Completeness in ADTs

In logical systems, the idea of *completeness* refers to the ability to prove all true statements. In ADT specifications this translates to proving all valid identities between values in the TOI(s). One way to ask about the adequacy of an ADT in this regard is by the question — when have we written enough equational axioms? Of course, this question can only be answered from an intuitive perspective since the answer depends on what we "intend" to specify. In ADTs a somewhat weaker property turns out to be technically helpful.

Definition: if an ADT specification <□,e> with pre-defined types $T_1, T_2, \ldots, T_n$ has the property that for every term t of the term algebra T(□,∅) whose sort is T_i $(1 \le i \le n)$, there is another term t' so that t ≡t' (or t □t' for final algebra view) and t' involves only the operations of T_i, then the specification is **sufficiently complete**.

So for instance in our example of the Stack of natural numbers, the sort of TOP(PUSH(s,i)) is the pre-defined type natural number, and to enjoy sufficient completeness, the specification must provide a means to eliminate the Stack operations that appear in this term — that is, we want to be able to determine the actual "natural number" that the signature promises. This is especially crucial in the final algebra view where objects are differentiated by the pre-defined results that can be "extracted" from them. Also, to ensure the no junk axiom, it is key in the initial algebra view.

Consistency in ADTs

In logical systems the idea of *consistency* refers to the impossibility of proving a false statement. We seek this same property in the ADT specifications we write, and we need to be aware of how things might go awry. In ADT specifications, the manifestation of consistency varies somewhat with the semantic viewpoint that we adopt.

Consistency from the Final Algebra Viewpoint

If there are no pre-defined types, then there is no option for distinguishing objects of the TOI, and so the final algebra view yields a trivial one element domain. In this case consistency is assured, but the specification can involve only trivial behavior. Hence in the final algebra view, there will invariably be one or more pre-defined types. In fact, in the final algebra view any ADT specification can be built up starting from the Boolean pre-defined ADT. If we assume Boolean is a pre-defined type, we say that an ADT is **inconsistent** if True = False may be proven. In practice, we may consider elements a and b known to be different in any pre-defined type, and if it can be proven that a = b, then the ADT is inconsistent. We shall see a little later how this may happen all too easily.

Consistency from the Loose Semantics Viewpoint

In the loose semantics point of view, an ADT specification is **consistent** if every equation t = t' which can be deduced is true in all the models of the ADT (recall that a *model* is an algebraic system with the same signature as the ADT that satisfies all the equations of the ADT).

Consistency from the Initial Algebra Viewpoint

Technically, inconsistency is impossible in the strict view of the initial algebra interpretation since two terms are equivalent only if they can be proven so. However, this requires that we disregard the hierarchical strategy of building up specifications from pre-defined types, a step we are unwilling to take. In practice the situation is not really different from the final algebra view. An initial algebra ADT is **inconsistent** if for some t of a pre-defined type, t ≡ t' can be deduced in the TOI when in the pre-defined type t ≠ t'. While the initial algebra view does not always necessitate pre-defined types, we usually find them crucial. If there is no pre-defined type, an initial algebra ADT is certain to be consistent. But if there are pre-defined types, we must be concerned with whether there are "collapsing" types.

Example of Inconsistency

Suppose we assume the natural numbers, Nat, with the usual operations (including =) as pre-defined, and we wish to provide an ADT specification of the (positive) rational numbers, Rat. We regard rationals as pairs in integers, and propose operations (among others):

num: Rat □ Nat	to yield the numerator of a Rational
den: Rat □ Nat	to yield the denominator of a Rational
make: Nat □ Nat □ Rat	to yield Rational with given numerator/denominator

We then introduce equations

num(make(n,d)) = n
den(make(n,d)) = d

and since for rational numbers $\frac{a}{b} = \frac{c}{d}$ if a*d = b*c, we include the equation

r = **if** num(r)*den(s) = num(s)*den(r) **then** s **else** r

But then we can conclude that make(0,1) = make(0,2) and hence
1 = den(make(0,1)) = den(make(0,2)) = 2 !?