Centrum voor Wiskunde en Informatica

Centre for Mathematics and Computer Science

J.C.M. Baeten, J.A. Bergstra

Design of a specification language by abstract syntax engineering

Computer Science/Department of Software Technology Report CS-R8934 September

1989

Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J.C.M. Baeten, J.A. Bergstra

Design of a specification language by abstract syntax engineering

Computer Science/Department of Software Technology Report CS-R8934 September

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 Research (N.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Design of a Specification Language by Abstract Syntax Engineering

J.C.M. Baeten
Dept. of Software Technology, Centre for Mathematics and Computer Science, P.O.Box 4079, 1009 AB Amsterdam, The Netherlands

J.A. Bergstra
Programming Research Group, University of Amsterdam, P.O.Box 41882, 1009 DB Amsterdam, The Netherlands
Department of Philosophy, State University of Utrecht, Heidelberglaan 2, 3584 CS Utrecht, The Netherlands

In this paper, we design a specification language in an entirely algebraic style. We describe the language in terms of abstract syntax only. We argue that this is the correct approach in language design.

1980 Mathematics Subject Classification (1985 revision): 68Q45, 68Q55, 68Q65, 68Q50.
1987 CR Categories: F.4.3, D.2.10, D.3.1, D.3.3.
Key words & Phrases: abstract syntax, specification languages, module algebra, ASF.
Note: This work is partially sponsored by ESPRIT contract 432, An Integrated Formal Approach to Industrial Software Development (METEOR).

1. INTRODUCTION

This report contains a first step towards a design of a specification language like ASF [BHK 89] in an entirely algebraic style. The language BMASF that is the result of this design effort is simpler than ASF because it has no parametrisation mechanism. It is better in the sense that imports and exports have been worked out in a more satisfactory way. We will first try to motivate in detail the reasons for the approach we have taken and then work out the language design. It should be noticed in advance that the language BMASF as such has no pretensions and that it is the method of its design via abstract syntax which is the real objective of the paper.

In order to clarify the motivation for our work we start by listing some points of view that we have developed about the design of specification languages. After observing several attempts to define specification languages it has become clear (to us) that the following phenomena seem to be unavoidable:

(i) Above a certain complexity one runs into semantic trouble because of unexpected interactions of features that are combined in the language being designed. These semantic problems are almost never solved by theoretical work because they depend on very special peculiarities of the features as they are embedded in the language.

Report CS-R8934
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

For instance, parameter passing mechanisms for abstract data types allow many degrees of freedom that have not been addressed by theoretical work and in which wrong language designs cannot yet be singled out by pointing at the way in which they depart from known theory. Also, if information hiding is present, the interaction between parametrisation and hiding becomes complicated. If on top of that, operational aspects enter the scene, one has to worry about the interaction between information hiding and the abstraction mechanisms of process theory.

(ii) If the language is designed in a syntax readable for human beings, there is always a next step in which an abstract syntax has to be designed. Typically, in the design of abstract syntax one attempts to define a more clear cut language with fewer semantic problems. Usually, this will not quite succeed because the abstract syntax has to be derived from the given concrete syntax and that will induce the introduction of features that are not really primitive ones.

(iii) Notoriously, one will feel a need to redesign parts of the language within two years after its conception. The difficulty is then to have the tools for the language written in such a way that their code or at least their design is somehow reusable. It is very unpleasant to construct tools for a language that has already been declared outdated by its chief designer. But exactly this phenomenon occurs time and again.

(iv) Upward compatibility is the slogan that should help in having previous work on a language reusable. Disappointingly, it is very difficult to design languages in an upwardly compatible way. More often than not, the redesign of a language will shed light on how to improve its existing version without adding new features. Moreover, these improvements may be needed if a second version of the language is to incorporate quite complex new features.

(v) The main difficulties are caused by the fact that a readable syntax for a language needs to be provided with efficient declaration, type inference and type checking schemes, because otherwise the human reader will soon loose his/her grasp of a piece of syntax due to enormous redundancy. (For computers this problem is much less pressing.) Exactly these mechanisms are closely connected with the design of the concrete vertical syntax and the particular packaging of the features that is employed. Those particularities are however quite hard to keep alive when a next version of the language is made. By the way, these problems would be less pressing if the ambition to provide a human readable textual syntax was given up in favour of a graphical or object-oriented way of working.

(vi) There is no place in the software engineering lifecycle for small scope specification languages (thematic languages and combination languages in the terminology of [BR 89]) that cannot be extended to wide spectrum languages. One will never tolerate the enormous overhead of recoding a formal specification. It should be noticed that adapting a complex specification to a formalism with slightly different mechanisms for import, export and parametrisation is fairly unpleasant. It follows that there is little reason to invest in small languages that have their key features designed in such a way that extension to larger languages is hardly possible, or obviously unrewarding. Thus it follows that in particular efforts aimed at the design of small and specialized specification languages should ensure that all features are incorporated in a generic (extendible) way.

To consider an example, ASF [BHK 89] is a language that has some of its key features worked out in a not entirely satisfactory way: in particular the normalization mechanism is such that normalization must be done inside out, since other strategies may lead to essentially different normal forms, and

besides this, the information hiding mechanism may only be applied to flat specifications, which is a rather pointless restriction. We view ASF as a step in a bootstrapping process. Its first use should be to assist in the design of much better languages of its own kind.

2. ABSTRACT SYNTAX DESIGN

The style of designing specification languages that is investigated in this report is to design abstract syntax only. The following working hypotheses (2.1 - 2.5) underlie the strategy. Of course validation of these assumptions is a difficult matter that requires substantial experimentation.

2.1 Abstract syntax can be represented in the format of algebraic specifications using many sorted algebra with total functions. At the level of abstract syntax, the issues of type checking, type inference, declarations and the use of declarations for type inference are totally absent.

2.2 All semantic problems of a language should be dealt with at the level of its abstract syntax. Language features that do not allow a coding in an abstract syntax are to be avoided.

2.3 Abstract syntax can be designed in an (almost) upwardly compatible way. If an abstract syntax specification is to be extended with new features, it almost never is a matter of just adding additional sorts functions and equations. Nevertheless, the modifications may well be very limited.

A typical example is that one needs additional structure in a name space. This will require a small redesign of the use of the names. But the general structure of an abstract syntax description will not be affected by that.

2.4 Abstract syntax can be described in such a way that all equations which describe the semantics of its key ingredients can be added without risking inconsistency of the full specification. Thus when designing abstract syntax, constructions must be avoided that are inconsistent with key semantic identities.

For instance, having an abstract syntax depend on a function that determines the length of an expression will usually prevent one to impose any non-trivial identities on the sort of those expressions. This is useful only if no such identifications are to be expected during the semantic analysis. On the other hand, when tooling a language one will use algorithms that act on the free term algebra of the abstract syntax and often these terms will have to be represented in a form that allows efficient manipulations (e.g. by the use of pointers).

2.5 An increasing family of languages can be designed at the level of abstract syntax. It is possible to analyse features at an abstract level in such a way that one can be confident that language extension will not lead to an entirely different view of the features. In order to guarantee this, it is crucial that many semantic equations can be imposed on the abstract syntax. Indeed it should be noticed that all generic constructions should be representation independent as much as possible. This is best guarantied by

having them consistent with a semantic model that identifies many systems. This implies that one will search for the most abstract semantics that is available.

This is a very tricky area because the more features one introduces, the less abstract the semantic model can be! In the case of process descriptions, this is a reason to use bisimulation semantics rather than less discriminating semantics, such as trace semantics. Features like interrupts, fair abstraction, deadlock analysis and structured operational semantics are harder if not impossible to model in trace semantics. It is also a reason to have only limited interest in fully abstract models. If later on, new features are added to a language, a fully abstract model may suddenly be inconsistent with the novel feature. This can happen to any other model as well, of course, but aiming at full abstraction clearly maximizes the risk.

3. AN OPEN PROBLEM ABOUT THE METHODOLOGY OF LANGUAGE DESIGN

Let us imagine a project that is carried out as follows:

(A) Design a family of specification languages with increasing expressive power and complexity at the level of abstract syntax and reuse almost all of each language description in the design of the next language.

(B) Generate in each stage an elementary tool set (ETS) consisting of a concrete textual (vertical, structured) syntax together with the following: a parser, a type checker, a prototyping tool, an interactive editor, a connection with a software engineering data base and a version management system, a cross reference information generator, and an automatic translation to and from earlier stages of the language, as well as an interface with specialized support programs for debugging, verification, proof editing, proof checking, graphical support and object-oriented representation.

(C) Have the overheads on (B) above so small that it is possible to work in (A) with many small steps rather than with a few big steps.

Now the question is whether or not such an approach is feasible. We have no opinion in advance what is the answer to this methodological question. Our point of view is that it is worth trying for the sake of a research project and that its practical value must de determined later on by other people. Thus the open problem is turned into a working hypothesis without any claim as to its validity.

Clearly our research can never generate a negative answer to the methodological question because inability to carry out the project proves nothing about its feasibility. By being successful, it can at best generate positive evidence. We will try to catch the motivation of the work in a few key phrases that may be useful in these times of inflation of terminology.

3.1 ABSTRACT SYNTAX ENGINEERING AND THE ABSTRACT SYNTAX ENGINEERING HYPOTHESIS

Abstract syntax engineering is the incremental design of languages via their abstract syntax. The hypothesis claims that this style of working is economically more efficient than the conventional design via BNF grammars.

3.2 ALGEBRAIC SPECIFICATION HYPOTHESIS (FOR ABSTRACT SYNTAX ENGINEERING)

This hypothesis claims that one gets benefit from the use of algebraic specifications in the case of abstract syntax engineering. In particular the restriction to many sorted algebra with finitely many sorts and total functions is supposed useful.

It may even be useful to structure the signature of the algebra itself as a finite partial algebra. Infinite signatures are to be avoided and must be counted as an indication of limited success in abstract syntax design (of course infinite signatures may play an important role in intermediate stages of a design).

3.3 MEANING IS A CONGRUENCE ON ABSTRACT SYNTAX

This claim is not exactly new. It was analysed in depth in [J 89] and named Frege's principle. The difficulty is to adhere to this slogan when the abstract syntax gets more complex. In the context of the abstract syntax engineering hypothesis, this means that the abstract syntax must be made compatible with most (and preferably all) semantic identities that come about during semantic analysis of fragments of the language.

3.4 THE HYPOTHESIS OF ABSENCE OF CANONICAL MEANING

From a certain complexity onwards, there is no canonical model for the semantics of an abstract syntax. In particular, there is no such thing as the 'real' practical meaning that users have in their mind and that theorists fail to write down in a concise way, due to an overemphasis of their mathematical standards and rigour. On the contrary, the practical user has often only an intuitive semantic view on fragments of an abstract language, and simply ignores the question how to integrate these fragments into a consistent picture.

In the paper [BHK 88] on module algebra, it has been emphasised that already a very few constructors for structured algebraic specifications generate a setting for which different useful semantic models can be found and the selection of a single most convincing model seems to be impossible. There is no indication that the practicioner's mind contains hidden semantic information that would resolve the semantic ambiguity of module algebra.

The practical consequence is that abstract syntax must preferably be designed in such a way that the full spectrum of semantic models known for fragments of the language can still be captured by considering the right congruence on the abstract syntax.

3.5 THE ABSTRACT SYNTAX DESIGN RULE: OPTIMIZATION OF LOOSE SEMANTICS IN ALGEBRAIC ABSTRACT SYNTAX ENGINEERING

This rule makes explicit the consequences of the hypothesis of absence of abstract syntax. It emphasizes that while designing an abstract syntax one should not optimize the fit with a given semantic model but rather ensure that a maximum of semantic options is left open. The algebraic specification of the abstract syntax will usually have a loose semantics in the sense that its initial algebra is not at all the model with the deepest semantic pretensions. Rather a family of models should exist that reflect the different ways in which fragments of the language can be provided with meaningful semantic models.

3.6 Initial overspecification is unavoidable and even beneficial

This final point lies at the heart of the algebraic approach. Once an algebraic specification of abstract syntax has been manufactured, the unavoidable question is: why have just these equations been selected and is this not a totally arbitrary choice? Moreover, the axioms may not allow semantics options that are popular in modern research because the equations identify too much.

The answer to this is that these problems are unavoidable in the algebraic method and that designers who dislike these uncertainties should not employ these techniques. Let us consider the case of group theory as an example. Once the axioms of groups have been written down, a substantial amount of useful theory can be generated. After some time, Abelian groups become important and this is no problem because the group axioms are consistent with the additional axiom of commutativity. Still later however, the success of group theory is such that one starts investigating semi-groups. This is a very different matter because an entire operation has been left out and the specification has been essentially weakened. Suppose that semi-groups after all are the really useful concept and the groups are a subcase of less interest. Then from a methodological point of view, the specification of groups formed an initial overspecification of the semi-groups. How much damage has this caused? We claim that no damage can be observed at all. On the contrary, the initial overspecification of a concept (semi-groups) that later on was found to be very useful has strongly guided the intuition on how to get the appropriate rewards from the algebraic model of the mechanisms involved. Even if semi-groups are the real thing after all, one can never ensure that future generations will not make big progress in the field of 'non-associative semi-groups'!

All of this boils down to the point of view that there will never be a truly undisputed set of axioms for any given signature, even if one has an agreement on the intuitions that the operators of the signature must support. This is true for every field of mathematics, including geometry and analysis, and it would be very optimistic to suppose that discrete systems theory constitutes an exception, just because these discrete systems are man made.

4. On the role of software support for algebraic specifications in abstract syntax engineering

The first role of software support is simply that with abstract syntax specifications becoming larger, type checking becomes useful. Although we have not been using the ASF system as described in [BHK 89], the type checking facilities of such a system will be needed to avoid mistakes in the specifications. We will discuss which further use a specification in a language as ASF might have.

Secondly, one might specify a normalization algorithm for structured specifications in BMASF using equations. If the resulting rewrite system is complete it can be prototyped by means of the ASF system. This implies that a non-trivial transformation has to be transformed into an executable (complete) TRS. Given a vertical syntax, one may specify decision algorithms that decide whether a specification satisfies certain design rules (e.g. not containing type-incorrect subexpressions).

Harder but still conceivable is that a syntax directed editor for a vertical syntax for BMASF is specified in ASF and that the ASF system is used to prototype such an environment. Further, one can

imagine a specification in ASF of a system that realizes an operational interpretation of a BMASF specification.

Also one may specify in ASF what exactly has to be done if separate type checking of a modular BMASF specification is to be realized.

5. PRELIMINARIES FOR THE BMASF SPECIFICATION

We need conventions for the use of names. These conventions are not given in the system ASF but should be workable when translating the specification to ASF.

5.1 SORTS AND LISTS.

Primitive sorts are named with identifiers made from one or more capital letters and from digits always beginning with a letter. Some constructed sort names may involve brackets. We will use only one sort constructor for lists. The underlying view is, that the algebra of sorts is a partial one and sort expressions are only defined if the axioms imply this.

The letter L always means that the sort denotes finite lists of another sort. So L(AB1) denotes the sort lists of AB1. If such lists are needed, the sort must be explicitly specified, and then the constructor operations and the empty list will come by default. We are using the notation of [BI 87] and [M 86] for lists. When needed, extensions of the 'automatic signature introduction mechanism' can be defined in order to allow more of the operators of [BI 87] and [M 86] to be used without declaration. A useful subset of these notations is selected in [J 89]: prefix, drop, map, reduce, right reduce, and transpose. Because none of the concrete notations proposed by Bird and Meertens overlaps in an unpleasant way with notations that we intend to use or to import from previous work on module algebra (or process algebra), we will make sure that these notations will not be overloaded in our proposals with quite different meanings as well, so that some notational consistency can be achieved in the end. Our subset is collected in the module FPN2; further extensions can be coded in extension modules when needed. Notice that further extensions require that standard names are introduced for function types just as for list types. So it is plausible to denote the type of functions from X to Y by F(X, Y). We get elements of this function space by an operator ^ applied to a function from X to Y.

The predefined lists of X are always structured by means of the following operators:

[]	the empty list of X-objects;
[_]	embedding X into L(X);
++	associative concatenation of lists;
_:_	prefixing a list with an object.

In addition, lists of fixed length can be denoted with the following notation:

[a, b, c, d] denotes a list with elements a, b, c, and d.

Notice that the list notation provides no type information. This implies a substantial overloading of the notation that must be resolved by making sure that the type of listed objects is clear without context information.

The reduce operator / must be applied to a function h from pairs of objects to objects (i.e. h: $X \times X \rightarrow X$). The value of reduce of a function on the empty list, /h([ ]), must be an element of X. This ele-

ment must be supplied, in each case, as the third argument of the reduce function, so reduce will have three arguments, viz. a function (element of F(CP(X, X), X)), a list (element of L(X)) and an element (of X).

In every application, h is commutative, associative, and has a unit element equal to /h([ ]) (i.e. h(x, /h([ ])) = x). The intuition is that reduce applies h to the list 'consecutively', so e.g.

/(+, [3,4,6], 0) = 3 + 4 + 6.

5.2 NATURAL NUMBERS.

There is a fixed sort NAT with function succ and constant 0.The cardinality function #: L(X) → NAT is automatically introduced with every list sort. We also have a fixed sort BOOL with functions &,∨,¬ and constants T,F.

5.3 CONSTANT NAMES AND FUNCTION NAMES.

Constant and function names can be systematically disambiguated by subscripting them with the sorts of their arity. These subscripts may be skipped in the presentation of a specification as long as disambiguation is possible in an unambiguous way (or: even more liberal but also less clear in its consequences, as long as all correct disambiguations that can be imagined can be proven equivalent by means of the axioms).

	module FPN0	list manipulation operators
	begin	
	begin signature	
	L(X)	sort of lists of X-objects (X is a parameter)
	[ ]: L(X)	empty list
	[_]: X → L(X)	embedding of X into L(X)
	_:_ : X × L(X) → L(X)	prefixing
	_++ _: L(X) × L(X) → L(X)	concatenation
	[_,...,_]	constructor notation scheme for finite name lists with flexible arity
	#: L(X) → NAT	length of list (automatically generated with L(X) and NAT)
	_∈ _: X × L(X) → BOOL	element of a list
	end signature	
	begin equations	
	variables x,y ∈ X, l,m ∈ L(X)	
1	x:[ ] = [x]	
2 n	x:[y1,...,yn] = [x,y1,...,yn] (n ∈ ℕ)	
3	[ ] ++ l = l	
4	(x:l) ++ m = x:(l ++ m)	
5	#([ ]) = 0	

```
6          #(x:l) = succ(#(l))
7          x ∈ [ ] = F
8          x ∈ y:l = eq(x,y) ∨ x∈l
       end equations
   end module FPN0

   module FPN1                          cartesian products
   begin
      begin signature
         CP(S1,...,Sn)                  Cartesian product of sorts S1 to Sn
         ⟨_,...,_⟩: S1 × ... × Sn → CP(S1,...,Sn)   construction of n-tuple
         πk:CP(S1,...,Sn) → Sk          k-th component (for all k with 1≤k≤n)
      end signature

      begin equations
      variables si ∈ Si (i=1,...,n)
9        πk(⟨s1,...,sk,...,sn⟩) = sk          i=1,...,n
      end equations
   end module FPN1

   module FPN2                          functional programming constructions
   begin
      begin signature
         F(X, Y)                        functions from X to Y (parameters X,Y)
         U^ : F(X,Y)                    embedding in function space, for U: X → Y
         _(_): F(X, Y) × X → Y          function application
         _∘_: F(X, Y) × F(Y, Z) → F(X, Z)   function composition
         *_: F(X, Y) × L(X) → L(Y)      map
         /: F(CP(X,X),X) × L(X) × X → X reduce
      end signature

      begin equations
      variables x,x' ∈ X, y ∈ Y, f ∈ F(X,Y), g ∈ F(Y,Z), h ∈ F(CP(X,X), X), l ∈ L(X), U: X → Y
10       U^(x) = U(x)
11       f∘g(x) = f(g(x))
12       *f([ ]) = [ ]
13       *f(x:l) = f(x):*f(l)
14       /(h, [ ], x') = x'
15       /(h, x:l, x') = h(x, /(h, l, x'))
      end equations
   end module FPN2
```

```
module BOOLEANS
begin
   begin signature
      BOOL                          sort of booleans
      T: BOOL                       true
      F: BOOL                       false
      _&_: BOOL × BOOL → BOOL       conjunction
      _∨_: BOOL × BOOL → BOOL       disjunction
      ¬_: BOOL → BOOL               negation
   end signature

   begin equations
   variables b,c ∈ BOOL
16    T ∨ b = T
17    b ∨ T = T
18    F ∨ F = F
19    ¬T = F
20    ¬F = T
21    b & c = ¬((¬b) ∨ (¬c))
   end equations
end module BOOLEANS

module NATURALS
begin
   begin signature
      NAT                           natural numbers
      0: NAT                        zero
      succ: NAT → NAT               successor
      eq: NAT × NAT → BOOL          equality
   end signature

   begin equations
   variables n,m ∈ NAT
22    eq(n,n) = T
23    eq(0, succ(n)) = F
24    eq(succ(n), 0) = F
25    eq(succ(n), succ(m)) = eq(n,m)
   end equations
end module NATURALS
```

Note that when one writes an equation it must be ensured that this equation has a proper typing. Therefore an equation such as

$$X = [succ(0)] +\!\!+ [succ(succ(0)), X]$$

is unreadable because there is no type assignment for X.

6. SPECIFICATION OF BMASF

Now we will start the design of the specification language BMASF. BMASF combines Basic Module Algebra (see [BHK 88]) with ASF. Figure 1 shows part of the signature of the first module, the module ELEMENTS (renamings, and signature of NAT and BOOL are not shown).

In the equations of this module, we will use *meta-variables*. These meta-variables range over a finite set of (regular) variables, and are used to cut down on the number of equations. It is straightforward to expand the equations in which meta-variables occur, in order to eliminate them.

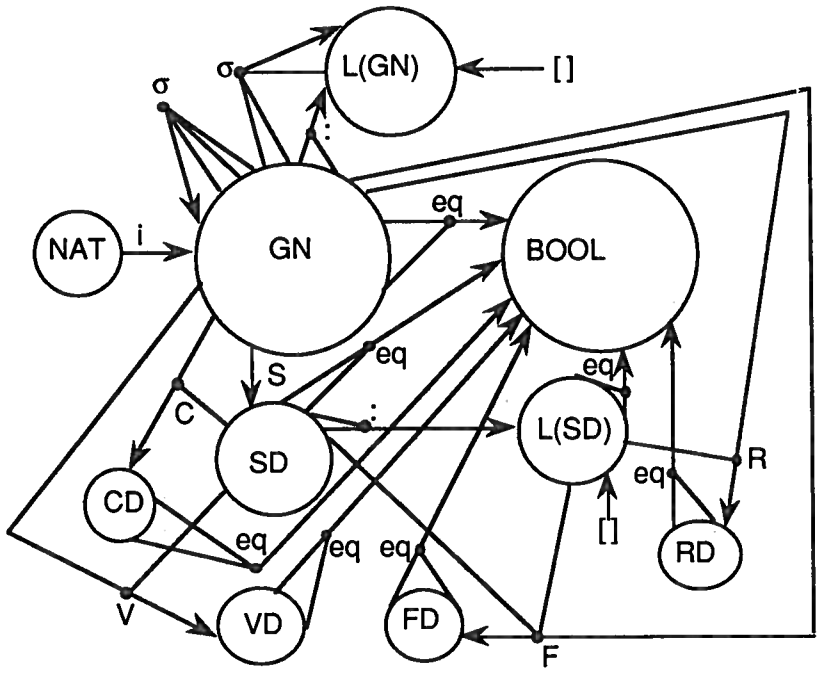

FIGURE 1. Part of the signature of ELEMENTS.

```
module ELEMENTS
begin
    begin signature
        GN                                          sort of (general) names
        eq_GN×GN→BOOL: GN × GN → BOOL               equality on names
        i_NAT→GN: NAT → GN                          embedding of numbers in names
        σ_GN×GN×GN→GN: GN × GN × GN → GN            name permutation
        σ_GN×GN×L(GN)→GN: GN × GN × L(GN) → L(GN)   name permutation on a list

        SD                                          sort of sort declarations
        S:_: GN → SD                                sort names
```

$eq_{SD\times SD\to BOOL}$: SD × SD → BOOL equality on sorts
$eq_{L(SD)\times L(SD)\to BOOL}$: L(SD) × L(SD) → BOOL equality on lists of sorts

CD sort of constant declarations
C:_:_: GN × SD → CD constant names
$eq_{CD\times CD\to BOOL}$: CD × CD → BOOL equality on constants

RD sort of relation declarations
R:_:_: GN × L(SD) → RD relation names
$eq_{RD\times RD\to BOOL}$: RD × RD → BOOL equality on relations

FD sort of function declarations
F:_:_→_: GN × L(SD) × SD → FD function names
$eq_{FD\times FD\to BOOL}$: FD × FD → BOOL equality on functions

VD sort of variable declarations
V:_:_: GN × SD → VD variable names
$eq_{VD\times VD\to BOOL}$: VD × VD → BOOL equality on variables

ATREN sort of atomic renamings
id: ATREN identity renaming
$r_{SD\times GN\to ATREN}$: SD × GN → ATREN permutation of sort names
$r_{CD\times GN\to ATREN}$: CD × GN → ATREN permutation of constant names
$r_{FD\times GN\to ATREN}$: FD × GN → ATREN permutation of function names
$r_{RD\times GN\to ATREN}$: RD × GN → ATREN permutation of relation names
$r_{VD\times GN\to ATREN}$: VD × GN → ATREN permutation of variable names
$\_\cdot\__{ATREN\times SD\to SD}$: ATREN × SD → SD application of atomic renaming
$\_\cdot\__{ATREN\times CD\to CD}$: ATREN × CD → CD application of atomic renaming
$\_\cdot\__{ATREN\times FD\to FD}$: ATREN × FD → FD application of atomic renaming
$\_\cdot\__{ATREN\times RD\to RD}$: ATREN × RD → RD application of atomic renaming
$\_\cdot\__{ATREN\times VD\to VD}$: ATREN × VD → VD application of atomic renaming

end signature

begin equations
variables n,m ∈ NAT, g,h,k,p ∈ GN, e ∈ L(GN), s,s',t,t' ∈ SD, l,j ∈ L(SD), c ∈ CD, f ∈ FD, q ∈ RD, v ∈ VD
meta-variables φ,ψ

26 eq(i(n),i(m)) = eq(n,m)
27 σ(g,h,g) = h
28 σ(g,h,h) = g
29 eq(g,k) = F & eq(h,k) = F ⇒ σ(g,h,k) = k
30 σ(g,h,[ ]) = [ ]
31 σ(g,h,k:e) = σ(g,h,k):σ(g,h,e)
32 eq(S:g, S:h) = eq(g,h)

```
33        eq([ ],[ ]) = T
34        eq([ ],s:l) = F
35        eq(s:l,[ ]) = F
36        eq(s:l,t:j) = eq(s,t) & eq(l,j)
37        eq(C:g:s, C:h:t) =  eq(g,h) & eq(s,t)
38        eq(R:g:l, R:h:j) = eq(g,h) & eq(l,j)
39        eq(F:g:l→s, F:h:j→t) = eq(g,h) & eq(s:l,t:j)
40        eq(V:g:s, V:h:t) =  eq(g,h) & eq(s,t))

41        r(S:g, g) = id
42        r(C:g:s, g) = id
43        r(R:g:l, g) = id
44        r(F:g:l→s, g) = id
45        r(V:g:s, g) = id
46        r(S:g, h) = r(S:h, g)
47        r(C:g:s, h) = r(C:h:s, g)
48        r(R:g:l, h) = r(R:h:l, g)
49        r(F:g:l→s, h) = r(F:h:l→s, g)
50        r(V:g:s, h) = r(V:h:s, g)
51        id·φ = φ                          for φ ∈ {s,c,f,q,v}
52        r(S:g, h)·(S:k) = S:σ(g,h,k)
53        r(S:g, h)·(C:k:(S:p)) = C:k:(S:σ(g,h,p))
54        r(S:g, h)·(F:k: (*S^:e) → S:p) = F:k: (*S^:σ(g,h,e)) → S:σ(g,h,p)
55        r(S:g, h)·(R:k: (*S^:e)) = R:k: (*S^:σ(g,h,e))
56        r(S:g, h)·(V:k:(S:p)) = V:k:(S:σ(g,h,p))
57        r(C:g:s, h)·(C:k:s) = C:σ(g,h,k):s
58        eq(s,t) = F  ⇒  r(C:g:s, h)·(C:k:t) = C:k:t
59        r(F:g:l→s, h)·(F:k:l→s) = F:σ(g,h,k):l→s
60        eq(l:s, t:j) = F  ⇒  r(F:g:l→s, h)·F:k:j→t) = F:k:j→t
61        r(R:g:l, h)·(R:k:l) = R:σ(g,h,k):l
62        eq(l,j) = F  ⇒  r(R:g:l, h)·(R:k:j) = R:k:j
63        r(V:g:s, h)·(V:k:s) = V:σ(g,h,k):s
64        eq(s,t) = F  ⇒  r(V:g:s, h)·(V:k:t) = V:k:t
65        r(φ,n)·ψ = ψ                      for φ ∈ {c,f,q,v}, ψ ∈ {s,c,f,q,v}, φ≠ψ
    end equations
end ELEMENTS
```

Next, we will describe signatures. First, we list the module ATOMICSIGNATURES, in which we embed the elements of the previous module. In turn, we embed the module ATOMICSIGNATURES into SIGNATURES. A picture of their signature is fig. 2, between the two modules.

module ATOMICSIGNATURES
begin
  begin signature
    ATSIG — sort of atomic signatures
    $i_{SD\to ATSIG}$: SD → ATSIG — embedding of sort declaration as atomic signature
    $i_{CD\to ATSIG}$: CD → ATSIG — embedding of constant declaration as at. sig.
    $i_{RD\to ATSIG}$: RD → ATSIG — embedding of relation declaration as at. sig.
    $i_{FD\to ATSIG}$: FD → ATSIG — embedding of function declaration as at. sig.
    $i_{VD\to ATSIG}$: VD → ATSIG — embedding of variable declaration as at. sig.
    $eq_{ATSIG\times ATSIG\to BOOL}$: ATSIG × ATSIG → BOOL — equality on atomic signatures
    $\_\cdot\_{}_{ATREN\times ATSIG\to ATSIG}$: ATREN × ATSIG → ATSIG — application of atomic renaming
  end signature

  begin equations
  variables s,t ∈ SD, c,d ∈ CD, r,q ∈ RD, f,g ∈ FD, v,w ∈ VD, n ∈ GN
  meta-variables φ,ψ
66 eq(i(s), i(t)) = eq(s,t)
67 eq(i(c), i(d)) = eq(c,d)
68 eq(i(r), i(q)) = eq(r,q)
69 eq(i(f), i(g)) = eq(f,g)
70 eq(i(v), i(w)) = eq(v,w)
71 eq(i(φ), i(ψ)) = F — for φ,ψ ∈ {s,c,q,f,v}, φ≠ψ
72 r·i(φ) = i(r·φ) — for φ ∈ {s,c,f,q,v}
  end equations
end module ATOMICSIGNATURES

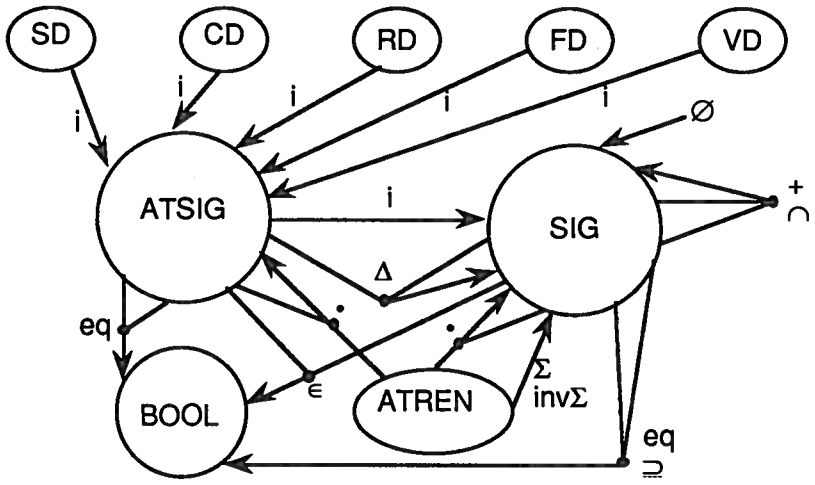

FIGURE 2. Signature of ATOMICSIGNATURES, SIGNATURES.

module SIGNATURES
begin

begin signature

SIG	sort of signatures
$i_{ATSIG \to SIG}$: ATSIG → SIG	conversion of atomic signatures into signatures
$\_+\_{}_{SIG \times SIG \to SIG}$: SIG × SIG → SIG	signature combination
$\varnothing_{SIG}$: SIG	empty signature
$eq_{SIG \times SIG \to BOOL}$: SIG × SIG → BOOL	equality on signatures
$\_\cap\_$: SIG × SIG → SIG	intersection
$\_\in\_{}_{ATSIG \times SIG \to BOOL}$: ATSIG × SIG → BOOL	element of a signature
$\_\Delta\_$: ATSIG × SIG → SIG	deletion of an element of a signature
$\_\supseteq\_$: SIG × SIG → BOOL	signature inclusion
$\_\cdot\_{}_{ATREN \times SIG \to SIG}$: ATREN × SIG → SIG	application of atomic renaming
$\Sigma_{ATREN \to SIG}$: ATREN → SIG	signature of an atomic renaming
invΣ: ATREN → SIG	sorts mentioned in, but invariant under renaming

end signature

begin equations

variables x,y,z ∈ SIG, n ∈ GN, s,t ∈ SD, l: L(SD), c,d ∈ CD, r,q ∈ RD, f,g ∈ FD, v,w ∈ VD, a ∈ ATSIG

meta-variables φ,ψ

73	x + ∅ = x	
74	x + x = x	
75	x + y = y + x	
76	(x + y) + z = x + (y + z)	
77	i(i(C:n:s)) = i(i(C:n:s)) + i(i(s))	
78	i(i(R:n:l)) = i(i(R:n:l)) + /(+^, *(i^∘i^)(l), ∅)	
79	i(i(F:n:l→s))) = i(i(F:n:l→s)) + /(+^, *(i^∘i^)(s:l), ∅)	
80	i(i(V:n:s)) = i(i(V:n:s)) + i(i(s))	
81	a ∈ ∅ = F	
82	i(s) ∈ i(i(t)) = eq(s,t)	
83	i(s) ∈ i(i(C:n:t)) = eq(s,t)	
84	i(s) ∈ i(i(R:n:l)) = s ∈ l	
85	i(s) ∈ i(i(F:n:l→t)) = s ∈ t:l	
86	i(s) ∈ i(i(V:n:t)) = eq(s,t)	
87	i(φ) ∈ i(i(ψ)) = eq(i(φ), i(ψ))	for φ ∈ {c,r,f,v}, ψ ∈ {s,c,r,f,v}
88	a ∈ (x+y) = a ∈ x ∨ a ∈ y	
89	∅ ∩ x = ∅	
90	x ∩ x = x	
91	x ∩ y = y ∩ x	
92	(x ∩ y) ∩ z = x ∩ (y ∩ z)	
93	i(s) ∈ x = F ⇒ i(i(s)) ∩ x = ∅	
94	i(C:n:s) ∈ x = F ⇒ i(i(C:n:s)) ∩ x = i(i(s)) ∩ x	

95 $i(R:n:l) \in x = F \Rightarrow i(i(R:n:l)) \cap x = /(+^, *(i^ \circ i^)(l), \emptyset) \cap x$
96 $i(F:n:l \to s) \in x = F \Rightarrow i(i(F:n:l \to s)) \cap x = /(+^, *(i^ \circ i^)(s:l), \emptyset) \cap x$
97 $i(V:n:s) \in x = F \Rightarrow i(i(V:n:s)) \cap x = i(i(s)) \cap x$
98 $a \in x = T \Rightarrow i(a) \cap x = i(a)$
99 $(x + y) \cap z = (x \cap z) + (y \cap z)$
100 $a \in x = F \Rightarrow a \Delta x = x$
101 $a \Delta i(a) = \emptyset$
102 $i(s) \Delta i(i(C:n:s)) = \emptyset$
103 $s \in l = T \Rightarrow i(s) \Delta i(i(R:n:l)) = i(s) \Delta /(+^, *(i^ \circ i^)(l), \emptyset)$
104 $s \in t:l = T \Rightarrow i(s) \Delta i(i(F:n:l \to t)) = i(s) \Delta /(+^, *(i^ \circ i^)(t:l), \emptyset)$
105 $i(s) \Delta i(i(V:n:s)) = \emptyset$
106 $a \Delta (x + y) = a \Delta x + a \Delta y$
107 $x = y + z \Rightarrow x \sqsupseteq y = T$
108 $a \in y = T \ \& \ a \in x = F \Rightarrow x \sqsupseteq y = F$
109 $eq(x,y) = x \sqsupseteq y \ \& \ y \sqsupseteq x$
110 $r \cdot \emptyset = \emptyset$
111 $r \cdot i(a) = i(r \cdot a)$
112 $r \cdot (x + y) = r \cdot x + r \cdot y$
113 $\Sigma(r(S:n,n)) = \emptyset$
114 $eq(n,m) = F \Rightarrow \Sigma(r(S:n,m)) = i(S:n) + i(S:m)$
115 $eq(n,m) = F \Rightarrow \Sigma(r(C:n:s, m)) = i(C:n:s) + i(C:m:s)$
116 $eq(n,m) = F \Rightarrow \Sigma(r(F:n:l \to s, m)) = i(F:n:l \to s) + i(F:m:l \to s)$
117 $eq(n,m) = F \Rightarrow \Sigma(r(R:n:l, m)) = i(R:n:l) + i(R:m:l)$
118 $eq(n,m) = F \Rightarrow \Sigma(r(V:n:s, m)) = i(V:n:s) + i(V:m:s)$
119 $inv\Sigma(r(s,n)) = \emptyset$
120 $eq(n,m) = F \Rightarrow inv\Sigma(r(C:n:s, m)) = i(i(s))$
121 $eq(n,m) = F \Rightarrow inv\Sigma(r(F:n:l,s, m)) = i(i(s)) + /(+^, *(i^ \circ i^)(l), \emptyset)$
122 $eq(n,m) = F \Rightarrow inv\Sigma(r(R:n:l, m)) = /(+^, *(i^ \circ i^)(l), \emptyset)$
123 $eq(n,m) = F \Rightarrow inv\Sigma(r(V:n:s, m)) = i(i(s))$

end equations
end module SIGNATURES

The equations for SIGNATURES express the fact that combination of signatures behaves like set union with $\emptyset$ as the empty set. Moreover there are axioms that imply that signatures are closed i.e. that sorts occurring in the arity of a function or relation also occur in the signature (77 - 80). The axioms here are modeled after the set in [BHK 88].

In the next module, we define expressions over a certain signature.

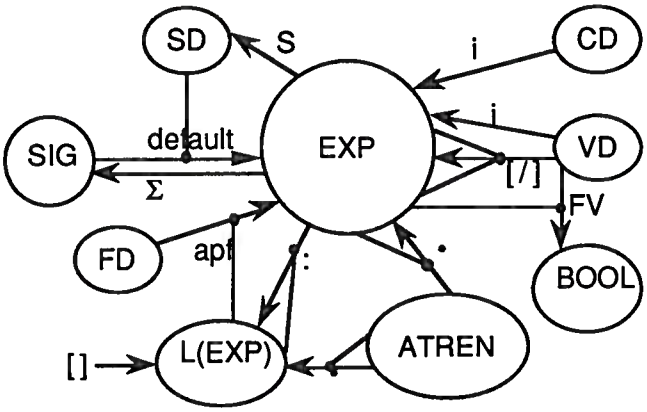

FIGURE 3. Signature of EXPRESSIONS.

```
module EXPRESSIONS
begin
  begin signature
    EXP                                  sort of expressions
    default: SIG × SD →EXP               default value of an expression
                                         (wrongly typed expression with base signature and sort)
    i_VD→EXP: VD → EXP                   conversion from variable to expression
    i_CD→EXP: CD → EXP                   conversion of constants to expressions
    apf: FD × L(EXP) → EXP               application of a function on an expression list
    Σ_EXP→SIG: EXP → SIG                 signature of expression
    S: EXP → SD                          sort of an expression
    [_/_]_VD×EXP→F(EXP,EXP): VD × EXP → F(EXP,EXP)
                                         substitution of an expression for a variable
    FV_VD×EXP→BOOL: VD × EXP → BOOL      variable occurs in expression
    _·_ATREN×EXP→EXP: ATREN × EXP → EXP  application of atomic renaming
    _·_ATREN×L(EXP)→L(EXP): ATREN × L(EXP) → L(EXP)   application of atomic renaming
  end signature

  begin equations
  variables n∈ GN, s,t∈ SD, f∈ FD, x∈ SIG, e∈ EXP, k∈ L(EXP), l∈ L(SD), v,w∈ VD, c∈ CD
124   S(i(V:n:s)) = s
125   S(i(C:n:s)) = s
126   S(default(x,s)) = s
127   eq(l, *S^(k)) = F ⇒ apf(F:n:l→s, k) = default(i(i(F:n:l→s)) + /(+^, *Σ^(k), ∅), s)
128   S(apf(F:n:l→s, k)) = s
129   Σ(default(x,s)) = x
130   Σ(i(v)) = i(i(v))
131   Σ(i(c)) = i(i(c))
132   Σ(apf(f, k)) = i(i(f)) + /(+^, *Σ^(k), ∅)
133   i(v)∈ x = T  ⇒  [v/e]⟪default(x,s)⟫ = default((i(v) Δ x) + Σ(e), s)
```

```
134         i(v)∈x = F  ⇒  [v/e]⟪default(x,s)⟫ = default(x,s)
135         eq(v,w) = T  ⇒  [v/e]⟪i(w)⟫ = e
136         eq(v,w) = F  ⇒  [v/e]⟪i(w)⟫ = i(w)
137         [v/e]⟪i(c)⟫ = i(c)
138         [v/e]⟪apf(f, k)⟫ = apf(f, *[v/e](k))
139         FV(v, e) = i(v) ∈ Σ(e)
140         r·default(x,s) = default(r·x, r·s)
141         r·i(v) = i(r·v)
142         r·i(c) = i(r·c)
143         r·[ ] = [ ]
144         r·(e:k) = (r·e):(r·k)
145         r·apf(f, k) = apf(r·f, r·k)
    end equations
end module EXPRESSIONS
```

The axioms for expressions must define the visible signature of all expressions, taking into account the rule that the signature of an expression is just the collection of all constants, variables, relations and function symbols that occur in it. An incorrectly typed expression should be equated with the default expression of the corresponding signature and sort (127). One needs that for all closed expressions of type EXPRESSION the visible signature can be calculated.

Next, we look at formulas over these expressions.

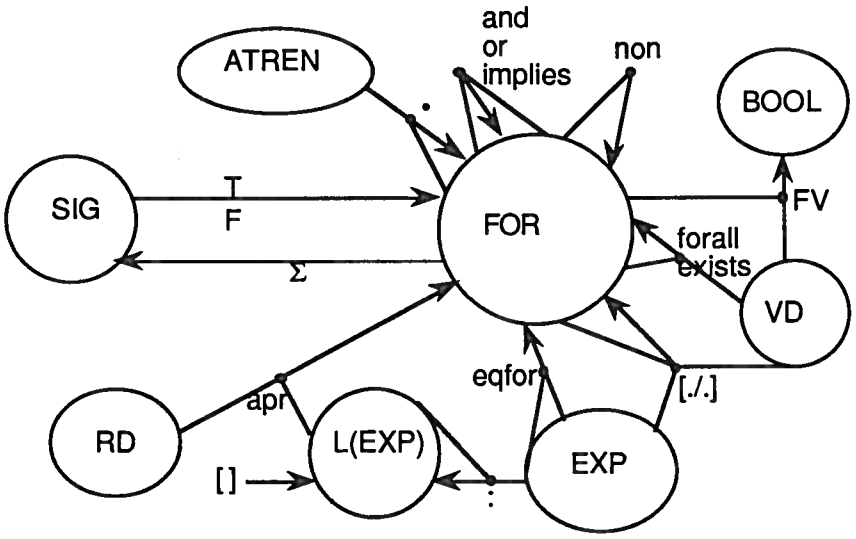

FIGURE 4. Signature of FORMULAS.

```
module FORMULAS
begin
    begin signature
        FOR                          sort of formulas
```

$T_{SIG \to FOR}$: SIG → FOR — constant formula true with signature
$F_{SIG \to FOR}$: SIG → FOR — constant formula false with signature
apr: RD × L(EXP) → FOR — application of a relation
eqfor: EXP × EXP → FOR — atomic formula equating two terms
forall: VD × FOR → FOR — universal quantification
exists: VD × FOR → FOR — existential quantification
and: FOR × FOR → FOR — conjunction
_or_: FOR × FOR → FOR — disjunction
non: FOR → FOR — negation
implies: FOR × FOR → FOR — implication
$\Sigma_{FOR \to SIG}$: FOR → SIG — signature of formula
$[\_/\_]_{VD \times EXP \to F(FOR,FOR)}$: VD × EXP → F(FOR,FOR) — substitution of an expression for a variable
$FV_{VD \times FOR \to BOOL}$: VD × FOR → BOOL — variable is free in formula
$\_\cdot\_{}_{ATREN \times FOR \to FOR}$: ATREN × FOR → FOR — application of atomic renaming

end signature

begin equations
variables x ∈ SIG, k ∈ L(EXP), s ∈ SD, r ∈ RD, l ∈ L(SD), n ∈ GN, p,q,e ∈ EXP,
f,g,h ∈ FOR, v,w ∈ VD

146 eq(l, *S^(k)) = F ⇒ apr(R:n:l, k) = F(i(i(R:n:l)) + /(+^, *Σ^(k), ∅))
147 Σ(T(x)) = x
148 Σ(F(x)) = x
149 Σ(apr(r, k)) = i(i(r)) + /(+^, *Σ^(k), ∅)
150 Σ(eqfor(p,q)) = Σ(p) + Σ(q)
151 Σ(forall(v, f)) = i(i(v)) + Σ(f)
152 Σ(exists(v, f)) = i(i((v)) + Σ(f)
153 Σ(f and g) = Σ(f) + Σ(g)
154 Σ(f or g) = Σ(f) + Σ(g)
155 Σ(non(f)) = Σ(f)
156 Σ(implies(f,g)) = Σ(f) + Σ(g)
157 non(f) or f = T(Σ(f))
158 non(f) and f = F(Σ(f))
159 implies(f, g or f) = T(Σ(f) + Σ(g))
160 f or f = f
161 f or g = g or f
162 implies((f or g) and (non(f) or g), g) = T(Σ(f) + Σ(g))
163 f and g = non(non(f) or non(g))
164 implies(f,g) = non(f) or g
165 f and implies(f,g) = (f and implies(f,g)) and g
166 i(v) ∈ x = T ⇒ [v/e]⟪T(x)⟫ = T((i(v) Δ x) + Σ(e))

167 $i(v) \in x = F \Rightarrow [v/e]\langle\langle T(x)\rangle\rangle = T(x)$
168 $i(v) \in x = T \Rightarrow [v/e]\langle\langle F(x)\rangle\rangle = F((i(v) \Delta x) + \Sigma(e))$
169 $i(v) \in x = F \Rightarrow [v/e]\langle\langle F(x)\rangle\rangle = F(x)$
170 $[v/e]\langle\langle apr(r, k)\rangle\rangle = apr(r, *[v/e](k))$
171 $[v/e]\langle\langle eqfor(p,q)\rangle\rangle = eqfor([v/e]\langle\langle p\rangle\rangle, [v/e]\langle\langle q\rangle\rangle)$
172 $eq(v,w) = T \Rightarrow [v/e]\langle\langle forall(w, f)\rangle\rangle = forall(w,f)$
173 $eq(v,w) = F \,\&\, FV(w,e) = F \Rightarrow [v/e]\langle\langle forall(w, f)\rangle\rangle = forall(w, [v/e]\langle\langle f\rangle\rangle)$
174 $eq(v,w) = T \Rightarrow [v/e]\langle\langle exists(w, f)\rangle\rangle = exists(w,f)$
175 $eq(v,w) = F \,\&\, FV(w,e) = F \Rightarrow [v/e]\langle\langle exists(w, f)\rangle\rangle = exists(w, [v/e]\langle\langle f\rangle\rangle)$
176 $[v/e]\langle\langle f \text{ or } g\rangle\rangle = [v/e]\langle\langle f\rangle\rangle \text{ or } [v/e]\langle\langle g\rangle\rangle$
177 $[v/e]\langle\langle non(f)\rangle\rangle = non([v/e]\langle\langle f\rangle\rangle)$
178 $FV(v, T(x)) = i(v) \in x$
179 $FV(v, F(x)) = i(v) \in x$
180 $FV(v, apr(r, [\,])) = F$
181 $FV(v, apr(r, e{:}k)) = FV(v, e) \vee FV(v, apr(r, k))$
182 $FV(v, eqfor(f,g)) = FV(v,f) \vee FV(v,g)$
183 $eq(v,w) = T \Rightarrow FV(v, forall(w, f)) = F$
184 $eq(v,w) = F \Rightarrow FV(v, forall(w, f)) = FV(v, f)$
185 $eq(v,w) = T \Rightarrow FV(v, exists(w, f)) = F$
186 $eq(v,w) = F \Rightarrow FV(v, exists(w, f)) = FV(v, f)$
187 $FV(v, f \text{ and } g) = FV(v,f) \vee FV(v,g)$
188 $FV(v, f \text{ or } g) = FV(v,f) \vee FV(v,g)$
189 $FV(v, non(f)) = FV(v,f)$
190 $FV(v, implies(f,g)) = FV(v,f) \vee FV(v,g)$
191 $FV(w, f) = F \,\&\, eq(S(i(v)),S(i(w))) \Rightarrow forall(v, f) = forall(w, [v/i(w)]f)$
192 $FV(w, f) = F \,\&\, eq(S(i(v)),S(i(w))) \Rightarrow exists(v, f) = exists(w, [v/i(w)]f)$
193 $eqfor(f,f) = T(\Sigma(f))$
194 $implies(eqfor(f,g) \text{ and } [v/p]f, [v/p]g) = T((i(i(v)) + \Sigma(f)) + (\Sigma(g) + \Sigma(p)))$
195 $forall(v, f) = non(exists(v, non(f))$
196 $FV(v, e) = F \Rightarrow implies([v/e]f, exists(v, f)) = T(\Sigma(e) + (i(i(v)) + \Sigma(f)))$
197 $FV(v, g) = F \Rightarrow implies(implies(f,g), implies(exists(v, f), g)) = T((\Sigma(f) + \Sigma(g)) + i(i(v)))$
198 $r \cdot T(x) = T(r \cdot x)$
199 $r \cdot F(x) = F(r \cdot x)$
200 $r \cdot apr(q, k) = apr(r \cdot q, r \cdot k)$
201 $r \cdot eqfor(f,g) = eqfor(r \cdot f, r \cdot g)$
202 $r \cdot (f \text{ or } g) = r \cdot f \text{ or } r \cdot g$
203 $r \cdot (non(f)) = non(r \cdot f)$
204 $r \cdot (exists(v, f)) = exists(r \cdot v, r \cdot f)$

end equations
end module FORMULAS

The situation with formulas is comparable with that of expressions, be it that the role of the default formula is now played by F(x) for the right signature (146). Moreover, the visible signature must be defined for all formulas. Then we need axioms that allow α-conversion of variables bound by existential and universal quantification (191, 192). Further, all axioms of predicate logic can be coded in the format of module algebra by writing them as equivalences between conjunctions of axioms (157 - 165, 193 - 197). In the following section on modules, one will find an axiom that allows us to split an atomic module consisting of the conjunction of two formulas, in a combination of atomic modules (205). Thus it will be possible to remove the long conjunctions that are generated by the coding of predicate logic inference rules in this section.

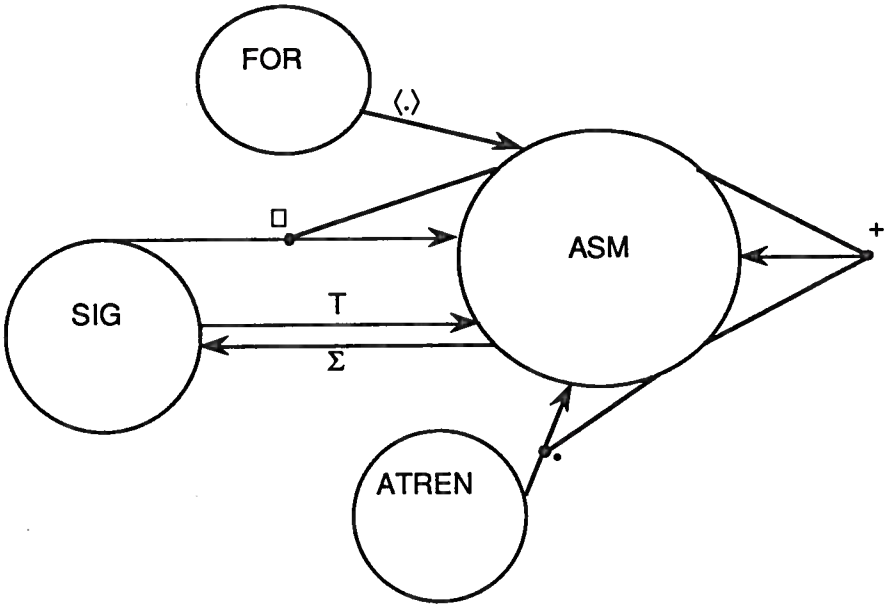

FIGURE 5. Signature of MODULES.

module MODULES
begin
  begin signature
    ASM — sort of algebraic specification modules
    ⟨_⟩: FOR → ASM — atomic module
    $T_{SIG \to ASM}$: SIG → ASM — embedding of signatures in ASM
    $\_+\_{ASM \times ASM \to ASM}$: ASM × ASM → ASM — combination of modules
    □: SIG × ASM → ASM — export operator
    $\Sigma_{ASM \to SIG}$: ASM → SIG — visible signature
    $\_\dot{}\_{ATREN \times ASM \to ASM}$: ATREN × ASM → ASM — application of atomic renaming
  end signature

  begin equations
  variables f,g ∈ FOR, u,v ∈ SIG, X,Y,Z ∈ ASM
205 ⟨f and g⟩ = ⟨f⟩ + ⟨g⟩

206 $\Sigma(\langle f\rangle) = \Sigma(f)$
207 $\Sigma(T(u)) = u$
208 $\Sigma(X + Y) = \Sigma(X) + \Sigma(Y)$
209 $\Sigma(u \triangleleft X) = u \cap \Sigma(X)$
210 $/+\hat{}([\,]) = T(\emptyset)$
211 $X + Y = Y + X$
212 $(X + Y) + Z = X + (Y + Z)$
213 $T(u + v) = T(u) + T(v)$
214 $X + T(\Sigma(X)) = X$
215 $X + (u \triangleleft X) = X$
216 $\Sigma(X) \triangleleft X = X$
217 $u \triangleleft (v \triangleleft X) = (u \cap v) \triangleleft X$
218 $u \triangleleft (T(v) + X) = T(u \cap v) + (u \triangleleft X)$
219 $u \supseteq \Sigma(X) \cap \Sigma(Y) \Rightarrow u \triangleleft (X + Y) = (u \triangleleft X) + (u \triangleleft Y)$
220 $\Sigma(r \cdot X) = r \cdot \Sigma(X)$
221 $r \cdot \langle g\rangle = \langle r \cdot g\rangle$
222 $r \cdot T(u) = T(r \cdot u)$
223 $r \cdot (X + Y) = r \cdot X + r \cdot Y$
224 $r \cdot (u \triangleleft X) = (r \cdot u) \triangleleft (r \cdot X)$
225 $r \cdot (r \cdot X) = X$
226 $\Sigma(r) \cap \Sigma(X) = inv\Sigma(r) \Rightarrow r \cdot X = X$

end equations
end module MODULES

These equations for algebraic specification modules require just the axioms of module algebra for the part of the signature written above (206 - 225).

Next, we look at declarations. When we combine declarations with algebraic specification modules, we extend the signature of such modules, and we will have to re-examine some axioms. A picture for the signature of the modules DECLARATIONS and ENVIRONMENTS can be found between the two.

module DECLARATIONS
begin
begin signature

DE	sort of declaration environments
$\langle asm:\_ = \_\rangle: GN \times ASM \to DE$	asm declaration
$\langle sig:\_ = \_\rangle: GN \times SIG \to DE$	signature declaration
$\langle bool:\_ = \_\rangle: GN \times BOOL \to DE$	boolean declaration
$imp_{GN \to SIG}: GN \to SIG$	import signature expression
$imp_{GN \to ASM}: GN \to ASM$	import module expression
$\_+\__{DE \times DE \to DE}: DE \times DE \to DE$	combination of declaration environments

$\varnothing_{DE}$: DE empty declaration environment
_·_ATREN×DE→DE: ATREN × DE → DE application of atomic renaming
_·_ATREN×BOOL→BOOL: ATREN × BOOL → BOOL application of atomic renaming
end signature

begin equations
variables p,q,q' ∈ DE, X ∈ ASM, u ∈ SIG, b ∈ BOOL, n ∈ GN
227 p + ∅ = p
228 p + q = q + p
229 (p + q) + q' = p + (q + q')
230 p + p = p
231 r·⟨asm: n=X⟩ = ⟨asm: n=r·X⟩
232 r·⟨sig: n=u⟩ = ⟨sig: n=r·x⟩
233 r·⟨bool: n=b⟩ = ⟨bool: n=r·b⟩
234 r·(D + E) = r·D + r·E
235 r·∅=∅
236 r·T = T
237 r·F = F
end equations
end module DECLARATIONS

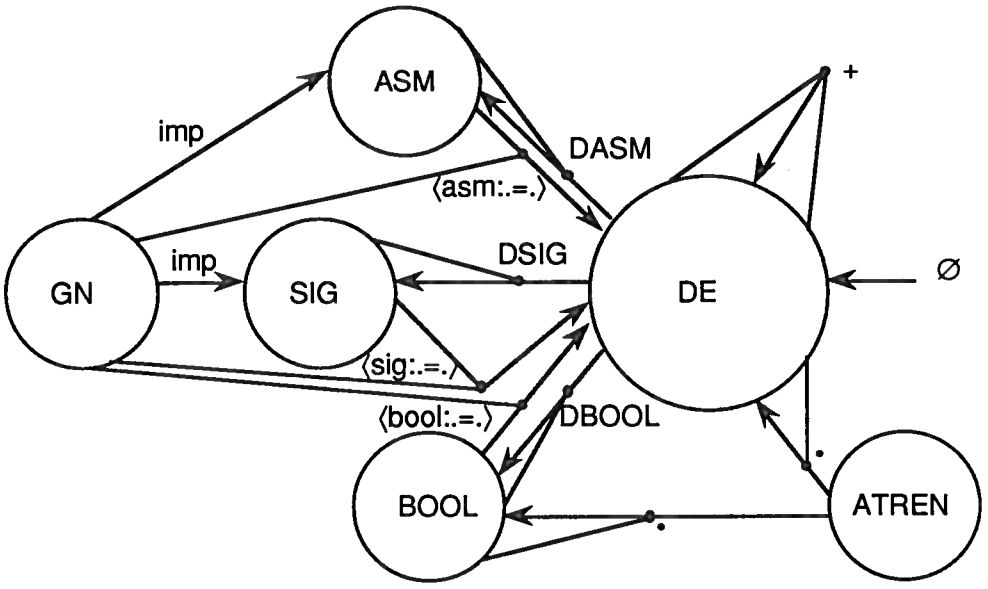

FIGURE 6. Signature of DECLARATIONS, ENVIRONMENTS.

The main equation for the abbreviation mechanism is a body replacement axiom that allows to replace a name with the module that it stands for (260, 261). Renamings have to be specified as well for the abbreviation declaration. Notice that since we allow declarations with module expressions, we also will get them with each target sort of ASM. In this case, this is only SIG, which in turn has BOOL as a target sort. The axioms are phrased so that the construct DASM (and DSIG, DBOOL) can be moved

to the outside, so that each module expression can be written in the form DASM(p, X), where X contains no declarations.

module ENVIRONMENTS
begin
  begin signature
    DASM: DE × ASM→ ASM asmodule with declarations
    DSIG: DE × SIG → SIG signature with declarations
    DBOOL: DE × BOOL → BOOL boolean with declarations
  end signature

  begin equations
  variables X,Y ∈ ASM, u,v,x ∈ SIG, b,c ∈ BOOL, p,q ∈ DE, s ∈ SD, v ∈ VD, n ∈ GN,
            l ∈ L(SD), a ∈ ATSIG, e ∈ EXP
238 X = DASM(∅, X)
239 u = DSIG(∅, u)
240 b = DBOOL(∅, b)
241 DASM(p, DASM(q, X)) = DASM(p + q, X)
242 DSIG(p, DSIG(q, u)) = DSIG(p + q, u)
243 DBOOL(p, DBOOL(q, b)) = DBOOL(p + q, b)
244 DASM(p, X) + DASM(q, Y) = DASM(p + q, X + Y)
245 DSIG(p, u) + DSIG(q, v) = DSIG(p + q, u + v)
246 DSIG(p, u) ∩ DSIG(q, v) = DSIG(p + q, u ∩ v)
247 DBOOL(p, b) & DBOOL(q, c) = DBOOL(p + q, b&c)
248 DBOOL(p, b) ∨ DBOOL(q, c) = DBOOL(p + q, b∨c)
249 ¬DBOOL(p, b) = DBOOL(p, ¬b)
250 Σ(DASM(p, X)) = DSIG(p, Σ(X))
251 T(DSIG(p, u)) = DASM(p, T(u))
252 eq(DSIG(p, u), DSIG(q, v)) = DBOOL(p + q, eq(u,v))
253 a ∈ DSIG(p, u) = DBOOL(p, a∈u)
254 a Δ DSIG(p, u) = DSIG(p, aΔu)
255 DSIG(p, u) ⊇ DSIG(q, v) = DBOOL(p + q, u⊇v)
256 DSIG(p, u) □ DASM(q, X) = DASM(p + q, u □ X)
257 ⟨asm: n = DASM(p,X)⟩ = p + ⟨asm: n = X⟩
258 ⟨sig: n = DSIG(p,u)⟩ = p + ⟨sig: n = u⟩
259 ⟨bool: n = DBOOL(p,b)⟩ = p + ⟨bool: n = b⟩
260 r·DASM(D, X)) = DASM(r·D, r·X)
261 r·(DSIG(D, x)) = DSIG(r·D, r·x)
262 r·(DBOOL(D, b)) = DSIG(r·D, r·b)

263 DASM(⟨asm: n = Y⟩, imp(n)) = DASM(⟨asm: n = Y⟩, Y)

264 DSIG(⟨sig: n = u⟩, imp(n)) = DSIG(⟨sig: n = u⟩, u)

265 $i(s) \in x = DBOOL(p, F) \Rightarrow i(i(s)) \cap x = \varnothing$

266 $i(C{:}n{:}s) \in x = DBOOL(p, F) \Rightarrow i(i(C{:}n{:}s)) \cap x = i(i(s)) \cap x$

267 $(R{:}n{:}l) \in x = DBOOL(p, F) \Rightarrow i(i(R{:}n{:}l)) \cap x = /(+^\wedge, {}^*(i^\wedge \circ i^\wedge)(l), \varnothing) \cap x$

268 $i(F{:}n{:}l \to s) \in x = DBOOL(p, F) \Rightarrow i(i(F{:}n{:}l \to s)) \cap x = /(+^\wedge, {}^*(i^\wedge \circ i^\wedge)(s{:}l), \varnothing) \cap x$

269 $i(V{:}n{:}s) \in x = DBOOL(p, F) \Rightarrow i(i(V{:}n{:}s)) \cap x = i(i(s)) \cap x$

270 $a \in x = DBOOL(p, T) \Rightarrow i(a) \cap x = i(a)$

271 $i(v) \in x = DBOOL(p, T) \Rightarrow [v/e](default(x,s)) = default((i(v) \Delta x) + \Sigma(e), s)$

272 $i(v) \in x = DBOOL(p, F) \Rightarrow [v/e](default(x,s)) = default(x,s)$

273 $i(v) \in x = DBOOL(p, T) \Rightarrow [v/e](T(x)) = T((i(v) \Delta x) + \Sigma(e))$

274 $i(v) \in x = DBOOL(p, F) \Rightarrow [v/e](T(x)) = T(x)$

275 $i(v) \in x = DBOOL(p, T) \Rightarrow [v/e](F(x)) = F((i(v) \Delta x) + \Sigma(e))$

276 $i(v) \in x = DBOOL(p, F) \Rightarrow [v/e](F(x)) = F(x)$

277 $DSIG(p, u) \supseteq \Sigma(X) \cap \Sigma(Y) \Rightarrow u \,\square\, (X + Y) = (u \,\square\, X) + (u \,\square\, Y)$

278 $\Sigma(r) \cap \Sigma(X) = DSIG(D, inv\Sigma(r) + v) \Rightarrow r \cdot X = X$

end equations

end module ENVIRONMENTS

7. MODELS OF BMASF

In this section, we give some heuristics on how to construct models for BMASF.

BMASF is a large specification and it is almost impossible to provide an elaborate account of its consistency and a survey of its various models in a reasonable and theoretically meaningful way. In particular, proving that it is a complete TRS or an attempt towards completion seems pointless to us. Semantically, one may say that the specification has an initial algebra and one needs some information about its structure and its homomorphic images.

The first step in the right direction is to view BMASF as an increasing sequence of specifications S1, S2, ..., Sk. Here, we discuss the following division:

S1 = modules ELEMENTS, ATOMICSIGNATURES and SIGNATURES;
S2 = S1 + module EXPRESSIONS;
S3 = S2 + module FORMULAS;
S4 = S3 + module MODULES;
S5 = S4 + module DECLARATIONS;
S6 = S5 + module ENVIRONMENTS.

7.1 SIGNATURES.

The initial algebra $\mathbb{I}(S1)$ is unproblematic, it can be described in terms of simple set theoretic constructions given the point of view that a signature is more or less a set of atomic signatures. Thus, the general names are to be interpreted by the natural numbers (one can choose to have character strings,

instead), for the various names one takes the appropriate kinds of pairs and tuples, and for the atomic signatures one takes the finite sets of the declarations of ingredients.

There is no reason to investigate any of its homomorphic images because of this, and one would be satisfied if $\mathbb{I}(S1)$ is the final algebra of S1 as well (which is probably true and otherwise can be made true by means of a harmless extension of the axioms).

7.2 EXPRESSIONS.

The initial algebra of S2 is an expansion of $\mathbb{I}(S1)$. To validate this, one must prove that every closed identity between terms over the signature of S1 provable from S2 is already provable from S1 (no confusion) and that every closed term over the signature S2 of a sort that occurs in the signature of S1 is provably equal to a closed term over the signature of S1 (no junk). One may prove the first fact in a model theoretic way by finding a model of S2 that is an expansion of the initial algebra of S1. The second fact requires inspection of the equations. By viewing them as rewrite rules one can prove with induction on the structure of closed expressions (of sort EXP) that S and Σ yield known values.

The model that proves 'no confusion' equates two expressions if they have the same signature and if in all algebras of that signature equipped with a default value for each sort, they compute the same polynomial functions with the understanding that minimal incorrectly typed expressions are replaced by the default corresponding to the target sort of their outermost function symbol. Notice that the defaults are never declared in the signature of an expression. These defaults are introduced in an external (meta) way. Thus, although the expressions seem to be over algebras of their signature, they must be interpreted in algebras with some additional structure, viz. the defaults for each sort.

As all expressions have a signature and not just a type, their interpretation in the models will take the form of a pair of a signature and a well-typed term over that signature (this term may involve the value default and may use just a part of the signature).

Thus, if $\mathbb{M}$ is a many-sorted algebra with an interpretation of the constant, relation and function symbols of our language, and if SIG is the class of signatures, then we are considering models of the form $\mathbb{M} \times SIG$, with as elements pairs of the form $\langle a,\sigma\rangle$ (a an element of a sort, σ a signature). Function application looks as follows:

$$f(\langle a_1,\sigma_1\rangle, ..., \langle a_n, \sigma_n\rangle) = \langle f(a_1, ..., a_n), \sigma_1 + ... + \sigma_n + f: t_1\times...\times t_n \rightarrow t\rangle$$

Notice that for all operations except substitution, the signature will increase.

7.3 FORMULAS

The initial algebra of S3 is an expansion of that of S2. This is proven in a similar way as in the case of S2. The model construction will equate two formulas if their signatures coincide and they have the same meaning in all algebras of that signature, where again we assume that these algebras have been equipped with default values for each sort and the application of a relation on a type incorrect parameter list leads to the default formula F for the appropriate signature.

The meaning of a formula is determined as in ordinary many sorted logic. One may or may not allow empty sorts, that will just lead to different variations of the theory. All sorts that have an expression in the language are non-empty, as the default element has to be identified.

7.4 MODULES

Models for S4 can be constructed as expansions of models for S3 similar to the construction of models for BMA in [BHK 88]. The objects of type FOR play the role of first order atomic modules and for each module a signature and a meaning is known. Models of module algebra that can be considered are $\mathbb{I}$(BMA[*fol*]), $\mathbb{M}$(*fol*), $\mathbb{M}_C$(*fol*), $\mathbb{T}$(*fol*) as defined in [BHK 88].

For S4 one can establish a normalization theorem that allows to rewrite every closed module expression into a form that features only a single occurrence of the export operator.

7.5 DECLARATIONS

Models of S5 are again expansions of selected natural models of S4. Given a model of S4, one may view a declaration environment as a system of definitions for signatures and modules. Two DE's can be identified if they define the same variety over the given models of S4. Stated differently, a DE defines a relation with attribute names in GN and with as domains for each attribute either the signatures or the modules of the chosen model. The relation contains exactly those pairs that satisfy the identities of the declarations in the chosen model of module algebra.

7.6 ENVIRONMENTS

The step towards S6 involves an extension of previous algebras rather than an expansion. Indeed, the transition to S6 introduces for three sorts (BOOL, SIG, ASM) new expressions involving declaration environments. It is claimed that models for S6 can be found by extending the appropriate models for S5 at these sorts with objects consisting of a pair of a declaration environment and an original expression of that sort.

Then one may consider the closed signature and module expressions involving declaration environments as mappings from the collections determined by their declarations to signatures resp. modules. This construction is entirely general in the sense that it works on a model of any theory in the place of module algebra.

7.7 AXIOMS

All axioms reflect valid assertions about the above interpretation of the language of BMASF. In fact BMASF is a small extension of the system BMA[*fol*] of [BHK 88]. It includes an explicit algebraic coding of first order logic. The only complication of that coding is the application of defaults to deal with incorrectly typed expressions. Moreover it includes a mechanism for the introduction of declarations and the expansion of declared names.

The axioms are to be judged on several criteria:

They must express facts about the operators that hold in the intended interpretation. They must allow all transformations that are involved in normalization procedures that allow rewriting expressions to the various normal forms that one may have in mind. Now it is hardly possible to survey the normal forms and normalization procedures that are relevant. Further as different tools may use different normalization mechanisms it is inappropriate to design the axioms on their ability to allow just one normalization or just a few key normalization mechanisms. The normalizations that are possible using the transformations described in our axioms include:

(1) the disjunctive and conjunctive normal form of propositional calculus;

(2) the prenex normal form of predicate logic;

(3) the normal form of module algebra for closed expressions without unexpanded imports;

(4) collecting all declarations in a declaration environment (where declared modules have no internal declarations);

(5) expanding all imports (for which a declaration is available) in a target expression;

(6) the flattening of module expressions to a normal form in the sense of module algebra; this may be done just partially because of inexpandable imports, otherwise we are in case (3);

(7) the decomposition of complex predicate calculus formulae into a sum of atomic module expressions using $\langle P \text{ and } Q \rangle = \langle P \rangle + \langle Q \rangle$; this is exactly what happens if one writes a long algebraic specification, though it may be inappropriate to call it normalization.

ACKNOWLEDGEMENT.

The authors thank Hans Mulder for many helpful comments.

REFERENCES.

[BHK 88] J.A. BERGSTRA, J. HEERING & P. KLINT, *Module algebra,* report CS-R8844, Centre for Math. & Comp. Sci., Amsterdam 1988 (revised version of report CS-R8617). To appear in JACM.

[BHK 89] J.A. BERGSTRA, J. HEERING & P. KLINT, *Algebraic specification,* ACM Press Frontier Series, Addison Wesley 1989.

[BR 89] J.A. BERGSTRA & G.R. RENARDEL DE LAVALETTE, *De plaats van formele specificaties in de software-technologie,* Informatie 31 (6), 1989.

[BI 87] R.S. BIRD, *An introduction to the theory of lists,* in: Logic of Programming and Calculi of discrete design (ed. M. Broy), Springer 1987, pp. 5-42.

[FJKR 87] L.M.G. FEIJS, H.B.M. JONKERS, C.P.J. KOYMANS & G.R. RENARDEL DE LAVALETTE, *Formal definition of the design language COLD-K,* METEOR/t7/PRLE/7, 1987.

[J 89] S.M.M. JOOSTEN, *The use of functional programming in software development,* Ph.D. thesis, Universiteit Twente, 1989.

[MV 88] S. MAUW & G.J. VELTINK, *A process specification formalism,* report P8814, Programming Research Group, University of Amsterdam 1988. To appear in Fund. Inf.

[M 86] L.G.L.T. MEERTENS, *Algorithmics – towards programming as a mathematical activity,* in: Math. & Comp. Sci. (eds. J.W. de Bakker e.a.), CWI Monograph 1, North Holland 1986, pp. 289-334.