Semantic Typing for
Parametric Algebraic Specifications

María Victoria Cengarle

Institut für Informatik
Ludwig-Maximilians-Universität München
Leopoldstr. 11b, 80802 Munich, Germany
cengarle@informatik.uni-muenchen.de

Abstract. The implementation relation of refinement of specifications is studied in the framework of the calculus of higher-order parameterization of specifications. An existing system of derivation of the relation among non-parametric specifications is enlarged so as to comprise parametric specifications. The new system is correct and complete under certain assumptions. By means of this system the calculus of parametric specifications can be enhanced with semantic types, and in this way a specification is a valid argument of a parametric specification as long as it shows the particular behavior demanded by the semantic parameter restriction. This typing can be derived, and so function application is conditional to the derivability of the parameter restrictions instantiated with the actual argument.

Keywords: algebraic specification, parametric specification, proof, implementation relation.

1 Introduction

A specification language is usually equipped with a notion of proof and a notion of implementation (see [6, 10]). The first one assists the inference of further relationships, other than the ones provided when the particular specification was defined. The second is a reflexive and transitive binary relation among specifications, which supports stepwise derivation: starting from an abstract description, a concrete specification or program is reached.

In [3] we have presented a specification language supporting programming in the large by two different means: specification-building operators and parameterization over higher-order variables. Specification-building operators allow the construction or combination of specifications, whose semantics is loosely defined as the collection of all models satisfying the inherited axioms. Parameterization abstracts just in the same way function definition does in traditional programming languages and is based on the simply typed \( \lambda \)-calculus. In this context, function application maps specification expressions into specification expressions, this means, parametric specifications are neither interpreted as specification morphisms nor define parametric algebras or structures (cf. [8]). Because of the parameter restrictions associated with the specification-building operators, requirements can be calculated that express the (syntactic) conditions to
be fulfilled by a specification in order to be a valid argument to a parametric specification. These requirements define themselves a parametric calculus and their validity can be derived. A contextual proof system was also provided that allows the derivation of formulas valid in a specification term under certain assumptions for its free variables. What is missing in our earlier work is any definition of an implementation relation, and thus of semantic typing to be explained below.

In the present paper we revisit the refinement implementation relation. We extend this partial order relation defined by signature equality and model class inclusion to parametric specifications in the standard way. We present a system of derivation of this extended relation based on the contextual proof system, that is, derivations are relative to a context of value assumptions. It extends the one presented in [10], and in the same way as there it generates proof obligations. The system is correct and complete w.r.t. denotable assumptions.

Let us motivate with an example. Suppose we have the specification \texttt{NAT} of natural numbers and the specification \texttt{INT} of integers. By examination of the corresponding axioms, we may conclude that the natural numbers are implemented by the integers, usually denoted by \texttt{NAT} \rightarrow \rightarrow \texttt{INT} (where signature adjustment may be necessary). Working with structured specifications, we can also prove that \texttt{SET} + \texttt{NAT} \rightarrow \rightarrow \texttt{LIST} + \texttt{INT}. There are two ways to test this latter relation. We can decompose the structure of both specifications and calculate the set of inherited axioms of both expressions (the specification arising by such a process is called a normal form in [11]). The drawback of this procedure is the loss of structure, and normalization itself can be a task of importance in the case of bigger specifications. The alternative is to take advantage of the structure, as for example in the system in [10], thus supporting the reuse of derivations. There we have an inference rule for each specification-building operator. Given that parameterization adds even more structure to the definition of specifications, we work exclusively with this structured inference system. This system extended allows the derivation of \texttt{SET}_{\text{par}}(\texttt{NAT}) \rightarrow \rightarrow \texttt{LIST}_{\text{par}}(\texttt{INT}), where \texttt{SET}_{\text{par}} and \texttt{LIST}_{\text{par}} are parameterized specifications of sets and lists, respectively.

These results enable the introduction of semantic conditions to be fulfilled by the argument passed to a parametric specification. That is, an actual parameter may need not only to have some syntactic characteristics but also to exhibit a particular behavior in order to be a valid argument of a parametric specification. In a simple case, the satisfaction of formulas may be expressed using the implementation relation, that is, an actual parameter is a valid argument if it proves a set of formulas. In general, different parameters may interact, for instance we can demand that a parameter implements another argument. We define further a mechanism for typing the semantics of the parameters of a specification term. The additional type of a parameter variable is a set of conditions. A condition is defined as a pair of specification terms (which may contain free occurrences of

\footnote{This inference depends on the logic chosen: it is not always the case that \texttt{INT} \vdash (\forall n)(0 \neq \text{succ}(n)). Such conclusions are valid in the so-called ultra-loose approach to semantics of specifications, which allows the reasoning with constructors; see [10, 11].}