Alternative approach to inheritance

Samsung R&D Institute Rus, LLC
Alexey V. Kanatov
Eugene Zouev
Dec 2nd 2015

Agenda

- Introduction
- Inheritance models overview
- Alternative approach to inheritance
- Summary

C++ Approach to Inheritance

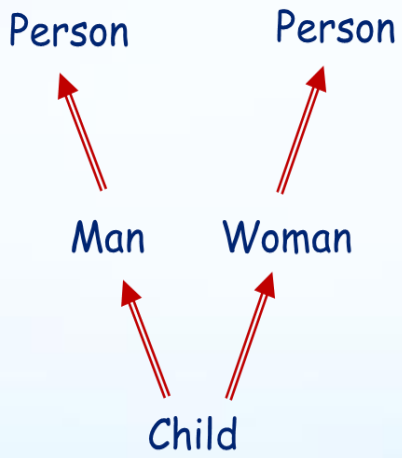
Multiple inheritance

Virtual base classes & virtual inheritance

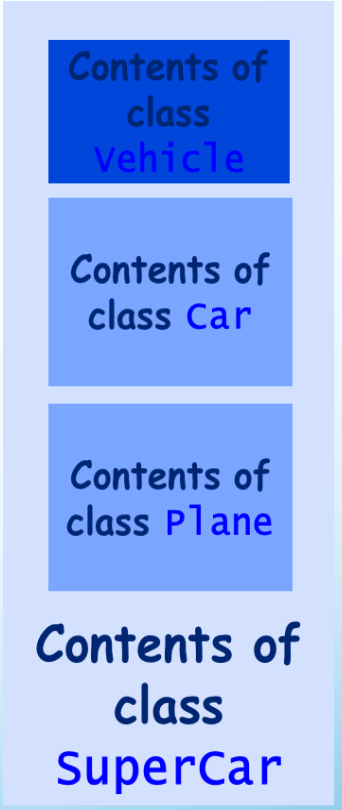
Abstract classes

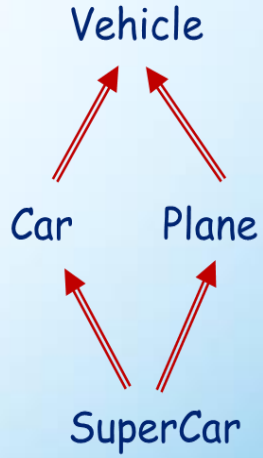
Virtual functions & overriding

The notion of "subobject"



DAG: directed acyclic graph

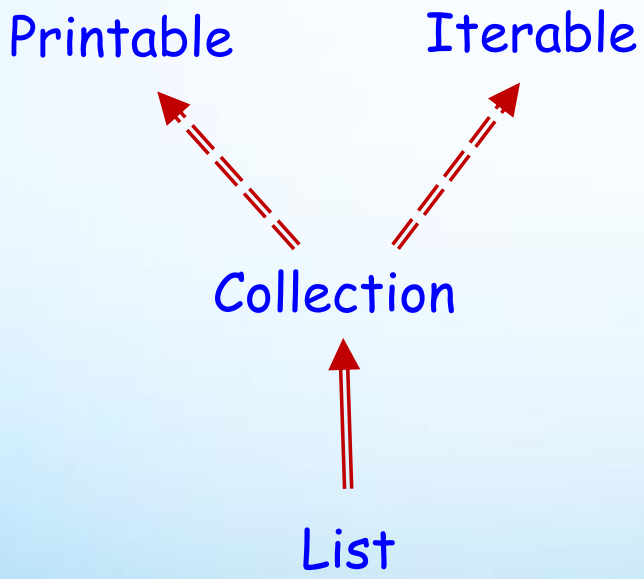




"Diamond" inheritance scheme

Java, C# Approach to Inheritance

- Single inheritance for classes
- Multiple inheritance for interfaces
- Abstract classes
- Virtual functions & overriding



Interfaces:
collections of abstract features

Class:
implementation of features

Derived class:
inherits base class' implementation &
overrides some features' implementations

“Niche” inheritance models

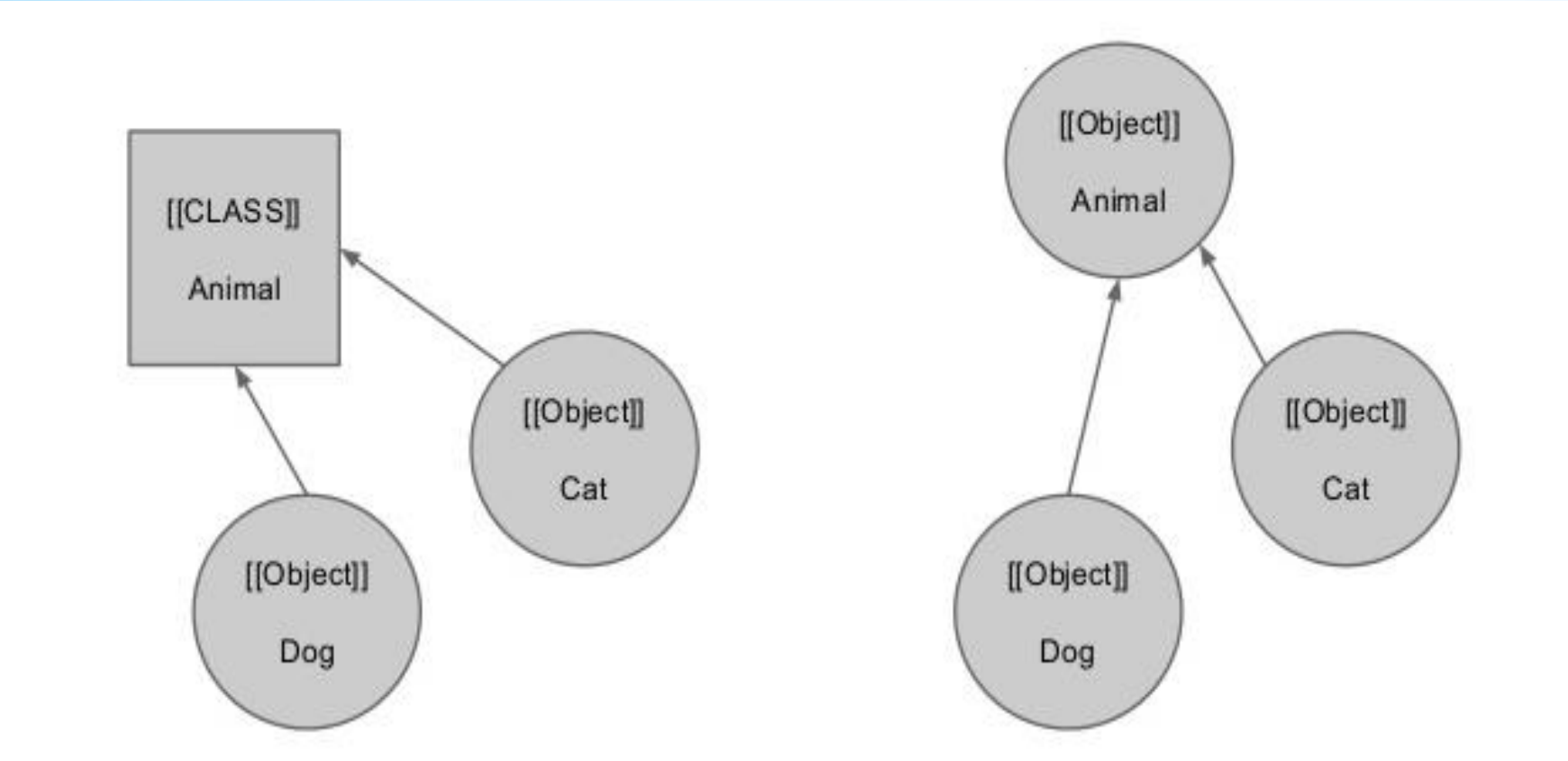
Zonnon Model

- No class inheritance: object are treated as implementations of a number of interfaces (“facets”)
- Interfaces can have default implementations; objects implementing interfaces can make use of default implementations
- No classes, no inheritance, no virtual functions, no overriding, no static fields

Eiffel Model

- Multiple inheritance aligned with genericity
- Very powerful feature adaptation while inheriting (rename, redefine, undefine, export, select)
- “Flat” object layout (no subobjects)
- Programming by Contract © – preconditions, postconditions and invariants aligned with inheritance

Classical vs. Prototypal Inheritance



Classic:
Inheritance is a static relationship;
it's set up for classes of objects

JavaScript:
Inheritance is a dynamic relationship;
it's set up for objects (via predefined "prototype" object property)

Inheritance – basic definitions

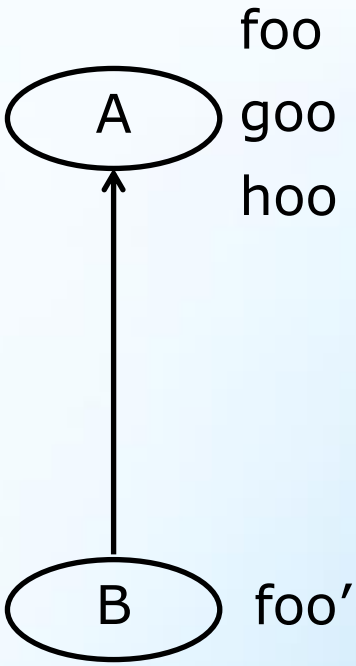
Unit – is a named set of features where feature can be either routine or data attribute (constant or variable). For this talk there is no difference between unit and class.

Inheritance – is a relation between units when features from the parent unit come to the child one and become its part. Inheritance is typically presented as directed graph.

Conformance (→) – is a relation between units based on the existence of the path in the inheritance graph from one unit to another.

Origin – the unit in which feature was declared first time

Seed – the version of the feature which was its first declaration



` means override

```
a: A
b: B
a := b
b := a
```

Origin for foo’ from B – unit A; Seed for foo’ from B – feature foo from A

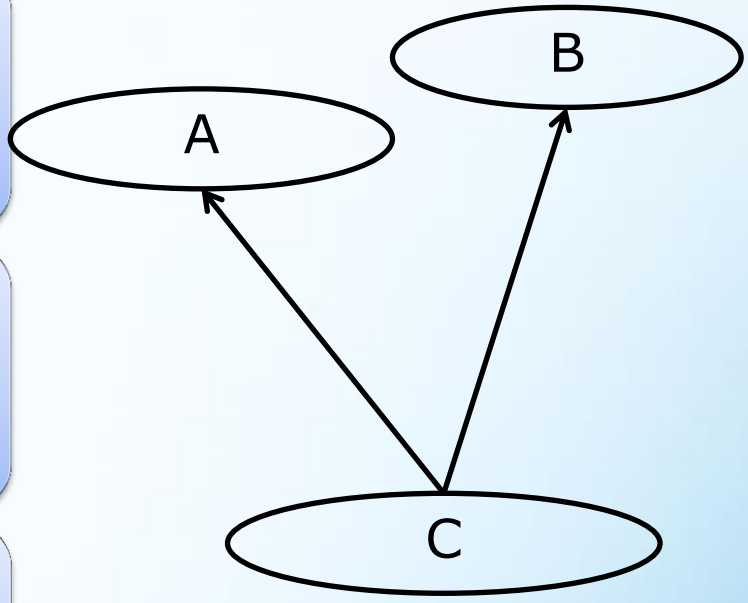
Inheritance – basic definitions

For unit C units A and B are parents

For units A and B unit C is a child (heir)

Graph direction highlights conformance. So, unit C conforms to A and B. C→A & C→B

Conformance defines if an object of one type can be assigned to an object of another type

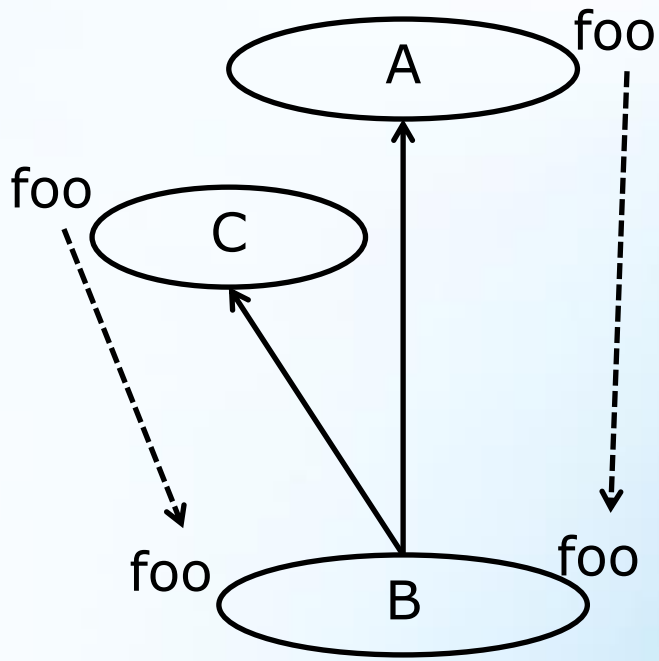


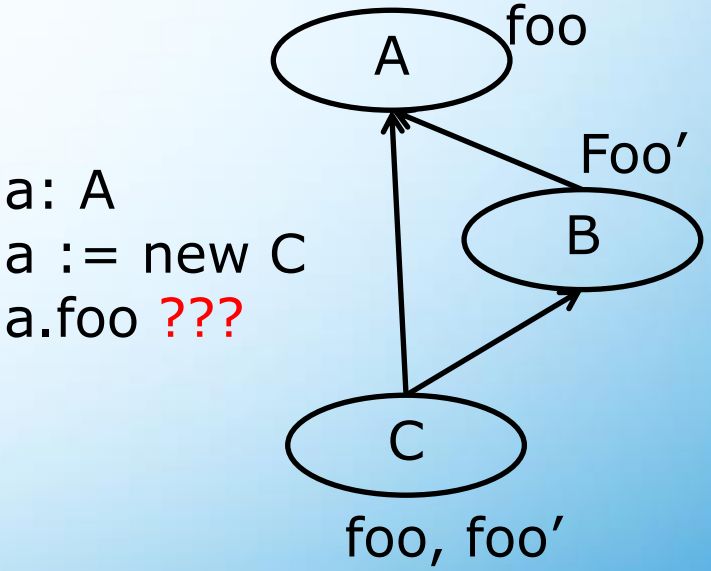
Inheritance – how it works

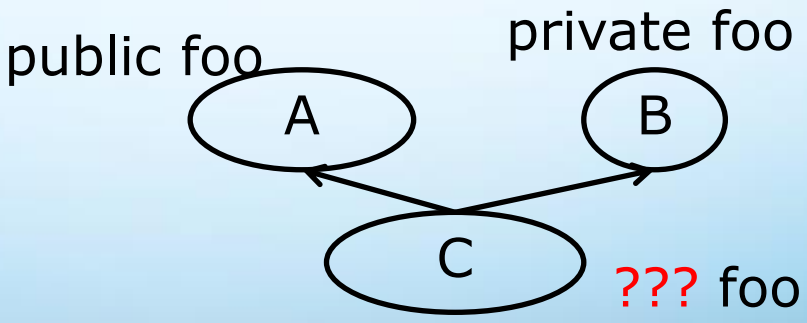
For every feature of the parent unit the place in the child class is to be found.

There are some issues when we apply inheritance:

- Name clashes.
- Versions ambiguity.
- Visibility conflict.







Inheritance – name clashes, overloading and ambiguity

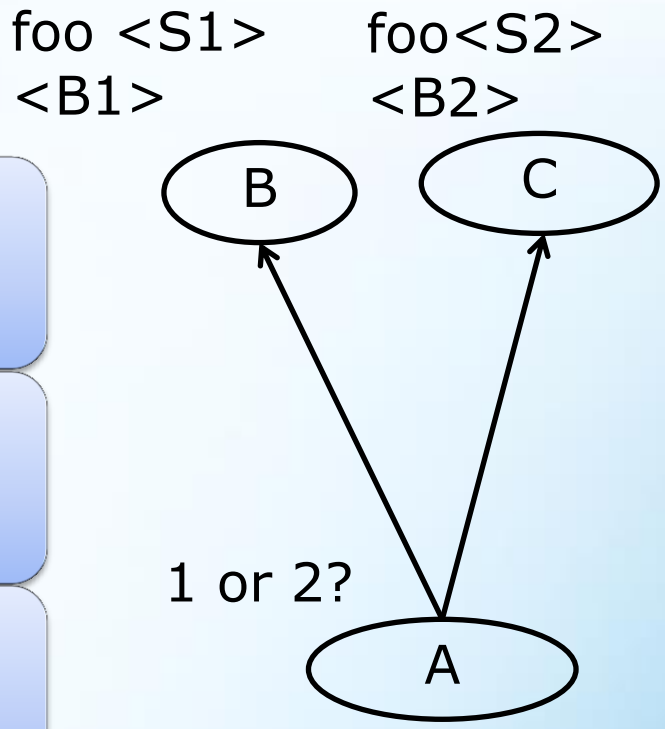
When we have name clash – the question which is to be addressed is how many copies of foo are in A?

if foo from B and foo from C is the same feature then there will be only 1 feature foo in A – that feature. The same means they both have the same origin and seed and the same signature and body for routines.

Otherwise we will have 2 features foo – and the name foo is overloaded.

Code a: A; a.foo (<parameters>) can be ambiguous feature call or resolved successfully depending on combinations of <S1><B1><S2><B2>

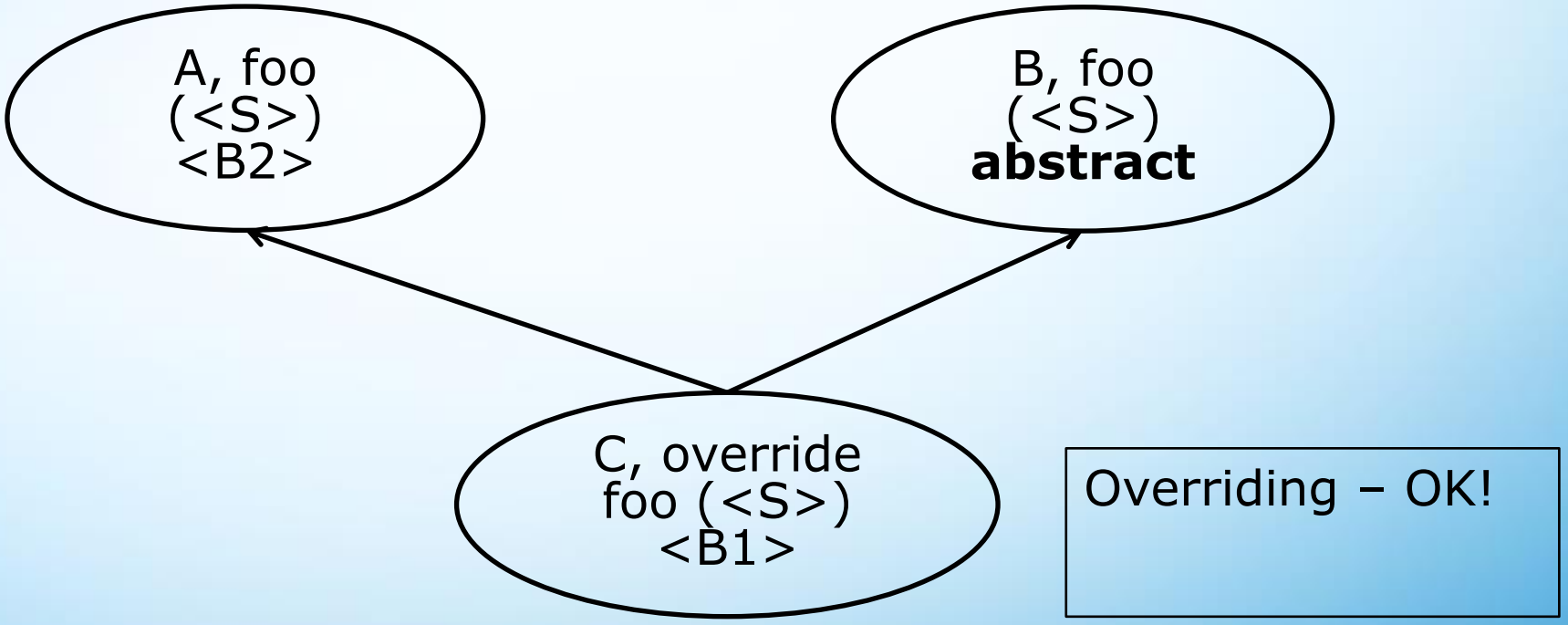
Note if we do not call the ambiguous feature then the code is valid and can work! So, we are not going to verify the full correctness of inheritance graph and every unit – we verify usage of features of units. If usage (feature call) can be verified then the program is correct. The only check to be done that inheritance graph has no cycles.



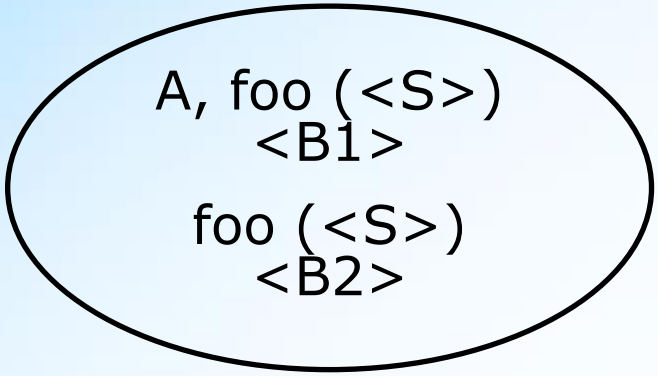
<S> - signature
<B> - body (routine)

Inheritance: overriding

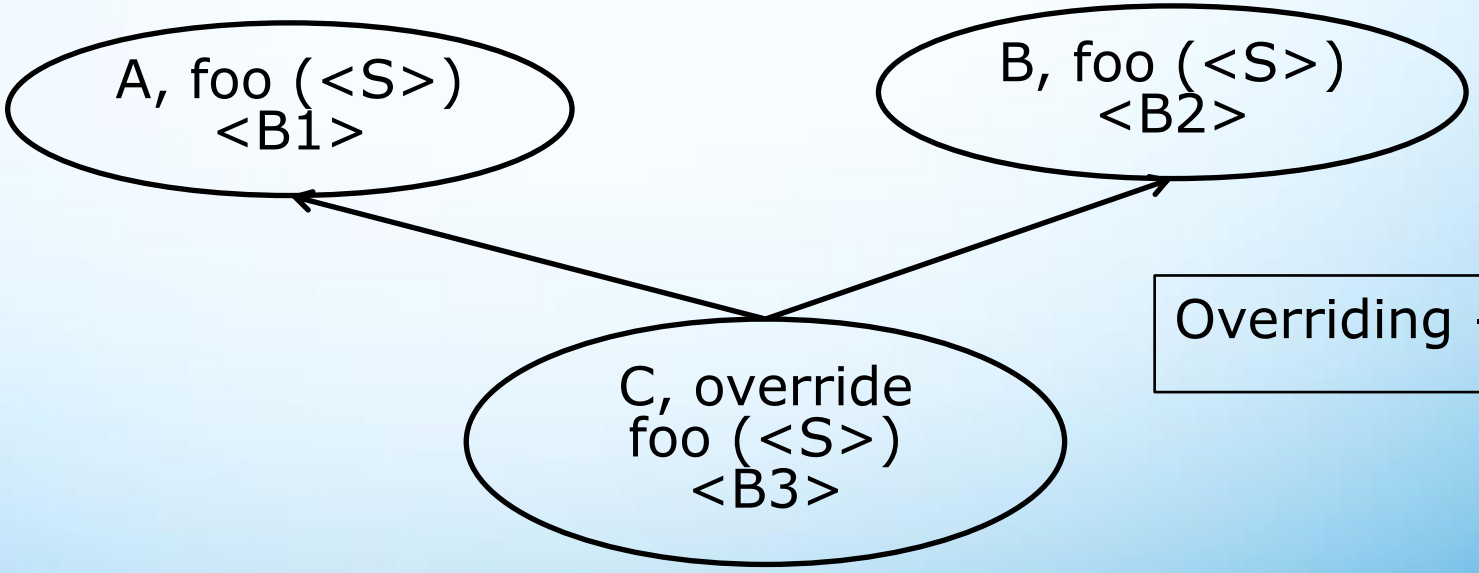
Overriding – we may specify the new version of the feature which will override all the previous versions it conforms to. If there are no such previous versions then it is compile time error – nothing to override. Simple example with identical signature



Inheritance and unit consistency: identical signatures

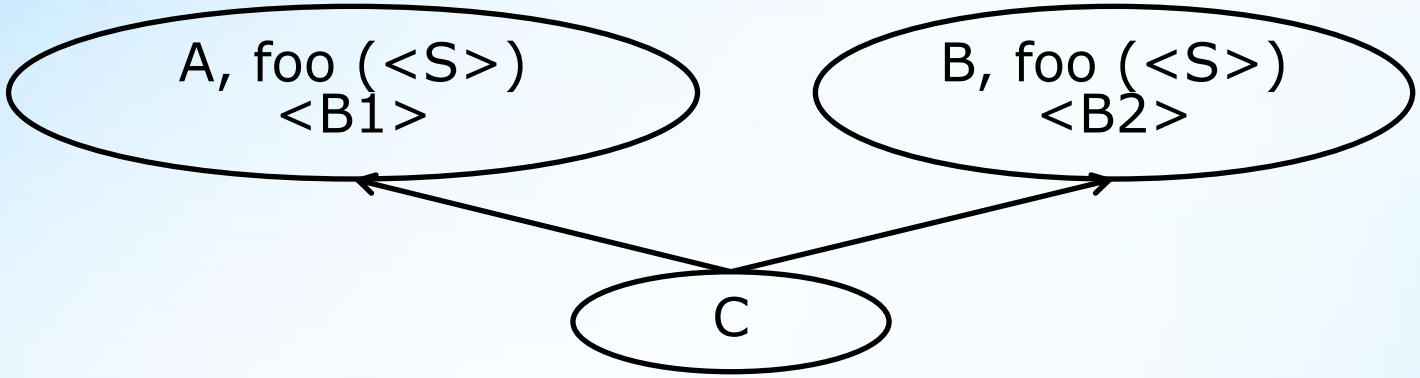


Compile time error – duplicated feature declaration!



Overriding – OK!

Inheritance: name clashes and overloading with identical signatures



If we try to access foo from the unit C and its descendants
A.foo (<exprS>) // OK!
B.foo (<exprS>) // OK!
foo (<exprS>) // Compile time error! Ambiguity!
If we try to access foo from the client code
c: C
c.foo (<exprS>) // Ambiguity! Compile time error!
But even in case of polymorphic assignment
a: A **is** C()
a.foo (<exprS>) // OK! Version from A is to be called
The key thing here that foo from A and foo from B come from different seeds!

Inheritance: name clashes and overloading: general scheme

2 routines foo (<S1>)<B1> and foo (<S2>) <B2> inherited

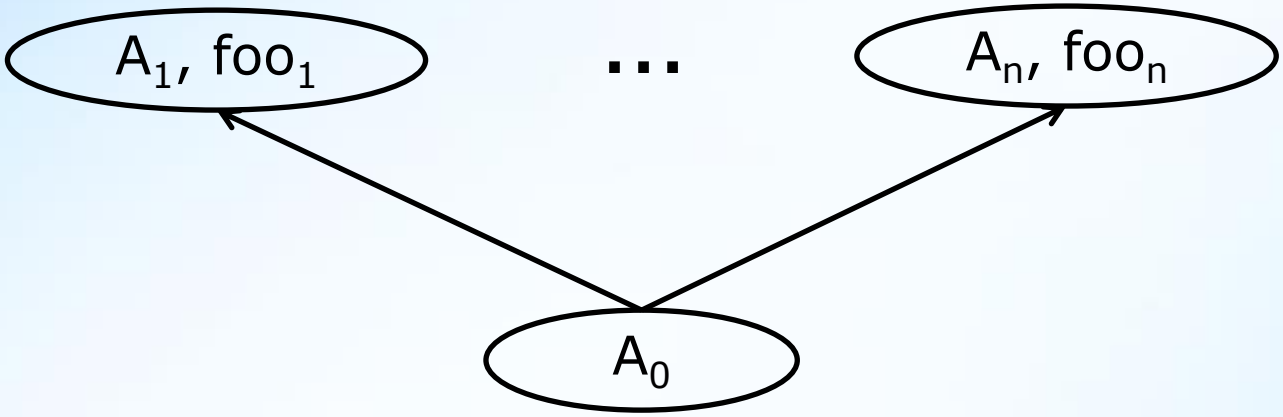
- S1 = S2
  - B1 = B2 – the same routine – OK!
  - B1 != B2 => ambiguity – on access compile time error!
- S1 != S2 – 2 different routines! To solve select case if they come the same origin and seed!
- S1 != S2 and override with S3→S1 and S3→S2 – OK!

2 attributes attr: T1 and attr: T2 inherited

- T1 = T2 – the same attribute – OK!
- T1 != T2- 2 different attributes! To solve select case if they come the same origin and seed!
- override attr : T3 – when T3 → T1 and T3 → T2 – OK!

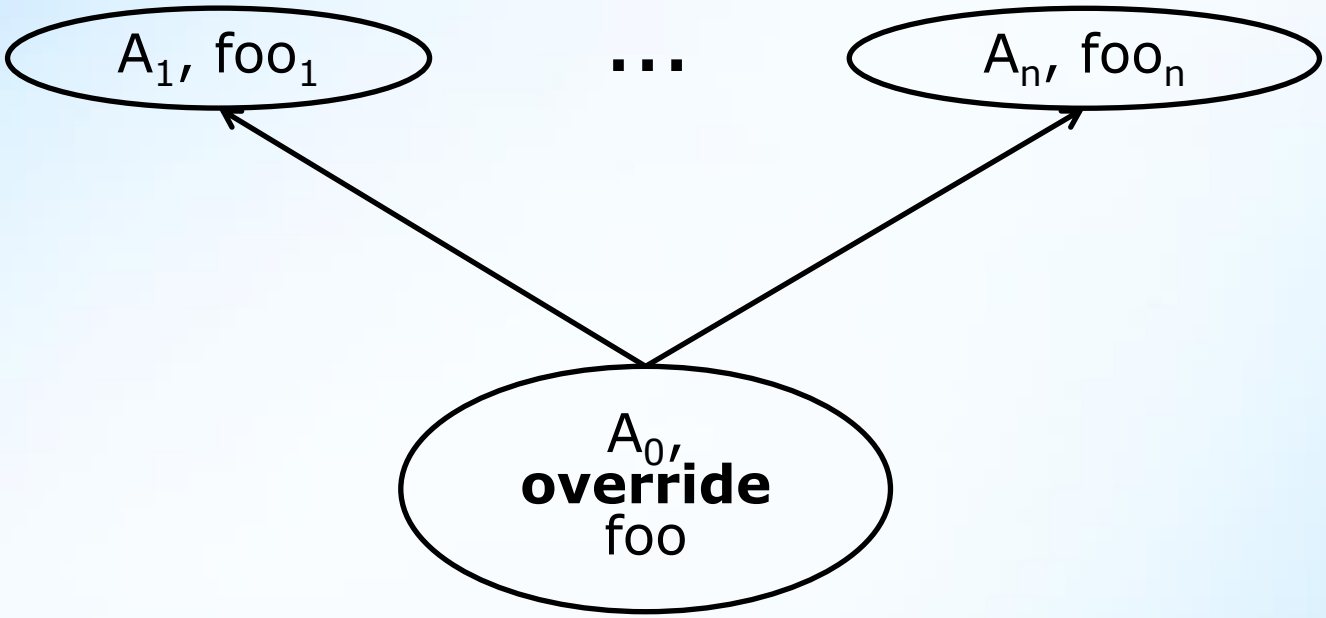
Routines and attributes are not much different while inheriting ☺

Inheritance: name clashes generalization I



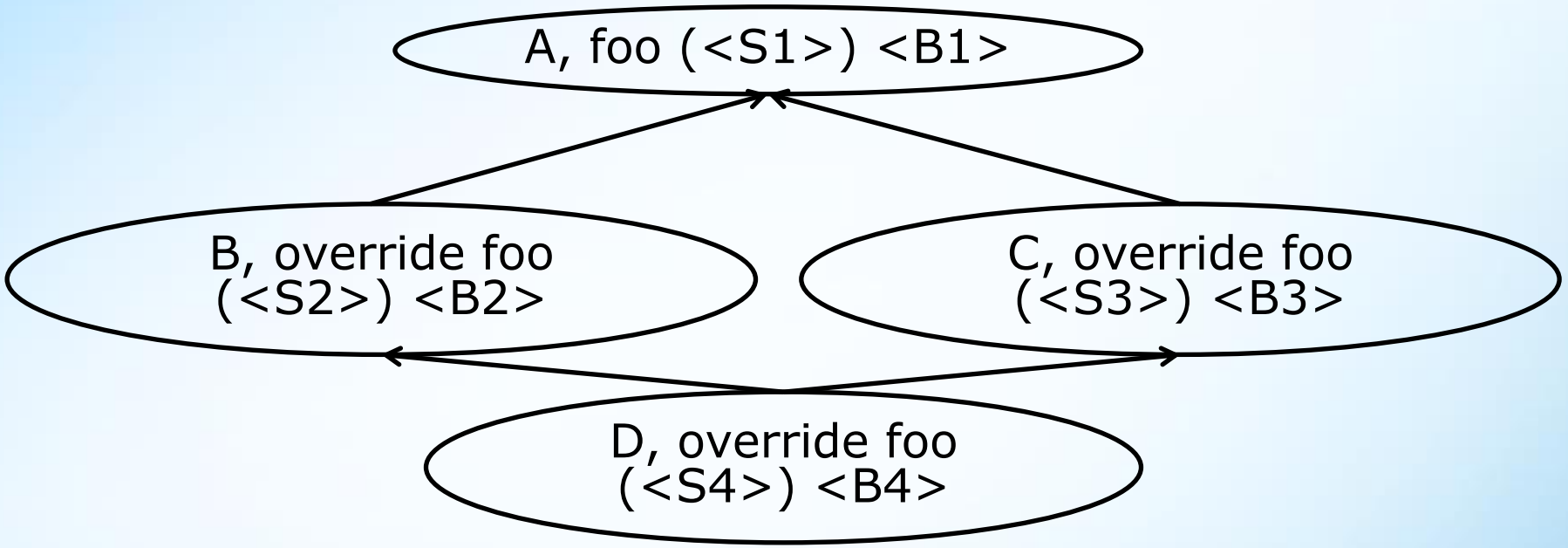
Let's consider A_0 features:
If A_0 states $A_1.foo_1$ **is abstract** then there must be one foo_i from A_0 which conforms to $A_1.foo_1$. In other words that is the mechanism how to kill a version while inheriting.
If A_0 states **override** $A_2.foo_2$ then it will override all versions from A_1 .. A_n to which it conforms to.
Then we have 2 sets (potentially empty)
foo_3 .. foo_k – the same feature – merged into one in A_0
foo_{k+1} .. foo_n – different overloaded features in A_0

Inheritance: name clashes generalization II



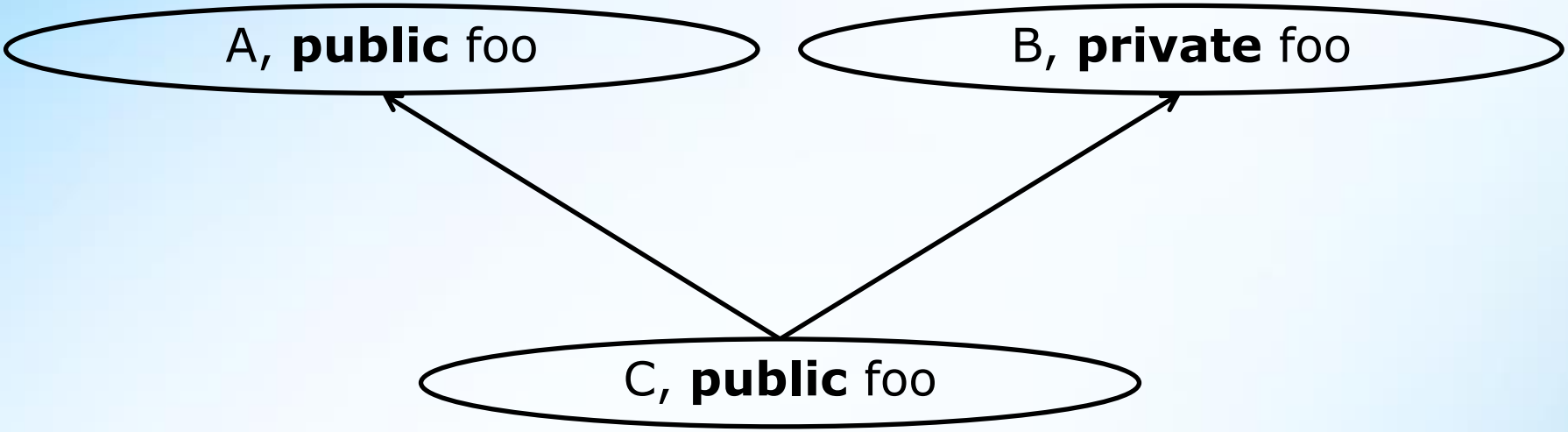
Let's consider A_0 features:
Some versions can be killed making them abstract. But for the remaining part we have:
2 sets (potentially empty)
foo_1 .. foo_k – overridden with foo from A_0
foo_{k+1} .. foo_n – different overloaded features in A_0

Inheritance: overloading: cat calls



S1 != S4, S2 → S1, S3 → S1, S4 → S2 & S4 → S3,
<exprType> → S4 (where → means conforms to)
a: A **is** D()
a.foo (<exprType>) // version from D must be called!
exprType → S1, but may not conform to S4 – that is a cat call! System wide check required.

Inheritance: visibility conflict



Regardless of the way how we keep one version of foo in C its visibility status must be public as at least in parent it was public

private … **private** => **public** or **private**

public … **private** = > **public**

Summary

Brief overview of existing approaches to inheritance given

Alternatives approach to inheritance presented, which combines power and generalization of multiple inheritance and at the same time avoiding complications of existing approaches.

- The key thing is whether compiler can resolve feature call or not! The unit may have arbitrary number of conflicting versions of the feature under one name.
- So, this inheritance may be called potentially conflicting inheritance with overloading and overriding.

Next steps is practical proof of concept ☺

THANK YOU!