Article:

Simons, A.J.H. (2005) The theory of classification part 17: multiple inheritance and the resolution of inheritance conflicts. Journal of Object Technology, 4 (2). 15 - 26. ISSN 1660-1769

JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 4, No.2, March-April 2005

The Theory of Classification Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts

Anthony J H Simons, Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK

1 INTRODUCTION

This is the seventeenth article in a regular series on object-oriented theory for non-specialists. Using a second-order λ-calculus model, we have previously modelled the notion of *inheritance* as a short-hand mechanism for defining subclasses by extending superclass definitions. Initially, we considered the inheritance of *type* [1] and *implementation* [2] separately, but later combined both of these in a model of *typed inheritance* [3]. By simplifying the short-hand inheritance expressions, we showed how these are equivalent to canonical class definitions. We also showed how classes derived by inheritance are type compatible with their superclass. Further aspects of inheritance have included method combination [4], mixin inheritance [5] and inheritance among generic classes [6].

Most recently, we re-examined the ⊕ inheritance operator [7], to show how extending a class definition (the *intension* of a class) has the effect of restricting the set of objects that belong to the class (the *extension* of the class). We also added a type constraint to the ⊕ operator, to restrict the types of fields that may legally be combined with a base class to yield a subclass. By varying the form of this constraint, we modelled the typing of inheritance in Java, Eiffel, C++ and Smalltalk. Object-oriented languages vary widely in their policies on inheritance. Some, like Smalltalk, Objective C and Java, only support *single inheritance*, whereby a class may have at most one parent class. Others, like Flavors, Eiffel, CLOS and C++, support multiple inheritance, whereby a class may have possibly many parent classes. In this article, we consider first the theoretical issues raised by combining multiple implementations. Then, we consider what it means for an object to belong to multiple parent classes, defining the notion of *multiple classification*.

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification, Part 17: Multiple Inheritance and the Resolution of Inheritance Conflicts", in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp 15-26 http://www.jot.fm/issues/issue_2005_03/column2

2 MULTIPLE RECORD COMBINATION

In the *Theory of Classification*, we model objects as simple records, whose fields map from labels to functions, representing their methods. Inheritance is modelled as a kind of record combination, in which *extra* fields are added to the fields of a *parent* object to yield the desired union of fields in the *child* object:

$$\text{child} = \text{parent} \oplus \text{extra}$$

In the resulting *child*, fields obtained from the *extra* extension may replace similarly-labelled fields obtained from the *parent*, modelling the notion of method overriding. This is assured by the right-handed preference of the ⊕ union with override operator [2, 7].

In single inheritance, the *child* obtains all the fields of its *parent*, adding the *extra* fields to these. Intuitively, in multiple inheritance, the *child* must somehow obtain all the fields of multiple parents, adding the *extra* fields to these. We can think of this initially as a kind of multiple record combination, in which the operator ⊕ is used many times to combine the fields of the parents and then combine this result with the *extra* fields:

$$\text{child} = \text{father} \oplus \text{mother} \oplus \text{extra}$$

However, this establishes a particular kind of multiple inheritance, in which records are combined in a strict left-to-right order, with fields in the later records overriding similarly-labelled fields of the earlier records. The above expression is evaluated in the order:

$$(\text{father} \oplus \text{mother}) \oplus \text{extra}$$

such that fields from *mother* will possibly override fields in *father*; and then fields in *extra* will possibly override fields in the result of the first combination. This rule is similar to the multiple mixin combination of Flavors [8, 5], and is most like the recency-based superclass ordering rule from LOOPS [9]. The fields that you get in the result of multiple inheritance expressions using ⊕ are critically dependent on the order in which you list the parent classes. But, as we shall see below, this is not the only strategy that could be proposed; nor is it perhaps the best strategy.

3 MULTIPLE INHERITANCE POLICIES

Theoretically, we start from the premise that the *child* should obtain at least the union of the fields of its *mother* and *father*. If there is no limit on the number of parent classes, this is a *distributed* union, of the form:

$$\text{parent1} \cup \text{parent2} \cup \text{parent3}$$

The first design choice in any programming language is whether this should be a simple union, or a disjoint union of fields. In a *simple union*, fields with the same labels on both sides are merged, so that only one copy of a field is retained. In a *disjoint union*, fields with the same labels on both sides are considered distinct, so they are not merged.

The simple union policy is usually adopted on the grounds that unique names should be chosen for fields everywhere, since the same name should always refer to the same property. It is perhaps the most theoretically challenging option, since it requires an automatic rule for merging fields. Every time two versions of a field with the same label are encountered, one must be preferred over the other, usually by determining the order of precedence among ancestor classes. Object-oriented languages have proposed many strategies for this, ranging from simple class pre-order [8, 9] to sophisticated topological sorting algorithms for linearising the multiple inheritance graph [10, 11]. Languages with automatic inheritance policies include: Flavors [8], LOOPS [9] and CLOS [11].

The disjoint union policy is usually adopted on the grounds that field-names were ill-chosen and therefore both inherited versions of the field are required in the child. This is the default policy in C++ [12] and for identically-labelled fields introduced at multiple points in Eiffel [13]. This policy gives rise to ambiguity when fields are accessed in the scope of the child. In C++, the two fields must be qualified explicitly by the name of each parent, to resolve the ambiguity, whereas in Eiffel, the programmer must rename one or other of the conflicting fields in the child's inheritance clause. A benefit of the disjoint union policy is that the fields of a child may always be computed locally from the fields of its immediate parents, without worrying about the order in which its more distant ancestors were declared. A disadvantage is that inheritance graphs with fork-join patterns will give rise to unnecessary duplications of fields that were declared in a common ancestor.

For this reason, Eiffel distinguishes the "repeated inheritance" of the same field, via multiple paths from a common ancestor, from the "multiple inheritance" of two distinct fields with the same names [13]. The default rule is to merge the common "repeatedly inherited" field. (The same effect can be achieved in C++ by using "virtual base classes", although this is a less elegant mechanism, required by the language's implementation strategy).

Merging or recombining fields is a complex issue. Over the years, object-oriented languages have proposed many different kinds of automatic rule for combining multiple parent classes. We shall consider a few here, to uncover their advantages and disadvantages.

LOOPS computes a local precedence order for the immediate parent classes. The child's fields take priority over the first parent's fields, which take priority over the second parent's fields, etc., on the basis that a child is "more like" its nearer parents than its distant ones [9]. Similarly, Flavors computes a global order for all classes, by merging all the local precedence orders (as defined above) and any duplicated classes in this list are eliminated, retaining the most recent copy nearest to the front. However, if the local orders are found to be globally inconsistent, for example by requiring class A to precede class B and, at the same time, class B to precede class A, the definition is rejected [8]. Both of these adopt recency-based criteria for choosing which field to retain, rather like Touretzky's "inferential distance" algorithm [14], according to which a class retains the version of the field that was defined closest to it in the inheritance graph. Where the graph forks and joins, all paths leading up to the joins are explored before the path beyond the join is considered.

In CLOS [11], all possible ordered pairs of classes are computed, such that the child precedes each of its parents separately, and parents precede each other pairwise according to their local ordering declared in the child. A topological sorting algorithm then computes a global order for all classes, preserving the pairwise constraints. Again, if no globally consistent order can be found, the definition is rejected [10]. The aim of this precedence-based scheme is to preserve, as much as possible, the local ordering of parents in a class, whether or not the class is included as the ancestor of another class. However, even this sophisticated algorithm for linearising the multiple inheritance graph delivers counter-intuitive results for certain lattices [15]. If inheritance is only a local short-hand for a canonical class definition, the way a child class inherits from its parents should not depend on unexpected interactions between the ordering of its more distant ancestors (see arguments in [13], especially pages 246-250).

4 INHERITANCE CONFLICT RESOLUTION

The *Theory of Classification* adopts a particular "reference model" of multiple inheritance, which captures aspects of both the automatic and explicitly-specified policies on multiple inheritance. The compromise is motivated by examining the nature of inheritance conflicts. An *inheritance conflict* arises when a class obtains the same named method from more than one parent – the resolution of this conflict determines which of these methods (perhaps both) should be incorporated in the child. We distinguish *accidental* and *recombinant* inheritance conflicts.

- An *accidental* conflict arises from inheriting two semantically distinct methods, introduced in two places, which accidentally have the same name.
- A *recombinant* conflict arises from inheriting two semantically related methods, which were originally introduced at a single point in the multiple inheritance graph.

As discussed above, an object-oriented language may prefer to merge conflicting definitions (simple union) or preserve both (disjoint union). Disjoint union really exists to support the resolution of accidental conflicts. We consider it inappropriate for the fundamental theoretical model to have to rectify poor naming conventions! Instead, we assume that accidental conflicts can be resolved by renaming one of the conflicting methods throughout, in the class hierarchy (here, perhaps better called a *heterarchy*, since it is a full directed, acyclic graph). By removing accidental conflicts, our model only has to provide for the resolution of recombinant conflicts, by simple union. But this is harder than at first it seems. We do not simply want to achieve Eiffel's "repeated inheritance", but also want to recognise that the common field may have been redefined in either, or both branches leading to the parents. Semantically, this is still the "same" operation (albeit in a refined form).

Ideally, a child class should be a deterministic synthesis of the most specific aspects of its parent classes, without undue prejudice to either parent. This is what is wrong with existing automatic inheritance schemes: they force the programmer to prioritise one whole parent class before the other, whereas what we really desire is a scheme that prioritises individual methods, by recency of their definition. However, we think it is inappropriate for objects to have to reason about the points at which methods were introduced in the inheritance graph, since this mitigates against the "locality" of inheritance expressions. The following cases may therefore be distinguished:

- the father and mother classes may have no fields in common; in which case multiple inheritance should lead to the straightforward concatenation of their fields;
- the father and mother classes may have some commonly-labelled fields, which are pairwise identical, since they are inherited from a common ancestor; in which case only one copy of each of these should be retained;
- the father and mother classes may have commonly-labelled fields which are *not* pariwise identical, due to further specialisation in one or other parent since their introduction; in which case automatic selection is impossible.

In the last case, even though the desire is to select the most specific redefinition of a method, a system that only has local knowledge of the immediate parents cannot detect which of the versions of this method was redefined most recently. Instead, the programmer must specify explicitly which method should be retained (sometimes a combination of both).

5 SYMMETRICAL COMBINATION

A variant of the $\oplus$ operator is defined below to allow the construction of multiple inheritance expressions. The new *symmetrical* record combination operator is written $\otimes$ and its specific character is that it treats both its left- and right-hand operands fairly, rather than preferring its right-hand operand, like $\oplus$. It is defined to obey the three principles described above in section 4. Where it can determine which field to select, it does so; and where it cannot, it leaves the result of the combination undefined, so that the programmer may later choose explicitly which field to include in the child class.

The symmetrical operator $\otimes$ is actually a short-hand for a typed second-order polymorphic function called *merge*, designed for merging two parent classes fairly:

$$
\begin{aligned}
&\text{merge} : \forall\text{Father}.\ \forall(\text{Mother M Father}).\ \text{Father} \rightarrow \text{Mother} \rightarrow (\text{Father} \wedge \text{Mother})\\
&= \lambda\text{Father}.\ \lambda(\text{Mother M Father}).\ \lambda(\text{father} : \text{Father}).\ \lambda(\text{mother} : \text{Mother}).\\
&\quad \{\ \text{label} \mapsto \text{value} \mid (\text{label} \in \text{dom(father)} \cup \text{dom(mother)}) \wedge\\
&\qquad\qquad (\text{label} \in \text{dom(father)} \wedge \text{label} \in \text{dom(mother)} \Rightarrow\\
&\qquad\qquad\qquad (\text{father(label)} = \text{mother(label)} \Rightarrow \text{value} = \text{father(label)}) \wedge\\
&\qquad\qquad\qquad (\text{father(label)} \neq \text{mother(label)} \Rightarrow \text{value} = \perp)) \wedge\\
&\qquad\qquad (\text{label} \in \text{dom(father)} \wedge \text{label} \notin \text{dom(mother)} \Rightarrow\\
&\qquad\qquad\qquad \text{value} = \text{father(label)}) \wedge\\
&\qquad\qquad (\text{label} \notin \text{dom(father)} \wedge \text{label} \in \text{dom(mother)} \Rightarrow\\
&\qquad\qquad\qquad \text{value} = \text{mother(label)})\ \ \}
\end{aligned}
$$

This says that "two records *father* and *mother* of the respective types *Father* and *Mother* may be merged, if these types satisfy a merging type-constraint M. The result of the merger is a map containing the union of fields from both parents, in which fields unique to the *father* or to the *mother* are inherited unchanged, but fields common to both *father* and *mother* are tested to see if their values are identical. If they are, then one copy is retained (the father's), but if they are not, then the result of the merger is undefined." In the above definition, $\perp$ denotes the undefined value and function-call expressions like: *father(label)* denote the *value* stored opposite the *label* in the *father* object, which is a map from labels to values. This definition uses the type constraint M that was introduced in the previous article [7], which says that two record types may be merged if their common fields have the same types. This is sufficient for second-order polymorphic typed inheritance, adopted in the *Theory of Classification* [7, 1].

We can now define the symmetrical operator $\otimes$ in terms of *merge*:

$$\forall\beta.\ \forall\varepsilon.\ \otimes_{\beta,\varepsilon} = \text{merge}\ [\beta,\varepsilon]$$

This creates a simply-typed version of $\otimes$ for each pair of records we wish to combine. Really, $\otimes$ is just a short-hand for *merge* with two types already supplied. To see how $\otimes$ works, we will seek to construct a *point3D* object by combining a two-dimensional *point* object with *zcoord*, a mixin object representing the third dimensional coordinate. Initially, we shall just observe the operation of $\otimes$ in isolation:

$$\text{point} = \mu\text{self}.\{x \mapsto 2,\ y \mapsto 3,\ \text{equal} \mapsto \lambda p.(\text{self}.x = p.x \wedge \text{self}.y = p.y\}$$

$$\text{zcoord} = \mu\text{this}.\{z \mapsto 5,\ \text{equal} \mapsto \lambda q.(\text{this}.z = q.z)\}$$

$$\text{point3D} = \text{point} \otimes \text{zcoord} \ \Rightarrow\ \{x \mapsto 2,\ y \mapsto 3,\ z \mapsto 5,\ \text{equal} \mapsto \perp\}$$

In this initial example, *point3D* is constructed by merging the definitions of the two parent objects, *point* and *zcoord*. The result contains copies of the unique methods inherited from both parents, but the *equal* method was defined in both *point* and *zcoord*, so by the definition of $\otimes$, the body of *equal* is undefined. This is because we cannot automatically determine which version of the method we should inherit (we assume by convention that *equal* was declared at a single point in the inheritance graph, and is intended to stand everywhere for the same semantic operation; although the calculus cannot determine this).

Now, the example was deliberately chosen to illustrate a further aspect of automatic multiple inheritance that is not usually considered in object-oriented languages. It would in fact be a mistake to prefer one version of the *equal* method over the other; what we really desire is to inherit *both* parent methods, suitably combined. We may use method combination [4], of the kind used to explain super-method invocation, to achieve this:

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

$$\text{genZcoord} = \lambda\text{this.}\{\text{z} \mapsto 5,\ \text{equal} \mapsto \lambda\text{q.}(\text{this.z} = \text{q.z})\}$$

$$\begin{aligned}\text{genPoint3D} = \lambda\text{me.}(&\lambda\text{father.}\ \lambda\text{mother.}\mathbf{(}\text{father} \otimes \text{mother} \\ &\qquad \oplus \{\text{equal} \mapsto \lambda\text{r.}(\text{father.equal(r)} \wedge \text{mother.equal(r)})\ \}\ \mathbf{)} \\ &\text{genPoint(me) genZcoord(me))} \\ \Rightarrow\ \lambda\text{me.}\{&\text{x} \mapsto 2,\ \text{y} \mapsto 3,\ \text{z} \mapsto 5, \\ &\text{equal} \mapsto \lambda\text{r.}(\text{me.x} = \text{r.x} \wedge \text{me.y} = \text{r.y} \wedge \text{me.z} = \text{r.z})\}\end{aligned}$$

The revised example uses generators instead of objects, because we wish to unify self-reference in the manner explained in the earlier article [2]. The two parent objects are created using *genPoint(me)* and *genZcoord(me)*, where *me* is the self-reference introduced by *genPoint3D*. These parent objects are bound internally to the variables *father* and *mother*, in exactly the same way that we bound an inherited object to *super* in [2]. There is no reason why we should not have many super-objects in multiple inheritance; and this is what opens the way to novel kinds of method combination. The multiple inheritance expression is resolved by first combining *father* ⊗ *mother*. Since this yields an undefined body for the *equal* method, the child object now specifies explicitly how to combine the two inherited versions: it is the logical-and of the *father's* and *mother's* implementations for *equal*. The new version of *equal* is given in a record of additional methods, to be added to the merger of the parents, using the usual ⊕ operator. After combination, it will override the undefined placeholder for *equal*, yielding the desired version of *equal* for a 3D point.

The notion of multiple *super*-objects has not been used before in any mainstream object-oriented language. It is powerful enough to model any kind of explicit method recombination, whether preferring the left-hand or right-hand version, or, as in this case, creating a fusion of both. In C++, you can get a similar effect by defining a method which explicitly calls both parent methods, qualifying these by their owning classes. In Eiffel, the same effect may be achieved by a combination of renaming and redefining. There is no limit on the number of *super*-objects, since ⊗ may be used to combine any number of parents fairly, whose common methods may be accepted, refused or combined in all possible ways in the child.

6 MULTIPLE CLASSIFICATION

Turning now to the issue of types, intuitively the type of the child object must be compatible with the types of each of its parents. In the first-order subtyping model (c.f. Java, C++), we may express this as a dual subtyping condition, which yields an interesting result in terms of type intersections:

$$(\text{Child} <: \text{Father}) \wedge (\text{Child} <: \text{Mother}) \Rightarrow \text{Child} <: (\text{Father} \wedge \text{Mother})$$

"If the *Child* is a subtype of the *Father* and also of the *Mother*, then the *Child* is a subtype of the intersection[1] of the *Father* and *Mother* types." Type intersections were introduced in the previous article [7]. We showed how merging two object types results in a new type whose extension-set is the intersection of the extension-sets of the original two types. The extension of the *Child* type is a subset of each of its parents' extensions, which fits nicely with the idea that all the *Child's* instances should also be considered as belonging to the *Father's* and *Mother's* types.

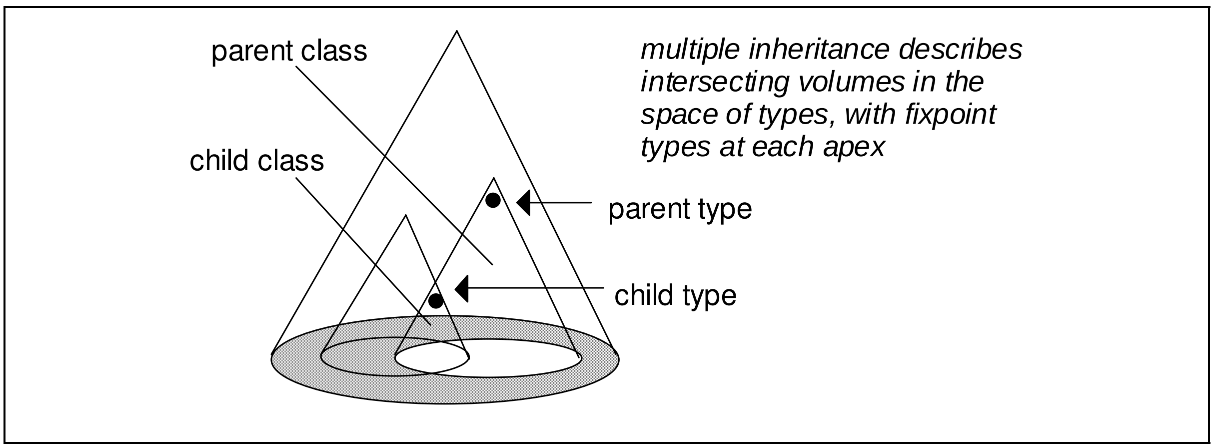


Figure 1: Multiple inheritance describes intersecting volumes

In the more sophisticated second-order parametric typing model [1, 3] (c.f. Eiffel, Smalltalk) there is a similar result which we can express using type generators:

$$\forall\tau \,.\, (\tau <: \text{GenFather}[\tau]) \wedge (\tau <: \text{GenMother}[\tau]) \Rightarrow \tau <: (\text{GenFather}[\tau] \wedge \text{GenMother}[\tau])$$

"If the self-type τ is a pointwise subtype of all types created using the *GenFather* generator and also of all types created using the *GenMother* generator, then it is a pointwise subtype of all intersection types created simultaneously from both generators."

[1] The symbol $\wedge$ is overloaded here to mean logical-and and type intersection. An intersection type contains the union of the fields of the two record type arguments.

The intersection type in the result is basically like the merger of two record types, in which τ has been replaced in turn by each self-type that satisfies the F-bound of both parents' generators. The constrained type expression: $\forall\tau <: \mathit{GenFather}[\tau] \wedge \mathit{GenMother}[\tau]$ is a new kind of F-bound, describing a family of types that occupy the intersection of the volumes described separately by each of the parent generators. Regular readers will recall that such a "space of types" is equivalent to the notion of a *class* in the *Theory of Classification*.

We can visualise this in figure 1. Here, the two parent classes are illustrated as intersecting cones. The volume where they intersect is the resulting child class you obtain by merging the two parents by multiple inheritance. (If the child were to add further methods, it would form a slightly smaller cone nesting inside the intersection volume). While polymorphic classes are represented by volumes, exact simple types are represented by points. If we recall that a class is a family of structurally-similar types, then all the exact types within a given volume satisfy the F-bound of the related generator and so possess at least the set of methods described in the generator's body. A type which resides inside the intersection volume therefore possesses at least the methods of both generators.

In figure 1, the point at the apex of a cone represents the least type that is still a member of that class. Mathematically, this simple type is the fixpoint of the generator used to describe the class. For multiple parent classes, we may describe this as:

$$\text{Father} = \mathbf{Y}\ \text{GenFather}, \quad \text{Mother} = \mathbf{Y}\ \text{GenMother}$$

but what is the least type that sits just inside the intersection volume (see figure 1)? This is the least type of the child class. Mathematically, we may create this *Child* type by merging the generators of the two parents and then taking the fixpoint of the result:

$$\text{Child} = \mathbf{Y}\ \lambda\tau.(\text{GenFather}[\tau] \wedge \text{GenMother}[\tau])$$

The least *Child* type is exactly the fixpoint of the type generator intersections (extensional definition). In other words, this is the fixpoint of the generator obtained by taking the union of the fields of both parent generators (intensional definition), after unifying the self-type τ. The *Child* type satisfies the F-bounds for each of its parent classes:

$$\text{Child} <: \text{GenFather}[\text{Child}], \quad \text{Child} <: \text{GenMother}[\text{Child}]$$

and the least *Child* is exactly equivalent to the type intersection:

$$\text{Child} = \text{GenFather}[\text{Child}] \wedge \text{GenMother}[\text{Child}]$$

by the fixpoint theorem. So, all the formal properties that were established in earlier articles for classes nesting in a single classification hierarchy [1, 3] also apply in a uniform way to overlapping classes created by multiple inheritance. It is pertinent to describe this notion as *multiple classification*, the idea that an object may have a type belonging simultaneously to more than one overlapping class.

7 CONCLUSION

In this article, we have developed a flexible formal model for multiple inheritance. By comparing existing programming language models for resolving inheritance conflicts automatically and then contrasting these with models that rely on explicit resolution by the programmer, we came up with a compromise that resolves fork-join "repeated inheritance" automatically, but relies on a form of explicit super-method combination to resolve other cases where the methods to be recombined have been redefined at some point in the inheritance graph. Reasoning globally about the points at which methods are introduced could also resolve this automatically, but we considered this inappropriate, since it conflicts with the view that inheritance should be a local short-hand for defining classes incrementally by extension from its immediate parents. We excluded accidental name conflicts from the formal model, on the grounds that these could be solved simply by renaming one or other method systematically in the inheritance graph.

Multiple inheritance is different from ordinary inheritance, because it involves symmetrical merging as well as asymmetric extension. To merge multiple parent classes fairly, a different combination operator $\otimes$ was defined, which treats both of its operands symmetrically. It constructs records containing the union of distinctly-labelled fields, merges identically-labelled fields that map to identical values, and declares the result of the merger to be undefined otherwise. This is the only sensible automatic choice, given that the programmer might wish to retain the left-hand, right-hand, or possibly a fusion of both versions of the method in the resulting child. The latter option has not been treated before in conflict-resolution schemes. The operator $\otimes$ was defined as a set of simply-typed operators created by instantiating a polymorphic typed function *merge*, in the same way that $\oplus$ was defined out of *extend* in the previous article [7].

Finally, it was shown that objects created by simultaneous extension from several parents have types which belong to multiple classes, in the *Theory of Classification*. The notion of multiple classification was visualised as creating intersections in the space of types, satisfying natural intuitions about belonging to more than one class.

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_07/column4

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

[3] 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

[4] A 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

[5] A J H Simons, "The theory of classification, Part 15: Mixins and the superclass interface", in *Journal of Object Technology*, vol. 3, no. 10, November-December 2004, pp. 7-18. http://www.jot.fm/issues/issue_2004_11/column1

[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] 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

[8] D A Moon, "Object-oriented programming with Flavors", *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl., pub. ACM Sigplan Notices, 21(11),* (ACM Sigplan, 1986), 1-6.

[9] D Bobrow and M Stefik, *The LOOPS Manual* (Palo Alto: Xerox PARC, 1983).

[10] D Bobrow, L DeMichiel, R Gabriel, S Keene, G Kiczales and D Moon, *Common Lisp Object System Specification, X3J13 Document 88-002R*, June, 1988.

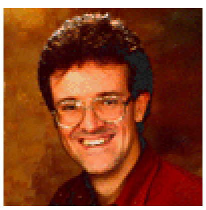
[11] S E Keene, *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS* (Reading MA: Addison-Wesley and Symbolics Press, 1989).

[12] B Stroustrup, "Multiple inheritance for C++", *Proc. European Users' Group Conf.*, (Helsinki, 1987).

[13] B Meyer, *Object-Oriented Software Construction, 1st ed.*, (Wokingham: Prentice-Hall, 1988). Note – cited page numbers refer to this earlier edition, rather than the later, greatly enlarged edition.

[14] D Touretzky, *The Mathematics of Inheritance Systems* (Palo Alto: Morgan Kaufmann, 1986).

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.