JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 7, July-August 2004

The Theory of Classification Part 13: Template Classes and Genericity

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

1 INTRODUCTION

This is the thirteenth article in a regular series on object-oriented type theory for non-specialists. Previous articles have gradually built up models of objects [1], types [2] and classes [3] in the λ-calculus. Inheritance has been shown to extend both type schemes [4] and implementations [5]. The most recent article [6] presented a model of a simple class hierarchy, with a root *Object* class, and various subclasses modelling geometric concepts, including a Cartesian *Point*, an abstract *Shape* class and a concrete *Rectangle* class. The aim was to demonstrate how natural intuitions about generalisation and specialisation could be expressed in the theoretical model, both at the type and implementation levels. Methods were written for abstract classes which also applied in a type-correct way to all classes beneath them in the class hierarchy, such as the *origin* method for *Shapes* [6].

However, abstract classes are not the only way in which generality can be expressed. Some object-oriented languages allow the introduction of type parameters, standing in place of actual types. These are known as *templates* in C++, or *generic parameters* in Ada or Eiffel[1]. The idea is that algorithms may be written without knowing full type information about all the elements involved. The actual types are supplied later, in a process known as *instantiating* the type parameters. In this article, we explore the consequences of adding generic classes to the Theory of Classification. Firstly, we look at some historical notions of polymorphism and type parameters. Secondly, we examine how to incorporate these into the type-level of the theory. Finally, we look at how introducing or instantiating type parameters can be combined with the process of deriving subclasses by inheritance.

[1] At the time of writing, several proposals exist for adding generic types to Java. One is actively being pursued for inclusion in the next revision of the language.

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification – Part 13: Template Classes and Genericity", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 15-25. http://www.jot.fm/issues/issue_2004_07/column2

2 TYPE ABSTRACTION AND POLYMORPHISM

It is tempting to think that the object-oriented family of languages was the first to generalise the notion of type. This is incorrect, although it is fair to say that the object-oriented family is the only group of languages to suppose that *systematic* sets of relationships exist between all the types (chiefly through the type hierarchy induced by the *subtype* [2] or *subclass* [4] relationships). The term used to describe generalisation over types is *polymorphism*, coming from the Greek *poly* (many) and *morphe* (form). The earliest strongly-typed programming languages were *monomorphic*, that is, variables were given a single type and could only be bound to values of this type. By contrast, a *polymorphic* language is one in which type constraints are systematically generalised and variables may be bound to values of more than one type. This opens the way to generic styles of programming, in which generic algorithms accept arguments of many different types.

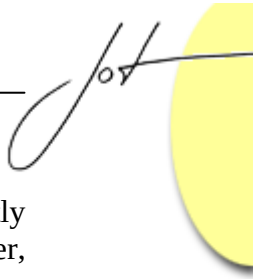
As long ago as the mid-1960s, Strachey and others [7, 8, 9] identified families of types that were sufficiently similar in structure that one could write polymorphic functions acting over them. These were typically the container types, such as *List* and *Stack*, for which functions like *cons*, *append*, *push* and *pop* could be written irrespective of the type of element they contained. Tennent [10] first proposed the use of type parameters to abstract over the unknown parts of these types, giving rise to the declaration style: *Stack[T]* representing a *Stack* of any element type *T*. So, it was possible to write a polymorphic *push* function that acted upon many different types of *Stack*, by giving it the parameterised type signature:

push (elem : T, stk : Stack[T]) : Stack[T]

Elsewhere, Strachey noted a tendency in programming languages to provide polymorphic functions in another way, simply by adding extra overloaded definitions to existing function names. The operator + might be used in one place to add *Integers* and *Reals*, but then also in another place to concatenate *Strings* and append *Lists*. Strachey therefore distinguished between:

- parametric polymorphism – provided by parameterised functions acting in a systematic way over a variety of types; and
- *ad hoc* polymorphism – provided by defining extra meanings for existing function names in an undisciplined way.

Today, these two forms of polymorphism are respectively known as *genericity* (or *templates*) and *overloading*. Strachey rejected *ad hoc* polymorphism on the grounds that it was not amenable to formal analysis. No semantic correspondence need exist between the different definitions overloaded on a single function name, for example: $x + y == y + x$ is true if x, y : *Integer*, but false if x, y : *String*. On the other hand, systematic parametric polymorphic mechanisms later entered into the designs of functional programming languages, such as ML [11]. In ML, sophisticated type inference is used at runtime to propagate actual type information into type parameters. Ada was the first



modular language to introduce generic packages, which had to be instantiated explicitly before use [12], generating a separate compiled image for each instantiation. However, parametric polymorphism existed even before these languages used it systematically. For example, in Pascal, the declaration:

myArray : ARRAY 1..10 OF Integer;

uses the ARRAY OF… special type constructor to build arrays. Such a type constructor can be readily explained as a Tennent-style parameterised polymorphic type:

Array[SubrangeType, ElementType]

in which the *SubrangeType* and *ElementType* are type parameters. Likewise, Pascal's SET OF… constructor can be considered a polymorphic type. Today, parametric polymorphism exists in all the strongly-typed functional languages, including ML, Hope, Miranda, Clean and Haskell. It is present in many object-oriented languages, such as Ada-95, Eiffel and C++, which have explicit parametric typing mechanisms.

3 A FORMAL MODEL OF POLYMORPHISM

Girard [13] and Reynolds [14] are independently credited with having provided the first formal model of polymorphism. They extended the simply-typed λ-calculus to include arguments standing for types, as well as for values. This is the (second-order) polymorphic typed λ-calculus, which we first introduced in the earlier article [3]. The differences between the simply-typed and polymorphic λ-calculus are here explained in more detail.

In the simply-typed λ-calculus, one can write functions whose arguments accept values that have types. For example, a function for constructing a coordinate object can be written[2]:

makeIntegerCoord : Integer → Integer → IntegerCoord
= λ(a : Integer).λ(b : Integer).{x ↦ a, y ↦ b}

This function accepts two arguments *a* and *b*, both values of the *Integer* type, and returns a record, whose *x* and *y* fields map to these *Integer* values. So, for example, we can create an *IntegerCoord* object at the location (2, 3) by constructing it:

makeIntegerCoord(2)(3) - *ie apply to value 2, then apply to value 3*
⇒ {x ↦ 2, y ↦ 3}

The type of the result is a record type, called *IntegerCoord* in the type signature of the function above. Technically, we should have defined this record type, before using it in the function's type signature, in the style:

IntegerCoord = {x : Integer, y : Integer}

[2] This style is slightly different from the previous article [6]. Here, we introduce each argument separately. Previously, we introduced the pair of Integers as a single argument.

Let us assume now that we want to generalise coordinates so that we can construct real-valued coordinates as well as integral-valued coordinates. Intuitively, we want to abstract over the type of the fields, and replace the hard-wired *Integer* type by a type parameter. The definition of *Coord* must therefore be turned into a type constructor function:

$$\text{Coord} = \lambda\tau.\{x : \tau, y : \tau\}$$

which accepts one type parameter, τ. We can create actual coordinate-types by applying this function to different arguments representing the type we desire for the *x* and *y* fields, for example:

$$\text{IntegerCoord} = \text{Coord[Integer]} = \{x : \text{Integer}, y : \text{Integer}\}$$
$$\text{RealCoord} = \text{Coord[Real]} = \{x : \text{Real}, y : \text{Real}\}$$

It is clear, therefore, that a type constructor function in the λ-calculus is the formal equivalent of a generic type in Ada or Eiffel, and the process of instantiating a generic type is modelled by applying the type constructor function to an actual type argument.

In the polymorphic typed λ-calculus, one may write functions that accept *both* type-arguments *and* value-arguments. The convention is for the type-arguments to be introduced before the value-arguments, mainly because the values might be of one of the introduced types. The polymorphic function for constructing a generic coordinate is written:

$$\forall\tau \,.\, \text{makeCoord} : \tau \rightarrow \tau \rightarrow \text{Coord}[\tau]$$
$$= \lambda\tau \,.\lambda(a : \tau).\lambda(b : \tau).\{x \mapsto a, y \mapsto b\}$$

Notice how the type declaration (the first line, above) is prefixed by the universal quantification $\forall\tau$, meaning "for all types τ". Then, the rest of the declaration says that *makeCoord* accepts two arguments of the τ type and constructs a Coord[τ] from this. Notice also how the implementation (the second line, above) expects the first argument to be a type, and binds this to the type variable τ. Thereafter, the subsequent arguments *a* and *b* are expected to be values of this same τ type, and the result is a record whose *x* and *y* fields map to these values, so the type of the coordinate is clearly dependent on the type of the arguments. In the type signature of *makeCoord*, this type-dependency was expressed in the result-type as: *Coord*[τ], because the record-type of the resulting coordinate is actually generated by applying the type-function *Coord* to whatever type τ was supplied as the first argument. We can create coordinate instances of different types in the following way:

makeCoord[Integer](2)(3) - *ie apply to the Integer type, then to 2, then to 3*
$\Rightarrow \{x \mapsto 2, y \mapsto 3\}$
makeCoord[Real](2.1)(3.4) - *ie apply to the Real type, then to 2.1, then to 3.4*
$\Rightarrow \{x \mapsto 2.1, y \mapsto 3.4\}$

This demonstrates that *makeCoord* is a polymorphic function in Strachey's original sense, in that it can be applied uniformly to values of different types. It is a parametric-polymorphic function in Tennent's sense, since the unknown part of the coordinate type is modelled using a type parameter.

4 GENERIC OBJECT TYPES

In a similar way, any kind of generic type can be constructed by replacing some parts of a simple type by type parameters. In previous articles [1, 3] we have seen that object types are often recursive, because their methods may accept or return objects of the same type. A recursive, simply-typed *IntegerStack* type can be written:

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

In this, $\mu\sigma$ introduces the recursion in the type and σ stands for the eventual *IntegerStack*, in the body. We may generalise this definition to create a generic *Stack* type constructor if we replace occurrences of *Integer* by a type parameter τ:

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

Here, $\lambda\tau$ introduces the parameter τ, standing for the element-type, ahead of $\mu\sigma$, which binds the recursion. This generic *Stack* definition has the form of a type function, which expects a type argument: τ and then returns a result, a recursive 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):

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

to see how this yields a recursive record type exactly like *IntegerStack*, above. We could also construct *Stack[Real]*, *Stack[Boolean]* and other types of *Stack*, each with different substitutions for the element-type τ.

Readers who have been following this series will know that the notation $\mu\sigma$ is actually a short-hand for constructing a recursive type from first principles, using a type generator [1]. To define a generic *Stack* from first principles, we need a type generator *GenStack*, which introduces the self-type argument σ as well as the element-type argument τ:

$$\begin{aligned}\text{GenStack} = \lambda\tau.\lambda\sigma.\{&\text{push} : \tau \to \sigma,\ \ \text{pop} : \to \sigma,\ \ \text{top} : \to \tau,\\ &\text{empty} : \to \text{Boolean},\ \ \text{size} : \to \text{Integer}\}\end{aligned}$$

GenStack is a type function accepting *two* type arguments. The order of introduction is significant: it is important to introduce the element-type τ before the self-type σ. This is because we want the element-type τ to be in scope when the self-type σ is declared. As a consequence, σ stands for the "whole of the self-type".

The relationship between *GenStack* and the generic *Stack* above is straightforward, but difficult to see at first. The order of parameters expects you to supply an element type

first, then to take the fixpoint of the resulting generator. For example we can create a fully-instantiated, recursive *RealStack* type by supplying {*Real*/τ} and then taking the fixpoint:

RealStack = (**Y** GenStack[Real])
⇒ {push : Real → RealStack, pop : → RealStack, top : → Real,
empty : → Boolean, size : → Integer}

This works because *GenStack[Real]* yields a generator of the form: λσ.{…} whose fixpoint can then be taken with **Y**, so binding σ recursively over the rest of the record. To create the generic *Stack* type, we somehow need to fix the recursion of σ without replacing the element-type parameter τ with any actual type. The trick is to re-introduce the parameter on the outside of the fixpoint:

Stack = λτ' . (**Y** GenStack[τ'])
= λτ'.μσ.{push : τ' → σ, pop : → σ, top : → τ',
empty : → Boolean, size : → Integer}

and this yields a type constructor function exactly like the *Stack* constructor above. The only difference here is that we supplied the new parameter {τ'/ τ} before taking the fixpoint, instead of some actual type, as in the *RealStack* example.

5 GENERIC CLASSES

The generic *Stack* above may best be described as a *generic type*, but not as a *generic class*. It is only a generic type, because the recursion of the self-type is fixed and the self-type cannot therefore evolve further under inheritance. A generic class may be defined by keeping the self-type open to extension. In this and the following sections, we shall develop a family of *List* classes, looking at how the typeful aspects evolve, but we will skip over the details of their implementations, for simplicity's sake.

Recall that a class is a family of types which all share some common structure, a minimum set of common methods [3, 4]. The class constraint is expressed using a bounded parameter, a type parameter with a restriction on the types which can replace it. For example, if all *Numbers* have at least a *plus* method, we can define a type generator for this record type:

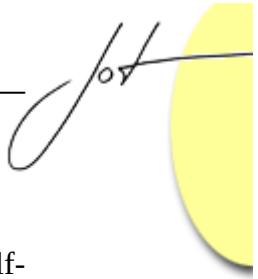
GenNumber = λσ.{plus : σ → σ}

and then express the class of *Numbers* using the generator function in the constraint, which is known as a *function bound*, or *F-bound* [15]:

∀(σ <: GenNumber[σ]) . σ

This says, "for all types σ that have at least as many methods as *GenNumber*[σ], σ is the entire class of numbers". This is how to express the membership of an ordinary class.

A generic class can be defined in the same way, using an F-bound. To define the base *List* class in the hierarchy, we first need to declare a type generator:



$$\text{GenList} = \lambda\tau\,.\lambda\sigma.\{\text{cons} : \tau \to \sigma,\ \text{head} : \to \tau,\ \text{tail} : \to \sigma,\ \text{equal} : \sigma \to \text{Boolean}\}$$

This is a type function with two type arguments: τ is the element-type and σ is the self-type, introduced within the scope of τ. The F-bound is constructed in a slightly more elaborate way, which takes the element-type into account:

$$\forall\tau\,.\forall(\sigma <: \text{GenList}[\tau][\sigma])\,.\,\sigma$$

This says, "for all element-types τ, and for all list-types σ that have at least as many methods as the type *GenList*[τ][σ], σ is that entire class of lists". The new aspect here is that the F-bound is expressed in terms of both τ and σ. This is because we must apply *GenList* to two type-arguments in order to release the record type in its body.

To validate this new kind of F-bound, describing the membership of a generic class, we shall define an actual list type that we expect to be in the class. To make things a little more difficult, this list type will have an extra *size* method, and a particular (instantiated) element type *Integer*. We shall call this type *IntSzList*, in recognition of the above. Its full type definition is given by:

$$\begin{aligned}\text{IntSzList} = \mu\sigma.\{&\text{cons} : \text{Integer} \to \sigma,\ \text{head} : \to \text{Integer},\ \text{tail} : \to \sigma,\\ &\text{equal} : \sigma \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\\ \Rightarrow \{&\text{cons} : \text{Integer} \to \text{IntSzList},\ \text{head} : \to \text{Integer},\ \text{tail} : \to \text{IntSzList},\\ &\text{equal} : \text{IntSzList} \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\end{aligned}$$

The real question is whether *IntSzList* is a member of the generic *List* class. To test this conjecture, we substitute {*Integer*/τ} and {*IntSzList*/σ} in the formula given above. This simplifies to the comparison:

$$\begin{aligned}&\text{IntSzList} <: \text{GenList}[\text{Integer}][\text{IntSzList}]\\ \Rightarrow\ &\{\text{cons} : \text{Integer} \to \text{IntSzList},\ \text{head} : \to \text{Integer},\ \text{tail} : \to \text{IntSzList},\\ &\qquad\text{equal} : \text{IntSzList} \to \text{Boolean},\ \text{size} : \to \text{Integer}\}\\ &\quad <:\ \{\text{cons} : \text{Integer} \to \text{IntSzList},\ \text{head} : \to \text{Integer},\ \text{tail} : \to \text{IntSzList}\\ &\qquad\text{equal} : \text{IntSzList} \to \text{Boolean}\}\\ \Rightarrow\ &\text{true, by record subtyping.}\end{aligned}$$

thereby demonstrating that *IntSzList* is a member of the class of generic *Lists*.

6 GENERIC INHERITANCE

Is it possible to introduce and adapt generic classes during inheritance? In practical object-oriented languages that combine generic polymorphism with subclassing, you can:

- introduce a subclass with extra type parameters, especially when the need to express genericity first arises in the hierarchy; and
- introduce a subclass with fewer type parameters, by instantiating some of the parent's parameters in the subclass.

The first property is necessary to allow generic classes to exist within the same class hierarchy as ordinary classes. The second property is necessary to allow specific subclass

instantiations of generic classes. We shall seek to demonstrate both these properties in the model, by seeing if we can adapt generators for generic classes from other generators.

First, we shall model the introduction of a generic class which inherits from a non-generic parent class. Let us assume that the class hierarchy has a root *Object* class with an *equal* method, as defined by the generator:

$$\text{GenObject} = \lambda\sigma.\{\text{equal} : \sigma \rightarrow \text{Boolean}\}$$
$$\forall(\sigma <: \text{GenObject}[\sigma]) \,.\, \sigma \qquad \text{-- } \textit{is the class of all Objects}$$

We wish to introduce our *List* subclass, that is, a family of generic lists with equality. It is relatively easy to define the generator *GenList* for this class by adapting *GenObject*:

$$\text{GenList} = \lambda\tau.\lambda\sigma.(\mathbf{GenObject[\sigma]} \cup \{ \text{cons} : \tau \rightarrow \sigma, \text{head} : \rightarrow \tau, \text{tail} : \rightarrow \sigma\})$$
$$\Rightarrow \lambda\tau.\lambda\sigma.(\text{equal} : \sigma \rightarrow \text{Boolean}, \text{cons} : \tau \rightarrow \sigma, \text{head} : \rightarrow \tau, \text{tail} : \rightarrow \sigma\}$$
$$\forall\tau \,.\forall(\sigma <: \text{GenList}[\tau][\sigma]) \,.\, \sigma \qquad \text{-- } \textit{is the class of all Lists}$$

because the new self-type of the list, σ, can be passed back as an argument to the *GenObject* generator (see bold highlight), such that the inherited equal method's self-type is adapted to the new self-type. Because we introduced σ inside the scope of τ, the new self-type implicitly stands for the "whole of the self-type" of the list, including the fact that it contains elements of the τ type. So, we have successfully demonstrated the introduction of a generic class.

To demonstrate the second property, we need to be able to define a subclass (with possibly extra methods) that also instantiates the generic element-type during inheritance. For this, we will introduce the generator *GenIntSzList* for a class of lists of *Integers*, with an additional *size* method. This generator will be defined by adapting the *GenList* generator, which has the extra element-type parameter τ, but the subclass generator will not have this, since it will have been instantiated by the *Integer* type.

$$\text{GenIntSzList} = \lambda\sigma.(\mathbf{GenList[Integer][\sigma]} \cup \{ \text{size} : \rightarrow \text{Integer}\})$$
$$\Rightarrow \lambda\sigma.(\text{equal} : \sigma \rightarrow \text{Boolean}, \text{cons} : \text{Integer} \rightarrow \sigma, \text{head} : \rightarrow \text{Integer},$$
$$\text{tail} : \rightarrow \sigma, \text{size} : \rightarrow \text{Integer}\}$$
$$\forall(\sigma <: \text{GenIntSzList}[\sigma]) \,.\, \sigma \qquad \text{-- } \textit{is the class of all IntSzLists}$$

The *GenIntSzList* generator clearly only has the self-type parameter, so it is no longer a generator for a generic class. The generic parameter τ was instantiated when *Integer* was supplied as one of the type arguments passed back to the *GenList* generator (see bold highlight), such that the inherited part of the record type has {*Integer*/τ} substituted everywhere. So, we have successfully demonstrated the removal of genericity during the operation of inheritance. This close integration of generic classes with inheritance and with old-fashioned type constructors, like Pascal's SET OF... was first demonstrated by Simons [16, 17], who also showed the important formal property of *confluence*. This property allows the same type to be derived either by instantiating, then inheriting; or by inheriting, then instantiating the parameters, and is an important symmetry property.

7 CONSTRAINED GENERICITY

The template types of C++ are exactly modelled by the universally-quantified type parameters provided in the Girard-Reynolds approach to polymorphism. This is because no restriction is placed on the possible types that might instantiate the parameters: the quantification $\forall\tau$ literally means "for all types τ". In practice, if you supply an unsuitable type for a type parameter in C++, this is not detected until the compiler generates a separate image for the instantiated code, because the compiler cannot check template class declarations.

In Eiffel, it is possible to check at the point of type-substitution whether suitable types are being supplied for a type parameter. This is because Eiffel also allows the expression of constraints on the type parameter, of the form: *SortedList [T → Comparable]*, meaning a *SortedList* of any element type *T* that conforms to the *Comparable* class. This is a more expressive kind of parametric polymorphism, since it allows a compiler to check the code for a generic class, before it is instantiated. All the calls made on variables of parametric type *T* can be checked, because we know that *T* is at least of the *Comparable* type.

Fortunately, the concept of restricting a type parameter to a certain family of types is captured exactly by an F-bound, which we have used so far to constrain the family of types in a class. It is particularly satisfying to find that F-bounds can also be used to model constrained generic types [17, 18]. To define the *SortedList* above, we first need to define a generator for the *Comparable* class, assuming that this supplies the methods *lessThan* and *equal*:

$$\text{GenComparable} = \lambda\sigma.\{\text{lessThan} : \sigma \to \text{Boolean}, \text{equal} : \sigma \to \text{Boolean}\}$$

The generator for a *SortedList* defines the operations that you would expect in such a list, such as an (ordered) *insert* operation, and *first* to extract the element at the head of the list.

$$\text{GenSortedList} = \lambda\tau\ .\lambda\sigma.\{\text{insert} : \tau \to \sigma, \text{first} : \to \tau, \text{rest} : \to \sigma\}$$

Finally, the F-bound can be constructed, to express the family of all those types that belong in the class of *SortedLists*:

$$\forall(\tau <: \text{GenComparable}[\tau]).\forall(\sigma <: \text{GenSortedList}[\tau][\sigma])\ .\ \sigma$$

This says, "for all those element-types τ which have at least the methods of *GenComparable*[τ], and then for all those list-types that have at least the methods of *GenSortedList*[τ][σ], σ is the entire class of sorted lists". This captures exactly Eiffel's notion of a generic class which has a *constrained generic type* parameter.

8 CONCLUSION

We have shown how *parametric polymorphism*, also known as *templates* in C++ and *genericity* in Ada and Eiffel, can be added to the Theory of Classification. We demonstrated how generic types could be created by abstracting over parts of simple types. A generic type is modelled as a type function expecting an actual type argument. We then extended this to model generic classes. A generic class is modelled by first creating a special type function, called a type generator, which has both element-type and self-type parameters. The notion of a generic class is formally all those types which satisfy the F-bound, expressed using the generator. We then showed how the generators for generic classes are well-behaved under inheritance, and can be extended at the same time as introducing, or instantiating the generic type parameters.

F-bounds have been especially useful in this aspect of the Theory of Classification. Cook originally used F-bounds just to model the self-types of classes and explain how these were modified under inheritance [15]. Simons integrated this use of F-bounds with generic classes in his Theory of Classification [16, 17], finding that the same modelling concept could be used everywhere. In a later paper, he also showed how all three of Eiffel's typing mechanisms (conformance, type anchors and constrained genericity) could be modelled by F-bounds, demonstrating the economy and power of the theory [18].

REFERENCES

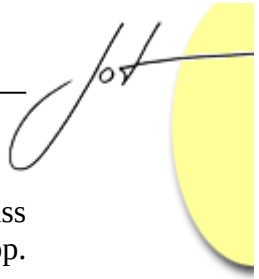
[1] A J H Simons, "The theory of classification, part 3: Object encodings and recursion", in *Journal of Object Technology*, vol. 1, no. 4, September-October 2002, pp. 49-57. http://www.jot.fm/issues/issue_2002_09/column4

[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://ww.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] A J H Simons, "The theory of classification, part 8: Classification and inheritance", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 55-64. http://www.jot.fm/issues/issue_2003_07/column4

[5] A J H Simons, "The theory of classification, part 9: Inheritance and self-reference", in *Journal of Object Technology*, vol. 2, no. 6, November-December 2003, pp. 25-34. http://www.jot.fm/issues/issue_2003_11/column2



[6] A J H Simons, "The theory of classification, part 12: Building the class hierarchy", in *Journal of Object Technology*, vol. 3, no. 5, May-June 2004, pp. 13-24. http://www.jot.fm/issues/issue_2004_05/column2

[7] C Strachey, *Fundamental Concepts of Programming Languages*, Oxford University: Programming Research Group, 1967.

[8] C Strachey, *Varieties of Programming Languages*, Oxford University: Programming Research Group, 1973.

[9] R Milne and C Strachey, *A Theory of Programming Language Semantics*, London: Chapman and Hall, 1976.

[10] R D Tennent, *Principles of Programming Languages*, Prentice Hall, 1981.

[11] R Milner, "A theory of type polymorphism in programming", in *J. Computer and System Sciences, 17*, (1978), pp. 48-375.

[12] J Ichbiah, J Barnes, J Heliard, B Krieg-Bruckner, O Roubine and B Wichmann, "Rationale and design of the programming language Ada", in *ACM Sigplan Notices, 14(6)*, (1979).

[13] J-Y Girard, "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur", *PhD Thesis*, Université Paris VII, (1972).

[14] J Reynolds, "Towards a theory of type structure", *Proc. Coll. Prog., New York, LNCS 19* (Berlin: Springer Verlag, 1974), pp. 408-425.

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

[16] A J H Simons, "A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language", *PhD Thesis*, Department of Computer Science, University of Sheffield (1995).

[17] A J H Simons, "A theory of class", in *Proc. 3rd Int. Conf. Object-Oriented Info. Sys.*, eds. D Patel, Y Sun and S Patel, (London: Springer Verlag, 1996), pp. 44-56.

[18] A J H Simons, "Rationalising Eiffel's type system", in *Proc. 18th Conf. Technology of Object- Oriented Languages and Systems (TOOLS Pacific)*, eds. C. Mingins, R. Duke and B. Meyer (Melbourne, 1995), pp. 365-377.

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.