Beyond C++: SLang

Eugene Zouev,
Innopolis University, Kazan

Alexey Kanatov
Samsung R&D Center, Moscow

Agenda

- Introduction
- Compilation units – anonymous procedures and units
- Operators – if & loop
- Approach to inheritance, feature call validity
- Null-safety and non-initialized attributes
- Constant objects
- Standard library basics
- Extended overloading
- Unit extensions
- Generics
- Dining philosophers
- Summary

Introduction

- **Authors’ background**: C++, Ada, Modula-2, Zonnon, Eiffel – battle ☺
- **Terminology**: feature – routine or attribute, attribute – variable or constant, routine – procedure or function; inheritance graph & conformance; module, type, class
- **Main task** is to give high-level overview of feature which could be of interest ☺. It is not possible to give full SLang description in 20 minutes. The book is to follow …

Compilation units

3 kinds:

- **Anonymous procedure:** sequence of operators
- **Standalone-routine:** scope, formal parameters, pre & post conditions, body
- **Unit**: named set of routines and attributes, invariant
    - Can be generic - type or constant expression of enumerated type parameterized
    - Unit defines type
    - Unit supports inheritance
    - Unit support direct usage (module)

Unit(module) name

```
StandardIO.put("Hello world!\n")
routine (“ha-ha-ha”)
```

New shorter name of the unit

```
use StandardIO as io
routine(aString: String) is
    io.put("Test!\n")
    c is C("This is a string“)
    io.put(c.string + “ “ + aString)
end
```

Standalone procedure

```
unit C
    string: String
    init (aString: as string) is
        string := aString
    end
end
```

Unit

Units - 3 in 1 (class, module, type)

Usage (module)

Client gets access to visible features of the module

Inheritance (class)

Unit inherits features of the base units treating them as classes

Typification (type)

Each unit defines a type. This type can be used to define attribute, local or argument

Usage(module)

```
StandardIO.put("Hello world!\n")
routine (C)
```

Inheritance(class)

```
unit C extend B, ~D use B
end
```

Typification (type)

```
routine(b: B) use D is
    D.foo
end
```

Usage(module)

```
unit B is
    foo is
    end
end
```

Inside units - definitions

Routines can be procedures or functions

- `a is end` // that is a procedure without parameters, one may put () after routine name
- `foo: T is end` // that is a function without parameters which returns an object of type T

Unit attributes can be variable or constant

- `variable: Type`
- `const constant: Type`

Routines may have locals which can be also variable or constant

- `variable is expression`
- `const constant is expression`

Inside units - example

```
unit X
        const constant1: Type is someExpression
        const constant2 is someExpression
        variable0: Type
        variable1: ?Type // variable1 is explicitly non-initialized.
        variable2 is someExpression
        variable3: Type is someExpression
        routine is
                const routineConstant1: Type is someExpression
                const routineConstant2 is someExpression
                routineVariable1: Type is someExpression
                routineVariable2 is someExpression
        end
        init is
                variable0 := someExpression // That is an assignment
                // constant1 := someExpression // Compile time error
        end
end
x is X; y is X.variable0
```

How to build a program?

Entry points:

- Anonymous procedure: First statement is the entry point

```
StandardIO.put("Hello world!\n")
routine ((“ha-ha-ha”))
```

- Visible stand-alone procedure

```
routine(strings: Array[String]) is
end
```

- Initialization procedure of some unit

```
unit C
    init is end
end
```

Global context:

- All top level units and stand-alone routines are mutually visible
- Name clashes are resolved outside of the language

Source 1:

```
        foo is end
        unit A is foo is do end end
```

Source 2:

```
        goo is end
```

Source 3:

```
        foo
        goo
        a is A
        a.foo
```

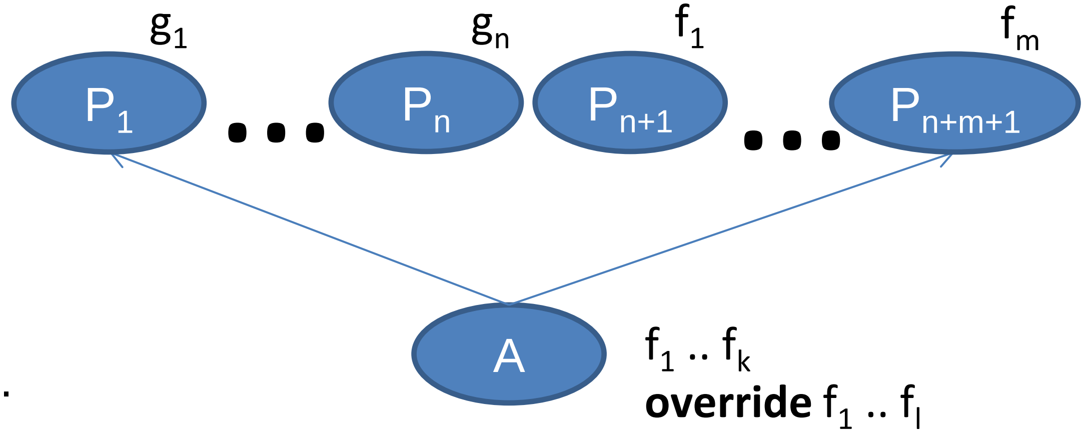
Operators – if & loop

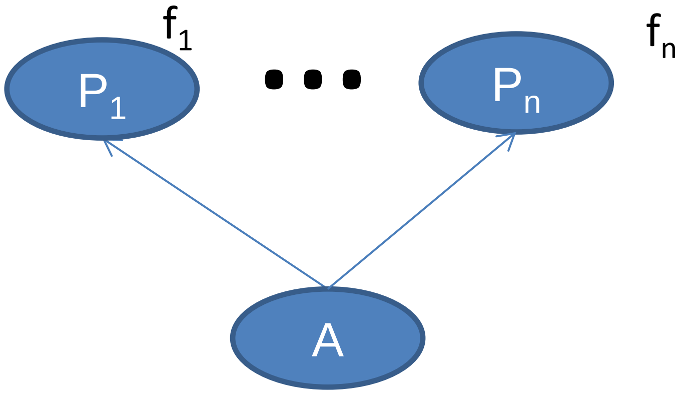
- One conditional statement and one loop
- 2 forms of conditional statements
- 3 forms of the loop

```
if condition then
        thenAction
else
        elseAction
end

if a is
   T1: action1 // where T1 is type
   E2: action2 // where E2 is expression
   else action3
end

while index in 1..10 loop
   body
end

loop
   body
while condition end

loop
   body
end
```

Approach to inheritance, feature call validity-1

- **Override in a unit:**
  - g_i is identical to g_j then only one g is inherited
  - g_1 .. g_n are inherited as is
  - f_1 .. f_k are introduced in A, new features
  - $_l \leq _m$, let f_1 .. f_l override some of f_1 .. f_m based on signature conformance then remaining (not overridden) of f_1 .. f_m are inherited as is
- **Override while inheriting**:
  - f_i will override f_1 .. f_k , where $_k < _n$, based on signature conformance
  - then A will have f_1 .. f_{n-k+1} features
- **Feature call validity**
  - Call is valid when it can be unambiguously resolved!
  - There is only one visible f in A with the signature (T_1..T_n) to which (ET_1..ET_n) conforms

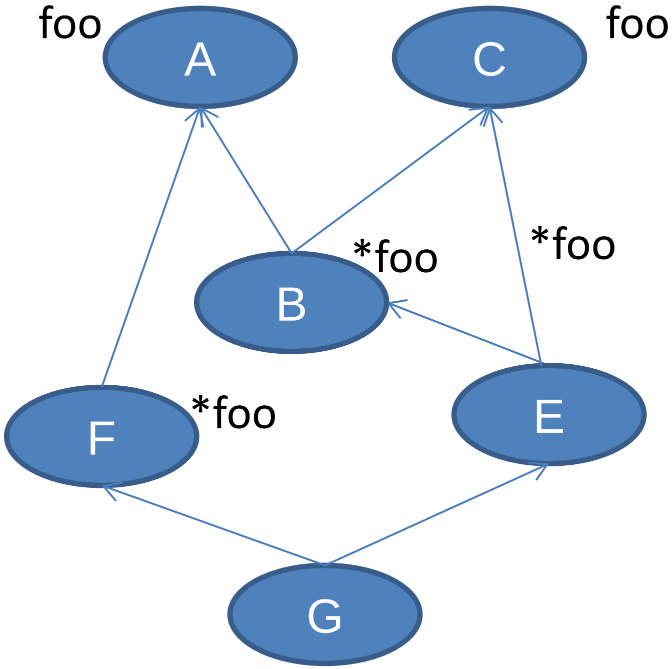




```
// P1..Pn – base units for A
// E1..En – expressions of types ETi
a is A
a.f(E1, .. En)
// Is it a valid feature call?
```

Approach to inheritance, feature call validity-2

- High-level approach: multiple inheritance with overloading and conflicting feature versions while checking feature call validity per call.
- Mandatory validity check for the inheritance graph :
  - No cycles in inheritance graph
  - All polymorphic version conflicts resolved (‘select’)



```
abstract unit A
    foo (T) is abstract
end
unit C
    foo (T) is end
end
unit B extend A, C
    override foo (T) is end
end
unit E extend C, B
    override C.foo
end
unit F extend A
    override foo (T1) is end
end
unit G extend F, E
    use E.foo
end
```

Null-safety and non-initialized attributes

Key principles:

- Every entity must be initialized before any access to its attributes or routines
- If one needs to declare an entity with no value, it is not possible to access its attributes or routines.
- There must be a mechanism how to check that some entity is a valid object of some type and safe access to its attributes/routines can be granted
- Entity which was declared as no-value entity may loose its value
- Not able to assign
- Works for value type
- There is no NULL/NIL/Void at all ☺

```
e1 is 5 // Type of e1 is deduced from 5
e2: Type is Expression /* Type of Expression
must conform to Type*/
unitAttr: Type /* init must assign value to
untiAttr*/

entity: ?A // entity has no value!!!


if entity is A then /* check if entity is of
type A or its descendant and only then deal
with it */
        entity.foo
end

? entity // detach the entity.

a: A is entity // Compile time error!

i: ?Integer
i := i + 5 // Compile time error!
if i is Integer then i := i + 5 end
```

Constant objects

- Every unit may define all known constant objects using **const is**
- Integer.1 is valid constant object of type Integer
- To skip unit name prefix use **use const**

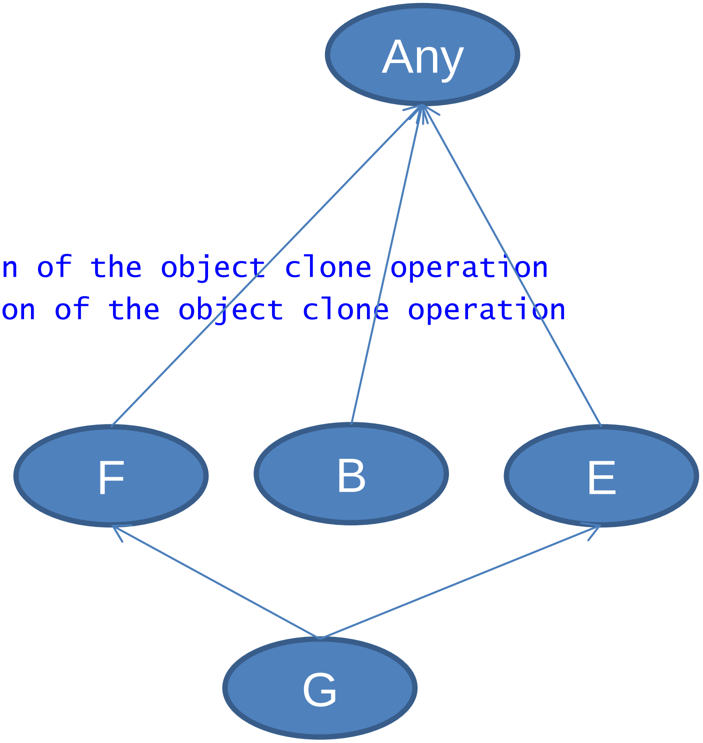
```
val unit Integer extend Integer
          [Platform.IntegerBitsCount] …
end
val unit Integer [BitsNumber: Integer] extend
Numeric, Enumeration is
   const minInteger is - (2 ^ (BitsNumber - 1))
   const maxInteger is 2 ^ (BitsNumber - 1) - 1
   const is /* That is ordered set defined as
range of all Integer constant values (objects) */
      minInteger .. maxInteger
   end
   init is
      data := Bit [BitsNumber]
   end
   hidden data: Bit [BitsNumber]
invariant
   BitsNumber > 0 /* Number of bits in Integer
must be greater than zero! *.
end
abstract unit Any use const Integer, Real,
Boolean, Character, Bit, String is
end
```

Constant objects - examples

```
unit WeekDay
   const is Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
Sunday end
end
use const WeekDay foo (Monday)
foo (day: WeekDay) is
   if day is
      Monday .. Friday: StandardIO.put (“Work day – go to the
office!\n”)
     Saturday, Sunday: StandardIO.put (“WeekEnd – do what you like!\n”)
   end
end
unit A
   const is a1.init, a2.init (T), a3.init (T1, T2)
   end
   init is end
   init (arg: T) is end
   init (arg1: T1; arg2: T2) is end
end
const x is A.a1
y is A.a2
```

Standard library basics: everything is defined

```
abstract unit Any use const Integer, Real, Boolean, Character, Bit, String is
   /// Shallow equality tests
   = (that: ? as this): Boolean is external
   final /= (that: ? as this): Boolean is return not ( this = that) end
   = (that: as this): Boolean is external
   final /= (that: as this): Boolean is return not ( this = that) end
   /// Deep equality tests
   == (that: ? as this): Boolean is external
   final /== (that: ? as this): Boolean is return not ( this == that) end
   == (that: as this): Boolean is external
   final /== (that: as this): Boolean is return not ( this == that) end
   /// Assignment definition
   hidden := (that: ? as this) is external
   hidden := (that: as this) is external
   /// Utility
   toString: String is external
   sizeof: Integer is external ensure return >= 0 end
end // Any
unit System is
   clone (object: Any): as object is external /// Shallow version of the object clone operation
   deepClone (object: Any): as object is external /// Deep version of the object clone operation
end // System
unit Platform is
   const IntegerBitsCount is 32
   const RealBitsCount is 64
   const CharacterBitsCount is 8
   const BooleanBitsCount is 8
   const PointerBitsCount is 32
   const BitsInByteCount is 8
end // Platform
```



Standard library basics: everything is defined

```
val unit Boolean extend Enumeration is
           const is false.init (0), true.init (1) end
           override < (other: as this): Boolean => not this => other
           override = (other: as this): Boolean => this.data = other.data
           succ: as this => if this then false else true
           pred: as this => if this then false else true
           override const first is false
           override const last is true
           const count is 2
           ord: Integer => if this then 1 else 0
           override sizeof: Integer => Platform.BooleanBitsCount / Platform.BitsInByteCount
           & alias and (other: as this): Boolean =>
                      if this then if other then true else false else false
           | alias or (other: as this): Boolean =>
                      if this = false then if other then true else false else true
           ^ alias xor (other: as this): Boolean =>
                      if this then if other then false else true else if other then true else false
           => alias implies (other: as this): Boolean => not this or other
           ~ alias not : Boolean => if this then false else true
           toInteger: Integer => if this then 1 else 0
           init (value: as this) is data := value.data end
           init is data := 0xb end
           hidden init (value: Integer) require value in 0..1 is data := value end
           hidden data: Bit [Platform.BooleanBitsCount]
invariant
           this and this = this /// idempotence of 'and'
           this or this = this /// idempotence of 'or'
           this and not this = false /// complementation
           this or not this = true /// complementation
end // Boolean
```

Extended overloading

Two units are different when they have different names or they have different number of generic parameters

```
i1: Integer is 5
i2: Integer[8] is 5

s1: String[3] is
“123”
S2: String is “123”

a1: Array[Integer, 3]
is (1, 2, 3)
a2: Array [Integer]
is
(1, 2, 3)
a3: Array [Integer,
(6,8)] is (1, 2, 3)
```

```
val unit Integer extend Integer
[Platform.IntegerBitsCount] … end
val unit Integer [BitsNumber: Integer] … end
abstract unit AString /* String abstraction */
… end

unit String [N:Integer] extend AString, Array
[Character, N] /* Fixed length string*/ … end
unit String extend Astring /* Variable length
String*/ … end

abstract unit AnArray [G] /* One dimensional
array abstraction*/ … end
unit Array [G->Any init (),
N: Integer|(Integer,Integer)]
extend AnArray [G] /* Static one dimensional
array*/ … end
unit Array [G -> Any init ()] extend AnArray
[G] /* Dynamic one dimensional array*/ … end
```

Unit extensions

- All sources are compiled separately
- Smart linking is required to support valid objects creation
- Source4 validity depends on what sources are included into the assembly

```
Source1:
unit A
   foo is local is A end
end
Source2:
extend unit A
   goo is end
end
Source3:
extend unit A extend B
   override too is end
end
unit B
   too is end
end
Source4:
a is A
a.too
a.foo
a.goo
```

Generics - example

- Standalone routines can be parameterized by type and /or value

```
x1 is factorial1 [Integer] (3) /* call to
factorial1 function will be executed at run-
time */

x2 is factorial2 [3] /*This call can be
processed at compile-time!!!*/

factorial1 [G->Numeric] (x: G): G is
   if x is
      x.zero, x.one: return x.one
   else
      return x * factorial1 (x - x.one)
   end
end

factorial2 [x:Numeric]: as x is
   if x is
      x.zero, x.one: return x.one
   else
      return x * factorial2 [x - x.one]
   end
end
```

Dining philosophers - example

```
philosophers is (concurrent Philosopher ("Aristotle"), concurrent Philosopher ("Kant"), concurrent
Philosopher ("Spinoza"), concurrent Philosopher ("Marx"), concurrent Philosopher ("Russell"))
forks is (concurrent Fork (1), concurrent Fork (2), concurrent Fork (3), concurrent Fork (4), concurrent
Fork (5))
check
   philosophers.count = forks.count or else philosophers.count = 1 and then forks.count = 2
   /* Задача валидна, если число вилок совпадает с числом философов или, если философ - один, то ему
просто нужны две вилки*/
end
loop /// Пусть философы едят бесконечно. Возможен и иной алгоритм симуляции …
   while seat in philosophers.lower .. philosophers.upper loop
      StandardIO.put ("Philosopher '" + philosophers (seat).name + "' is awake for lunch\n")
      eat (philosophers (seat), forks (seat), forks (if seat = philosophers.upper then forks.lower else
seat + 1)
   end
end
eat (philosopher: concurrent Philosopher; left, right: concurrent Fork) is
   /* Процедура - eat с тремя параллельными параметрами, вызов которой и образует критическую секцию
параметризованную ресурсами, которые находятся в эксклюзивном доступе для этой секции */
   StandardIO.put ("Philosopher '" + philosopher.name + "' is eating with forks #" + left.id + " and #" +
right.id + "\n")
end
unit Philosopher is
   name: String
   init (aName: as name) is name := aName end
end
unit Fork is
   id: Integer
   init (anId: as id) is id := anId end
end
```

Summary

Presented

- Key concepts of SLang
  - Units, standalone routines, usage-inheritance-typification
  - Alternative approach to inheritance
  - NULL-safety and non-initialized data 2 in 1

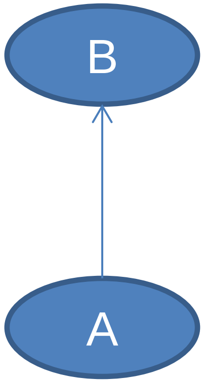
Status

- Short introduction to the language (PP presentation)
- 3 conference papers
- The full **language reference** (in progress)
- Front end compiler implementation (in progress)

THANK YOU VERY MUCH!!!

Conformance

1. Unit A conform to unit B if there is a path in inheritance graph from A to B.
2. Signature foo conforms to signature goo if every type of signature foo conforms to corresponding type of signature goo.



```
unit B
end

unit A extend B
end
```

goo (T_1, T_2, … T_n)

foo (U_1, U_2, … U_n)

if for $_i$ in $_1$ .. $_N$

U_i conforms to T_i

We *can* – therefore we *must*