Algebraic Specification Constituents

Data Domains
An algebraic abstract data type may involve pre-defined types. These can be assumed to be previously defined ADTs. Thus we are permitted to develop hierarchical definitions. To a very limited extent we may involve *primitive* (or concrete) types — types whose behavior is known a priori. Several distinct varieties of data may be involved (i.e., it may be heterogeneous). For example, for sets of Integers we have both set values and Integer values involved, as in the Pascal expression “2 **in** s1+s2”. These data domains are called **sorts**. An ADT specification provides a *name* for each of the sorts it introduces. However, no representation of the data values themselves is provided — only the sort names. This is one of the characteristics that accounts for the word 'abstract' in the name. One (and sometimes several) of the sorts of the ADT being defined is distinguished as the object(s) of primary interest and is referred to as the **type (or types) of interest** (TOI).

Operation signatures (syntax)
This component provides the names of the operations and their domain/range characteristics. Also, we may provide notational conventions (i.e., in-fix, pre-fix, or post-fix), and (rarely) precedence conventions. These properties are called the **signatures** of the functions.

It is assumed that the operations are “pure functions” — that is, there are no side-effects. Also, for the time being we will assume that the functions are total — they are defined for all the values in their domain. The treatment of partial functions is a non-trivial issue we will explore later. Note that constants are viewed as functions with no arguments.

For example, for an ADT Stack of Integer describing push-down stacks of Integers we have the pre-defined type Integer, the TOI sort Stack, and the operations and their signatures:
PUSH: Stack $\times$ Integer $\rightarrow$ Stack
POP: Stack $\rightarrow$ Stack
NEW: $\rightarrow$ Stack
TOP: Stack $\rightarrow$ Integer.

We expect the arguments to functions to conform to the type requirements indicated in the signatures. This would mean that PUSH(NEW, 2) is a valid expression while PUSH(2, NEW) is not. Since this information determines exactly what the well-formed expressions are, this may be referred to as 'syntax'. Legal **expressions** (or **terms**) are defined inductively. We assume that there is an unlimited collection of variable names associated with each sort. Then

- each variable and constant of sort S is a valid expression (or term) of sort S,
- if e_i is a valid expression of sort S_i ($1 \le i \le n$) and the ADT operation f has the signature f: $S_1 \times S_2 \times \ldots \times S_n \rightarrow S_0$, then $f(e_1, e_2, \ldots, e_n)$ is a valid expression (or term) of sort S_0.

This collection of well-formed expressions is the “language” of the ADT.

If Σ is a signature, the set of well-formed expressions (or terms) over Σ which use variables drawn from the set V is denoted by $T(\Sigma,V)$. If the variables are not restricted we write $T(\Sigma)$. A term that includes no variable is called **ground**.

Operation properties (semantics)
No algorithms are provided for the operations — we have no information about the values of the TOI, so this is outside the realm of possibility. We seek to specify the *results* of the operations while avoiding a bias toward any particular implementation. This is the other characteristic that accounts for the word 'abstract' in the name.

Properties of the operations consist of equations between two expressions (of the same sort), and are understood to assert that the two expressions evaluate to the same result for all values of the variables involved. An algebraic ADT specification encompasses a finite collection of equations.

For instance for the Stack example, we have for all s: Stack and i: Integer

```
   POP(PUSH(s, i)) = s
   TOP(PUSH(s,i)) = i.
```

The reason that this may be referred to as 'semantics' is that the behavior (i.e., results) of the operations can be inferred from these constraints independent of the details of any implementation. For example,

```
     TOP(POP(PUSH(PUSH(NEW,1), 2)))
   = TOP(PUSH(NEW, 1))  by the first axiom
   = 1  by the second axiom.
```

The inference rules used for deducing “behavior” are those familiar for reasoning about equalities, namely, for all x, y, and z

- x = x,
- if x = y, then y = x,
- if x = y and y = z, then x = z, and
- we may “substitute equals for equals”, e.g., if x = y, then f(x) = f(y), etc.