Types in Z

Z has been designed as a strongly typed language. This can be useful in a practical way by clarifying the kind of object (and hence its properties) under consideration. It is also essential from the theoretical point of view when working with sets in a very general setting, in particular when defining sets by describing the properties of their members as we do in **Z**. It is well known that paradoxes can easily arise when sets are described by unconstrained properties. One famous case is Russell's paradox. Suppose we say a set S is *ordinary* if $S \notin S$. Since to identify a set, we must describe its members, being ordinary (or not) is an apparently reasonable property to attach to a set. But then consider the collection $\mathcal{O}$ of all and only ordinary sets, and the question: is $\mathcal{O}$ ordinary? Of course, the answer must be either yes or no. However,

if yes, $\mathcal{O}$ *is* ordinary, then by the definition of ordinary, $\mathcal{O} \notin \mathcal{O}$ and we have an ordinary set not in $\mathcal{O}$, an impossibility;

if no, $\mathcal{O}$ is *not* ordinary, then by the definition of ordinary, $\mathcal{O} \in \mathcal{O}$ and we have an non-ordinary set in $\mathcal{O}$, an impossibility.

Hence we have a logically impossible situation -- a paradox -- that arises from seemingly natural methods of defining sets. One method to avoid such logical catastrophes is to impose the discipline of typing on the sets that are considered. It goes beyond the scope of our class to develop the assurance that this is sufficient to avoid these logical pitfalls however.

Since **Z** is a strongly typed language, the nature of types is a primary issue. In many presentations this is given minimal attention. However, every variable that appears in a Z specification must be declared, and the declaration identifies its type. Also, every function (and predicate) has prescribed types for its arguments and result. Finally, every expression has a type associated with it that is determined by the types of its constituents and the operations performed on them.

Types generally provide a set of values, together with a collection of operations that may be performed on these values. But not every set of values is a type. The following are *basic types* of **Z**:

- in a generic declaration such as **[X, Y]**, X and Y are (names of) basic types (or *given types*) -- all operations and their properties must be described by the specification,
- * the signed integers $\mathbb{Z} = \{ \dots -2, -1, 0, 1, 2, \dots \}$ together with all their familiar operations and properties is a basic type; additional operations and their properties may be described by the specification
- * the subsets of values of a type T constitute a type, $\mathbb{P}$ T, with the familiar set operations and properties
- * the *Cartesian product* (n-tuples), written $T_1 \times T_2 \times \dots \times T_n$,for types $T_1, T_2, \dots, T_n$ denotes the collection of values $(x_1, x_2, \dots, x_n)$ where x_i is a value of type T_i $(1 \le i \le n)$. Operations on n- tuples are the *projection functions*, $p_i(x_1, x_2, \dots, x_n) = x_i$ $(1 \le i \le n)$. The most common special case is for n=2, and 2-tuples are often called *pairs*.

For a type T, we write x:T, and for a set T we write x[T. However, commonly corruptions of types are used. For instance, we may see a declaration n:ℕ, while ℕ is a set not a proper type. Strictly speaking the type of n is ℤ, plus there is an implicit assertion n≥0. So while a clear distinction is sometimes not made, the means in the previous paragraph are the required ways to form legitimate **Z** types; one additional means called *free types* will be discussed later.

Since values have types, and operations define types for their arguments and results, expressions constructed from several operations provide type requirements as well. Expression type requirements are determined by direct application of the type characteristics of the values and operations it is composed from.