Chapter 4
Working within an arbitrary logical system

Several approaches to specification were discussed in Chapter 2. Each approach involved a different logical system as a part of its mathematical underpinnings. We encountered different definitions of:

Signatures: “ordinary” many-sorted signatures, signatures containing bool, true and false (for final and reachable semantics), error signatures, order-sorted signatures;

Algebras (on a signature $\Sigma$): “ordinary” $\Sigma$-algebras, error $\Sigma$-algebras, partial $\Sigma$-algebras, order-sorted $\Sigma$-algebras;

Logical sentences (on a signature $\Sigma$): $\Sigma$-equations, conditional $\Sigma$-equations, error $\Sigma$-equations (with safe and unsafe variables), $\Sigma$-definedness formulae, order-sorted $\Sigma$-equations; and

Satisfaction (of a $\Sigma$-sentence by a $\Sigma$-algebra): of a $\Sigma$-equation by a $\Sigma$-algebra, of an error $\Sigma$-equation by an error $\Sigma$-algebra, of a $\Sigma$-equation by a partial $\Sigma$-algebra, of a $\Sigma$-definedness formula by a partial $\Sigma$-algebra, of an order-sorted $\Sigma$-equation by an order-sorted $\Sigma$-algebra.

All of these choices can be combined to obtain many different logical systems and hence different approaches to specification, e.g. partial error specifications with conditional axioms. There are also several alternative approaches to the specification of partial algebras and likewise for the specification of error handling. Furthermore, there are many other variations that have not been considered, including the following (some of them briefly mentioned in Section 2.7.6):

- polymorphic signatures which permit polymorphic type constructors (rather than just sorts) and operations having polymorphic types;
- continuous algebras to handle infinite data objects such as streams;
- higher-order algebras to handle higher-order functions (i.e. functions taking functions as arguments and/or yielding functions as results);
- relational structures to model specifications containing predicates;
- inequations and conditional inequations;
- first-order formulae, with and without equality;
• various modal logics, including algorithmic, dynamic, and temporal logics, for formulating properties of (possibly non-functional) programs.

Some of these variations depart quite considerably from the usual algebraic framework presented in Chapters 1 and 2. But none of them (and very few of the others considered in the literature) are artificial, resulting merely from a theoretician’s toying with formal definitions. All of them arise from the practical need to specify different aspects of software systems, often reflected by diverse features of different programming languages.

The resulting wealth of choice of definitions of the basic concepts is not a bad thing. None of the logical systems used in specifications is clearly better than all the others — and we should not expect that such a “best” system will ever be developed. In theory, we can imagine putting all of the above concepts together, producing a single logical system where signatures, algebras, sentences and the satisfaction relation would cover as special cases all we have considered up to now. But the result would be so huge and complex as to make it unmanageable. Moreover, what would we do if one day somebody pointed out that yet another view of software is important and should be reflected in specifications, and hence included in the logical system we use? Scrap everything and start again?

Different specification tasks may call for different systems to express most conveniently the properties required. Moreover, different logical systems may be appropriate for describing different aspects of the same software system, and so a number of logical systems may be useful in a single specification task. It is thus important that the designer of a software system be able to choose which logical system(s) to use.

An unfortunate effect of this necessary wealth of choice is that different researchers have tended to adopt different combinations of basic definitions in their work, sometimes varying their choices from one paper to the next. This makes it difficult to build on the work of others, to compare the results obtained for different logical systems, and to transfer results from one system to another. Such results include not only mathematical definitions and theorems, but also practically useful tools supporting software specification, development and verification produced at great expense of effort, time and money.

In fact, much of the work done turns out to be independent of the particular choice of the basic definitions, although this is often not obvious. The main objective of this chapter, and one of the main objectives of this book, is to lay out the mathematical foundations necessary to make this independence explicit. We achieve this using the notion of an institution, which formalises the informal concept of a logical system devised to fit the purposes of specification theory; see Section 4.1 below for the definition. Our thesis is that building as much as possible on the notion of an institution brings important benefits to both the theory and the practice of software specification and development. On the one hand, this allows much work on theories, results, and practical tools to be done just once for many different specific logical systems; on the other hand it forces, via abstraction, a better understanding of and deeper insight into the essence of the concepts and results.