This is a repository copy of *The theory of classification part 20: modular checking of classtypes*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/79263/

Version: Published Version

Article:

The Theory of Classification Part 20: Modular Checking of Classtypes

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

1   INTRODUCTION

This is the final article in an informal series on the *Theory of Classification,* which has considered the theoretical notions of *type* and *class* in object-oriented languages. The series began by constructing models of objects, types, classes and inheritance, then branched out into interesting areas such as mixins, multiple inheritance and generic classes. The core of our argument has been that the notions of *class* and *type* are distinct, but both can be described formally in the λ-calculus. Strongly-typed object-oriented programming languages are largely based on the idea that "a class is a type" and "subclassing is subtyping" [1]. In earlier articles, we demonstrated why this is not really satisfactory as a formal model of classes and classification. A type system based on first-order types and subtyping:

- fails to capture natural relationships between recursive types (whose methods accept or return values of the same type as themselves), since recursive types can have no proper subtypes [2];
- loses type information when methods are inherited [2], requiring the use of *type downcasting* everywhere to recover the most specific type of the object returned by a general method, which is tantamount to breaking the type-system;
- conflicts with the notion of *type classes* adopted elsewhere in functional programming languages like Haskell, which use *type parameters* to express this notion [3].

Instead, we have argued that a class is a *family of related types*, which can only be expressed in a second-order type system with polymorphism [4]. Classical polymorphism is represented using type parameters that range over many different actual types, but object-oriented programming requires a kind of polymorphism where type parameters receive only certain related types that satisfy a particular interface description. Mathematically, this is constructed by placing constraints on type parameters, called *function bounds*, or *F-bounds*, which have form: $\tau <: F[\tau]$, where F is a type function, describing the shape of the interface that the type τ is expected to satisfy [5]. However,

while this gives a much more satisfying account of classes and classification, very few programming languages have ventured into this new and exciting territory. In this final article, we try to understand why this is so; and what practical problems remain to be solved in the modular checking of class-types.

2   TRADING MODULARITY AND EXPRESSIVENESS

A first-order type system has two things to commend it. Firstly, it is quite simple to implement a type-checker that can check types for exact correspondence, or for subtype compatibility with a given type. The type of the source object can be compared with that of the target variable to see if the former can be converted up to the latter, using subtyping rules like those we discussed in [1]. Secondly, code that has been checked once need never be checked again, or recompiled in new contexts. This is because the type system can never reveal more specific information about an object that is passed into a more general variable (which we have called the "type loss problem"), so the code need only be checked once over the most general type that it can accept. This, more than any other reason, is why object-oriented languages have been slow to take up the new insights into the nature of classes and classification: the desire to have *modular* and *incremental* compilation. Without this, it would not be possible to build industrial-scale systems.

In the last two articles, we showed that full support for the notion of classes and subclassing requires a distinction to be drawn between simple, monomorphic *types* and polymorphic *classes*, the latter formally expressed using type parameters [6]. However, this means that type checking rules are more complicated. The compiler has to keep track of sets of type parameters, one for each variable with a "class-type", and has to know how to substitute one parameter for another when values are passed, and also check that the various constraints on the parameters will allow the given substitutions [3].

Now, different type substitutions may happen upon different occasions. For example, consider a polymorphic method for moving graphical shapes on a screen, that accepts *Integer* coordinates and returns the moved object:

$$move : \forall(\tau <: \text{GenShape}[\tau]) . \tau \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \tau)$$

This method is defined for a polymorphic class of *Shapes*, expressed by $\tau <: GenShape[\tau]$ in the style described in earlier articles [2, 4]. Below, we assume that *Square* and *Circle* are exact types that satisfy this F-bound constraint. Now, if *move* is legally invoked on different actual shapes on different occasions, say on a *Square* and a *Circle*, this will cause the two different type substitutions *{Square/τ}* and *{Circle/τ}*, yielding two differently-typed versions of the *move* method. These variants will have have the exact types:

$$move : \text{Square} \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Square})$$
$$move : \text{Circle} \rightarrow (\text{Integer} \rightarrow \text{Integer} \rightarrow \text{Circle})$$
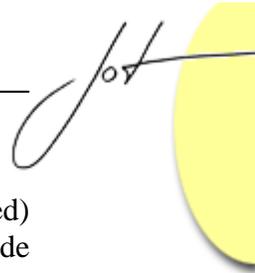
for the duration of the binding of the (polymorphic) argument to the (exactly-typed) objects. Does this mean that we must compile two different versions of the source-code for *move*?

Well, if *Circle* and *Square* were passed by value, then we should need multiple compilations of the source to handle the different physical layout of each type – this would be analogous to the *template mechanism* in C++, in which multiple copies of the template code are compiled, one copy for each distinct type-instantiation. However, it is more likely that the objects will be passed by reference and both will share the same physical layout (in the low-order bytes) for storing information about their screen location. This case therefore has more similarities with Java 1.5's treatment of type parameters. They are used in the typechecker to eliminate the need for explicit type downcasting, but are erased later in the virtual machine, in which objects are treated in the same way as in the usual subtyping approach. So, having flexible typing for *move* does not necessarily require multiple compilations of this method.

Nonetheless, the polymorphic typing situation is quite different from the kind of typing that is possible in a first-order type system. In the latter, the result of *move* can only ever have the general type *Shape*, which is typically not useful, especially if we want to do something else with the moved object (for which we would first have to perform a *type downcast*). But, in the second-order type system, we recover the exact type of the moved object straight away. This is good, from the point of view of expressiveness.

Now, consider the context in which *move* was called. What does this context expect the result-type to be? It knows the result must be some type $\tau <: GenShape[\tau]$, but, on different occasions, it receives back objects of the more specific types *Square* and *Circle*. It is possible to optimise further method invocations on these results, on the basis of this type analysis. For example, imagine that the *Shape* class declares an abstract method *area(),* which has distinct implementations in all subclasses. The method *area()* may be statically bound, if we can tell in advance that the target is definitely a *Square* or a *Circle*, rather than some unknown kind of *Shape*, for which we would have to insert a dynamically-bound call, to detect the exact type at run-time. However, using the more expressive polymorphic type system, we can propagate exact type information back into the calling context and choose to bind the *area()* method statically. The cost of this is that we must compile multiple versions of the context code.

Simons et al. first analysed these kinds of parametric issues as part of a wider optimisation strategy for object-oriented compilers, using the experimental language *Brunel* as an exemplar for the techniques [7]. They discovered that a fully parametric type system gives you the choice of using more or less of the exact type information available, to tailor the optimisation of bindings. The early binding algorithm described in [7] bears some similarities to Chambers and Ungar's notion of pre-emptive type analysis in the untyped language *Self* [8]. However, the trade-off is that the more type information you use, the more copies of the object-code you generate. In *Brunel*, a global compilation approach was adopted, in which a full type analysis of the whole program was performed and compiler switches could be set to control the amount of early binding and code duplication.

The early binding approach does not transfer over to a modular compilation strategy. When compiling modules incrementally, only partial type information is available. For example, we must compile the methods of the *Shape* class, without any knowledge that it will eventually have two subclasses *Circle* and *Square*, which we previously assumed were the "leaves" in the class hierarchy, becoming the exact types of objects used in the program. Obviously, the compiler could not know in advance whether these are in fact leaf-nodes, or whether they too might eventually be specialised further. The best that can be done is to insert a dynamically-bound call for abstract methods like *area()*. No further optimisation can be performed. However, the parametric type information can be retained and used to avoid type downcasting on the result of *move()*.

3   A UNIFICATION APPROACH TO PARAMETRIC TYPING

Java 1.5 is introducing a form of parametric type analysis that captures an aspect of the strategy we describe above. However, parameters will be used only for certain generic classes, like those in the *Java Collections* framework, and will only characterise the element-types of these collections. In our view, this same approach could be used to characterise all class-types, in the manner described in [3]. All variables marked with a "class-type" are polymorphic and so should be treated in a parametric way. The parametric type information could be used everywhere in the type-checker to obtain more exact type information, but erased in the runtime model. This would allow code modules (classes, in Java) to be incrementally compiled, provided that most calls were bound dynamically, as is usual in Java. But it would also allow some optimisations and static binding of methods for simple leaf-classes, like *Integer* and *Boolean*, for which the compiler could be told that no further specialisations were intended. (In fact, the Java simple types *int, float*, etc. could be merged with the class-types in a single-rooted class hierarchy).

The most significant challenge to the development of proper, parametric type-checkers is the problem of unifying different polymorphic types. This happens whenever one polymorphic variable is passed as an argument or result to another method, where the formal and actual types of the variables may be distinct. However, there exists at least one programming language, Prolog, which already has a similar algorithm at the heart of its interpreter. This is the *most general unifier* (MGU) algorithm, which calculates the most general term that can result from the unification of two other terms.

For those unfamiliar with Prolog, this is a language in which the programmer constructs logical expressions, in a declarative style, and program execution is then analogous to solving the simultaneous equation expressed by all the terms. Terms are structured as *predicates*, which may contain grounded values (written in lowercase) or variables (written capitalised). So, the following two terms may exist:

loves(john, Loved).   *loves(Lover, mary).*

and it is possible to see that these terms may be unified, yielding the MGU:

loves(john, mary).   with substitutions: {*john/Lover, mary/Loved*}

In this unification, the variable *Loved* in the left-hand term receives the value *mary* from the right-hand term; and the variable *Lover* in the right-hand term receives the value *john* from the left-hand term in a symmetrical act of merging. In logic, this is equivalent to saying: "these terms can be unified, provided that the *Lover* stands for *john* and the *Loved* person stands for *mary*". The important thing to note is that both the left- and right-hand terms contributed some of the specific values to the resulting unified term.

It is not difficult to see how this kind of substitution mirrors the process a polymorphic type checker must go through when a polymorphic variable receives an object of some exact type. However, this is an even better analogy for when two polymorphic variables have their types merged, for example when a polymorphic variable with the type $\tau <: F[\tau]$ is passed into a method, whose formal argument has the type $\sigma <: G[\sigma]$. For the duration of the binding, $\sigma == \tau$ and therefore the dual constraint must apply: $\tau <: F[\tau] \wedge \tau <: G[\tau]$. In earlier articles dealing with inheritance and multiple inheritance, we called this an *intersection type* [9, 10] because the type variable τ is being constrained to accept two different, overlapping sets of types and therefore accepts the intersection of these sets. Accordingly, we used $\tau <: F[\tau] \wedge G[\tau]$ to denote an intersection on the parameter τ.

The important thing to note is that this merging of type-constraints is even-handed: it doesn't matter whether $\tau <: F[\tau]$ or $\tau <: G[\tau]$ is the more restricting F-bound, since any type replacing τ must satisfy both constraints. So, this mechanism is adequate to handle specialisation (when a polymorphic type is replaced by a more restricted polymorphic type [9]) and also the kind of symmetrical type-merger that happens with multiple inheritance (when the polymorphic types of two parent classes become unified in the child [10]). The latter case also extends to languages with a combination of single inheritance and multiple interface satisfaction (the λ-calculus model treats all class-like and interface-like types in the same way). So, parametric type checkers will in future need to perform unification on type variables and compute the pool of merged constraints; but fortunately this is no more difficult than unification in Prolog.

4   DISGUISING THE TYPE PARAMETERS

Another of the challenges to be faced is how to make genuinely polymorphic[1] languages attractive to programmers. The previous article [3] reported how such languages tend to become swamped by the proliferation of type parameters. If each class requires its own self-type parameter, then a class with "class-typed" polymorphic variables in its attributes and methods needs a distinct parameter for each such variable that could eventually be

---

[1] By this, we mean languages with second-order type systems; the kind of type aliasing performed in first-order subtyping is not technically polymorphism, but something much weaker.

bound to a distinct type. If the classes describing these elements also contain further "class-typed" variables of their own, then our original class has a declaration which is already three layers of parameters deep! We showed how the order of declaration was significant, in that the type parameters for the inner element classes have to be declared first, on the outside, and the dependent type parameters standing for the outer composite classes have to be declared within their scope (in a second-order type system). Essentially, any polymorphic structure must expose, in its interface, all of the different type parameters which could be bound to a different type at some point during the execution of the program.
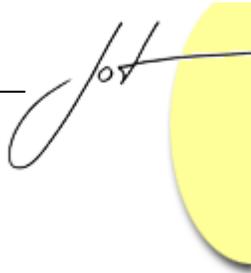
Various attempts have therefore been made to disguise the existence of type parameters and the many substitution operations that must be performed on them. Perhaps the most careful and thorough of these treatments is Bruce's *matching*. This is an alternative to F-bounds that establishes flexible type compatibility relationships between class-types. Bruce and his co-workers started building type-safe experimental object-oriented languages in the early 1990s. TOOPL and TOOPLE were functional-style object languages (rather like the λ-calculus models we have used in this series), which supported both simple subtyping and a new treatment of the *self*-type using a distinguished type variable called *MyType* [11]. Originally, the motivation for *MyType* arose from considering the same problems with subtyping in the presence of recursive types that led Cook to devise F-bounded quantification [5]; and Bruce's early treatments relied on an F-bounded explanation. However, in later work, Bruce defined complete and consistent type rules for *MyType* which dispensed with explicit F-bounds altogether. The later languages TOIL and PolyTOIL were styled more like imperative object-oriented languages, with variable reassignment [12].

The first advantage gained through using a distinguished *MyType* is that this type variable is implicitly defined within each class-type. The meaning of *MyType* is rather like the type parameter σ in F-bounded constructions like: $\sigma <: GenMyType[\sigma]$, but for each new class, *MyType* is implicitly rebound to refer to the *self*-type of the new class-type. The implicit declaration of *MyType* can be seen below in the type declaration of the abstract *Comparable*, which defines abstract methods *lessThan* and *equal*; and in the type declaration of the concrete *BoxedInteger*, which is essentially a wrapper for a simple *int* type:

Comparable = ObjectType { lessThan : MyType $\rightarrow$ bool; equal : MyType $\rightarrow$ bool }

BoxedInteger = ObjectType { lessThan : MyType $\rightarrow$ bool; equal : MyType $\rightarrow$ bool; getValue : void $\rightarrow$ int; setValue : int $\rightarrow$ void }

where *ObjectType* is the keyword introducing the new object types (these examples are adapted from [12]). Notice how *MyType* occurs freely inside both definitions, but stands in each case for a different polymorphic type. In our approach using F-bounds, we would introduce two generators, which declare two self-type parameters σ and τ up-front, and then construct F-bounds to use in type expressions:

$$\text{GenComparable} = \lambda\sigma.\{\ \text{lessThan} : \sigma \rightarrow \text{bool}; \text{equal} : \sigma \rightarrow \text{bool}\ \}$$
$$\text{GenBoxedInteger} = \lambda\tau.\{\text{lessThan} : \tau \rightarrow \text{bool}; \text{equal} : \tau \rightarrow \text{bool};$$
$$\text{getValue} : \text{void} \rightarrow \text{int}; \text{setValue} : \text{int} \rightarrow \text{void}\ \}$$

$$\forall(\sigma <: \text{GenComparable}[\sigma]).\text{some\_type\_expr\_using}(\sigma)$$
$$\forall(\tau <: \text{GenNumType}[\tau]).\text{some\_type\_expr\_using}(\tau)$$

Now, in PolyTOIL you can declare variables with object types directly, for example, it is legal to declare myInteger : BoxedInteger, in which all internal occurrences of MyType are eventually resolved (in the type rules) to refer to a BoxedInteger, recursively. The parameter MyType is replaced by the actual type of the object receiver, when a method is invoked upon it. In our approach using F-bounds, this requires taking a fixpoint first:

$$\text{BoxedInteger} = \mathbf{Y}\ \text{GenBoxedInteger};\qquad \text{recursively bind }\{\text{BoxedInteger}/\tau\}$$
$$\text{myInteger} : \text{BoxedInteger}$$

The second innovation in Bruce's approach is the way in which subclass relationships can be expressed directly between these object types, using the novel *matching* mechanism. You can assert the usual subclass relationship as: *BoxedInteger <# Comparable* (read this as "*BoxedInteger* matches *Comparable*"), where "<#" is the new matching operator. Bruce describes the matching relation as:

"the same as subtyping in the absence of the *MyType* construct, but differs in the presence of *MyType*, because *MyType* implicitly has different meanings in different types." [13].

In fact, matching behaves in a similar manner to F-bounded inclusion, in the presence of *MyType*, but in a similar manner to simple subtyping elsewhere. In our approach, we would have to establish a second-order pointwise subtyping relationship between the two corresponding type generators, to ensure that the two parameters were unified before interfaces were compared, and then that one interface were longer than the other:

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

Bruce's rules simply compare the structure of object types, in which all occurrences of *MyType* are considered equivalent when determining if one type matches another. This dispenses with some of the fiddly detail of parameters. Matching has the same expressive power as F-bounds, for example, note that while *BoxedInteger <# Comparable*, the subtyping relationship *BoxedInteger <: Comparable* does not hold, because *MyType* occurs as a method argument type (in contravariant position). The type rules for inheritance ensure that *MyType* evolves smoothly to represent the self-type of inheriting classes, which dispenses with another layer of type substitutions in the explicit approach. Finally, if the programmer so wishes, it is also possible to declare explicit type parameters in PolyTOIL. For example, the element-type of a *SortedList* may be given as

elt : T <# Comparable, to denote any type which matches *Comparable*. This is the analogue of $\forall(\tau <: GenComparable[\tau])$. *elt : τ* in our approach. In later work, Bruce developed a version of matching with "hash types" that was sufficiently expressive that subtyping could be dropped altogether [14]. This is closer to our approach, which recognises only exact simple types, or parametric polymorphic types.

5   IMPLICIT CLASS-TYPE SUBSTITUTIONS

Perhaps the trickiest issue for future compilers, with the more thorough kind of type analysis we have been proposing here, is to keep track of all the subtle changes to type descriptions that happen as a result of objects being mutually related to each other. This can lead to some hidden evolution in the types of expressions, of which the programmer may not be aware! Consider a class hierarchy describing the various kinds of *Vehicle* that exist, together with the different kinds of *Location* in which such vehicles are typically kept. The root concepts could be described in Java as:

```java
class Vehicle {
  private Person myOwn;
  private Location myLoc;
  public Vehicle(Person p) { myOwn = p; myLoc = null; }
  public Vehicle(Person p, Location c) {
    myOwn = p; myLoc = c; c.keep(this); }
  public Person owner() { return myOwn; }
  public Location keptAt() { return myLoc; }
  public void keepAt(Location c) {
    myLoc = c; if (c.keeps() != this) c.keep(this); }
}

class Location {
  private String myAdr;
  private Vehicle myVeh;
  public Location(String a) { myAddr = a; myVeh = null; }
  public Location(String a, Vehicle v) {
    myAdr = a; myVeh = v; v.keepAt(this); }
  public String address() { return myAdr; }
  public Vehicle keeps() { return myVeh; }
  public void keep(Vehicle v) {
    myVeh = v; if (v.keptAt() != this) v.keepAt(this); }
}
```

We can build a pair of mutually-referencing objects by constructing a *Vehicle* and a *Location* in either order, since their constructors set up the reciprocal references:

```java
Person wal = new Person("Wallace");
Location lcn = new Location("42 West Wallaby Street");
Vehicle veh = new Vehicle(wal, lcn);
```
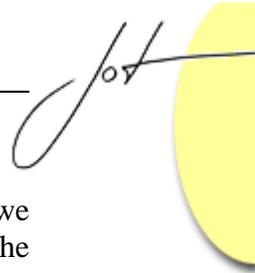
Now, the intention is that these classes should be specialised in pairs, for example, we might create *Car/Garage*, or *Aircraft/Hangar*, or *Ship/Port* pairs. But what happens if the programmer only specialises one half of this mutual relationship?

```
class Car extends Vehicle {
  public Car(Person p, Location c) { super(p, c); }
}
Person wen = new Person("Wendolene");
Car car = new Car(wen);
Location loc = new Location("3 Town Square", car);
```

In Java, the result of enquiring *loc.keeps()* always has the type *Vehicle* (we are suffering from the "type-loss" problem again), but dynamically it contains an instance of *Car*. In a parametric type system, we would expect to be able to recover the exact vehicle-type. This is because, when the *Location* is constructed with a value of the exact type *Car*, this type is propagated into the vehicle-type parameter τ of *Location's* polymorphic variable *myVeh*, which we imagine might have the type:

$$\forall(\tau <: \text{GenVehicle}[\tau]) . \text{myVeh} : \tau \qquad \text{which then becomes…}$$
$$\text{myVeh} : \text{Car} \qquad \text{…after substituting } \{\text{Car}/\tau\}.$$

This is exciting from the viewpoint of type analysis; but notice that we have created a new, unforeseen type. We expected eventually to specialise *Vehicle* and *Location* in step with each other, producing *Car* and *Garage*, such that the *Garage.keeps()* method returns a *Car*, and the *Car.keptAt()* method returns a *Garage*. Because we only specialised one half of the mutual relationship, we created a new intermediate type variant, a *Location'* whose *keeps()* method returns a *Car*, rather than a *Vehicle*. This type is neither a *Location*, nor a *Garage*, but something in between.

Palsberg and Schwartzbach were the first to report such intermediate types in object-oriented languages [15]. They were using a *type substitution* mechanism, which has only slightly less expressive power than the full parametric mechanism used in our approach[2]. They discovered that checkers which perform full type analysis will inevitably synthesise many intermediate versions of types, as a result of the evolution of other closely-related types. The consensus nowadays is that a mutually-referring set of types creates another enclosing formal structure, a *closure*, which is specialised as a whole, when any one of the related types is specialised. This, then, is the challenge facing the designers of future object-oriented compilers with smart type analysis, implicit type evolution and incremental compilation.

---

[2] If you systematically substitute type X by type Y within some scope, then all Xs must change into Ys. But with type parameters, you can declare different parameters P <# X and Q <# X, choosing to substitute {Y/P} and {Z/Q}, so this gives you slightly finer control over which substitutions happen together.

6 CONCLUSIONS

Maybe in the future we will see object-oriented languages that exemplify the Theory of Classification in full. I'd like to think that one day, we could have a programming language that is based on a few simple concepts, which is as type-safe as Pascal and as expressive as Eiffel (or Algol-68, or whatever the last really good programming language was). In my crystal ball, this language has to distinguish the theoretical notions of class and type, to allow programmers to understand clearly when simple, or polymorphic typing is intended. It will relate all built-in and programmer-defined types and support obvious, intuitive notions of classification, for example, that the simple types Integer and Boolean are first-class members of the class/type hierarchy and fit underneath an abstract class of Numbers, whose abstract arithmetic-methods are appropriately specialised when they are implemented in Integer and Real. Multiple classification will be possible, such that both Complex numbers and Sets will be considered PartiallyOrdered types, Complex and Integer numbers will be considered kinds of Number, and Sets and Bags kinds of UnorderedCollection. Interfaces will be the same thing as abstract classes.

Incremental compilation will continue to be supported and dynamic binding will be the norm, with some static optimisations performed on the standard leaf-types. The syntax of these languages may start out using explicit type parameters everywhere (such as the cutting-edge work on Haskell *type classes*), but the parameters may eventually disappear inside the compiler, maybe at the loss of a small amount of flexibility and expressiveness. The compiler's ability to perform early type analysis will improve and I expect that in future, code modules will be compiled, which retain their type parameters, such that when the modules are linked and bound at their call-sites, exact type information will be propagated throughout the web of type-constraints, allowing the call-site to extract precisely-typed results. The binding of such parametric modules will result in a bi-directional flow of type-information, yielding solutions such as the "most general intersecting type", computed using unification algorithms.

Throughout my work in this area, I have been standing on the shoulders of giants. I owe particular thanks to Willam Cook, Kim Bruce and Luca Cardelli for formative conversations in the early 1990s and occasional exchanges since then. If you have been stimulated by this informal series of articles on the typing and semantics of object-oriented languages, the next stage might be to get to grips with the details of the type rules, perhaps in [13, 16]. If you have comments, insights or critiques to make, please feel free to contact me by email. If you would like to help bring about the "language with class", then I have a PhD project in this area that needs a good student.

REFERENCES

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

[2]   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*, pp 13-22, http://www.jot.fm/issues/issue_2003_05/column2

[3]   A J H Simons, "The theory of classification, part 19: The proliferation of parameters", *Journal of Object Technology, 4(5), July-August 2005*, pp 37-48, http://www.jot.fm/issues/issue_2005_07/column4

[4]   A J H Simons, "The theory of classification, part 8: Classification and inheritance", *Journal of Object Technology, 2(4), July-August 2003*, pp 55-64, http://www.jot.fm/jot/issues/issue_2003_07/column4/index_html.

[5]   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.

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

[7]   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.

[8]   C Chambers and D Ungar, "Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs", *Proc. 5th ACM Conf. Prog. Lang. Design and Impl.,* pub. *ACM Sigplan Notices, 25(6),* (1990), 150-164. Reprinted in: *Lisp and Symbolic Computation, 4(3),* (1991), 283-310.

[9]   A J H Simons, "The theory of classification, part 16: Rules of extension and the typing of inheritance", *Journal of Object Technology, 4 (1), January-February 2005*, pp 13-25, http://www.jot.fm/issues/issue_2005_01/column2

[10]   A J H Simons, "The theory of classification, Part 17: Multiple inheritance and the resolution of inheritance conflicts", *Journal of Object Technology, 4 (2), March - April 2005,* pp 15-26, http://www.jot.fm/issues/issue_2005_03/-column2.

[11]   K Bruce, J Crabtree, A Dimock, R Muller, T Murtaugh and R van Gent, "Safe and decidable type checking in an object-oriented language", Proc. 8th ACM Conf. Obj.-Oriented. Prog. Sys., Lang. and Appl., (1993), 29-46.

[12]   K Bruce, A Schuett and R van Gent, "PolyTOIL: A type-safe, polymorphic object-oriented language," *ACM Trans. Prog. Langs. and Sys., 25(2),* March (2003), 225-290.

[13]   K B Bruce, *Foundations of Object-Oriented Languages: Types and Semantics*, (Cambridge MA: MIT Press, 2002).

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

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

[16]   M Abadi and L Cardelli. *A Theory of Objects. Monographs in Computer Science,* Springer-Verlag, 1996.

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.