**Embedded or enclosed terms are synonyms**

**No need for embedded units!**

Let’s consider a simple example with embedded units. Inside of unit A, we have declared unit B which is accessible only within the body of A and it can be used for internal purposes of A. B has access to the context of A. That is the classical approach to embedded routines with 2 cases of top-level unit usage

unit A

attr: Type

unit B

foo **do**

attr.routine(…) // case #1: That is what we can get

**end**

**end**

foo **do**

b **is** B // case #2: That is what we can get from embedded units

b.foo // activation of case #1

**end**

**end**

But do we need this indeed? Case #2 can be easily implemented if unit B is at the same level as A. case #1 is trickier. We directly access attributes of the top-level unit. So, it creates additional client dependency between B and A. In fact, it is similar to the code

unit B

foo **do**

client.attr.routine(…) // case #1: That is what we can get

**end**

**init** (c: **as** client) **do** client := c **end**

{} client: A

**end**

unit A

attr: Type

foo **do**

b **is** B.**init** (**this**)

b.foo // activation of case #1

**end**

**end**

So, it looks there are no strong reasons for embedded units at all!

**No need for embedded routines!**

Let’s consider a simple example with embedded routine. Inside of routine foo, we have declared routine goo which is accessible only within the body of foo and it can be used for internal purposes of foo. Goo has access to the context of foo. That is the classical approach to embedded routines.

foo (T1, … Tn) **do**

goo (U1, … Um) **do**

**…**

**end**

goo (U1, …, Um)

**end**

During the recent decade, we observe convergence between object-oriented and functional programming approaches and routine types and expressions give a solid alternative for embedded routines

foo (T1, … Tn) **do**

goo **is routine** (U1, … Um) **do**

**…**

**end**

goo (U1, …, Um)

**end**

So, there is no more need for embedded routines as a functional style of programming allows reaching the same result keeping static type checks in place and providing even better flexibility due to reacher control mechanisms

foo (T1, … Tn) **do**

**const** goo **is routine** (U1, … Um) **do**

/\* In this case, this is 100% embedded routine

If we keep goo as a variable it can be attached to another routine or it can be returned as some function result to the whole world outside of foo \*/

**…**

**end**

goo (U1, …, Um)

**end**

One may exclaim that embedded routine is just a short cut of writing instead of **const** goo **is routine** the programmer can just type goo, that is very close to the true statement with the only exception. If a routine is declared in an ordinary way we need explicitly create a routine object from it

foo (T1, … Tn) **do**

**…**

**end**

**const** goo **is routine** (U1, … Um) **do**

**…**

**end**

// Call is the same

foo (T1, … Tn)

goo (U1, …, Um)

// Meaning in expressions is different

… := foo // Semantic error

… := **routine** foo /\* OK, lvalue will be attached to the routine object of foo \*/

… := goo // OK, lvalue will be attached to routine object goo

It looks beneficial to treat any routine declaration in form of just <routine\_name> as const <routine\_name> is routine <signature> and then treat the routine name as a reference to the routine object. But functions ruin this ideal picture

foo (T1, … Tn): RT1 **do**

**…**

**end**

**const** goo **is routine** (U1, … Um): RT2 **do**

**…**

**end**

// Call is the same

x1 **is** foo (T1, … Tn)

x2 **is** goo (U1, …, Um)

// Meaning in expressions is different

… := foo (…) // That is a valid call to foo !!!

… := **routine** foo // OK, lvalue will be attached to the routine object

So, let’s just drop the concept of embedded routines as they required in the limited number of cases staying with routine typoes and expressions.

P.S. In fact, it leads to very interesting considerations – every unit has only data attributes

**unit** A

**const** foo **is routine** (<signature>) <body>

**const** goo **is** Type

variable\_attribute: Type

**const init is routine do**

variable\_attribute := …

**end**

**end**

**unit B extend A**

**override** **const** foo **is routine** (<conformantSignature>) <newBody>

**override** **const** goo **is** confromantType /\* No, that is **NOT** a semanic error, it is safe override constant attribute while inheriting \*/

**override** variable\_attribute: confromantType

**const init is routine do**

variable\_attribute := …

**end**

**end**

So, all members of the unit are data attributes together with rules for overriding while inheriting! ☺

This leads to another dramatic change – all attributes are constants by default and then we need keyword var to highlight variables. Then all attributes by default use SSA form – single static assignment and all exceptions are explicitly marked with **’var’** keyword. Here comes an example of SLang++ :-)

**I am sorry but I am about to redesign the basic foundation of SLang … ☹**

**unit** A

foo [ **is rtn** ][<Signature>] **do** /\* That is unit routine!!! In fact a constant attribute of the unit\*/

localConstant **is** someExpression

**var** localVariable **is** anotherExpression

**end**

**var** too1: **rtn** (T1, T2) /\* that is a procedure with 2 arguments

to be initialized within init procedure \*/

**var** too2: **rtn** **as** foo

// that is a routine with the type the same as foo has

goo **is** Type // That is a constant attribute of the unit

**var** variable\_attribute: Type /\* That is a variable attribute of the unit \*/

**init do**

variable\_attribute := …

too1 := **rtn** (U1, U2) **do**

// where U1 conforms to T1 and U2 conforms to T2

**end**

**end**

demo do // that is a short cut for demo **is** **rtn** () **do**

r1 **is rtn** demo /\* That is a constant of routine type which refers to routine called demo (itself)\*/

**while** i **in** 1..10 **do**

// For every loop iteration i is a constant!!!

**end**

**do**

// programmer can assign to j within the loop body

**while var** j **in** 1 .. 100 **do**

**end**

**end**

Standalone routines stays unchanged syntactically. The only conceptual change that standalone rouitines can be treated as both local attributes of the anonomous routine of the current compilation source (unit) and as global attributes as they can be accessed from other compilation units

foo [ **is rtn** ][<Signature>] **do** /\* That is in fact global constant

attributes!!! \*/

…

**end**

And standalone routine **cannot** be declared in var form, like

**var** foo [ **is rtn** ][<Signature>] **do**

…

**end**

The bottleneck is former **const deep** construction. Now, the keyword **const** stays to highlight contact objects and their import. And it looks deep constants will look like

**deep** tree: Tree[Element]

Maybe another keyword should be introduced to highlight the deep nature of the constant. Immutable? Rigid? Instead, fo deep to use rigid? It is short at least and not related to BM terminology :-)

**rigid** tree: Tree[Element] // The whole tree object is immutable

**const** // It will work for constant object definitions as well

**rigid** a.**init** (1,2), // The whole a is immutable

b, /\* Can not assign to b, but its internal attributes can be changed calling b procedures \*/

**rigid** c // The whole c object is immutable

**end**

Of course when we say that unit is a collection of attributes (data members) it does not directly imply that all objects of the unit type will have physically data fields for all constant attributes. Some constant attributes will be shared across all objects (routines) some will be spread across the code – like Integer, Real or Boolean constants. But this is part of potential optimization which are subject to be implemented by the particular compiler.

--- end of the document