The VTP project : modular abstract syntax specification

Didier Austry

UNITÉ DE RECHERCHE
NRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1219

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

THE VTP PROJECT : MODULAR ABSTRACT SYNTAX SPECIFICATION

Didier AUSTRY

Mai 1990

RR_1219

Le Projet VTP: Specification de Syntaxe Abstraite Modulaire

The VTP project: Modular Abstract Syntax Specification

Didier Austry

SEMA GROUP
c/o I.N.R.I.A. Sophia Antipolis
Route des Lucioles
06 560 Valbonne FRANCE
austry@mirsa.inria.fr

Résumé. Ce rapport étudie la spécification de syntaxes abstraites modulaires. Nous y décrivons une syntaxe abstraite comme un ensemble de modules paramétrés encapsulant des définitions d'opérateurs et de sortes ainsi qu'un certain nombre d'opérations de base. La plus remarquable de ces opérations est l'extension de sortes d'un module dans un autre. Cette proposition s'inspire principalement du langage de module de Standard ML. Ce rapport constitue un rapport intermédiaire du projet VTP.

Mots clés: Syntaxe abstraite, modularité, Standard ML, projet Centaur.

Financement: Ce projet est partiellement financé par le Ministère de la Recherche et de la technologie, projet n^o 88 S 698.

Abstract. This report examines the modular specification of abstract syntaxes. We describe an abstract syntax as a collection of parameterized modules encapsulating operator and sort definitions along with some basic operations on them. The sort extension from one module into another one is the most remarquable of these operations. This proposition takes much of its source from the Standard ML module language. This report is a part of a series of report on the VTP project.

Keywords: Abstract syntax, modularity, Standard ML, the Centaur project.

Grant: This project is partially supported by the french Minister of Research and Technology, under contract number 88 S 698.

The VTP Project: Modular Abstract Syntax Specification†

Didier Austry
SEMA GROUP, Sophia-Antipolis

† This project is partially supported by the french Minister of Research and Technology under contract 88 S 698

The VTP Project: Modular Abstract Syntax Specification

Didier Austry
Sema-Group, Sophia-Antipolis

Contents

1 Modular Abstract Syntax **3**

1.1 Presentation 3

1.2 Modularity: Problems and Proposals 3

1.2.1 The Problem 3

1.2.2 Proposals 4

1.2.3 Our Problem 5

1.3 The Core language 6

1.3.1 Sort And Operator Definitions 6

1.3.2 The Sort Inclusion And Renaming 7

1.4 The Module Language 8

1.4.1 The Formalism 8

1.4.2 Subformalisms 10

1.4.3 Parameterized Formalisms 10

1.4.3.1 Formalisms and Signatures 11

1.4.3.2 Formalism Instantiations 13

1.4.3.3 Constraints on Result Formalism 13

1.4.4 The Sharing 14

1.4.5 The Open and Hiding Instructions 15

1.5 Advanced Operations 16

1.5.1 The Sort Extension 16

1.6 Parameterized Formalisms as Arguments 18

2 The Static Semantics **20**

2.1 Presentation 20

2.2 The Formal Specification of Basic Structures 21

2.3 The Core language 22

2.3.1 The Sort Declaration 23

2.3.2 The Sort Inclusion 23

2.3.3 The Sort Renaming 24

2.3.4 The Sort Extension 24

2.3.5 The Operator Declaration 24

2.4 The Module Language 25

2.4.1 The Formalism Matching 25

A The Nogom Metal Abstract Syntax **26**

B The Simplest Nogom Specification **29**

C Nogom With a Block Instruction **33**

D Nogom With Subprogram Declarations **34**

E The Complete Nogom Specification **36**

Chapter 1

Modular Abstract Syntax

1.1 Presentation

This report is part of a series of reports on the specifications of the VTP project. The goal of this project is to undertake a new version of the current VTP [6].

The VTP primitives allow the creation and manipulation of *trees*. Very often, such trees are used to represent *fragments of abstract syntaxes*. It is proposed in this project, among other improvements, to study the *modularity* of such abstract syntax definitions.

This report presents the description of language constructs aiming at solving this problem. We do not claim any originality. Our problem is very fundamental but also very classical in the software engineering field. Thus, we have adapted already existing methods and tailored them to our own needs. The module facility of the STANDARD ML language [5] is our principal source of inspiration with some addings from the CLEAR specification language [1] and the ASF formalism [2].

1.2 Modularity: Problems and Proposals

1.2.1 The Problem

Modularity is the corner stone of current problems in software engineering. Solving it solves two related problems:

- *Organization*: cutting a program into manageable pieces.

- *Reusability*: having at one's disposal a library of predefined, generic, components.

These two problems induce another one:

- *Compositionality*: having at one's disposal language constructs aiming at describing connections between predefined components.

1.2.2 Proposals

Related works were studied in the domain of abstract data type specifications, such as the CLEAR specification language or more recently the ASF formalism. The CLEAR proposal [1] was one of the first to tackle the problem of compositionality. The constructs proposed were:

- The *module*, grouping sort definitions, operator definitions, and equations between operators.
- The *parameterization* of modules, allowing the definition of generic modules. The formal parameters are specified by other module specifications.
- Several *operations* between modules, such as:
  - `Enrich`, allowing the addition of sorts or operators in a predefined module.
  - `Hide`, allowing the use of only a part of a predefined module.
  - `Combine`, allowing the combination of two predefined modules.

Similar in spirit is the ASF proposal [2]. ASF is a formalism aiming at specifying *abstract data types*. The basic object is the *module* grouping, as in CLEAR, definitions of *sorts*, *operator declarations*, and *semantic equations*. ASF uses the notion of a *signature*, in its constructs: sorts and operator declarations make up the signature of a module.

ASF supports modular construction of specifications. The language constructs provided in this regard are the following:

- Sorts and functions can be *exported*. Only exported sorts and functions are part of the module signature.
- Modules can be *imported* by another one.
- Modules can be parameterized. Formal parameters are specified as a signature in the body of the parameterized module.

The differences between these two proposal are rather syntactic. Only the constructs provided at the module level are different but they clearly play the same role: *hiding* information or *reusing* information.

Soon after CLEAR, D. Mac Queen proposed a similar but, perhaps, more economical view of modules in the context of the STANDARD ML language [5, 4]. The basic concepts are:

- The *structure*, grouping together STANDARD ML declarations.
- The *functor*, acting as a function from structures to structures. The functor is the only constructor. No constructor on modules is provided.
- The *signature*, playing the role of a type of a structure.

  For example, signatures specify the constraints the functor's formal parameters must respect. In fact, this situation is completely general: a structure (or the body of a functor) can always be constrained to satisfy a certain signature. Thus, the signature provides the means of *hiding information* from a module. This use is called the *thinning* of a structure in the STANDARD ML terminology.

  The functor and signature concepts, used together, are powerful enough to express all kinds of constructions. For example, the `hiding` construct of CLEAR or the `export` construct of ASF are simulated by the thinning of a structure.

The strategic idea of the STANDARD ML module language is the distinction between the core language and the module language.

What can be done at the module language level is: defining basic units—grouping together entities of the core language with a common interface—and parameterizing of basic units—abstracting over part of basic units.

What cannot be done at the module language level is: defining modules recursively—only hierarchical composition of modules is allowed—and parameterizing entities from the core language. The module language only manipulates entities at the module level.

1.2.3 Our Problem

These concepts were the guiding ideas we followed in the design of our language aiming at describing *modular abstract syntax.*

Our problem is, in a sense, much simpler. We just specify abstract datatypes, without semantic equations as in CLEAR or ASF and without any associated values or functions as in STANDARD ML.

But, from the module point of view, the problem is the same. We need all the basic constructs of the STANDARD ML module language: definitions grouped

in *separated modules*, *parameterizations* of modules, and *constraints* on modules.

These constructs take the following form in our context:

- The *formalism* is the basic unit which groups together sort and operator definitions. A formalism specifies an abstract syntax and is really a *signature* in the STANDARD ML terminology.
- The formalisms are *parameterized* over formalism names. But in contrast to STANDARD ML, these parameters are constrained by *other formalisms*, as in CLEAR. This is a natural choice because formalisms are signatures as well.

  Thus formalisms play alternatively two roles:

  - the role of *basic values* of the module language,
  - the role of *constraints* over these basic values.

We now describe our proposal, piece by piece, taking care of distinguishing between the core language and the module language.

Note. Throughout the following, the NOGOM language will be our favorite example. NOGOM is a simple algorithmic language with procedure and block definitions but it is complex enough to illustrate our main constructs. The complete METAL description of the NOGOM abstract syntax can be found in the appendix.

1.3 The Core language

First, we describe the fundamental operations of the core language, *sort* and *operator* definitions. Second, we describe the different provided operations: the *sort inclusion* and the *sort renaming*.

1.3.1 Sort And Operator Definitions

The goal of our language is the specification of *abstract syntaxes*. An abstract syntax consists in:

- The definition of *sorts*, another name for type or CENTAUR Phylum.
- The definition of *operators*, with their sort arguments and the result sort.

  Three kinds of operators can be defined: atomic operators, fixed arity operators, and list operators.

Examples. Here are simple examples used in the construction of the abstract syntax of NOGOM.

- The integer and boolean sorts. These are atomic operators:

```
sort Bool
booltrue:  -> Bool
boolfalse: -> Bool

sort Int
int:  -> Int
```

- The sorts can be defined simultaneously. Take, for example, the following definitions of the `Ident` and `Ident_s` sorts:

```
sort Ident Ident_s
ident:                -> Ident
ident_s : {Ident}+    -> Ident_s
```

  Here, the `ident_s` operator is a list operator with a non-null number of arguments.

- The beginning of an `Exp` sort definition, for algebraic expressions, can look like the following one:

```
sort Exp Exp_s
exp_s:     {Exp}*      -> Exp_s
exp_and:   Exp # Exp   -> Exp
exp_not:   Exp         -> Exp
exp_plus:  Exp # Exp   -> Exp
exp_mult:  Exp # Exp   -> Exp
```

  Here, the `exp_s` operator is a list operator with a possibly empty list of arguments. The other operators are fixed arity operators.

Note. Sort are implicitely mutually recursively defined. See examples in section 1.4.4.

We now describe the operations provided in the core language: the *sort inclusion* and the *sort renaming.*

1.3.2 The Sort Inclusion And Renaming

An abstract syntax definition always asks for the definition of sorts with other sorts *as subsorts.* NOGOM provides several examples of such a case: the litterals of the language consist in the union of booleans and integers, these litterals

are, in turn, part of the `Exp` sort, the sort `Exp_Or_Empty` is just the `Exp` sort plus the `empty` operator, and so on...

Examples. These examples are written in the following manner:

- The `Litt` sort definition:

```
sort Litt
sort Bool < Litt
sort Int  < Litt
```

- The `Exp_Or_Empty` sort definition:

```
sort Exp_Or_Empty
sort Exp < Exp_Or_Empty
empty:   -> Exp_Or_Empty
```

From time to time, for clarity reasons, it is interesting to *rename* already defined sorts.

Example. Let the following definition be the embryo of the `Decl` sort declaration, without procedures and functions declarations.

```
sort Type_Name = Ident
sort Decl Decl_s
decl_s:   {Decl}*                    -> Decl_s
var_decl: Ident # Type_Name # Exp    -> Decl
```

Since we want to stress that the second argument of `var_decl` is a type name and not a simple identifier, we rename `Ident` to emphasize this use.

1.4 The Module Language

First, we describe the *formalism* which is the module unit of our language. Second, we describe the different provided operations: the *subformalism* construct and the *parameterization* of formalisms.

1.4.1 The Formalism

The *formalism* is the basic unit of the module language. A formalism just groups together sort and operator definitions.

The basic operations on formalisms are:

- The *declaration* of formalisms. A declaration associates a name with a computed formalism expression. We can refer to the content of a formalism, with a *qualified name* (see note below) as in STANDARD ML.
- The description of formalism *expressions*. They define the content of a formalism declaration. They are delineated by the keywords **form** and **end**. Formalism expressions are just *core language declarations* plus some language facilities for manipulating identifiers (see section 1.4.5).

Examples. Let us complete two of the previous examples:

- The formalism IDENT:

```
formalism IDENT =
   form
      sort Ident
      ident:                -> Ident
   end
```

- Sort declarations and operators declarations can be mixed as in:

```
formalism VALUE =
   form
     sort Bool
     booltrue:  -> Bool
     boolfalse: -> Bool

     sort Int
     int:  -> Int
   end
```

Note. Qualified Names

Once we have encapsulated sort definitions, we need a syntactic mean to access them *inside* a formalism definition. We use, for that, *qualified names*: sorts are accessed as field components of records, with a dot notation.

Example. `VALUE.Bool` refers to the sort `Bool` defined in the formalism `VALUE`.

These qualified names will be naturally generalyzed to *path names* when *subformalisms* will be introduced (see section 1.4.2).

We now describe the different provided operations: the *subformalism* construct and the *parameterization* of formalisms. Furthermore, we discuss several points related to the parameterization of formalisms: the formalisms as *constraints*, the *formalism instantiations*, the **hiding** construct, and the **sharing** construct.

1.4.2 Subformalisms

Take a look, for example, at the DECL declaration: `var_decl` uses the `Ident` and `Exp` sorts. We can perfectly use these sorts as global names, referring to formalism already defined. But a more structured way of looking at the DECL formalism is to emphasize the *dependence* on `Ident` and `Exp` and to *localize* these dependences, at the same time. This can take the form of *subformalism* declarations.

Example. We just give here a sketch of the DECL specification:

```
formalism DECL =
  form
    formalism EXP =
       form
          sort Exp
          ...
       end
    formalism IDENT =
        form
           sort Ident
           ...
        end

    sort Type_Name = IDENT.Ident
    ...
    var_decl: IDENT.Ident # Type_Name # EXP.Exp  -> Decl
  end
```

Note. Path names.

We can refer to the `Ident` or `Exp` sorts defined inside the subformalism of DECL in generalizing the qualified names to allow more complex names denoting the *complete path* to follow to access these objects.

Example. The path `DECL.EXP.Exp` refers to the sort `Exp` defined above inside DECL.

1.4.3 Parameterized Formalisms

Subformalism specifications is not enough as a modular tool. Written this way, specifications will become *larger and larger*, without *sharing* common subformalisms. And finally, we want a specification to be completely *independent* of any previously defined formalism. Hence, all that naturally appeals to the use

of *parameterization.*

A parameterized formalism looks like a traditional function. In fact, parameterized formalisms correspond roughly to *functors* in the STANDARD ML terminology (see below for an exact correspondence). We give:

- The list of the formalism parameters in parentheses.
- Their associated "types" which are just formalisms as well. More on that in the next section.
- The body of the formalism, which is a formalism expression that can use the formal parameters as subformalisms.

Before giving examples, we must precise the role of formalisms as signatures.

1.4.3.1 Formalisms and Signatures

We want to emphasize the fact that formalisms are also *signatures.* We declare a formalism to declare an *abstract syntax* and abstract syntaxes are signatures: by definition, a signature is the specification of types and associated operators. The fact that we manipulate it as a value must not obscure its real nature.

Thus at the same time we declare a formalism, we have declared a signature and we can use it as such. The signature plays here the role of a *constraint* put on formalism parameters, as in STANDARD ML.

Note. Comparison with the STANDARD ML module language.

In STANDARD ML, there is a clear distinction between basic values, the structures, and signatures. But structures are more complex objects, containing code for values or functions and signatures are pertinent to describe *what* is defined inside the structure, forgetting the *how.* Since in our case the values and the signatures are the same, we don't want to make the language heavy with superfluous syntactic differences.

Furthermore, in STANDARD ML, only structures are subject to parameterization in the form of functors. Signatures are simple objects which can be associated to structures and to functors. There are no parameterized signatures. In this regard, we have to precise the exact signature corresponding to a *parameterized formalism.* We follow the same path as in STANDARD ML: the signature associated with a functor contains the structure parameters as *substructures* (see [4] and the examples below).

Example. The parameterization of the `DECL` formalism.

The `DECL` formalism needs two parameters, the `EXP` and `IDENT` formalisms. We have only to know that the actual parameter for `EXP` contains the `Exp`

sort and the actual parameter for IDENT contains the Ident sort. Thus, the specification can looks as follows:

```
formalism EXPSIG =
   form
      sort Exp
   end

formalism IDENTSIG =
   form
      sort Ident
   end

formalism DECL_F (EXP:EXPSIG; IDENT:IDENTSIG) =
   form
     sort Type_Name = IDENT.Ident
     sort Decl_s Decl
     decl_s:   {Decl}*                              -> Decl_s
     var_decl: IDENT.Ident # Type_Name # EXP.Exp -> Decl
   end
```

Note. Notice the use of *qualified names* to access sorts inside the formal parameter.

We can simply write this specification in the following manner, with formalism constraints given immediately:

```
formalism DECL_F (EXP: form sort Exp end ;
                  IDENT: form sort Ident end) =
   form
       ...
   end
```

The form keyword introduces here an anonymous formalism expression.

As an alternative way of specifying the DECL_F formalism, assuming that we have already defined the EXP and IDENT formalisms, we use them as constraints in the DECL_F parameter specification. This choice simplifies the specification but restricts the possible instantiations of EXP and IDENT since they have to satisfy the complete EXP constraint.

```
formalism DECL_F2 (EXP:EXP; IDENT:IDENT) =
   form
```

```
   ...
end
```

The signature corresponding to a parameterized formalism is obtained by adding the parameter specifications as *subformalisms* of the body of the parameterized formalism.

Example. The DECL_F formalism has the implicit signature:

```
formalism EXP: EXPSIG
formalism IDENT: IDENTSIG
sort Type_Name ...
decl_s: ...
var_decl: ...
```

1.4.3.2 Formalism Instantiations

An instance of DECL_F2 is created in the following way:

```
formalism DECL = DECL_F2 (EXP, IDENT)
```

where EXP and IDENT are the two previously described formalisms.

We have to check that the actual parameters *match* the constraints for the formal parameters. In our context, this task is simple. We just have to verify a kind of inclusion between formalism.

In the previous example, we just have to check that the EXP formalism as an actual parameter satisfies the EXP formalism as a constraint.

1.4.3.3 Constraints on Result Formalism

We can also constrain the body of a parameterized formalism to match a given signature (as we give the type of a function result in a traditional language).

Example. Assume we want to only export the Decl and Decl_s sort of the DECL_F formalism. We can define a DECL_RES formalism as:

```
formalism DECL_RES =
   form
      sort Decl_s Decl
   end
```

The header of our DECL_F formalism can be now:

```
formalism DECL_F (EXP: EXPSIG; IDENT: IDENTSIG) : DECL_RES =
   form
      ...
   end
```

We can also type the result with an anonymous formalism as in the case of a parameter specification.

1.4.4 The Sharing

Trying to complete the NOGOM specification, we write a sketch of an INST_F formalism containing the specification of the NOGOM instructions without block and procedure call instructions:

```
formalism INST_F (EXP:EXPSIG; IDENT:IDENTSIG)  =
   form
     sort Tag_Name = IDENT.Ident
     sort Inst_s Inst Else_Part

     sort Inst_s <  Else_Part
     empty: -> Else_Part

     inst_s: {Inst}+                    -> Inst_s
     null:                              -> Inst
     tag: Tag_Name # Inst_s             -> Inst
     exit: IDENT.Ident                  -> Inst
     assign: IDENT.Ident # EXP.Exp      -> Inst
     if : EXP.Exp # Inst_s # Else_Part  -> Inst
   end
```

We now have to create the complete NOGOM formalism in declaring that a NOGOM program consists in a list of declarations and a list of instructions. We first define a PROGBODY_F formalism to specify the program body and create the PROG_F formalism with this formalism as parameter. This will allow to resuse later PROG_F.

```
formalism PROGBODY_F(DECL : DECL_F; INST : INST_F)  =
   form
     sort Progbody
     progbody : DECL.Decl_s # INST.Inst_s -> Progbody
   end
```

```
formalism PROG_F(PROGBODY: form sort Progbody end;IDENT:IDENT)=
   form
     sort Prog
     prog: IDENT.Ident # PROGBODY.Progbody  -> Prog
   end
```

The alert reader may have noticed that, in the PROGBODY_F specification, we implicitely use two occurences of the formalism EXP: one in DECL_F and one in INST_F, but nothing says that these two subformalism DECL_F.EXP and INST_F.EXP are *the same* EXP *formalism.* We must *state explicitely* that they are identical. This is the purpose of the **sharing** construct.

Syntactically, we add **sharing** instructions where we write constraint instructions, as in parameter specifications.

Example. The header of our PROG_F specification is now:

```
formalism PROGBODY_F(DECL:DECL_F; INST:INST_F
                     sharing DECL.EXP=INST.EXP) =
   form
     ...
   end
```

Note. Since we have the same problem for IDENT, we have to modify all our formalism declarations to state this sharing properties. The reader can find this corrected version of the NOGOM specification in the appendices.

1.4.5 The Open and Hiding Instructions

As typing repeatedly full names will become rapidly boring, STANDARD ML provides a means to abbreviate the access to internal entities: the **open** instruction allows the user to see objects inside a module without prefixing them.

Example. Let us write again a sketch of the DECL_F formalism:

```
formalism DECL_F (EXP:EXPSIG; IDENT:IDENTSIG) =
   form
     open EXP IDENT
     sort Type_Name = Ident
     ...
     var_decl: Ident # Type_Name # Exp -> Decl
   end
```

As a last facility, we have added an operation, the **hiding** instruction, which reminds of the **Hide** construct in CLEAR.

Assume we have already written a big specification and we want to use it in a different context where some parts are irrelevant. We do not want to rewrite the complete specification except for a few definitions. This is the role of the **hiding** instruction.

Example. To only export the **Decl** sort in the DECL_F and using the DECL_RES declaration, we write the following specification:

```
formalism DECL_F (EXP: EXPSIG; IDENT: IDENTSIG) :
        DECL_RES hiding sort Decl_s =
  form
    ...
  end
```

1.5 Advanced Operations

We now describe more complex operations found either in the core language, the *sort extension* or in the module language, the use of *parameterized formalisms as arguments.*

1.5.1 The Sort Extension

To complete the NOGOM specification, we must add a **block** instruction and procedures declarations:

- We easily add a **block** instruction to the INST_F formalism. This instruction has the following specification:

  ```
  block : Decl_s # Inst_s  -> Inst
  ```

  Thus we add a DECL parameter to the INST_F specification. The reader can find this extension, the INST_F2 formalism, and the corresponding PROG_F2, in the appendices.

- We easily add procedure or function declarations in modifying the DECL_F specification. The heart of this modification rests on the specification of the body of such declarations:

  ```
  sort Progbody
  progbody : Decl_s # Inst_s      -> Progbody
  progdecl : Progspec # Progbody -> Decl
  ```

where `Progspec` speficies the procedure and function header.

Such declarations need an `Inst_s` argument, thus we add an `INST_F` parameter to the `DECL_F` specification. The reader can find a variant of this extension, the `PROGBODY_F3` specification, and the corresponding `PROG_F3`, in the appendices.

But now, we get the following problem: how can we combine the last two constructions in one shot, in face of the mutual dependencies of `INST_F` and `DECL_F`?

These dependencies cannot be simulated at the module level: there is neither *mutually recursive definition* of modules nor *sorts parameterization* (which would solve simply our problem, as the reader can check).

And we want to stress the difficulty:

- We do not want to specify `Inst` and `Decl`, at the same level, in the same formalism. We would lose all the benefits of the modularity in this case.
- We cannot use sort inclusions neither, for two symmetric reasons:
  - the addition of the `block` instruction must *modify the extension* of the `Inst` sort, appearing in the `DECL_F` formalism,
  - *at the same time*, the addition of procedure declarations must *modify the extension* of the `Decl` sort, appearing in the `INST_F` formalism.

Thus we need a new operation at the *core language level* to be able to express this kind of dependencies. We call it the *sort extension*,

Example. For NOGOM, before creating the complete formalism, we first define a `SUBPROG_F` formalism whose role is to define what the procedure declarations need and we *extend* the two sorts, `Inst` and `Decl` while reusing the `PROGBODY` formalism already defined in section 1.4.4.

```
formalism SUBPROG_F (IDENT:IDENT)=
   form
     open IDENT
     sort Type_Name = IDENT
     sort Param_Decl_s Param_Decl Progspec
     param_decl_s : {Param_Decl}*     -> Param_Decl_s
     param_decl   : Ident # Type_Name -> Param_Decl

     procspec : Ident # Param_Decl_s          -> Progspec
     functspec :Ident # Param_Decl_s # Type_Name -> Progspec
```

```
  end

formalism PROGBODY_F4 (PROGBODY: PROGBODY_F; SUBPROG:SUBPROG_F
                       sharing SUBPROG.IDENT=PROGBODY.IDENT) =
  form
    open PROGBODY.DECL PROGBODY.IDENT
    extend sort Inst Decl
    block : Decl_s # Inst_s                -> Inst
    progdecl : Progspec # SUBPROG.Subprog  -> Decl
  end
```

As for the previous examples, the complete text of the NOGOM specification can be found in the appendices.

Notes.

- It is possible to rename the extended sort while extending it:

  ```
  sort Inst' = extend sort Inst
  ...
  ```

- We have not found any similar operation in the current litterature on the subject. We think it would be interesting to study this operation from a semantic point of view, in particular from a typechecking point of view.

1.6 Parameterized Formalisms as Arguments

We want to be able to pass *parameterized formalisms as arguments* of other parameterized formalisms. We motivate that on the following example.

Example. The creation of a PROG formalism can be:

```
formalism DECL = DECL_F(NOG_EXP, IDENT)
formalism INST = INST_F(NOG_EXP, IDENT)

formalism PROG = PROG_F(DECL, INST, IDENT)
```

Thus we must instantiate the different arguments of PROG_F before calling it. This unnecessarily multiplies formalism instances. *Local declarations* is a first way to avoid this problem (as in STANDARD ML).

But another way to follow is to allow parameterized formalisms as arguments. This does not heavily complicate the syntax or the static semantics of our language but offers much possibilities.

Example. The PROG_F can be written in this way:

```
formalism PROG_F(DECL_F : DECL_F; INST_F : INST_F;
                 EXP : EXPSIG; IDENT : IDENT;
        sharing DECL_F.IDENT=INST_F.IDENT=EXP.IDENT=IDENT
                and DECL_F.EXP=INST_F.EXP= EXP)  =
   form
     formalism DECL = DECL_F (EXP,IDENT)
     formalism INST = INST_F (EXP,IDENT)
     sort Prog
     prog: DECL.Ident # DECL.Decl_s # INST.Inst_s -> Prog
   end
```

We now pass as arguments all the formalisms needed for the instantiation of the arguments inside the body of PROG_F.

The creation of PROG can now be:

```
formalism PROG = PROG_F (DECL_F, INST_F, NOG_EXP, IDENT)
```

Notes.

- We have considered the possibility of giving back a *parameterized formalism as result* of another parameterized formalism. But, this time the syntax has to be slightly changed and the typechecking will be more involved. We prefer to forbid this possibility, for simplicity reasons.
- This kind of specifications has also its drawbacks: in the example, the instantiations of the parameters, DECL_F and INST_F, with their actual arguments preclude any other use of the PROG_F formalism with different DECL or INST formalisms. The reader can consult the appendices to see such examples of reuses.

Chapter 2

The Static Semantics

2.1 Presentation

In this section, we provide insights on the static semantics of our language. We don't completely describe it for the following two reasons:

- On the module language side, the static semantics is completely similar to the static semantics of STANDARD ML(see [3] or [8]).
- On the core language side, the static semantics is, in part, similar to the static semantics of METAL [9].

We describe and give formal rules for the most interesting part of the task, the part concerning the elaboration of the core language structures: formalisms, sorts, and operators.

As in STANDARD ML, the static semantics must elaborate *environments*. These environments consist in list of bindings between *formalism names* and *elaborated formalism expressions*. Thus, we must precise the exact structure of an elaborated formalism and of its components. This is the subject of the following section.

Note. Static semantics versus dynamic semantics.

As in STANDARD ML module language, the static semantics does all the interesting job. We don't have to evaluate environments, just to elaborate them. On the core language side, evaluation consists in building the formalism elaborated by the static semantics. All the information needed to the implementation will be in the formalism environment built by the static semantics. Thus, even from a dynamic semantics point of view, the static semantics is the essential part.

2.2 The Formal Specification of Basic Structures

A *formalism* is principally a collection of sorts and operators. Furthermore, we need to keep track of the subformalism declarations to do the matching between formalisms.

Twos remarks are in order:

- The subformalisms play only a role during the static semantics phase. They disappear during the evaluation phase.
- The storing of operators and sorts is evidently redundant. This simplifies the formal description. Evident optimizations must be taken into account in the implementation.

A *sort* is a collection of operators. Furthermore, we need to keep track of the *subsorts* and of the sorts in which the sort has been extended.

An *operator* is described by its type, i.e. by giving the sort argument list—we call it the source list—and the target sort.

We can now give the specification of these structures in the form of a STANDARD ML signature:

```
signature FORMALISM =
   sig
     datatype  Formalism =
         formalism of Formalism * Sort list * Operator list

     datatype Sort =
         sort of Operator list * Sort list * Sort list

     datatype Operator =
         operator of Sort list * Sort

   end
```

In fact, we do not use this signature directly. We use basic constructors, creating and accessing the corresponding components. We give another signature specifying these constructors:

```
signature FORMALISM =
   sig
     type Formalism
     type Sort
     type Operator
```

```
  val makesort: Ident * Sort -> Sort
  val include:  Sort * Sort  -> Sort
  val extend: Sort * Sort    -> Sort

  val makeop: Ident * Sort list * Sort -> Operator
  val addop: Sort * Operator           -> Operator
end
```

2.3 The Core language

In this section, we give the formal rules for elaborating formalism structures.

We assume the following simplifications:

- We don't describe the basic verifications which are done by the METAL compiler: double definitions of sorts and operators, unknown names, circular definition of sorts, and so on...
- We ignore the **open** and **include** instructions, They are just like syntactic sugar and add nothing to the expressiveness of the language.
- We ignore the subformalism declarations. This is just recursive analysis and elaboration.
- We don't describe the basic operations on sorts, operators, or formalism used in these rules. They are elementary and could be added to the signature description given above.
- As in the STANDARD ML formal definition, we use *injections* of substructures into environments with a single overloaded operation, the $\oplus$ operator.

Finally, we assume that these rules are part of rules elaborating *sequences* of declarations. This explains in part the form of these rules.

These rules have the following TYPOL judgement:

```
ENVIRONMENT, FORMALISM |- CORE_LANGUAGE : FORMALISM,ENVIRONMENT
```

where the **ENVIRONMENT** argument is the global environment currently built and the **FORMALISM** argument is the currently elaborated formalism.

This judgement says that, given an environment and a formalism, these rules elaborate a new formalism and environment. The environment serves as a reference to global names and is *modified* by the **extend** operation.

Note. In the following rules, we have adopted the following naming conventions:

- uppercase identifiers range over structures or constructors on these structures,
- italic lowercase identifiers denote metavariables ranging over lexical categories such as sort identifiers, and so on...
- typewriter style identifiers denote keywords of the object language.

2.3.1 The Sort Declaration

The sort declaration adds a new sort structure to the current formalism. The MAKESORT constructor builds a new sort to be injected into the current formalism. It must verify that the sort is not already defined.

Rule 2.1 *The Sort Declaration*

$$\frac{\vdash \text{MAKESORT}(\mathit{sortid}) : \text{SORT}}{\text{ENV}, \text{F} \vdash \text{sort } \mathit{sortid} : \text{F} \oplus \text{SORT}, \text{ENV}}$$

The sort can be declared with a **sharing** specification. We don't take care here of such constraints. They are simply managed as links between structures (see [7]).

2.3.2 The Sort Inclusion

The sort inclusion modifies the *subsort* field of the corresponding sort and thus modifies the current formalism. The INCLUDE constructor has to give back the modified formalism. The INCLUDE constructor takes already defined sorts as arguments.

Rule 2.2 *The Sort Inclusion*

$$\frac{\text{ENV} \vdash \mathit{sortid}_1 : \text{SORT} \qquad \text{F} \vdash \mathit{sortid}_2 : \text{SORT}' \qquad \text{F} \vdash \text{INCLUDE}(\text{SORT}', \text{SORT}) : \text{F}'}{\text{ENV}, \text{F} \vdash \text{sort } \mathit{sortid}_1 < \mathit{sortid}_2 : \text{F}', \text{ENV}}$$

2.3.3 The Sort Renaming

The sort renaming adds a new sort structure to the current formalism. This new sort is a copy of the corresponding sort. The reader may notice that MAKESORT takes here a supplementary argument, the value of the new defined sort.

Rule 2.3 *The Sort Renaming*

$$\frac{\text{ENV} \vdash \textit{sortid}_2 : \text{SORT} \qquad \vdash \text{MAKESORT}(\textit{sortid}_1, \text{SORT}) : \text{SORT}'}{\text{ENV}, \text{F} \vdash \text{sort } \textit{sortid}_1 = \textit{sortid}_2 : \text{F} \oplus \text{SORT}', \text{ENV}}$$

2.3.4 The Sort Extension

The sort extension is based on two operations:

- The creation of a new sort structure, containing the extended sort as a *subsort*. We can use the INCLUDE constructor after the creation of the new sort.
- The modification of the *extended_sort* field of the extended sort with the new sort. Thus, the EXTEND constructor has to modify the current environment.

Rule 2.4 *The Sort Extension*

$$\frac{\begin{array}{ll} \vdash \text{MAKESORT}(\textit{sortid}_1) : \text{SORT} & \text{F} \vdash \text{INCLUDE}(\text{SORT}, \text{SORT}') : \text{F}' \\ \text{ENV} \vdash \textit{sortid}_2 : \text{SORT}' & \text{ENV} \vdash \text{EXTEND}(\text{SORT}', \text{SORT}) : \text{ENV}' \end{array}}{\text{ENV}, \text{F} \vdash \text{sort } \textit{sortid}_1 = \text{extend sort } \textit{sortid}_2 : \text{F}', \text{ENV}'}$$

2.3.5 The Operator Declaration

The operator declaration adds a new operator structure to the current formalism, once we get back the source and target sort. The $\sum_i$ denotes a list of sorts.

Rule 2.5 *The Operator Declaration*

$$\frac{\begin{array}{ll} \text{F} \vdash \text{TARGET} : \text{SORT} & \vdash \text{MAKEOP}(\textit{opid}, \sum_i \text{SORT}_i, \text{SORT}) : \text{OP} \\ \text{ENV}, \text{F} \vdash \text{SOURCE} : \sum_i \text{SORT}_i & \text{F} \vdash \text{ADDOP}(\text{SORT}, \text{OP}) : \text{F}' \end{array}}{\text{ENV}, \text{F} \vdash \textit{opid} : \text{SOURCE} \rightarrow \text{TARGET} : \text{F}' \oplus \text{OP}, \text{ENV}}$$

2.4 The Module Language

Here we just sketch the description of the static semantics concerning the module part of our language. It is very similar to the static semantics of STANDARD ML. We urge the reader to consult the Formal Definition of STANDARD ML [3].

First, we should provide basic definitions about the management of names: consistency, well-formedness and cycle fredom. They insure a correct naming of formalisms and subformalisms. They are identical with the STANDARD ML definitions.

Second, we should provide formal rules for the module constructs. They also look like the corresponding rules of the STANDARD ML Formal Definition.

Note. In fact, we should describe the additional rules due to parameterized formalism as arguments. But we don't do that here.

2.4.1 The Formalism Matching

The major difference between our module language and the STANDARD ML module language rests on the management of the *signature/formalism matching*. In STANDARD ML, the signature matching is relatively involved, taking care of polymorphim and different type definitions. Moreover, the signature matching is defined between a *structure* and a *signature*, thus the signature matching claims for a *structure instantatiation* (see [8]).

In our case, the formalism matching is directly defined between two formalisms and we just have to take care of the inclusion of the matching formalism into the instance formalism. Regarding subformalism and sharing checks, the definition is as in STANDARD ML.

Appendix A

The Nogom Metal Abstract Syntax

```
chapter PROGRAM
   abstract syntax
      prog -> IDENT PROGBODY ;
      progbody -> DECL_S INSTR_S ;
      PROG ::= prog;
      PROGBODY   ::= progbody ;
end chapter ;

chapter DECLARATIONS
   abstract syntax
      decl_s   -> DECL * ... ;
      const_sdecl   -> IDENT_S TYPENAME EXPR ;
      var_sdecl   -> IDENT_S TYPENAME EXPR_OR_EMPTY ;

      DECL_S   ::=  decl_s ;
      DECL   ::= DECL_S const_sdecl var_sdecl progspec ;
      TYPENAME ::= IDENT;
      EXPR_OR_EMPTY  ::= EXPR empty ;
end chapter ;

chapter 'SUBPROGRAM DECLARATIONS'
   abstract syntax
      progspec  -> PROGSPEC PROGBODY IDENT ;
      procspec   -> IDENT PARAMDECL_S ;
      functspec   -> IDENT PARAMDECL_S TYPENAME ;
      paramdecl_s   -> PARAMDECL + ... ;
      paramdecl   -> IDENT MODE IDENT ;
      inoutmode   -> implemented as SINGLETON ;
      inmode   -> implemented as SINGLETON ;
```

```
      PROGSPEC   ::= procspec functspec ;
      PARAMDECL_S   ::= paramdecl_s empty ;
      PARAMDECL ::= paramdecl ;
      MODE ::= inoutmode inmode ;
end chapter ;

chapter INSTRUCTIONS
   abstract syntax
      instr_s  -> INSTR + ... ;

      INSTR_S ::=  instr_s;
end chapter ;

chapter 'SIMPLE INSTRUCTIONS'
   abstract syntax
      null    -> implemented as SINGLETON ;
      tag   -> TAGNAME INSTR_S ;
      exit    -> TAGNAME ;
      assign   -> IDENT EXPR ;
      return   -> EXPR_OR_EMPTY ;
      proccalln   -> IDENT EXPR_S ;
      proccall0   -> IDENT ;

      TAGNAME ::= IDENT;
end chapter ;

chapter 'COMPOUND INSTRUCTIONS'
   abstract syntax
      if -> EXPR INSTR_S ELSE_PART ;
      while   -> EXPR INSTR_S ;
      block   -> DECL_S INSTR_S ;

      ELSE_PART ::= INSTR_S null;
      INSTR   ::= INSTR_S null assign tag exit proccalln
                  progbody proccall0 return if while block;
end chapter ;

chapter EXPRESSIONS
   abstract syntax
      expr_s   -> EXPR + ... ;
      and   -> EXPR EXPR ;
      or   -> EXPR EXPR ;
```

```
      equal  -> EXPR EXPR ;
      inf    -> EXPR EXPR ;
      sup    -> EXPR EXPR ;
      plus -> EXPR EXPR ;
      minus    -> EXPR EXPR ;
      times  -> EXPR EXPR ;
      not    -> EXPR ;
      functcalln -> IDENT EXPR_S ;
      functcall0 -> IDENT ;

      EXPR_S   ::= expr_s ;
      EXPR   ::= and or equal inf sup plus minus times not
                    functcalln functcall0 LITTERAL IDENT;
end chapter ;

chapter 'IDENTIFIERS and LITTERALS'
   abstract syntax
      ident_s -> IDENT + ... ;
      ident -> implemented as IDENTIFIER ;
      empty    -> implemented as SINGLETON ;
      int    -> implemented as INTEGER ;
      booltrue   -> implemented as SINGLETON ;
      boolfalse  -> implemented as SINGLETON ;

      IDENT_S   ::= ident_s ;
      IDENT   ::= ident ;
      LITTERAL   ::= INTEGER booltrue boolfalse ;
      INTEGER ::= int;
end chapter ;
```

Appendix B

The Simplest Nogom Specification

We construct here an abstract syntax for a language with independent sets of instructions and declarations.

```
formalism IDENT =
   form
      sort Ident
      ident: -> Ident
   end

formalism BOOL =
   form
     sort Bool
     booltrue:  -> Bool
     boolfalse: -> Bool
   end

formalism INT =
   form
     sort Int
     int: -> Int
   end

formalism LITT  =
   form
      sort Litt
      sort BOOL.Bool < Litt
      sort INT.Int   < Litt
   end
```

```
formalism EXP_F =
   form
     sort Exp_s Exp
     exp_s:     {Exp}*     -> Exp_s
     exp_and:   Exp # Exp  -> Exp
     exp_not:   Exp        -> Exp
     exp_plus:  Exp # Exp  -> Exp
     exp_mult:  Exp # Exp  -> Exp
   end

formalism NOG_EXP_F(EXP : form sort Exp end;
                    LITT:LITT; IDENT:IDENT) =
   form
     sort EXP.Exp     < Exp
     sort LITT.Litt   < Exp
     sort IDENT.Ident < Exp
   end

-- The signatures specifying the DECL and INST arguments.

formalism EXPSIG =
   form
      formalism IDENT: IDENT
      formalism LITT: LITT
      sort Exp
   end

formalism DECLSIG =
   form
     formalism EXP : EXPSIG
     formalism IDENT : IDENT
     sort Decl_s
   end

formalism INSTSIG =
   form
     formalism EXP : EXPSIG
     formalism IDENT : IDENT
     sort Inst_s
   end

formalism DECL_F (EXP : EXPSIG; IDENT : IDENT
                  sharing IDENT = EXP.IDENT)=
```

```
   form
     formalism EXP_OR_EMPTY =
        form
          sort Exp_Or_Empty
          sort EXP.Exp  < Exp_Or_Empty
          empty:        -> Exp_Or_Empty
        end
     open IDENT EXP_OR_EMPTY
     sort Type_Name = Ident
     sort Decl_s Decl
     decl_s:   {Decl}*                              -> Decl_s
     var_decl: Ident # Type_Name # Exp_Or_Empty -> Decl
   end

formalism INST_F (EXP : EXPSIG; IDENT : IDENT
                  sharing IDENT = EXP.IDENT)  =
   form
     open IDENT EXP
     sort Tag_Name = Ident
     sort Inst_s Inst Else_Part

     sort Inst_s <  Else_Part
     empty: -> Else_Part

     inst_s: {Inst}+              -> Inst_s
     null:                        -> Inst
     tag: Tag_Name # Inst_s       -> Inst
     exit: Ident                  -> Inst
     assign: Ident # Exp          -> Inst
     if : Exp # Inst_s # Else_Part-> Inst
   end

-- The construction of the NOGOM formalism

formalism PROGBODY_F(DECL : DECLSIG; INST : INSTSIG;
                     EXP : EXPSIG; IDENT : IDENT;
        sharing DECL.IDENT=INST.IDENT=EXP.IDENT=IDENT
                and DECL.EXP=INST.EXP= EXP)  =
   form
     sort Progbody
     progbody : DECL.Decl_s # INST.Inst_s -> Progbody
   end
```

```
formalism PROG_F(PROGBODY: form sort Progbody end;IDENT:IDENT)=
   form
     sort Prog
     prog: IDENT.Ident # PROGBODY.Progbody  -> Prog
   end

-- Instantiations. We will later use DECL, INST, and PROGBODY.

formalism NOGOM_EXP = NOG_EXP_F(EXP, LITT, IDENT)
formalism DECL = DECL_F(NOGOM_EXP,IDENT)
formalism INST = INST_F(NOGOM_EXP,IDENT)
formalism PROGBODY = PROGBODY_F(DECL, INST)
formalism PROG1 = PROG_F (PROGBODY, IDENT)
```

Appendix C

Nogom With a Block Instruction

We add here a **block** operator. This instruction needs declarations; hence we define a new INST_F formalism with a DECL argument for extending the **Inst** sort. The generation of the PROG2 formalism uses the previous PROGBODY_F and PROG_F.

```
formalism INST_F2 (INST:INSTSIG; DECL:DECLSIG;
                   EXP:EXPSIG; IDENT:IDENT
                   sharing DECL.IDENT = EXP.IDENT = IDENT)=
   form
     extend sort Inst
     block : DECL.Decl_s # Inst_s -> Inst
   end

-- The new formalism with a local declaration

local formalism INST2 = INST_F2(INST,DECL,,NOGOM_EXP,IDENT)
   in formalism PROG2 = PROG_F(PROGBODY_F(DECL, INST2),IDENT)
```

Appendix D

Nogom With Subprogram Declarations

We add here subprogram declarations. This time, we define a new DECL_F with taking PROGBODY as parameter (to get the Subprog sort)

```
formalism SUBPROG_F (IDENT:IDENT)=
   form
     open IDENT
     sort Type_Name = IDENT
     sort Param_Decl_s Param_Decl Progspec
     param_decl_s : {Param_Decl}*      -> Param_Decl_s
     param_decl   : Ident # Type_Name -> Param_Decl

     procspec : Ident # Param_Decl_s          -> Progspec
     functspec :Ident # Param_Decl_s # Type_Name -> Progspec
   end

formalism PROGBODY_F3 (PROGBODY: PROGBODY_F; SUBPROG:SUBPROG_F
                       sharing SUBPROG.IDENT=PROGBODY.IDENT)=
   form
     open PROGBODY.DECL PROGBODY.IDENT

     extend sort Decl
     progdecl : Progspec # SUBPROG.Subprog  -> Decl
   end

-- The new formalism with another instance of PROGBODY
```

```
formalism SUBPROG = SUBPROG_F(IDENT)
local formalism PROGBODY3 = PROGBODY_F3(PROGBODY, SUBPROG)
      in formalism PROG3 = PROG_F(PROGBODY3, IDENT)
```

Appendix E

The Complete Nogom Specification

We add here a block instruction *and* subprogram declarations. We need to extend the Decl and Inst sorts.

```
formalism PROGBODY_F4 (PROGBODY: PROGBODY_F; SUBPROG:SUBPROG_F
                       sharing SUBPROG.IDENT=PROGBODY.IDENT)=
   form
     open PROGBODY.DECL PROGBODY.IDENT
     extend sort Inst Decl
     block : Decl_s # Inst_s                -> Inst
     progdecl : Progspec # SUBPROG.Subprog  -> Decl
   end

local formalism PROGBODY4 = PROGBODY_F4(PROGBODY, SUBPROG)
      in formalism PROG4 = PROG_F(PROGBODY4, IDENT)
```

Bibliography

[1] "An Informal Introduction to Specifications Using Clear", R.M. Burstall, J.A. Goguen, in *The Correctness Problem in Computer Science*, Boyer and Moore eds, Academic Press, 1981.

[2] "The Algebraic Specification Formalism ASF", in *Algebraic Specification*, J.A. Bergstra, J. Heering, P. Klint eds, Addison Wesley, 1989.

[3] "The Definition of Standard ML, Version 2", R. Harper, R. Milner, M. Tofte, Report ECS-LFCS-88-62, Edinburgh University, 1988.

[4] "Introduction to Standard ML", R. Harper, Report ECS-LFCS-86-14, Edinburgh University, 1986.

[5] "Modules For Standard ML", D. MacQueeen, Polymorphism Newsletter, II-2, October 1985. (An earlier version appeared in *Proceedings 1984 ACM Symposium on LISP and Functional Programming*).

[6] "The Virtual Tree Processor", B. LANG in *Generation of Interactive Programming Environment*, Intermediate Report, J. Heering, J. Sidi, A. Verhoog Eds., CWI Report CS-R8620, Amsterdam, May 1986.

[7] "An Implementation of Standard ML Modules", D. MacQueeen, Research Report, AT&T Bell Laboratories, 1986.

[8] "Four Lectures On Standard ML", M. Tofte, Report ECS-LFCS-89, Edinburgh University, 1989.

[9] "METAL : A formalism to specify formalisms", G. Kahn, B. Lang, B. Melese, E. Morcos, INRIA, research report, April 1983.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399