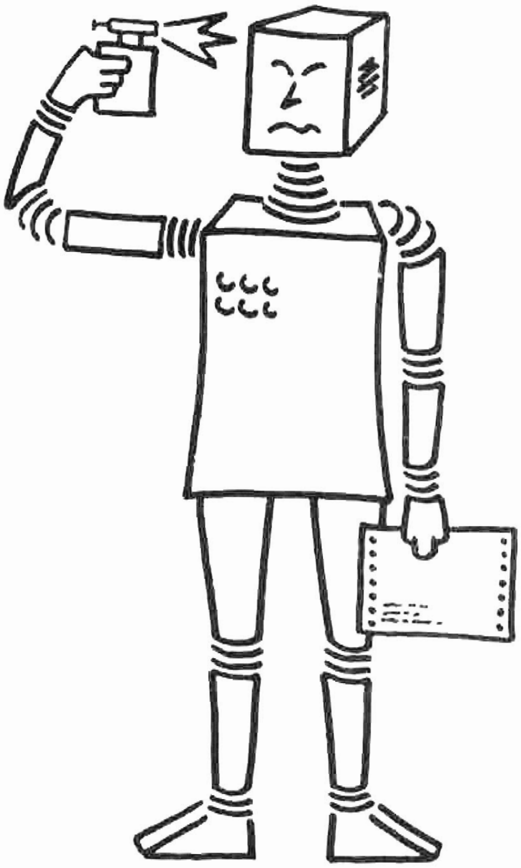
SEKI-PROJEKT

SEKI MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types

Ch. Beierle, A. Voß

Memo SEKI-83-07 May 1983

CANONICAL TERM FUNCTORS AND PARAMETERIZATION-BY-USE FOR THE SPECIFICATIONS OF ABSTRACT DATA TYPES

Ch. Beierle, A. Voß

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern
West Germany

May 1983

Abstract

Algebraic and algorithmic specification methods for abstract data types are combined in the specification language ASPIK covering the whole scope from high level requirements and formal specifications to functional programs. The link between axiomatic and algorithmic specifications is provided by the notion of canonical term functor, a generalization of canonical term algebra. Specifications are structured hierarchically and the new concept of parameterization-by-use offers a flexible means to refine such hierarchies. These features are illustrated by several examples of ASPIK specifications.

Druck: Abteilung Foto-Repro-Druck der Universität Kaiserslautern

Contents

1. Introduction

Abstract data types (ADTs) have turned out to be a useful and powerful concept for the description of data types ([Zi 75], [Gut 75], [GHM 78]). The predominant method for specifying ADTs is the axiomatic specification method (e.g. [GTW 78], [EKTWW 80], [BDPPW 80]). The purpose of abstract specifications is to provide a method for specifying problems and solution approaches without talking about undue details of representation. However, a specification method must not only provide a sufficient level of abstraction but should also be close to the ways of reasoning of those developing specifications: specifications are meant to bridge the gap between informal descriptions and executable programs. Whereas the level of abstraction in axiomatic specifications is very high, it poses problems as well. It may often be easier and more natural to think in terms of models instead of axiom systems, though still on a high level of abstraction. In the axiomatic approach problems of proving consistency and completeness properties arise. In [Kl 80] and [Lo 81] algorithmic specification methods are used that are intended to overcome some of these difficulties. Whereas axiomatic specifications merely describe the desired properties of operations, carriers and operations must be defined explicitly as sets and functions in algorithmic specifications.

Algorithmic specifications therefore constitute a lower level of abstraction than axiomatic specifications. Since both levels are useful they are incorporated in our specification language ASPIK. To close the gap between both levels we use canonical term functors, a generalization of the notion of canonical term algebra ([GTW 78]). Axiomatic specifications are taken to be loose specifications, whereas canonical term functors are used to define so-called Σ-fixes, a constraint mechanism similar to initial or generating constraints in [HKR 80], [BG 80], [SW 82].

ASPIK supports the development of hierarchical specifications. In contrast to other approaches (e.g. [BG 80], [Sa 81], [Eh 81], [Ba 81]), in our parameterization-by-use concept no distinction is made between parameter, parameterized, and non-parameterized specifications. Instead, every specification hierarchically lower than some other specification may be regarded as a formal parameter and replaced by some fitting actual parameter. This represents a new type of abstraction: when writing down specifications one does not have to care about parameterization; only when creating an instantiation the parts regarded as formal w.r.t. that instantiation must be identified.

ASPIK is integrated in a program development and verification system ([RS 80]). It is supported by the interactive INTERLISP system SPESY ([KRST 83]), comprising a guiding input facility, a syntax oriented editor, a file manager and a symbolic interpreter for algorithmic specifications. Various properties of ASPIK specifications are proved by an automatic theorem prover ([BES 81]).

In Sec. 2, the language ASPIK is presented by working through several example specifications in 2.1. An abstract syntax and the main concepts of ASPIK semantics are sketched in 2.2.

2. The specification language ASPIK

ASPIK is intended to be a specification language supporting the software development cycle from high level requirement definitions down to executable programs. It allows the use of loose specifications to formalize requirements. Starting with loose specifications they may be tightened and refined over several steps finally coming up with algorithmic specifications. Algorithmic specifications can be viewed as abstract, but complete, executable programs.

2.1. Specifications in ASPIK

In ASPIK both algorithmic and loose specifications as well as all intermediate forms follow a single specification scheme thus documenting the link between specifications and supporting the step from a specification to a refinement thereof.

A specification consists of a unique name, a specification header and a specification body, the latter being empty in the case of loose specifications. The language constructs as sketched in Fig. 2.0 will be discussed in 2.1.1 and 2.1.2 using the specifications BOOL (Fig. 2.1), NAT (Fig. 2.2), LIMIT (Fig. 2.3), ELEM (Fig. 2.4) and BOUNDED-STACK (Fig. 2.5) for illustration.

2.1.1. Specification header

The header of an ASPIK specification SP describes SP to the outside world. The header

- says which other specifications it is based upon,
- gives the name and arity of the sorts resp. functions it provides as accessible to the outside, and
- states properties met by these functions.

2.1.1.1. Hierachies

ASPIK supports the hierarchical development of specifications. The hierarchical relationship between specifications is expressed by their use-clauses. A specification SP containing

<u>use</u> SP1, ... , SPn

denotes an implicit combination of SP1, ... , SPn enriched by the new sorts and/or operations introduced by SP (see 2.1.1.2). The use-relationship establishes a partial ordering on specifications; its transitive closure must not introduce any cycles, thus guaranteeing its hierarchical nature.

NAT only uses spec BOOL which is used directly or indirectly by

spec <spec-name>		
use ...	hierarchy of specifications	specification header
sorts ... ops ...	signature of specification	
props...	properties of specification	
spec body		specification body
constructors ... auxiliaries ... define auxiliaries ... define carriers...	definition of carriers	
define constructor ops ... private ops ... define ops ...	definition of operations	
endspec		

Figure 2.0: Syntactic structure of ASPIK-specification

```
spec BOOL

   sorts bool

   ops   true, false: ---> bool
         not: bool ---> bool
         and, or: bool bool ---> bool

   props not(true)        = false
             not(false)   = true
          not(and(x,y))   = or(not(x),not(y))


spec body

   constructors  true, false

   define constructor ops
      true  := *true
      false:= *false

   define ops
      not(x):= case x is
                  *true  : false
                  *false: true
               esac

      and(x,y):= case x is
                    *true  : y
                    *false: false
                 esac

      or(x,y)  := case x is
                    *true  : true
                    *false: y
                 esac

endspec
```

Figure 2.1: The specification BOOL

```
spec NAT

   use BOOL

   sorts nat

   ops     0: ---> nat
           succ: nat ---> nat
           add : nat nat ---> nat
           le  : nat nat ---> bool

   props         add(x,y) = add(y,x)
           add(x,add(y,z)) = add(add(x,y),z)
             le(x,add(x,y)) = true

spec body

   constructors  0, succ

   define constructor ops
      0 := * 0
      succ(x) := *succ(x)

   define ops
      add(x,y):= case x is
                   *0        : y
                   *succ(x´): succ(add(x´,y))
                 esac

      le(x,y):= case x is
                 *0        : true
                 *succ(x´): case y is
                              *0        : false
                              *succ(y´): le(x´,y´)
                            esac
                esac

endspec
```

Figure 2.2: The specification NAT

```
spec LIMIT

   use NAT

   ops    limit: ---> nat

   props  le(limit,succ**100(0)) = true
                le(succ(0),limit) = true

endspec
```

Figure 2.3: The specification LIMIT

```
spec ELEM

   use BOOL

   sorts elem

endspec
```

Figure 2.4: The specification ELEM

```
spec BOUNDED-STACK

   use BOOL, NAT, LIMIT, ELEM

   sorts stack
   ops   empty: ---> stack
         push: stack elem --> stack
         pop: stack ---> stack
         top: stack ---> elem
         empty?, full?: stack ---> bool

   props full?(s) = false ==> pop(push(s,e)) = s
         full?(s) = false ==> top(push(s,e)) = e
         empty?(empty) = true
spec body

   constructors empty, push

   auxiliaries depth: stack --> nat

   define auxiliaries
      depth(s):= case s is
                   *empty     : 0
                   *push(s´,e): succ(depth(s´))
                 esac

   define carrier
      is-stack(s):= case s is
                      *empty      : true
                      *push(s´,e): if not(is-stack(s))
                                     then false
                                     else le(succ(depth(s´)),limit)
                    esac

   define constructor ops
      empty=: *empty
      push(s,e):= if le(succ(depth(s)),limit)
                   then *push(s,e)
                   else error-stack

   define ops
      pop(s):= case s is
                 *empty     : error-stack
                 *push(s´,e): s´
               esac

      top(s):= case s is
                 *empty     : error-elem
                 *push(s´,e): e
               esac

      empty?(s):= case s is
                    *empty     : true
                    *push(s´,e): false
                  esac

      full?(s):=  not(le(succ(depth(s)),limit))

endspec
```

Figure 2.5: The specification BOUNDED-STACK

every other specification. This convention makes boolean constants and functions, and thus the if-then-else language construct, available in every ASPIK specification. BOOL is the only specification without a use-clause; it is given in Fig. 2.1. The sorts and operations provided by all specifications used constitute the imported interface.

2.1.1.2. The signature

The sorts and ops clauses introduce new (also referred to as ´public`) sorts and operation names. Together with the imported interface, they constitute the exported interface which is provided to each specification using this one. In NAT, a new sort nat is introduced following the key word sorts, as well as new operations 0, succ, add, and le following the key word ops. Their domains and codomains may contain only sort names from the exported interface.

2.1.1.3. Properties

In this clause, properties of the public functions may be given. In the ASPIK version currently supported by SPESY, the properties may be expressed by equations over the operation names and variables. The equations are implicitly universally quantified and may be conditional. The conditions themselves are conjunctions of one or more equations.
The properties allow for axiomatic ASPIK specifications. An algebra is a model of a specification SP only if it fulfills all of SP´s properties. However, in contrast to the initial approach in e.g. [GTW 78], all model algebras are considered, not just initial ones. Thus, the user may write down a specification stating only some axioms as requirements; later on, the specification may be refined by a tighter one having more axioms or by a specification with an algorithmic definition part, i.e. a non-empty specification body.
In NAT, some properties of add and le are stated, e.g.

commutativity and associativity of the operation add. LIMIT gives properties of limit in relationship to le, 0 and succ.

2.1.1. Specification body

For loose specifications the body is empty; otherwise for the new sorts and operation names carriers and functions must be defined representing a CTA resp. CTF as a model of the properties. For the formal definition, the reader is referred to 2.2.2. First, CTAs are considered.

2.1.2.1. Definition of carriers

The carriers of a CTA are subsets of the Herbrand universes over its signature. In the simplest case, such a subset is generated by a subset Σ' of the functions in the algebra's signature Σ, ($\Sigma' \subseteq \Sigma$) . In other cases the Herbrand universe $H(\Sigma,s)$ of some sort s may not be isomorphic to the intended data type carrier. Therefore, ASPIK provides a mechanism to define a subset $C_s \subseteq H(\Sigma,s)$, such that C_s will be a CTA-carrier for sort s. The definition of CTA-carriers in ASPIK may be done in three steps with the second and third step being optional.

Following the key word <u>constructors</u> a list of function names with codomain s is given for every new sort s introduced in this specification. Since the carrier elements are terms build from the constructors, we will use the <u>'*-notation'</u> to distinguish carrier elements from CTA-function applications: the terms in the Herbrand universe $H(\Sigma,s)$ and thus the elements of the carrier C_s are prefixed by '*'. NAT has constructors 0 and succ for sort nat, BOUNDED-STACK has empty and push for stack.

If the CTA carrier of sort s is a proper subset of the Herbrand universe a characteristic predicate is-s must be defined. In order to facilitate its definition, <u>auxiliaries</u> may be introduced. These are functions on the Hebrand universe as opposed to e.g. the functions defined on the carrier only: in BOUNDED-STACK a function depth is used yielding the number of `push´ occurrences in a given stack term.

The characteristic predicate is-s is defined following the key word define carriers. Simple syntactic restrictions (automatically checked by SPESY) guarantee that only proper CTA carriers can be defined: since all subterms of a carrier element must be carrier elements themselves (subterm property), the definition of is-s must start with a case analysis over the syntactic structure of a term t and for every case is-s(t) may yield true only if is-s´(t´) yields true for every subterm t´ of t and every new sort s´. The carrier of sort stack contains all terms over empty and push such that the depth of the term is not greater than the given limit.

In algorithmic specifications the ASPIK language constructs auxiliaries, define auxiliaries, define carriers are optional. In case they are missing the trivial characteristic predicate yielding constantly true is assumed. Thus, the carrier of sort nat in specification NAT is just the Herbrand universe over 0 and succ, i.e. {*0, *succ(0), *succ(succ(0)), ...}.

Since only new sorts and operations are defined in the specification body, and since imported sorts are referenced via imported operations only, the carrier structure of imported sorts is invisible to the importing specification and the same syntactical scheme may be employed to define CTAs as well as CTFs. If all specifications used are algorithmic, the scheme yields a CTA. It yiedls a CTF, if some specifications used are loose. The specification body of BOUNDED-STACK is an example for a CTF definition because it uses sort elem of the loose specificaion ELEM.

2.1.2.2. Definition of operations

Every new operation must be defined on the respective carriers. This is achieved by dividing the operation definitions into two steps. First, operations are defined corresponding to the constructors that were used to define the carriers, second, the other new operations are defined possibly by means of some private operations (also called hidden functions in e.g. [TWW 82]). As opposed to implicit operation definitions by equations ([GTW 78]), ASPIK provides a definition technique that is similar to the algorithmic specification method used in [Kl 80] and [LO 81]. The left hand side (lhs) of an operation definition consists of the name of the operation to be defined applied to variables of appropriate sorts; the right hand side (rhs) is an operation scheme over the lhs variables. An operation scheme is one of the following:

- public operation term
- if-then-else scheme
- case scheme
- let scheme

A public operation term is a possibly nested application of public functions to appropriate variables.
An if-then-else scheme is the usual conditional where the condition is a public operation term of sort bool. The then- and else-branches are again operation schemes.
A case scheme relies upon the fact that the arguments of public functions are elements from CTA-carriers. Depending on the syntactic structure of the arguments one can give different function values. A simple pattern matching is employed where only the outermost operator is relevant.
In a let scheme, a variable can be introduced as an abbreviation for a term.

Following define constructor ops the operations corresponding to the constructors of every public sort are defined. Two main conditions must be met:

(1) <u>constructor property</u>
Whenever a term $*op(t_1,...,t_n)$ is in the carrier of sort s, $op(t_1,...,t_n)$ must yield $*op(t_1,...,t_n)$.
In BOUNDED-STACK the constructor property enforces

empty := *empty and push(s,e) := *push(s,e)

if the depth of stack s is less than limit.

(2) <u>operations closed on carriers</u>
Whenever a constructor term $*op(t_1,...,t_n)$ is not in the carrier of sort s, $op(t_1,...,t_n)$ must <u>not</u> yield $*op(t_1,...,t_n)$.
In BOUNDED-STACK push(s,e) must not yield *push(s,e) if the depth of s is equal to limit since it must yield an element of the stack carrier.

In ASPIK, the above conditions are guaranteed automatically. The constructor definitions are defined along the characterisitic predicates. For each constructor the corresponding right hand side of the characteristic predicate´s case scheme is transformed into an operation scheme, where, roughly speaking, a true-branch is substituted by the corresponding carrier term $*op(t_1,...,t_n)$, and a false-branch has to be filled in with some user given operation scheme. Thus, SPESY generates parts of the constructor definitions automatically. In NAT, is-nat is assumed to be constantly true; SPESY generates the complete operation definitions for both 0 and succ. in BOUNDED-STACK, the definition for empty is generated as well, and in the push definition only the else-branch must be filled in by the user. It cannot be automatically generated, since one might want to define a ´forgetful` bounded stack by setting push(s,e) := s iff s is already full.

<u>Private operations</u> are not accessible from the outside.

Therefore, they are not declared in the specification header but in its body. All auxiliaries are automatically available as private operations, the difference being that auxiliaries are defined on the Herbrand universe while private operations operate on the carrier sets. Private operations as well as all public operations other than constructors are defined algorithmically following the key word define ops. An example for the use of private operations is given in BOUNDED-STACK, where operation full? is defined in terms of depth.

Closedness of all private and public operations is again enforced by simple syntactic restrictions. Because of the hierarchical relationships of the different types of operations their closedness w.r.t. the carrier sets may finally be reduced to the closedness of the constructors operations: Except for the automatically generated parts of the constructor operation definitions, the right hand sides of the defining operation schemes may not contain *-prefixed terms from the Herbrand universe explicitly; instead, function applications of the constructor operations evaluating to carrier elements must be used. Termination of all algorithmically defined operations remains to be shown, i.e. auxiliaries, characteristic predicates, constructor operations, private and public operations. For proving termination an automatic theorem prover will be used ([RES 81]).

2.1.3. Parameterization-by-use

ASPIK provides a parameterization concept that was designed according to the following principles:

- Both formal and actual parameters are specifications.
- Loose and algorithmic specifications may be used as formal as well as actual parameters.
- In a specification itself no parameters are declared; only when instantiating a specification the formal parameters are indicated.

```
spec TWENTY

   use NAT

   public ops twenty: ---> nat

spec body

   define ops
      twenty:= succ**20(0)

endspec
```

Figure 2.6: The specification TWENTY

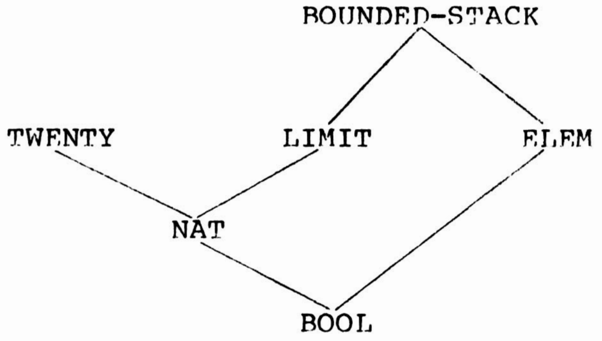


Figure 2.7: The hierarchy of specifications with BOUNDED-STACK

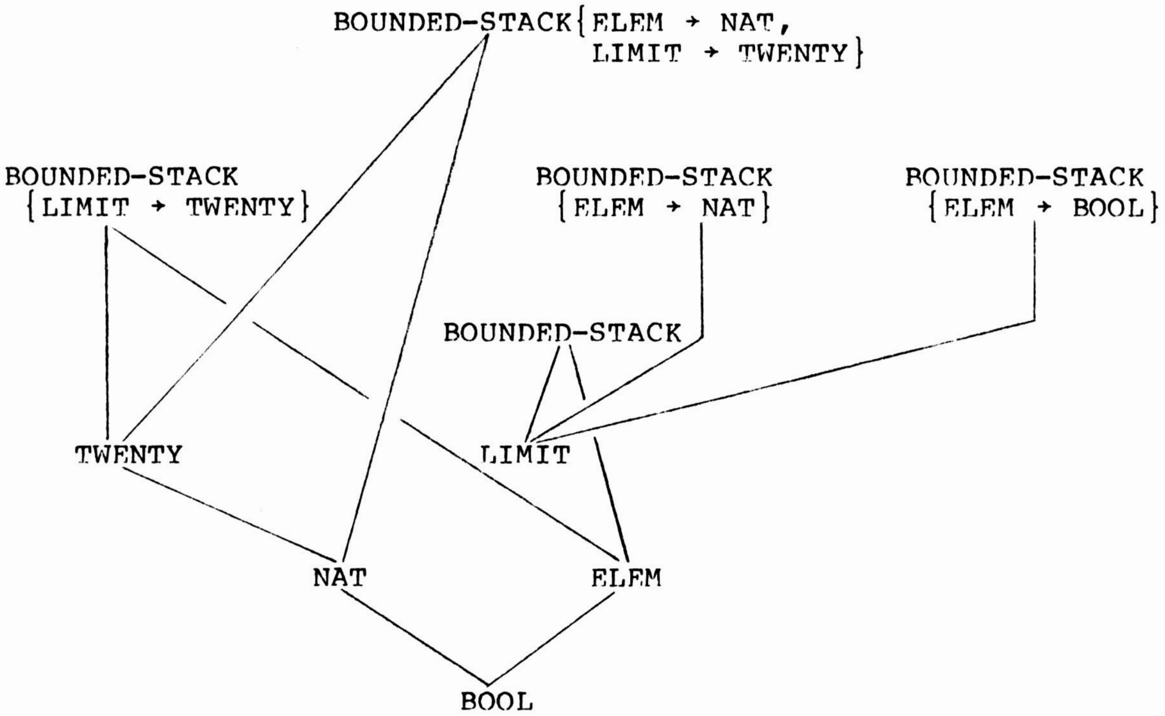


Figure 2.8: The hierarchy with some instances of BOUNDED-STACK

- All specifications used by a specification may serve as formal parameters.

This yields a highly flexible system of parameter specification and instantiation. Its formal treatment is discussed in 2.2.2 whereas in this section some illustrating examples are given.

The use-clauses of all specifications define a hierarchy of specifications that can be represented by an acyclic graph. They induce a partial order on specifications, with minimum element BOOL. Since every used specification is included in the specification itself as a subspecification the idea of instantiation amounts to consider some used specifications as formal parameters and to replace (or actualize) them by some other specifications regarded as actual parameters. Such an instantiation must be compatible with the hierarchical structure of the specifications: every specification used by a formal parameter must be used by the corresponding actual parameter (see also 2.2.2).

The hierarchy of Fig. 2.7 is generated by the specifications of Fig. 2.1. - 2.6. An instantiation of BOUNDED-STACK could be produced by actualizing the used specification LIMIT - as formal parameter - by specification TWENTY as actual parameter, thus restricting the maximal depth of a stack to 20. The result of this instantiation process is denoted by

(1) BOUNDED-STACK{LIMIT $\xrightarrow{\sigma}$ TWENTY

where σ is part of a specification morphism mapping new sorts and operations of LIMIT to sorts and operations in TWENTY's exported interface:

σ: ops limit $\rightarrow$ twenty

The complete specification morphism is obtained by extending the given part by mapping all used specifications that are not parameters identically to themselves. σ must be compatible with the arity of the operations and all properties of the formal parameter translated by σ must be met by the actual parameter. While the first condition is easily checked the second one has to be proved. This could be done by an automatic theorem prover.

The specification denoted by (1) originates from BOUNDED-STACK by substituting the specification name TWENTY for LIMIT in the use clause and the operation name twenty for limit in the specification body.

Some more examples for instantiations are:

(2) BOUNDED-STACK {ELEM → NAT
    sorts elem → nat}

(3) BOUNDED-STACK {ELEM →BOOL
    sorts elem → bool}

(4) BOUNDED-STACK {ELEM → NAT
    sorts elem → nat;
    LIMIT → TWENTY
    ops limit → twenty}

Considering the hierarchy extented by (1) - (4) in Fig. 2.8, new instantiations are now possible, e.g.

(5) BOUNDED-STACK $\overset{f}{\text{\{LIMIT → TWENTY\}}}$ $\overset{g}{\text{\{ELEM → NAT\}}}$

(6) BOUNDED-STACK $\overset{g}{\text{\{ELEM →NAT\}}}$ $\overset{f}{\text{\{LIMIT → TWENTY\}}}$

where: f: ops limit → twenty
g: sorts elem → nat

It should not matter in which sequence independent parameters are actualized or whether they are actualized in parallel, thus both the specifications denoted by (5) and (6) should be identified with (4). In ASPIK, this is indeed the case. Instantiations are denoted by *specification terms*. ASPIK semantics refer every specification term to a specification and to a node in the specification hierarchy, e.g. the different terms (4) - (6) denote the same specification under ASPIK semantics. Thus, the use-clause of a specification may not only contain simple specification names but also specification terms such as (2) - (4). More sophisticated examples of parameterization-by-use exhibiting these and other aspects can be found in [BGV 83].

2.2. Outline of a formal definition of ASPIK

2.2.1. Abstract ASPIK syntax

Note: The syntax is given in a BNF-like notation.
Terminal symbols are underlined.

```
spec:: spec specid
       [comment text]
       [use spec-term...]
       [sorts sortid...]
       [ops op-header...]
       [props property...]
       [spec-body  body]
       endspec

spec-term:: specid [(spec-map...)]...

spec-map::  spec-term → spec-term
                 sig-map
sig-map:    [sorts (sortid → sortid)...]
            [ops   (opid → opid)...]
```

```
opheader::   opid ...:[sortid...] → sortid

property::   equation|inequation|cond-equation

equation::   term=term

inequation:: term=/=term

cond-equation:: equation[& equation]...=>
                              (equation|inequation)

body: [carrier-part]
      op-part

carrier-part:: constructors  opid...
               [[auxiliaries op-header...
                [define auxiliaries  op-body...]]
                define carriers  op-body...]

op-part:: [define constructors op-body...]
          [private ops  op-header...]
          [define ops  op-body...]

op-body:: opid[( varid...)] := op-scheme

op-scheme:: term|if-scheme|case-scheme|let-scheme

term:: varid|opid[(  term...)]

let-scheme:: let (varid = term)...
             in  op-scheme

case-scheme:: case varid is
                (opid[( varid..)]: op-scheme)...
                [otherwise op-scheme]
              esac
```

```
if-scheme:: if term
            then  op-scheme
            [elsif term
            then  op-scheme]...
            else  op-scheme
```

2.2.2 Semantics

The objective of the algebraic specification method is the definition of abstract data types. Data types are usually regarded as algebras or classes of algebras. ASPIK definitions denote hierarchies of specifications as induced by the use relationship. Such hierarchical specifications have classes of algebras as models, constituting the abstract data types.

2.2.2.1 SPEC - the category of specifications

A signature Σ is a set of sorts S together with a set of operators each operator having an arity in $S^{*}x$ S. A signature morphism σ is a translation of sorts to sorts and operators to operators such that the arities are preserved. SIG is the category of signatures, $S\vec{I}G$ the subcategory of SIG with only signature inclusions as morphisms.

A Σ-algebra is an S-indexed family of sets together with an $S^{*}xS$-indexed family of functions. A Σ-algebra morphism is an S-indexed family of functions f_s such that the functions are preserved. ALG(Σ) is the category of Σ-algebras. A (conditional) Σ-equation e is a pair of two Σ-terms over an S-indexed family of variables (together with a list of Σ-equations as condition). A Σ-algebra satisfies e iff it satisfies every ground instance of e.

Given a Σ'-algebra A' and a signature morphism $\sigma: \Sigma \rightarrow \Sigma'$ the Σ-restriction of A' along σ is denoted by $A^{\sigma}|_{\Sigma}$ or just $A|_{\Sigma}$ if σ is understood. It is the Σ-algebra A defined by $A_s = A'_{\sigma(s)}$ and $op_A = \sigma(op)_{A'}$.

With these preliminaries we can now give the formal definition of a canonical term functor.

Definition

Let Σ, Σ' be signatures with $\Sigma \subseteq \Sigma'$, $C(\Sigma)$ a subcategory of $ALG(\Sigma)$. A functor

$$g: C(\Sigma) \to ALG(\Sigma')$$

is a canonical term functor (CTF) iff

$\forall A \in C(\Sigma)$ conditions (i) to (iv) hold:

(i) $g(A)|_{\Sigma} = A$ (persistency)

(ii) $g(A)_s \subseteq T_{\Sigma'-\Sigma}(A)$ for all $s \in \Sigma'-\Sigma$

(term property)

$\forall$ op $\in \Sigma'-\Sigma$. op: $s_1 \dots s_n \to s$ (where $s \in \Sigma'-\Sigma$):

(iii) $op(t_1, \dots, t_n) \in g(A)_s$

$\Rightarrow [\forall i \in \{1,\dots n\} \,.\, s_i \in \Sigma'-\Sigma \Rightarrow t_i \in g(A)_{si}]$

(subterm property)

(iv) $op(t_1, \dots, t_n) \in g(A)_s$

$\Rightarrow op_{g(A)}(t_1, \dots, t_n) = op(t_1, \dots, t_n)$

(constructor property)

leer Signatur

The concept of a CTF is a generalization of the notion of canonical term algebra (CTA) as introduced in [GTW 78]. A Σ-algebra A is a CTA iff the constant functor g_A: $ALG(\phi) \to ALG(\Sigma)$ yielding A is a CTF. Some other useful facts are also easy to prove: every CTF is strongly persistent, the composition of CTFs yields again a CTF, and the application of a CTF to a CTA yields a CTA.

The body of an ASPIK specification can be evaluated to a CTF. Its source is given by the combination of the use-clause entries yielding $C(\Sigma)$ in the definition above and its target is given by that combination enlarged by the public sorts and operations. Just like the properties in the specification header restrict the class of model algebras the specification body evaluated to a CTF represents a restriction as well.

Definition

Let Σ', Σ'', Σ be signatures, $\Sigma' \subseteq \Sigma''$.

A Σ-fix f is a pair

$$(g\colon C(\Sigma') \to ALG(\Sigma''), \sigma\colon \Sigma'' \to \Sigma)$$

consisting of a canonical term functor g and signature morphism σ.

A Σ-fix is similar to initial or generating constraints ([HKR 80], [BG 80], [SW 82]). Let Σ, Σ', Σ'' be as above, A a Σ-algebra, A" its Σ''-restriction. Then A satisfies a Σ-fix (g,σ) if

- g is applicable to the Σ'-restriction of A", i.e. $A''|_{\Sigma'}$ is in the domain of g
- the carriers of sorts in $\Sigma''-\Sigma'$ and the corresponding operations in A" are "defined just as" in the algebra obtained by applying g to the Σ'-restriction of A".

Since the second condition is guaranteed for the Σ'-part of A" by the persistency of g, the "defined just as" is captured by an isomorphism between the two algebras:

Definition

A Σ-algebra A satisfies a Σ-fix

$$f = (g\colon C(\Sigma') \to ALG(\Sigma''), \sigma\colon \Sigma'' \to \Sigma)$$

iff

$$(A|_{\Sigma''})|_{\Sigma'} \in C(\Sigma') \quad \text{and}$$
$$g((A|_{\Sigma''})|_{\Sigma'}) \cong A|_{\Sigma''}$$

For a set of Σ-equations E and a set of Σ-fixes F $ALG(\Sigma,E,F)$ is the subcategory of $ALG(\Sigma)$ with algebras satisfying E and F.

Like data constraints in Clear [BG 80] a Σ-fix $f = (g,\sigma)$ may be translated by a signature morphism $\sigma' : \Sigma \to \Sigma'$ yielding the Σ'-fix $(g,\sigma' \circ \sigma)$. A specification representation $SP=(\Sigma,E,F)$ with Σ-equations E and Σ-fixes F represents the specification $SP' = (\Sigma,E',F')$ where (E',F') is the closure of (E,F), i.e. the sets of Σ-equations resp. Σ-fixes satisfied by all algebras in $ALG(\Sigma,E,F)$. A specification morphism $\sigma\colon(\Sigma,E,F) \to (\Sigma',E',F')$ is a signature morphism $\sigma:\Sigma \to \Sigma'$ such that $\sigma(E) \subseteq E'$ and $\sigma(F) \subseteq F'$. SPEC is the category of specifications, SP$\vec{E}$C the subcategory of SPEC with only inclusions as morphisms.

2.2.2.2 Specification hierarchies

In the category SPEC the hierarchical structure of specifications is not represented. In the semantics of ASPIK this is achieved by special diagrams in SP$\vec{E}$C, called specification hierarchies. Let AO be a well founded irreflexive partial order with a minimal element such that every element has only finitely many predecessors. Let AO also denote the induced path category. A specification hierarchy is a functor H: AO $\to$ SP$\vec{E}$C where the minimal element is mapped to the specification BOOL. Since there is at most one morphism between any two objects in SP$\vec{E}$C, H is determined by its object part: if there is a path from A to B in AO then there must be an inclusion $H(A) \to H(B)$, thus reflecting precisely a use-relationship between the two specifications.

A sequence of ASPIK-specifications yields a specification hierarchy. Having already evaluated the first n-1 specifications to a hierarchy H, the nth specification, say <u>spec</u> SP, yields the hierarchy H' generated from H in two steps:

(1) AO is enlarged by the new element SP where some already existing element A is smaller than SP iff SP uses A. As sketched in 2.2.2.1, <u>spec</u> SP is evaluated to a triple $(\Sigma \cup \Sigma', E \cup E', F \cup F')$ where $(\Sigma,E,F,)$ is the union of all specifications used. Σ' contains SP's public sorts and operations (prefixed by 'SP' in order to avoid unwanted name clashes), E' is the set of SP's properties and F' contains the CTF SP's body is evaluated to. This specification representation yields the specification that is the label of SP under H'.

(2) In the second step all specification terms in normal form involving SP are considered. A normal form term must not contain trivial parameter replacements like id: SP $\to$ SP nor any sequential replacements, e.g. (4) in section 2.1.3 is in

normal form, but (5) and (6) are not. Every specification term can be transformed into an equivalent normal form term ([BV 83]). For every normal form term a new node is introduced that is labeled with the instantiation object of the specificaion term. By repeating this process inductively a <u>closed</u> hierarchy H' is generated such that every specification term can be mapped to a node n with $H'(n)$ being the corresponding instantiation object.

In [BV 83] hierarchies are studied in more detail. The results reported there are applicable to specification hierarchies as well.

Acknowledgement

We would like to thank our colleagues from the SEKI-Projekt at the University of Kaiserslautern for interesting discussions, H. Klaeren for helpful criticism on an earlier version of this paper, and D. Kilgore for excellent typing. This research is supported by the Bundesministerium für Forschung und Technologie under contract IT.8302363.

<u>References</u>

[Ba 81] Bauer, F.L. et al.: Report on a wide spectrum language for program specification and development. TU München, Inst.f.Informatik, Report TUM-I8104, May 1981.

[BGV 83] Beierle, Ch., Gerlach, M., Voß, A.: Parameterization without parameters - the history of a hierarchy of specifications. SEKI-Projekt, Univ. Kaiserslautern, FB Informatik (in preparation).

[BV 83] Beierle, Ch., Voß, A.: Parameterization-by-use for hierarchically structured objects. SEKI-Projekt, Univ. Kaiserslautern, FB Informatik, May 1983.

[BES 81] Bläsius, K., Eisinger, N., Siekmann, J., Smolka, G., Herold, A., Walther, C.: The Markgraf Carl Refutation Procedure, Proc. 7th IJCAI, 1981.

[BDPPW 80] Broy, M., Dosch, W., Partsch, H., Pepper, P., Wirsing, M.: On hierarchies of abstract data types, TU München, Inst. für Informatik, TUM-I8007, May 1980.

[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.

[Eh 82] Ehrich, H.-D.: On the theory of specification, Implementation and Parametrization of Abstract Data Types. JACM Vol. 29, No. 1, Jan. 1982, pp. 206-227.

[Eh 81] Ehrig, H.: Algebraic Theory of Parameterized Specification with Requirements, Proc. 6th CAAP, Genova, 1981.

[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.

[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.

[Gut 75] Guttag, J.V.: The specification and application to programming of abstract data types. Ph.D. thesis, Univ. of Toronto, 1975.

[GHM 78] Guttag,J., Horowitz, E., Musser, D.: Abstract Data Types and Software Validation, CACM, Vol. 21, No. 12 (Dec. 1978), pp. 1048-1064.

[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.

[Kl 80] Klaeren, H.: A simple class of algorithmic specifications of abstract software modules. Proc. 9th MFCS 1980, LNCS Vol. 88, pp 362 -374.

[KRST 83] Kücke, R., Rome,E., Sommer, W., Thomas, C.: Das SPEC-System (SPESY): Benutzerhandbuch, SEKI-Projekt, Univ. Kaiserslautern, FB Informatik, 1983.

[Lo 81] Loeckx, J.: Algorithmic specification of abstract data types. Proc. 8th ICALP, LNCS 115, July 1981, pp. 129-147.

[RS 80] Raulefs, P., Siekmann, J.: Programmverifikation. Darstellung des Forschungsvorhabens, Univ. Bonn/Karlsruhe, Inst.f. Informatik, Aug. 1980.

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

[Sa 81] Sannella, D.T.: A new semantics for Clear. Report CSR -79-81, Dept. of Computer Science, Univ. of Edinburgh, 1981.

[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.

[Zi 75] Zilles, S. N.: Abstract specifications for data types, IBM Research Laboratory, San Jose, California, 1975.