JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 2, No. 1, January-February 2003

The Theory of Classification Part 5: Axioms, Assertions and Subtyping

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

1 INTRODUCTION

This is the fifth 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 subtyping from the *syntactic* point of view, that is, judging type compatibility by the type signatures of an object's methods. In terms of the dimensions of type checking in figure 1, we have considered *exact* type correspondence (box 2, see earlier article [1]) and *subtyping*, a more flexible kind of type correspondence (box 5, see previous article [2]). This allows us to determine whether a given type provides enough operations to satisfy a given interface and whether the supplied operations have suitable type signatures.

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

Figure 1: Dimensions of Type Checking

However, component compatibility is not just a matter of observing the conventions on type signatures. An object could offer all the expected operations, but still execute in a completely perverse way (see earlier article [3]). It is equally important to know whether a component *behaves* in the way expected by the program in which it is used. For this, an approach is required which can model the *semantics* of object types and capture precisely how they execute. Semiformal methods for capturing behaviour include statecharts [4] and various assertion languages, such as OCL [5]. A means of incorporating assertions

Cite this column as follows: Anthony J.H. Simons: *The Theory of Classification, Part 5: Axioms, Assertions and Suptyping,* in Journal of Object Technology, vol. 2, no. 1, January-February 2003, pages 13-21. http://www.jot.fm/issues/issue_2003_01/column2

[6] into a practical programming language was first introduced by Eiffel [7]. This expresses the meaning of operations in terms of the *preconditions* and *postconditions* that they satisfy, together with *invariants* characterising the unchanging properties of object types.

All of these approaches are incomplete realisations of the more fundamental *algebraic* approach to defining the meaning of abstract datatypes. In this article, we consider the exact specification of a type's *behaviour* (box 3 in figure 1) and also the relationship between algebraic specification and subtyping, which will allow us to prove when one object *behaves in a subtype-conformant way* to another (box 6 in figure 1).

2 INITIAL AND FINAL ALGEBRAS

Mathematicians have been experimenting with notions of abstract types and classification since the early part of the 20th century. Much of this work falls within the remit of *formal algebra* and *category theory*. An algebra is an abstract type definition, consisting of a set of elements (a *sort*) and a collection of operations acting on the set, characterised by their type signatures and logical axioms (see earlier article [3]). Some algebras are related to each other, in that they have the same operations, but the properties of the operations may vary slightly from algebra to algebra. As an example, consider that the List type, with *cons, head* and *tail* can be mapped onto an abstract Stack type with *push, top* and *pop*. Such a mapping relationship is called a *homomorphism* (literally, "same form"), and where an inverse mapping also exists, this is called an *isomorphism* ("identical form").

In the universe of algebras, families of algebras exist in which elements and operations that are distinct in one algebra become merged and indistinguishable in other algebras. Consider that if "+" is an abstract operation with just the property of *associativity* (for example, like "+" used to append Strings in Java), then "a + b" and "b + a" will mean different things. However, if in another algebra, "+" also has the property of *commutativity* (for example, like "+" used to add Integers), then "a + b" and "b + a" will mean the same thing. In the universe of algebras, homomorphisms are arrows running from the algebras with more distinguishable elements to the algebras with fewer distinguishable elements. At one end of this universe, an algebra called the *initial algebra* exists[1], whose elements are more distinguishable than in any other. At the opposite end, a *final algebra* exists, whose elements are the least distinguishable.

[1] Technically, an *initial algebra* is one from which a unique homomorphism maps to every other algebra. Similarly, a *final algebra* is one into which a unique homomorphism maps from every other algebra. By this definition, initial and final algebras may not exist in certain semantic domains.

3 AXIOMATIC SEMANTICS

When defining an algebraic type, the first concern is to establish under what semantics the axioms of the algebra will be interpreted. The Ordinal type[2] from an earlier article [3] was defined using *final algebra semantics*, in which we assume that all elements of the type are equivalent, unless we can prove them to be distinct:

$$
\begin{array}{lll}
\text{Ordinal} = \exists\ \text{ord} \,.\, \{\text{first:} \rightarrow \text{ord};\ \ \text{succ: ord} \rightarrow \text{ord}\} & & \\
\forall x, y : \text{Ordinal} \,. & & \\
\qquad & \text{succ}(x) \neq \text{first}() & (1) \\
\wedge \qquad & \text{succ}(x) \neq x & (2) \\
\wedge \qquad & \text{succ}(x) = \text{succ}(y) \Leftrightarrow x = y & (3)
\end{array}
$$

The onus is on showing that the first() element is distinct from any successor succ(x), that any element x is distinct from its immediate successor succ(x), and that by induction all the elements of the type are eventually distinct, such that Ordinal is inhabited by a series of monotonically increasing elements: first(), succ(first()), succ(succ(first())) ...

We change our approach to define the behaviour of object types. The Stack algebra below is defined using *initial algebra semantics*, in which we assume that all the elements of the algebra are distinct, unless we can prove them to be equal. Note in particular how axiom (6) asserts when two Stacks can be judged to be equivalent; and how axiom (5) asserts under what conditions the element you retrieve is equivalent to the one you previously inserted:

$$
\begin{array}{lll}
\text{Stack} = \forall T.\ \mu stk.\{\text{push} : T \rightarrow \text{stk};\ \ \text{pop} : \rightarrow \text{stk};\ \ \text{top} : \rightarrow T;\ \ \text{empty} : \rightarrow \text{Boolean}; & & \\
\qquad \text{size} : \rightarrow \text{Natural}\};\ \ \text{newStack} : \rightarrow \text{Stack}; & & \\
\forall e : T \,.\, \forall s : \text{Stack} \,. & & \\
\qquad & \text{newStack}().\text{empty}() & (1) \\
\wedge \qquad & \neg s.\text{push}(e).\text{empty}() & (2) \\
\wedge \qquad & \text{newStack}().\text{size}() = 0 & (3) \\
\wedge \qquad & s.\text{push}(e).\text{size}() = 1 + s.\text{size}() & (4) \\
\wedge \qquad & s.\text{push}(e).\text{top}() = e & (5) \\
\wedge \qquad & s.\text{push}(e).\text{pop}() = s & (6)
\end{array}
$$

In the signature of Stack's operations, the use of ∀T "for all types T" indicates a *generic* Stack definition, since T is later used as the element-type. μ *stk* indicates that Stack is a recursive record type in which *stk* refers to the eventual Stack type. All Stack operations

[2] I am indebted to Kim Bruce for pointing out that the third Ordinal axiom is required to allow the rule of induction to operate as intended. Originally, I missed this.

are defined as methods that accept and return elements of these and other types (which we assume are defined in associated algebras). The initial constructor for a Stack is not a member of the record, since a Stack instance does not create itself. The axioms which define the meaning of Stack's operations are equations which refer to arbitrary stacks s : Stack and elements e : T. Each axiom asserts a boolean expression which holds true for the type. Axioms are combined using "∧" logical *and*. All other properties of Stacks can be inferred from these axioms.

4 INDUCTIVE DEFINITIONS

The strategy for defining the behaviour of an object type uses an inductive approach, similar to recursive function definition in a functional programming language. In the axioms defining the meaning of *size*, note how there is a base case (3) for new Stacks, and a step case (4) for arbitrary Stacks *s*. The step case always defines a property in terms of something simpler that is closer to the base case (a *recurrence relationship*). Thereafter, the size of any Stack can be derived using repeated application of these rules.

The equations always relate pairs of methods on the left-hand side and assert that a nested invocation of these methods is equivalent to something else on the right-hand side (think of Stack axioms (1) and (2) as being "equivalent to true" on the right-hand side). How do you decide which pairs of methods to relate; and how do you know when sufficient axioms have been defined? If too many axioms are supplied, a theorem prover might waste time exploring redundant solutions that could be derived from other axioms. To help with this problem, the functions of the type are sometimes divided into three groups:

- constructors - the smallest set of functions returning the type, which, taken together, can generate *every single instance* of the type;
- transformers - the remaining functions which return the type, but which are non-primitive in the sense that they can be defined in terms of primitive constructors;
- observers - functions returning something other than the type, typically because they inspect part of the type or compute some value from it.

Note that *constructors* mean more than the usual object-oriented sense of the word. In a pure functional calculus, pushing an element onto a Stack means creating a new Stack object onto which the extra element has been added. Accordingly, *new* and *push* are both algebraic constructors for the Stack. With these two operations, we can create every single possible Stack instance. Consequently, *pop* is a non-primitive transformer and its result can be defined in terms of other constructors. Likewise, *top, empty* and *size* are observers.

Thereafter, the maximum number of axioms to define is decided. You need no more than an axiom for *each constructor paired with every other non-constructor*. From this, you would expect to define at most $2 \times 4 = 8$ axioms for Stack. However, only 6 were supplied above; this means that in certain contexts, some of Stack's methods are undefined, an issue to which we shall return below.

5 DEDUCTIVE REASONING

Let us now assume that we wish to derive some property of Stacks. For example, what is the *size* of a Stack after a sequence of *push* and *pop* operations? The problem corresponds to simplifying a nested method expression, such as:

newStack().push(e1).push(e2).pop().size()

To simplify this, we look for axioms which relate suitable pairs of operations on their left-hand side, and substitute the corresponding equivalent expressions on their right-hand side. Working backwards from the end of the expression, and given ∀e : T, ∀s : Stack:

- There is no axiom for s.pop().size(); but this is not an omission, since pop is not a primtive constructor and we can derive its meaning elsewhere. Instead, we look further.

- There is an axiom (6) with s.push(e).pop() on the left-hand side, so if we make the substitutions: s ← newStack().push(e1) and e ← e2, then the corresponding right-hand side resubstitution is: s → newStack().push(e1), giving the simplification:

  newStack().push(e1).push(e2).pop().size() ⇒ newStack().push(e1).size() [by axiom 6].

- There is an axiom (4) with s.push(e).size() on the left-hand side, so if we make the substitutions: s ← newStack() and e ← e1, then the corresponding right-hand side resubstitution is: 1+s.size() → 1+newStack().size(), giving the simplification:

  newStack().push(e1).size() ⇒ 1+newStack().size() [by axiom 4].

- Finally, there is an axiom (3) giving newStack().size() directly:

  1+newStack().size() ⇒ 1+0 [by axiom 3].

and the final answer 1+0 = 1 is obtained using the algebra for Natural numbers in a similar fashion. This is the right answer and, hopefully, the one the reader expected!

6 ERRORS AND DEFERRED DEFINITIONS

The two axioms omitted from Stack's equations were those relating: newStack().top() and newStack().pop(). This is because the meanings of *top* and *pop* are deliberately left undefined for *new* Stacks. "Undefined" can be interpreted variously to signify that a method's meaning has *not yet* been fully specified, or that the method's result *is an error* in this context. In the case of *top* and *pop*, it is clear that these should always raise

exceptions with a new Stack. A function that is not defined in all contexts is known as a *partial function*.

There may be other valid reasons for "saying less about" a type than just to specify error cases. Consider first that a Queue type looks quite similar to a Stack, except that some of its functions behave in a slightly different way:

Queue = ∀T. μque.{push : T → que; pop : → que; top : → T
empty : → Boolean; size : → Natural}; newQueue : → Queue
∀e : T . ∀q : Queue .

	newQueue().empty()	(1)
∧	¬q.push(e).empty()	(2)
∧	newQueue().size() = 0	(3)
∧	q.push(e).size() = 1 + q.size()	(4)
∧	q.push(e).top() = e **if** q.empty()	(5a)
	= q.top() **otherwise**	(5b)
∧	q.push(e).pop() = q **if** q.empty()	(6a)
	= q.pop().push(e) **otherwise**	(6b)

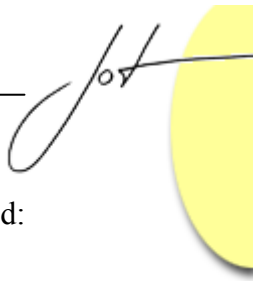
The differences are in axioms (5) and (6), which assert FIFO properties for a Queue, contrasting with the LIFO properties asserted above for a Stack. Queue's equations also demonstrate how a right-hand side can be split into several cases, using **if**-clauses. The reader should experiment with some examples, in the deductive style shown above, to see that the recurrence relation in axiom 6 causes elements to be added and removed in the right order. (The recurrence works by driving *pop* backwards, until it encounters a base case).

Given that Queue and Stack are *syntactically* identical (their methods have identical type signatures), it should be possible to create the type of an interface, or supertype, to which both Stack and Queue conform. Since both of these collections dispense their elements in a particular order, we shall call their abstract supertype *Dispenser*:

Dispenser = ∀T. μdsp.{push : T → dsp; pop : → dsp; top : → T;
empty : → Boolean; size : → Natural}; newDispenser : → Dispenser
∀e : T . ∀d : Dispenser .

	newDispenser().empty()	(1)
∧	¬d.push(e).empty()	(2)
∧	newDispenser().size() = 0	(3)
∧	d.push(e).size() = 1 + d.size()	(4)

It is clear that both Stack and Queue satisfy the above definition, since they have the identical signatures, and obey the identical axioms (1) - (4). The fact that axioms (5) and (6) are missing means that, at the level of generality described by *Dispenser*, we cannot



yet say anything about the order in which elements are inserted, accessed and removed: the specifications of *top* and *pop* are deferred. We say that Dispenser is *underspecified*.

7 UNDERSPECIFICATION AND SUBTYPING

The rules governing axioms and semantic subtyping follow from this. If a type is underspecified, then a subtype may be created by adding suitable axioms giving the missing meanings of the underspecified operations. Note that Stack and Queue define mutually exclusive axioms (5) and (6), such that one could never be a subtype of the other; yet both are semantic subtypes of Dispenser, since they only add to Dispenser's existing axioms and do not violate any of them.

Consider now that, in just one case, Stacks and Queues actually do behave identically in regard to *push*, *pop* and *top* - this is when they contain a single element. We could add this common information to the Dispenser type by writing *partial* axioms:

$\wedge$ d.push(e).top() = e **if** d.empty() (5a)
$\wedge$ d.push(e).pop() = d **if** d.empty() (6a)

These two axioms express, for both Stacks and Queues, that if you *push* an element into an empty container, this is the *top* element, and the one that is removed by *pop*. From this, it is clear that some notion of *axiom refinement* must happen in subtypes. In the case of Queue, we simply *complete* the partial axioms by adding parts (5b) and (6b). In the case of Stack, we drop the **if**-condition instead, such that Stack's axioms (5) and (6) cover more than the 1-element case. Dispenser's partial axioms are therefore still satisfied by both Stack and Queue. If we refine an axiom in a subtype, the subtyping condition is this: the refined axiom *must logically entail* the original axiom.

Finally, we can show a relationship between semantic and syntactic subtyping. Why does adding axioms, or strengthening axioms to cover more cases, create a subtype? Consider defining a set S by comprehension in relation to a set T, such that S contains all those elements in T which satisfy the extra axiom p(x):

$$S = \{\forall x \in T \mid p(x)\}$$

It is clear that, if all elements of T pass the test p(x), then S = T. However, if some elements of T fail the test, then $S \subset T$. Therefore, we can assert that: $S \subseteq T$, and this also means that S is a subtype of T [2].

8 CONCLUSION

We have developed an algebraic calculus for reasoning about the *complete* behaviour of object types, and demonstrated the effects of axioms upon subtyping. When seeking to apply the results of this analysis to assertion languages like OCL [5] and object-oriented languages like Eiffel [7] we have to translate from pure algebra into a piecemeal treatment in terms of invariants, pre- and postconditions. It is useful to think in terms of *strengthening* assertions:

- strengthening an *invariant* is identical to strengthening the axioms of an algebra, since the invariant applies constantly to the object type as a whole;
- strengthening a *method postcondition* corresponds either to strengthening the axioms on the result-type of the method, or strengthening the axioms on the object type itself; or possibly to both of these;
- strengthening a *method precondition* corresponds either to strengthening the axioms on the argument-types of the method, or strengthening the axioms on the object type itself; or possibly to both of these.

Since there is a direct relationship between axiom strengthening and subtyping, we can immediately apply our existing object subtyping rules [2] to derive subtyping rules governing the strengthening, or weakening of assertions. Recall that an object type which adds to the methods of another object type is a subtype. The subtype will define the semantics of the extra methods, providing more axioms, which is consistent with being a subtype. Some of these extra axioms may appear as strengthened invariants, which is also consistent. Apart from this, an object type may sometimes replace methods, so we must consider under what conditions this results in a subtype.

A method is a valid replacement for another if it obeys the function subtyping rule [2]. 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. Translating this into assertions, the method's preconditions may *possibly be weakened* (but never strengthened) and the method's postconditions may *possibly be strengthened* (but never weakened). In this regard, assertions also follow the contravariant argument-type and covariant result-type rules. Eiffel [7] obeys similar rules regarding the weakening of its preconditions and strengthening of its postconditions. The only area of conflict is where a precondition also affects the object type itself. To satisfy the contravariant rule, the precondition can only be weaker, but to satisfy object subtyping, the invariant can only be stronger. In practice, weaker preconditions can co-exist with stronger invariants, since the same object must satisfy the stronger of the two and doesn't care that the method would accept something less strict than itself.

REFERENCES

[1] A. J. H. Simons: *The Theory of Classification, Part 2: The Scratch-Built Typechecker*, Journal of Object Technology, vol. 1, no. 2, July-August 2002, pages 47-54. http://www.jot.fm/issues/issue_2002_07/column4

[2] 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, pages 27-35. http://www.jot.fm/issues/issue_2002_11/column2

[3] A. J. H. Simons, *The Theory of Classification, Part 1: Perspectives on Type Compatibility*, Journal of Object Technology, vol. 1, no. 1, May-June 2002, pages 55-61. http://www.jot.fm/issues/issue_2002_05/column5

[4] D. Harel and A. Naamad, The STATEMATE semantics of statecharts, *ACM Trans. Soft. Eng. Method., 5(4),* 1996, pages 293-333.

[5] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, (Reading MA : Addison Wesley, 1999).

[6] C A R Hoare, Proof of correctness of data representations, *Acta Informatica, 1,* 1972, pages 271-281.

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

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