Article:
Simons, A.J.H. (2005) The theory of classification part 18: The theory of classification part 18: polymorphism through the looking glass. The Journal of Object Technology, 4 (4). pp. 7-18. ISSN: 1660-1769

https://doi.org/10.5381/jot.2005.4.4.c1

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 4, May–June 2005

The Theory of Classification Part 18: Polymorphism through the Looking Glass

Anthony J H Simons, Department of Computer Science, University of Sheffield

1 INTRODUCTION

In this, the eighteenth article in a regular series on object-oriented type theory, we look at how object-oriented languages might evolve in the future, given that the formal notion of *class* is now better understood than at the outset. Object-oriented languages were the first family to suppose that there might be systematic sets of relationships between all the program data types and use this as the basis for a kind of type compatibility. However, the early formal models chosen were based on simple types and subtyping [1] and struggled in practice to support all the obvious, systematic relationships that programmers intuitively recognised [2]. For a while, objects were thought to have *class* and *type* independently, where *class* was demoted to a mere implementation construct. Later, it was realised that the notion of *class* is also a typeful construct that requires at least a bounded second-order λ-calculus model to explain it [3]. We have developed this model in the *Theory of Classification*, showing how it deals properly with typed inheritance [4, 5] and generic types [6] in a consistent framework.

However, current object-oriented languages fall short of what is actually possible in a language that *really* supports the notion of class. The majority still treat classes for the most part as if they were the same thing as simple types, and it only becomes clear that something more sophisticated is intended when dynamic binding in these languages is examined, showing dispatching behaviour equivalent to higher-order functions [7]. The additional template mechanisms of C++ and Java (from version 1.5) are intended partly to compensate for the lack of expressiveness caused by treating classes as simple types. But do we really need all these separate typing mechanisms? What would a language look like that consistently supported the higher-order notion of *class* throughout?

Cite this column as follows: Anthony Simons: "The Theory of Classification: Part 18: Polymorphism through the Looking Glass", in *Journal of Object Technology*, vol. 4, no. 4, May - June 2005, pp 7-18 http://www.jot.fm/issues/issue_2005_05/column01

2 THE HALFWAY HOUSE

In the very first article of this series [8], we described three increasingly more flexible kinds of plug-in type compatibility, in the context of supplying a component to match an interface:

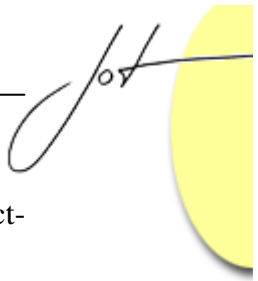
- correspondence: the component is identical in type and its behaviour exactly matches the expectations made of it when calls are made through the interface;
- subtyping: the component is a more specific type, but behaves exactly like the more general expectations when calls are made through the interface;
- subclassing: the component is a more specific type and behaves in ways that exceed the more general expectations when calls are made through the interface.

An example of *correspondence* is the strong monomorphic typing exhibited in languages like Pascal or Modula-2, in which every object is of a single type and may only be passed to variables of exactly the same type. Pascal's strong *name equivalence* rule means that even structurally equivalent types are considered distinct, if their type names are distinct (in contrast to C++'s *typedefs*, which are only aliases for the base type).

An example of *subtyping* is where a subrange object is coerced to a base type variable, so that the base type's function may be executed, such as where two *SmallInt* objects are passed to an *Integer plus* function and the result is returned as an *Integer*. The function originally expected *Integer*s, but could handle subtypes of *Integer* and convert them. Note that no dynamic binding is implied or required. Also, a simply-typed first-order calculus (with subtyping) is adequate to explain this behaviour.

An example of *subclassing* is where the functions of the interface are systematically replaced by functions appropriate to the new type, such as where a *Numeric* type's abstract functions *plus, minus, times* and *divide*, are replaced by retyped versions for a *Complex* type. Rather than coerce a *Complex* object to a *Numeric*, the call to *plus* through *Numeric* executes the replacement *Complex plus* function. This could be achieved through dynamic binding; or alternatively through template instantiation (in which the parameter *Numeric* is replaced throughout by an actual *Complex* type), requiring at least a second-order calculus.

What are the important differences between the simple subtyping and subclassing approaches? In the subtyping approach, the *Integer plus* function treats its *SmallInt* arguments exactly as if they were plain *Integers*. It returns a result of the general type *Integer* and does not know or care whether the result is still in the range of a *SmallInt*. On the other hand, in the subclassing approach, there is an obligation to propagate type information about the actual argument and result types of *Complex's plus* back to the call-site. Whereas the interface expected a *Numeric*, once this was bound to a *Complex* number, the second argument was also forced to be a *Complex* number. Furthermore, the result-type, which was formerly *Numeric*, is now known also to be of the *Complex* type. This means that the caller of *plus* must know how to deal with more specific types than originally specified in the interface. From our point of view, this is exciting stuff, in the



true spirit of classification, and something worth exploiting in the design of object-oriented languages.

However, the majority of languages only practise a halfway-house approach, which is *subtyping* with *dynamic binding*. This is similar to subtyping, except that the subtype may provide a replacement function that is executed instead. Recalling the earlier example, this is like the *SmallInt* type providing its own version of the *plus* function which wraps the result back into the *SmallInt* range. Syntactically, the result is acceptable as an *Integer*, but semantically it may yield different results from the original *Integer plus* function (when wrap-around occurs). From the type perspective, we still only know that the result is of the *Integer* type (rather than *SmallInt*) because there is no way of propagating type information about the actual arguments through to the result of the function. So, we have a situation where more specific functions are executed, but externally we cannot see that their type has changed. This gives rise to the phenomenon of "type loss" in C++ and Java, requiring corrective use of type downcasting to recover the most specific types of returned objects [2].

3 POLYMORPHISM REVISITED

Stopping at the halfway house constitutes a failure of nerve in the design of object-oriented languages. At the heart of this problem is the inability to distinguish the notions of *class* and *type* in the syntax of programming languages. If an object-oriented language implemented this distinction properly, a programmer should never have to perform type downcasting, but the language could always recover the most specific types of returned objects for itself. To make this distinction absolutely clear, in the *Theory of Classification*:

- a *type* always refers to a simple monomorphic type, a first-order construct;
- a *class* always refers to a polymorphic type, a second-order (or higher) construct.

As stated previously [8], the term *polymorphism* has less to do with the dynamic binding of methods and properly describes the generalised types of variables that may receive values of more than one type. In conventional programming languages, we consider that type constructors, such as *Stack[T]* or *Map[K,V]* are polymorphic, because they contain type variables standing for possibly many types, and may be adapted to specific types by parameter instantiation. In object-oriented languages, we also consider that a variable of "class-type" is polymorphic and can be made to receive actual objects of possibly many types, where these are restricted by the class hierarchy to be of some "subclass-type" of the target variable. These polymorphic mechanisms seem on the surface to be quite different, but they are fundamentally the same.

In the λ-calculus, polymorphism always requires a type parameter, standing for the generalised type; and when a polymorphic variable binds to a specific type, this type is propagated into the parameter, throughout the whole parameterised expression. The

formal model therefore brings together the notions of class-based polymorphism and template-based polymorphism. In earlier articles [2, 3], we deliberately drew out the similarity between classical Girard-Reynolds [9, 10] univeral polymorphism:

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

$$\forall\tau \,.\, \text{insert} : \tau \times \text{List}[\tau] \to \text{List}[\tau]$$

in which you could give functions truly generalised types (where τ ranges over absolutely any type) and Cook et al.'s [11, 12] function bounded polymorphism:

$$\forall(\tau <: \text{GenNumeric}[\tau]) \,.\, \text{plus} : \tau \times \tau \to \tau$$

$$\forall(\tau <: \text{GenComparable}[\tau]) \,.\, \text{insert} : \tau \times \text{SortedList}[\tau] \to \text{SortedList}[\tau]$$

in which you could give functions class-types (where τ ranges over *only* those types which have at least the functions specified in the interface of the bounding generator function). F-bounded polymorphism is more general than universal polymorphism (since you can type more things using F-bounds, for example you can type *SortedLists* of *Comparable* elements with F-bounds, whereas you can only type plain *Lists* of universal elements, without them). This can be shown formally by recasting Girard-Reynolds polymorphism as a special case of F-bounded polymorphism:

$$\text{GenUniversal} = \lambda\sigma.\{\} \qquad \text{// the content-free constraint}$$

$$\forall(\tau <: \text{GenUniversal}[\tau]) \,.\, \text{identity} : \tau \to \tau$$

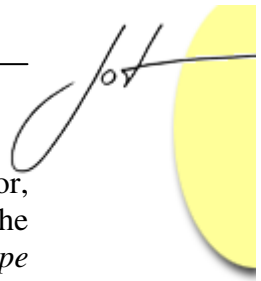
$$\forall(\tau <: \text{GenUniversal}[\tau]) \,.\, \text{insert} : \tau \times \text{List}[\tau] \to \text{List}[\tau]$$

That is, we constrain τ to range over those types which have at least the functions of the universal interface, but this interface is trivial (empty), so τ ranges again over any type.

There are two practical consequences of this discussion. The first is that, wherever a polymorphic variable is required in our programming language, we should always model its type using some kind of type parameter in the formal calculus. The fact that object-oriented languages don't make the type parameters explicit for their classes is one of the reasons why the notions of *class* and *type* get so confused. The second is that we do not need separate mechanisms to explain template-based and class-based polymorphism. The class parameters constrained by F-bounds are adequate for both purposes [6], being more general than classical unconstrained parameters.

4 DISTINGUISHING CLASS AND TYPE

In current object-oriented languages, objects and variables are "typed" using the names of the classes like type identifiers. These identifiers are used ambiguously, to describe either



an object or value with an exact type (a monomorphic *type* in the theory), or, alternatively, to describe a variable with a flexible type (a polymorphic *class* in the theory). What we should like is for object-oriented languages to indicate a *class*, or a *type* unambiguously.

Informally, it is possible to infer the intended semantics of class identifiers from the program context in which they appear. In a C++ or Java-like language, when we create an object, we usually intend to create something with a fixed implementation and an exact type:

```
… = new Point;                  // exactly typed object creation
```

In this context, we do not expect to obtain some instance of a subclass of *Point*, but rather an exact instance of the exact type *Point*. On the other hand, when we declare a program variable of the *Point* class, it is clear that we intend this to be flexible, capable of receiving values that might be more specific than a *Point*, but which are at least of this class:

```
Point p = …                     // polymorphic variable declaration

accept(Point p) { … }           // polymorphic method arguments
```

In this context, we do not expect these variables to be restricted to accepting only objects of the exact *Point* type, but rather any type which is at least a subclass of *Point*. We can model these differences in the λ-calculus, to make them explicit.

Recall that a class is defined essentially as a flexible, open-ended implementation, parameterised by *self*, with a corresponding polymorphic type, parameterised by σ, the self-type [4]. We give the type-shape of the class using a type generator, followed by the implementation-shape using an object generator, which is typed using the type generator as the F-bound, restricting what types may eventually replace the self-type:

$$\text{GenPoint} = \lambda\sigma.\{x : \text{Integer}, y : \text{Integer}, \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

$$\text{genAPoint} : \forall(\tau <: \text{GenPoint}[\tau]).\tau \rightarrow \text{GenPoint}[\tau]$$
$$\text{genAPoint} = \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(\text{self} : \tau).$$
$$\{ x \mapsto 2, y \mapsto 3, \text{equal} \mapsto \lambda(q : \tau).(\text{self}.x = q.x \wedge \text{self}.y = q.y)\}$$

In our C++ or Java-like programming language, when we declare a variable of the *Point* class, what we are really asserting is the polymorphic typing *p0* : τ, where τ is a type parameter constrained to range over any type in the *Point* class:

Programming Language	**Formal Model**
Point p;	p0 : ∀(τ <: GenPoint[τ]) . τ

Table 1: Polymorphic variable declaration

On the other hand, when we create an exact instance of the *Point* type, we must fix both the type and the implementation. In the calculus, this is done by taking the fixpoint of the type generator and of the object generator [4]:

$$\begin{aligned}
&\text{Point} = \mathbf{Y}\ \text{GenPoint}\\
&= \mu\sigma.\{x : \text{Integer}, y : \text{Integer}, \text{equal} : \sigma \to \text{Boolean}\}\\
&\Rightarrow \{x : \text{Integer}, y : \text{Integer}, \text{equal} : \text{Point} \to \text{Boolean}\}, \quad \text{after unrolling;}
\end{aligned}$$

$$\begin{aligned}
&\text{aPoint} = \mathbf{Y}\ \text{genAPoint}[\mathbf{Y}\ \text{GenPoint}] = \mathbf{Y}\ \text{genAPoint}[\text{Point}]\\
&= \mathbf{Y}\ \lambda(\text{self} : \text{Point}).\{\ x \mapsto 2, y \mapsto 3,\\
&\qquad\qquad \text{equal} \mapsto \lambda(q : \text{Point}).(\text{self}.x = q.x \wedge \text{self}.y = q.y)\}\\
&= \mu(\text{self} : \text{Point}).\{\ x \mapsto 2, y \mapsto 3,\\
&\qquad\qquad \text{equal} \mapsto \lambda(q : \text{Point}).(\text{self}.x = q.x \wedge \text{self}.y = q.y)\}\\
&\Rightarrow \{\ x \mapsto 2, y \mapsto 3, \text{equal} \mapsto \lambda(q : \text{Point}).(\text{aPoint}.x = q.x \wedge \text{aPoint}.y = q.y)\},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{after unrolling.}
\end{aligned}$$

This creates the exact instance *aPoint* of the exact *Point* type. We can therefore model the meaning of object creation expressions in our programming language:

Programming Language	**Formal Model**
new Point;	p1 = **Y** genAPoint[**Y** GenPoint]; p1 : Point

Table 2: Exactly-typed object creation

Here, we have taken the liberty of introducing the temporary variable *p1* in the formal model, so that we can initialise this variable to the rather complex object creation expression and then see that it has an exact type, which is the *Point* type we expected. The temporary variable is simply a convenience, to save repeating longer expressions. In section 6 below, we will use a similar approach to analyse program behaviour in step-by-step detail.

5 TYPE CHECKING WITH FIRST-ORDER TYPES

First, we shall introduce a test-case that exemplifies some of the difficulties identified with type systems that check types in the first-order model (with simple types and subtyping). The example code fragment, expressed in our C++ or Java-like language, is a cut-down version of the infamous “Eiffel type failure” problem first identified by Cook [13]:

```
Point p = new Point3D;        // alias a more specific Point3D
Point q = new Point;          // create a standard Point
Boolean b = p.equal(q);       // dynamically invoke the specific equal
```

Programmers expect a *Point3D* instance to be type-compatible with a *Point* variable, but in the first order model, this is not the case. To explain why the above fragment is problematic, we should define the *Point3D* class, which describes a three-dimensional point, whose interface extends that of a standard two-dimensional *Point*:

$$\text{GenPoint3D} = \lambda\sigma.\{x : \text{Integer}, y : \text{Integer}, z : \text{Integer}, \text{equal} : \sigma \to \text{Boolean}\}$$

$$\text{genAPoint3D} : \forall(\tau <: \text{GenPoint3D}[\tau]).\tau \to \text{GenPoint3D}[\tau]$$
$$\text{genAPoint3D} = \lambda(\tau <: \text{GenPoint3D}[\tau]).\lambda(\text{self} : \tau).$$
$$\{x \to 2, y \mapsto 3, z \mapsto 5,$$
$$\text{equal} \mapsto \lambda(r : \tau).(\text{self.x} = \text{r.x} \wedge \text{self.y} = \text{r.y} \wedge \text{self.z} = \text{r.z})\}$$

In particular, an instance *aPoint3D : Point3D*, created from these generators by taking the fixpoints (see section 4) will have an extra *z* field; and when *aPoint3D* tests itself for equality against another point, it will compare all of its *x, y* and *z* fields.

In the original "type failure" scenario, the programming language expected the subtyping relationship *Point3D* <: *Point* to hold. In fact, we now know that these types are *not* in a subtyping relationship, because the retyping of *Point3D*'s *equal* method violates the function subtyping rule [1]. However, Eiffel allowed subclasses to retype their methods with more specific argument types, since it is unlikely in practice that we should want a *Point3D* to compare itself with more general kinds of point.

An undetected type failure arises as follows. First, we create a specific *Point3D* instance and assign it (by polymorphic aliasing) to the more general variable *p : Point*. This is permitted by the (faulty) assumption that *Point3D* <: *Point*. Then, we create another instance *q : Point*. Finally, we invoke *p.equal(q),* at which moment the undetected type failure occurs. Statically, the type of *equal* is *Point.equal : Point* → *Boolean*, so it appears to be legal to pass in the given argument *q : Point*. However, *p* currently contains a dynamic instance of *Point3D* and the version of the *equal* method which is actually invoked is *Point3D.equal : Point3D* → *Boolean*. This receives the too-general argument *q : Point*, and during the execution of the method body, an attempt is made to access the *z* field of a plain *Point*, which will cause the program to crash, generating a memory segmentation fault.

Cook originally proposed to fix this problem by forcing Eiffel to conform to strict subtyping rules [13]. Redefined argument types for *equal* would therefore not be allowed. Although this technically satisfies subtyping, we have seen how this results in a strictly less expressive language [2]. In particular, the *equal* method, which is required by every class, may only be typed with the most general kind of argument (usually, the root class *Object*), and it may never be retyped with more restricted types of argument. Instead, redefined versions of *equal* have to accept *Object* arguments and use runtime-checked type downcasting internally, to recover the more specific dynamic type of the argument,

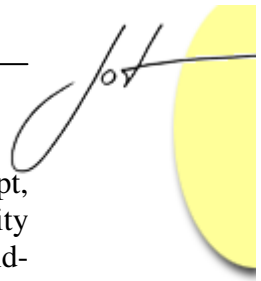
before comparison can be made. This merely pushes the type failure problem back into the run-time.

6 TYPE CHECKING WITH SECOND-ORDER CLASSES

Classes are type-recursive, meaning that their methods often accept or return arguments of the *self*-type. So it is natural to want these arguments and results to become uniformly specialised along with the class itself. We want to allow a *Point3D* to specialise the argument type of its *equal* method. However, we still want to avoid unchecked type failures.

Programming Language	**Formal Model**
Point p …	p0 : ∀(τ <: GenPoint[τ]).τ
… new Point3D;	p1 = **Y** genAPoint3D[**Y** GenPoint3D]; p1 : Point3D
Point p = new Point3D;	p2 = (p0 := p1); {Point3D / τ } p2 : Point3D
Point q …	q0 : ∀(σ <: GenPoint[σ]).σ
… new Point;	q1 = **Y** genAPoint [**Y** GenPoint]; q1 : Point
Point q = new Point;	q2 = (q0 := q1); {Point / σ } q2 : Point
Boolean b …	b0 : Boolean
… p.equal …	p2 : Point3D; p2.equal : Point3D → Boolean
… p.equal(q);	q2 : Point; p2.equal : Point3D → Boolean; p2.equal(q2 : Point) : ERROR Point ≠ Point3D

Table 3: Polymorphic checking with type substitution



In the *Theory of Classification*, we take the view that a class is not a first-order concept, but a second-order, polymorphic concept. One of the advantages this brings is the ability to relate closed recursive types to each other, by relating their generators in a (second-order) *pointwise* subtyping relationship [3]. This allows us to specialise argument and result types uniformly, in line with programmers' intuitions about classes. However, the recursive types themselves do not enter into simple subtyping relationships, so we cannot type-check them in the usual first-order system. By properly distinguishing the polymorphic notion of *class* from the monomorphic notion of *type*, we may type-check the same fragment of object-oriented code in a second-order model, showing that polymorphic assignment really involves the *propagation of types* into polymorphic type parameters. This is a very powerful checking mechanism, capable of resolving many of the difficulties formerly identified with object-oriented type systems.

On the left-hand side of table 3, the expressions in the programming language are broken down into small steps, in order to examine the types of these expressions in the formal model on the right-hand side. On the first row, we declare a polymorphic *Point* variable and show this to have a F-bounded parametric type. On the second row, we create an exact *Point3D* object and show this to have the exact *Point3D* type. On the third row, we assign the specific instance to the general variable. This is where the new type-checking principle first comes into play. At the moment of polymorphic aliasing, the exact type of the object is propagated into the type parameter of the variable, shown by the substitution: {Point3D / τ}. As a consequence, we obtain a new context *p2* after the assignment (*p0 := p1*), in which the type of the bound variable expression has been updated.

This is how the type mismatch is eventually detected. When checking the program expression: *p.equal(q)*, the model can predict the type of the *equal* method, and its expected argument type, since statically it knows that this is selected from *p2*. At the same time, the model knows the type of the actual argument, from the context *q2*. The formal and actual argument types are shown to conflict (*Point* ≠ *Point3D*), so the checker can raise a type mismatch at compile time. Not only do we spot the type error at compile time, but we do this without having to restrict the expressiveness of the language. We still allow *Point3D* objects to be passed into polymorphic variables $p : \forall(\tau <: GenPoint[\tau]).\tau$, so long as this does not conflict with other typing requirements further down the line. For example, the following code fragment is readily accepted by this checking algorithm:

```
Point p = new Point3D;        // alias a more specific Point3D
Point3D q = new Point3D;      // create a specific Point3D
Boolean b = p.equal(q);       // dynamically invoke the specific equal
```

since, at the moment of selection, the *equal* method has the type *Point3D.equal : Point3D* → *Boolean*. As a consequence, it can happily accept the actual argument *q : Point3D*. The following code fragment is also acceptable:

```
Point p = new Point3D;        // alias a more specific Point3D
Point q = new Point3D;        // alias another specific Point3D
Boolean b = p.equal(q);       // dynamically invoke the specific equal
```

because the type substitution {*Point3D* / τ <: *GenPoint*[τ]} is made consistently when both *p* and *q* alias values of exact types, before the typing of the *equal* method invocation is considered. Though the variables *p, q* were originally declared with general polymorphic types, new type contexts are established by the polymorphic aliasing.

7 CONCLUSION

Any object-oriented language that truly supports the notion of *class* should be able to distinguish contexts where simple types, or polymorphic classes are intended. The secret to success is to preserve the underlying type parameter in expressions where polymorphism is intended. When polymorphic variables alias each other, this has the effect of substituting one type parameter for another, possibly strengthening the F-bound constraint (this is because unifying two type variables requires that you accept the more restricting of the two type constraints – see the previous discussion on *intersection types* in [5]). When polymorphic variables alias objects with exact types, these types are substituted into the type parameters. As a consequence, it is always clear whether an expression has a polymorphic, or fixed type, in a given context.

Object-oriented lanaguages that adopted this simple rule could remove a lot of clutter from their syntax. To start with, there would be no need to have both this kind of (genuine) polymorphic typing *and* subtyping. So, type checkers that performed parametric substitutions would not also have to perform subtyping coercions. If two simple types turned out not to be the same, the checker could immediately rule them as mutually incompatible! Secondly, there would be no need for separate syntactic treatments of template-based and class-based polymorphism, since both would be handled using the same underlying F-bounded parametric mechanism. However, the type instantiation process might happen at run-time as well as at compile-time (this unifies the notions of dynamic binding and template instantiation). Thirdly, we would have to consider more carefully the scope of type substitutions made when polymorphic aliasing occurs. We would expect, for example, that a polymorphic method would bind type parameters on entry, but release these bindings on exit, so that the method could be applied to an object of some different type on another occasion. What then is the scope of a polymorphic assignment? We saw above that binding one type rules out subsequent assignments to different types. The scope of an assignment would have to be defined carefully, with rules for "undoing" an assignment and recovering the old polymorphic type of the variable.

The advantages of (genuine) polymorphic typing do not stop there. For example, type propagation may have considerably stronger and pervasive effects on the behaviour of a piece of software. The C++ Standard Template Library makes use of this when it defines template *allocators* for handling the memory management aspects of regular data types. Substituting different actual allocators can alter the efficiency of the whole program. In fact, parametric substitution is related to reflective meta-programming and,

when properly exploited, can produce most of the pervasive benefits claimed by aspect-oriented programming. By understanding the true polymorphic nature of the *class*, we may yet obtain much simpler, yet more powerful programming languages.

REFERENCES

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

[2] A J H Simons, "The theory of classification, part 7: A class is a type family", *Journal of Object Technology,* vol.2 no. 3, May-June 2003, pp 13-22. http://www.jot.fm/issues/issue_2004_05/column2

[3] 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

[4] A J H Simons, "The theory of classification, part 11: Adding class types to object implementations", in *Journal of Object Technology*, vol. 3, no. 3, March-April 2004, pp. 7-19. http://www.jot.fm/issues/issue_2004_03/column1

[5] A J H Simons, "The theory of classification, part 16: Rules of extension and the typing of inheritance", in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 13-25. http://www.jot.fm/issues/issue_2005_01/column2

[6] A 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

[7] W Harris, "Contravariance for the rest of us", *Journal of Object-Oriented Programming,* Nov-Dec 1991, 10-18.

[8] A J H Simons, "The theory of classification, part 1: Perspectives on type compatibility", *Journal of Object Technology*, vol. 1 no. 1, May-June 2002, pp 55-61. http://www.jot.fm/issues/issue_2002_05/column5

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

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

[11] 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.

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

[13] W Cook, "A proposal for making Eiffel type safe", Proc. 3rd European Conf. Object-Oriented Prog., 1989, 57-70; reprinted in *Computer Journal 32(4),* 1989, 305-311

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching Quality 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.