Where are the Laws of Object-Oriented Programming?

Paulo Borba*
Departamento de Informática
Universidade Federal de Pernambuco

Abstract

In this position paper we overview on-going research work aimed at finding and formalizing laws of object-oriented programming. Rather than formal development, our main interest is to guide, justify and document informal object-oriented programming practices. In particular, we focus on laws that support software evolution practices such as introducing common analysis and design patterns. Moreover, contrasting with previous work on programming laws, we investigate both *universal* and *method-specific* laws.

Keywords: Formal Methods, Object-Oriented Programming, Refinement Calculus, Laws of Programming, Programming Language Semantics.

1 Introduction

The laws of imperative programming are well established and have been useful both for assisting software development and for providing precise axiomatic programming language semantic definitions [14, 17]. In fact, besides being used as guidelines to informal programming practices, programming laws establish a sound basis for formal and rigorous software development methods. Moreover, axiomatic semantic definitions are an important tool for the design of correct compilers and code optimizers [22].

Similarly, the laws of package[1] composition [10] are well established and useful both for reasoning about the structure of large systems and for justifying the implementation of powerful package languages. Whereas imperative programming laws determine the properties of small grain constructs and units such as conditionals, sequential composition, loops, and procedures, the laws of package composition determine the properties of large grain (parameterized) units and operators for importing, renaming, and instantiating them.

*Supported in part by CNPq, grant 521994/96–9. Email: `phmb@di.ufpe.br`. WWW: `http://www.di.ufpe.br/~phmb`.

[1]We use the word 'package', instead of 'module', in order to avoid confusion with smaller grain programming units such as classes and procedures, which some authors consider to be modules too [23].

But where are the laws of object-oriented programming? Some laws have been informally described on the object-oriented literature [15, 1, 20], but most of them are still on the minds of object-oriented programmers which intuitively apply them everyday. In fact, the laws of object-oriented programming are not well established yet. So, although object-oriented programming is widely used nowadays, there is no comprehensive set of laws to help developers understand and use the properties of medium grain programming units and mechanisms such as classes, inheritance, and subtyping. Furthermore, some of the laws of imperative programming, which deal with small grain units and constructs, are not directly applicable to similar object-oriented units and constructs. For example, due to dynamic binding [16], the laws of procedure call are not valid for method call.

In this position paper we overview on-going research work aimed at finding and formalizing laws of object-oriented programming. The formalization of those laws is essential for determining the situations when the laws may be applied, and for precisely indicating code modifications resulting from the application of a given law. However, rather than formal development, our main interest is to guide, justify, and document informal object-oriented programming practices. In particular, we focus on laws that support software evolution (maintenance) practices; for example, we consider laws for safely introducing common architecture, analysis and design patterns, assisting the transition from anti-patterns to patterns [11, 7].

Contrasting with previous work on programming laws [14, 17, 10], we investigate both *universal* and *method-specific* laws. Universal laws are useful for any system, whereas method-specific laws are only useful for systems developed by following a specific programming method. Indeed, as requested by Hoare and others [14], method-specific laws determine deeper and more specific properties, which can be used on a restricted but large range of situations.

This paper is organized as follows. We first discuss some aspects of mathematical frameworks being developed to validate the laws. Following that, we present class restructuring laws and explain how they may help to introduce design patterns and improve modularity. We use a small subset of Java [13] for illustrating the laws, but anyone with a basic knowledge of object-oriented programming (and formal methods) shall be able to understand this paper. In Section 4 we discuss synchronization laws for increasing either safety or liveness in concurrent object-oriented programs. After that we give some method-specific laws for supporting the progressive implementation method [5]. At the end there are some concluding remarks.

2 Refinement of Object-Oriented Programs

In order to formalize and validate the laws of a programming or specification language, we must precisely define the syntax and semantics of that language. Furthermore, we must provide the language with a formal notion of refinement establishing which programs are 'better' than others[2] [14, 17]. The programming laws shall reflect the

[2]Strictly speaking, we could alternatively simply have a notion of equivalence between programs. The meaning of 'better' and 'equivalent' will naturally depend on the behavior that one wishes to observe from the execution of programs.

properties of this notion so that we can also call them 'refinement laws'. In fact, the laws must be proved correct with respect to refinement.

Towards defining semantics, refinement, and universal laws for sequential object-oriented programming, a joint CNPq and NSF project has been proposed [19]. In fact, the laws in Section 3 are in the scope of that project, and the denotational semantic frameworks resulting from the project will be useful to validate the method-specific laws presented in Section 5. However, the synchronization laws (see Section 4) demand a semantic framework that deals with concurrency; so they will be validated using an operational framework for concurrent object-oriented programming [2, 6, 3, 4].

The denotational frameworks mentioned above will be developed by extending imperative programming frameworks [17, 21, 18] in two ways:

- adapting semantics and the standard notions of algorithmic and data refinement to consider additional language features such as subclass polymorphism, object references (pointers), and type test and guard; and
- defining a notion of refinement for classes.

In particular, refinement of classes will be defined in terms of the adapted notions and we shall have specific theorems (laws) relating them.

A major issue in adapting refinement to object-oriented development is due to subclass polymorphism. For example, consider a standard class `Stack` implementing integer stacks and a variable `s` of that type. We might expect to have

$$\texttt{s.push(5) ; s.pop()} \sqsubseteq \texttt{skip}$$

but this actually does not hold in general because `s` might store objects of any subclass of `Stack`; for example, that would not hold for objects of a subclass `UndoStack` having an additional `undo` method for undoing the last pop operation.

Our solution for the problem above is to have "typed" refinement relations [6] such as

$$\texttt{s.push(5) ; s.pop()} \sqsubseteq_{\texttt{Stack}} \texttt{skip}$$

indicating a signature of visible operations for which the behavior of `skip` cannot be distinguished from the behavior of '`s.push(5) ; s.pop()`'. That would hold for objects of `UndoStack` for two reasons: we are implicitly saying that the `undo` method should not be considered; and we assume behavioral subtyping in the sense that redefined methods of `UndoStack` should preserve the semantics of the original `Stack` methods.

As a consequence of having "typed" relations, and also because of the support for hidden methods and attributes (instance variables), we should only expect a weaker monotonicity (congruence) property for object-oriented development [2, 3]. For instance, for commands $\texttt{c}$ and $\texttt{c}'$,

$$\texttt{c} \sqsubseteq_{\texttt{Stack}} \texttt{c}'$$

implies

$$\texttt{P[c]} \sqsubseteq_{\texttt{Stack}} \texttt{P[c}'\texttt{]},$$

for any program context `P` formed only by the visible operations of `Stack`. Indeed, contexts formed by `undo` or `elems[index + 1]` (for an implementation of stacks with an array `elems` and an `index`) would not preserve the refinement

`s.push(5) ; s.pop()` $\sqsubseteq_{\texttt{Stack}}$ `skip`.

Similarly, we should also have "typed" relations for refinement of classes:

`Stack` $\sqsubseteq_{\texttt{Stack}}$ `UndoStack`,

indicating that `Stack` can be replaced by `UndoStack` in any context formed only by the visible operations of the first. That would actually establish that `UndoStack` is a behavioral subclass of `Stack` (objects of `UndoStack` preserve all properties of `Stack` objects, if the visible operations only of `Stack` are considered).

3 Class Restructuring Laws

Laws that justify class restructuring practices shall be extremely useful for software evolution, when programmers are interested on restructuring code in order to achieve modularity and, consequently, improve software reuse and extensibility [16]. Therefore, we shall certainly provide medium grain laws that justify informal and common restructuring practices such as splitting a class like

```
class Person {
   private String name;
   private String street;
   private String city;
   void m() {
     street = exp; ...
     x = ...street...; ...
   }
   Person (String nm, ...) { this.name = nm; ... }
   ⋮
}
```

into two other classes such as

```
class ModularPerson {
   private String name;
   private Address address;
   void m() {
     address.setStreet(exp); ...
     x = ...address.getStreet()...; ...
   }
   ModularPerson (String nm, ...) {
     address = new Address();
     this.name = nm; ...
   }
   ⋮
}
```

which corresponds to Person but where direct access to the instance variables street and city are replaced by corresponding set and get method calls using the address attribute, and

```
class Address {
   private String street;
   private String city;
   void setStreet(String s) { this.street = s; }
   String getStreet() { return this.street; }
   ⋮
}
```

which simply encapsulates the attributes of Person that are associated to the address concept.

Such class restructuring practices are sound only if certain constraints are satisfied. And revealing those usually subtle constraints is one of the main benefits gained from the formalization and validation of the laws. In fact, the illustrated class split practice is only valid if the introduced attribute, address in our example, stores only unaliased objects. Otherwise, for example, the street of a given person (object of ModularPerson) could be changed by invoking the method setStreet of another person, given that they may share the same Address object. This misbehavior is certainly not possible for objects of Person, and that is why the practice is not valid in general.

The class ModularPerson satisfies the class split constraint because the instance variable address is set to a new object at initialization time, and the remaining code assigns no value to it. Alternatively, we could have used promises [8]—special purpose annotations that can be automatically checked for validity—to document that constraint: address would be qualified as unique, indicating that it stores only unaliased object references. In this case, the initial address of a person could be provided at initialization time, and a tool could check the uniqueness of Address objects provided at the creation of ModularPerson objects.

3.1 Formalization

A concise way to formalize and represent the laws is achieved by the use of class operators for building classes from smaller pieces, as if we were defining an algebra for classes. For example, the '$\langle \bullet \quad \bullet \rangle$' operator constructs classes so that

$$\langle A \bullet M \bullet I \rangle$$

denotes the class formed by the private attributes[3] in A, the methods in M, and the initializers (constructors) in I. The '$\otimes$' operator puts attributes together, so the expression

$$\langle A \otimes B \bullet M \bullet I \rangle$$

[3]Hereafter we consider that classes have only private attributes; so we omit the word 'private' for brevity.

denotes the class formed by the attributes in A and in B, besides the methods and initializers respectively in M and I.

Using those operators, the restructuring practice previously discussed in this section could be precisely defined by indicating that the class above is refined by the following class:

$$\langle A \otimes c \bullet M_c^B \bullet I_c^B \lhd \texttt{c=new C();} \rangle$$

where c is a new attribute and

- M_c^B, similar to I_c^B, denotes the methods obtained from M by replacing direct accesses to the variables in B by indirect accesses via c (using `set` and `get` methods);
- $I \lhd$ `code` denotes the initializers obtained by inserting `code` at the beginning of each initializer in I[4];
- $name(c)$ and $type(c)$ are respectively `c` and `C`, which is the name of the class

  $$\langle B \bullet \texttt{set}^B \oplus \texttt{get}^B \bullet \emptyset_B \rangle$$

  basically formed by methods `get` and `set` for each attribute in B, and the default initializer for the attributes of B [13].

More formally, this law should be stated in the following way:

$$\langle A \otimes B \bullet M \bullet I \rangle \sqsubseteq_{\Sigma(M,I)} \langle A \otimes c \bullet M_c^B \bullet I_c^B \lhd \texttt{c=new C();} \rangle$$

where $\Sigma(M, I)$ denotes the signatures of the operations in M and I[5].

Surprisingly, this law turns out to be also useful for justifying the use of the Item-Item description analysis pattern [9]—which indicates how to model objects and their descriptions—for improving modularity. This shows that the class restructuring laws may be used for guiding the introduction of patterns, assisting the transition from anti-patterns to patterns [11, 7]. In fact, we are interested on laws that specify how a bad analysis or design can be refined to a well known effective solution. We are also studying laws that indicate how a design or programming mistake can be removed without affecting the behavior of that specification or program, and assuring that software quality factors are improved.

Other useful class composition and decomposition laws can be formalized in the same way. For example, consider methods m and m' having the same signature. We have that

$$\langle A \bullet M \oplus m \bullet I \rangle \sqsubseteq \langle A \bullet M \oplus m' \bullet I \rangle$$

provided that $m \sqsubseteq m'$. This law basically indicates that method refinement implies class refinement. Similarly, we have that the refinement of a 'server class' implies the refinement of a 'client class':

$$\langle A \otimes a \bullet M \bullet I \rangle \sqsubseteq \langle A \otimes a' \bullet M \bullet I \rangle$$

provided that $name(a) = name(a')$ and $type(a) \sqsubseteq type(a')$.

[4]Strictly speaking, `code` should be inserted right after the `super` call of initializers that refer to superclass initializers [13].

[5]Hereafter we use $\langle A \bullet M \bullet I \rangle \sqsubseteq \langle A' \bullet M' \bullet I' \rangle$ for $\langle A \bullet M \bullet I \rangle \sqsubseteq_{\Sigma(M,I)} \langle A' \bullet M' \bullet I' \rangle$.

4 Synchronization Laws

It is extremely difficult to develop and maintain concurrent programs because their behavior cannot be easily predicted. Moreover, testing, which is not so effective for assuring correctness of sequential programs, is even less effective for concurrent programs, since those are often highly nondeterministic due to the vast range of possible interferences that may occur [4].

Thus, in order to deal with concurrent programs, it is important to follow practical guidelines for safely introducing, moving, and removing synchronization mechanisms without leading to deadlock or to undesirable interference. Such guidelines can, for example, be derived from common design patterns for structuring concurrent object-oriented programs [15]. Synchronization laws formalize those guidelines and, therefore, may assist programmers to apply design patterns to an otherwise badly designed concurrent program.

Each synchronization law will typically help to correctly increase either liveness or safety. For example, a useful law for increasing liveness establishes that we can remove the Java `synchronized` qualifier [13] from a method as long as some constraints are satisfied:

$$\langle A \otimes b \bullet M \oplus sync(m) \bullet I \rangle \sqsubseteq \langle A \otimes b \bullet M \oplus m \bullet I \rangle$$

provided that

- $body(m)$ is in the form '$\mathtt{b.n}(\overline{\mathtt{e}})$', where $name(b)$ is `b`, `n` is any method name, and $\overline{\mathtt{e}}$ is an arbitrary list of expressions formed only by variables and constants[6]; and
- $type(b)$ is a fully synchronized class—its methods are all synchronized, and it has no public instance variables [15].

Roughly, this law establishes that it is sound to remove synchronization from a method m that simply invokes another method that is always atomically executed. The justification is that m will be atomically executed anyway.

Although the provisos above are quite restrictive, that law may be useful for refining a surprisingly large range of systems. In fact, any system structured using the Facade design pattern [11]—which provides a unified interface for all services of a system—would benefit from that law for justifying that a main *business (application) server class* such as

```
class MyBusiness {
   private CustomerFile customers;
   private ProductList products;
   synchronized void add_customer(Customer c) { customers.add(c); }
   synchronized void setPrice(ProductCode code, double price) {
      products.setPrice(code,price);
   }
   ⋮
}
```

[6]We assume that method bodies are well typed, so that we do not need to impose further constraints on `b`, `n`, and $\overline{\mathtt{e}}$.

can be optimized by removing the `synchronized` qualifier from the methods above, assuming that the classes `CustomerFile` and `ProductList` serialize the insertion, updating, querying, and deletion of customer and product records.

We believe that synchronization laws like the one discussed in this section can be very helpful for the systematic development of practical concurrent Java programs, making it easier to bridge the gap between sequential and concurrent Java programming [4].

5 Progressive Implementation Laws

Synchronization and class restructuring laws are universal in the sense that they might be useful for any system. On the other hand, method-specific laws may only be used for systems developed by following a specific programming method. Those laws determine deeper and more specific properties, which rely on constraints implicitly satisfied by the method. So method-specific laws are simpler to apply than corresponding universal laws. And, although they have limited applicability, they can be used on a large range of situations.

In this section we consider a method-specific law for supporting the progressive implementation method (Pim) [5], which proposes patterns for structuring object-oriented client-server and distributed systems. In particular, those patterns are useful for supporting the progressive introduction of distribution, concurrency, and persistence. By initially abstracting from those subtle aspects, programmers can quickly develop a non-trivial, monolithic, sequential, non-persistent system prototype useful for capturing and validating user requirements. As requirements become more well understood, this prototype can be progressively transformed into a complete version of the system, by carefully dealing with the aspects mentioned above, one at a time.

Before presenting the method-specific law, we shall give some more details about the method. Pim proposes the use of the Facade pattern as in Section 4, and determines that main *business collection classes* such as `ProductList` should be initially implemented in the following way[7]:

```
class ProductList {
   private ProductCollection productlist;
   synchronized void setPrice(ProductCode code, double price) {
      Product p = productlist.search(code);
      p.setPrice(price);
   }
   ProductList (ProductCollection prdlist) {
      this.productlist = prdlist;
   }
   ⋮
}
```

where `ProductCollection` is a Java interface [13] containing the signature of typical operations offered by collections. This approach guarantees a certain independence between the business and data layers [1], so that the products in a list may be stored

[7] For simplicity, we omit the details about the exceptions that may be raised by several methods.

by several data structures—*data collection classes* implementing `ProductCollection`—without impacting clients of `ProductList`.

However, in order to keep product lists on persistent storage (such as relational and object-oriented databases), the class `ProductList` should be slightly changed to update the data collection whenever products are updated[8]. For instance, the body of `setPrice` should be modified to

```
Product p = productlist.search(code);
p.setPrice(price);
productlist.update(p);
```

which guarantees that changes to a run-time object are reflected on the persistent storage too.

This kind of practice is necessary to support Pim and there should exist laws to guide and justify it. In particular, the following law establishes that a transient data collection may be replaced for a persitent one, as long as some modifications are made to `ProductList`[9]:

$$PL \dashv \texttt{new ProductTransColl()} \sqsubseteq_{Sig} \mathit{update}(PL) \dashv \texttt{new ProductPersColl()}$$

where $\mathit{update}(PL)$ represents the class obtained from `ProductList` by making the modifications illustrated above, `ProductTransColl` and `ProductPersColl` are respectively the names of proper transient and persistent implementations of `ProductCollection`, and Sig is the signature of $\mathit{update}(PL)$ plus typical database operations for opening, commiting and rolling back transactions. The notation '$C \dashv c$' denotes a command c that is supposed to be executed on a program context that includes the class C. We have been using so far '$c \sqsubseteq c'$' to denote '$C \dashv c \sqsubseteq C \dashv c'$'.

By saying that a law is only valid for systems developed following Pim, we may rely on several constraints that are necessary for the soundness of the law. In particular, for the example above, we rely on constraints like the following:

- the methods of `ProductCollection` are only invoked inside `ProductList`;
- variables and parameters might be declared to have the type `ProductCollection`, but not the type of any implementation of this interface; and
- `ProductList` does not have methods that, directly or indirectly, return references to objects of `Product`.

Those constraints would otherwise have to be verified for each application of the law. So, besides being quite focused, another advantage of method-specific laws is to abstract from tedious constraints that are implicitly implied by the proper use of a programming method.

[8] For the sake of efficiency, that usually cannot be automatically done by implementations of `ProductCollection` using databases or files.

[9] Of course, this law can be easily generalized for `Customer` or any other class whose objects are stored on main business collection classes.

6 Conclusion

This position paper overviews on-going research work aimed at finding and formalizing laws of object-oriented programming. In particular, we hope that it is now clear how laws of object-oriented programming look like. Many others laws are being considered, but they have not been formally proved yet since the necessary mathematical frameworks are still being developed or adapted.

The formalization of nontrivial laws is essential for revealing subtle details that are usually only clear to very experienced programmers. By revealing those details we provide important assistance for software development and evolution. In particular, we have shown that the synchronization and class restructuring laws are powerful tools for allowing the widespread use of analysis and design patterns during software evolution as well.

We have concentrated here on laws useful for refining programs in order to increase one of the following aspects: efficiency, modularity (and consequently extensibility and reusability), liveness, safety, and persistence. This is a consequence of our interest on supporting informal software evolution practices, rather than formal development. So we have not considered laws for refining specifications or for deriving code from specifications [17]. These kinds of law are, however, being considered elsewhere [19].

Contrasting with previous work on programming laws, we discussed both *universal* and *method-specific* laws. The later are simpler to use and provide more powerful assistance because they are focused and abstract from tedious constraints that are implicitly implied by the proper use of a programming method. Those laws are not only useful for the direct application of a given method, but also for the implementation of automated tools that support the method.

The formalization of laws by using class operators might turn out to define a suitable algebra for classes, which could be then easily encoded in a theorem proving tool such as OBJ [12], providing a modular encoding of the theory and allowing the semi-automatic derivation of new laws from already existing laws.

Acknowledgements

Much of Section 2 was written for a joint project proposal with several other colleagues: Augusto Sampaio and Ana Cavalcanti at Universidade Federal de Pernambuco, Uday Reddy at University of Illinois at Urbana-Champaign, and David Naumann at Stevens Institute. The work reported on Section 3 is also in the context of that project and those people have contributed with important discussions. The class splitting law presented there was extensively explored by Gustavo Bueno, under the supervision of Augusto Sampaio and myself. I am also grateful to the members of the LinES research group—specially André Santos, Augusto Sampaio and Ana Cavalcanti—for making important suggestions for improving the work presented here.

References

[1] Scott Ambler. *Building Object Applications that Work*. Cambridge University Press and Sigs Books, 1998.

[2] Paulo Borba. *Semantics and Refinement for a Concurrent Object Oriented Language*. PhD thesis, Oxford University, Computing Laboratory, Programming Research Group, July 1995.

[3] Paulo Borba. A refinement theory for concurrent object oriented languages. In *X Brazilian Symposium on Software Engineering*, pages 39–55, São Carlos, Brazil, 15-18th October 1996.

[4] Paulo Borba. Systematic development of concurrent object-oriented programs. In *Brazil–US Workshop on Formal Foundations of Software Systems, Electronic Notes in Theoretical Computer Science 7 (To appear)*. Rio de Janeiro, Brazil, 5-9th May 1997.

[5] Paulo Borba. Progressive implementation. In preparation, 1998.

[6] Paulo Borba and Joseph Goguen. Refinement of concurrent object oriented programs. In Stephen Goldsack and Stuart Kent, editors, *Formal Methods and Object Technology,* Chapter 11. Springer-Verlag, 1996. Also appeared as Technical Report TR-18-95, Oxford University, Computing Laboratory, Programming Research Group, November 1995.

[7] William Brown et al. *Anti Patterns: Refactoring Software, Architectures and Projects in Crisis*. Wiley Computer Publishing, 1998.

[8] Edwin Chan, John Boyland, and William Scherlis. Promises: Limited specifications for analysis and manipulation. In *20th International Conference on Software Engineering*, 1998.

[9] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152–159, 1992.

[10] Razvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation, 1993. *Proceedings* of Workshop on Logical Frameworks (Edinburgh, Scotland, May 1991).

[11] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] Joseph Goguen and Joseph Tardo. An introduction to OBJ: A language for writing and testing software specifications. In Marvin Zelkowitz, editor, *Specification of Reliable Software*, pages 170–189. IEEE, 1979. Reprinted in *Software Specification Techniques*, Nehan Gehani and Andrew McGettrick, editors, Addison Wesley, 1985, pages 391–420.

[13] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[14] C.A.R. Hoare, J. Michael Spivey, Ian Hayes, Jifeng He, Carroll Morgan, A. William Roscoe, Jeff Sanders, Ib Sorenson, and Bernard Sufrin. Laws of programming. *Communications of the Association for Computing Machinery*, 30(8):672–686, 1987.

[15] Doug Lea. *Concurrent Programming in Java.* Addison-Wesley, 1997.

[16] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[17] Carroll Morgan. *Programming from Specifications.* Prentice Hall, second edition, 1994.

[18] David Naumann. Predicate transformer semantics of an oberon-like language. In *IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi*, 1994.

[19] David Naumann, Uday Reddy, Paulo Borba, Ana Cavalcanti, and Augusto Sampaio. Towards a practical calculus of object-oriented programming. Research project proposal jointly submitted to the CNPq and the NSF, 1998.

[20] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[21] Uday Reddy. Objects and classes in Algol-like languages. In *Fifth International Workshop on Foundations of Object-Oriented Languages*, 1998.

[22] Augusto Sampaio. *An Algebraic Approach to Compiler Design.* World Scientific, 1997.

[23] David Watt. *Programming Language Concepts and Paradigms.* Prentice Hall, 1990.