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
```

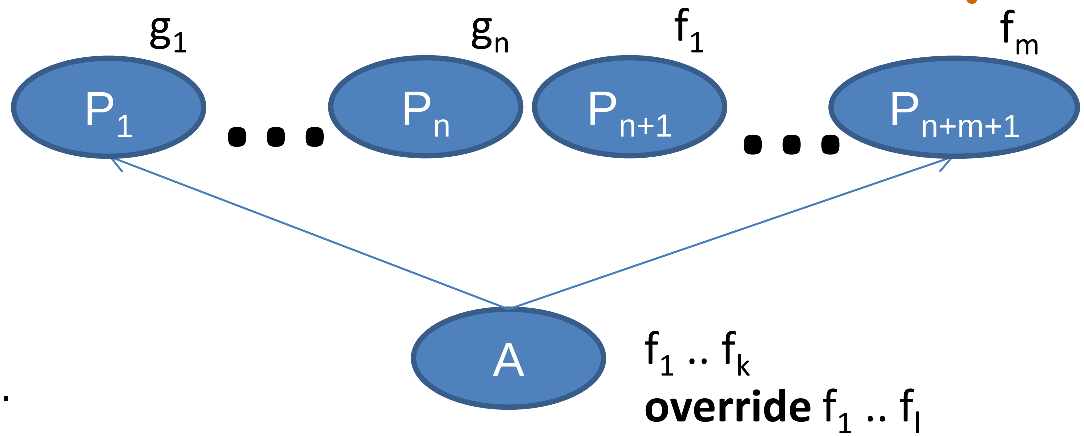
Operators – if & loop

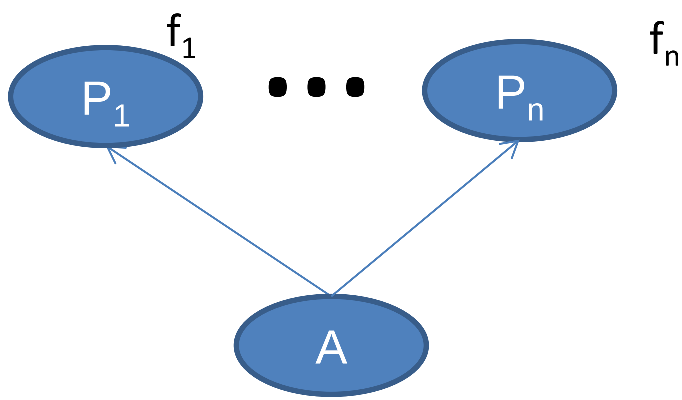
- One conditional statement and one loop
- 2 forms of conditional statements
- 3 forms of the loop

```
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
  - $l \le m$, 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

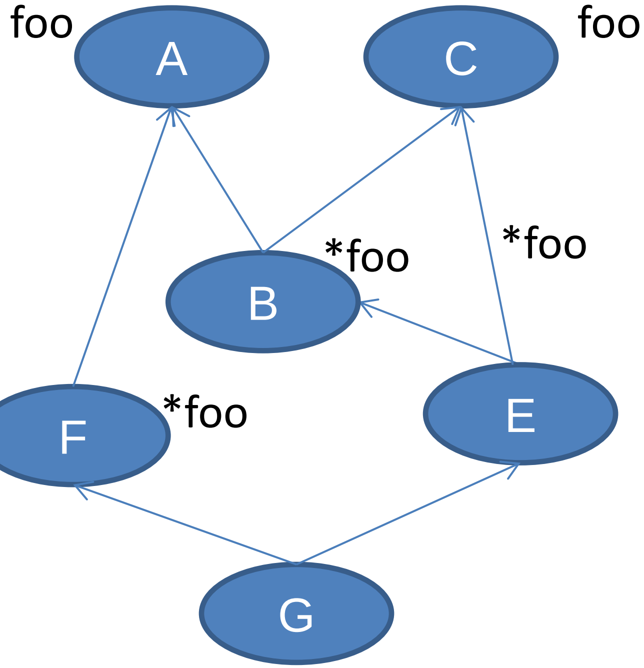





```
// P1..Pn – base units for A
// E1..En – expressions of types ETi
a is A
a.f(E1, .. En)
// Is it a valid feature call?
```

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**

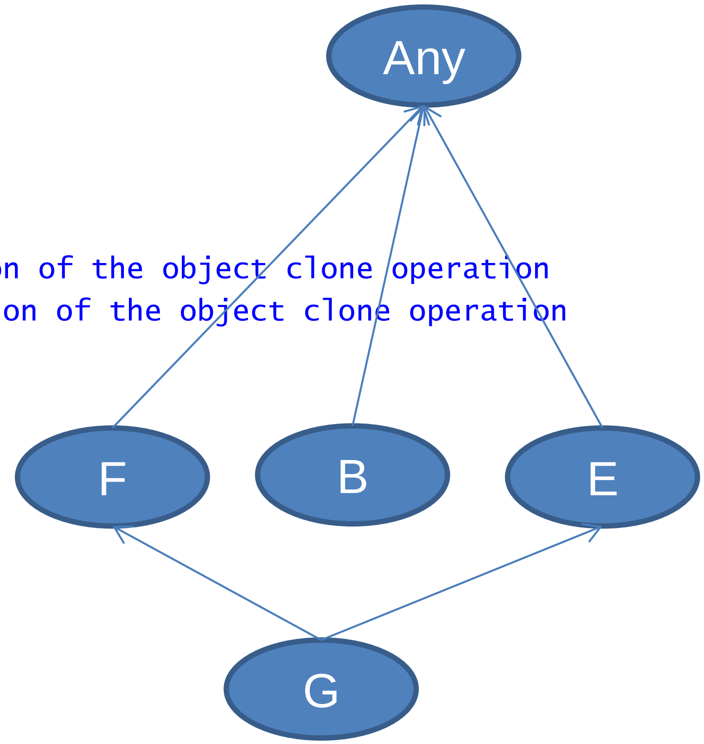
```
val unit Integer extend Integer
       [Platform.IntegerBitsCount] …
end
val unit Integer [BitsNumber: Integer] extend
Numeric, Enumeration is
   const minInteger is - (2 ^ (BitsNumber - 1))
   const maxInteger is 2 ^ (BitsNumber - 1) - 1
   const is /* That is ordered set defined as
range of all 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

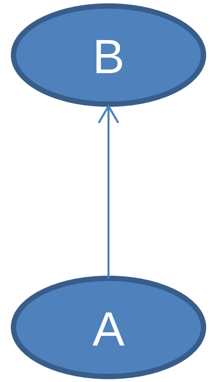
Status

- 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 $(T_1,\ T_2,\ \dots\ T_n)$

foo $(U_1,\ U_2,\ \dots\ U_n)$

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

U_i conforms to T_i

We *can* – therefore we *must*

? and typeof instead of NULL and type casts

- Value types case - entity: **? val** Type
- Consider rather expressive example:

var i: **?**Integer

i := i + 5 // Not valid!!! Compile time error

if i **is** Integer **then** i := i + 5 **end** /* That is a correct code */

if *i* ***is*** *Integer i := i +5 /* short form of if with one statement. It has no else part!!!*/*

? and typeof check instead of NULL and type casts

- Let's review in details how it works

c: **?**C

if c **is**

C1: /* if c is attached to an object which type conforms to C1 then one may work with c as it has static type C1*/

c.call_feature_from_C1

C: // the same for C

else /* Here we are – as there was a when clause with C type entity else clause means that c is actually detached. If there is no such clause then c can be either detached or attached to an object which type does not conform to all other when alternatives */

end

So, it allows to do both – run-time check for dynamic types and check for initialization.

? and typeof instead of NULL and type casts

- Let's see how typeof works

if c **is** C1 **then** /* if c is attached to an object which type conforms to C1 then one may work with c as it has static type C1*/

c.call_feature_from_C1

elseif c is C **then** // the same for C

else /* Here we are – as there was a when clause with C type entity else clause means that c is actually detached. If there is no such clause then c can be either detached or attached to an object which type does not conform to all other when alternatives

*/

end

while c **is** C1 **loop**

/*This loop works while type of c conforms to C1*/

end

‘?’ and ‘is’ instead of NULL and type casts

Power of if-case statement

if <expression> **is**

<expression1>**:**

<expression2> .. <expression3>**:**

Type1**:**

Type2|Type3|type4**:**

else

end

The statement above is equivalent to

if <expression> = <expression1> **then**

elseif <expression> **in** <expression2> .. <expression3> **then**

elseif <expression> **is** Type1 **then**

elseif <expression> **is** Type2|Type3|type4 **then**

else

end

2 kinds of unit attributes.

1. Potentially non-initialized entity (a**: ?**Type)
2. Entity which will (must) be initialized by every unit construction procedure (a**:** Type)

So, for latter kind attributes it is not possible to access features of such attributes inside constructors' bodies. In other words some object will be valid if and only if when its attributes will be initialized by one of its initialization procedures. This allows not to create artificial initialization procedures and gives additional flexibility for programmers.

2 kinds of unit attributes. Example.

```
a is Account (Customer())
StandardIO.put (a.customer.name) // OK
unit Account
      customer: Customer
      init (aCustomer: like customer) is
            StandardIO.put (customer.name) /*
            Compile time error*/
      end
      foo is
            StandardIO.print (customer.name) // OK!
      end
end
/*Objects of type Account are valid if and only is the
customer attribute was initialized.*/
```

Assertions (II)

```
unit Stack [G] // Interface of unit Stack
        push (e: G)
                ensure
                        count = old count + 1 // Push done
        pop: G
                require
                        count > 0 // stack not empty
                ensure
                        count = old count – 1 // pop done
        count: Integer
invariant
        count >= 0  // Consistent stack
end // Stack
```

Constant objects

One may ask why do we need constant objects while we have const attributes? Const attribute is part of the unit object while constant object is not. Let's consider example with modelling days of the week.

```
abstract unit Day
        isWorkDay: Boolean is abstract
        isWeekEndDay: Boolean is abstract
end // Day
unit WorkDays extend Day
        const is Monday, Tuesday, Wednesday, Thursday, Friday end
        override const isWorkDay is True
        override const isWeekEndDay is False
end
unit WeekEndDays extend Day
        const is Saturday, Sunday end
        override const isWorkDay is False
        override const isWeekEndDay is True
end
unit WeekDay extend Day
        const use WorkDays, WeekEndDays is end
        override isWorkDay: Boolean is
                this in Monday .. Friday
        end
        override isWeekEndDay: Boolean is
                this in Saturday .. Sunday
        end
end // WeekDay
```

Range types

```
unit WeekDay
        const is Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday end
        isWorkDay: Boolean is
                return this in Monday .. Friday
        end
        isWeekEndDay: Boolean is
                return this in Saturday .. Sunday
        end
end // WeekDay

use WeekDay
workDay: Monday .. Friday is Monday
weekEnd: Saturday | Sunday is Saturday
weekDay: WeekDay is Monday

workDay := weekDay // Error
weekDay  := workDay // OK
weekEnd := workDay  // Error
```

Tuples

- Tuple is a group of something ☺ (Integer, Real, Boolean) – tuple of types. Tuple type is a kind of anonymous unit.
- (a: Integer; b: Boolean) – tuple with named fields
- (5, 6, 7) – tuple of Integer values. Tuple expression. It conforms to Array [Integer] as all types are identical. So, we initialize arrays with tuple expressions!
- a: (Integer, Real) – type of a is a tuple with 2 unnamed fields of types Integer and Real.
- x: (Integer, 5, Real, flag: Boolean) – That is a tuple as well
- Conformance for tuples: tuple T1 -> tuple T2 if for every i = 1..n T1i -> T2i when n = T1.count and n <= T2.count. Note that is the basis for functions with “growing” number of parameters
- Then every routine has only 1 parameter – tuple, possibly empty. And it returns a tuple with 0 or more elements. Procedure is a function which returns empty tuple ☺ So, we can just ignore what it returns like void in old plain C ☺

Tuples – WIP!

If we have foo declared as foo (args: ()) then we can call foo like

foo (e1, e2, e3) /* that is call to foo with the tuple (e1, e2, e3), where e1, e2, e3 – 3 expressions and T1, T2, T3 are types of these expressions*/

//So, we can assign a tuple to a variable and then

t: (T1, T2, T3) **is** (e1, e2, e3)

foo (t)

t1 **is** (e1, e2, e3, e4) // Type of t1 is deduced from types of e1, e2, e3,e4

foo (t1) // Valid as well !

foo (e1) // Calls foo with 1 argument

foo (e1, e2) // Calls foo with 2 arguments

foo (e1, e2, e3) // Calls foo with 3 arguments

foo (arg1: T1; arg2: T2; arg3: T3)

foo (arg1: T1; arg2: T2)

foo (arg1: T1)

So, a: () is (1, True, “String”) is a valid variable of type empty tuple declaration with initial value the tuple with 3 elements.

Tuples

/*So, Tuple may be typed – (Integer, Real, Boolean)*/

t2 **is** (Integer, Real, Boolean) /* That is in fact call to Tuple constructor and it will work only when Integer, Real and Boolean have init with no arguments!!! */

t2(1) := 5; t2 (2) := 5.5; t2 (3) := True

/*So, tuple may have named fields*/

t3 **is** (i: Integer; r: Real; b: Boolean)

t3.i := 5; t3.r := 5.5; t3.b := False

t4: (Integer, Real, Boolean) **is** (5, 5.5, True)

t5 **is** (5, 5.5, True)

//Note!

goo (x: Integer) **is** StandardIO.print (“goo 1\n”) **end**

goo (x: (Integer)) **is** StandardIO.print (“goo 2\n”) **end**

/* These are 2 different routines!*/

goo (5) // output -> goo 1

t6: (Integer) **is** (5)

goo (t6) // output -> goo 2

goo ((5)) // output -> goo 1 as we treat (<expr>) as expression!!!

goo (5,6,7,8) // output -> goo 2

Tuples

```
unit () // That is a pseudo unit. It just describes what features every tuple has
		count: Integer /* the number of elements in the Tuple*/
		type (position: Integer): RTTypeDescriptor // That is retrospection API
				require position in 1 .. count /// Valid position
		override assign | := (other: like this) is init (other) end
		value | () (position: Integer): Any
				require position in 1 .. count /// Valid position
		setValue | () (position: Integer, aValue: Any)
				 require position in 1 .. count /// Valid position
		type (fieldName: String): RTTypeDescriptor
				require hasFiled (fieldName) /// Valid field name
		value | . (name: String): Any
				 require hasFiled (fieldName) /// Valid field name
		setValue | () (name: String, aValue: Any)
				 require
						position in 1 .. count /// Valid position
						hasFiled (fieldName) /// Valid field name
		hasFiled (name: String): Boolean
		init (other: like this) is
				count := other.count
				while pos in 1 .. other.count loop setValue (pos, other.value (pos)) end
		end
invariant
		 count >= 0 /// Consistent tuple
end
```

Tuples - assertions

If we have a tuple – what is the invariant of such tuple? The answer is straightforward - default invariant is True. And that is why feature setValue will always work. But if one needs to specify tuple invariant to protect its integrity we may consider to allow adding invariant to tuples. See example below

use StandardIO

t **is** (f1: Integer; f2: Real; f3: Boolean **invariant** f1 >= f2 **implies** f3)

print (“t.f1 = “, t.f1, “, t.f2 = ”, t.f2, “, t.f3 = ”, t.f3, ‘\n’)

/* Output will be 0 0.0 False ☺ as init with no arguments for Integer, Real and Boolean do exactly this*/

t.f1 := 5 /* Will trigger invariant violation as 5 0.0 False dies not match the invariant */

t := (5, 1.0, True) // OK. Invariant preserved!

print (“t.f1 = “, t.f1, “, t.f2 = ”, t.f2, “, t.f3 = ”, t.f3, ‘\n’)

// Output will be 5 1.0 True ☺

t(2) := 4.99 // OK. Invariant preserved!

Tuples: Arrays

```
a:  Array [Integer] is (1,2,3,4); a(1) := 6; i1 is a(4)
unit Array [G->Any init ()]  ///   WIP!, dimensions: (Discrete?????)]
/* We can put info Array only objects which has constructor with empty signature !!! We are always safe!!!*/
        item  | () (pos: Integer) require lower <= pos and then pos <= upper is
                getArrayItem (data, pos)
        end
        setItem | () (pos: Integer; value: G) require lower <= pos and then pos <= upper  is
                setArrayItem (data, pos, value)
        end
        count: Integer is  upper – lower + 1 end
        lower: Integer
        upper: Integer
        init (n: Integer; value: G) is lower := 1; upper := n; fill (value); end
        init (n: Integer) is init (n, G()) end
        init (l, u: Integer) is lower := l; upper := u; fill (G()); end
private:
        fill (value: G) is
                data := allocateArray (lower, upper, sizeof (G))
                while i in lower .. upper loop  setItem (i, value) end
        end
        data: Pointer
        getArrayItem (d: Pointer, …) is external end
invariant
        count >= 0 /// Consistent array count – greater than zero
        lower <= upper /// Consistent  array range – lower is not greater than upper
end
```

Tuples: Variable number of arguments

```
// Let's consider the following routine
foo (arguments : ()) is
        while argument in arguments loop
                // Type of argument is deduced as Any!!!
                if argument is
                Integer: // Do something with argument of type Integer
                Real : // Do something with argument of type Real
                Boolean : // Do something with argument of type Boolean
                Character :  // Do something with argument of type Character
                String : // Do something with argument of type String
                else // Do something with argument of type Any
                end
        end
end
// It can be called in many different ways
foo (1, 2, 3)
foo ("String", True, Boolean, Integer)
foo (T1, T2, T3, T4)
// Another caveat
goo (arg1: T1; arg2: T2)
goo (E1, E2, E3, E4) /* Should expressions E3 and E4 be evaluated ? My guess is NO as they are
goo does not have arguments of type tuple!!!*/
```

Routine types

foo **is end** // That is a procedure without arguments

f **is routine** foo // That is lambda based on foo

f /* that is a call to a procedure which is associated with f. So, one may guess that f can be passed to other routines, stored and called later when necessary*/

goo (i: Integer; b: Boolean; t: Type) **is end**

g: **routine** (Integer, Boolean, Type) **is routine** goo

/*Type inference allows just to write */

g1 **is routine** goo

g1 (5.5, “String”, f1) /* Compile time error!!! So, we have type safe lambdas!!! */

l1 : **routine** (T1; T2; T3) /* That is non-attached lambda - **?** In front of lambda is assumed here*/

l2 : **routine** (arg1: T1; arg2: T2; arg3: T) **is** arg1.foo **end** /* That is inline lambda */

l1 := l2 // Type of T2 conforms to type of l1

l1 := f // Type of f does not conform to type of l1 – compile time error

Routine types

Let's see the example

foo: Type **is end** /* foo is a function which returns objects of type Type*/

f **is routine** foo /* f is a object of functional type. Its derived type is **routine** : Type*/

a **is routine** f /* a is the same object of functiuonal type*/

t1 **is** f /* t will be declared of type Type and initialized with the results of the call to f */

t2 **is** foo // The same semantics as t1

So, if one likes to define an object of functional type use of keyword **routine** is mandatory!

Routine types - example

```
g is routine (a, b, c: Real): (x1:Real, x2:Real)
        require a /= 0 /// First parameter can not be zero
        is // That is inline lambda
                d is b*b – 4*a*c
                return  if d >= 0 then ((-b + Math.sqrt (d))/2/a, (-b - Math.sqrt (d))/2/a)
                else () // Empty tuple ☺
        end
a is StandardIO.readReal
if a = 0 then  StandardIO.put (“That is not a square equation!!!\n”)  else
        b is StandardIO.readReal
        c is StandardIO.readReal
        x is g (a, b, c)
        if x.count = 2 then
                StandardIO.put (“X1= ”, x [1], “\n”, “X2 = ”, x[2], “\n”)
        else
                StandardIO.put (“Equation with coefficients a= ”, a, “, b = ”, b, “, c = ”,
                c, “ has no valid square equation roots”)
        end
end
```

Predefined and core units

/*There are only 1 predefined unit - Bit. So, all other units can be constructed from Bit

Can we call Platform a predefined module – not sure ☺ It is just an essential part of the Kernel library (libc ☺)*/

unit Platform //In fact we define ILP here …

        const integerBits **is** integerBytes * bitsInByte /* Type deduction works here – no need to mention Integer */

        const integerBytes **is** 4

        const realBits **is** realBytes * bitsInByte

        const realBytes **is** 8

        const bitsInByte **is** 8

end

Predefined and core units

```
val unit Bit [N: Integer]
	() (index: Integer; value: Boolean)
		require index in 0..N /// Valid index
	alias () (index: Integer): Boolean
		require index in 0..N /// Valid index
	^ (distance: Integer): like this
	and (other: like this): like this
	or (other: like this): like this
	xor (other: like this): like this
	=> (other: like this): like this
Invariant this and this = this
end // Bit
```

Statements and expressions: if & case expressions

a := **if** <Boolean_expr>

then <then_expr>

else <else_expr>

c := **if** <expression> **is**

<case_expr1>**:** <expression1>

<case_expr2>**:** <expression2>

…

else <else_expr>

- Отсюда есть темы, претендующие на новизну.
  1. Множественное наследование при наличии overloading and overriding
  2. Multi-types (type-safe duck typing)
      a: T1|T2
  3. New variant of Null-safety in fact Null-absense.
  4. Анонимный код - последовательность операторов. StandardIO.putString ("Hello world!\n") - законченная программа.
  5. ref and val types of objects of all types.
  6. Units - 3 in 1 concept – modules, classes and types together.

Lambda (routines as 1st class citizens) WIP!!

foo **is end** // That is a procedure without arguments

f: Routine [(), ()] = **routine** foo

f.call /* that is a call to foo which is associated with f. So, one may guess that f can be passed to other routines, stored and called */

goo (i: Integer; b: Boolean; t: Type) **is end**

g: Routine [(Integer, Boolean, Type), ()] = **routine** goo

/*Type inference allows just to write */

f1: Routine **is routine** foo

g1: Routine **is routine** goo

f1(5, 6, True) // Is a valid call!!!

g1 (5.5, “String”, f1) /* Compile time error!!! So, we have type safe lambdas!!! */

/* Note that just routine name is ambiguous due to overloading one need to specify the signature to remove ambiguity*/

Routine types

abstract unit Routine [Arguments->(), Result]

arguments: like Arguments

abstract apply (args: Arguments)

// That is a procedure call

abstract apply (args: Arguments): Result

// That is a function call

end

unit Procedure [Arguments -> ()] **extend** Routine [Arguments, ()]

apply (args: Arguments)

// That is a procedure call

hidden apply (args: Arguments): Result

// That is a function call

end

unit Function [Arguments -> (), Result] **extend** Routine [Arguments, Result]

hidden apply (args: Arguments)

// That is a procedure call

apply (args: Arguments): Result

// That is a function call

end

Мультитипы

Постановка задачи: есть две сущности разных (не конформных друг другу) типов, с общими свойствами (features). Как написать общий код для работы с этими свойствами, не вводя общего родителя (базового класса)?

Для решения этой задачи и предлагается понятие **мультитипа**.

Введение в язык этого понятия вместе с соответствующим механизмом контроля может заменить правила структурной эквивалентности типов без нарушения принципов статической типизации.

Мультитипы (Пример)

- number: Integer | Real | Double /*Таким образом, атрибуту number можно присваивать сущности типов Integer, Real, Double или их наследников. Соответственно, можно обращаться к тем свойствам мультитипа, которые присутствуют во всех трех типах.*/
- number := number1 + number2
- /*Т.е. свойство сложения, которое обозначается инфиксной операцией +, должно присутствовать в типах Integer, Real и Double. Кроме того, вызов вида number.+(number) должен быть правильным для всех видов сочетаний Integer.+(Integer), Real.+(Real) и Double.+(Double).*/

Неинициализированные переменные и нулевые указатели

Нулевые (пустые) указатели (ссылки) - часть более общей проблемы - ошибки при попытке работе с *неинициализированными атрибутами*.

3 базовых принципа:

- Каждый атрибут должен получить значение до первого обращения к его свойствам.
- Если нужно описать атрибут без значения, то нельзя обращаться к его свойствам.
- Должен быть механизм безопасного перехода от неинициализированного атрибута к инициализированному.

Неинициализированные переменные и нулевые указатели (Пример)

attr1 **is** 5 // Явная инициализация и неявная типизация

attr2: Integer // Явная типизация и неявная инициализация

attr3: **?**Integer /* Явная типизация и *отсутствие значения*.*/

attr1 := attr2; attr2 := attr1; attr3 := attr2 // Все валидно!

attr1 := attr3; attr2 := attr3 // Ошибка компиляции

if attr3 **typeof** Integer **then** // Внутри - тип attr3 Integer!

    attr3 := attr3 + 5

    attr1 := attr3

end

?attr3 // Потеря значения и смена типа!

Multi-types

- Fur

```
number: Integer | Real |
    myType
```

Атрибуту `number` можно присваивать сущности типов `Integer`, `Real`, `myType` или их наследников. Соответственно, можно обращаться к тем свойствам мультитипа, которые *присутствуют* во всех трех типах.

```
number := number1 + number2
```

Свойство сложения, которое обозначается инфиксной операцией `+`, должно присутствовать во всех типах, образующих мультитип. Кроме того, вызов вида `number.+(number)` должен быть правильным для всех видов сочетаний `Integer.+(Integer)`, `Real.+(Real)` и `myType.+(myType)`.

- ther generalization of inheritance

```
// Example ///
a1, a2: Integer | Real | String is 0

a1 := a1 + a2

a1 := “string”

a1 := a1 + a2
```

Проблема:

Пусть имеются две или более сущности разных (не конформных друг другу) типов, с общими св

Решение:

Понятие **мультитипа**

Введение в язык этого понятия вместе с соответствующим механизмом контроля может заменить правила структурной эквивалентности типов без нарушения принципов статической типизации.