Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

N.W.P. van Diepen

Implementation of modular algebraic specifications

Computer Science/Department of Software Technology Report CS-R8801 January

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Implementation of Modular Algebraic Specifications

N.W.P. van Diepen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The foundation of implementation of algebraic specifications in a modular way is investigated. Given an algebraic specification with visible and hidden signature an *observing* signature is defined. This is a part of the visible signature which is used to observe the behaviour of the implementation.
Two correctness criteria are given for the implementation with respect to the observing signature. An algebraic correctness criterion guarantees initial algebraic semantics for the specification as seen through the observing signature, while allowing freedom for other parts of the signature, to the extent that even final semantics may be used there. A functional correctness criterion allows one to prove the correctness of the implementation for one observing function in Hoare Logic. The union over all observing functions of such implementations provides an actual implementation in any programming language with semantics as described above.

1980 Mathematics Subject Classification (1985): 68N99 [software], 68Q55 [theory of computing - semantics].
1987 CR Categories: D.2.1 SOFTWARE ENGINEERING: Requirements, Specifications; D.2.2 SOFTWARE ENGINEERING: Tools and Techniques - *modules and interfaces*; D.2.m SOFTWARE ENGINEERING: Reusable software; D.3.1 PROGRAMMING LANGUAGES: Formal Definitions and Theory - *semantics*; D.3.3 PROGRAMMING LANGUAGES: Language Constructs - *modules, packages*; F.3.1 LOGICS AND MEANINGS OF PROGRAMS: Specifying and Verifying and Reasoning about Programs; F.3.2 LOGICS AND MEANINGS OF PROGRAMS: Semantics of Programming Languages - *Algebraic Approaches to Semantics*.

Key words & phrases: modular algebraic specification, implementation of algebraic specifications, functional implementation.

Note 1: Partial support has been received from the European Communities under ESPRIT project no. 348 (Generation of Interactive Programming Environments - GIPE).
Note 2: This paper has been accepted for *ESOP '88* (European Symposium on Programming), to be held at Nancy, France, on 21-25 March 1988.

1. INTRODUCTION

An algebraic specification is a mathematical structure consisting of sorts, functions (and constants) over these sorts, and equations describing the relation between the functions and constants. It is a convenient tool to specify static and dynamic semantics of programming languages, see e.g. Goguen and Meseguer ([GM82], [GM84], [MG85]) for more detail on algebraic specification, and [BHK85], [BDMW81] and [Die86] for examples. The implementation of an algebraic specification usually consists of the conversion of the equations into a *term rewriting system*, either directly or through the completion procedure of Knuth-Bendix. More details can be found in [HO80] and [ODo85]. The performance of such an implementation is rather slow in general, compared with algorithms written in conventional programming languages, while the specification must have certain properties to be implemented in this way at all. The aim of this paper is to provide another implementation strategy, based on pre- and postconditions, allowing the application of more classical programming and optimization techniques.

Report CS-R8801
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The research for this paper was motivated by ESPRIT Project 348 — Generation of Interactive Programming Environments —, in which the implementation of various aspects of programming languages from formal definitions (including modular algebraic specifications) plays a key role. The project is reported upon in [GIP86] and [GIP87]. This paper has independent significance and can be read without knowledge of the project.

1.1. Modular algebraic specifications

Algebraic specifications have been introduced to provide a description style for data types in an algebraically — mathematically — nice way. The mathematical notion of a (many-sorted) *algebra* used here is a structure consisting of carrier sets and typed functions (including constants) over these sets, together with a set of equations, specifying the behaviour of the functions. The combination of a set of sorts (the names of the carrier sets) and a set of functions (which include constants, unless stated otherwise), is called the *signature* of the algebra.

The algebraic specifications studied in this paper have additional organization primitives, prompted by both theoretical and practical considerations. Central issue is the *modular structure* imposed on the algebraic specifications. An algebraic specification can import another algebraic specification as a *module*, meaning that it adds the sorts, functions and equations of the imported specification to its own. Sorts or functions with the same name are only allowed when they are the same (they must originate in the same module), otherwise they must be renamed.

The modular approach naturally leads to two other primitives, a *parameter* mechanism, and the occurrence of *hidden* (local, auxiliary) sorts and functions. Hidden sorts and functions are used in the equations of the module in which they are defined, but they are not included in the exported or *visible* sorts and functions. Only the latter are included in the algebra associated with the module. Hidden sorts and functions make it easier to write many specifications by providing local definitions. Also, they are necessary to specify properties needing an infinite number of equations (when defined without hidden sorts and functions) in a finite way (see Bergstra and Tucker [BT83], [BT82]).

The equations used are *conditional equations*, i.e. equations which are valid only when certain conditions are satisfied. The semantics provided are of an *initial* nature, though this will be modified in the paper. Initial algebra semantics are described by 'no junk', meaning that there are no more sorts than specified and that for the sorts specified all elements can be reached via functions specified, and 'no confusion', meaning that everything which is equal in the algebra can be proved equal with the equations provided. These semantics are usually intuitively clear.

1.2. Implementation of algebraic specifications

Once an algebraic specification has been written there is no clearcut way to derive a working program from it. In general, any model of the algebra can be seen as an implementation. Certain models are preferable, though.

A strategy followed quite often to implement a model satisfying initial semantics (an initial model) is to transform the specification into a *term rewriting system*. The easiest way to do this is to give every equation a direction, say from left to right, and to view the set of directed equations as a set of *rewrite rules*, transforming one term over the signature into another. This system can be found in various places in the literature ([BK86], [DE84], [FGJM85], [GMP83], [ODo85], [Die86]), but the success of this method depends on the properties of the directed version of the (in principle undirected) set of equations, combined with the technique used for rewriting. Turning an equation around (writing B=A instead of A=B) or writing the equations in a different order may have significant consequences for the behaviour (both in speed and in termination) of the implementation, while the specification has not been changed, except textually.

An additional problem is what to do with the modular structure when fitting a modular specification in a term rewriting system. Transparent semantics can be obtained by a *normalization step* (as described by Bergstra, Heering and Klint [BHK86]), flattening all imports into one module (renaming hidden functions and sorts where necessary). The approach above can be applied to the normalized module. It may be debated whether the loss of the structure in the specification is sufficiently motivated by the transparency of

the semantics.

The present paper aims at a more module-oriented implementation, giving semantics to implement an *observing signature* (a signature through which one can observe the visible signature, of which it is a subsignature) in a functional way, using descriptions of the observing functions in *Hoare Logic* (see e.g. the text books [LS84], [Bac86]). The significance for the semantics of the import construct will be touched upon briefly. The main advantage of this approach is that it permits the implementation of modules in a, possibly low-level, efficient way from the high-level specification. This allows the construction of a library of efficient basic modules upon which more sophisticated algebraic specifications may depend.

1.3. Related work

Implementation techniques for pure initial semantics are burdened with the obligation to implement the initial algebra semantics faithfully. This generally slows down the implementation, since often an initial specification demands too much detail, as has been discussed by Baker-Finch [Bak84].

Meseguer and Goguen [MG85] also provide an implementation criterion for algebraic specifications. They focus on observable sorts, while the present paper takes a functionally oriented point of view. Their approach is a special case of the approach presented here. They retain initial algebra semantics for their specifications but loosen the restriction on the models for implementation. In the present paper the semantics are modified.

There is a strong resemblance to *abstract data type* theory as practiced in the verification of correctness of programs (cf. Jones [Jon80]). After all, an algebraic specification is a nice way to describe a data type. While the specifications look similar, the point of view is different. Constructor functions (i.e. functions describing the data type) really construct the type in algebraic specifications, while they only serve as a description tool in [Jon80].

Techniques which use *term rewriting systems* have the advantage of allowing (semi-)automatic translation schemes, but pay the price with severe restrictions on the set of equations allowed. Perhaps the overhead of a completion procedure for generation of rewrite rules is needed, e.g. the Knuth-Bendix procedure (see [HO80] and [ODo85] for more detail). The technique presented here allows for faster implementations, but does not support automatic translation.

1.4. An outline of this paper

In the next section brief introductions to both specification formalisms used (the algebraic specification formalism ASF and Hoare Logic) are given.

Section 3 starts with an example to illustrate certain disadvantages of the initial algebra approach to motivate the theoretical framework leading to an algebraic implementation notion in the second half. An example giving an implementation according to this notion follows in section 4, which may be read before section 3.2 to get the flavour, or in the order provided to convince oneself of the rigour of the approach.

The functional implementation notion is described as an extension of the algebraic notion in section 5, preceded by an example to show the insufficient strength of the latter notion for our purpose. The example of section 4 is implemented in an imperative language in the following section according to this notion.

Finally some remarks are made and areas of further research indicated in section 7.

2. THE FORMALISMS INFORMALLY

2.1. Algebraic specifications

The algebraic specifications in this paper are presented in the specification formalism *ASF* (for Algebraic Specification Formalism), of which a complete treatment of syntax and semantics is given by Bergstra, Heering and Klint in [BHK87]. The choice of ASF is not essential to the paper, so there is no need to explain this formalism in great detail. Various features (renaming, infix operators) are not used in the paper and will not be discussed here at all.

A specification in ASF consists of a list of modules. Every module has a unique name and contains the following sections, all of them optional:

- A *parameters* section, which contains a list of sorts and functions to be bound to fully define the current module. For instance, a stack may be parameterized by the sort of the items to be put on it. Binding the items to some particular sort produces a full definition.
- An *exports* section, containing the sorts and (typed) functions *visible* to the outside world (which is an importing module).
- An *imports* section, containing a list of modules. The sorts and functions exported by the modules in the list are imported in the current list and exported again. Parameters can be bound in this section. The sorts and functions of this and the preceding section provide the visible signature in the corresponding formal treatment of the algebraic specification.
- A *sorts* section, and a *functions* section, providing the hidden signature of the algebraic specification. These two sections together are often informally dubbed the *hidden* section. Definitions here are local to the module.
- A *variables* section, which quantifies the list of typed variables universally over the equations presented in the
- *equations* section, containing positive conditional equations (i.e., equations of the form $A=B$ or $A=B$ **when** $P_1=Q_1 \wedge \cdots \wedge P_n=Q_n$). Equations consist of open terms generated by all exported functions (these include imported functions), and all hidden functions. Every equation is labeled for easy reference.

The equations which hold in a module are all equations of the module itself and the equations of the (directly and indirectly) imported modules with proper renaming of hidden functions, sorts and variables from the imported modules to ensure independence.

A special place in ASF has the function `if`, which is predefined. `if` has three arguments of which the first must be of type `BOOL` and the second and third of the same, but arbitrary, type. It is approximately defined by the following equation scheme:

For every sort (with variables `x`, `y` of this sort) one adds:

```
[if.1] if(true, x, y)  = x
[if.2] if(false, x, y) = y
```

The semantics of a module is defined by the initial algebra over the export (visible) signature and the function `if` as defined by the equations above. This will be encountered in more detail in the remainder of the paper.

Various ASF-specifications will appear in the sequel, to which one is referred for examples.

2.2. Hoare logic and abstract data types

Hoare logic is a well-known technique to describe the behaviour of programs in both imperative and functional languages. It has found its way into various text books, e.g. [LS84] and [Bac86], which provide more rigour. Briefly, Hoare logic allows one to write

$$\{P\}\ S\ \{Q\},$$

meaning that evaluation of program S in a state in which *precondition* P holds results in a state in which *postcondition* Q holds. These conditions describe the state vector, i.e., the variables and their contents, of the program. Various proof rules and proof techniques are available to verify such a program.

In the paper some functions specified in an algebraic way will be specified in an equivalent Hoare logic way by giving conditions on its input terms and its output. Such a specification is independent of the actual implementation program, which may be changed (and preferably optimized). Since Hoare logic techniques can be formulated for many languages the ultimate program could be written in any appropriate language, at a cost in interfacing. Hence a large degree of language independence for the implementation is achieved, allowing various kinds of optimization strategies.

One way to interpret an algebraic specification is as a high level specification of an *abstract data type*. Hence the implementation strategy for algebraic specifications presented here bears a more than casual resemblance to the theory of implementation of abstract data types. An abstract data type is some type together with a set of functions on the type. An implementation is a more concrete (i.e., closer to machine

level) type with a corresponding set of functions which model the abstract type and functions. This is done by providing a translation back and forth between the abstract and concrete types, such that the abstract functions are simulated correctly by the combination of the translation to the concrete type, application of the concrete function and the translation back to the abstract type (cf. Jones [Jon80]).

The scheme in the paper basically relaxes the translation conditions for all terms in the initial model of the algebraic specification by demanding translations for specific terms only. This stems from the functional orientation: only the input terms need to be translated and only the output terms need to be translated back. Section 5 provides more detail.

3. ALGEBRAIC IMPLEMENTATION

3.1. Initial algebra semantics and reusability

The question we want to consider is the following. Suppose we have an algebraic specification and we want to make in some way an efficient implementation for further use by someone else, as in a library. What is the interaction between efficiency and semantics?

Initial algebra semantics have much in favour. They are characterized by 'no junk', i.e., it is clear which objects exist, and 'no confusion', i.e., closed terms (terms without variables) are only equal if they can be proved equal using equational logic. While these two characterizations are clearly desirable in many circumstances this is not always the case.

The 'no confusion' condition generates overspecification in the sense that terms might be distinguished from each other without necessity. If the writer of a specification does not care about whether two terms are equal or not (in the common case that their usage is identical), and hence does not specify their equality, they are unequal. This puts a burden on the implementor of the specification to provide this inequality, not allowing a possibly more efficient identification. Since it is not possible to specify which terms must be unequal the only tool available is the precise definition of the opposite property – equality – by extending the number of equations. This puts a burden on the shoulders of the specifier, who has to provide these additional equations. While the extra amount of work is undesirable it is also not clear in general what additional equations are necessary and whether a sufficient set can be found at all. For discussions see [Bak84], [Kam83] and [Wan79].

An example will serve to illustrate this. A very common datatype is the bounded array. Suppose a specifier wants to define an array of natural numbers of length ten, indexed from 0 to 9. It should be possible to put natural numbers into the array at certain indices and to retrieve them again, getting the last one. Initially, all entries are set to 0, and of course they can be reset to this value, simply by entering a 0 in every slot. Out of bound indices are simply ignored. In practice one would probably want to have a more robust version, but this will be at the cost of a longer specification. The following specification is a natural way to describe such a bounded array.

```
module BoundedArray
begin

exports
        begin
        sorts ARR
        functions
                newarr  :                    -> ARR
                put     : NAT # NAT # ARR -> ARR
                maxindex:                    -> NAT
                get     : NAT # ARR          -> NAT
        end

imports Booleans, Natnumbers
```

```
variables
        i,j,v : -> NAT
        arr   : -> ARR

equations
[1] maxindex = 9
[2] get(i,newarr) = 0
[3] get(i,put(j,v,arr)) = if(greater(i,maxindex),
                             0,
                             if(equal(i,j),
                                v,
                                get(i,arr)))
[4] put(i,v,arr) = arr
                   when greater(i,maxindex) = true
end BoundedArray
```

This specification contains just about what one wants to specify if the output behaviour of function `get` is the only thing of concern. Equation 1 fixes `maxindex` and equation 4 says that additions above this key have no effect. Equations 2 and 3 describe the behaviour of function `get`. If a user imports this module and restricts the use of the result of functions in the module to `get` only, it will behave as a bounded array, so no need is felt to extend the specification.

The problem is, that function `put` is hardly specified. This is natural, since the writer of module `BoundedArray` concentrated `get`, the function he wanted to specify. Indeed, in terms of functionality of `get` there is nothing wrong with the specification of `put` as it is. However, since we are using initial semantics, it is possible for someone importing module `BoundedArray` to extend the number of functions on sort ARR with

```
sum      : NAT # ARR        -> NAT
```

and to add the following equations:

```
[5] sum(i,newarr) = 0
[6] sum(i,put(j,v,arr)) = if(greater(i,maxindex),
                             0,
                             if(equal(i,j),
                                add(v,sum(i,arr)),
                                sum(i,arr)))
```

This new function makes a summation over *all* entries ever put into a certain index value of the array. It is a well-defined new function in the sense that no unexpected identifications of terms in other sorts than ARR (which we are redefining) occur. Of course, the writer of module `BoundedArray` intended to have only the last entry in hand.

How can the specification be remedied? The straightforward way to get rid of function `sum` is to put restrictions on the terms of sort ARR allowed. Old entries should be forgotten. The following equation specifies that:

```
[7] put(i,v1,put(j,v2,arr)) = if(equal(i,j),
                                 put(i,v1,arr),
                                 put(j,v2,put(i,v1,arr)))
```

Now the definition of `sum` would result in undesired identifications in sort `NAT`. Hence the function is ruled

out (it is still allowed to define it like this, but the resulting module will have unintended properties).

Addition of this one equation seems to be fine, but how can we be sure that the story ends here? This requires a non-trivial proof, for instance giving an isomorphism between the initial algebraic model generated by the specification, and the data type to be modeled.

In the example, the combination of equations 1 through 4 and 7 does not allow that. The terms `newarr` and `put(7,0,newarr)` cannot be proved equal by these equations, though they both describe the array containing exclusively zeros. Of course, the equation

```
[8] newarr = put(i,0,newarr)
```

can be added , but then the question whether the set of equations (now 1 through 4 and 7 and 8) fixes what was intended returns again.

Actually, these six equations are sufficient. A proof could proceed as follows. First a set of canonical forms is defined, e.g. the set of terms with at most one `put` for every key in order of the keys and without entries of value `0`. Obviously a bijection between the set of canonical forms and arrays of length ten can be found. It also can be proved that every well-formed term of sort `ARR` is equationally equal to exactly one of these canonical terms. So the term model has exactly the same structure.

The surest way to supply a structural answer in initial algebra semantics is to add a constructor function actually making an isomorphic image of the object wanted. In the example this is a bounded array of length `maxindex+1`, so for instance a function with 10 holes in it. Then the suitable additional specifications must be provided in terms of this constructor function. In the example this would become a function:

```
arr: NAT # NAT # NAT # NAT # NAT # NAT # NAT # NAT # NAT # NAT -> ARR
```

and in addition to equations 1 through 4 the equations (`v0...v9` are variables of sort `NAT`):

```
[n0] newarr = arr(0,0,0,0,0,0,0,0,0,0)
[p0] put(0,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v,v1,v2,v3,v4,v5,v6,v7,v8,v9)
[p1] put(1,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v0,v,v2,v3,v4,v5,v6,v7,v8,v9)
             ...
[p9] put(9,v,arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v9))
            = arr(v0,v1,v2,v3,v4,v5,v6,v7,v8,v)
```

Apart from the question whether such an *ad hoc* solution can be found in general this is contrary to the amount of detail one wants to specify algebraically. For this two important considerations can be given, one philosophical and one practical:

- Algebraic specification is a higher level programming formalism. While the formalism is powerful enough to express a computer up to bit level if necessary, this is a waste of effort. There are more than enough lower level programming languages already.
- An algebraic specification (and indeed any specification) is made with a certain use in mind. This use is what has to be specified in detail, since that is what has to be implemented. Other details specified are peripheral in the sense that one might have chosen another description. The less detail is fixed in these peripheral specifications the more freedom an implementor has for optimizing it. The choice of models for implementation should not be restricted to one model (up to isomorphism), but rather be as broad as the

we only wanted a convenient function to enter natural numbers in a "behind-the-screen" data representation.

The exact form of the data representation is of little interest to the user of function `get`, as long as this use is not affected. In this example probably a simple array of length 10 is what you want. But changing the value of `maxindex` to some large number might make a sparse array approach or a hash-table the better implementation.

For the remainder of the paper we distinguish three important subsets in the signature of an algebraic specification:

- The **visible** signature which generates the terms existing in the specification for the outside world.
- The **hidden** signature, which is necessary to obtain finite initial algebra specifications on the one hand and handy as a shorthand mechanism and alternative data description on the other hand. The complete signature is the union of the visible and the hidden signature.
- The **observing** signature, which restricts the terms generated by the visible signature. It contains the functions through which visible terms may be used and the sorts with terms which may be used as observing terms. A term is an observing term if both the head function and its sort are in the observing signature. This signature is the subset of the visible signature one wants to be implemented as specified. In the example this is the signature with sort `NAT` and function `get`.

The choice of the functions and sorts in the observing signature depends on the goal one has in mind for the specification. Making this signature bigger enhances the possibilities for use but restricts the freedom of the implementor. So one can opt for a fast, but narrowly applicable implementation, or for a more generally usable, but slower implementation.

Of course, the speed of a certain function in an observable signature is not only dependent on the signature but also on the implementations of other (not necessarily observing) functions. One can for instance trade the speed of an insertion function for the speed of a retrieval function by gearing the underlying data structure (at this level of abstraction represented by the visible functions that are not observing and the hidden functions) to the other task.

The consequences of this tripartition for the theory are investigated in the next section.

3.2. A theory of algebraic implementation

This section is devoted to the development of a theory for the subsequently introduced notion of *algebraic implementation* with respect to an *observing signature*. Roughly speaking two algebraic specifications are algebraic implementations of each other when the behaviour of the observing functions is the same in both specifications. An annotated example is provided in section 4. The reader may wish to read the example first, referring back to notations and details in this section when necessary.

3.2.1. Notations (algebraic specifications)

In the rest of the paper the following conventions are used:

a. A **signature** Σ is a tuple (S,F) in which S is a set of sorts and F a set of typed functions. (Note that there is no intrinsic relation between the sorts in S and F.) Often an element of F is denoted by its name only, providing typing when necessary. Two functions with the same name, but different typing are different functions.

b. A **complete signature** $\Sigma = (S,F)$ is a signature in which for all $f: s_1 \times \cdots \times s_k \rightarrow s \in F$ holds that all sorts in the typing of f are available in S, so $s_1, \cdots, s_k, s \in S$.

c. For a signature Σ is $T(\Sigma)$ the **set of closed terms**; $T_s(\Sigma)$ is the subset of terms of sort s from $T(\Sigma)$.

d. Union, intersection and inclusion are defined for signatures Σ_1, Σ_2 ($\Sigma_i = (S_i,F_i)$), as:

$$\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2,\ F_1 \cup F_2)$$

$$\Sigma_1 \cap \Sigma_2 = (S_1 \cap S_2,\ F_1 \cap F_2)$$

$$\Sigma_1 \subseteq \Sigma_2 = S_1 \subseteq S_2 \wedge F_1 \subseteq F_2$$

e. An **algebraic specification** is a tuple (Σ_V, Σ_H, E) with
   - $\Sigma_V = (S_V, F_V)$ a complete signature (the **visible** signature),
   - $\Sigma_H = (S_H, F_H)$ a signature (the **hidden** signature) such that $\Sigma_V \cup \Sigma_H$ (the **internal** signature) is a complete signature, and
   - E a set of equations over $T(\Sigma_V \cup \Sigma_H)$.

f. Let (Σ_V, Σ_H, E) be an algebraic specification and let $t, t' \in T(\Sigma_V \cup \Sigma_H)$. For an equation $e \in E$, t **and** t' **are equal through direct substitution in equation** e (i.e., in one step) is written as:

$$t =_e t'.$$

t **and** t' **are equationally equal**, i.e., equal through zero or more direct substitutions in one or more equations from E, is written as:

$$t =_E t'.$$

3.2.2. Definitions (Σ_O-observability and -equality)
Let (Σ_V, Σ_H, E) be an algebraic specification and $\Sigma_O = (S_O, F_O)$ (the **observing** signature) a signature such that $\Sigma_O \subseteq \Sigma_V$.

a. The **set of closed Σ_O-terms over Σ_V**, also called the **set of observing terms** is the set of terms in $T(\Sigma_V)$ of sort in S_O and head function symbol in F_O. It is defined as:

$$T(\Sigma_O, \Sigma_V) = \{t \in T(\Sigma_V) \mid \exists f{:}s_1 \times \cdots \times s_k \rightarrow s \in F_O,\ s \in S_O\ \exists u_1 \in T(\Sigma_V) \cdots \exists u_k \in T(\Sigma_V)$$
$$[t \in T_s(\Sigma_V) \wedge t = f(u_1, \cdots, u_k)]\ \}.$$

The **set of closed Σ_O-terms over Σ_V of sort** s is written as $T_s(\Sigma_O, \Sigma_V)$.
Note that it is possible to have functions in F_O whose output sort is not in in S_O, or sorts in S_O which cannot be reached from F_O. This choice of notation is motivated by the function-oriented approach of this paper. By choosing a set of observing functions F_O and all visible sorts S_V for S_O all sorts in S_V which cannot be reached do not influence $T(\Sigma_O, \Sigma_V)$. For symmetry reasons the definition is formulated in such a way that one can also restrict the sorts and not the functions, as will be done in point e below. Alternatively, it is possible to define S_O as the set of sorts in the range of F_O. This does not affect the theory.

b. Where no confusion can arise the following abbreviations are used:

$$T_O = T(\Sigma_O, \Sigma_V)$$
$$T_V = T(\Sigma_V)$$
$$T_{s,O} = T_s(\Sigma_O, \Sigma_V)$$
$$T_{s,V} = T_s(\Sigma_V)$$

c. A **context** (for sort s) $T(\circledast_s)$ is a term with a missing subterm of sort s. The **empty context** (i.e., a context in which the top term is missing) is written as $\circledast_s$.
A term $t \in T_{s,V}$ is **Σ_O-observable** if and only if there exists a context $T(\circledast_s)$ such that $T(t) \in T_O$;
a Σ_O-observable term $t \in T_{s,V}$ is **directly Σ_O-observable** if and only if $t \in T_O$ (hence the empty context $T(\circledast_s) = \circledast_s$ satisfies $T(t) \in T_O$);
a Σ_O-observable term $t \in T_{s,V}$ is **indirectly Σ_O-observable** if and only if $t \notin T_O$ (hence the empty context T does not satisfy $T(t) \in T_O$).

d. **Σ_O-equality** (i.e., equality with respect to observations through the observing signature Σ_O) is defined as follows for two terms $t, t' \in T_{s,V}$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow \forall T(\circledast_s)\,[T(t), T(t') \in T_O \rightarrow T(t) =_E T(t')\,].$$

Where no confusion can arise $\sim_{E,\Sigma_O}$ is abbreviated to $\sim_O$.

e. Let $f \in F_V$. A term $t \in T(\Sigma_V)$ is **f-observable** if and only if t is $(S_V, \{f\})$-observable;
two terms in $T(\Sigma_V)$ are **f-equal** if and only if they are $(S_V, \{f\})$-equal.
Let $s \in S_V$. A term $t \in T(\Sigma_V)$ is **s-observable** if and only if t is $(\{s\}, F_V)$-observable;
two terms in $T(\Sigma_V)$ are **s-equal** if and only if they are $(\{s\}, F_V)$-equal.
Since the definition in case c ignores unreachable sorts and functions to unavailable sorts, only terms with head symbol f in the first two cases, and with range s in the last two, are relevant.
The notion of observability via a sort corresponds to the notion in [MG85], and is underlying the behavioural equivalence notion in [ST85]. In the latter paper, the observational equivalence notion is very general, since it is parameterized with the logic used to reason about observations. Thus Σ_O-equality corresponds to observational equivalence under conditional equational logic in the terms of [ST85]. By concentrating on one logic more can be said about the implementation in the present paper.

3.2.3. Some facts about Σ_O-observability and -equality

In final algebra semantics terms are equal unless they can be proved different, so in models with final semantics there is the maximum amount of 'confusion' consistent with the inequalities which must exist in the model. As such, Σ_O-equality is a notion from final algebra semantics. If you want to show that two closed terms are different you have to find a context for which these terms behave differently, thus proving their inequality. If no such context can be found the terms cannot be distinguished from each other. In initial semantics, on the contrary, they are distinguished unless they are equationally equal, in other words, unless they can be transformed into each other via equations from E, thus proving their equality.

Let (Σ_V, Σ_H, E) be an algebraic specification, $\Sigma_O \subseteq \Sigma_V$, and $t, t' \in T_{s,V}$, then the following facts hold:

a. $T(\Sigma_V, \Sigma_V) = T(\Sigma_V)$
Observing through the visible signature gives all visible terms. This follows immediately from definition 3.2.2.a.

b. If $\Sigma'_O \subseteq \Sigma_O$ then $T(\Sigma'_O, \Sigma_V) \subseteq T(\Sigma_O, \Sigma_V)$.
A smaller observing signature results in less observing terms. Again this follows from definition 3.2.2.a.

c. If $\Sigma'_O \subseteq \Sigma_O$ then $t \sim_{E,\Sigma_o} t' \rightarrow t \sim_{E,\Sigma'_o} t'$.
This follows from the definition of Σ_O-equality and fact b, since there are less contexts in $T(\Sigma'_O, \Sigma_V)$ than in $T(\Sigma_O, \Sigma_V)$ to show the difference between t an t'.

d. If t and t' are not Σ_O-observable they are Σ_O-equal.
Since there is no context to show the difference between t and t' this follows from the definition.

e. If t is Σ_O-observable and t' is not then they are Σ_O-equal.
The argument for fact d holds here also.

f. It should be noted that in cases d and e both $t \sim_{E,\Sigma_v} t'$ and $\neg\, t \sim_{E,\Sigma_v} t'$ can be true. Take for example:

$$\Sigma_V = (\{s\}, \{a, b\}) \text{ with } a, b \in s, \text{ and}$$

$$\Sigma_O = (\{s\}, \{a\}) \text{ for case e, or}$$

$$\Sigma_O = (\{s\}, \varnothing) \text{ for case d.}$$

When the set of equations is empty the following holds:

$$a \sim_{\varnothing,\Sigma_o} b \text{ and } \neg\, a \sim_{\varnothing,\Sigma_v} b,$$

while $E = \{a = b\}$ has as result:

$$a \sim_{\{a=b\},\Sigma_o} b \text{ and } a \sim_{\{a=b\},\Sigma_v} b.$$

The following lemma states that the initial algebraic structure is retained on directly observable terms. So only indirectly observable and unobservable terms can lose their initial behaviour. It follows immediately (corollary 3.2.5) that no restriction on the observability (i.e., $\Sigma_O = \Sigma_V$) retains the initial algebraic structure.

3.2.4. Initial Algebra Lemma
For $t, t' \in T_s(\Sigma_O, \Sigma_V)$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow t =_E t'.$$

Proof:
$\Rightarrow$ Immediately from definition 3.2.2.d.
$\Leftarrow$ For all contexts $T(\bullet_s)$ such that $T(t)$, $T(t') \in T(\Sigma_O, \Sigma_V)$, $t =_E t'$, and hence $T(t) =_E T(t')$, holds.□

3.2.5. Corollary ($\Sigma_O = \Sigma_V$ preserves initial algebra semantics)
For $t, t' \in T_s(\Sigma_V)$:

$$t \sim_{E,\Sigma_V} t' \Leftrightarrow t =_E t'.$$

3.2.6. Witness Existence Lemma
The following lemma formulates a nice fact for proofs with observable terms. Two terms are Σ_O-equal unless there is a context proving the opposite. Hence two terms are Σ_O-equal when there is no common context. So it is important to have at least one common context. In this lemma existence of a witness context is proven for Σ_O-observable terms of the same sort.
Lemma: For two Σ_O-observable terms $t, t' \in T_{s,V}$ there exists a context $T(\bullet_s)$ such that $T(t)$, $T(t') \in T_O$.
Proof:
a. If $t, t' \in T_{s,O}$ the empty context $T(\bullet_s) = \bullet_s$ is fulfilling the condition.
b. If t is indirectly Σ_O-observable there exists a non-empty context $T(\bullet_s)$ such that $T(t) \in T_O$. Since T is non-empty the head function f is in F_O with range in S_O. Hence $T(t') \in T_O$.□
In the proof, case a corresponds to the initial algebra equality, and case b to the final algebra (observable only) equality.

3.2.7. Σ_O-equality as congruence: a problem with transitivity
We would have liked to use the notation of $=_O$ instead of $\sim_O$ since it should define a congruence similar to $=_E$. However, there are some problems connected with the final nature of $\sim_O$ and the initial nature of $=_E$. A congruence $\sim$ satisfies the following laws:
- *symmetry*, i.e., $t \sim t$;
- *reflexivity*, i.e., if $t \sim t'$ then also $t' \sim t$;
- *transitivity*, i.e., if $t \sim t'$ and $t' \sim t''$; then also $t \sim t''$;
- *the substitution property*, i.e., if $t_1 \sim t_1' \wedge \cdots \wedge t_n \sim t_n'$ holds then also $f(t_1, \cdots, t_n) \sim f(t_1', \cdots, t_n')$ holds.

For terms in $T(\Sigma_V)$ reflexivity, symmetry and the substitution property of $\sim_O$ follow immediately from the corresponding properties of $=_E$ and definition 3.2.2.d. However, in section 3.2.3 facts d and e show that transitivity is not guaranteed on $T(\Sigma_V)$. Since these facts deal with terms that are not observable, this is no real problem. However, $\sim_O$ is also not transitive on the subset of Σ_O-observable terms in $T(\Sigma_V)$. This is illustrated in the following example.

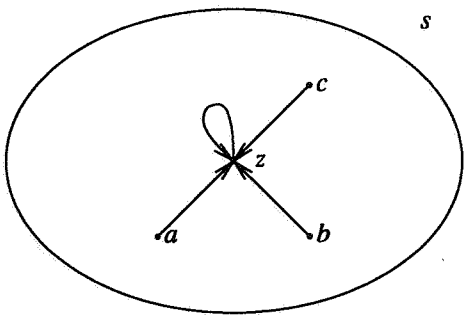


FIGURE 3.1

Let $\Sigma_V = (\{s\}, \{a, b, c, z, f\})$ with $a, b, c, z \in s$ and $f : s \to s$, E consists of the equation $f(x) = z$ with x a variable of sort s, and $\Sigma_O = (\{s\}, \{a, b, z, f\})$. This structure is shown in FIGURE 3.1. Then $a \sim_O c$, since $f(a) =_E f(c), f(f(a)) =_E f(f(c))$, etc., and similarly $c \sim_O b$, though $\neg a \sim_O b$.

Still, this is not unreasonable. If one only looks at Σ_O any relation involving c is irrelevant. The new structure is given in FIGURE 3.2, forgetting the dashed arrow.

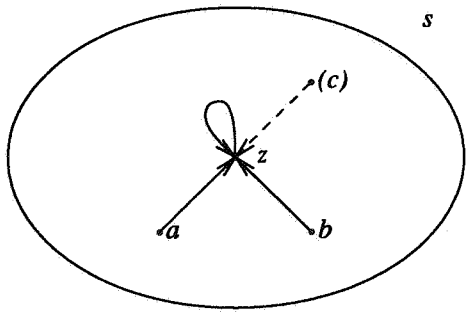


FIGURE 3.2

If later on one would want to add a new constant named c in FIGURE 3.2 then c could be a new name for an old constant like a, b, or z, or even a completely new constant. So if c is observably not equal to one or more of constants, this would rule out some of the possibilities. Hence, the freedom allowed when introducing c would be limited in an undesirable way.

The precise criteria conserving transitivity, and hence making $\sim_O$ a congruence are given in Theorem 3.2.8. Some important classes of observable signatures that are transitivity conserving are given in a corollary (3.2.9).

3.2.8. Transitivity Theorem

Let t, t', t'' be Σ_O-observable terms of sort s such that $t \sim_O t'$ and $t' \sim_O t''$, then

$$\neg t \sim_O t'' \Leftrightarrow t, t'' \in T_O \wedge t' \in T_V - T_O \wedge \forall T(\circledast_s)\, T(t), T(t'') \in T_O \to [T(t) \neq_E T(t'') \leftrightarrow T(\circledast_s) = \circledast_s]$$

Proof

$\Leftarrow$ The empty context $T(\circledast_s) = \circledast_s$ is a context for t and t'', since they are directly Σ_O-observable. Hence $t \neq_E t''$ and thus $\neg t \sim_O t''$.

$\Rightarrow$ The three parts of the conjunction are treated in sequence.
If t is indirectly Σ_O-observable all contexts $T(\circledast_s)$ such that $T(t), T(t'') \in T_O$ are non-empty and hence satisfy $T(t') \in T_O$. Hence for any such context $T(t) =_E T(t') =_E T(t'')$, and thus $t \sim_O t''$. From this

contradiction it follows that t (and by symmetry t'') is directly Σ_O-observable.
If t' is directly Σ_O-observable, $t =_E t'$ and $t' =_E t''$ hold, since t and t'' are directly observable, and hence $t \sim_O t''$. Hence t' must be indirectly Σ_O-observable.
Since $\neg t \sim_O t''$ there exists a context $T(\bullet_s)$ such that $T(t) \neq_E T(t'')$ and $T(t), T(t'') \in T_O$. If $T(\bullet_s)$ is non-empty then also $T(t') \in T_O$, and hence $T(t) =_E T(t') =_E T(t'')$. Thus $T(\bullet_s)$ must be the empty context $\bullet_s$.□

3.2.9. Transitivity conserving constraints
The transitivity theorem states that two *directly* Σ_O-observable terms can be Σ_O-unequal, even though there is an *indirectly* Σ_O-observable term which is Σ_O-equal to both. This is the case in the example in 3.2.7. Hence Σ_O-inequality is stronger for directly observable terms than for indirectly observable terms.

The theorem above provides necessary and sufficient conditions for transitivity. This may be unwieldy to use in practice. However, it is conveniently possible to give criteria, that are important from the point of view of implementation, to check whether $\sim_O$ is an equivalence relation. Intuitively, the implementation of directly observable terms only has to follow the initial algebra semantics (Lemma 3.2.4), while indirectly observable and unobservable terms are less demanding for the implementation. The criteria are formulated below:
Corollary:
Let T_s be the subset of Σ_O-observable terms from $T_{s,V}$. The relation $\sim_{E,\Sigma_O}$ is an equivalence relation on T_s if either of the following holds:
a. $T_{s,O} = \emptyset$;
b. $T_{s,O} = T_s$;
c. for all $t \in T_s$ there is precisely one $t' \in T_{s,O}$ such that $t' \sim_O t$;
d. $\Sigma_O = (S_O, F_V)$ for some $S_O \subseteq S_V$.
Notes:
ad a. Sort s is not directly observable. Consequently its internal representation may be changed without altering the directly observable sorts.
ad b. T_s has to be implemented with initial algebra semantics.
ad c. There is exactly one directly observable term Σ_O-equal to any term of T_s. This term plays the role of a canonical form and has to be implemented faithfully. All other terms may be implemented by their canonical equivalent.
ad d. If $s \in S_O$ then all constructor functions for terms of sort s are available, hence $T_s = T_{s,O}$ (case b).
If $s \notin S_O$ then no constructor function for terms of sort s qualifies as outermost function in T_O and hence $T_{s,O} = \emptyset$ (case a).
This case states that for s-observability $\sim_O$ is a congruence for terms of any sort $s' \in S_V$, including s itself. Hence it is a rephrasing of the well-known fact that observability through a sort conserves the congruence (see [MG85]).
Generally $\sim_O$ will be a congruence. If that is the case it is usually written as $=_O$ in the sequel. Similarly $\sim_{E,\Sigma_O}$ becomes $=_{E,\Sigma_O}$ and $\sim_{E,\Sigma_V}$ becomes $=_{E,\Sigma_V}$.

3.2.10. Definition (Algebraic Implementation)
The following definition represents the central notion in this section, namely the notion of implementation for an algebraic specification relative to an observing signature. Intuitively, two specifications are algebraic implementations of each other when they have the same congruence on the observable terms. This is inherently an almost symmetric notion: if a small specification implements part of a large one then the large specification implements the same part of the small one (and more, but that is redundant) if the set of observable terms is the same. We provide the following *definition*:
- Let (Σ_V, Σ_H, E) and $(\Sigma'_V, \Sigma'_H, E')$ be algebraic specifications and Σ_O be a signature such that $\Sigma_O \subseteq \Sigma_V \cap \Sigma'_V$.
$(\Sigma'_V, \Sigma'_H, E')$ is a **Σ_O-implementation of** (Σ_V, Σ_H, E) if and only if for all $s \in S_V$ and for all Σ_O-observable terms $t, t' \in T_s(\Sigma_V)$:

$$t \sim_{E,\Sigma_O} t' \Leftrightarrow t \sim_{E',\Sigma_O} t'.$$

3.2.11. Some facts about algebraic implementations

a. If $\Sigma_O \subseteq \Sigma_V$ then an algebraic specification (Σ_V, Σ_H, E) is a Σ_O-implementation of itself. As an even more trivial special case (Σ_V, Σ_H, E) is a Σ_V-implementation of itself.

b. If $(\Sigma'_V, \Sigma'_H, E')$ is a Σ_O-implementation of (Σ_V, Σ_H, E) and $\Sigma'_O \subseteq \Sigma_O$ then $(\Sigma'_V, \Sigma'_H, E')$ is also a Σ'_O-implementation of (Σ_V, Σ_H, E).

c. Σ_O-implementation is a symmetric relation on the class of algebraic specifications with the same set of Σ_O-observable terms.

d. Σ_O-implementation is also a transitive relation under the conditions of case c.

While the facts above provide some idea about the usefulness of the definition two important properties have to be proved. Of course we want to conserve the property in initial algebra semantics that the hidden signature and the set of equations may be changed as long as this does not affect the congruence on the visible signature. This is proved in lemma 3.2.12.

Next, in the central theorem a functionally oriented criterion is given for an algebraic implementation. This serves as a starting point for section 5, in which a notion of functional implementation will be given.

3.2.12. Initial Algebra Implementation Lemma

Let $(\Sigma_V, \Sigma'_H, E')$ be a Σ_V-implementation of (Σ_V, Σ_H, E), then for all $s \in S_V$ and for all $t, t' \in T_{s,V}$:

$$t =_E t' \Leftrightarrow t =_{E'} t'.$$

Proof:

All terms in T_V are Σ_V-observable. Hence for all $s \in S_V$ and for all $t, t' \in T_{s,V}$:

$$t =_{E,\Sigma_V} t' \Leftrightarrow t =_{E',\Sigma_V} t'.$$

According to corollary 3.2.5 $t =_{E,\Sigma_V} t' \Leftrightarrow t =_E t'$ and $t =_{E',\Sigma_V} t' \Leftrightarrow t =_{E'} t'$, hence $t =_E t' \Leftrightarrow t =_{E'} t'$.□

3.2.13. Algebraic Implementation Theorem

Let (Σ_V, Σ_H, E) and $(\Sigma'_V, \Sigma'_H, E')$ be algebraic specifications and $\Sigma_O \subseteq \Sigma_V \cap \Sigma'_V$.
If for all $f \in F_O$, $f : s_1 \times \cdots \times s_k \rightarrow s_0$, with $s_0 \in S_O$, for all $t \in T_{s_0}(\Sigma_V)$ and for all $(u_1, \cdots, u_k) \in (T(\Sigma_V))^k$ $f(u_1, \cdots, u_k) =_E t \Leftrightarrow f(u_1, \cdots, u_k) =_{E'} t$ holds, then $(\Sigma'_V, \Sigma'_H, E')$ is a Σ_O-implementation of (Σ_V, Σ_H, E).

Proof:

Let $s \in S_V$ and $t, t' \in T_s(\Sigma_V)$ be Σ_O-observable, then

$$\begin{aligned}
t \sim_{E,\Sigma_O} t' &\Leftrightarrow \forall T(\bullet_s)\, [T(t), T(t') \in T(\Sigma_O, \Sigma_V) \rightarrow T(t) =_E T(t')] && \text{(by 3.2.2.d)}\\
&\Leftrightarrow \forall T(\bullet_s)\, T(t), T(t') \in T(\Sigma_O, \Sigma_V)\, [\exists g \in F_O, u_1, \cdots, u_k \in T(\Sigma_V) && \text{(by 3.2.2.a)}\\
&\qquad (g(u_1, \cdots, u_k) = T(t) \wedge g(u_1, \cdots, u_k) =_E T(t'))]\\
&\Leftrightarrow \forall T(\bullet_s)\, T(t), T(t') \in T(\Sigma_O, \Sigma_V)\, [\exists g \in F_O, u_1, \cdots, u_k \in T(\Sigma_V)\\
&\qquad (g(u_1, \cdots, u_k) = T(t) \wedge g(u_1, \cdots, u_k) =_{E'} T(t'))]\\
&\Leftrightarrow \forall T(\bullet_s)\, T(t), [T(t') \in T(\Sigma_O, \Sigma_V) \rightarrow T(t) =_{E'} T(t')] && \text{(by 3.2.2.a)}\\
&\Leftrightarrow t \sim_{E',\Sigma_O} t'. && \text{(by 3.2.2.d)}
\end{aligned}$$

□

Note: this theorem is sufficiently strong to describe the behaviour of a function up to the congruence defined by $\sim_{E,\Sigma_O}$, if such a congruence exists. An example of the use of the theorem is given in the next section. A more restrictive definition of implementation, strong enough to describe functional implementation, is given in section 5.

4. AN EXAMPLE: TABLES

In this section, two definitions of elementary data structures for objects of arbitrary sort ELEM are given. Both data structures support storage and retrieval with elements of an arbitrary sort KEY as selection criterion. Equality is defined on this sort through function eq.

The first specification describes a sort TABLE. An element of this sort is a list of all entries with corresponding keys in the data structure. This data structure can be searched from the last entry to the first, but anything faster than linear search is not accommodated for.

The second specification uses a hidden sort TREE to implement the same data in a search-tree. This is possible when we have a total ordering on sort KEY. In the example function lt, combined with eq, provides such an ordering.

The specifications below are parameterized with sorts KEY and ELEM at the specification level. To guide the intuition, one should think of two (perhaps equal) sorts which are already well-known, e.g. CHAR for ELEM and INT for KEY. ASF does not provide a semantics for unbound parameter sorts, since there is no mechanism to force restrictions on the actual parameter (like the total ordering in the example). In the paper restrictions are given in the commentary, so it will be clear what the semantics should be.

The remainder of this section is devoted to a fairly detailed proof sketch of the fact that the modules Tables and Tables-as-trees are algebraic implementations of each other when one observes through the retrieve function lookup only, in other words, with respect to lookup-equality. So we take for the observing signature ({BOOL, ELEM, KEY, TABLE}, {lookup}), or ({ELEM}, {lookup}), since ELEM is the range of lookup.

The simple specification is given below:

```
module Tables -- original specification
begin
  parameters Keys-and-Elements
    begin
      sorts KEY, ELEM
      functions eq: KEY # KEY -> BOOL -- equality
    end Keys-and-Elements
  exports
    begin
      sorts TABLE
      functions
        nulltable:                      -> TABLE
        tableadd : KEY # ELEM # TABLE -> TABLE
        lookup   : KEY # TABLE        -> ELEM
        errorelem:                      -> ELEM
    end
  imports Booleans

  variables key, key1, key2: -> KEY
            elem           : -> ELEM
            table          : -> TABLE
  equations
    [1] lookup(key, nulltable) = errorelem
    [2] lookup(key1, tableadd(key2,elem,table))
            = if(eq(key1,key2),
                 elem,
                 lookup(key1,table))
end Tables
```

This specification speaks for itself. It is similar to the first BoundedArray specification in section 3.1.

Function tableadd gives the same problems as function put in that module. To avoid them we restrict the set of observing terms to those with function lookup as outermost symbol. Hence an implementor of this module can concentrate on the correct implementation of lookup.

To get an efficient implementation of lookup more detailed information about sort KEY is needed. If KEY is a small set something similar to a bounded array is feasible. If a hash function could be defined, a hash table might be used as implementation. Each of these structures can be algebraically specified as hidden structure, thus providing an algebraic specification which upon implementation gives an equivalent, but more efficient, implementation of lookup.

For this example it is assumed that a total ordering can be defined on the set KEY with the functions eq and lt (lower-than). The total ordering allows the definition of a binary search-tree. This is done in module Tables-as-trees below:

```
module Tables-as-trees
begin
  parameters Keys-and-Elements
    begin
      sorts KEY, ELEM
      functions eq: KEY # KEY -> BOOL -- equality
                lt: KEY # KEY -> BOOL -- lower-than --new
                -- eq and lt must provide a total ordering on sort KEY
    end Keys-and-Elements

  exports
    begin
      sorts TABLE
      functions
        nulltable:                      -> TABLE
        tableadd : KEY # ELEM # TABLE -> TABLE
        lookup   : KEY # TABLE        -> ELEM
        errorelem:                      -> ELEM
    end
  imports Booleans

  -- hidden section
  sorts TREE                                          --new
  functions
    tree     : TREE # KEY # ELEM # TREE ->  TREE   --new
    niltree  :                          ->  TREE   --new
    treeadd  : KEY # ELEM # TREE        ->  TREE   --new
    lookuptr : KEY # TREE               ->  ELEM   --new
    tbltotree: TABLE                    ->  TREE   --new

  variables key, key2   : -> KEY
            elem, elem2 : -> ELEM
            table       : -> TABLE
            tree1, tree2: -> TREE
```

```
equations                                                    -- all new
  [h1] tbltotree(nulltable) = niltree
  [h2] tbltotree(tableadd(key,elem,table))
                          = treeadd(key,elem,tbltotree(table))

  [h3] treeadd(key,elem,niltree)
                          = tree(niltree,key,elem,niltree)
  [h4] treeadd(key,elem,tree(tree1,key2,elem2,tree2))
                          = if(eq(key,key2),
                               tree(tree1,key,elem,tree2),
                               if(lt(key,key2),
                                  tree(treeadd(key,elem,tree1),
                                       key2,elem2,tree2),
                                  tree(tree1,key2,elem2,
                                       treeadd(key,elem,tree2))))

  [h5] lookuptr(key,niltree) = errorelem
  [h6] lookuptr(key,tree(tree1,key2,elem,tree2))
                          = if(eq(key,key2),
                               elem,
                               if(lt(key,key2),
                                  lookuptr(key,tree1),
                                  lookuptr(key,tree2)))
  [h7] lookup(key,table) = lookuptr(key,tbltotree(table))
end Tables-as-trees
```

Note that all equations contain hidden sorts. Equation h7 defines `lookup` in terms of `lookuptr`, the retrieval function on trees, itself defined in h5 and h6. Equations h1 through h4 define the build-up of a tree from a table.

It is possible to declare all hidden sorts and functions visible rather than hidden. The effect would be that module `Tables-as-trees` would still be an implementation with respect to `lookup`-observability of module `Tables`, but not the other way around. The reason for the latter is the existence of observable terms containing constructor functions for `TREE` in module `Tables-as-trees`, terms which are not existent in module `Tables`.

The following proof sketch first defines a well-formedness predicate *searchtree* for terms of sort `TREE`, since not all constructible terms are search-trees. Then it is proved that the predicate *searchtree* is invariant over the insertion function `treeadd`, and that the retrieval function `lookuptr` is well-defined for single additions to a tree which satisfies this predicate. Finally the equivalence between the two specifications is proved with induction on the number of insertions.

4.1. Definition (well-formedness of searchtrees)
The predicate *searchtree*(`t`) for a term `t` of sort `TREE` describes the well-formedness of a tree as search-tree. It will be used in the proof to derive properties about the behaviour of the data structure generated by function `treeadd`. This holds in particular for the behaviour observed through function `lookuptr`, which is needed to derive the behaviour of function `lookup`. The predicate is defined as follows (with `t1`, `t2` of sort `TREE`, `j`, `k`, `l` of sort `KEY`, and `e` of sort `ENTRY`):

- *searchtree*(`niltree`) = true;
- *searchtree*(`tree(t1,k,e,t2)`) =
  searchtree(`t1`) ∧ *searchtree*(`t2`) ∧
  ∀ `j` ∈ *set-of-keys*(`t1`) [`lt(j,k)` = true] ∧
  ∀ `l` ∈ *set-of-keys*(`t2`) [`lt(k,l)` = true],

with *set-of-keys*(t) for terms t of sort TREE a set containing all keys in t. Formally:
- *set-of-keys*(niltree) = ∅;
- *set-of-keys*(tree(t1,k,e,t2)) = *set-of-keys*(t1) ∪ {k} ∪ *set-of-keys*(t2).

4.2. Well-formedness lemma for trees

Two important properties of the behaviour of well-formed trees (i.e., terms satisfying predicate *searchtree*) are formulated. Case *a* states that the predicate *searchtree* is an invariant under insertion in the tree. Case *b* states that a well-formed tree behaves properly with respect to function lookuptr after insertion. These facts provide technical steps for the proof of lookup-equivalence in section 4.3.

For the remainder of section 4, let k, k', ... be of sort KEY, e, e', ... be of sort ENTRY, and t, t1, ... be of sort TREE, then we can formulate the following

Lemma:

a. *searchtree*(t) → *searchtree*(treeadd(k',e',t)).

b. *searchtree*(t) →

[eq(k,k') = true → lookuptr(k,treeadd(k',e',t)) = e'] ∧

[eq(k,k') = false → lookuptr(k,treeadd(k',e',t)) = lookuptr(k,t)].

Proof by induction on the number of nodes in the tree (omitted).

4.3. Proof of lookup*-equality*

The following proof uses induction with respect to the number of insertions using function tableadd. The well-formedness predicate *searchtree* makes the proof straightforward.

In this proof the equivalence defined by the equations from module Tables is called $=_{Tb}$ and from module Tables-as-trees $=_{Tr}$. Equivalence according to an equation numbered *i* is written $=_i$.

According to Theorem 3.2.13 it is sufficient to prove for all pairs (k, t) ∈ $T_{KEY,V} \times T_{TABLE,V}$ and for all terms e ∈ $T_{ELEM,V}$

$$\text{lookup}(k,t) =_{Tb} e \Leftrightarrow \text{lookup}(k,t) =_{Tr} e.$$

First we assume that e does not contain the function lookup. The proof then proceeds with induction on the length of terms in $T_{TABLE,V}$, which is defined in the obvious way, with multiple occurrences of the same key counted for every occurrence separately.

For the table of length 0, nulltable, both lookup(k,nulltable) $=_{Tb}$ errorelem and lookup(k,nulltable) $=_{Tr}$ errorelem obviously hold. Now let

$$\text{lookup}(k,t) =_{Tb} e \Leftrightarrow \text{lookup}(k,t) =_{Tr} e$$

be proved for all tables of length $n \geq 0$ and e not containing lookup, and let t' = tableadd(k',e',t), with e' not containing function lookup, be a table of length $n+1$.

- If eq(k,k') $=_{Tb/Tr}$ true then
  lookup(k,t') $=_2$ e', and
  lookup(k,t') $=_{h7}$ lookuptr(k,tbltotree(tableadd(k',e',t)))
  $=_{h2}$ lookuptr(k,treeadd(k',e',tbltotree(t)))
  $=_{Tr}$ e',
  with the last equation following from lemma 4.2.
- If eq(k,k') $=_{Tb/Tr}$ false then
  lookup(k,t') $=_2$ lookup(k,t), and
  lookup(k,t') $=_{h7,h2}$ lookuptr(k,treeadd(k',e',tbltotree(t)))
  $=_{Tr}$ lookuptr(k,tbltotree(t))
  according to lemma 4.2. The induction hypothesis states that lookuptr(k,tbltotree(t)) $=_{Tr}$ lookup(k,t).

The proof can now be extended to general terms e ∈ $T_{ELEM,V}$ by replacing such terms by terms not containing function lookup, starting with the innermost occurrence(s) of this function. It can easily be seen that any term in $T_{ELEM,V}$ containing one occurrence of lookup is equivalent in either module to a term containing no occurrence of lookup. The soundness of such a replacement per term with one lookup was proved

above. Since e contains a finite number of occurrences of this function this series of replacements terminates. Hence the proof sketch is complete for general $e \in T_{ELEM,V}$.

5. FUNCTIONAL IMPLEMENTATION

5.1. The functional view

The implementation Theorem (3.2.13 in section 3.2) gives an algebraically clean criterion for implementation. However, it is not sufficient as a tool to fix implementations of functions in the classical sense: a function has a certain result value for every combination of input values. Of course the result value should depend on the input values, but it should not depend on the implementation.

The violation of this property is shown in the example below:

Let

$\Sigma_V = (\{s, t\}, \{a, b, p, q, f\})$ with $a,b \in s, p,q \in t$ and $f: s \rightarrow t$,

$\Sigma_O = (\{s, t\}, \{a, b, f\})$,

$\Sigma_H = \emptyset$,

$E = \{f(a)=p, f(b)=p\}$, and

$E' = \{f(a)=q, f(b)=q\}$.

The Σ_O-observable terms in T_V are a, b, $f(a)$ and $f(b)$. Obviously $f(a) =_{E,\Sigma_O} f(b)$ and $f(a) =_{E',\Sigma_O} f(b)$. Hence $(\Sigma_V, \emptyset, E)$ and $(\Sigma_V, \emptyset, E')$ are Σ_O-implementations of each other. However, f clearly has different result values.

Additional restrictions are needed to be able to view a term in T_O as a function (the header function) defined on tuples in T_V and with range T_V. In initial algebra semantics the 'result' is the congruence class defined by the set of equations E. Hence any term in the congruence class will do, since it fixes (for specific E and Σ_V) the class. So we need a canonical form, which is a representative for every congruence class. In a confluent and terminating term rewriting system this canonical form is called 'normal form', and it is defined by the system itself.

The following three sets of terms within T_V are induced by Σ_O:

- the *directly* Σ_O-observable terms,
- the *indirectly* Σ_O-observable terms, and
- the terms *reachable* from T_O, i.e., terms not necessarily in T_O but in the congruence class of some term in T_O.

Note that the last two sets may overlap. The input values for functions in F_O with range in S_O form a subset of the union of the first two sets. Any element of the first and the third sets could be in the range of a function in F_O.

The directly Σ_O-observable terms do not necessarily contain a desired result value. For example, a specification of *string-of-characters* might contain a function *length* from strings to integers. The set of *length*-observable terms contains the *length* function applied to numerous strings of various length, but it does not contain the integers, which is clearly the desired set of result values.

In the subsection below this idea is formalized for a specific observing function. The function has input terms, which should be well-typed, and an output term, depending on the input terms and the set of equations, which must be in a certain set of canonical terms. There is an obvious link with the theory of *abstract data types* (cf. Jones [Jon80]) here. The well-typedness of the input terms serves as precondition and the equations and a characterization of the set of canonical terms serve as postcondition.

In general one has more than one observing function, so some preliminary work has to be done to allow a decomposition of the observing set of functions into singletons.

5.2. A theory of functional implementation

5.2.1. Definitions (input-, reachable and canonical terms)
Let (Σ_V, Σ_H, E) be an algebraic specification and $\Sigma_O \subseteq \Sigma_V$. Then
a. the set $I(\Sigma_O, \Sigma_V)$ of **Σ_O-input terms over Σ_V** is defined as:

$$I(\Sigma_O, \Sigma_V) = \{t \in T(\Sigma_V) \mid \exists f \in F_O\ f: s_1 \times \cdots \times s_k \to s,\ s \in S_O\ \exists i \leq k\ t \in T_{s_i}(\Sigma_V)\}.$$

b. the set $R(\Sigma_O, \Sigma_V)$ of **Σ_O-reachable terms over Σ_V** is defined as:

$$R(\Sigma_O, \Sigma_V) = \{t \in T(\Sigma_V) \mid \exists t' \in T(\Sigma_O, \Sigma_V)\ t =_E t'\}.$$

Note that terms containing hidden functions and sorts are not considered reachable.
c. A set $C(\Sigma_O, \Sigma_V) \subseteq R(\Sigma_O, \Sigma_V)$ is a **set of canonical terms** if and only if

$$\forall t, t' \in C(\Sigma_O, \Sigma_V)\ [t =_E t' \to t = t'].$$

d. A set of canonical terms $C(\Sigma_O, \Sigma_V)$ is **complete** if and only if

$$\forall t \in T(\Sigma_O, \Sigma_V)\ \exists t' \in C(\Sigma_O, \Sigma_V)\ t =_E t'.$$

e. A **reduction to canonical terms** $\twoheadrightarrow_{C(\Sigma_O, \Sigma_V)}$ (abbreviated $\twoheadrightarrow_C$ or even $\twoheadrightarrow$) is defined as follows:

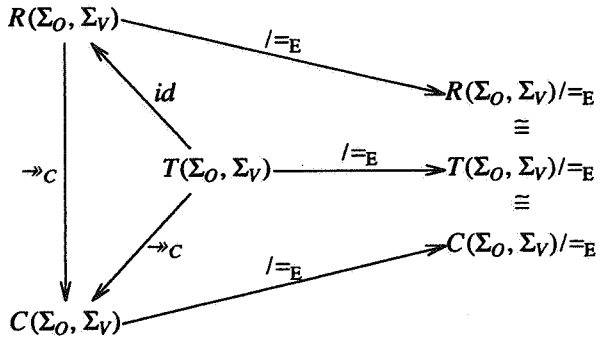
$$t \twoheadrightarrow_{C(\Sigma_O, \Sigma_V)} t' \Leftrightarrow t \in R(\Sigma_O, \Sigma_V) \wedge t' \in C(\Sigma_O, \Sigma_V) \wedge t =_E t'.$$

f. Analogous to the definitions of T_O and $T_s(\Sigma_O, \Sigma_V)$ the following shorthand conventions are adopted:

$$I_O = I(\Sigma_O, \Sigma_V),$$
$$R_O = R(\Sigma_O, \Sigma_V),$$
$$C_O = C(\Sigma_O, \Sigma_V),$$
$$I_{s,O} = I_s(\Sigma_O, \Sigma_V) = I(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V),$$
$$R_{s,O} = R_s(\Sigma_O, \Sigma_V) = R(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V) \text{ and}$$
$$C_{s,O} = C_s(\Sigma_O, \Sigma_V) = C(\Sigma_O, \Sigma_V) \cap T_s(\Sigma_V).$$

5.2.2. Some facts
a. $R(\Sigma_O, \Sigma_V) \supseteq T(\Sigma_O, \Sigma_V)$.
b. Every term in $I(\Sigma_O, \Sigma_V)$ is Σ_O-observable.
c. The converse of fact b does not hold, i.e., not every Σ_O-observable term is a Σ_O-input term.

5.2.3. Lemma
Let $C(\Sigma_O, \Sigma_V)$ be a complete set of canonical terms. The terms in $C(\Sigma_O, \Sigma_V)$ with the (adapted) functions on $R(\Sigma_O, \Sigma_V)$ form a canonical term algebra if the operation of these functions on $R(\Sigma_O, \Sigma_V)$ is restricted to reach $C(\Sigma_O, \Sigma_V)$ by application of the reduction to canonical terms $\twoheadrightarrow_C$ after the normal application of the function in $R(\Sigma_O, \Sigma_V)$. Then $C(\Sigma_O, \Sigma_V)$ as a term algebra is isomorphic to $R(\Sigma_O, \Sigma_V)/=_E$.
Proof (sketch):
Since $C(\Sigma_O, \Sigma_V)$ is complete the following diagram commutes:



By definition $C(\Sigma_O, \Sigma_V) \cong C(\Sigma_O, \Sigma_V)/=_E$ and hence $C(\Sigma_O, \Sigma_V) \cong R(\Sigma_O, \Sigma_V)/=_E$.□

5.2.4. Functional decomposition of a reduction to canonical terms

Next we want to pursue a 'divide and conquer' strategy to provide an implementation of a reduction to canonical terms $\twoheadrightarrow_C$. The decomposition chosen is made on the typed head symbol (the "function") of the term to be reduced. This allows for a separate implementation for each function. The total implementation of $\twoheadrightarrow_C$ can be constructed from the union of these separate implementations.

It should be noted that a reduction to canonical terms $\twoheadrightarrow_C$ as a total map from R_O to C_O is fixed by C_O and the congruence $/=_E$. This follows from the definition of C_O, since for every term in R_O exactly one term in C_O is in the same congruence class. So it is possible to define the map $\twoheadrightarrow_C$ as a union of partial maps to the set of canonical terms.

It is also possible to define a complete set of canonical terms implicitly by defining a (possibly partial) map $\twoheadrightarrow$ from T_O to R_O for which the following holds:

1) $\forall t \in T_O, t' \in R_O\ [t \twoheadrightarrow t' \Leftrightarrow t =_E t']$

2) $\forall t \in T_O\ card(\{t' \in R_O \mid \exists t'' \in T_O\ [t =_E t'' \wedge t'' \twoheadrightarrow t']\}) = 1.$

It can easily be seen that the range of $\twoheadrightarrow$ is a complete set of canonical terms. So a reduction to canonical terms can be described by its behaviour on terms in T_O. A well-known example of such an implicit definition is the set of normal forms defined by a confluent and terminating term rewriting system.

5.2.5. Definitions (functional implementation)

a. Let $\twoheadrightarrow_C$ be a reduction to canonical terms and let $\Sigma \subseteq \Sigma_O$. Then $\twoheadrightarrow_{\Sigma,C}$ is defined as the restriction of $\twoheadrightarrow_C$ to the domain $T(\Sigma, \Sigma_V)$.

b. Let (Σ_V, Σ_H, E) be an algebraic specification, Σ_O be a signature such that $\Sigma_O \subseteq \Sigma_V$, and C_O be a complete set of canonical terms. Then a map $\twoheadrightarrow$ is a **functional implementation** if and only if

$$\forall t \in T_O, t' \in C_O\ [t \twoheadrightarrow t' \leftrightarrow t =_E t'].$$

5.2.6. Some facts about functional implementations

a. Let (Σ_V, Σ_H, E) be an algebraic specification, $\Sigma_O \subseteq \Sigma_V$, C_O a set of canonical terms, and $\twoheadrightarrow_C |_{T_o}$ the restriction of reduction to canonical terms $\twoheadrightarrow_C$ to domain T_O. Then:

$$\twoheadrightarrow_C |_{T_o} = \bigcup_{f \in F_o} \twoheadrightarrow_{(S_o,\{f\}),C}.$$

Hence $\twoheadrightarrow_C |_{T_o}$ – and according to section 5.2.4 thus by extension $\twoheadrightarrow_C$ – can be defined for each function in Σ_O separately.

b. Let (Σ_V, Σ_H, E) and $(\Sigma'_V, \Sigma'_H, E')$ be Σ_O-implementations of each other (so $\Sigma_O \subseteq \Sigma_V \cap \Sigma'_V$). Then a functional implementation $\twoheadrightarrow_C \subseteq T_O \times C_O$ of $(\Sigma'_V, \Sigma'_H, E')$ is also a functional implementation fo (Σ_V, Σ_H, E) if

for all $t \in T_O$, $t' \in C_O$ $t =_E t' \leftrightarrow t =_{E'} t'$ holds.

c. If two algebraic specifications are Σ_O-implementations of each other and both have a functional implementation then these implementations are isomorphic.

5.2.7. Concrete representation

Eventually, we want to convert an algebraic specification into a working computer program. For this a representation function ι from the set of input terms I_O to the concrete representation of input terms is needed to be able to execute implemented functions. When confusion arises the restriction of ι to the domain $I_{s,O}$ will be written as ι_s. Additionally, a set of retrieval functions ρ_s from concrete representations of output terms to the set of canonical terms $C_{s,O}$ is needed.

This is formalized in the following

Definitions:

Let I be a set of data types for a programming language L. I is an **implementation in L of I_O and R_O** if there is a total function ι: $I_O \rightarrow$ I (the **implementation function**) and a set of (partial) functions $\{\rho_s: I \rightarrow I_{s,O} \cup R_{s,O} \mid s \in S_O\}$ (the **retrieval functions**) such that $\rho_s(\iota(t)) =_O t$ for all $t \in I_{s,O}$.

Generally, if $I_{s,O}$ is not empty then $\iota(I_{s,O})$ will correspond to a subset of a data type in L. It could very well happen that two different input types are implemented by the same data type, so differently named retrieval functions are needed for every sort. Only one name (ι) is needed for the implementation function, since the sort of the argument provides type information.

5.2.8. Implementation theorem

Let (Σ_V, Σ_H, E) be an algebraic specification, $\Sigma_O \subseteq \Sigma_V$, and C_O a set of canonical terms. Let I be an implementation in a programming language L of I_O and R_O with implementation function ι and retrieval functions $\{\rho_s \mid s \in S_O\}$, and $S(x_1, \cdots, x_n)$, storing its result in c, a program operating on I. Then the statement $S(x_1, \cdots, x_n)$, describes a functional implementation $\twoheadrightarrow_{(S_O,\{f\}),C}$ for $f: s_1 \times \cdots \times s_n \rightarrow s$, $s \in S_O$, if the following holds:

$$\{c_1 \in s_1 \wedge \cdots \wedge c_n \in s_n \wedge k_1 = \iota(c_1) \wedge \cdots \wedge k_n = \iota(c_n)\}$$

$$S(k_1, \ldots, k_n)$$

$$\{\rho_s(c) \in C_{s,O} \wedge f(c_1, \ldots, c_n) =_E \rho_s(c)\}.$$

Proof: Let function F: $s_1 \times \cdots \times s_n \rightarrow s$ be defined by $F(a_1, \ldots, a_n) = a$ if and only if $\rho_s(c) = a$ after execution of $S(\iota(a_1), \ldots, \iota(a_n))$, i.e., F is the function defined by S. Then $f(a_1, \ldots, a_n) =_E F(a_1, \ldots, a_n)$ and $F(a_1, \ldots, a_n) \in C_{s,O}$. Hence $f(a_1, \ldots, a_n) \twoheadrightarrow_{(S_O,\{f\}),C} F(a_1, \ldots, a_n)$ holds.□

5.2.9. Decidability of the conditions

It is a pleasant property of Theorem 5.2.8 that in practice satisfaction of the precondition can be computed if the implementation function ι can be computed. Since the terms in I_O are typed, a typechecking algorithm provides the statements on membership of the input terms. Generally there are no extra restrictions to ensure computability, since obviously the implementation has to be computed anyway.

The decidability of the postcondition depends on the computability of the retrieval function ρ_s, the decidability of the check on membership of the set of canonical terms C_O, and the decidability of the congruence $=_E$. The first condition is necessarily fulfilled for the same reasons as the computability of the implementation function. The second depends on the definition of C_O, which will allow computation in practical cases (who wants a canonical form wild enough to be unrecognizable as such?). The decidability of $=_E$ is not ensured in general. So a separate proof may be needed. Of course, for many classes this congruence is decidable. For specifications where the congruence is undecidable, e.g. an algebraic specification of a programming language, an implementation will to provide at least a partial decision procedure, even when it cannot be completed.

6. AN EXAMPLE: TABLES REVISITED

To illustrate the use of Theorem 5.2.8 an implementation of module `Tables` in an imperative language is given. Though the implementation again uses trees there is an important difference with the algebraic implementation in section 4 in the sense that recursion is eliminated.

The language Pascal (described in [JW78]) is chosen for the imperative implementation. This choice is motivated by its availability and by its convenient type system. Of course, any other imperative language would serve as well. It should be noted that a functional implementation is very well possible, even in Pascal, but we want to illustrate the possibility to give a correct implementation in a non-functional way.

Generally it is easier to derive a functional program from an ASF-specification, since writing an algebraic specification has strong similarities to functional programming. The specification of `Tables-as-trees`, for instance, is easily converted into a functional program for `lookup`. Thus a functional implementation has the advantage of being easily derived from the specification, and also of being faster in general than a term rewriting implementation.

The first step in the implementation is the choice of a data structure. This is provided for by the following data type declarations:

```
type  key = integer;
      elem = char;
      pointer = ↑tree;
      tree =record
                  l,r: pointer;
                  k: key;
                  e: elem;
            end;
```

In a concrete program it is necessary to bind the sorts *key* and *elem*. The choice for integers and characters is arbitrary; the only prerequisite is that an ordering must be established on the keys. A node in a tree has four fields, a left and right pointer to subtrees, and information fields for key and element.

The values `niltree` and `errorelem` require different treatment in Pascal. For the first we can use the standard notion *nil*, the second has to be declared as variable and set to some unused value.

The auxiliary functions on *key* pose no problems with the current choice, since integers are already ordered, though of course this could be much more complicated:

```
function eq (a,b:key): boolean;
begin eq := (a=b) end;

function lt (a,b:key): boolean;
begin lt := (a<b) end;
```

Next the implementation function ι must be defined. The domain of ι is $T_{\text{KEY},V} \cup T_{\text{ELEM},V} \cup T_{\text{TABLE},V}$ and its range is the union of the data types *key*, *elem* and *tree* (or rather *pointer* to *tree*) already indicated above. Since a specification of the terms of type ELEM and KEY has not been given in section 3 an identification with *elem* and *key* is assumed, so ι is 'defined' backwards by $\iota(t)=t$ for $t \in T_{\text{KEY},V} \cup T_{\text{ELEM},V}$. Hence also $\rho_{\text{ELEM}}(t)=t$, the only retrieval function needed for the example. For $t \in T_{\text{TABLE},V}$ a definition of ι can be provided as follows:

ι(`nulltable`) = *nil*

ι(`tableadd(key, elem, table)`) = *ptr*
when *treeadd*(ι(`key`),ι(`elem`),*ptr*)
is executed with *ptr* = ι(`table`).

This definition uses procedure *treeadd* defined below. It should be noted that function ι restricted to terms of type `TABLE` plays the same role as function `tbltotree` in section 4. Evidently, procedure *treeadd* below and function `treeadd` in section 4 are closely related also. A procedure with a variable parameter is a common way to handle data structures in a language like Pascal. A function definition would have the advantage of a more elegant definition of function ι, but the definition below shows that other programming styles can be handled too.

```
procedure treeadd (ky:key; el: elem; var root: pointer);
var cur, anc: pointer;
  inserted: boolean;
begin
  cur:= root;
  inserted:= false;
  while not inserted do
  begin
    if cur = nil
    then
      begin
        new(cur);
        cur↑.l:= nil; cur↑.r:= nil;
        cur↑.k:= ky; cur↑.e:= el;
        if root = nil
        then root:=cur
        else
          if lt(ky, anc↑.k)
          then anc↑.l:=cur
          else anc↑.r:=cur;
        inserted:= true
      end
    else
      begin
        if eq(ky,cur↑.k)
        then begin cur↑.e:= el; inserted:= true end
        else
          begin
            if lt(ky,cur↑.k)
            then begin anc:=cur; cur:=cur↑.l end
            else begin anc:=cur; cur:=cur↑.r end
          end
      end
  end
end;
```

The proof of correctness of this implementation closely resembles the proof sketch in section 4. Hence it will be an even more concise sketch. Following the lead in section 4.1 we provide two well-formedness predicates on structures of type *pointer* (to *tree*), again called *searchtree* and *set-of-keys*. They are defined as follows (*ptr* of type *pointer* to *tree* and *j*, *k*, *l*, resp. *e*, of type *key*, resp. *entry*):

- *searchtree*(*nil*) = true;
- *searchtree*(*ptr*) = *searchtree*(*ptr*↑.*l*) ∧ *searchtree*(*ptr*↑.*r*) ∧
  ∀ *j* ∈ *set-of-keys*(*ptr*↑.*l*) [*lt*(*j*,*ptr*↑.*k*) = *true*] ∧
  ∀ *l* ∈ *set-of-keys*(*ptr*↑.*r*) [*lt*(*ptr*↑.*k*,*l*) = *true*];

and
- *set-of-keys(nil)* = ∅;
- *set-of-keys(ptr)* = *set-of-keys(ptr↑.l)* ∪ {*ptr↑.k*} ∪ *set-of-keys(ptr↑.r)*.

This allows us to state the well-formedness of implemented terms by providing the following parallel to Lemma 4.2.*a*:

6.1. Second well-formedness lemma for trees (part a)
Let *ptr* be of type *pointer* and $t \in T_{TABLE,V}$. Then

$$ptr = \iota(t) \rightarrow searchtree(ptr).$$

Proof by induction on the number of nodes in the list (omitted).

Next we provide the function *lookuptr*:

```
function lookuptr (ky: key; root: pointer): elem;
var cur: pointer;
  searched: boolean;
begin
  cur:= root;
  searched:= false;
  while not searched do
  begin
    if cur = nil
    then begin lookuptr:= errorelem; searched:= true end
    else
      begin
        if eq(ky,cur↑.k)
        then begin lookuptr:= cur↑.e; searched:= true end
        else
          begin
            if lt(ky,cur↑.k)
            then cur:= cur↑.l
            else cur:= cur↑.r
          end
      end
  end
end;
```

Presently, a lemma similar to lemma 4.2.*b* can be formulated. It states that *lookuptr* is well-defined for single additions to a well-formed tree.

6.2. Second well-formedness lemma for trees (part b)
Let k, k′ be of sort KEY, e of sort ENTRY, and t of sort TREE, and let $ptr' = \iota(\texttt{treeadd(k', e, t)})$. Then

$$ptr = \iota(t) \rightarrow$$
$$[eq(\iota(k),\iota(k')) = true \rightarrow lookuptr(\iota(k), ptr') = e] \wedge$$
$$[eq(\iota(k),\iota(k')) = false \rightarrow lookuptr(\iota(k), ptr') = lookuptr(\iota(k), ptr)]$$

The *proof* follows directly from the observation that $ptr' = \iota(\texttt{treeadd(k', e, t)})$ is defined in terms of $ptr = \iota(t)$.

According to Theorem 5.2.8 it is now sufficient to prove (E the set of equations from module Tables):

$\{k \in KEY \wedge tbl \in TABLE \wedge ky = \iota(k) \wedge root = \iota(tbl)\}$

$elt := lookuptr(ky, root)$

$\{\rho_{ELEM}(elt) \in C_{ELEM,O} \wedge lookup(k, tbl) =_E \rho_{ELEM}(elt)\}$.

This follows immediately from lemma 6.2, and the definition of ρ_{ELEM}.

7. Conclusions

7.1. The results

The paper provides a functionally oriented (black box) approach to the implementation of modular algebraic specifications. The main advantages are listed below.

- It provides a theoretical background for the *separate implementation* of modules.
- The implementation above is based on the *initial* behaviour of certain functions, the observing functions. This provides an intuitively clear semantics.
- A correctness criterion for implementations is given in Hoare logic, allowing the application of *standard optimization techniques*. In algebraic terms this means that functions which are not observing may have more or less *final* semantics.
- The combination of separate implementation and (hence separate) optimization allows the construction of a *library* of (possibly optimized) modules.

The loss of the *initial algebra semantics* might instead be listed as a disadvantage. Terms are only judged different when they have different effects (**confusion** is allowed) and other invisible terms (**junk**) may be introduced. On the one hand, precisely these two "undesirable" effects allow the introduction of optimal implementations. On the other hand, they make the semantics of a module less clear to the user (i.e., someone writing a module importing the optimized module). This problem is minimized by the fact that the criteria for use of the module, allowing the set of observing terms only, are rather easy.

7.2. Further research

Both theoretical and practical extensions of the work in this paper are planned for. The former include:

- Investigation of the significance for *import semantics* in an algebraic specification formalism with initial algebra semantics, except for observable imports.
- The *combination* with an implementation as a *term rewriting system* of the importing module of an observable implementation has to be investigated to allow for automatic translation of modules built around an observable module.

Of more practical nature are

- The design and implementation of a *module library*, containing efficient (e.g. built-in) implementations.
- The construction of an *implementation* of modules on top of the module library, using the normalization (i.e., elimination of imports, renamings and bindings by combining modules) semantics from ASF [BHK87] for the top level modules.

Acknowledgements

Discussions with P.R.H. Hendriks, P. Klint, and especially J. Heering greatly improved the content of this paper. They, J. Rekers, and the referees for ESOP '88 made useful comments on one or more earlier versions.

REFERENCES

[Bac86] R.C. BACKHOUSE (1986). *Program Construction and Verification*, Prentice-Hall.

[Bak84] C. BAKER-FINCH (1984). "Acceptable models of algebraic semantics," *Australian Computer Science Communications*, vol. 6, no. 1, pp. 5-1/10, Proceedings of the Seventh Australian Computer Science Conference, Adelaide, ed. C.J. Barter.

[BHK85] J.A. BERGSTRA, J. HEERING, and P. KLINT (1985). "Algebraic definition of a simple programming language," Report CS-R8504, Centre for Mathematics and Computer Science, Amsterdam.

[BHK86] J.A. BERGSTRA, J. HEERING, and P. KLINT (1986). "Module algebra," Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam.

[BHK87] J.A. BERGSTRA, J. HEERING, and P. KLINT (1987). "ASF - an algebraic specification formalism," Report CS-R8705, Centre for Mathematics and Computer Science, Amsterdam.

[BK86] J.A. BERGSTRA and J.W. KLOP (1986). "Conditional rewrite rules: confluence and termination," *Journal of Computer and System Sciences*, vol. 32, no. 3, pp. 323-362.

[BT82] J.A. BERGSTRA and J.V. TUCKER (1982). "The completeness of the algebraic specification methods for computable data types," *Information and Control*, vol. 54, no. 3, pp. 186-200.

[BT83] J.A. BERGSTRA and J.V. TUCKER (1983). "Initial and final algebra semantics for data type specifications: two characterization theorems," *SIAM Journal on Computing*, vol. 12, no. 2, pp. 366-387.

[BDMW81] M. BROY, W. DOSCH, B. MÖLLER, and M. WIRSING (1981). "GOTOs – a study in the algebraic specification of programming languages," in *GI – 11. Jahrestagung*, ed. W. Brauer, Informatik-Fachberichte, vol. 50, pp. 109-121, Springer-Verlag.

[Die86] N.W.P. VAN DIEPEN (1986). "A study in algebraic specification: a language with goto-statements," Report CS-R8627, Centre for Mathematics and Computer Science, Amsterdam.

[DE84] K. DROSTEN and H.-D. EHRICH (1984). "Translating algebraic specifications to Prolog programs," Informatik-Bericht Nr. 84-08, Technische Universität Braunschweig.

[FGJM85] K. FUTATSUGI, J.A. GOGUEN, J.-P. JOUANNAUD and J. MESEGUER (1985). "Principles of OBJ2", in *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pp. 52-66, ACM.

[GIP86] J. HEERING, J. SIDI & A. VERHOOG (eds.), (1986). "Generation of interactive programming environments - GIPE. Intermediate report," Report CS-R8620, Centre for Mathematics and Computer Science, Amsterdam.

[GIP87] N.N. (1987). *GIPE, Generation of Interactive Programming Environments*, ESPRIT Project 348 second annual review report, SEMA-METRA.

[GM82] J.A. GOGUEN and J. MESEGUER (1982). "Universal realization, persistent interconnection and implementation of abstract modules," in *Proceedings 9th International Conference on Automata, Languages and Programming*, eds. M. Nielsen & E.M. Schmidt, Lecture Notes in Computer Science, vol. 140, pp. 265-281, Springer-Verlag.

[GM84] J.A. GOGUEN and J. MESEGUER (1984). "Equality, types, modules, and (why not?) generics for logic programming," *Journal of Logic Programming*, vol. 2, pp. 179-210.

[GMP83] J.A. GOGUEN, J. MESEGUER and D. PLAISTED (1983). "Programming with parameterized abstract objects in OBJ", in *Theory and Practice of Software Technology*, eds. D. Ferrari, M. Bolognani & J.A. Goguen, pp. 163-193, North-Holland.

[HO80] G. HUET and D.C. OPPEN (1980). "Equations and rewrite rules: a survey," in *Formal Language Theory, Perspectives and Open Problems*, ed. R.V. Book, pp. 349-405, Academic Press.

[JW78] K. JENSEN and N. WIRTH (1978). *Pascal: User Manual and Report* (second edition), Springer-Verlag.

[Jon80] C.B. JONES (1980). *Software Development: a Rigorous Approach*, Prentice-Hall.

[Kam83] S. KAMIN (1983). "Final data types and their specification," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 1, pp. 97-123.

[LS84] J. LOECKX and K. SIEBER (1984). *The Foundation of Program Verification*, Wiley-Teubner.

[MG85] J. MESEGUER and J.A. GOGUEN (1985). "Initiality, induction, and computability," in *Algebraic Methods in Semantics*, eds. M. Nivat & J.C. Reynolds, pp. 459-541, Cambridge University Press.

[ODo85] M.J. O'DONNELL (1985). *Equational Logic as a Programming Language*, MIT Press.

[ST85] D. SANNELLA and A. TARLECKI (1985). "On observational equivalence and algebraic specification," in *Mathematical Foundations of Software Development. Proceedings International Joint Conference on Theory and Practice of Software Development, TAPSOFT '85*, eds. H. Ehrig, C. Floyd, M. Nivat & J. Thatcher, Lecture Notes in Computer Science, vol. 185, pp. 308-322, Springer-Verlag.

[Wan79] M. WAND (1979). "Final algebra semantics and data type extensions," *Journal of Computer and System Sciences*, vol. 19, pp. 27-44.