
Article:
Simons, A.J.H. (2003) The theory of classification part 7: a class is a type family. Journal of Object Technology, 2 (3). 13 - 22. ISSN 1660-1769

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 2, No. 3, May-June 2003

The Theory of Classification Part 7: A Class is a Type Family

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

1 INTRODUCTION

This is the seventh article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. So far, we have built up a model of objects as simple records, which are instances of corresponding record types [1]. Initially, we took the seemingly attractive 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* [2], linking this with an algebraic description of subtype-compatible semantic behaviour [3]. Naturally, the *class*-construct in an object-oriented language also defines an implementation for the instances of the class, but there is no formal contradiction here in considering just the typeful aspects of class declarations. Subtyping does indeed provide a simple, flexible model for type compatibility, but we shall find that it is not always as useful as we might expect.

We shall discover circumstances in which a type system based on subtyping breaks down, providing less than useful information. Object-oriented languages like Smalltalk and Eiffel exhibit sophisticated, systematic kinds of behaviour which cannot adequately be described in terms of types and subtyping. By appealing to natural notions of classification in biology, we shall demonstrate the extent to which the subtyping model fails to capture the intuitive notion of a *class*. In this article, we shall define the notion of *class* formally, and prove that it is more than just a type. To understand this, we will need to extend our formal model to include type polymorphism. This requires the *second-order* λ-calculus and notions of *universal* [4, 5] and *function-bounded quantification* [6, 7].

2 THE PROBLEM OF RECURSIVE CLOSURE

There used to be a popular rhyming couplet that joked about the terminology used in the biological classification of the animal kingdom:

Cite this column as follows: Anthony 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

Cats have kittens, dogs have puppies,
But guppies just have baby guppies.

While every species has young of its own kind, the biologists seemed to have left some holes in the taxonomy, surely an oversight! What is more disturbing is that most object-oriented languages will assert that Dog, Cat and Guppy, by virtue of being kinds of Animal, all mate with an Animal and produce an offspring, which is (you guessed) an Animal, something forced upon recursive types by the rules of subtyping. To see why, let us express the reproducing behaviour of all animals (ignoring litters of offspring) formally as:

$$\text{Animal} = \mu\sigma.\{..., \text{mate} : \sigma \to \sigma, ...\}$$

This defines Animal as a recursive record type whose methods (most of which are not shown) include the *mate* method. In such a recursive definition, σ is the self-type, a placeholder for the eventual type name, bound recursively using μ to refer to the whole record (the earlier articles [1, 2] explain this notation). Once the recursion is established, we can access the type of the *mate* method as:

$$\text{Animal.mate} : \text{Animal} \to \text{Animal}$$

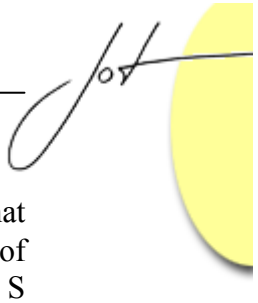
showing that an Animal mates with an Animal and produces an Animal offspring, suitably capturing the general notion. Intuitively, we should like to introduce the Animal subclasses Dog, Cat (and even Guppy), such that these creatures all mate with, and produce young of their own kind, in the uniformly specialised style:

$$\text{Dog.mate} : \text{Dog} \to \text{Dog}$$
$$\text{Cat.mate} : \text{Cat} \to \text{Cat}$$

We might expect Cat and Dog to be subtypes of Animal; however this is not the case, since they both redefine the signature of the *mate* method a way that violates subtyping. The function subtyping rule allows a subtype function to have more general arguments and a more specific result [2]. Here, the Dog type replaces the signature Animal $\to$ Animal with the retyped signature Dog $\to$ Dog, which unhelpfully specialises both argument and result. The best we could do while still preserving subtyping is to break with uniform specialisation and invent strangely retyped versions of *mate* which still accept Animal arguments:

$$\text{Dog} = \mu\sigma.\{..., \text{mate} : \text{Animal} \to \sigma, ...\}$$
$$\text{Dog.mate} : \text{Animal} \to \text{Dog}$$

This ensures that Dogs produce puppies, but still allows a Dog to mate with any kind of Animal, which seems intuitively wrong, but is formally correct by the rules of subtyping. The Animal type declared that its *mate* method always accepts an argument of at least the Animal type and we cannot go back on this, particularly if we expect to invoke *Dog.mate* dynamically through an Animal variable and supply any legal Animal argument.



In general, recursion interacts poorly with subtyping. If any type T has a method that is *closed* over its own type: T.m : T → T, then it is impossible to specialise the type of this method uniformly. Let us assume that such a type S existed, with a method S.m : S → S. If we want to establish a subtype relationship S <: T, then for the replaced method m to be a subtype, we have to show, on the result-side, that S <: T, which is consistent, and also show, on the argument-side, that T <: S, which is precisely the opposite of the relationship we seek. The only condition under which both S <: T and T <: S is if S = T!

This is one reason why redefined methods cannot change their types in languages with both type recursion and subtyping. Consider that in Java, the *equals* method is recursively typed at the root: Object.equals : Object → Boolean, making every Java type recursive. Recursion fixes the type of this method, which can never be specialised. This leads to a damaging lack of expressiveness: any redefinition of *equals* must still accept a generic Object argument, yet it is usually a semantic error to seek to equate a Cat with anything other than another Cat. In practice, programmers compare like with like, but they can only do this by *type downcasting* the argument of the *equals* method, forcibly overriding the natural type system.

3 THE PROBLEM OF TYPE LOSS

A different but related problem arises when recursively-typed methods are inherited and invoked in a subtype object. So far, we have not modelled the notion of inheritance in any detail, but let us invent a simple rule to create subtypes by record extension [2]. Since records are merely maps from labels to values, and maps are really just sets, we can combine records using set union. Let us assume that we wish to define a hierarchy of numeric types, and that the basic Number type provides a primitive notion of addition:

$$\begin{aligned}\text{Number} &= \mu\sigma.\{\text{plus} : \sigma \to \sigma\},\\ &= \{\text{plus} : \text{Number} \to \text{Number}\}\end{aligned}$$

after unrolling the recursion. We can seek to derive other numeric types by extending this, yielding for example the Natural, Integer, Real and Complex numbers. In particular, the Integer type offers a full range of arithmetical methods:

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

This defines the Integer type by extending the Number type with a record of additional fields, and then fixing the recursion. After unrolling Number to yield the corresponding record type, we can compute the union of fields, yielding the recursive record type:

$$\begin{aligned}\text{Integer} &= \mu\sigma.\{\text{plus} : \text{Number} \to \text{Number}, \text{minus} : \sigma \to \sigma,\\ &\qquad \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\},\\ &= \{\text{plus} : \text{Number} \to \text{Number}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\qquad \text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

after unrolling the Integer recursion. Curiously, while the locally declared methods are all typed in terms of Integer, the inherited *plus* method is already fixed in terms of Number. As a consequence, the intuitively reasonable arithmetic expression:

```
i, j, k : Integer
i.plus(j).minus(k)
```

fails even to typecheck! This is because the sub-expression *i.plus(j)* returns a result of the general type Number, for which the *minus* method is not defined. This is the problem of type-loss under inheritance, something with which Java and C++ programmers will be familiar. It arises because the type recursion in Number fixes the signature of the *plus* method, and this more general type is still retained in Integer, after the union of fields. In practice, Java and C++ programmers have to use *type downcasting* to override the natural type system, if they wish expressions like this to compile, something like:

```
i, j, k : Integer
((Integer) i.plus(j)).minus(k)
```

Type downcasting is typically considered a last resort, a dirty trick to be used on occasions when the natural type system doesn't help. Here, we have shown how type downcasting has to be used systematically all the time to overcome deficiencies in the type system. This is a strong indicator that subtyping is not the most appropriate formal model for object-oriented languages. Instead, we need a more expressive type system.

4 QUANTIFICATION OVER TYPES

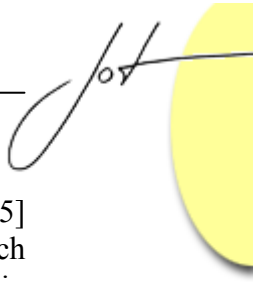
Working backwards from the desired goal, it seems that our intuitive notion of classification requires a type system in which recursive types can have methods that are closed over their own type, but which are nonetheless related to each other in some systematic way. We want to be able to support *families* of related types that behave in similar ways, such as the numeric types which all provide addition:

$$\text{Integer.plus} : \text{Integer} \to \text{Integer}$$
$$\text{Complex.plus} : \text{Complex} \to \text{Complex}$$
$$\text{Natural.plus} : \text{Natural} \to \text{Natural}$$

and somehow be able to assert that these all belong to the *class of numbers*. There is clearly a systematic pattern here, in which all related numeric types τ have a *plus* method with the type signature $\tau.\text{plus} : \tau \to \tau$. We can get close to this idea with universal quantification:

$$\forall \tau \,.\, \tau.\text{plus} : \tau \to \tau$$

which says that "all types τ have a method *plus* which accepts and returns a value of the same type τ." This is still not quite right, since we want *plus* to be defined only for the numeric types, not for absolutely every type.



Universal quantification was adopted independently by Girard [4] and Reynolds [5] as a way of introducing *type parameters*, variables which range over types (ie which receive types as their bound value). They are used in the *second-order* λ-calculus, in which functions can accept both value- and type-arguments. This was found useful to express the notion of *polymorphism*, describing functions that work uniformly on families of types. *Parametric polymorphism* was built into the early functional programming languages ML and Hope, and exists in object-oriented languages as the *templates* in C++, or *generic types* in Ada and Eiffel. Before we can understand the usefulness of type parameters in modelling polymorphism, we need to explain the idea of different *orders* of calculus, and understand something of how to construct and simplify expressions.

5 SIMPLY-TYPED AND POLYMORPHIC CALCULUS

A zero-order system has no variables, but only sets of values. A first-order system has functions, whose bound variables range over simple values. A second-order system has functions, whose bound variables range over both values and simple types. The λ-calculus is a basic functional language. In its primitive form, it is *untyped*, so we cannot yet say anything about its order. About the simplest function you can write in the untyped λ-calculus is the *identity* function, which accepts an argument and returns it unchanged:

λx.x	the untyped identity function
λx.x 3 ⇒ 3	apply identity to an Integer
λx.x 'a' ⇒ 'a'	apply identity to a Character

Recall that "λx" means "a function of x" and everything after the dot is the function body, here just "x". Placing the function next to a value *applies* the function to this value, rather like calling a function in a programming language. Applying λx.x to the integer 3 simply binds the argument x←3 then returns the body x, which has the substituted value 3 (see also [1]). In the untyped calculus, we can apply λx.x to anything, such as integers, characters or even other functions (in which case we would have a *higher-order* system).

We may attach simple types to the function's arguments in the *simply-typed* λ-calculus. This is a first-order system, since simply-typed variables can only range over basic values. If we so wish, we can restrict the identity function to accept only Integer values:

λ(x:Integer).x	a typed identity function
λ(x:Integer).x 3:Integer ⇒ 3	type-safe application
λ(x:Integer).x 'a':Character ⇒ error	type-incorrect application

The difference here is that the types of the formal argument and the actual value must match, otherwise the application is deemed illegal, a type error. We can say that this identity is a *monomorphic* function, since it is defined only for a single type, Integer. We say that identity has the type: Integer → Integer.

In the *second-order* λ-calculus, functions have extra arguments standing for types, which are introduced ahead of the arguments standing for values of these types. A *polymorphic* version of identity is given by the following, in which τ is a *type parameter:*

λτ.λ(x:τ).x	a polymorphic identity function
λτ.λ(x:τ).x Integer ⇒ λ(x:Integer).x	instantiation with Integer
λτ.λ(x:τ).x Character ⇒ λ(x:Character).x	instantiation with Character
λτ.λ(x:τ).x Integer 3:Integer ⇒ 3	instantiation and application
λτ.λ(x:τ).x Character 'a':Character ⇒ 'a'	instantiation and application

Identity now expects a type argument: λτ and then a value of this type: λ(x:τ). If we apply identity just to a type, such as Integer, then we bind τ←Integer and return the body, which after the type substitution is the simply-typed version of the function. This models the notion of *type parameter instantiation* in C++ or Eiffel, in which type parameters are replaced by actual types. We may apply the resulting simply-typed function to a value of the expected type, as before. Second-order functions expect to be applied to a type, then to a value in that order. We say that polymorphic identity has the type: ∀τ.τ → τ, since it applies to any type and returns a result of the same type.

6 GENERIC OBJECT TYPES

This kind of construction can be used to extend our formal model of object types and allows us to define polymorphic types (generic, or templated types). Let us start with a *monomorphic* recursive record type for an IntegerStack:

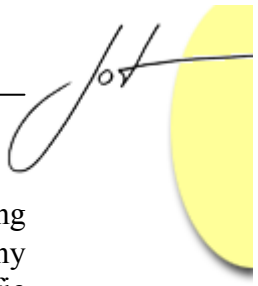
$$\text{IntegerStack} = \mu\sigma.\{\text{push} : \text{Integer} \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \text{Integer},\ \text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}$$

As before, σ is a recursive placeholder for the eventual IntegerStack. We may modify this definition to create a *polymorphic* type if we replace occurrences of Integer by a type parameter. We must introduce the parameter at the head of the type definition:

$$\text{Stack} = \lambda\tau.\mu\sigma.\{\text{push} : \tau \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \tau,\ \text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}$$

Here, λτ introduces the parameter τ standing for the element-type, ahead of μσ, which binds the recursion in the rest of the record. This Stack definition now has the form of a *type function*, that is, a function which expects a type argument: τ and then returns a result, a record type in which τ will be bound to some actual type. To see how this works, we can apply Stack to the Integer type (*ie* call Stack with Integer as its actual type argument):

$$\text{Stack[Integer]} = \mu\sigma.\{\text{push} : \text{Integer} \to \sigma,\ \text{pop} : \to \sigma,\ \text{top} : \to \text{Integer},\ \text{empty} : \to \text{Boolean},\ \text{size} : \to \text{Integer}\}$$



and the result we obtain is identical to the IntegerStack type from above, after substituting {Integer/τ} in the record body. This is interesting, because we may apply Stack to any type we fancy, such as Stack[Character] or Stack[Boolean], creating specific instantiations of the polymorphic type. There is no restriction on the actual element type we could supply, so this kind of polymorphism is sometimes known as *universal polymorphism*. We can express the polymorphic types of individual methods using universal quantification:

$$\forall\tau \,.\, \text{Stack}[\tau].\text{push} : \tau \rightarrow \text{Stack}[\tau]$$
$$\forall\tau \,.\, \text{Stack}[\tau].\text{pop} : \rightarrow \text{Stack}[\tau]$$
$$\forall\tau \,.\, \text{Stack}[\tau].\text{top} : \rightarrow \tau$$

which we read as: "for all types τ, a Stack of τ has a *push* method that accepts an argument of the type τ, and returns a Stack of τ", and similarly for the other methods. Clearly, the generic Stack type expresses something about a family of related Stack-types, but this is still not the notion of *class* that we are seeking to capture.

7 FUNCTION-BOUNDED QUANTIFICATION

It was Cook [6, 7] who first realised that in order to model a class as a polymorphic family of related types, the key lay in making the self-type flexible, so that it could refer to a different actual type in every member of the type family. In an earlier article [1] we introduced *type generators* for recursive types, in which the self-type σ is a parameter and is not yet bound. Type generators are similar to our *type functions* for generic types, except that the self-type parameter σ eventually stands for the whole type, not for a part of it.

Type generators can be used, exactly like type functions above, to create different instantiated versions of a parameterised record type. To see how this works, we revisit the Number type, but this time express it as a type generator, in which the self-type is not recursively fixed, but is a parameter introduced by λσ:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \rightarrow \sigma\}$$

GenNumber is a generator for a family of related record types which have the general structure of numbers with a plus method. To show this, we can apply GenNumber to other numeric types, and this has the effect of adapting the self-type σ, which is substituted by whatever type-argument we supply:

$$\text{GenNumber}[\text{Integer}] = \{\text{plus} : \text{Integer} \rightarrow \text{Integer}\}$$
$$\text{GenNumber}[\text{Real}] = \{\text{plus} : \text{Real} \rightarrow \text{Real}\}$$
$$\text{GenNumber}[\text{Complex}] = \{\text{plus} : \text{Complex} \rightarrow \text{Complex}\}$$

This looks promising, in that we are able to construct record types with a *plus* method that is uniformly specialised to specific numeric types. We shall use this adaptive ability below.

The ideal type for an Integer is a recursive type whose methods are all closed over its own type, which we can unroll to a record type expressed in terms of Integers:

$$\begin{aligned}\text{Integer} &= \mu\sigma.\{\text{plus} : \sigma \to \sigma, \text{minus} : \sigma \to \sigma, \text{times} : \sigma \to \sigma, \text{divide} : \sigma \to \sigma\},\\ &= \{\text{plus} : \text{Integer} \to \text{Integer}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\qquad \text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

Although this type can never be a subtype of the recursive Number *type* from section 3 above (because it uniformly specialises plus : Number → Number to plus : Integer → Integer), it is nonetheless a subtype of a *specially adapted* GenNumber *generator*, that is:

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

which we can demonstrate by unrolling Integer to a record (on the left-hand side) and then evaluating GenNumber[Integer] (on the right-hand side) and comparing the two records:

$$\begin{aligned}&\{\text{plus} : \text{Integer} \to \text{Integer}, \text{minus} : \text{Integer} \to \text{Integer},\\ &\qquad \text{times} : \text{Integer} \to \text{Integer}, \text{divide} : \text{Integer} \to \text{Integer}\}\\ <:\quad &\{\text{plus} : \text{Integer} \to \text{Integer}\}\end{aligned}$$

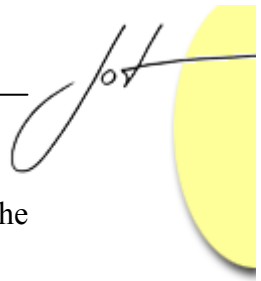
This satisfies the record subtyping rule [2]. The left-hand side contains more fields than the right-hand side, a simple case of record extension. It turns out that all the other numeric types (with more methods than Number) can be shown to enter into a similar relationship with a suitably-adapted version of the generator, for example:

$$\begin{aligned}&\text{Real} <: \text{GenNumber}[\text{Real}]\\ &\text{Complex} <: \text{GenNumber}[\text{Complex}]\end{aligned}$$

and it follows intuitively that any type τ satisfying: $\tau <: \text{GenNumber}[\tau]$ belongs to the family of numeric types which share at least the plus-method. From this, Cook realised that a class is a polymorphic family of types that satisfy a constraint, or *bound* [6, 7], expressed using a generator function. Whereas universal quantification introduces type parameters that range over any type, Cook's *function-bounded quantification* introduces type parameters that only range over a restricted group of types which satisfy the constraint. The whole *class of numbers* can be expressed formally as the type family:

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

meaning "all those types which are subtyes of the adapted GenNumber generator". What is unusual about this special kind of quantification is that the parameter τ appears on both sides of the <: subtyping constraint; but it turns out that this is exactly what is necessary to express the notion of a family of recursively closed types that have a shared minimum structure.



A suitable polymorphic type for the plus method may now be given, by restricting the family of types to those in the class of numbers:

$$\forall(\tau <: \text{GenNumber}[\tau]) \; . \; \tau.\text{plus} : \tau \rightarrow \tau$$

The constraint $\tau <: \text{GenNumber}[\tau]$ is known as a *function bound*, or *F-bound* for short. F-bounded quantification was a revolutionary discovery, because it captured exactly the kind of polymorphism present in object-oriented languages, in which methods apply to families of types sharing a minimum common structure.

8 CONCLUSION

We have formally defined the notion of class, using Cook's F-bounded quantification to express the idea that a class is *a family of types that share a minimum common structure.* This is a radical departure from the earlier view that a programmer's class corresponds to a simple type. Although the languages Java and C++ adopt this simpler view, we found that there were sufficient reasons to challenge this view, particularly the evidence from the frequent use of type downcasting needed to overcome inadequacies of first-order type systems based on types and subtyping. We are moving towards a second-order type system, in which a programmer's class really corresponds to a polymorphic type. We showed how polymorphism is modelled systematically using type parameters, and explained the relationship between universal quantification, which supports the definition of generic types, and F-bounded quantification, which supports the definition of classes:

$$\text{GenAnimal} = \lambda\sigma.\{\text{mate} : \sigma \rightarrow \sigma\}$$
$$\forall(\tau <: \text{GenAnimal}[\tau]) \; . \; \tau.\text{mate} : \tau \rightarrow \tau$$

and even satisfies natural intuitions about biological classification in which animals reproduce their own kind. Mathematics, as someone once said, is pure poetry.

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-Decembe 2002, pp. 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[3] A J H Simons, “The theory of classification, part 5: Axioms, assertions and subtyping”, in *Journal of Object Technology*, vol. 2, no. 1, January-February, pp. 13-21. http://www.jot.fm/issues/issue_2003_01/column2

[4] J-Y Girard, “Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur”, *PhD Thesis,* Universite Paris VII, 1972.

[5] J Reynolds, “Towards a theory of type structure”, *Proc. Coll. Prog., New York, LNCS 19* (Springer Verlag, 1974), 408-425.

[6] W Cook, “A denotational semantics of inheritance”, *PhD Thesis,* Brown University, 1989.

[7] 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), 273-280

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.