**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.

**Content**

[**1 Introduction**](#Inheritance)

[**2 Basic assumptions and key concepts of proposed programming language constructs**](#Basic_assumptions)

[**3 Multi-structured compilation units**](#Multi_structured_units)

[**4 Inheritance and its usage**](#Inheritance)

[**5 Generic units**](#Generic_units)

[**6 Assertions**](#Assertions)

[**7 Value and reference classes and entities**](#Val_and_ref)

[**8 Non-conventional data structures: Tuples and multi-types**](#tuples_multi_types)

[**9 Null safety: typeof predicate instead of NULL and type casts**](#type_of)

[**10 Language implementation**](#Implementation)

**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 to existing languages depending on particular needs. So, that looks like the way how evolution happens. 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. Therefore, the 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 characterized with high-level 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 efficient (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.

**2 Basic assumptions and key concepts of proposed programming language constructs**

Today’s reality is that majority of software developers are used to object-oriented programming with genericity of different kinds; also, 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 these aspects. Namely, we will present the concept of multi-structured compilation unit, multi-type entities, declarations with immediate and deferred entity initializations, multiple inheritance with overloading and overriding, and units as compile time concept. All the 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 © [Meyer]) and tuples. As majority of these concepts are already presented in other programming languages they will be described briefly and only when necessary to illustrate the key concepts of our research.

**3 Multi-structured compilation units**

**3.1 From code snippets…**

The progress in software engineering reached the stage when a lot of software was already implemented. However, 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 two major tasks. First is to quickly solve particular problem based on what was already done. The second one is to develop new framework or library for further reuse or sophisticated software. So, the idea that the same programming language should combine ability for fast development of code snippets and big software components can be solved providing support for different kinds of compilation units. Simplest compilation unit is just a sequence of statements whereas more structured and complicated compilation unit could be a set 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 the console or a 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 declaration statement can be added

**var** ch **is** 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 **is** Char(readChar())  
myRoutine()  
myRoutine **is**  
 // Some statements...  
**end** myRoutine

It contains use directive with the semantics well-known from Ada [Ada] and Modula-2 [Modula-2] – no more need for prefixing write and readChar in this source with StandardIO. The call to the myRoutine routine is the last statement of the anonymous routine which starts with write and ends with the myRoutine call. In other words if the 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 developers 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 parentheses. var keyword is well-known from Pascal-like languages (and nowadays Scala [Scala]) and has the similar meaning denoting the kind of the entity being declared. There is a discussion still whether we really need var specifier or may just omit it in most cases.

**3.2 …To program units**

Now it’s time to talk about basic notions 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 on the presence of a value to be returned as a result a procedure or a function is defined. So, routine is a function if a return type is defined for it. If there is no return value or the 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 the difference. Routines can be anonymous (just set of statements, there is no way for programmer to call such a routine by name), be part of a class or a module (for classes they are called methods as well) and be standalone routines which are at the same level as classes and modules (like in Ada and C++ [C++]).

Next are class and module. Both concepts are not new and they have big fraction of common stuff between them. They both are named containers of features; they both are compile-time concepts. This allows using the term *unit* which denotes that we do not care whether we talk about class or module. Modules are typically used to group a set or related resources (data and functionality) which leads to single module instance at run-time, while classes imply that programmer can create unlimited number of class instances (objects) at run-time. That makes the essential difference between module and class. Class in fact defines a type while module does not [Szipersky].

Some examples of classes and modules with comments are following:

**module** M0 **is**  
 data **is** Integer  
 incData **is**  
 data := data + 1  
 **end**

decData **is**  
 data := data – 1  
 end

**ctor** // Module initializer  
 io.write ("Module M0 initialized.\n")  
**end** M0

**module** M1 : M0 **is** // Module can inherit from other module(s)  
 ...  
 **ctor**  
 io.write ("Module M1 initialized.\n")  
end M1

**module** M2 : M0 **is** ...  
 **ctor**  
 io.write ("Module M2 initialized.\n")  
**end** M2

**class** X : A, B **use** M1 **as** m1, M2 **as** m2 **is**

// The function returns pair of values (tuple)  
 showData: (Integer, Integer) **is**  
 (m1.data, m2.data)  
 **end**

r1 **is**  
 m1.incData  
 m2.incData  
 **end** r1

r2 **is**  
 m1.decData  
 m2.incData  
 **end** r2

**ctor** **is**  
 io.write ("Object of type X initialized.\n")  
 **end** ctor

**end** X

Units contain features; features can be either routines or attributes. A routine can be function or procedure and attribute can be mutable or immutable. The only difference between them is that immutable attribute gets its value at declaration and keeps it till the end of its scope while mutable attribute may get a new value.

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 deduced by the compiler based on the declaration context (e.g., from the first initialization of an entity). Examples of the simplest cases of mutable and immutable declarations follow:

**var** entity **is** Type  
 // Type of ‘entity’ is explicitly defined as Type  
const5 **is** 5  
 // Type of ‘const5’ is type of 5, i.e., Integer

As a consequence, immutable entity cannot be the target of the assignment statement while immutable one can. Entity is either unit’s attribute or local attribute inside a routine or reference to a current object (“this”).

immutableAttr **is** *some-expression* // The type of the attribute  
 // gets deduced from the expression*.*

**var** mutableAttr **is** Type(*some-expression*) // Constructor of type T is used to initialize the entity

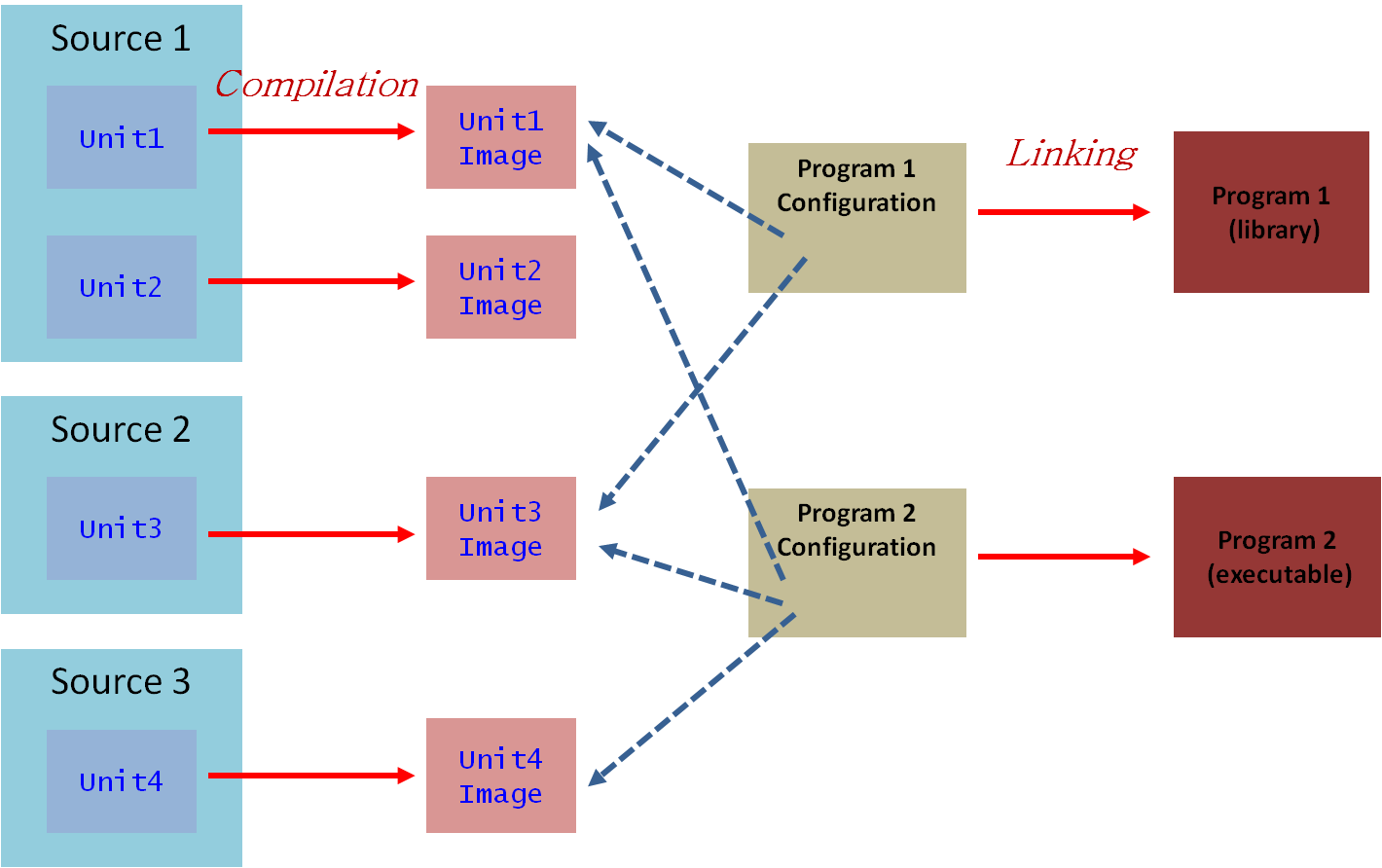
immutableAttr := *expression* // Compile time error!  
mutableAttr := <expression>  
 // Valid assignment provided if types are OK

At the top of the source with program code we have standalone and anonymous routines, classes and modules. And there could be some relations between them. Routines may use classes as types while declaring entities. They can also use modules getting access to modules features. Modules and classes can inherits features from each other and use other modules. Usage of the module can be associated with the source with the program code and then all routines and units get access to this module. So, there are two basic kinds of relations - inheritance and usage. Inheritance can be conformant and non-conformant one. Conformant one allows assignments of descendant objects to ancestor ones while non-conformant does not. Usage allows getting access to features of the module without module name prefix or with a different prefix to resolve name clash. Inheritance means that one unit makes all features of all its ancestors its own features potentially adding new and overriding inherited ones. It is important to mention that inheritance works on per feature basis not unit-wide. As unit may have several parents multiple inheritance is treated as normal for both classes and modules and is a method of code reuse. The name clash issue is resolved not at the unit level but at the level of the call to a particular unit feature. In other words unit may have features with the same name, with different signatures (including conformant ones) inherited from different parents and compiler will be validating the access to these features not full correctness of the unit in terms of inheritance. This will be described later in more details section [Inheritance and its usage].

At the moment flat model is considered as the proper one – no nested routines, classes or modules. It makes units and standalone routines atomic elements for massive reuse. Together with type-parameterized and/or constant-parameterized classes and standalone routines it empowers programmers to develop highly reusable code. Also it is important to note that all units in the same source are mutually visible. And all names of classes, modules and standalone routines are in the same name context. So, let’s see how one can build programs and libraries from these components.

**3.2 Overall program composition**

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



<Будет краткое объяснение, в частности, слова о configuration file>

**4 Inheritance and its usage**

As inheritance is one of important relations between units let’s dig into details how it works. Syntactically it is expressed as every unit may explicitly state the list of parents and depending on the type of inheritance from the particular parent state if conformance will work or not. So, example below show this

**class** A: B’, C, D, F’ **is**

/\* class A inherits features from units B, C, D and F. For B it will be non-conformant inheritance while for C conformant one\*/

**end** A

a is A

**var** b is B

**var** c is C

b := a /\* compile-time error. Type of a does not conform to type of b\*/

c := a // all is OK

In case of inheriting from module conformance is not applied as module is not a type and we cannot define an entity of module type. So, let’s state that D is a module and it inherits from class E

**var** e **is** E

e := a // Is valid

**var** f **is** F

f := a /\* is not valid, as type of a does not conform to type of f\*/

**module** D: E **is**

**end** D

**class** E **is**

**end** E

**class** F **is**

**end** F

It implies that if there is mixture of classes and modules while inheriting there could be a need to explicitly specify some class ancestor(s) as a parent(s) for a class and state that conformance does not work.

**4.1 Inheritance feature focused.**

Let’s consider the following declarations

**class** A:B,C **is**

**end** A

**class** B **is**

foo <S> // <S> states for some signature foo has

**end** B

**module** C **is**

foo <S>

**end** C

How many copies of foo are in A? If both foo are growing from the same seed and signatures are identical (and body is the same for routines) than there will be only one copy otherwise two different features (regardless of the seed the same or not).

So, if to consider the following code

a **is** A

a.foo (<Valid parameters>) // this call can be ambiguous or not

Please note it works for both routines and attributes! One more example to highlight how inheritance works

**class** A: B, C **is**

**end** A

**class** B **is**

foo (<S1>) <B1> // <Bx> states for the routine body

**end** B

**class** C **is**

foo (<S2>) <B2>

**end** C

Depending on different combinations of S1-B1-S2-B2 we may have different cases. Some will be leading to ambiguity if we try to call the feature. So, if we do not call the ambiguous feature the code is valid and can work! So, the general principle not to verify the inheritance graph fully – we verify usage of features of units. If usage (feature call) can be verified then the program is correct. The only check to be done that inheritance graph does not have cycles.

**4.2 Overriding**

Programmer can declare a feature in a unit which will override nearly all previous versions of this feature. It works in a straightforward way if all signatures are identical and creates some complicated cases when signatures are conformant. If signatures are not conformant that is compile time error. Here is a simple example for the case with identical signatures. Concept of abstract class and feature is introduced as well in similar manner like other object-oriented languages define it.

**abstract class** B **is**

foo (<S>) **is** **abstract**

**end** B

**class** A **is**

foo (<S>) **is** <B2>

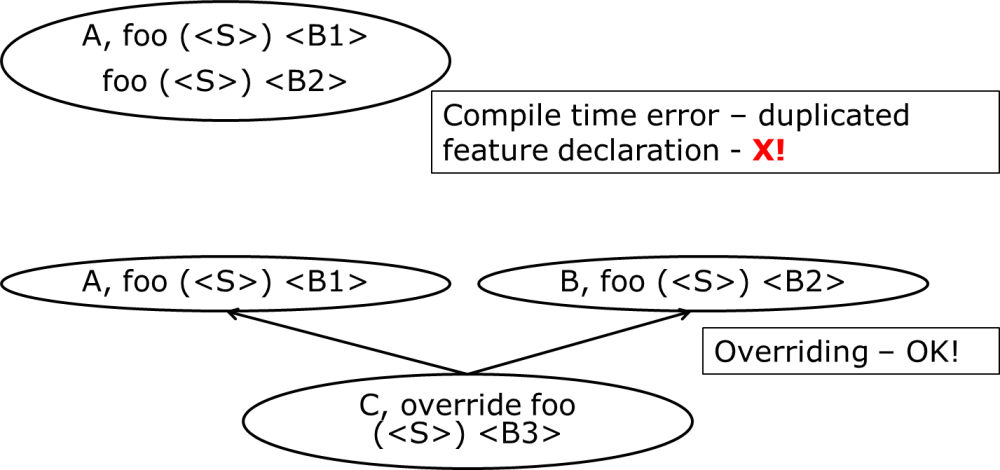
**end** A

**class** C: A, B **is**

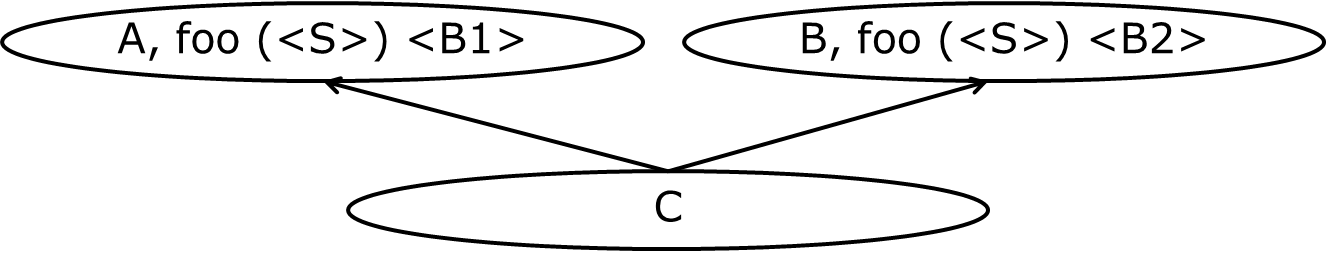
**override** foo (<S>) **is** <B1> /\*Both versions of foo coming from A and B are replaced with version foo from C\*/

**end** C

Of course one cannot declare more than one feature with the same name and signature but different body in one unit. Picture below illustrates that.

****

Let’s consider another example with identical signatures and no overriding. Picture below shows inheritance graph.

****

Then if one likes to access feature foo from the unit C body or its descendants

A.foo (<parameters>) /\* OK! Call version of foo coming from parent A\*/

B.foo (<parameters>) /\* OK! Call version of foo coming from parent B\*/

foo (<exprS>) // Ambiguity! Cannot resolve this call

If we try to access foo from the client code

c **is** C

c.foo (<parameters>) // Ambiguity! Cannot resolve this call

So, in such case there is no way to call feature foo from the client. But in case of polymorphic assignment it may work because no ambiguity happens

A **is** A = C()

a.foo (<exprS>) // OK! Version from A is to be called

The key thing here is that foo from A and foo from B come from different seeds!

**4.3 General scheme**

Let’s consider that two routines foo (<S1>) is <B1> and foo (<S2>) is <B2> inherited by a class. Then

S1 = S2 and B1 = B2 => the same routine – all is OK!

S1 = S2 and B1 != B2 => it will lead to ambiguity on access implying compile time error!

S1 != S2 – these are 2 different routines! If they come from the same origin and seed then select is to be applied like Eiffel does for the same case.

S1 != S2 and override occurs with S3 ->(conforms to) S1 and S3->S2 – all is OK!

Let’s consider two variables variable: T1 and variable: T2 inherited by a class

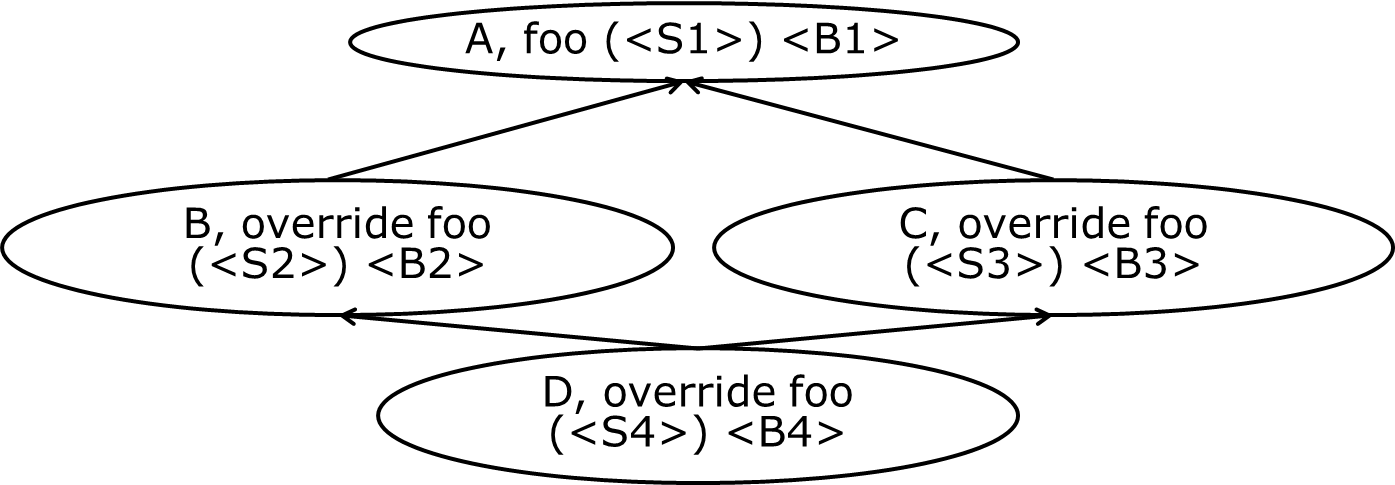
T1 = T2 – the same variable – all is OK!

T1 != T2- these are two different variables in the class! To resolve select case!

override variable : T3 – when T3 -> T1 and T3 -> T2 – all is OK!

Routines and variables are not much different in terms of inheritance. It is not yet fully decided whether to allow overriding of immutable attributes.

**4.4 Cat calls**

****

S2->S1, S3->S1, S4->S2 & S4 -> S3, S5 -> S1

a **is** A = D()

a.foo (<S2>) // version from B must be called! OK!

a.foo (<S3>) // version from C must be called! OK!

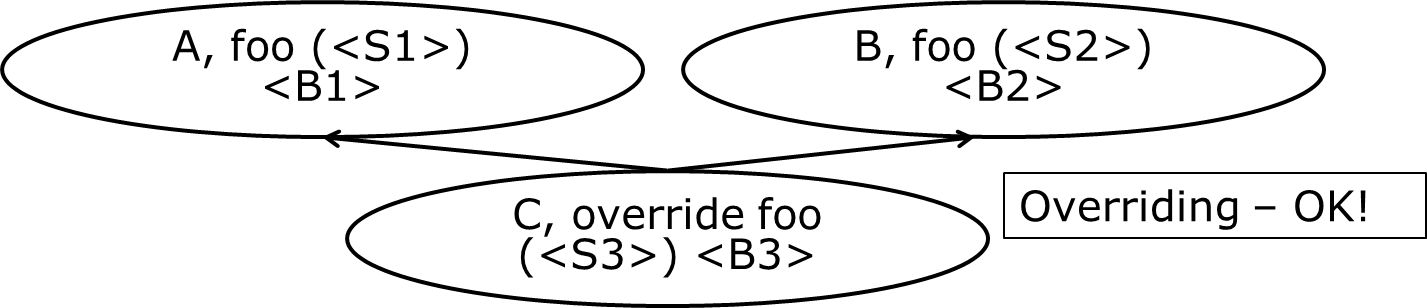
a.foo (<S4>) // ambiguity

a.foo (<S5>) // cat call

Work to resolve the cat calls is being done by computer science specialist and may be reused. Conservative approach is to consider system wide analysis for call validity.

**4.5 Power of overriding**

Let’s remind that again that overriding is the mechanism which allows to state the fact that within the current class there is a feature which overrides all versions of this features coming thru different parents. The list of features being overridden is determined by the conformance of the current feature signature to overridden features’ signatures. Better to looks at the picture below

****

If <S3> conforms to <S1> and to <S2> then there is only one feature foo in C with signature <S3>. If <S3> conforms to <S1> or to <S2> then there will be two features foo. One will be with <S3> signature and another with non-conforming to <S3> signature. Also note that in case of routines abstract status of <B1>, <B2> and <B3> can be arbitrary.

So, we may override abstract routine with abstract one. We may provide an effective body with overriding. We may merge several abstract routines into one using overriding. We can also do the opposite – if we have an effective routine (routine with the internal or external body) we may override it with an abstract routine. This gives us full flexibility to control the level of abstraction while inheriting.

Also if we like to select one version of several versions which come from parent units then we can use override with no body but the name of the current class instead. See the example below

**class** A: B, C, D **is**

**override** (D) foo // 1st variant of syntax

**override** D.foo // 2nd variant of syntax

**end** A

Please note that such form of overriding will override only versions to which signature of D.foo conforms to.

So, inheritance together with overloading and overriding allows using multiple inheritance for building proper units and focusing not on full correctness and uniqueness of every unit but on verification of feature calls.

**5 Generic units**

Any class or standalone routine can be defined generic – meaning that it is parameterized. 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] **is** ... **end**

There are two options for type: constrained genericity and non-constrained one:

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

Generic implementation recommendations and details are not a part of the language definition; we should not make any assumptions on whether every new instantiation implies new portion of code or not. It is just a different instantiation.

Another caveat here is how to create object of generic type inside a unit. In particular, the notation for specifying of 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[G, G1 **ctor**(G)] ...

**6 Assertions**

Assertions are not new; systematic use of assertions has big pluses in terms of increasing software reliability. Eiffel [Meyer] has a very powerful mechanism of Design by Contract (C) and it seems reasonable just to reuse it in the context of design decisions described above. So, suggested approach is to support 3 types of assertions: preconditions, postconditions and unit invariants. Loops are out of scope for now as practice shows that programmers nearly ignore loop-related assertions. We may consider support assertion check within the body of any routine at any place, then we may use assert keyword for that. In general assertion is a valid Boolean expression optionally labeled by a name. Routine precondition is a set of assertions which must be all True at routine entrance. It is started with keyword require. Routine postcondition started with keyword ensure is a set of assertions which must be all True at routine exit. Unit invariant is a set of assertions which must be all True to any unit routine entrance and exit, except construction procedures – invariant must be True after its call. invariant is the keyword and invariant section is put at the end of the unit.

**6.1 Assertions example**

Let’s consider example of class Stack and how assertions define semantics of its run-time behavior. Assertion will use old expression in postconditions. Old expression allows referring to the value of an entity which it had at the entry point of the routine. Comments in the class interface give nearly all the information about the class and its features. Class interface does not contain the routines implementation and it is not a substance of the language, it is just a form which should be generated by class interface tool for more convenient analysis of the class or module. Class interface should not contain the private features of the class or module.

**class** Stack[G] // Interface of class Stack

push (e: G) **is**

**ensure**

/\*Set of assertions which define behavior of push \*/

“count was incremented”: count = **old** count + 1

“element was added into stack”: has (e)

**end** push

pop: G is

**require**

“stack not empty”: count > 0

**ensure**

“count after pop decremented”: count = **old** count – 1

**end** pop

count: Integer // The number of elements in the Stack

has (e: G): Boolean /\* It returns True if element is in the Stack\*/

**ctor** /\* It means that class Stack has a construction procedure with no arguments \*/

**invariant**

“Stack consistent”: count >= 0

**end** Stack

So, systematic use of assertion allows fixing the semantics of the software and dynamically checking that execution does the expected job at run-time. As assertions are parts of routines which belong to units like classes and modules let’s see how they are affected by inheritance.

**6.2 Assertions and inheritance**

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

**class** MyStack[G]: Stack[G] **is**  
 **override** push (e: G) **is  
 ensure then** *Some-boolean-expresssion-1*

**ctor**

**invariant** *Some-boolean-expresssion-2*

**end** MyStack

Invariants are AND-ed while inheriting.

**6.3 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 continues 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 unit invariant is violated it implies that the unit 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.

Example:

/\* Here we create a stack and try to pop immediately after creation\*/  
stack is Stack[Integer]  
i **is** Integer = stack.pop /\* Precondition violation: stack not empty \*/

We can provide flexible control over assertions – first approach is to select (at compile time) which kind of assertions we like to have enabled at run-time at routine, unit, group of units or library/program wide level. The simplest scheme is just to tell the compiler assertions/preconditions are on or off. Also we can 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 a smarter compiler implementation.

**7 Value and reference classes and entities**

We like to distinguish two things: object and reference to object. It may have some implementation similarities to C-struct and pointer to the struct. The key thing is that this difference is to be reflected only in the semantics of the assignment. All feature calls of an object have the same semantics for both reference and value entities.

entity.featureCall /\* It does not matter if entity is a reference or value – the key thing we need to call a feature of the entity\*/

It’s possible to determine the type of the run-time entity while declaring it. Let’s start with an example to present the concept:

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

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

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

a0 **is** A // a0 is reference entity

a1 **is val** A // a is value entity

b0 **is** B // b0 is value entity

b1 **is ref** B // b1 is reference entity

**class** A

**end** A

**val class** B **:** A

**end** B

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: all attributes of val1 are filled with corresponding attributes from val2. It implies that some val2 may have more attributes than val1. That how conformance works here\*/

val1 := ref1 /\* field-by-field copy: all attributes of val1 are filled with corresponding attributes from an object which ref1 refers to at run-time.\*/

Implicit outcome – all arguments are passed by value.

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

**8.1 Tuples**

***[Этот раздел не правил. Надо бы переписать его в более «повествовательном» стиле.***

***АК: Попробовал чуть-чуть подправить … Смотри …***

***]***

Tuple is a group of “something”. “Something” can be type, expression or field declaration. For example

(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

So, tuples are just natural part of every programming language, every time when we have some sequence in the programming language we have a tuple. To deal with tuples let’s consider conformance rule.

Tuple T1 conforms to tuple T2 if for every i in 1..n T1i conforms to T2 i when n = T1.count and n <= T2.count. Note that such definition of conformance is the basis for functions with “growing” number of parameters. In other words any function can be called with more parameters than it was declared having static type check in place. Then we may assume that every routine has only one parameter – a tuple, possibly empty and it returns a tuple as well with zero or more elements. Then procedure is a function which always returns empty tuple. And then any function can be called from procedure call statement like 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 an entity

t **is** Tuple(expr1, expr2, expr3)

foo(t)

/\* That is a feature call with a tuple. ((t)) is identical to (t)\*/

t1 **is** Tuple(e1, e2, e3, e4)  
foo(t1) // Valid as well!

// Tuples may be typed:  
t2 **is** Tuple(Integer, Real, Boolean)  
t2.1 := 5; t2.2 := 5.5; t2.3 := True

// Tuples may have named fields  
t3 **is** Tuple(i: Integer, r: Real; b: Boolean)  
t3.i := 5; t3.r := 5.5; t3.b := False

**8.2 Multi-types**

Can an entity have several types? Yes. If entity is declared specifying several types then such 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

e **is** 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 belongs 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 a compiled form only or 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. One more example to highlight the mechanism

StandrdIO.write (

add (5, 5), “”,

add (5.5, 15.5), “”,

add (1.5, 3), “”,

add (5, 3.5), “”

)

add (e1, e2 **is** Integer|Real) **is**

e1 + e2  
**end** add

Of course classes Integer and Real must have ‘+’ with proper signatures like

‘+’ (Integer, Integer)

‘+’ (Integer, Real)

‘+’ (Real, Real)

‘+’ (Real, Integer)

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

**9.1 Null safety problem**

Null pointer is the constant which requires checks before we can work with dereferenced entity. So, safe programming implies checking for Null in every point where we use a reference. The simplest optimization is to split the world in two parts – the first one (which is safe) entity cannot be Null and another entity can be potentially Null. Then in the safe world every call to a feature is safe – no more need for Null check in the program code or run-time – it just cannot be Null. In the unsafe world we cannot access features without either run-time transformation into safe entity or just a check that actual run-time value is not Null. So, another issue which we have that we cannot get rid of Null unsafe code at all – we cannot know initialization of some entities. For example in class List the reference to the next List element can be a valid List element or… non-initialized. So, we replace the concept of Null constant with the concept of non-initialized entity. And at the same time we go to the next level of abstraction and solve two issues – Null safety for feature calls and controlled approach to entities with deferred initialization.

**9.2 The solution taken: entities with deferred initialization and typeof predicate**

As we wish entities to be initialized at the time of their declaration and wish not to have non-initialized entities. That is great but we cannot guarantee the proper initialization for variables when declaring them, or we force programmers to invent artificial objects or use some Boolean flags. So, in fact we have two distinct kinds of entities: ones will be initialized at declaration time while others can be either initialized or non-initialized (or better say their initialization can be deferred) or may lose their initialization. To support 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 potentially non-initialized into another one declared as always initialized one and ability to de-initialize an entity. The following language constructs seem to solve this:

entity **is** ?T  
 // After this declaration the entity is considered  
 // non-initialized. It implies that we cannot  
 // do any operation based on this entity

?entity  
 // Explicit de-initialization of the entity.  
 // It becomes non-initialized again

**if** entity **typeof** T **then** entity.foo **end**  // Inside then-part of the if statement  
 // entity is definitely of type T therefore  
 // call to entity.foo is definitely safe

Let’s consider rather expressive example:

i **is** ?Integer // declaring non-initialized variable  
i := i + 5 // Not valid; compile time error  
**if** i **typeof** Integer **then** i := i + 5 **end** // Safe code

So, **typeof** operator works as runtime check that entity has the dynamic type conforming to the type specified. So, it mixes the run-time type check with initialization check.

It is essential to note that such scheme works for both reference and value objects.

Let’s see how **typeof** works with more examples.

**C is ?C**

**if** c **typeof** C1 **then** // if c contains 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  
 // check else clause means that c is actually  
 // non-initialized.  
 // If there is no such clause C type check then c can be  
 // either non-initialized or attached to an object which  
 // type does not conform to all other when alternatives  
**end**

Very similar considerations are also applicable to loops and data structures. See example of List class below:

l1 **is** List[Integer](5)  
l2 **is** ?List [Integer] // The empty list  
l2 := l1  
**if** l2 **typeof** List[Integer] **then**  
 l2.add(128)  
 **while** l2 **typeof** List[Integer] **loop** StandardIO.write(" ", l2.item)  
 l2 := l2.next  
 **end  
end if**

**class** List[G] **is**  
 item **is** G  
 next **is** **?**List[G]  
  
 setItem (other: G) **is** item := other  
 **end** setItem  
  
 add (other: G) **is**  
 next := List[G](other)  
 **end** add  
  
 **ctor** (element: G) **is**  
 setItem (element)  
 **?**next  
 **end ctor  
  
end** List

Case multiple-choice statements can be used as well.

foo (c: **?**C) **is  
 case** c **of  
 when** T1 **then** // work with c as entity of type T1  
 **when** T2 **then** // ...  
 **when** T2 **then** // ...  
 **else** // ...  
 **end case  
end** foo

**10 Language implementation**

The compiler is being written in the language itself using bootstrapping technology. The compiler for a reduced subset of the language (as an initial version for bootstrapping process) is being written in C# for speeding up the work. After the initial compiler is bootstrapped (i.e., has been re-written in its own language), it will be ported into the native environment for further stepwise improvements.

The LLVM infrastructure [LLVM] is used as compiler’s back end. The primary hardware platform is ARM.

**Что ещё написать в этом разделе?- уж больно он маленький получился...**

**А я вот сомневаюсь что нам сейчас надо иметь такой раздел … Мы хотим говорить о некоторых аспектах языка а не о том как его делать … Но если хочешь то тогда надо быть готовым отвечать на вопросы про ЛЛВМ … А я не готов … Просто сказать что все так делают … Как-то не то …**

**References**

1. Bertrand Meyer: Eiffel: The Language: <http://www.amazon.com/Eiffel-Language-Prentice-Object-Oriented-Series/dp/0132479257>
2. ECMA Standard 367 -- Eiffel: Analysis, Design, and Programming Language: <http://www.ecma-international.org/publications/standards/Ecma-367.htm>
3. ISO/IEC 8652:2012: Information technology -- Programming languages – Ada: <http://www.iso.org/iso/catalogue_detail?csnumber=61507> **Евгений проверь плиз!**
4. Стандарт С++ **Евгений это твоё плиз!**
5. Niklaus Wirth: Programming in Modula-2: [http://download-v2.springer.com/static/pdf/87/bfm%253A978-3-642-83565-0%252F1.pdf?token2=exp=1430301183~acl=%2Fstatic%2Fpdf%2F87%2Fbfm%25253A978-3-642-83565-0%25252F1.pdf\*~hmac=24af9ace696499d5252e41492993c08419b328a7cb7c88245a00016bdc8ffaae](http://download-v2.springer.com/static/pdf/87/bfm%253A978-3-642-83565-0%252F1.pdf?token2=exp=1430301183~acl=%2Fstatic%2Fpdf%2F87%2Fbfm%25253A978-3-642-83565-0%25252F1.pdf*~hmac=24af9ace696499d5252e41492993c08419b328a7cb7c88245a00016bdc8ffaae)
6. Clemens A. Szyperski: Import is Not Inheritance Why We Need Both: Modules and Classes. <http://research.microsoft.com/en-us/um/people/cszypers/pub/ecoop92.pdf>
7. The LLVM compiler infrastructure. <http://llvm.org/>
8. Scala - **Евгений это твоё плиз!**