);
  }
  case (is Branch) {
      …
  }
```

PATTERN MATCHING

CORRESPONDENCIA DE PATRONES

PATTERN MATCHING ADDS DESTRUCTURING

```
Tree tree = …;

switch (tree)
case (Leaf(content)) {
    print(content);
}
case (Branch(left,right)) {
    …
}
```

A PATTERN CAN BE COMPLEX

```
case (Branch(Leaf(x),Leaf(y))) {
    ...
}
```

TYPE FUNCTIONS

FUNCIONES DE TIPOS

Or, alternatively,

"Higher-kinded types"

"Type constructor polymorphism"

"Higher-order generics"

A FUNCTION BETWEEN VALUES

```
function f(Float x) => x*x;
```

A FUNCTION BETWEEN TYPES

```
alias F<T> => List<List<T>>;
```

- In all modern languages a function is a value — it may be passed as a reference between other functions
- Similarly, we can consider a type function to be a type, passing it as a type argument to another generic declaration

ABSTRACT OVER VALUE FUNCTIONS

```
Stream<Float> map(Float(Float) f,
                  Stream<Float> stream)
    => …;
```

f is an unknown value function that accepts just one value

ABSTRACT OVER TYPE FUNCTIONS

```
S<Float> fmap<S>(Float(Float) f,
                 S<Float> stream)
        given S<E>
    => …;
```

`S<E>` is an unknown type function that accepts just one type

INDIRECT APPLICATION OF A TYPE FUNCTION

```
List<List<Float>> stream = …;
```

```
List<List<Float>> result
        = fmap<F>(f, stream);
```

ANONYMOUS VALUE FUNCTION

```
(Float x) => x*x;
```

ANONYMOUS TYPE FUNCTION

```
<T> => List<List<T>>;
```

INDIRECT APPLICATION OF ANONYMOUS FUNCTIONS

```
List<List<Float>> stream = …;
```

```
List<List<Float>> result
      = fmap<<T> => List<List<T>>>
                ((x) => x*x, stream);
```

GENERIC FUNCTION REFERENCES

REFERENCIAS A FUNCIONES GENÉRICAS

Or, alternatively,

"Higher-rank polymorphism"

A NON-GENERIC FUNCTION

```
function f(Float x) => x;
```

Has the type `Float(Float)`

A GENERIC FUNCTION

```
function id<T>(T x) => x;
```

Give me a type, I'll give you back a function

What is its type?
I want to write `T(T)` but `T` `is` unknown!

A GENERIC FUNCTION

```
function id<T>(T x) => x;
```

It's type is a type function!

```
<T> => T(T)
```

Give me a type, I'll give you back a function type

A function of this form

```
ReturnType f<TypeParameters>(ParameterTypes)
    => …;
```

has the type

```
<TypeParameters> => ReturnType(ParameterTypes)
```

which is a function between types!

A SILLY EXAMPLE

```
[Integer,Float,String]
      h(<T> => T(T) g)
            => [g(0), g(0.0), g("")];
```

Give me a generic function, I'll apply it to three totally different types!

```
value [i, f, s] = h(id);
```

STRUCTURAL TYPING

TIPADO ESTRUCTURAL

IN A SYSTEM WITH NOMINAL TYPING, THE RELATION BETWEEN A TYPE AND IT IMPLEMENTATION IS EXPLICIT

```
interface Quack {
    void quack();
}

class Duck() satisfies Quack {
    void quack() => print("quack");
}
```

IN STRUCTURAL TYPING, WE DON'T HAVE AN EXPLICIT INHERITANCE RELATIONSHIP

```
interface Quack {
    void quack();
}

struct Duck() {
    void quack() => print("quack");
}
```

INSTEAD, THE SUBTYPE RELATIONSHIP IS VERIFIED WHEN WE ASSIGN A VALUE TO A POLYMORPHIC TYPE

```
Quack quack = Duck();
```

Do you notice a problem? — These "structural" types are only *partially* structural …
they still depend on member names

- *Unions, intersections, flow typing:* Ceylon, TypeScript, flow, Crystal
- *Pattern matching:* OCaml, Haskell, Scala, Rust, many others
- *Type functions:* Haskell, Scala, Ceylon
- *Higher rank polymorphism:* Haskell, Ceylon
- *Structural typing:* OCaml, Rust, Go, Scala