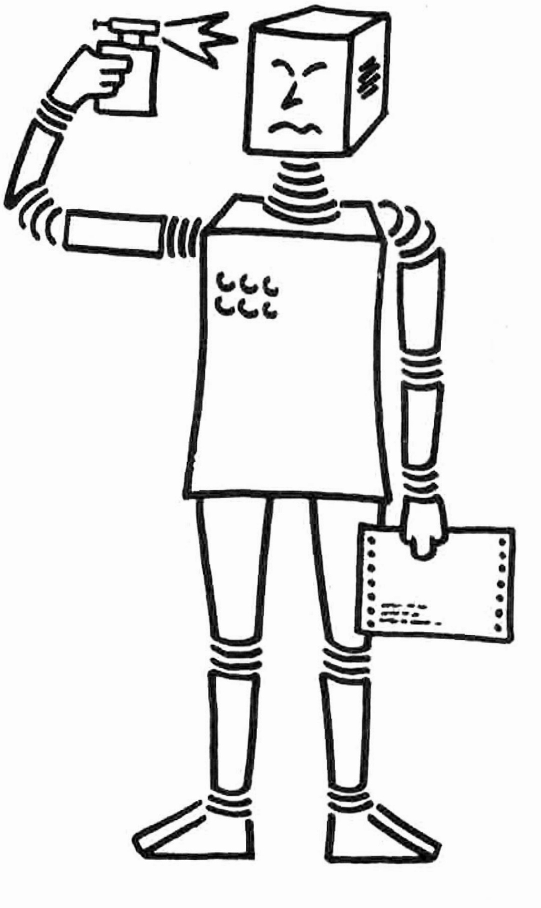
SEKI-PROJEKT

SEKI MEMO

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



Parameterization without Parameters
in:
The History of a Hierarchy of Specifications

Ch. Beierle, M. Gerlach, A. Voss

Memo SEKI-83-09 September 1983

Parameterization without Parameters in: The History of a Hierarchy of Specifications

Ch. Beierle
M. Gerlach
A. Voss

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

Abstract

We are going to tell you the history of a hierarchy of specifications dealing with the evolutions, revolutions and conquests of the family of sorting methods. However, this hierarchy and its history are so intricate that a special language is used to describe them: ASPIK - a specification language for hierarchies consisting of formal requirements and particular models arbitrarily mixed or produced by applications and abstractions.

This research was supported by the Bundesministerium für Forschung und Technologie under contract IT.8302363.

Contents

1.	Prologue	1
2.	A crash course in ASPIK	2
2.1	Loose and fixed specifications	2
2.2	Parameterization-by-use	4
2.3	Canonical term functors	4
2.4	Applications	5
3.	The hi-story of a hierarchy	7
3.1	A datatypist sorts lists of natural numbers	7
3.2	A chief datatypist sorts arbitrary lists	12
3.3	A chief abstract datatypist sorts systematically	15
3.4	A Master in ASPIK divides and conquers	22
4.	The specifications	24
4.1	The specifications of the datatypist	24
4.2	The specifications of the chief datatypist	28
4.3	The specifications of the chief abstract datatypist	31
4.4	The specifications of the Master in ASPIK	38
5.	The moral	39
References		40

1. Prologue

We are going to tell you the hi-story of a hierarchy. You will hear about the evolutions and revolutions in the family of sorting methods (as recorded by [Da 76]), and you will see how they came up to divide and conquer.

However, this hierarchy and its history are highly intricate, so intricate that a special language is required to describe them: ASPIK - a specification language for hierarchies consisting of formal requirements and particular models arbitrarily mixed or produced by applications and abstractions. So, unless you are already familiar with [BV 83a], [BV 83b], we shall offer you a crash course in ASPIK.

2. A crash course in ASPIK

ASPIK is a specification language for abstract data types. A specification as sketched in Fig. 2.0 may use others such that arbitrary specification hierarchies can be constructed serving as a type library.

2.1. Loose and fixed specifications

Every specification in a hierarchy is either loose or fixed. A loose specification may introduce new sorts and operation names (ops), and it may impose properties (props). It is interpreted loosely in the way that every algebra which supplies carriers for the sorts and functions for the operation names such that the properties are satisfied is a model of the specification. In this way, a loose specification expresses formal requirements. Other approaches dealing with loose specifications are e.g. [Eh 81], [HKR 80], [BG 80], [Sa 81], [BDPPW 80], and [Ba 81].

A fixed specification consists of a header and a body. The header is a loose specification. In the body, the new sorts and operation names are constructively defined in a way consistent with the properties. That means, the body supplies a particular model of the header. Thus, fixed specifications support the algorithmic definition of abstract data types. These models may serve as prototypes, because their operations are executable though abstractly defined algorithms. Other algorithmic approaches are described in [Kl 80] and [Lo 81].

Since loose and fixed specifications may be deliberately mixed in a hierarchy, the notions "loose" and "fixed" are always relative to the specifications being used: a stack may be fixed w.r.t. its loose elements, an arbitrary ordering postulated for the characters is loose w.r.t. the fixed characters. Due to this mixture, ASPIK supports all stages of development from formal requirements to executable prototypes in a uniform way.

```
spec <spec-name>

   use ...                              } hierarchy of          ⎤
                                          specifications        ⎥
                                                                ⎥
   sorts ...                            } signature of          ⎥  specification
   ops ...                                specification         ⎥     header
                                                                ⎥
   props...                             } properties of         ⎥
                                          specification         ⎦

spec body

   constructors ...                     ⎤                       ⎤
                                        ⎥ definition            ⎥
   auxiliaries ...                      ⎥    of                 ⎥
                                        ⎥ carriers              ⎥
   define auxiliaries ...               ⎥                       ⎥
                                        ⎥                       ⎥
   define carriers...                   ⎦                       ⎥  specification
                                                                ⎥     body
   define constructor ops ...           ⎤                       ⎥
                                        ⎥ definition            ⎥
   private ops ...                      ⎥    of                 ⎥
                                        ⎥ operations            ⎥
   define ops ...                       ⎦                       ⎦

endspec
```

Figure 2.0: Syntactic structure of an ASPIK-specification

2.2. Parameterization-by-use

Orthogonal to the hierarchical use relation is the refinement relation which expresses a selection of models. In more detail, to refine a specification SP1 by another one SP2, a specification morphism must be defined: the sorts and operation names of SP1 must be associated to those of SP2 such that the properties of SP1 are satisfied by the associated operations in SP2, and if SP1 is fixed SP2 must be fixed in the same way.
The refinement relation is exploited to generate new hierarchies as applications or abstractions of old ones. An application is obtained by viewing some specifications in a hierarchy as formal parameters to be substituted by fitting actual parameters, where an actual parameter fits if it refines the formal one. Vice versa, if specifications in a hierarchy are replaced by more general ones, an abstraction is obtained.
Abstractions and applications are inverse mechanisms allowing to develop specifications by deliberately switching from the concrete to the abstract and back to the concrete direction. The flexibility of these two mechanisms depends on the fact that the specifications to be substituted need not be declared as static parameters of the hierarchy, but need to be identified no sooner than the latest possible moment: when they are substituted by other specifications. This feature of ASPIK is called parameterization-by-use. A mechanism similar to our applications is proposed in [ZLT 82]. Previous approaches dealing with explicit parameterization can be found in [Eh 81] and [EKTWW 81].
Two novel features of ASPIK deserve further explanation: canonical term functors as constraint mechanism and applications as realization of the parameterization-by-use concept.

2.3. Canonical term functors

The particular model defined in a fixed specification is a canonical term algebra [GTW 78]. Its carriers consist of terms whose subterms all belong to the carriers as well. They are defined by constructors generating sets of terms which may be cut down by characteristic predicates (in define carriers). Sometimes auxiliaries are needed to define the characteristic predicates. Every constructor induces an operation that yields the corresponding term if it belongs to the carrier. They are separately defined (in define constructor ops) to stress this condition. The remaining operations are defined in define ops and may use some private operations (declared in private ops) and auxiliaries.

The advantages of canonical term algebras are twofold. Since their carriers consist of terms built from abstract operation names, they provide an abstract way to define a particular model. Moreover, the recursive definition of these carriers allows to do structural induction. This is demonstrated in [Pa 79].

However, in ASPIK a fixed specification is fixed only w.r.t. to the specifications used, which may be loose. Therefore, the notion of "canonical term algebra" is generalized to "canonical term functor". A canonical term functor is a functor mapping every algebra which is a model of the specifications used to a canonical algebra which is a model of the specification at hand. Syntactically a canonical term functor can be defined by the same scheme as a canonical term algebra, provided no reference is made to the carriers of inherited sorts other than by inherited operations - thus the structure of the carrier elements is hidden to the outside.

Canonical term functors constitute the constraint mechanism of ASPIK: they constrain or fix the loose models of the specification header to the particular ones defined by the body. Other constraint mechanisms are presented in [HKR 80], [BG 80], [EWT 82], [SW 82a], [SW 82b], and [ZLT 82].

2.4. Applications

It is natural to think of a node in a hierarchy as a unit that is neither to be confused with other nodes nor multiplied via different uses: the same operation name op introduced in different specifications SP1 and SP2 shall denote different operations - namely SP1.op and SP2.op. On the other hand, combining two specifications both using SP1 shall provide exactly one operation - namely SP1.op.

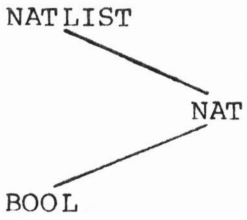
This is desirable not only for an explicitly defined specification SP1 but also if SP1 is an application (or abstraction). Therefore, every hierarchy in ASPIK is implicitly extended to include a node for every conceivable application: the hierarchy is closed under applications.

The closure construction is complicated by the following problem: Applications are syntactically denoted by application terms. Two application terms might differ just in the order of actualizing independent specifications. Or one application term might perform an actualization in a single step which is split into several steps in another term. In both cases the application terms should be semantically equivalent. This equivalence is realized in ASPIK by determining normal forms such that two application terms reducing to the same normal form are semantically equivalent. Accordingly, in order to close a hierarchy under applications new nodes are implicitly added for the normal forms only.

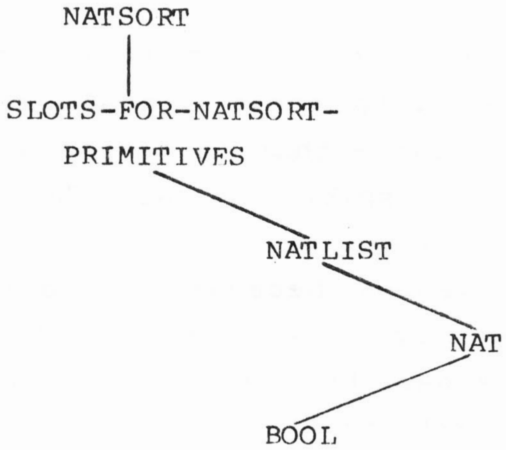
3. The hi-story of a hierarchy

3.1. A datatypist sorts lists of natural numbers

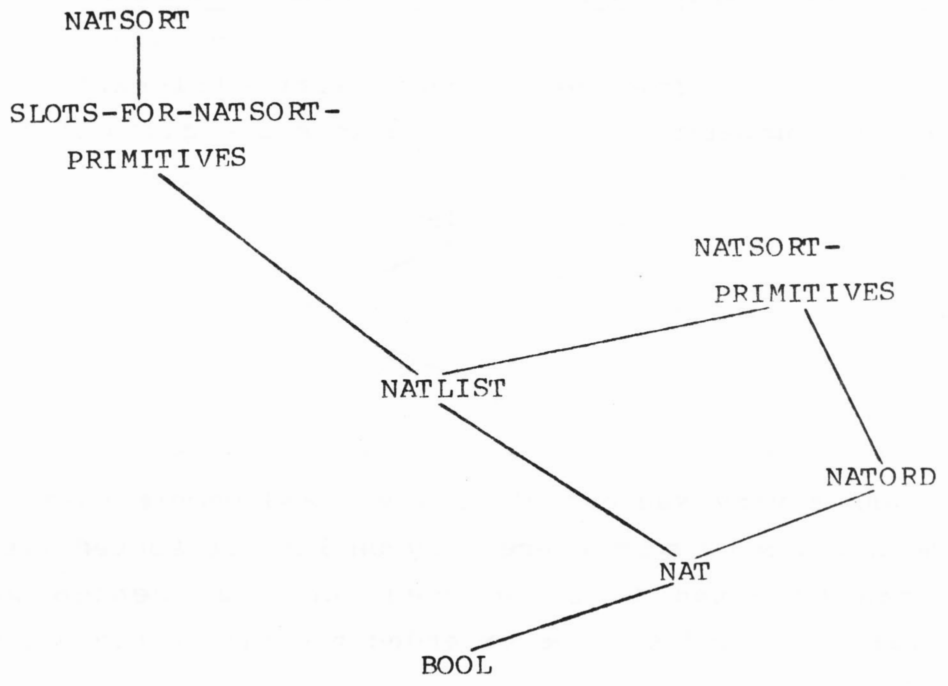
Once upon a time there was a little hierarchy for lists of natural numbers. (The specifications are given in Section 4.)



It was a very successful datatype and people used it heavily. So the day was to come where they called for sorted lists.
A man educated in structured top down design was appointed datatypist, and soon he extended the little hierarchy to:



NATSORT provided a sorting algorithm based on primitive operations which were declared but not yet defined in SLOTS-FOR-NATSORT-PRIMITIVES. Trying to define them in a separate specification NATSORT-PRIMITIVES, the datatypist discovered that the ordering of the natural numbers was missing in NAT. So he added the operation <= in an extra specification NATORD on top of NAT. NATLIST and NATORD together supplied the base for the algorithms of NATSORT-PRIMITIVES.



"To finish my task I need a mechanism to refine NATSORT by inserting the primitives into their slots", the datatypist explained to the aspikialist - that is a specialist in ASPIK. "There are two replies", the aspikialist said, "a short practical one and a long theoretical one.

Practically you are finished, because in the meantime your hierarchy has considerably grown: Beside the nodes added explicitly further nodes have been entered implicitly - among them a node solving your problem:

```
NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES -ml→ NATSORT-PRIMITIVES)
with ml: ops  part1        =  min
              part2        =  allbutmin
              combine      =  putmin
              simple?      =  simple?
              simple-sort  =  simple-sort
```

associates the primitives with their slots." The perplexed face of the datatypist made the aspikialist continue with the theoretical explanation.

"Theoretically a closure construction is applied to your hierarchy each time you add a node. As a result there is a node for every application you may conceive. An application or instantiation of a node like NATSORT is obtained by considering some nodes below it as formal parameters and replace them by suitable nodes as actual parameters. In your case SLOTS-FOR-NATSORT-PRIMITIVES is the formal parameter which is actualised by the NATSORT-PRIMITIVES."

"Let me see if I got it", the datatypist replied eagerly. "After I had added NATORD I could have moved NATLIST thereupon and build the NATSORT-PRIMITIVES on top. This could have been achieved by simply using NATLIST(NAT -m2→ NATORD) in NATSORT-PRIMITIVES, where m2 relates the sorts and operations of NAT identifically to themselves." (Furtheron we will omit the maps because they are rather evident.)

"Exactly. There even is an application node in your hierarchy corresonding to this alternative:

NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES → NATSORT-PRIMITIVES)
(NATLIST →NATLIST(NAT -m2→ NATORD))

Maybe I should tell you the closure construction in some more detail:

As soon as you add a new node,

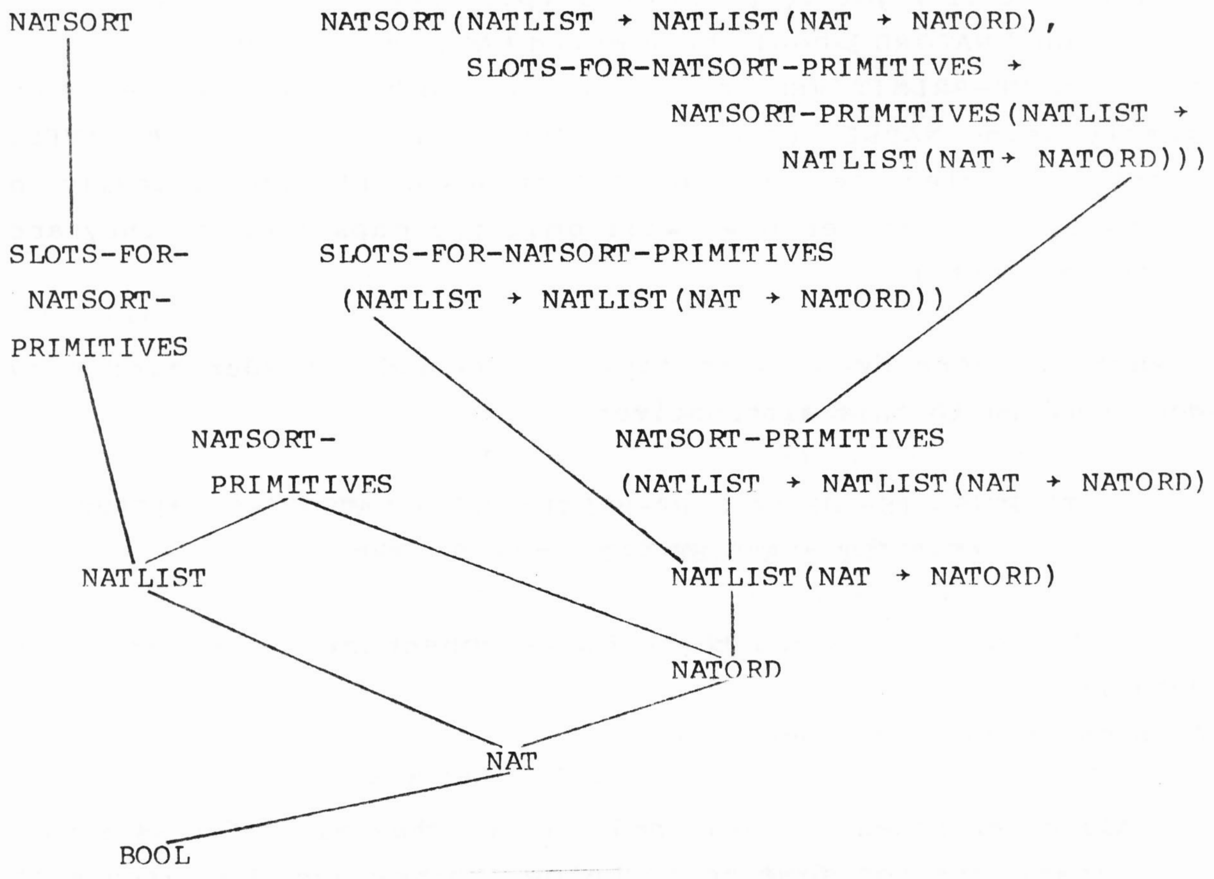
1. All other nodes are examined whether they might fit as actual parameters for some nodes below the new one. For each such application a node is created on top of the actual parameters and the nodes which remain unsubstituted.

2. Vice versa all nodes explicitly entered by the user are

checked whether the new node might fit as actual parameter for a node below themselves. Again for each such application a new node is created.

3. Step 2 is repeated for all nodes introduced by the closure construction.

It doesn´t matter that the last step produces an infinite number of application nodes, since each node actually uses only a finite number of other nodes.
Here is a relevant section of your hierarchy:



"Tremendous." The datatypist was overwhelmed. "But is there no way to avoid these huge application terms?". "Surely, for example both the application terms

```
    NATSORT(NATLIST → NATLIST(NAT → NATORD),
            SLOTS-FOR-NATSORT-PRIMITIVES
            →NATSORT-PRIMITIVES(NATLIST → NATLIST(NAT → NATORD)))
```

and

```
    NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES → NATSORT-PRIMITIVES)
           (NATLIST → NATLIST(NAT → NATORD))
```

denote the same hierarchical specification. The first, longer term is in normal form while the other one is more readable since it is shorter, and better reveals its historical development. You can transform the shorter term into the longer one by two normalization rules. The first rule

```
     S(F1 → A1) = S(F1 → A1, F2 → F2(F1 → A1))
                  if S uses F2 uses F1
```

makes implicit parameters explicit.
This rule repeatedly applied to the shorter term produces

```
     NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES → NATSORT-PRIMITIVES)
            (NATLIST → NATLIST(NAT → NATORD),
               NATSORT-PRIMITIVES → NATSORT-PRIMITIVES
                 (NATLIST → NATLIST(NAT → NATORD))).
```

To transform this term into its normal form, we need a rule that allows to contract replacement sequences where an actual parameter of the first replacement is considered as a formal parameter in the second one:

```
     S(F1 → A1) (F2 → A2,A1 → A1´) = S(F2 →A2,F1 → A1´)
                  if S uses F2.
```

Here I have another pair:

```
     NATSORT(NAT → NATORD,
             NATLIST → NATLIST(NAT → NATORD),
             SLOTS-FOR-NATSORT-PRIMITIVES
```

```
→ SLOTS-FOR-NATSORT-PRIMITIVES
      (NAT → NATORD,
       NATLIST → NATLIST(NAT → NATORD)))
```

is in normal form. According to the first rule, it can be abbreviated to

```
NATSORT(NAT → NATORD).
```

This very short term is an indirect application term, since the nodes between NAT and NATSORT do not occur (explicitly) as parameters. So you see, usually every term in normal form can be avoided by an equivalent one that is easier to read. However, the normal forms are technically convenient in order to construct the closure of a hierarchy since they neither contain any implicit parameters nor replacement sequences."

"All right", resumed the datatypist, " as a practical man I shall tell everybody who wants to sort lists of natural numbers that he has to use:

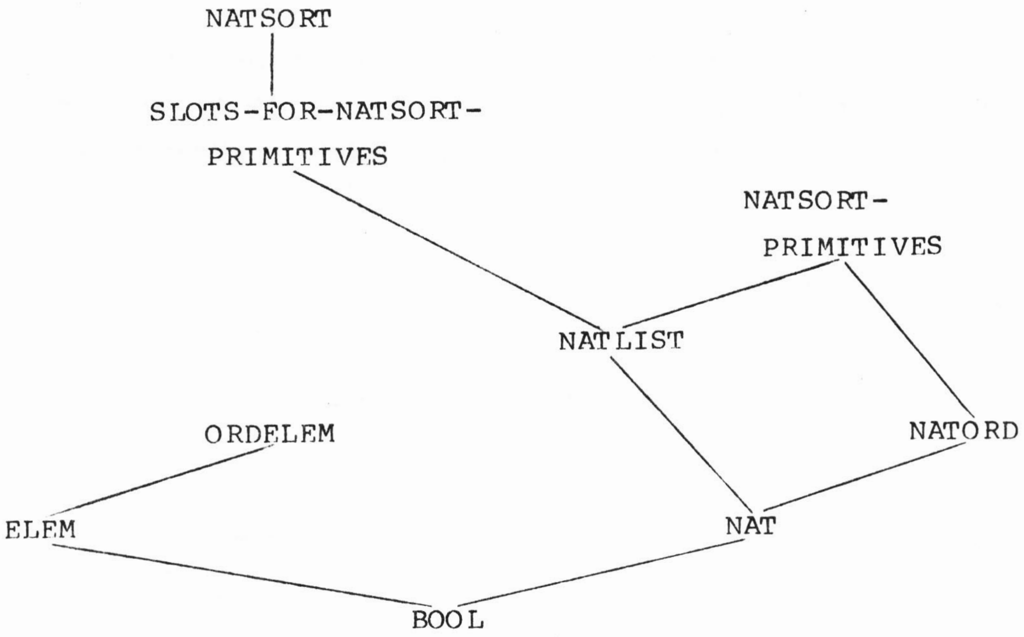
```
NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES → NATSORT-PRIMITIVES)."
```

Our people used this sorting facility happily ever after and, grateful as they were, the datatypist was advanced to a chief datatypist.

3.2 A chief datatypist sorts arbitrary lists

Our story would have ended here had not a new need arisen: "We want to sort any lists."
The chief datatypist had a close look at NATLIST and recognized that its LIST-part depended on nothing more than some sort elem with its equality-relation. These minimal requirements are expressed in the specification ELEM on top of BOOL. Similarly, the NATSORT-PRIMITIVES minimally required a linear ordering of

sort elem, which is expressed in the specification ORDELEM on top of ELEM. He presented these extensions of the hierarchy to the aspikialist:



"Last time we met you shocked me with all those implicit nodes. Today our roles are reversed: I need even more nodes than your application nodes! More precisely, I need a specification like NATSORT but NAT shall be replaced by ELEM and NATORD by ORDELEM." "I´m not shocked at all", replied the aspikialist, " nor can I help you. Your problem has been recognized and its solution will be forwarded to the next release of ASPIK. The new version will support so called generalization or abstraction terms. The one you need might look like

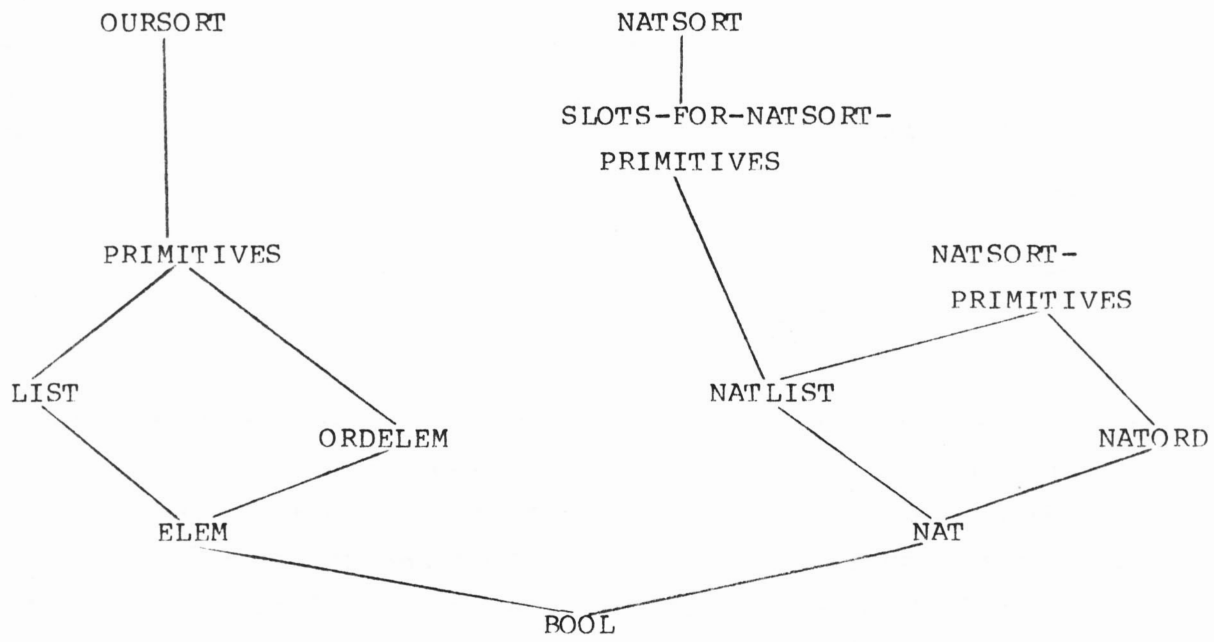
NATSORT(NAT ← ELEM, NATORD ← ORDELEM).

In some way, abstraction terms are inverse to application terms. So you could re-apply your term

NATSORT(NAT ← ELEM, NATSORT ← ORDELEM)
(ELEM → NAT, ORDELEM → NATORD)

and get back NATSORT. Thus generalization terms can be handled symmetrically to application terms, except that mixed terms must be coped with. Yet for the time being I must ask you to simulate the generalization by handmade specifications."

Thus the chief datatypist created an abstraction LIST from NATLIST, used it in an abstraction PRIMITIVES from NATSORT-PRIMITIVES, which in turn was used by an abstraction OURSORT from NATSORT.



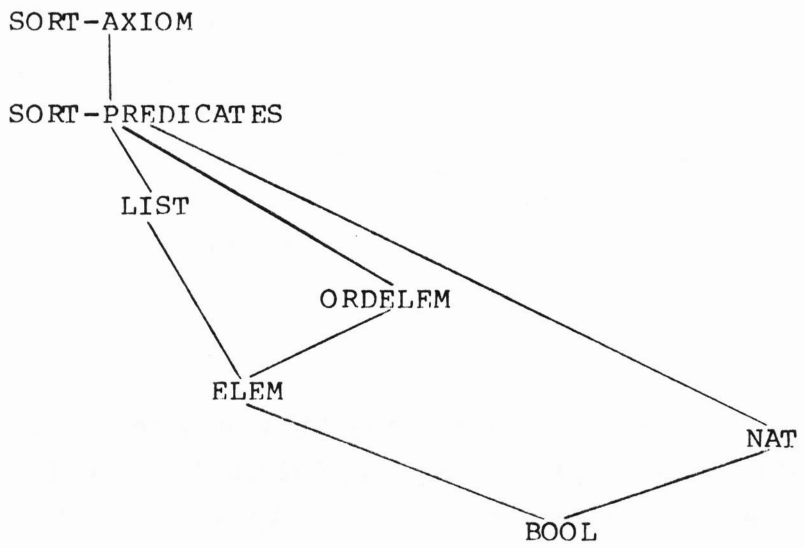
He returned the new hierarchy to his people and instructed them: "All you do is specify your concrete elements with their equality relation and a linear ordering. Then you can use

OURSORT(ELEM → your concrete elements,
ORDERELEM → the specification with your concrete ordering)."

This worked out fine, and so the chief datatypist was appointed chief abstract datatypist.

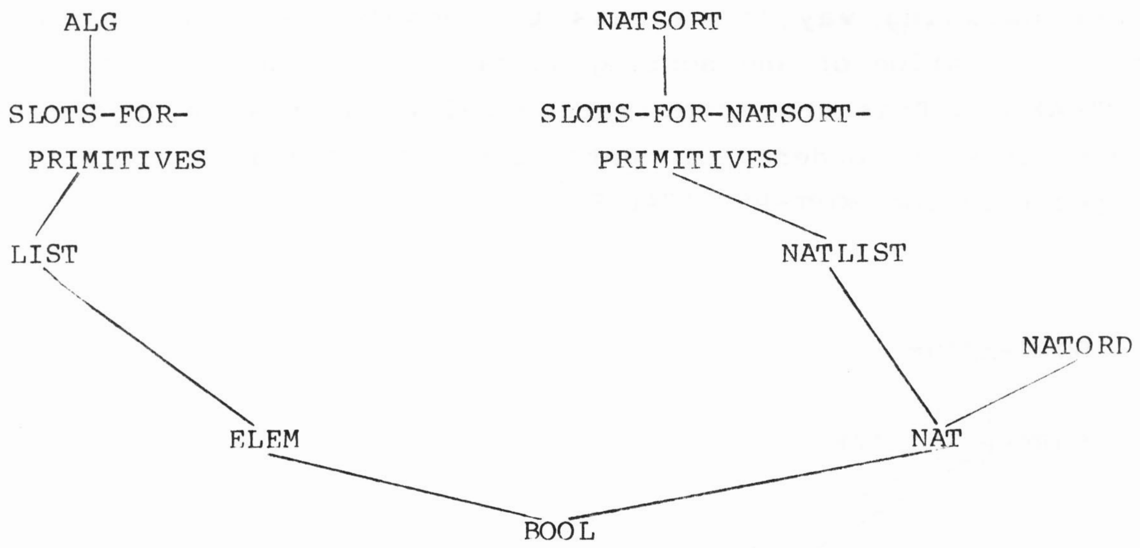
3.3 A chief abstract datatypist sorts systematically

Even now our story has not yet reached its end, since among the increasing number of OURSORT applications some were criticized of being too slow. Faster sorting algorithms were required.
To solve the sorting problem once and for ever, the chief abstract datatypist started a fresh, systematic approach. The least obliging way to express the problem was an axiomatic characterization of any sorting operation in a new specification SORT-AXIOM. This characterization relied upon some predicates about lists and ordered elements also expressed axiomatically in a specification SORT-PREDICATES.

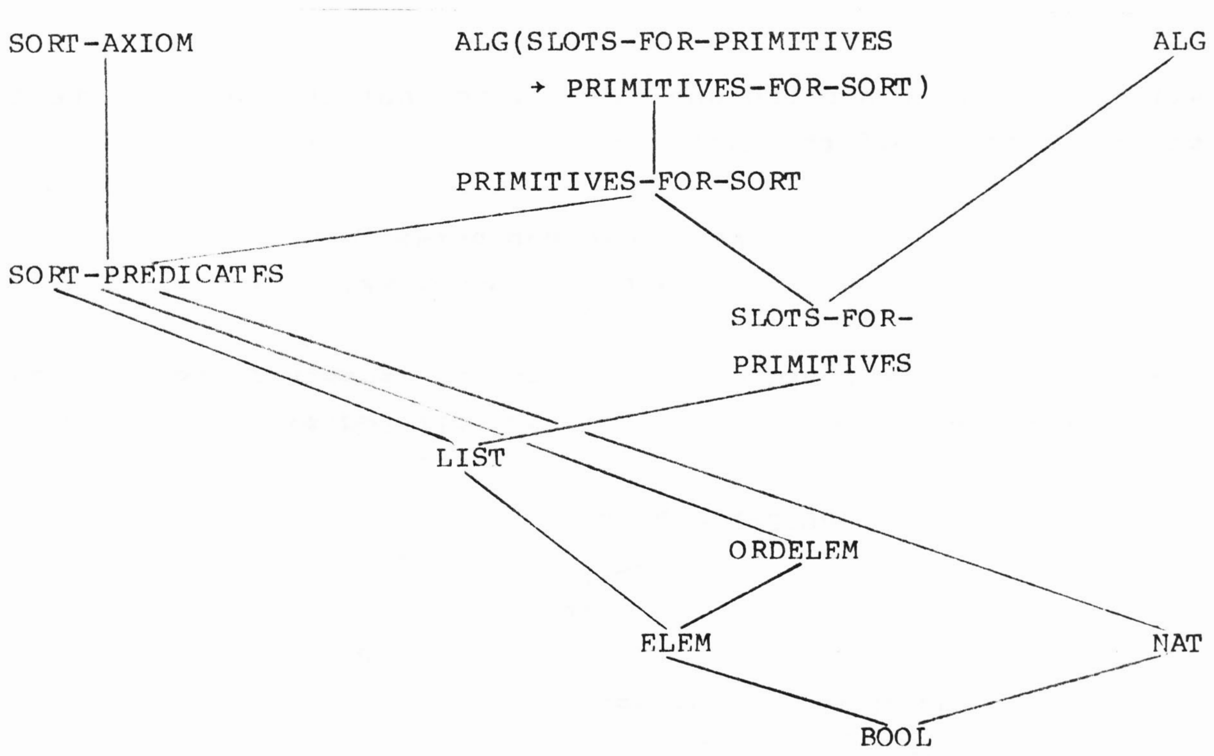


The various sorting methods were supposed to be refinements of SORT-AXIOM. More precisely, SORT-AXIOM as a formal parameter should be replaceable by such a specification providing some sorting algorithm as actual parameter.

First, he wanted to arrive at a very general algorithmic refinement of SORT-AXIOM. Reviewing his hierarchy he found that the NATSORT-algorithm using the SLOTS-FOR-NATSORT-PRIMITIVES was just the right thing to start with. Both specifications could be adapted to the current situation by generalizing them to use ELEM instead of NAT and LIST instead of NATLIST. Simulating the generalization by hand he obtained the specifications ALG with SLOTS-FOR-PRIMITIVES below it.



To get a refinement of SORT-AXIOM he added a specification PRIMITIVES-FOR-SORT on top of SLOTS-FOR-PRIMITIVES and SORT-PREDICATES. It contained axioms to admit only such primitives that the algorithm became in fact a sorting algorithm. The application ALG(SLOTS-FOR-PRIMITIVES → PRIMITIVES-FOR-SORT) constituted the first refinement of SORT-AXIOM.

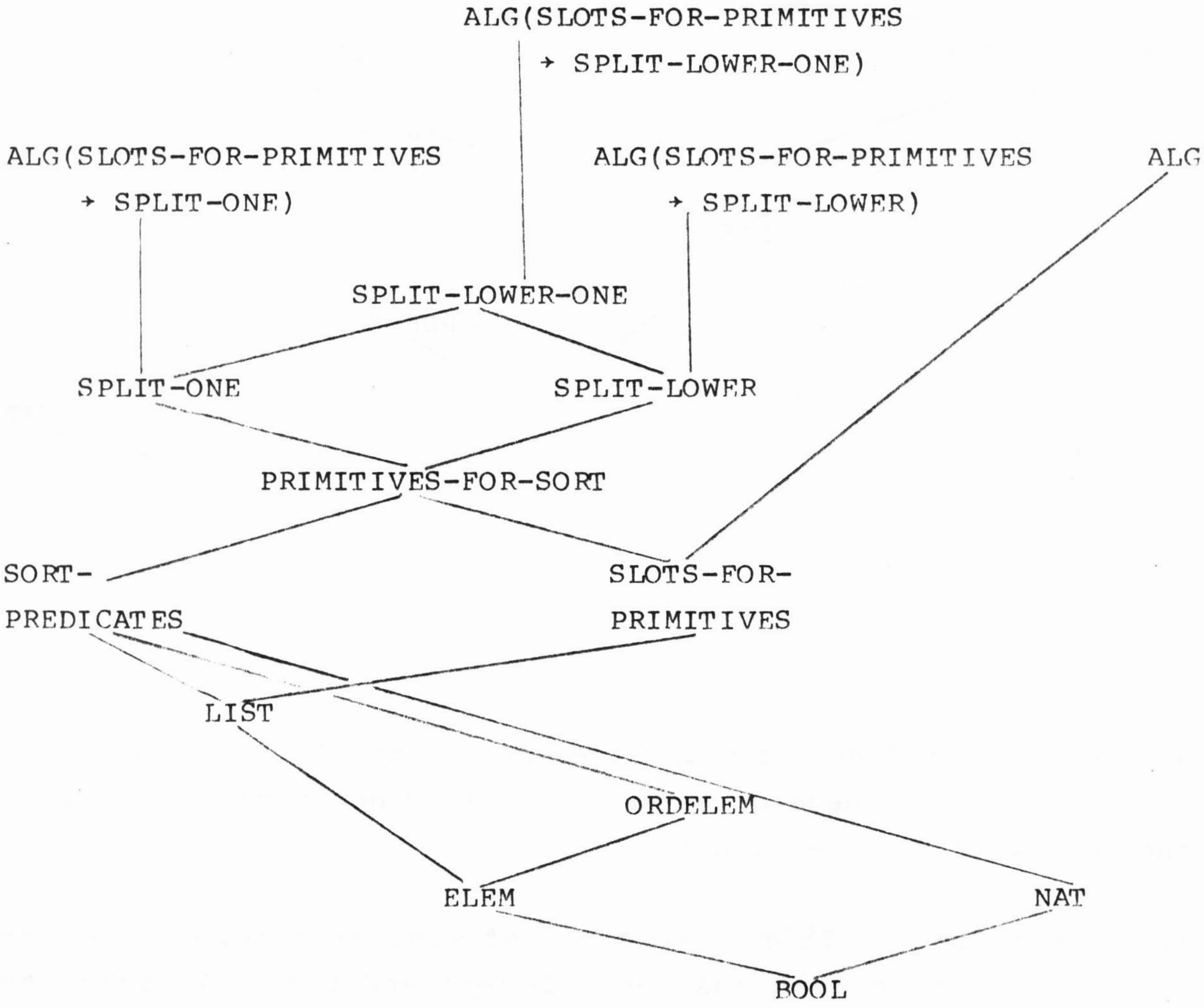


Now he had to refine the primitive operations. He remembered that it is possible to develop several sorting algorithms by answering the following two questions:

1. Is the list split up into parts of similar size, or does the first part contain just one element and the other part the remainder of the list?

2. Is the comparison of the elements done when splitting up the list, or is it done when combining the two sorted parts?

Accordingly, the chief abstract datatypist added two specifications SPLIT-ONE and SPLIT-LOWER to his hierarchy on top of PRIMITIVES-FOR-SORT. SPLIT-ONE was expressing the choice to split up the list into one element and the rest, while SPLIT-LOWER was an axiomatic characterization of all sorting algorithms doing the comparison when splitting up the list. The two specifications were combined in a third one expressing that the least element was to be split off the list.



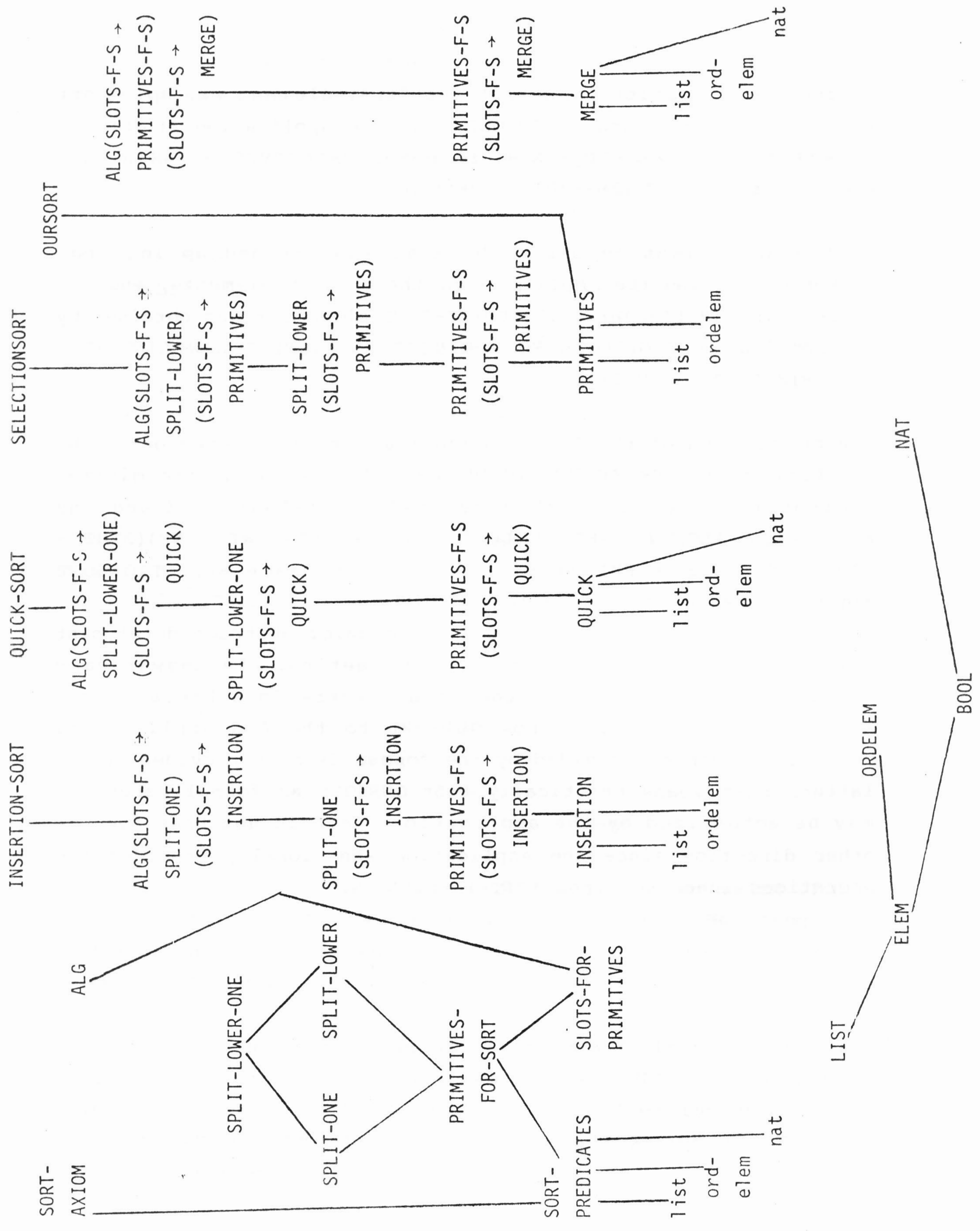
So he came up with four possibilities for sorting by using SPLIT-ONE, SPLIT-LOWER, SPLIT-LOWER-ONE or none of them. He recognized that the corresponding sorting algorithms were Insertionsort, Quicksort, Selectionsort and Mergesort.

For Mergesort he wrote a specification MERGE. It split the list in the middle and therefore refined neither SPLIT-ONE nor SPLIT-LOWER. He obtained the MERGE-SORT algorithm from the application ALG(SLOTS-FOR-PRIMITIVES → PRIMITIVES-FOR-SORT)(SLOTS-FOR-PRIMITIVES → MERGE).

In the specification INSERTION the first element was split off the list thus refining SPLIT-ONE, and the application ALG(SLOTS-FOR-PRIMITIVES → SPLIT-ONE)(SLOTS-FOR-PRIMITVES → INSERTION) yielded the INSERTION-SORT algorithm.

In the specification QUICK the list was divided up into two halves containing the smaller resp. the greater elements, and was so refining SPLIT-LOWER. The QUICK-SORT algorithm was obtained by the application ALG(SLOTS-FOR-PRIMITIVES → SPLIT-LOWER)(SLOTS-FOR-PRIMITIVES → QUICK).

The primitives of the Selectionsort algorithm turned out to be identical to the PRIMITIVES of OURSORT. As they split the minimal element off the list, they refined SPLIT-LOWER-ONE and the application ALG(SLOTS-FOR-PRIMITIVES → SPLIT-LOWER-ONE)(SLOTS-FOR-PRIMITIVES → PRIMITIVES) yielded the same SELECTIONSORT algorithm as the specification OURSORT.
"Can you tell me the difference?", the chief abstract datatypist asked the aspikialist at their next meeting. The answer came promptly: "Nothing easier than that. There is a hierarchical specification morphism from OURSORT to the ALG application, because everything provided by the former is also provided by the latter. That means practically that OURSORT as formal parameter may be actualized by the application. This is not true for the other direction since the application additionally provides the operations inherited from SORT-PREDICATES."


SORT-AXIOM
ALG
SPLIT-LOWER-ONE
SPLIT-LOWER
SPLIT-ONE
PRIMITIVES-FOR-SORT
SLOTS-FOR-PRIMITIVES
SORT-PREDICATES
list
ord-elem
nat
INSERTION-SORT
ALG(SLOTS-F-S → SPLIT-ONE) (SLOTS-F-S → INSERTION)
SPLIT-ONE (SLOTS-F-S → INSERTION)
PRIMITIVES-F-S (SLOTS-F-S → INSERTION)
INSERTION
ordelem
QUICK-SORT
ALG(SLOTS-F-S → SPLIT-LOWER-ONE) (SLOTS-F-S → QUICK)
SPLIT-LOWER-ONE (SLOTS-F-S → QUICK)
PRIMITIVES-F-S (SLOTS-F-S → QUICK)
QUICK
SELECTIONSORT
ALG(SLOTS-F-S → SPLIT-LOWER) (SLOTS-F-S → PRIMITIVES)
SPLIT-LOWER (SLOTS-F-S → PRIMITIVES)
PRIMITIVES-F-S (SLOTS-F-S → PRIMITIVES)
PRIMITIVES
OURSORT
ALG(SLOTS-F-S → PRIMITIVES-F-S) (SLOTS-F-S → MERGE)
PRIMITIVES-F-S (SLOTS-F-S → MERGE)
MERGE
LIST
ELEM
ORDELEM
BOOL
NAT

"Very impressive, your old little hierarchy", the aspikialist commented on the hierarchy of sorting methods. "And I´m happy to see that you´ve become an expert in the laws of application terms." "Have I ?", the chief abstract datatypist wondered. "At least intuitively. Look, for instance you put

```
    ALG(SLOTS-FOR-PRIMITIVES → SPLIT-ONE)
       (SLOTS-FOR-PRIMITIVES → INSERTION)
```

on top of

```
    SPLIT-ONE(SLOTS-FOR-PRIMITIVES → INSERTION)
```

on top of

```
    PRIMITIVES-FOR-SORT(SLOTS-FOR-PRIMITIVES → INSERTION).
```

But without transforming the SPLIT-ONE application into its normal form you can hardly see that it uses the PRIMITIVES-FOR-SORT application:

```
    SPLIT-ONE(SOLTS-FOR-PRIMITIVES → INSERTION) =
    SPLIT-ONE(SLOTS-FOR-PRIMITIVES → INSERTION,
              PRIMITIVES-FOR-SORT
              → PRIMITIVES-FOR-SORT(SLOTS-FOR-PRIMITIVES
                                   → INSERTION))
```

according to the rule:

```
    S(F1 → A1) =  S(F1 → A1, F2 → F2(F1 → A1))
                    if S uses F2 uses F1
```

which allows to make implicit parameters explicit, as you may recollect.
In order to see that the ALG application uses the SPLIT application, you have to contract the two replacements:

```
    ALG(SLOTS-FOR-PRIMITIVES → SPLIT-ONE)
       (SLOTS-FOR-PRIMITIVES → INSERTION) =
    ALG(SLOTS-FOR-PRIMITIVES
        → SPLIT-ONE(SLOTS-FOR-PRIMITIVES
```

```
→ INSERTION))
```

according to a simpler version of the contraction rule:

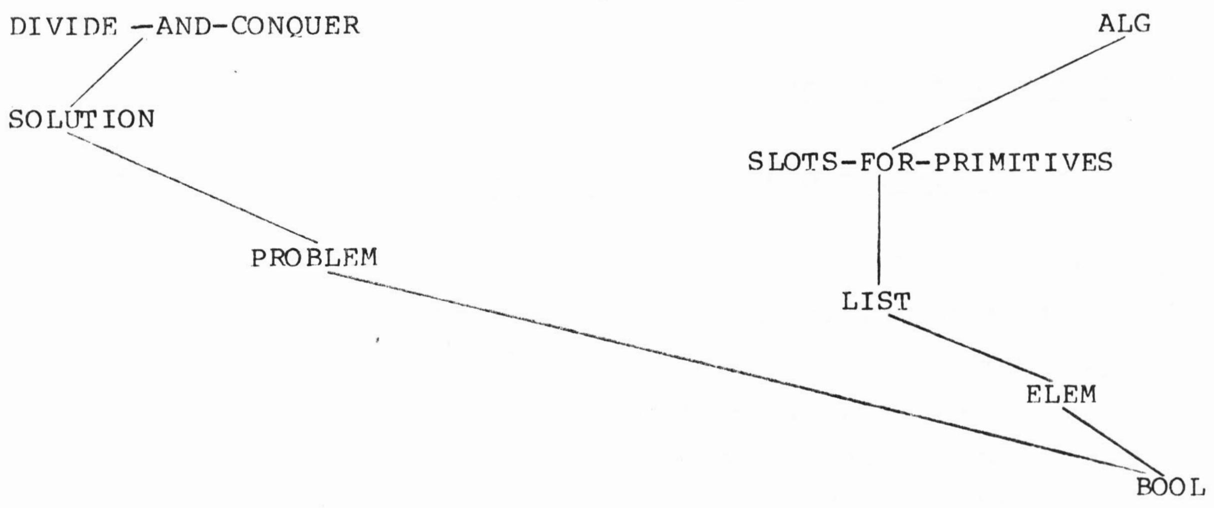
```
S(F1 → A1)(F1 → A2) =  S(F1 → A1(F1 → A2))
                           if A1 uses A2.
```

The hierarchy of sorting methods satisfied everybody and so the chief abstract datatypist was appointed Master in ASPIK.

3.4. A Master in ASPIK divides and conquers

For some months this was the end of the story - though not of the hierarchy which was continuously enlarged by ELEM and ORDELEM refinements. So what happened? There was a masters meeting at Passau attended by our Master in ASPIK. There, listening to the talk of master Veloso, he recognized that ALG could be generalized to the Divide-and-Conquer paradigm. He just had to devise a signature for decomposable PROBLEMs on top of BOOL and a signature for composable SOLUTIONs on top of PROBLEM. Then the DIVIDE-AND-CONQUER approach could be abstracted as

```
ALG(SLOTS-FOR-PRIMITIVES ← PROBLEM
      sorts  list    =  problem
      ops    part1   =  subproblem1
             part2   =  subproblem2
             simple? =  simple?,
    SLOTS-FOR-PRIMITIVES ← SOLUTION
      sorts  list         =  solution
      ops    simple-sort  =  simple-sort
             combine      =  combine)
```



By this uttermost generalization the Master in ASPIK rose to the rank of a Divisor and Conqueror in ASPIK and - finally - they all lived happily ever after.

4. The specifications

4.1 The specifications of the datatypist

```
spec BOOL
/* standard definition of the booleans */
   sorts bool;
   ops   true, false : --> bool
         _and_, _or_ : bool bool --> bool
         not : --> bool;
spec body
   constructors  true, false;
   define ops
      b1 and b2 = case b1 is *true: b2
                             *false: false
                  esac
      b1 or b2 = case b1 is *true: true
                            *false: b2
                 esac
      not(b) = case b is *true: false
                         *false: true
               esac
endspec
```

```
spec NAT
/* standard definition of the natural numbers */
   use   BOOL;
   sorts nat;
   ops   0 : --> nat
         succ, pred : nat --> nat
         _+_, _-_ : nat nat --> nat
         eq-nat : nat nat --> bool;
```

```
spec body
   constructors 0, succ;
   define ops
      n1 + n2 = case n1 is *0: n2
                           *succ(n): n + succ(n2)
                esac
      pred(n) = case n is *0: error-nat
                          *succ(n1): n1
                esac
      n1 - n2 = case n2 is *0: n1
                           *succ(n): pred(n1) - n
                esac
endspec


spec NATLIST
/* lists of natural numbers */
   use   NAT;
   sorts list;
   ops   empty : --> list
         put : nat list --> list
         first : list --> nat
         rest : list --> list
         append : list list --> list
         empty?, simple? : list --> bool
         in? : nat list --> bool;
spec body
   constructors empty, put;
   define ops
      first(l) = case l is *empty: error-nat
                           *put(n,l1): n
                 esac
      rest(l) = case l is *empty: error-list
                          *put(n,l1): l1
                esac
```

```
      append(l,l2) = case l is *empty: l2
                                  *put(n,l1): put(n,append(l1,l2))
                     esac
      empty?(l) = case l is *empty: true
                            otherwise: false
                  esac
      simple?(l) =  case l  is *empty: true
                                *put(n,l1): empty?(l1)
                    esac
      in?(n,l) = case l is *empty: false
                           *put(nl,l1): eq-nat(n,nl) or in?(n,l1)
                 esac
endspec


spec SLOTS-FOR-NATSORT-PRIMITIVES
/* names for NATSORT`s primitive operations */
   use   NATLIST;
   ops   part1, part2 : list --> list
         combine : list list --> list
         simple-sort : list --> list;
endspec


spec NATSORT
/* sorts lists of natural numbers (by selection) */
   use   SLOTS-FOR-NATSORT-PRIMITIVES;
   ops   sort : list --> list;
spec body
   define ops
      sort(l) = if simple?(l)
                then simple-sort(l)
                else combine(sort(part1(l)),part2(l)))
endspec
```

```
spec NATORD
/* the standard ordering of the natural numbers */
   use    NAT;
   ops    _<=_ : nat nat --> bool;
spec body
   define ops
      n1 <= n2 = if eq-nat(n1,0)
                 then true
                 elsif eq-nat(n2,0)
                 then false
                 else pred(n1) <= pred(n2)
endspec


spec NATSORT-PRIMITIVES
/* NATSORT`s primitive operations */
   use    NATLIST, NATORD;
   ops    min, allbutmin : list list --> list
          putmin : list list --> list
          simple-sort : list --> list;
spec body
   private ops min-elem : list --> nat;
   define ops
      min(l) = put(min-elem(l),empty)
      allbutmin(l) = if eq-nat(first(l),min-elem(l))
                     then rest(l)
                     else put(first(l),allbutmin(rest(l)))
      min-elem(l)  = if simple?(l)
                     then first(l)
                     else let m = min-elem(rest(l)) in
                          if first(l) <= m
                          then first(l)
                          else m
endspec
```

```
spec NATSORT(SLOTS-FOR-NATSORT-PRIMITIVES --> NATSORT-PRIMITIVES
             ops part1 = min
                 part2 = allbutmin
                 combine = putmin
                 simple-sort = simple-sort)
/* sorts lists of natural numbers (by selection) */
   use    NATSORT-PRIMITIVES;
   ops    sort : list --> list;
spec body
   define ops
      sort(l) = if simple?(l)
                then simple-sort(l)
                else putmin(sort(min(l)),sort(allbutmin(l)))
endspec
```

4.2 The specifications of the chief datatypist

```
spec ELEM
/* a sort with its equality */
   use    BOOl;
   sorts  elem;
   ops    eq-elem : elem elem --> bool;
endspec
```

```
spec ORDELEM
/* a reflexive, linear ordering on elem */
   use    ELEM;
   ops    _<=_ : elem elem --> bool;
   props  all x,y : x <= x  =  true
                    if x <= y  =  true, y <= z  =  true
                    then x <= z = true
                    if x <= y  =  true, y <= x  = true
                    then x = y;
endspec
```

```
spec LIST
/* lists of anything.
   It simulates the abstraction: NATLIST(NAT <-- ELEM). */
   use   ELEM;
   sorts list;
   ops   empty : --> list
         put : elem list --> list
         first : list --> elem
         rest : list --> list
         append : list list --> list
         empty?, simple? : list --> bool
         in? : elem list --> bool;
spec body
   constructors empty, put;
   define ops
      first(l) = case l is *empty: error-elem
                           *put(n,ll): n
                 esac
      rest(l) = case l is *empty: error-list
                          *put(n,ll): ll
                esac
      append(l,l2) = case l is *empty: l2
                               *put(n,ll): put(n,append(ll,l2))
                     esac
      empty?(l) = case l is *empty: true
                            otherwise: false
                  esac
      simple?(l) =  case l  is *empty: true
                               *put(n,ll): empty?(ll)
                    esac
      in?(n,l) = case l is *empty: false
                           *put(nl,ll): eq-elem(n,nl) or in?(n,ll)
                 esac
endspec
```

```
spec PRIMITIVES
/* OURSORT`s primitive operations.
   It simulates the abstraction:  NATSORT(NAT <-- ELEM,
                                          NATLIST <-- LIST,
                                          NATORD <-- ORDELEM). */

   use   LIST, ORDELEM;
   ops   min, allbutmin : list list --> list
         putmin : list list --> list
         simple-sort : list --> list;
 spec body
   private ops min-elem : list --> elem;
   define ops
      min(l) = put(min-elem(l),empty)
      allbutmin(l) = if eq-elem(first(l),min-elem(l))
                     then rest(l)
                     else put(first(l),allbutmin(rest(l)))
      min-elem(l)  = if simple?(l)
                     then first(l)
                     else let m = min-elem(rest(l)) in
                          if first(l) <= m
                          then first(l)
                          else m
endspec


spec OURSORT
/* sorts lists of anything (by selection).
   It simulates the abstration: NATSORT(NAT <-- ELEM,
                                        NATLIST <-- LIST,
                                        NATORD <-- ORDELEM,
                                        NATSORT-PRIMITIVES<--
                                                   PRIMITIVES). */

   use   PRIMITIVES;
   ops   sort : list --> list;
 spec body
   define ops
```

```
      sort(l) = if simple?(l)
                then simple-sort(l)
                else combine(sort(part1(l)),sort(part2(l)))
endspec
```

4.3 The specifications of the chief abstract datatypist

```
spec SORT-PREDICATES
/* predicates needed in SORT-AXIOM */
   use    LIST, ORDELEM, NAT;
   ops    permutation? : list list --> bool
          sorted? : list --> bool
          occurrences : elem list --> nat
          length : list --> nat;
   props all e, e1, e2, l,l1, l2 :
              occurrences(e,empty) = 0
              occurrences(e,put(e,l)) = succ(occurrences(e,l))
              if e1 =/= e2
              then occurrences(e1,put(e2,l)) = occurrences(e1,l)
              permutation?(l1,l2) =
                 eq-nat(occurrences(e,l1),occurrences(e,l2))
              if simple?(l) = true then sorted?(l) = true
              if simple?(l)  =  false ,
                 first(l) <= first(rest(l)) = true,
                 sorted?(rest(l)) = true
              then sorted?(l) = true
              length(empty) = 0
              length(put(e,l)) = succ(length(l));
endspec


spec SORT-AXIOM
/* axiomatic definition of all sort operations */
   use    SORT-PREDICATES;
   ops    sort : list --> list;
```

```
   props all l : permutation?(sort(l)) = true
                 sorted?(sort(l)) = true;
endspec


spec SLOTS-FOR-PRIMITIVES
/* names for SORT`s primitive operations
   It simulates the abstraction:
   SLOTS-FOR-NATSORT-PRIMITIVES(NAT <-- ELEM,
                                NATLIST <-- LIST). */
   use   LIST;
   ops   part1, part2 : list --> list
         combine : list list --> list
         simple-sort : list --> list;
endspec


spec ALG
/* sorts lists of anything algorithmically.
   It simulates the abstraction:
   NATSORT(NAT <-- ELEM,
           NATLIST <-- LIST,
           NATORD <-- ORDELEM,
           SLOTS-FOR-NATSORT-PRIMITIVES<-- SLOTS-FOR-PRIMITIVES)
                                                              */
   use   SLOTS-FOR-PRIMITIVES;
   ops   sort : list --> list;
spec body
   define ops
      sort(l) = if simple?(l)
                then simple-sort(l)
                else combine(sort(part1(l)),sort(part2(l)))
endspec
```

```
spec PRIMITIVES-FOR-SORT
/* axiomatic characterization of ALG`s primitive operations */
   use    SLOTS-FOR-PRIMITIVES, SORT-PREDICATES;
   props all l: if simple?(l) = true
                then sorted?(simple-sort(l)) = true
                if simple?(l) = false,
                   sorted?(part1(l)) = true,
                   sorted?(part2(l)) = true,
                then sorted?(combine(part1(l),part2(l))) = true
                if simple?(l) = false
                then permutation?(combine(part1(l),part2(l)),l)
                         = true
                if simple?(l) = true
                then permutation?(simple-sort(l),l) = true
                if simple?(l) = false
                then length(part1(l)) <= pred(length(l)) = true
                if simple?(l) = false
                then length(part2(l)) <= pred(length(l)) = true;
endspec


spec SPLIT-ONE
/* one element must be split off */
   use    PRIMITIVES-FOR-SORT;
   props all l, l1, l2 : if simple?(l) = false
                         then simple?(part1(l)) = true;
endspec


spec SPLIT-LOWER
/* the list must be split into a lower and an upper half */
   use    PRIMITIVES-FOR-SORT;
   props all l, x, y : if simple?(l) = false,
                          in?(x,part1(l)) = true,
                          in?(y,part2(l)) = true,
                       then x <= y = true;
```

```
endspec


spec SPLIT-LOWER-ONE
/* the minimal element must be split off */
   use    SPLIT-ONE, SPLIT-LOWER;
endspec


spec MERGE
/* primitive operations of the merge-sort algorithm */
   use    LIST, ORDELEM, NAT;
   ops    merge : list list --> list
          firsthalf, secondhalf : list --> list
          simple-merge : list --> list;
spec body
   private ops secondhalf1, difference : list list --> list
                half-of : nat --> nat
                length : list --> nat;
   define ops
      merge(l,m) = if empty?(l)
                   then m
                   elsif empty?(m)
                   then l
                   elsif first(l) <= first(m)
                   then put(first(l),merge(rest(l),m))
                   else put(first(m),merge(l,rest(m))
      secondhalf(l) = secondhalf1(l,l)
      secondhalf1(l1,l2) = if length(l2) <= half-of(length(l1))
                           then l2
                           else secondhalf1(l1,rest(l2))
      difference(l1,l2) = if length(l1) <= length(l2)
                          then empty
                          else put(first(l1),
                                   difference(rest(l1),l2))
      half-of(n) = if n <= succ(succ(0))
```

```
                    then succ(0)
                    else succ(half-of(n - succ(succ(0))))
      length(l) = if empty?(l)
                  then 0
                  else succ(length(rest(l)))
endspec


spec MERGE-SORT
/* the merge-sort algorithm */
   use    ALG(SLOTS-FOR-PRIMITIVES --> PRIMITIVES-FOR-SORT)
             (SLOTS-FOR-PRIMITIVES --> MERGE
              ops part1 = firsthalf
                  part2 = secondhalf
                  combine = merge
                  simple-sort = simple-merge);
endspec


spec INSERTION
/* primitive operations for the insertion-sort algorithm */
   use    LIST, ORDELEM;
   ops    list-of-first : list --> list
          insert : list list --> list
          simple-insert : list --> list;
spec body
   define ops
      list-of-first(l) = put(first(l),empty)
      insert(ll,l) = if empty?(ll)
                     then l
                     elsif first(ll) <= first(l)
                     then put(first(ll),l)
                     else put(first(l),insert(ll,rest(l)))
      simple-insert(l) = l
endspec
```

```
spec INSERTION-SORT
/* the insertion-sort algorithm */
   use    ALG(SLOTS-FOR-PRIMITIVES --> PRIMITIVES-FOR-SORT)
              (SLOTS-FOR-PRIMITIVES --> INSERTION
               ops part1 = list-of-first
                   part2 = rest
                   combine = insert
                   simple-sort = simple-insert);
endspec


spec QUICK
/* primitive operations of the quick-sort algorithm */
   use    LIST, ORDELEM, NAT;
   ops    lower-part, upper-part : list --> list
          simple-quick : list --> list;
spec body
    private ops half-of : nat --> nat
                lower-part1, upper-part1 : list elem --> list
                middle-elem :list --> elem
                nth-elem : list nat --> elem
                length : list --> nat;
 define ops
      lower-part(l) = lower-part1(l,middle-elem(l))
      upper-part(l) = upper-part1(l,middle-elem(l))
      simple-quick(l) = l
      middle-elem(l) = nth-elem(l,half-of(length(l)))
      half-of(n) = if n <= succ(succ(0))
                   then succ(0)
                   else succ(half-of(n - succ(succ(0))))
      length(l) = if empty?(l)
                  then 0
                  else succ(length(rest(l)))
      nth-elem(l,n) = if n <= succ(0)
                      then first(l)
                      else nth-elem(rest(l),pred(n))
```

```
      lower-partl(l,m) = if empty?(l)
                         then empty
                         elsif first(l) <= m
                         then put(first(l),
                                  lower-partl(rest(l),m))
                         else lower-partl(rest(l),m)
      upper-partl(l,m) = if empty?(l)
                         then empty
                         elsif not(first(l)) <= m
                         then put(first(l),
                                  upper-partl(rest(l),m))
                         else upper-partl(rest(l),m)
endspec


spec QUICK-SORT
/* the quick-sort algorithm */
   use   ALG(SLOTS-FOR-PRIMITIVES --> PRIMITIVES-FOR-SORT)
             (SLOTS-FOR-PRIMITIVES --> QUICK
              ops partl = lower-part
                  part2 = upper-part
                  combine = append
                  simple-sort = simple-quick);
endspec


spec SELECTION-SORT
/* the selection-sort algorithm */
   use   ALG(SLOTS-FOR-PRIMITIVES --> PRIMITIVES-FOR-SORT)
             (SLOTS-FOR-PRIMITIVES --> PRIMITIVES
              ops partl = min
                  part2 = allbutmin
                  combine = append
                  simple-sort = simple-sort);
endspec
```

4.4 The specifications of the Master in ASPIK

```
spec PROBLEM
/* decomposable problems */
   use    BOOL;
   sorts  problem;
   ops    subproblem1, subproblem2 : problem --> problem
          simple-problem? : problem --> bool;
endspec


spec SOLUTION
/* composable solutions */
   use    PROBLEM;
   sorts  solution;
   ops    simple-solution : problem --> solution
          combine-solutions : solution solution --> solution;
endspec


spec DIVIDE-AND-CONQUER
/* the divide-and-conquer approach.
   Apart from renaming sort to solve, it simulates the abstraction:
   ALG(SLOTS-FOR-PRIMITIVES <-- PROBLEM
       sorts list = problem
       ops   part1 = subproblem1
             part2 = subproblem2
             simple? = simple-problem?
       SLOTS-FOR-PRIMITIVES <-- SOLUTION
       sorts list = solution
       ops   simple-sort = simple-solution
             combine = combine-solutions). */
   use    SOLUTION;
   ops    solve;
spec body
   define ops
```

```
     solve(p) = if simple-problem?(p)
                then simple-solution(p)
                else combine-solutions(solve(subproblem1(p)),
                                       solve(subproblem2(p)))
endspec
```

5. The moral

The moral of the story is:
In ASPIK you need not be afraid of software cycles.

- If you have a model in mind right from the beginning it´s ok to write it down at once.
- If you recognize parts of your hierarchy as parameters later on it´s not too late. ASPIK´s parameterization-by-use allows you to introcuce your parameters as late as you need them.
- If you realize the general qualities of your specifications later rather than sooner you needn´t start afresh. You can generalize whenever you want.

References

[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.

[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.

[BV 83a] Beierle, Ch., Voß, A.: Parameterization-by-use for hierarchically structured objects. SEKI-Projekt, Memo SEKI-83-08, Univ. Kaiserslautern, FB Informatik, May 1983.

[BV 83b] Beierle, Ch., Voß, A.: Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types. SEKI-Projekt, Memo SEKI-83-07, May 1983.

[Da 76] Darlington, J.: A Synthesis of Several Sorting Algorithms, D.A.I. Research Report 23, University of Edinburgh, June 1976.

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

[EKTWW 81] Ehrig, H., Kreowski, H.-J., Thatcher, J., Wagner,E. , Wright, J.: Parameter Passing in Algebraic Specification Languages, Workshop Program Specification, Aarhus 81.

[EWT 82] Ehrig, H., Wagner, E., Thatcher, J.: Algebraic Constraints for Specifications and Canonical Form Results, TU Berlin, FB Informatik (20), Bericht Nr. 82-09, June 1982.

[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.

[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.

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

[Pa 79] Padawitz, P.: Proving the Correctness of Implementations by Exclusive Use of Term Algebras, TU Berlin, FB Informatik (20), Bericht Nr. 79-8, June 79.

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

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

[SW 82b] Sannella, D.T., Wirsing, M.: A Kernel Language for Algebraic Specificiation and Implementation, Draft, Dep. of Computer Science, University of Edinburgh, Institut f. Informatik, TU München, 1982.

[ZLT 82] Zilles, S.N., Lucas, P., Thatcher, J.W.: A Look at Algebraic Specifications, IBM Research Division, Yorktown Heights, New York, San Jose, California, Zurich, Switzerland, 1982.