JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 4, July-August 2003

The Theory of Classification Part 8: Classification and Inheritance

Anthony J H Simons, Department of Computer Science, University of Sheffield, U.K.

1 INTRODUCTION

This is the eighth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. In earlier articles, we explored the view that a programmer's *class* in C++ or Java corresponds in some way to a *type* in the formal model, and that a compatible *subclass* therefore corresponds to a *subtype* [1, 2]. In the last article, simple subtyping was found to be inadequate to express systematic relationships between recursive types [3]. Instead, we found that a second-order model with type parameters was needed. In this model, a programmer's class corresponds to a *bounded polymorphic type*, representing a family of similar types which share a minimum common structure and behaviour. The family likeness was expressed using a constraint, known as a *function bound* or *F-bound* [4] on the type parameter, which ensured that the parameter could only be replaced by types having at least the structure and behaviour specified in the *F-bound*.

This opens up a completely different formal notion of *class*, and consequently of *inheritance*. Whereas before, we thought of a class as a type, clearly it is now the pattern for a family of related types. Likewise, whereas inheritance was formerly the simple extension of a type, in the new model it is the extension of a general pattern. In this article, we explore further the differences between classes and types, developing the alternative formal model of classification and inheritance, which is quite different from subtyping [5].

2 SPECIFICATION VERSUS IMPLEMENTATION

So far in the *Theory of Classification*, I have been seeking to show how the intuitive notion of *class* in object-oriented languages has a *strictly formal* interpretation that is more general than the simple notion of *type*. At this point, I usually come up against a long-held prejudice among practically-minded programmers that the "real" difference between a class and a type is that a class is merely a programming language construct, whereas a type is the formal description of this. In other words, there is out there the entrenched view that "type = specification" and "class = implementation". In the following, I hope to show that this view is a red herring in our thinking about classification in object-oriented languages.

In the early days, we struggled to understand the formal nature of novel object-oriented language features, especially inheritance, which could sometimes be used in a strict way, to derive a family of related types, and sometimes in an opportunistic way, to extend implementations [6]. This led some to believe that objects have *class* and *type* independently [7, 8], asserting that an inheritance hierarchy was merely a convenience for describing shared implementation, whereas a separate type hierarchy was necessary to describe the subtyping relationships between the same objects. In some cases, the "classes" and "types" for the very same objects could be linked in different orders (see figure 1).

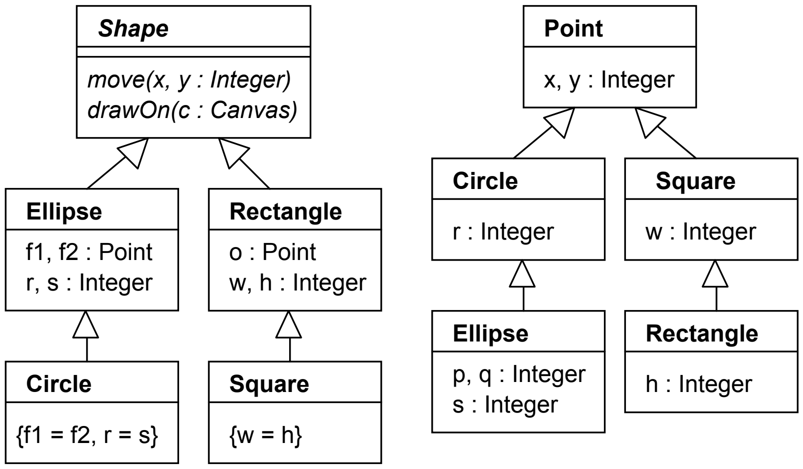

Figure 1: Sharing type (left) and implementation (right)

Now, while this is an interesting issue, it relates to *conceptual design* more than it relates to *type theory* and the notions we have been discussing here. The left-hand hierarchy expresses the conceptual family of Shapes, while the right-hand hierarchy expresses how you might conveniently derive extended records by adding variables, although it leads to conceptual nonsense: a Circle is not a kind of Point, for example. Nonetheless, it is quite possible to define a strange Circle type that is genuinely a subtype of Point - in the theory of subtyping, this would be perfectly legitimate for many definitions of Circle and Point, so long as you obeyed the rules [2]. The real issue here is one of discipline versus opportunism in conceptual design, and is not really related to types and subtyping.

Another blow to the "class = implementation" viewpoint is that programming languages like Pascal and C had types with concrete implementations long before the object-oriented notion of class was popular - nothing *necessarily* forces "class" to mean implementation. In Computer Science, we have always talked in terms of *abstract* and *concrete* types. Abstract types are formal, described in terms of operation signatures and axioms; concrete types have a representation in a programming language. So, why not extend this notion to classes, which can also be both abstract (formal and typeful) and concrete (practical and implemented)?

Modern object-oriented languages, like C++ and Eiffel, link the type hierarchy directly to the implementation hierarchy. Some allow further expression of type compatibility apart from the main implementation hierarchy, like Objective C and Java, which have separate interfaces to express common type relationships that cut across the main divisions in the class hierarchy. Earlier languages like POOL-T [8] developed completely independent implementation- and type-hierarchies, in the hope of preserving "pure subtyping" in a language with multiple, variant implementations. However, even the separate subtype hierarchy of POOL-T is defeated by recursive types and inescapably suffers from the same restrictions and lack of expressiveness that we identified previously for subtyping [3].

3 CLOSED VERSUS OPEN

A more satisfying way of distinguishing object-oriented classes from traditional simple types is to realise that a type is closed, in the sense of being complete, whereas a class is open-ended, in the sense of being subject to arbitrary subdivision and further specialisation. In older object-based languages like Modula-2 and Ada (pre-95, before the addition of inheritance), the type of an object is expressed *exactly* by its interface. In later object-oriented languages with polymorphic inheritance, the class of an object is understood to express only the *minimum* interface which members of the class must satisfy. This subtle difference between *exact* and *minimum* interfaces is what characterises the essential difference between a traditional programmer's type and a modern class. Taxonomic classification in biology also follows this pattern: a mammal is defined as something with (at least) warm blood and hair that bears and suckles live young. This is not a complete or finished definition - biologists don't exclaim: "Look, there's an instance of a mammal!", but rather identify dogs, cats or gerbils and show that these belong to the class of mammals, by virtue of having the four essential mammalian properties.

Mathematically, we can capture the same distinction between simple, closed types and open-ended polymorphic classes. In the previous article [3], we showed that a basic *class of Numbers* may be expressed using the F-bound:

$$\forall(\tau <: \text{GenNumber}[\tau]), \qquad \text{where: GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma\}.$$

According to this definition, the parameter τ may range over all kinds of numeric types, so long as they have *at least* a *plus* method with the specified signature. By contrast, the *exact Number type* is a recursive type, created from first principles as the least fixed point of the generator GenNumber (see earlier article [1] for a full explanation):

$$\text{Number} = \mathbf{Y}\,[\text{GenNumber}] = \text{GenNumber}[\text{GenNumber}[\text{GenNumber}[...]]]$$
$$\Rightarrow \{\text{plus} : \text{Number} \to \text{Number}\}, \qquad \text{at the limit of recursion.}$$

This is a very general numeric type *with only a plus* method - we are unlikely to use direct instances of this type (like mammal, above), but instead we will want to use instances of Integer, Real, or Complex.

Note that exact types, like Number, are fixed, whereas classes are open-ended and flexible. If we say n : Number, we assert that n is a variable that can only receive objects of exactly the Number type. On the other hand, if we say $x : \forall(\tau <: \text{GenNumber}[\tau])$, we assert that x is a variable that can receive objects of any numeric type in the class of Numbers, even types having more than the minimum required methods. This is the difference between monomorphism (exact typing) and bounded polymorphism (constrained flexible typing). In the following, *types* are always exact and *classes* are always polymorphic.

4 RELATING CLASSES AND TYPES

There is an interesting relationship between classes and types. It turns out that the exact Number type is the *least type* which is still a member of the polymorphic Number class. In other words, it has just enough fields to satisfy the F-bound constraint:

$$\text{Number} <: \text{GenNumber}[\text{Number}],$$

which unrolls (on the left) and evaluates (on the right) to give:

$$\{\text{plus} : \text{Number} \to \text{Number}\} \ <: \ \{\text{plus} : \text{Number} \to \text{Number}\}$$

from which it is clear that both sides are equal. In fact, for this one case alone, we could rewrite the subtyping condition as a type equivalence:

$$\text{Number} = \text{GenNumber}[\text{Number}],$$

and the reader may recall that this is exactly the same formula that identifies Number as the fixpoint of the generator GenNumber, that is, a type which is unchanged by the application of the generator [1]. From this, we know that there is exactly one type which "only just" satisfies the conditions for class membership - and this is always the recursive type created from the generator that is used to express the F-bound constraint.

We can visualise this in figure 2 by drawing the Number class as a cone-shaped bounded volume, representing a space of possible numeric types that satisfy the F-bound, and the corresponding "least type" Number, which is only just a member of this class, as the point at the apex of this cone. There are many other numeric types which belong to the Number class, which have more than the minimum required methods: figure 2 also shows the Integer type as a point in the space enclosed by the Number cone.

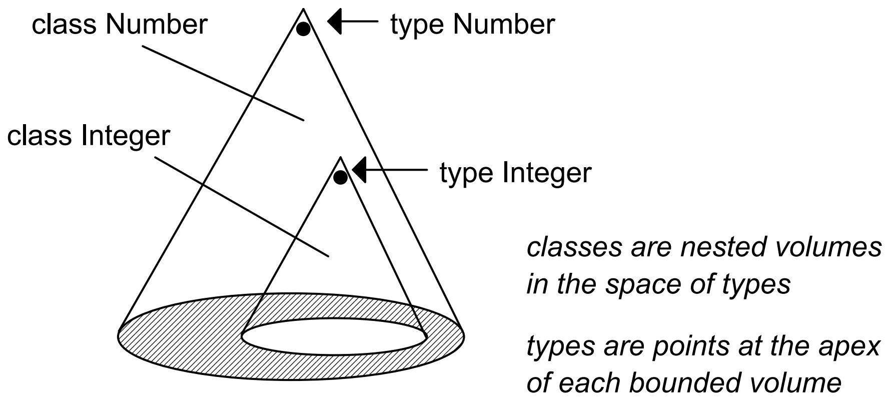
class Number
type Number
class Integer
type Integer
classes are nested volumes in the space of types
types are points at the apex of each bounded volume

Figure 2: Classes as volumes containing types as points

Can we demonstrate this mathematically? Recall that the exact Integer type is given by:

$$\text{Integer} = \mu\sigma.\{\text{plus} : \sigma \rightarrow \sigma,\ \text{minus} : \sigma \rightarrow \sigma,\ \text{times} : \sigma \rightarrow \sigma,\ \text{divide} : \sigma \rightarrow \sigma\},$$

$$\Rightarrow \quad \{\text{plus} : \text{Integer} \rightarrow \text{Integer},\ \text{minus} : \text{Integer} \rightarrow \text{Integer},\ \text{times} : \text{Integer} \rightarrow \text{Integer},\ \text{divide} : \text{Integer} \rightarrow \text{Integer}\},$$

after unrolling the recursion. We then apply the test for class membership, manipulating the expression on both sides, by unrolling the recursive type (on the left) and applying the type generator (on the right) to yield a comparison between record types:

$$\text{Integer} <: \text{GenNumber[Integer]} \quad \Rightarrow$$

$$\{\text{plus} : \text{Integer} \rightarrow \text{Integer},\ \text{minus} : \text{Integer} \rightarrow \text{Integer},\ \text{times} : \text{Integer} \rightarrow \text{Integer},\ \text{divide} : \text{Integer} \rightarrow \text{Integer}\}$$
$$<: \quad \{\text{plus} : \text{Integer} \rightarrow \text{Integer}\}$$

which is true by the record subtyping rule [2]. We have therefore shown that the Integer type is indeed in the class of Numbers. As well as the least type Number, we may expect many more numeric types like Integer, which have *more than* the minimum required methods, to be members of this class. Such numeric types could include Real, Complex, Fraction and any numeric type with at least a *plus* method. This expresses exactly the object-oriented notion of class membership.

5 A MODEL OF CLASSIFICATION

Figure 2 also visualises how we would expect a class of Integers to nest inside the class of Numbers. This is drawn using "stacking cones" to show that the volume occupied by the Integer class is contained within the Number class. Intuitively, we ought to be able to partition (divide up) the space of Numbers into sub-spaces, corresponding to the spaces occupied by the more specific numeric classes; however, we have yet to demonstrate this idea mathematically.

The Integer class is constructed in the same way as the Number class, first by defining a type generator, a type function which accepts the self-type as an argument:

$$\text{GenInteger} = \lambda\sigma.\{\text{plus} : \sigma \rightarrow \sigma,\ \text{minus} : \sigma \rightarrow \sigma,\ \text{times} : \sigma \rightarrow \sigma,\ \text{divide} : \sigma \rightarrow \sigma\}$$

then by expressing the polymorphic Integer class using an F-bound constructed using the same generator:

$$\forall(\tau <: \text{GenInteger}[\tau])$$

This constraint ensures that the parameter τ may range *only* over those numeric types which have *at least* the methods: *plus, minus, times* and *divide* with the specified signatures. It should be clear now that the exact Integer *type* is the least member of this class:

$$\text{Integer} <: \text{GenInteger}[\text{Integer}], \qquad \text{because Integer} = \text{GenInteger}[\text{Integer}].$$

How can we show that the Integer *class* nests inside the Number class? Intuitively, the purpose of classification is to show that objects which belong to specific classes also belong to the more general classes. We therefore want to be able to show that, if any actual type satisfies the constraint for the Integer class, it will also satisfy the constraint for the Number class. In other words, we seek the condition under which:

$$\tau <: \text{GenInteger}[\tau] \;\Rightarrow\; \tau <: \text{GenNumber}[\tau]$$

Intuitively, this holds true for these two particular generators, because GenInteger defines strictly more methods than GenNumber and otherwise it has an identical *plus* method. So, any type satisfying $\tau <: \text{GenInteger}[\tau]$ is bound to satisfy $\tau <: \text{GenNumber}[\tau]$. If GenInteger had not defined a *plus* method with a matching signature, then no types could satisfy both constraints. The condition we seek therefore depends on the structure of one generator including all the structure of the other generator.

This sounds somewhat similar to a record subtyping rule [2], except that the generators are not record types, they are type functions with *distinct* self-type arguments. Though we cannot make a rule to compare the generators directly, we can do this indirectly using a rule that compares *instantiations* of the generators, since this yields proper record types. In fact, we want to compare *all possible* instantiations of the two generators with the *same type*. This is known as a *pointwise subtyping* condition, which we write as:

$$\forall\tau\ .\ \text{GenInteger}[\tau] <: \text{GenNumber}[\tau]$$

We read this as: "For all types τ, the record type you get by instantiating GenInteger with τ is always a subtype of the record type you get by instantiating GenNumber with τ". Literally, the constraint expresses two things:

- the subclass generator must have a larger interface than the superclass generator
- the self-types must be made to refer to the same type when making any comparison

and this is the condition under which the GenInteger generator stands in a subclass-to-superclass relationship with the GenNumber generator.

Abstracting away from numeric types, we obtain the subclass rule relating *any* two generators, which we shall name (arbitrarily) GenSub and GenSuper:

$$\frac{\forall\tau\ .\ \text{GenSub}[\tau] <: \text{GenSuper}[\tau]}{\forall t\ .\ t <: \text{GenSub}[t]\ \Rightarrow\ t <: \text{GenSuper}[t]} \qquad \text{[Subclass]}$$

"If the generators GenSub and GenSuper stand in a pointwise subtyping relationship, then any type which satisfies the Sub-class constraint will also satisfy the Super-class constraint". This is the rule which describes how classes are nested inside each other. From this, we may derive another rule, which allows us to infer how types which belong to one class also belong to more general classes:

$$\frac{t <: \text{GenSub}[t],\quad \forall\tau\ .\ \text{GenSub}[\tau] <: \text{GenSuper}[\tau]}{t <: \text{GenSuper}[t]} \qquad \text{[Classify]}$$

"If t is a member type of the Sub-class, and the Sub-class is nested inside the Super-class, then t is also a member type of the Super-class." These are two of the most important rules in the theory of classification describing the subclass relationship and transitive class membership.

6 A MODEL OF INHERITANCE

In our presentation, we want to distinguish carefully the notions of *classification* and *inheritance*. Classification describes the way in which exact types belong to classes and the way in which classes are nested inside one another. Inheritance describes an extension mechanism for defining more specialised classes incrementally. That is, inheritance is merely a short-hand. This means that any class we could develop incrementally using inheritance must look the same as if we had defined it as a whole, from first principles. As a challenge, we shall attempt to derive the Integer class from the Number class.

Previously, we adopted a simplistic model of inheritance, using record type extension [3]. In this approach, a record type is considered to be a set of fields, which can be extended using the set union operator $\cup$ to create a larger record type, which is therefore a subtype of the original record [2]. The basic model for record type extension is:

$$\text{Subtype} = \text{Basetype} \cup \text{Extension}$$

However, we found an anomaly when extending recursive record types, which is that fields obtained from the base type were fixed with the wrong types when combined in the extended record type [3]. If we derive an Integer type naïvely from a Number type, we find that the *plus* method for Integer ends up with intuitively the wrong type signature:

$$\begin{aligned}&\text{Number} = \mu\sigma.\{\text{plus} : \sigma \to \sigma\},\\ &\Rightarrow \{\text{plus} : \text{Number} \to \text{Number}\}, \qquad \text{after unrolling the recursion;}\end{aligned}$$

$$\text{Integer} = \mu\sigma.(\text{Number} \cup \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})$$

$$\begin{aligned}\Rightarrow \{&\text{plus} : \text{Number} \to \text{Number}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

Essentially, the reason why this doesn't work is because we are extending simple types using a subtyping model and are hitting the limits of subtyping in the context of recursion. The self-types of Number and Integer are fixed independently, such that after record union, there is no single, uniform self-type, resulting in a kind of "self-type schizophrenia".

The insight offered by Cook and others [5] is that inheritance should not be modelled as the *extension of simple types*, but rather it is the *extension of the patterns described by their generators*. With generators, we are better able to control the binding of the self-types, so that these refer uniformly to the same type in the combined result. Starting with the GenNumber generator, we shall derive a GenInteger generator incrementally from it, but also rebind the self-type of the base generator at the same time:

$$\text{GenNumber} = \lambda\tau.\{\text{plus} : \tau \to \tau\}$$

$$\begin{aligned}\text{GenInteger} &= \lambda\sigma.(\text{GenNumber}[\sigma] \cup\\ &\qquad\qquad \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})\\ &= \lambda\sigma.(\{\text{plus} : \sigma \to \sigma\} \cup \{\text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\})\\ &= \lambda\sigma.\{\text{plus} : \sigma \to \sigma, \text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\}\end{aligned}$$

This yields exactly the GenInteger that we hoped for, equivalent to the generator that we defined from first principles in section 5. Note how the inherited *plus* method is typed in terms of σ, the self-type of GenInteger, rather than in terms of τ, the old self-type of GenNumber. All the methods of GenInteger are uniformly typed in terms of σ, solving the "self-type schizophrenia" problem.

The trick at the heart of this derivation is the type-instantiation of the GenNumber generator, obtained through the application: GenNumber[σ]. Recall that GenNumber is a type-function, with a formal argument τ. When GenNumber is applied to σ, we substitute

$\{\sigma/\tau\}$ throughout in the body. The result of the application is a base record of method types: $\{\text{plus} : \sigma \rightarrow \sigma\}$ which is subsequently unioned with the extension record to yield the result.

To obtain the exact Integer type, we may take the fixpoint of the GenInteger generator that we have just derived, to bind the self-type argument recursively:

$$\text{Integer} = \mathbf{Y}\,[\text{GenInteger}] = \text{GenInteger}[\text{GenInteger}[\text{GenInteger}[...]]]$$

$$\begin{aligned}\Rightarrow\ &\{\text{plus} : \text{Integer} \rightarrow \text{Integer},\ \text{minus} : \text{Integer} \rightarrow \text{Integer},\\ &\quad \text{times} : \text{Integer} \rightarrow \text{Integer},\ \text{divide} : \text{Integer} \rightarrow \text{Integer}\}, \qquad \text{at the limit.}\end{aligned}$$

This has the type signature that we desire, namely one that is uniformly typed in terms of the Integer self-type, rather than one which is schizophrenic. This seems therefore to be a more satisfactory model for explaining incremental derivations in a class hierarchy.

7 CONCLUSION

We have exposed the heart of the theory of classification, using Cook's F-bounded quantification to express the idea that a class is *a family of types that share a minimum common structure.* By contrasting the notion of *type* in older object-based languages with the notion of *class* in newer object-oriented languages with inheritance, we have argued that extensibility and polymorphism are what properly characterise classes apart from traditional programmer's types. Thereafter, we have used *class* to refer to a polymorphic family and *type* to refer to one member of this family.

We developed a second-order model of class membership, in which recursive types with similar structure could be shown to belong in the same class. We extended this to a model of subclassing, based on a pointwise relationship between generators, to show how classes are nested hierarchically. Finally, we showed how generator adaptation provided a more satisfactory model of incremental inheritance, avoiding problems of schizophrenic self-reference. The languages Smalltalk and Eiffel adopt this more sophisticated model of inheritance, in which inherited references to self and the self-type are redirected to refer uniformly to the subclass.

While we dispelled one myth about classes and implementation, it is clear that we have only covered the *typeful* aspects of classes here. We defer a treatment of the *concrete* aspects of classes, especially the inheritance of implementation, to a later article. This will help us further in understanding the differences between the classification model of Smalltalk and Eiffel, and the subtyping model of Java and C++.

REFERENCES

[1] A J H Simons, "The theory of classification, part 3: Object encodings and recursion", in *Journal of Object Technology, vol. 1, no. 4, September-October 2002*, pp. 49-57. http://www.jot.fm/issues/issue_2002_09/column4

[2] A J H Simons, "The theory of classification, part 4: Object types and subtyping", in *Journal of Object Technology, vol. 1, no. 5, November-December 2002*, pp. 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[3] A J H Simons, "The theory of classification, part 7: A class is a type family", in *Journal of Object Technology, vol. 2, no. 3, May-June 2003*, pp. 13-22. http://www.jot.fm/issues/issue_2003_05/column2

[4] P Canning, W Cook, W Hill, W Olthoff and J Mitchell, "F-bounded polymorphism for object-oriented programming", *Proc. 4th Int. Conf. Func. Prog. Lang. and Arch*. (Imperial College, London, 1989), pp. 273-280.

[5] W Cook, W Hill and P Canning, "Inheritance is not subtyping", *Proc. 17th ACM Symp. Principles of Prog. Lang.*, (ACM Sigplan, 1990), pp. 125-135.

[6] M Sakkinen, "Disciplined inheritance", *Proc. 3rd European Conf. Object-Oriented Prog.,* (Nottingham: British Computer Society, 1989), pp. 3-24.

[7] A Snyder, "Encapsulation and inheritance in object-oriented programming languages", *Proc. 1st ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 21(11),* (ACM Sigplan, 986), pp. 38-45.

[8] P America , "Designing an object-oriented language with behavioural subtyping", *Proc. Conf. Foundations of Object-Oriented Lang*., (1990), pp. 60-90.

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching in the Department of Computer Science, University of Sheffield, where he leads object-oriented research in verification and testing, type theory and language design, development methods and precise notations. He can be reached at a.simons@dcs.shef.ac.uk.