Toward formal development of programs from algebraic specifications: parameterisation revisited*

Donald Sannella† Stefan Sokołowski‡ Andrzej Tarlecki§

Abstract

Parameterisation is an important mechanism for structuring programs and specifications into modular units. The interplay between parameterisation (of programs and of specifications) and specification (of parameterised and of non-parameterised programs) is analysed, exposing important semantic and methodological differences between specifications of parameterised programs and parameterised specifications. The extension of parameterisation mechanisms to the higher-order case is considered, both for parameterised programs and parameterised specifications, and the methodological consequences of such an extension are explored.

A specification formalism with parameterisation of an arbitrary order is presented. Its denotational-style semantics is accompanied by an inference system for proving that an object satisfies a specification. The formalism includes the basic specification-building operations of the ASL specification language and is institution independent.

1 Introduction

Modular structure is an important tool for organizing large and complex systems of interacting units. When a system is decomposed into self-contained modules with well-defined interfaces, the number of possible interactions between parts of the system is greatly reduced. This makes it possible to understand each module in relative isolation from the other modules in the system.

The application of modular structure to the organization of "dynamic" systems such as programs and machines is well known. Here, interactions between parts of a system involve transmission of data or physical contact between bits of metal. Its application to "static" systems such as algebraic specifications is perhaps less obvious but just as important. Interactions here are more implicit and insidious, where axioms meant to specify one function can indirectly constrain the possible implementations of other functions as well.

The first algebraic specification language which provided the means to structure specifications was CLEAR [BG 80]. Since then the need for structure in specifications has become universally recognized, and mechanisms for structuring specifications appear in all modern algebraic specification languages including CIP-L [Bau 85], ASL [SW 83], [Wir 86], ACT ONE [EM 85] and the Larch Shared Language [GHW 85].

An important structuring mechanism is *parameterisation*. This allows modules to be defined in a generic fashion so that they may be applied in a variety of contexts which share some common characteristics. A parameterised program module F [Gog 84] (an ML functor [MacQ 86]) may be applied to

*To appear in *Acta Informatica*.

†Department of Computer Science, University of Edinburgh, Edinburgh, UK.

‡Institute of Computer Science, Polish Academy of Sciences, Gdańsk, Poland.

§Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland.

any non-parameterised program module A_{arg} matching a given import interface A_{par}. The result is a non-parameterised program module $F(A_{arg})$, a version of F in which the types and functions in A_{par} have been instantiated to the matching types and functions in A_{arg}. An example of a parameterised program module is a parser module which takes a lexical analyser module as argument. Similarly, a parameterised specification P may be applied to any non-parameterised specification SP_{arg} fitting a certain signature Σ_{par} (or specification SP_{par}) to yield a specification $P(SP_{arg})$. A standard example is a specification *Stack-of-X* which takes a specification of stack elements and produces a specification of stacks containing those elements. In some algebraic specification frameworks, parameterisation is implicit in the sense that no distinction is made between parameterised and non-parameterised specifications (see for example LOOK [ETLZ 82], ASPIK [Voß 85] and the unified algebra framework [Mos 89a], [Mos 89b]) but the idea is the same.

The above discussion has dealt with two distinct classes: programs and specifications. Appropriate parameterisation mechanisms give rise to two new (and distinct) classes: parameterised programs and parameterised specifications. It is possible to specify parameterised programs, just as it is possible to specify non-parameterised programs. The result is *not* a parameterised specification; it is a non-parameterised specification of a parameterised program. In work on algebraic specification there has been a tendency to ignore this distinction and use one flavour of parameterisation for both purposes. This has sometimes led to misunderstanding and confusion. In this paper, we argue that the distinction is important both for semantical reasons and because the two kinds of specifications belong to different phases of program development: parameterised specifications are used to structure requirements specifications, while specifications of parameterised programs are used to describe the modules which arise in the design of an implementation. The most natural way to structure a specification often reflects the way that an implementation would be structured. But this is not always the case, and then the lack of a distinction causes real problems.

It is natural to consider what happens when parameterisation mechanisms are extended to the higher-order case in which parameterised objects are permitted as arguments and as results. This makes sense both for parameterised specifications and for specifications of parameterised programs. The consequences of such an extension are explored in this paper. It is shown that re-interpreting the concept of constructor implementation in [ST 88b] in these terms leads to a natural extension to deal with implementations of specifications of parameterised programs. This in turn supports an extension of the methodology for formal development of ML programs from specifications presented in [ST 89] to the case of higher-order ML functors.

The paper is organized as follows. After some preliminary definitions in Section 2, Section 3 surveys four of the approaches to parameterisation found in algebraic specification languages. This is not intended as an exhaustive overview of the literature on parameterisation; the four approaches discussed were found to be useful as a means of introducing the ideas in this paper, and many other related and important studies have been omitted (e.g. [Ehr 82] and [Gan 83], just to mention two). In Section 4, the similarities and differences between these approaches are analysed. The distinction between parameterised specifications and specifications of parameterised programs is brought to light, and the consequences of this distinction are investigated. This part of the paper may be summarized by the following slogan:

$$\text{parameterised (program specification)} \neq \text{(parameterised program) specification}$$

In Sections 5 and 6, the technical and methodological consequences of extending parameterisation to the higher-order case are considered. Section 7 presents a specification formalism built on the institution-independent kernel specification language in [ST 88a] which supports the specification of arbitrarily high-order parameterised programs, as well as extending the mechanism in [ST 88a] for defining first-order parameterised specifications to the higher-order case. This section is based on a more extensive presentation of this formalism in [ST 91a]. Finally, Section 8 contains conclusions and some ideas for future work.

2 Preliminaries

Throughout the paper we assume that the reader is familiar with the basic concepts of logic and universal algebra. In particular we will freely use the notions of: algebraic many-sorted signature, usually denoted by Σ, Σ', Σ_1, etc.; algebraic signature morphism $\sigma : \Sigma \to \Sigma'$ (this yields the category of signatures $\mathcal{SIGN}$); Σ-algebra; Σ-homomorphism; Σ-isomorphism; Σ-equation; first-order Σ-sentence (the set of all Σ-sentences will be denoted by $Sen(\Sigma)$); and satisfaction relation between Σ-algebras and Σ-sentences. These all have the usual definitions (see e.g. [ST 88a]) and a standard, hopefully self-explanatory notation is used to write them down. We also make minor use of the pushout construction of category theory.

For any signature Σ, the class of all Σ-algebras is denoted by $Alg(\Sigma)$. We will identify this with the category of Σ-algebras and Σ-homomorphisms whenever convenient. If $\sigma : \Sigma \to \Sigma'$ is a signature morphism then $\_|_\sigma : Alg(\Sigma') \to Alg(\Sigma)$ is the reduct functor defined in the usual way (the notation $\_|_\Sigma$ is sometimes used when σ is obvious). Now, given a signature morphism $\sigma : \Sigma \to \Sigma'$ and functor $F : Alg(\Sigma) \to Alg(\Sigma')$, we say that F is *(strongly) persistent along* σ if for every algebra $A \in Alg(\Sigma)$, $F(A)|_\sigma = A$.

For any signature Σ, by a Σ-*presentation* we mean any set of Σ-sentences. Any Σ-presentation Φ determines the class of its models, written $[\![\Phi]\!]$, which consists of all Σ-algebras that satisfy all the sentences in Φ. By a Σ-*theory* we mean any Σ-presentation which is closed under semantical consequence, i.e., a set Φ of Σ-sentences is a Σ-theory if all the sentences that hold in $[\![\Phi]\!]$ are in Φ.

The most basic assumption of work on algebraic specification is that software systems are modelled as algebras, abstracting away from the concrete details of algorithms and code and focussing on their functional behaviour. Roughly, the signature of the algebra gives the names of data types and of operations available to the user of the system, and the algebra itself gives the semantics of the particular realizations of these data types and operations defined by the system. Consequently, to specify a software system viewed in this way means to give a signature (fix the abstract syntax available to the user) and define a class of algebras over this signature, that is, describe a class of admissible realizations of the system.

One way to give a specification of a system is to present a list of axioms over a given signature and describe in this way the properties that the operations of the system are to satisfy. This view of a software-system specification as a Σ-presentation (for an appropriate signature Σ) is perhaps the simplest possible, but has a number of disadvantages. Most notably, any specification of a real software system given in this style would comprise a very long, unstructured, and hence unmanageable list of axioms.

To cope with this problem, a number of so-called *specification languages* have been designed, which allow specifications to be built in a structured manner using a predefined set of *specification-building operations*. According to the brief discussion above, the most essential feature of any such specification formalism is that every specification SP over a given signature Σ (we will say that SP is a Σ-specification) unambiguously determines the class of admissible realizations of the system being specified, i.e. a class of Σ-algebras (sometimes referred to as *models* of the specification). Thus, any Σ-specification SP denotes a class of Σ-algebras $[\![SP]\!] \in Pow(Alg(\Sigma))$[1]. A specification SP is called

[1] $Pow(X)$, for any class X, denotes the "class of all subclasses" of X. This raises obvious foundational difficulties. We disregard these here, as they may be resolved in a number of standard ways. For example, for the purposes of this paper we could assume that algebras are built within an appropriate universal set, and deal with sets, rather than classes, of algebras.

consistent if it has at least one model, i.e. $[\![SP]\!] \neq \emptyset$. See [ST 88a], [ST 92] for a more extensive discussion of the semantics of specifications.

As a starting point for the presentation of specifications in this paper, we recall here the simple yet powerful specification-building operations defined in [ST 88a] (with the slight difference that signatures are regarded as specifications in their own right here with **impose** Φ **on** Σ in place of $\langle \Sigma, \Phi \rangle$). This was in turn based on the ASL specification language [SW 83], [Wir 86]. The main use of these operations is in examples where they should be more or less self-explanatory. The particular choice of specification-building operations is not important for the purposes of this paper.

- If Σ is a signature, then Σ is a Σ-specification with the semantics:

$$[\![\Sigma]\!] = Alg(\Sigma)$$

- If SP is a Σ-specification and Φ is a set of Σ-sentences, then **impose** Φ **on** SP is a Σ-specification with the semantics:

$$[\![\textbf{impose } \Phi \textbf{ on } SP]\!] = \{A \in [\![SP]\!] \mid A \models \Phi\}$$

- If SP is a Σ-specification and $\sigma : \Sigma' \to \Sigma$ is a signature morphism, then **derive from** SP **by** σ is a Σ'-specification with the semantics:

$$[\![\textbf{derive from } SP \textbf{ by } \sigma]\!] = \{A|_\sigma \mid A \in [\![SP]\!]\}$$

- If SP is a Σ-specification and $\sigma : \Sigma \to \Sigma'$ is a signature morphism, then **translate** SP **by** σ is a Σ'-specification with the semantics:

$$[\![\textbf{translate } SP \textbf{ by } \sigma]\!] = \{A' \in Alg(\Sigma') \mid A'|_\sigma \in [\![SP]\!]\}$$

- If SP and SP' are Σ-specifications, then $SP \cup SP'$ is a Σ-specification with the semantics:

$$[\![SP \cup SP']\!] = [\![SP]\!] \cap [\![SP']\!]$$

- If SP is a Σ-specification and $\sigma : \Sigma' \to \Sigma$ is a signature morphism, then **minimal** SP **wrt** σ is a Σ-specification with the semantics:

$$[\![\textbf{minimal } SP \textbf{ wrt } \sigma]\!] = \{A \in [\![SP]\!] \mid A \text{ is minimal in } Alg(\Sigma) \text{ w.r.t. } \sigma\}^2$$

  where a Σ-algebra A is *minimal w.r.t.* σ if it has no non-trivial subalgebra with an isomorphic σ-reduct (cf. [ST 88a]).

- If SP is a Σ-specification, then **iso-close** SP is a Σ-specification with the semantics:

$$[\![\textbf{iso-close } SP]\!] = \{A \in Alg(\Sigma) \mid A \text{ is isomorphic to } B \text{ for some } B \in [\![SP]\!]\}$$

- If SP is a Σ-specification, $\sigma : \Sigma \to \Sigma'$ is a signature morphism and Φ' is a set of Σ'-sentences, then **abstract** SP **wrt** Φ' **via** σ is a Σ-specification with the semantics:

$$[\![\textbf{abstract } SP \textbf{ wrt } \Phi' \textbf{ via } \sigma]\!] = \{A \in Alg(\Sigma) \mid A \equiv^\sigma_{\Phi'} B \text{ for some } B \in [\![SP]\!]\}$$

  where $A \equiv^\sigma_{\Phi'} B$ means that A is observationally equivalent to B w.r.t. Φ' via σ (see [ST 87], [ST88a] for details).

[2] This is slightly different from the definition in [ST 88a].

The above definitions were given in [ST 88a] in the framework of an arbitrary *institution* [GB 84]. This means that the specification-building operations defined above are actually independent of the underlying logical system, that is, of the particular definitions of the basic notions of signature, algebra, sentence and satisfaction relation. In this paper it will be convenient to use some other specification-building operations defined in the standard framework of first-order logic, as follows:

- If SP is a Σ-specification and SP' is a Σ'-specification, then $SP+SP'$ is a $(\Sigma\cup\Sigma')$-specification with the semantics:

  $$[\![SP + SP']\!] = \{A \in Alg(\Sigma \cup \Sigma') \mid A|_{\Sigma} \in [\![SP]\!] \text{ and } A|_{\Sigma'} \in [\![SP']\!]\}$$

  (this is expressible using union and **translate** as defined above, see [ST 88a]).

- If SP is a Σ-specification, S is a set of sort names, Ω is a set of ranked operation names such that adding S and Ω to Σ yields a well-formed signature Σ', and Φ' is a set of Σ'-sentences, then **enrich** SP **by sorts** S **opns** Ω **axioms** Φ' is a Σ'-specification with the semantics:

  $$[\![\textbf{enrich } SP \textbf{ by sorts } S \textbf{ opns } \Omega \textbf{ axioms } \Phi']\!] = \{A \in Alg(\Sigma') \mid A|_{\Sigma} \in [\![SP]\!] \text{ and } A \models \Phi'\}$$

  (this is expressible using **translate** and **impose** as defined above, see [ST 88a]).

- If SP is a Σ-specification and S is a set of sort names in Σ, then **reachable** SP **on** S is a Σ-specification with the semantics:

  $$[\![\textbf{reachable } SP \textbf{ on } S]\!] = \{A \in [\![SP]\!] \mid A \text{ is generated on } S\}$$

  where A is said to be *generated on* S if it has no proper subalgebra having the same carriers of sorts not in S (this is expressible using **minimal** as defined above, see [ST 88a]).

For example we can now define:

$$
\begin{array}{lll}
\text{Bool} & =_{def} & \textbf{reachable} \\
& & \quad \begin{array}{ll}
\textbf{sorts} & \text{bool} \\
\textbf{opns} & \text{true}, \text{false} : \ \to \text{bool} \\
& \ldots \\
\textbf{axioms} & \text{true} \neq \text{false} \\
& \ldots
\end{array} \\
& & \textbf{on } \{\text{bool}\}
\end{array}
$$

$$
\begin{array}{lll}
\text{Nat} & =_{def} & \textbf{reachable} \\
& & \quad \begin{array}{lll}
\textbf{enrich} & \text{Bool} & \\
\textbf{by} & \textbf{sorts} & \text{nat} \\
& \textbf{opns} & \text{zero} : \ \to \text{nat} \\
& & \text{succ} : \text{nat} \to \text{nat} \\
& & > : \text{nat} \times \text{nat} \to \text{bool} \\
& & \ldots \\
& \textbf{axioms} & \forall n : \text{nat}.\ \text{succ}(n) > \text{zero} = \text{true} \\
& & \ldots
\end{array} \\
& & \textbf{on } \{\text{nat}\}
\end{array}
$$

Note that at the semantical level, each of the specification-building operations introduced above is a function mapping classes of algebras to classes of algebras. We will assume that when viewed this way, all specification-building operations are monotone w.r.t. the inclusion ordering on classes. This is indeed the case for all the above operations.

If SP is a Σ-specification and SP' is a Σ'-specification, then a *specification morphism* from SP to SP', $\sigma : SP \to SP'$, is a signature morphism $\sigma : \Sigma \to \Sigma'$ such that for all $A' \in [\![SP']\!]$, $A'|_\sigma \in [\![SP]\!]$. This yields the category of specifications $\mathcal{SPEC}$ (with composition and identities inherited from the category of signatures $\mathcal{SIGN}$). $\mathcal{SPEC}$ is co-complete, with colimits in $\mathcal{SPEC}$ determined by colimits in $\mathcal{SIGN}$ (cf. [GB 84], [ST 88a]). Note that this definition of specification morphism works for any specification formalism with a semantics in the style presented above.

3 Overview of parameterisation mechanisms

A specification language provides a certain number of specification-building operations such as those defined above. As we have mentioned, such operations may be viewed as functions mapping specifications to specifications. More complex functions of this type may be defined as combinations of the elementary specification-building operations provided. We can use λ-abstraction to write such functions down, where the semantics of application is given by β-conversion. This is the approach adopted in ASL [SW 83], [Wir 86]; the particular definition below is taken from [ST 88a] with a minor syntactic modification (viz., the use of *Spec*(_) to conform with conventions to be introduced later).

Definition 3.1 An *ASL-style parameterised specification* has the form $\lambda X{:}Spec(\Sigma_{par}).\ SP_{res}[X]$ where Σ_{par} is a signature and $SP_{res}[X]$ is a Σ_{res}-specification which may contain one or more uses of X as a Σ_{par}-specification. The result of applying such a parameterised specification to a Σ_{par}-specification SP_{arg} is defined as follows:

$$(\lambda X{:}Spec(\Sigma_{par}).\ SP_{res}[X])(SP_{arg}) =_{def} SP_{res}[SP_{arg}/X]$$

where $SP_{res}[SP_{arg}/X]$ is SP_{res} with all occurrences of X replaced by SP_{arg}. □

This naturally extends to higher-order parameterised specifications in the standard way. Recursion is also no problem since all specification-building operations are monotonic with respect to the inclusion of model classes.

A characteristic feature of the specification-building operations defined in Section 2 and in [ST 88a] is that the signature of $SP_{res}[SP_{arg}/X]$ is not dependent on SP_{arg}: $SP_{res}[SP_{arg}/X]$ is a Σ_{res}-specification for any Σ_{par}-specification SP_{arg}. Thus a parameterised specification $\lambda X{:}Spec(\Sigma_{par}).\ SP_{res}[X]$ describes a function mapping Σ_{par}-specifications to Σ_{res}-specifications.

This function is only defined for specifications over the indicated parameter signature Σ_{par}. In the framework of [ST 88a], it is possible to apply a parameterised specification $\lambda X{:}Spec(\Sigma_{par}).\ SP_{res}[X]$ to a Σ_{arg}-specification SP_{arg} where $\Sigma_{arg} \supset \Sigma_{par}$, but only by first applying **derive** to SP_{arg} (via the inclusion morphism $\iota : \Sigma_{par} \hookrightarrow \Sigma_{arg}$); the same trick works for any SP_{arg} where there is a signature morphism $\sigma : \Sigma_{par} \to \Sigma_{arg}$. The part of SP_{arg} which is thereby "forgotten" does not reappear in the result of the application.

A different approach is taken in CLEAR [BG 80], where parameterised specifications are used to uniformly enrich given argument specifications. In this approach, everything which is in the argument specification carries through to the overall result. This is achieved by making the "fitting" of the argument specification to the indicated parameter signature an explicit part of the argument-passing mechanism.

Definition 3.2 A *CLEAR-style parameterised specification* is a specification morphism $P : SP_{par} \to SP_{res}$, where SP_{par} is a Σ_{par}-specification and SP_{res} is a Σ_{res}-specification. The overall result of applying such a parameterised specification to a Σ_{arg}-specification SP_{arg} via a specification morphism $\sigma : SP_{par} \to SP_{arg}$ (a so-called *fitting morphism*) is defined as the specification SP'_{res} where

$$\begin{tikzcd}
SP_{arg} \arrow[r] & SP'_{res} \\
SP_{par} \arrow[u, "\sigma"] \arrow[r, "P"'] & SP_{res} \arrow[u]
\end{tikzcd}$$

is a pushout in the category of specifications $\mathcal{SPEC}$. $\square$

SP'_{res} may be defined more explicitly as follows:

$$SP'_{res} =_{def} (\textbf{translate } SP_{arg} \textbf{ by } \widehat{P}) \cup (\textbf{translate } SP_{res} \textbf{ by } \widehat{\sigma})$$

where $\widehat{P}$ and $\widehat{\sigma}$ are given by the following pushout diagram in the category of signatures $\mathcal{SIGN}$:

$$\begin{tikzcd}
\Sigma_{arg} \arrow[r, "\widehat{P}"] & \Sigma'_{res} \\
\Sigma_{par} \arrow[u, "\sigma"] \arrow[r, "P"'] & \Sigma_{res} \arrow[u, "\widehat{\sigma}"']
\end{tikzcd}$$

This still defines a function taking specifications to specifications, but the signatures of the argument and of the overall result specifications are not fixed. Further differences with respect to ASL-style parameterisation are that the argument specification is required to "semantically" fit the parameter specification SP_{par} rather than just to "syntactically" fit the parameter signature Σ_{par} as before, and that the argument specification is always explicitly included in the overall result specification.

ACT ONE [EM 85] adopts a similar style of parameterisation to that of CLEAR except that it has an additional layer of semantics. Namely, in addition to the way in which application of a parameterised specification to an argument specification builds an overall result specification as above, a parameterised specification describes a functor mapping individual models of the parameter specification to models of the result specification.

Definition 3.3 An *ACT ONE-style parameterised specification* and application of such a parameterised specification to an argument specification are defined exactly as for CLEAR-style parameterised specifications, except that the only specifications considered are presentations with equational axioms. The model-level semantics of a parameterised specification $P : SP_{par} \rightarrow SP_{res}$ is the free functor $F_P : [\![SP_{par}]\!] \rightarrow [\![SP_{res}]\!]$ (the left adjoint to the P-reduct functor $\_|_P : [\![SP_{res}]\!] \rightarrow [\![SP_{par}]\!]$). $\square$

In ACT ONE, an important issue is the compatibility of the specification-level semantics with the model-level semantics, where the model-level semantics of an unparameterised specification is the class of its initial models. It turns out that everything works out fine when the free functor defined by a parameterised specification is persistent. In this case, for any argument specification SP_{arg} and fitting morphism $\sigma : SP_{par} \to SP_{arg}$, $F_P : [\![SP_{par}]\!] \to [\![SP_{res}]\!]$ lifts to a functor $F_{\widehat{P}} : [\![SP_{arg}]\!] \to [\![SP'_{res}]\!]$ (via the amalgamation lemma) which is free with respect to $\_|_{\widehat{P}}$ and so maps the initial models of SP_{arg} to the initial models of SP'_{res}.

The model-level semantics of ACT ONE-style parameterisation has a completely different flavour from the previous parameterisation mechanisms. ACT ONE parameterised specifications are not purely construed as specification-building operations but also as tools to construct models of the overall results out of models of the arguments. This is very much like module parameterisation mechanisms in modular programming languages such as Standard ML [MacQ 86], which was the starting point for work on the Extended ML specification language [ST 85], [ST 89], [San 91], [ST 91b]. Such modularisation mechanisms provide the means to structure programs "in the large" so that programs may be decomposed into a number of (possibly generic) self-contained units with well-defined interfaces [Gog 84]. The Standard ML programming language comprises two layers: an essentially functional programming language, and a language for defining program units (*structures*), interfaces (*signatures*) and parameterised program units (*functors*). Simplifying somewhat, we can think of structures as algebras, ML signatures as algebraic signatures, and functors as *parametric algebras*, i.e. functions mapping algebras to algebras (cf. OBSCURE [LL 88] where these are called *algebra modules*). Extended ML provides a means of specifying such parametric algebras (cf. the notion of *cell* in [Sch 87]). Extended ML functor specifications are like Standard ML functors except that the result of applying the functor to an argument algebra is (loosely) specified rather than defined explicitly by means of code. To simplify the presentation, we consider only those functors in which the result includes the parameter.

Definition 3.4 An *Extended ML functor specification* has the form **functor** $F(X : SP_{par}) : SP_{res}$ where SP_{par} is a Σ_{par}-specification and SP_{res} is a Σ_{res}-specification with $\Sigma_{par} \subseteq \Sigma_{res}$. Its semantics is a function F which to any algebra $A \in [\![SP_{par}]\!]$ assigns a Σ_{res}-specification $F(A)$ with semantics defined as follows:

$$[\![F(A)]\!] = \{B \in [\![SP_{res}]\!] \mid B|_{\Sigma_{par}} = A\}$$

A Standard ML functor which maps Σ_{par}-algebras to Σ_{res}-algebras satisfies this specification iff for each algebra $A \in [\![SP_{par}]\!]$ it yields an algebra in $[\![F(A)]\!]$.

The semantics of an Extended ML functor specification F as above extends pointwise to any argument Σ_{par}-specification SP_{arg} such that $[\![SP_{arg}]\!] \subseteq [\![SP_{par}]\!]$: $F(SP_{arg})$ is a Σ_{res}-specification with semantics defined as follows:

$$[\![F(SP_{arg})]\!] = \bigcup\{[\![F(A)]\!] \mid A \in [\![SP_{arg}]\!]\}$$

$\square$

Note that the function described by an Extended ML functor specification is persistent by definition, in the sense that any algebra B in $[\![F(A)]\!]$ inherits an unmodified copy of A. For Standard ML functors, where both the argument and the body of the functor are defined by Standard ML code, this embodies the fact that the code of the argument cannot be modified by adding the additional code in the body. This constraint is retained when generalising to specifications of Standard ML functors as in Extended ML. In Extended ML, however, this may lead to inconsistency since the axioms in the result specification may impose new requirements on the argument: for some algebras $A \in [\![SP_{par}]\!]$, $[\![F(A)]\!]$ may be empty, as happens when the requirements imposed in SP_{res} are inconsistent with the particular realization of SP_{par} given by A. There will then be no Standard ML functor which satisfies the specification since such a functor must produce a result for all algebras which satisfy SP_{par}.

In this section we described a number of different parameterisation mechanisms appearing in specification languages. One obvious difference between them concerns the technicalities of parameter passing, which in ASL is based on β-reduction in a λ-calculus style, while CLEAR and ACT ONE use a pushout-based approach. Advocates of the pushout approach argue for its convenience, since for an arbitrarily large argument the overall result always includes the whole argument. This is not the case in ASL, as discussed above.

The pushout approach seems fully justified in a formalism used to gradually construct a single specification by successively adding pieces. The idea is to incorporate in this specification all potentially useful components, as otherwise they may be lost. However, a real specification language will incorporate an environment of named specifications, with explicit scoping mechanisms like those in declarative programming languages. Once a specification is introduced into the environment, it remains there and all of its components are permanently accessible. Whichever parameterisation mechanism is used, there is no danger that some components of an actual parameter will inadvertently become unavailable. If needed, they can always be taken from the environment, subject only to the constraints of the type system and the scoping mechanisms. Specification languages like Extended ML are designed to be sufficiently permissive to allow this (cf. [Tar 92]).

Such differences will to a large extent be disregarded in this paper; although they are of great importance for practical purposes, the difference is a matter of taste and convenience rather than of a more fundamental nature.

4 Parameterised specifications vs. specifications of parametric algebras

4.1 Concepts and semantic objects

An essential difference between the parameterisation approaches presented in Section 3 may best be seen if we compare the ASL-style and Extended ML-style parameterisation mechanisms. ASL-style parameterised specifications are defined entirely on the level of specifications, without any reference to the algebras which specifications are used to describe. Thus, they accept specifications as arguments and yield specifications as results. Defining the semantics of Extended ML functor specifications at this level was possible only via the more basic level of algebras: an Extended ML functor specification describes a function taking single algebras to classes of algebras, which is viewed as a definition of a class of Standard ML functors. The pointwise extension to specifications is *a posteriori*, and in fact plays no part in the Extended ML program development methodology proposed in [ST 89], [ST 91b]. CLEAR-style parameterised specifications are similar in this respect to ASL-style parameterisation, while ACT ONE-style parameterisation has elements of both embodied in its two levels of semantics. This distinction seems fundamental as it reflects the role which both kinds of specifications play in the program development process.

There are two kinds of entities involved in the process of developing a software system from a specification. On one hand we have software systems, modelled as algebras. On the other hand we have specifications which describe classes of algebras. Both software systems and specifications may (and should) be presented in a structured way, using mechanisms such as parameterisation. This gives rise to both parametric (or generic) software systems, and parameterised specifications. It is easy to confuse two distinct notions: specifications of parametric software systems, and parameterised specifications of software systems. The first is a (non-parameterised) specification of a parametric algebra. Extended ML-style functor specifications are of this kind. The second is a parameterised specification of a (non-parametric) algebra. ASL-style and CLEAR-style parameterised specifications are of this kind. Of course, it is possible to combine these two notions to obtain (for example) parameterised specifications of parametric algebras, etc.

A technical consequence of the above considerations is that the semantic objects modelling parameterised specifications and specifications of parametric algebras are quite different.

Definition 4.1 The denotation of a parameterised specification P with parameter signature Σ_{par} and result signature Σ_{res} is a *monotone* function

$$[\![P]\!] : Pow(Alg(\Sigma_{par})) \to Pow(Alg(\Sigma_{res}))$$

mapping classes of Σ_{par}-algebras to classes of Σ_{res}-algebras. The collection of all such functions is called $Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$; we write $P : Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ to indicate the "type" of P. □

We will use the notation of ASL to define parameterised specifications. The definition of the application of a parameterised specification P (Definition 3.1) yields a monotone function $[\![P]\!]$ as above (cf. Section 7.2). When needed, we will use the same notation with a parameter (requirement) specification instead of just a parameter signature (cf. Section 7). The denotation of a parameterised specification P with parameter specification SP_{par} and result signature Σ_{res} is a monotone function

$$[\![P]\!] : Pow([\![SP_{par}]\!]) \to Pow(Alg(\Sigma_{res}))$$

mapping classes of SP_{par}-models to classes of Σ_{res}-algebras.

Definition 4.2 The denotation of a specification Q of Σ_{res}-algebras parameterised by Σ_{par}-algebras is a class

$$[\![Q]\!] \in Pow(Alg(\Sigma_{par}) \to Alg(\Sigma_{res}))$$

of functions mapping Σ_{par}-algebras to Σ_{res}-algebras. The collection of all such functions is called $Spec(\Sigma_{par} \to \Sigma_{res})$; we write $Q : Spec(\Sigma_{par} \to \Sigma_{res})$ to indicate the "type" of Q. □

We will use a notation like $\Pi A{:}\Sigma_{par}.\, SP[A]$ for specifications of parametric algebras. $[\![\Pi A{:}\Sigma_{par}.\, SP[A]]\!]$ is the class of functions F which map any Σ_{par}-algebra A to an algebra $F(A) \in [\![SP[A]]\!]$. Here, $SP[A]$ is a specification in which A stands for a Σ_{par}-algebra. A can be used in SP via a new specification-building operation which turns algebras into specifications: if A is a Σ-algebra then $\{A\}$ is a Σ-specification having A as its only model. We can readily generalise this and use an arbitrary specification rather than a signature to define the class of algebras over which the "parameter" A ranges (cf. Section 7). The denotation of a specification Q of Σ_{res}-algebras parameterised by SP_{par}-models is a class

$$[\![Q]\!] \in Pow([\![SP_{par}]\!] \to Alg(\Sigma_{res}))$$

of functions mapping SP_{par}-models to Σ_{res}-algebras. Extended ML functor specifications may be modelled as specifications of parametric algebras: **functor** $F(X : SP_{par}) : SP_{res}$ specifies the class of Standard ML functors F which are parametric algebras in the class defined by $\Pi X{:}SP_{par}.\, SP_{res}$.

The following two simple examples illustrate the notions introduced above. Other examples will follow in later sections.

Example 4.3 Let

Key $=_{def}$	**enrich**	Nat	
	by	**sorts**	key
		opns	hash : key → nat

where *Nat* is as defined in Section 2, and let

$$
\begin{array}{lll}
\text{HashTable1} =_{def} & \lambda X : \mathit{Spec}(\text{Key}). & \\
& \textbf{enrich } X & \\
& \textbf{by} \quad \textbf{sorts} & \text{array} \\
& \qquad\;\; \textbf{opns} & \text{empty} : \; \to \text{array} \\
& & \text{used} : \text{nat} \times \text{array} \to \text{bool} \\
& & \text{put} : \text{nat} \times \text{key} \times \text{array} \to \text{array} \\
& & \text{get} : \text{nat} \times \text{array} \to \text{key} \\
& & \text{add} : \text{key} \times \text{array} \to \text{array} \\
& & \text{putnear} : \text{nat} \times \text{key} \times \text{array} \to \text{array} \\
& & \text{present} : \text{key} \times \text{array} \to \text{bool} \\
& & \text{searchnear} : \text{nat} \times \text{key} \times \text{array} \to \text{bool} \\
& \qquad\;\; \textbf{axioms} & \text{used}(i, \text{empty}) = \text{false} \\
& & \ldots \; \textit{(axioms for arrays)} \ldots \\
& & \text{add}(k, a) = \text{putnear}(\text{hash}(k), k, a) \\
& & \text{used}(i, a) = \text{false} \Rightarrow \text{putnear}(i, k, a) = \text{put}(i, k, a) \\
& & \text{used}(i, a) = \text{true} \wedge \text{get}(\text{hash}(k), a) = k \Rightarrow \\
& & \qquad \text{putnear}(i, k, a) = a \\
& & \text{used}(i, a) = \text{true} \wedge \text{get}(\text{hash}(k), a) \neq k \Rightarrow \\
& & \qquad \text{putnear}(i, k, a) = \text{putnear}(i + 1, k, a) \\
& & \text{present}(k, a) = \text{searchnear}(\text{hash}(k), k, a) \\
& & \text{used}(i, a) = \text{false} \Rightarrow \text{searchnear}(i, k, a) = \text{false} \\
& & \text{used}(i, a) = \text{true} \wedge \text{get}(\text{hash}(k), a) = k \Rightarrow \\
& & \qquad \text{searchnear}(i, k, a) = \text{true} \\
& & \text{used}(i, a) = \text{true} \wedge \text{get}(\text{hash}(k), a) \neq k \Rightarrow \\
& & \qquad \text{searchnear}(i, k, a) = \text{searchnear}(i + 1, k, a)
\end{array}
$$

(Each axiom is implicitly universally quantified over all its free variables; this convention will be used in examples throughout the rest of this paper.)

HashTable1 is a parameterised specification. Given a specification *SP* describing a particular choice for keys and perhaps constraining *hash* in some fashion (e.g. to have a particular subset of the natural numbers as its range), *HashTable1*(*SP*) is a specification of hash tables containing such keys and using such a hash function. For instance, let

$$
\begin{array}{lll}
\text{String} =_{def} & \textbf{enrich } \text{Nat} & \\
& \textbf{by} \quad \textbf{sorts} & \text{string} \\
& \qquad\;\; \textbf{opns} & \text{nil} : \; \to \text{string} \\
& & \text{a} : \; \to \text{string} \\
& & \text{b} : \; \to \text{string} \\
& & \text{c} : \; \to \text{string} \\
& & \cdot : \text{string} \times \text{string} \to \text{string} \\
& & \text{hash} : \text{string} \to \text{nat} \\
& \qquad\;\; \textbf{axioms} & s \cdot \text{nil} = s \\
& & \text{nil} \cdot s = s \\
& & s \cdot (t \cdot v) = (s \cdot t) \cdot v \\
& & \text{hash}(\text{nil}) = 0
\end{array}
$$

Let Σ_{Key} and Σ_{String} be the signatures of *Key* and *String* respectively, and let $\sigma : \Sigma_{Key} \to \Sigma_{String}$ be the obvious signature morphism (mapping *key* to *string* and leaving the rest of the signature of *Key* unchanged). Then

$$\text{HashString} \;=_{def}\; \text{HashTable1}(\textbf{derive from } \text{String } \textbf{by } \sigma)$$

specifies hash tables containing strings, with the hash function constrained so that the empty string is hashed into position 0 of the table. Note that adding further axioms to *String* which constrain its class of models and then repeating the construction of *HashString* with this new version of *String* would yield a specification with fewer models than *HashString* as above. This demonstrates the monotonicity of the function which *HashTable* denotes.

The above example can be made slightly more realistic by adding a use of **derive** to the body of *HashTable1* to hide some operations (e.g. *putnear*) and to change the name of the sort *array* to *hashtable*. □

Example 4.4 Let

$$\begin{array}{lllll}
\text{HashTable2} & =_{def} & \Pi A : \text{Key}. & & \\
 & & \quad\textbf{enrich} & \{A\} & \\
 & & \quad\textbf{by} & \textbf{sorts} & \text{array} \\
 & & & \textbf{opns} & \text{empty} : \;\to \text{array} \\
 & & & & \text{used} : \text{nat} \times \text{array} \to \text{bool} \\
 & & & & \ldots\ \textit{(as in the previous example)}\ \ldots \\
 & & & \textbf{axioms} & \text{used}(i, \text{empty}) = \text{false} \\
 & & & & \ldots\ \textit{(as in the previous example)}\ \ldots
\end{array}$$

where *Key* is as in Example 4.3. *HashTable2* is a specification of a parametric algebra. If H is in the class defined by *HashTable2*, then H is a function mapping algebras to algebras. For any algebra A satisfying the specification *Key*, $H(A)$ is an algebra which realizes hash tables containing the keys in A and using the hash function in A. $H(A)$ is required to satisfy the axioms in the body of *HashTable2* for any A satisfying *Key*, so any H satisfying *HashTable2* will be a universally applicable parameterised implementation of hash tables, in the sense that it is required to exhibit correct behaviour for any choice of keys and any choice of hash function over those keys. □

The above two examples illustrate the essential difference of intention underlying (instantiated) parameterised specifications vs. specifications of parametric algebras. To realize *HashString* (Example 4.3), the implementor must provide an implementation of hash tables for a hash function of his/her choice (subject only to the constraint that $hash(nil) = 0$). To realize *HashTable2* (Example 4.4), the implementor must provide an implementation of hash tables which works for *any* hash function, since the hash function will be supplied later as an argument.

4.2 The Galois connection

There is a natural connection between the semantic domains of parameterised specifications and specifications of parametric algebras having the same parameter and result signatures. This relationship is captured as follows. (The technicalities below do not depend on the fact that we deal with the meanings of specifications here; we just apply some standard ideas of lattice theory [Bir 48] to our specific concepts.) Note that all of the following extends to the case where we have a parameter specification instead of just a parameter signature.

Recall that $Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ stands for the function space $Pow(Alg(\Sigma_{par})) \to Pow(Alg(\Sigma_{res}))$ with elements $\mathcal{P}$ (these are monotonic functions corresponding to the denotations of parameterised specifications P). Similarly, $\mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$ stands for $\mathit{Pow}(\mathit{Alg}(\Sigma_{par}) \to \mathit{Alg}(\Sigma_{res}))$ with elements $\mathcal{Q}$ (these correspond to the denotations of specifications Q of parametric algebras). As usual, in examples we will avoid stressing the distinction between a (parameterised) specification and its denotation, and use the same name to refer to both.

Definition 4.5 For any $\mathcal{P} \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$, let $\mathcal{P}^\sharp \in \mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$ be defined by

$$\mathcal{P}^\sharp = \{F : \mathit{Alg}(\Sigma_{par}) \to \mathit{Alg}(\Sigma_{res}) \mid \text{for all } A \in \mathit{Alg}(\Sigma_{par}),\ F(A) \in \mathcal{P}(\{A\})\}.$$

For any $\mathcal{Q} \in \mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$, let $\mathcal{Q}^\dagger \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$ be defined by

$$\mathcal{Q}^\dagger(C) = \{F(A) \mid F \in \mathcal{Q}, A \in C\}$$

for any class C of Σ_{par}-algebras. □

Example 4.6 Recall the parameterised specification *HashTable1* in Example 4.3 and the specification *HashTable2* in Example 4.4. We have:

$$\begin{aligned} \mathrm{HashTable1}^\sharp &= \mathrm{HashTable2} \\ \mathrm{HashTable2}^\dagger &= \mathrm{HashTable1} \end{aligned}$$

□

Proposition 4.7

1. $\mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$ forms a complete lattice with the set inclusion ordering.
2. $\mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$ forms a complete lattice with the natural extension of the set inclusion ordering on $\mathit{Spec}(\Sigma_{res}) = \mathit{Pow}(\mathit{Alg}(\Sigma_{res}))$: for $\mathcal{P}_1, \mathcal{P}_2 \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$, $\mathcal{P}_1 \leq \mathcal{P}_2$ iff for all $C \subseteq \mathit{Alg}(\Sigma_{par})$, $\mathcal{P}_1(C) \subseteq \mathcal{P}_2(C)$.
3. For $\mathcal{Q}_1, \mathcal{Q}_2 \in \mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$, if $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$ then $\mathcal{Q}_1^\dagger \leq \mathcal{Q}_2^\dagger$.
4. For $\mathcal{P}_1, \mathcal{P}_2 \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$, if $\mathcal{P}_1 \leq \mathcal{P}_2$ then $\mathcal{P}_1^\sharp \subseteq \mathcal{P}_2^\sharp$.
5. For $\mathcal{Q} \in \mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$, $\mathcal{Q} \subseteq (\mathcal{Q}^\dagger)^\sharp$.
6. For $\mathcal{P} \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$, $\mathcal{P} \geq (\mathcal{P}^\sharp)^\dagger$.

Thus we have defined a Galois connection [Bir 48].
Proof Immediate from the definitions, but note that 6 relies on the monotonicity of $\mathcal{P}$. □

Corollary 4.8 The maps $(\_)^\sharp$ and $(\_)^\dagger$ defined above are isomorphisms between the sublattices of $\mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$ and $\mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$ consisting of their closed elements, i.e. of parameterised specifications of the form $(\mathcal{P}^\sharp)^\dagger$ and of specifications of parametric algebras of the form $(\mathcal{Q}^\dagger)^\sharp$, respectively. □

Definition 4.9 $\mathcal{Q} \in \mathit{Spec}(\Sigma_{par} \to \Sigma_{res})$ is called *regular* if for every family of functions $F_A \in \mathcal{Q}$, $A \in \mathit{Alg}(\Sigma_{par})$, the function $F : \mathit{Alg}(\Sigma_{par}) \to \mathit{Alg}(\Sigma_{res})$ defined by $F(A) = F_A(A)$ for $A \in \mathit{Alg}(\Sigma_{par})$ is an element of $\mathcal{Q}$ as well. □

Proposition 4.10 $\mathcal{Q} \in Spec(\Sigma_{par} \to \Sigma_{res})$ is regular iff it is closed, i.e. iff $\mathcal{Q} = (\mathcal{Q}^{\dagger})^{\sharp}$.
Proof ($\Leftarrow$): Directly from the definition; for all $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$, $\mathcal{P}^{\sharp}$ is regular.
($\Rightarrow$, $\subseteq$): By Proposition 4.7.5.
($\Rightarrow$, $\supseteq$): Consider any $F : Alg(\Sigma_{par}) \to Alg(\Sigma_{res})$ such that $F(A) \in \mathcal{Q}^{\dagger}(\{A\})$ for all $A \in Alg(\Sigma_{par})$. The definition of $\mathcal{Q}^{\dagger}$ implies now that for all $A \in Alg(\Sigma_{par})$, there is $F_A \in \mathcal{Q}$ such that $F(A) = F_A(A)$. Thus $F \in \mathcal{Q}$ since $\mathcal{Q}$ is regular. □

Definition 4.11 $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ is called *additive* if for every class $C \subseteq Alg(\Sigma_{par})$, $\mathcal{P}(C) = \bigcup\{\mathcal{P}(\{A\}) \mid A \in C\}$. □

Fact 4.12 If $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ is not additive then $P \gneq (\mathcal{P}^{\sharp})^{\dagger}$. □

Definition 4.13 $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ is called *globally inconsistent* if $\mathcal{P}(C) = \emptyset$ for all $C \subseteq Alg(\Sigma_{par})$. □

Definition 4.14 $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ is said to *preserve consistency* if $\mathcal{P}(C) \neq \emptyset$ for all non-empty $C \subseteq Alg(\Sigma_{par})$. □

Fact 4.15 If $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ does not preserve consistency then $\mathcal{P}^{\sharp}$ is inconsistent and $(\mathcal{P}^{\sharp})^{\dagger}$ is globally inconsistent. □

Proposition 4.16 $\mathcal{P} \in Spec(\Sigma_{par}) \to Spec(\Sigma_{res})$ is closed, i.e. $\mathcal{P} = (\mathcal{P}^{\sharp})^{\dagger}$, iff either $\mathcal{P}$ is globally inconsistent, or $\mathcal{P}$ is additive and preserves consistency.
Proof ($\Rightarrow$): Directly from the definition. For all $\mathcal{Q} \in Spec(\Sigma_{par} \to \Sigma_{res})$, $\mathcal{Q}^{\dagger}$ is additive. Moreover, either $\mathcal{Q} = \emptyset$, and then $\mathcal{Q}^{\dagger}$ is globally inconsistent, or $\mathcal{Q} \neq \emptyset$, and then $\mathcal{Q}^{\dagger}$ preserves consistency.
($\Leftarrow$): If $\mathcal{P}$ is globally inconsistent, then $\mathcal{P}^{\sharp} = \emptyset$ and so $\mathcal{P} = (\mathcal{P}^{\sharp})^{\dagger}$ trivially. Otherwise, by additivity of $\mathcal{P}$ and of $(\mathcal{P}^{\sharp})^{\dagger}$ (the latter follows from the proof of the opposite implication, above) and by Proposition 4.7.6, it is enough to show that $\mathcal{P}(\{A\}) \subseteq (\mathcal{P}^{\sharp})^{\dagger}(\{A\})$. For all $A' \in Alg(\Sigma_{par})$, $\mathcal{P}(\{A'\}) \neq \emptyset$, since $\mathcal{P}$ preserves consistency. Consequently, for all $B \in \mathcal{P}(\{A\})$, there is $F \in \mathcal{P}^{\sharp}$ such that $F(A) = B$, and thus $B \in (\mathcal{P}^{\sharp})^{\dagger}(\{A\})$. □

Corollary 4.17 There is a 1-1 correspondence between consistent regular specifications of parametric algebras and additive parameterised specifications which preserve consistency. □

Example 4.18 The parameterised specification *HashTable1* in Example 4.3 is additive and preserves consistency. The specification *HashTable2* in Example 4.4 is regular and consistent. In fact, *HashTable1* and *HashTable2* are in the correspondence mentioned in Corollary 4.17 (cf. Example 4.6). □

Some explanation of the properties embodied in the above technicalities will make their significance clearer.

$\mathcal{Q} \in Spec(\Sigma_{par} \to \Sigma_{res})$ is regular iff it is a Cartesian product of a family of classes of algebras indexed by Σ_{par}-algebras. This is a natural condition which is met by all the specifications we are going to write (and indeed by all specifications expressible in Extended ML). The Π-notation used above and later formally introduced in Section 7 for defining specifications of parametric algebras has a semantics which may be decomposed into two stages. Consider $\Pi A{:}\Sigma_{par}.\, SP[A]$, where $SP[A]$ is a Σ_{res}-specification. First, this directly defines a function $\mathcal{F} : Alg(\Sigma_{par}) \to Pow(Alg(\Sigma_{res}))$ which maps any Σ_{par}-algebra A to the class of algebras $[\![SP[A]]\!]$. This is then used to determine a class of functions $\Pi_{A \in Alg(\Sigma_{par})}\mathcal{F}(A) \in Spec(\Sigma_{par} \to \Sigma_{res})$ (Π is used here as the Cartesian product symbol) which is the meaning of the specification $\Pi A{:}\Sigma_{par}.\, SP[A]$. Clearly, any specification defined in this way is regular. Violating regularity would require use of specification mechanisms which constrain the instantiation of a parametric algebra on an argument by relating it to the instantiation of this parametric algebra on other arguments. For example, the "stability" of Standard ML functors (see [Sch 87], [ST 89], [ST 92]) is such a constraint. Thus, if a specification language provided a way to require stability of parametric algebras (Standard ML functors) it might allow non-regular classes of functors to be specified.

With regard to additivity of parameterised specifications, we have assumed that all specification-building operations (and therefore also parameterised specifications) are monotonic on model classes. This ensures that $\mathcal{P}(C) \supseteq \bigcup\{\mathcal{P}(\{A\}) \mid A \in C\}$ for every $\mathcal{P} \in \mathit{Spec}(\Sigma_{par}) \to \mathit{Spec}(\Sigma_{res})$ and $C \subseteq \mathit{Alg}(\Sigma_{par})$. Thus, the only way for a parameterised specification to be non-additive is for the opposite inclusion to fail. This may happen if it operates on the class of models of its argument as a whole rather than "pointwise". The following example of an ASL parameterised specification shows the difference.

Example 4.19 Let

$$\begin{array}{lll} \Sigma_{\mathrm{ab}} =_{def} & \textbf{sorts} & \mathrm{s} \\ & \textbf{opns} & \mathrm{a} :\to \mathrm{s} \\ & & \mathrm{b} :\to \mathrm{s} \end{array}$$

and

$$\begin{array}{lll} \Sigma_{\mathrm{c}} =_{def} & \textbf{sorts} & \mathrm{s} \\ & \textbf{opns} & \mathrm{c} :\to \mathrm{s} \end{array}$$

with signature morphisms $\sigma_{ca}, \sigma_{cb} : \Sigma_c \to \Sigma_{ab}$ where $\sigma_{ca}(s) = \sigma_{cb}(s) = s$, $\sigma_{ca}(c) = a$ and $\sigma_{cb}(c) = b$. Now, consider the ASL parameterised specification

$$\mathrm{P} =_{def} \lambda X{:}\mathit{Spec}(\Sigma_{\mathrm{ab}}).\, (\textbf{derive from } X \textbf{ by } \sigma_{\mathrm{ca}}) \cup (\textbf{derive from } X \textbf{ by } \sigma_{\mathrm{cb}}).$$

P is *not* additive. Consider two Σ_{ab}-algebras, A, B with $A_s = B_s = \{0,1\}$ and $a_A = 0, b_A = 1, a_B = 1, b_B = 0$. It is easy to see that $[\![P]\!](\{A\}) = [\![P]\!](\{B\}) = \emptyset$ while $[\![P]\!](\{A,B\}) = \{C, D\}$ where $C_s = D_s = \{0,1\}$, $c_C = 0$ and $c_D = 1$. □

It is interesting to observe that each of the ASL specification-building operations used in this example (and all those defined in Section 2) are additive. Paradoxically, it was nevertheless possible to use them to form a non-additive parameterised specification. The reason is that there is a hidden non-additive operation built into the notation, namely the "diagonalisation" function which allows an argument to be used repeatedly as in $\lambda X{:}\mathit{Spec}(\Sigma).\, (\ldots X \ldots X \ldots)$. It is also possible to imagine full-fledged specification-building operations which are non-additive. For example, one might like to define a class of algebras by first specifying the boundaries of the admissible behaviour and then applying a specification-building operation which fills in all those algebras which exhibit behaviour within these boundaries. Such an operation would be non-additive.

Example 4.19 is also an example of a parameterised specification which does not preserve consistency (e.g. $[\![P]\!](\{A\}) = \emptyset$) without being globally inconsistent (e.g. $[\![P]\!](\{A,B\}) \neq \emptyset$). Among the specification-building operations defined in Section 2, **derive**, **translate** (with respect to injective signature morphisms), **iso-close** and **abstract** preserve consistency. A parameterised specification preserves consistency iff it is in principle realizable by a parametric algebra: for every model of the argument specification there is a model of the result specification. The **use** operation in the PLUSS specification language is explicitly designed to have this property, as described in [Bid 88] (cf. [GM 88]).

Example 4.20 Let Σ_{Ord} be the signature obtained by adding a sort *elem* and an operation $lt : elem \times elem \to bool$ to the signature of the specification *Bool* defined in Section 2. Then, let

$$
\begin{array}{lllll}
\text{Sort} & =_{def} & \lambda X : \mathit{Spec}(\Sigma_{\text{Ord}}). & & \\
 & & \textbf{enrich} & X & \\
 & & \textbf{by} & \textbf{sorts} & \text{list} \\
 & & & \textbf{opns} & \text{nil} : \ \to \text{list} \\
 & & & & \text{cons} : \text{elem} \times \text{list} \to \text{list} \\
 & & & & \text{is-in} : \text{elem} \times \text{list} \to \text{bool} \\
 & & & & \text{is-sorted} : \text{list} \to \text{bool} \\
 & & & & \text{sort} : \text{list} \to \text{list} \\
 & & & \textbf{axioms} & \text{is-in}(x, \text{nil}) = \text{false} \\
 & & & & \text{is-in}(x, \text{cons}(x, l)) = \text{true} \\
 & & & & x \neq y \Rightarrow \text{is-in}(x, \text{cons}(y, l)) = \text{is-in}(x, l) \\
 & & & & \text{is-sorted}(\text{nil}) = \text{true} \\
 & & & & \text{is-sorted}(\text{cons}(x, l)) = \text{true} \Leftrightarrow \\
 & & & & \qquad (\forall y : \text{elem. is-in}(y, l) = \text{true} \Rightarrow \text{lt}(y, x) = \text{false}) \\
 & & & & \qquad \wedge \text{is-sorted}(l) = \text{true} \\
 & & & & \text{is-in}(x, l) = \text{is-in}(x, \text{sort}(l)) \\
 & & & & \text{is-sorted}(\text{sort}(l)) = \text{true}
\end{array}
$$

Sort is a parameterised specification. Given a specification of data elements with a binary relation, it builds a specification of lists of these elements together with an operation which sorts lists with respect to the given relation.

Here are some specifications which constitute admissible arguments for *Sort*:

$$
\begin{array}{lllll}
\text{Ord} & =_{def} & \textbf{enrich} & \text{Bool} & \\
 & & \textbf{by} & \textbf{sorts} & \text{elem} \\
 & & & \textbf{opns} & \text{lt} : \text{elem} \times \text{elem} \to \text{bool} \\
 & & & \textbf{axioms} & \text{lt}(x, y) = \text{true} \Rightarrow \text{lt}(y, x) = \text{false} \\
 & & & & \text{lt}(x, y) = \text{true} \wedge \text{lt}(y, z) = \text{true} \Rightarrow \text{lt}(x, z) = \text{true}
\end{array}
$$

This specifies a strict (i.e. irreflexive) partial ordering. Then, *Sort*(*Ord*) is a specification of topological sorting.

$$
\begin{array}{lllll}
\text{LOrd} & =_{def} & \textbf{enrich} & \text{Ord} & \\
 & & \textbf{by} & \textbf{axioms} & \text{lt}(x, y) = \text{false} \wedge \text{lt}(y, x) = \text{false} \Rightarrow x = y
\end{array}
$$

This specifies a total (linear) ordering. Then, *Sort*(*LOrd*) is a specification of ordinary sorting, except that the *sort* operation is permitted to remove duplicate elements.

$$
\begin{array}{lllll}
\text{Tot} & =_{def} & \textbf{enrich} & \text{Bool} & \\
 & & \textbf{by} & \textbf{sorts} & \text{elem} \\
 & & & \textbf{opns} & \text{lt} : \text{elem} \times \text{elem} \to \text{bool} \\
 & & & \textbf{axioms} & \text{lt}(x, y) = \text{true} \\
 & & & & \exists x, y : \text{elem}.\ x \neq y
\end{array}
$$

This specifies a "total" relation with at least two different values of sort *elem*. Then, *Sort*(*Tot*) is inconsistent! (Suppose a and b are two different values of sort *elem*; then according to the axioms for *is-in* and the penultimate axiom in the body of *Sort*, we have that $sort(cons(a, cons(b, nil)))$ is either $cons(a, cons(b, nil))$ or $cons(b, cons(a, nil))$, or a list with duplicates for which the same argument applies. Since the first axiom of *Tot* requires that $lt(a, b) = lt(b, a) = true$, we have *is-sorted*$(cons(a, cons(b, nil))) = false$ and *is-sorted*$(cons(b, cons(a, nil))) = false$, so neither of these two values satisfies the last axiom of *Tot*.) Thus *Sort* does not preserve consistency although it is not globally inconsistent.

Let $SP_{Sort}[X]$ denote the body of *Sort*. Then $Sort^\sharp$ may be given explicitly as follows:

$$\mathrm{Sort}^\sharp \quad = \quad \Pi A{:}\Sigma_{\mathrm{Ord}}.\, \mathrm{SP}_{\mathrm{Sort}}[\{A\}]$$

Since *Sort* does not preserve consistency, $Sort^\sharp$ is inconsistent and $(Sort^\sharp)^\dagger$ is globally inconsistent (Fact 4.15). Thus $Sort \gneqq (Sort^\sharp)^\dagger$. Intuitively, $Sort^\sharp$ is a specification of a parametric algebra which for any set of elements with a binary relation builds an algebra of lists of elements together with, among others, an operation to sort lists with respect to the relation. Unfortunately, for non-antisymmetric binary relations a sorting operation satisfying the axioms in *Sort* cannot exist, and so $Sort^\sharp$ cannot be implemented ($Sort^\sharp$ is inconsistent).

We can modify *Sort*, restricting the range of admissible parameters so that the binary relation *lt* is forced to be a (strict) partial ordering:

$$\widehat{\mathrm{Sort}} \quad =_{def} \quad \lambda X{:}Spec(\mathrm{Ord}).\, \mathrm{SP}_{\mathrm{Sort}}[X]$$

Then we have

$$\widehat{\mathrm{Sort}}^\sharp \quad = \quad \Pi A{:}Spec(\mathrm{Ord}).\, \mathrm{SP}_{\mathrm{Sort}}[\{A\}]$$

It is easy to check that on the domain determined by *Ord* as above, $\widehat{Sort}$ preserves consistency and is additive, and so (Proposition 4.16) $Sort = (\widehat{Sort}^\sharp)^\dagger$.

□

4.3 Methodological consequences

The upshot of the above deliberations is that there are two distinct things: parameterised specifications, and specifications of parametric algebras. Corollary 4.17 characterises the proper subclasses of these two classes which essentially coincide. The properties of additivity and preserving consistency characterise the class of parameterised specifications which can be adequately viewed as consistent specifications of parametric algebras. Regularity of a specification of a parametric algebra ensures that it can be regarded as a parameterised specification. This does not mean that these properties are to be viewed as requirements to be imposed on all parameterised specifications and specifications of parametric algebras. We believe there is a role for both non-additive and non-consistency-preserving parameterised specifications in the process of software development. Non-regular specifications of parametric algebras are useful as well, although in Extended ML [ST 89], [ST 91b] we decided to introduce the only potential non-regular specification-building operation (the requirement of stability) at a different level.

Corollary 4.17 is not meant to suggest either that the subclasses of specifications having these properties should be identified. There is an important methodological distinction between parameterised specifications and specifications of parametric algebras. Parameterised specifications are tools for building requirements specifications in a structured way. The structure which is thereby introduced makes the requirements specification easier to understand, reason about and use; it is not meant to impose any restriction on the structure of the eventual implementation. In contrast, specifications of parametric algebras are used in the process of designing an implementation. They are introduced for the express purpose of imposing structure on the desired implementation, breaking the problem into self-contained chunks which may be tackled independently. Once the job is completed, the result is a collection of self-contained modules with precisely-specified interfaces, all of which may later be reused in other systems.

The structure of the requirements specification may suggest a possible way of decomposing the specified task into subtasks. However, if the parameterised specifications involved do not preserve consistency then it will not be possible to provide implementations of the corresponding subtasks (Fact 4.15). In this case it is necessary to seek an alternative way of decomposing the problem. Less dangerously, if the parameterised specifications involved are not additive then imposing the structure of the specification on the solution may exclude some of the models of the original requirements specification (Fact 4.12). In this case, it may be useful to seek an alternative way of decomposing the problem which admits more, possibly better, solutions. Even when all the parameterised specifications involved are both additive and preserve consistency, the structure of the requirements specification may not be the best structure for the implementation (for example, for efficiency reasons) and so the implementation team should not be compelled to use it. Thus, in any case, the designer should not be forbidden from seeking an alternative way of decomposing the problem; the need to eventually obtain an algebra which realizes the requirements specification should be the only constraint. See [FJ 90] for a similar conclusion supported by evidence from a practical example.

Example 4.21 The following simple example illustrates some of the points made above.
Let

$$
\begin{array}{lll}
\mathrm{Bunch} & =_{def} & \textbf{reachable} \\
 & & \quad \textbf{enrich}\ \mathrm{Nat} \\
 & & \quad \textbf{by}\ \ \begin{array}[t]{ll}
\textbf{sorts} & \mathrm{elem}, \mathrm{bunch} \\
\textbf{opns} & \mathrm{empty} : \ \to \mathrm{bunch} \\
 & \mathrm{add} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
 & \mathrm{removeone} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
 & \mathrm{count} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{nat} \\
\textbf{axioms} & \mathrm{count}(a, \mathrm{empty}) = 0 \\
 & \mathrm{count}(a, \mathrm{add}(a, B)) > 0 = \mathrm{true} \\
 & a \neq b \Rightarrow \mathrm{count}(a, \mathrm{add}(b, B)) = \mathrm{count}(a, B) \\
 & \mathrm{count}(a, B) = 0 \Rightarrow \mathrm{count}(a, \mathrm{removeone}(a, B)) = 0 \\
 & \mathrm{count}(a, B) \neq 0 \Rightarrow \\
 & \qquad \mathrm{count}(a, \mathrm{removeone}(a, B)) = \mathrm{count}(a, B) - 1 \\
 & a \neq b \Rightarrow \mathrm{count}(b, \mathrm{removeone}(a, B)) = \mathrm{count}(b, B)
\end{array} \\
 & & \textbf{on}\ \{\mathrm{bunch}\}
\end{array}
$$

Bunch is a generalisation of finite sets, bags, lists, etc. The intention is that *count* counts the number of occurrences of a given element in a bunch and *removeone* removes one occurrence of an element from a bunch. The operation *add* is not constrained except that adding an element to a bunch does not change the number of occurrences of other elements in the bunch, and leaves in the bunch at least one occurrence of the indicated element. In a realization of bunches using bags or lists, *add* would add one occurrence of the indicated element. There are also realizations of bunches in which *add* would add more than one occurrence of the element, and even realizations in which, under some circumstances, *add* would decrease the number of occurrences of the "added" element (as long as it does not remove them all).

$$
\begin{array}{lll}
\mathrm{Set} & =_{def} & \textbf{reachable} \\
& & \quad \textbf{enrich } \mathrm{Nat} \\
& & \quad \textbf{by} \quad \begin{array}[t]{ll}
\textbf{sorts} & \mathrm{elem}, \mathrm{bunch} \\
\textbf{opns} & \mathrm{empty} : \ \to \mathrm{bunch} \\
& \mathrm{add} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
& \mathrm{removeone} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
& \mathrm{count} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{nat} \\
\textbf{axioms} & \mathrm{add}(a, \mathrm{add}(b, B)) = \mathrm{add}(b, \mathrm{add}(a, B)) \\
& \mathrm{add}(a, \mathrm{add}(a, B)) = \mathrm{add}(a, B) \\
& \mathrm{count}(a, \mathrm{empty}) = 0 \\
& \mathrm{count}(a, \mathrm{add}(a, B)) = 1 \\
& a \neq b \Rightarrow \mathrm{count}(a, \mathrm{add}(b, B)) = \mathrm{count}(a, B) \\
& \mathrm{count}(a, B) = 0 \Rightarrow \mathrm{removeone}(a, B) = B \\
& \mathrm{count}(a, B) = 0 \Rightarrow \mathrm{removeone}(a, \mathrm{add}(a, B)) = B
\end{array} \\
& & \textbf{on } \{\mathrm{bunch}\}
\end{array}
$$

Set is a specialization of *Bunch* (in the sense that $[\![Set]\!] \subseteq [\![Bunch]\!]$) in which each element occurs at most once in a bunch (so *count* is never greater than 1) and *add* is required to be idempotent and commutative.

$$
\mathrm{Delete} =_{def} \lambda X{:}Spec(\mathrm{Bunch}).\ \begin{array}[t]{l}
\textbf{enrich } X \\
\textbf{by} \quad \begin{array}[t]{ll}
\textbf{opns} & \mathrm{delete} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
\textbf{axioms} & \mathrm{count}(a, \mathrm{delete}(a, B)) = 0 \\
& a \neq b \Rightarrow \mathrm{count}(a, \mathrm{delete}(b, B)) = \mathrm{count}(a, B)
\end{array}
\end{array}
$$

The parameterised specification *Delete* takes any specification *SP* with $[\![SP]\!] \subseteq [\![Bunch]\!]$ and adds an operation *delete* which removes all occurrences of the indicated element from a bunch. Note that *Delete* is additive and preserves consistency.

The parameterised specification *Delete* may be applied to the specification *Bunch*:

$$
\mathrm{BunchDelete} =_{def} \mathrm{Delete}(\mathrm{Bunch})
$$

Suppose that *BunchDelete* is the requirements specification we are to implement. As a requirements specification, this conveys exactly the same information as the following equivalent non-parameterised specification:

$$
\begin{array}{l}
\textbf{enrich} \quad \textbf{reachable} \\
\qquad\qquad \textbf{enrich } \mathrm{Nat} \\
\qquad\qquad \textbf{by} \quad \begin{array}[t]{ll}
\textbf{sorts} & \mathrm{elem}, \mathrm{bunch} \\
\textbf{opns} & \mathrm{empty} : \ \to \mathrm{bunch} \\
& \mathrm{add} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
& \mathrm{removeone} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
& \mathrm{count} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{nat} \\
\textbf{axioms} & \ldots
\end{array} \\
\qquad\quad \textbf{on } \{\mathrm{bunch}\} \\
\textbf{by} \quad \begin{array}[t]{ll}
\textbf{opns} & \mathrm{delete} : \mathrm{elem} \times \mathrm{bunch} \to \mathrm{bunch} \\
\textbf{axioms} & \ldots
\end{array}
\end{array}
$$

The requirements specification *BunchDelete* is correctly realized by any model of the following enrichment of *Set*:

$$\begin{array}{lll} \mathrm{SetDelete} & =_{def} & \textbf{enrich}\ \mathrm{Set} \\ & & \textbf{by}\ \ \textbf{opns}\quad \mathrm{delete} : \mathrm{elem} \times \mathrm{bunch} \rightarrow \mathrm{bunch} \\ & & \qquad \textbf{axioms}\quad \mathrm{delete}(a, B) = \mathrm{removeone}(a, B) \end{array}$$

The definition of *delete* as *removeone* takes advantage of the fact that in *Set*, each element occurs at most once in a bunch.

Now, suppose we view the definition

$$\mathrm{BunchDelete} =_{def} \mathrm{Delete}(\mathrm{Bunch})$$

as a design specification which decomposes the task of implementing *BunchDelete* into two subtasks:

1. Implement *Bunch*.

2. Implement $Delete^\sharp$, where $Delete^\sharp$ is given explicitly as follows:

$$\begin{array}{lll} \mathrm{Delete}^\sharp = \Pi A{:}\mathrm{Bunch}. & \textbf{enrich}\ \{\mathrm{A}\} & \\ & \textbf{by}\ \ \textbf{opns} & \mathrm{delete} : \mathrm{elem} \times \mathrm{bunch} \rightarrow \mathrm{bunch} \\ & \qquad \textbf{axioms} & \mathrm{count}(a, \mathrm{delete}(a, B)) = 0 \\ & & a \neq b \Rightarrow \mathrm{count}(a, \mathrm{delete}(b, B)) = \mathrm{count}(a, B) \end{array}$$

Suppose we regard *SetDelete* as consisting of two separate parts: an implementation of *Bunch* (i.e. *Set*) together with an enrichment of this which adds *delete* defined as *removeone*. This is not a correct implementation of the design specification. The first part is indeed a correct implementation of *Bunch*, but the second part is not a correct implementation of $Delete^\sharp$. It works only in the context of the particular implementation of *Bunch* we have chosen since it takes advantage of some of its properties which are not shared by other possible implementations. Thus it disobeys the principle of modular decomposition by which separate modules are to be implemented independently. The natural implementation of $Delete^\sharp$ would define *delete* to repeatedly apply *removeone* until all occurrences of the given element are removed. Applying this implementation to any model of *Set* gives the same algebra as the extension embodied in *SetDelete* (this does not happen in all examples — see Example 4.22) but the "code" would be quite different. Most significantly, the modular version would be reusable in other contexts since its correctness is preserved under a change of the implementation of *Bunch*. □

Example 4.22 A variation on the above example is the following. Consider an enrichment $\widehat{Bunch}$ of *Bunch* above by operations to choose an element of a non-empty bunch, to test whether a bunch is empty, and to form the union of two bunches. Let $\widehat{Set}$ be an analogous enrichment of *Set*, and let $\widehat{Delete}$ be like *Delete* but with the range of admissible parameters determined by $\widehat{Bunch}$ rather than by *Bunch*. Then

$$\begin{array}{lll} \widehat{\mathrm{Delete}}^\sharp = \Pi A{:}\widehat{\mathrm{Bunch}}. & \textbf{enrich}\ \{\mathrm{A}\} & \\ & \textbf{by}\ \ \textbf{opns} & \mathrm{delete} : \mathrm{elem} \times \mathrm{bunch} \rightarrow \mathrm{bunch} \\ & \qquad \textbf{axioms} & \ldots \end{array}$$

One possible implementation of $\widehat{Delete}^\sharp$ would code $delete(a, B)$ by first initialising the result to *empty*, and then, while B is non-empty, choosing an element of B, removing it from B, and if the chosen element is different from a, adding it to the result. Such an implementation is quite natural under an assumption that the *count* operation is relatively "expensive" and the other operations are relatively "cheap" — which is quite likely under (for example) the most obvious list implementation of bunches.

Applying this implementation of $\widehat{Delete}^{\sharp}$ to an implementation of $\widehat{Set}$ yields, of course, an implementation of $\widehat{Delete}(\widehat{Set})$.

However, if we were to realize $\widehat{Delete}(\widehat{Set})$ directly, even assuming that we use a similar idea for the implementation, then the most natural algorithm for $delete(a, B)$ would proceed as above only to the point where the first (and only) occurrence of a in B is encountered. Then the search would stop, and the union of the result accumulated so far with the remainder of B would be the answer (since these two parts are disjoint, forming their union is cheap). The algebra thus obtained would be different from that obtained by instantiating the parametric implementation of $\widehat{Delete}^{\sharp}$ sketched above. □

5 Higher-order parameterisation

The ASL-style λ-calculus approach to parameterised specifications, outlined in Section 3, extends naturally to a higher-order parameterisation mechanism. (A very limited form of this in the pushout approach to parameterisation is provided by *parameterised parameter passing* [EM 85].) By allowing both the arguments and results of parameterised specifications to be parameterised specifications of an arbitrary complexity, as suggested in [ST 88a], we obtain a hierarchy of higher-order parameterised specifications. This hierarchy is indexed by a class of types $\mathcal{T}$ defined as the least class such that:

- For any signature Σ, $\Sigma \in \mathcal{T}$; and
- For any types $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \rightarrow \tau_2 \in \mathcal{T}$.

Definition 5.1 The semantic domain $ParSpec(\tau)$ of (denotations of) parameterised specifications of type $\tau \in \mathcal{T}$ is defined inductively as follows:

- If $\tau = \Sigma$, then $ParSpec(\tau) =_{def} Pow(Alg(\Sigma))$ (ordered by inclusion); this will be written as $Spec(\Sigma)$ as well.
- If $\tau = \tau_1 \rightarrow \tau_2$, then $ParSpec(\tau) =_{def} ParSpec(\tau_1) \rightarrow ParSpec(\tau_2)$, the class of monotone functions from $ParSpec(\tau_1)$ to $ParSpec(\tau_2)$ (ordered by pointwise extension of the ordering on $ParSpec(\tau_2)$). □

This covers first-order parameterisation (see Definition 4.1) since $ParSpec(\Sigma_1 \rightarrow \Sigma_2) = Spec(\Sigma_1) \rightarrow Spec(\Sigma_2)$. In general, $ParSpec(\ldots\Sigma_1\ldots \rightarrow \ldots\Sigma_2\ldots)$ is the same as $(\ldots Spec(\Sigma_1)\ldots \rightarrow \ldots Spec(\Sigma_2)\ldots)$. Similar definitions may be formulated for parameter specifications in place of parameter signatures, which will be done in Section 7. For now we will use an obvious modification of the notation introduced in Section 3.

Example 5.2 The requirements specification in Example 4.21 may be rephrased using higher-order parameterisation. The most natural way to view *Bunch* is as a parameterised specification, parameterised by the type of elements. As before, *Bunch* serves as a parameter for *Delete*, which then becomes a higher-order parameterised specification.

Let

$$\text{Elem} \;=_{def}\; \textbf{sorts}\;\; \text{elem}$$

and

$$
\begin{array}{lll}
\text{P-Bunch} =_{def} & \lambda \mathit{ESP}{:}\mathit{Spec}(\text{Elem}). & \\
& \quad \textbf{reachable} & \\
& \quad\quad \textbf{enrich } \mathit{ESP} + \text{Nat} & \\
& \quad\quad \textbf{by}\ \ \textbf{sorts} & \text{bunch} \\
& \quad\quad\quad\ \ \textbf{opns} & \text{empty} : \ \to \text{bunch} \\
& & \text{add} : \text{elem} \times \text{bunch} \to \text{bunch} \\
& & \text{removeone} : \text{elem} \times \text{bunch} \to \text{bunch} \\
& & \text{count} : \text{elem} \times \text{bunch} \to \text{nat} \\
& \quad\quad\quad\ \ \textbf{axioms} & \text{count}(a, \text{empty}) = 0 \\
& & \text{count}(a, \text{add}(a, B)) > 0 = \text{true} \\
& & a \neq b \Rightarrow \text{count}(a, \text{add}(b, B)) = \text{count}(a, B) \\
& & \text{count}(a, B) = 0 \Rightarrow \text{count}(a, \text{removeone}(a, B)) = 0 \\
& & \text{count}(a, B) \neq 0 \Rightarrow \\
& & \qquad \text{count}(a, \text{removeone}(a, B)) = \text{count}(a, B) - 1 \\
& & a \neq b \Rightarrow \text{count}(b, \text{removeone}(a, B)) = \text{count}(b, B) \\
& \quad \textbf{on } \{\text{bunch}\} &
\end{array}
$$

Let Σ_{Bunch} be the result signature of *P-Bunch*. Then $\mathit{P\text{-}Bunch} : \mathit{Spec}(\mathit{Elem}) \to \mathit{Spec}(\Sigma_{Bunch})$.

$$
\begin{array}{lll}
\text{P-Delete} =_{def} & \lambda \mathit{BSP}{:}\mathit{Spec}(\text{Elem}) \to \mathit{Spec}(\Sigma_{\text{Bunch}}). & \\
& \lambda \mathit{ESP}{:}\mathit{Spec}(\text{Elem}). & \\
& \quad \textbf{enrich } \mathit{BSP}(\mathit{ESP}) & \\
& \quad \textbf{by}\ \ \textbf{opns} & \text{delete} : \text{elem} \times \text{bunch} \to \text{bunch} \\
& \quad\quad\ \ \textbf{axioms} & \text{count}(a, \text{delete}(a, B)) = 0 \\
& & a \neq b \Rightarrow \text{count}(a, \text{delete}(b, B)) = \text{count}(a, B)
\end{array}
$$

We now have

$$\text{P-Delete} : (\mathit{Spec}(\text{Elem}) \to \mathit{Spec}(\Sigma_{\text{Bunch}})) \to (\mathit{Spec}(\text{Elem}) \to \mathit{Spec}(\Sigma_{\text{Delete}}))$$

where Σ_{Delete} is Σ_{Bunch} together with the new operation $\mathit{delete} : \mathit{elem} \times \mathit{bunch} \to \mathit{bunch}$.

We can thus apply *P-Delete* to *P-Bunch*, and then $\mathit{P\text{-}Delete}(\mathit{P\text{-}Bunch}) : \mathit{Spec}(\mathit{Elem}) \to \mathit{Spec}(\Sigma_{Delete})$ is a parameterised specification of bunches with the *delete* operation, parameterised by the type of elements. Finally, $\mathit{P\text{-}Delete}(\mathit{P\text{-}Bunch})(\mathit{Elem}) : \mathit{Spec}(\Sigma_{Delete})$ is a non-parameterised specification of bunches of arbitrary elements with the *delete* operation. This is the same as the requirements specification *BunchDelete* of Example 4.21. □

Our earlier discussion said that it is often possible to turn a requirements specification directly into a design specification, leading towards an implementation having the same structure as the requirements specification. Applying this design strategy to the above example naturally leads to the need for higher-order parameterisation of software modules. We will be talking about software modules (corresponding to *P-Delete*) parameterised by software modules (corresponding to *P-Bunch*) which are themselves parameterised (by a realization of *Elem*). To model this we need higher-order parametric algebras. The generalisation is not difficult: we introduce a hierarchy of higher-order parametric algebras indexed by the class $\mathcal{T}$ of types.

Definition 5.3 The class $\mathit{Alg}(\tau)$ of parametric algebras of type $\tau \in \mathcal{T}$ is defined inductively as follows:

- If $\tau = \Sigma$, then $\mathit{Alg}(\tau)$ is the class of all Σ-algebras.

- If $\tau = \tau_1 \rightarrow \tau_2$, then $Alg(\tau) =_{def} Alg(\tau_1) \rightarrow Alg(\tau_2)$, the class of all functions from $Alg(\tau_1)$ to $Alg(\tau_2)$.

The semantic domain $Spec(\tau)$ of (denotations of) specifications of parametric algebras of type $\tau \in \mathcal{T}$ is defined as $Pow(Alg(\tau))$, ordered by inclusion. $\square$

This covers first-order parameterisation (see Definition 4.2), since $Spec(\Sigma_1 \rightarrow \Sigma_2)$ as defined here is the same as $Spec(\Sigma_1 \rightarrow \Sigma_2)$ as defined there. This will be generalised further in Section 7. For now we will use an obvious modification of the notation introduced in Section 3 for specifications of parametric algebras.

As before, there is a natural connection between the semantic domains of parameterised specifications and specifications of parametric algebras having the same type. The following definition generalises Definition 4.5.

Definition 5.4 For any type $\tau \in \mathcal{T}$, we define by induction:

- For any $\mathcal{P} \in ParSpec(\tau)$, let $\mathcal{P}^\sharp \in Spec(\tau)$ be defined by:
  - If $\tau = \Sigma$, $\mathcal{P}^\sharp =_{def} \mathcal{P}$.
  - If $\tau = \tau_1 \rightarrow \tau_2$, then $\mathcal{P}^\sharp =_{def} \{F \in Alg(\tau) \mid \text{for all } A \in Alg(\tau_1),\ F(A) \in \mathcal{P}(\{A\}^\dagger)^\sharp\}$.
- For any $\mathcal{Q} \in Spec(\tau)$, let $\mathcal{Q}^\dagger \in ParSpec(\tau)$ be defined by:
  - If $\tau = \Sigma$, $\mathcal{Q}^\dagger =_{def} \mathcal{Q}$.
  - If $\tau = \tau_1 \rightarrow \tau_2$, then $\mathcal{Q}^\dagger(C) =_{def} \bigsqcup_{A \in C^\sharp} \{F(A) \mid F \in \mathcal{Q}\}^\dagger$ for any $C \in ParSpec(\tau_1)$, where $\sqcup$ is the least upper bound in $ParSpec(\tau_2)$ with respect to the ordering of Definition 5.1.

$\square$

Proposition 4.7 and Corollary 4.8 carry over to this more general framework as well (the monotonicity of parameterised specifications is needed to show this). It is possible to further generalise this to the case where we have parameter specifications rather than parameter types, but the details are rather involved.

Armed with the above technicalities, let us go back to our example.

Example 5.5 Suppose that

$$\text{BunchDelete} =_{def} \text{P-Delete(P-Bunch)(Elem)}$$

is the requirements specification we are to implement. Viewing this definition as a design specification, we decompose the task of implementing *BunchDelete* into three subtasks:

1. Implement *Elem*.

2. Implement $P\text{-}Bunch^\sharp$, where $P\text{-}Bunch^\sharp$ is given explicitly as follows:

$$\begin{array}{lllll}
\text{P-Bunch}^\sharp = & \Pi A{:}\text{Elem}. & & & \\
& \quad \textbf{enrich } \{A\} + \text{Nat} & & & \\
& \quad \textbf{by} & \textbf{sorts} & \text{bunch} & \\
& & \textbf{opns} & \text{empty} : \ \to \text{bunch} & \\
& & & \text{add} : \text{elem} \times \text{bunch} \to \text{bunch} & \\
& & & \text{removeone} : \text{elem} \times \text{bunch} \to \text{bunch} & \\
& & & \text{count} : \text{elem} \times \text{bunch} \to \text{nat} & \\
& & \textbf{axioms} & \text{count}(a, \text{empty}) = 0 & \\
& & & \text{count}(a, \text{add}(a, B)) > 0 = \text{true} & \\
& & & a \neq b \Rightarrow \text{count}(a, \text{add}(b, B)) = \text{count}(a, B) & \\
& & & \text{count}(a, B) = 0 \Rightarrow \text{count}(a, \text{removeone}(a, B)) = 0 & \\
& & & \text{count}(a, B) \neq 0 \Rightarrow & \\
& & & \qquad \text{count}(a, \text{removeone}(a, B)) = \text{count}(a, B) - 1 & \\
& & & a \neq b \Rightarrow \text{count}(b, \text{removeone}(a, B)) = \text{count}(b, B) &
\end{array}$$

3. Implement $P\text{-}Delete^{(\sharp)}$, where $P\text{-}Delete^{(\sharp)}$ is defined as a restriction of $P\text{-}Delete^\sharp$ to the domain of interest as follows:

$$\begin{array}{llll}
\text{P-Delete}^{(\sharp)} =_{def} & \Pi F{:}\text{P-Bunch}^\sharp. & & \\
& \quad \Pi A{:}\text{Elem}. & & \\
& \qquad \textbf{enrich } \{F(A)\} & & \\
& \qquad \textbf{by} \ \ \textbf{opns} & \text{delete} : \text{elem} \times \text{bunch} \to \text{bunch} & \\
& \qquad \quad \ \ \textbf{axioms} & \text{count}(a, \text{delete}(a, B)) = 0 & \\
& & a \neq b \Rightarrow \text{count}(a, \text{delete}(b, B)) = \text{count}(a, B) &
\end{array}$$

Given any realization of these tasks, that is:

1. any algebra $A \in [\![Elem]\!]$;
2. any parametric algebra $F \in [\![P\text{-}Bunch^\sharp]\!]$; and
3. any higher-order parametric algebra $G \in [\![P\text{-}Delete^{(\sharp)}]\!]$

an implementation of the requirement specification *BunchDelete* may be constructed, namely $G(F)(A) \in [\![BunchDelete]\!]$. □

We stress once more that turning a requirements specification directly into a design specification is not the only way to proceed. It is important to allow the design specification to take on a completely different structure from the requirements specification if necessary.

The above example illustrates that higher-order parameterisation is sometimes necessary to present specifications and to structure implementations in a natural way. Another more extended example is given in the appendix of [SST 90]; see also [Sok 90].

6 Program development

Higher-order parameterisation is not only useful for purposes of presentation, as we saw in the previous section; it also comes in during the process of developing modular programs from specifications.

The view of program development presented in [ST 88b], [ST 92] and further elaborated for the Extended ML framework in [ST 89] is based on the notion of a *constructor implementation*:

Definition 6.1 Let SP be a specification of (parametric) algebras of type $\tau \in \mathcal{T}$ and let $SP_1, \ldots, SP_n$ be specifications. A *constructor from* $\langle SP_1, \ldots, SP_n \rangle$ *to* τ is a function $\kappa : [\![SP_1]\!] \to \ldots \to [\![SP_n]\!] \to Alg(\tau)$. We say that SP *is implemented by* $\langle SP_1, \ldots, SP_n \rangle$ *via* κ, written $SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle$, if for all $A_1 \in [\![SP_1]\!], \ldots, A_n \in [\![SP_n]\!]$, $\kappa(A_1) \cdots (A_n) \in [\![SP]\!]$. □

If $SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle$, then given any realizations $A_1 \in [\![SP_1]\!], \ldots, A_n \in [\![SP_n]\!]$, the constructor κ yields a realization of SP, namely $\kappa(A_1) \cdots (A_n) \in [\![SP]\!]$. For technical convenience we have assumed here that constructors, which may naturally be viewed as multi-argument functions, are given in their curried form.

The program development process builds a tree of constructor implementations having the original requirements specification as its root and specifications of subtasks yet to be achieved as leaves.

$$SP \underset{\kappa}{\leadsto} \begin{cases} SP_1 \underset{\kappa_1}{\leadsto} \cdots \\ \vdots \\ SP_n \underset{\kappa_n}{\leadsto} \begin{cases} SP_{n1} \underset{\kappa_{n1}}{\leadsto} \cdots \\ \cdots \\ SP_{nm} \underset{\kappa_{nm}}{\leadsto} \cdots \end{cases} \end{cases}$$

This process is finished once a tree is obtained which has specifications with known implementations, given as parameterless constructors, as its leaves.

$$SP \underset{\kappa}{\leadsto} \begin{cases} SP_1 \underset{\kappa_1}{\leadsto} \langle\rangle \\ \vdots \\ SP_n \underset{\kappa_n}{\leadsto} \begin{cases} SP_{n1} \underset{\kappa_{n1}}{\leadsto} \left\{ SP_{n11} \underset{\kappa_{n11}}{\leadsto} \langle\rangle \right. \\ \cdots \\ SP_{nm} \underset{\kappa_{nm}}{\leadsto} \langle\rangle \end{cases} \end{cases}$$

Then the composition of the constructors in the tree yields a realization of the original requirements specification. The above tree yields

$$\kappa\,(\kappa_1)\ \cdots\ (\kappa_n(\kappa_{n1}(\kappa_{n11}))\cdots(\kappa_{nm})) \in [\![SP]\!].$$

An obvious observation here is that constructors (in their curried form) are parametric algebras.

This view was presented in [ST 88b] but was limited there to specifications of non-parametric algebras; in particular the type τ of algebras specified by SP was always a signature. In the current framework it is natural to consider the generalisation of this view to the case where any of the specifications involved in the development process is a specification of (possibly higher-order) parametric algebras.

For example, one might wish to use an implementation such as $SP \underset{\kappa}{\leadsto} \langle SP_1, SP_2 \rangle$ where $SP : Spec(\Sigma)$, $SP_1 : Spec(\Sigma_1)$ and $SP_2 : Spec(\Sigma_1 \to \Sigma)$. The constructor κ would be a function $\kappa : [\![SP_1]\!] \to [\![SP_2]\!] \to Alg(\Sigma)$ defined by $\kappa(A_1)(F_2) = F_2(A_1)$. A correct implementation is obtained provided that for any $A_1 \in [\![SP_1]\!]$ and $F_2 \in [\![SP_2]\!]$, $F_2(A_1) \in [\![SP]\!]$. This holds, for instance, if we let $SP_2 =_{def} \Pi A{:}SP_1.\, SP$, which decomposes the problem of implementing SP into the problem of implementing SP_1 and (independently) implementing SP_2. The latter is the problem of implementing SP given an (arbitrary) implementation of SP_1.

In general, higher-order parameterisation allows us to restrict to constructors which are of a particularly simple form. Namely, any constructor implementation $SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle$ may be replaced by the decomposition $SP \underset{app}{\leadsto} \langle SP_\kappa, SP_1, \ldots, SP_n \rangle$, where $SP_\kappa = \Pi X_1{:}SP_1. \cdots \Pi X_n{:}SP_n.\, SP$ and $app = \lambda F{:}SP_\kappa.\, \lambda X_1{:}SP_1. \cdots \lambda X_n{:}SP_n.\, F X_1 \ldots X_n$, and then the realization of SP_κ by the constructor κ (which is correct since $SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle$ means $\kappa \in [\![SP_\kappa]\!]$). Such a decomposition captures the decision to realize SP in terms of realizations of $SP_1, \ldots, SP_n$, turning the problem of finding the necessary construction into yet another subtask. If $n = 0$ then this is pointless since no progress is made towards a realization. Even if $n > 0$, from the methodological point of view one may question whether it is appropriate to postpone the problem of finding a constructor in this manner. This is exactly the controversy between advocates of the top-down development style (as captured by $SP \underset{\kappa}{\rightsquigarrow\!\!\!>} \langle SP_1, \ldots, SP_n \rangle$) and the bottom-up development style (as captured by $SP \underset{app}{\rightsquigarrow\!\!\!>} \langle SP_\kappa, SP_1, \ldots, SP_n \rangle$). We believe that steps of both kinds are useful, and indeed in this formulation the distinction is somewhat blurred, with a spectrum of possibilities between these two extremes.

Example 6.2 Example 5.5 may be rephrased as a higher-order constructor implementation step

$$\text{BunchDelete} \underset{\kappa}{\rightsquigarrow\!\!\!>} \langle \text{Elem}, \text{P-Bunch}^\sharp, \text{P-Delete}^{(\sharp)} \rangle$$

where $\kappa : [\![Elem]\!] \to [\![P\text{-}Bunch^\sharp]\!] \to [\![P\text{-}Delete^{(\sharp)}]\!] \to Alg(\Sigma_{Delete})$ is defined by $\kappa(A)(F)(G) =_{def} G(F)(A)$ for arbitrary $A \in [\![Elem]\!]$, $F \in [\![P\text{-}Bunch^\sharp]\!]$ and $G \in [\![P\text{-}Delete^{(\sharp)}]\!]$. □

If we restrict attention to the case in which all the specifications involved are Extended ML functor specifications (recall that these amount to specifications of first-order parametric algebras), the framework obtained corresponds to the Extended ML formal program development methodology described in [ST 89], [ST 91b] (modulo issues of behavioural equivalence). The main kind of development step in this framework is the decomposition of an Extended ML functor specification into a number of simpler Extended ML functor specifications. The constructor involved in this step describes how to build a functor which realizes the original functor specification out of functors which realize the simpler functor specifications.

Example 6.3 A simple case is the decomposition of the specification

$$FSP \;=_{def}\; \Pi X{:}SP_{in}.\, SP_{out}$$

into the two specifications

$$\begin{aligned} GSP &=_{def} \Pi X{:}SP_{in}.\, SP' \\ HSP &=_{def} \Pi Y{:}SP'.\, SP_{out} \end{aligned}$$

via the constructor

$$\lambda G{:}GSP.\, (\lambda H{:}HSP.\, (\lambda X{:}SP_{in}.\, H(G(X))))$$

This splits the task of implementing FSP into first implementing SP' given any model of SP_{in}, and separately implementing SP_{out} given any model of SP'. □

Example 6.4 When the decomposition is as in Example 6.3, the implementor of HSP cannot use the model of SP_{in} which is available to the implementor of GSP. This suggests that a possibly more flexible decomposition of FSP might be into the specifications

$$\begin{aligned} GSP &=_{def} \Pi X{:}SP_{in}.\, SP' \\ HSP &=_{def} \Pi X{:}SP_{in}.\, (\Pi Y{:}SP'.\, SP_{out}) \end{aligned}$$

via the constructor

$$\lambda G{:}GSP.\, (\lambda H{:}HSP.\, (\lambda X{:}SP_{in}.\, H(X)(G(X))))$$

The uncurried version of this (where HSP is a two-argument functor specification) is the best we could have done in the first-order framework described in [ST 89], [ST 91b]. □

Example 6.5 We can use higher-order parameterisation to go further than in Example 6.4 and make the realization of GSP available to the implementor of HSP. We decompose the task of implementing FSP into the specifications

$$\begin{array}{lll} GSP & =_{def} & \Pi X{:}SP_{in}.\ SP' \\ HSP & =_{def} & \Pi X{:}SP_{in}.\ (\Pi G{:}GSP.\ SP_{out}) \end{array}$$

via the constructor

$$\lambda G{:}GSP.\ (\lambda H{:}HSP.\ (\lambda X{:}SP_{in}.\ H(X)(G)))$$

The implementor of HSP now has more powerful tools available to use in building a realization of SP_{out}. Previously, only a realization X of SP_{in} and a particular realization $G(X)$ of SP' were made available; now G itself is available as well, which means that $G(X)$ as well as applications of G to other algebras satisfying SP_{in} can easily be obtained.

Note that there is nothing methodologically ugly about this: there are two independent implementation tasks to perform, one of implementing GSP and the other of implementing HSP, and a strict separation between the two tasks is preserved. The implementor of HSP is provided with a realization of GSP, but he can rely only on those features of this realization which are explicitly stated in GSP. $\Box$

The use of higher-order parameterisation as in the above example is not merely a specious generalisation. Consider the problem of implementing an interpreter for a programming language. An obvious subtask of this is to implement stacks, since stacks are useful in various places including the parser (stacks of parse trees) and the evaluator (stacks of data values). Using first-order parameterisation as in Example 6.4, we would decompose the task of implementing the interpreter into the following subtasks:

1. Implement stacks of arbitrary elements.
2. Implement parse trees.
3. Implement data values.
4. Implement the interpreter, using parse trees, data values, stacks of parse trees and stacks of data values.

However, the implementor of subtask 4 may notice that yet another kind of stacks are required, perhaps in the symbol table. In a strict first-order regime, he/she would either have to re-implement stacks for this purpose, or else go back to the designer and ask for the specification of his/her subtask to be modified to provide the new instantiation of stacks. But using higher-order parameterisation as in Example 6.5, subtask 4 would become

4. Implement the interpreter, using parse trees, data values and stacks of arbitrary elements.

Stacks of parse trees and of data values would be constructed during the realization of this subtask, as would stacks of other kinds of elements if the need arises. An alternative decomposition would be into just two subtasks:

1. Implement stacks of arbitrary elements.
2. Implement the interpreter, using stacks of arbitrary elements.

This leaves the implementation of parse trees and data values as potential lower-level subtasks of subtask 2. Such a decomposition introduces a bottom-up flavour into our top-down design methodology, as mentioned above. It may turn out that the implementor of subtask 2 does not need stacks at all; they are provided as tools which might come in handy for the task at hand.

In some such cases, explicit higher-order parameterisation may be avoided. An environment of previously-defined modules, all of which are available for use in subsequent module definitions, allows some higher-order dependencies of the kind illustrated above to be left implicit. However, this trick does not work when dependencies become complex and deeply nested, and anyway it seems advisable to keep dependencies explicit rather than trying to sweep them under the carpet.

7 A kernel specification formalism

7.1 Introducing the language

In the preceding sections we have argued for the use of both parameterised specifications and specifications of parametric algebras (and of their higher-order counterparts) in software specification and development. In this section, we present a specification formalism which extends in an essential way the kernel specification language presented in [ST 88a] by adding a simple yet powerful parameterisation mechanism which allows us to define and specify parametric algebras of arbitrary order, as well as extending the mechanism in [ST 88a] for defining first-order parameterised specifications to the higher-order case. This is achieved by viewing specifications on one hand as specifications of objects such as algebras or parametric algebras, and on the other hand as objects themselves to which functions (i.e. parameterised specifications) may be applied. Consequently, the language allows specifications to be specified by other specifications, much as in CLEAR [BG 80] or ACT ONE [EM 85] parameterisation where the parameter specification specifies the permissible argument specifications (see Section 3).

The view of specifications as objects enables the use of a uniform parameterisation mechanism, functions defined by means of λ-abstraction, to express both parameterised specifications and parametric algebras. There is also a uniform specification mechanism to specify such functions, Π-abstraction (Cartesian-product specification, closely related to the dependent function type constructor in e.g. NuPRL [Con 86]). This may be used to specify (higher-order) parametric algebras as well as (higher-order) parameterised specifications. There is no strict separation between levels, which means that it is possible to intermix parameterisation of objects and parameterisation of specifications, obtaining (for example) algebras which are parametric on parameterised specifications or specifications which are parameterised by parametric algebras. We have not yet explored the practical implications of this technically natural generalisation.

The language does not include notation for describing algebras, signatures, signature morphisms, or sets of sentences. Such notation must be provided separately, for example as done for ASL in [Wir 86]. The definition of the language is independent of this notation; moreover, it is essentially *institution independent*, with all the advantages indicated in [GB 84], [ST 88a].

The language has just one syntactic category of interest, which includes both specifications and objects that are specified, with syntax as follows:

$$
\begin{array}{lcll}
\mathit{Object} & = & \mathit{Signature} & \left.\begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \end{array}\right\} \text{\textit{Simple specifications}} \\
& \mid & \textbf{impose } \mathit{Sentences} \textbf{ on } \mathit{Object} & \\
& \mid & \textbf{derive from } \mathit{Object} \textbf{ by } \mathit{Signature\text{-}morphism} & \\
& \mid & \textbf{translate } \mathit{Object} \textbf{ by } \mathit{Signature\text{-}morphism} & \\
& \mid & \textbf{minimal } \mathit{Object} \textbf{ wrt } \mathit{Signature\text{-}morphism} & \\
& \mid & \textbf{iso-close } \mathit{Object} & \\
& \mid & \textbf{abstract } \mathit{Object} \textbf{ wrt } \mathit{Sentences} \textbf{ via } \mathit{Signature\text{-}morphism} & \\
& \mid & \mathit{Object} \cup \mathit{Object} & \text{\textit{Other specifications}} \\
& \mid & \Pi\, \mathit{Variable}{:}\mathit{Object}.\ \mathit{Object} & \\
& \mid & \{\mathit{Object}\} & \\
& \mid & \textbf{Spec}(\mathit{Object}) & \\
& \mid & \mathit{Variable} & \text{\textit{Other objects}} \\
& \mid & \mathit{Algebra\text{-}expression} & \\
& \mid & \lambda\, \mathit{Variable}{:}\mathit{Object}.\ \mathit{Object} & \\
& \mid & \mathit{Object}(\mathit{Object}) &
\end{array}
$$

As usual, we have omitted the "syntax" of variables. The other syntactic categories of the language above are algebra expressions, signatures, sets of sentences and signature morphisms — as mentioned above, the details of these are not essential to the main ideas of this paper and we assume that they are provided externally. Algebra expressions may contain occurrences of object variables. We will assume, however, that variables do not occur in signatures, signature morphisms and sentences, which seems necessary to keep the formalism institution-independent. This requirement may seem overly restrictive, as it seems to disallow the components of a particular algebra to be used in axioms; one would expect to be able to write something like $\Pi X{:}\Sigma.\ (\ldots X.op \ldots)$. Fortunately, using the power of the specification-building operations included in the language, it is possible to define a more convenient notation which circumvents this restriction (see the appendix of [SST 90]).

We have used the standard notation for Π- and λ-objects to suggest the usual notions of a free and of a bound occurrence of a variable in a term of the language, as well as of a closed term. As usual, we identify terms which differ only in their choice of bound variable names. We define substitution of objects for variables in the usual way: $Obj[Obj'/X]$ stands for the result of substituting Obj' for all free occurrences of X in Obj in such a way that no unintended clashes of variable names take place. This also defines the usual notion of β-reduction between objects of the language: $(\ldots(\lambda X{:}SP.\ Obj)(Obj')\ldots) \rightarrow_\beta (\ldots Obj[Obj'/X]\ldots)$. Then, $\rightarrow_\beta^*$ is the reflexive and transitive closure of $\rightarrow_\beta$.

The first seven kinds of specifications listed above (simple specifications) are taken directly from [ST 88a] (see Section 2). The particular choice of these seven operations is orthogonal to the rest of the language and will not interfere with the further development in this paper. We have singled out the union operation — we will use it for arbitrary, not necessarily "simple" specifications (this generalisation w.r.t. [ST 91a] makes the formalism more flexible, but otherwise does not seem to cause any extra technical difficulties). The other three kinds of specifications are new. Π-abstraction is used to specify parametric objects.[3] To make this work, it must be possible to use objects in specifications. The $\{\_\}$ operation provides this possibility by allowing objects to be turned into (very tight) specifications. The next clause allows a specification which defines a class $\mathcal{C}$ of objects to be turned into a specification which defines the class of specifications defining subclasses of $\mathcal{C}$. This is compatible with the use of parameter specifications in parameterised specifications as in

[3]The notation $SP \rightarrow SP'$ is often used for $\Pi X{:}SP.\ SP'$ if X does not actually occur in SP'.

CLEAR and ACT ONE. For example, the declaration **proc** $P(X : SP) = \ldots$ in CLEAR introduces a parameterised specification P, where the parameter (or *requirement*) specification SP describes the admissible arguments of P. Namely, if SP defines a class of objects $\mathcal{C} = [\![SP]\!]$ then P may be applied to argument specifications SP_{arg} defining a subclass of $\mathcal{C}$, i.e. such that $[\![SP_{arg}]\!] \subseteq [\![SP]\!]$ (we disregard the parameter fitting mechanism). In our formalism this would be written as $P =_{def} \lambda X{:}\mathbf{Spec}(SP). \ldots$.

The syntax of other objects is self-explanatory.

The richness of the language may lead to some difficulty in recognizing familiar concepts which appear here in a generalised form. The following comments might help to clarify matters:

- A specification is (an object which denotes) a class of objects. If the objects of this class are algebras, then this specification is a specification in the usual sense.
- $\Pi X{:}(\ldots).(\ldots)$ denotes a class of mappings from objects to objects. If these objects are algebras, then this is a class of parametric algebras, i.e. a specification of a parameterised program.
- $\lambda X{:}(\ldots).(\ldots)$ denotes a mapping from objects to objects. If these objects are specifications in the usual sense, then this is a parameterised specification.

The semantics of the language, presented in the next section, gives more substance to the informal comments above concerning the intended denotations of certain phrases.

As pointed out above, we assume that the sublanguage of expressions defining algebras is to be supplied externally (with a corresponding semantics — see Section 7.2). Even under this assumption, it would be possible to include institution-independent mechanisms for building algebras from other algebras (amalgamation, reduct, free extension, etc.) in the language, which could lead to a powerful and uniform calculus of specified modular programs. This is an interesting possibility for future work but it is outside the scope of this paper.

7.2 Semantics

We have chosen the syntax for objects in the language so that their semantics should be intuitively clear. We formalise it by defining for any environment ρ, which assigns meanings to variables, a partial function $[\![\_]\!]\rho$ mapping an object Obj to its meaning $[\![Obj]\!]\rho$. It is defined below by structural induction on the syntax of objects. The use of the meta-variable SP instead of Obj in some places below is intended to be suggestive (of objects denoting object classes, used as specifications) but has no formal meaning. This convention will be used throughout the rest of the paper.

Simple specifications:

$[\![\Sigma]\!]\rho = Alg(\Sigma)$

$[\![\mathbf{impose}\ \Phi\ \mathbf{on}\ SP]\!]\rho = \{A \in [\![SP]\!]\rho \mid A \models \Phi\}$

if $[\![SP]\!]\rho \subseteq Alg(\Sigma)$ and $\Phi \subseteq Sen(\Sigma)$ for some signature Σ

$[\![\mathbf{derive\ from}\ SP\ \mathbf{by}\ \sigma]\!]\rho = \{A|_\sigma \mid A \in [\![SP]\!]\rho\}$

if $[\![SP]\!]\rho \subseteq Alg(\Sigma)$ and $\sigma : \Sigma' \to \Sigma$ is a signature morphism for some signatures Σ and Σ'

... *similarly for the other forms, based on the semantics given in Section 2* ...

Other specifications:

$[\![SP \cup SP']\!]\rho = [\![SP]\!]\rho \cap [\![SP']\!]\rho$
  if $[\![SP]\!]\rho$ and $[\![SP']\!]\rho$ are classes of values
$[\![\{Obj\}]\!]\rho = \{[\![Obj]\!]\rho\}$
  if $[\![Obj]\!]\rho$ is defined
$[\![\Pi X{:}SP.\, SP']\!]\rho = \Pi_{v \in [\![SP]\!]\rho}[\![SP']\!]\rho[v/X]$[4,5]
  if $[\![SP]\!]\rho$ is a class of values and for each $v \in [\![SP]\!]\rho$, $[\![SP']\!]\rho[v/X]$ is a class of values
$[\![\mathbf{Spec}(SP)]\!]\rho = Pow([\![SP]\!]\rho)$
  if $[\![SP]\!]\rho$ is a class of values

Other objects:

$[\![X]\!]\rho = \rho(X)$
$[\![A]\!]\rho = \ldots$ *assumed to be given externally* $\ldots$
$[\![\lambda X{:}SP.\, Obj]\!]\rho = \{\langle v \mapsto [\![Obj]\!]\rho[v/X]$[5]$\rangle \mid v \in [\![SP]\!]\rho\}$
  if $[\![SP]\!]\rho$ is a class of values and for each $v \in [\![SP]\!]\rho$, $[\![Obj]\!]\rho[v/X]$ is defined
$[\![Obj(Obj')]\!]\rho = [\![Obj]\!]\rho([\![Obj']\!]\rho)$
  if $[\![Obj]\!]\rho$ is a function and $[\![Obj']\!]\rho$ is a value in the domain of this function

In the above definition, it is understood that a condition like "$[\![SP]\!]\rho \subseteq Alg(\Sigma)$" implicitly requires that $[\![SP]\!]\rho$ is defined. An object's meaning is undefined unless the side-condition of the appropriate definitional clause holds.

It is easy to see that the semantics of an object of the language depends only on the part of the environment which assigns meanings to variables which occur free in the object. In particular, the meaning of a closed object is independent from the environment. This allows us to omit the environment when dealing with the semantics of closed objects and write simply $[\![Obj]\!]$ to stand for $[\![Obj]\!]\rho$ for any environment ρ whenever Obj is closed.

Of course, the above remark is true only provided that the sublanguage of algebra expressions and its semantics assumed to be given externally have this property. In the following, we will take this for granted. We will also assume that the sublanguage satisfies the following substitutivity property: for any algebra expression A, variable X and object Obj, for any environment ρ such that $v = [\![Obj]\!]\rho$ is defined, $[\![A[Obj/X]]\!]\rho$ is defined if and only if $[\![A]\!]\rho[v/X]$ is defined, and if they are defined then they are the same. This ensures that the following expected fact holds for our language (the standard proof by induction on the structure of objects is omitted):

Fact 7.1 For any objects Obj, Obj' and variable X, for any environment ρ such that $v' = [\![Obj']\!]\rho$ is defined, $[\![Obj[Obj'/X]]\!]\rho$ is defined if and only if $[\![Obj]\!]\rho[v'/X]$ is defined, and if they are defined then

$$[\![Obj[Obj'/X]]\!]\rho = [\![Obj]\!]\rho[v'/X]$$

□

Corollary 7.2 β-reduction preserves the meaning of objects. That is, for any objects Obj and Obj' such that $Obj \rightarrow_\beta^* Obj'$, for any environment ρ, if $[\![Obj]\!]\rho$ is defined then so is $[\![Obj']\!]\rho$ and $[\![Obj]\!]\rho = [\![Obj']\!]\rho$. □

4 Π on the right-hand side of this definition denotes the usual Cartesian product of an indexed family of sets. That is, $\Pi_{x \in S} C_x$ is the set of all functions with domain S mapping any $x \in S$ to an element of C_x.

5 As usual, $\rho[v/X]$ is the environment which results from ρ by assigning v to the variable X (and leaving the values of other variables unchanged).

The reader might feel uneasy about the fact that we have not actually defined here any domain of values, the elements of which are assigned to objects of the language as their meanings. A naive attempt might have been as follows:

$$Values = Algebras \mid Pow(Values) \mid Values \rightharpoonup Values$$

Clearly, this leads to serious foundational problems, as the recursive domain definition involves "heavy recursion" (cf. [BT 83]) and hence cannot have a set-theoretic solution (even assuming that we consider here a set *Algebras* of algebras built within a fixed universe). However, since the formalism we introduce is not intended to cater for any form of self application of functions or non-well-foundedness of sets, the equation above attempts to define a domain of values of objects which is undesirably rich. The well-formed[6] objects of the language can easily be seen to form a hierarchy indexed by "types" (see Section 7.4). Thus, we can define a corresponding cumulative hierarchy of sets of values, and then define the domain of the meanings of objects as the union of sets in the hierarchy, much in the style of [BKS 88] (see [BT 83] where the idea of using hierarchies of domains in denotational semantics is discussed in more detail). Another, less "constructive", possibility is to work within a fixed universal set of values of objects containing the "set" of all algebras [Coh 81].

7.3 Proving satisfaction

We are interested in determining whether or not given objects satisfy given specifications. We use the formal judgement $Obj : SP$ to express the assertion that a closed object Obj satisfies a closed specification SP, i.e. that $[\![Obj]\!] \in [\![SP]\!]$, and generalise it to $X_1 : SP_1, \ldots, X_n : SP_n \vdash Obj : SP$ stating the assertion that an object Obj satisfies a specification SP in the *context* $X_1 : SP_1, \ldots, X_n : SP_n$, i.e. under the assumption that objects $X_1, \ldots, X_n$ satisfy specifications $SP_1, \ldots, SP_n$, respectively. The inference rules listed below allow us to derive judgements of this general form. For the sake of clarity, though, we have decided to make contexts implicit in the rules and rely on the natural deduction mechanism of introducing and discharging assumptions (all of the form $X : SP$ here) to describe the appropriate context manipulation. For example, in (R2) below, $[X : SP]$ is an assumption which may be used to derive $SP' : \mathbf{Spec}(SP'')$, but is discharged when we apply the rule to derive its conclusion. Whenever necessary, we will use the phrase "the current context" to refer to the sequence of currently undischarged assumptions. We say that an environment ρ is *consistent* with a context $X_1 : SP_1, \ldots, X_n : SP_n$ if for $i = 1, \ldots, n$, $\rho(X_i) \in [\![SP_i]\!]\rho$.

Simple specifications:

$$\frac{\Sigma \text{ signature}}{\Sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{\mathbf{impose}\ \Phi\ \mathbf{on}\ SP : \mathbf{Spec}(\Sigma)}$$

$$\frac{SP : \mathbf{Spec}(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{derive\ from}\ SP\ \mathbf{by}\ \sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{translate}\ SP\ \mathbf{by}\ \sigma : \mathbf{Spec}(\Sigma')}$$

$$\frac{SP : \mathbf{Spec}(\Sigma) \qquad \sigma : \Sigma' \to \Sigma}{\mathbf{minimal}\ SP\ \mathbf{wrt}\ \sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma)}{\mathbf{iso\text{-}close}\ SP : \mathbf{Spec}(\Sigma)}$$

$$\frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi' \subseteq Sen(\Sigma') \qquad \sigma : \Sigma \to \Sigma'}{\mathbf{abstract}\ SP\ \mathbf{wrt}\ \Phi'\ \mathbf{via}\ \sigma : \mathbf{Spec}(\Sigma)}$$

[6] An intuitive understanding of the notion of well-formedness involved is sufficient here (we hope) — we introduce it formally in Section 7.3.

Other specifications:

$$\frac{Obj : SP}{\{Obj\} : \mathbf{Spec}(SP)} \tag{R1}$$

$$\frac{SP : \mathbf{Spec}(SP_{any}) \qquad \begin{array}{c}[X : SP] \\ SP' : \mathbf{Spec}(SP'')\end{array}}{\Pi X{:}SP.\, SP' : \mathbf{Spec}(\Pi X{:}SP.\, SP'')} \tag{R2}$$

$$\frac{SP : \mathbf{Spec}(SP')}{\mathbf{Spec}(SP) : \mathbf{Spec}(\mathbf{Spec}(SP'))} \tag{R3}$$

Union:

$$\frac{SP_1 : \mathbf{Spec}(SP) \qquad SP_2 : \mathbf{Spec}(SP)}{SP_1 \cup SP_2 : \mathbf{Spec}(SP)} \tag{R4}$$

$$\frac{SP_1 \cup SP_2 : \mathbf{Spec}(SP_{any}) \qquad Obj : SP_1 \qquad Obj : SP_2}{Obj : SP_1 \cup SP_2} \tag{R5}$$

λ-expressions:

$$\frac{SP : \mathbf{Spec}(SP_{any}) \qquad \begin{array}{c}[X : SP] \\ Obj : SP'\end{array}}{\lambda X{:}SP.\, Obj \;:\; \Pi X{:}SP.\, SP'} \tag{R6}$$

$$\frac{Obj : \Pi X{:}SP.\, SP' \qquad Obj' : SP}{Obj(Obj') : SP'[Obj'/X]} \tag{R7}$$

$$\frac{Obj : SP \qquad SP \to_\beta^* SP'}{Obj : SP'} \tag{R8}$$

$$\frac{Obj : SP \qquad SP' : \mathbf{Spec}(SP_{any}) \qquad SP' \to_\beta^* SP}{Obj : SP'} \tag{R9}$$

Trivial inference:

$$\frac{Obj : SP_{any}}{Obj : \{Obj\}} \tag{R10}$$

"Cut"

$$\frac{Obj : SP \qquad SP : \mathbf{Spec}(SP')}{Obj : SP'} \tag{R11}$$

Semantic inference:

$$\frac{SP : \mathbf{Spec}(\Sigma) \qquad [\![A]\!]\rho \in [\![SP]\!]\rho \text{ for all } \rho \text{ consistent with the current context}}{A : SP} \tag{R12}$$

$$\frac{SP, SP' : \mathbf{Spec}(\Sigma) \qquad [\![SP]\!]\rho \subseteq [\![SP']\!]\rho \text{ for all } \rho \text{ consistent with the current context}}{SP : \mathbf{Spec}(SP')} \tag{R13}$$

Some of these rules involve judgements (Σ signature, $\Phi \subseteq Sen(\Sigma)$, $\sigma : \Sigma \to \Sigma'$) which are external to the above formal system. This is a natural consequence of the fact that the language does not include any syntax for signatures, sentences, etc. More significantly, there are two rules which involve model-theoretic judgements, referring to the semantics of objects given above.

Following the usual practice, in the sequel we will simply write "$Obj : SP$" meaning "$Obj : SP$ is derivable".

The rules labelled *Simple specifications* characterise the well-formedness of Σ-specifications built using the underlying specification-building operations included in the language. They directly incorporate the "syntactic" requirements of Section 2 on the use of these operations. Rules (R1), (R2) and (R3) play a similar role for the other specification-forming operations: singleton specification, Cartesian-product specification and $\mathbf{Spec}(\_)$, respectively. Notice, however, that their specifications are given here in a form which is as tight as possible. For example, for any $SP : \mathbf{Spec}(\Sigma)$ and $Obj : SP$, rule (R1) allows us to deduce $\{Obj\} : \mathbf{Spec}(SP)$ rather than just $\{Obj\} : \mathbf{Spec}(\Sigma)$.

The first of the two rules related to the union operation, (R4), embodies the characterisation of well-formed union specifications. The other, (R5), gives the obvious way to prove that an object satisfies a (well-formed) union specification. The two rules are not quite satisfactory, as they do not seem to sufficiently capture the interplay between union and the other operations — more work is needed here.

The rules related to λ-expressions and their applications to arguments are quite straightforward. Rules (R6) and (R7) are the usual rules for λ-expression introduction and application, respectively. The assumption $SP : \mathbf{Spec}(SP_{any})$ in rule (R6) asserts the well-formedness of the specification SP (see also (R2), (R9), (R10)). Whenever the meta-variable SP_{any} is used below, it will play the same role as part of a well-formedness constraint. Notice that in order to prove $\lambda X{:}SP.\, Obj : \Pi X{:}SP.\, SP'$, we have to prove $Obj : SP'$ "schematically" for an arbitrary unknown $X : SP$, rather than for all values in the class $[\![SP]\!]\rho$ (for the appropriate environments ρ).

Rules (R8) and (R9) embody a part of the observation that β-reduction preserves the semantics of objects (Corollary 7.2). Rule (R8) allows for β-reduction and rule (R9) for well-formed β-expansion of specifications. A particular instance of the latter is

$$\frac{Obj' : SP'[Obj/X] \qquad (\lambda X{:}SP.\, SP')(Obj) : \mathbf{Spec}(SP_{any})}{Obj' : (\lambda X{:}SP.\, SP')(Obj)}$$

That is, in order to prove that an object satisfies a specification formed by applying a parameterised specification to an argument, it is sufficient to prove that the object satisfies the corresponding β-reduct.

However, we have not incorporated full β-equality into our system; rules (R8) and (R9) introduce it only for specifications. In particular, we have not included the following rule, which would allow well-formed β-expansion of objects:

$$\frac{Obj : SP \qquad Obj' : SP_{any} \qquad Obj' \rightarrow^*_\beta Obj}{Obj' : SP}$$

An instance of this would be:

$$\frac{Obj_1[Obj_2/X] : SP \qquad (\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP_{any}}{(\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP}$$

Hence, in order to prove that a structured object $(\lambda X{:}SP_2.\, Obj_1)(Obj_2)$ satisfies a specification SP, it would suffice to show that the object is well-formed and to prove that its β-reduct $Obj_1[Obj_2/X]$ satisfies the specification. We think that this is not methodologically desirable: a proof of correctness of a program should follow the structure of the program, without any possibility of flattening it out. So, to prove $(\lambda X{:}SP_2.\, Obj_1)(Obj_2) : SP$ we have to find an appropriate specification for the parameterised program $\lambda X{:}SP_2.\, Obj_1$, say $\lambda X{:}SP_2.\, Obj_1 : \Pi X{:}SP_2.\, SP_1$ such that $SP_1[Obj_2/X] = SP$ (actually, $SP_1[Obj_2/X] : \mathbf{Spec}(SP)$ is sufficient).

The other part of β-equality for objects, β-reduction, although not derivable in the system, is admissible in it[7] (see [ST 91a] for a proof sketch):

Lemma 7.3 The following rule is an admissible rule of the system

$$\frac{Obj : SP \qquad Obj \rightarrow^*_\beta Obj'}{Obj' : SP}$$

□

It might be interesting to enrich the system by the β-reduction rule for objects given in the above lemma, or even more generally by some "operational semantics rules" for (the computable part of) the object language. This, however, would be quite orthogonal to the issues of object specification considered in this paper. Therefore, to keep the system as small and as simple as possible, the rule is not included in the system.

Rules (R10) and (R11) embody trivial deductions which should be intuitively straightforward. Notice that $SP : \mathbf{Spec}(SP')$, as in the premise of (R11), asserts that specification SP imposes at least the same requirements as SP'.

Rules (R12) and (R13) refer directly to the semantics of objects. They embody the semantic verification process which is a necessary component of inference in the above formal system. These rules are deliberately restricted to the non-parametric case, since this is the point at which an external formal system is required; parameterisation is handled by the other rules. We do not attempt here to provide a formal system for proving the semantic judgements $[\![A]\!]\rho \in [\![SP]\!]\rho$ and $[\![SP]\!]\rho \subseteq [\![SP']\!]\rho$ for all environments ρ consistent with the current context. This is an interesting and important research topic, which is however separate from the main concerns of this paper; some considerations and results

[7] A rule is *admissible* in a deduction system if its conclusion is derivable in the system provided that all its premises are derivable. This holds in particular if the rule is *derivable* in the system, that is, if it can be obtained by composition of the rules in the system.

on this may be found in e.g. [ST 88a] and [Far 92]. It is not possible to give a set of purely "syntactic" inference rules which is sound and complete with respect to the semantics above because of the power of the specification mechanisms included in the language (this is already the case for the subset of the language excluding parameterisation, presented in Section 2).

As mentioned earlier, to make the rules as clear and readable as possible, the presentation of the system omits a full formal treatment of contexts. In particular, we should add two rules to derive judgements that a context is well-formed (here, $\langle\rangle$ is the empty context):

$$\frac{}{\langle\rangle \text{ is a well-formed context}}$$

$$\frac{\Gamma \text{ is a well-formed context} \qquad X \text{ is not in } \Gamma \qquad \begin{array}{c}[\Gamma]\\ SP : \mathbf{Spec}(SP_{any})\end{array}}{\Gamma, X : SP \text{ is a well-formed context}}$$

and then axioms $X_1 : SP_1, \ldots, X_n : SP_n \vdash X_k : SP_k$, for $k = 1, \ldots, n$, where $X_1 : SP_1, \ldots, X_n : SP_n$ is a well-formed context. It is important to realise that contexts are *sequences*, rather than sets, and so we allow the variables $X_1, \ldots, X_k$ to occur in SP_{k+1}.

We will continue omitting contexts throughout the rest of the paper. All the definitions and facts given below (as well as above) are correctly stated for closed objects only, but are meant to be naturally extended to objects in a well-formed context. This will be done explicitly only within proofs where it is absolutely necessary. Similarly, we will omit in the following the environment argument to the semantic function for objects; all the environments thus implicitly considered are assumed to be consistent with the corresponding context. We hope that this slight informality will contribute to the readability of the paper without obscuring the details too much.

The following theorem expresses the soundness of the formal system above with respect to the semantics given earlier.

Theorem 7.4 For any object Obj and specification SP, if $Obj : SP$ is derivable then $[\![Obj]\!] \in [\![SP]\!]$ (that is, $[\![SP]\!]$ is defined and is a class of values and $[\![Obj]\!]$ is defined and is a value in this class).
Proof (sketch) By induction on the length of the derivation and by inspection of the rules. A complete formal proof requires, of course, a careful treatment of free variables and their interpretation (cf. the remark preceding the theorem). Thus, for example, rule (R6) really stands for:

$$\frac{\Gamma \vdash SP : \mathbf{Spec}(SP_{any}) \qquad \Gamma, X : SP \vdash Obj : SP' \qquad X \text{ is not in } \Gamma}{\Gamma \vdash \lambda X{:}SP.\, Obj : \Pi X{:}SP.\, SP'}$$

where Γ is a context. In the corresponding case of the inductive step we can assume that

1. $[\![SP]\!]\rho \in [\![\mathbf{Spec}(SP_{any})]\!]\rho$ for all environments ρ consistent with context Γ, and

2. $[\![Obj]\!]\rho \in [\![SP']\!]\rho$ for all environments ρ consistent with context $\Gamma, X : SP$

and then we have to prove that $[\![\lambda X{:}SP.\, Obj]\!]\rho \in [\![\Pi X{:}SP.\, SP']\!]\rho$ for all environments ρ consistent with context Γ. That is, taking into account the semantics of λ- and Π-expressions as given in Section 7.2, we have to prove that for all environments ρ consistent with context Γ

- $[\![SP]\!]\rho$ is defined and is a class of values — which follows directly from assumption (1) above, and then

- for all values $v \in [\![SP]\!]\rho$,

– $[\![Obj]\!]\rho[v/X]$ is defined,
– $[\![SP']\!]\rho[v/X]$ is defined and is a class of values, and
– $[\![Obj]\!]\rho[v/X] \in [\![SP']\!]\rho[v/X]$,

which in turn follow directly from assumption (2) above.

The cases corresponding to the other rules of the system require similar, straightforward but tedious analysis. Notice that the proofs about the rules concerning application and β-reduction, (R7), (R8) and (R9), crucially depend on Fact 7.1 and Corollary 7.2. □

It is natural to ask if the above formal system is also complete with respect to the semantics. It turns out not to be complete. One reason for incompleteness is that the formal system does not exploit the semantical consequences of inconsistency. For example, for any inconsistent specification SP we have that $[\![SP]\!] \in [\![\mathbf{Spec}(SP_{any})]\!]$ for any SP_{any} such that $[\![SP_{any}]\!]$ is a class of values. The corresponding formal judgement $SP : \mathbf{Spec}(SP_{any})$ is not derivable when (for example) SP and SP_{any} are simple specifications over different signatures. If the formal parameter specification in a λ- or Π-expression is inconsistent then similar difficulties arise (cf. [MMMS 87] for a discussion of the related issue of "empty types" in typed λ-calculi). This topic deserves further study; it would be nice to identify all sources of incompleteness and the effect of the deliberate omission of a rule allowing for well-formed β-expansion of objects.

Definition 7.5 An object Obj is *well-formed* if $Obj : SP$ for some SP. □

This also defines the well-formed specifications since specifications are objects.

Checking whether an expression in the language is well-formed must in general involve "semantic" verification as embodied in rules (R12) and (R13). In fact, checking the well-formedness of objects is as hard as checking if they satisfy specifications: $Obj : SP$ if and only if $(\lambda X{:}SP.\ \text{(any constant)})(Obj)$ is well-formed.

An easy corollary to the soundness theorem is the following:

Corollary 7.6 Any well-formed object Obj has a well-defined meaning $[\![Obj]\!]$. □

Since specifications do not form a separate syntactic category of the language, in the above discussion we have used the term "specification" and the meta-variable SP rather informally, relying on an intuitive understanding of the role of the objects of the language. This intuitive understanding may be made formal as follows:

Definition 7.7 An object SP is called a *specification* if for some SP_{any}, $SP : \mathbf{Spec}(SP_{any})$. □

Corollary 7.8 The meaning of a specification is a class of values: if $SP : \mathbf{Spec}(SP_{any})$ then $[\![SP]\!] \subseteq [\![SP_{any}]\!]$. □

Note that this covers ordinary Σ-specifications, specifications of (higher-order) parametric algebras, specifications of (higher-order) parameterised specifications, etc. The following theorem shows that this is indeed consistent with our previous informal use of the term.

Theorem 7.9 If $Obj : SP$ then SP is a specification. □

Even though this theorem captures an intuitively rather obvious fact, its inductive proof (given in [ST 91a], omitted here) is surprisingly long and relatively complicated. Unfortunately, this seems to be typical of many proofs dealing with "syntactic" properties of λ-calculi.

7.4 Type-checking

Inference in the system presented in the previous section has a purely "type-checking" component on which the "verification" component is in a sense superimposed. We try to separate this "type-checking" process below. The concept of type we use must cover signatures (as "basic types" of algebras) and "arrow types" (types of functions) which would be usual in any type theory, as well as "specification types" which are particular to the formalism presented here: as we have stressed before, the type of a specification is distinct from the type of objects the specification specifies.

Definition 7.10 The class of types $\mathcal{T}$ is defined as the least class such that:

- for any signature Σ, $\Sigma \in \mathcal{T}$;
- for any types $\tau_1, \tau_2 \in \mathcal{T}$, $\tau_1 \rightarrow \tau_2 \in \mathcal{T}$; and
- for any type $\tau \in \mathcal{T}$, $\mathbf{Spec}(\tau) \in \mathcal{T}$.

$\square$

Under the standard notational convention that arrow types of the form $\tau \rightarrow \tau'$ stand for Π-types of the form $\Pi X{:}\tau.\, \tau'$ where X does not actually occur in τ', types as defined above are well-formed specifications.

We define type $Type(Obj)$ for an object Obj of our system by induction as follows:

Simple specifications:

$$\frac{\Sigma \text{ signature}}{Type(\Sigma) = \mathbf{Spec}(\Sigma)} \qquad \frac{Type(SP) = \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{Type(\textbf{impose } \Phi \textbf{ on } SP) = \mathbf{Spec}(\Sigma)}$$

... and similarly for other simple specifications ...

Other specifications:

$$\frac{Type(Obj) = \tau}{Type(\{Obj\}) = \mathbf{Spec}(\tau)} \qquad \frac{Type(SP) = \mathbf{Spec}(\tau)}{Type(\mathbf{Spec}(SP)) = \mathbf{Spec}(\mathbf{Spec}(\tau))}$$

$$\frac{Type(SP) = \mathbf{Spec}(\tau) \qquad \begin{array}{c}[Type(X) = \tau] \\ Type(SP') = \mathbf{Spec}(\tau')\end{array}}{Type(\Pi X{:}SP.\, SP') = \mathbf{Spec}(\tau \rightarrow \tau')}$$

Union:

$$\frac{Type(SP_1) = \mathbf{Spec}(\tau) \qquad Type(SP_2) = \mathbf{Spec}(\tau)}{Type(SP_1 \cup SP_2) = \mathbf{Spec}(\tau)}$$

λ-expressions:

$$\frac{Type(SP) = \mathbf{Spec}(\tau) \qquad \begin{array}{c}[Type(X) = \tau] \\ Type(Obj) = \tau'\end{array}}{Type(\lambda X{:}SP.\, Obj) = \tau \rightarrow \tau'} \qquad \frac{Type(Obj) = \tau \rightarrow \tau' \qquad Type(Obj') = \tau}{Type(Obj(Obj')) = \tau'}$$

Algebra expressions:

$$\frac{A \text{ is an algebra expression denoting a } \Sigma\text{-algebra}}{Type(A) = \Sigma}$$

Note that the semantic inference rules (R12), (R13), the trivial inference rule (R10), the "cut" rule (R11), (R5) and the β-reduction and β-expansion rules (R8) and (R9), which do not introduce new well-formed objects, do not have counterparts in the above definition.

Clearly, the above definition depends on a judgement whether or not an algebra expression denotes an algebra over a given signature. We will assume that such "type-checking" of algebra expressions is defined externally in such a way that it is consistent with the semantics (i.e., if A is a well-formed algebra expression denoting a Σ-algebra then indeed $[\![A]\!] \in Alg(\Sigma)$). Moreover, we will assume that it is substitutive: if A is an algebra expression denoting a Σ-algebra under an assumption $Type(X) = \tau$ then for any object Obj with $Type(Obj) = \tau$, $A[Obj/X]$ is an algebra expression denoting a Σ-algebra as well.

The above rules (deliberately) do not define $Type(Obj)$ for all object expressions of our language. However, if a type is defined for an object, it is defined unambiguously. An object Obj is *roughly well-formed* if its type $Type(Obj)$ is defined. There are, of course, roughly well-formed objects that are not well-formed. The opposite implication holds, though:

Theorem 7.11 $Type(Obj)$ is well-defined for any well-formed object Obj. In particular:

1. If $Obj : SP$ then $Type(SP) = \mathbf{Spec}(Type(Obj))$.
2. If SP is a specification then $Type(SP) = \mathbf{Spec}(\tau)$ for some type τ.
3. If $Obj : \Pi X{:}SP.\, SP'$ then $Type(Obj) = \tau \rightarrow \tau'$, where $Type(SP) = \mathbf{Spec}(\tau)$, for some types τ and τ'.

□

We omit the proof here: the first part of the theorem follows by induction on the length of the derivation of $Obj : SP$ (this proof is sketched in [ST 91a]). The other two parts follow directly from this.

The above theorem states that a necessary condition for an object to satisfy a specification is that both are roughly well-formed and the type of the object is consistent with the type of the specification. Of course, nothing like the opposite implication holds. As pointed out earlier, proving that an object satisfies a specification must involve a verification process as embodied in the two rules of semantic inference.

One might now expect that any well-formed object Obj "is of its type", i.e. $Obj : Type(Obj)$. This is not the case, though. The problem is that both λ- and Π-expressions include parameter *specifications* rather than just parameter *types*, and so functions denoted by λ-expressions and specified by Π-expressions have domains defined by specifications, not just by types. This is necessary for methodological reasons: we have to be able to specify permissible arguments in a more refined way than just by giving their types. However, as a consequence, objects denoted by λ- and Π-expressions in general do not belong to the domain defined by their types, and so we cannot expect that such expressions would "typecheck" to their types.

To identify the purely "type-checking" component in our system we have to deal with objects where parameter specifications are replaced by their types. Formally, for any roughly well-formed object Obj, its version $Erase(Obj)$ with parameter specifications erased is defined by "rounding up" parameter specifications to parameter types. A full inductive definition is given in [ST 91a]; two crucial cases are:

$$\begin{array}{rcl} Erase(\Pi X{:}SP.\, SP') & =_{def} & \Pi X{:}\tau.\, Erase(SP') \\ & & \quad \text{where } Type(Erase(SP)) = \mathbf{Spec}(\tau) \\ Erase(\lambda X{:}SP.\, Obj) & =_{def} & \lambda X{:}\tau.\, Erase(Obj) \\ & & \quad \text{where } Type(Erase(SP)) = \mathbf{Spec}(\tau) \end{array}$$

Now, by a tedious but straightforward induction one may show that for any roughly well-formed object Obj, $Erase(Obj)$ is well-formed and has the same type as Obj. Joining this with Theorem 7.11, we conclude that a necessary condition for an object Obj to satisfy a specification is that $Erase(Obj)$, the version of the object where parameter specifications have been "rounded up" to parameter types, has a type which is consistent with the type of the specification. This necessary condition embodies the purely type-checking component of any proof that an object satisfies a specification.

It is important to realize that the type-checking of $Erase(Obj)$ may be performed within the original system, since $Type(Erase(Obj)) = \tau$ if and only if $Erase(Obj) : \mathbf{Spec}(\tau)$. Moreover, this can be done separately from the semantic verification part, without any reference to the meanings of objects and specifications. We present below the corresponding proper fragment of the original system:

Simple specifications:

$$\frac{\Sigma \text{ signature}}{\Sigma : \mathbf{Spec}(\Sigma)} \qquad \frac{SP : \mathbf{Spec}(\Sigma) \qquad \Phi \subseteq Sen(\Sigma)}{\mathbf{impose}\ \Phi\ \mathbf{on}\ SP : \mathbf{Spec}(\Sigma)}$$

... and just as before for other simple specifications ...

Other specifications:

$$\frac{Obj : \tau}{\{Obj\} : \mathbf{Spec}(\tau)} \qquad \frac{\begin{array}{c}[X : \tau]\\ SP' : \mathbf{Spec}(\tau')\end{array}}{\Pi X{:}\tau.\, SP' : \mathbf{Spec}(\tau \rightarrow \tau')} \qquad \frac{SP : \mathbf{Spec}(\tau)}{\mathbf{Spec}(SP) : \mathbf{Spec}(\mathbf{Spec}(\tau))}$$

Union:

$$\frac{SP_1 : \mathbf{Spec}(\tau) \qquad SP_2 : \mathbf{Spec}(\tau)}{SP_1 \cup SP_2 : \mathbf{Spec}(\tau)}$$

λ-expressions:

$$\frac{\begin{array}{c}[X : \tau]\\ Obj : \tau'\end{array}}{\lambda X{:}\tau.\, Obj : \tau \rightarrow \tau'} \qquad \frac{Obj : \tau \rightarrow \tau' \qquad Obj' : \tau}{Obj(Obj') : \tau'}$$

Algebra expressions:

$$\frac{A \text{ is an algebra expression denoting a } \Sigma\text{-algebra}}{A : \Sigma}$$

We hope that a comparison of the above with the system presented in Section 7.3 will clearly illustrate the intuitive difference between typed λ-calculi, like the one above, and "specified" λ-calculi, like the one in Section 7.3.

7.5 An example

Example 7.12 Let us look again at Example 5.2.

First, since $Elem$ is just a signature, $Type(Elem) = \mathbf{Spec}(Elem)$. Moreover, we have

$$\mathrm{Elem} : \mathbf{Spec}(\mathrm{Elem})$$

Then, *P-Bunch* as defined before is well-formed (modulo the necessary translation of +, **enrich** and **reachable** into the operations provided by the system) and has type $\mathbf{Spec}(\mathit{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\mathit{Bunch}})$. Again, because *Elem* is a trivial specification, we have

$$\text{P-Bunch} : \mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Bunch}})$$

It is possible, however, to derive a much tighter specification of *P-Bunch* than its type:

$$\text{P-Bunch} : \Pi E{:}\mathbf{Spec}(\text{Elem}).\ \mathbf{Spec}(\text{P-Bunch}(E))$$

Then another, perhaps more adequate, version of *P-Delete* may be defined as follows:

$$\begin{array}{lll} \text{P-Delete}' =_{def} & \lambda \mathit{BSP}{:}(\Pi E{:}\mathbf{Spec}(\text{Elem}).\ \mathbf{Spec}(\text{P-Bunch}(E))). & \\ & \quad \lambda \mathit{ESP}{:}\mathbf{Spec}(\text{Elem}). & \\ & \qquad \mathbf{enrich}\ \mathit{BSP}(\mathit{ESP}) & \\ & \qquad \mathbf{by}\ \ \mathbf{opns} & \text{delete} : \text{elem} \times \text{bunch} \rightarrow \text{bunch} \\ & \qquad \phantom{\mathbf{by}}\ \ \mathbf{axioms} & \text{count}(a, \text{delete}(a, B)) = 0 \\ & & a \neq b \Rightarrow \text{count}(a, \text{delete}(b, B)) = \text{count}(a, B) \end{array}$$

Then

$$\mathit{Type}(\text{P-Delete}') = (\mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Bunch}})) \rightarrow (\mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Delete}}))$$

much as in Example 5.2. However, we do *not* have

$$\text{P-Delete}' : (\mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Bunch}})) \rightarrow (\mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Delete}}))$$

The type of *P-Delete*$'$, viewed as a specification, requires the specified objects (which are higher-order parameterised specifications) to be applicable to any specification of the type $\mathbf{Spec}(\mathit{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\mathit{Bunch}})$, which is not the case with *P-Delete*$'$ as defined here.

We can, however, show that

$$\text{P-Delete}' : (\Pi E{:}\mathbf{Spec}(\text{Elem}).\ \mathbf{Spec}(\text{P-Bunch}(E))) \rightarrow (\mathbf{Spec}(\text{Elem}) \rightarrow \mathbf{Spec}(\Sigma_{\text{Delete}}))$$

and the tightest specification we can derive for *P-Delete*$'$ is

$$\text{P-Delete}' : \Pi B{:}\big(\Pi E{:}\mathbf{Spec}(\text{Elem}).\ \mathbf{Spec}(\text{P-Bunch}(E))\big).\ \big(\Pi E{:}\mathbf{Spec}(\text{Elem}).\ \mathbf{Spec}(\text{P-Delete}'(B)(E))\big)$$

□

8 Concluding remarks

In this paper we have discussed parameterisation and its role in the process of software specification and development. We have especially stressed two points. The first is that there should be a clear distinction between parameterised specifications and specifications of parameterised software:

$$\text{parameterised (program specification)} \neq \text{(parameterised program) specification}$$

Both concepts are important and useful, but they are modelled by different semantical objects and, more significantly, they play different roles in the process of software development. The methodological consequences of this distinction were discussed in detail in Section 4.3.

The second point is that it is natural and useful to generalise parameterisation to the higher-order case. Specifications of higher-order parametric program modules arise naturally and give extra flexibility in the process of systematic software development. This was discussed in Sections 5 and 6.

Spurred by these methodological considerations, in Section 7 we introduced an institution-independent specification formalism that provides a notation for parameterised specifications and specifications of parametric objects of an arbitrary order, as well as any mixture of these concepts. The formalism incorporates the kernel specification-building operations described in [ST 88a] based on those in the ASL specification language [SW 83], [Wir 86]. The basic idea was to treat specifications, which specify objects, as objects themselves. This collapsing together of the two levels, that of objects and that of their specifications, led (perhaps surprisingly) to a well-behaved inference system for proving that an object satisfies a specification with a clearly identified formal type-checking component.

The formalism presented deals explicitly with two levels of objects involved in the process of software development: programs (viewed as algebras) and their specifications (viewed as classes of algebras) — both, of course, arbitrarily parameterised. Aiming at the development of an institution-independent framework, we decided to omit from our considerations yet another level of objects involved, namely that of algebra components (such as data values and operations on them). In particular institutions, however, it may be interesting to explicitly consider this level as well, and to intermix constructs for dealing with this level with those for the other two levels mentioned above. This would lead to entities such as algebras parametric on data values, specifications parameterised by functions on data, functions from algebras and specifications to data values, etc.

Just as the kernel ASL-like specification formalism it builds on, the presented system is too low-level to be directly useful in practice. We view it primarily as a kernel to be used as a semantic foundation for the development of more user-friendly specification languages. Easier to use notations can be devised, with their semantics defined by translation into the formalism of Section 7.

The material in Section 7 is more tentative than that in the remainder of the paper, and clearly some of the details of the design of the specification formalism deserve further consideration. The choice of operations used to build simple specifications is not essential; we have chosen here those of ASL (**derive**, **translate**, etc.) but any reasonably expressive set of operations would suffice, and most of the subsequent technical development would require little or no modification. Adding e.g. an intersection operation (dual to union) to the present system would be completely unproblematic. A less straightforward extension would be to add recursion for building specifications as in [SW 83], [ST 88a]: for a parameterised specification P, **fix** P would be a specification denoting the greatest fixed point of the (monotone) function $[\![P]\!]$ on classes of objects. Yet another thing to consider is the possible benefits of making the $(\_)^\sharp$ and $(\_)^\dagger$ operators of Sections 4.2 explicitly available. It is not clear how the system of rules in Section 7.3 could be enriched to cope with these additions though.

The presented system provides an appropriate foundation for the Extended ML specification language and program development methodology as presented in [ST 89]. Indeed, one of the main stimuli to write this paper was our inability to express the semantics of the current version of Extended ML directly in terms of the kernel specification-building operations in ASL: Extended ML functor specifications are specifications of parametric objects, and these were not present in ASL. The task of writing out a complete semantics of Extended ML in terms of the specification formalism presented here remains to be done. We expect that some technicalities, like those which arise in connection with ML type inheritance, will cause the same problems as in [ST 89]. Some others, like the use of behavioural equivalence and the concept of functor stability in the Extended ML methodology, although directly related to the **abstract** operation in the formalism presented here, require further study in this more general framework. Finally, properties of ML functors such as persistency, which cause difficulties in other specification formalisms, will be easy to express here.

One of the interesting possibilities the system presented in Section 7 offers is that it incorporates the concept of specification refinement, cf. [ST 88b]. Namely, we can define $SP \rightsquigarrow SP'$ (read: SP' is a refinement of SP — this is equivalent to $SP \underset{id}{\leadsto} SP'$ in the notation of Section 6) as $SP' : \mathbf{Spec}(SP)$. This also covers refinements of specifications of (higher-order) parametric algebras, due to the following derivable rule:

$$\frac{\begin{array}{c}[X : SP] \\ SP' \leadsto SP''\end{array} \qquad \text{(i.e. } SP'' : \mathbf{Spec}(SP')\text{)}}{\Pi X{:}SP.\, SP' \leadsto \Pi X{:}SP.\, SP'' \qquad \text{(i.e. } \Pi X{:}SP.\, SP'' : \mathbf{Spec}(\Pi X{:}SP.\, SP')\text{)}}$$

We can also "internalize" $SP \underset{\kappa}{\leadsto} \langle SP_1, \ldots, SP_n \rangle$, for example as $\kappa : \Pi X_1{:}SP_1. \cdots \Pi X_n{:}SP_n.\, SP$. The consequences of such an internalisation of development steps seem worth exploring. This was one of the ideas underlying the design of the SPECTRAL specification language [KS 91] which can be seen as a higher-level and more user-friendly version of a subset of the formalism presented here.

The formal properties of the system presented in Section 7 need much further study. For example, it seems that the "cut" rule should be admissible (although not derivable) in the remainder of the system. The standard properties of β-reduction, such as the Church-Rosser property and termination (on well-formed objects) should be carefully proven, probably by reference to the analogous properties of the usual typed λ-calculus. For example, the termination property of β-reduction on the well-formed objects of the language should follow easily from the observation that the *Erase* function introduced in Section 7.4 preserves β-reduction, which allows us to lift the corresponding property of the usual typed λ-calculus to our formalism. The system is incomplete, as pointed out earlier. It would be useful to identify all the sources of this incompleteness, for example by characterising an interesting subset of the language for which the system is complete. One line of research which we have not followed (as yet) is to try to encode the formalism we present here in one of the known type theories (for example, Martin-Löf's system [NPS 90], the calculus of constructions [CH 88] or LF [HHP 87]). It would be interesting to see both which of the features of the formalism we propose would be difficult to handle, as well as which of the tedious proofs of some formal properties of our formalism (cf. the proofs sketched for Theorems 7.9 and 7.11 in [ST 91a]) would turn out to be available for free under such an encoding.

Note added in proof.
Our attention has recently been drawn to certain intriguing similarities between some of the rules presented in Section 7.3 and those in the paper "Structural subtyping and the notion of power type" by Luca Cardelli in *Proc. 15th ACM Symp. on Principles of Programming Languages*, San Diego, 70–79 (1988). Among other things, the **Spec** operator here is closely related to Cardelli's Power type-formation operator, our "cut" rule (R11) corresponds to his Power elimination rule, and our rule (R3) corresponds to his Power subtyping rule.

Acknowledgements: Thanks to David Aspinall, Jan Bergstra, Jordi Farrés, John Fitzgerald, Marie-Claude Gaudel, Joseph Goguen, Claudio Hermida, Cliff Jones, Stefan Kahrs, Ed Kazmierczak, Bernd Krieg-Brückner, Jacek Leszczyłowski, Fernando Orejas, Lincoln Wallen and Martin Wirsing for interesting discussions and suggestions on this topic. Thanks to the referees for comments on an earlier version of the paper.

This research was supported by the University of Edinburgh, the University of Bremen, the Technical University of Denmark, the University of Manchester, and by grants from the Polish Academy of Sciences, the (U.K.) Science and Engineering Research Council, ESPRIT, and the Wolfson Foundation.

9 References

[Note: LNCS n = Springer Lecture Notes in Computer Science, Volume n]

[Bau 85] Bauer, F.L. *et al* (the CIP language group). *The Wide Spectrum Language CIP-L.* LNCS 183 (1985).

[Bid 88] Bidoit, M. The stratified loose approach: a generalization of initial and loose semantics. *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland. LNCS 332, 1–22 (1988).

[BKS 88] Borzyszkowski, A.M., Kubiak, R. and Sokołowski, S. A set-theoretic model for a typed polymorphic λ-calculus. *Proc. VDM-Europe Symp. VDM – The Way Ahead*, Dublin. LNCS 328, 267–298 (1988).

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of CLEAR, a specification language. *Proc. of Advanced Course on Abstract Software Specification*, Copenhagen. LNCS 86, 292–332 (1980).

[Bir 48] Birkhoff, G. *Lattice Theory.* American Mathematical Society (1948).

[BT 83] Blikle, A. and Tarlecki, A. Naive denotational semantics. *Information Processing 83, Proc. IFIP Congress'83*, Paris, ed. R. Mason, 345–355, North-Holland 1983.

[Coh 81] Cohn, P.M. *Universal Algebra.* Reidel (1981).

[Con 86] Constable, R.L. *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall (1986).

[CH 88] Coquand, T. and Huet, G. The calculus of constructions. *Information and Computation* 76, 95–120 (1988).

[Ehr 82] Ehrich, H.-D. On the theory of specification, implementation, and parametrization of abstract data types. *Journal of the Assoc. for Computing Machinery 29*, 206–227 (1982).

[EM 85] Ehrig, H. and Mahr, B. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics.* Springer (1985).

[ETLZ 82] Ehrig, H., Thatcher, J.W., Lucas, P. and Zilles, S.N. Denotational and initial algebra semantics of the algebraic specification language LOOK. Report 84-22, Technische Universität Berlin (1982).

[Far 92] Farrés-Casals, J. Verification in ASL and Related Specification Languages. Ph.D. thesis, report CST-92-92, Dept. of Computer Science, Univ. of Edinburgh (1992).

[FJ 90] Fitzgerald, J.S. and Jones, C.B. Modularizing the formal description of a database system. *Proc. VDM'90 Conference*, Kiel. Springer LNCS 428 (1990).

[Gan 83] Ganzinger, H. Parameterized specifications: parameter passing and implementation with respect to observability. *Trans. Prog. Lang. Syst. 5*, 318–354 (1983).

[GM 88] Gaudel, M.-C. and Moineau, T. A theory of software reusability. *Proc. 2nd European Symp. on Programming*, Nancy. LNCS 300, 115–130 (1988).

[Gog 84] Goguen, J.A. Parameterized programming. *IEEE Trans. Software Engineering SE-10*, 528–543 (1984).

[GB 84] Goguen, J.A. and Burstall, R.M. Introducing institutions. *Proc. Logics of Programming Workshop*, Carnegie-Mellon. LNCS 164, 221–256 (1984).

[GHW 85] Guttag, J.V., Horning, J.J. and Wing, J. Larch in five easy pieces. Report 5, DEC Systems Research Center, Palo Alto, CA (1985).

[HHP 87] Harper, R., Honsell, F. and Plotkin, G. A framework for defining logics. *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Cornell, 194–204 (1987).

[KS 91] Krieg-Brückner, B. and Sannella, D. Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in SPECTRAL. *Proc. Colloq. on Combining Paradigms for Software Development, Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, Brighton. Springer LNCS 494, 313–336 (1991).

[LL 88] Lehmann, T. and Loeckx, J. The specification language of OBSCURE. *Recent Trends in Data Type Specification, Selected Papers from the 5th Workshop on Specification of Abstract Data Types*, Gullane, Scotland. LNCS 332, 131–153 (1988).

[MacQ 86] MacQueen, D.B. Modules for Standard ML. in: Harper, R., MacQueen, D.B. and Milner, R. Standard ML. Report ECS-LFCS-86-2, Univ. of Edinburgh (1986).

[MMMS 87] Meyer, A.R., Mitchell, J.C., Moggi, E. and Statman, R. Empty types in polymorphic lambda calculus. *Proc. 14th ACM Symp. on Principles of Programming Languages*, 253–262; revised version in *Logical Foundations of Functional Programming*, ed. G. Huet, Addison-Wesley (1990), 273–284.

[Mos 89a] Mosses, P. Unified algebras and modules. *Proc. 16th ACM Symp. on Principles of Programming Languages*, Austin, 329–343 (1989).

[Mos 89b] Mosses, P. Unified algebras and institutions. *Proc. 4th IEEE Symp. on Logic in Computer Science*, Asilomar, 304–312 (1989).

[NPS 90] Nordström, B., Petersson, K. and Smith, J.M. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford Univ. Press (1990).

[San 91] Sannella, D. Formal program development in Extended ML for the working programmer. *Proc. 3rd BCS/FACS Workshop on Refinement*, Hursley Park. Springer BCS Workshop series (1991).

[SST 90] Sannella, D., Sokołowski, S. and Tarlecki, A. Toward formal development of programs from algebraic specifications: parameterisation revisited. Report 6/90, FB Informatik, Universität Bremen (1990).

[ST 85] Sannella, D. and Tarlecki, A. Program specification and development in Standard ML. *Proc. 12th ACM Symp. on Principles of Programming Languages*, New Orleans, 67–77 (1985).

[ST 87] Sannella, D. and Tarlecki, A. On observational equivalence and algebraic specification. *J. Comp. and Sys. Sciences 34*, 150–178 (1987).

[ST 88a] Sannella, D. and Tarlecki, A. Specifications in an arbitrary institution. *Information and Computation 76*, 165–210 (1988).

[ST 88b] Sannella, D. and Tarlecki, A. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica 25*, 233–281 (1988).

[ST 89] Sannella, D. and Tarlecki, A. Toward formal development of ML programs: foundations and methodology. Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (1989); extended abstract in *Proc. Colloq. on Current Issues in Programming Languages*, Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Barcelona. LNCS 352, 375–389 (1989).

[ST 91a] Sannella, D. and Tarlecki, A. A kernel specification formalism with higher-order parameterisation. *Proc. 7th Intl. Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 274–296 (1991).

[ST 91b] Sannella, D. and Tarlecki, A. Extended ML: past, present and future. *Proc. 7th Intl. Workshop on Specification of Abstract Data Types*, Wusterhausen. Springer LNCS 534, 297–322 (1991).

[ST 92] Sannella, D. and Tarlecki, A. Toward formal development of programs from algebraic specifications: model-theoretic foundations. *Proc. Intl. Colloq. on Automata, Languages and Programming*, Vienna. Springer LNCS 623, 656–671 (1992).

[SW 83] Sannella, D. and Wirsing, M. A kernel language for algebraic specification and implementation. *Proc. Intl. Conf. on Foundations of Computation Theory*, Borgholm, Sweden. LNCS 158, 413–427 (1983).

[Sch 87] Schoett, O. Data abstraction and the correctness of modular programming. Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

[Sok 90] Sokołowski, S. Parametricity in algebraic specifications: a case study. Draft report, Institute of Computer Science, Polish Academy of Sciences, Gdańsk (1990).

[Tar 92] Tarlecki, A. Modules for a model-oriented specification language: a proposal for MetaSoft. *Proc. 4th European Symp. on Programming*, Rennes. Springer LNCS 582, 451–472 (1992).

[Voß 85] Voß, A. Algebraic specifications in an integrated software development and verification system. Ph.D. thesis, Universität Kaiserslautern (1985).

[Wir 86] Wirsing, M. Structured algebraic specifications: a kernel language. *Theor. Comp. Science 42*, 123–249 (1986).