
Article:
Simons, A.J.H. (2003) The theory of classification part 6: the subtyping inquisition. Journal of Object Technology, 2 (2). 17 - 26. ISSN 1660-1769

JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 2, March-April 2003

The Theory of Classification Part 6: The Subtyping Inquisition

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

1 INTRODUCTION

This is the sixth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The series has been investigating the notion of simple object types and has so far developed a *theory of subtyping* which judges object type compatibility from both the *syntactic* point of view, that is, by the type signatures of an object's methods [1] and from the *semantic* point of view, that is, by the logical axioms asserted on an object's methods [2]. In terms of the dimensions of type checking in figure 1, we have considered the *exact* type correspondence of signatures (box 2) and behaviour (box 3), and the *subtype* correspondence of modified signatures (box 5) and refined behaviour (box 6).

	Schemas	**Interfaces**	**Algebras**
Exact	**1**	**2**	**3**
Subtyping	**4**	**5**	**6**
Subclassing	**7**	**8**	**9**

Figure 1: Dimensions of Type Checking

This kind of theory provides a much-needed tool for analysing the type safety and behavioural correctness of programming languages. In the late 1980s and early 1990s, the possibility that object-oriented languages might be insecure in their type systems, when judged according to subtyping [3], caused quite a stir, particularly in the software engineering community. This led to the greater prominence of subtype-conformant languages [4], but also sparked a new interest in the way object-oriented languages really seemed to behave [5]. The debate swung between the desire to force languages into obeying subtyping and the desire to develop more sophisticated formal models of object-

Cite this column as follows: Anthony J.H. Simons: "The Theory of Classification. Part 6: The Subtyping Inquisition", in *Journal of Object Technology*, vol. 2, no. 2, March-April 2003, pp. 17-26.
http://www.jot.fm/issues/issue_2003_03/column2

oriented classes and inheritance. In this article, we take the former side (as devil's advocate, since we shall take the latter side in a subsequent article) and examine a number of popular object-oriented languages for their type safety, asking of each candidate subclass: "Are you, or have you ever been, a proper subtype?".

2 A PRACTICAL TEST FOR SUBTYPING

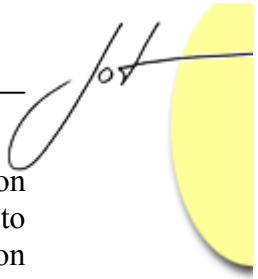
How well do popular object-oriented languages follow the rules of subtyping? How type secure are they generally? A quick survey may reveal hidden weaknesses or unappreciated strengths in your favourite language. Recall that our theory deals in terms of *objects* and *object types* (so far, we have not used the term *class* in any formal sense). Practical object-oriented languages have concrete *classes* which define the structure and behaviour of objects created at runtime. A class in this sense has both an implementation aspect and a typeful aspect, in that class names are usually also treated as type identifiers. We shall assume the same correspondence when making typing judgements here.

The main syntactic type rules of interest derive from the *record subtyping* and *function subtyping* rules [1]. These can be expressed informally for classes-viewed-as-types in the following way:

- Where a class S is intended to be a subtype of a class T, it must obey the extension rule: S may add to the methods of T, but never remove any methods from T; and the overriding rule: S may replace some methods of T, so long as the replacement methods R_i in S are subtypes of the corresponding methods M_i in T that they replace.
- Where a method R is intended to be a subtype replacement for a method M, it must obey the argument contravariance rule: any arguments of R may be more general than corresponding arguments in M, but never more specific; and the result covariance rule: the result of R may be more specific than the result of M, but never more general.

The main semantic behavioural rules of interest derive from the *addition of new constraints* and the *generalisation of constraints* [2], both of which are said to *strengthen* an axiom. These are converted into the more familiar assertion format below:

- Where a class S is intended to be a behavioural subtype of a class T, it must obey the invariant strengthening rule: S may have a stronger invariant than T, but never weaker; and the behavioural conformance rule: any replacement methods R_i in S must be behavioural subtypes of the corresponding methods M_i in T that they replace.
- Where a method R is intended to be a behavioural subtype replacement for a method M, it must obey the precondition weakening rule: R may weaken M's precondition, but never strengthen it; and the postcondition strengthening rule: R may strengthen M's postcondition, but never weaken it.



In Eiffel [6], strengthening is typically obtained by adding extra assertions in conjunction (combined with logical AND) with the existing set, making the constraint harder to satisfy. Weakening is obtained by providing alternative assertions in disjunction (combined with logical OR) with the existing set, making the constraint easier to satisfy.

3 SMALLTALK AND OBJECTIVE C

Smalltalk [7] is an interesting language to evaluate against these rules, since it is considered by some to be an *untyped* language. From a schema-based perspective (the first column in figure 1), everything is implemented as a uniform object type. Further than this, the static types of variables are not given, so they cannot be checked at all. However, it is reasonable to think of objects at runtime as having a type, corresponding to their class identifier. This type is used implicitly during method lookup to select from method dispatch tables. Type checking is dynamic, in the sense that type errors only appear as "message not understood" exceptions at runtime, after a search for a given method in the class hierarchy has failed. Smalltalk is therefore usually considered to be weakly type checked, since it cannot detect incorrect invocations at compile time.

Fundamentally, Smalltalk expects a subclass to add to the methods of its superclass, which follows the extension rule, and method overriding is supported, on a name-equivalence basis. A weakness is where a replacement method sometimes *derails* the operation of the original version: for example, though all *Collection*s may *add:* elements, a *FixedSizeCollection* may not, so it redefines the *add:* method to raise an exception. This is tantamount to removing a method in a subclass, which violates the extension rule. Smalltalk's method overriding rule is slightly stronger than it first appears, due to the syntactic checking that is carried out upon the distinctive infix message syntax. For example, in the Smalltalk expression:

```
myArray at: 3 put: 42.
```

the *at:put:* message requires exactly two arguments, one inserted after each colon, though the precise types of these arguments cannot be specified. If this method were overridden, the replacement method would have exactly the same name and so it would expect exactly the same number of arguments. This allows us to assert that Smalltalk has a rudimentary interface check in its method overriding. However, we cannot say that the arity of methods is statically checked against invocations, since a message with fewer arguments would simply be considered a different method instead.

Objective C [8] allows the programmer to mix plain C code, which is statically checked according to the rules of C, with dynamically checked object message expressions, written in the same style as Smalltalk. By default, all objects have the static type *id*, the base type of all object references. A variation on this is where the programmer may assert that an object is of a more specific object type than *id*. This does not affect the dynamic binding of methods, but does allow a compiler to check whether a method of a given name exists for that type. Given the contrasting declarations:

```
id squareOne;
Shape * squareTwo;
```

messages sent to *squareOne* will not be statically checked, but messages sent to *squareTwo* can be checked to see if they are defined for objects of at least the type *Shape** (pointer to *Shape*). A further feature of Objective C is the ability to attach type *protocols* (an early foreshadowing of Java interfaces) to any class, independent of its position in the class hierarchy. Variables bearing a protocol-type may be checked in a similar fashion. Smalltalk and Objective C ultimately have weak syntactic type checking. It is possible in either language to write expressions which compile, but fail at runtime due to type-related errors.

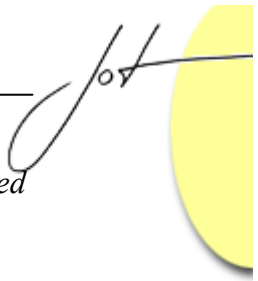
4 C++ AND JAVA

Two languages which come closer to following the syntactic rules of subtyping are C++ [9] and Java [10]. Variables are strongly typed in both languages, such that all expressions may be checked against the declared types of methods. By default, C++ binds methods statically unless you request dynamic binding (with the *virtual* keyword). Java's methods are dynamically bound, unless marked *final*, in which case compilers may choose to bind them statically, since they will never be overridden.

Like Smalltalk, these languages expect a subclass to add to the methods of a superclass. Although it is still possible to derail methods in C++ and Java, the temptation to do this is much reduced. Smalltalk's dependence on derailment arises from having only a single classification hierarchy in which to factor out all behaviours. As a result, some generic methods are declared which do not strictly apply to every subclass. By contrast, it is possible to apply multiple and overlapping classification schemes in C++ using multiple inheritance; and in Java using interfaces. In any case, the C++ programming culture tends to avoid large, monolithic class hierarchies.

A different threat to C++ comes through *public, protected* and *private* modes of inheritance. Only in the *public* form of inheritance does a subclass inherit the method interface of its superclass unchanged; in the other two forms, inherited methods become secrets of the subclass. This is equivalent to withdrawing a method in a subclass; however, the C++ compiler recognises that this violates subtyping and correctly disallows aliasing through superclass variables. *Friend*-declarations in C++ are a different matter, which we consider alongside *selective exports* in Eiffel, below.

The method overriding rule in C++ and Java expects a replacement method to have *exactly the same* type signature as the original. This is actually more restrictive than the function subtyping rule requires - more general argument types and more specific result types are allowed in a subtype method, even if accepting more general arguments has limited practical application [3]. Another reason lies behind the choice of this simpler, but stricter rule.



In both languages, method names are not unique within a class, but *overloaded* versions may exist, so long as they can be distinguished by the types of their arguments:

```
void setDate(int, int, int);      // set using day, month, year
void setDate(String);             // set using "dd/mm/yy" string
```

A compiler must be able to resolve the most specific type of an expression to select a unique overload. In C++, this requires a complex series of type conversions. It is difficult to combine this with a mechanism for inferring which of several overloaded versions should be replaced by a more specialised method type (the same replacement might apparently override more than one original version). So, for this pragmatic reason, overriding is restricted to methods having exactly the same type. Some recent C++ compilers allow the returned self-type (the type of *this*, the current object) to be specialised during method overriding, since overloading is resolved using argument types alone.

In terms of the syntactic rules of subtyping, Java and C++ are fairly secure. The weakness in C++ comes from the ease with which a programmer can override the type system and convert one type into another, even when this is not suitable. C++ allows both explicit and implicit type coercions between its numeric, character and boolean types, often losing information in the process (eg a *long int* converted up to a *float*). The most potentially damaging use of typecasting is where the type information attached to a pointer is thrown away ("casting to *void**") or the pointer is converted arbitrarily into another pointer type. For example:

```
Square* mySquare = (Square*) myShape;
```

is only safe if *myShape* holds an object pointer of at least the type *Square**, but in practice it could hold any type of pointer. The typecast (*Square**) is not checked in C++, such that a program could continue to run with an unsuitable value in *mySquare*, leading to a system crash. In Java, this kind of type conversion is checked at runtime, rasing an exception if *myShape* does not refer to an object of at least the *Square* type. Modern C++ has tried to address this problem by advising programmers to use the similarly-checked type-conversion operators *static_cast, dynamic_cast* and *const_cast*, instead of simply retyping variables.

C++ promotes the use of both *value* and *reference* types (here, we mean pointers), which has extra implications for the implementation schema. Exact typing (box 1 in figure 1) allows the compiler to reserve the exact amount of storage required for an object, or a pointer. Subtyping (box 2 in figure 1) is subject to different physical constraints for values and pointers. A pointer can be coerced to a supertype without difficulty and the primary data is preserved via one level of indirection, though access is limited to those fields declared in the supertype. A value can be coerced to a supertype, but any additional subclass fields are truncated, since storage is allocated in the variable itself, rather than via one level of indirection. For this reason, no dynamic binding can be applied to value-types.

C++ also has a parametric type mechanism, known as *templates*. Our theory does not yet include a treatment of parametric types. However, C++ does not check template class definitions; instead, the compiler only checks fully *instantiated* templates, which are no different from regular classes. Our existing rules may handle these.

5 EIFFEL AND TRELLIS

Eiffel [6] is one language that can be evaluated against both the syntactic and semantic subtyping rules, since it supports object and method specification using executable assertions. Checking Eiffel's syntactic type system is still a challenge, as it offers three different type mechanisms, known as *conformance*, *constrained genericity* and *type anchoring*:

- *conformance* is the regular class-subclass type compatibility relationship;
- *constrained genericity* is a parametric typing mechanism with class constraints;
- *type anchoring* is an entirely novel mechanism linking the types of variables.

We only have space to consider the first of these mechanisms in the current article; the other two mechanisms are based on insights that will form the basis for a complete re-appraisal of the formal notion of *class* and *classification* in a later article.

Like other languages, Eiffel expects a subclass to add to the methods of its superclasses and possibly redefine some. Early versions of the language ran into problems over selective inheritance, whereby a subclass could withdraw a method that was exported (ie declared public) in a superclass. From version 3.0 this was fixed and the default behaviour is to inherit all export declarations unchanged. However, selective inheritance crept in via the back door with the *undefinition* mechanism, whereby a method's effective implementation could be suspended in a subclass, turning it back into a *deferred* (ie abstract) method. While this appears to obey the letter of the law (the method remains in the subclass's public interface), it breaks the spirit of the law (it is illegal to invoke non-implemented methods) and we cite method derailment as a precedent!

Eiffel's overriding rules are ambitious. Not only can you replace methods with retyped versions, but you can also redefine attribute types in a subclass. Unfortunately, Eiffel's overriding rules are faulty from the viewpoint of subtyping. This was the infamous "Eiffel type failure" headline in 1989 [3]. Eiffel assumes that *everything* may be uniformly specialised in a subclass: attributes, method arguments and method results.

This rubs up against strict subtyping in two places. Firstly, it is incorrect to specialise method arguments in an overriding method. This violates the contravariant requirement for argument-types, which asserts that these can only be more general in a subtype method [1]. Secondly, the specialisation of attributes only works for attribute access. At some point, an attribute must also be initialised by assignment. The assignment operation may formally be considered to have the type signature: assign : $\tau \rightarrow$ void, where τ is the type of the value being assigned to the attribute. It is clear that

contravariance is violated if a more specialised value of type σ is assigned: assign : $\sigma \rightarrow$ void. Figure 2 shows how breaking contravariance may potentially lead to a runtime crash (this example uses Eiffel 5.0 syntax):

```
class POINT
create  make                    -- declare initialiser-method for a point
feature {ANY}                   -- public read-only access to attributes and methods
        x, y : INTEGER;         -- coordinates of a point, initially 0 by default
        make(nx, ny : INTEGER) is do x := nx; y := ny end;
        equal(other : POINT) : BOOLEAN is
                do Result := (x = other.x and y = other.y) end
end -- POINT

class HOTPOINT inherit POINT
        redefine equal          -- with unsafe covariant argument specialisation
feature {ANY}
        on : BOOLEAN;           -- currently selected, initially false by default
        toggle is do on := not on end;
        equal(other : HOTPOINT) : BOOLEAN is
                do Result  := (x = other.x and y = other.y and on = other.on) end
end -- HOTPOINT

genpt, point : POINT;           -- declarations
hotpt : HOTPOINT;
same : BOOLEAN;

create point.make(3, 5);        -- create standard point at (3, 5)
create hotpt.make(3, 5);        -- create hotpt at (3, 5) with on = false by default
genpt := hotpt;                 -- alias hotpt through genpt variable
same := genpt.equal(point);     -- invoke hotpt's equal, with only a point arg!!
```

Figure 2: Eiffel Covariant Type Failure Example

The salient issue is that *equal : POINT* $\rightarrow$ *BOOLEAN* is replaced by *equal : HOTPOINT* $\rightarrow$ *BOOLEAN*, incorrectly specialising the argument type. When the replacement method is invoked through the general variable *genpt : POINT*, statically it appears to be safe to supply *point : POINT* as its argument. However, when *HOTPOINT*'s *equal* method executes by dynamic binding, it tries to access the non-existent *on* attribute of this plain POINT argument. If unchecked, this will cause a memory segmentation fault.

Eiffel's designer eventually decided to fix both of these problems in a non-standard way [11]. Rather than change the type rules to obey strict subtyping, covariant argument-type redefinitions are flagged, such that unsafe combinations of aliasing and polymorphic invocation are detected immediately. The same technique is used to trap the polymorphic invocation of methods which have been suspended in descendent classes (both are known as polymorphic CAT-calls, standing for *change* in *availability* or *type*). Technically, this solution works, although mathematically it does not address the basic soundess issue.

By contrast, Eiffel's semantic redefintion rules follow *exactly* the semantic subtyping requirements as stated in section 2 above. This means there is an inconsistency between Eiffel's syntactic and semantic redefinition rules: the conflict is over redefined method arguments, between the precondition weakening rule and the (incorrect) argument specialisation rule. One accepts more, the other less. Paradoxically, the correct semantic rules were derived by thinking about contracts between client and supplier objects. The language Trellis [4] applies the same thinking to its syntactic type rules, in which redefined method arguments may only become more general. Trellis is the only language in our survey whose syntactic type rules follow exactly the subtyping requirements in section 2.

A different type-related issue is raised by Eiffel's *selective export* mechanism. A class may export separate lists of features to ANY (public visibility), to NONE (protected visibility) or to an arbitrary set of client classes. This has the curious consequence that the public interface of a class may appear to change, depending on who its client is! Types are no longer fixed, but are chameleon-like interfaces that change colour according to context. C++ raises similar issues with its *friend* declarations. Arbitrary functions, or whole classes, may be declared a *friend* of another class, in which case the friends have total freedom of access. So, the public interface of a class may appear different to its friends than to other clients. Friendship declarations are not inherited. So, aliasing an object through a superclass variable offering extra friendship privileges may break its intended encapsulation. Eiffel's selective exports are inherited unchanged, which is better.

6 CONCLUSION

The subtyping inquisition has bared the type systems of several popular object-oriented languages and found most of them guilty of violating subtyping in one way or another. Syntactically sound subtyping is exhibited by Trellis and Java, although Java is less flexible in its redefinition rule. C++ would be as good as Java, were it not for unchecked typecasting. Eiffel is retrospectively type-safe, due to the polymorphic CAT-call rule, even though it strictly violates soundness. None of the surveyed languages apart from Eiffel seriously promote verifying the behaviour of a class. Where full use is made of Eiffel's assertion mechanism, then a subclass may be shown to conform to the behaviour of its parent class. However, in all these languages, it is still possible to redefine methods to execute in arbitrary ways, resulting in unpredictable behaviour in substitutable components.

The fact that these faults do not give rise to system crashes more often than they do is explained mostly by the fact that programmers strive to write code in a consistent way, adopting style guidelines over and above what the type systems are capable of checking. It typically takes more than one unusual circumstance to trigger a type-related fault - for example, the Eiffel type-failure examples [3] were manufactured retrospectively by theoreticians, working backwards from the formal rules of subtyping. Up until that point, no system failures had been reported as being due to this particular

fault. In practice, you are less likely to want to compare a HOTPOINT with a more general POINT than you are with another object of the same type. So, how is it that subtyping cannot express this? Such will be the focus of the next article in the series.

REFERENCES

[1] A J H Simons. "The theory of classification, part 4: Object types and subtyping", *Journal of Object Technology, vol. 1, no. 5, November-December 2002*, pp. 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[2] A J H Simons. "The theory of classification, part 5: Axioms, assertions and subtyping", *Journal of Object Technology,vol. 2, no. 1, January-February 2003*, pp. 13-21. http://www.jot.fm/issues/issue_2003_01/column2

[3] W Cook. "A proposal for making Eiffel type safe", *Proc. 3rd European Conf. Object-Oriented Prog.*, 1989, 57-70; reprinted in *Computer Journal 32(4),* 1989, 305-311.

[4] C Schaffert, T Cooper, B Bullis, M Kilian and C Wilpolt. "An introduction to Trellis/Owl", *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub. *ACM Sigplan Notices, 21(11),* 1986, 9-16.

[5] W Cook and J Palsberg. "A denotational semantics of inheritance and its correctness", *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.,* pub. *ACM Sigplan Notices, 24(10),* 1989, 433-443.

[6] B Meyer. *Object-Oriented Software Construction*, 2nd edn., Prentice Hall, 1995.

[7] A Goldberg and D Robson. *Smalltalk-80: The Language and its Implementation,* Addison Wesley, 1983.

[8] B J Cox and A J Novobilski. *Object-Oriented Programming: an Evolutionary Approach,* 2nd edn., Addison Wesley, 1991.

[9] B Stroustrup. *The C++ Programming Language*, 3rd edn., Addison Wesley, 1997.

[10] C S Horstmann and G Cornell. *Core Java 2, Volume 1 - Fundamentals*, Sun Microsystems Press, 2003.

[11] B Meyer. "Beware polymorphic cat-calls", *Eiffel forum at 18th Conf. Tech. Object-Oriented Lang. and Sys. (TOOLS Pacific),* Melbourne, 1995; also available through: http://archive.eiffel.com/doc/manuals/technology/typing/cat.html.

About the author

Anthony Simons is a Senior Lecturer and Director of Teaching in the Department of Computer Science, University of Sheffield, where he leads object-oriented research in verification and testing, type theory and language design, development methods and precise notations. He can be reached at a.simons@dcs.shef.ac.uk.