JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, no. 5, November-December 2002

The Theory of Classification Part 4: Object Types and Subtyping

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

1 INTRODUCTION

This is the fourth article in a regular series on object-oriented type theory, aimed specifically at non-theoreticians. The "Theory of Classification" will explain the behaviour of languages such as Smalltalk, C++, Eiffel and Java in a consistent framework, modelling features such as classes, inheritance, polymorphism, message passing, method combination and templates or generic parameters. So far, we have covered the foundations of type theory and symbolic calculi. Along the way, we shall look at some further theoretical approaches, such as F-bounds and matching. In this article, we focus on the fundamental notion of subtyping.

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

Figure 1: Dimensions of Type Checking

Previous articles introduced the notion of type [1], motivated rules for constructing types [2] and compared different formal encodings for objects [3]. We consolidate here on the *functional closure* encoding of objects and object types that we intend to use in the rest of the series. Previously, we noted how component compatibility can be judged from different perspectives, according to *representation*, *interface* or full *behavioural* correspondence; and type checking may be *exact*, or according to more flexible schemes which we called *subtyping* and *subclassing* [1, 2]. This yields the nine possible combinations shown again in figure 1, of which the darker shaded areas interest us the most. In this fourth article, we shall construct syntactic subtyping rules (box 5 in figure 1) to determine when it is safe to substitute one object in place of another, where a different type was possibly expected.

2 OBJECTS AND OBJECT TYPES

In the following, we refer to *objects* and *object types*. Objects are modelled in the calculus as simple records, whose fields consist of labels which map to values [2, 3]. The values can be of any kind, for example, simple integer values, or functions (representing methods), or indeed other objects. In the model, each object contains all of its own methods - the calculus does not bother to represent indirection to a shared table of methods, which is merely an implementation strategy in object-oriented languages. The calculus only has to capture the notion of field selection, using the record selection "dot" operator, which then works for both attribute access and method invocation [2].

In the calculus, we define things using "name = expression" to associate names with equivalent values (or types, below). The names are simply convenient abbreviations for the associated definitions. Simple objects may be written directly in the calculus, for example an arbitrary instance may be defined as a literal record:

$$\text{aPerson} = \{\text{name} \mapsto \text{"John"};\ \text{surname} \mapsto \text{"Doe"};\ \text{age} \mapsto 25;\ \text{married} \mapsto \text{false}\}$$

The type of an object follows automatically from the types of the values used for its fields. A suitable corresponding type for the instance above may be given as:

$$\text{Person} = \{\text{name} : \text{String};\ \text{surname} : \text{String};\ \text{age} : \text{Integer};\ \text{married} : \text{Boolean}\}$$

Object types are modelled in the calculus as record types, that is, records whose fields consist of labels which map to types (note the use of ":" to indicate "has the type", rather than "↦" meaning "maps to the value"). We always use the term *object type* to denote the type of an object, to distinguish straightforward types from the more subtle notion of *class*, which potentially has other interpretations.

3 RECURSIVE OBJECTS AND OBJECT TYPES

Objects are frequently recursive, where they refer to their own methods, via *self*. Likewise, object types are recursive where the type signatures of their methods accept or return objects of the same type [3]. Since recursion is not built-in as a primitive notion, we must appeal to the fixpoint theorem to construct recursive objects and types [3]. A conventional notation is used to denote an object that has been recursively constructed:

$$\text{aPoint} = \mu\text{self} . \{x \mapsto 2; y \mapsto 3; \text{equal} \mapsto \lambda p.(p.x = \text{self}.x \wedge p.y = \text{self}.y)\}$$

in which μ*self* indicates that *self* is recursively bound in the record after the dot. This is really a short-hand for defining an *object generator*, a function of *self* (note the difference: λ*self*):

$$\text{genAPoint} = \lambda\text{self} . \{x \mapsto 2; y \mapsto 3; \text{equal} \mapsto \lambda p.(p.x = \text{self}.x \wedge p.y = \text{self}.y)\}$$

and then constructing aPoint from this generator, using the fixpoint combinator **Y** (see [3]):

$$\text{aPoint} = (\mathbf{Y}\ \text{genAPoint}) \Rightarrow \text{genAPoint}(\text{genAPoint}(\text{genAPoint}(...)))$$

Since it is inconvenient to have to keep on appealing to this construction in every recursive definition, we use the μ-convention to denote recursive definitions directly. Recursive object types may also be denoted using this convention. A suitable corresponding type for the instance above is given as:

$$\text{Point} = \mu\sigma . \{x : \text{Integer}; y : \text{Integer}; \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

in which μσ indicates that σ, the self-type of points, is recursively bound in the record type after the dot. Note in passing how the type of equal is a function type (arrow type) and that equal accepts an argument having the same self-type, which accounts for Point being a recursive type. In type definitions, the μ-convention is a short-hand for the whole rigmarole of defining a *type generator*, a type function of σ (σ is the function's type parameter):

$$\text{GenPoint} = \lambda\sigma . \{x : \text{Integer}; y : \text{Integer}; \text{equal} : \sigma \rightarrow \text{Boolean}\}$$

and then constructing the recursive Point type using **Y** (in a similar manner [3]):

$$\text{Point} = [\mathbf{Y}\ \text{GenPoint}] \Rightarrow \text{GenPoint}[\text{GenPoint}[\text{GenPoint}[...]]]$$

In the rest of this article, we will use the μ-convention throughout for recursive objects and recursive types. In later articles, object generators and type generators will acquire a new importance in the definition of the notion of *class*. It is theoretically sound to use **Y** to "fix" both recursive objects and recursive types in the same universe, provided that records contain fields of functions and do not refer directly to themselves [4].

4 SIMPLE SUBTYPING

Object-oriented languages take the ambitious view that, in the universe of types, all types are *systematically related* in a type hierarchy. Types lower in the hierarchy are somehow compatible with more general types higher in the hierarchy. By contrast, older languages like Pascal treated every type as distinct and unrelated[1] The more flexible object-oriented notion of type compatibility is based on a safe substitution property. Liskov is frequently cited as the source of this idea [5], popularly known as the *Liskov Substitutability Principle* (LSP), although Cardelli and Wegner [6] deserve shared credit:

> "What is wanted here is something like the following substitution property: if for each object o_1 of type S there is an object o_2 of type T such that, for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." [5]

These authors all identify substitutability with subtyping. In fact, subtyping is just one formal approach which satisfies the principle of safe substitution - there are other more subtle and equally formal approaches, which we shall consider in later articles.

Henceforth, we shall use "<:" to mean "is a subtype of". The notion of subtyping derives ultimately from subsets in set theory. In exactly the same way that: $x : X$ ("x is of type X") can be interpreted as $x \in X$ ("x is a member of the set X") in the model, so the subtyping relationship, for example: $Y <: X$ ("Y is a subtype of X") can be interpreted consistently as the subset relationship $Y \subseteq X$ ("Y is a subset of X") in the model. Objects may belong to a hierarchy of increasingly more general types: consider that if $y : Y$ and $Y <: X$ holds, then $y : X$ is also true. The set-theoretic model supports this directly: If $y \in Y$ and $Y \subseteq X$, then it follows that $y \in X$, by the definition of subsets:

$$Y \subseteq X ::= \forall y \,.\, y \in Y \Rightarrow y \in X \qquad \text{[Definition of a subset]}$$

While this translation works immediately for primitive types, the object substitutability principle is couched in terms of *equivalent behaviour* after a substitution. The behaviour of an object is characterised by its methods; so we are obliged to consider the type-relationships between all corresponding methods in the original and substitute objects.

5 FUNCTION SUBTYPING

For a substitute object $y : Y$ to behave exactly like the original object $x : X$, then every method of X must have a corresponding method in Y that behaves like the original method of X. If we are only interested in *syntactic* compatibility [1] (that is, between interfaces), this reduces to checking the type signatures of related pairs of methods. In many object-oriented languages, the correspondence between the methods of X and Y is at least partly assured by defining Y as an extension of X - in this case, Y may inherit the majority of X's methods unchanged. However, we must cater for the general case, in which Y substitutes different methods in place of those in X (known as method *redefinition*, or *overriding*).

Consider a method f of X, which we shall call $f_X : D_X \rightarrow C_X$, to indicate that it is a function accepting an argument of some type D and yielding a result of some other type C. This is to be replaced by a substitute method f of Y, which we shall call $f_Y : D_Y \rightarrow C_Y$, with the intention that Y should still behave like X. Under what conditions can f_Y be safely substituted in place of f_X? From the syntactic point of view, there are two obligations:

- f_Y must be able to handle at least as many argument values as f_X could accept; we express this as a constraint on the domains (argument types): $D_Y \supseteq D_X$; and

- f_Y must deliver a result that contains no more values than the result of f_X expected; we express this as a constraint on the codomains (result types): $C_Y \subseteq C_X$.

A helpful way to think about these obligations is to consider how a program might fail if they were broken. What if f_Y accepted fewer argument values than f_X? In this case, there might be some valid arguments supplied to f_X in the original working program that were not recognised by f_Y after the substitution, causing a run-time exception. What if f_Y delivered more result values than f_X? In this case, the call-site expecting the result of f_X might receive a value that was outside the specified range when f_Y was invoked instead.

From this, we can motivate the *function subtyping* rule, which expresses under what conditions a function type $D_Y \rightarrow C_Y$ is a subtype of another function type $D_X \rightarrow C_X$. This uses the same style of natural deduction rule as before [2]:

$$\frac{D_X <: D_Y,\ \ C_Y <: C_X}{D_Y \rightarrow C_Y\ <:\ D_X \rightarrow C_X} \qquad \text{[Function Subtype]}$$

"If the domain (argument type) D_Y is larger than the domain D_X, and the codomain (result type) C_Y is smaller than the codomain C_X, then the function type $D_Y \rightarrow C_Y$ is a subtype of the function type $D_X \rightarrow C_X$." Note how there is an assymmetry in this rule: in the subtype function, the codomain is also a subtype, but the domain is a supertype. For this reason, we sometimes say that the domains are *contravariant* (they are ordered in the opposite direction) and the codomains are *covariant* (they are ordered in the same direction) with respect to the subtyping relationship between the functions.

The function subtyping rule is expressed formally using a single argument and result type. To extend this to methods accepting more than one argument, we recall the fact that a single type variable in one rule may be deemed equivalent to a product type in another rule [2]. In this case, the contravariant constraint applies to the two products as a whole, and so we need a *product subtyping* rule to break this down further:

$$\frac{S_1 <: T_1,\ \ S_2 <: T_2}{S_1 \times S_2\ <:\ T_1 \times T_2} \qquad \text{[Product Subtype]}$$

"The product type $S_1 \times S_2$ is a subtype of the product type $T_1 \times T_2$, if the corresponding types in the products are also in subtyping relationships: $S_i <: T_i$." The consequence for function subtyping is that, in a multi-argument function, all of the overriding (subtype) function's arguments must be supertypes of those in the function they replace.

6 RECORD SUBTYPING

The last piece of the subtyping jigsaw is to determine under what conditions a whole object type Y is a subtype of another object type X. Recall that object types are basically record types, whose fields are function types, representing the type signatures of the object's methods. According to Cardelli and Wegner [6], one record type is a subtype of another if it can be coerced to the other. For example, a Person type with the fields:

Person = {name : String; surname : String; age : Integer; married : Boolean}

might be considered a subtype of a DatedThing type having fewer fields:

DatedThing = {name : String; age : Integer}

because a Person can always be coerced to a DatedThing by "forgetting" the extra surname and married fields. Intuitively, this also satisfies the LSP, since a Person instance may always be used where a DatedThing was expected; any method invoked through a DatedThing variable will also exist in a Person object. This motivates the first part of the record subtyping rule (*record extension*):

$$\frac{a_1, \ldots a_k, \ldots a_n : A}{\{a_1 : T_1, \ldots a_n : T_n\} <: \{a_1 : T_1, \ldots a_k : T_k\}} \quad \textit{for}\ 1 \le k \le n \qquad \text{[Record Extension]}$$

"If there are distinct labels a_i, then a longer record constructed with n fields, having the types $a_i : T_i$, is a subtype of a shorter record constructed with only the first k fields, provided that the common fields all have the same types." This rule basically asserts that adding to the fields of a record creates a subtype record.

Defining new types of object by extension is clearly a common practice in object-oriented programming languages. However, there is also the possibility of overriding some methods in the extended object. What requirement should we place on a record type if *all* of its fields were to change their types? According to the reasoning above which led to the function subtyping rule, any replacement methods should be subtypes of the originals. We can confirm this using a simpler example of field-type redefinition. If PositiveInteger <: Integer, then

PositiveCoordinate = {x : PositiveInteger; y : PositiveInteger}

is intuitively a subtype of the more general:

Coordinate = {x : Integer; y : Integer} because the set of all PositiveCoordinates is contained within the set of all Coordinates (the positive subset occupies the upper right quadrant of the Cartesian plane). This motivates the second part of the record subtyping rule (*record overriding*):

$$\frac{a_i : A,\ S_i <: T_i}{\{a_i : S_i\} <: \{a_i : T_i\}}\ \textit{for}\ i = 1..n \qquad \text{[Record Overriding]}$$

"A record that has n labelled fields of the types $a_i : S_i$ is a subtype of a record having the same labelled fields of the different types $a_i : T_i$, provided that each type S_i is a subtype of the corresponding type T_i." This rule uses the index i to link the corresponding labels, types and subtypes appropriately. Combining the record extension rule with the overriding rule gives the complete, but more complicated looking, *record subtyping* rule:

$$\frac{a_i : A,\ \ S_1 <: T_1, \ldots\ S_k <: T_k}{\{a_1 : S_1, \ldots\ a_n : S_n\} <: \{a_1 : T_1, \ldots\ a_k : T_k\}}\ \textit{for}\ 1 \leq k \leq n \qquad \text{[Record Subtyping]}$$

"A longer record type $\{a_i : S_i\}$ with n fields is a subtype of a shorter record type $\{a_i : T_i\}$ with only k fields, provided that, in the first k fields that they have in common, every type S_i is a subtype of the corresponding type T_i." The record subtyping rule generalises the previous two rules. If $S_i = T_i$, for i = 1..k, then it reduces to the record extension rule. If k = n, then it reduces to the record overriding rule.

To complete the picture, we must say a little about recursion and subtyping. The subtyping rules given above depend on having complete type information about all the elements that make up the types. One of these elements could be the self-type, in a recursive type. We cannot make definite assertions about subtyping between recursive types, unless we first make some assumptions about the corresponding self-types. Cardelli [7] expresses this (here slightly simplified, ignoring the context) as the rule:

$$\frac{\sigma <: \tau\ \vdash\ S <: T}{\mu\sigma.S\ <:\ \mu\tau.T}\ \sigma \text{ free only in S, } \tau \text{ free only in T} \qquad \text{[Recursive Subtype]}$$

"If assuming that σ is a subtype of τ allows you to derive that S is a subtype of T, then the recursive type $\mu\sigma.S$ is a subtype of the recursive type $\mu\tau.T$, where S may contain occurrences of the self-type σ and T may contain occurrences of the self-type τ." This rule sets up the relationship between the self-type variables and the types which depend on them. In a later article, we shall revisit the interactions between recursion and subtyping, which proves to be quite a thorny problem for object-oriented type systems.

7 CONCLUSION

We have reconstructed the classic set of subtyping rules for object types, including rules for recursive types. These rules have the following impact on object-oriented languages that wish to preserve subtyping relationships. A class or interface name may be understood as an abbreviation for the type of an object, where the type is expressed in full as the set of method type signatures owned by the class (or interface). A subclass or derived interface may be understood as a subtype, if it obeys the rule of record subtyping. In particular, the subtype may add (but not remove) methods, and it may replace methods with subtype methods.

A method is a valid replacement for another if it obeys the function subtyping rule. In particular, the subtype method's arguments must be of the same, or more general (but not more restricted) types; and its result must be of the same, or a more restricted (but not more general) type. Very few languages obey both the covariant and contravariant parts of the function subtyping rule (Trellis [8] is one example). Languages such as Java and C++ are less flexible than these rules allow, in that they require replacement methods to have exactly the same types. Partly, this is due to interactions with other rules for resolving name overloading; but also it reflects a certain weakness in the type systems of languages based on subtyping, which we will start to explore in the next article.

REFERENCES

[1] A J H Simons, *The theory of classification, part 1: Perspectives on type compatibility*, in Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 55-61. http://www.jot.fm/issues/issue_2002_05/column5.

[2] A J H Simons, *The theory of classification, part 2: The scratch-built typechecker*, in Journal of Object Technology, vol. 1, no. 2, July-August 2002, pages 47-54. http://www.jot.fm/issues/issue_2002_07/column4.

[3] A J H Simons, *The theory of classification, part 3: Object encodings and recursion*, in Journal of Object Technology, vol. 1, no. 4, September-October 2002, pages 49-57. http://www.jot.fm/issues/issue_2002_09/column4

[4] K Bruce and J Mitchell, PER models of subtyping, recursive types and higher-order polymorphism, *Proc. 19th ACM Symp. Principles of Prog. Langs.,* (1992), 316-327.

[5] B Liskov, Data abstraction and hierarchy, *ACM Sigplan Notices, 23(5),* (1988), 17-34.

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

[7] L Cardelli, Amber, *Combinators and Functional Programming Languages, LNCS, 242* (1986), 21-47.

[8] 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.

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

[1] Apart from subrange types in Pascal, which were considered compatible with their base types.