Integrating Disequations in the Algebraic and Logic Programming Language LPG

D. Bert and R. Echahed
LGI-IMAG, BP 53, 38041 Grenoble cedex 9, France
Phone: +33 76 51 42 69 Fax: +33 76 51 33 79
Email: {bert,echahed}@imag.fr

Abstract : This paper gives a brief description of the recent improvements of the algebraic and logic programming language LPG. We mainly introduce the possibility to use disequations (negation of equations), which involve user's definable functions, in the definition of predicates. Then we describe the operational semantics of LPG which is based on an extension of $SLD-$resolution in order to deal with equations and disequations. Also, we give new sufficient conditions to ensure the completeness of narrowing strategies, to solve equations and disequations, in the presence of *conditional* term rewrite systems.

1 Introduction

During the last few years, many languages have been proposed to combine functional and logic programming paradigms, see for instance [13] for a recent survey. LPG (for *Langage de Programmation Générique*)[2] is one of these proposals. The LPG language has been designed and implemented in 1985-86 at the IMAG Institute. It is a generic, logic and functional programming language intended to experiment new concepts in the field of specification, prototyping and high-level programming languages. LPG is based on concepts of algebraic specifications, abstract data types and logic programming. Thus, with respect to functional languages, LPG offers the possibility to define predicates or relations and to solve goals that include defined functions and logical variables. With respect to classical logic programming languages, LPG is generic, i.e. it offers the possibility to define predicates that are parameterized by other predicates or functions. This feature allows to write "higher-order like" programs such as the transitive closure of a relation given as parameter. It is strongly typed and allows user's definable functions.

In this paper, we give an overview of the main features of LPG and describe its operational semantics. Section 2 is devoted to remind general background about algebraic specifications and rewrite systems, which are useful in the sequel. In section 3, we present an short introduction of the language and we focus on three points : *(1)* the way of defining functions and predicates by means of rewrite rules and Horn clauses extended by disequations. *(2)* Modularity primitives, structuring and specification building operations, such as generic modules and instantiation. *(3)* Facilities provided by the language to use generic objects. Section 4 presents the operational semantics of LPG, that is to say the principle of resolution of logical goals and the inference rules used by LPG to "solve" equations and disequations. At last, we give some insights into the possible future directions to improve the system.

2 Preliminaries

We recall some important notions and notations. See [7, 15] for the missing definitions.

A many-sorted first order *signature* Σ is a tuple (S, Ω, Π) where S is a set of *sorts*, Ω is an S^*-indexed family of *operation* sets, and Π is an S^+-indexed family of *predicate* sets. For every sort $s \in S$, every set Π_{ss} includes the symbols of equality (`==`) and disequality (`=/=`). Let $\mathcal{X} = (\mathcal{X}_s | s \in S)$ be an S-sorted, countably infinite set of *variables*. We denote by $\mathcal{T}(\Sigma, \mathcal{X})_s$ the set of *terms* of sort s built from Σ and $\mathcal{X}$. $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of all terms, i.e., the union $\cup_{s \in S} \mathcal{T}(\Sigma, \mathcal{X})_s$. By $\mathcal{V}ar(t)$ we denote the set of variables occurring in a term t. A term t is called *ground term* if $\mathcal{V}ar(t) = \emptyset$. A term is called *linear* if it does not contain multiple occurrences of one variable. We denote by $Pos(t)$ the set of positions of subterms of t. $Pos^*(t)$ is a subset of $Pos(t)$ that includes the positions of non variable subterms of t. If $p \in Pos(t)$ then $t_{|p}$ is the subterm of t at position p and $t[s]_p$ is the term which is obtained by replacing the subterm at position p by the term s. An *equation* (resp. a *disequation*) is a pair denoted by t`==`t' (resp. t`=/=`t') where t and t' are terms. A specification is a tuple $SP = (\Sigma, E, Cl)$ where Σ is a first order signature, E is a set of conditional equations of the form l`==`r `<==` t_1`==`$t'_1, \ldots, t_n$`==`t'_n, and Cl is a set of Horn clauses of the form $P(t_1, \ldots, t_n)$ `<==` $A_1, \ldots, A_m$ where $P \in \Pi - \{$`==`,`=/=`$\}$ and each A_i may be an equation, a disequation or any other atom formed by predicates in Π.

Let SP be a specification. We denote by $Mod(SP)$, the class of all models of SP. This class admits an initial model (up to isomorphism), also called *standard model* and noted SM_{SP}. Note that the existence of SM_{SP} is not guaranteed if we use disequations in the premises of conditional equations. Indeed, if we consider the conditional equation $f(x)$`==2 <==` $f(x)$`=/=1`, it is clear that there are models of this equation that are not comparable (by homomorphism). We denote by $=^i_E$ the inductive congruence generated by E over $\mathcal{T}(\Sigma, \mathcal{X})$ for any variable set $\mathcal{X}$. $\mathcal{T}(\Sigma)/E$ denotes the quotient term algebra. The relation $\neq^i_E$ is defined to be exactly the negation of $=^i_E$.

A *substitution* is a mapping $\sigma: \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ with $\sigma(x) \in \mathcal{T}(\Sigma, \mathcal{X})_s$ for all variables $x \in \mathcal{X}_s$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. Sometimes we will identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. Substitutions are extended to morphisms on $\mathcal{T}(\Sigma, \mathcal{X})$ by $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$ for every term $f(t_1, \ldots, t_n)$. A substitution σ is called *ground substitution* if $\sigma(x)$ is a ground term for all $x \in \mathcal{D}om(\sigma)$. The *composition of two substitutions* σ and τ is defined by $(\sigma \circ \tau)(x) = \sigma(\tau(x))$ for all $x \in \mathcal{X}$. We also note the composition by $\sigma\tau$. We will note the identity substitution by id (for each x we have $id(x) = x$). The *restriction* $\sigma_{|V}$ of a substitution σ to a set V of variables is defined by $\sigma_{|V}(x) = \sigma(x)$ if $x \in V$ and $\sigma_{|V}(x) = x$ if $x \notin V$. If V is a set of variables, we write $\sigma = \sigma'[V]$ iff $\sigma_{|V} = \sigma'_{|V}$. $\leq$ denotes the usual preorder on substitutions: $\sigma \leq \sigma'[V]$ iff there exists β such that $\beta \circ \sigma = \sigma'[V]$. This preorder is extended to the equational theory $=^i_E$ as follows: $\sigma \leq^i_E \sigma'[V]$ iff $\exists \rho, \rho\sigma =^i_E \sigma'[V]$, where $\sigma =^i_E \sigma'[V]$ iff $\forall x \in V, \sigma(x) =^i_E \sigma'(x)$.

A *conditional rewrite rule* is a directed conditional equation of the form $l \to r$ `<==` t_1`==`$t'_1, \ldots, t_n$`==`t'_n such that l is not a variable, $\forall i \in \{1,..,n\}, \mathcal{V}ar(t_i$`==`$t'_i) \subseteq \mathcal{V}ar(l)$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A *variant* of a rewrite rule r is obtained from r by a variable renaming. A conditional term rewrite system (CTRS for short) is a set of conditional rewrite rules. The rewrite relation associated with a $CTRS$ R is obtained by interpreting the equality signs in the conditional part of a rewrite rule as joinability. That is to say, $t \rightarrow_R t'$ if there exists a position $p \in Pos^*(s)$, a variant $l \rightarrow r$ `<==` c of a rewrite rule in R and a substitution σ such that $t_{|p} = \sigma(l)$, $t' = t[\sigma(r)]_p$ and $\sigma(u) \downarrow_R \sigma(u')$ for every equation u==u' in c. The joinability of two terms t and t', denoted by $t \downarrow t'$, means that there exists a term u such that $t \rightarrow_R^* u$ and $t' \rightarrow_R^* u$ where $\rightarrow_R^*$ denotes the reflexive and transitive closure of $\rightarrow_R$. The relation $\rightarrow_R$ is terminating if there is no infinite derivation $t \rightarrow_R t_1 \rightarrow_R \ldots$. The relation $\rightarrow_R$ is confluent if $\forall t_1, t_2, t_3, t_1 \rightarrow_R^* t_2$ and $t_1 \rightarrow_R^* t_3 \Rightarrow \exists t_4 \mid t_2 \rightarrow_R^* t_4$ and $t_3 \rightarrow_R^* t_4)$.

A *goal* is a conjunction of atoms $A_1, \ldots, A_m$. Let R be a $CTRS$. A goal G (conditionally) narrows into a goal H if there exists an equation or disequation $e \in G$, a position $p \in Pos^*(e)$, a variant $l \rightarrow r$ `<==` c of a conditional rewrite rule in R and a substitution σ such that σ is the most general unifier of $e_{|p}$ and l, and $H = \sigma(G - \{e\} \cup \{e[r]_p\} \cup c)$. We write $G \rightsquigarrow_{[e,p,l \rightarrow r \texttt{ <== } c,\sigma]} H$ or $G \rightsquigarrow_\sigma H$.

Let G be a goal, a substitution σ is a *solution* of G iff $\sigma(G)$ is valid in SM_{SP}. A set S of substitutions is called *complete set of solutions* of G iff, in addition to some technical conditions on variables we are omitting here, every substitution σ in S is a solution of G (soundness) and for every ground solution of G, say θ, there exists a substitution β in S such that $\beta \leq_E^i \theta[\mathcal{V}ar(G)]$ (completeness).

A signature $\Sigma = (S, \Omega, \Pi)$ is called *constructor-based* if the set of operations is split into two disjoint sets C and D. C is the set of constructors and D is the set of defined operations. A specifications SP is constructor-based if its signature is. A term t is called *innermost* if it is of the form $f(t_1, \ldots, t_n)$ such that $f \in D$ and every t_i does not contain any operation in D.

3 The Language

3.1 Basic modules and relations

In LPG, specifications are decomposed into a collection of reusable modules. Semantic conditions ensure the independence of the modules. There are three kinds of specification modules, namely *properties*, *types* and *enrichments*. All the modules can be composed of a signature part and an axiomatization part. The signature part is a first order signature $\Sigma = (S, \Omega, \Pi)$. The axiomatization of operators is achieved by conditional equations (or rather conditional rewrite rules) whereas the predicates semantics is provided by Horn clauses, where the bodies of the clauses may include classical literals, equations (`t == t'`) and/or disequations (`t =/= t'`). The axiomatization is structured in two layers: the algebraic axiomatization of the operators, like in algebraic specification languages, without any reference to predicate symbols, and a second layer for the axiomatization of the predicates which can mix all kinds of logical atoms.

Modules of "types" are designed to define abstract data types. In these modules, the list of operators may be divided into *constructors* and *defined operators*. This means that the term algebra of the data type is generated by constructors only and that terms with other operators do have a meaning expressed solely with constructors. To weaken this constraint LPG allows to declare that some terms are "exception-terms" and have got no meaning[1]. So

[1]The preconditions and exceptions mechanism of LPG will not be presented in this paper.

```
type nat-dot
sorts
   dot, line
constructors
   c-dot:  (nat,nat) → dot
   c-line :  (dot,dot) → line
operators
   dist :  (nat,nat) → nat
   sign :  line → bool
variables
   i,j,k,l :  nat
equations
   1 :  dist(i,j) ==> (i - j) + (j - i)
   2 :  sign(c-line(c-dot(i,j),c-dot(k,l))) ==> (i <= k and j <= l) or
                                                (k <= i and l <= j)
end nat-dot
```

Figure 1: Data type of dots and lines.

type specifications must contain declarations of new sorts, constructors and possibly operators and predicates. Modules of "enrichments" are used to declare only defined operators or predicates on types already declared. The need of separating some operators or predicates from the declaration of their sort "of interest" appears when parameterization is introduced. As examples, look at the Figure 1 defining the sorts `dot` of dots with natural coordinates and `line` of straight lines. For that we assume the data types `NAT` with a sort `nat` and usual operators on natural numbers as well as `BOOL` with a sort `bool` and boolean operators are already defined in the current context of modules. Actually, these data types belong to the predefined library of LPG. We define a function `dist` that computes a "distance" between two naturals and the function `sign` that evaluates to `true` whenever the considered line has a positive slope. Then, several predicates on `line` are defined in an *enrichment* module (see Figure 2). A clause declaration is written as "`Q(...) <==` $atom_1$`,...,`$atom_k$". In the equations of defined operators the equality symbol can be written as "`==>`" to mean that these axioms have to be used as rewriting rules from left to right, for tools like the symbolic evaluator or the solver of goals. The left hand sides of such oriented rules must be innermost terms.

Modules of "properties" are designed to define equational theories which characterize classes of types. For example, in programming, usual theories are the characterization of any set of values (`Any`) or an equivalence relation (Figure 3). The property `Equality` uses the sort `bool`[2].

A specific feature of LPG is the definition of relations between properties : the relation "P_2 satisfies P_1" means that the signature of P_1 is included in P_2 and the axioms of P_1 hold in P_2, via the renaming defined by the relation. The relation "P_2 inherits P_1" means that

[2]By convention in LPG, this property is used to define an operator which is supposed to give the observational equality on the elements of an abstract data type. So, it is called "Equality" although the right first order axiomatization of equality can be done only for each data type because of the congruence axioms.

```
enrichment on-nat-dot
predicates
   vertical, horizontal :  line
   not-parallel :  (line,line)
variables
   i,j,k,l,m,n,o,p :  nat
   a,b :  line
clauses
   1:  vertical(c-line(c-dot(i,j),c-dot(i,k))) <== k =/= j
   2:  horizontal(c-line(c-dot(i,j),c-dot(k,j))) <== i =/= k
   3:  not-parallel(a,b) <== sign(a) =/= sign(b)
   4:  not-parallel(c-line(c-dot(i,j),c-dot(k,l)),c-line(c-dot(m,n),c-dot(o,p)))
                           <== dist(i,k) * dist(m,o) =/= dist(j,l) * dist(n,p)
end on-nat-dot
```

Figure 2: Predicates on lines.

```
property Any
sorts t
end Any
```

```
property Equality
sorts t
operators
   =:  (t,t) → bool
variables
   x,y,z:  t
equations
   1:  x = x == true
   2:  x = y == y = x
   3:  (x = y and y = z) ⇒ (x = z) == true
satisfies
   Any [t]
end Equality
```

Figure 3: Properties of any data type and any data type with equal operator.

```
property Partial-Order
sorts s
operators
   ≤,=:  (s,s) → bool
variables
   x,y,z:  s
equations
   1:  x ≤ x == true
   2:  x ≤ y and y ≤ x ⇒ x = y == true
   3:  x ≤ y and y ≤ z ⇒ x ≤ z == true
satisfies
   Equality [s operators =]
end Partial-Order
```

```
property Total-Order
sorts s
operators
   ≤,=:  (s,s) → bool
variables
   x,y:  s
equations
   1:  x ≤ y or y ≤ x == true
inherits
   Partial-Order [s operators ≤,=]
end Total-Order
```

Figure 4:

the signature of P_1 is included in P_2, as previously, and the axioms of P_1 are added in the property P_2. At last, the relation "P combines $P_1, P_2, \ldots$" means that all the properties P_i are inherited in P, and P is exactly the "union" of these properties. An example is the definition of the property of an associative and commutative operator by combination of the property of a commutative operator and the property of an associative operator. In Figure 4, we present the properties of partial and total order. Notice that property relations can associate sorts, operators and predicates with different names (e.g. `t` $\mapsto$ `s`, in `Partial-Order` satisfies `Equality`, Figure 4).

Property modules are used to specify generic parameters of generic types or enrichments. If a generic type/enrichment is parameterized by a property P, then it can be instantiated by any "model" of P. The generic parameter is introduced by the keyword "**requires**". See the specification `Sorting` in Figure 5 which requires a data type with a total order to define the generic *sort* operator and the generic predicate *ordered* on sequences.

At last, it is often interesting to specify that a data specification is a model of a property, thus giving a better information on the data being specified. This is the idea of the "views" in the OBJ language [12] and also in recent languages for computer algebra. A simple application of this method is to specify that some operators are commutative, or associative, or both, if they satisfy the adequate property. Another example is the specification `Eq-Seq` which contains a model of the property `Equality` with the sort (or rather the elements of sort) `seq[elem]` and the operator `=` on sequences. The syntactic form of model declarations in LPG is shown in figure 5.

3.2 Using generic sorts, operators and predicates

In LPG, a library is a set of specification modules, like properties, types and enrichments, together with the relations between properties, declarations of models and `requires` relations. Like it is shown in the examples, importation as well as instantiation of generic modules is *implicit* in the language. So, LPG provides mechanisms and notations to refer to declarations of instantiated modules. The basic mechanism for creating an instantiation of a generic module

```
enrichment Eq-Seq
requires Equality [elem operators eq]
operators
   =:  (seq[elem],seq[elem] → bool
variables
   x,y:  elem
   s,u:  seq[elem]
equations
   1:  nil = nil ==> true
   2:  cons(x,s) = nil ==> false
   3:  nil = cons(x,s) ==> false
   4:  cons(x,s) = cons(y,u) ==>
                       eq(x,y) and s = u
models
   Eq_seq:  Equality[seq[elem] operators =]
end Eq-Seq
```

```
enrichment Sorting
requires Total-Order [elem operators ≤,eq]
operators
   sort:  seq[elem] → [elem]
   add:  (elem,seq[elem]) → seq[elem]
predicates
   ordered :  seq[elem]
variables
   x,y:  elem
   s:  seq[elem]
equations
   1:  sort(nil) ==> nil
   2:  sort(cons(x,s)) ==> add(x,sort(s))
   3:  add(x,nil) ==> [x]
   4:  add(x,cons(y,s)) ==> if x≤y
                            then cons(x,cons(y,s))
                            else cons(y,add(x,s))
                            endif
clauses
   1:  ordered(s) <== sort(s) == s
end Sorting
```

Figure 5: Definition of equality and sorting of generic sequences

D is produced by using the objects (sorts, operators and predicates) declared in D with a model of the required property of D. In many cases, the models declared in the library are sufficient to determine what is the instance of an object. For example, if the library contains the declaration of the model :

```
Nat-Order:  Total-Order[nat operators ≤,=]
```

then a term like "`sort.Nat-Order([2,4,1])`" is the call of the operator to sort the sequence `[2,4,1]` using the order "≤". A shorter form of this call is "`sort[≤,=]([2,4,1])`" because the type checking inference rules of LPG allow to determine the sort of the model from the sort of the argument (`sequence of nat`). At last, if the model `Nat-Order` is the unique model of the property `Total-Order` for the sort `nat`, then the notation "`sort([2,4,1])`" is not ambiguous and corresponds again to the same call. This method of determination of the instances of operators can be applied to the predicates as well. The second form of instantiation (`sort[≤,=]([2,4,1])`) is allowed if any suitable model for instantiation is not declared. In this case, LPG is able to build a model on the fly, from the type checking of the arguments and of operators (or predicates) given between brackets, without taking into account the declarations of the current library. So, a call of the form `sort[≥,=]([2,4,1])` is the sorting of the sequence parameter with the order ≥, even if the model `Nat-Order` is declared. Obviously, the LPG system checks that the effective generic parameters are syntactically consistent with the required property (checking of the profiles of the effective operators and predicates with respect to the effective sorts).

For the instantiation of generic sorts, one can use the first form (`s.m` where `m` is a model of the required property of the module of the declaration of `s`), or the form `s[s1,s2,...]`, where `s1,s2,...` is the list of effective sorts corresponding to the sorts of the instantiated model. The complete model must be deduced by LPG from these sorts and the models declared in the library, like in the third form of instantiation of operators and predicates. This mechanism is very powerful because it can be used (finitely) recursively like in the expression `[[[1,2]]] = [[[1],[2]]]` (which returns the result `false`), where the operator has the profile `= : seq[seq[seq[nat]]],seq[seq[seq[nat]]])` $\rightarrow$ `bool`, and causes three implicit instantiations of the module `Eq-Seq`, each time with different models, that are :

```
Eq-seq:  Equality[seq[seq[nat]] operators =]
Eq-seq:  Equality[seq[nat] operators =]
Eq-nat:  Equality[nat operators =]
```

The two first models are two instances of the models declared in the module `Eq-Seq`. The last model is either declared as such in the current library (the `=` operator has the profile `(nat,nat)` $\rightarrow$ `bool`, or is deduced from the model `Nat-Order` and the properties inclusion :

```
Equality ⟶ Partial-Order ⟶ Total-Order
```

The theory that allows to deduce models from the declared ones is based upon the transitivity of composition of some arrows like inclusion of properties or model declarations. We do not detail this mechanism here. For more explanations, see the reference manual of LPG [5], or [6].

To conclude this section, we present an example of definitions of generic predicates which can be seen as higher-order predicates or meta-operators on predicates. They are the definition of the graph of a function, and definitions of union, intersection, inverse, etc. of binary relations, in Figure 6.

Now, assume that the relation `id0` is simply the pair $\{$`(0,0)`$\}$, then the "predicate expression" `union[id0,inverse[graph[succ]]]` is the relation of the graph of the function `pred` on natural numbers (with `pred(0) == 0`). Another example of the use of higher-order predicates, is the definition of classical family relationships. Given the following relations on data type `person` :

```
husband:  (person,person)
mother:  (person,person)
```

with an obvious meaning[3], then these other relations can be defined as instances of generic predicates :

```
wife(x,y) <== inverse[husband](x,y)
spouse(x,y) <== union[husband,wife](x,y)
father(x,y) <== compose[mother,husband](x,y)
parents(x,y) <== union[father,mother](x,y)
grand-parents(x,y) <== compose[parents,parents](x,y)
children(x,y) <== inverse[parents](x,y)
```

[3] The associations `husband` and `mother` could be defined as functions and the relation should be build by means of the predicate `graph` without any problem. This illustrates very well the integration paradigm.

```
property Function
sorts t1,t2
operators            -- characterization of any
   f:  t1 → t2      -- function from t1 to t2.
satisfies
   Any[t1], Any[t2]
end Function
```

```
enrichment For-Graph
   requires Function [t1,t2 operators f]
predicates            -- definition of the graph
   graph:  (t1,t2)  -- of the function f.
variables
   x :  t
clauses
   1:  graph(x,f(x))
end For-Graph
```

```
property Bin-Relation
sorts t
predicates       -- characterization of any
   r:  (t,t)    -- binary relation on t.
satisfies
   Any[t]
end Bin-Relation
```

```
enrichment Relation-Calculus1
   requires Bin-Relation [t predicates r]
predicates
   reflex:  (t,t)       -- reflexive relation containing r.
   not-reflex:  (t,t)   -- relation r without identity.
   inverse:  (t,t)      -- inverse of the relation r.
variables
   x,y :  t
clauses
   1:  reflex(x,x)
   2:  reflex(x,y) <== r(x,y)
   3:  not-reflex(x,y) <== r(x,y), x =/= y
   4:  inverse(x,y) <== r(y,x)
end Relation-Calculus1
```

```
property Bin--Rel2
sorts
predicates            -- characterization of two
   r1,r2:  (t,t)    -- binary relations on t.
combines
   Bin-Relation [t predicates r1],
   Bin-Relation [t predicates r2],
end Bin-Rel2
```

```
enrichment Relation-Calculus2
   requires Bin-Rel2 [t predicates r1,r2]
predicates
   union:  (t,t)       -- relation r1 or r2.
   inter:  (t,t)       -- relation r1 and r2.
   compose:  (t,t)     -- composition r1;r2.
variables
   x,y :  t
clauses
   1:  union(x,y) <== r1(x,y)
   2:  union(x,y) <== r2(x,y)
   3:  inter(x,y) <== r1(x,y), r2(x,y)
   4:  compose(x,y) <== r1(x,z), r2(z,y)
end Relation-Calculus2
```

Figure 6: Definition of higher-order predicates

```
brother-or-sister(x,y) <== not-reflex[compose[parents,children]](x,y)
brother-or-sister-in-law(x,y) <==
          union[union[compose[brother-or-sister,spouse]
                    ,compose[spouse,brother-or-sister]]
               ,compose[spouse,compose[brother-or-sister,spouse]]](x,y)
```

3.3 Flattening instantiated specifications

From a semantical point of view, a completely instantiated specification with importations, instantiations, etc. is equivalent to a single flat specification defining a finite set of sorts, operators and predicates. So, tools like the interpreter of closed terms, or the solver of goals can be defined without taking into account generic specifications. Evaluations or resolutions are always performed in a term algebra, which is the *initial* model of the flattened specification. The semantics of modules is defined in such a way results of instantiations and importations give the same model as this one obtained after flattening. In an evaluation session, terms written for the interpreter or the solver must be completely instantiated. So, they refer to a virtual specification containing all the (instantiated) sorts, operators and predicates used in the terms, all the operators and predicates used in the rewrite rules and clauses defining these operators and predicates, together with copies of all the needed instantiated equations and clauses. Actually for the interpreter, copies of equations are not performed, because they are partially compiled, so making the interpreter fast enough. For instance, the resolution of the goal :

```
union[id0,inverse[graph[succ]]](x,0) ?
```

is equivalent to the resolution of the goal :

```
P1(x,0) ?
```

in the context of declarations of figure 7. We can expect that the result will be `x == 0` for the first solution and `x == succ(0)` for the second solution.

4 Operational Semantics of LPG

Let SP be a completely instantiated LPG specification, i.e. a collection of functions and predicates. Classical operational semantics of functional languages consists in a calculus that allows the "evaluation" (or normalization) of terms ; whereas operational semantics of logic programming languages aims to solve goals. A functional logic programming language provides these two possibilities. In LPG, a goal is a conjunction $A_1, \ldots, A_m, \ldots, A_n$ where every A_i is either an equation, a disequation or any other literal formed by predicates in SP. In the sequel we represent such a goal as a multiset $\{A_1, \ldots, A_n\}$. The operational semantics of LPG is based on one complete inference rule, an extension of the SLD-resolution, called SLDEI-resolution [9, 2]. This inference rule is parameterized by an algorithm of solution of equations and disequations in initial models and may be easily extended in order to cope with other new constraints.

```
type Evaluation_context
sorts nat
constructors
   0 :  → nat
   succ :  nat → nat
predicates
   P1 :  (nat,nat)                   -- union[id0,inverse[graph[succ]]].
   P2 :  (nat,nat)                   -- inverse[graph[succ]].
   P3 :  (nat,nat)                   -- graph[succ].
   id0 :  (nat,nat)                  -- non generic predicate.
variables
   x,y:  nat
clauses
   1:  P1(x,y) <== id0(x,y)          -- clause 1 of union.
   2:  P1(x,y) <== P2(x,y)           -- clause 2 of union.
   3:  P2(x,y) <== P3(y,x)           -- clause defining inverse.
   4:  P3(x,succ(x))                 -- clause defining graph
   5:  id0(0,0)                      -- clause defining id0.
end Evaluation_context
```

Figure 7: Flattened context of evaluation of `union[id0,inverse[graph[succ]]](x,0)`

Definition 4.1 (*SLDEI*-**resolution**)
Let $SP = (\Sigma, E, Cl)$ be a constructor-based specification, G a goal and θ a substitution. The inference rules of SLDEI-resolution *are :*

- *Resolution:*

$$\frac{(G \cup \{P(t_1, \dots, t_n)\}; \theta)}{(\sigma(G \cup \{D_1, \dots, D_k\}); \sigma\theta)}$$

if $P \in \Pi$.
$P(v_1, \dots, v_n)$`<==`$D_1, \dots, D_k$ *is a variant of a clause in* Cl *and* σ *belongs to a complete set of solutions of* $\{v_1$`==`$t_1, \dots, v_n$`==`$t_n\}$.

- *Unification:*

$$\frac{(G \cup \{t\texttt{==}t'\}; \theta)}{(\sigma(G); \sigma\theta)}$$

if σ *belongs to a complete set of solutions of* t`==`t'.

- *Disunification:*

$$\frac{(G \cup \{t\texttt{=/=}t'\}; \theta)}{(\sigma(G); \sigma\theta)}$$

if σ *belongs to a complete set of solutions of* t`=/=`t'.

We denote by $\rightarrow_{SLDEI}$ the relation on goals and substitutions induced by the inference rules above. Then, we say that a substitution σ is *a solution computed by SLDEI-resolution* for a goal G if there exists a derivation

$$(G; id) \rightarrow^{+}_{SLDEI} (\emptyset; \sigma)$$

Theorem 4.2 *Let $SP = (\Sigma, E, Cl)$ be a constructor-based specification and G a goal. The set of solutions computed by SLDEI-resolution for the goal G is a complete set of solutions of G.*

The proof of the completeness of SLDEI-resolution is quite similar to the one given in [9]. The consideration of disequations does not cause any substantial problem since the restriction of the use of disequations (Recall that disequations are not allowed in the bodies of conditional equations) ensure the existence of initial models. In other words, these conditions ensure the continuity of a transformation on Herbrand E-models and then the existence of its least fixed point.

4.1 Solution of Equations and Disequations

In order to define an effective operational semantics for our language, we need actual methods which compute complete sets of solutions of equations and disequations. In following, we briefly present the inference rules that underlie such an algorithm which we have designed and implemented in LPG. This algorithm is based on narrowing [14]. It is well known that narrowing-based algorithms develop a very large search tree, computing very often redundant informations. Thus, many strategies of narrowing have been proposed. One of the most interesting improvement of narrowing-based algorithms consist in using the so called *narrowing position selection strategies* (or *narrowing strategies* for short). A narrowing strategy NS is a partial function from terms to positions (sequence of naturals). $NS : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{N}_+^*$. $NS(t)$ is defined to be a position p such that the term t is narrowable at position p. $NS(t)$ is defined if and only if such a position p exists. By abuse of notation, we extend the domain of strategies to atoms and goals as follows: We write $NS(t\texttt{==}t')$ resp. $NS(t\texttt{=/=}t')$ to denote a narrowing position either in t or in t' when it exists. For a goal G, we write $NS(G)$ to denote a narrowing position $NS(e)$ such that e is an equation or a disequation in G.

Unfortunately, the use of such strategies does not always preserve the completeness of narrowing algorithms (see [8] for a counter example). So, sufficient conditions on specifications that ensure the completeness of such strategies (particularly the *outermost strategy*) have been proposed [8, 17, 10]. In this paper, we extend the conditions in [10] to the case of CTRS's. In [10] we have proved that, in addition to termination and confluence of unconditional TRS's, a narrowing strategy is complete if it is uniform. A narrowing strategy NS is called *uniform* iff for any term $t \in \mathcal{T}(\Sigma, \mathcal{X})$, if $NS(t)$ is defined then for every ground normalized substitutions $\mu, \mu(t)$ is reducible at position $NS(t)$.

The following example shows that these conditions do not ensure any more the completeness of narrowing strategies in the case of conditional TRS's.

Example 4.3 *Consider the following CTRS. It is borrowed from [16] and modified in order to satisfy the uniformity condition [10].*

$f(d) \to a$ `<==` $d\texttt{==}b, d\texttt{==}c$
$f(c) \to a$
$f(a) \to c$

$$
\begin{array}{ll}
d & \rightarrow b \\
d & \rightarrow c \\
b & \rightarrow c \texttt{<==} f(d)\texttt{==}a
\end{array}
$$

This CTRS is confluent and terminating. Consider the goal $f(d)\texttt{==}a$. The identity substitution is a solution for this goal. However, we can find some uniform narrowing strategy that leads to no solution. Indeed, consider the following derivation. At each step, we underline the equation we narrow.

$$
\begin{array}{ll}
\underline{f(d)\texttt{==}a} & \rightsquigarrow a\texttt{==}a, \underline{d\texttt{==}b}, d\texttt{==}c \\
& \rightsquigarrow a\texttt{==}a, d\texttt{==}c, \underline{f(d)\texttt{==}a}, d\texttt{==}c \\
& \rightsquigarrow \ldots
\end{array}
$$

Hence we can manage to develop an infinite derivation that leads of course to no solution. However, at each step, the chosen position is uniform.

To overcome this problem we need to reason upon a further relation [16] induced by CTRS on goals. We note this relation $\mapsto$ which is defined by :

Definition 4.4 *Let R be a CTRS and S and T two goals. We write $S \mapsto_R T$ if there exists a goal e in S, a position $p \in Pos^*(e)$, a variant $l \rightarrow r\texttt{<==}c$ of a rewrite rule in R and a substitution σ such that*

- $e_{|_p} = \sigma(l)$
- $T = S - \{e\} \cup \{e[\sigma(r)]_p\} \cup \sigma(c)$
- *for every equation $s\texttt{==}t$ in c, $\sigma(s) \downarrow_R \sigma(t)$*

The advantage of this relation consists in having a simple correspondence between conditional narrowing and conditional rewriting [16] as it is the case for unconditional TRS's. Thus the completeness proof of conditional narrowing becomes quite similar to unconditional narrowing and the conditions for the completeness of narrowing strategies concern rather the relation $\mapsto_R$ than $\rightarrow_R$. Indeed, to ensure the completeness of a narrowing strategy NS, we require for $\mapsto_R$ to be confluent (which is implied by the confluence of $\rightarrow_R$) and terminating. Note that termination of $\rightarrow_R$ does not imply termination of $\mapsto_R$ (see the example above). From the narrowing strategy NS, we require uniformity between conditional narrowing and the relation $\mapsto_R$, i.e. if $NS(t)$ is defined, for every ground normalized substitution μ, $\mu(t)$ is reducible by $\mapsto_R$ at position $NS(t)$. In the following definition we give the inference system $EQUALITY$ which consists of the main inference rules of the algorithm of solution of equations and disequations we have implemented in LPG.

Definition 4.5 (Inference system $EQUALITY$)
Let $SP = (\Sigma, R, Cl)$ be a constructor-based specification such that $\mapsto_R$ is confluent and terminating, NS a uniform narrowing strategy, G a goal consisting of equations and disequations and θ a substitution. The inference rules for the solution of equations and disequations are :

- *Uniform Narrowing 1:*

$$\frac{(G \cup \{t\texttt{==}t'\}; \theta)}{(\sigma(G \cup \{(t\texttt{==}t')[r]_p\} \cup c); \sigma\theta)}$$

$$\textit{if} \;\; t\texttt{==}t' \leadsto_{[p, l \to r \texttt{<==} c, \sigma]} \sigma(\{t\texttt{==}t'[r]_p\} \cup c) \;\; \textit{and} \; p = NS(t\texttt{==}t').$$

- *Uniform Narrowing 2:*

$$\frac{(G \cup \{t\texttt{=/=}t'\}; \theta)}{(\sigma(G \cup \{(t\texttt{=/=}t')[r]_p\} \cup c); \sigma\theta)}$$

$$\textit{if} \;\; t\texttt{=/=}t' \leadsto_{[p, l \to r \texttt{<==} c, \sigma]} \sigma(\{t\texttt{=/=}t'[r]_p\} \cup c) \;\; \textit{and} \; p = NS(t\texttt{=/=}t').$$

- *Success 1:*

$$\frac{(G \cup \{t\texttt{==}t'\}; \theta)}{(G; \sigma\theta)}$$

if t and t' are not narrowable and σ = mgu(t, t')

- *Success 2:*

$$\frac{(G \cup \{k(t_1, \ldots, t_n)\texttt{=/=}k'(t'_1, \ldots t'_m)\}; \theta)}{(G; \theta)}$$

if k and k' are two different constructors

- *Decomposition:*

$$\frac{(G \cup \{k(t_1, \ldots, t_n)\texttt{=/=}k(t'_1, \ldots t'_n)\}; \theta)}{(G \cup \{t'_i\texttt{=/=}t'_i\}; \theta)}$$

if k is a constructor and $1 \leq i \leq n$

- *Instantiation 1:*

$$\frac{(G \cup \{x\texttt{=/=}k(t_1, \ldots, t_n)\}; \theta)}{(\sigma(G) \cup \{k'(z_1, \ldots, z_m)\texttt{=/=}\sigma(k(t_1, \ldots, t_n)); \sigma\theta)}$$

if x is a variable, k and k' are constructors, $z_1, \ldots, z_m$ are new variables and $\sigma = \{x \mapsto k'(z_1, \ldots, z_m)\}$

- *Instantiation 2:*

$$\frac{(G \cup \{x\texttt{=/=}y\}; \theta)}{(\sigma(G) \cup \{k_1(x_1, \ldots, x_m)\texttt{=/=}k_2(y_1, \ldots, y_n); \sigma\theta)}$$

if x and y are different variables, k_1 and k_2 are constructors, $x_1, \ldots x_m, y_1, \ldots, y_n$ are new variables and $\sigma = \{x \mapsto k_1(x_1, \ldots, x_m), y \mapsto k_2(y_1, \ldots, y_n)\}$

We denote by $\to_{EQU}$ the relation on goals and substitutions induced by the inference rules above. Let G be a conjunction of equations and disequations. Then, we say that a substitution σ is *a solution computed by the system EQUALITY* for the goal G if there exists a derivation

$$(G; id) \to^+_{EQU} (\emptyset; \sigma)$$

Theorem 4.6 *Let $SP = (\Sigma, R, Cl)$ be a constructor-based specification such that $\mapsto_R$ is confluent and terminating, NS a uniform narrowing strategy, G a goal consisting of equations and disequations. The set of solutions computed by the inference system $EQUALITY$ for the goal G is a complete set of solutions of G.*

The proof of the completeness of the inference system $EQUALITY$ may be found in [3]. If we consider unconditional TRS's we can retrieve, within the system $EQUALITY$, on one hand the rules for solution of equations using narrowing strategies and on the other hand the rules of solution of disequations given in [11] that we have corrected and improved by the use of narrowing strategies.

The actual operational semantics of LPG includes further rules that improve the implementation. Thus, we rather use normal narrowing, i.e., after each narrowing step, the considered term is normalized. The use of inductive rules while normalization is also allowed. Other simplification rules are used such as :

- *Simplify 1:*

$$\frac{(G \cup \{t\texttt{==}t\}; \theta)}{(G; \theta)}$$

- *Simplify 2:*

$$\frac{(G \cup \{x\texttt{==}t)\}; \theta)}{(\sigma(G); \sigma\theta)}$$

if x is a variable such that $x \notin \mathcal{V}ar(t)$ and $\sigma = \{x \mapsto t\}$

- *Simplify 3:*

$$\frac{(G \cup \{x\texttt{=/=}t)\}; \theta)}{(G; \theta)}$$

if x is a variable such that $x \in \mathcal{V}ar(t)$ and t contains no defined operator (occur-check)

Other rules are used to detect failures such as :

- *Fail 1:*

$$\frac{(G \cup \{t\texttt{=/=}t\}; \theta)}{Failure}$$

- *Fail 2:*

$$\frac{(G \cup \{k_1(t_1, \ldots, t_n)\texttt{==}k_2(s_1, \ldots, s_m)\}; \theta)}{Failure}$$

if k_1 and k_2 are different constructors

A disequation of the form $x\texttt{=/=}t$ such that $x \notin \mathcal{V}ar(t)$ and t is a constructor term, i.e., t contains no defined oparations, has an infinite solution when x ranges over an infinite domain. To overcome this infinite computation, these kind of disequations are detected and presented as such in the solutions (see Figure 8 for a sample session).

```
{1} vertical (a) ?
for all k_1 :  nat
for all j_1 :  nat
for all i_1 :  nat
a == c_line(c_dot(i_1,j_1),c_dot(i_1,k_1))
k_1 =/=j_1
more solutions?  y
No more solutions.
```

```
{1} x + y == x * y ?
x == 0
y == 0
more solutions?  y
x == 2
y == 2
more solutions?  n
{1} [0 | s] =/=[0] ?
s =/=nil
more solutions?  y
No more solutions.
```

Figure 8: Sample session of the Solver.

4.2 Tools

The current environment of LPG includes mainly two tools:

- An interpreter : This tool evaluates functional expressions like in

  ```
  {1} sort[<=,= ]([1,25,3,27]) ==>
  [ 1, 3, 25, 27]
  {2} sort[>=,=]([1,25,3,27]) ==>
  [ 27, 25, 3, 1]
  ```

  This tool uses a compiled code and runs on an abstract machine described in [2].

- A solver : This tool solves goals. It attempts to find (à la PROLOG) values for the variables that make the considered goal hold. This tool implements the SLDEI-resolution together with the narrowing-based algorithms of solution of equations and disequations. The computation of the solver may be performed either by depth-first strategy or breadth-first strategy according to user's indications. The selected atom at each step of the solver is the left-most one. The current implemented narrowing strategy is the innermost-leftmost strategy.

5 Conclusion

In LPG, predicates and functions are well distinguished. This fact allowed us to use disequations within the bodies of Horn clauses without any problem. In other words, any completely instantiated LPG specification has an initial model. In languages based on conditional equational logic, the use of disequations in the specifications will not be so immediate. Indeed, the presence of disequations in the bodies of equational clauses does not always ensure the existence of an initial model.

We have used narrowing-based algorithms to solve equations and disequations. We have generalized our previous conditions [10] on specifications and strategies to ensure the completeness of narrowing strategies in presence of conditional term rewrite systems.

The use of narrowing strategies allows to prune drastically the narrowing search tree. However, the conditions required for the use of such strategies may seem to be restrictive. In order to deal with a larger class of specifications, *needed narrowing* [1] is a very good candidate. Furthermore, narrowing strategies does not always avoid useless branches. Thus, other techniques must be used in order to detect wasted computations. In [4] we have investigated a new method, based on abstract analysis techniques, which allows to avoid narrowing over some unsatisfiable equations or disequations.

References

[1] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, Oregon, January 1994.

[2] Didier Bert and Rachid Echahed. Design and implementation of a generic, logic and functional programming language. In *Proceedings of the First European Symposium on Programming, ESOP'86*, number 213 in Lecture Notes in Computer Science, pages 119–132. Springer-Verlag, 1986.

[3] Didier Bert, Rachid Echahed, and Peter Habermehl. Uniform narrowing strategies for semantic unification and disunification. Technical report, IMAG, University of Grenoble. Forthcoming, 1994.

[4] Didier Bert, Rachid Echahed, and Bjarte M. Østvold. Abstract rewriting. In *Proceedings of the 3rd International Workshop on Static Analysis, WSA'93*, number 724 in Lecture Notes in Computer Science, pages 178–192. Springer-Verlag, 1993.

[5] Didier Bert, Rachid Echahed, and Jean-Claude Reynaud. Reference manual of the LPG specification language and environment, release with disequations. Technical report, IMAG, University of Grenoble, 1994.

[6] Didier Bert and Catherine Oriat. Structuring specifications in the algebraic and logic language LPG. Technical report, IMAG, University of Grenoble, 1994.

[7] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[8] R. Echahed. On completeness of narrowing strategies. In *Proc. CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[9] Rachid Echahed. Sur l'intégration des langages algébriques et logiques. These de l'Institut National Polytechnique de Grenoble, 1990.

[10] Rachid Echahed. Uniform narrowing strategies. In *Proceedings of the 3rd International Conference on Algebraic and Logic Programming*, number 632 in Lecture Notes in Computer Science, pages 259–275. Springer-Verlag, 1992.

[11] Maribel Fernandez. Narrowing based procedures for equational disunification. Technical Report 764, LRI, University of Paris Sud, July 1992.

[12] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. of 12th ACM POPL*, pages 51–60. ACM, 1985.

[13] Michael Hanus. The integration of functions into logic programming : A survey. Technical Report MPI-I-94-201, Max-Planck-Institut fur Informatik, 1994.

[14] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th Conference on Automated Deduction*, number 87 in Lecture Notes in Computer Science, pages 318–334. Springer-Verlag, 1980.

[15] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1984.

[16] A. Middeldorp and E. Hamoen. Counterexamples to completeness results for basic narrowing (extended abstract). In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 244–258, Volterra, Italy, September 1992.

[17] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.