UNIFYING INITIAL AND LOOSE SEMANTICS OF PARAMETERIZED SPECIFICATIONS IN AN ARBITRARY INSTITUTION

HUBERT BAUMEISTER

ABSTRACT. In this paper we are going to present a theory of parameterized abstract datatypes as the model-theoretic level of parameterized specifications. We will show that parameterized abstract datatypes allow us to model the main approaches to the semantics of parameterized specifications, the loose approach (e.g. [GB90]) and the free functor semantics (e.g. [TWW82, EM85]), using the same formalism. As a consequence we obtain that, when using data constraints [Rei86, GB90, OSC89, EM90] in a specification language, this language is able to cope with both the loose and the free functor semantics at the same time. To be independent of a specific logic this theory is developed in the context of an arbitrary institution.

1. INTRODUCTION

Specifications denote on the presentation level a set of formulas (called axioms) over some signature Σ and on the model-theoretic level *abstract datatypes*, which are classes of Σ-structures. *Parameterized specifications* are on the presentation level morphisms in the category **Spec** of specifications from a formal parameter specification to a body specification, which is an enrichment of the formal parameter. The application of a parameterized specification to a specification is then modeled by a pushout in **Spec**. On the model-theoretic level two different approaches have been introduced in the literature.

In the first one parameterized specifications are viewed as functions or classes of functions from parameter structures to body structures. These functions are often called *datatype constructors*. The most popular representative of this view is the free functor semantics of [TWW82, EM85], which associates to each parameterized specification the class of all free functors from the models of the formal parameter specification to those of the body. This semantics is e.g. used in the definition of the specification language ACT ONE [Cla87]. A more general approach allowing arbitrary datatype constructors is presented by Lipeck [Lip82, EGL89]. Another example is the semantics of EML-functors which is a set of functions from parameter structures to body structures [ST88].

In the second approach parameterized specifications are viewed as functions from classes of structures to classes of structures. Representatives are e.g. the specification languages ASL [SW83] and Clear [BG80]. Other examples are the concept of loose extensions [OSC89], behavioural canons [Rei86] and cells [Sch86].

We are going to present a model-theoretic view of parameterized specifications based on parameterized abstract datatypes, which represent functions from abstract datatypes to abstract datatypes. A parameterized abstract datatype is a morphism in the category **Adt** of abstract datatypes from a formal parameter abstract datatype to a body abstract datatype, where the structures of the body are strongly persistent extensions of some

parameter structures. The application of a parameterized abstract datatype to an actual parameter is then modeled by a pushout in **Adt**.

If the body of a parameterized abstract datatype B contains for every parameter structure a unique strongly persistent extension then B represents a strongly persistent functor from parameter structures to body structures. Since abstract datatypes are arbitrary classes of structures we are able to find for each strongly peristent functor on structures a parameterized abstract datatype representing it.

If the body of B contains for every parameter structure at least one strongly persistent extension then B represents a function from parameter structures to classes of body structures or equivalently a class of strongly persistent functions from parameter structures to body structures. Although it is not possible to represent every class of functions that way (see [SST90] for details) we are able to show that the class of free functors and thus the free functor semantics of a parameterized specification B can be represented by a parameterized abstract datatype. In other words, we will present the conditions for a parameterized abstract datatype to map the class of initial models of the parameter specification $spec_A$ of B to the class of initial models of the result specification $B(spec_A)$. A consequence of these conditions is that specification languages in the style of Clear are able to model both the loose approach and the free functor approach to the semantics of parameterized specifications.

Since it is possible to model both functions and classes of functions with parameterized abstract datatypes we propose to view parameterized abstract datatypes as a basic tool for studying the model–theoretic side of parameterized specifications. An application of this theory can be found in [Bau90] where it is used to study parameterized specifications in the presence of behavioural abstraction.

Our treatment is a slight generalization of the approach by Burstall and Goguen [BG80, GB90]. They view the semantics of a specification *spec* as a *theory* which can in turn be seen as the class of all structures satisfying the axioms of *spec*[1] and the semantics of a parameterized specification as a theory morphism. In our approach we allow arbitrary classes of structures not only those that can be represented by a set of formulas. An approach similar to ours can be found in [Lip82, EGL89] with the exception that Lipeck requires a functor from the models of the parameter to those of the body in addition to an abstract datatype morphism.

The theory of parameterized abstract datatypes will be developed in the framework of an arbitrary institution [GB90] and thus be independent of a specific logic. By doing this we will summarize the results on the existence of amalgamated sums and its implication in the amalgamation and extension lemma found in the literature (see [EM85, Lip82, WE85, EPO89, Poi89] among others).

To summarize, the approach of the present paper is in spirit similar to the ones mentioned before. Our technical improvement is that we only employ a minimal number of primitive requirements for our treatment of parameterized specifications.

[1]In [GB90] a theory is really seen as a set of formulas closed under logical consequence. See also footnote 8.

2. Specifications and Abstract Datatypes

2.1. Institutions. The notion of institution has been introduced by Goguen and Burstall [GB90] to define the semantics of the specification language Clear [BG80] independent of an underlying logic. An institution is parameterized by the notion of signatures, of interpretations of signatures, of formulas over a signature and by the notion of validity of a formula in an interpretation of a signature.

Definition 2.1 (Institution). An *institution* $\mathcal{I} = (\mathbf{Sign}, Mod, Sen, \models)$ consists of

- a category of *signatures* **Sign**;
- a functor Mod: **Sign** $\rightarrow$ $\mathbf{Cat}^{op}$ assigning to each signature Σ the category of all possible interpretations of the "symbols" in Σ and to each signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ a forgetful functor $Mod(\sigma): Mod(\Sigma') \rightarrow Mod(\Sigma)$[2]. The objects in $Mod(\Sigma)$ will be called *structures*;
- a functor Sen: **Sign** $\rightarrow$ **Set**, assigning to each signature Σ the set of *formulas* that can be build from symbols of Σ and to each signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ a translation $Sen(\sigma)$[3] of Σ–formulas to Σ'–formulas;
- and a family of *satisfaction relations* $\models = (\models_\Sigma \subseteq |Mod(\Sigma)| \times |Sen(\Sigma)|)_{\Sigma \in |\mathbf{Sign}|}$, indicating if a Σ–formula f is valid in a structure m ($m \models_\Sigma f$ or for short $m \models f$,)

where the *satisfaction condition* holds:
For all signature morphism $\sigma: \Sigma \rightarrow \Sigma'$, formulas $f \in Sen(\Sigma)$ and structures $m' \in Mod(\Sigma)$

$$m'|_\sigma \models f \text{ iff } m' \models \sigma(f).$$

In [GB90] some examples of institutions are given, among them the institution of equational logic, conditional equational logic, first order logic with and without equality and horn clause logic.

We extend the notion of satisfaction and logical consequence to sets of formulas. Let F and F' be sets of Σ–formulas, then we write

$$\begin{aligned} m \models F &\text{ iff } \forall f \in F : m \models f \text{ and} \\ F \models F' &\text{ iff } \forall m \in Mod(\Sigma) : m \models F \Rightarrow m \models F'. \end{aligned}$$

$F \models F'$ means that F' is a subset of all logical consequences of F.

2.2. Specifications. A many–sorted signature and a set of equations form a specification in equational logic. Extended to an arbitrary institution $\mathcal{I}$ a *specification* $spec = \langle \Sigma, F \rangle$ consists of a signature $\Sigma \in \mathbf{Sign}$ and a set of Σ–formulas $F \subseteq Sen(\Sigma)$, called the *axioms* of *spec*.

A *specification morphism* $\sigma: \langle \Sigma, F \rangle \rightarrow \langle \Sigma', F' \rangle$ is a signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ satisfying $F' \models \sigma(F)$, e.g. all formulas in $\sigma(F)$ follow from the formulas in F'. Specifications together with specification morphisms form the *category* **Spec** *of specifications*. This definition of **Spec** is an extension of the definition of CATSPEC in [EM85] to an arbitrary institution as can be found, for instance, in [EM90].

Two functors $Sign$ and Mod are defined on **Spec**, returning the signature and the category of its models, respectively. The functor $Sign$ from **Spec** to **Sign** takes each

[2]The application of $Mod(\sigma)$ to a structure m will be written as $m|_\sigma$.
[3]The application of $Sen(\sigma)$ to a formula f will be abbreviated by $\sigma(f)$.

specification $spec = \langle \Sigma, F \rangle$ to its signature Σ and each **Spec**–morphism to its corresponding signature morphism. The functor Mod: **Adt** $\rightarrow$ **Cat**^{op} maps a specification $spec = \langle \Sigma, F \rangle$ to the full subcategory of $Mod(\Sigma)$, which has as objects all structures satisfying F[4]

$$|Mod(spec)| := \{m \in Mod(\Sigma) \mid m \models F\}.$$

The objects of $Mod(spec)$ are called models of *spec*. For each **Spec**–morphism σ from *spec* to *spec'* the functor Mod yields the forgetful functor $Mod(Sign(\sigma))$ restricted on $Mod(spec')$.

We define operations on specifications to build new specifications from existing ones. The operations used in this paper are the union and translation of specifications. The *union* of two specifications $spec_1 = \langle \Sigma, F_1 \rangle$ and $spec_2 = \langle \Sigma, F_2 \rangle$ over the same signature is given by the union of their axioms:

$$spec_1 \cup spec_2 := \langle \Sigma, F_1 \cup F_2 \rangle.$$

If $spec = \langle \Sigma, F \rangle$ is a specification and $\sigma\colon \Sigma \rightarrow \Sigma'$ is a signature morphism, then the *translation* of *spec* via σ is defined by

$$\sigma(spec) := \langle \Sigma', \sigma(F) \rangle.$$

With these two operations the application of a parameterized specification to actual parameters will be formulated. More complex specification building operations can be defined with the help of this instantiation mechanism.

2.3. Abstract Datatypes. Abstract datatypes are intended to form the semantic meaning of a specification. A datatype can be seen as many–sorted signature Σ together with a Σ–algebra A, where Σ contains the sorts and operations defined on the datatype and the algebra A is an interpretation of the symbols in Σ [GTW76]. An abstract datatype abstracts from a concrete interpretation (algebra) of the symbols in Σ to a class of interpretations (class of algebras) having the same "properties." E.g., one might require that all algebras in an abstract datatype should be isomorphic to a standard model. This is what [GTW76] considers "abstract" to mean in "abstract datatype". Another concept is to require that all algebras behave like a standard model [SW83, ST84, Sch86] or to take all algebras satisfying a set of formulas (e.g. [BG80]).

In the context of an arbitrary institution $\mathcal{I}$ an *abstract datatype* $adt = \langle \Sigma, \mathbf{M} \rangle$ consists of a signature $\Sigma \in$ **Sign** together with a subcategory **M** of $Mod(\Sigma)$. The objects of **M** are called models of *adt*.

We do not require **M** to be a full subcategory of $Mod(\Sigma)$ as done in [SW83, ST85, EGL89]. One reason is that these requirements are not necessary for the theory developed in this paper. The other reason is that we will always have full subcategories anyway. The loose and initial semantics of specification will in arbitrary context yield full subcategories and the operations on abstract datatypes defined in this paper will preserve this property.

An *abstract datatype morphism* σ from $\langle \Sigma, \mathbf{M} \rangle$ to $\langle \Sigma', \mathbf{M}' \rangle$ is a signature morphism σ from Σ to Σ' satisfying the condition that the objects and arrows of $\mathbf{M}'|_\sigma$ are contained in the objects and arrows of **M**.[5] Abstract datatypes together with abstract datatype morphisms form the *category* **Adt** *of abstract datatypes* [SW83, EGL89, ST85].

[4]The class of objects of a category **C** are denoted by $|\mathbf{C}|$ and the class of arrows by $arr(\mathbf{C})$.

[5]$\mathbf{M}'|_\sigma$ is not required to be a subcategory of **M**, because $\mathbf{M}'|_\sigma$ may not be a category.

As with specifications, the functors *Sign* and *Mod* are defined on abstract datatypes, yielding its signature and its models. The functor $Sign$: **Adt** $\rightarrow$ **Sign** takes an abstract datatype to its signature and each **Adt**–morphism to its corresponding signature morphism. The functor Mod: **Adt** $\rightarrow$ **Cat**^{op} returns for each abstract datatype $adt = \langle \Sigma, \mathbf{M} \rangle$ the category $\mathbf{M}$ and for each **Adt**–morphism σ the forgetful functor $Mod(Sign(\sigma))$ restricted to $Mod(adt')$.

Operations on abstract datatypes are used to define new abstract datatypes from existing ones. In the literature many operations have been introduced for various purposes. For the purpose of this paper we need only union and translation (see [ST85, SW83, Lip82, EGL89] for other operations).

The *union* of two abstract datatypes adt_1 and adt_2 over the same signature Σ is given by the intersection of their model categories[6]

$$\begin{aligned} Sign(adt_1 \cup adt_2) &:= \Sigma \\ Mod(adt_1 \cup adt_2) &:= Mod(adt_1) \cap Mod(adt_2). \end{aligned}$$

The *translation* of an abstract datatype adt of signature Σ by a signature morphism σ: $\Sigma \rightarrow \Sigma'$ is defined by

$$\begin{aligned} Sign(\sigma(adt)) &:= \Sigma' \\ |Mod(\sigma(adt))| &:= \{m' \in |Mod(\Sigma')| \mid m'|_\sigma \in |Mod(adt)|\} \\ arr(Mod(\sigma(adt))) &:= \{h' \in arr(Mod(\Sigma')) \mid h'|_\sigma \in arr(Mod(adt))\} \end{aligned}$$

Evidently, $Mod(\sigma(adt))$ is again a category.

2.4. The Semantics of Specifications. The semantics of a specification is generally seen as some function[7] *sem* taking specifications to abstract datatypes over the same signature

$$sem: |\mathbf{Spec}| \rightarrow |\mathbf{Adt}\text{—}, \text{ with } Sign(spec) = Sign(sem(spec)).$$

It is not required that all models of $sem(spec)$ satisfy the axioms of *spec*. This allows for instance behavioural semantics, where the models only have to be behaviouraly equivalent to models of *spec* [Sch86].

Various concrete semantics of this kind have been defined in the literature. In this paper, we are concentrating on the loose [BG80, SW83] and initial semantics [GTW76] of specifications.

Definition 2.2 (Loose Semantics). The *loose semantics* of a specification *spec* is the abstract datatype

$$loose(spec) := \langle Sign(spec), Mod(spec) \rangle.$$

Definition 2.3 (Initial Semantics). The *initial semantics* of a specification *spec* is the abstract datatype

$$init(spec) := \langle Sign(spec), \mathbf{M} \rangle,$$

where $\mathbf{M}$ is the full subcategory of $Mod(\Sigma)$ consisting of all initial objects of $Mod(spec)$.

[6] $\cap$ denotes the intersection of two categories by intersecting their classes of objects and of arrows. The intersection of two categories again is a category.

[7] We do not require *sem* to be a functor as in the case of initial semantics a nonpersistent **Spec**–morphism has no corresponding **Adt**-morphism.

In an arbitrary institution $Mod(init(spec))$ may be empty for some specifications. This is not the case in many interesting institutions such as the institution of equational logic, where any $Mod(spec)$ has an initial object (see e.g. [GTW76]).

One may ask, if the operations defined on abstract datatypes and those defined on specifications are compatible with a given semantics *sem*, in other words, if

$$\begin{aligned} sem(spec_1 \cup spec_2) &= sem(spec_1) \cup sem(spec_2) \text{ and} \\ sem(\sigma(spec)) &= \sigma(sem(spec)) \end{aligned}$$

hold true.

Theorem 2.4. *It holds that*

$$\begin{aligned} loose(spec_1 \cup spec_2) &= loose(spec_1) \cup loose(spec_2) \text{ and} \\ loose(\sigma(spec)) &= \sigma(loose(spec)), \end{aligned}$$

for specifications $spec_1$, $spec_2$ *and* $spec$, *with* $Sign(spec_1) = Sign(spec_2)$, *and a signature morphism* σ *from* $Sign(spec)$ *to a signature* Σ'.

The last theorem does not hold in the case of initial semantics. Take for example the specification BOOL with sort *bool* and constants *true* and *false* of sort *bool*. The initial algebra of BOOL has exactly two elements of sort *bool*. Now take BOOLF to be BOOL together with the equation $true = false$. The initial algebra of BOOLF has only one element of sort *bool*. The union of BOOLF and BOOL yields again BOOLF, but the union of $init$(BOOL) and $init$(BOOLF) has no models.

As an example for the incompatibility with translation, take the specification BOOLF2, where BOOLF2 is BOOL extended by a constant c of sort *bool*, and σ the inclusion of $Sign$(BOOL) in $Sign$(BOOLF2). Then, the initial algebra A of $Mod(\sigma(\text{BOOL})) = Mod(\text{BOOLF2})$ has 3 elements of sort *bool*, but each algebra $B \in Mod(\sigma(init(\text{BOOL})))$ has only 2 elements of sort *bool*, because $B|_\sigma$ has to be an initial object of Mod(BOOL).

3. PARAMETERIZATION

3.1. Parameterized Specifications. A parameterized specification defines a function taking an admissible parameter specification to a result specification. The result specification is the parameter extended by the definitions of the body of the parameterized specification. This section contains the material about parameterized specifications in [EM85] and extends it to an arbitrary institution as, for instance, in [EM90].

A parameterized specification consists of a formal parameter specification and a body specification, which is an enrichment of the (possibly renamed) formal parameter.

Definition 3.1 (Parameterized Specification).
A *parameterized specification* $B(X : spec_F)_\beta : spec_B$ is a **Spec**–morphism

$$\beta\colon spec_F \to spec_B,$$

where $spec_F$ is called *the formal parameter* and $spec_B$ *the body* of B.

Each specification which satisfies the requirements of the formal parameter, i.e.,provides all symbols (sorts and operations) possibly renamed by a parameter passing morphism α, and all formulas of the formal parameter, is an admissible parameter for the parameterized specification.

Definition 3.2 (Admissible Parameter). A specification $spec_A$ is an *admissible parameter* for a parameterized specification $B(X : spec_F)_\beta : spec_B$ wrt. a *parameter passing morphism* α: $Sign(spec_F) \to Sign(spec_A)$, if α is also a **Spec**–morphism from $spec_F$ to $spec_A$.

In the following we will say for short $spec_A$ is an admissible parameter for B and will assume that the corresponding parameter passing morphism is called α.

The application of a parameterized specification B to an admissible parameter is the extension of the parameter by the definitions given in the body of B. In equational logic this corresponds to the disjoint union of sorts, operations and equations of the parameter and the body, where symbols contained in the formal parameter are not duplicated. Therefore this process is modeled on the level of specifications as the construction of a pushout object.

Definition 3.3 (Application).
The result $B(spec_A)$ of the *application* of $B(X : spec_F)_\beta : spec_B$ to an admissible parameter $spec_A$ is the pushout object $spec_{po}$ of the following diagram

$$\begin{array}{ccc} spec_F & \xrightarrow{\beta} & spec_B \\ {\scriptstyle\alpha}\downarrow & \text{po.} & \downarrow{\scriptstyle\alpha'} \\ spec_A & \xrightarrow[\beta']{} & spec_{po} \end{array}$$

The existence of pushouts in **Spec** is guaranteed, whenever the corresponding signatures have a pushout in **Sign**. This is due to the property of the functor *Sign* to create pushouts. In the case where **Sign** is finitely cocomplete (e.g. has all finite colimits or equivalently has an initial object and all pushouts [Mac88]) **Spec**, too, is finitely cocomplete. Then the initial object in **Spec** is $\langle \Sigma_0, \emptyset \rangle$, where Σ_0 is the initial object in **Sign**. As shown in [EM85] the category of many sorted signatures is finitely cocomplete and therefore $B(spec_A)$ always exists in the institution of equational logic.

Theorem 3.4 (*Sign* **creates pushouts**). *A diagram* $D = spec_A \xleftarrow{\alpha} spec_F \xrightarrow{\beta} spec_B$ *has a pushout* $P = spec_A \xrightarrow{\beta'} spec_{po} \xleftarrow{\alpha'} spec_B$ *in* **Spec**, *whenever* $Sign(D)$ *has a pushout* $P_{\mathbf{Sign}}$ *in* **Sign**. *In this case* $Sign(P)$ *equals* $P_{\mathbf{Sign}}$ *and*

$$spec_{po} = \beta'(spec_A) \cup \alpha'(spec_B).$$

3.2. Parameterized Abstract Datatypes. Parameterized abstract datatypes are intended to be the semantic counterpart of parameterized specifications. Most constructions of this section will hence be the same as in the previous one, except that objects and morphisms of **Adt** are used instead of the objects and morphisms of **Spec**.

This treatment has been proposed by [ST85] and is a slight generalization of [BG80, GB90] where parameterized specifications are semantically viewed as theory morphisms. A theory can be seen as an abstract datatype *adt*, where the models are the models of a

specification *spec* over the same signature

$$Mod(adt) = Mod(spec).$$

Therefore the category **Th** of theories can be understood as a full subcategory of **Adt**.[8]

A parameterized abstract datatype defines a function mapping abstract datatypes to abstract datatypes. The models of the result are extensions of parameter models by the new symbols and their interpretation as defined by the body of the parameterized abstract datatype.

Definition 3.5 (Parameterized Abstract Datatype).
A *parameterized abstract datatype* $B(X : adt_F)_\beta : adt_B$ is an **Adt**-morphism

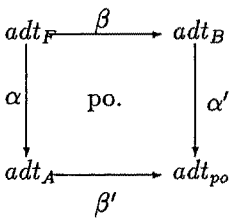
$$\beta\colon adt_F \to adt_B,$$

where adt_F is called the *formal parameter* and adt_B the *body* of B.

Definition 3.6 (Admissible Parameter). An abstract datatype adt_A is an *admissible parameter* wrt. a parameter passing morphism $\alpha\colon Sign(adt_F) \to Sign(adt_A)$ for a parameterized abstract datatype $B(X : adt_F)_\beta : adt_B$ if α is an **Adt**–morphism from adt_F to adt_A.

As in the case of specifications we will speak of admissible parameters adt_A and assume the parameter passing morphism is denoted by α.

Definition 3.7 (Application). The *application* of $B(X : adt_F)_\beta : adt_B$ to an admissible parameter adt_A ($B(adt_A)$) is defined as the pushout object adt_{po} of the following diagram.



As in the case of parameterized specifications the functor $Sign\colon$ **Adt** $\to$ **Sign** creates pushouts [Lip82, ST85, EGL89] and therefore $B(adt_A)$ always exists whenever **Sign** has all pushouts. Again we have that **Adt** is finitely cocomplete whenever **Sign** is. The initial object in **Adt** then is $\langle \Sigma_0, Mod(\Sigma_0)\rangle$. Especially for the institution of equational logic **Adt** is finitely cocomplete.

Theorem 3.8 (*Sign* **creates Pushouts**). *A diagram* $D = adt_A \xleftarrow{\alpha} adt_F \xrightarrow{\beta} adt_B$ *has a pushout* $P = adt_A \xrightarrow{\beta'} adt_{po} \xleftarrow{\alpha'} adt_B$ *in* **Adt**, *whenever* $Sign(D)$ *has a pushout* $P_{\mathbf{Sign}}$ *in* **Sign**. *In this case* $Sign(P)$ *equals* $P_{\mathbf{Sign}}$ *and*

$$adt_{po} = \beta'(adt_A) \cup \alpha'(adt_B).$$

[8]Another characterization of a theory is to consider a theory as a specification, in which the axioms are closed under logical consequence. This view is taken by [GB90] and has the disadvantage that theories are not closed under union whereas theories seen as abstract datatypes are (see theorem 2.4).

The application of a parameterized abstract datatype to another parameterized abstract datatype can be defined in the same way as it is done for parameterized specifications in [EM85]. Due to lack of space, we will omit this construction here.

3.3. Parameterized Abstract Datatypes as Functors. In [SST90] it is argued that there are two kind of parameterizations, parameterized specifications, which are functions on abstract datatypes, and specifications of parametric structures, which are functions on structures. Above we have indicated how a parameterized abstract datatype can be viewed as a function mapping actual parameter abstract datatypes to a pushout abstract datatype. This section now studies parameterized abstract datatypes from the viewpoint of functions on structures. To do this, the function B_R is defined, describing the effect of a parameterized abstract datatype B on the models of an admissible parameter adt_A.

Definition 3.9 (B_R). For a parameterized abstract datatype B and an admissible parameter adt_A the function B_R takes every model m_A of adt_A to the set of all models m of $B(adt_A)$ with $m|_{\beta'} = m_A$

$$B_R(m_A) := \{m \in Mod(B(adt_A)) \mid m|_{\beta'} = m_A\}.$$

In the same way B_R is defined for morphisms $h_A \in Mod(adt_A)$

$$B_R(h_A) := \{h \in Mod(B(adt_A)) \mid h|_{\beta'} = h_A\}.$$

The semantics of an Extended ML functor in [ST88, SST90] can be formulated now with the help of parameterized abstract datatypes. For an Extended ML functor

$$\textbf{functor } F(X : spec_{par}) : spec_{bod}$$

its semantics is defined as a function F_{sem} mapping a model m of $spec_{par}$ to a set of models of $spec_{bod}$

$$F_{sem}(m) := \{m_B \in Mod(spec_{bod}) \mid m_B|_{spec_{par}} = m\}.$$

But this is the definition of $B_R(m_A)$ for a parameterized abstract datatype

$$B(X : loose(spec_{par}))_\iota : loose(spec_{bod}),$$

where ι is the injection of $spec_{par}$ into $spec_{bod}$.

Definition 3.10 (Consistency). A parameterized abstract datatype B is *consistent for* an admissible parameter adt_A, if for every model m_A and morphism h_A of $Mod(adt_A)$ the sets $B_R(m_A)$ and $B_R(h_A)$ are not empty. B is called consistent, if B is consistent for all admissible parameters.

If a parameterized abstract datatype B is consistent for adt_A, then B can be viewed as a family of functions $\mathcal{F}$ from $Mod(adt_A)$ to $Mod(B(adt_A))$ given by

$$\mathcal{F} := (F : Mod(adt_A) \to Mod(B(adt_A)) \mid F(m_A) \in B_R(m_A)).$$

Note that, if there exists a strongly persistent functor F from the models of adt_A to the models of $B(adt_A)$ wrt. $Mod(\beta')$,[9] then B is consistent for adt_A. Take for example a model m_A of adt_A, then, because F ist strongly persistent, $F(m_A)|_{\beta'}$ equals m_A. Therefore $F(m_A)$ is an element of $B_R(m_A)$ and $B_R(m_A)$ not empty.

[9] that is, $Mod(\beta')F = Id$.

Definition 3.11 (Uniqueness). A parameterized abstract datatype B is called *unique* for an admissible parameter adt_A if for every object m_A and morphism h_A of $Mod(adt_A)$ $B_R(m_A)$ and $B_R(h_A)$ contain at most one element.

B is called unique if it is unique for every admissible parameter.

Theorem 3.12 (Parameterized Abstract Datatypes as Functors).
If a parameterized abstract datatype B is unique and consistent for an admissible parameter adt_A, then there exists a unique strongly persistent functor B_F from $Mod(adt_A)$ to $Mod(B(adt_A))$.

For every object m_A and morphism h_A of $Mod(adt_A)$ B_F is defined by the only element of $B_R(m_A)$ and $B_R(h_A)$.

3.4. Semantics of Parameterized Specifications. In the loose approach to abstract datatypes the semantics of a parameterized specification is a parameterized abstract datatype (e.g. [BG80]), while in the initial approach the semantics is the class of all left adjoint functors[10] to the forgetful functor $Mod(\beta)$ [TWW82, EM85, EGL89].

For the loose approach we have

$$ploose(B(X : spec_F)_\beta : spec_B) := B(X : loose(spec_F))_\beta : loose(spec_B).$$

It is now natural to ask for *compatibility* with the corresponding semantics of specifications. The semantics of the parameterized specification $psem(B)$ applied to the semantics of an actual parameter specification $sem(spec_A)$ should be the same as the semantics of the pushout $B(spec_A)$

$$psem(B)(sem(spec_A)) = sem(B(spec_A)).$$

The compatibility of *ploose* with *loose* is a result of the compatibility of the union and the translation of specifications with loose semantics (see theorem 2.4), because

$$\begin{aligned} ploose(B)(loose(spec_A)) &= \beta'(loose(spec_A)) \cup \alpha'(loose(spec_B)) \\ &= loose(\beta'(spec_A) \cup \alpha'(spec_B)) \\ &= loose(B(spec_A)). \end{aligned}$$

Another useful result is that for a parameterized specification B and a parameterized abstract datatype B' we have, if $Mod(adt_i)$ is a subcategory of $Mod(spec_i)$ $(i = F, B, A)$, then $Mod(B'(adt_A))$ is a subcategory of $Mod(B(spec_A))$ or in other words, if the models of adt_i satisfy the formulas of $spec_i$, then the models of $B'(adt_A)$ satisfy the formulas of $B(spec_A)$.

The inital approach to the semantics of parameterized specifications and their compatibility with *init* is more complex and will be treated in section 5.

[10]Because all left adjoint functors are isomorphic, the semantics of a parameterized specification is often regarded as only one free functor.

4. Amalgamated Sum and Extension Lemma

4.1. Amalgamation. It is desirable to construct the models of $B(adt_A)$ from the models of the parameter adt_A and the models of the body adt_B. But in an arbitrary institution there may be no extension m of a model m_A of adt_A in $Mod(B(adt_A))$, although an extension of $m_A|_\alpha$ exists in $Mod(adt_B)$. For the construction of $B(adt_A)$ from adt_A and adt_B to exist, the institution $\mathcal{I}$ has to have unique amalgamated sums.

Definition 4.1 (Amalgamated Sum). An institution $\mathcal{I} = \langle \mathbf{Sign}, Mod, Sen, \models \rangle$ has *unique amalgamated sums*, if for every pushout

$$\begin{array}{ccc} \Sigma_F & \xrightarrow{\beta} & \Sigma_B \\ {\scriptstyle\alpha}\downarrow & \text{po.} & \downarrow{\scriptstyle\alpha'} \\ \Sigma_A & \xrightarrow[\beta']{} & \Sigma_{po} \end{array}$$

in **Sign** the following holds:
For every $m_B \in Mod(\Sigma_B)$ and $m_A \in Mod(\Sigma_A)$, with $m_B|_\beta = m_F = m_A|_\alpha$, there exists a unique $m \in Mod(\Sigma_{po})$ with

$$m|_{\beta'} = m_A \text{ and } m|_{\alpha'} = m_B.$$

Moreover, for every two morphisms $h_B \in Mod(\Sigma_B)$ and $h_A \in Mod(\Sigma_A)$ with $h_B|_\beta = h_F = h_A|_\alpha$, there exists a unique morphism $h \in Mod(\Sigma_{po})$ with

$$h|_{\beta'} = h_A \text{ and } h|_{\alpha'} = h_B.$$

The structure m and morphism h are called *amalgamated sum* or *amalgamation* [EM85] of m_B and m_A wrt. m_F and of h_B and h_A wrt. h_F, respectively. We will write in this case

$$\begin{aligned} m &= m_B +_{m_F} m_A \text{ and} \\ h &= h_B +_{h_F} h_A, \end{aligned}$$

respectively.

The institution of equational logic and all examples of institutions found in [GB90] have unique amalgamated sums [EM85, ST85], including first order predicate logic.

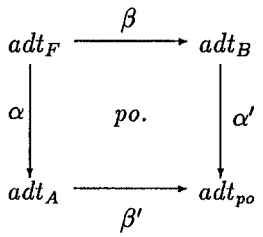
A characterization for an institution to have unique amalgamated sums is that the functor Mod from **Sign** to $\mathbf{Cat}^{op}$ transforms pushouts in **Sign** to pullbacks in **Cat**.

Theorem 4.2. *An institution $\mathcal{I} = \langle \mathbf{Sign}, Mod, Sen, \models \rangle$ has unique amalgamated sums, iff Mod preserves pushouts, that is, Mod transforms pushouts in* **Sign** *to pullbacks in* **Cat**.

The existence of unique amalgamated sums is characteristic for pullbacks in **Cat**. A proof that pullbacks follow from the existence of unique amalgamated sums is part of the amalgamation lemma of [EM85]. A proof for the other direction is given in [Lip82] and [WE85].

The last theorem is important, as it shows that the existence of amalgamated sums depends only on the category of signatures **Sign** and the functor *Mod* of an institution, but not on the form of the sentences or the satisfaction relation. This means that since equational logic has unique amalgamated sums so does for instance conditional equational logic.

Theorem 4.3 (Amalgamation Lemma).
If an institution has unique amalgamated sums then, for each pushout in **Adt**



and for any two structures $m_A \in Mod(adt_A)$ *and* $m_B \in Mod(adt_B)$ *with* $m_A|_\alpha = m_F = m_B|_\beta$ *their amalgamation* $m_A +_{m_F} m_B$ *is in* $Mod(adt_{po})$.

Conversely, for every model m *of* adt_{po} *there exist structures* $m_A \in Mod(adt_A)$ *and* $m_B \in Mod(adt_B)$ *such that* m *is the amalgamation of* m_A *and* m_B.

This also holds for morphisms h_i *of* $Mod(adt_i)$ $(i = F, B, A, po)$.

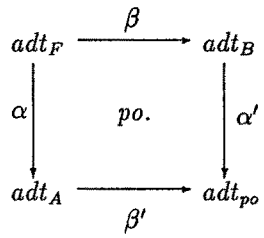
The proof follows directly from the definition of adt_{po} because $Mod(adt_{po})$ can be reformulated as

$$Mod(adt_{po}) = \{m \mid m|_{\beta'} \in Mod(adt_A) \text{ and } m|_{\alpha'} \in Mod(adt_B)\}.$$

The amalgamation lemma mainly states that the model categories of the abstract datatypes, together with the forgetful functors, form a pullback in **Cat**.

4.2. Extension Lemma. As a result of the amalgamation lemma, the models of $B(adt_A)$ can be constructed from the models of the actual parameter and the body. This unique construction allows to extend properties, which hold for the formal parameter, to all admissible parameters. These lemmata are usually called extension lemmata. The classical extension lemma [EM85] is the unique extension of a free functor F from $Mod(spec_F)$ to $Mod(spec_B)$ which is strongly persistent wrt. $Mod(\beta)$ to a strongly persistent, free functor F' from $Mod(spec_A)$ to $Mod(B(spec_A))$ wrt. $Mod(\beta')$ in equational logic. The proof of this extension lemma depends only on the existence of unique amalgamated sums and thus can be formulated more abstract in an arbitrary institution as follows (s.a. [Lip82, EGL89, EPO89, Poi89]):

Theorem 4.4 (Extension Lemma I). *If an institution has unique amalgamated sums, then the following holds:*
Given a pushout in **Adt**



and a functor $F\colon Mod(adt_F) \to Mod(adt_B)$

(1) *If* F *is strongly persistent wrt.* $Mod(\beta)$*, then there exists a unique functor* F' *from* $Mod(adt_A)$ *to* $Mod(adt_{po})$ *which is strongly persistent wrt.* $Mod(\beta')$ *with*

$$F \circ Mod(\alpha) = Mod(\alpha') \circ F'.$$

(2) *Moreover, if* F *is strongly persistent and left adjoint to* $Mod(\beta)$*, then* F' *is left adjoint to* $Mod(\beta')$.

The functor F' is defined as follows. For each structure m_A and morphism h_A in $Mod(adt_A)$ we have:

$$\begin{aligned} F'(m_A) &:= m_A +_{m_A|_\alpha} F(m_A|_\alpha) \text{ and} \\ F'(h_A) &:= h_A +_{h_A|_\alpha} F(h_A|_\alpha). \end{aligned}$$

It is not necessary that $F(m_F)|_\beta = m_F$ for all models m_F of adt_F to get a strongly persistent functor F' from $Mod(adt_A)$ to $Mod(B(adt_A))$. It suffices that F restricted on $Mod(adt_A)|_\alpha$ is strongly persistent, since only for models m_A of adt_A it is required that $F(m_A|_\alpha)|_\beta$ equals $m_A|_\alpha$ to make the amalgamation in the definition of F' work.

Theorem 4.5 (Extension Lemma II). *If an institution* $\mathcal{I}$ *has unique amalgamated sums and if a parameterized abstract datatype* B *is consistent for its formal parameter, then* B *is consistent for all admissible parameters.*

If B *is unique for its formal parameter, then it is unique for all admissible parameters.*

This holds, because for each model m_A of adt_A, $B_R(m_A)$ can be uniquely constructed from the amalgamated sums of m_A and the elements of $B_R(m_A|_\alpha)$, therefore B is consistent and unique for adt_A, whenever B is consistent and unique for its formal parameter.

5. Compatibility of *pinit* with Initial Semantics

The importance of free functors F in the initial semantics of parameterized specifications is the fact that they construct for each model m_A of the actual parameter specification a universal arrow $\langle m_A, \eta\colon m_A \to F(m_A)|_{\beta'}\rangle$ from m_A to $Mod(\beta')$. A universal arrow has the property that, if m_A is an initial object in $Mod(spec_A)$, then $F(m_A)$ is an initial object in $Mod(B(spec_A))$.

Definition 5.1 (Universal Arrow). Given a functor $V\colon \mathbf{D} \to \mathbf{C}$ and a morphism η from c to $V(d)$ then $\langle c, \eta\rangle$ is a universal arrow from c to V, if for every $f\colon c \to V(d')$ in $\mathbf{C}$ there exists a unique morphism $\tilde{f}\colon d \to d'$ in $\mathbf{D}$ with $V(\tilde{f})\eta = f$.

The notion of β–freeness defined here, is a special case of an universal arrow, where η is the identity.

Definition 5.2 (β–free). Be β: $spec_F \rightarrow spec_B$ a **Spec**–morphism. A model m_B of $spec_B$ is called β–free, if $\langle m_B|_\beta, id_{m_B|_\beta}\rangle$ is a universal arrow from $m_B|_\beta$ to $Mod(\beta)$.

It is important to notice that this definition of β–freeness does not require an adjunction; but if for every $m_F \in Mod(spec_F)$ a β–free model m_B of $spec_B$ exists, then any choice of a model m_B for each m_F determines a free functor from $Mod(spec_F)$ to $Mod(spec_B)$ [Mac88] which is strongly persistent wrt. $Mod(\beta)$.

This definition of β–freeness is equivalent to the one found in [GB90]. In [GB90] m_B is β–free, if there exists a free extension of $m_B|_\beta$ which is isomorphic to m_B via the counit morphism ϵ_{m_B}. Our notion seems to be more appropriate if we do not assume the existence of an adjoint situation while the definition of [GB90] is easier to check in the presence of an adjunction. Similar requirements to β–freeness are the core of the definitions of data constraints [GB90], initial restrictions [Rei86] and free generating constraints [WE85, Ehr89, OSC89].

Lemma 5.3. *Let m be β–free for a* **Spec***–morphism β from $spec_F$ to $spec_B$ and be m' a model of $spec_B$ isomorphic to m, then m' is β–free, too.*

In the case where unique amalgamated sums exist, the β–freeness of a structure m_B extends to the β'–freeness of a structure m with $m|_{\alpha'} = m_B$.

Lemma 5.4 (Extension Lemma III). *If an institution has unique amalgamated sums, then the following holds:*
Given a pushout in **Spec**

$$\begin{array}{ccc} spec_F & \xrightarrow{\beta} & spec_B \\ {\scriptstyle\alpha}\downarrow & po. & \downarrow{\scriptstyle\alpha'} \\ spec_A & \xrightarrow[\beta']{} & spec_{po} \end{array}$$

and a β–free model m_B of $spec_B$, then each model m of $spec_{po}$ with $m|_{\alpha'} = m_B$ is β'–free.

Now we can proceed to define a semantic function *pinit* on parameterized specifications which is compatible with the initial semantics of specifications. The trick is to only allow β–free extensions of formal parameter models as the models of the body.

Definition 5.5 (*pinit*). The initial semantics $pinit(B)$ of a parameterized specification $B(X : spec_F)_\beta : spec_B$ is defined as the parameterized abstract datatype $B(X : adt_F)_\beta : adt_B$, where

- $adt_F = loose(spec_F)$ and
- $adt_B = \langle Sign(spec_B), \{m \in Mod(spec_B) \mid m \text{ is } \beta\text{–free}\}\rangle$.[11]

The following theorem is more general then needed for the compatibility of *pinit* with *init*. It handles the case where $Mod(adt_F)$ is an arbitrary subcategory of $Mod(spec_F)$. This can be useful when excluding unwanted admissible parameters. Take, for example, as formal parameter the specification of a partial order in equational logic. A part of

[11] the full subcategory of $Mod(Sign(spec_B))$ is meant.

this specification is a specification of BOOL with sort *bool*. An admissible parameter may be a specification which defines a partial order together with a new constant of sort *bool* and no equations involving the new constant. This parameter is admissible as long as all equalities of the formal parameter follow from the admissible parameter, but its initial model has at least 3 elements of sort *bool* and therefore, we would not like to have this model as an admissible parameter. With the use of data constraints (e.g. [GB90, Rei86, Ehr89, OSC89]) we could have added a constraint to the formal parameter specification ensuring that the BOOL part of any partial order is the initial Boolean algebra.

Theorem 5.6. *In an institutuion I which has unique amalgamated sums, the following holds:*
Given a parameterized specification $B(X : spec_F)_\beta : spec_B$, a parameterized abstract data-type $B_a(X : adt_{F_a})_{\beta_a} : adt_{B_a}$ and a specification $spec_A$ such that

(1) $Sign(\beta) = Sign(\beta_a)$,
(2) $Mod(adt_{F_a})$ *is a subcategory of* $Mod(spec_F)$,
(3) $|Mod(adt_{B_a})| = \{m \in Mod(spec_B) \mid m \text{ is } \beta\text{–free and } m|_\beta \in Mod(adt_{F_a})\}$
(4) B_a *is consistent for its formal parameter,*
(5) $spec_A$ *is an admissible parameter for* B,
(6) $init(spec_A)$ *is an admissible parameter for* B_a *and*
(7) $init(spec_A)$ *has at least one model*

then

$$B_a(init(spec_A)) = init(B(spec_A)).$$

Proof. ($\subseteq$) Assume m_A is an initial object of $Mod(spec_A)$ and m is an element of $B_{aR}(m_A)$. Because of the extension lemma for β–freeness (lemma 5.4) and the fact that $m|_{\alpha'}$ is β–free, m is β'–free. Then m is also an initial object of $Mod(B(spec_A))$ since $m|_{\beta'} = m_A$ is initial in $Mod(spec_A)$.

($\supseteq$) For each initial object m of $Mod(B(spec_A))$ it is to show that $m|_{\beta'}$ is an initial object of $Mod(spec_A)$, $m|_{\alpha'}$ is β–free and $(m|_{\alpha'})|_\beta$ is a model of adt_{F_a}. If $m|_{\beta'}$ is an initial object of $Mod(spec_A)$ then $(m|_{\beta'})|_\alpha = (m|_{\alpha'})|_\beta$ is a model of adt_{F_a} because $init(spec_A)$ is an admissible parameter for B_a.

Since $Mod(init(spec_A))$ is not empty and B_a is consistent, there exists, due to the first part of this proof, an inital object m' of $Mod(B(spec_A))$ such that $m'|_{\beta'}$ is an initial object of $Mod(spec_A)$ and $m'|_{\alpha'}$ is β–free. Then m' has to be isomorphic to m and, because initiality and β–freeness are closed under isomorphism and functors preserve isomorphism, we have that $m|_{\beta'}$ is initial in $Mod(spec_A)$ and $m|_{\alpha'}$ is β–free. □

Corollar 5.7 (Compatibility of *pinit* with initial semantics). *In an institutuion I which has unique amalgamated sums, the following holds:*
Given a parameterized specification $B(X : spec_A)_\beta : spec_B$ and an admissible parameter $spec_A$ such that

(1) $pinit(B)$ *is consistent for its formal parameter and*
(2) $Mod(spec_A)$ *has an initial object*

then

$$pinit(B)(spec_A) = init(B(spec_A))$$

The conditions for the compatibility of *pinit* with *init* are equivalent to those for the correctness of parameter passing in the free functor semantics of [TWW82, EM85]. The condition that $init(spec_A)$ has at least one model is always true in equational logic [GTW76]. Because $pinit(B)$ is consistent for adt_F, there exists for each $m_F \in Mod(spec_F)$ a structure $m_B \in pinit(B)_R(m_F)$ which is β–free. Any choice of a structure m_B for each m_F determines a unique left adjoint functor F from $Mod(spec_F)$ to $Mod(spec_B)$ [Mac88], where F is strongly persistent wrt. $Mod(\beta)$. The extension lemma for consistency (lemma 4.5) and for β–freeness (lemma 5.4) then yield a functor F' from $Mod(spec_A)$ to $Mod(B(spec_A))$ and therefore correspond to the classical extension lemma for a strongly persistent, free functor in [EM85].

On the other hand each strongly persistent, free functor $F\colon Mod(spec_F) \to Mod(spec_B)$ induces an abstract datatype adt_B, where $Mod(adt_B)$ is $F(Mod(spec_F))$ closed under isomorphism. Then all models of adt_B are β–free and the parameterized abstract datatype $B(X : adt_F)_\beta : adt_B$ is consistent because F yields for each $m_F \in Mod(spec_F)$ a β–free structure $F(m_F)$ with $F(m_F) \in B_R(m_F)$.

6. Conclusion and related work

In this paper we have presented a theory of parameterized abstract datatypes to study the model theory of parameterized specifications. We have shown that the most fundamental approaches to the semantics of parameterized specifications, the loose and initial approach, can be modeled within this theory.

Using dataconstraints and constraint institutions as introduced by Burstall and Goguen [GB90, BG80] for the semantics of Clear, we can formulate the initial semantics of a parameterized specification B in an institution $\mathcal{I}$ by the loose semantics of a parameterized specification B' in the corresponding constraint institution $\mathcal{C}(I)$. For each B there is a B' such that $pinit(B)$ equals $ploose(B')$. Thus it is possible in the software development process to proceed in one and the same specification language from loose requirement specifications to more and more concrete specifications, representing, in the final stage, exactly one model up to isomorphism,.

This approach has been taken by Orejas et al. [OSC89] for the specific institution of equational logic with constraints. Their loose extensions can be modeled by consistent parameterized abstract datatypes and parameter passing is done by the notion of refinement. They also give proof theoretic conditions for the consistency of parameterized abstract datatypes where the models of the formal parameter are those satisfying a set of equations together with a set of constraints.

A similar approach to that of Orejas et al. is taken by Reichel [Rei86]. There initiality restrictions together with behavioural semantics are investigated in conditional equational logic. The difference is that the body of a parameterized abstract datatype is required to be a recursive extension of the parameter. A recursive extension of a parameter corresponds roughly to a free extension of that parameter. It is not possible to formulate loose extensions of the formal parameter.

The purpose of the use of constraints by Ehrig in [Ehr89] is to eliminate unwanted admissible parameter specifications that can not be eliminated by adding equations only. E.g., if the formal parameter contains a specification of the booleans using only equations then a parameter specification with *true* and *false* equated could not be avoided as

an admissible parameter. Constraints are not used to abandon the free functor in the semantics of parameterized specifications as done in this paper, thus it is not possible to model the loose semantics of parameterized specifications of Clear.

In the course of our studies we have motivated two desirable requirements an institution should satisfy. The category **Sign** of signatures should be finitely cocomplete to ensure that all applications of parameterized specifications and parameterized abstract datatypes to specifications and abstract datatypes are well defined. Secondly, the functor *Mod* should preserve finite colimits. This is necessary for the existence of unique amalgamated sums and therefore for an amalgamation and extension lemma. These requirements are fullfilled in many interesting institutions such as equational logic, conditional equational logic, horn clause logic and first order predicate logic with and without equality.

Both requirements allow us also to define constraints that describe the initiality of a structure m in the category of the models of a specification *spec*. If m is β–free for β: $spec_0 \rightarrow spec$, where $spec_0$ is the initial object in **Spec**, then m is an initial object in $Mod(spec)$. Since $spec_0$ is initial in **Spec**, β always exists. If *Mod* preserves finite colimits it takes the initial object of **Sign** to the terminal object **1** of **Cat**. Because **1** contains exactly one object this object is trivially initial in **1** and so is m in $Mod(spec)$ if m is β–free.

The theory presented in this paper is based mainly on pushouts but it should be easy to extend to arbitrary colimits. These are useful when defining more complex parameterized specifications as in [Kre89] or when dealing with environments of shared specifications as in [BG80].

Acknowledgements. This paper is an extension of a part of my diploma thesis at the University of Dortmund [Bau90]. I would like to thank my supervisor Harald Ganzinger for his support and Hagen Huwig for fruitful discussions.

References

[Bau90] Hubert Baumeister. Über die Stabilität parameterisierter algebraischer Spezifikationen (About the stability of parameterized algebraic specifications). Master's thesis, Universität Dortmund, July 1990.

[BG80] R.M. Burstall and J.A. Goguen. The semantics of Clear, a specification language, February 1980.

[Cla87] Ingo Claßen. Revised ACT ONE: Categorical constructions for an algebraic specification language. In H. Ehrig, H. Herrlich, J.-J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science with Aspects from Topology*, number 393 in LNCS, pages 124–141. Springer, 1987.

[EGL89] Hans-Dieter Ehrich, Martin Gogolla, and Udo Walter Lipeck. *Algebraische Spezifikation abstrakter Datentypen (Algebraic Specification of Abstract Datatypes).* B. G. Teubner Stuttgart, 1989.

[Ehr89] Hartmut Ehrig. A categorical concept of constraints for algebraic specifications. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical Methods in Computer Science*, number 393 in LNCS. Springer Verlag, 1989.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1990.

[EPO89] H. Ehrig, P. Pepper, and F. Orejas. On recent trends in algebraic specification. In *ICALP 89*, pages 263–288, 1989.

[GB90] J. A. Goguen and R. Burstall. INSTITUTIONS: Abstract model theory for specification and programming, 1990.

[GTW76] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types, 1976.

[Kre89] Hans-Jörg Kreowski. Colimits as parameterized data types. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical methods in Computer Science with aspects from Topology*, pages 36–49. Springer, 1989.

[Lip82] Udo Lipeck. *Ein algebraischer Kalkül für einen strukturierten Entwurf von Datenabstraktionen (An Algebraic Calculus for a Structured Design of Data Abstraction).* PhD thesis, Universität Dortmund, 1982.

[Mac88] Saunders Mac Lane. *Categories for the working mathematician.* Graduated Texts in Mathematics. Springer, fourth edition, 1988.

[OSC89] F. Orejas, V. Sacristán, and S. Clerici. Development of algebraic specifications with constraints. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical methods in Computer Science with aspects from Topology*, pages 36–49. Springer, 1989.

[Poi89] Axel Poigné. Foundations are rich institutions, but institutions are poor foundations. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Categorical methods in Computer Science with aspects from Topology*, pages 82–101. Springer, 1989.

[Rei86] H. Reichel. Software specification by behavioural canons. TU Magdeburg, 1986.

[Sch86] Oliver Schoett. *Data Abstraction and the Correctness of Modular Programming.* PhD thesis, University of Edinburgh, 1986.

[SST90] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: parameterization revisited, 1990.

[ST84] Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. Technical Report CSR-172-84, University of Edinburgh, 1984.

[ST85] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. Draft 1, March 1985.

[ST88] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: Foundations and methodology. draft (short version), September 1988.

[SW83] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification. Internal Report CSR–131-83, University of Edinburgh, September 1983.

[TWW82] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: Parameterization and the power of specification techniques. *acm Transactions on Programming Languages and Systems*, 4(4):711–732, October 1982.

[WE85] E. G. Wagner and H. Ehrig. Canonical constraints for parameterized data types. Research Report RC 11248 (50666), IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598 / USA, Jul 1985.

UNIVERSITÄT DORTMUND, INFORMATIK V, POSTFACH 50 05 00, W–4600 DORTMUND-50, GERMANY

E-mail: huba@ls5.informatik.uni-dortmund.de