Towards a New Way of Parameterization*

Radu Grosu Dieter Nazareth

Technische Universität München, Fakultät für Informatik
D-80290 München, Germany

E–Mail: {grosu,nazareth}@informatik.tu-muenchen.de

Abstract

Classical approaches to parameterization in axiomatic specification languages require the user to *explicitly* handle specification instantiation. This often makes specifications less readable and manageable. We therefore present a new parameterization mechanism which allows *implicit* instantiation. However, since this mechanism is less powerful as the first one we show how to combine them to achieve both elegance and power. We included both mechanisms in the specification language SPECTRUM.

1 Introduction

Parameterization has a long history and was from the very beginning recognized as one of the most important structuring mechanisms in specification construction. Essentially, parameterization for specifications corresponds to λ–abstraction for functions: it allows to designate some parts of a specification as formal parameters. The specification abstraction obtained is called *parameterized* and can subsequently be instantiated with actual specifications "fitting" the formal ones.

Parameterized specifications were first used in the specification language CLEAR [2]. Since then, plenty of approaches were developed with different semantics or abstraction and instantiation mechanisms. Among them, probably best known are the CLEAR–style parameterized specifications, the ASL–style parameterized specifications [17], the ACT-ONE–style parameterized specifications [4, 5] and the (Extended) ML functor specifications [16][1]. In all these approaches the user must *explicitly* deal with specification instantiation. This makes the specification language very powerful but also more awkward to use.

A very interesting attitude was adopted in the functional language ML [10] where the formal parameters are restricted to sorts. This restriction decreases the abstraction level to sort abstraction but it enables the definition of a type inference algorithm. As a consequence the user is not constrained anymore to deal explicitly with sort instantiation. This is performed *implicitly* by the system. This kind of parameterization is known as ML-style parametric polymorphism. It is easy to use but also less powerful. Therefore ML also provides for *functor* declarations.

A significant improvement of the implicit parameterization mechanism was done by introducing *type classes* in the language Haskell (see [11]). Sort variables belonging to a class are restricted to range only over sorts owning the functions stated in the class declaration. From an algebraic point of view, class declarations can be understood as formal parameter specifications which are restricted to have only one *sort of interest* (i.e. one sort variable) and no axioms. In other words type classes are a step towards specification abstraction which further maintains the pleasant features of an implicit parameterization mechanism. They also give an elegant solution to the overloading problem of ML. However, Haskell type classes are not as powerful as ML functors. The absence of a functor mechanism is one of the most criticized points in the Haskell design. Several efforts are currently made to improve Haskell in this respect.

The SPECTRUM specification language philosophy is to give the user maximal flexibility and manageability without losing specification power. This is why we provide the user both with a type class mechanism and with a parameterization mechanism.

Our classes are similar to the Haskell ones but have a different semantics (see [7] for details). Moreover, they are also allowed to contain axioms. As a consequence they provide an easy-to-use alternative to a very large number of classical parameterized specifications.

*This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM.

[1] For a comparison of the above approaches see [15].

In order not to decrease specification power we also provide a "classical" parameterization mechanism. Despite of their conceptual overlapping both mechanisms fit together well. Firstly, it is obvious that parameterization will be used only in cases where type classes alone are not powerful enough. Secondly, it turns out that specifications having a type class polymorphism eliminate a parameterization nesting level if used as formal parameters. This improves the readability and flexibility of the classical parameterization mechanism.

The paper is structured as follows. In Section 2 we give a short presentation of the language SPECTRUM. This is by no means complete and it is only intended to aid the understanding of the examples from this paper. A thorough presentation can be found in [1]. In Section 3 we review the syntax and semantics of parameterized specifications. Section 4 and 5 show how parametric polymorphism and type classes can be used to model parameterization. These will be combined with an ASL–like parameterization in Section 6. Finally in Section 7 we draw some conclusions and point out some future work.

2 The Specification Language SPECTRUM

SPECTRUM is an axiomatic specification language which borrows concepts both from algebraic languages (e.g. LARCH [9]) and from type theoretic languages (e.g. LCF [20]). An informal presentation with many examples illustrating its properties is given in [1]. We briefly summarize its principal characteristics.

As in algebraic specification languages, a SPECTRUM specification consists of a signature and an axioms part. However, in contrast to most algebraic languages, the semantics of a specification in SPECTRUM is loose, i.e. it is not restricted to initial models or even term generated ones. Loose semantics leaves a large degree of freedom for later implementations. SPECTRUM is also not restricted to equational or conditional-equational axioms, since it does not primarily aim at executable specifications. One can use full first order logic to write very abstract and non-executable specifications or only use its constructive part to write specifications which can be understood and executed as programs.

The influence from type theory is twofold. On the type level SPECTRUM uses shallow predicative polymorphism with type classes. The other influence of type theory is in the language of terms and their underlying semantics. SPECTRUM incorporates the entire notation for typed λ-terms. As a consequence functions are first class citizens. The models of SPECTRUM specifications are assumed to be certain continuous algebras. Therefore SPECTRUM supports partial and non–strict functions as well as higher–order functions in the sense of domain theory.

For example, the following specification defines a transitive relation[2]:

```
Transitive = {
    --The Signature
    sort Elem;
    .≤. : Elem × Elem → Bool        prio 6;
    --The Axioms
    .≤. strict total;               --Definedness axioms
    axioms ∀ a, b, c : Elem in
        a ≤ b ∧ b ≤ c ⇒ a ≤ c;
    endaxioms; }
```

It introduces the sort Elem and the boolean function .≤. in the signature part. The properties of .≤. are stated in the axioms part. Every algebra which has the above signature and satisfies the axioms is a model of this specification.

Since writing well-structured specifications is one of our main goals, a flexible language for structuring specifications has been designed for SPECTRUM. This structuring is achieved by using so-called specification building operators which map a list of argument specifications into a result specification. The language for these operators was originally inspired by ASL [17]. The current version also borrows concepts from Haskell, LARCH and PLUSS [6]. In this paper we will use only the .+., **rename** and **enriches** operators[3].

Two specifications SP_1 and SP_2 are combined by writing $SP_1 + SP_2$. Combination is modeled by taking the union of signatures and of axioms. As a consequence the properties of an identifier occurring both in SP_1 and in SP_2 are combined in $SP_1 + SP_2$. To avoid this effect by keeping identifiers from coinciding, to force it by making identifiers to coincide or to give a more appropriate name to an identifier, SPECTRUM provides the **rename** operator.

[2] Comments are preceded by --.

[3] For a full description see [1].

The process of building specifications hierarchically, by adding new sort and function symbols together with their corresponding axioms to a given specification, can be expressed as follows:

SP' = { **enriches** SP; ... new spec. text ... }

We are now able to discuss the parameterization and polymorphism concepts.

3 Parameterization Overview

In most algebraic specification languages a parameterized specification consists of two parts: the formal parameters part and the body. The formal parameters part, also called *specification interface,* describes syntactically (by the signature) and semantically (by the axioms) the required properties for the actual parameters. The *body* describes how actual parameters are used to build the result specification. For example, a parameterized specification of lists can be defined in SPECTRUM as follows:

LIST_{param} =
param X = {**sort** Elem};
body { **enriches** X;
−−Constructors
[]: List;
cons: Elem × List → List;
first: List → Elem;
rest: List → List;
.⊕.: List × List → List **prio 6:left**;
cons, first, rest, .⊕. **strict**; cons, rest, .⊕. **total**;
List **freely generated by** [], cons;
axioms ∀ a : Elem, l, m : List **in**
first [] = ⊥;
first(cons(a,l)) = a;
rest [] = [];
rest(cons(a,l)) = l;
[]⊕l = l;
cons(a,m)⊕l = cons(a,m⊕l);
endaxioms; }

In this case the interface is trivial and it only requires the existence of a sort. Since the purpose of the interface is to *select* and not to construct algebras, it usually does not contain constructors for the constrained sorts. This is the reason why the interface is often called *requirement theory.*

There was a long debate about the semantics of a parameterized specification (see [15]). From a syntactical point of view parameterization can be understood as "typed" specification abstraction. The actual parameters must imply the formal ones (modulo a signature morphism). Hence, a parameterized specification can be considered as a function taking specifications into specifications.

From a semantical point of view a parameterized specification can be understood as a function taking algebras satisfying the requirement specification into algebras satisfying the specification body[4]. More generally, if the body of the parameterized specification is loosely interpreted, one can obtain for one actual parameter algebra a set of body algebras. Therefore, in this view a parameterized specification is actually a set of functions. In [15] this is called a specification of parametric algebras.

4 Parameterization with Parametric Polymorphism

Strachey [18] distinguished between two major kinds of polymorphism, parametric polymorphism and ad-hoc polymorphism. *Parametric polymorphism* is obtained when a function works uniformly on an infinite range of sorts with a common structure. This is achieved by abstracting from a concrete sort in the sort expression of a function. *Ad-hoc polymorphism*, in contrast, allows a function to work differently on finitely many sorts not

[4]Since the algebras are usually allowed to contain more sorts and functions as the requirement signature, it is actually the reduct who must satisfy the requirement theory.

necessarily having a related structure. This kind of polymorphism is better known as *overloading* and can be simulated in SPECTRUM (see [1]).

The parametric polymorphism adapted for SPECTRUM is the simple predicative polymorphism that can be found in many modern functional programming languages, e.g. ML[5]. In contrast to more complex versions of parametric polymorphism like e.g. System F [8], ML polymorphism has no constructs for explicit sort abstraction or application. A sort inference system computes the implicit instantiations of polymorphic values by static program analysis. This feature enables a very simple parameterization mechanism without explicit instantiation and renaming.

In the parameterized specification LIST_{param} the sort List is defined as the sort of interest. The elements of this sort are constructed by the constructors [] and cons. The construction is based on the basic sort Elem. The result sort List, however, does not reflect the fact that the constructed lists have elements of sort Elem. This information is lost. If the specification is instantiated by a concrete specification, we need a renaming mechanism to distinguish between different kinds of lists. This deficiency can be overcome by the use of sort constructors.

Sort constructors are function symbols on the sort level denoting domain constructors which take a (possibly empty) sequence of domains as argument and yield a domain as result[6]. The user can e.g. define a unary sort constructor List which, applied to an arbitrary sort, yields the sort of lists where the elements are of the sort given as parameter. Sort constructors of arity zero like e.g. Nat are called *basic sorts*.

In parametric polymorphism sort abstraction is achieved by the use of *sort variables*. Sort variables take the role of absolutely free sorts in parameterized specifications. Sort variables and sort constructors are the basic elements of our sort language, used to build complex sort expressions. If the sort constructor List and the basic sort Nat are defined, the following examples are correct sort expressions, where α is a sort variable and $\times$ and $\longrightarrow$ are built-in infix sort constructors:

List (List Nat)
$\alpha \times \mathsf{List}\ \alpha \longrightarrow \mathsf{List}\ \alpha$

In the example above the sort variable α can be instantiated by an arbitrary sort expression. In the signature of a specification we can define a polymorphic value by using sort variables in the sort expression of the definition. We can e.g. define a polymorphic list constructor in the following way:

cons: $\alpha \times \mathsf{List}\ \alpha \longrightarrow \mathsf{List}\ \alpha$;

Now we give a polymorphic list specification which is in some sense equivalent to the parameterized specification LIST_{param}. The example shows how parameterization is replaced by parametric polymorphism.

LIST_{poly} = {
sort List α; ——*Sort constructor List*
[]: List α; ——*Constructor*
cons: $\alpha \times$ List $\alpha \longrightarrow$ List α; ——*Constructor*
first: List $\alpha \longrightarrow \alpha$;
rest: List $\alpha \longrightarrow$ List α;
$.\oplus.$: List $\alpha \times$ List $\alpha \longrightarrow$ List α **prio 6:left**;
cons, first, rest, $.\oplus.$ **strict**; cons, rest, $.\oplus.$ **total**;
List α **freely generated by** [], cons;
axioms $\forall$ a : α, l, m : List α **in**
first [] = $\perp$;
first(cons(a,l)) = a;
rest [] = [];
rest(cons(a,l)) = l;
[]$\oplus$l = l;
cons(a,m)$\oplus$l = cons(a,m$\oplus$l);
endaxioms; }

[5] Therefore sometimes also called ML polymorphism.
[6] In the following we will not distinguish between the syntactical notion 'sort' and the semantical notion 'domain'.

Of course the first difference to the parameterized version is the lack of a parameter specification. The second difference is the definition of a sort constructor List. The sort variable α is used as a formal parameter, indicating the unarity of the constructor. Next, all functions in the signature are defined polymorphically. With the help of the sort variable α we abstract from a concrete element sort. The sort variable α takes the role of the dummy sort Elem. The use of the sort constructor enables us to relate the element sort with the list sort. Finally, in the axioms part we also use a sort variable instead of the dummy sort. The axioms are valid for all instantiations of the sort variable. By using sort variables in the axioms part a uniform behaviour of the polymorphic functions can be achieved.

As we already pointed out, semantically a parameterized specification can be seen as a set of functors mapping algebras to algebras. In a polymorphic specification the task of a functor is shared by a sort constructor and the corresponding polymorphic functions. The sort constructor maps algebras into sorts and the polymorphic functions map algebras into (functional) values[7]. Hence, a polymorphic specification already provides a restricted form of parameterization.

As soon as the specification LIST_{poly} is combined with another specification module, the constructor List can be used to build complex sort expressions. The polymorphic functions can be applied to arbitrary operands matching the argument sort. This can be achieved without instantiating a parameterized specification or renaming a function symbol. Instantiation is implicitly done by the sort inference system. Because of the sort constructor even the sort of interest has not to be renamed. This makes polymorphism a very user friendly parameterization mechanism.

Further, it is important for the design of proof support systems that the specification LIST_{poly} is needed only once in such a system. In particular, it is quite easy to prove theorems "generically" in the polymorphic specification.

5 Parameterization with Sort Classes

In the last section we saw that parametric polymorphism together with user defined sort constructors in many cases gives an elegant alternative to the classical parameterization mechanism. However, although very simple and elegant, parametric polymorphism alone is rarely sufficient in practice. Normally we do not want to abstract from all possible sorts, but only from those which satisfy some requirements.

Suppose we want to specify a predicate $.\in.$, testing whether an element is contained in a list. Unfortunately, $.\in.$ is not monotonic on non-flat sorts. However, because we want to implement this function later[8], we have to restrict the possible domains of $.\in.$ to flat sorts. The same problem arises in the functional programming language ML. Because the equality of values is not computable on the non-flat function space, the polymorphic equality is restricted to flat domains by tagging the sort variables[9].

Sort classes, introduced by Wadler and Blott [19], generalize this tagging mechanism by shifting from an unsorted to a sorted parametric polymorphism. This concept, firstly implemented in the functional language Haskell is also available in SPECTRUM.

We now show how to model constrained parameterized specifications with the help of sort classes. For this purpose we extend the list specification by the predicate $.\in.$. Using classical parameterization mechanism this can be achieved in the following way:

$\mathsf{Equality}_{param}$ = {
  sort Elem;
  $.==.$: Elem × Elem → Bool;
  $.==.$ **strict total**;
  axioms ∀ a, b : Elem **in**
    (a == b) ⇔ (a = b);
  endaxioms; }

LIST'_{param} =
  param X = $\mathsf{Equality}_{param}$;
  body {
    enriches LIST_{param}(X);
    $.\in.$: Elem × List → Bool;

[7] In the polymorphic setting each algebra must be uniquely identified by its carrier of interest.
[8] It is well-known (see [8]) that non-monotonic functions are not implementable.
[9] Of course this tag propagates to functions based on the equality.

```
    .∈. strict total;
    axioms ∀a, x : Elem, l : List in
        ¬(a ∈ []);
        a ∈ cons(x,l) ⇔ (a == x) ∨ a ∈ l;
    endaxioms; }
```

Equality$_{param}$ is the requirement theory. It selects those actual parameters which contain a boolean function .==. coinciding with the strong built-in equality .=. on the defined values of sort Elem. This function is usually called *weak equality*. The requirement theory can be modelled by a sort class as follows:

```
Equality_poly = {
    class EQ;
    .==. : α :: EQ ⇒ α × α → Bool;
    .==. strict total;
    axioms α :: EQ ⇒ ∀ a, b : α in
        (a == b) ⇔ (a = b);
    endaxioms; }
```

The specification defines the class EQ of sorts containing a weak equality. Technically, this is obtained by introducing a new class EQ and by restricting the equality function .==. by the premise $\alpha :: EQ$. This premise states that in each application of .==. the sort variable can only be replaced by a sort expression of class EQ. The same premise can be found in the axioms part. Hence, the axioms are valid for all sorts of class EQ. The sort of the object variables a and b must be of class EQ, because otherwise the application of the equality function .==. would result in a sort error.

Based on this specification the specification LIST$'_{poly}$ can be written without parameter specification.

```
LIST'_poly = { enriches LIST_poly + Equality_poly;
    .∈. : α :: EQ ⇒ α × List α → Bool;
    .∈. strict total;
    axioms α :: EQ ⇒ ∀a, x : α, l : List α in
        ¬(a ∈ []);
        a ∈ cons(x,l) ⇔ (a == x) ∨ a ∈ l;
    endaxioms; }
```

The sort class EQ is used to restrict the range of the sort variable α. The predicate .∈. is only applicable on those sorts where the weak equality .==. is available.

Now we have defined functions that work on all sorts of class EQ, but we have not defined any sort to be of class EQ. Which sorts actually belong to the sort class EQ is explicitly controlled by the specifier. Consequently, the specifier has control over the possible instances of the polymorphic list operations.

Assuming a specification NAT which contains a sort Nat, the following example shows how Nat is defined to be of class EQ.

```
ListUser = { enriches LIST'_poly + NAT ;
    Nat :: EQ;
    ... }
```

In this specification both .==. and .∈. can be applied to elements of sort Nat. Declaring Nat to be of class EQ yields a consistent specification because Nat is a flat sort. On non-flat sorts the declaration would lead to an inconsistent specification. The generation of proof obligations is not part of the language but part of the methodology.

When modeling parameterization with sort classes, the classical functor is replaced by three constructs. The sort constructor and the polymorphic functions play the same role as explained in section 4. The only difference is the sort restriction in the polymorphic functions .==. and .∈.. The third construct needed is the explicit declaration that a sort belongs to a particular class, as shown above. In fact this declaration replaces the actual application of a parameterized specification, because only this declaration allows the restricted functions to be applied to elements of this sort. There is, however, a difference between these two approaches. If the parameterized specification LIST$'_{param}$ is applied to some specification, this specification must already provide an equality predicate .==. satisfying the interface axioms. In our approach, the explicit class declaration

defines the predicate .==. to work on the elements of the declared sort. In a further example we will go into this difference again.

Up to now we have only defined basic sorts to be of a particular class. Similar to Haskell, SPECTRUM also allows to provide non-basic sort constructors with a class information. Sort constructors can even be overloaded with class information, i.e. it is allowed to define more than one class information for a particular sort constructor. This feature enables us to express the fact, that a constructed sort belongs to some class, if the parameter of the sort constructor belongs to a particular class.

Let us extend the specification LIST'_{poly} by the following declaration:

List :: (EQ)EQ;

This means, that if the sort constructor List is applied to a sort of class EQ, the result is also of class EQ. It defines the argument and result class of the sort constructor List. If the sort Nat is of class EQ, the function .==. can now be applied to elements of the sort List Nat, List List Nat, and so on. Thus, with the help of sort constructors together with a class information it can be achieved that one explicit class declaration of a sort entails infinitely many class declarations of constructed sorts. This propagation of class memberships is an important advantage in contrast to classical parameterization. It can not be achieved with parameterized specifications. In the example each nested list would have to be instantiated and renamed explicitly.

Partial Order on Sort Classes

Suppose we want to enrich the specification List'_{poly} by a minimum function computing the minimal element of a list. To specify such a function we need the assumption that there exists a total order .≤. on the element sort. The specification of a total order, however, is based on the equality .==.. In a parametric specification this would lead to a hierarchical interface specification.

The sort class system of SPECTRUM offers a mechanism to deal with hierarchical theories, i.e. refinement and enrichment of theories. The user can define a partial order on sort classes, namely that a sort class is a subclass of another sort class. This leads to an order-sorted (order-classified to be precise) polymorphism concept. Semantically, it means that if a sort belongs to a class, it also belongs to all superclasses of this class. This implies that if a function is restricted to the sorts of a particular class, the function is also available on the sorts of all subclasses. The following example demonstrates this feature by enriching the specification $\mathsf{Equality}_{poly}$.

Ordering = { **enriches** $\mathsf{Equality}_{poly}$;
  class TO **subclass of** EQ;
  .≤.: α :: TO ⇒ $\alpha \times \alpha \rightarrow$ Bool;
  .≤. **strict total**;
  axioms α :: TO ⇒ ∀ a, b, c: α **in**
    a ≤ a; —*reflexivity*
    a ≤ b ∧ b ≤ c ⇒ a ≤ c; —*transitivity*
    a ≤ b ∧ b ≤ a ⇒ a == b; —*antisymmetry*
    a ≤ b ∨ b ≤ a; —*totality*
  endaxioms; }

Based on this specification we can now enrich LIST'_{poly} by a minimum function as follows:

LIST''_{poly} = {**enriches** LIST'_{poly} + Ordering;
  min : α :: TO ⇒ List $\alpha \rightarrow \alpha$;
  min **strict**;
  axioms α :: TO ⇒ ∀ e : α, s : List α **in**
    s ≠ [] ⇒ min(s) ∈ s ∧ (e ∈ s ⇒ min(s) ≤ e);
  endaxioms; }

To use the minimum function on lists over natural numbers we must declare the sort Nat to be of class TO. Because TO is a subclass of EQ this declaration automatically entails that Nat is of class EQ.

As already mentioned there is a difference between the application of a parameterized specification and the declaration of a class membership. If we declare the sort Nat to be of class TO no order predicate .≤. must be defined on Nat. The declaration only defines the requirements for such a predicate on the natural numbers

but does not fix a particular order. This is also in contrast to the functional programming language Haskell where an instantiation of a class must always provide an executable body for each function of the class. But this would be to restrictive for a specification language and is only necessary if the specification should be executable.

6 Classical Parameterization

Parametric polymorphism combined with sort classes i.e. "order–sorted parametric polymorphism" is powerful enough to model most of the classical examples for parameterized specifications. In all these examples sort abstraction is done only over one sort or in other words the requirement theories have only one sort of interest.

However, there are cases in which we want to abstract not only over one sort but over n–tuples of sorts. A typical example is the theory of vector spaces. Here we abstract from tuples of sorts of the form (α, β) :: scalars × vectors which satisfy the vector space properties. There are also cases in which we want to abstract from n-ary sort constructors (not only basic sorts). A typical example is a container structure which has lists and vectors as instances. For such cases, in order to keep the sort class mechanism simple, we provide *specification abstraction*[10]. A parameterized specification of the container theory is the following one:

```
CONTAINER_param =
    param X = {sort Elem};
    body { enriches X + NAT;
        sort Container;
        [] : Container;
        cons : Elem × Container → Container;
        nth : Container × Nat → Elem;
        len : Container → Nat;
        --Container axioms }
```

$\mathsf{CONTAINER}_{param}$ can subsequently be applied to any specification if it is also indicated which sort in this specification corresponds to Elem. In general one has to give a mapping (a renaming list) from the signature of the formal parameter to the signature of the actual parameter if this is not an inclusion. As an example we can instantiate $\mathsf{CONTAINER}_{param}$ with NAT as follows:

```
NatContainer = CONTAINER_param(NAT via [Elem to Nat])
```

The result of the above instantiation is the body of $\mathsf{CONTAINER}_{param}$ with NAT substituted for each occurrence of X and Nat substituted for each occurrence of Elem.

Using the parameterized container theory we can define a variation of the well known map function. This variation applies an argument function to all elements of a container and yields another container with duplicates eliminated. To eliminate the duplicates we need a boolean function to test for equality.

```
MAP_param =
    param
        X = {sort X_Elem};
        Y = rename Equality_param by [ Elem to Y_Elem];
        Z = CONTAINER_param;
    body {  enriches X + Y
            + rename Z(X via [ Elem to X_Elem ]) by [ Container to X_Container,
                                                      [] to X_[],
                                                      cons to X_cons,
                                                      nth to X_nth, len to X_len ]
            + rename Z(Y via [ Elem to Y_Elem ]) by [ Container to Y_Container,
                                                      [] to Y_[],
                                                      cons to Y_cons,
                                                      nth to Y_nth, len to Y_len ];
            map : (X_Elem → Y_Elem) × X_Container → Y_Container;
            --Map axioms }
```

[10] According to [15] this mechanism allows to construct parameterized specifications in contrast to specifications of parametric algebras.

We can subsequently instantiate this specification with actual specifications for X, Y and Z. It is important to note that the instances of parameterized specifications have to be legal, non-parameterized specification expressions. Furthermore, we do not allow partial instantiations. Suppose we are given the parameterized specifications LIST and VECT of lists and vectors both having at least the $\mathsf{CONTAINER}_{param}$ functions and satisfying its axioms. Further, we assume that NAT provides at least a weak equality .==. on Nat. Then we can build for example the following instances:

ListMap = MAP_{param}(NAT **via** [X_Elem **to** Nat], NAT **via** [Y_Elem **to** Nat], LIST **via** [Container **to** List])

VectMap = MAP_{param}(NAT **via** [X_Elem **to** Nat], NAT **via** [Y_Elem **to** Nat], VECT **via** [Container **to** Vector])

Because NAT satisfies the trivial theory as well as the equality theory and LIST and VECT satisfy the container theory the instances have the properties we expect them to have. As with sort classes, proof obligations are treated at the methodology level.

As probably noted, classical parameterization alone involves a lot of complications. First, we have to allow parameterized specifications as parameters and then we have to take care of name clashes. Furthermore, classical parameterization alone does not take advantage of the new facilities of SPECTRUM by requiring the specifications LIST and VECT to be parameterized in a classical and not in a polymorphic way.

However, when combined with sort classes, parameterization becomes a simple and very powerful mechanism. For example, we can write a polymorphic version of containers as follows:

$\mathsf{CONTAINER}_{poly}$ = { **enriches** NAT;
sort Container α;
[]: Container α;
cons : $\alpha \times$ Container $\alpha \rightarrow$ Container α;
nth : Container $\alpha \times$ Nat $\rightarrow \alpha$;
len : Container $\alpha \rightarrow$ Nat;
--Container axioms }

We can now use $\mathsf{CONTAINER}_{poly}$ to define the parameterized specification:

MAP_{poly} =
param X = $\mathsf{CONTAINER}_{poly}$;
body { **enriches** X + $\mathsf{Equality}_{poly}$;
map : β::EQ $\Rightarrow$ ($\alpha \rightarrow \beta$) $\times$ Container $\alpha \rightarrow$ Container β;
--Map axioms }

Subsequently, MAP_{poly} can be instantiated with the polymorphic versions of LIST and VECTOR easily by writing:

ListMap = MAP_{poly}(LIST **via** [Container **to** List])

VectMap = MAP_{poly}(VECT **via** [Container **to** Vector])

Obviously, the second version is easier to write and to understand as the first one, because it takes advantage of the new facilities of our language SPECTRUM. Moreover, no new complications are introduced when combining both approaches.

7 Conclusions and Future Work

We presented a specification formalism combining the elegance of *implicit* with the power of *explicit* parameterization. In contrast to e.g. Extended ML which only combines parametric polymorphism with explicit parameterization our language combines type-classed parametric polymorphism with explicit parameterization. Moreover, we are not aware of any other algebraic specification language which includes type classes. The only logical system we know to include type classes is the Isabelle [14] theorem prover. The experience we gained by some case studies encouraged us to consider the combination of both parameterization mechanisms to be very useful. However, further case studies have to be done.

A very important application of type classes which we did not treat in this paper is overloading. Type classes not only solve the overloading problem from ML but also allow user controlled overloading. This is essential in every algebraic formalism.

An interesting question is how far the implicit mechanism can be extended. For example [3] presents a class concept allowing also to abstract from sort constructors and [12, 13] try to cope with classes abstracting over tuples of sorts.

Acknowledgement

For comments on draft versions and stimulating discussions we would like to thank our colleagues C. Facchi, R. Hettler, H. Hussmann, F. Regensburger, B. Rumpe and O. Slotosch. Special thanks go to Tobias Nipkow whose work inspired our treatment of type classes, to Martin Wirsing who influenced the classical parameterisation mechanism and to Manfred Broy whose role was decisive in the design of the SPECTRUM language.

References

[1] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The requirement and design specification language SPECTRUM: An informal introduction – Version 1.0, Part I,II. Technical Report TUM-I9311, TUM-I9312, Technische Universität München, 1993.

[2] R. Burstall and J. Goguen. The semantics of CLEAR, a specification language. In *Proceedings of Advance Course on Software Specification*, volume 86 of *LNCS*, pages 292–332, 1980.

[3] K. Chen, P. Hudak, and M. Odersky. Parametric Type Classes. In *ACM Conference on LISP and Funtional Programming*, 1992.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[5] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constrains.* Springer, 1990.

[6] M.-C. Gaudel. Towards Structured Algebraic Specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.

[7] R. Grosu and F. Regensburger. The Logical Framework of SPECTRUM. Draft, 1993.

[8] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[9] J. Guttag, J. Horning, and J. Wing. Larch in Five Easy Pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.

[10] R. Harper, D. MacQueen, and R. Milner. Standard ML. *Report ECS-LFCS-86-2, Univ. Edinburgh*, 1986.

[11] P. Hudak, S. P. Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.

[12] M. Jones. Type Inference For Qualified Types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, 1991.

[13] S. Kaes. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *Proc. ACM Conf. Lisp and Functional Programming*, 1992.

[14] T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[15] D. Sannella, S. Sokolowski, and A. Tarlecki. Toward formal development of programs from algebraic specifications: parameterization revisited. Technical Report 6/90, Universität Bremen, 1990.

[16] D. Sannella and A. Tarlecki. Program specification and development in Standard ML. In *Proc. 12 ACM Symp. on Principles of Programming Languages*, pages 67–77, 1985.

[17] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, Sept. 1983.

[18] C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[19] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[20] C. Wadsworth, M. Gordon, and R. Milner. Edinburgh lcf: A mechanised logic of computation. In *LNCS 78.* Springer, 1979.