**Alexey Kanatov**  
Samsung Research and Development Institute, Moscow, Russia  
Alexey.V.Kanatov@gmail.com

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

**Inheritance – what’s new?**

**Abstract**. The paper gives a brief overview of existing approaches to inheritance in programming languages and suggests alternative approach to it keeping multiple inheritance as the general mechanism for the software reuse. Approach is based on overloading and overriding with conflicts resolution at call sites based on conformance instead of full validity of the system inheritance graph.

**Keywords**: multiple inheritance, overloading, overriding, conflicting version, origin and seed, conformance.

**Content**

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

**Overview of approaches to inheritance**

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

**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.

**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

**unit** A **extends ~**B, C, D, ~F

/\* unit 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

b **is** B

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

e **is** E

e := a // Is valid

f **is** F

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

**unit** D **extends** E

**end** D

**unit** E

**end** E

**unit** F

**end** F

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

**4.1 Inheritance feature focused.**

Let’s consider the following declarations

**unit** A **extends** B,C

**end** A

**unit** B

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

**end** B

**unit** C

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

**unit** A **extends** B, C

**end** A

**unit** B

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

**end** B

**unit** C

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 unit and feature is introduced as well in similar manner like other object-oriented languages define it.

**abstract unit** B

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

**end** B

**unit** A

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

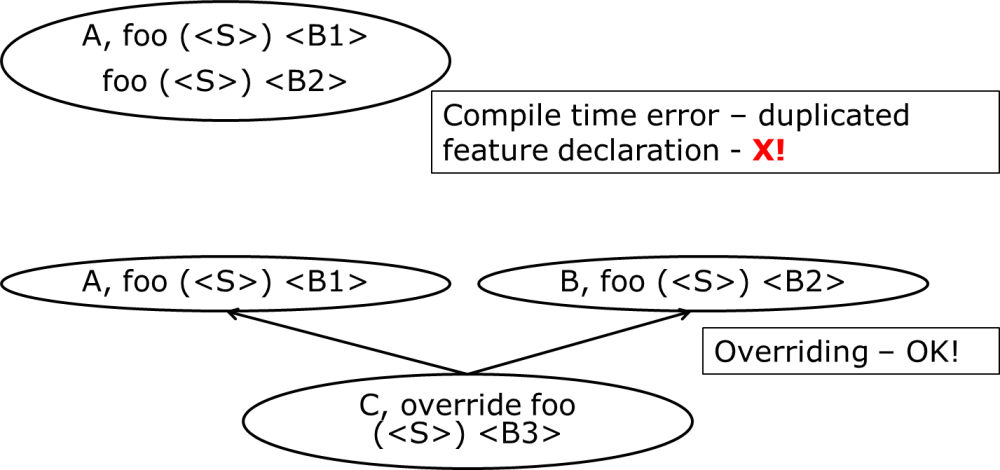
**end** A

**unit** C **extends** A, B

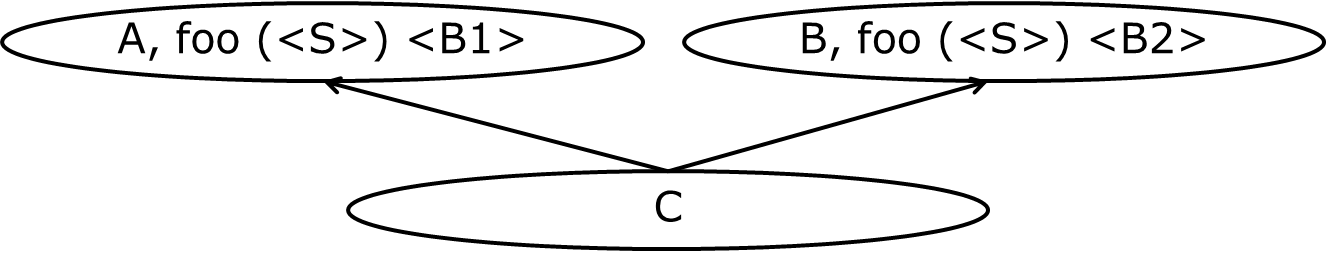
**override** foo (<S>) **is** <B1> /\*Both versions of foo coming from A and B are replaced with version foo from C\*/

**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

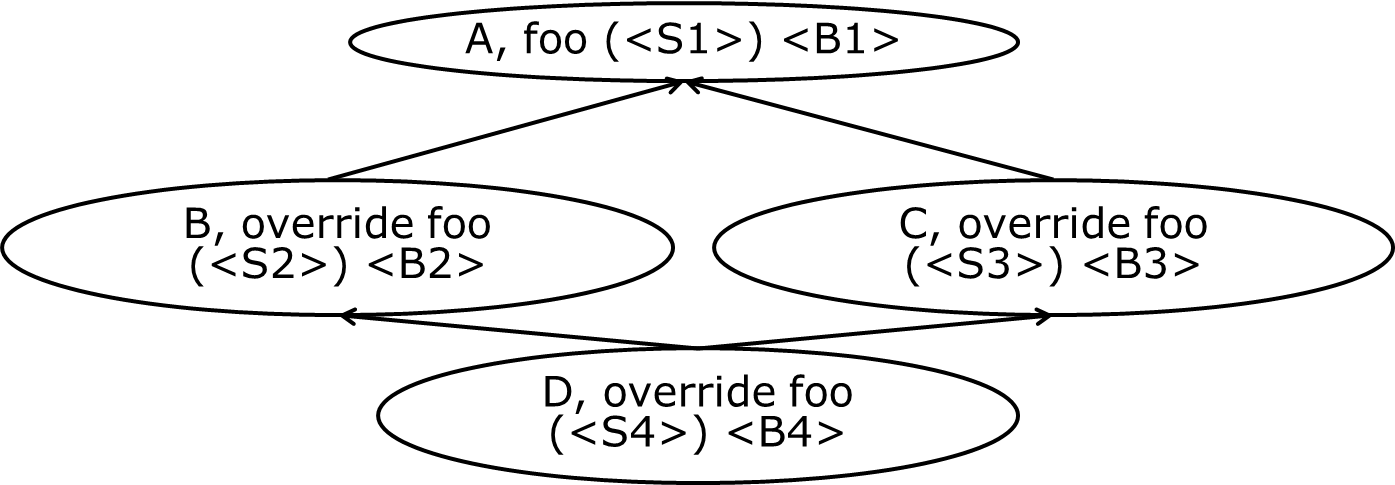
T1 = T2 – the same variable – all is OK!

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!

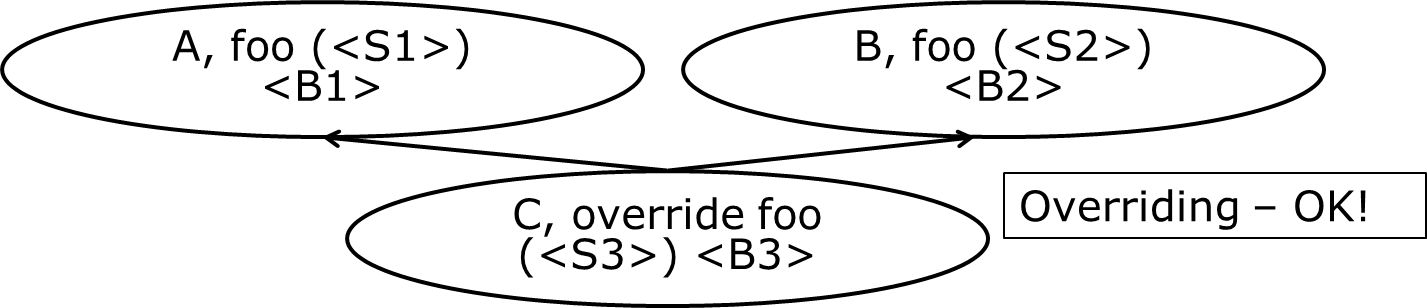
a.foo (<S4>) // ambiguity

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 unit 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 unit instead. See the example below

**unit** A **extends** B, C, D

**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.

* Cуть подхода: множественное наследование с перегрузками и конфликтами + проверка правильности обращения к свойствам.
* Необходимые проверки графа
  + Отсутствие циклов в графе наследования
  + Разрешение конфликтов версий для полиморфизма
* Необходимые проверки при обращении
  + Наличие свойств(а) и его доступность
  + Конформность типов параметров и аргументов
  + Cat calls

# АЛЬТЕРНАТИВНЫЙ ПОДХОД К НАСЛЕДОВАНИЮ

Наследование как вид отношений между контейнерами дает возможность расширять возможности новых контейнеров, используя уже имеющиеся и доступные свойства существующих. Основные проблемы, которые возникают при реализации наследования,– это конфликт имен, неоднозначность версий свойства при полиморфном присваивании, а также согласование статусов экспорта свойств. Отмеченные проблемы возникают в основном при множественном наследовании, поэтому во многих современных языках возможности наследования ограничивают единичным наследованием, предлагая в качестве паллиативного решения понятие интерфейса (С#, Java) или «протокола» (Swift [7]).

Можно, однако, сохранить механизм множественного наследования со всеми его преимуществами, если не ставить задачей контроль правильности всего графа наследования для всех свойств всех контейнеров - как делается, например в языке Eiffel [5], котрый имеет мощный, но непривычный аппарат адаптации свойств при наследовании. Вместо этого можно ограничиться решением более простой задачи: проверки на наличие однозначного разрешения *обращения к свойству* контейнера (validity of the call).

Данный раздел содержит обсуждение преимуществ предложенной альтернативы. Рассмотрим схематический пример контейнера, в котором заданы две одноименные подпрограммами с различными сигнатурами.

**unit** A  
 foo(*signature1*)  
 foo(*signature2*)  
**end** А

Общая задача может быть сформулирована так: является ли правильным контейнер А? Иными словами, при каком соотношении сигнатур он непротиворечив? Вместо этой общей задачи поставим более конкретную задачу, которая исходит из *использования* контейнера в качестве типа. Пусть имеется обращение к подпрограмме foo:

a **is** A  
a.foo(*arguments*)

Мы хотим проверить, является ли корректным обращение к a.foo с некоторыми аргументами. Такое обращение будет правильным, если в А есть свойство или несколько свойств с именем foo, и типы аргументов вызова имеют однозначное разрешение среди сигнатур всех свойств foo. Если типы всех аргументов конформны только сигнатуре *signature1*, то перед нами обращение к первой версии foo, если только сигнатуре *signature2* – то ко второй foo. Если же аргументы вызова конформны обеим сигнатурам, то фиксируется неоднозначность, что трактуется компилятором как ошибка. При этом сам контейнер A с набором указанных подпрограмм не считается некорректным. Что же касается самого контейнера A, то единственная проверка, которая представляется для него необходимой,– контроль двух сигнатур на идентичность.

Механизм перегрузки (overloading) может использоваться также для разрешения конфликтов по именам при наследовании. Рассмотрим, как это может работать, на следующем примере.

**unit** B **extends** A  
 foo(*signature3*)  
 **override** foo (*signature4*)  
**end** B

Контейнер В вводит *новое* свойство foo с сигнатурой *signature3*. Единственное ограничение на *signature3* заключается в требовании неидентичности с *signature4*, определенной в этом же контейнере.

В то же время наличие префикса **override** для foo с сигнатурой *signature4* приводит к перекрытию (overriding) *обеих* foo из базового контейнера А, при условии, что *signature4* конформна обеим сигнатурам *signature1* и *signature2*. Если *signature4* конформна только одной foo из базового контейнера, то другая подпрограмма наследуется в В.

Иными словами, основой механизма разрешения конфликтов перекрытия служит конформность сигнатур. Рассмотрим несколько более сложный пример.

**unit** T1 ... **end**  
**unit** T2 **extends** T1 ... **end**  
**unit** T3 **extends** T1 ... **end**  
**unit** T4 **extends** T2, T3 ... **end**

**unit** A  
 foo(arg1: T1; arg2: T2)  
 foo(arg1: T1; arg2: T3)  
**end** А

**unit** B **extends** A  
 foo (arg1: T4)  
 **override** foo (arg1: T4; arg2: T4)  
**end** B

В контейнерах А и В имеются по две подпрограммы foo с различными сигнатурами. Рассмотрим различные случаи использования этих подпрограмм[[1]](#footnote-1).

**var** a **is** A

Создается объект а, типом которого считается контейнер A. При создании производится вызов процедуры инициализации из A.

a.foo(T1(),T2())

Разрешение однозначно: перед нами обращение к первой foo из А.

a.foo(T1(),T3())

Однозначное обращение ко второй foo из А.

a.foo(T1(),T4())

Неоднозначность: могут быть вызваны обе версии A.foo.

a := B()  
a.foo(T1(),T2())

Однозначное обращение к первой foo. При выполнении будет вызвана версия foo из В с сигнатурой (T4,T4). Представленный случай может служить иллюстрацией так называемых «cat calls»: нарушением типизации вследствие ковариантного переопределения сигнатур и полиморфного присваивания. При проведении полной проверки корректности данного вызова (на уровне всей собираемой системы) он будет отвергнут компилятором.

a.foo(T1(),T3())

Это однозначное обращение ко второй foo из А. При выполнении, как и в примере выше, будет вызвана версия foo из контейнера В с сигнатурой (T4, T4), перекрывающая «базовую» версию.

a.foo(T1(),T4())

Неоднозначность: обе версии foo конформны аргументам.

В заключение данного раздела отметим, что предложенный механизм разрешения неоднозначностей в случае множественного наследования, основанный на анализе полиморфного использования, позволяет не решать общую теоретическую задачу проверки правильности графа наследования, а просто решать конкретные маленькие задачи, с которыми сталкивается программист-практик, оставаясь только с одной концепцией наследования для построения повторно используемого ПО.

**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!**

1. В последующих примерах нотация вида T() обозначает создание нового объекта данного контейнера (то есть, в данном случае он рассматривается как тип). При этом вызывается процедура инициализации контейнера без параметров, если она имеется. В такой нотации можно опускать пустые круглые скобки. Кроме того, перед именем контейнера можно поставить служебное слово **new**, чтобы явно подчеркнуть создание объекта (экземпляра), например **new** T. Наконец, допускается конструкция явного вызова процедуры инициализации экземпляра: T.**init**(*arguments*). Все эти варианты синтаксиса считаются семантически эквивалентными, однако здесь для простоты используется только вид нотации, указанный первым. [↑](#footnote-ref-1)