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??

¿Por qué son tan complicados los sistemas de tipado estático?

- Un lenguaje de tipado estático facilita le creación de herramientas que razonan sobre el código

- Ningún sistema de tipado estático puede verificar todo que un humano puede comprobar

- Pero, es much mas rápido y no se equivoca!

También mas rápido que los tests!

- Un sistema de tipado estático demasiado sencillo nos roba expresividad — nos impide para abstraer
- Un sistema mas avanzado presta la habilidad to abstraer, con costo de complejidad

UNION AND INTERSECTION TYPES

TIPOS UNION Y INTERSECCIÓN

UN TIPO UNION ES UNA ELECCIÓN ENTRE TIPOS

```
Null | String arg
     = process.arguments[0];

value len = arg.size; //error!
```

Ojo: no tiene nada que ver con "union" en C

AYUDA MUCH CON INFERENCIA DE TIPOS GENÉRICOS

```
File file = …;
Url url = …;

value arg = ArrayList(file, url);
```

`arg` tiene el tipo inferido `ArrayList<File|Url>`

UN TIPO INTERSECCIÓN ES UNA COMBINACIÓN DE TIPOS

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

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

EL COMPILADOR SABE RAZONAR CON UNIONES Y INTERSECCIONES

```
Object & <String|Null>
⇒ Object & String | Object & Null
⇒ String | Nothing
⇒ String
```

FLOW-SENSITIVE TYPING

TIPADO SENSITIVO AL FLUJO

TIPADO SENSITIVO AL FLUJO ADAPTA LOS TIPOS CON LAS ASERCIONES QUE APARECEN EN EL CÓDIGO

```
Null | String arg
     = process.arguments[0];

if (is Object arg) {
    //tiene que ser un String
    value len = arg.size;
}
```

SUM TYPES

TIPOS SUMA

O, alternativamente,

"Algebraic data types"

"Tagged union types"

UN TIPO SUMA ES UNA ELECCIÓN ENTRE CASOS DISJUNTOS

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

//el tipo suma
class Tree() of Leaf | Branch {}
```

TRATAMOS CON LOS CASOS DE UN TIPO SUMA CON SWITCH

```
Tree tree = …;

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

PATTERN MATCHING

CORRESPONDENCIA DE PATRONES

CORRESPONDENCIA DE PATRONES AGREGA LA HABILIDAD DE DESTRUCTURAR

```
Tree tree = …;

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

UN PATRÓN PUEDE SER COMPLEJO

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

TYPE FUNCTIONS

FUNCIONES DE TIPOS

O, alternativamente,

"Higher-kinded types"

"Type constructor polymorphism"

"Higher-order generics"

UNA FUNCIÓN ENTRE VALORES

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

UNA FUNCIÓN ENTRE TIPOS

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

- En lenguajes modernos, una función es un valor — la puedo pasar como referencia entre otras funciones
- Igual, podemos tratar con una función de tipos como un tipo, pasándola como un argumento de tipos

ABSTRAER SOBRE FUNCIONES DE VALORES

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

f es una función de valores desconocido que acepta un solo valor

ABSTRAER SOBRE FUNCIONES DE TIPOS

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

`S<E>` *es una función de tipos desconocido*
que acepta un solo tipo

APLICACIÓN

```
List<List<Float>> stream = …;
```

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

UNA FUNCIÓN ANÓNIMA ENTRE VALORES

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

UNA FUNCIÓN ANÓNIMA ENTRE TIPOS

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

APLICACIÓN CON FUNCIÓNES ANÓNIMAS

```
List<List<Float>> stream = …;
```

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

GENERIC FUNCTION REFERENCES

REFERENCIAS A FUNCIONES GENÉRICAS

O, alternativamente,

"Higher-rank polymorphism"

UNA FUNCIÓN NO GENÉRICA

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

tiene el tipo `Float(Float)`

UNA FUNCIÓN GENÉRICA

Dame un tipo, te doy una función

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

Qué tipo tiene?

`T(T)`

`T` es desconocido!

UNA FUNCIÓN GENÉRICA

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

Su tipo es una función de tipos!

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

Dame un tipo, te doy un tipo de función

Un función de la siguiente forma

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

tiene el tipo

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

que es una función entre tipos!

UN EJEMPLO TONTO

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

Dame una función genérica, la aplico a tres diferentes tipos!

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

STRUCTURAL TYPING

TIPADO ESTRUCTURAL

EN UN SISTEMA DE TIPADO NOMINAL, LA RELACIÓN ENTRE TIPO Y IMPLEMENTACIÓN ES EXPLÍCITO

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

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

EN UN SISTEMA DE TIPADO ESTRUCTURAL, NO TENEMOS QUE ESPECIFICAR HERENCIA

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

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

ES VERIFICADO CUANDO ASIGNAMOS UN VALOR A LA INTERFAZ

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

Notas un problema? — Los tipos "estructurales" son *parcialmente* estructurales ... aun dependen de los nombres de miembros

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