
Article:
Simons, A.J.H. (2004) The theory of classification part 12: building the class hierarchy. Journal of Object Technology, 3 (5). 13 - 24. ISSN 1660-1769

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 5, May-June 2004

The Theory of Classification Part 12: Building the Class Hierarchy

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

1 INTRODUCTION

This is the twelfth article in a regular series on object-oriented type theory for non-specialists. Readers following the series will by now have gained some experience in theoretical models and their uses. 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 combined the type and implementation aspects of inheritance [6], introducing the typed notation that will be used in the rest of this series. The theoretical model has already revealed some interesting differences between the notions of *type* and *class*. We have shown that one group of languages, exemplified by C++ and Java, uses *subtyping* [2, 3, 5] as the basis for type compatibility, whereas another group, exemplified by Smalltalk and Eiffel, uses *subclassing* [3, 4, 5], which is a distinct formal relationship in the *Theory of Classification*.

In this article, we review the whole model so far, to demonstrate more of its formal modelling power. The theoretical model is able to represent the whole spread of object-oriented concepts, such as objects, types, classes, abstract classes and interfaces. It can handle the notion of object creation, class extension through inheritance, type compatibility and interface matching. We shall build up a simple model of a class hierarchy, demonstrating all of these concepts. First, we shall briefly review the elements of the model.

2 SIMPLE OBJECTS AND TYPES

Objects are modelled as simple records, whose fields are values, representing attributes, or functions, representing methods. Types are modelled as record types, whose fields are

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification, Part 12: Building the Class Hierarchy", in *Journal of Object Technology*, vol. 3, no. 3, May-June 2004, pp. 13-24.
http://www.jot.fm/issues/issue_2004_05/column2

the corresponding type signatures of the attributes or methods. The following introduces a simple co-ordinate type and an instance of the type:

$$\text{Coord} = \{x : \text{Integer}, y : \text{Integer}\}$$

$$\text{coord} : \text{Coord} = \{x \mapsto 2, y \mapsto 3\}$$

We use capitalisation to indicate type names, and lower case for object names. The above coordinate is a specific instance of Coord at the location (2, 3). To construct such an instance, we can define the constructor-function *makeCoord*:

$$\text{makeCoord} : \text{Integer} \times \text{Integer} \to \text{Coord}$$
$$= \lambda(a, b : \text{Integer} \times \text{Integer}).\{x \mapsto a, y \mapsto b\}$$

This accepts a pair of arguments *a, b*, and returns a record, corresponding to a Coord instance. We can then create *coord* from first principles by the constructor-call:

$$\text{coord} = \text{makeCoord}(2, 3) \Rightarrow \{x \mapsto 2, y \mapsto 3\}$$

We access its fields using the "dot" syntax: $coord.x \Rightarrow 2$, $coord.y \Rightarrow 3$. The fields *x, y* have fixed values once the object is constructed. We can think of these fields either as constant public attributes, or else we can think of them as "unary methods", functions that accept no argument and access an encapsulated value[1]. We typically take the second view, so that all record fields can be thought of as functions.

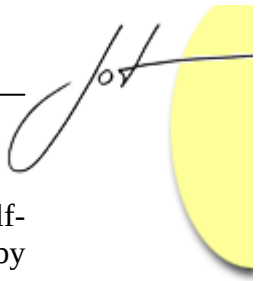
3 RECURSIVE OBJECTS AND TYPES

Objects are frequently recursive, since methods may invoke further methods on *self*, the current object. Types are also recursive, since methods may accept or return arguments of the same type as the *self*-type. These two kinds of recursion are related, but independent. We use a standard technique in the λ-calculus to introduce and bind the recursion, called *fixpoint finding* [1] and separate fixpoints are needed at the object-level and the type-level. The technique involves the use of *generators* (also called *functionals*), which are functions of the self-value (or self-type). The following introduces type- and object-generators for a more sophisticated Cartesian point type, which has a recursive *equal* method:

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

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

[1] It is possible to think of simple fields as functions accepting an empty argument, eg: $x : Empty \to Integer$, and think of all method invocations as supplying the empty value ε automatically, eg: $coord.x(\varepsilon) \Rightarrow 2$.



The type generator *GenPoint* is not yet a record type, but rather a function of σ (the self-type parameter) that returns a record type. We construct the recursive *Point* type by binding σ recursively to the function body, using the fixpoint finder **Y** (see [1] for full details):

$$\text{Point} = \mathbf{Y}[\text{GenPoint}]$$
$$\Rightarrow \{x : \text{Integer}, y : \text{Integer}, \text{equal} : \text{Point} \rightarrow \text{Boolean}\}$$

In this way, we build the recursive *Point* type from first principles. Note how, once the definition is complete, we may *unroll Point* (denoted by the evaluation step ⇒) to a record type containing recursive reference to the *Point* type.

Likewise, the object generator *genPoint* is not yet an object instance, but rather a function that returns this instance. It accepts two arguments: τ stands for the self-type and *self* for the self-value. To create a point-instance from this generator, we need to supply a suitable type for τ and then bind *self* recursively over the rest of the function body. In the first step, we choose to supply the type *Point*, and the returned result is a generator, a simple function of *self*:

$$\text{genPoint}[\text{Point}] = \lambda(\text{self} : \text{Point}).\{x \mapsto 2, y \mapsto 3,$$
$$\text{equal} \mapsto \lambda(p : \text{Point}).(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}$$

In the second step, this simple generator may be recursively fixed using **Y**:

$$\text{point} = \mathbf{Y}(\text{genPoint}[\text{Point}])$$
$$\Rightarrow \{x \mapsto 2, y \mapsto 3, \text{equal} \mapsto \lambda(p : \text{Point}).(\text{point}.x = p.x \wedge \text{point}.y = p.y)\}$$

to bind *self* over the rest of the body. Note again how, after unrolling, the defined *point* instance contains recursive references to *point*. If we consider that the type *Point* was itself the result of a fixpoint operation, we could equally define *point* in the style:

$$\text{point} = \mathbf{Y}(\text{genPoint}[\mathbf{Y}\ \text{GenPoint}])$$

illustrating how it takes two fixpoints to bind the recursions at object- and type-level in each instance. From a theoretician's point of view, this presents some challenges. Bruce and Mitchell [7] were the first to show that **Y** existed both at both the type- and object-levels, and prove that fixpoint operations converged at infinity, provided that objects were records of functions. (Convergence fails if an object has a field which directly returns *self*; however, we can always convert the *identity* method into a function accepting an empty argument – see the earlier footnote[1]).

4 CONSTRUCTING OBJECT INSTANCES

In the previous article [6], we settled on a style for defining all classes using type- and object-generators, as illustrated above. It turns out that this is most useful, because we can create subclasses by adapting generators [4, 5], in a style that mimics inheritance.

However, the exact process of constructing *distinct* object instances was not fully described. In fact, *genPoint* is a generator for a *specific* instance, with *point.x* $\Rightarrow$ 2 and *point.y* $\Rightarrow$ 3. To make this function more general-purpose, we should force it to accept extra initialisation arguments:

$$\begin{aligned}&\text{initPoint} : \forall(\tau <: \text{GenPoint}[\tau]).\ (\text{Integer} \times \text{Integer}) \rightarrow \tau \rightarrow \text{GenPoint}[\tau]\\ &= \lambda(\tau <: \text{GenPoint}[\tau]).\lambda(a, b : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \tau).\\ &\qquad \{x \mapsto a,\ y \mapsto b,\ \text{equal} \mapsto \lambda(p : \tau).(\text{self}.x = p.x \wedge \text{self}.y = p.y)\}\end{aligned}$$

Here, *initPoint* is an extended version of *genPoint*, which accepts a type τ, then a pair of Integer arguments *a, b*, then the usual value for *self*. This function can be used to construct objects having distinct *x, y* field values in the following style:

$$\begin{aligned}&p1 = \mathbf{Y}(\text{initPoint}[\text{Point}](4, 5))\\ &\Rightarrow \{x \mapsto 4,\ y \mapsto 5,\ \text{equal} \mapsto \lambda(p : \text{Point}).(p1.x = p.x \wedge p1.y = p.y)\}\end{aligned}$$

$$\begin{aligned}&p2 = \mathbf{Y}(\text{initPoint}[\text{Point}](6, 7))\\ &\Rightarrow \{x \mapsto 6,\ y \mapsto 7,\ \text{equal} \mapsto \lambda(p : \text{Point}).(p2.x = p.x \wedge p2.y = p.y)\}\end{aligned}$$

Note the order of application in the creation of *p1*: first we supply the precise type, *Point*; then we supply the initialisation values (4, 5). This returns a simple object generator, having the form: λ(*self* : Point).{ … }, which we can fix using **Y** to bind *self* recursively over the rest of the generator body.

If we wanted, we could define a simple object constructor, *makePoint*, in the same style as *makeCoord* (see section 2), which uses the extended function *initPoint* internally:

$$\begin{aligned}&\text{makePoint} : \text{Integer} \times \text{Integer} \rightarrow \text{Point}\\ &= \lambda(a, b : \text{Integer} \times \text{Integer}).\ \mathbf{Y}\ (\text{initPoint}\ [\mathbf{Y}\ \text{GenPoint}]\ (a, b))\end{aligned}$$

$$\begin{aligned}&p1 = \text{makePoint}(4, 5)\\ &p2 = \text{makePoint}(6, 7)\end{aligned}$$

In this, you can think of **Y** as taking a fixed snap-shot of the flexible type-structure and object-structure represented by the two generators. Constructors in practical object-oriented languages always create a specific instance of a specific type.

5 INVOKING OBJECT METHODS

Having created two distinct instances of *Point*, we can simulate their behaviour in the theory, by evaluating expressions representing method invocations, such as:

$$\begin{aligned}&p1.\text{equal}(p2)\\ &\Rightarrow \lambda(p : \text{Point}).(p1.x = p.x \wedge p1.y = p.y)\ (p2) \quad \text{- by selecting } \textit{equal} \text{ from } \textit{p1}\\ &\Rightarrow p1.x = p2.x \wedge p1.y = p2.y \quad \text{- by substituting } \{\textit{p2/p:Point}\}\end{aligned}$$

$\Rightarrow$ 4 = p2.x $\wedge$ p1.y = p2.y - by selecting *x* from *p1*
$\Rightarrow$ 4 = 6 $\wedge$ p1.y = p2.y - by selecting *x* from *p2*
$\Rightarrow$ false $\wedge$ p1.y = p2.y - by *Integer*.=
$\Rightarrow$ false - by *Boolean*.$\wedge$

The steps shown above are mostly single evaluation steps in the calculus, showing on the right which rule applied in each case. In particular, we use the record selection rule to access the methods for *equal* and *x*, and the function application rule when applying the result of *p1.equal* to the argument *p2*. The details of primitive *Integer* and *Boolean* operations are omitted here. We assume that suitable definitions exist for these types.

6 ROOTING A CLASS HIERARCHY

Within the theory, we can model the notion of a class hierarchy, starting with a root class of all objects. We shall later derive other classes by extending the root class. In this example, we shall assume that the root class only defines a single method, *equal*, and that the default implementation of this method is identity of reference, represented by ==.

$$\text{GenObject} = \lambda\sigma.\{\text{equal} : \sigma \rightarrow \text{Boolean}\}$$

$$\text{genObject} : \forall(\tau <: \text{GenObject}[\tau]).\ \tau \rightarrow \text{GenObject}[\tau]$$
$$= \lambda(\tau <: \text{GenObject}[\tau]).\ \lambda(\text{self} : \tau).\ \{\ \text{equal} \mapsto \lambda(o : \tau).(\text{self} == o)\}$$

The type generator *GenObject* describes the type-shape of the most general kind of object, saying that it has an *equal* method. The object generator *genObject* provides the default implementation of the *equal* method, and restricts the applicability of this method to arguments of the type $\lambda(o : \tau)$, where τ ranges over the family of types expressed by the constraint: $\forall(\tau <: \textit{GenObject}[\tau])$. In earlier articles [3, 4] we described how this expresses exactly the notion of a class, a family of related types. This constraint [8] accepts all record types that have at least an *equal* method, with a type signature $\sigma \rightarrow$ *Boolean*.

The *Point* from section 3 above appears to match this pattern. It is possible to derive *Point*'s generators from *Object*'s generators, in the style that mimics the operation of inheritance in object-oriented programming. Recall that $\oplus$ is union with override, creating an extended record by combining a base record with a record of extra methods [5]:

$$\text{GenPoint} = \lambda\tau.(\mathbf{GenObject[\tau]} \cup \{\text{x} : \text{Integer}, \text{y} : \text{Integer}\})$$
$$\Rightarrow \lambda\tau.\{\text{equal} : \tau \rightarrow \text{Boolean}, \text{x} : \text{Integer}, \text{y} : \text{Integer}\}$$

$$\text{initPoint} : \forall(\sigma <: \text{GenPoint}[\sigma]).\ (\text{Integer} \times \text{Integer}) \rightarrow \sigma \rightarrow \text{GenPoint}[\sigma]$$
$$= \lambda(\sigma <: \text{GenPoint}[\sigma]).\lambda(\text{a, b} : \text{Integer} \times \text{Integer}).\lambda(\text{self} : \sigma).$$

genObject[σ](self) ⊕
{x ↦ a, y ↦ b, equal ↦ λ(p : σ).(self.x = p.x ∧ self.y = p.y)}

⇒ λ(σ <: GenPoint[σ]).λ(a, b : Integer × Integer).λ(self : σ).
{x ↦ a, y ↦ b, equal ↦ λ(p : σ).(self.x = p.x ∧ self.y = p.y)}

The bold highlights indicate how the old type-generator and object-generator are reused in the definition of the new, extended functions. At the type-level, we added the types of the *x, y* fields. At the object-level, we added implementations for these, and also redefined *equal* to compare the *x, y* field values, rather than test for reference identity. The right-handed preference of ⊕ causes the new version of *equal* to override the default version provided in the root class, when the extra record is combined with the base record [5].

To establish that the *Point*- and *Object*-classes are in a proper hierarchical relationship, we need to show that the *Point*-interface is compatible with the *Object*-interface. This is done formally using the *Classify* rule [4], in which we have to compare the two type-generators for a pointwise subtyping relationship:

∀τ . GenPoint[τ] <: GenObject[τ]

We can show that the relationship holds for a single dummy exemplar type *t*:

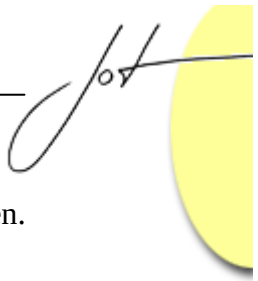
for some t, GenPoint[t] <: GenObject[t]
⇒ {x : Integer, y : Integer, equal : t → Boolean}
<: {equal : t → Boolean}
⇒ true - by record subtyping [2].

and by generalising *t* to all types ∀τ, assert that the relationship holds everywhere. As a result, we maintain that all types belonging to the *Point*-class will also belong to the *Object*-class.

7 INTERMEDIATE ABSTRACT CLASSES

We can also define the notion of an abstract class. This is expressed by a pair of generators in which full type information is given in the type-generator, but some of the implementation information is left undefined in the object-generator. Earlier, we used ⊥ to represent the *undefined* value [1]. In λ-calculus, any expression which simplifies to ⊥ is the formal equivalent of raising an exception in a practical programming language. This is also quite suitable for representing abstract methods, since it is an error to invoke them (instead, you would expect the abstract methods to be overridden in a concrete subclass).

We seek to define a *Shape*-class, the abstract ancestor of geometric shapes, such as *Circle* and *Rectangle*, which provides a concrete *origin* method, indicating its screen



coordinates, but an abstract *area* method, for which no implementation can yet be given. Furthermore, we define this class by extension from the root *Object*-class above.

GenShape = λτ.(GenObject[τ] ∪ { **origin : Point, area : Integer** })
⇒ λτ.{equal : τ → Boolean, **origin : Point, area : Integer** }

initShape : ∀(σ <: GenShape[σ]). Point → σ → GenShape[σ]
= λ(σ <: GenShape[σ]). λ(p : Point). λ(self : σ).
genObject[σ](self) ⊕ { **equal ↦ λ(s : σ).(self.origin.equal(s.origin)),**
origin ↦ p, area ↦ ⊥ }

⇒ λ(σ <: GenShape[σ]). λ(p : Point). λ(self : σ).
{ **equal ↦ λ(s : σ).(self.origin.equal(s.origin)), origin ↦ p, area ↦ ⊥** }

In the type-generator *GenPoint*, the extra type-information is highlighted in bold. The object-function *initShape* is written in the same style as *initPoint*, with extra initialisation arguments, since we want to be able to supply the initial *Point* coordinate *p* at which a *Shape* is to be located. Given this argument *p*, we can define *origin* to return *p*, and can redefine *equal* to compare the *origins* of two *Shapes* – this in turn uses the *equal* method defined earlier for *Points*. Note the use of ⊥ in the body of the *area* method, indicating that this is so far undefined.

To demonstrate what happens in the model when you try to invoke an abstract method, we exceptionally provide a simple constructor for abstract Shapes, so that we can create an instance and invoke the abstract method (normally, no constructor would be provided, since the class is abstract):

makeShape : Point → Shape
= λ(p : Point). **Y** (initShape [**Y** GenShape] (p))

p1 = makePoint(4, 5) ⇒ … - detail omitted for clarity
s1 = makeShape(p1) ⇒
{ equal ↦ λ(s : Shape).(s1.origin.equal(s.origin)), origin ↦ p1, area ↦ ⊥ }
s1.origin ⇒ p1 - concrete, by selecting *origin* in *s1*
s1.area ⇒ ⊥ - abstract, by selecting *area* in *s1*

The result of attempting to select an abstract method is the undefined value ⊥, equivalent to raising an exception.

8 FINAL CONCRETE CLASSES

We now seek to derive a concrete *Rectangle*-class as a subclass of the abstract *Shape*-class, with an implementation of its *area* method. The type-generator declares the extra signatures for the *width* and *height* fields, while the object-function supplies

implementations for these, and also provides a concrete implementation for the *area* method, and re-implements the *equal* method to compare *width* and *height*, in addition to *origin*:

$$\begin{aligned}
&\text{GenRectangle} = \lambda\sigma.(\text{GenShape}[\sigma] \cup \{\ \textbf{width : Integer, height : Integer}\ \})\\
&\Rightarrow \lambda\sigma.\{\text{equal} : \sigma \to \text{Boolean}, \text{origin} : \text{Point}, \text{area} : \text{Integer}, \textbf{width : Integer,}\\
&\qquad\qquad \textbf{height : Integer}\ \}
\end{aligned}$$

$$\begin{aligned}
&\text{initRectangle} : \forall(\tau <: \text{GenRectangle}[\tau]).\\
&\qquad (\text{Point} \times \text{Integer} \times \text{Integer}) \to \tau \to \text{GenInteger}[\tau]\\
&= \lambda(\tau <: \text{GenRectangle}[\tau]).\ \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\ \lambda(\text{self} : \tau).\\
&\qquad \text{initShape}[\tau]\ (\text{p})\ (\text{self})\ \oplus\ \{\ \textbf{area} \mapsto \textbf{(self.width} \times \textbf{self.height),}\\
&\qquad\quad \textbf{equal} \mapsto \textbf{λ(r : τ).(self.origin.equal(r.origin)} \wedge \textbf{self.width = r.width}\\
&\qquad\qquad\qquad\qquad \wedge \textbf{self.height = r.height),}\\
&\qquad\quad \textbf{width} \mapsto \textbf{w, height} \mapsto \textbf{h}\ \}\\
&\Rightarrow \lambda(\tau <: \text{GenRectangle}[\tau]).\ \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\ \lambda(\text{self} : \tau).\\
&\qquad \{\ \text{origin} \mapsto \text{p},\ \textbf{area} \mapsto \textbf{(self.width} \times \textbf{self.height),}\\
&\qquad\quad \textbf{equal} \mapsto \textbf{λ(r : τ).(self.origin.equal(r.origin)} \wedge \textbf{self.width = r.width}\\
&\qquad\qquad\qquad\qquad \wedge \textbf{self.height = r.height),}\\
&\qquad\quad \textbf{width} \mapsto \textbf{w, height} \mapsto \textbf{h}\ \}
\end{aligned}$$

By the usual operation of $\oplus$, the right-hand, concrete version of *area* is preferred, overriding the abstract version inherited from the *Shape*-class; likewise, the new version of *equal* is preferred, replacing the inherited version. The *Rectangle*-class is now fully defined, with suitable implementations for all of its methods.

In the object-function *initRectangle*, the initialisation-value is of a different type than the argument supplied to *initShape*, since we need to initialise a rectangle with (all of) its *origin*, *width* and *height* values. Formally, this initialiser is a single value, a tuple of the product type Point × Integer × Integer, whose projections we access implicitly. Notice how the initialiser passed back to *initShape* in the inheritance-expression (on the left of $\oplus$) is just the *p : Point* value, the first projection from the current initialiser. This models the notion of passing back some construction arguments to a superclass.

We can provide a simple constructor for *Rectangle* objects, in the usual way, which fixes the object- and type-level recursions to construct an instance of this exact type:

$$\begin{aligned}
&\text{makeRectangle} : \text{Point} \times \text{Integer} \times \text{Integer} \to \text{Rectangle}\\
&= \lambda(\text{p, w, h} : \text{Point} \times \text{Integer} \times \text{Integer}).\\
&\qquad \textbf{Y}\ (\text{initRectangle}\ [\textbf{Y}\ \text{GenRectangle}]\ (\text{p, w, h}))
\end{aligned}$$

and with this, we may construct a number of distinct *Rectangle* instances:

p1 = makePoint(4, 5) ⇒ … - detail omitted for clarity
r1 = makeRectangle(p1, 2, 3) ⇒
{ origin ↦ p1, area ↦ (r1.width × r1.height),
equal ↦ λ(r : Rectangle).(r1.origin.equals(r.origin) ∧
r1.width = r.width ∧ r1.height = r.height),
width ↦ 2, height ↦ 3 }
r2 = makeRectangle(p1, 6, 7) ⇒
{ origin ↦ p1, area ↦ (r2.width × r2.height),
equal ↦ λ(r : Rectangle).(r2.origin.equals(r.origin) ∧
r2.width = r.width ∧ r2.height = r.height),
width ↦ 6, height ↦ 7 }

The instances *r1, r2 : Rectangle* have different values for their *width* and *height* fields, although they happen to share their *origin* in this example. Also, their *area* and *equal* methods invoke further methods (recursively) on the same instance, as intended.

9 TYPE COMPATIBILITY AND INTERFACE MATCHING

The *Rectangle* instances should be type-compatible with the *Shape*-class interface, and transitively with the *Object*-class interface. Methods defined for these general classes should be type-correct when applied to instances of the specific *Rectangle* type. One way of checking this is to see whether *Rectangle* is included in the type-family of each class [4]. First, we want to see if *Rectangle* is in the class of *Shapes*:

Rectangle <: GenShape[Rectangle]
⇒ {equal : Rectangle → Boolean, origin : Point, area : Integer,
width : Integer, height : Integer}
<: {equal : Rectangle → Boolean, origin : Point, area : Integer}
⇒ true, - by record subtyping [2];

and secondly, whether *Rectangle* is in the class of *Objects*:

Rectangle <: GenObject[Rectangle]
⇒ {equal : Rectangle → Boolean, origin : Point, area : Integer,
width : Integer, height : Integer}
<: {equal : Rectangle → Boolean}
⇒ true, - by record subtyping [2].

In both cases, the answer is yes, because *Rectangle* extends the interface provided by each superclass. In particular, the *equal* method was originally declared in the *Object*-class, with the type: ∀(τ <: *GenObject*[τ]). τ.*equal* : τ → *Boolean*, so applying this to a *Rectangle* instance simply produces the substitution: {*Rectangle*/τ} and yields the specifically-typed version: *Rectangle.equal : Rectangle* → *Boolean*. Similarly, the *area* method was originally declared in the *Shape*-class with the type: ∀(τ <: *GenShape*[τ]).

$\tau.area$: *Integer*, so applying this to a *Rectangle* instance produces the same substitution: {*Rectangle*/τ} and yields the specifically-typed version: *Rectangle.area : Rectangle* $\rightarrow$ *Integer*.

Looking at the unrolled version of the *Rectangle* type above, we can see that the redefined versions of these methods have exactly the same types. So, *Rectangle* matches all the expected superclass interfaces, as intended. These interfaces were defined as part of the type-information associated with a class. However, some practical programming languages allow the definition of interfaces that are not associated with any class.

In the theory, the concept of an independent *interface* may be represented exactly by a type-generator, without a corresponding object-generator. If we wanted to specify (for example) a *Locatable* interface for all object types providing an *origin* method, we could do so using just the type generator:

$$GenLocatable = \lambda\sigma.\{origin : Point\}$$

and then give *Locatable* variables the polymorphic type: $\forall(\tau <: GenLocatable[\tau])$. It is clear that both the *Shape* and *Rectangle* classes satisfy this interface, by the pointwise subtyping condition expected in the Classify rule [4]:

$$\forall\tau\ .\ GenShape[\tau] <: GenLocatable[\tau]$$
$$\forall\tau\ .\ GenRectangle[\tau] < : GenLocatable[\tau]$$

and the interested reader is encouraged to prove this, using the same kind of strategy as demonstrated in section 6 above.

10 CONCLUSION

This article has focused on two main areas: how to construct specific object instances, and the development of a simple class hierarchy. The overall aim was to demonstrate how the *Theory of Classification* can model a variety of object-oriented concepts, including objects, types, classes, abstract classes and interfaces.

Regarding object construction, we showed how generators can be extended into flexible object-creation functions with initialisation arguments. Furthermore, initialisation values may be passed back to the superclass functions, mimicking the behaviour of real object-oriented languages. From a formal perspective, creating a unique instance of an exact type always involves taking a double fixpoint, one for the type-generator and one for the object-generator.

Regarding class hierarchy development, we gave examples of recognisable classes, with mixtures of default, abstract and concrete methods. Note especially how the type- and implementation-aspects were able to evolve independently, according to need. A class may introduce a new method, may declare an abstract method, or may re-implement an existing method, replacing it with a more appropriate version. All of this is handled

uniformly within the theory. Furthermore, we have shown that classes derived by inheritance (using generators) give rise to object constructors, which create objects of exact types (after fixpoints are taken). These exact types match the expected interfaces of their superclasses, and also of separately-declared interfaces, where appropriate. The matching condition (*Classify* [4]) is exactly the same in both cases, demonstrating the economy of the theory.

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_08/column4

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

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

[4] 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 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

[7] K Bruce and J Mitchell, "PER models of subtyping, recursive types and higher-order polymorphism", *Proc. 19th ACM Symp. Principles of Prog. Langs.*, (1992), 316-327.

[8] 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.