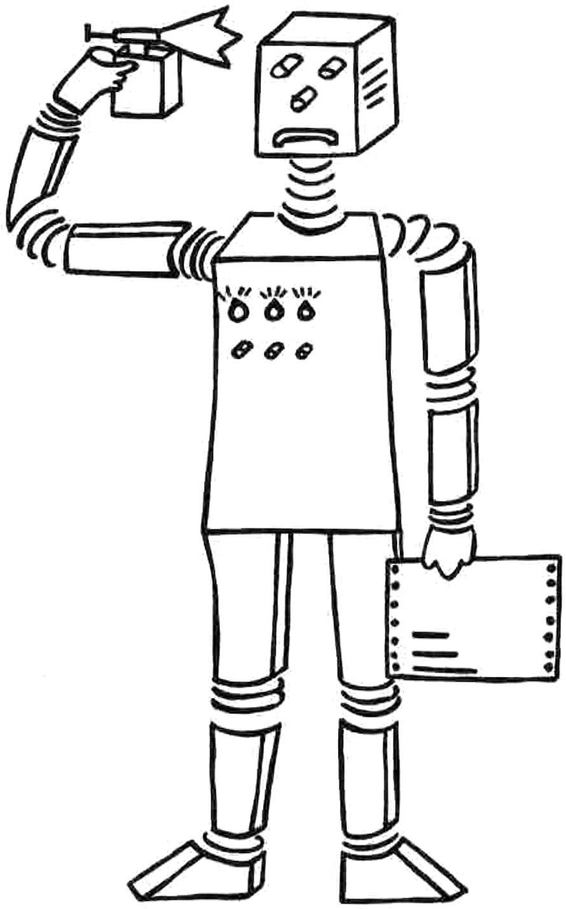
SEKI-REPORT

Artificial
Intelligence
Laboratories

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



Stepwise software development:
Combining axiomatic and algorithmic
approaches in algebraic specifications

Christoph Beierle, Angelika Voß

August 1986 SEKI-REPORT SR-86-15

Stepwise software development: Combining axiomatic and algorithmic approaches in algebraic specifications

Christoph Beierle, Angelika Voß
Fachbereich Informatik
Universität Kaiserslautern
Postfach 30 49
6750 Kaiserslautern
West Germany
UUCP: ... mcvax!unido!uklirb!beierle

Abstract

Much of the software development activity can be carried out using formal specifications that have a precise and well defined semantics, making it possible to formally verify the correctness of the development steps. In order to support this claim we present an algebraic specification method that provides both axiomatic and algorithmic techniques and illustrate it by working through an example development. Our method is realized in the specification development language ASPIK, which is a core component of an integrated software development and verification system. The semantics of ASPIK is based on the new notion of canonical term functor which generalizes the notion of canonical term algebra, and we show how this notion allows a uniform integration of axiomatic and algorithmic approaches by using the concept of algorithmic constraints.

Keywords

Algebraic specification, algorithmic constraint, axiomatic and algorithmic definitions, canonical term functor, software development, verification,

Contents

1. Introduction

Many software development models view the software development process to consist of a sequence of succesive phases where each subsequent phase refines (models, implements, etc.) the result of the previous phase; for a survey see e. g. [Hün 80]. In most of the models the first phases usually deal with informal descriptions consisting of e.g. texts in natural language and graphical representations. Often, the first phase dealing with formalized objects and having a rigorous semantics is the coding phase; i.e. the first formalized problem description is the problem solution itself, namely the program. Obviously, this makes it impossible to check in some mathematically precise way the consistency of the problem solution with the preceding problem descriptions or specifications since the latter must be given in a formal language as well.

During the last decade a lot of work in formal semantics of programming languages has been done and the development of rigorous specification methods laid the basis for a study of the relationship between a program and its specification. In this paper we argue that much of the software development activity can be carried out using formal specifications that have a well defined semantics, making it possible to verify formally the correctness of the development steps. In particular, we show how the paradigms of "stepwise-refinement" and "verify-while-develop" are realized in this approach.

Our work is based on the algebraic specification method which was first suggested by Zilles ([Zil 74]), Guttag ([Gut 75]), and the ADJ group ([GTW 78]). We use a so-called loose approach where each specification has many non-isomorphic models in general; examples of loose approaches are the canon specifications of [HKR 80], or Clear [BG 77, 80], CIP-L [CIP 85] and Look [ZLT 82]. On the other hand we combine loose axiomatic specifications with an algorithmic definition technique as suggested e.g. by Cartwright [Cart 80], Klaeren [Kl 80, 84], and Loeckx [Lo 81, 84] and integrate both in a uniform way.

Our approach was developed for the Integrated Software Development and Verification (ISDV) project and is employed in SPESY, a prototype system resulting from the ISDV project ([BGGORV 83], [BV 85], [BOV 86]).

This paper is organized as follows:

In section 2 we state some algebraic preliminaries used in the sequel and fix our notation. In section 3 we describe the development scenario of our approach, which is illustrated in section 4 by working through an example. In section 5 we show how this scenario is realized in the ISDV system by our specification development language ASPIK and its suppport environment SPESY. In section 6 the formal semantics based on the new notion of canonical term functor is outlined, and section 7 contains a summary and an outlook.

2. Preliminaries: Algebraic Specifications

As suggested by Morris [Mor 73] "types are not sets", but a collection of data together with operations that can be performed on these data, and according to Liskov and Zilles [LZ 74] "an abstract data type defines a class of abstract objects which is completely characterized by the operations available in these objects". This led to the questions of how to specify the behaviour of the operations without referring in any way to the representation of the objects. The algebraic approach to abstract data type specifications that has been accepted as the most promising one was first carried out by Zilles ([Zil 74]), Guttag ([Gut 75]) and the ADJ group ([GTWW 75 a]).

The formalization given by the ADJ group defines the notions of "signature" as name space, "algebra" as representing a concrete data type, and "specification" as defining an abstract data type by a class of isomorphic algebras.

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 $op: s_1 \dots s_n \rightarrow s$ means that op has argument sorts $s_1 \dots s_n$ and target sort s.

A Σ-algebra $A = \langle \{ A_s \mid s \in S \}, \{op_A: A_{s1} \times \dots \times A_{sn} \rightarrow A_s \mid op: s_1 \dots s_n \rightarrow s \in Op \} \rangle$ 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 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. 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. Only equations are considered in [GTW 78], positive conditional equations in [TWW 78], and universal Horn sentences in [EKTWW 80], whereas in the loose approach of [CIP 85] arbritrary first order formulas are allowed. Other loose approaches are e.g. [BG 77, 80, 81], [HKR 80], [SW 82], [ZLT 82], and [EWT 82].

3. Software development using loose algebraic specifications

There are several reasons in favour of a loose approach as a basis for formalized software development. Firstly, in the early phases of design and specification one would like to have a rather rich language with enough expressive power so that the properties and characteristics of the operations and functions under consideration can be specified directly without having to take into account any particular restrictions on the types of admissible sentences. Secondly, from the controversy about whether the initial or the terminal approach is "best" (c. f. [MG 85]), it seems apparent that both should be complemented by a technique without such a universial initial or terminal constraint. Thirdly, whereas in a fixed approach a complete set of axioms is required right from the beginning (c. f. the sufficient completeness problem in e.g. [EKP 78], [Pad 83]), in the loose approach a specification can be gradually refined by making more design decisions and thereby restricting the class of models.

This can be done by elaborating the signature and adding sentences. More formally, a refinement $\varphi: SP_1 \rightarrow SP_2$ between the specifications SP_1 and SP_2 is a type preserving translation $\varphi: \Sigma_1 \rightarrow \Sigma_2$ of the signature of SP_1 to that of SP_2 which respects the sentences E_1 of SP_1 in the sense that for every p

$\in E_1$ the translation $\varphi(p)$ is contained in the sentences E_2 of SP_2. Our model of specification development via refinements is sketched in figure 3.1. It proceeds from an abstract specification which has a small signature, few sentences but many models over intermediate specifications with a more elaborated signature, more sentences and fewer models down to a rather concrete specification which has a completely elaborated signature and a sufficient set of sentences in order to obtain only isomorphic models. Every model of this concrete specification may be used as a prototype.

In order to support such a specification development process which gradually proceeds from the abstract to the concrete level, there should be types of sentences supporting a very high level of abstraction, like axiomatic predicate calculus formulas. On the other hand there should also be more concrete types of sentences like constructive definitions and algorithms. Moreover, both types of sentences should be arbitrarily combinable such that the intermediate specifications may have both axiomatic and algorithmic sentences.

While there are fixed approaches providing constructive and algorithmic techniques ([Kl 84], [Lo 84]) and loose approaches providing axiomatic techniques ([BG 80], [CIP 85]) but none integrating both, we propose a loose approach providing both axiomatic and algorithmic techniques as required above. The key ideas are:

1. Parameterize a constructive technique so that loose axiomatic specifications may be used as formal parameter descriptions.
2. Define the semantics of a parameterized constructive description as a function from parameter algebras to constructively extended algebras.
3. Consider parameterized constructive descriptions as algorithmic sentences.
4. Allow both axiomatic and algorithmic sentences to occur in the set E of a specification $SP = \langle \Sigma, E \rangle$.

4. How to build a tower

Before going into more formal details, we will illustrate our approach by working through an example development.

The problem of building under certain constraints a tower as high as possible is described informally in section 4.1 and stepwise formalized and

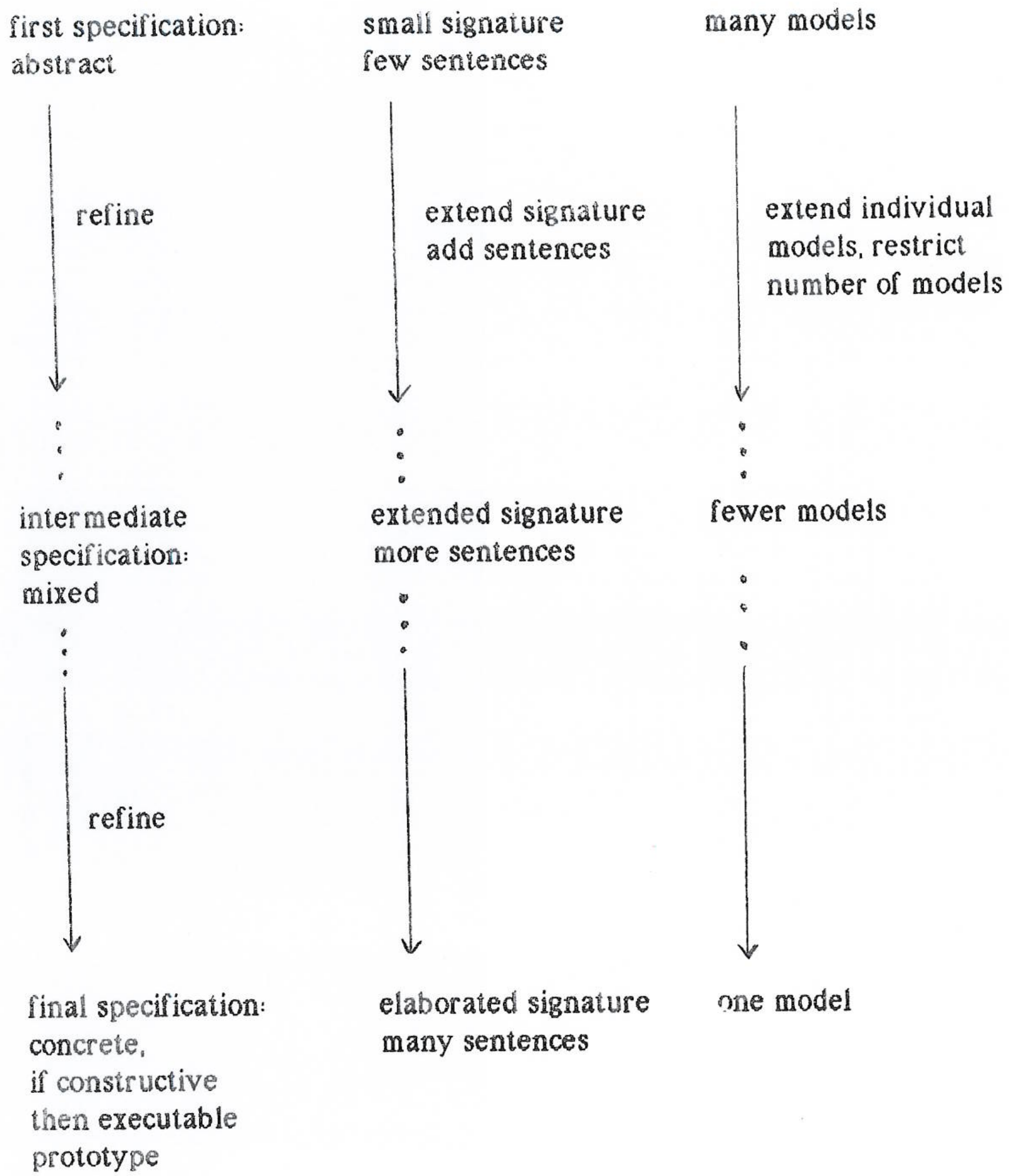


Figure 3.1: The refinement process supported by loose approaches

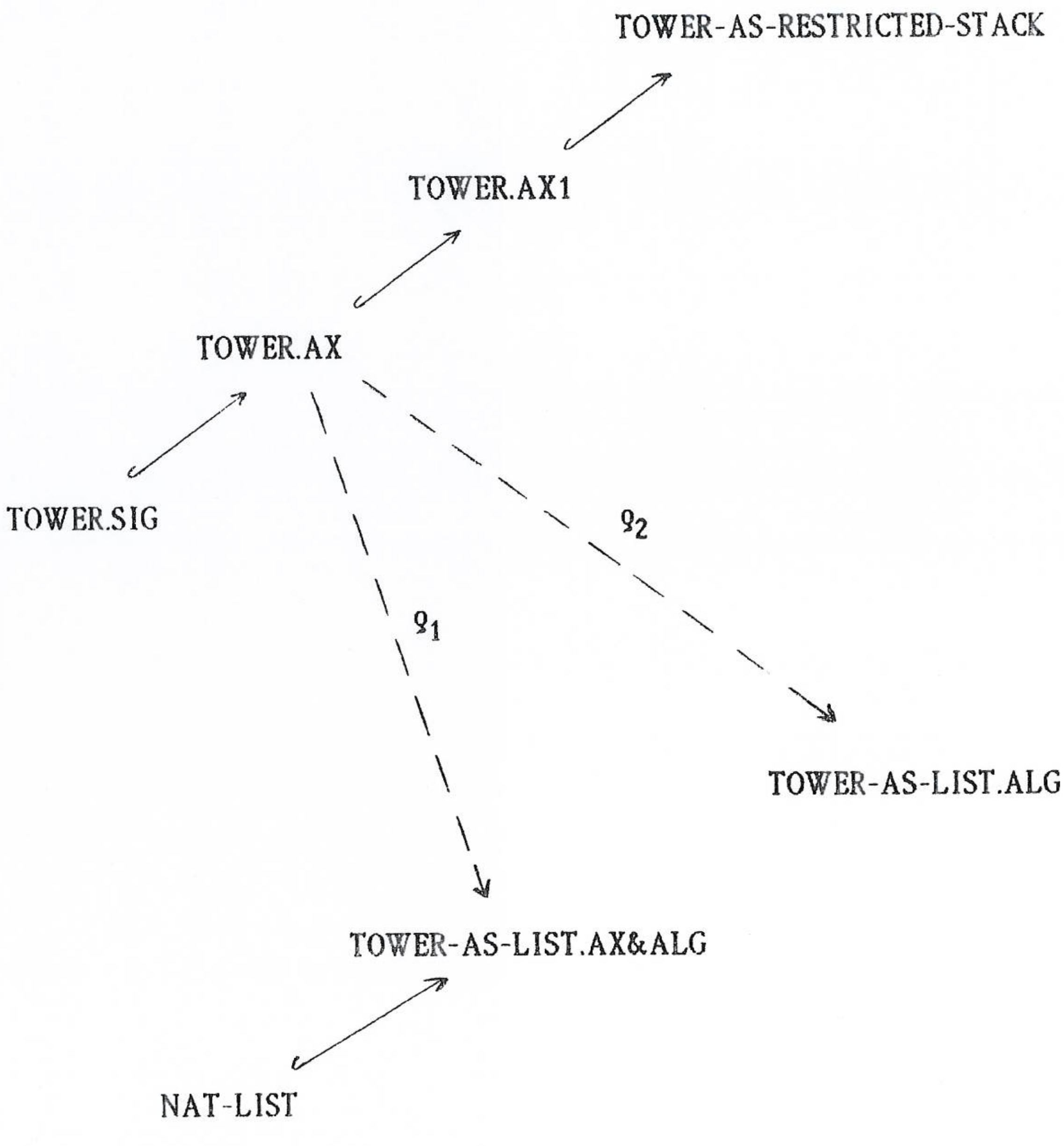


Figure 4.1: The specifications for the highest-tower problem and their refinement relations

refined in the the following subsections. An overview of the resulting specifications and their refinement relations is given in figure 4.1. In section 4.2 we develop the specification TOWER.SIG of the signature in which to express our problem. In section 4.3 we add first order predicate formulas in order to obtain the axiomatic specification TOWER.AX. Using lists of natural numbers as specified in NAT-LIST, the axiomatic description is algorithmically refined in two steps in sections 4.4 and 4.5, respectively, yielding the specifications TOWER-AS-LIST.AX&ALG and TOWER-AS-LIST.ALG. For each refinement step we give the verification conditions that have to be proven in order to ensure the correctness of the refinements. In section 4.6 we discuss alternative development steps where e.g. TOWER.AX is refined successively first to an axiomatic specification TOWER.AX1 and finally to an algorithmic specification TOWER-AS-RESTRICTED-STACK.

4.1. Informal problem description

In a room some blocks are lying around. They shall be used to build as high a tower as possible. Naturally, the size of the tower is limited by the ceiling. All blocks are cubes and, for simplification, we may assume the lengths of the blocks and the height of the room to be integer valued. The solution must fix neither the size or number of the blocks nor the height of the room.

4.2 The signature specification

In order to formalize the problem we may abstract from the notions of room and block. All we need is a natural number constant ceiling for the height of the room and a function #blocks (n) giving the number of blocks of length n for each natural number n.

A tower is either flat for else constructed by successively putting a block on top of another tower. It may be destructed similarly by repeatedly removing the block on the top. Since the only relevant property of blocks is their length we may represent the blocks of the tower by their lengths. Thus, e.g. put-on is a function taking a tower and a natural number and yielding again a tower: put-on: tower nat → nat.

Every tower t has a height and uses a certain number #used (t,n) of blocks of length n. With these two predicates we may characterize admissible towers. Finally we are asked to build the highest admissible tower.

We will not start from scratch but assume a specification BOOL of the booleans (for the predicates) and a specification NAT of the natural numbers for measuring to be already available. Letting BOOL ∪ NAT denote their componentwise union we may specify our vocabulary as follows:

```
spec    TOWER.SIG = BOOL ∪ NAT ∪
sorts   tower
ops     #blocks:        nat → nat
        ceiling:        nat
        flat:           tower
        put-on:         tower nat → tower
        remove:         tower → tower
        top:            tower → nat
        height:         tower → nat
        #used:          tower nat → nat
        admissible:     tower → bool
        highest-tower:  → tower
endspec
```

Our notation of the specification should be self-explanatory. However, beside the explicitly declared operation names we have an error constant error-s for each sort s, i.e. error-tower in the example above, which may be used to specify undefined operation calls.

4.3 A first axiomatic specification

According to the informal description we do not make any assumptions about the ceiling or the number and size of the blocks given by #blocks. Hence we may directly proceed to characterize the height of a tower t as the sum of its elements, and the number #used(t, n) of blocks of length n used for t as the number of occurrences of "n" in t. Then we may describe t as admissible exactly if its height does not exceed the ceiling and if it does not use more blocks than available according to #blocks. The operation top shall return the topmost element of a non-flat tower and put-off shall remove that element.

Since we are only interested in admissble towers, we should restrict these conditions on the operations to admissible towers only in order to avoid over-specification (see (2) below). The inductive description starts with the "flat" tower which is evidently admissible (1).

Finally, the highest tower must be admissible and it must be of maximal height w. r. t. all admissible towers (3).

We specify these conditions by adding them as axioms to the signature specification TOWER.SIG:

```
spec TOWER.AX = TOWER.SIG ∪
     axioms
     ⎧ admissible (flat) = true
(1)  ⎨ height (flat) = 0
     ⎩ ∀ n : nat. #used (flat, n) = 0

     ⎧∀ t: tower.  ∀ n : nat.
     ⎪   admissible (put-on (t, n)) = true
     ⎪      ⇔   admissable (t) = true &
     ⎪          height (t) ≤ ceiling &
     ⎪          #used (t, n) ≤ #blocks (n)
     ⎪
     ⎪∀ t: tower.   ∀ n, m : nat.
(2)  ⎨   admissible (put-on (t, n)) = true & n ≠ m
     ⎪   ⇒    height (put-on (t, n)) = height (t) + n          &
     ⎪        #used (put-on (t, n), n) = #used (t, n) + 1     &
     ⎪        #used (put-on (t, n), m) = #used (t, m)         &
     ⎪        remove (put-on (t, n)) = t                      &
     ⎩        top (put-on (t, n)) = n

     ⎧admissible (highest-tower) = true
(3)  ⎨∀ t : tower. admissible (t)  = true
     ⎩                  ⇒ height (t) ≤ height (highest-tower) = true
endspec
```

As in the signature specification in section 4.1, error-constants are implicitly declared in order to support a concise and convenient notation. For the same reason we also have implicit axioms for error propagation, such as top (error-stack) = error-nat.

Moreover, all axioms are interpreted such that quantified variables are not bound to errors: For example, under our interpretation, the second axiom of (1) is equivalent to the standard interpretation of:

$\forall$ n: nat. n $\neq$ error-nat $\Rightarrow$ #used(flat, n) = 0

Since the specification TOWER.AX includes the specification TOWER.SIG, the inclusion map

TOWER.SIG $\rightarrow$ TOWER.AX
x $\mapsto$ x   for all x

describes a refinement relation. It restricts the models of TOWER.SIG to those algebras that satisfy the axioms of TOWER.AX. The latter still contain non-isomorphic models since we have not yet specified top and remove for flat towers, top, remove, height, and #used for non-admissible tower, nor have we excluded elements of sort tower that cannot be constructed with the given operations.

4.4 A partially algorithmic specification

Looking at the argument and target sorts of operations flat, put-on, remove, and top there is a close similarity to the basic operations of data type list. In fact, when translating tower to list, flat to nil, put-on to cons, remove to cdr, and top to car we observe that the standard lists over natural numbers satisfy all axioms in TOWER.AX concerning these operations. That means, given a specification NAT-LIST of such lists we should be able to extend it to a refinement of TOWER.AX by supplying axiomatic or algorithmic definitions for the remaining tower-operations in TOWER.AX. For that purpose we first discuss the algorithmic specification NAT-LIST:

```
spec NAT-LIST = BOOL ∪ NAT ∪
   sorts list
   ops      nil  :  → list
            cons :  list nat → list
            car  :  list → nat
            cdr  :  list → list
            nil? :  list → bool
   algorithmic definitions
            constructors nil, cons
```

```
    define constructor ops
        nil = *nil
        cons (l, n) = *cons (l, n)
    define ops
        car (l) = case l is  *nil: error-nat
                             *cons (l1, n1) : n1
                  esac
        cdr (l) = case l is  *nil: error-list
                             *cons (l1, n1): l1
                  esac
        nil?(l) = case l is  *nil: true
                             otherwise: false
                  esac
endspec
```

Like TOWER.SIG, NAT-LIST includes BOOL ∪ NAT, adding the list-specific signature part. Following the key word algorithmic definitions these new components are defined constructively, first the data sets and then the operations.

The list data set (or list carrier) is defined by declaring nil and cons as constructors generating the Herbrand-Universe {*nil, *cons(nil, n_1), *cons (cons(nil, n_1), n_2), ... | $n_i \in N$} ∪ {error-list} of all terms built from nil, cons and the natural numbers, and adding the error constant separately. Note that the prefix * is used to distinguish data objects from operation applications.

Our specification method allows to restrict this term-generated set by supplying an algorithmically defined characteristic predicate is-list: list → bool. It is required to respect subterms so that the restricted carrier is still closed under subterms (subterm property). This restriction guarantees that structural recursive definitions are well-defined and that structural induction is available as a proof method.

In the NAT-LIST specification a define-carriers clause with such a characteristic predicate is omitted since we are interested in all terms generated from nil and cons. In 4.6 we will give an example of a non-trivial characteristic predicate.

The operations are defined in two steps, starting with the constructor operations. Their definition following the keyword define constructors ops is constrained so that

$$cop(x_1, ..., x_n) = *cop(x_1, ..., x_n)$$

whenever $*cop(x_1, ... ,x_n)$ is in the carrier. This constructor property establishes the special meaning of the constructors and guarantees that the constructor operations are well-defined. In the NAT-LIST specification the constructor property completely determines nil and cons as operations since the list carrier is unrestricted. The definition of the constructor operation is non-trivial whenever the carrier is restricted.

Following the keyword define ops the remaining operations are defined using if-then-else schemes, case-schemes which distinguish between the possible constructors, and recursion. Previously defined operations, constructor operations and error-constants may be used as basic operations, these restrictions guaranteeing that the remaining operations are also well-defined.

As an example, consider the definition of car(l) in the NAT-LIST specification. The case-scheme distinguishes whether l is *nil or composed by "cons"ing a natural number n1 to a list l1. In the first case car is defined to yield an error, in the second case it returns the element n1.

In [BV 85] we have elaborated purely syntactic conditions guaranteeing the semantic subterm- and constructor-properties and the well-definedness of all operations. These conditions are checked by the SPESY system. Even more, as far as possible, these conditions are exploited to generate parts of the specification automatically.

Our algorithmic definitions with their explicit definitions constitute a particular restriction on the models: Every model must have a carrier that is isomorphic to the term carrier given in the algorithmic definition. Moreover, all operations on that carrier must be functionally equivalent to the algorithmic operation definitions which are interpreted with a least fix-point semantics. Thus, the algorithmic definitions constitute a constraint mechanism as it is needed in every approach considering all models of a specification (c. f. [HKR 80] and [BG 80]). Details of the semantical interpretation of algorithmic definitions by canonical term functors are given in section 6.

In order to extend NAT-LIST to a specification TOWER-AS-LIST.AX&ALG as refinement of TOWER.AX we have to add names for the missing operations. Again no assumptions are made about #blocks and ceiling. The operations sum (corresponding to height in TOWER.AX), #used, and admissible are algorithmically defined for arbitrary lists so as to satisfy the corresponding axioms of TOWER.AX. The problem of devising an algorithm to compute the maximal-list (corresponding to the highest tower) is deferred until a later stage; therefore, the corresponding axioms (3) from TOWER.AX are simply translated:

```
spec TOWER-AS-LIST.AX&ALG = NAT-LIST ∪
  ops #blocks:   nat → nat
      ceiling:   → nat
      sum:       list → nat
      #used:     list nat → nat
      admissible: list → bool
      maximal-list: → list

  algorithmic definitions
      define ops
              sum(l) = if nil?(l)
                       then 0
                       else car(l) + sum(cdr(l))
              #used(l, n) = if nil?(l)
                            then 0
                            else if cdr(l) = n
                                 then 1 + #used (cdr(l), n)
                                 else #used (cdr(l), n)
              admissible(l) =    if nil?(l)
                                 then true
                                 else  if sum(l) ≤ ceiling and
                                             #used(l, car(l)) ≤ #blocks(car(l))
                                       then admissible(cdr(l))
                                       else false
```

axioms

$$(4)\ \begin{cases} \text{admissible (maximal-list)} = \text{true} \\ \forall\, l: \text{list. admissible}(l) = \text{true} \\ \quad \Rightarrow \text{sum}(l) \le \text{sum(maximal-list)} = \text{true} \end{cases}$$

endspec

Since TOWER-AS-LIST.AX&ALG includes NAT-LIST, it obviously refines this specification. In contrast the refinement relation to TOWER.AX is non-trivial and is described by the following signature translation ϱ_1:

ϱ_1: TOWER.AX → TOWER-AX-LIST.AX&ALG

tower ↦ list
flat ↦ nil
put-on ↦ cons
remove ↦ cdr
top ↦ car
height ↦ sum
highest-tower ↦ maximal-list
x ↦ x otherwise

In order to establish ϱ_1 as a correct refinement relation we have to translate the axioms of TOWER.AX via ϱ_1 to formulas over the signature of TOWER-AS-LIST.AX&ALG and prove their validity in the latter specification:

axioms

$$(1)\ \begin{cases} \text{admissible (nil)} = \text{true} \\ \text{sum (nil)} = 0 \\ \forall\, n : \text{nat. \#used (nil, n)} = 0 \end{cases}$$

$$(2)\ \begin{cases} \forall\, t: \text{list. } \forall\, n : \text{nat.} \\ \quad \text{admissible (cons (t, n))} = \text{true} \\ \qquad \Leftrightarrow \text{admissible (t)} = \text{true } \& \\ \qquad\quad \text{sum (t)} \le \text{ceiling } \& \\ \qquad\quad \text{\#used (t, n)} \le \text{\#blocks (n)} \\ \\ \forall\, t: \text{list. } \forall\, n, m : \text{nat.} \\ \quad \text{admissible (cons (t, n))} = \text{true } \& \ n \ne m \\ \quad \Rightarrow \ \text{sum (cons (t, n))} = \text{sum (t)} + n \qquad \& \\ \quad \vdots \end{cases}$$

⋮

$$\left.\begin{array}{ll} \#used\ (cons\ (t, n), n) = \#used\ (t, n) + 1 & \& \\ \#used\ (cons\ (t, n), m) = \#used\ (t, m) & \& \\ cdr\ (cons\ (t, n)) = t & \& \\ car\ (cons\ (t, n)) = n & \end{array}\right.$$

$$(3)\left\{\begin{array}{l} admissible\ (maximal\text{-}list) = true \\ \forall\ t : list.\ admissible\ (t) \quad = true \\ \qquad \Rightarrow sum\ (t) \quad \le sum\ (maximal\text{-}list) = true \end{array}\right.$$

From the algorithmic definitions of TOWER-AS-LIST.AX&ALG we can prove (1) - (3) by induction on the list carrier set, thus ensuring the correctness of the refinement relation ρ_1.

4.5 A fully algorithmic specification

In order to give a fully algorithmic solution to our problem we are left to devise an algorithm to compute the maximal admissible list. This algorithm will then be added to the algorithmic definitions of TOWER-AS-LIST.AX&ALG, yielding the specification TOWER-AS-LIST.ALG.

The idea behind the algorithm is to generate all relevant admissible towers and to remember the highest one constructed so far. On termination the latter will be the highest admissible tower at all. Since the height of a tower does not change when permuting its components we need to consider only towers with decreasing block-size as relevant towers. In order to generate them we extend a current admissible tower by a new block - called gapfiller - which is the largest block not yet used for the current tower and still fitting in the gap between tower and ceiling. Since we need to consider only relevant towers, blocks that are larger than the topmost block of the current tower need not be considered as gapfillers. If there are no more gapfillers available we backtrack by repeatedly removing the topmost block from the current tower until a new gapfiller is found to be put on the cut-off tower.

Realizing this idea we have four operations:

gapfiller(l, gap) determines the size of the largest block not yet used in l that fits into "gap".

try(l, maxl) and backtrack(l, maxl) cooperate by extending resp. destroying the current list l so that all relevant admissible lists are generated. The maximal list encountered so far is remembered as "maxl".

maximal-list starts the search by calling try with a current list containing only the size of the largest block filling the gap between floor and ceiling.

All but the last operation are declared as <u>private ops</u> because they shall not be visible outside the specification TOWER-AS-LIST.ALG:

```
spec TOWER-AS-LIST.ALG = TOWER-AS-LIST.AX&ALG ∪
    algorithmic definitions
     private ops  gapfiller: list nat → nat
                  try: list list → list
                  backtrack: list list → list

     define ops
       maximal-list = let base = gafiller (nil, ceiling) in
                          if  base = 0
                          then nil
                          else try (cons (nil, base), nil)

       gapfiller(l, gap) = if gap = 0
                           then 0
                           else if #blocks(gap) - #used(l, gap) > 0
                                then gap
                                else gapfiller(l, gap -1)

       try(l, maxl) = let ontop = gapfiller(l, min (car(l), ceiling - sum(l))) in
                          if ontop = 0
                          then if sum(l) > sum(maxl)
                               then backtrack(l, l)
                               else  backtrack(l, maxl)
                          else try(cons(l, ontop), maxl)
       backtrack(l, maxl) = let  destroyed = cdr(l) in
                            let  ontop = gapfiller(destroyed, car(l) - 1) in
                                 if ontop = 0
                                 then  if sum(destroyed) = 0
                                       then l
```

else backtrack(destroyed, maxl)
else try(cons(destroyed, ontop), maxl).

endspec

Since TOWER-AS-LIST.ALG includes TOWER-AS-LIST.AX&ALG it is obviously a refinement of the latter specification. The only question is whether TOWER-AS-LIST.ALG is still consistent, which means that the algorithmic definition of maximal-list satisfies the two axioms (4) for that operation. To answer this question we can rephrase the axioms as

```
admissible(maximal list) = true
admissible(nil) = true ⇒
                    sum(nil) ≤ sum(maximal-list) = true
∀ l: list. ∀ n:nat.
    (admissible(l) = true ⇒
                    sum(l) ≤ sum(maximal-list) = true)
                    ⇒
    (admissible(cons(l,n)) = true ⇒
                    sum(cons(l,n)) ≤ sum(maximal-list) = true)
```

for a proof by induction on the structure of the lists.

Refinement relations are closed under composition. Therefore, we can compose ϱ_1 with the inclusion from TOWER-AS-LIST.AX&ALG to TOWER-AS-LIST.ALG to obtain the refinement relation

ϱ_2: TOWER.AX	→	TOWER-AS-LIST.ALG	
x	↦	$\varrho_1(x)$	for all x

Since the operations ceiling and #used are left unrestricted according to our informal problem statement, the specification TOWER-AS-LIST.ALG still has non-isomorphic models. However, for every fixed interpretation of these two operations, there are only isomorphic models due to the interpretation of the algorithmic definitions. Therefore, TOWER-AS-LIST.ALG is a complete algorithmic specification with respect to the problem statement.

4.6 Alternatives

As already mentioned, the first axiomatic specification TOWER.AX is not complete for the following reasons:

- the data set of sort tower may contain unreachable objects
- it may contain non-admissible towers
- the behaviour of the operations top and remove applied to the flat tower is not specified
- except for admissible, the behaviour of all operations w. r. t. non-admissible towers is not specified.

We may answer these open questions except for the first one by adding axioms to TOWER.AX that

- determine the tower data set to consist of all admissible towers only
- specify that top and remove result in errors when applied to the flat tower

yielding a more precise axiomatic specification TOWER.AX1. In order to exclude unreachable elements first order formulas are in general not powerful enough. Instead, second order formulas or a constraint mechanism as mentioned in section 4.4 must be used.

It is interesting to see that the algorithmic specification TOWER-AS-LIST.ALG, which was naturally developed from our first but still incomplete axiomatic specification TOWER.AX, is not a refinement of the refined axiomatic specification TOWER.AX1 since the list data set includes non-admissible lists. In order to obtain an algorithmic refinement TOWER-AS-RESTRICTED-STACK of TOWER.AX1 we could specifiy a type of restricted stacks that exactly correspond to the admissible towers by taking the admissible operation as characteristic predicate. Since these restricted stacks are isomorphic to the subset of admissible lists we can use an analogous algorithm to compute the highest restricted stack.

5. The specification development language ASPIK

The specification method presented in section 4 is not immediately suited for practical use since it lacks any structuring, parameterization, and instantiation facilities. Such mechanisms are provided in our specification development language ASPIK which allows to define hierarchically structured, loose axiomatic and algorithmic specifications as well as hierarchically structured maps defining refinement and implementation relations between hierarchical specifications.

As an example we show in figure 5.1 how the specification TOWER-AS-LIST.AX&ALG of section 4.4 could be hierarchically composed of several ASPIK specifications which are sketched in figure 5.2.

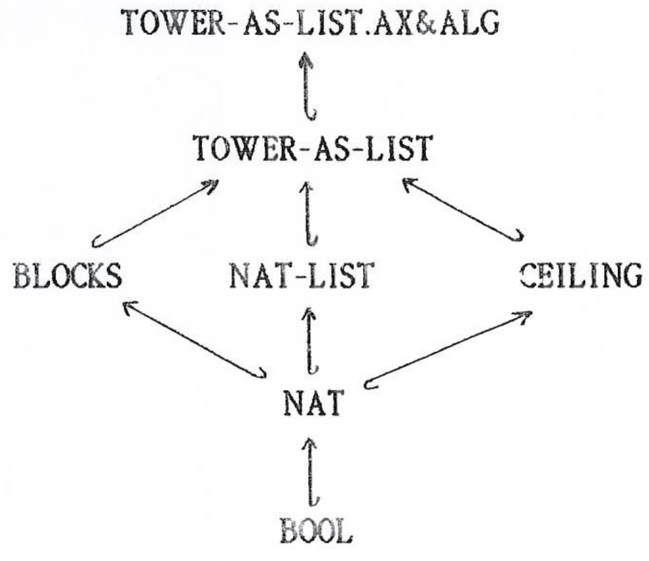


Figure 5.1: Hierarchical structure of the tower specifications in ASPIK

```
spec  BOOL  ...   endspec

spec  NAT
   use  BOOL
    ...
endspec

spec  BLOCKS
   use  NAT
   ops  #blocks : nat → nat
    ...
endspec

spec  CEILING
   use  NAT
   ops  ceiling : → nat
    ...
endspec

spec NAT-LIST
   use  NAT
   sorts  list
   ops       nil        : → list
             cons       : list nat → list
             car        : list → nat
             cdr        : list → list
             nil?         list → bool
   spec body  ...  ‹ algorithmic definitions  part of NAT-LIST in section 4.4›
endspec

spec  TOWER-AS-LIST
   use BLOCKS, CEILING, NAT-LIST
   ops    sum : list → nat
          #used : list nat → nat
          admissible : list → bool
   spec body
          ...‹algorithmic  definitions  part  of  TOWER-AS-LIST.AX&ALG  in
                                                               section 4.4›
endspec
```

```
spec TOWER-AS-LIST.AX&ALG
   use TOWER-AS-LIST
   ops maximal-list : → list
   props ... <axioms-part of TOWER-AS-LIST.AX&ALG in section 4.4>
endspec
```

Figure 5.2: Specifications for the highest-tower problem in ASPIK

A hierarchical ASPIK specification can be instantiated by substituting arbitrary subspecifications in its hierarchy by other hierarchical specifications via refinement maps describing the replacements to be performed. This concept disposes entirely of the notion of formal parameters since the specifications to be substituted need only to be identified at instantiation time, together with the actual parameters. In contrast to usual parameterization concepts where the formal parameters must be statically declared, this concept of ASPIK is called dynamic parameterization.

As an example consider the term

TOWER-AS-LIST.AX&ALG { CEILING → NAT }

defining a particular instance of the specification TOWER-AS-LIST.AX&ALG where "CEILING → NAT" is the ASPIK object

```
map CEILING → NAT
  base NAT
  ops ceiling = 15
endmap
```

describing a refinement relation from the specification CEILING to the NAT specification.

6. Semantics: Canonical term functors and algorithmic constraints

In this section we describe the concepts underlying the semantics of our approach to algebraic specifications. In section 6.1 we recall the definition of canonical term algebra introduced by the ADJ group in [GTW 78] and show how this notion can be generalized to canonical term functors, providing suitable models for a parameterized constructive definition. In section 6.2 we explain the meaning of our algorithmic specifications in terms of canonical term functors. In section 6.3 we show how canonical term functors can be interpreted as algorithmic constraints on the specified models, and in section 6.4 we discuss how the integration of axiomatic and alorithmic techniques in the semantics of specifications is achieved.

6.1 Canonical term algebras and functors

Term algebras have played an important role in abstract data type theory. They allow to define a particular algebra by explicitly introducing its carrier sets and by defining its operations on these carrier sets still in a very abstract way and without having to invent some fancy representation: the carriers are just syntactic items, i.e. well-formed terms over the algebra's signature, and the operations act on these terms by composing or decomposing them.

Term algebras are used for the quotient term algebra construction in the initial approach of [GTW 78]. In the quotient term algebra one has to deal with equivalence classes while in a canonical term algebra (cta) a representative is chosen for each equivalence class. By imposing a certain discipline on the choice of the representatives the structure of the terms can be exploited. For example, proofs can be done by structural induction as in the cited paper of the ADJ group or in [Pad 79].

The power of the cta concept is demonstrated in [GTW 78] by showing that for every equational specification an initial cta exists. However, the proof is non-constructive, and in general there is no algorithm which generates an initial cta from an equational specification.

This is the reason why we devised a constructive definition method based on the notion of cta.

Definition 6.1 [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) $\forall op: s_1 ... s_n \to s \in Op \,.$

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

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

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

Since the models underlying our overall approach are strict algebras we specialize the definition to strict canonical term algebras:

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 adressable 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-constant implicitly.

Definition 6.2 [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) $\forall s \in S \,.\, error\text{-}s \in A_s$.

For practical purposes, the concept of cta is not adequate enough: Instead of defining an entire cta from scratch we would like to compose them from reusable "cta-pieces", some of which may have been defined previously and stored in a library. Except for some elementary ctas such "cta-pieces" correspond to instances of "parameterized ctas". To write parameterized ctas we would like to start with a class of "parameter algebras" for which we do not assume any term structure. They are extended by new carriers and operations obeying the cta-requirements (i.e. term, subterm, and constructor properties) relative to the parameter algebras. Obviously, this extension should not change the old sorts and operations in the parameter algebras. By defining the extension not only on the parameter algebras but also on the homomorphisms between them we get a strongly persistent functor from the parameter algebras to the extended algebras.

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 6.3 [term-, subterm-, constructor property]

Let Σ, Σ' be signatures such that $\Sigma \subseteq \Sigma'$.
Let $A \in Alg(\Sigma)$ and $A' \in Alg(\Sigma')$.

(1) A' has the $(\Sigma' - \Sigma)$-term property w.r.t. A

iff

$$\forall s \in \Sigma' - \Sigma.\ A_s' \subseteq T_{\Sigma' - \Sigma}(A)_s$$

(2) A' has the $(\Sigma' - \Sigma)$-subterm property w.r.t. A

iff

$$\forall s \in \Sigma' - \Sigma.\ \forall op: s_1 \dots s_n \to s \in \Sigma' - \Sigma$$
$$op(t_1,\dots,t_n) \in A_s'$$
$$\Rightarrow t_1 \in A_{s1}' \ \&\ \dots\ \&\ t_n \in A_{sn}'$$

(3) A' has the $(\Sigma' - \Sigma)$-constructor property w.r.t. A

iff

$$\forall s \in \Sigma' - \Sigma.\ \forall op: s_1 \dots s_n \to s \in \Sigma' - \Sigma$$
$$op(t_1,\dots,t_n) \in A_s'$$
$$\Rightarrow op_A'(t_1,\dots,t_n) = op(t_1,\dots,t_n)$$

Definition 6.4 [canonical term functor, ctf]

Let $\iota: \Sigma \rightarrow \Sigma'$ be a signature inclusion, and let $C \subseteq Alg(\Sigma)$ and $C' \subseteq Alg(\Sigma')$ be subcategories closed under isomorphisms.
A functor

$$g: C \rightarrow C'$$

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 w.r.t. A
(2.3) g(A) has the $\Sigma' - \Sigma)$-constructor property w.r.t. A

The requirement in Definition 6.4 that C and C′ are closed under isomorphisms is not essential; it corresponds to the concept of abstract data type theory that isomorphic algebras are considered to be 'equal'.

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) in Definition 6.2, we define a strict ctf to be a ctf between two categories of strict algebras which satisfies the error requirement w.r.t. the new carriers.

Definition 6.5 [strict ctf]

Let $\iota: \Sigma \rightarrow \Sigma'$ be a signature inclusion, and let $C \subseteq EAlg(\Sigma)$ and $C' \subseteq EAlg(\Sigma')$ be subcategories closed under isomorphisms. A functor

$$g: C \rightarrow C'$$

is a strict (Σ,Σ')-ctf

iff

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

As an example for a strict ctf let C be the category of one-sorted algebras, and C′ the category of lists over arbitrary elements where the list operations have the usual names: nil, cons, car, and cdr. Now let g be a functor $g: C \rightarrow C'$ whose object part is defined by extending every one-sorted algebra A with carrier set $\{e_i \mid i \in I\}$ by the list carrier set

$\{nil, cons(nil,e_1), cons(cons(nil,e_1),e_2),... \mid i \in I\}$

and by the usual list operations such that e.g.

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

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 $cons(t,e_i)$ 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 $cons_{g(A)}(t,e_i) = cons(t,e_i)$ for every term $cons(t,e_i)$ in the list carrier.

The concept of ctf was motivated by the idea to compose ctas piecewise. Hence, a constant ctf should yield a cta, a ctf applied to a cta should yield a cta, and a ctf applied to a ctf should yield again a ctf. These properties are verified in the following three facts both for the ordinary and the strict version.

Fact 6.6 [constant ctfs are ctas]

Let Σ be a signature, $A \in Alg(\Sigma)$ [resp. $A \in EAlg(\Sigma)$] and

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

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

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.

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

Fact 6.7 [ctfs can be composed]

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.

Fact 6.8 [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.

6.2 Algorithmic definitions of canonical term functors

We now turn to the question of how to define ctfs constructively. Because of the results in the previous subsection this will also give us a definition method for ctas. It consists essentially of three components:

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

An obvious choice for (1) is to take a loose algebraic specification $\langle\Sigma,E\rangle$ and to let EAlg($\langle\Sigma,E\rangle$) denote the class of parameter algebras. For (2) and (3) we can direcly adapt the respective parts of the algorithmic-definitions construct described in section 4.

For example, the loose specification ELEM = $\langle\langle\{elem\},\emptyset\rangle,\emptyset\rangle$ with the single sort elem denotes the class of arbitrary one-sorted algebras.
By simply replacing ELEM for NAT in the specification NAT-LIST in section 4.4 we obtain a parameterized constructive specification ELEM-LIST of lists over arbitrary elements.

6.3 Algorithmic constraints

In section 4.4 we explained for the specification NAT-LIST how its list carrier is explicitly constructed as a set of terms, how its operations are alorithmically defined, and we pointed out how the subterm and constructor properties are satisfied.

According to this description the formal meaning of our algorithmic definitions is a ctf. More precisely, let SP be an algorithmic specification (like NAT-LIST or ELEM-LIST) with parameter specification part $\langle\Sigma,E\rangle$ (like NAT or ELEM) and new signature part

$$\Sigma_{new} = \langle S_{new}, Op_{new}\rangle \quad \text{(like } \langle\{list\},\{nil,cons,car,cdr,nil?\}\rangle\text{)}.$$

Then SP denotes a strict canonical term functor

$$ctf_{SP}: EAlg(\Sigma,E) \to EAlg(\langle\Sigma \cup \Sigma_{new},E\rangle)$$

(like our construction of lists over the NAT-algebra or over an ELEM-algebra).

An arbitrary $(\Sigma \cup \Sigma_{new})$-algebra A (like the algebra of standard lists over the natural numbers) satisfies the algorithmic specification SP exactly if A is generated - up to isomorphisms - from its Σ-reduct - (in our example the NAT-algebra) by the functor ctf_{SP}.

A little more formally, let $\iota: \Sigma \to \Sigma \cup \Sigma_{new}$ be the signature inclusion and let $EAlg(\iota): EAlg(\Sigma \cup \Sigma_{new}) \to EAlg(\Sigma)$ be the corresponding forgetful functor, which forgets the new sorts and operations of Σ_{new}. Then

A satisfies SP

$$\Leftrightarrow$$

$$A \cong ctf_{SP}(EAlg(\iota)(A))$$

In our example the algebra A of standard lists over the natural numbers satisfies $ctf_{NAT-LIST}$. But an algebra A′ obtained by adding terms like "default-list" or "cons(nil, overflow)" as new elements to the list carrier of A does not satisfy $ctf_{NAT-LIST}$. Such elements are called unreachable, and constructs that allow to exclude unreachable elements are called constraints.

For example, [HKR 80], [BG 80] and [EWT 83] use a constraint mechanism involving a free functor which is specified by equational theories. 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) whereas the other approaches also require that generated elements must be distinct ("no-confusion" condition).

Due to the term property of ctfs our algorithmic definitions exclude unreachable elements and therefore serve as a constraint mechanism in our specification method. This is the reason why we use the term "algorithmic constraints" for our algorithmic definitions.

6.4 Integration of axiomatic and algorithmic techniques

Usually, a specification is a pair $\langle\Sigma,E\rangle$ consisting of a signature Σ and a set E of sentences. Often, sentences are equations or other logical formulas. But, more generally, any item p may be viewed as a sentence provided we have a satisfaction condition telling us whether a Σ-algebra A satisfies p.

Taking again the specification NAT-LIST from section 4.4, Σ is the complete signature of NAT-LIST and contains e.g. sorts bool, nat, and list. The axiomatic components of NAT-LIST are first-order formulas. They can obviously be put into the set E of sentences. The algorithmic components of NAT-LIST are algorithmic constraints (for the natural numbers and the lists over natural numbers). So far, no algorithmic approach like [Kl 84] or [Lo 84] has considered algorithmic definitions as sentences. But since we have a satisfaction condition for algorithmic constraints, namely the condition in section 6.3 about whether an algebra A satisfies an algorithmic constraint ctf_{SP}, we may add our algorithmic constraints to the set E of sentences. Thus, E may contain an arbitrary mixture of axiomatic first order formulas and algorithmic constraints representing a uniform integration of axiomatic and algorithmic techniques.

7. Conclusions

We presented a specification method that allows to formalize much of the software development process so that the individual development steps can be proved to be correct by formal verification methods. We demonstrated why loose algebraic specifications are particularly suited for this purpose. Whereas previous approaches provide either axiomatic or algorithmic definitions our approach integrates both techniques in a uniform way, using the new notions of canonical term functor and algorithmic constraints. All ideas presented here are realized in the specification development language ASPIK which has been implemented as a core component of an integrated software development and verification system ([BV 85], [BOV 86b]).

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.

[BGGORV 83] Beierle, C., Gerlach, M., Göbel, R., Olthoff, W., Raulefs, P., Voß, A.: Integrated Program Development and Verification. In: H.-L. Hausen (ed.): Symposium on Software Validation. North Holland Publ. Co., Amsterdam, 1983.

[BOV 86] Beierle, C., Olthoff, W., Voß, A.: Software development environments integrating specification and programming languages. In.: H.-W. Wippermann (ed): Software Architektur und modulare Programmierung. Proceedings German Chapter of the ACM, Teubner Verlag, Stuttgart, 1986.

[BOV 86b] Beierle, C., Olthoff, W., Voß, A.: Automatic theorem proving in the ISDV system. Proc. 8th Conference on Automated Deduction, LNCS 230, 1986.

[BOV 86c] Beierle, C., Olthoff, W., Voß, A.: Towards a formalization of the software development process. Proc. Software Engineering ´86, Southampton, U.K., 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.

[EKMP 82] Ehrig, H., Kreowski, H.-J., Mahr, B., Padawitz, P.: Algebraic Implementation of Abstract Data Types. Theoretical Computer Science Vol. 20, 1982, pp. 209-254, (also:) Bericht Nr. 80-32, Fachbereich Informatik, Techn. Univ. Berlin 1980.

[EKP 78] Ehrig, H.,Kreowski, H.J.,Padawitz, P.: Stepwise specification and implementation of abstract data types. Proc. 5th ICALP, LNCS Vol. 62, 1978, pp. 203-206.

[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.

[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.

[GTWW 75a] Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Abstract data types as initial algebras and the correctness of data representations. Proc. of Conf. on Computer Graphics, Pattern Recognition and Data Structures, 1975.

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

[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.

[Hün 80] Hünke, H. (ed.): Software Engineering Environments. North Holland Publ. Co., Amsterdam, 1980.

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

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

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

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

[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)

[LZ 74] Liskov, B.H., Zilles, S.N.: Programming with Abstract Data Types. SIGPLAN Notices Vol. 9, 1974, No. 4, pp. 50-59.

[MG 85] Meseguer, J., Goguen, J.: Initiality, induction, and computability. In.: M. Nivat, J. Reynolds (eds): Algebraic Methods in Semantics. Cambridge University Press, 1985, pp. 460 - 541.

[Mor 73] Morris, F.L.: Types are not sets. Proc. ACM POPL, 1973, pp. 120 - 124.

[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.

[TWW 78] Thatcher, J.W., Wagner, E.G., Wright, J.B.: Data Type Specification: Parameterization and the Power of Specification Techniques. Proc. 10th Annual ACM Symposium on Theory of Computing. 1978, pp. 119-132.

[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.

[Zil 74] Zilles, S.N.: Algebraic specifications of data types, Project MAC Prog. Rep. 11, MIT pp. 52-58, 1974.

[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.