# A Theory of Class

**Anthony J H Simons**

*Department of Computer Science, University of Sheffield,*
*Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.*
Email: a.simons@dcs.shef.ac.uk  Fax: +44 114 278 0972

***ABSTRACT**: We present a mathematical theory of class. The theory is general, in that it encompasses many different approaches to type abstraction, such as* type constructors, generic parameters, classes, inheritance *and* polymorphism. *The theory is elegant, in that it is based on a simple generalisation of F-bounds. The theory has timely implications for emerging OMG standards and future language designs.*

***KEYWORDS**: object-oriented, type theory, F-bounds, class, classification, generic parameters, polymorphism, inheritance*

## 1. Introduction

Programming languages as different as *Ada*, *Smalltalk*, *ML* and *Eiffel* offer various attractive, but partial views of type abstraction. The motivation for the current work arises from a dissatisfaction with the number of different mechanisms used to explain type abstraction and type compatibility, such as *type constructors*, *generic parameters*, *classes*, *inheritance*, and *polymorphism*. Accordingly, the focus here is on finding a unifying framework within which all the above mechanisms are related. Central to this is a properly-constructed mathematical notion of *class*.

In section 2, some background history is given on formal models of classification, commenting on the strengths of each approach. The state of the art is a mixture of different models of type polymorphism and subtyping. The focus of this article is to argue that one model, F-bounded quantification (Canning *et al.*, 1989a), is more general than each approach individually and, if adopted to describe all forms of polymorphism, yields a great simplification in the formal description of object-oriented languages. Section 3 provides a guide to the λ-calculus foundations for the theory, which is not just intended for theory experts. Section 4 describes the F-bounded model of class inheritance (Cook *et al.*, 1990). Section 5 presents a new generalisation of this theory that describes type constructors, generic parameters and classes with internal polymorphic components in the same framework. Section 6 concludes by relating the theory to the often-misunderstood notion of *class*. Suggestions for the simplification of object-oriented languages are given.

## 2. A Perspective on Formal Models of Classification

Many different approaches to characterising what is meant by a *class* have been proposed over the years. Influenced by the ability of Smalltalk's classes to provide arbitrary extensions to object implementations, many were led to believe that objects have *class* and *type* independently (Snyder, 1987; America, 1990), since the structure of object extensions did not conform well to theories of types and subtyping. There was much discussion of separating *specification* inheritance from *implementation* inheritance (Liskov, 1987; Sakkinen, 1989) and the notion of *class* was relegated to a unit of implementation, because of the concern to preserve a clear, type-related interpretation of inheritance.

Researchers modelling pure types and subtyping (Cardelli and Wegner, 1985; Cardelli, 1988; America, 1990) eventually determined that, though a *class* could be treated as a *type* (for example, in *Trellis* (Schaffert *et al*., 1986) and *Modula-3* (Cardelli *et al*., 1989), the overall effect was to restrict the usefulness of the static type system. An example illustrates how it is impossible to type check statically an intuitively sound method expression involving a mixture of local and inherited (supertype) methods:

```
class Number = {plus : Number → Number}                    recursion unrolled
class Integer = Number ∪ {minus : Integer → Integer}
i, j, k : Integer;
i.plus(j).minus(k);                                     minus applied to a Number
```

because of the phenomenon of *type-loss*, something which profoundly affects the type system of *C++* (Stroustrup, 1991) and leads to the unsafe programming practice of *type-downcasting* (Meyers, 1992) to recover lost type information. *Eiffel* (Meyer, 1992) and *Sather* (Omohundro, 1994) interpret the notion of *class* ambiguously as "sometimes" like a type, since redefined methods may sometimes have type signatures that contravene subtyping rules (Cook, 1989), preventing any simple identification of *class* with *type* and making the formal interpretation of *class* unclear.

The most fruitful foundation for object-oriented languages has come from mathematical theories of polymorphism. These, which date back to the original Girard-Reynolds formulation of universal quantification (Girard, 1972; Reynolds, 1974), invariably introduce type parameters, which range over families of types. Cardelli and Wegner (Cardelli and Wegner, 1985) generalised universal quantification with a subtype-bounded theory of polymorphism, also known as *bounded universal quantification*, or *inclusion polymorphism*, which described families of non-recursive types possessing a minimal interface. Later, Canning and others (Canning *et al*., 1989a) determined that subtype-bounded polymorphism did not function in the presence of type-recursion and was no more expressive than simple subtyping in this case. They proposed an alternative function-bounded theory of polymorphism, also known as *F-bounded quantification*, which correctly handles recursive types. In the latter two approaches, a *class* is considered a second-order generalisation over a family of types, more accurately reflecting the intuitions of object-oriented programmers about classification.

For some time, then, a collection of different mechanisms for handling type compatibility and type substitution have coexisted. Girard-Reynolds universal quantification is still widely used to characterise the *parametric polymorphism* of languages like *ML* (Milner, 1978) or *Ada* (Ichbiah *et al*., 1979) and also to describe the in-built *array of...* type constructor found in conventional languages. Cardelli and Wegner (Cardelli and Wegner, 1985) considered that *inclusion* and *parametric polymorphism* were distinct forms of universal polymorphism, further distinguished from *ad-hoc* varieties such as overloading or explicit type coercions. F-bounded polymorphism was devised chiefly to explain the compatibility of subclass and superclass interfaces (Canning *et* *al.*, 1989b), demonstrating how the inheritance relationship is not the same thing as subtyping (Cook *et al.*, 1990). This promoted the use of F-bounded parameters to describe the *self*-type of recursive objects, in order to show how this type changes when it is inherited. Other researchers have since developed formal languages in which the *self*-type is described in this way, but type compatibility between object fields is either handled by subtyping (Bruce *et al.*, 1993) or else the field-types match exactly (Eifrig *et al.*, 1994). In another article (Simons, 1995a), we have argued that object-oriented languages now have too many mechanisms to handle type polymorphism: for example, *Eiffel* has *conformance* (a kind of pseudo-subtyping), *constrained generic* and *anchored types* (both kinds of F-bounded polymorphism); whereas *C*++ has *subtyping* and *templates* (a kind of parametric polymorphism). It is unnecessary to have all these different mechanisms to obtain the kinds of behaviour we want in object-oriented languages.

## 3. A Guide to λ-Calculus Foundations

The theoretical model is motivated by considering that an object is a dynamic entity from which you obtain behaviour by selecting its methods. An object is therefore modelled as a record, a set of fields containing simple functions representing its methods (Canning *et al.*, 1989a, 1989b; Cook *et al.*, 1990). In the following, *p1, p2* and *p3* are records representing cartesian point objects. The fields of such a record are selected using the dot "." operator:

$p1 = \{x \mapsto 3, y \mapsto 4\}$
$p1.x \;\Rightarrow\; 3$

where *x* and *y* are labels which map "$\mapsto$" to values. In general, the values associated with the labels are simple functions; and these functions may refer to each other:

$p2 = p3 = \mu self.\{x \mapsto 2, y \mapsto 5, eq \mapsto \lambda p.(self.x = p.x \textbf{ and } self.y = p.y)\}$
$p2.eq(p3) \;\Rightarrow\; \textbf{true}.$

Note that the value selected by the label *eq* is a function λ*p*.(...), which is applied to another point object, in the style of a binary method expecting an argument. Now, the method *eq* refers to other methods *x* and *y* in the same object. Since the current object is not passed as an argument to the *eq* method, the variable *self* is introduced, standing implicitly for the whole object, such that *self.x* and *self.y* select fields in the same object. At the head of the record, μ*self* is a convention binding *self* to the whole record; this is explained below.

It is so demonstrated that even very simple objects like *p2* are recursive, because their methods invariably call each other. Recursion presents a problem from the theoretical viewpoint, since a recursive definition *g = f(g)* is not so much a definition as an equation which some *g* may satisfy. To motivate the existence of a *unique* solution requires a brief exposition of fixpoint theory. Recursion is handled in the λ-calculus by introducing a recursion variable, then binding this variable to the desired structure. The recursive point object above would be introduced by first defining a function of *self*:

$\phi p = \lambda self.\{x \mapsto 2, y \mapsto 5, eq \mapsto \lambda p.(self.x = p.x \textbf{ and } self.y = p.y)\}$

$\phi p$ is called an *object generator*, since it is a pattern for all points having fields x=2 and y=5, etc; however *self* is not yet bound to anything. We distinguish object *generator* names from objects by prefixing them with $\phi$. It would be possible to apply this function:

$\phi$p(p1) $\Rightarrow$ {x $\mapsto$ 2, y $\mapsto$ 5, eq $\mapsto$ $\lambda$p.(p1.x = p.x **and** p1.y = p.y)}

such that *self* is bound throughout to *p1*. However, what we want is to bind *self* to *p2*, giving the recursive paradox:

p2 = $\phi$p(p2)

Fixpoint theory says that it is possible gradually to approximate *p2* by applying the generator $\phi p$ an increasing number of times. If $\perp$ is the undefined value, then the recursive object *p2* is the limit of the infinite sequence:

p2 = $\phi$p($\phi$p($\phi$p( ... $\phi$p($\perp$) ...)))

There is a distinguished function **Y**, known as the *fixpoint finder*, which may be used to construct infinite self-applications of a generator. **Y** is not itself recursive, but:

p2 = **Y**($\phi$p)
= $\mu$self.{x $\mapsto$ 2, y $\mapsto$ 5, eq $\mapsto$ $\lambda$p.(self.x = p.x **and** self.y = p.y)}

will always construct a unique recursive object from an object generator. So, recursion is established from first principles. Formally, *p2* is known as the *least fixed point* of the generator $\phi p$. The rest of the theory makes widespread use of generators to define the recursive structure of objects and types, which are later fixed using **Y**. It is conventional to indicate a recursively-fixed variable with $\mu$.

The types of objects are also modelled as records, whose fields represent the types of methods. A record type is in general recursive:

Point = $\mu\sigma$.{x : Integer, y : Integer, eq : $\sigma \rightarrow$ Boolean}
Point.x $\Rightarrow$ Integer
Point.eq $\Rightarrow$ $\sigma \rightarrow$ Boolean

Here, *x*, *y* and *eq* are labels which map ":" to field types. These generally take the form of function signatures, since methods may accept further arguments. In the definition of *Point*, the method *eq* accepts another object that is the same type as *Point*. This is handled by introducing a recursion variable $\sigma$ standing for the *self*-type of points, which is bound to the whole record type using $\mu\sigma$. This in turn is merely a notational convention to indicate that the recursive type *Point* is the least fixed point of a *type generator* $\Phi$*Point*:

$\Phi$Point = $\Lambda\sigma$.{x : Integer, y : Integer, eq : $\sigma \rightarrow$ Boolean}
Point = **Y**[$\Phi$Point]

$\Phi$*Point* is a type function $\Lambda\sigma$.(...) expecting a single type argument, standing for the *self*-type, and which returns a record type as a result. When $\Phi$*Point* is fixed using **Y**, $\sigma$ is bound to the whole of the record type body, yielding the recursive type *Point*. We distinguish type *generator* names from types by prefixing them with Φ; also we use lower case names for objects and capitalise types.

## 4. The Theory of F-Bounds and Inheritance

Type generators play an important role in describing classes and inheritance. Whereas Cardelli and Wegner (Cardelli and Wegner, 1985) suggested that a *class* was the family of all those types $\tau$ that were subtypes of a given base type: $\forall(\tau \subseteq F)$, Canning and others (Canning *et al.*, 1989a) showed how this failed to adapt the *self*-type properly for recursive types. Instead, they expressed the constraint on the class in terms of the application of a type generator: $\forall(\tau \subseteq \Phi F[\tau])$. To see how this works, consider that we would intuitively like a labelled point *lp*:

$$lp = \mu self.\{x \mapsto 2,\ y \mapsto 5,\ eq \mapsto \lambda p.(self.x = p.x \ \mathbf{and}\ self.y = p.y),\ lab \mapsto \text{"vertex 1"}\}$$

to belong to the *class* of ordinary points, since it has all methods of *p2*, plus an extra method which returns the label string. The type of *lp* is the recursive type *LabPoint*:

$$LabPoint = \mu\sigma.\{x : Integer,\ y : Integer,\ eq : \sigma \rightarrow Boolean,\ lab : String\}$$

We can show that *lp* is not a member of the *type Point,* because the method *Point.eq* has the type *Point → Boolean*, whereas *LabPoint.eq* has the type *LabPoint → Boolean*. Furthermore, *LabPoint* is not a subtype of *Point*, because their respective *eq* functions are not in a subtyping relationship (Cook, 1989; Canning *et al.*, 1989a):

$$LabPoint \rightarrow Boolean \not\subseteq Point \rightarrow Boolean, \qquad \textit{because}\ Point \not\subseteq LabPoint$$

and this prevents subtype-bounded polymorphism $\forall(\tau \subseteq \textit{Point})$ from saying anything useful in this case. Instead, function-bounded polymorphism allows the class membership constraint to be expressed with respect to a *type generator* $\forall(\tau \subseteq \Phi\textit{Point}[\tau])$, rather than a type. We have to show that *LabPoint* is one of the types $\tau$ satisfying the bound:

$$\begin{aligned}&LabPoint \subseteq \Phi Point[LabPoint] \quad \Leftrightarrow\\ &\{x : Integer,\ y : Integer,\ eq : LabPoint \rightarrow Boolean,\ lab : String\}\\ &\qquad \subseteq \{x : Integer,\ y : Integer,\ eq : LabPoint \rightarrow Boolean\}\end{aligned}$$

which is now demonstrably true by Cardelli's record subtyping rule (Cardelli and Wegner, 1985; Cardelli, 1988). In particular, note how the *self*-type of *Point* is adapted by the application Φ*Point*[*LabPoint*]. This expresses formally exactly what object-oriented programmers mean when they say that a subclass object *matches the interface* of a superclass (OMG, 1991), even though there is no simple subtyping relationship between instances of the subclass and superclass.

Classification, or the ability to order classes in a hierarchy, is therefore defined with respect to generators. Rather than saying that one type is a subtype of another: $F \subseteq G$, we say that a pointwise subtyping relationship must hold between their generators: $\forall\tau.\Phi F[\tau] \subseteq \Phi G[\tau]$. This is sometimes also known as *second-order subtyping*. The intuitive notion that objects are members of an increasingly more general series of classes is captured in the rule:

$$\frac{\Gamma \vdash t \subseteq \Phi F[t], \quad \Gamma \vdash \forall s.\Phi F[s] \subseteq \Phi G[s]}{\Gamma \vdash t \subseteq \Phi G[t]} \quad \text{CLASSIFY}$$

which says that if, in the current context "$\Gamma$" you can derive "$\vdash$" that an object type t belongs to a class $t \subseteq \Phi F[t]$ and if the class generator $\Phi F$ extends the interface expressed by the class generator $\Phi G$, then the object type is also a member of the class $t \subseteq \Phi G[t]$. Rules such as this express exactly the kind of type compatibility required for object-oriented programming.

Inheritance, or the ability to derive subclasses from superclasses, is also expressed using generators. The existence of a record combination operator $\oplus$ is assumed, which performs field union with overriding (this, and other primitives, may be defined in $\lambda$-calculus terms). Intuitively, we should like to create a *LabPoint* object by adding to the fields of a *Point* object. However, the simple strategy:

lp = p2 ⊕ {lab ↦ "vertex 1"}
= {x ↦ 2, y ↦ 5, eq ↦ λp.(p2.x = p.x **and** p2.y = p.y), lab ↦ "vertex 1"}

does not work, because in the combined set of fields, *self* refers back to *p2,* instead we want *self* to refer to the resulting record, *lp*. By extending object *generators* instead of objects:

ϕlp = λself.(ϕp(self) ⊕ {lab ↦ "vertex 1"})
= λself.{x ↦ 2, y ↦ 5, eq ↦ λp.(self.x = p.x **and** self.y = p.y), lab ↦ "vertex 1"}

we may redirect *self* onto the intended object. The generator expression ϕ*lp* introduces a new *self* variable, which is passed to the old point generator ϕ*p*(*self*). This application returns an "adapted" record in which *self* now refers to the result. These "inherited" fields are combined with the additional fields using ⊕. Finally, *self* is bound recursively in *lp* using:

lp = **Y**(ϕlp)

Realistically, the object *lp* should be given a different *eq* method, which compares labels as well as coordinates. We could provide a completely new method to override the old one:

```
ϕlp = λself.(ϕp(self) ⊕ {eq ↦ λp.(self.x = p.x and self.y = p.y and self.lab = p.lab),
                         lab ↦ "vertex 1"})
```

however, it is more satisfying to extend the functionality of the "inherited" method. By introducing a new variable *super* = ϕ*p*(*self*), we maintain a handle on the adapted form of the parent object in inheritance expressions:

```
ϕlp = λself.(λsuper.(super ⊕ {eq ↦ λp.(super.eq(p) and self.lab = p.lab),
                              lab ↦ "vertex 1"})
             ϕp(self))
```

In the record of additional methods, *super.eq*(*p*) selects the inherited method from *super*, which, as we have established, is the adapted form of a *Point* object in which *self* has been redirected. The adapted *eq* method is provably equivalent to the earlier version where it was redefined in full.

This model captures exactly the notion of *method combination* in *Smalltalk*, or invoking base class functions in *C*++.

We have motivated inheritance from an untyped perspective. Object types evolve in parallel with the types of their individual fields; for example the $\Phi$*LabPoint* type generator:

$\Phi\text{LabPoint} = \Lambda\sigma.(\Phi\text{Point}[\sigma] \cup \{\text{eq} : \sigma \to \text{Boolean}, \text{lab} : \text{String}\})$
$= \Lambda\sigma.\{\text{x} : \text{Integer}, \text{y} : \text{Integer}, \text{eq} : \sigma \to \text{Boolean}, \text{lab} : \text{String}\}$

may be defined as an extension of the $\Phi$*Point* type generator. Now, by adding types to the object inheritance model, we may show that such derivations are also type consistent:

$\phi\text{lp} : \forall(\tau \subseteq \Phi\text{LabPoint}[\tau]).\tau \to \Phi\text{LabPoint}[\tau]$

```
$\phi\text{lp} = \Lambda(\tau \subseteq \Phi\text{LabPoint}[\tau]).\lambda(\text{self} : \tau).$
        $(\lambda(\text{super} : \Phi\text{Point}[\tau]).(\text{super} \oplus$
                $\{\text{eq} \mapsto \lambda(\text{p} : \tau).(\text{super.eq(p)}$ **and** $\text{self.lab} = \text{p.lab}), \text{lab} \mapsto \text{"vertex 1"}\})$
        $\phi\text{p}[\tau](\text{self}))$
```

*Typed* object generators have an additional type argument $\Lambda(\tau \subseteq \Phi\text{F}[\tau])$, expressing the fact that the object generators are polymorphic and may be adapted to objects of different types. The constraint on the type argument: $\tau \subseteq \Phi$*LabPoint*$[\tau]$, ensures that any object derived from $\phi$*lp* has at least the interface of a *LabPoint*. Both *self* and the argument *p* in the *eq* method have this type $\tau$, ensuring that *eq* compares objects of the same type. The type of *super* is $\Phi$*Point*$[\tau]$, a *Point* type adapted by substituting the new *self*-type $\tau$. This is type-correct, since the CLASSIFY rule allows us to infer $\tau \subseteq \Phi$*Point*$[\tau]$ from $\tau \subseteq \Phi$*LabPoint*$[\tau]$ and the pointwise condition $\forall$s.$\Phi$*LabPoint*[s] $\subseteq \Phi$*Point*[s]. Assuming that the superclass generator $\phi$*p* is now also typed, $\phi$*p*$[\tau]$(*self*) adapts the inherited super-record of methods by supplying the *self*-type and *self*-reference of the resulting object. This expresses exactly the notion of redirecting *self*-reference in *Smalltalk* and the automatic adaptation of anchored types: *like Current* in *Eiffel* (Meyer, 1992). The record combination operator $\oplus$ then combines two sets of fields in which *self* refers to the whole derived object, whose type is $\tau$.

## 5. The Theory of F-bounds and Generic Polymorphism

The chief achievement of Cook's work (Cook *et al*., 1990) was in recognising that to model inheritance, *self* had to be rebound to refer to the correct structure. For this reason, generators were used in which *self* and its type were flexible. F-bounds were used to constrain the *self*-types that belong to a class. As a result of this focus, subsequent work has only used F-bounds to characterise the *self*-type as it evolves under inheritance (Bruce *et al*., 1993; Eifrig *et al*., 1994). We have further generalised the F-bounded model to incorporate other kinds of polymorphism, such as standard *parametric polymorphism* as found in *ML* (Milner, 1978), *C*++ templates (Stroustrup, 1991) and *Ada*'s generic packages (Ichbiah *et al*., 1979); and the *constrained generic parameters* as found in *Eiffel* (Meyer, 1992). The claim is that this single theory (which eventually is *higher-order*) captures *every* kind of type abstraction present in these languages.

To motivate the more general theory, consider that a simple recursive integer list may be constructed as an object having the recursive record type:

$\text{IntList} = \mu\sigma.\{\text{add} : \text{Integer} \rightarrow \sigma, \text{head} : \text{Integer}, \text{tail} : \sigma\}$

A list object *il : IntList* has the structure of a single cell, whose *head* returns an *Integer* and whose *tail* returns an *IntList*. To turn this into a polymorphic list type, we must parameterise over the element-type. The Girard-Reynolds [Gira72, Reyn74] style of declaration has:

$\text{List} = \Lambda\tau.\mu\sigma.\{\text{add} : \tau \rightarrow \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

Note that $\tau$ is introduced before $\sigma$, such that $\sigma$ is declared within the scope of $\tau$. We say that $\sigma$ is *closed over* $\tau$. The order of introduction determines that *List* is a *homogeneous* list type, in the style of *ML* lists, whose elements are all of the same type. When the recursion is bound using $\mu$, the type of the list's tail is fixed as the same as the *self*-type, ensuring that subsequent list elements have the same type as the *head*. *List* is actually a *type function*:

$\text{List} : \forall\tau.\tau \rightarrow \text{List}[\tau]$

rather than a *simple type*, because it expects any given type $\tau$ as an argument and then returns a recursive list type *List*[$\tau$] constructed from this element type as a result. Most programming languages offer built-in *type constructors* that work like this, such as the *Array of ...*, *Set of ...*, constructions to be found in *Pascal* and other languages.

The universal quantification $\forall\tau$ of Girard-Reynolds allows no restrictions on the type of list element. However, a much richer family of list types may be constructed if we permit the introduction of restrictions on the element type. Consider that a sorted list has the type:

$\text{SortList} = \Lambda(\tau \subseteq \Phi\text{Order}[\tau]).\mu\sigma.\{\text{add} : \tau \rightarrow \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

where $\Phi$*Order* is a type generator expressing the interface of all those types possessing the total ordering relationships <, >, <= and >=.

$\Phi\text{Order} = \Lambda\sigma.\{< : \sigma \rightarrow \text{Boolean}, > : \sigma \rightarrow \text{Boolean}, <= : \sigma \rightarrow \text{Boolean}, >= : \sigma \rightarrow \text{Boolean}\}$

Now, the *add* method for sorted lists may be written with the foreknowledge that every element will be comparable with its fellows. This generalisation to include F-bounded restrictions captures a notion similar to *constrained generic parameters* in Eiffel (Meyer, 1992). *SortList* is a type function expecting an element type $\tau \subseteq \Phi$*Order*[$\tau$] and returning a recursive sorted list type. The element-type must be supplied for the list to be useable:

```
isl : SortList[Integer];
isl.add(3) ...
```

capturing the notion of static type parameter replacement in generic schemes.

It should be clear that ordinary Girard-Reynolds parametric polymorphism, which characterises *C++ templates* and Eiffel's *non*-constrained generic parameters, is included in this model, since a universal bound is a special case of an F-bound:

$\text{List} = \Lambda\tau.\mu\sigma.\{\text{add} : \tau \to \sigma, \text{head} : \tau, \text{tail} : \sigma\} \Leftrightarrow$
$\text{List} = \Lambda(\tau \subseteq \Phi\text{Top}[\tau]).\mu\sigma.\{\text{add} : \tau \to \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

where $\Phi\text{Top} = \Lambda\sigma.\{\}$ is a generator for the type with no restrictions on its interface. It should also be clear that an F-bound, rather than a Cardelli-Wegner subtype bound, is necessary to characterise the element type of *SortList*. To see this, consider that the alternative:

$\text{SortList2} = \Lambda(\tau \subseteq \text{Order}).\mu\sigma.\{\text{add} : \tau \to \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

would make it impossible to find a useful type to substitute for $\tau$, since the only true subtypes ($\tau \subseteq Order$) must have comparison methods with the type ($\tau \supseteq Order$) $\to$ *Boolean* to satisfy the record subtyping rule (Cardelli and Wegner, 1985) this effectively makes *Order* the only substitutable type.

To integrate generic polymorphism with the model of classification, it is necessary to show how the type scheme is well-behaved under inheritance. A polymorphic list *class* is obtained by generalising over the polymorphic list *type*, yielding a type function which may then be used to describe an F-bound on the family of list types belonging to the class:

$\Phi\text{List} = \Lambda(\tau \subseteq \Phi\text{Top}[\tau]).\Lambda\sigma.\{\text{add} : \tau \to \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

This function expects two type arguments, one for the element-type and one for the recursive type of the list. The objects which belong to this class may have at least the interface of a *List* having elements with at least the interface of *Top*. We express this type constraint as:

$\forall(\tau \subseteq \Phi\text{Top}[\tau]).\forall(\sigma \subseteq \Phi\text{List}[\tau, \sigma]).\sigma$

Note how the bound on the *self*-type $\sigma$ is expressed in terms of $\Phi List[\tau, \sigma]$, since the type-function for lists requires two type arguments. This construct captures exactly the notion of *generic classes* in *Eiffel* (as stated above, *Eiffel* is ambiguous in its treatment of *class* - our model distinguishes and properly relates the notions of generic *type* and generic *class*).

To see how a generic class may be adapted in all the ways desired in object-oriented programming, consider how applying the type function to an element-type yields the standard form of a generator seen above:

$\Phi\text{IntList} = \Phi\text{List}[\text{Integer}] = \Lambda\sigma.\{\text{add} : \text{Integer} \to \sigma, \text{head} : \text{Integer}, \text{tail} : \sigma\}$

This captures the notion of inheritance with parameter replacement. The derived class, whose objects have one of the types in the family $\forall(\sigma \subseteq \Phi IntList[\sigma]).\sigma$, is restricted to those lists having *Integer* elements, but it is still extensible - for example, a further class may be derived having an extra method to *sum* the elements, in the usual manner of inheritance:

$\Phi\text{SumIntList} = \Lambda\sigma.(\Phi\text{IntList}[\sigma] \cup \{\text{sum} : \text{Integer}\})$

Instead of replacing the generic parameter with a simple type, it is equally possible to restrict the bound on the parameter and so derive a generator for the *class* of sorted lists:

$\Phi$SortList = $\Lambda(\tau \subseteq \Phi\text{Order}[\tau]).\Lambda\sigma.(\Phi\text{List}[\tau, \sigma] \cup \{\text{add} : \tau \rightarrow \sigma\})$
$= \Lambda(\tau \subseteq \Phi\text{Order}[\tau]).\Lambda\sigma.\{\text{add} : \tau \rightarrow \sigma, \text{head} : \tau, \text{tail} : \sigma\}$

This is achieved by introducing the new parameter $\tau \subseteq \Phi$*Order*$[\tau]$ and applying the $\Phi$*List* generator to this and the new *self*-type. This application is type-correct, since the CLASSIFY inheritance rule allows us to derive $\tau \subseteq \Phi$*Top*$[\tau]$ from $\tau \subseteq \Phi$*Order*$[\tau]$ and the pointwise condition $\forall$s.$\Phi$*Order*[s] $\subseteq \Phi$*Top*[s]. Objects in the class of sorted lists have a type satisfying:

$\forall(\tau \subseteq \Phi\text{Order}[\tau]).\forall(\sigma \subseteq \Phi\text{SortList}[\tau, \sigma]).\sigma$

and we may prove that these objects are also members of the ordinary list class. Consider that we may replace $\tau$ by some type *Integer* $\subseteq \Phi$*Order*[*Integer*]. The generator $\Phi$*IntSortList* = $\Phi$*SortList*[*Integer*] may be used as the bound for the extensible "sorted list of integer" type family. The CLASSIFY rule allows us to infer that all objects of this class are also members of the extensible "list of integer" family, bounded by $\Phi$*IntList* = $\Phi$*List*[*Integer*]. This in turn is a smaller family than the entire list family, shown above. Now, by generalising from *Integer* to all types $\tau \subseteq \Phi$*Order*$[\tau]$, the assertion is proved on a pointwise basis.

Although this example did not extend the interface of $\Phi$*List* (the *add* method was redefined to enforce ordered element insertion), there is no reason why we should not extend the interface at the same time. Consider a merge-sorted list derived from a general list:

$\Phi$MrgSortList = $\Lambda(\tau \subseteq \Phi\text{Order}[\tau]).\Lambda\sigma.(\Phi\text{List}[\tau, \sigma] \cup \{\text{add} : \tau \rightarrow \sigma, \text{merge} : \sigma \rightarrow \sigma\})$
$= \Lambda(\tau \subseteq \Phi\text{Order}[\tau]).\Lambda\sigma.\{\text{add} : \tau \rightarrow \sigma, \text{head} : \tau, \text{tail} : \sigma, \text{merge} : \sigma \rightarrow \sigma\}$

Here, $\sigma$ is rebound so that *add* and *tail* also return merge-sorted lists. It can be proven that object types belonging to the class: $\forall(\tau \subseteq \Phi$*Order*$[\tau]).\forall(\sigma \subseteq \Phi$*MrgSortList*$[\tau, \sigma]).\sigma$ are also members of the sorted list class, using a pointwise argument like that given above; and therefore also members of the ordinary list class.

## 6. Implications of the Theory of Class

A theory has been presented, which unites the treatment of inheritance based polymorphism with generic parameter based polymorphism. In particular, a type model based on F-bounds has been shown powerful enough to capture all existing forms of parametric polymorphism. Classical *ML* parametric polymorphism and *C*++ templates were shown to be a special case of F-bounded polymorphism, applied to types. It was demonstrated that the theory held when parameterised types were generalised to generic classes containing internal polymorphic components. In particular, types were well-behaved under inheritance and the formal model captured all the intuitions present in *Eiffel's* generic classes. The theory shows how the notion of classification extends properly to parameterised structures: there are families of parameterised types which enclose each other and which are describable within the same framework as ordinary classes. In particular, ordinary classes were shown to be equivalent to generic classes in which the generic parameters were replaced by types. Generic classes could have their parameters restricted and their interfaces extended.

The theory makes a strong claim that *class* is a well-founded mathematical notion. Unlike earlier work (Snyder, 1987; America, 1990) which relegated *class* to an implementation construct, it has been shown here that, just as a simple *type* is inhabited by a monomorphic family of objects having identical method type signatures, a *class* is inhabited by a polymorphic family of objects, belonging to different types satisfying the class F-bound. In particular, there is no need to divorce type from implementation, since the (mono- or polymorphic) type of an object follows from the structure of its method interface. The current OMG standard (OMG, 1991) skirts round problematic definitions of *class* by talking in terms of *types* (= simple types) and *interfaces* (= generators for F-bounds). In this theory, a recursive type is the least fixed point of the generator for a class interface. The two concepts are therefore properly linked.

The theory has implications for future language design - currently, languages like *Eiffel* and *C++* have too many mechanisms for handling polymorphism (Simons, 1995a). In particular, subtyping is too weak, conformance is incorrect (Cook, 1989), constrained generic parameters and anchored types describe the same kind of polymorphism as class inheritance (Simons, 1995a), which is best described using F-bounds. F-bounds also provide a formal explanation for *where-clauses* (Day, *et al.*, 1995), *matching* (Abadi and Cardelli, 1995; Bruce, 1994; Bruce *et al.*, 1993) *open types* (Eifrig *et al.*, 1994) and similar expressions of constraint upon interfaces.

Future work in this field must provide a suitably secure typing for $\oplus$, the record combination operator. Cook's operator (Cook *et al.*, 1990) was simply-typed. We have generalised this to a second-order and higher-order operator (Simons, 1995b). A second-order theory is necessary to extend classes whose components are parameterised over simple types. A higher-order theory is necessary to extend classes whose components are parameterised over type functions (ie over classes and type constructors) and also to integrate properly with *heterogeneous* polymorphism within the same model (Simons, 1995b).

## References

Abadi, M. and Cardelli, L., (1995), 'On subtyping and matching', *Proc. 9th European Conf. Object-Oriented Prog.*, Aarhus, Denmark, pub. *Lecture Notes in Computer Science, 952*, ed. W. Olthoff, Springer Verlag, 145-167.

America, P., (1990), 'Designing an object-oriented language with behavioural subtyping', *Proc. Conf. Foundations of Object-Oriented Lang.*, 60-90.

Bruce, K., Crabtree, J., Murtagh, T. and van Gent, R., (1993), 'Safe and decidable type-checking in an object-oriented language', *Proc. 8th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 28 (10),* 29-46.

Bruce, K., (1994), 'A paradigmatic object-oriented programming language: design, static typing and semantics', *J. of Func. Prog., 4(2),* 127-206.

Canning, P., Cook, W., Hill, W., Olthoff, W. and Mitchell, J., (1989a), 'F-bounded polymorphism for object-oriented programming', *Proc. 4th Int. Conf. Func. Prog. Lang. and Arch.*, Imperial College London, September, 273-280.

Canning, P., Cook, W., Hill, W. and Olthoff, W., (1989b), 'Interfaces for strongly-typed object-oriented programming', *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang and Appl.,* pub. *ACM Sigplan Notices, 24 (10),* 457-467.

Cardelli, L. and Wegner, P., (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys, 17(4),* 471-521.

Cardelli, L., (1988), 'Structural subtyping and the notion of power type', *Proc. 15th ACM Symp. Principles of Prog. Langs.,* 70-79.

Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B. and Nelson, G., (1989), 'Modula-3 report (revised)', *Tech. Rep. 52*, Digital Equipment Corporation Systems Research Centre.

Cook, W., (1989), 'A proposal for making Eiffel type-safe', *Proc. 3rd European Conf. Object-Oriented Prog.,* 57-70.

Cook, W., Hill, W. and Canning, P., (1990), 'Inheritance is not subtyping', *Proc. 17th ACM Symp. Principles of Prog. Lang.,* 125-135.

Day, M., Gruber, R., Liskov, B. and Meyers, A. C., (1995), 'Subtypes vs. where clauses: constraining parametric polymorphism', *Proc. 10th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices 30(10),* 156-168.

Eifrig, J., Smith, S., Trivonov, V. and Zwarico, A., (1994), 'Application of object-oriented type theory: state, decidability and integration', *Proc. 9th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 29 (10),* 16-30.

Girard, J.-Y., (1972), 'Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur', PhD Thesis, Université Paris VII.

Ichbiah, J. *et al*. (1979), 'Rationale and design of the programming language Ada', *ACM Sigplan Notices, 14(8).*

Liskov, B., (1987), 'Data abstraction and hierarchy', *Addendum to Proc. 2nd ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* 17-34.

Meyer, B., (1992), *Eiffel: the Language,* Prentice Hall.

Meyers, S., (1992), *Effective C++: 50 Ways to Improve your Programs and Designs,* Addison-Wesley.

Milner, R., (1978), 'A theory of type polymorphism in programming', *J. Computer and System Sciences, 17,* 348-375.

The Object Management Group (1991), *The Common Object Request Broker: Architecture and Specification, Revision 1.1,* OMG.

Omohundro, S., (1994), *The Sather 1.0 Specification*, ICSI Berkley.

Reynolds, J. C., (1974), 'Towards a theory of type structure', *Proc. Coll. sur la Programmation, pub. LNCS 19,* Springer Verlag, 408-425.

Sakkinen, M., (1989), 'Disciplined inheritance', *Proc. 3rd European Conf. Object-Oriented Prog.,* 3-24.

Shaffert, C., Cooper, T., Bullis, B., Killian, M. and Wilpolt, C., (1986), 'An introduction to Trellis/Owl', Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl, 9-16.

Simons, A. J. H., (1995a), 'Rationalising Eiffel's type system', *Proc. 18th Conf. Techn. Object-Oriented Lang. and Sys.*, eds. R Duke, C Mingins and B Meyer, Prentice-Hall, 365-377.

Simons, A. J. H., (1995b), *A Language with Class*, PhD Thesis, University of Sheffield.

Snyder, A., (1987), 'Inheritance and the development of encapsulated software components' in: Research Directions in Object-Oriented Programming, eds. B Shriver and P Wegner, MIT Press, 165-188.

Stroustrup, B., (1991), *The C++ Programming Language, 2nd Edn.,* Addison-Wesley.