**Alexey Kanatov**  
Samsung Research and Development Institute, Moscow, Russia  
a.kanatov@samsung.com

**Eugene Zouev**  
Innopolis University, Kazan, Russia  
eugene.zueff@gmail.com

**Towards a better design of a further programming language**

**Abstract**. The paper describes investigations on what can be made to make programming practice more convenient and more efficient keeping code safety and reuse at high levels. Some major concepts and improvements which can be added into existing or new programming language are presented. Among them are anonymous code sequences, multiple inheritance mixed with overloading and overriding, multi-types for increasing reuse, null/void absence and modules/classes program architecture. These new/updated concepts work well with well-proven structured, object-oriented, generic and functional programming paradigms.

**Keywords**: programming language, language semantics, type inference, unified type system, mutli-types.

**1. Introduction**

Evolution implies constant search for new ways of solving different challenges which software engineering practice faces since its creation in the middle of the 20th century. Question how to find a balance between program code reliability, performance, readability and convenience for average programmers probably cannot be answered in a single way and this leads to different attempts to propose new languages, new extensions in the existing languages depending on particular needs. So, that looks like the way how evolution happens. So, authors were involved in different compiler related projects during last 30 years and could not avoid the temptation to share some ideas and suggestions to approach the desired balance based on what was done and potentially could be done. So, this article focuses on few aspects which may bring value for the software engineering practice. There is a great interest in the world to script languages which are characterized with high-level of dynamic nature which has its pluses and minuses and there is a plenty of code developed in languages designed 20 and more years ago, managed languages made a big step ahead in terms of increasing the reliability of the software development process and we observe evolution of these languages as well. So, several proposals which will be described below aim to keep the ability to produce effective (optimized) machine code as well as ability to generate manageable code in parallel with static typing and dynamic checks which can minimize the risk of software failures.

*[Евгений – я не уверен, что нам надо “ругать” существующие языки. Фразы про постоянно возрастающие требования я помню еще со времен Ады-83…. И может пока мы будем в статье говорить о некоторых предложениях о новых/улучшенных конструкциях и подходах, а не о целостной концепции языка …*

*Современные потребности: возрастает объем и сложность программных систем, повышаются требования к быстродействию, надежности, устойчивости и т.д. Современные языки реализации – как системного, так и прикладного уровня – явно не соответствуют возросшим требованиям. Слова о недостатках существующих языков (пара абзацев).*

*Отсюда – желание спроектировать и реализовать язык, который бы отвечал изменившимся потребностям.]*

***[2 Requirements and features of a new language]***

**2. Basic assumptions and key concepts of proposed programming language constructions**

Today’s reality is that majority of software developers are used to object-oriented programming with genericity of different kinds and the tendency adding elements of functional programming into imperative programming languages is the trend. In this article we will focus on updates and enhancements which are based on this trend. Namely we will present the concept of multi-structured compilation unit, multi-type entities, declaration with immediate and deferred initialization of entities, multiple inheritance with overloading and overriding, collections as compile time concept. All these concepts we focus on work well with imperative, object-oriented, generic and functional programming, strict separation of the program from execution environment, static typing with type inference, null-safety, access control, structured assertions (Design by Contract (C)) and tuples. As majority of these concepts are already presented in some programming languages they will be described briefly and only when necessary to illustrate the key concepts of this paper.

*[SLang's trinity: simplicity, reliability, concurrency. <Коротко объяснить, что под этим понимается: один абзац>.*

*Более подробно:*

* *Multi-paradigm support: imperative, object-oriented, generic and function programming.*
* *Separation program from execution environment; the notion of a configuration file.*
* *Strong typing with type inference support; the notion of immutable objects.*
* *Null-safety and full access control.*
* *Compile-time type and constant constraints for generic units.*
* *Pre- and postconditions for routines and invariants for classes.*
* *Lightweight and easy-to-use concurrency support.*
* *Flat type system with uniform access: all program types are (either system-defined or user-defined) classes.]*

***[3 Common language structure]***

**3. Multi-structured compilation unit**

Development of software engineering reached the stage when a lot of software was already implemented though that does not imply that there will be no new software development. New software and modification of existing one is being done in the context of what was already implemented focusing on particular pieces of today’s interest. So, we may distinguish 2 tasks. First - quickly solve particular problem based on what was already done. And second – develop new framework or library for further reuse or sophisticated software complex. So, the idea that the same programming language should combine ability for fast development of code snippets and big software complexes can be solved providing support for different types of compilation units. Simplest compilation unit is a sequence of statements and more structured and complicated compilation unit is a sequence of routines, modules and classes. These two approaches in fact do not contradict each other as the first one is just an optional starting part of any compilation source. As an example the classical Hello world example can be expressed in just one line.

StandardIO.write("Hello world!\n")

where StandardIO is the name of the module which has set of features to deal with input/output. Its construction procedure will establish default connection to console or file before the main program starts or at first usage. The key thing is that semantics of the program is rather straightforward and follows the C-style notion here. If one needs to develop a bit more functional program the following can be added

var ch: Char = StandardIO.readChar()

So, it allows adding more and more functionality just adding more and more statements. And at some point one will encounter the need of some common code which will be used – routines. Again while the program is simple enough the same source can be used to store the source code of routines. See example below

use StandardIO

write("Hello world!\n")

var ch: Char = readChar()

myRotuine ()

myRoutine is

// Some statements …

end myRoutine

It contains use statement with the semantics well-known from Ada and Modula-2 – no more need for prefixing write and readChar in this source with StandardIO. And call to routine myRoutine is the last statement of the anonymous routine which starts with write and ends with myRoutine call. In other words if compiler encounters routine, class or module it is the end of sequence of statements. So, if the source starts with routine, class or module it has no default entry point.

Just few comments on syntax used. Adding the name of the routine, class, module or statement name after end is optional but allows to understand the exact meaning of the end. Style of comments is taken from C/C++/Java. Call to routine with no arguments may skip empty parenthesis. var and const keywords are well-known from Pascal, Module-2 languages and have the similar meaning denoting the type of the entity being declared.

*[The language supports two basic kinds of usage: it can be used either as a usual programming language. As such it supports the full set of features, see below. The second option is to use it to create short code snippets (scripts0 for various purposes. For example, the following code is the complete valid program; its semantics seems to be obvious without any extra explanations:*

*StandardIO.write("Hello world!\n")*

*For more complicated purposes, the language defines a full set of program building blocks: modules, classes, routines (procedures and functions). Modules are system-managed units aggregating related program resources of any kind.* Classes are … Routines are … Краткое объяснение существа каждой единицы.]

Now it is the time to spend some time talking about basic terms like module, class, attribute, routine, and entity. Let’s start with routines – named set of statements, block of code which has some well-defined semantics. Depending of the presence of return value procedures and functions are defined. So, routine is a function if it has as return value. If there is no return value or return value is empty tuple it is called procedure. The key difference is that functions are called from expressions and we must handle the returning value, while procedures are called as a statement and there is no need to think about return value. Hello world example had a function call to readChar and procedure call to write – that is the illustration of this difference. Routines can be anonymous (just set of statements, there is no way for programmer to all such routine by name), part of classes or modules (for classes they are called methods as well) and routines which are at the same level as classes and modules – standalone routines (like in Ada and C++). Attributes are data. And they are part of module or class. Attributes can be constant or variable. The only difference between them is that constant attribute gets its value at declaration and keeps it till the end of the execution while variable attribute may get new value. In other words constant attribute can be the target of the assignment statement while variable one can.

var variableAttribute: Type = <expression>

const constantAttribute: Type = <expression>

constantAttribute := <expression> // Compile time error!!!

variableAttribute := <expression> /\* Can be a valid assignment if types are OK\*/

There is a variety of relationships between program units. Three main kinds of relationships are allowed: composition, using and inheritance. <Краткое объяснение существа всех видов отношений>. For example, modules are **used** by other units, classes may **inherit** from other classes composing conventional object-oriented class hierarchies (with multiple inheritance), functions and classes can be **parts** of modules, and can be **nested**, routines are **methods** of classes (with possible overriding).

All program units (except modules) can be defined as type-parameterized and/or constant-parameterized, see Section X.

Below is the common scheme of how program is composed, gets compiled and linked/

<Моя картинка из слайдов.>

<Краткое объяснение.>

**3 Variables, types, operations and statements**

**3.4 Static typing and type inference**

The language supports static typification that means that every entity has a type associated with it explicitly or implicitly. Type strictly defines which features can be invoked based on this entity. This is the basis for program correctness check at compile time. Type is specified explicitly when entity is declared or can be derived by the compiler based on first initialization of an entity.

Examples:

**var** entity: Type // Type of entity is explicitly defined as Type  
**const** constant = 5 // Type of constant is type of 5, eg., Integer

**4 Non-conventional data structures: Tuples and multi-types**

**4.1 Tuples**

Tuple is a group of something ☺

(Integer, Real, Boolean) – tuple of types

(5, 6, 7) – tuple of Integer values – it conforms to Array [Integer]. So, we initialize Arrays with Tuples!

(a: Integer; b: Boolean) – tuple with named fields

Conformance for tuples: tuple T1 conforms to 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 ☺

foo (e1, e2, e3) /\* that is call to foo with the tuple (e1, e2, e3)\*/

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

t: Tuple = (e1, e2, e3) /\* type deduction may deduce different types instead of Tuple\*/

foo (t) // ☺ That is feature call with a tuple. ((t)) identical to (t) ☺

t1: Tuple = (e1, e2, e3, e4)

foo (t1) // Valid as well !

/\*So, Tuple may be typed – Tuple [Integer, Real, Boolean]\*/

t2: Tuple [Integer, Real, Boolean]

t2[1] := 5; t2 [2] := 5.5; t2 [3] := True

/\*So, Tuple may have named fields\*/

t3: Tuple [i: Integer; r: Real; b: Boolean]

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

class Tuple [T->Tuple] // That is a pseudo class

count: Integer /\* the number of elements in the Tuple\*/

type (position: Integer): ClassDescriptor // That is retrospection API

require “Valid position: position <= count

value alias [](position: Integer): Any

require “Valid position”: position <= count

setValue alias [] (position: Integer, aValue: Any)

require “Valid position”: position <= count

type (fieldName: String): ClassDescriptor

require “Valid field name”: hasFiled (fieldName)

value alias . (name: String): Any

require “Valid field name”: hasFiled (fieldName)

setValue alias [] (name: String, aValue: Any)

require “Valid position”: position <= count

require “Valid field name”: hasFiled (fieldName)

hasFiled (name: String): Boolean

invariant

“Consistent tuple”: count>= 0

end

a: Array [Integer] = (1,2,3,4); a[1] := 6; i1: Integer = a[4]

///class Array [G->Any ctor ()] // We can put info Array only objects which has constructor with empty signature !!! We are Null safe!!!

class Array [G]

item alias [] (pos: Integer) is

require lower <= pos and then pos <= upper

setItem alias [] (pos: Integer; value: G)

require lower <= pos and then pos <= upper

count: Integer is

count := upper – lower + 1

end

lower: Integer

upper: Integer

ctor (n: Integer; value: G) is

lower := 1; upper := n; init (value);

end

ctor (n: Integer) is

ctor (n, G())

end

ctor (l, u: Integer) is

lower := l; upper := u; init (G());

end

private:

init (value: G) is

while i in lower..upper loop // This loop is to be parallel loop!

setItem (i, value)

end

end

data: Pointer

invariant

“Consistent count”: count >= 0

“Consistent range”: lower <= upper

end

**4.2 Multi-types**

Can an entity have several types? Yes. If entity is declared specifying several types then such a declaration defines multi-type entity and access to any feature is processed in a bit more sophisticated way rather than access to single type entity feature. Let’s consider example

**var** e: T1 | T2 | T3  
e.foo(E1, E2, ...)

where T1, T2, T3 are types, and E1, E2, … are expressions. So, declaration of e is valid when T1 does not conform to T2 and T3 and this is true for any other pair. The call e.foo is valid when feature foo belong to T1, T2 and T3 and types of E1, E2, … conform to corresponding types of arguments of features foo in T1, T2 and T3 respectively.

What does it give to a programmer: yet another aspect of reuse. When code of T1, T2 and T3 is available in compiled form only, when the inheritance clause of T1, T2 and T3 cannot be changed one can still develop the universal code which will work with object of types T1, T2, T3 and their descendants. (“Duck typing”? ☺)

**5 Null safety: typeof predicate instead of NULL and type casts**

**5.1 Null safety problem**

Краткое описание проблемыж почему современные языки ее не решают и каким средствами они для этого обходятся. Может быть, один разительный (короткий) пример.

**5.2 The solution taken: undefined values and typeof predicate**

As every entity is to be initialized at the time of declaration we do not have non-initialized entities. That is great but we cannot guarantee the proper initialization for collection variables when declare them, or we force programmers to invent artificial objects or some Boolean flags. So, in fact we have two states – entity is initialized (in other words attached) and non-initialized (detached). Does it imply the need of NULL, NIL, Void or any other special “value” that reflects that entity is detached – probably no. What we need is the language mechanism to declare entities which initialization is deferred to some point at execution, check if an entity was initialized or not, safely transform initialized entity which was declared as detachable into another one declared as attached one and detach an entity. So, we introduce the following language constructs:

entity: ?T /\* after this declaration entity is considered detached or non-initialized. It implies that we cannot do any operation based on this entity\*/

?entity // explicitly detach this entity. It becomes un itialized

**if** entity **typeof** T **then** entity.foo **end** // safe call

Let’s consider rather expressive example:

**var** i: ? Integer  
i := i + 5 // Not valid; compile time error  
**if** i **typeof** Integer **then** i := i + 5 **end** // Safe code, or...

**if** i **typeof** Integer i := i +5  
 // ...A shorter form of if with only one statement.  
 // It has no else part!

<Здесь нужно подробное объяснение семантики операции typeof. Кстати, я бы отказался от слов attached и detachable – уж больно Майером пахнет ☺>.

Let’s see how **typeof** works.

**if** c **typeof** 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  
**elsif c typeof** 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**

The very similar considerations are applicable also to loops and multiple-choice statements. For example:

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

**6 Value and reference classes and objects**

**6.1 Value and reference classes**

**6.2 Creation of objects**

So, we may determine the type of the run-time entity while declaring it. Let’s start with an example to present the concept:

o1: **ref** Type // o1 is the reference to an object of type Type

o2: **val** Type // o2 is the object itself - value

o3: **separate** Type // o3 is a proxy to an object which is to be  
 // processed by another processing element: CPU,  
 // core, thread, web server – whatever.

If Type was declared like class Type or ref class type – default is to create reference object. If Type was declared like val class Type – default is to create value objects. If Type was declared like separate class Type – meaning is straightforward. However, it’s possible to change the default construction behavior with explicit notice what kind of object is to be created like in examples above.

Implication of the way we create objects is the way we assign them

ref1 := val1 // clone val1 and reference is stored in ref1

ref1 := ref2 // ref1 will point to the same object as ref2

val1 := val2 // field-by-field copy only fields which we have in val1  
 // will be copied from val2

val := ref // field-by-field copy only fields which we have in  
 // objected referred by ref will be copied from val2

separate1 := separate2 // like ref1 := ref2

ref |= val // We would like to have a reference attached to my val  
 // object. This is under debates. however

Implicit outcome – all arguments are passed by value!

**8 Generic units**

Any program unit (except modules) and any collection can be defined generic and any standalone routine can be generic. Two kinds of generic parameters are supported: type as a parameter, and constant expression of an enumerated type as a parameter:

**class** Vector[T, length: Integer] ... **end**

// Any type can be an argument **class** Array[G] ... **end**

For type as a parameter there are also two options – constrained genericity and non-constrained one:

// Constraint: argument can be only type derived from Comparable **class** SortredArray [G->Comparable] ... **end**

Generic implementation details is not a part of the language definition – we should not assume that every new instantiation implies new portion of code.

Another caveat here is how to create object of generic type inside collection. So, the mechanism to pass the info on what constructor must be used is to be provided. For example:

**class** SomeClass [G->Constraint **ctor**(*signature*)]

**class** SomeClass [G **ctor**(Integer, Boolean)]

**class** AnotherClass [G **ctor**]

**class** AnotherClass [G1, G **ctor**(G1)]

**9 Assertions**

Абзац-другой о том, что это вообще такое, зачем нужно и как другие языки без этих радостей обходятся (и как они несчастны из-за этого, сами о том не подозревая...).

**9.1 The approach taken**

The approach suggested for the language is to support 3 types of assertions: preconditions, postconditions and collection invariants. Loops are out of scope for now as practice shows that programmers nearly ignore loop-related assertions. We may consider support assertion check with in the body of any routine, then we may use assert or check keyword.

Assertion is a optionally labeled with the identifier valid Boolean expression. Precondition: set of assertions which must be all true at routine entrance started with keyword require. Postcondition: set of assertions which must be all true at routine exit started with keyword ensure. Collection invariant: set of assertions which must be all true to any collection routine entrance and exit. invariant is the keyword. Example:

**class** Stack [G] // Interface of class Stack  
 push (e: G) **is**  
 **ensure** Push\_done: count = **old** count + 1  
 ...  
 **end** push

pop: G is  
 **require** Stack\_not\_empty: count > 0  
 **ensure** Pop\_done: count = **old** count – 1  
 ...  
 **end** pop

count: Integer

**invariant** Stack\_is\_consistent: count >= 0  
**end** Stack

**9.2 Assertions and inheritance**

When we override some routine its precondition will be OR-ed with preconditions of all overridden versions. While for the postcondition AND is applied. For the purpose of clarity it is better to keep “require else” and “ensure then” operators while overriding:

**class** MyStack[G] : Stack[G]  
 **override** push (e: G) **is  
 ensure then** some\_boolean\_expresssion\_1

**invariant** some\_boolean\_expresssion\_2

**end** MyStack

**9.3 Assertions: runtime behavior**

When assertion is checked at run-time we have two cases. The first one is that the assertion is evaluated to true and execution continue its normal way. The second option is that assertion is evaluated to false. In this case the normal execution of the program does not make sense as if precondition was violated it implies that routine cannot do its job and the problem is on the caller’s side. If postcondition is violated then the problem is in the callee’s body. If collection invariant is violated it implies that collection is no longer in a consistent state. In all these cases we need to signal immediately – that is an exception case. So, 3 types of assertions imply that 3 different exceptions are to be triggered. In all cases exception is sent to the caller and depending on absence or presence of the exception handling execution is stopped or continued. When execution is stopped the identifier which was used to label particular assertion will play the role of the helper string which will help to understand the reason of the exception and will guide the programmer how to fix the issue.

Examples:

stack: Stack[String]  
Stack.push("A string") // Postcondition violation: pushDone WHY???

// Here we create a stack and try to pop immediately after creation  
stack: Stack[Integer]; i: Integer  
i := stack.pop // Precondition violation: stackNotEmpty

We may provide flexible control over assertions – first approach is to select at compile time what kind of assertions we like to have enabled at run-time at routine, collection, group of collections or library/program wide level. The simplest scheme is just to tell to the compiler assertions/preconditions on or off. Also we may control assertions at run-time using run-time configuration file – then it allows to change the way we check assertions without recompilation. All these mechanisms are not part of the language and may be adjusted by particular implementation. Of course smart implementation will allow to speculatively execute the routine body in parallel checking its preconditions and class invariant to minimize the performance impact of assertions checking at runtime. The similar technique can be applied for postconditions leaving the routine. But again that is part of smart compiler implementation.

**10 Language implementation**

The compiler is being written in the language itself using bootstrapping technology. The LLVM infrastructure is used as compiler’s back end. The primary hardware platform is ARM.

**References**

Майер. Язык Эйфель (или стандарт ECMA)  
Стандарт Ады  
Стандарт С++  
Книга по Модула-2  
Статья Шиперского о модулях и классах  
Ссылка на LLVM.