Theory and practice of canonical term functors in abstract data type specifications

Christoph Beierle
TK LILOG, IBM Deutschland GmbH
Postfach 800880, D-7000 Stuttgart 1
EARN/BITNET: BEIERLE at DSØLILOG

Angelika Voss
GMD, Gruppe Expertensysteme
Postfach 1240, D-5205 St. Augustin 1
USENET: AVOSS%GMDXPS at GMDZI

Abstract: Term algebras have been widely used in the theory of abstract data types. Here, the concept of canonical term algebra is generalized to the notion of canonical term functor, which is useful for various aspects of abstract data type specifications. In particular, we show how canonical term functors provide a constraint mechanism in loose specifications and how they constitute a junction between axiomatic and constructive approaches. These concepts are the semantic base for the specification development language ASPIK which has been implemented as a core component of an integrated software development and verification system.

1. Introduction

During the last decade the field of abstract data type theory has received much attention, yielding numerous papers on various approaches. To mention only a few, there is the initial approach proposed by the ADJ group ([GTW 78], see also [EM 85]) which is based on equational specifications and was later generalized to conditional equations and universal Horn clauses, the terminal approach advocated by e.g. [GGM 76], [Wa 79], [Kam 80], the loose approaches of Clear [BG 80] or CIP-L [CIP 85], the algorithmic approaches of [Cart 80], [KL 84], [Lo 84], etc., for a more complete list of references see e.g. [EM 85].

In many of these approaches term algebras have played an important role, and the concept of canonical term algebra as introduced in [GTW 78] has been used also in e.g. [Pad 79] and [KL 83]. Here we generalize canonical term algebras to the new notion of canonical term functors and show how this notion provides a powerful concept both under theoretical and practical aspects of abstract data types. It eases the stepwise development and verification of specifications, provides a constraint mechanism in loose specifications and can be used as a junction between high level axiomatic and lower level algorithmic or constructive approaches.

The concept of canonical term functor has already been exploited extensively in design and implementation of the specification development language ASPIK which together with its support environment SPESY is a core component of the ISDV system, an integrated software development and verification system [BV 85], [BOV 86].

This paper is organized as follows: Section 2 contains some preliminaries about algebraic

specifications and fixes our notation. In section 3 we recall the definition of canonical term algebra and show how it can be generalized to canonical term functors. Additionaly, we define strict versions of both concepts, supporting partial operations and a simple error handling mechanism. Composability of canonical term functors and other properties are proved. In section 4 various applications are described, section 5 briefly discusses ASPIK and its support in SPESY, and section 6 contains concluding remarks.

Acknowledgements: This work was performed at the Universität Kaiserslautern and was supported in part by the Bundesministerium für Forschung und Technologie (IT 8302363) and the Deutsche Forschungsgemeinschaft (SFB 314).

2. Preliminaries: Algebraic specifications

A signature $\Sigma = \langle S, Op \rangle$ consists of a set S of sorts or types and an $S^* \times S$ - sorted set Op of typed operation names. For $op \in Op$ the notation $op: s_1 \ldots s_n \rightarrow s$ means that op has argument sorts $s_1 \ldots s_n$ and target sort s.
A Σ-algebra $A = \langle (A_s \mid s \in S), (op_A: A_{s1} \times \ldots \times A_{sn} \rightarrow A_s \mid op: s_1 \ldots s_n \rightarrow s \in Op) \rangle$ provides a data set or carrier A_s for each sort s and an operation op_A for each operation symbol op in Op. A Σ-homomorphism $h: A \rightarrow A'$ is an S-sorted family of functions $(h_s: A_s \rightarrow A_s' \mid s \in S)$ such that h commutes with the algebra operations in A and A'. Alg(Σ) denotes the category of all Σ-algebras together with all Σ-homomorphisms.
A specification $SP = \langle \Sigma, E \rangle$ consists of a signature Σ and a set E of sentences over Σ. This defines the class of $\langle \Sigma, E \rangle$-algebras which are all Σ-algebras satisfying the sentences E. According to the ADJ approach the isomorphism class of the initial $\langle \Sigma, E \rangle$-algebra is the abstract data type specified by SP.

The initial approach of the ADJ-group is an example of a so-called fixed approach where a specification has only isomorphic models. Fixed approaches were generalized to so-called loose approaches where a specification $SP = \langle \Sigma, E \rangle$ may also have non-isomorphic models, for example, the class of all Σ-algebras satisfying E is considered, not just the initial ones. Whereas the initial as well as the terminal approach (e.g. [Wa 79], [Kam 80]) have to restrict the types of admissible sentences in order to guarantee the existence of an initial (resp. terminal) model, there is no such need in a loose approach: Equations are considered in [GTW 78], positive conditional equations in [TWW 82], and universal Horn sentences in [EKTWW 80], whereas in the loose approach of [CIP 85] arbitrary first order formulas are allowed. Other loose approaches are e.g. [BG 77, 80, 81], [HKR 80], [SW 82], [ZLT 82] and [EWT 82]. Beside logical formulas, these approaches need so-called constraints as another type of sentences in order to exclude unreachable elements or non-standard interpretations. In sections 4.3 and 4.4 we will show how algorithmic definitions may be used for this purpose.

3. Canonical term functors

We introduce canonical term functors as generialization of canonical term algebras in section 3.1. In section 3.2 we specialize the definition to strict canonical term functors

that are better suited as semantics of algebraic definitions. Properties of both types of canonical term functors are given in section 3.3.

3.1 Definition

In the initial approach of the ADJ-group, an abstract data type is defined as the isomorphism class of the initial algebra of an equational specification $SP = \langle \Sigma, E \rangle$. Such an initial algebra can be obtained by taking the free Σ-term algebra T_Σ and imposing the congruence generated by E on the carriers. The resulting quotient algebra $T_{\Sigma/E}$ has equivalence classes as its carrier elements. This definition in terms of equivalence classes is rather abstract. Sometimes, a more concrete definition is preferable, for example in order to compute terms over initial specifications. This role of a concrete initial algebra can be played especially by a canonical term algebra (cta), which exists for every equational specification. Unfortunately, the existence proof is non-constructive [GTW 78], and in general there is no algorithm to generate an initial cta from a specification.

A cta is obtained from the quotient term algebra $T_{\Sigma/E}$ by choosing a representative out of each equivalence class. Two restrictions are imposed on this selection: The first guarantees that the carriers are closed under subterm formation (subterm property), the second guarantees that the carriers are operation-generated. We call this the constructor property since the generating operations are often called constructors.

Definition 3.1 [canonical term algebra, cta]

Let $\Sigma = \langle S, Op \rangle$ be a signature and $A \in Alg(\Sigma)$ an algebra. A is a canonical Σ-term algebra (Σ cta, or just cta) iff

(1) $\forall s \in S . A_s \subseteq T_{\Sigma,s}$ (term property)

(2) $\forall op: s_1 \ldots s_n \to s \in Op .$

$op(t_1,\ldots,t_n) \in A_s$

$\Rightarrow t_1 \in A_{s1} \;\&\; \ldots \;\&\; t_n \in A_{sn}$ (subterm property)

$\&\; op_A(t_1,\ldots,t_n) = op(t_1,\ldots,t_n)$ (constructor property).

The initial approach to ADTs was extended to parameterized ADTs ([TWW 78], [EKTWW 80]). A parameterized specification $PSP = \langle FSP, \Sigma, E \rangle$ consists of a formal parameter specification $FSP = \langle F\Sigma, FE \rangle$, a signature Σ extending $F\Sigma$ ($F\Sigma \subseteq \Sigma$) and equations E extending FE ($FE \subseteq E$). The formal parameter specification denotes the class of all FSP-algebras to be parameter algebras. The parameterized specification PSP denotes the free functor $free_{PSP}$: $Alg(FSP) \to Alg(\langle \Sigma, E \rangle)$ that maps every parameter algebra A to its free extension $free_{PSP}(A)$, which can again be defined very abstractly in terms of equivalence classes over the free term algebra $T_\Sigma(A)$ generated from A.

Therefore the same reasons that led to consider a cta instead of a quotient term algebra also apply in the parameterized case: Sometimes it is preferable to have a more concrete

definition than just the implicit definition of a free functor by equivalence classes. This role can be played by the so-called canonical term functor (ctf), which we define by a straightforward generalization of the cta definition: Since the parameter algebras must not be affected, a ctf should be strongly persistent, and the term-, subterm-, and constructor properties must now be restricted to the new sorts. In order to ease the precise definition of these ideas we first introduce some auxiliary notions for expressing the cta-requirements relative to a parameter algebra A.

Definition 3.2 [term-, subterm-, constructor property]
Let Σ, Σ' be signatures such that $\Sigma \subseteq \Sigma'$. Let $A \in Alg(\Sigma)$ and $B \in Alg(\Sigma')$.

(1) B has the $(\Sigma' - \Sigma)$-term property w.r.t. A iff
$$\forall s \in \Sigma' - \Sigma.\ B_s \subseteq T_{\Sigma' - \Sigma}(A)_s$$

(2) B has the $(\Sigma' - \Sigma)$-subterm property iff
$$\forall s \in \Sigma' - \Sigma\ .\ \forall\ op: s_1 \dots s_n \to s \in \Sigma' - \Sigma\ .$$
$$op(t_1,\dots,t_n) \in B_s$$
$$\Rightarrow t_1 \in B_{s1}\ \&\ \dots\ \&\ t_n \in B_{sn}$$

(3) A' has the $(\Sigma' - \Sigma)$-constructor property iff
$$\forall s \in \Sigma' - \Sigma\ .\ \forall\ op: s_1 \dots s_n \to s \in \Sigma' - \Sigma\ .$$
$$op(t_1,\dots,t_n) \in B_s$$
$$\Rightarrow op_B(t_1,\dots,t_n) = op(t_1,\dots,t_n)$$

Definition 3.3 [canonical term functor, ctf]
Let $\iota: \Sigma \to \Sigma'$ be a signature inclusion, and let $C \subseteq Alg(\Sigma)$ and $C' \subseteq Alg(\Sigma')$ be subcategories. A functor $g: C \to C'$ is a canonical (Σ, Σ')-term functor ((Σ, Σ')-ctf, or just ctf iff

(1) g is strongly persistent:
$$Alg(\iota) \circ g = id_C$$

(2) For every $A \in C$, (2.1) - (2.3) hold:
(2.1) g(A) has the $(\Sigma' - \Sigma)$-term property w.r.t. A
(2.2) g(A) has the $(\Sigma' - \Sigma)$-subterm property
(2.3) g(A) has the $(\Sigma' - \Sigma)$-constructor property

As an example, let $ELEM = \langle \langle \{elem\}, \emptyset \rangle, \emptyset \rangle$ specify all one-sorted algebras as parameters. Let $\Sigma LIST = \langle \{elem, list\}, \{nil, cons, car, cdr\} \rangle$ be the signature of linear lists with elements of sort elem. Define g as a functor $g: Alg(ELEM) \to Alg(\Sigma LIST)$ whose object part extends every parameter algebra $A \in Alg(ELEM)$ by the list carrier $\{nil,\ cons(e_1, nil),\ cons(e_2, cons(e_1, nil)) \dots \mid e_1, e_2, \dots \in A_{elem}\}$ and by the usual list operations such that e.g.

$cons_{g(A)}(e_2, cons(e_1, nil)) = cons(e_2, cons(e_1, nil))$

$car_{g(A)}(cons(e_2, cons(e_1, nil))) = e_2$

$cdr_{g(A)}(cons(e_2, cons(e_1, nil))) = cons(e_1, nil)$.

Then g is a ctf for the following reasons:

1. g is strongly persistent since the parameter algebra A is not modified.

2. g(A) has the term property because the list objects are term generated by the new operations nil and cons over the elements of A.
3. g(A) has the subterm property since for every list carrier element $cons(e_i,t)$ t is also in the list carrier.
4. g(A) has the constructor property since for the constructor operations nil and cons we have $nil_{g(A)} = nil$ and $cons_{g(A)}(e_i,t) = cons(e_i,t)$ for every term $cons(e_i,t)$ in the list carrier.

3.2 Strict canonical term functors

So far we have considered all total algebras as models. But there are many reasons to consider also partial algebras, e. g. partially defined operations or non-terminating recursion, c. f. [CIP 85] among others. A particular method to deal with partial algebras is to extend the operation domains by a new element ('undefined'), and to extend the operations such that they are strict w. r. t. the new element. We say that a strict algebra has carriers with a minimal element, called the error element, and strict operations propagating the error elements. Whereas $Alg(\Sigma)$ denotes the class of all Σ-algebras, we use $EAlg(\Sigma)$ to denote the class of all strict algebras. To make the error elements addressable in our specifications we introduce error constants error-s for each sort s in a signature Σ yielding the signature $Err(\Sigma)$. Thus, a strict Σ-algebra in particular is an ordinary $Err(\Sigma)$-algebra.

Now we can replace Σ by $Err(\Sigma)$ and $Alg(\Sigma)$ by $EAlg(\Sigma)$ in the definition of cta. Additionally we require that in every carrier the error element is represented by the error constant. The latter requirement is not necessary, but convenient since it allows to define the error constants implicitly.

Definition 3.4 [strict cta]

Let $\Sigma = \langle S,Op \rangle$ be a signature and $A \in EAlg(\Sigma)$ a strict Σ -algebra. A is a strict Σ-cta iff

(1) A is an (ordinary) $Err(\Sigma)$-cta

(2) $\forall\, s \in S$. error-$s \in A_s$.

Just as we obtained the definition of a strict cta from the definition of cta by adding the implicit error constants and the error requirement (2), we define a strict ctf to be a ctf between two categories of strict algebras and add the error requirement w.r.t. the new carriers.

Definition 3.5 [strict ctf]

Let $\iota: \Sigma \to \Sigma'$ be a signature inclusion, and let $C \subseteq EAlg(\Sigma)$ and $C' \subseteq EAlg(\Sigma')$ be subcategories. A functor $g: C \to C'$ is a strict (Σ,Σ')-ctf iff

(1) g is an $(Err(\Sigma), Err(\Sigma'))$-ctf

(2) $\forall\, s \in \Sigma'\text{-}\Sigma\ .\ \forall\, A \in C.$ error-$s \in g(A)_s$.

As an example consider again the ctf g defining lists over arbitrary elements from section 3.1. Supplementing the missing operation definitions we would have difficulties to define $car_{g(A)}$ (nil) in a total algebra approach. Assuming g as a strict ctf that extends strict ELEM-algebras A, we can define $car_{g(A)}$ (nil) := error-$elem_A$ to yield the error element of sort elem in algebra A. This corresponds exactly to a partial algebra approach (as in the CIP project [CIP 85]) when we forget the error elements and analogously restrict the algebra operations to partial operations. Thus $car_{g(A)}$(nil) = error-$elem_A$ means that $car_{g(A)}$(nil) is undefined if g(A) was a partial algebra, and the fact that $cons_{g(A)}(car_{g(A)}(nil), nil)$ would also be undefined in a partial algebra g(A) is reflected in the strictness of the operations in a strict algebra g(A). However, taking the approach of strict algebras we can stay within the simpler framework of total algebras.

3.3 Properties

We motivated the concept of ctfs as concrete counterparts of free functors defined by equivalence classes in the semantics of parameterized specifications, just like ctas are concrete counterparts of initial quotient term algebras in the semantics of non-parameterized specifications. Therefore, we would like to have the following correspondences between ctfs and free functors:

(1) Since a constant free functor defines an initial algebra, a constant ctf should yield a cta.
(2) Since the composition of free functors yields again a free functor, the composition of ctfs should yield again a ctf.
(3) Since the application of a free functor to an initial algebra yields again an initial algebra, the application of a ctf to a cta should yield again a cta.

These properties hold both for ordinary and for strict ctfs due to the following three facts:

Fact 3.6 [constant ctfs are ctas]
Let Σ be a signature, $A \in Alg(\Sigma)$ [resp. $A \in EAlg(\Sigma)$] and

$$1_A: Alg(\langle\emptyset,\emptyset\rangle) \to Alg(\Sigma)$$

[resp. $1_A: EAlg(\langle\emptyset,\emptyset\rangle) \to EAlg(\Sigma)$]

be the constant functor yielding A. Then we have:
A is a [strict] Σ-cta $\Leftrightarrow$ 1_A is a [strict] $(\langle\emptyset,\emptyset\rangle,\Sigma)$-ctf.

Proof
1_A is strongly persistent since the only object in $Alg(\langle\emptyset,\emptyset\rangle)$ [resp. $EAlg(\langle\emptyset,\emptyset\rangle)$] is the empty algebra with no sorts and no operations, and for $\Sigma - \langle\emptyset,\emptyset\rangle = \Sigma$ [resp. $Err(\Sigma) - Err(\langle\emptyset,\emptyset\rangle) = Err(\Sigma)$], the [strict] Σ-term, [strict] Σ-subterm, and [strict] Σ-constructor properties w. r. t. the empty algebra coincide with the conditions for a [strict] Σ-cta.

The next two facts state that ctfs are closed under composition and that a ctf applied to a cta yields again a cta.

Fact 3.7 [ctfs are closed under composition]

Let $g_1: C_1 \to C_2$ be a [strict] (Σ_1,Σ_2)-ctf and $g_2: C_2 \to C_3$ be a [strict] (Σ_2,Σ_3)-ctf. Then

$$g_2 \circ g_1: C_1 \to C_3$$

is a [strict] (Σ_1,Σ_3)-ctf.

Proof

$g_2 \circ g_1$ is strongly persistent since both g_1 and g_2 are strongly persistent. For $A \in C_1$, $g_2(g_1(A))$ has the [strict] $(\Sigma_3\text{-}\Sigma_2)$-term property w. r. t. $g_1(A)$ since g_2 is a [strict] (Σ_2,Σ_3)-ctf, and $g_1(A)$ has the [strict] $(\Sigma_2\text{-}\Sigma_1)$-term property w. r. t. A since $g_1$is a [strict] (Σ_1,Σ_2)-ctf. Thus, $g_2(g_1(A))$ has the [strict] $(\Sigma_3\text{-}\Sigma_1)$-term property w. r. t. A since g_2 is strongly persistent. A similar argument shows that $g_2(g_1(A))$ has the [strict] $(\Sigma_3\text{-}\Sigma_1)$-subterm and [strict] $(\Sigma_3\text{-}\Sigma_1)$-constructor properties.

Fact 3.8 [application of ctfs to ctas]

Let $g: C \to C'$ be a [strict] (Σ,Σ')-ctf, and A a [strict] Σ-cta with $A \in C$. Then g(A) is a [strict] Σ'-cta.

Proof

According to Fact 3.6, A can be identified with the constant functor I_A, and I_A can be corestricted to C since $A \in C$. The composition $g \circ I_A$ is a [strict] ctf due to Fact 3.7. Since $g \circ I_A$ is a constant functor from the category containing just the empty algebra, its value g(a) is a [strict] Σ'-cta according to Fact 3.6.

4. Applications

4.1. Proofs and stepwise verification

The power of the cta concept is demonstrated in [GTW 78] by showing that for every equational specification an initial cta exists. Similarly, we can prove an analogous result for the parameterized case.

Fact 4.1 [existence of a ctf]

Let $PSP = \langle FSP, \Sigma, E \rangle$ be a parameterized equational specification with parameter specification $FSP = \langle F\Sigma, FE \rangle$ such that the induced free functor is persistent. Then there exists a free functor

$$ctf_{PSP}: Alg(FSP) \to Alg(\Sigma, E)$$

which is a ctf.

As in the non-parameterized cta case the proof of this fact which is given in the appendix is non-constructive, but similarly as for ctas there is often a natural choice for selecting the canonical representatives for a ctf.

Knowing that ctfs exist, we can use them like ctas instead of quotient term algebras with the advantage that one can exploit the term structure in induction proofs (c.f. [GTW 78], [Pad 79], [Kl 83] and - for a variant of ctas - [EM 85]). Moreover, if one has a system of

equational parameterized specifications of say sets over lists over some elements and corresponding ctfs one can verify them w.r.t. given properties in a stepwise manner by considering first the set ctf and then the list ctf, or the other way around. This modularization also allows us to do the set verification part only once (e.g. showing that the union operation is commutative) and to use it for other instances like sets over arrays as well. Such a modularization is used extensively in our ISDV system [BV 85, BOV 86].

4.2 Constraints

Whereas in a fixed approach to abstract data type specifications like the initial or terminal one there is no need for a constraint mechanism, such a mechanism is needed in a loose approach where all models of a specification are considered. The reason is that at least for some substructures one wants to allow only a standard interpretation. For example, a specification of the natural numbers should not allow for a model with additional elements like ∞ that cannot be generated by the usual natural numbers operations. The loose approaches of [HKR 80], [BG 80] and [EWT 83] use a constraint mechanism involving a free functor. The hierarchy constraints proposed in [SW 82] are weaker in the sense that - apart from requiring true $\neq$ false - they only exclude unreachable elements ("no-junk" condition) while the other approaches also require that generated elements must be distinct ("no-confusion" condition).

The functor involved in these constraint mechanisms is the free functor defined by equational theories. In [GB 83] the more general case of data constraints in so-called liberal institutions is considered where an institution is liberal if each of its theory morphisms gives rise to a free functor. While the equational institution is liberal, many other institutions like the institution of first order predicate logic are not. Therefore, we propose to allow as definition of the desired functor not only a theory morphism with its induced free functor but any other functors and functor definition methods as well. We illustrate this idea first by introducing a general concept of functor constraints and then by describing a constructive ctf definition method in the next subsection.

Definition 4.2 [functor constraints and their satisfaction]

Let $f: Alg(\Sigma, E) \to Alg(\Sigma \cup \Sigma_{new}, E)$ be a functor, let $\iota: \Sigma \to \Sigma \cup \Sigma_{new}$ be the signature inclusion, and let $\sigma: \Sigma \cup \Sigma_{new} \to \Sigma'$ be a signature morphism. Then the pair (f, σ) constitutes a Σ'-functor constraint.

An arbitrary Σ'-algebra A satisfies (f, σ) exactly if its $(\Sigma \cup \Sigma_{new})$-reduct along σ is generated - up to isomorphisms - from its Σ-reduct by the functor f, i.e.

A satisfies (f, σ)

$\Leftrightarrow$

$Alg(\iota)(Alg(\sigma)(A)) \in Alg(\Sigma, E)$ &

$Alg(\sigma)(A) \cong f(Alg(\iota)(Alg(\sigma)(A)))$

where $Alg(\sigma)$ is the forgetful functor corresponding to the signature morphism σ.

(For the strict version replace Alg by EAlg.)

As an example, consider the ctf $g: Alg(ELEM) \to Alg(\Sigma LIST)$ from section 3 and an algebra A of lists over the natural numbers. Then A satisfies the functor constraint $(g, id_{\Sigma LIST})$ if

the list part of A corresponds exactly to the standard lists over the natural numbers. But an algebra A' obtained from A by adding terms like "default-list" or "cons(overflow, nil)" as new elements of sort list does not satisfy the constraint.

Note that like in the data constraints of [GB 83], the signature component $\sigma: \Sigma \cup \Sigma_{new} \to \Sigma'$ in a functor constraint (f, σ) serves as a means for translating such a Σ' constraint by a signature morphism $\sigma': \Sigma' \to \Sigma''$ to a Σ''-constraint $(g, \sigma' \circ \sigma : \Sigma \cup \Sigma_{new} \to \Sigma'')$.
This allows us to derive the satisfaction condition for functor constraints; the proof is analogous to the case of data constraint in [GB 83].

Fact 4.3 [satisfaction condition for functor constraints]
Let (g, σ) and σ' be as above, and let $A'' \in Alg(\Sigma'')$ be an algebra. Then $Alg(\sigma')(A'')$ satisfies (g, σ) iff A'' satisfies $(g, \sigma' \circ \sigma)$. (For the strict version replace Alg by EAlg.)

Proof
A'' satisfies $(g, \sigma' \circ \sigma)$ iff $Alg(\iota)(Alg(\sigma' \circ \sigma))(A'') \in Alg(\Sigma,E)$ and $Alg(\sigma' \circ \sigma)(A'') \equiv g(Alg(\iota)(Alg(\sigma' \circ \sigma)(A')))$ due to Definition 4.2. Since the functor Alg respects composition this is equivalent to $Alg(\sigma \circ \iota)(Alg(\sigma')(A'')) \in Alg(\Sigma,E)$ and $Alg(\sigma)(Alg(\sigma')(A'')) \equiv g(Alg(\sigma \circ \iota)(Alg(\sigma')(A')))$, which in turn is equivalent to $Alg(\sigma')(A'')$ satisfies (g,σ) according to Definition 4.2. (For the strict version replace Alg by EAlg.)

Since the satisfaction condition holds we can extend the types of admissible sentences in a specification (c.f. section 2) by functor constraints and in particular by ctf constraints.

4.3 Parameterized algorithmic definitions

Whereas so far only free functor constraints defined by theory morphisms have been considered in the literature, we now describe a definition method for ctf constraints.

While the implicit definition method for free functors via equational theories or more generally via theory morphisms represents a very high level of abstraction we think that for the more concrete ctfs a more constructive definition method is appropriate. Constructive or algorithmic definition techniques in the framework of abstract data types have been proposed in [Cart 80], [Kl 84], and [Lo 84], but none of them exploits the specific advantages of ctas nor do they support a rigorous parameterized approach. On the other hand, for the definition of ctf domains we would like to allow a broad range of different specification techniques since our ctf concept does not make any specific assumptions about the parameter algebras. Therefore, we distinguish the following two components of our ctf definition method:

(1) definition of the class of parameter algebras
(2) definition of the new carriers and the new operations.

For (1) we can assume an arbitrary loose specification $\langle \Sigma, E \rangle$ denoting the class $Alg(\Sigma, E)$ (resp. $EAlg(\Sigma, E)$ in the strict case) as the domain of the ctf. E may contain just equations, or formulas in first order predicate logic, or constraints, etc. For (2) we will describe a definition method for strict ctfs that can be modified to yield a method for the definition

of ordinary ctfs.

Since our constructive definition method for strict ctfs has been realized in the specification development language ASPIK [BV 85], we will illustrate it by working through the ASPIK specification LIMITED-STACK as shown in figure 4.4; (this figure was produced by the ASPIK support environment SPESY). The dashed lines in figure 4.4 indicate parts that obey certain syntactic conditions to be discussed in the sequel w.r.t. the individual clauses. These conditions guarantee that every ASPIK specification has a well defined ctf semantics.

1. The use clause contains the two specification names ELEM and LIMIT where ELEM contains just the single sort ELEM and LIMIT extends a specification NAT of the natural numbers by a constant limit of sort nat to be used as the maximal size of the stacks. Semantically, the use clause defines the class of parameter algebras for a ctf which in this case is the class of all one-sorted algebras combined with the natural numbers with an additional natural number constant. As pointed out above, ELEM and LIMIT could have been specified by any suitable specification method.

2. The sorts and ops clauses introduce the names and arities of the new sorts and operations.

3. In the spec-body clause, the new carriers and operations are defined separately

3.1 Definition of new carriers:
The operation symbols empty and push in the constructors clause generate the Herbrand universe of terms over these operation symbols. Prefixing the terms with the symbol `*´, it is the set (*empty, *push(empty, e_1), *push(push(empty, e_1), e_2), ...| $e_i \in A_{elem}$) $\cup$ {error-stack} of all terms built from empty, push and elem- objects in the parameter algebra A (without error-$elem_A$); the stack error constant is added separately. Thus, the term property is satisfied. Note that the prefix * is used to distinguish data objects from operation applications. Thus, *push(st,e) is an element of the Herbrand universe, while push(st,e) is a term that may evaluate to *push(st,e) or to error-stack depending on the depth of st. The auxiliary function depth is introduced and defined in order to be used in the definition of the characteristic predicate is-stack in the define-carriers clause. This characteristic predicate restricts the term-generated Herbrand universe to stack terms that do not exceed the given limit, yielding the carrier for sort stack. Note that is-stack must respect subterms so that the restricted carrier is still closed under subterms (subterm property). This semantic property is guaranteed by a simple syntactic condition that requires the explicit subterm test in the definition of is-stack, (see figure 4.4).

3.2 Definition of new operations:
In the define-constructor-ops clause the constructors empty and push are defined so as to satisfy the constructor property, which requires empty := *empty and push(st,e) := *push(st, e) for all stack terms st below the limit. To satisfy this requirement the characteristic predicate's definition can be transformed into definitions of the

```
spec LIMITED-STACK
    /* STANDARD ALGORITHMIC DEFINITION OF A LIMITED-STACK. PUSH ON A FULL  */
    /* STACK, POP OR TOP OF AN EMPTY STACK RESULT IN ERRORS */
  use ELEM
      LIMIT ;
  sorts STACK;
  ops   EMPTY:   --> STACK
        EMPTY?,FULL?: STACK  --> BOOL
        PUSH: STACK ELEM  --> STACK
        POP: STACK  --> STACK
        TOP: STACK  --> ELEM;
spec-body
  constructors EMPTY
               PUSH;
  auxiliaries DEPTH: STACK  --> NAT;
  define-auxiliaries
      DEPTH(ST) = case ST is
                    * EMPTY : ZERO
                    * PUSH(STO,ELO) : SUC(DEPTH(STO))
                  esac;
  define-carriers
      IS-STACK(ST) = case ST is
                       * PUSH(STO,ELO) : if NOT(IS-STACK(STO))
                                         then FALSE
                                         else (DEPTH(STO) LT LIMIT)
                       otherwise TRUE
                     esac;
  define-constructor-ops
      PUSH(STO,ELO) = if (DEPTH(STO) LT LIMIT)
                      then * PUSH(STO,ELO)
                      else ERROR-STACK
      EMPTY = * EMPTY;
  define-ops
      EMPTY?(ST) = case ST is
                     * EMPTY : TRUE
                     * PUSH(STO,ELO) : FALSE
                   esac
      FULL?(ST) = NOT((DEPTH(ST) LT LIMIT))
      POP(ST) = case ST is
                  * EMPTY : ERROR-STACK
                  * PUSH(STO,ELO) : STO
                esac
      TOP(ST) = case ST is
                  * EMPTY : ERROR-ELEM
                  * PUSH(STO,ELO) : ELO
                esac;
endspec
```

Figure 4.4 The ASPIK specification LIMITED-STACK

constructor operations by replacing every true-branch by the *-prefixed constructor term. For every false-branch an arbitrary term may be supplemented. By requiring that it must not contain any *-prefixed constructors it is guaranteed to lie in the restricted carrier. In our example, *push(st, e) is not accepted by is-stack when st is full to the limit. In this case push(st, e) is defined to yield the error element of sort stack. The remaining operations are defined in the define-ops clause using the constructor operations, but again no *-prefixed constructors. As a consequence, these operations are also guaranteed to be closed on the carriers.
Auxiliary operations like depth need not be redefined as new operations, they are closed on the carriers because they, too, must not be defined using the *-prefixed constructors.

All operations may be defined via if-then-else schemes, case-schemes w.r.t. new sorts, and recursion. To explain the semantics of the recursive definitions by a least fixpoint construction we use strict algebras: Their carriers are flat cpos with the error element as

bottom. As a consequence the ctfs defined by our language are strict ctfs. Besides, strict algebras provide a simple built-in error handling mechanism that propagates errors via strict operations.

The syntactic clauses discussed above do not contain an explicit definition of the morphism part of a ctf, because it can be derived from the information already given. For example, if $h : A \to A'$ is a morphism in EAlg(ELEM) the corresponding LIMITED-STACK extension of h maps a stack carrier object $*push(push(empty, a_2), a_1)$ with $a_i \in A_{elem}$ to $*push(push(empty, h(a_2)), h(a_1))$.

In this section we could only indicate the conditions that guarantee that every constructive ctf definition denotes a well defined ctf. [BV 85] contains a denotational semantic definition with a complete set of context sensitive conditions and correctness proofs. The algorithmic approach of [Lo 84] also uses term sets as carriers but in principle there is no syntactic correspondence to the algebra operations like in a cta approach. In [Lo 84] the carriers may be restricted by a characteristic predicate and additionally an algorithmically defined equivalence relation generates congruence classes on the restricted carriers. We think the latter may be better suited for a high level axiomatic approach than for a lower level constructive one.
[Lo 84] does not provide any syntactic conditions for a well-defined semantics. In general, rather complex and difficult proofs may be necessary to ensure that the operations are restrictable to the restricted carriers, that the equivalence operation is reflexive, symmetric, and transitive and that it defines a congruence relation (i.e. it must be compatible with all operations). [Kl 84] allows only primitive recursive definitions which makes a least fixpoint semantics superfluous. If we restrict our ctf definition method to primitive recursive definitions and additionally exclude error elements and constants we obtain a definition method for ordinary ctfs where the same context conditions can be used.

4.4 Integration of axiomatic and constructive techniques

The parameterized constructive ctf definition technique described in the previous subsection can be used as a constraint mechanism according to section 4.2. Since it is independent of the sentences used to specify the parameter class of the ctfs, it can extend many different approaches. In particular, extending axiomatic techniques based on equational logic or first order predicate calculus by ctf constraints, axiomatic and constructive methods presenting different levels of abstraction are integrated in a uniform framework. In such a framework the stepwise development scenario can be realized by moving gradually from high level, purely axiomatic definitions through intermediate forms to completely constructive definitions representing executable prototypes ([BV 85], [BOV 86]).

5. ASPIK and SPESY

ASPIK is a specification development language for the stepwise development of hierarchical, loose specifications, their refinements and implementations. A specification

may contain two types of sentences: arbitrary first order formulas and ctf constraints that are defined in the constructive technique described in section 4.3.

The constructive ctf definition technique is highly supported by the ASPIK support environment SPESY:

- Context-sensitive conditions guarantee that every ctf definition has a well defined semantics. SPESY checks these conditions and additionally exploits them to generate parts of a ctf definition automatically. In figure 4.4 these parts are indicated by dashed lines.
- Being constructive, ctf definitions are amenable to interpretation: SPESY provides an interpreter for terms built from constructively defined operations.

Due to the integration of axiomatic and constructive techniques stepwise specification development as sketched in section 4.4 can be carried out within ASPIK. All development steps can be verified formally, e.g. refining an axiomatically specified subpart A by a constructive ctf definition C requires a proof that C satisfies the properties given in A.
Such proofs are done stepwise along the hierarchical structure as suggested in section 4.1. The corresponding proof tasks are formulated by SPESY, and passed to one of its associated automatic theorem provers ([Karl 84], [Tho 84]). SPESY processes the results of the provers and a reason maintenance component surveys the validity of proved assertions after any manipulations like editing in the specification hierarchy ([BV 85], [BOV 86]).

6. Conclusions

We introduced the notion of canonical term functor as a generalization of canonical term algebra and defined strict versions for both concepts. After proving some properties of ctfs, we addressed their applications in a constraint mechanism, an integration of axiomatic and constructive techniques, and in the specification development language ASPIK and its support environment SPESY.

Appendix: Proof of Fact 4.1

The free functor $free_{PSP}$: Alg(FSP) → Alg (Σ, E) can be defined by sending every A ∈ Alg(FSP) to $T_\Sigma(A)_{/\equiv}$ (c. f. Theorem 7 in [TWW 82]). From $T_\Sigma(A)_{/\equiv}$ we will construct an isomorphic algebra C(A) by selecting a single representative for each equivalence class. Then sending A to C(A) still yields a free functor. By showing that the FΣ-reduct of C(A) is A and that C(A) has the (Σ - FΣ)-term, -subterm, and -constructor properties w. r. t. A we conclude that this functor is also a ctf.

The rest of this proof generalizes the one for the cta case given in [GTW 78]: We define a family $\langle C_n \mid n \geq 0 \rangle$ of subsets of $T_\Sigma(A)$ such that $C = \bigcup \{C_n \mid n \geq 0\}$ is the set of representatives. The sets C_n are defined inductively on the depth of terms so that:

(1) $t \in C_n$ implies depth(t) ≤ n

(2) if $t \in T_\Sigma(A)$ such that the $\equiv$-equivalence class of t has a representative of depth $\leq$ n, then there is a unique representative $t^* \in T_{\Sigma - F\Sigma}(A)$

(3) for any op: $s_1 \ldots s_m \to s \in \Sigma$ with $s \in \Sigma - F\Sigma$ $op(t_1, \ldots t_m) \in C_n$ implies $(t_1, \ldots, t_m) \subseteq C_{n-1}$

In the following the elements of $F\Sigma$ are called old, the elements of $\Sigma - F\Sigma$ are called new. For any old sort $s \in F\Sigma$ we observe that $A_s \cong T_\Sigma(A)_{/\equiv,s}$ since the free functor is persistent. Therefore, the elements of A can be taken as unique representatives for all eqivalence classes in $T_\Sigma(A)_{/\equiv,s}$ with $s \in F\Sigma$. For the set $T_0 := \{op: \to s \in \Sigma - F\Sigma \mid s \in \Sigma - F\Sigma\}$ of constants of a new sort s we choose a subset $C'_0 \subseteq T_0$ such that for each $op \in T_0$ there exists a unique $op^* \in C'_0$ with $op \equiv op^*$ (which obviously can be done). Since the elements of A are treated as constants having depth 0 in the definition of $T_\Sigma(A)$, $C_0 := A \cup C'_0$ satisfies conditons (1) - (3).

Now assume that C_n satisfies (1) - (3). Let T_{n+1} be the set of equivalence classes having a representative of depth n + 1, but no representative of depth $\leq$ n. C_{n+1} is given by C_n together with a single representative $op(t_1^*,\ldots,t_m^*)$ of depth n +1 for each class in T_{n+1} which can be chosen so that $t_i^* \in C_n$ since C_n contains representatiaves for all terms of depth $\leq$ n. Furthermore, both the target sort s of op: $s_1 \ldots s_m \to s$ and op itself must be in $\Sigma - F\Sigma$ since for all old sorts there are representatives of depth 0 due to the peristency of the free functor. Thus C_{n+1} also satisfies (1) -(3).

C is the carrier of our desired algebra C(A). The operations of C(A) are obtained by restricting the operations in $T_\Sigma(A)_{/\equiv}$ to the representatives, i.e. $op_{C(A)}(t_1^*,\ldots,t_m^*) := (op(t_1,\ldots,t_m))^*$. The definition of C immediatly guarantees that the $F\Sigma$-reduct of C(A) has the $(\Sigma - F\Sigma)$ - term and subterm properties w.r.t. A. For $s \in \Sigma - F\Sigma$, op: $s_1 \ldots s_m \to s \in \Sigma - F\Sigma$, and $op(t_1,\ldots,t_m) \in C(A)$ we have $op_{C(A)}(t_1,\ldots,t_m) = (op(t_1,\ldots,t_m))^* = op(t_1,\ldots,t_m)$ since the representatives in C(A) are unique. Thus, C(A) also has the $(\Sigma - F\Sigma)$ - constructor property which completes the proof.

References

[BG 77] Burstall, R.M., Goguen, J.A.: Putting Theories together to Make Specifications. Proc. 5th IJCAI, 1977, pp. 1045-1058.

[BG 80] Burstall, R.M., Goguen, J.A.: The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. LNCS Vol.86, pp. 292-332.

[BG 81] Burstall, R.M., Goguen, J.A.: An informal introduction to specifications using Clear. in: The Correctness problem in Computer Science (Eds. R.S. Boyer, J.S. Moore). Academic Press 1981.

[BOV 86] Beierle, C., Olthoff, W., Voß, A.: Towards a formalization of the software development process. Proc. Software Engineering '86, Southampton, U.K. (Eds. D. Barnes, P. Brown). Peter Peregrinus Ltd., pp. 130-144, 1986.

[BV 85] Beierle, C., Voß, A.: Algebraic Specifications and Implementations in an Integrated Software Development and Verification System. Memo SEKI-85-12, FB Informatik, Univ. Kaiserslautern, (joint SEKI-Memo containing the Ph.D. thesis by Ch. Beierle and the Ph.D. thesis by A. Voß), Dec. 1985.

[Cart 80] Cartwright, R.: A constructive alternative to abstract data type definitions. Proc. 1980 LISP Conf., Stanford University, pp. 46-55, 1980.

[CIP 85] CIP Language Group: The Munich Project CIP, Vol. I: The Wide Spectrum Language CIP-L. LNCS, Vol. 183, 1985.

[EKTWW 80] Ehrig, H., Kreowski, H.-J., Thatcher, J., Wagner, E., Wright, J.: Parameterized data types in algebraic specification languages, Proc. 7th ICALP, LNCS Vol. 85, 1980, pp. 157-168.

[EM 85] Ehrig, H., Mahr, B.: fundamentals of Algebraic Specificiations 1 - Equations and Initial Semantics, Springer Verlag, 1985.

[EWT 82] Ehrig, H., Wagner, E., Thatcher, J.: Algebraic Constraints for specifications and canonical form results. Draft version, TU Berlin, June 1982.

[EWT 83] Ehrig, H., Wagner, E., Thatcher, J.: Algebraic specifications with generating constraints, Proc. ICALP 83, LNCS 154, 1983, pp. 188-202.

[GB 83] Goguen, J.A., Burstall, R.M.: Institutions: Abstract Model Theory for Program Specification. Draft version. SRI International and University of Edinburgh, January 1983, revised 1985.

[GGM 76] Giarratana, V., Gimona, F., Montanari, V.: Observability concepts in abstract data type specifications. 5th MFCS, LNCS 45, 1976, pp. 576-587.

[GTW 78] Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: Current Trends in Programming Methodology, Vol.4, Data Structuring (ed. R. Yeh), Prentice-Hall, 1978, pp. 80-144. also: IBM Research Report RC 6487, 1976.

[HKR 80] Hupbach, U.L., Kaphengst, H., Reichel, H.: Initial algebraic specifications of data types, parameterized data types, and algorithms. VEB Robotron, Zentrum für Forschung und Technik, Dresden, 1980.

[Kam 80] Kamin S.: Final data type specifications: a new data type specification method. 7th POPL, Las Vegas, 1979.

[Karl 84] Karl Mark G Raph: The Markgraf Karl Refutation Procedure. SEKI-Projekt, Memo SEKI-MK-34-01, Univ. Kaiserslautern, 1984.

[Kl 83] Klaeren, H.: Algebraische Spezifikation. Springer Verlag, 1983.

[Kl 84] Klaeren, H.: A constructive method for abstract algebraic software specification. TCS, Vol.30, No. 2, pp. 139 - 204, Aug. 1984.

[Lo 84] Loeckx, J.: Algorithmic specifications: A constructive specification method for abstract data types. Bericht A 84/03, Fachrichtung Informatik, Universität des Saarlandes, April 1984. (to appear in TOPLAS)

[Pad 79] Padawitz, P.: Proving the correctness of implementations by exclusive use of term algebras. Bericht Nr. 79-8, TU Berlin, Fachbereich Informatik, 1979.

[Pad 83] Padawitz, P.: Correctness, Completeness, and Consistency of Equational Data Type Specifications. Dissertation, TU Berlin, Fachbereich Informatik, Bericht Nr. 83-15, 1983.

[SW 82] Sannella, D.T., Wirsing, M.: Implementation of parameterized specifications, Proc. 9th ICALP 1982, LNCS Vol. 140, pp 473 - 488.

[Tho 84] Thomas, Ch.: RRLab - Rewrite Rule Labor. Entwurf, Spezifikation und Implementierung eines Software-werkzeuges zur Erzeugung und Vervollständigung von Rewrite-Rule Systemen. SEKI-Projekt, Memo SEKI-84-01, Univ. Kaiserslautern, FB Informatik, 1984.

[TWW 82] Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data Type Specification: Parameterization and the Power of Specification Techniques. ACM TOPLAS Vol. 4, No. 4, Oct. 1982, pp. 711-732.

[Wa 79] Wand, M.: Final algebra semantics and data type extensions. J. Comp. Syst. Sci. 19, 1979.

[ZLT 82] Zilles, S.N., Lucas, P., Thatcher, J.W.: A Look at Algebraic Specifications. RJ 3568 (41985), IBM Research Division Yorktown Heights, New York, 1982.