Generics, tuples, pattern-matching, compile-time class constants and static arrays, NULL(Void)-safety is here as well:

1. Generics – let’s use [] brackets for generics
2. Tuples – let’s use () brackets for tuples

So, then we have Array [G] that is generic class Array with one formal generic parameter called G. if we have Array (1, 15, 64, 75) – that is an object of type Array with values – 1, 15, 64, 75

That is class text

class Array [G]

end

That is how one can use tuples

a: (field1: Integer; field2: Real) – that is unnamed class type which has two fields. Or we can just state that is a tuple with 2 fields.

That is how one can work with fields of the tuple.

a.field1 := 5

a.field2 := 15.7

So, tuple is in fact an object. If it is an object then we have a very interesting cases like

Can we dynamically extend tuples – yes we can. So, we have for example class RunTime which has the following routines

RunTime.add (t1, t2: []) – it adds t2 to t1 – like add ax, bx in IA ☺

….

Two types of inspect instruction – in both cases inspect instruction is a pattern matching mechanism

Variant A - dynamic type matching

Variant B – dynamic value matching

Let’s consider examples to understand the concept starting with A

case

some\_variable

when t1: TYPE1 then

-- Here we have local variable t1 of TYPE1 assigned with the value of some\_variable as its dynamic type conforms to TYPE1.

when t2: TYPE2 then

when t3: TYPE3 then

else

--- Different type! Or Void is some\_variable was declared as detachable object

end -- case

Then the second example about inspecting value of an object at runtime

enum: Integer is red, green, blue;

case

some\_variable

when 5 then

when Array(1,2,5,75) then ---: BTW – that is static Array!!!! Need to think how we deal with static and dynamic arrays!!!

when TYPE\_A(field1:=54; field2 := TYPE\_B(field1:= “STRING”)) then --- so, in fact we introduce the concept of constant objects – the objects which are created at compile time

when 6..12, 27, 74 then ---- rather straightforward meaning

when red then

when blue, green then

else

end

Can we mix value test and dynamic type test? Probably no

At the same time we know that every case instruction can be written as chain of if statements like

if equal (some\_variable, 5) then

elseif equal (some\_variable, Array(1,2,5,75]) then

elseif equal (some\_variable, TYPE\_A(field1:=54; field2 := TYPE\_B(field1:= “STRING”])) then

elseif equal (some\_variable, 27) or else equal (some\_variable, 74)

then

elseif equal (some\_variable, red) then

elseif equal (some\_variable, blue) or else equal (some\_variable, green) then

elseif some\_variable is integer\_variable: Integer and then 6 <= integer\_variable and then integer\_variable <= 12 then

else

end

Can compiler generate more efficient code than just converting into if sequence – of course. Will it optimize the case when inspected variable is of type Integer or CHARCTER all when clauses has Integer or CHARACTER constants – of course! So, we keep the efficiency of classical switch(C)/case(Pascal)/inspect(Eiffel) and add pattern matching capabilities for more complicated objects.

Just to remind that we have two syntax forms of dynamic object test in form of inspect as shown above and in form of if statement used to deal with 6..12 case

~~But as we started to use for object creation the name of the type with square brackets let’s see if we need a keyword new/create for objects creation as a syntax construction~~

~~a: TYPE~~

~~a := new TYPE.creation\_procedure\_call ()~~

~~So, we can try of course something like this~~

~~[TYPE]a.creation\_procedure\_call ()~~

~~it will work and we may save one keyword. All forms will work~~

~~[TYPE\_WITH\_NO\_CREATION\_PROCEDURE]a~~

~~[]a.creation\_procedure\_of\_the type\_used\_for\_declaration\_of\_a ()~~

~~[]a ---- looks ugly - but the meaning is straightforward – create an object~~

~~And this is identical in terms of syntax to what we had in Eiffel from its birth but instead of !!a we are forced to write []a ☹ In this case I prefer !! ☺~~

So, it looks it is necessary to highlight the difference between compile-time created object and run-time created ones … But going back to inspect value test – it can have not only compile-time created objects but run-time created ones too like below

inspect

some\_variable

when TYPE (field1 := some\_expression) then

when some\_object then

else

end ---- inspect

some construct TYPE (field1 := some\_expression) will be fully evaluated at run-time and the second case is a function call or variable access which will be dynamically dispatched and executed at runtime. So, we keep the possibility to do compile time optimization and dynamic flexibility within the same language construct … And support different types of pattern matching: dynamic type matching and compile time and runtime value matching. So, I like pattern matching and it does not require case classes ☺

Plus it looks there is something about static and dynamic arrays – let’s see

a: Array [Integer] ---- that is dynamic array – it has no elements

a := (1, 24, 23, 45) ---- now it was initialized with compile time constant array but it was declared as dynamic array and it is dynamic array!

a.add (75) --- now the array looks like 1, 24, 23, 45, 75

a.add ((3, 5,6)) -- and now it is 1, 24, 23, 45, 75, 3, 5, 6

but if we remember that we can use value instantiated temples and use constant as a value then

we declare

b: STATIC\_ARRAY [1..5, Integer]

Of course class STATIC\_ARRAY inherits from Array and it will be necessary to think about how to deal with indices and NULL safety … WIP

Some assumptions more precisely some rules taken from Eiffel ☺

Conformance rules

Any tuple conforms to () – empty tuple. Tuple is a reference object unless it is declared expanded explicitly.

e0: val ()

e1: val (Integer)

e2: val (Integer; Real)

e0 := e1 ---- it works!!! How many fields has e0 after assignment? Yes – ZERO!

e1 := e2 ---- it works as well!!! How many fields has e1 after assignment? Yes – 1 of type INETGER and it has the same value as e2 first field.

So, we keep the semantics of the assignment

reference1 := reference2 --- both references are attached to the same object

expanded := reference --- field by field copy

reference := expanded -- new reference object created as a shallow copy of expanded

expanded := expanded --- field by field copy

r0 : () // that is in fact the top of the classes hierarchy!!! That is ANY/OBJECT/whatever and it raises the question if we need this reserved name ANY/OBJECT …. WIP!!!

r1: (Integer)

r2: (Integer; Real)

r0 := r1 // both references are attached to the same object

r1 := r2 // both references are attached to the same object

WIP !!!!

class List [G]

item: G

next: ?List [G]

add (an\_item: like item) is

next := List [G] (an\_item)

end add

ctor, setItem (an\_item: like item) is

item := an\_item

end

ctor (items: Array [G]) is

while anItem:

end

end List

l1: ?List [Integer] //

l2: List [Integer] (5)