JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 4, No. 1, January-February 2005

The Theory of Classification Part 16: Rules of Extension and the Typing of Inheritance

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

1 INTRODUCTION

This is the sixteenth article in a regular series on object-oriented type theory for non-specialists. Earlier articles have built up λ-calculus models of objects [1], classes [2], inheritance [3, 4] and generic template types [5]. *Classification* describes the way in which typed objects fit into a hierarchy of classes, which nest inside each other [3]. *Inheritance* is the short-hand mechanism for definining a subclass by extension from another class, specifying only the additions and modifications to the base class [4]. Previously, we have modelled the inheritance of *type* [3] and *implementation* [4] and combined both of these in a model of *typed inheritance* [6]. We showed how short-hand inheritance expressions can be simplified to yield a canonical subclass definition that is type compatible with the base superclass from which it was derived. Further aspects of inheritance have included method combination [7], mixin inheritance [8] and inheritance among generic classes [5].

The current article examines the mechanism of inheritance in more detail, looking at the constraints on what may or may not be added to a class during inheritance. Most object-oriented languages have restrictions on the types of overriding methods, to ensure that the resulting subclass is still type compatible with the superclass. This requires more precise rules about the typing of $\oplus$, the inheritance operator. Previously, we thought of $\oplus$ as a polymorphic map override operator that could combine two maps of any types, irrespective of the types of the fields. We now require inheritance to be properly typed, in the second-order F-bounded λ-calculus, so that we can restrict the kinds of extension that are deemed legal. The extended type resulting from inheritance is shown to be an *intersection type* [9].

2 MERGING RECORDS AND RECORD TYPES

In the *Theory of Classification*, we model *objects* as simple records of functions, representing their methods, and *object types* as the corresponding record types, containing the signatures of these methods. This leads to a model of inheritance based on record union with overriding, denoted by the operator ⊕. Intuitively, this creates a longer record by combining two shorter records:

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

in which the *derived* result contains the union of all the fields of *base* and *extra*, but fields from *extra* are preferred over any identically-labelled fields from *base* in the result [3]. This right-handed preference of ⊕ models the notion of *overriding* in object-oriented languages, in which the *derived* class may replace some of the methods present in the *base* class with redefined versions supplied in the *extra* extension.

So far, we have always used ⊕ in a context where the replacement fields have the same types as the original versions. As a consequence, we have been able to construct *Derived* record types using simple set union ∪ to merge the two sets of type signatures in the corresponding *Base* and *Extra* type-records:

$$\text{Derived} = \text{Base} \cup \text{Extra}$$

We think of objects as sets of maplets from labels to functions, and object types as corresponding sets of maplets from labels to function types. It is reasonable to think of the merger of two record types as the set union of their respective sets of maplets, since any fields with identical labels will also have identical types.

Merging in the subtyping model

In the subtyping model [10], we must consider the possibility that fields of different types may be merged. This is because the record subtyping rule allows fields to be replaced by subtype fields. In this case, we cannot use ∪ to merge the record types. The following is a plausible definition of a record subtype by extension:

$$\text{Vehicle} = \{\text{owner} : \to \text{Person}, \text{home} : \to \text{Location}\}$$

$$\begin{aligned}\text{Car} &= \text{Vehicle} \oplus \{\text{home} : \to \text{Garage}, \text{range} : \text{Litres} \to \text{Kilometres}\} \\ &= \{\text{owner} : \to \text{Person}, \text{home} : \to \text{Garage}, \text{range} : \text{Litres} \to \text{Kilometres}\}\end{aligned}$$

That is, we wish to obtain the subtyping relationship *Car* <: *Vehicle*. According to the record subtyping rule [10], this requires *Car* to have at least as many fields as *Vehicle* (it has one more) and requires any replacement fields to be subtypes. The field *home* : → *Location* is replaced by *home* : → *Garage*, so the subtyping condition is only met if

Garage <: *Location*, which is plausible. We use ⊕, rather than ∪, to combine the two record types, because we wish the right-hand types of any common fields to be retained in the result.

The above model could be used in languages like Java and C++, which are based on types and subtyping. However, in these languages, a replacement method is typically expected to have *exactly* the same type as the method it replaces. In more recent versions of C++, the type of *this* is allowed to be more specific in the result of a replacement method. The overriding rules of Trellis are closest to the subtyping model, allowing both method argument and result types to change in accordance with the function subtyping rule [10].

Merging in the subclassing model

In the subclassing model, we merge generators and type generators, rather than objects and object types [3, 4]. A curious thing happens when merging records according to the (F-bounded) parametric model: the parameters are instantiated or replaced before any record combination occurs. This means that references to different parametric types on the left and right hand sides become unified before record combination. As a result, record combination always merges records whose common fields have the same types. The following illustrates a simple type generator example, in which a (somewhat simplified) *Integer*-class generator is defined by extension from a *Number*-class generator:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma,\ \text{equal} : \sigma \to \text{Boolean}\}$$

$$\begin{aligned}\text{GenInteger} &= \lambda\tau.(\text{GenNumber}[\tau] \cup \{\text{minus} : \tau \to \tau,\ \text{equal} : \tau \to \text{Boolean}\})\\ &= \lambda\tau.\{\text{plus} : \tau \to \tau,\ \text{minus} : \tau \to \tau,\ \text{equal} : \tau \to \text{Boolean}\}\end{aligned}$$

We obtain the subclass relationship: ∀τ. *GenInteger*[τ] <: *GenNumber*[τ]. This is achieved by making sure that *GenInteger* has more fields than *GenNumber* and that the common fields are typed in terms of parameters which can be unified before record combination occurs. In the inner type-record combination expression, *GenNumber*[τ] causes the substitution of {τ/σ} in the body of *GenNumber*, such that the record types on both sides of the union ∪ have field types which refer to the self-type uniformly as τ; and, in particular, the common field *equal* has the same type: τ → *Boolean* on each side of the union.

In fact, it is always the case, in the subclassing model, that self-type parameters are unified before record combination. Likewise, generic type parameters may be unified before combination [5] (see also 5.4 below). So, the simpler union ∪ of type-records appears to be all that we need in the subclassing model.

3 SUBSETS, SUBTYPES AND TYPE INTERSECTIONS

Throughout this series, we have been careful to distinguish the notation for the subset $\subseteq$ and subtype <: relationships. This is because the relationship between the two depends on whether we are thinking about the set of values in a concrete type, or the set of type signatures in a record type. These two set-theoretic constructions are different, and they correspond to different subtyping relationships.

Concrete versus abstract representation

The fundamental relationship is that types may be modelled as sets. When we assert that an element is of a particular type, this is equivalent (in the model) to asserting that the element is a member of a particular set:

$x : T \equiv x \in T$ "x is of type T means that x is in the set T"

From this it follows that a subtype may be modelled as a subset:

$S <: T \equiv S \subseteq T$ "S is a subtype of T means that S is a subset of T"

In the universe of types, we want to show that if $x : S$ and $S <: T$, then $x : T$ also. In the universe of sets, $x \in T$ follows from $x \in S$ and $S \subseteq T$, by the definition of the subset relationship:

$S \subseteq T \equiv \forall x . x \in S \Rightarrow S \in T$ "S is a subset of T means that if x is in S, then x is also in T"

This is the fundamental relationship, which applies to types defined *concretely* as sets. When we move to defining types *abstractly*, in terms of their syntactic signatures, then the relationship is different. A record type with more signatures denotes a subtype of a record type with fewer signatures. For example, if the following record types are defined:

$S = \{plus : Integer \rightarrow Integer, minus : Integer \rightarrow Integer\}$

$T = \{plus : Integer \rightarrow Integer\}$

then it is clear that S is the larger record type and contains the signatures of T, which we express as $T \subseteq S$ in the universe of signature-based types. However, it is also clear that S denotes a subtype of T, because every object that satisfies the interface S will also satisfy the interface T. The record subtyping rule (see [10]) expresses this fact.

Intensional versus extensional definition

There are grounds for confusion here: in one model, we say: $S \subseteq T$; but in the other model, we say: $T \subseteq S$. The difference is that, in the first model, we are comparing sets of values, but in the second model, we are comparing sets of type signatures. To see how these both ultimately reflect the same subtyping relationship, we have to distinguish the *intensional* and *extensional* definitions of a type.

- The *extension* of a type is the enumeration of the set of elements that it contains, for example, the *Boolean* type has the extension: {*false*, *true*}
- The *intension* of a type is the enumeration of the set of properties that characterise the type, for example, the (existentially defined [1]) *Boolean* type has the intension:

$$\begin{aligned}\text{Boolean} \equiv \exists \text{b}.\{&\text{not} : \text{b} \rightarrow \text{b}, \text{and} : \text{b} \times \text{b} \rightarrow \text{b}, \text{or} : \text{b} \times \text{b} \rightarrow \text{b},\\ &\text{implies} : \text{b} \times \text{b} \rightarrow \text{b}\}\end{aligned}$$

followed by a set of axioms defining the meanings of these operations.

To unify the concrete and abstract views of a type, it is easiest to imagine the extension of the type, that is, the set of values (or objects) populating the type. This is the usual view adopted in type theoretical treatments. In object-oriented programming, we usually characterise a class intensionally, that is, by its properties (type signatures). From this, we have to imagine the extension of the class, that is, the possible set of objects which could populate it.

Intersection types

Here, we try to establish the relationship between intensional (signature-based) types and extensional (value-based) types. Earlier, we modelled type extension as the union of type-records: *Derived* = *Base* ∪ *Extra*. In terms of sets of signatures, this means that *Base* ⊆ *Derived* and *Extra* ⊆ *Derived*, that is, both *Base* and *Extra* contain a subset of the signatures of *Derived*, which is a longer record type. By the record subtyping rule [10], a longer record type with more field signatures is a subtype. According to this, the direction of the subtyping relationship is *contravariant* with the direction of the signature subsets: *Derived* <: *Base* and also *Derived* <: *Extra*. This is a fundamental property of type hierarchies: the larger the interface, the smaller the set of objects which may satisfy it.

From this, we may reason about the extensions of each type. Instances of the *Derived* type may also be considered instances of the *Base* type (and instances of the *Extra* type), by the subtyping rule of subsumption. So, the extension set of the *Base* type is larger than that of the *Derived* type; likewise the extension set of the *Extra* type is larger than that of the *Derived* type. Since elements of the *Derived* extension are also members of the *Base* and *Extra* extensions, the membership of the *Derived* extension is precisely the *intersection* of the memberships of the *Base* and *Extra* extensions.

For this reason, the kinds of types created by merging signature-based types are sometimes known as *intersection types*. Instead of writing: *Base* ∪ *Extra* (in the world of signatures), we write: *Base* ∧ *Extra* (in the world of sets), to denote the intersection of the *Base* and *Extra* types. Much of the fundamental research on this was done by Compagnoni and Pierce in the mid-1990s [9, 11]. They developed a type system called

"System $F^{\omega}\wedge$", pronounced "System F-omega-meet", a higher-order type system with intersection types.

4 CONSTRAINING THE INHERITANCE FUNCTION

Many object-oriented languages have strict rules about method overriding, during inheritance, because they wish to preserve type compatibility (either subtyping, or subclassing) in the derived type. In C++ or Java, any replacement method must have *exactly* the same type as the original method it replaces. This imposes a constraint on the inheritance function, which we should like to capture in the model. We shall try to capture this constraint in a general enough way that it will apply both to the first-order *subtyping* model of inheritance, as found in Java, and also in the second-order *subclassing* model of inheritance, which is a more appropriate general model for object-oriented programming, in which polymorphic *classes* and simple *types* are actually distinct notions.

The *extend* inheritance function

Inheritance is only well-defined if the *Extra* record provides fields whose types "merge" with the types of the *Base* record. This "merge" condition is expressed as a constraint M between the two record-types in the following F-bounded, second-order definition of the inheritance function *extend*, which we shall now use in place of the earlier unconstrained $\oplus$ map override operator:

$$\begin{aligned}
&\text{extend} : \forall \text{Base}.\ \forall(\text{Extra M Base}).\ \text{Base} \rightarrow \text{Extra} \rightarrow (\text{Base} \wedge \text{Extra})\\
&= \lambda\text{Base}.\ \lambda(\text{Extra M Base}).\ \lambda(\text{base} : \text{Base}).\ \lambda(\text{extra} : \text{Extra}).\\
&\qquad\quad \{\ \text{label} \mapsto \text{value} \mid (\text{label} \in \text{dom(base)} \cup \text{dom(extra)}) \wedge\\
&\qquad\qquad\qquad (\text{label} \in \text{dom(extra)} \Rightarrow \text{value} = \text{extra(label)}) \wedge\\
&\qquad\qquad\qquad (\text{label} \notin \text{dom(extra)} \Rightarrow \text{value} = \text{base(label)})\ \}
\end{aligned}$$

This definition says that: "*extend* takes two type arguments, *Base* and *Extra*, where *Extra* must satisfy the type-merge condition with *Base*, then two record arguments, *base : Base* and *extra : Extra*, and constructs a result by merging the two records, which has the intersection type (*Base* $\wedge$ *Extra*). The result is a map of label-value pairs, such that the set of labels is the union of the domains of *base* and *extra*, and the values are preferentially taken from *extra*, if the label is present in *extra*, otherwise taken from *base*." (Note that *base(label)* maps to the *value* opposite *label* in the *base* map [4]).

Readers will note that the body of *extend* is identical to the body of $\oplus$ in earlier articles [4]. These two functions are essentially the same, except that *extend* is now properly-defined in the second-order λ-calculus, with type arguments (*Base* and *Extra*) as well as value arguments (*base* and *extra*). The type arguments were conveniently omitted from the earlier definition of $\oplus$, which we imagined could be applied directly to two record values. We can retrospectively define the operator $\oplus$ in terms of *extend*:

$$\forall\beta.\ \forall\varepsilon.\ \oplus_{\beta,\,\varepsilon} = \text{extend}\,[\beta,\varepsilon]$$

This creates a simply-typed version of ⊕ for each pair of records we wish to combine. Really, ⊕ is just a short-hand for *extend* with two types already supplied.

The M type-merge condition

The all-important type-merge condition M is a constraint that restricts the record-types that are allowed to be substituted for the *Extra* type argument. Although this is a rather special condition, constructed for the purpose of typing inheritance, it is syntactically no different from other kinds of restriction, such as the F-bound: $\forall(\tau <: F[\tau])$, which restricts a type τ to be a subtype of some generator expression. Here, we restrict *Extra* to range over those record types whose field-types enter into a particular relationship with the types of the *Base* fields. The constraint M is defined as follows:

$$\begin{aligned}&\forall\text{Base}.\ \forall\text{Extra}.\ \text{Extra M Base} ::= \\ &\qquad \forall\text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}.\ \text{Base(label)} = \text{Extra(label)}\end{aligned}$$

This says: "For all types *Base* and *Extra*, the type-merger condition *Extra* M *Base* is defined as being satisfied if, for all common fields in *Base* and *Extra* with identical labels, the corresponding types are also equal".

For this, we must assume that the notion of "type equality" is well-defined. In full, this might be expressed by a whole set of rules. For the model of inheritance used in the *Theory of Classification*, we require the following kinds of type equality:

$t = t$	-- identity of simple types
$\tau = \tau$	-- identity of type parameters
$(S \times T) = (S \times T)$	-- equality of product types, where $S, T ::= t \mid \tau$
$(S \rightarrow T) = (S \rightarrow T)$	-- equality of function types, where $S ::= t \mid \tau \mid T \times T$ and $T ::= t \mid \tau$

where t is a simple type, τ is a type parameter, and S, T are metavariables ranging over simple types and type parameters. (Type rules sometimes use metavariables like this to save having to repeat the same rule for simple types and parametric types).

Constrained typed inheritance

The result of *extend* is well-defined if *Extra* M *Base* ("*Extra* merges with *Base*"). This rule constrains inheritance just enough to behave exactly like typed inheritance in Java, but disallows certain other kinds of inheritance For example, the Trellis-style of inheritance in section 2.1 is now ruled out by the type-merge condition, because a field is replaced by a field which has a *different* type. The *Base* and *Extra* records have the types:

$$\text{Base} = \{\text{owner} : \to \text{Person}, \text{home} : \to \text{Location}\}$$
$$\text{Extra} = \{\text{home} : \to \text{Garage}, \text{range} : \text{Litres} \to \text{Kilometres}\}$$

and the common labels in *dom(Base)* $\cap$ *dom(Extra)* $\Rightarrow$ *{home}*. However, when we compare the corresponding types, we find that: *Base(home)* $\Rightarrow$ *Location* and *Extra(home)* $\Rightarrow$ *Garage*. So, we establish that common field-types are not identical: *Base(home)* $\neq$ *Extra(home)*, and therefore that M is not satisfied. To pass the type-merge condition, the *Extra* record would have to redefine the *home* field with the same type: *home* : $\to$ *Location*, as in Java.

By deliberate design, the same type-merger rule allows the kind of unions of type-records we require for the merger of parameterised record types, which are used in the subclassing model of inheritance. Repeating the example from section 2.2:

$$\text{GenNumber} = \lambda\sigma.\{\text{plus} : \sigma \to \sigma, \text{equal} : \sigma \to \text{Boolean}\}$$

$$\begin{aligned}\text{GenInteger} &= \lambda\tau.(\text{GenNumber}[\tau] \cup \{\text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\})\\ &= \lambda\tau.\{\text{plus} : \tau \to \tau, \text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}\end{aligned}$$

The *Base* and *Extra* records have the following types, after the parameter substitution $\{\tau/\sigma\}$:

$$\text{Base} = \{\text{plus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}$$
$$\text{Extra} = \{\text{minus} : \tau \to \tau, \text{equal} : \tau \to \text{Boolean}\}$$

and the common labels in *dom(Base)* $\cap$ *dom(Extra)* $\Rightarrow$ *{equal}*. When we compare the corresponding types, we find that: *Base(equal)* $\Rightarrow \tau \to$ *Boolean*, and: *Extra(equal)* $\Rightarrow \tau \to$ *Boolean*. Intuitively, these two types are identical; formally we would need to appeal to the equality of two function-types (see 4.2) based on the identity of the two argument type parameters τ and the identity of the two simple *Boolean* result types. Ultimately, the condition M is satisfied, so this is a legal extension.

5 VARIATIONS ON TYPED INHERITANCE

The standard "reference" model of inheritance consists of the *extend* inheritance function and the M type-merger constraint. This allows a record to be extended only if overriding fields have the same types as in the original fields they replace. The resulting intersection type is always a record-type consisting of the union of the signatures of the *Base* and *Extra* record types, since common fields have the same types. We now consider a number of object-oriented languages and examine how their models of typed inheritance differ from this reference model.

Inheritance in Smalltalk

Smalltalk is not strongly typed. However, certain rules are still observed about inheritance. A method can only override another method if its untyped "signature" is structurally similar, for example, the method *at:put:* always has the structural form:

at: anIndex put: anItem

Any method in a descendant class must have the same name and structural form in order to override this method. So, the "arity" of method arguments and results is always preserved, although nothing can be said about the individual types of each argument or result. Smalltalk can distinguish product types $\sigma \times \tau$ from basic types τ, but apart from this, all basic types (and parameters, considering that *self* has an F-bounded parametric type) are indistinguishable, and so must be considered equivalent. So, for Smalltalk, we should have to redefine the notion of type equality to allow $s = t = \sigma = \tau$ for all simple types s, t and all parameters σ, τ.

Inheritance in Trellis

The type-merger condition above is too restricting to describe inheritance in Trellis. Trellis allows full subtyping in its overriding rules, that is, methods may be replaced by other methods whose arguments have more general types and whose results have more specific types, according to the contravariant and covariant parts of the function subtyping rule. To handle Trellis, we should modify our definition of M:

$$\begin{aligned}&\forall \text{Base}.\ \forall \text{Extra}.\ \text{Extra}\ \text{M}_{\text{Trellis}}\ \text{Base} ::= \\ &\qquad \forall \text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}.\ \text{Extra(label)} <: \text{Base(label)}\end{aligned}$$

This now allows field types in the *Extra* record to be subtypes of common fields in the *Base* record. The resulting intersection type *Base* ∧ *Extra* may contain finer intersections of field types, for example, the extension of *Vehicle* in 2.1:

$$\begin{aligned}&\{\text{owner} : \to \text{Person},\ \text{home} : \to \text{Location}\} \wedge \\ &\quad \{\text{home} : \to \text{Garage},\ \text{range} : \text{Litres} \to \text{Kilometres}\} \\ &\Rightarrow \{\text{owner} : \to \text{Person},\ \text{home} : \to (\text{Location} \wedge \text{Garage}), \\ &\qquad \text{range} : \text{Litres} \to \text{Kilometres}\} \\ &\Rightarrow \{\text{owner} : \to \text{Person},\ \text{home} : \to \text{Garage},\ \text{range} : \text{Litres} \to \text{Kilometres}\}\end{aligned}$$

requires the nested intersection: *Location* ∧ *Garage* = *Garage*. (Constructively, *Garage* is the largest type which is a subtype of both *Garage* and *Location*).

Inheritance in Java and C++

The original type-merger condition describes exactly the constraint on inheritance in Java, in which all replacement methods must have exactly the same types as the methods they replace. This strict equality *nearly* describes the condition in C++, apart from the relaxation that applies to returned self-types. We can express this relaxation as:

$$\begin{array}{l}\forall\text{Base}.\ \forall\text{Extra}.\ \text{Extra}\ \text{M}_{\text{C++}}\ \text{Base} ::= \\ \quad \forall\text{label} \in \text{dom(Base)} \cap \text{dom(Extra)}. \\ \qquad \forall\sigma.\ \text{Base(label)} \neq (\sigma \rightarrow \text{Base}) \Rightarrow \text{Base(label)} = \text{Extra(label)} \wedge \\ \qquad \forall\sigma.\ \text{Base(label)} = (\sigma \rightarrow \text{Base}) \Rightarrow \text{Extra(label)} = (\sigma \rightarrow \text{Base} \wedge \text{Extra})\end{array}$$

saying that replacement methods must have the identical types *unless* they return the self-type, in which case methods of the function type: $\sigma \rightarrow$ *Base* must be replaced by methods of the function type $\sigma \rightarrow$ (*Base* $\wedge$ *Extra*). The resulting intersection type *Base* $\wedge$ *Extra* will be a subtype of *Base*.

C++ may also have type parameters in its method signatures, if the *template class* mechanism is used (and so will Java, from version 1.5 onward). The notion of type equality must therefore allow for the comparison both of exact types and type parameters (see 4.2).

Inheritance in Eiffel

The overriding rules of Eiffel allow methods to be replaced by methods whose arguments and results are both uniformly specialised. This is not legal within a simple subtyping regime; but Eiffel is not based on the subtyping model of inheritance. Elsewhere, Eiffel implicitly evolves the self-type (the type of *current*) under inheritance and anchors other types to the self-type, especially in binary methods[1] such as the infix "+" method in the *Numeric* class:

infix "+" (arg : **like** *current*) : **like** *current*

Because of this, it is tempting to think of Eiffel as following the F-bounded subclassing model of inheritance, in which "like current" is actually a parametric type σ of the kind: $\forall(\sigma <: \text{GenNumeric}[\sigma])$. Eiffel also has generic and *constrained* generic parameters:

class SortedList [T $\rightarrow$ Comparable] … **end**

which are *exactly* the same notion as F-bounds. Think of the constrained type parameter T as a parametric type: $\forall(\tau <: \text{GenComparable}[\tau])$. So, it makes most sense to think of Eiffel as belonging to the second-order family of languages, along with Smalltalk and Flavors.

This being the case, the reference definition of type-merge is adequate to capture Eiffel's model of inheritance. You simply have to imagine that *all* Eiffel class-types are in fact parametric types, which are only fixed when object instances are created. The model of inheritance unifies all the type parameters before combining the records. We illustrate this with a parametric version of the example from 2.1 above:

[1] A binary method is one which accepts an argument of the same type as self. It is binary in the sense that it deals with two objects of the same type.

$$\begin{aligned}
&GenVehicle = \lambda(\pi <: GenPerson[\pi]).\lambda(\theta <: GenLocation[\theta]).\\
&\qquad \lambda\sigma.\{owner : \to \pi,\ home : \to \theta\}
\end{aligned}$$

$$\begin{aligned}
&GenCar = \lambda(\pi <: GenPerson[\pi]).\lambda(\theta <: GenGarage[\theta]).\\
&\qquad \lambda(\rho <: GenLitres[\rho]).\lambda(\kappa <: GenKilometres[\kappa]).\\
&\qquad\qquad \lambda\sigma.(GenVehicle[\pi,\theta]\ \cup \{home : \to \theta,\ range : \rho \to \kappa\})
\end{aligned}$$

$$\begin{aligned}
&= \lambda(\pi <: GenPerson[\pi]).\lambda(\theta <: GenGarage[\theta]).\\
&\qquad \lambda(\rho <: GenLitres[\rho]).\lambda(\kappa <: GenKilometres[\kappa]).\\
&\qquad\qquad \lambda\sigma.\{owner : \to \pi,\ home : \to \theta,\ range : \rho \to \kappa\}
\end{aligned}$$

The subclass generator *GenCar* reintroduces all the parametric types used within the class, and substitutes these new parameters inside the body of the parent generator, through the application: GenVehicle[π,θ] before merging this adapted record with the record of extra methods. So, all common fields have the same types before record combination is computed, and the simple union of signatures is all that is required. The notion of type equality must allow for equality of simple types (such as Eiffel's *Integer* and *Real* types) and equality of type parameters for all class-types (see 4.2). Simons first proposed a unified parametric model of Eiffel's type system in 1995 [12], in which the self-type, anchored types, constrained generic types and ordinary class-types were all modelled using F-bounded parameters.

6 CONCLUSION

In this article, we have revisited the notion of typed inheritance. *The Theory of Classification* describes two models of inheritance, one a first-order model based on subtyping (Java, C++) and the other a second-order model based on subclassing (Smalltalk, Eiffel). Objects are modelled as records, or maps from labels to methods, so inheritance may be modelled as map union with override. Previously, the classical function override operator $\oplus$ was used without any constraints on the types of the records being combined. Here, we have introduced an F-bounded second-order definition of the inheritance function, called *extend*, with a constraint M on the type of extension that may legally be combined with any record.

We showed how, in the reference model, the constraint merely has to ensure that replacement fields have the same types as the fields they replace. This works for Java-style inheritance (first order) and also for Eiffel-style inheritance (second-order) in which field types may be parametric as well as simply-typed. Variations on this allow replacement fields to be subtypes (Trellis), or a mixture of type-equal and subtype fields (C++). One observation emerging from this is that the ability to replace fields with subtype fields is not a frequent requirement in object-oriented languages. The subclassing model of inheritance only requires type-equality, because all the field types are unified prior to combination, whether by parameter unification [3], or instantiation [5]. Simons and Bruce were the first to note the poor match between simple subtyping and natural models of inheritance [13, 14]. This is what originally motivated the *Theory of Classification*.

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_09/column4

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

[3] A J H Simons, "The theory of classification, part 8: Classification and inheritance", in *Journal of Object Technology*, vol. 2, no. 4, July-August 2003, pp. 55-64. http://www.jot.fm/issues/issue_2003_07/column4

[4] A J H Simons, "The theory of classification, part 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

[5] 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

[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] 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

[8] 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

[9] A Compagnoni and B Pierce, "Multiple inheritance via intersection types", *Technical Report ECS-LFCS-93-275, University of Edinburgh*, (Edinburgh: LFCS, 1993).

[10] 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

[11] A Compagnoni, "Subtyping in $F^{\omega}\wedge$ is decidable", *Technical Report ECS-LFCS-94-281, University of Edinburgh*, (Edinburgh: LFCS, 1994).

[12] A J H Simons, “Rationalising Eiffel’s type system”, *Proc. 18th Conf. Tech. Object-Oriented Lang. and Sys.*, eds. R Duke, C Mingins and B Meyer, (Melbourne : Prentice Hall, 1995), 365-377.

[13] A J H Simons, “A language with class: The theory of classification exemplified in an object-oriented language”, PhD Thesis, University of Sheffield (Sheffield, Department of Computer Science, 1995).

[14] K B Bruce, A Fiech and L Petersen, “Subtyping is not a good “match” for object-oriented languages”, *Proc. European Conf. Obj-Oriented Prog. 1997, pub. LNCS 1241*, (Jyväskylä: Springer Verlag, 1997) 104-127.

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.