JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 3, No. 1, January-February 2004

The Theory of Classification Part 10: Method Combination and Super-Reference

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

1 INTRODUCTION

This is the tenth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. Recent articles have presented a formal model of classification that differs from the conventional model of types and subtyping [1, 2]. The popular object-oriented languages fall into two groups – those based on simple types and subtyping, such as C++ and Java, and those based on polymorphic classes and subclassing, such as Smalltalk and Eiffel [2, 3]. A programmer's class has a formal interpretation at the typeful level [2] and a concrete interpretation the implementation level [3]. In the theory, we have been careful to distinguish *classification* from *inheritance*.

Classification is the hierarchical relationship between classes, whereby one class is judged to be a subclass of another, according to type rules governing subclassing [2]. Inheritance, on the other hand, is a short-hand mechanism for defining a new class in relation to an existing class[1], specifying what is new or different, and otherwise *inheriting* all existing features [3]. This presents an interesting formal challenge. If inheritance is indeed merely a short-hand, we should be able to prove in our theory that a class constructed by inheritance is equivalent to a class defined as a whole, from first principles [3]. This challenge is made more complicated by the possibility of *method combination*, the merging of local and inherited versions of a method. Furthermore, Smalltalk and Java can invoke inherited versions of methods through a pseudo-variable called *super*. So, our

[1] Popular books on object-oriented programming sometimes confuse these notions, referring to the hierarchical relationship as "inheritance". Strictly speaking, inheritance is just the extension mechanism. However, an inheriting class will typically be a subclass, but only by virtue of obeying the rules about classification [2].

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification, Part 10: Method Combination and Super-Reference", in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 43-53. http://www.jot.fm/issues/issue_2004_01/column4

theory must be able to explain the meaning of *super*, and show how super-method invocations are eventually equivalent to ordinary method invocations.

2 METHOD COMBINATION

So far, our treatment of inheritance and method overriding has assumed that individual methods are always replaced as a whole. For example, in the previous article [3] we replaced an *equal* method of a two-dimensional point:

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

in which *self* refers to a Point2D instance, by an *equal* method of a three-dimensional point:

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

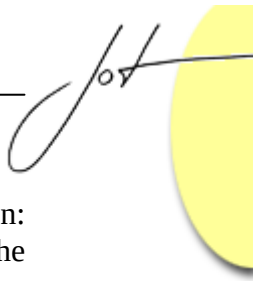
in which *self* refers to a Point3D instance. On the surface, the overriding method appears to be syntactically quite similar to the method that was replaced. In most object-oriented languages, programmers don't have to write out such replacement methods long-hand; instead the languages offer a short-hand mechanism for adapting the inherited version of the method. The idea is that the overriding method may somehow reuse the code body of the inherited method and need only specify what additional computations take place. This is known as *method combination*.

In C++ and Eiffel, method combination is achieved simply by naming the local and inherited versions of the method differently. In C++ it is always possible to qualify method names globally by their owning class, in the style: *ParentClass::method* and *ChildClass::method*. So, the redefined body of the *Child*'s *method* may invoke *ParentClass::method* explicitly, by referring to its global name. In Eiffel, the name of a method may be locally renamed when it is inherited, in the style: ***class*** *Child* ***inherit*** *Parent* ***rename*** *method* ***as*** *old_method* … ***end***. Then, the body of *Child*'s *method* may invoke *old_method*. In both of these cases, invoking an inherited method inside a redefined version is no different from invoking any other differently-named method.

Smalltalk and Java achieve method combination in a much more interesting way. These languages have a pseudo-variable called *super*, which somehow allows a programmer to invoke an inherited version of a method inside the redefined version of the same method, without renaming the method. In the theoretical model, we can express Point3D's *equal* method more simply as:

$$\text{equal} \mapsto \lambda p.(\text{super.equal}(p) \wedge \text{self.z} = p.z)$$

In this, *super* somehow stands for the current object in the context of the parent class. Whereas an invocation *self.equal(…)* would call the local version, *super.equal(…)* is deemed to call the inherited version of the same method. It is quite difficult to understand exactly what object *super* refers to! By the end of this article, we hope to answer this



important question. However, one consideration is that the sub-expression: *super.equal(p)* must eventually be shown to be equivalent to the portion of code in the long-hand version of the method: $self.x = p.x \wedge self.y = p.y$.

3 RENAMING METHODS

To handle method combination in C++ and Eiffel, we must first consider how to rename methods in the theoretical model. In C++, we could either decide that the global names exist permanently as alternative, duplicate names for the same methods, or we could introduce them on demand, when we want to combine methods. In Eiffel, however, we must allow methods to be renamed locally, on demand, during inheritance. Recall that inheritance is modelled as record combination in the theoretical model [3], in which a *base* record is extended by combining it with a record of *extra* methods, to yield a *derived* record:

$$\text{derived} = \text{base} \oplus \text{extra}$$

Intuitively, we now want to rename methods in the base record, then combine this with the extra record. For this, we need a renaming operator ®, and expect to use it in the following way:

$$\text{derived} = (\text{base} \circledR \text{renamings}) \oplus \text{extra}$$

in which *renamings* is a map from original labels to revised labels. In the same style as the union with override operator [3], we may define the renaming operator to accept an object record (a map from labels to methods) and a record of renamings (a map from labels to labels) and return a new object record in which some labels have been replaced:

$$\forall \alpha, \beta \,.\, \circledR : (\alpha \to \beta) \times (\alpha \to \alpha) \to (\alpha \to \beta)$$
$$\circledR = \lambda(f{:}\alpha \to \beta).\lambda(g{:}\alpha \to \alpha).$$
$$\{\, k \mapsto v \mid \forall h \in \mathrm{dom}(f) \,.\, v = f(h) \wedge (h \in \mathrm{dom}(g) \Rightarrow k = g(h)) \wedge (h \notin \mathrm{dom}(g) \Rightarrow k = h) \,\}$$

The top line is a polymorphic type signature [1], saying that ® takes two maps with the individual type signatures $(\alpha \to \beta)$ and $(\alpha \to \alpha)$, and returns a map with the signature $(\alpha \to \beta)$. The type α is the label-type, and β is the method-type in the object record. Note how the object-type of the result is unchanged, reflecting that methods have merely been renamed.

The full definition follows. This says that ® 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 |). For all original labels *h* in the domain of the base object *f*, if *h* is also listed in the domain of the renamings *g*, the resulting label *k* is equal to the translation *g(h)*, otherwise

k is equal to the original label *h*. The corresponding values *v* are always equal to *f(h)*, as in the original object map. Recall that *f(h)*, *g(h)* denote range values by appealing to the functional interpretation of maps: *f(h)* "applies" the map *f* to the domain value *h*, yielding the corresponding range value.

We can use this operator to rename methods of an object. Here is a simple Point2D instance:

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

in which we wish to rename the *equal* method with a longer, qualified name, *old_equal*:

$$\text{aPoint2D}\ \textcircled{R}\ \mathbf{\{equal \mapsto old\_equal\}} = \mu\ \text{self}\ .\ \{\ x \mapsto 3,\ y \mapsto 5,\ \text{identity} \mapsto \text{self},\ \text{old\_equal} \mapsto \lambda p.(\text{self.x} = p.x \wedge \text{self.y} = p.y)\ \}$$

The renaming operator ® takes, as its operands, the object and a map of renamings (see bold highlight), yielding an object in which the *equal* method has been renamed.

4 METHOD COMBINATION WITH RENAMING

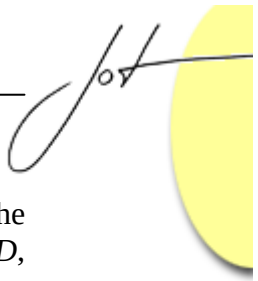
This allows us to construct a model of inheritance with method combination supported through renaming. The following is an expression to derive *aPoint3D* from *aPoint2D* by renaming its *equal* method and then supplying a redefined version, which refers back to the old version:

$$\text{aPoint3D} = \mu\ \text{self}\ .\ (\ \mathbf{(aPoint2D\ \textcircled{R}\ \{equal \mapsto old\_equal\})} \oplus \{\ z \mapsto 2,\ \text{equal} \mapsto \lambda q.(\mathbf{self.old\_equal(q)} \wedge \text{self.z} = q.z)\ \}\ )$$

The left-hand operand of the record combination operator ⊕ is a renaming expression (see first bold highlight), in which the *equal* method is renamed before combination. The right-hand operand of ⊕ is a record of extra methods, in which the redefined *equal* method invokes the renamed method *old_equal* in its body (see second bold highlight), thereby benefiting from the more succinct syntax offered by method combination.

We want to show that this method combination syntax is equivalent to a regular, wholesale method replacement. Simplifying the renaming expression yields a base record (see bold highlight):

$$\text{aPoint3D} = \mu\ \text{self}\ .\ (\ \mathbf{\{\ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto aPoint2D,\ old\_equal \mapsto \lambda p.(\ aPoint2D.x = p.x \wedge aPoint2D.y = p.y)\ \}} \oplus \{\ z \mapsto 2,\ \text{equal} \mapsto \lambda q.(\text{self.old\_equal(q)} \wedge \text{self.z} = q.z)\ \}\ )$$



which in turn may be combined with the record of extra methods (see [3]) to yield the following unusual record, in which inherited self-references are resolved to *aPoint2D*, while the local self is bound over *aPoint3D*:

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

The resulting object has both the renamed method *old_equal* and the redefined version *equal*. This is quite normal in languages with renaming. Note that we have deliberately given these two functions different formal argument names, *p* and *q*, in order to observe what happens to object references in the next stage. We now want to understand the precise meaning of the redefined *equal* in detail. To do this, we may simplify the embedded call to the old version of the method (see bold highlight above). This simplification is equivalent to replacing the call by the inlined body of the *old_equal* method, after substituting the argument {*q/p*}, *viz* the evaluation:

$$\text{self.old\_equal}(q) \Rightarrow (\text{aPoint2D.x} = q.x \wedge \text{aPoint2D.y} = q.y)$$

This internal simplification is performed exactly like any other function simplification; and may be thought of as an evaluation in which the call is replaced by the body after arguments have been substituted. In this case, the actual argument *q* (a Point3D instance) is substituted in place of the formal argument *p* (a Point2D variable) yielding the simplified form:

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

This demonstrates formally how the derived *equal* method does indeed compare all of the x, y and z dimensions of the Point3D instance *q*. However, note how the body of this method suffers from the same kind of schizophrenia that we have noted before [2, 3] when dealing with simple object types. The local binding is *self* ← *aPoint3D*, whereas the inherited *self* ← *aPoint2D*. After the internal simplification, the combined method is comparing mixtures of 2D and 3D points for the equality of their x and y fields! For this reason, we cannot yet regard this kind of method combination as wholly equivalent to defining a replacement method wholesale.

5 FLEXIBLE METHOD COMBINATION WITH RENAMING

The problem is one of ensuring uniform self-reference in both local and inherited methods. Recall that in Eiffel, self-reference is flexible, such that inherited occurrences of self (known as *current* in Eiffel) are redirected to refer to the derived object. In the formal model, this requires the use of an object generator to express the object definition [3]:

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

This is different from a plain object in that *self* is a formal argument of the generator. As such, we can bind it to different values, representing the different objects that *self* may range over. Inheritance is modelled as an adaptation on generators [3]. We now add the renaming scheme to this model:

$$\text{genAPoint3D} = \lambda\ \text{self}\ .\ (\ \mathbf{(genAPoint2D(self)\ ®\ \{equal \mapsto old\_equal\})} \oplus \{\ z \mapsto 2,\ \text{equal} \mapsto \lambda q.(\text{self.old\_equal}(q) \wedge \text{self.z} = q.z)\ \}\ )$$

The critical difference is in the way self-reference is modified, through the generator application: *genAPoint2D(self)*, yielding an adapted record in which *self* refers to the new object. After simplifying the renaming-expression (see bold highlights above and below), this yields:

$$\text{genAPoint3D} = \lambda\ \text{self}\ .\ (\ \mathbf{\{\ x \mapsto 3,\ y \mapsto 5,\ identity \mapsto self,\ old\_equal \mapsto \lambda p.(self.x = p.x \wedge self.y = p.y)\ \}} \oplus \{\ z \mapsto 2,\ \text{equal} \mapsto \lambda q.(\text{self.old\_equal}(q) \wedge \text{self.z} = q.z)\ \}\ )$$

and after record combination, this yields a result in which both *equal* and *old_equal* methods co-exist, but all self-reference is now uniform:

$$\text{genAPoint3D} = \lambda\ \text{self}\ .\ \{\ x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self},\ \text{old\_equal} \mapsto \lambda p.(\text{self.x} = p.x \wedge \text{self.y} = p.y),\ \text{equal} \mapsto \lambda q.(\mathbf{self.old\_equal(q)} \wedge \text{self.z} = q.z)\ \}$$

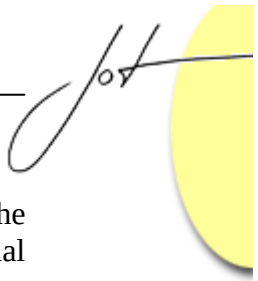
We now simplify the body of the *equal* method, by expanding inline the call to *old_equal*:

$$\text{genAPoint3D} = \lambda\ \text{self}\ .\ \{\ x \mapsto 3,\ y \mapsto 5,\ z \mapsto 2,\ \text{identity} \mapsto \text{self},\ \text{old\_equal} \mapsto \lambda p.(\text{self.x} = p.x \wedge \text{self.y} = p.y),\ \text{equal} \mapsto \lambda q.(\mathbf{self.x = q.x \wedge self.y = q.y} \wedge \text{self.z} = q.z)\ \}$$

This yields the final result (see highlights) in which *self* refers uniformly to the current object. This is a satisfactory outcome, in that it meets our requirement that method combination should be provably equivalent to redefining the method wholesale. Because of its special treatment of *current*, Eiffel's method combination with renaming follows this semantics.

6 THE MEANING OF SUPER

However, it may be considered inelegant for objects to keep both the old and new versions of a method. Having to rename methods is irksome and keeping both versions is redundant, especially if you only want the old version once, during method combination.



For this reason, languages like Smalltalk and Java provide "one time access" to the inherited versions of methods, when redefining the same methods, through a special pseudo-variable called *super*:

$$\text{equal} \mapsto \lambda p.(\textbf{super.equal(p)} \wedge \text{self.z} = \text{p.z})$$

It is clear that *super* must refer in some sense to the current object, somewhat like *self*, yet different from the point of view of method lookup. The operational explanation of *super*-method invocation is typically that "the search for a method starts in the immediate superclass of the class of *self*" [4]. Other attempts at describing *super* sometimes say that it is "the inherited object" or "the embedded parent-part of the current object". In fact, the meaning of *super* is subtle and varies from language to language, as we shall show below.

We may construct a model of *super* from the operational description of method lookup. The most obvious way of invoking the "next most general" version of a method in our theory is to ensure that we select it from the "next most general" version of the object-instance with which we are dealing. In other words, if our most recently derived object is expressed as:

$$\text{derived} = \text{base} \oplus \text{extra}$$

such that we would expect *derived.method* to invoke the latest version of some *method*, then *base.method* should be the expression that invokes the next most general version of the same *method*, skipping over any redefinition supplied in the *extra* record. From this analysis, it seems clear that *super* is equivalent to the *base* object record, in a record combination expression.

In the theory, this object is always supplied as the left-hand operand to the record combination operator $\oplus$ (see bold highlights below). Recall that in record combination with simple object records (see section 6 in [3]) we have:

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

which indicates that this operand is in fact equivalent to an instance of the base type. This gives the meaning of *super* in Java, a language based on simple types and subtyping in our theory. The variable *super* corresponds to the embedded parent-part of the current object, *self*, or more exactly to the inherited parent-part before any fields are replaced during record combination. It has the type *super : Point2D* and behaves exactly like an instance of the base type, in that self-references in super-methods refer back to *super*.

On the other hand, in flexible record combination with object generators (see section 7 in [3]) we have the completely different left-hand operand:

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

which is an adapted record, obtained by applying the base generator to the new self reference (of the derived generator). This gives the exact meaning of *super* in Smalltalk, a

language based on classes and subclassing in our theory. The variable *super* is like an adapted instance of the parent type, in which self-reference is redirected to refer to the current, derived object. It has an adapted type given by the application of a type generator *super : GenPoint3D[Point2D]* and behaves differently from an instance of the base type, in that self-references in super-methods refer to the whole of the current, derived object, rather than to an embedded parent instance.

7 METHOD COMBINATION WITH SUPER-REFERENCE

In order to model method combination with super-reference in our theory, we need to introduce the *super* variable, and bind it to a value standing for the (possibly adapted) parent instance, such that we may refer to *super* throughout the record combination expression, in the style:

$$\textbf{super} \oplus \{ z \mapsto 2, \text{equal} \mapsto \lambda q.(\textbf{super}.\text{equal}(q) \wedge \text{self}.z = q.z) \}$$

The variable may be introduced as the formal argument of a function: λ*super.(…)* which is later bound to a suitable value. The order of variable introduction is dictated by the need for *super* to be bound outside the scope of record combination, but inside the scope of *self*:

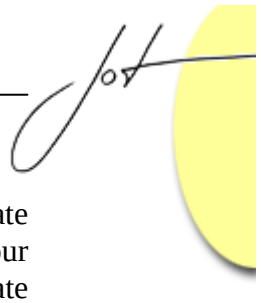
$$\begin{aligned}\text{aPoint3D} = \mu\,\text{self} .\ &\textbf{(}\lambda\textbf{super} .\ \textbf{(}\text{super} \oplus \{ z \mapsto 2,\\ &\qquad \text{equal} \mapsto \lambda q.(\text{super}.\text{equal}(q) \wedge \text{self}.z = q.z) \}\ \textbf{)}\\ &\textbf{aPoint2D)}\end{aligned}$$

Here, the expression λ*super.(…)* is a function, binding *super*, whose body is a record combination expression that contains free references to *super* as intended. This super-function is then applied to the value: *aPoint2D*. To verify that this is equivalent to regular record combination, we may simplify the super-function application internally, with the binding *super* $\leftarrow$ *aPoint2D*, to obtain:

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

Firstly, we see that *super* is replaced by the desired parent object on the left-hand side of the combination operator. Secondly, we find that *super.equal(…)* translates exactly into an expression invoking the *equal* method of a parent instance, which is promising, since it clearly skips the current version. To simplify this internally, we replace the call by the inlined body of the parent's method, after substituting the argument {*q/p*} as before. After record combination:

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



this yields a non-uniform solution similar to that in section 4 above, but without duplicate versions of the *equal* method. This model of method combination explains the behaviour of languages like Java, in which *super* always resolves to an instance of the immediate parent type.

The more flexible kind of method combination with object generators may be modelled as:

$$\begin{aligned}\text{genAPoint3D} \;=\; \lambda\,\text{self}\,.\; &\mathbf{(\lambda super\,.\,(}\text{super} \oplus \{\; z \mapsto 2,\\ &\qquad \text{equal} \mapsto \lambda q.(\text{super.equal}(q) \wedge \text{self.z} = \text{q.z})\;\}\;\mathbf{)}\\ &\mathbf{genAPoint2D(self))}\end{aligned}$$

in which the super-function is bound differently with *super* ← *genAPoint2D(self)*. Note in passing how *self* must be bound before we can bind *super*. This time, *super* does not denote a parent instance, but rather an adapted object, the result of applying the parent generator to *self*. To explore further what this means, we may simplify the super-function application internally, yielding:

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

From this, it appears that the *super.equal(…)* invocation is equivalent to invoking *equal* on an adapted parent object. This is quite subtle, because self-reference is redirected in this object onto the new *self* of 3D points. We can illustrate this more graphically by expanding *super*:

$$\begin{aligned}\text{super} &= \text{genAPoint2D}(\mathbf{self_{3D}})\\ &= \{\; x \mapsto 3,\; y \mapsto 5,\; \text{identity} \mapsto \mathbf{self_{3D}},\\ &\qquad \text{equal} \mapsto \lambda p.(\;\mathbf{self_{3D}}.x = p.x \wedge \mathbf{self_{3D}}.y = p.y)\;\}\end{aligned}$$

From this, it is clear that *super.equal* will select the inherited *equal* method body, in which *self* refers back to the local Point3D instance. Furthermore, *super.equal(q)* will produce the argument substitution {*q/p*}, where *q* is implicitly a variable of the type Point3D:

$$\text{super.equal}(\mathbf{q}) \;\Rightarrow\; (\text{self}_{3D}.x = \mathbf{q}.x \wedge \text{self}_{3D}.y = \mathbf{q}.y)$$

When this subexpression is replaced in the body of the local *equal* method, we obtain a combined *equal* method, which has consistent self-reference and an argument in the same type:

$$\text{equal} \mapsto \lambda q.(\;\text{self}_{3D}.x = q.x \wedge \text{self}_{3D}.y = q.y \wedge \text{self}_{3D}.z = q.z)$$

Method combination using super-reference is thereby proved to be equivalent to redefining the method wholesale. Note how the more subtle semantics of *super* is needed for this to work out fully. Cook et al. were the first to identify this formal interpretation of *super* in Smalltalk [5, 6], from the operational description of inheritance in that language.

8 CONCLUSION

We have constructed four different models for inheritance with method combination. Two involve the renaming of methods, in the style of C++ and Eiffel. Two involve the use of the pseudo-variable *super*, in the style of Java and Smalltalk. While C++ and Java have a simple model of inheritance, based on the extension of object records, Smalltalk and Eiffel have a more subtle model of inheritance, based on the extension of object generators. Hereafter in our theory, we shall refer to the simple extension model as *derivation*, to distinguish it from "genuine" *inheritance*, in which self-references are redirected to refer to more specific objects [3, 5].

Considering each approach to method combination individually, the first uses derivation and renaming. The second uses inheritance and renaming, of which Eiffel is the exemplar. The third uses derivation and super-reference, of which Java is the exemplar. The fourth uses inheritance and super-reference, of which Smalltalk is the exemplar. An even simpler model exists for C++, if we allow objects to have two names for each method, one local and one global:

$$\text{equal} \mapsto \lambda p.(\text{self.x} = p.x \wedge \text{self.y} = p.y)$$
$$\text{Point2D\_equal} \mapsto \lambda p.(\text{self.x} = p.x \wedge \text{self.y} = p.y)$$

and adopt the convention that only the local names are ever overridden. In this way, we can express the combined *equal* method for Point3D as:

$$\text{equal} \mapsto \lambda q.(\text{self.Point2D\_equal}(q) \wedge \text{self.z} = q.z)$$

which simplifies in accordance with our first model, above. The method combination strategies using inheritance were shown to be equivalent to wholesale method replacement, demonstrating again the usefulness of the theoretical model. The model also provided the pseudo-variable *super* with two semantic interpretations, corresponding to the meanings of this variable in Java and Smalltalk. We also provided an original renaming operator ® to account for Eiffel's behaviour.

REFERENCES

[1] 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

[2] 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

[3] 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

[4] A Goldberg and D Robson, *Smalltalk 80: The Language and its Implementation*, Addison-Wesley, 1983.

[5] 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.

[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.