Tarea Independiente 03/11/2025

David Núñez Franco

November 12, 2025

Inventario de Conceptos Claves

- Eficiencia de un evaluador: Problema. Soluciones JIT (contraste AOT)
- Tipificación como una forma de evaluación.
- Type-safety (memory pollution)
- Covariancia, contravariancia invariancia

Ejercicio 1

```
Vea el API de Function en Java
Note el método compose que compone la Function<T, R>  con otra <V, R> Function
interface Function<T, R>{
 //...
 default <V> Function<V,R> compose(Function<? super V,? extends T> before)
 //...
}
Explicación:
? super V es un tipo ? desconocido, del que solo sabemos que es supertipo de
V (V <: ?)

? extends T es un tipo ? desconocido, solo sabemos que es subtipo de
T (? <: T)

? super V en esencia se usa para declarar contravariante en V y ? extends
T covariante en T

Es decir, para una función (de un argumento)
Contravariancia en el argumento: la función puede aceptar algo más general que V
(o sea al menos acepta V).

Covariancia en el retorno: función puede producir algo más específico que T
```

(a lo más T). Por ejemplo:
Apple es más específico que Fruit (toda manzana es fruta, Apple <: Fruit)
Thing es más general que Fruit (toda fruta es una cosa, Fruit <: Thing).

Explique este enredo construyendo un ejemplo que use compose con manzanas, frutas y cosas o algo así. Debe compilar y correr.

Solución

```
import java.util.function.Function;

class Thing {}                    // supertipo mas general
class Fruit extends Thing {}      // intermedio
class Apple extends Fruit {}      // subtipo mas especifico

public class VarianceComposeDemo
{
    public static void main(String[] args) {
        // this: Function<T, R> con T = Fruit, R = String
        Function<Fruit, String> fruitToReport =
        f -> "OK: recibi un " + f.getClass().getSimpleName()
            + " y regreso String";

        // before: Function<? super V, ? extends T>
        // Elegimos V = Apple, T = Fruit
        // Contravarianza en el argumento: acepto algo MAS
           GENERAL que Apple -> Thing
        // Covarianza en el retorno: produzco algo MAS
           ESPECIFICO que Fruit -> Apple
        Function<Thing, Apple> thingToApple =
        f -> new Apple();

        // compose: (fruitToReport * thingToApple) :
           Function<V,R> -> Function<Thing, String>
        Function<Thing, String> pipeline = fruitToReport.
           compose(thingToApple);

        // Demostracion 1: paso un Thing; before lo consume
           (contravariante) y produce Apple (covariante)
        System.out.println(pipeline.apply(new Thing()));

        // Demostracion 2: tambien funciona con subtipos de
           Thing
        System.out.println(pipeline.apply(new Fruit()));
        System.out.println(pipeline.apply(new Apple()));

```

```
                // Caso alterno: before que acepta exactamente Apple
                    y devuelve un subtipo de Fruit (Apple)
                Function<Apple, Apple> idApple =
                a -> a;
                Function<Apple, String> applePipeline =
                   fruitToReport.compose(idApple);
                System.out.println(applePipeline.apply(new Apple()))
                   ;
            }
        }
```

```
        OK: recibi un Apple y regreso String
        OK: recibi un Apple y regreso String
        OK: recibi un Apple y regreso String
        OK: recibi un Apple y regreso String
```

Listing 1: output

Ejercicio 2

En nuestro demo typer.pl: a) Maneje los demás operadores aritméticos *, -, /, ** (reutilice código, no haga copy-and-paste). Maneje también el menos (-) unario como en -(x + 666).

b) Añada un tipo double que acepte int (pero no al contrario). Una operación de un double con un int en cualquier orden siempre da double.

c) Permita que en la suma + si al menos un argumento es string el operador se se interpretaría como concatenación de hileras. ¿Es eso una buena idea?

Solución

```
        % typer(+Expr, +Ctx, -Type) Type is the type of Expr in
           context Ctx. Recursive algoritm.

        % CONTEXTO %
        context_find(null, _, _) :- fail.
        context_find(ctx(Locals, Parent), X, TX) :-
        ( context_locals_find(Locals, X, TX) -> true ; context_find(
           Parent, X, TX) ).
        context_locals_find(C, X, V) :- member([X, V], C).

        % PRIMITIVOS Y SUBTIPADO %
        type_primitive(any).
        type_primitive(int).
        type_primitive(double).
        type_primitive(string).
```

```
type_primitive(boolean).

% int <: double %
type_accept(any, _).
type_accept(T, T) :- type_primitive(T).
type_accept(double, int). % un int es aceptable donde se
   espera un double
% arrow (contravariante en el argumento, covariante en el
   resultado) %
type_accept(X >> Y, A >> B) :-
type_accept(A, X),
type_accept(Y, B).

% CASOS BASE %
typer(N, _, int)     :- integer(N).
typer(F, _, double)  :- float(F).
typer(S, _, string)  :- string(S).
typer(B, _, boolean) :- member(B, [true,false]).
typer(X, C, TX)      :- atom(X), context_find(C, X, TX).

% OPERADORES %
% Binarios soportados: +, -, *, /, ** con promocion numero y
    caso especial para + con string
typer(L + R, C, string) :- % concatenacion de strings
(typer(L, C, string) ; typer(R, C, string)),
!.
typer(L + R, C, T) :-
typer(L, C, TL),
typer(R, C, TR),
bin_numeric_result(+, TL, TR, T).

typer(L - R, C, T) :-
typer(L,C,TL), typer(R,C,TR),
bin_numeric_result(-, TL, TR, T).
typer(L * R, C, T) :-
typer(L,C,TL), typer(R,C,TR),
bin_numeric_result(*, TL, TR, T).
typer(L / R, C, T) :-
typer(L,C,TL), typer(R,C,TR),
bin_numeric_result(/, TL, TR, T).
typer(L ** R, C, T) :-
typer(L,C,TL), typer(R,C,TR),
bin_numeric_result(**, TL, TR, T).

% unario menos: -(Expr)
typer(-E, C, T) :-
typer(E, C, TE),
```

```
58          unary_numeric_result('-', TE, T).
59
60          % REGLAS DE TIPADO NUMERICO %
61          % solo tipos numericos validos para aritmetica
62          num_type(int).
63          num_type(double).
64
65          % promocion numerica
66          % - Si alguno es double, el resultado es double
67          % - Si ambos son int, el resultado es int
68          promote_numeric(T1, T2, double) :- (T1 == double ; T2 ==
               double), !.
69          promote_numeric(int, int, int).
70
71          % resultado binario numerico con promocion; falla si no es
               numerico
72          bin_numeric_result(_, TL, TR, T) :-
73          num_type(TL), num_type(TR),
74          promote_numeric(TL, TR, T).
75
76          % resultado unario numerico; preserva el tipo
77          unary_numeric_result('-', T, T) :-
78          num_type(T).
79
80          % -------------------- PRUEBAS --------------------
81          test_typer_0 :-
82          writeln('>>> Typer testing 0'),
83          E = x,
84          C = ctx([[x,int]], null),
85          (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
               format('>>> ~w failed~n', [E])).
86
87          test_typer_1 :-
88          writeln('>>> Typer testing 1'),
89          E = 666,
90          C = ctx([[x,int]], null),
91          (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
               format('>>> ~w failed~n', [E])).
92
93          test_typer_2 :-
94          writeln('>>> Typer testing 2'),
95          E = true,
96          C = ctx([], null),
97          (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
               format('>>> ~w failed~n', [E])).
98
99          test_typer_3 :-
```

```
        writeln('>>> Typer testing 3 (+ int)'),
        E = x + 666,
        C = ctx([[x, int]], null),
        (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
           format('>>> ~w failed~n', [E])).

        test_typer_4 :-
        writeln('>>> Typer testing 4 (+ string concat)'),
        E = "hola" + 3,
        C = ctx([], null),
        (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
           format('>>> ~w failed~n', [E])).

        test_typer_5 :-
        writeln('>>> Typer testing 5 (* double promotion)'),
        E = 2.0 * 3,
        C = ctx([], null),
        (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
           format('>>> ~w failed~n', [E])).

        test_typer_6 :-
        writeln('>>> Typer testing 6 (unary -)'),
        E = -(x + 666),
        C = ctx([[x,int]], null),
        (typer(E, C, TE) -> format('>>> ~w :: ~w~n', [E, TE]) ;
           format('>>> ~w failed~n', [E])).

        :-  test_typer_0,
        test_typer_1,
        test_typer_2,
        test_typer_3,
        test_typer_4,
        test_typer_5,
        test_typer_6.
```

```
        ?- [typer].
        >>> Typer testing 0
        >>> x :: int
        >>> Typer testing 1
        >>> 666 :: int
        >>> Typer testing 2
        >>> true :: boolean
        >>> Typer testing 3 (+ int)
        >>> x+666 :: int
        >>> Typer testing 4 (+ string concat)
        >>> hola+3 :: string
        >>> Typer testing 5 (* double promotion)
```

```
13          >>> 2.0*3 :: double
14          >>> Typer testing 6 (unary -)
15          >>> - (x+666) :: int
16          true.
```

Listing 2: output