DESIGN AND IMPLEMENTATION OF A GENERIC, LOGIC AND FUNCTIONAL PROGRAMMING LANGUAGE[†]

Didier BERT, Rachid ECHAHED

ABSTRACT

This paper presents the broad outlines of LPG, a language designed for generic specification and programming. In this language one may specify different modules which can represent either particular algebras (ADTs), families of algebras (generic data types and enrichments) or Σ-structures (theories). This language is based on Horn clause logic with equality which permits logic and functional programming to be combined. As modules in LPG can be generic, an instantiation mechanism is needed ; such a mechanism is described here as well as the interpreter and an E-unification algorithm, thus making LPG a powerful programming language.

[†] This work is supported in part by the "Centre National de la Recherche Scientifique, GRECO-Programmation", and in part by the "Centre National d'Etudes des Télécommunications", under grant 84/5B.008.790.92.45./PAA.

Authors' address : LIFIA / IMAG BP 68 -38402 Saint Martin d'Hères CEDEX FRANCE.

1-INTRODUCTION

A specification language is a tool with which one may specify problems, and then deduce the resulting consequences. In order to verify the correctness of these consequences, the language must be based upon some logical system. [Goguen and Burstall 84] give a generalization of such systems by introducing the notion of "institution". Some examples of institutions are Horn clause logic without equality that underlies ordinary PROLOG, and conditional equational logic on which some specification languages are based , like OBJ2 [Futatsugi et al. 84] and CLEAR [Burstall and Goguen 77] [Burstall and Goguen 80]. Another institution which generalizes the two examples above is Horn clause logic with equality. This gives rise to interesting specification languages where logic and functional programming are combined. EQLOG [Goguen and Meseguer 84] is such a language.

For several years, we have been working on the design and implementation of an applicative language, called LPG, which is both a specification language with an algebraic formalism, and a functional programming language in which some parts of specification can be executed with good efficiency [Bert 83]. One of the aims of the project is to give the user the greatest facilities for parameterizing declarations (generic units). This choice allows a static type checking to be done, thus providing an increased security of programming together with a better efficiency at run time, while relaxing constraints imposed by the usual non generic typed languages. We have now included a predicate definition mechanism and an algorithm for solving equations and evaluating predicates in a uniform way. In order to include these facilities, our language is underlied by Horn clause logic with equality.

In this paper we present an actual implementation of high level notions. An interesting point is the instantiation mechanism : generic operators look like functional forms [Backus 78] or second order operators, and the same applies to generic predicates.

2-MAIN FEATURES OF LPG

2.1 Program units

LPG (for "Langage de Programmation Générique") allows abstract data types to be defined, following the algebraic specification method. Other program units are enrichments of data types, and "properties". Syntactically, a program unit is a presentation (S, Ω, Π, C) where S is a set of sorts, Ω a family of operator sets indexed by $S^* \times S$, Π a family of predicate sets indexed by S^*, such that $\Omega \cap \Pi = \emptyset$. For each sort s, there is an equality predicate symbol $=_s$, hereafter noted "==". C is a set of Horn clauses, of the form :

$P_0 \Leftarrow\!= P_1, \dots, P_n \qquad (n \geq 0)$

Where P_i is either $Q(t_1, \dots, t_m)$ or $t_1 == t_2$, with Q a predicate symbol and $t_1, \dots, t_m$ terms. Each term or predicate must be well typed with respect to the arity of the operator or predicate symbols that compose it.

For reasons of structuration and efficiency, we hierarchically separate the axiomatization of the "==" predicate symbol from that of the other predicates. The axioms (equations) defining "==" may be conditional, i.e. of the three following forms :

$t_1 == t_2$
or $t_1 ==$ **if** b **then** t_2 **fi** which means : $t_1 == t_2 \Leftarrow\!= b == \text{true}$
or $t_1 ==$ **if** b **then** t_2 **else** t_3 **fi** which means : $t_1 == t_2 \Leftarrow\!= b == \text{true}$
$t_1 == t_3 \Leftarrow\!= b == \text{false}$

Note that t_2 and t_3 may also include conditional expressions.

Moreover, the set of operator symbols Ω is divided into "constructors" and other operators ; these ones must be sufficiently completely defined [Guttag and Horning 78] with respect to the constructors. Examples of program units in LPG are :

Example 1 : Data type of the natural numbers :

```
type Nat
sorts nat
constructors
   0 :   ->nat
```

```
    s  : nat ->nat
operators
    +, * : (nat, nat) -> nat
variables
    n, m : nat
axioms
    1 :  0 + n   == n
    2 : s(m) + n == s(m+n)
    3 :  0 * n   == 0
    4 : s(m) * n == n + m * n
end
```

Example 2 : Enrichment of the data types Nat and Bool (Bool is supposed already defined) ; cats_birds (c, b, h, l) holds iff when c is the number of cats and b the number of birds, h is the total number of heads and l the total number of legs.

```
enrich On_bool_and_nat
operators
    =     : (nat, nat) -> bool
predicates
    odd, even : nat
    cats_birds : (nat, nat, nat, nat)
variables
    n, m, cats, birds : nat
axioms
    1 : 0 = 0 == true
    2 : s(n) = 0 == false
    3 : 0 = s(m) == false
    4 : s(n) = s(m) == n = m
clauses
    1 : even(0)
    2 : even(s(n)) <== odd(n)
    3 : odd(s(n)) <== even(n)
    4 : cats_birds(cats, birds, cats + birds, 4 * cats + 2 * birds)
end
```

In order to define generic (parameterized) units, classes of data types, operators and predicates –that is to say classes of algebras– need to be characterized. In LPG, this is made by property declarations (like theories in OBJ2 and CLEAR).

Example 3 : Declaration of the associativity property of an operator :

```
prop Assoc
sorts t
operators
    * :  (t, t) -> t
variables
    x, y, z : t
axioms
    1 : x * (y * z) == (x * y) * z
end
```

In the example below, we give the presentations of properties which characterise respectively any data type (cf. TRIV in CLEAR [Burstall and Goguen 80]), any operator, the category of "linear sequence" structures and any binary relation.

Example 4 :

```
prop Ftype
sorts t
end

prop Cat_Seq
sorts  t  dom
operators
```

```
prop One_operator
sorts  t1  t2
operators
    f : t1 -> t2
end

prop Relation
sorts t
```

```
    nil_op : -> dom
    cons_op : (t, dom) -> dom
end
```

```
predicates
    p : (t, t)
end
```

2.2 Semantics of data types and properties

A model of a first order signature $\Sigma = (S, \Omega, \Pi)$ consists of sets A_s for each $s \in S$, functions $f_\omega : A_{s1} \times \ldots \times A_{sn} \rightarrow A_s$ for each operator symbol $\omega : (s1, \ldots, sn) \rightarrow s$, $\omega \in \Omega$ and subsets p_π over $A_{s1} \times \ldots \times A_{sn}$ for each predicate symbol $\pi : (s1, \ldots, sn)$, $\pi \in \Pi$. We say that $p_\pi(a1, \ldots, an)$ "holds" iff $(a1, \ldots, an)$ is in p_π.

Let I_M be the interpretation of a signature Σ by a model M, we note $I_M(t)$ the value of a term t in M, $I_M[f](t(X))$ the value of t(X) which extends f : X -> M, where X is a set of variables and t(X) is a term containing variables in X. Identically, == is interpreted by the actual equality in M, and $I_M[f](\pi(t_1(X), \ldots, t_n(X)))$ holds in M, iff $p_\pi(I_M[f]\,(t_1(X)), \ldots, I_M[f]\,(t_n(X)))$ holds.

A model M satisfies a clause $P \Leftarrow\!\!= P_1, \ldots, P_n$ iff for every f : X -> M where X is the set of variables of the clause, $I_M[f]\,(P)$ holds in M, whenever $I_M[f]\,(P_i)$ holds in M for all i. A model M satisfies a set C of clauses iff it satisfies every clause in C. **Mod** (Σ) denotes the class of models of a signature Σ, and **Mod** (Σ, C) the class of Σ-models which satisfy C.

Semantics of a property presentation pp = (Σ, C) is the class Mod(Σ, C). On the contrary, semantics of a data type presentation dtp = (Σ, C) is given by the "standard" model of dtp, which is the Herbrand universe on the equivalence classes determined by the congruence relation generated by == on the terms. If a new presentation contains "old" sorts, operators or predicates, its interpretations must not change their own ones. It is particularly the case for enrichments, or for properties importing sorts of data types.

In LPG, we denote a model of a property named p by :

p [$T_1, \ldots, T_k$ **operators** $F_1, \ldots, F_m$ **predicates** $P_1, \ldots, P_n$]

where $T_1, \ldots, T_k$ are the carriers of an (heterogeneous) algebra (here they are sorts of data types), $F_1, \ldots, F_m$ are maps over the algebra associated to the operator symbols of the property p, and $P_1, \ldots, P_n$ are predicates interpreting predicate symbols of p. For example, Assoc[nat **operators** +] denotes a model of Assoc and therefore indicates that "+" on naturals is associative, if the semantic conditions are satisfied.

From a dual point of view, a theory associated to a presentation p is the set of the clauses which hold in every model of p [Goguen and Burstall 84]. As in CLEAR, we need the notion of morphism between signatures and between theories. Let us briefly recall the definitions.

Definition 1 : A signature morphism $\Phi : (S, \Omega, \Pi) \rightarrow (S', \Omega', \Pi')$ is a triple (f, g, h) with f : S -> S' a map on sorts, g : $\Omega \rightarrow \Omega'$ a sort-preserving map on operators, and h : $\Pi \rightarrow \Pi'$ a sort-preserving map on predicates.

Let $\Phi : \Sigma \rightarrow \Sigma'$ be a signature morphism, there exists a function $\hat{\Phi} : T_\Sigma(X) \rightarrow T_{\Sigma'}(X')$ which "translates" terms over the signature Σ into terms over Σ'. Similarly, it is possible to deduce the map $\bar{\Phi}$: Mod(Σ') -> Mod(Σ), which, for each Σ'-model M' associates a Σ-model M.

Definition 2 : A theory morphism $\Phi : (\Sigma, C) \rightarrow (\Sigma', C')$ is a signature morphism $\Phi : \Sigma \rightarrow \Sigma'$, such that $\bar{\Phi}(\mathrm{Mod}(\Sigma', C')) \subseteq \mathrm{Mod}(\Sigma, C)$.

In LPG, it is possible to declare theory morphisms. This is made by a "satisfies", "inherits" or "combines" statement [Bert 83] within property units. For example, let us consider the property that expresses the equality :

Example 5 :

```
prop Equality
sorts  t
```

```
operators
    = : (t, t) -> bool
variables
    x, y, z : t
axioms
    1 : x = x == true
    2 : x = y == y = x
    3 : (x = y and y = z) implies (x = z) == true
satisfies Ftype[t]
end
```

In this case, the "satisfies" declaration means that there exists a theory morphism between Ftype (theory without operators nor predicates) and the theory Equality. Therefore, any model "Equality[T **operators** equ]" is also a model "Ftype[T]", by "forgetting" the operator "equ" and the equations attached to it.

Informally, whenever a statement "satisfies p_1[...], ..., p_n[...]" occurs in a property unit p, this means that there exists a theory morphism between each of the theories p_1, ..., p_n and p. A statement "inherits p_0[...]" occurring in a property unit p has the same meaning as "satisfies p_0[...]", but in addition the set of axioms in p is augmented by the axioms in p_0 (up to renaming induced from the signature morphism). Finally, a statement "combines p_1[...], ..., p_n[...]" in a property p means that there are no more equations in p apart those in p_1, ..., p_n, imported into p (up to renaming) ; obviously, there are theory morphisms between p_1, ..., p_n and p.

Now, we give in the example below, the presentations of partial and total order properties where morphism declarations are more expressive :

Example 6 :

```
prop Partial_Order
sorts t
operators
    ≤, = : (t,t) -> bool
variables
    x, y, z : t
axioms
    1 : x ≤ x == true
    2 : x = y ==  x ≤ y and y ≤ x
    3 : (x ≤ y and y ≤ z) implies (x ≤ z) == true
satisfies
    Equality[t operators =]
end
```

```
prop Total_Order
sorts t
operators
    ≤, = (t, t) -> bool
variables
    x, y, z : t
axioms
    1 : x ≤ y or y ≤ x == true
inherits
    Partial_Order[t operators ≤, =]
end
```

2.3 Generic data types

An abstact data type (or an enrichment) is generic if it is parameterized by a property, called "**required**" property. This property characterizes the class of all models which may be actual parameters of the declared unit. For example, the data type of linear sequences, parameterized by the type of the elements is defined by :

Example 7 :

```
type Seq requires Ftype[elem]
sorts seq
constructors
    nil    : -> seq[elem]
    <+     : (elem, seq[elem]) -> seq[elem]
end
```

Ftype[elem] denotes a "formal" model of the required property. A formal model "p[...]" parameterizing a generic unit G gives the image of sorts, operators and predicates of the property p, inside G. Therefore, it defines a theory morphism. In the example 7, the morphism is F_seq : Ftype -> Seq such that F_seq = (f, g, h) where "$f : t \mapsto elem$" and "$g, h : \emptyset \mapsto \emptyset$". Following [Goguen and Burstall 84], the so defined morphism induces between the models a functor F_seq$^\$$: Mod(Ftype) -> Mod(Seq), called the free functor, which is left adjoint to the forgetful functor F_seq* : Mod(Seq) -> Mod(Ftype). The semantics of this generic data type is the set of models F_seq$^\$$(Mod(Ftype)), and from the theory point of view, we shall be able to speak about the theory Seq constrained by F_Seq, as it is defined in CLEAR [Burstall and Goguen 80]. In the case of non generic data types, like Nat, we can propose the same semantics, by considering the morphism F_Nat : Empty -> Nat {0, s} where Empty is the empty theory defined by :

```
prop Empty
end
```

and where Nat {0, s} means the theory Nat with only the two bracketed operators. F_Nat$^\$$(Empty) is reduced to only one (up to isomorphism) model, which is the initial algebra of the category Mod(Nat{0, s}), like in the usual semantics of abstract data types [Goguen et al. 78]. Let us notice that it always exists a morphism between Empty and any other theory.

Particular conditions should be required to insure that the data types passed as parameters are "protected", but we do not develop these restrictions here, see [Goguen and Meseguer 82] [Thatcher et al. 82] [Ehrig et al. 84].

2.4 Enrichments, Model declaration

An enrichment is a set of operator or predicate declarations which may require specific properties. For example, the operator of equality on sequences must require equality on the elements. Sorting operators must require a total or partial order, and so on. Example 8 shows such enrichments and indicates also how to declare models of properties in LPG, in a "**models**" rubric :

Example 8 :

```
enrich Equ_Seq requires Equality[elem operators eq]
operators
   = : (seq[elem], seq[elem]) -> bool
variables
   a, b  :  elem
   s, u  :  seq[elem]
axioms
   1 : nil = nil == true
   2 : a <+ s = nil == false
   3 : nil = b <+ u == false
   4 : a <+ s = b <+ u == eq(a, b) and s = u
models
   E_Seq : Equality [seq[elem] operators =]
end
```

```
enrich On_Nat_Seq
operators
   rev_iota : nat -> seq[nat]
variables
   n : nat
axioms
   1 : rev_iota (0) == nil
   2 : rev_iota (s(n)) == s(n) <+ rev_iota (n)
end
```

This model declaration is semantically valid only if the axioms of the equality hold in the theory Equ_Seq, i.e. the morphism E_Seq : Equality -> Equ_Seq defined by :

$$\begin{array}{rcl} t & \mapsto & \text{seq[elem]} \\ = :(t, t) \text{ -> bool} & \mapsto & = : (\text{seq[elem], seq[elem]}) \text{ -> bool} \end{array}$$

is indeed a theory morphism.

Let us note that if this model declaration is valid, then Ftype[seq[elem]] is also a declared model (of Ftype), without any extra verification, because of the morphism Ftype -> Equality (cf. example 5).

The reader has probably noted that in the first examples, some theory morphism and model declarations have been omitted. The reason is that it was too early to introduce them.

2.5 Instantiation of sorts and operators

The declarations of program units are made in an environment. This environment is composed of the names of the theories (e.g. Seq, Assoc, Ftype, ...), but also of the sorts, operators, predicates and models of the data theories. These names are individually parameterized by the required property of the unit in which they have been declared. An explicit instance of a sort (sort expression) is given by the association between the generic sort and a model of the required property. For example, we can write : "seq.Ftype[nat]" for the data type of the sequences of natural numbers.

If we have declared the data type set as following :

Example 9 :

```
type Set requires Equality[elem operators eq]
sorts set
constructors
    empty :  -> set[elem]
    insert : (elem, set[elem]) -> set[elem]
...
end
```

an explicit occurrence of "set" is : (*) set.Equality[nat **operators** =]

where "=" is the equality operator over the natural numbers. If "Ftype[nat]" is a declared model in the environment, we can simply write "seq[nat]" for "seq.Ftype[nat]" ; from the actual sort, LPG tries to reconstitute the complete instance of the sort "seq". Identically, if "Equality[nat **operators** =]" is a model named E_nat in the environment, writing "set[nat]" is the same thing as writing "set.E_nat", or the expression (*). Because of the generic declarations of models, like E_Seq in Equ_Seq, it is possible to combine sorts without any intermediary declaration, as in the sort expression "set[seq[seq[nat]]]" which denotes the sort of the data theory of the sets of sequences of sequences of natural numbers. In this expression, The equality model in the data type "seq[seq[nat]]" is completely deduced from the declarations, without extra efforts. Sort expressions can freely be used in the profiles of the operators. We can notice that within the definition of Set, the expression "set[elem]" is indeed an implicit instance of "set" with the formal model "set.Equality[elem **operators** eq]". In this same program unit, "seq[elem]" would mean "seq.Ftype[elem]" where Ftype[elem] is deduced from the formal model by the theory morphism " Ftype -> Equality".

Operators and predicates defined in generic units can also be directly instantiated in a term or a literal, according to the type of the operands. For example, in LPG, knowing that "3" is a denotation of a natural number and "{1, 2}" a denotation of a set of natural numbers, then the term "insert(3, {1, 2})" contains an occurrence of "insert" instantiated by the model "Equality[nat **operators** =]". In the same way, knowing that [a, b, c] is a denotation of a sequence (for a <+ (b <+ (c <+ nil))), the term "insert([[2]], { [[1, 2], [3]], [nil] })" is an instance of "insert" with the model "Equality[seq[seq[nat]] **operators** =]" where the operator "=" is an occurrence of the "=" in Equ_Seq, instantiated itself by the model "Equality [seq[nat] **operators** =]", and so on. The user is not bothered with complete instantiations of the generic operators or predicates ; the system undertakes to do it, from the type checking and the model declarations.

There exists another kind of operator/predicate instantiation, that is the explicit instantiation. Some examples allow this case to be illustrated.

Example 10 :

```
enrich Ex1 requires One_operator [t1, t2 operators f]
operators
    alpha : seq[t1] -> seq[t2]
predicates
    graph : (t1, t2)
variables
    a : t1
    s : seq[t1]
axioms
    1 : alpha (nil) == nil
    2 : alpha (a <+ s) == f (a) <+ alpha (s)
clauses
```

```
enrich Ex2 requires Relation[t predicates r]
predicates
    reflex : (t, t)
variables
    x, y : t
clauses
    1 : reflex (x, y) <== r (x, y)
    2 : reflex (x, x)
end
```

```
    1 : graph (a, f (a))
end

enrich Ex3 requires Total_Order[elem operators ≤, =]
operators
    sort : seq[elem] -> seq[elem]
    insert : (elem, seq[elem]) -> seq[elem]
predicates
    ordered? : seq[elem]
variables
    x, y : elem
    s : seq[elem]
axioms
    1 : sort(nil) == nil
    2 : sort(x <+ s) == insert(x, sort(s))
    3 : insert(x, nil) == x <+ nil
    4 : insert(x, y <+ s) == if x ≤ y then x <+ (y <+ s)
                             else y <+ insert(x, s)
                             fi
clauses
    1 : ordered? (s) <== sort(s) == s
end
```

```
enrich Ex4 requires Cat_Seq[t, dom operators n, f]
operators
    hom : seq[t] -> dom
variables
    a : t
    s : seq[t]
axioms
    1 : hom (nil) == nil
    2 : hom (a <+ s) == f(a, hom(s))
end
```

The operator "alpha" applies the formal operator "f" to each element of the sequence parameter ; "graph" is the relation of the graph of f, for any f ; "reflex" is the least reflexive relation which contains the formal relation r ; "sort" is a sorting operator ; "ordered?" is a predicat which holds whenever its sequence parameter is ordered (according to the total order used) ; "hom" (for homomorphism) transforms a sequence of elements to a term where the constructors nil and <+ are replaced by n and f respectively. An explicit instance of a generic operator or predicate is given by the operator or predicate symbol followed by the actual operators and predicates of the instantiating model. The term "alpha[+]" is an operator expression with the model "One_operator [(nat, nat), nat **operators** +]", and with the profile "seq[(nat, nat)] -> seq[nat]". Similarly, "graph [+]" is a predicate expression which denotes the graph of the operation "+". Operator/predicate expressions have the same prerogatives as operator/predicate symbols, and the explicit instantiation mechanism may be nested at any level, like in "reflex [graph [alpha [alpha [s]]]]" which is a predicate denoting a subset of (seq [seq [nat]], seq [seq [nat]]).

More generally, in the generic context of a formal model M, a sort, operator or predicate expression is either a formal parameter given in M, or an effective symbol with an instantiating model. A model contains recursively sort, operator and predicate expressions. Considering these expressions as trees, we call "ending model" a model occurring at a leaf of such a tree. By construction, an ending model is always of the form Φ (M), including "Empty" as model of a non generic symbol. Thus, such an expression τ is :

- s_i, ω_i, π_i : a formal sort, operator or predicate of M ;
- u.p'[t_1, ... **operators** f_1, ... **predicates** p_1, ...]
  where p' is the required property of u , and t_1..., f_1..., p_1... are not formal ;
- v.Φ (M) where Φ throws down M to the required model of v.

Let p be the required property of the generic context, and let M' be p[T_1, ..., T_k **operators** F_1, ..., F_m **predicates** P_1, ..., P_n], the instantiation of τ by M', noted τ.M' is defined by :

(1) s_i.M' = T_i
(2) ω_i.M' = F_i
(3) π_i.M' = P_i
(4) (u.p'[t_1, ... **operators** f_1, ... **predicates** p_1, ...]).M' = u.p'[t_1.M', ... **operators** f_1.M', ... **predicates** p_1.M', ...]
(5) (v.Φ (M)).M' = v.Φ (M')

Inductively, to instantiate a term f(a_1, ..., a_n) or a formula Q(a_1, ..., a_m) by a model M', we have :

f (a_1, ..., a_n).M' = f.M'(a_1.M', ..., a_n.M')
Q (a_1, ..., a_m).M' = Q.M' (a_1.M', ..., a_m.M')

3–LOGIC PROGRAMMING IN LPG

Given a specification SP (i.e. a set of specifications) and a goal statement "$<== P_1, ..., P_n$" (i.e. a Horn clause whose head is empty) where the atomic formulae $P_1, ..., P_n$ are completely instantiated (containing no formal symbol) , we wish to find "solutions" of these formulae, that is to say values of the variables which make the clause valid in the theory SP. As in ordinary logic programming [Van Emden and Kowalski 76], our algorithm is built on the resolution principle [Robinson 65] that we briefly recall below :

Given two clauses :

$A <== A_1, ..., A_i, ..., A_n$

$B <== B_1, ..., B_m$

Suppose there exists a substitution σ such that $\sigma(A_i) = \sigma(B)$, by applying the resolution principle, the following clause called resolvent is deduced :

$$\sigma(A <== A_1, ..., A_{i-1}, B_1, ..., B_m, A_{i+1}, ..., A_n)$$

In our case, the substitution σ is not the most general unifier as in traditional logic programming, but an E-unifier depending on the set E of equations that define the functions. Furthermore, when B is generic (so is A_i), the actual models instantiating B and A_i must be the same ; otherwise LPG deduces the model instantiating B from that of A_i.

Starting from a goal statement C_0 and using the set of clauses of the specification SP, the algorithm tries to find sequences $C_0, C_1,...,C_q$, one at a time, where C_i ($i \geq 1$) is either a clause in SP or a resolvent and C_q is the empty clause. As in ordinary PROLOG, the strategy used is depth-first (with backtracking), the atomic formulae of a goal are evaluated from left to right and clauses are considered in their order of declaration.

Solving an equation $t_1 == t_2$ modulo a set of equations is called **E-unification**. Two terms t and t' are said E-unifiable iff there exists a substitution σ, called E-unifier of t and t', such that $\sigma(t) =_E \sigma(t')$ with $=_E$ being the congruence generated by the set E of equations. Let U(t, t', E) denote the set of all unifiers of t and t'. When $E=\emptyset$, U(t, t', E) contains, if it is not empty, one unifier (up to renaming) called the most general unifier. Unfortunately, such a unique unifier does not always exist as soon as $E \neq \emptyset$. That is why the notion of complete set of E-unifiers is introduced.

Definition 3 : Let E be a set of equations, t and t' be two terms and W be a set of variables containing V, where V is the set of variables occurring in t and t'. A set S of substitutions is called a **complete set of E-unifiers** of t and t' away from W iff :

(1) $\forall \sigma \in S, D(\sigma) \subseteq V$ and $I(\sigma) \cap W = \emptyset$

(2) $\forall \sigma \in S, \sigma \in U(t, t', E)$

(3) $\forall \sigma \in U(t, t', E), \exists \sigma' \in S, \sigma' \leq_E \sigma\ [V]$

S is said to be minimal iff it satisfies the further condition :

(4) $\forall \sigma, \sigma' \in S, \sigma \leq_E \sigma'\ [V] \Rightarrow \sigma = \sigma'$

Note that $D(\sigma)$ denotes the domain of the substitution σ (i.e. the set of variables x such that $\sigma(x) \neq x$), $I(\sigma)$ denotes the set of variables introduced by σ (i.e. the set of variables occurring in $\sigma(x)$ for all x in $D(\sigma)$) and $\leq_E$ [V] stands for the preorder on substitutions defined by : $\sigma \leq_E \sigma'\ [V]$ iff $\forall x \in V, \sigma(x) \leq_E \sigma'(x)$ where $t \leq_E t'$ iff $\exists\ \theta, \theta(t) =_E t'$.

In the case where a set of equations E possesses a canonical term rewriting system, an E-unification algorithm has been given [Hullot 80]. This algorithm is based on the narrowing relation.

Definition 4 : Let $\mathfrak{R}$ be a term rewriting system. A term t is said $\mathfrak{R}$-narrowable into t' at occurrence u, with t/u (the subterm of t at occurrence u) not a variable, iff there exists a rule $g_i \rightarrow d_i$ in $\mathfrak{R}$ such that t/u and g_i are unifiable. Let μ be their most general unifier, then t' is obtained by replacing in $\mu(t)$ the subterm $\mu(t/u) = \mu(t)/u$ by $\mu(d_i)$. This is denoted by : $t \text{–N–>}_{[u,\ i,\ \mu]}\ t'$

Let us recall a theorem that gives a non deterministic algorithm to compute a complete set of E-unifiers.

Theorem [Hullot 80] : Given an equational theory defined by a canonical term rewriting system $\mathfrak{R}$. Let t and t' be two terms, H a new function symbol (H $\notin \Omega$), and V be a set of variables containing those occurring in t and t'. Let S be the set of substitutions σ such that σ is in S iff there exists a -N->_derivation :

$$M_0 = H(t, t') \text{-N->}_{[u0,\ i0,\ \sigma 0]} M_1 = H(t_1, t'_1) \text{-N->}_{[u1,\ i1,\ \sigma 1]} \cdots \text{-N->}_{[u(n-1),\ i(n-1),\ \sigma(n-1)]} M_n = H(t_n, t'_n)$$

such that t_n and t'_n are unifiable. θ_n is normalized (with $\theta_{i+1} = \sigma_i\theta_i$ and $\theta_0 = \varepsilon$) and $\sigma = \mu\theta_n$ where μ is the most general unifier of t_n and t'_n. S is a complete set of E-unifiers of t and t' away from V.

To implement this algorithm, we have chosen depth-first strategy for its efficiency. Terms are always normalized. When applying one step of narrowing relation, equations are considered in their order of declaration and the subterm chosen to be rewritten, if it exists, is the innermost and leftmost one as it is mentioned in [Fribourg 84] and [Jouannaud et al. 83]. Furthermore, we do not deduce an E-unifier as soon as t_n and t'_n are unifiable but only when in addition $H(t_n, t'_n)$ is not narrowable. So, the set S of E-unifiers computed does not always satisfy the condition : $\forall\ \sigma \in U(t, t', E),\ \exists\ \sigma' \in S, \sigma' \leq_E \sigma$; but when σ is ground this condition is satisfied.

Moreover, when E is such that for every operator f of arity $s_1, \ldots, s_n$ and every n-uple of ground terms $(t_1, \ldots, t_n)$, there exists at most one equation whose left hand side matches $f(t_1, \ldots, t_n)$, then S satisfies the further condition : $\forall\ \sigma, \sigma' \in S, \sigma \leq_E \sigma' \Rightarrow \sigma = \sigma'$.

Another implementation of this algorithm is presented in [Rety et al. 85].

Example 11 : A query is followed by a question mark. Variables must be declared, like within a program unit. After each solution the user may type a carriage return to obtain another solution if it exists, otherwise, he may type "no" to stop the process. Here we list all the solutions given by the algorithm.

```
variables
   i, j, cats, birds, heads : nat
end
i + 1 == 3 + j ?
for all j : nat     (solution 1)
i == s(s(j))
.

cats_birds (cats, birds, heads, 6) ?
cats == 0          (solution 1)
birds == 3
heads == 3

cats == 1          (solution 2)
birds == 1
heads == 2
.

ordered?[<=, =] ([0, i, 2]) ?
i == 0             (solution 1)
i == 1             (solution 2)
i == 2             (solution 3)
.

ordered?[>=, =]([0, i, 2]) ?
.                  (no solution)

ordered?[>=, =] ([3, 1, 1]) ?
ok
.

reflex[graph[alpha[s]]]([0, i], [j, s(s(0))]) ?
i == s(0)          (solution 1)
```

j == s(0)

i == s(s(0)) (solution 2)
j == 0
.

4–THE INTERPRETER

One may wish to use specification language as a programming language for rapid prototyping and therefore programs must run with a certain efficiency. For this reason, we have written an interpreter for a subset of the equations which reduces very quickly a ground term to its normal form.

The equations considered by the interpreter are syntactically defined with a "==>" symbol instead of the "==" symbol. They must meet the conditions :

(1) the root of the left hand side is the operator defined by the equation ;

(2) the operands of the root operator are composed only of constructors and variables (in the left hand side) ;

(3) all the variables in the right hand side must occur in the left hand side.

These equations are called "runable" equations. The system LPG translates every runable equation into a pair (f, c), where f is a filter corresponding to the left hand side, and c is a text in an intermediary code (I_code), corresponding to the right hand side. Interpretation is performed by a stack machine which is essentially composed of four stacks :

–	the value stack	(v_st)
–	the parameter stack	(p_st)
–	the generic context stack	(c_st)
–	the return address stack	(r_st)

The values are represented in a heap by n-ary trees, refered from v_st and p_st. A node contains the code for a constructor, or is a single value for the predefined values : nat, string, ... ; the children of a node n refer to the operands of the constructor assigned in n. A filter is represented like a value, except that, in this case, a leaf may be a variable. The filter algorithm takes a pair (f, v) where f is a filter and v a value, and if succeeds, initializes the parameter stack for the evaluation of the right hand side associated to the filter f. Some particular cases must be added to deal with predefined values (such as integers), but we do not detail them here. Let us notice that the interpreter never performs type checking, because that was made during parsing ; at the run time, operators are sure of the right typing of their operands. It is an advantage of the strongly typed languages.

As usual, r_st contains the return addresses ; the address of the top is the next statement to execute, after the evaluation of the I_code of the current operator. The c_st stack refers also to the I_code, in the following way : each element of c_st indicates the beginning of the representation of an effective operator in the I_code.

For example, the piece of the I_code for the right hand side of the equation "alpha(a <+ s) ==> f(a) <+ alpha(s)" is only :

<u>statement</u> :	<u>operand</u> :	<u>comment</u> :
value_of_var	1	value of a
formal_call	0	call of f
value_of_var	0	value of s
call	index(alpha)	call of alpha
cartesian_prod	2	
call	index(<+)	call of <+
pop_p_st	2	pop a and s from p_st
return		

Compiling the expression "alpha[alpha[+]](x)" where x is a variable of type seq[seq[(nat, nat)]] gives the I_code :

```
        value_of_var         index(x)              value of x
        jump_to              E4
E1      jump_to              E3                    beginning of alpha[+]
E2      call                 index(+)              beginning of +
        return                                     end of +
E3      push_c_st            E2                    install the generic context of alpha
        call                 index(alpha)
        pop_c_st             1                     remove the generic context
        return                                     end of alpha[+]
E4      push_c_st            E1                    install the generic context
        call                 index(alpha)          call of the outermost alpha
        pop_c_st             1                     remove the generic context
```

Operations on a stack st are , in a Pascal-like language :

```
    procedure push (v : value ; var st : stack) ;
    procedure pop (i : integer ; var st : stack) ;      <*  to pop i elements from the stack  *>
    function top (i : integer ; st : stack) : value ;   <*  to take the ith value under the top  *>
```

Let C be the label of the statement to be executed, the interpretation procedure IP is described by :

```
    (1)   IP(jump_to, i)           :  C := i ;
    (2)   IP(return)               :  C := top(0, r_st) ; pop(1, r_st) ;
    (3)   IP(pop_p_st, i)          :  pop(i, p_st) ; C := C + 1 ;
    (4)   IP(push_c_st, i)         :  push(i, c_st) ; C := C + 1 ;
    (5)   IP(pop_c_st, i)          :  pop(i, c_st) ; C := C + 1 ;
    (6)   IP(value_of_var, i)      :  push(top(i, p_st), v_st) ; C := C + 1 ;
    (7)   IP(formal_call, i)       :  push(C+1, r_st) ; C := top(i, c_st) ;
    (8)   IP(cartesian_prod, i)    :  form the cartesian product cp of the i values at the top of v_st ;
                                      pop(i, v_st) ; push(cp, v_st) ; C := C + 1 ;
    (9)   IP(call, i)              :  push(C+1, r_st) ;
                                      if i is a constructor :
                                             form the value v with a node i and the value top(0, v_st) for arguments,
                                             if any ; pop(1, v_st) ; push(v, v_st) ; pop(1, r_st) ; C := C+1 ;
                                      else   if i is an operator with parameters :
                                             let (f, c) be the first runable equation of i such that filter(f, top(0, v_st))
                                             succeeds, then C:= c ;
                                             otherwise the axiomatization of the operator coded by i is not
                                             sufficiently complete ;
                                      else   <*  i is an operator without parameter *>
                                             let(ø, c) be a runable equation for i then C := c ;
                                             othewise no complete axiomatization.
```

We have just shown a sketch of the interpreter. The complete LPG provides precondition and exception facilities. Moreover some built-in input-output functions may be called in the runable equations. Experiments with LPG as a functional programming language have been very pleasant and easy. The important fact is that "compiling" an instance of a generic operator consists only in passing operators as actual parameters –the generic context– to execute the code of the operator equations. Two different instances of the same operator are compiled in two different bindings of the generic parameters for the same piece of code.

Example 12 : An invocation of the interpreter is made by typing a ground term followed by the "==>" symbol. The symbol " ^ " stands for the last value (term) computed by the interpreter :

```
[[1, 2], [3]] = [[1, 2], [3]] ==>                     (implicit instantiation of "=")
true                                                  (result)

sort[<=, =] ([6, 4, 2, 3]) ==>                        (explicit instantiation of the operator "sort")
[2, 3, 4, 6]

alpha[rev_iota] (^) ==>
[[2, 1], [3, 2, 1], [4, 3, 2, 1], [6, 5, 4, 3, 2, 1]]

hom[nil, +] (^) ==>                                   ("+" appends two sequences. We suppose this operator defined elsewhere)
[2, 1, 3, 2, 1, 4, 3, 2, 1, 6, 5, 4, 3, 2, 1]

hom[1, *] (^) ==>
207360
```

5-CONCLUSION AND FUTURE WORK

We have presented the broad outlines of a project designed for generic specification and programming. In this language, one may define classes of Σ-structures (properties), particular algebras (abstract data types) and families of algebras (generic data types and enrichments). An E-unification algorithm has been implemented in order to solve equation systems and to evaluate predicates, thus generalizing ordinary logic programming. The characteristics of the instantiation of sorts and operators, as well as the declaration of runable equations make it an applicative programming language. A few other useful features already implemented, but not presented here, are raising and handling exceptions, and partial binding of the arguments of an operator (curryfication). These features together with the user defined "functional forms" like "alpha" and "hom", make LPG a very powerful programming tool.

Some verifications such as the validity of model declarations and theory morphism declarations ($\bar{\Phi}(\mathrm{mod}(\Sigma', C') \subseteq \mathrm{Mod}(\Sigma, C))$ are not performed in the present implementation, nor is it checked that enrichments preserve the hierarchy. On the other hand, one often would like to prove theorems in a given specification. So, it seems worthwhile to give the user these facilities ; that is why we are building an environment for specification analysis. This environment will include the interpreter and the "resolver" presented here as well as other tools such as a completion algorithm, a system for transformation of programs [Bert and Bensalem 85], a theorem-prover (the connection with the demonstrator OASIS [Barberye et al. 85] is in the process of being done), and so on.

6-ACKNOWLEDGEMENTS

Special thanks are due to P. DRABIK for his participation in implementing the LPG system. We are also grateful to J.M. HUFFLEN for his assistance in typing a first draft.

7-REFERENCES

[Backus 78] J. Backus : Can Programming be Liberated From Von Neumann Style ? A functional style of algebra of programs. CACM, vol 21, no. 8, pp. 613-641, August 1978.

[Barberye et al. 83] G. Barberye, T. Joubert, M. Martin, M. Mouffron, E. Paul : Manuel OASIS. Note technique CNET, NT/PAA/CLC/LSC/959, 1983.

[Bert 83] D. Bert : Refinements of Generic Specifications With Algebraic Tools. Proceedings of the IFIP 9th World Computer Congress, pp. 815-820, Paris, September 1983.

[Bert and Bensalem 85] D. Bert, S. Bensalem : Algebra of Strongly Typed Functional Programs. RR. IMAG-561-LIFIA-33, Grenoble, 1985.

[Burstall and Goguen 77] R. M. Burstall, J. A. Goguen : Putting theories together to make specifications. Proceedings of 5th International Joint Conference on Artificial Intelligence, pp. 1045-1058, Cambrige (Massachussets), 1977.

[Burstall and Goguen 80] R. M. Burstall, J. A. Goguen : The Semantics of CLEAR, a specification language. Proceedings of Advanced Course on Abstract Software Specification, LNCS, no. 86, pp. 292-332, Copenhagen, 1980.

[Ehrig et al. 84] H. Ehrig, H. J. Kreowski, J. W. Thatcher, E. W. Wagner, J. Wright : Parameter Passing in Algebraic Specification Languages. Theoretical Computer Science, no. 28, pp. 45-81, North-Holland, 1984.

[Fribourg 84] L. Fribourg : Handling Function Definitions Through Innermost Superposition and Rewriting. LITP, Rapport no. 84-69, Paris, 1984.

[Futatsugi et al. 84] K. Futatsugi, J. A. Goguen, J-P. Jouannaud, J. Meseguer : Principles of OBJ2. CRIN, Rapport no. 84-R-066, Nancy, 1984.

[Goguen et al. 78] J. A. Goguen, J. W. Thatcher, E. W Wagner : An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data types. Current Trends in Programming Methodology, vol. 4 : Data Structuring, chap. 5, Prentice Hall, 1978.

[Goguen and Burstall 84] J. A. Goguen, R. M. Burstall : Introducing Institutions. Proceedings Logics and Programming Workshop, pp. 221-256, 1984.

[Goguen and Meseguer 82] J. A. Goguen, J. Meseguer : Universal Realisation, Persistent Interconnection and Implementation of Abstract Modules. 9th Colloquium on Automata, Languages and Programming, LNCS, no 140, pp. 265-281, 1982.

[Goguen and Meseguer 84] J. A. Goguen, J. Meseguer : Equality, Types, Modules and Generics for Logic Programming. Proceedings of International Conference on Logic Programming, Uppsala, 1984.

[Guttag 78] J. V. Guttag : The Algebraic Specification of Abstract data types. Acta Informatica, no. 10, 1978.

[Hullot 80] J-M. Hullot : Canonical Forms and Unification. Proceedings 5th Conference on Automated Deduction. LNCS, no. 87, pp. 318-334, 1980.

[Jouannaud et al. 83] J-P. Jouannaud, C. Kirchner, H. Kirchner : Incremental Constructions of Unification Algorithms in Equational Theories. Automata, Languages and Programming, pp. 361-375, Barcelona, 1983.

[Rety et al. 85] P. Rety, C. Kirchner, H. Kirchner, P. Lescanne : Narrower : a new algorithm for unification and its application to logic programming. First International Conference on Rewriting Techniques and Applications. Dijon, 1985.

[Robinson 65] J. A. Robinson : A Machine-Oriented Logic Based on the Resolution Principle. JACM, vol. 12, no. 1, pp. 23-41, 1965.

[Thatcher et al. 82] J. W. Thatcher, E. W. Wagner, J. B. Wright : Data Type Specification : Parameterization and the Power of Specification Techniques. ACM TOPLAS, vol. 4, no. 4, pp. 711-732, 1982.

[Van Emden and Kowalski 76] M. H. Van Emden, R. A. Kowalski : The semantics of Predicate Logic as a Programming Language. JACM, vol. 23, no. 4, pp. 733-742, 1976.