JOURNAL OF OBJECT TECHNOLOGY

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

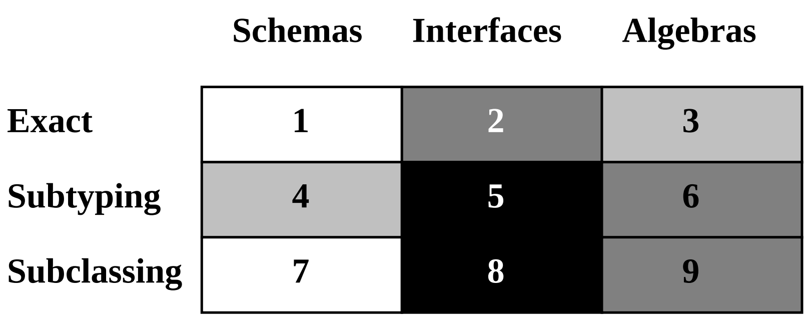
Vol. 2, No. 6, November-December 2003

The Theory of Classification Part 9: Inheritance and Self-Reference

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

1 INTRODUCTION

This is the ninth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The previous article demonstrated 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* [1]. A class can be modelled as an *F-bounded polymorphic type*, representing a family of similar types which share a minimum common structure and behaviour [2, 3]. In this second-order model, which provides an alternative to first-order subtyping [2, 4], even recursively-defined classes can be shown to nest properly inside each other [1], a more sophisticated kind of type compatibility that we call *subclassing* in figure 1 (box 8). Type rules were defined for class membership, sub-classification and class extension by inheritance.



	Schemas	Interfaces	Algebras
Exact	1	2	3
Subtyping	4	5	6
Subclassing	7	8	9

Figure 1: Dimensions of Type Checking

The previous article concentrated only on the *typeful* aspects of classes. In this article, we now turn to the *concrete* aspect of classes and the detailed modelling of method implementations. We want to be able to explain in the formal model how an object's structure is extended or altered during inheritance. In particular, we want to understand the process of method overriding and the meaning, after inheritance, of the special selfreferential variable *self*, also known as *this*, or *current* in different object-oriented languages.

2 OBJECT IMPLEMENTATIONS

Several articles ago, we considered three different formal encodings of simple objects [5]. We preferred the λ-calculus encoding, which represents recursive objects as functional closures, denoting simple records of methods. One reason for choosing this model is because it fits so well with the F-bounded explanation of polymorphic classes, since both models rely on the use of *generators*, special functions which accept *self* (or, the *self*-type) as their argument. Later, we shall link the object model with the type model, in a combined second-order F-bounded λ–calculus. For the time being, we shall deal with objects in an untyped way.

A simple two-dimensional point object at the co-ordinate (3, 5) may be represented as a record, whose fields store values and functions, representing the attributes and methods of the object. Each field consists of a label which maps to a value (a simple value, or a function):

$$\begin{aligned} \text{aPoint2D} = \mu\ \text{self} \,.\, \{\ & \text{x} \mapsto 3,\ \text{y} \mapsto 5,\ \text{identity} \mapsto \text{self}, \\ & \text{equal} \mapsto \lambda\text{p}.(\text{self.x} = \text{p.x} \wedge \text{self.y} = \text{p.y})\ \} \end{aligned}$$

In the above, *self* is the recursion variable, a placeholder equivalent to the eventual definition of the whole object *aPoint2D*, which contains embedded references to *self* (technically, we say that μ binds *self* to the resulting definition – see [5] for an explanation of this convention). Following the dot is a record of fields, enclosed in braces {…}. The first two fields, labelled *x* and *y*, map to simple values in the model. This is because we want *aPoint2D.x* and *aPoint2D.y* to return the simple values directly, rather like accessor methods. The third field *identity* is a method for returning the object itself; the fourth field *equal* maps to another function λp.(…), representing a method that compares *aPoint2D* with the argument *p*, which is assumed to be another Point2D instance.

This is where it becomes clear why *aPoint2D* is a recursive object: inside the body of the *equal* method, *aPoint2D* must invoke further methods *x* and *y* upon itself, to perform the field-by-field comparison with the argument *p*. To enable this, the body of the *equal* method needs a handle on the top-level object *aPoint2D*, granted through the recursion variable *self*. Any object that needs to invoke nested methods on itself must be recursive, so this is quite a common occurrence in practice. The *identity* method is also recursive, returning the object itself.

This theoretical use of *self* corresponds exactly to the usual meaning of *self* (in Smalltalk), *this* (in Java and C++) and *current* (in Eiffel). In the model, you always invoke nested methods explicitly through the recursion variable *self*. In some of these programming languages, you can omit the receiver of a nested message to the same object, which is implicitly understood to be *self* (*this*, or *current*). This is just a syntactic sugaring in the programming language.

3 OBJECT GENERATORS

Readers who have been following this series will know by now that a recursive definition is created from first principles using a generator, a function which abstracts over the point of recursion. Previously, we used *type generators* to build recursive types [1, 2]. Here, we introduce *object generators* to build recursive objects:

$$\begin{aligned}\text{genAPoint2D} = \lambda\ \text{self} .\ \{\ & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y)\ \}\end{aligned}$$

In this function, *self* is not yet bound to any value – it is the argument of the function. The generator can be used to build the recursive object by infinite self-application [5]:

$$\text{aPoint2D} = \text{genAPoint2D}(\text{genAPoint2D}(\text{genAPoint2D}(\ldots)))$$

and for convenience's sake, we may use the fixpoint finder **Y** to build this infinite sequence:

$$\mathbf{Y}\ (\text{genAPoint2D}) = \text{genAPoint2D}(\text{genAPoint2D}(\text{genAPoint2D}(\ldots))) = \text{aPoint2D}$$

Later, we will see more expressions of this kind to describe the final fixing of the recursive structure of an object. Roughly speaking, an object generator *genAPoint2D* is a template for the structure of points like *aPoint2D*, in which *self* does not yet refer to anything. It turns out that this is a useful construction, since it will allow us to explain how *self* can refer to different objects.

4 EXTENDED OBJECT IMPLEMENTATIONS

Our goal is to model the inheritance of implementation. This can be thought of as a kind of extension or adaptation of an object, to produce an extended object with more fields, some of which may be modified, in the sense that the labels map to different values than in the original object. The challenge we set ourselves is to derive a three-dimensional point *aPoint3D* at the co-ordinate (3, 5, 2) by extending the simple two-dimensional *aPoint2D* in some fashion.

To begin with, we jump ahead to describe what we want *aPoint3D* to look like, at the end of the derivation process. Ultimately, we want it to have an extra *z* dimension and a modified version of the *equal* method, as if it had been defined from scratch, as a whole, in the following way:

$$\begin{aligned}\text{aPoint3D} = \mu\ \text{self} .\ \{\ & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self},\\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\end{aligned}$$

In the above, *self* is the placeholder variable, equivalent to the eventual definition of the object *aPoint3D*, which also contains embedded references to *self* in its *identity* and *equal* methods. Note the difference in the meaning of *self*: whereas before μ bound *self* $\leftarrow$ *aPoint2D*, here μ binds *self* $\leftarrow$ *aPoint3D*. The object denoted by *self* changes, depending on the binding context. From the practical point of view, this is also desirable, since the body of the modified *equal* method is only valid if *self* stands for *aPoint3D* (*viz*: you could not access the *z*-method of *aPoint2D*).

In the previous article [1], we constructed a simple model for inheritance, in which we treated a record as a set of fields and used set union $\cup$ to build a larger derived record by taking the union of the fields of the base record and a record of extra fields:

derived = base $\cup$ extra

This only works for very simple records, which have unique fields and no recursion:

aCoord2D = { x $\mapsto$ 3, y $\mapsto$ 5 }	- base record
zField = { z $\mapsto$ 2 }	- extra record
aCoord3D = aCoord2D $\cup$ zField = { x $\mapsto$ 3, y $\mapsto$ 5, z $\mapsto$ 2 }	

We cannot construct *aPoint3D* in this way, because the simple union of fields would create a result in which there were two different versions of *equal*, since the original and redefined versions of this field are not actually identical, therefore the union would preserve two copies.

5 UNION WITH OVERRIDE

Instead, a different operator must used, called *union with override*: $\oplus$. This is a standard mathematical operator, defined for maps (rather than sets), that combines two maps in a certain way, which we define below. A *map* is a collection of pairs of values, called *maplets*, with an arrow from the left- to the right-hand value in each pair. A map is written like:

{a $\mapsto$ x, b $\mapsto$ y, c $\mapsto$ z, ….}   “a maps to x, b maps to y, c maps to z, ...”

and can be viewed variously as a lookup table, in which each maplet relates a key (on the left) to a corresponding value (on the right), or alternatively as a function[1] from the domain {a, b, c, … } to the range {x, y, z, … }. Where a map models a function, all the domain values are unique (but the range could contain duplicates).

An advantage in modelling objects as records is that they can also be thought of as maps: each field is a maplet consisting of a label (the domain) that maps to a value (the range). In object maps, the labels always have the same type (viz: Label), but the values

[1] The operator $\oplus$ is sometimes called *function override* in formal methods like Z, precisely because functions in Z are modelled as maps.

could be of many different types. For this reason, we give the union with override operator the following definition:

$$
\begin{aligned}
&\forall\alpha, \beta, \gamma \,.\, \oplus : (\alpha\to\beta) \times (\alpha\to\gamma) \to (\alpha\to\beta\cup\gamma)\\
&\oplus = \lambda(f{:}\alpha\to\beta).\lambda(g{:}\alpha\to\gamma).\\
&\qquad\quad \{\, k \mapsto v \mid (k \in \mathrm{dom}(f) \cup \mathrm{dom}(g)) \wedge\\
&\qquad\qquad\quad (k \in \mathrm{dom}(g) \Rightarrow v = g(k)) \wedge\\
&\qquad\qquad\quad (k \notin \mathrm{dom}(g) \Rightarrow v = f(k)) \,\}
\end{aligned}
$$

The top line is a polymorphic type signature [2], saying that $\oplus$ takes two maps with the individual type signatures $(\alpha\to\beta)$ and $(\alpha\to\gamma)$, and returns a map with the signature $(\alpha\to\beta \cup\gamma)$. Notice how the type of the domain α is the same in each case (for labels), but the types of the ranges β, γ are possibly different. The range in the result is a union type $\beta\cup\gamma$, formed by merging the types of the ranges of each argument. This is consistent with the type unions used in the previous article [1].

The full definition follows. This says that $\oplus$ takes two argument maps, *f* and *g* (with the given types) and produces a result map (the whole expression in braces). This result is the set of all those maplets $k \mapsto v$ that satisfy the following conditions (after the vertical bar |). The domain values *k* are obtained by taking the union of the domains of each argument map. This ensures that the domain of the result contains all the unique domain values from both maps. The range values *v* are obtained according to the asymmetric rule: if *k* is in the domain of the right-hand map *g*, use the corresponding range value *g(k)* from the right-hand map; otherwise use the corresponding range value *f(k)* from the left-hand map. This works, because every *k* must either be in the domain of the left-, or right-hand maps, or both. Note that *f(k), g(k)* denote range values by appealing to the functional interpretation of maps: *f(k)* "applies" the map *f* to the domain value *k*, yielding the corresponding range value.

When used with records, the union with override operator has the effect of merging two records, but preferring the fields from the right-hand side in the case of conflicting labels. Where there are no label-conflicts, it takes the union of the fields. If there are label-conflicts, it discards fields on the left-hand side and chooses fields from the right-hand side. This is exactly the behaviour we need to model record extension with overriding.

Another use for this operator is to model field updates in an object. We shall look at this first, as a simple way of illustrating the behaviour of $\oplus$. Consider updating the x position of the simple co-ordinate from above:

$$
\begin{aligned}
&\mathrm{aCoord2D} = \{\, x \mapsto 3, y \mapsto 5 \,\} && \text{- starting state values}\\
&\mathrm{newCoord2D} = \mathrm{aCoord2D} \oplus \{x \mapsto 7\} && \text{- override the x value}\\
&\qquad\quad = \{\, x \mapsto 7, y \mapsto 5 \,\} && \text{- modified state values}
\end{aligned}
$$

This can be used to model a primitive notion of field reassignment, although in the calculus we always create and return new objects (the λ-calculus is purely functional, after all).

6 SCHIZOPHRENIC SELF-REFERENCE

Following this example, it would seem natural to define the extended object *aPoint3D* by combining the record for *aPoint2D* with a record of the extra methods (*z* and the modified *equal*) that we want it to have. The overriding behaviour of ⊕ will ensure that the result gains just one copy of the *equal* method, from the right-hand record of extra methods, which has the following structure:

$$\{ z \mapsto 2, \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}$$

Notice how this record also implies a recursion somewhere, because the *equal* method contains free references to *self*. To which object should this *self* refer? Looking at the body of the *equal* method, it is clear that we intend it to refer to *aPoint3D*, because we want to invoke the nested method *self.z*, which is not valid for *aPoint2D*. The only way to make *self* refer to the resulting object is to bind it outside the record combination (see bold highlight):

$$\begin{aligned} \text{aPoint3D} \;=\; & \boldsymbol{\mu}\,\textbf{self . (}\ \text{aPoint2D} \oplus \{ z \mapsto 2, \\ & \quad \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}\ \textbf{)} \end{aligned}$$

This says that *aPoint3D* is a recursive object, formed by taking the union-with-override of *aPoint2D* and a record of extra methods, in which *self* refers to *aPoint3D*. Unrolling the definition of *aPoint2D* (see bold highlight) this gives:

$$\begin{aligned} \text{aPoint3D} \;=\; & \mu\,\text{self . (}\ \boldsymbol{\{ x \mapsto 3, y \mapsto 5,}\ \textbf{identity} \mapsto \textbf{aPoint2D,} \\ & \qquad \textbf{equal} \mapsto \boldsymbol{\lambda}\textbf{p.(aPoint2D.x = p.x} \wedge \textbf{aPoint2D.y = p.y) \}} \\ & \quad \oplus \{ z \mapsto 2, \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \}\ ) \end{aligned}$$

We can now judge how ⊕ will combine the fields of these two records. The result should contain fields having all the labels *{x, y, z, identity, equal}* and the right-hand version of *equal* should be preferred, overriding the left-hand version. After record combination, this simplifies to:

$$\begin{aligned} \text{aPoint3D} \;=\; & \mu\,\text{self . } \{ x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \text{aPoint2D}, \\ & \qquad \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z) \} \end{aligned}$$

This is the correct result, but on closer inspection, it may not be quite what was expected. Unrolling the definition of *aPoint3D* reveals what has become of all the references to *self*:

$$\begin{aligned} \{ & x \mapsto 3, y \mapsto 5, z \mapsto 2, \text{identity} \mapsto \textbf{aPoint2D}, \\ & \quad \text{equal} \mapsto \lambda p.(\textbf{aPoint3D}.x = p.x \wedge \textbf{aPoint3D}.y = p.y \wedge \textbf{aPoint3D}.z = p.z) \} \end{aligned}$$

From this it is apparent that *aPoint3D* is a schizophrenic object, in two minds about itself! The inherited method *identity* thinks that *self* ← *aPoint2D*, while the locally added method *equal* thinks that $self \leftarrow aPoint3D$. How did this confusion arise? Recall that the recursive object *aPoint2D* was defined in isolation, such that *self* was bound over a different record:

$$\begin{aligned} \text{aPoint2D} = \mu\, \text{self} \,.\, \{\, & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self}, \\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y) \,\} \end{aligned}$$

All *self*-reference in this object inevitably refers to *aPoint2D*. So, when we inherit any methods from *aPoint2D* that contain *self*, this always refers to *aPoint2D*. In an earlier article [2], we described the problem of type-loss in languages such as C++ and Java, when methods referring to the self-type are inherited. The current example explains in more detail why this type-loss occurs: it is because inherited *self* always refers to the old object.

7 REDIRECTING SELF-REFERENCE

It was Cook and Palsberg who first described the more flexible behaviour of *self* in languages like Smalltalk and Eiffel [6]. They drew analogies between object self-reference and recursive function derivations . Figure 2 shows the two contrasting cases.

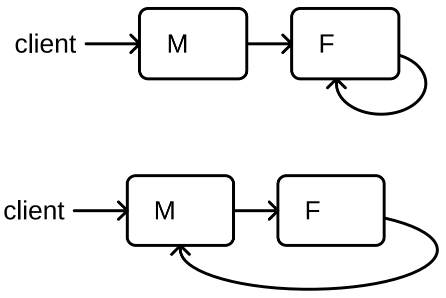


Figure 2: Naïve, and mutually recursive derivations

In the first example, a simple recursive function F calls itself, indicated by the loop. A modification to F is modelled as a derived function M which calls F. Recursive calls in F are unaffected, so the encapsulation of F is preserved. This is like the treatment of *self* in languages such as Java and C++. In these languages, a derived object does not modify the self-reference of the base object.

In the second example, the derived function M is *mutually recursive* with F. Recursive calls in F *are* affected by the modification, in the sense that they refer back to M, instead of to F. This is like the treatment of *self* in languages such as Smalltalk and Eiffel. In these languages, a derived object implicitly modifies the inherited *self*-references of the base object, such that these refer instead to the derived object.

To model inheritance in Smalltalk and Eiffel, we must redirect inherited *self*-references to refer to the derived object. In the formal model, this is accomplished by using generators instead of records. A generator is a function of *self*, in which the argument *self* is unbound (see section 3 above). The generator for 2D point objects is given by:

$$\begin{aligned} \text{genAPoint2D} = \lambda\, \text{self}_{2D}\,.\,\{\, & x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self}_{2D}, \\ & \text{equal} \mapsto \lambda p.(\text{self}_{2D}.x = p.x \wedge \text{self}_{2D}.y = p.y)\ \} \end{aligned}$$

and now we seek to derive a generator for 3D point objects, by inheritance. The key strategy is to make sure that the old *self$_{2D}$* is replaced by the new *self* before any record fields are combined. This is achieved by applying *genAPoint2D* to the new self-argument for 3D points (see bold highlight):

$$\begin{aligned} \text{genAPoint3D} = \lambda\, \text{self}\,.\,(\ & \mathbf{genAPoint2D(self)} \oplus \{\, z \mapsto 2, \\ & \text{equal} \mapsto \lambda p.(\ \text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\ ) \end{aligned}$$

This creates an instance of the generator body in which the substitution {*self/self$_{2D}$* } has taken place. This body has the form of a record (see bold highlight):

$$\begin{aligned} \text{genAPoint3D} = \lambda\, \text{self}\,.\,(\ & \mathbf{\{\ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto self,} \\ & \mathbf{equal \mapsto \lambda p.(\ self.x = p.x \wedge self.y = p.y)\ \}} \\ & \oplus\ \{\, z \mapsto 2,\ \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \}\ ) \end{aligned}$$

Both records on the left- and right-hand sides now refer to exactly the same *self*. We may combine them using the union with override operator, yielding the final simplified expression:

$$\begin{aligned} \text{genAPoint3D} = \lambda\, \mathbf{self}\,.\,\{\, & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \mathbf{self}, \\ & \text{equal} \mapsto \lambda p.(\mathbf{self}.x = p.x \wedge \mathbf{self}.y = p.y \wedge \mathbf{self}.z = p.z)\ \} \end{aligned}$$

This produces the generator that we desired, in which all self-reference is uniform (see bold highlight). To create the derived recursive object, all we need to do is take the fixpoint of the generator, which has the effect of binding *self* to the resulting record:

$$\text{aPoint3D} = \mathbf{Y}\,(\text{genAPoint3D})$$

$$\begin{aligned} = \mu\, \text{self}\,.\,\{\, & x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self}, \\ & \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y \wedge \text{self}.z = p.z)\ \} \end{aligned}$$

We have now succeeded in our challenge, for this object has the desired structure and uniformity that we anticipated in section 4, but it required a more sophisticated model of inheritance. This model of implementation inheritance is somehow more satisfying, in that it allows inherited code to adapt to the derived object. For example, the inherited *identity* method will now return the derived object *aPoint3D*, rather than the old object *aPoint2D*.

8 CONCLUSION

We have constructed two different models of implementation inheritance and found that these correspond broadly to the mechanisms used in the major object-oriented languages. The core idea of extending and modifying object templates can be modelled using records and the union with override operator ⊕, for which we provided a satisfying typed definition. In Cook's earlier work [4, 3, 6], no general typed definition was ever given for ⊕, so this aspect is novel in our Theory of Classification. The operator captures the basic mechanism for record combination with overriding for all languages. After this, object-oriented languages diverge, falling into two groups.

In the first group, which includes Java and C++, self-reference is fixed as soon as the object template is defined. When such an object is extended, although self-reference in the additional methods refers to the *derived object*, self-reference in the inherited methods always refers to the *base object*. In a suitably deep inheritance hierarchy, this means that objects may contain many versions of self, each referring to a different embedded ancestor-object! We described this as a kind of schizophrenia in self-reference. It is also linked to the problem of type-loss when methods passing values of the self-type are inherited [2]. Nonetheless, this model is the one used in all languages based on *types and subtyping*.

In the second group, which includes Smalltalk and Eiffel, self-reference is open to modification, *even after* the object template is defined. When such an object is extended, self-reference in the inherited methods is always redirected to refer to the derived object. All references to self are uniform, which is good for consistency, but they are interpreted locally in the object concerned. This is a novel feature of object-oriented languages, anticipating other kinds of adaptive programming. It is interesting to note how this flexibility came about. In Smalltalk, all methods are dynamically interpreted, so references to *self* are only bound at runtime to mean the current receiver. In Eiffel, the recursion variable *current* was originally conceived as a macro, to be expanded locally to refer to the new class. This had the effect of building implicit type redefinition into the language. The flexible model of inheritance is used in all languages based on *classes and subclassing* [1].

Formally, the difference between the two models of inheritance is very slight: it depends only on *when* fixpoints are taken. In the subtyping model, the recursion in the base object is fixed *before* record combination. In the subclassing model, the recursion is only fixed *after* record combination. For this reason, we can claim that the theory is both elegant and economical.

REFERENCES

[1] 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_04/column4

[2] 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

[3] 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.

[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] 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

[6] W Cook and J Palsberg, "A denotational semantics of inheritance and its correctness", *Proc. 4th ACM Conf. Obj.-Oriented Prog. Sys. Lang. and Appl.*, pub. *Sigplan Notices, 24(10)*, (ACM Sigplan, 1989), pp. 433-443.

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.