OO languages late-binding signature

Antoine Beugnard
ENST Bretagne, BP 832, 29285 BREST CEDEX, FRANCE
Antoine.Beugnard@enst-bretagne.fr

Abstract

Most comparisons among OO languages focus on structural or philosophical features but rarely on dynamic ones. Beyond all these structural properties, late-binding is, to our opinion, the key property of OO paradigm; the operational consequence of inheritance use. All OO languages use late-binding, but do they all have the same interpretation? We show that the answer is no, not very surprisingly, but that almost each language has its own interpretation.

We propose a simple procedure to compare late-binding interpretation of OO languages and introduce a late-binding signature of OO programming languages. This procedure can be used to study language interactions as we will show it for the Microsoft .NET framework.

1 Introduction

Most comparisons among OO languages [Sei87, HZ93, SO91, ISE01, Bro97, Wol89] focus on structural or philosophical features but rarely on dynamic ones. For instance, comparison criterions are the ability to distinguish *types* and *classes*, to offer *single* or *multiple* inheritance, to accept or not *assertions*, to manage or not *exceptions*, to accept covariant redefinition or not, the nature of late-binding: simple or multiple, etc. Late-binding is, to our opinion, the key property of OO paradigm; the operational consequence of inheritance use. All OO languages use late-binding, but do they all have the same interpretation? To answer this question we propose a simple procedure that produces a table for each language and that can be considered as its signature. Moreover, this procedure can be used to study language interactions as we will show it for the Microsoft .NET framework.

The paper is organized as follows. Next section introduces the procedure to compare late-binding operational variants. Section 3 gives the results obtain with 9 different programming languages and section 4 the results obtain when languages interact via the Microsoft .NET framework. We conclude with perspectives of this work.

2 The test procedure

The comparison technique relies on a simple scenario. We first define a small package containing four classes. The Up class offers two services cv and ctv. Methods cv and ctv require one parameter each. Parameters are instances of classes Top, Middle or Bottom with the inheritance relationships Bottom ⟶ Middle ⟶ Top (where A ⟶ B means A is a subclass of B). The method body consists in printing out the class where it is defined (Up).

```
class Top
class Middle subclass of Top
class Bottom subclass of Middle

class Up
   method cv(Top t)
      print Up
   method ctv(Bottom b)
      print Up
```

Then we specialize class Up with a Down subclass

```
procedure main
– receiving objects
        Up u, ud;
        Down d;
– possible parameters
        Top t = new Top();
        Middle m = new Middle();
        Bottom b = new Bottom();
```

– First test	– Second test	– Third test
u := new Up();	d := new Down();	ud := new Down();
u.cv(t);	d.cv(t);	ud.cv(t);
u.cv(m);	d.cv(m);	ud.cv(m);
u.cv(b);	d.cv(b);	ud.cv(b);
u.ctv(t);	d.ctv(t);	ud.ctv(t);
u.ctv(m);	d.ctv(m);	ud.ctv(m);
u.ctv(b);	d.ctv(b);	ud.ctv(b);

Table 1: The three tests

that redefines the two services as follows:

```
class Down subclass of Up
   -- a covariant redefinition of cv
   method cv(Middle m)
      print Down
   -- a contravariant redefinition of ctv
   method ctv(Middle m)
      print Down
```

In order to observe the behavior of late-binding, a client calls all (18) possible parameter combinations as shown in table 1. Note that results of columns 2 and 3 are identical for languages that do not require object declaration.

In order to avoid any attempt on class or method name interpretation, and to concentrate on runs only, we have chosen names with only mnemonic connotations.

The scenario proposes both covariant and contravariant method redefinitions. Covariant redefinition means that the argument type varies in the same way as inheritance hierarchy, i.e. Down ⟶ Up and Middle ⟶ Top. Contravariant redefinition means that the argument varies in the opposite way, i.e. Down ⟶ Up and Middle ⟵ Bottom. A long controversy opposes computer scientists in order to decide what redefinition is the good one. Theorizers were in favor of contravariance since it is semantically sound and simple. Practitioners observe that concrete programs often use covariance. In [Cas95] G. Castagna unifies the two points of view showing that they could be used together for different purposes; the contravariance rule captures code substitutivity (always replace) while the covariant rule characterizes code specialization (replace in some special cases).

calls	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Down	Down

Table 2: An example of results

Another common OO semantics used is invariance. We could have added a method inv(Middle m) in Up and Down with exactly the same declaration in both classes. For the sake of brevity, we ignore this case in the following tests since all languages deal with it in the same way[1].

When neither covariance nor contravariance are accepted by a language, one uses method overloading, i.e. the capacity to use the same method name with different parameter types (signature). This approach is strongly criticized by B.Meyer [Mey97] who argues that if programmers want to change the signature of a service, it is much better to change the name of the service than to use the same name with a different type or number of arguments.

The result of a test consists of a 3x6 slots table. One column per receiver object (u, d, d declared as u). The content of the slot names the class where the code has actually been found. When a compilation error occurs the result is "Error" and when a runtime error occurs the result is "Run. Error". Table 2 shows an example of results.

Such a table shows the expected results for a lan-

---

[1] but for compilation error detection.

guage. It also gives some information on the compiler's features. For instance, slot (5,3)[2] triggers an error in table 2. The reason is that when calling ctv(m) we imagine the programmer expects to find only services declared in class Up, even if s/he knows that a more specialized object can actually be used. If an error is not detected, that means that the Up class and its clients should be recompiled each time a subclass redefines some of its methods. That means *it is impossible to build an incremental safe compiler.*

3 Single language signatures

Next tables (3 to 11) show results found with 9 popular OO languages where all parts of the scenario are programmed in the same language. We used the following languages: C++ [Str97], C# [Lib01], CLOS [Ste90], Dylan [Cra96], Eiffel [Mey92], Java [AGH00], OCaml [RV98], Smalltalk [GR83] and VisualBasic [Cor99]. We compiled the same program (in the syntax of each language) with gcc from Cygnus cygwin beta 20 and Microsoft Visual C++ 6.0 for C++, the GNU smalleiffel [CC01] and the Eiffel workbench 4.5 from ISE [Mey01] for Eiffel, the JDK1.3 from SUN for Java and the Squeak [IKM+97] for Smalltalk, and Visual Studio .NET beta 2 [Mic01] for C# and VisualBasic respectively.

The interesting point is that there almost all different! The case of Smalltalk (table 10) and OCaml (table 9) is interesting since they seem identical, but for more complex type relationships the OCaml compiler would reject some calls.

OO semantics does not have a single interpretation, so does OO really exists ? Facts show that the operational behavior of OO languages is defined by compilers designers with a limited understanding of consequences on OO programs. For instance, Java (table 8) rejects slot (6,2) while C++ (table 3) accepts it, and C++ rejects slot (1,2) while Java accepts it! Eiffel (table 7) rejects contravariant redefinition rules on principle. VisualBasic (table 11) prefers the most specialized parameter than the most specialized receiver on slot (6,2). OCaml rejects method overloading making impossible to mix methods found in Up and Down in column 3. Dylan, CLOS, Smalltalk, Eiffel (slot (1,3)) accept runtime errors.

calls	u	d	ud
cv(t)	Up	Error	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Down	Up

Table 3: C++ results

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Down	Up

Table 4: C# results

4 Language interaction

To get deeper in OO dynamic understanding we have used Microsoft .NET framework to make interlanguage cooperation tests. We have played the described scenario using the three languages Visual Stu-

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Run. Error	Run. Error	Run. Error
ctv(m)	Run. Error	Down	Down
ctv(b)	Up	Down	Down

Table 5: CLOS results

---

[2] Results are referenced by (line, column) in a [1..6]x[1..3] domain.

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Run. Error	Run. Error	Run. Error
ctv(m)	Run. Error	Down	Down
ctv(b)	Up	Down	Down

Table 6: Dylan results

calls	u	d	ud
cv(t)	Up	Error	Down
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Error	Error	Error
ctv(m)	Error	Error	Error
ctv(b)	Up	Up	Up

Table 7: Eiffel results

calls	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Error	Up

Table 8: Java results

appels	u	d	ud
cv(t)	Up	Down	Down
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Up	Down	Down
ctv(m)	Up	Down	Down
ctv(b)	Up	Down	Down

Table 9: OCaml results

calls	u	d	ud
cv(t)	Up	Down	Down
cv(m)	Up	Down	Down
cv(b)	Up	Down	Down
ctv(t)	Up	Down	Down
ctv(m)	Up	Down	Down
ctv(b)	Up	Down	Down

Table 10: Smalltalk/Squeak results

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Up	Up

Table 11: VisualBasic results

dio .NET offers (VisualBasic, C++ and C#). Structural interactions are resolved via the use of an intermediate language; method calls, inter-language inheritance, parameters transferts, data representation are efficiently treated. But, method lookup remains language dependent. Dynamic properties of languages are not well taken into account. Tables 12, 13 and 14 show results of the scenario where Up, Top, Middle and Bottom are programmed with C++, Down with C#, VisualBasic and C++ respectively, and the client with VisualBasic[3].

Column 2 is the most significant since all 3 are different, see slot (1,2) and (6,2). Columns 1 and 3 are identical since all tested languages use an invariant redefinition semantics. This means that the choice of a programming language to define the Down class is not neutral, or differently said, the Down component can not be replaced by another Down component programmed in another language without changing the global behavior.

---

[3] see http://perso-info.enst-bretagne.fr/˜ beugnard/papiers/lb-sem.html for all other results.

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	*Up*	Up

Table 12: VisualBasic, C#, C++ results

appels	u	d	ud
cv(t)	Up	Up	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Down	Up

Table 13: VisualBasic, VisualBasic, C++ results

appels	u	d	ud
cv(t)	Up	*Error*	Up
cv(m)	Up	Down	Up
cv(b)	Up	Down	Up
ctv(t)	Error	Error	Error
ctv(m)	Error	Down	Error
ctv(b)	Up	Down	Up

Table 14: VisualBasic, C++, C++ results

5 Conclusion

We have presented an original and pragmatic process to compare OO languages. The test could be improved by the association of a class specific method associated to each parameter class. Such improvement would detect safer compilers and show more runtime errors for the Eiffel and Smalltalk languages.

We propose here a kind of *language signature* represented by a 3x6 table. This signature reveals the operational behavior of a language and may be used to better understand language interaction. For instance, one can imagine an operator on signatures to forecast language interaction behavior.

Efforts made to unify OO approach like UML are facing a real problem. Should we accept all variants and define specialized version of UML (UML4java, UML4C++, etc.) or could we also define a unified late-binding semantics? We propose to adopt a unified signature (table 2 for instance proposes a "most specialized receiver choice") and to develop language transformation (to be defined) that will generate the selected behavior from the one implemented in the language.

We have defined a very pragmatic approach to get precise understanding of late-binding operational semantics. Tables enable to recognize languages as a signature does. To be used to better understand language interaction this approach need now to be formalized.

References

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[Bro97] Benjamin M. Brosgol. A comparison of the object-oriented features of ada 95 and java. *ACM*, pages 213–229, 1997.

[Cas95] Guiseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, March 1995.

[CC01] Dominique Colnet and Suzanne Collin. SmallEiffel the GNU eiffel compiler. http://www.loria.fr/projets/SmallEiffel/, 2001.

[Cor99] Microsoft Corporation. *Microsoft Mastering : Microsoft Visualbasic 6.0 Development (Dv-Dlt Mastering).* Microsoft Corporation, August, 1999.

[Cra96] Iain D. Craig. *Programming in Dylan.* Springer, 1996.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[HZ93] Robert Henderson and Benjamin Zorn. A comparison of object-oriented programming in four modern languages. Technical Report CU-CS-641-93, Department of Computer Science, University of Colorado, Boulder, Colorado, July, 1993.

[IKM$^+$97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, A practical smalltalk written in itself. In *Conference Proceedings of OOPSLA '97, Atlanta*, volume 32(10) of *ACM SIGPLAN Notices*, pages 318–326. ACM, October 1997.

[ISE01] ISE. Object-orient languages: A comparison, 2001. http://www.eiffel.com/doc/manuals/technology/oo_comparison/page.html.

[Lib01] Jesse Liberty. *Programming C#.* O'Reilly, 2001.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction, Second Edition.* Prentice Hall, 1997.

[Mey01] Bertrand Meyer. Interactive software engineering. http://eiffel.com/, 2001.

[Mic01] Microsoft. Visual studio .NET beta, 2001. http://msdn.microsoft.com/vstudio/nextgen/beta.asp.

[RV98] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[Sei87] Ed Seidewitz. Object-oriented programming in smalltalk and ADA. In *Object-Oriented Programming Systems, Languages and Applications*, pages 202–213, 1987.

[SO91] Heinz W. Schmidt and Stephen M. Omohundro. CLOS, eiffel and sather: A comparison. Technical Report TR-91-047, International Computer Science Institute, September, 1991.

[Ste90] G.L. Steele. *Common Lisp - The Language.* Digital Press, 1990.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

[Wol89] Wayne Wolf. A practical comparison of two object-oriented languages. *IEEE Software*, pages 61–68, September, 1989.