JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 4, No. 5, July - August 2005

The Theory of Classification Part 19: The Proliferation of Parameters

Anthony J H Simons, Department of Computer Science, University of Sheffield, UK

1 INTRODUCTION

The *Theory of Classification* is an informal series of articles considering the formal notions of type and class in object-oriented languages. The series began by constructing models of objects, types, and inheritance, then branched out into interesting areas such as mixins, multiple inheritance and generic classes. Most recently, we returned to the core argument that simple types and subtyping are formally different concepts from polymorphic classes and subclassing. The previous article [1] described how object-oriented languages were unclear about the distinction between simple and polymorphic types, so we described one possible approach in which class names (which are used like type identifiers) were interpreted unambiguously either as simple types, or polymorphic classes, according to context.

In the current article, we consider in more detail the kinds of manipulations performed upon polymorphic class-types. These are expressed using function-bounded type parameters of the form: $\tau <: F[\tau]$, where F is a type function, describing the shape of the interface that the type τ is expected to satisfy [2]. In the following, we motivate the need for parameters and bounded parameters, then examine what happens when we require a language to express all of its polymorphism in this way. The result is a proliferation of parameters and constraints, which has both good and bad consequences. On the positive side, it is clear over what types the polymorphic variables may range. On the negative side, the syntax of such languages becomes inflated with parameters and is therefore somewhat unwieldy.

Cite this column as follows:Anthony J H Simons: “The Theory of Classification Part 19: The Proliferation of Parameters”, in *Journal of Object Technology*, vol. 4, no. 5, July-August 2005, pp.37-48 http://www.jot.fm/issues/issue_2005_07/column4

2 POLYMORPHISM NEEDS PARAMETERS

Throughout this series, we have made it clear that wherever polymorphism is intended, this formally requires the use of a type parameter, to stand for the group of types over which the function is defined [3]. For example, consider how the *identity* function applies to a value of any type and returns an identical value, and the result is of the same type as the argument. To define such a function in C++, we must use the template mechanism:

```
template <class T>
T identity (const T arg)       // argument is not modified
{  return arg;                 // returns a copy of arg
}
```

in which the line "`template <class T>`" introduces the type parameter `T`, which stands for any type. This function can be applied to values of many different types in C++, including primitive types, pointers and structured object types:

```
int x = identity(5);                // returns an int
double y = identity(3.14);          // returns a double
Point* p = identity(new Point);     // returns a Point* pointer
Point q = identity(Point(2, 3));    // returns a Point value
```

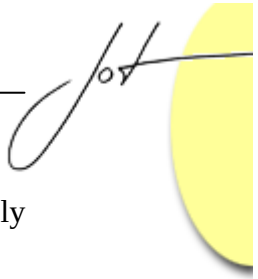
The polymorphic function appears to be "smart" because it somehow detects the type of its argument and returns a value of the exact same type. In C++ this is accomplished by the compiler, which statically detects the argument's type and creates a specific instantiation of the template function at compile time. In fact, for every distinct instantiation of the *identity* function, the compiler must generate a new copy of the *identity* function. In the end, it is as if the above program contained four different *identity* functions, with the overloaded type signatures:

```
int identity(int arg);
double identity(double arg);
Point* identity(Point* arg);
Point identity(Point arg);
```

and then C++'s normal rules for selecting one or other overloaded function were used to determine which function to apply.

In the λ-calculus, which is a very simple symbol-manipulation system, we bind the type parameter explicitly, supplying the intended type. The calculus cannot detect by itself that a value has any particular type unless we tell it so. All polymorphic functions therefore accept type arguments and value arguments, in that order [3]. The identity function is defined:

identity : $\forall\tau \,.\, \tau \rightarrow \tau$	// for all τ, accept a τ and return a τ
identity = $\lambda\tau \,.\, \lambda(\text{arg} : \tau) \,.\, \text{arg}$	// bind τ, then bind *arg* : τ, then return *arg*



and we apply it to values of different types in the following style, in which we supply first the type argument, then a value argument of that type:

identity int 5 $\Rightarrow$ 5
identity double 3.14 $\Rightarrow$ 3.14

Readers will recall that a λ-function simply binds its arguments in a left-to-right fashion and so doesn't need to put the arguments in parentheses. In earlier articles, we have sometimes adopted the convention of wrapping type arguments in brackets [] and value arguments in parentheses () to help the reader visualise what is going on inside the function.

Second-order functions like the above are really two functions, one nested inside the other. The outer function is a type-function and the inner function is a value-function. In the above examples, we supplied both the type argument and then the value argument. If we only supply the first type argument, then the result we get back is the second function, which is the body of the first function, in which the type parameter τ has been replaced:

identity int $\Rightarrow$ λ(arg : int) . arg // replace τ by int: {int/τ}
identity double $\Rightarrow$ λ(arg : double) . arg // replace τ by double: {double/τ}

and this models the effect of C++'s type instantiation, since it returns a specific version of the identity function, ready to be applied to a value of one particular type.

3 CLASSES NEED BOUNDED PARAMETERS

Throughout this series, we have argued that class-types are also polymorphic things and therefore need to be modelled using type parameters [3, 4, 1]. The main difference between a universal polymorphic function, such as *identity*, and the kinds of function that belong to a restricted class, such as *plus*, *minus*, *times* and *divide*, is that we want these functions to apply not just to any old type, but only to those types which are considered to be "at least some kind of number". In earlier articles, we demonstrated that this required the introduction of *constraints*, or *bounds* on the type parameter. We defined the interface of a number class using a type function:

GenNumber = λσ.{plus : σ → σ, minus : σ → σ, times : σ → σ, divide : σ → σ}

and then used this to constrain a type parameter, in the function-bounded style [2, 4]:

∀(τ <: GenNumber[τ]) . (… some definition involving τ …)

How does this constraint express what we mean by a class? The intention is that τ may only range over certain types in a type family [3]. What the constraint literally says is "all types τ that are subtypes of the type you get when you apply *GenNumber* to the τ type".

To unpack this further, recall that we started with a simple model of objects as records, whose labelled fields are functions, representing the object's methods. The types

of objects are therefore represented by record types, whose fields are the corresponding type signatures of the object's methods. The notion of subtyping we need, in order to understand this constraint, is therefore record subtyping [5]. The record subtyping rule permits a type with more fields to be a subtype of a type with fewer fields.

So, returning to the original problem, imagine that we expect the *Integer* type to belong to the class of numbers. Therefore, *Integer* must be one of the types that satisfies the above constraint, in other words, we expect the following to be true:

Integer <: GenNumber[Integer], which we can re-express as:

Integer <: {plus : Integer → Integer, minus : Integer → Integer,
times : Integer → Integer, divide : Integer → Integer}

by applying *GenNumber[Integer]* and seeing what kind of record-type we get when we substitute {*Integer* / σ} in the body of the generator. So, what this constraint is really saying is that the *Integer* type must have at least as many methods as the record type on the right-hand side. This is easy enough to satisfy if we give *Integer* all of those methods; and we could possibly give it more methods, such as: *modulo : Integer → Integer*, which returns a remainder.

So, this kind of bounded type parameter captures exactly the sort of constraint we need when defining groups of functions that apply to all the types in a given class and only to those types. We may write the polymorphic type of *plus* and *minus* in the style of methods:

$\forall(\tau <: \text{GenNumber}[\tau])\ .\ \tau.\text{plus} : \tau \rightarrow \tau$
$\forall(\tau <: \text{GenNumber}[\tau])\ .\ \tau.\text{minus} : \tau \rightarrow \tau$

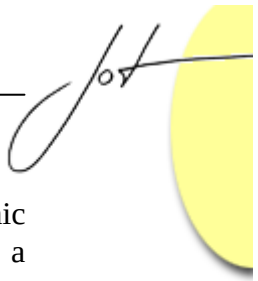
This says that, for all numeric types τ, selecting the *plus* or *minus* methods from an object of type τ will return a function that accepts the remaining argument of the same type τ and this will also return a result of the type τ. Alternatively, we can write the polymorphic type of *plus* and *minus* in the style of regular functions:

$\text{plus} : \forall(\tau <: \text{GenNumber}[\tau])\ .\ \tau \rightarrow (\tau \rightarrow \tau)$
$\text{minus} : \forall(\tau <: \text{GenNumber}[\tau])\ .\ \tau \rightarrow (\tau \rightarrow \tau)$

which are now clearly seen to accept two arguments, the first of which is the receiver object, from which the method is to be selected. The remaining argument and result type are as above.

4 BOUNDED PARAMETERS IN FUNCTIONAL LANGUAGES

Our notion of a class is exactly the same as the notion of *type classes* in the strongly-typed functional programming languages. The language Haskell [6] defines polymorphic functions using type parameters and sometimes needs to assert that these parameters



range over certain restricted classes of types. In the following example of a polymorphic function to compute the `length` of a list, "`[]`" denotes a list-type and "`a`" denotes a type parameter:

```
length :: [a] -> Int
length nil = 0                              -- empty case
length (head : tail) = 1 + length tail      -- non-empty case
```

The first line is a type signature, saying that `length` takes a list of any polymorphic element type "`[a]`" and returns a result of the `Int` type. This is a universal polymorphic function, rather like *identity* earlier. The `length` can be computed, irrespective of what type of element we choose for the list.

However, the polymorphic function for list membership cannot be defined in quite the same way. To determine list membership, the body of the `elem` function needs to be able to compare the supplied value with successive elements of the supplied list, using the equal function "`==`". So, the polymorphic element type must be one that is *in the class of types possessing an equal function.* In Haskell, this is represented by the constraint "`Eq a`", which has the sense "any polymorphic type `a` in the `Eq` class". Elsewhere, Haskell defines the `Eq` class as the class of all those types possessing "`==`" and "`/=`" (not equal). The definition of `elem` is given by:

```
elem :: Eq a => a -> [a] -> Bool
elem x nil = False              -- empty case
elem x (head : tail)                -- non-empty case
    | x == head = True          -- found the element
    | otherwise = elem x tail   -- keep searching
```

in which the constraint "`Eq a =>`" on the first line has the sense: "provided that the type `a` is a member of the `Eq` class, then this function takes an element of the type `a` and a list of the type `[a]` and returns a `Bool` result". The effect of this class constraint is to ensure that `elem` can only apply to lists, whose elements have a well-defined equal function. If this constraint were not present, then the Haskell compiler would refuse to compile the definition of `elem`, because it could not guarantee that "`x == head`" was a well-typed expression. This, by the way, contrasts with the approach taken in C++, which allows you to write arbitrary expressions involving variables with parametric types, because C++ doesn't compile or check any of its template definitions. It simply waits for these to be instantiated, and then checks that all the calls are well-typed for each instantiation, separately.

In the λ-calculus, we can express such a class of types with equality, using a type generator to constrain the polymorphic type parameter to accept only types that have at least the two functions *equal* and *notEqual*:

$$\text{GenEqual} = \lambda\sigma.\{\text{equal} : \sigma \rightarrow \text{Boolean}, \text{notEqual} : \sigma \rightarrow \text{Boolean}\}$$

$$\forall(\tau <: \text{GenEqual}[\tau]) \,.\, (\ \ldots \text{ some definition involving } \tau \ \ldots\ )$$

The above F-bound provides the exact meaning of Haskell's "`Eq a =>` …" constraint. It was quite satisfying to find this convergence between the notion of class put forward in the *Theory of Classification*, and the notion of "type classes" in functional programming languages, because it means that we are all probably on the right track!

5 CLASSES WITHIN CLASSES NEED NESTED PARAMETERS

After a while, the realisation comes that everywhere you want polymorphism, you formally need another type parameter. This has interesting consequences when you consider more complex classes that are built up out of a number of other classes as their "elements". Simple examples include the kind of generic classes we considered in an earlier article [7]. For example, if we have a *List* class whose self-type is expressed as the parameter σ, and if this *List* has a method *insert* accepting elements of some other class, whose polymorphic type is expressed as the parameter τ, then in which order should these be declared?

The solution to this quandary is found by thinking about the notion of type dependency. The element-type may exist by itself, but the list-type depends in some way upon the element-type in its definition. That is, for whatever element type τ we imagine, the list self-type σ somehow depends on the type of τ. This is natural, since we would expect an *IntegerList* to depend on its *Integer* elements, whereas a *RealList* would have *Reals* as its elements. We therefore introduce τ before σ, since this keeps within the *second-order* λ-calculus, in which type parameters only range over simple types [3]. That is, we introduce σ in a context in which τ is already bound to some actual type, so σ can range over simple types also.

To see how the type parameters stack up, let us construct the type signature for a generic *List* class with a method *elem* to test whether the inserted elements are members of the list. From section 4 above, we know that this puts a constraint on the element type, which must have an *equal* method to compare itself with other elements. Therefore, we will first require a type generator to express the shape of the element type:

$$\text{GenEqual} = \lambda\tau.\{\text{equal} : \tau \rightarrow \text{Boolean}, \text{notEqual} : \tau \rightarrow \text{Boolean}\}$$

The shape of the list's interface is expressed through a second type generator:

$$\text{GenList} = \lambda\tau.\lambda\sigma.\{\text{insert} : \tau \to \sigma,\ \text{head} : \tau,\ \text{tail} : \sigma,\ \text{length} : \text{Integer},\ \text{elem} : \tau \to \text{Boolean}\}$$

which now recognises that there are two parameters involved. The first is τ for the element-type. We can see this by applying *GenList* to some actual element type, say *Integer*, to see what kind of type-expression this produces:

$$\text{GenList[Integer]} = \lambda\sigma.\{\text{insert} : \text{Integer} \to \sigma,\ \text{head} : \text{Integer},\ \text{tail} : \sigma,\ \text{length} : \text{Integer},\ \text{elem} : \text{Integer} \to \text{Boolean}\}$$

The result substitutes $\{Integer/\tau\}$ in the body of the generator, returning a nested type function beginning: $\lambda\sigma.\{\ldots\}$. This looks exactly like the shape of a regular type generator for a non-generic class, and so it is. The second type parameter σ stands for the self-type of this class. We need this because we are dealing with polymorphic classes, not simple types. That is, the above definition is the minimum common interface for a variety of different list-types, which might include more specialised list-types, such as sorted lists.

What is the form of the F-bound constraint that ensures that our list self-type σ ranges over only those lists which (i) have elements with an *equal*-method and (ii) have a list-interface that includes all of the methods: *insert, head, tail, length* and *elem*? This is a constraint constructed using both of the above two type generators:

$$\forall(\tau <: \text{GenEqual}[\tau]).\forall(\sigma <: \text{GenList}[\tau, \sigma]).\ (\ \ldots \text{ some definition involving } \tau \text{ and } \sigma \ \ldots\ )$$

What is interesting here is the double quantification. On the outside, we assert that the element type τ may only range over subtypes of *GenEqual*[t], that is, types with at least the methods: *equal* and *notEqual*. Then, on the inside, we assert that the self-type σ may only range over subtypes of *GenList*[τ, σ], that is, types with at least the methods: *insert, head, tail, length* and *elem*, provided that τ is of the earlier specified type. This is the way in which the constraint for the list-type depends on the element-type. Translating this into object-oriented programming terms, the polymorphic type of a class depends on the polymorphic types of the other classes which it references internally. More precisely, it depends on those classes which affect the shape of its external interface.

6 SUBCLASSING USES PARAMETER SUBSTITUTION

Assume now that we wish to derive a more specialised kind of *List* class, say a *SortedList* of elements which are inserted automatically in ascending order. *SortedList* has two new operations *least* and *greatest*, to return the smallest and largest elements and otherwise has all the operations of a *List*. Below, we define the polymorphic type of the *SortedList* class by inheritance, constructing the new constraint partly from information present in the old one. Afterwards, we show how the new constraint preserves the old constraint.

To permit the sorting of elements when they are *inserted*, elements must be comparable with each other using *lessThan*. This means that the constraint on the element-type will be stronger than the one required for a plain *List* element. The new type generator must at least have the interface of *Equal*, otherwise the old *List* membership method *elem* would not work. The new generator *GenComparable* is therefore defined to extend the interface of *GenEqual*:

$$\begin{aligned}\text{GenComparable} &= \lambda\omega.(\text{GenEqual}[\omega] \cup \{\text{lessThan} : \omega \to \text{Boolean},\\ &\qquad\qquad \text{greaterThan} : \omega \to \text{Boolean}\})\end{aligned}$$

$$\begin{aligned}&= \lambda\omega.\{\text{equal} : \omega \to \text{Boolean}, \text{notEqual} : \omega \to \text{Boolean},\\ &\qquad \text{lessThan} : \omega \to \text{Boolean}, \text{greaterThan} : \omega \to \text{Boolean}\}\end{aligned}$$

In the above, the new element type parameter is ω. This is introduced on the outside, then the result is formed internally as the union of the old interface and the extra methods. Note that the inherited interface is given by: *GenEqual*[ω]. This essentially applies the old generator to the new parameter, and creates a record type in which $\{\omega/\tau\}$ has been substituted throughout:

$$\text{GenEqual}[\omega] = \{\text{equal} : \omega \to \text{Boolean}, \text{notEqual} : \omega \to \text{Boolean}\}$$

and this inherited record type can be safely unioned with the additional method signatures. A similar trick is used to extend the *List* generator to produce the *SortedList* generator:

$$\text{GenSortedList} = \lambda\omega.\lambda\psi.(\text{GenList}[\omega, \psi] \cup \{\text{least} : \omega, \text{greatest} : \omega\})$$

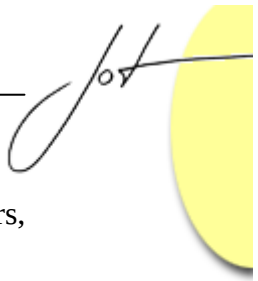
$$\begin{aligned}&= \lambda\omega.\lambda\psi.\{\ \text{insert} : \omega \to \psi\ \text{head} : \omega, \text{tail} : \psi, \text{length} : \text{Integer},\\ &\qquad\qquad \text{elem} : \omega \to \text{Boolean}, \text{least} : \omega, \text{greatest} : \omega\}\end{aligned}$$

Note again how this creates an inherited version of the old *List* interface by applying the old generator to the new parameters: *GenList*[ω, ψ]. This returns a record type in which $\{\omega/\tau, \psi/\sigma\}$ have been substituted throughout:

$$\begin{aligned}\text{GenList}[\omega, \psi] = \{\ &\text{insert} : \omega \to \psi\ \text{head} : \omega, \text{tail} : \psi,\\ &\text{length} : \text{Integer}, \text{elem} : \omega \to \text{Boolean}\}\end{aligned}$$

and this inherited record type can be safely unioned with the new method signatures. Previously, we noted that the reason for substituting parameters like this was to ensure, in the inheriting class, that the self-type was consistently denoted by ψ (rather than a mixture of new ψ and old σ) [3, 4]. Now that we are dealing with nested classes, the rules must be applied recursively for the element type, which must be consistently ω throughout.

The stronger constraint for our new *SortedList* class, which ensures that the type paramter ψ ranges over only those types which are at least *SortedLists* of *Comparable*



elements, is given by a nested F-bound that is written in terms of the two new generators, but is otherwise similar to the nested F-bound for a *List*:

$$\forall(\omega <: \text{GenComparable}[\omega]).\forall(\psi <: \text{GenSortedList}[\omega, \psi]).$$
$$(\ \ldots \text{ some definition involving } \omega \text{ and } \psi \ \ldots\ )$$

7 SUBCLASSING IS THE INCREASING OF TYPE CONSTRAINTS

What is even more sophisticated in this model of inheritance is the preservation of the constraints on all the type parameters. Technically, we are substituting parameters bounded by one set of constraints with new parameters bounded by a different set of constraints. Are the substitutions all legal? To do this, we check the expectations made internally by the type generators.

When we apply: *GenEqual*[ω], we substitute {ω/τ}. Now, the τ-parameter expects to receive a type satisfying: $\tau <:$ *GenEqual*[τ], in other words, a type with at least the interface of the *Equal* class. It so happens that we are replacing one parameter with another parameter, rather than an actual type. So instead, we have to consider all the types that might be allowed by the new parameter. The new parameter ω expects to receive a type satisfying: $\omega <:$ *GenComparable*[ω], in other words, any type with at least the interface of the *Comparable* class. So, we have to ensure that all types that we could substitute for ω will also be acceptable types for the parameter τ. Formally, we determine this using the pointwise subtyping rule [4], checking the assertion:

$$\forall\tau\ .\ \text{GenComparable}[\tau] <: \text{GenEqual}[\tau]$$

By inspection, the *Comparable* interface includes the *Equal* interface, no matter what value we supply for τ, so this substitution is proven legitimate.

A similar process happens when we apply: *GenList*[ω, ψ], causing the substitutions {ω/τ, ψ/σ}. We follow the same argument for ω, and then a similar argument for ψ as it replaces the parameter σ. Eventually, we conclude that any type which we could substitute for ψ will also be an acceptable type for the parameter σ, as a result of the pointwise rule:

$$\forall\tau\ .\ \forall\sigma\ .\ \text{GenSortedList}[\tau, \sigma] <: \text{GenList}[\tau, \sigma]$$

This expresses the assertion that the interfaces of *SortedList* and *List* stand in a pointwise subtyping relationship, no matter what types we substitute for τ and σ. By inspection of the two interfaces, we conclude this holds, so the substitution is proven legitimate.

In general, the derivation of subclasses follows a process of monotonic restriction (steadily increasing, or stable constraint) on all the type parameters involved. The kinds of restriction that are permitted can be modelled as a kind of commuting diagram, shown in figure 1.

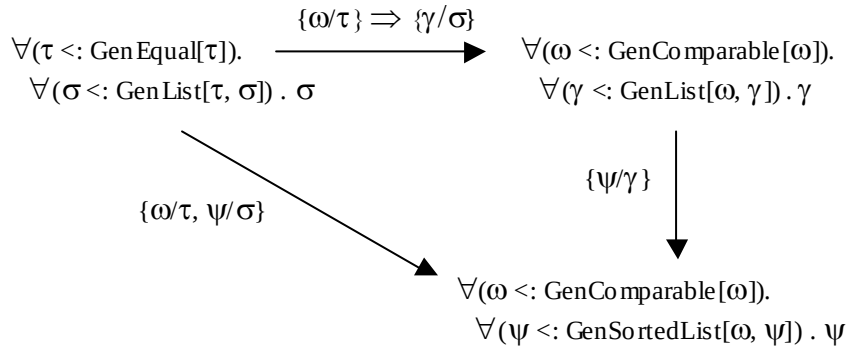


Figure 1: Commuting diagram of increasing constraints

This shows that you can start with a polymorphic list σ of elements τ with equality, and can choose to restrict the element-type, giving a similar list γ of comparable elements ω with less-than ordering, and then constrain the list further to a sorted list ψ with comparable elements ω. Alternatively, both restrictions on the element-type and list-type may be carried out simultaneously, which is illustrated by the diagonal path. A fourth substitution path, changing a list σ to a sorted list ψ *without* changing the element type is not possible, according to the diagram. This is because a sorted list depends on having a comparable element-type.

One new thing that the diagram shows is that substituting one of the "type elements" of the main class, here, the element-type of the list, leads to a change in the self-type also. Merely choosing to substitute {ω/τ} with a more constrained element-type entails the substitution of the self-type {γ/σ} in figure 1. This is something rarely considered in informal treatments of object-oriented type systems. Why has the self-type of the list changed? Well, the result of one access method is typed: *head* : τ for the original list; but after we have substituted {ω/τ}, then we would expect this method to return a different type: *head* : ω. Therefore, this must be the *head* of a different type of list. Jens Palsberg and Michael Schwartzbach were the first to explore the consequential effects of type substitutions to any significant degree in object-oriented type systems [8]. This is still a relatively new area, the subject of continuing research.

8 CONCLUSIONS

We have shown how a proper treatment of object-oriented polymorphism involves the use and manipulation of constrained type parameters. In particular, the polymorphic type intended by a class identifier in object-oriented programs is quite a complex notion. It is a parametric type, restricted to receive only types with a certain interface structure. But more interestingly, the parametric type depends in turn on all the type elements of the class in question. So, the polymorphic aliasing of one of these element-types may also affect the type of the whole class. During the operation of inheritance, many type substitutions are performed, both of the element-types and the class's self-type. This

follows a pattern which gradually increases the constraints on all type parameters monotonically, restricting the types which may eventually be valid members of the subclass.

Very few attempts have been made so far to build a practical object-oriented programming language based on entirely parametric treatments of polymorphism. One attempt was by Simons *et al.* in the early 1990s [9, 10]. In the experimental language *Brunel*, simple types were written in the usual way as: `x:Integer; y:Boolean;` and polymorphic class-types were written using explicit type parameters: `p:P`, where `#Point[P]` introduced the parameter and expressed the F-bound constraint that `P` is in the class of *Points*. However, the language eventually proved unwieldy, due to the stacking up of type parameters. Every class that itself contained further class-elements had to declare the element-types up front. For larger classes, this was a considerably high overhead and eventually was judged impractical for a real programming language. It seems that the most practical way ahead should be to design languages that can keep track automatically of all the complex, and interdependent type substitutions. This might eventually lead to a whole new kind of compiler technology.

REFERENCES

[1] A J H Simons, "The theory of classification, part 18: Polymorphism through the looking glass", *Journal of Object Technology, 4(4), May-June 2005*, 7-18. http://www.jot.fm/issues/issue_2005_05/column1

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

[3] A J H Simons, "The theory of classification, part 7: A class is a type family", *Journal of Object Technology, 2(3), May-June 2003*, 13-22. http://www.jot.fm/issues/issue_2003_05/column2

[4] A J H Simons, "The theory of classification, part 8: Classification and inheritance", *Journal of Object Technology, 2(4), July-August 2003*, 55-64. http://www.jot.fm/issues/issue_2003_07/index_html

[5] A J H Simons, "The theory of classification, part 4: Object types and subtyping", *Journal of Object Technology, 1(5), November-December 2002*, 27-35 http://www.jot.fm/issues/issue_2002_11/column2

[6] S Peyton-Jones et al., *The Haskell 98 Language and Libraries: the Revised Report* (Cambridge, UK: CUP, 2003), 270pp. Also pub. as special edn. *Journal of Functional Programming, 13(1)*, January, 2003. Online version: http://www.haskell.org/onlinereport/

[7] A J H Simons, "The theory of classification, part 13: Template classes and genericity", *Journal of Object Technology, 3 (7), July-August 2004,*, 15-25. http://www.jot.fm/issues/issue_2004_07/index_html

[8] J Palsberg and M I Schwartzbach, *Object-Oriented Type Systems* (Chichester: John Wiley, 1994).

[8] A J H Simons, Low E-K and Ng Y-M, "An optimising delivery system for object-oriented software", *Object-Oriented Systems, 1 (1)* (1994), 21-44.

[9] A J H Simons, *A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language*, PhD Thesis, Department of Computer Science, University of Sheffield (Sheffield, 1995), 255pp.

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching Quality 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.