Mathematical Use Cases lead naturally to non-standard Inheritance Relationships: How to make them accessible in a mainstream language?

Marc Conrad[a], Tim French[a], Carsten Maple[a], Sandra Pott[b]

[a]University of Luton, LU1 3JU, UK
[b]University of York, YO10 5DD, UK

ABSTRACT

Conceptually there is a strong correspondence between Mathematical Reasoning and Object-Oriented techniques. We investigate how the ideas of Method Renaming, Dynamic Inheritance and Interclassing can be used to strengthen this relationship. A discussion is initiated concerning the feasibility of each of these features.

Keywords: Object Oriented Programming, Mathematics, Inheritance, Algebra

1. INTRODUCTION

A strong relationship has already been identified between inheritance relationships and algebraic structuring of "pure" mathematics.[1,2] This relationship is best explored in a highly dynamic object oriented programming language. In practice mathematicians in the course of their undergraduate studies might be exposed only (if at all!) to one of the mainstream languages, such as Java, C++, or perhaps in the near future to C#.

This paper proposes how a mainstream language could be "smoothly" extended to embody new inheritance relationships so as to make them accessible to a mathematical community. We discuss only Java for simplicity, however the proposed additions are, in principle, valid for any mainstream language that only supports inheritance in a manner similar to that of Java, such as C++ or C#.

The aim of the paper is to open a discussion about the feasibility of such extensions. We note that there are basically four possible courses of implementation: 1) Adding the features to Java/C++/C# themselves (that would imply negotiations with the responsible groups/companies); 2) Providing an "add-on" to the mainstream language (as a library or pre-processor); 3) Developing a new language that extends the existing mainstream language; 4) Developing a new language and educating users (universities, mathematics departments and so forth) so as to use this language for mathematical purposes.

2. OBJECT-ORIENTATION AND MATHEMATICS

In the field of Computer Algebra there are already packages that offer support for the object-oriented paradigm. For example, Axiom[3] has type hierarchies ordered in an inheritance-like structure and similarly Mupad[4] explicitly enables the defining of child classes of existing classes as groups, fields, etc.

The author's approach focuses on an object-oriented implementation of mathematical structures in an axiomatic manner.[2,5] The philosophy therein is that postulated properties of a domain are reflected as abstract methods. For example an algebraic ring *has* by definition addition and multiplication. Of course, addition and multiplication are not known *algorithmically* for an arbitrary (unspecified) ring: they cannot be implemented. Therefore the mathematical entity "algebraic ring" is implemented as an abstract base class. This design follows the GoF[6] mediator pattern : The abstract ring class is an abstract *mediator* whilst the elements of the ring are the mediated colleagues.

Email: Marc.Conrad@luton.ac.uk, Tim.French@luton.ac.uk, Carsten.Maple@luton.ac.uk, sp23@york.ac.uk

3. NON-STANDARD INHERITANCE

In this section we address Method Renaming, Dynamic Inheritance, and Interclassing. Each of these three subsections is divided into the presentation of the mathematical Use Cases together with a discussion. For all three features we provide two example Use Cases: An elementary (rather basic) example to introduce and cement the ideas and a more advanced example to demonstrate the power of the method.

3.1. Overriding with Renaming

3.1.1. Mathematical Use Cases

A group is a set with an operation and certain properties. In a concrete situation there is often a standard notation for the group operation. The most familiar are $+$ for addition in an additive group and $*$ or $\times$ for multiplication in a multiplicative group.

Similarly the *composition* of two endomorphisms in the ring of endomorphisms over a vector space becomes matrix *multiplication* in the special case of vector spaces of finite dimension.

3.1.2. Discussion

In these two examples it becomes clear that the renaming of a certain operation after specialization is a familiar task in mathematics. Especially the second example where composition becomes multiplication shows that renaming is able to reflect nontrivial mathematical relationships. This is even more evident in a language that supports operator overloading (as C++): The group operation $a \circ b$ is renamed in a concrete application as either $a \cdot b$ or $a + b$. A well known example can be found in cryptography: A public key/private key algorithm can be formulated for a certain class of abelian, finite groups. The two kinds of groups that are used in practice are $(\mathbf{Z}/n\mathbf{Z})^*$ (a *multiplicative* group) and elliptic curves (*additive* groups). A generic approach[7] dealing with both kinds of groups needs to be supported by a renaming mechanism.

The renaming of an operation after it has been overridden is hardly a new feature in object-oriented contexts. In Eiffel renaming is the preferred method of choice to avoid ambiguity in multiple inheritance relationships.[8] Renaming also exists as a standard feature in Python.[9] Adding this concept to Java would be an easy step to improve the usability of Java within "mathematical context". For example, C# already provides an *overrides* keyword and from here it would be a comparatively small step to extend this syntax by specifying *what* it is that is overridden.

3.2. Dynamic Inheritance

3.2.1. Mathematical Use Cases

Assume we start with an inheritance relationship with *Field* as a child class of *Ring*. For some rings, for example $\mathbf{Z}/n\mathbf{Z}$, it cannot be decided by a compiler a priori if it is a field or not. ($\mathbf{Z}/n\mathbf{Z}$ is a ring if and only if n is a prime number).

In algebraic ring theory we have an even more extended inheritance hierarchy with (for example) *Euclidian Ring*, *Noetherian Ring*, *Principal Ideal Ring* as classes located between *(commutative) Ring* and *Field*. For instance if we restrict consideration to only the class of quadratic orders $\mathbf{Z}[\sqrt{d}]$ with $d \in \mathbf{N}$ we find Euclidian rings and Principal Ideal Rings for various values of d.[10]

3.2.2. Discussion

Dynamic Inheritance is hardly a "new" feature. A C++ implementation (or rather a workaround) is already discussed in.[11] Related to this is the concept of *predicate classes*[12] that is implemented in Cecil.[13] In Self,[14] the object itself can decide on its parent objects thereby giving maximum flexibility. Kniesel[15] proposes a Java extension featuring Dynamic Inheritance. Dynamic Inheritance is also supported in Lava as part of the Darwin project.[16]

The most appropriate approach may well be reclassification as introduced in *Fickle*[17]: An object is related to a *Root Class* (in the Use Cases the class *Ring*). Then it can be reclassified to each child class (called *State*

Classes) of this Root class. A special operator *!!* in *Fickle* reclassifies an object from one State Class to another when both belong to the same Root Class.[17]

The translation of Java into *Fickle* described in[18] may serve as a roadmap for an implementation of reclassification in Java (although it is not straightforward). For a code example that illustrates the proposed syntax of Java reclassification see.[19]

3.3. Interclassing

3.3.1. Mathematical Use Cases

Assume for the moment that Euclid is a contemporary mathematician who has just discovered Euclidian division (also known as division with remainder), and that he wants to add Euclidian division into existing mathematical software that features a ring/field implementation as described in the previous section. The proper place for a Euclidian Ring – a ring with Euclidian division – is between the ring and field. Not every ring is Euclidian and every field is trivially a Euclidian ring.[20]

A typical, non-fictional, example taken from Functional Analysis is that of Triebel-Lizorkin spaces, which were introduced in the 1970's as simultaneous generalizations of a number of well-known classes of function spaces, such as L^p spaces, Hardy spaces, the space of functions of bounded mean oscillation (BMO), Lipschitz spaces and Sobolev spaces.[21] A Triebel-Lizorkin space is a specialization of a Banach function space. A more recent example is that of so-called real Q-spaces, which are simultaneous generalizations of the space BMO and certain other Banach function spaces.[22]

This "interclassing" in Mathematics is often motivated by the desire to create a unifying framework for several known classes of mathematical objects in a certain context (as in the first of the two examples mentioned above), or to bring existing mathematical techniques to new applications (the second example).

3.3.2. Discussion

Note that the problem of interclassing is substantially different from the problem of run-time reclassification described earlier in section 3.2. Here, we start with a class hierarchy that may be arranged in a package and that may not even contain any source code. We want to extend this class hierarchy by adding a class on a well defined position in an inheritance tree. Even if the source code is available it may not be desirable to change this code, especially if the class library is well established and the addition of the new class has an *experimental* character, or is only relevant for a specialized application area.

Outside of a mathematical context, the idea of *interclassing* is already discussed by Rapicault and Napoli.[23] Crescenzo and Lahire[24] describe an implementation using the OFL model. However, in terms of pragmatic usage OFL is inadequate as it requires *de facto* the learning of OFL as an additional language, namely the understanding of the correct use of hyper-generic parameters. Also, in using hyper-generic parameters, the developer of a library already unnecessarily restricts possible extensions.

A concept developed in LPC[25] called "shadowing" may be a useful technique for the implementation of interclassing. Essentially a shadow is a proxy-object that can be added at run-time and receives all messages determined to the shadowed object (hence "shadowing" the "proxied" object). The concept is evaluated in more detail in two preprints by the authors[19, 26] and has been implemented in Java.[27]

4. CONCLUSION AND PERSPECTIVE

This paper describes work in progress. However a simple (in terms of usability) incorporation of Method Renaming, Dynamic Inheritance, and Interclassing in a mainstream language would radically simplify the implementation of mathematical structures in a wide range of use cases, even in areas that are currently merely considered as practically not accessible by programming (such as Functional Analysis).

The paper was motivated by disappointment with the traditional way of implementing "mathematics" within mainstream Computer Algebra Systems and experiments using the com.perisic.ring Java package.[5] However, it seems that the question of "implementing" mathematical structures in an object oriented context is strongly linked to (and may be dominated by) the issue of how best to represent these structures.

Moreover, it may be fruitful to discuss the problems of Method Renaming, Dynamic Inheritance and Interclassing independently from any implementation language in the context of the UML. Having appreciated the usefulness of a UML representation of mathematical structures, the use cases provided in this paper may lead to future examination of suitable extensions to UML, for instance how Reclassification should be modelled in a Sequence Diagram etc.

REFERENCES

1. M. Conrad, *Abstract Classes - pure computer science meets pure mathematics*, Seminar talk, York 2003, http://ring.perisic.com/info/york2003.
2. M. Conrad, T. French, "Exploring the synergies between the Object-Oriented paradigm and Mathematics: a Java led approach," to appear in *Int. J. Math. Educ. Sci. Technol.*
3. Tim Daly et. al. *Axiom Computer Algebra System*, http://savannah.nongnu.org/projects/axiom.
4. The MuPAD Research Group. *MuPAD – The Open Computer Algebra System,* http://www.mupad.de.
5. M. Conrad. *com.perisic.ring – A Java package for multivariate polynomials*, http://ring.perisic.com.
6. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns,* Addison-Wesley, 1995.
7. C. Maple, M. Conrad, T. French, "A Novel Flexible Approach to Document Encryption Using a MathML Extension to the W3C XML Digital Certificate Standard," in *Proceedings of the IADIS International Conference on e-Society* (2003)
8. Betrand Meyer, "Overloading vs. Object Methodology," *Journal of Object-Oriented Programming*, October/November 2001.
9. Jeremy Hylton, *Introduction to Object-Oriented Programming in Python (Outline),* http://www.python.org/j̃eremy/tutorial/outline.html, Januar 2000.
10. Kenneth Ireland, Michael Rosen, *A Classical Introduction to Modern Number Theory, 2nd ed.* Springer-Verlag, New York, 1995
11. James Coplien, *Advanced C++ programming styles and idioms*, Addison-Wesley 1992.
12. C. Chambers, "Predicate classes," in: *Proceedings of the ECOOP'93*, volume 707 of *Lecture Notes in Computer Science*, pages 268–296, Kaiserslautern, Germany, July 1993.
13. C. Chambers, *The Cecil Language: Specification & Rationale*, avialable at: http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.
14. The Self Group, *Self*, http://research.sun.com/research/self
15. Günter Kniesel, *Darwin & Lava - Object-based Dynamic Inheritance ... in Java*, Poster presentation at ECOOP 2002.
16. *The Darwin Project*, http://javalab.iai.uni-bonn.de/research/darwin.
17. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini and P. Giannini, "Fickle: Dynamic object reclassification," in: *ECOOP'01*, LNCS **2072** (2001), pp. 130–149.
18. D. Acnona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, "A type preserving translation of Fickle into Java," in: *Electronic Notes in Theoretical Computer Science* 62 (2001). Available at: http://www.elsevier.nl/locate/entcs/volume62.html
19. M. Conrad, T. French, C. Maple, S. Pott, *Approaching Inheritance from a "Natural" Mathematical Perspective and from a Java driven viewpoint: a Comparative Review*, Preprint available from: http://ring.perisic.com.
20. Serge Lang, *Algebra*, third ed., Addison-Wesley, 1993.
21. H. Triebel, "Theory of Function Spaces," *Monographs in Mathematics*, vol 78, Birkhäuser Verlag Basel, 1983
22. M. Essén, S Janson, L. Peng and J. Xiao, "Q-spaces of several real variables," *Indiana University Mathematics Journal*, vol 49, no 2(2000), 575 – 615
23. P. Rapicault, A. Napoli, "Evolution d'une hirarchie de classes par interclassement," in: *LMO'2001, Hermes Sc. Pub. "L'objet"*, vol. 7 - no. 1–2/2001
24. Pierre Crescenzo, Philippe Lahire, "Using Both Specialisation and Generalisation in a Programming Language: Why and How?" In: *Advances in Object-Oriented Information Sytems*, OOIS 2002 Workshops, Montpellier, pages 64–73, September 2002.

25. Ronny Wikh, *LPC*, available at: http://genesis.cs.chalmers.se/coding/lpcdoc/lpc.html (last update 2003)
26. M. Conrad, T. French, C. Maple, "Object Shadowing - a Key Concept for a Modern Programming Language," Submission to the *2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity* (Workshop 5 of ECOOP 2004).
27. M. Conrad. *The com.perisic.shadow package*, http://perisic.com/shadow.