Order-sorted ADTs

The idea of sort ordering is to allow the specification to prescribe a subset relation between types of data. For example, in ordinary mathematics Integer < Rational for these two kinds of numbers since every Integer is a Rational number. In the computer programming context, this addition allows a specification to describe inheritance behavior (operations defined on a superset are defined on the subset) and expands ADT expressiveness into the object-oriented domain. It turns out that it also provides a means to deal clearly and succinctly with exceptional behavior specification, avoiding numerous extra equations.

Definition: an **order-sorted ADT** is an algebraic ADT augmented with a p*artial order*, $\leq$, on the collection of sorts. The partial order on sorts provides a *subsort order* that imposes a restriction on the algebras that serve as models. Namely, for sorts s_1 and s_2, if $s_1 \leq s_2$, an algebra $\mathbf{A}$ is a model of the ADT only if all equations are satisfied *and* $\mathbf{A}_{s1} \subseteq \mathbf{A}_{s2}$, where $\mathbf{A}_s$ is the collection of all values in $\mathbf{A}$ of sort s (the carrier of s). The subsort order is given by a collection of declarations, $s_i \leq s_j$, that is understood to determine a partial order (i.e., $s_i \leq s_j \wedge s_j \leq s_k \Rightarrow s_i \leq s_k$).

To form the term algebra, $T(\Sigma,\varnothing)$, wherever a term of sort s_2 is allowed, any term of sort s_1 may appear, if $s_1 \leq s_2$. We also allow operation "overloading", that is the same function name is permitted to have more than one signature. However, we require the **monotonicity condition**, that if $\sigma\colon w_1 \rightarrow s_1$, $\sigma\colon w_2 \rightarrow s_2$, and $w_1 \leq w_2$, then $s_1 \leq s_2$. The monotonicity condition reflects the assumption that the function $\sigma\colon w_2 \rightarrow s_2$ "extends" the function $\sigma\colon w_1 \rightarrow s_1$ — that is, on the common elements of sort w_1, these are the same functions. For instance, with Int<Rat, +: Int, Int $\rightarrow$ Int and +: Rat, Rat $\rightarrow$ Rat are the same when applied to integers.

An object of an order-sorted ADT may have several sorts (types) since sets of type values are allowed to have a nesting relationship. In the case of number relationships, a value can be both a natural number and an (signed) integer, or both an integer and a rational number (or all three). An operation defined for one sort is therefore expected to be defined for subsorts. For instance, division is defined for rational numbers, and therefore it is defined for integers (but yields a rational).

In the order-sorted case, we never have a single TOI — as long as there are subsorts, we will have multiple TOIs. For each sort s, we will have equivalence between objects of that sort, $\equiv_s$. This is an equivalence determined in the usual way, but there is the extra proviso that if $s_1 \leq s_2$, then $\equiv_{s1} \subseteq \equiv_{s2}$. For instance, with Int<Rat, if two objects are equal as integers, they are also valid rationals, and they must be equal as rationals as well.

Definition: the **initial algebra** of an order-sorted ADT consists of the equivalence classes of well-formed ground terms $T(\Sigma,\varnothing)$ just as in the non order-sorted case, where equivalence must be provable from the equational axioms. An equivalence class may contain terms of several sorts.

Definition: the **final algebra** of an order-sorted ADT consists of the indistinguishability classes of ground terms T(Σ,∅) just as in the non order-sorted case, where to be indistinguishable two objects must be members of a common sort so that the same operations are applicable to them, and must yield the same element of a pre-defined type whenever the same operations are applied to them.

The sort of a class in either case is the smallest sort of any term contained in the class.

Example — Order-Sorted Stack of Integers

Signature (Σ)

sorts: Stack, NeStack, and Int (pre-defined)
subsorts: NeStack < Stack

NEW: → Stack
PUSH: Stack × Int → NeStack
POP: NeStack → Stack
TOP: NeStack → Int

Equations

POP(Push(s,i)) = s
TOP(PUSH(s,i)) = i

This is it — errors and all! The intuition is that the subsort NeStack (for non-empty stacks) isolates the exceptional element for the ADT. Actually errors are implicitly indicated by the impossibility of applying some operations to the values that would produce exceptions. We understand that it is incorrect (technically impossible according to the operation signatures) to apply either POP or TOP to the empty stack NEW, so the specification leaves unsaid further details about these circumstance.

The initial and final algebra views are the same for this example, as we will see as we examine the equivalence classes. First of all note that the order-sorted signature substantially changes the term algebra and the equivalence classes — that is, well-formed terms must respect the type signature of the operations, so terms of the form POP(POP(…)) are not “type respecting” and do not belong to the collection of well-formed ground terms, T(Σ,∅). In fact, the signature alone now forbids all terms that would cause an exception (or error)!

NEW and PUSH still serve as constructors in this order-sorted version. If we consider the initial algebra of equivalence classes of terms, the canonical representatives of equivalence classes are still provided by these operations, however the equivalence classes contain different terms than in the non-sorted version.

[NEW] = {NEW} ∪
{POP(PUSH(NEW,i)) | i∈Int} ∪
{POP(PUSH(POP(PUSH(NEW,i1)),i2)) | i1,i2∈Int} ∪ …

[PUSH(NEW,i1)] = {PUSH(NEW,i1)} ∪
{POP(PUSH(PUSH(NEW,i1),i2)) | i2∈Int} ∪
{PUSH(POP(PUSH(NEW,i2)),i1) | i2∈Int} ∪ …

etc.

Notice that the class algebra $T(\Sigma,\emptyset)/\equiv$ is sufficiently complete — every well-formed ground term of sort Int can be proven equivalent to some term in Int.

Can we POP "twice in a row" in this ADT? Note that
POP(POP(PUSH(PUSH(NEW,1),2)))
is ill-typed (i.e., not in $T(\Sigma,\emptyset)$), but
POP(PUSH(PUSH(NEW,1),2))
is well formed and equivalent to PUSH(NEW,1). And POP(PUSH(NEW,1)) $\equiv$ NEW, so POP(PUSH(NEW,1)) $\in$ [NEW], but
POP(POP(PUSH(PUSH(NEW,1),2)))
is not among the well-formed terms $T(\Sigma,\emptyset)$! So applying the POP operation *unconditionally* twice in a row is not possible! Substituting equals for equals is restricted by the requirement to form terms that are type correct. This provides an implicit indication of the necessity of dynamic testing of the result of a POP operation — it may be of type NeStack, or it may not. In the initial algebra, to apply an operation f to an equivalence class [t], there must be a term t'$\in$[t] so that f(t') is well formed, then the result f([t]) is [f(t')].

In the initial algebra, the sort NeStack consists of all the equivalence classes except [NEW] — [NEW] has no terms of sort NeStack, and all the other classes do. Hence we see that this specification is sufficiently complete since for all the terms t in NeStack classes, TOP(t) is equivalent to a term in pre-defined type Int. Also in the final algebra, terms from two different classes can be distinguished by suitable use of POP and TOP so indistinguishability classes are the same.