Negation in Conclog

J.-M. Jacquet
Center for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

This paper presents a new constructive form of the negation-as-failure rule dedicated to concurrent executions and, based on it, a parallel execution model of general Horn clauses. Referring to the completion understanding of the programs, this model has been proved sound and as complete as possible when the resolution rule and the negation-as-failure rule are used. Furthermore, it does not suffer from the floundering problem : in contrast, negative literals can be used to produce computed answers as the positive literals can.

This new form of negation is based on an equational framework. It introduces generalizations of substitutions, terms and related concepts of instantiation and unification. A theory of such concepts is also presented in the paper.

Keywords : constructive negation, parallel logic programming, parallel language, theory of substitutions and terms.

1 Introduction

Although it is theoretically not necessary ([Tarnlund, 1977]), negation is one of the very desirable features of any logic programming language. Theoretically, it can take three places : in the head of clauses, in the body of clauses and in queries. Negation in the head of clauses provides the full power of first order logic and, consequently, needs a much more expensive form of resolution than the SLD one. One attractive feature of Horn clauses with SLD-resolution is precisely that it can be implemented very efficiently. Hence, the only places where not predicates are usually allowed to take place are in queries and in body of clauses. As a result, there is no means left to derive negative information so that the following trick is generally added to SLD-resolution. It consists of solving not(p) by first solving p and then inverting the answers, thus stating the provability of not(p) if p is found unprovable and vice-versa. This is known as negation as failure. Its great advantage is the possibility to implement it with essentially no extra-cost to the resolution system. Its main drawback is that it is not real negation. More specifically, variables have to be carefully manipulated, the result generally depending of the clause and the goal selection strategies. For instance, with Prolog depth first search strategy and with regards to the one clause program

p(a).

the reduction of not(p(X)) fails whereas that of not(p(b)) succeeds. In fact, with Prolog depth first search, the goal not(p($\vec{X}$)), with $\vec{X}$ the variables occurring in p, is interpreted as $\neg(\exists \vec{X}\, p(\vec{X}))$ whereas the expected interpretation is $\exists \vec{X}\, (\neg p(\vec{X}))$. These two interpretations are, of course, not always the same. To force the equality, it is of current and safe practice to reduce negative literals only when they are ground. They can thus be used only as tests and not to produce answers, as the reduction of positive literals does. Furthermore, one problem - known as the floundering problem - occurs when the reduction has to select one subgoal in a conjunction composed only of non-ground negative literals.

Recently, a new form of the negation-as-failure rule, called constructive negation ([Chan, 1988], [Chan, 1989]), has been proposed to remedy this situation. An alternative form dedicated to concurrent executions is presented in this paper. It is also constructive and, consequently, avoids the floundering problem. It issues from the design of a concurrent logic programming language, named Conclog, but is not dedicated to it. It rests on an equational perception of the computation and has induced the generalization of several basic concepts. All of these generalized forms have been extensively studied in [Jacquet, 1989]. They are used here only as a basis for the paper and are just sketched. Their reason for existence may be motivated as follows. Interpreting the executions in equational terms, it is not difficult to imagine that handling negation leads to negate some equations and, therefore, to solve systems composed of equations and inequations. In general, the set of solutions of these systems cannot be represented by a finite number of substitutions. Our remedy to this situation is to introduce negative information in the substitutions. This produces constructs of the form (θ,not ω) where θ is a substitution and ω is a set of bindings presented in a normal form with respect to θ. Such constructs are called n-substitutions. In a symmetric way, negative information has also been introduced in terms. This results in so-called extended terms, of the form (t,Ω) where Ω is a set of "not ω" constructs, just introduced but normalized here with respect to t. Related concepts of unification and instantiation have then been extended to these new terms and substitutions.

Given these extended concepts, our constructive approach to negation basically consists of the following. Let, for any n-substitution $\nu=(\{X_1/t_1,\ldots,X_p/t_p\},\text{not}\{Y_1/u_1,\ldots,Y_q/u_q\})$, eq($\nu$) be

$$X_1=t_1 \wedge \ldots \wedge X_p=t_p \wedge Y_1\neq u_1 \wedge \ldots \wedge Y_q\neq u_q \,.$$

Assume the reduction of G has $\nu_1, \ldots, \nu_m$ as computed n-substitutions. Then return as computed answer n-substitutions to not(G) the n-substitutions $\mu_1, \ldots, \mu_n$ such that the negation of eq(ν_1) $\vee \ldots \vee$ eq(ν_m) is equivalent to eq(μ_1) $\vee \ldots \vee$ eq(μ_n). Of course, this basic scheme is slightly adapted to cope correctly with the quantification of the variables.

A parallel execution model of general Horn clauses based on the above features is also presented in the paper. Referring to the completion understanding of the programs, it has been proved sound and as complete as possible when the resolution principle and the negation-as-failure rule are employed. As a snapshot, it essentially rests on and-parallelism, or-parallelism and on the constructive negation introduced above. Reconciliation of n-substitutions is introduced as a means of combining n-substitutions issued from conjoined subgoals in order to produce n-substitutions for the whole goal. Negative information and extended unification are furthermore used to constrain the reduction process. Any computation under the model can thus be basically seen as the construction of the and/or/not search tree associated with the query and the program under consideration. However, this tree is not constructed in its pure form but slice by slice. At the end of the construction of each slice, branches that are detected useless are killed and some n-substitutions are published in order to avoid, in a "a priori" way, the production of such useless branches. The creation of the branches is, of course, made concurrent and independent so that n-substitutions might be delivered as answers to queries whereas others are still under computation. This parallel execution model corresponds to the second phase of the design of a concurrent logic programming language, named Conclog, where extra-logical features such as sequentialization, commit operators, guarded constructs are introduced in a third step to ensure efficiency and practicability.

The main originalities of our work are best presented by comparing it with related negations and execution models. Negation by complex solutions ([Khabaza, 1984]) also tackles negation in a parallel context by means of a reconciliation calculus. It does not use n-substitutions but more complex constructs, called complex solutions, of the form $\{P, \sim N_1, \ldots, \sim N_m\}$ where P and the N_i's are sets of bindings. Although the negated sets $\sim N_i$ can be viewed as negative pieces of information associated with the positive one P, no negative information has been coupled with the terms and the notions of instantiation and unification have not been extended to cope with it. As a result, some derivations that are removed in our model of execution are performed in Khabaza's model. Another major difference is that negation by complex solutions does not take into account sets of variables, intended to represent variables differently quantified, whereas we will do. The consequence is that the Conclog model is sound and is complete in some situations whereas the model of [Khabaza, 1984] is neither sound nor complete.

Negation by constraints ([Wallace, 1987]) uses directly the completed definition of the procedures to reduce the negative literals. This results in handling first order formulae which are far more complex than our extended terms. As another difference, negation by constraints take place in a sequential context and thus does not provide any counterpart for reconciliation and binding publication. Finally, negative information plays also a role in the deduction by means of so-called constraint sets but those are structured in a way that suits well the sequential executions but does not fit the parallel ones.

The SLD-CNF rule ([Chan, 1988], [Chan, 1989]) is another constructive form of the negation as failure rule. It consists of an extension of the SLD-resolution rule whose main feature is to reduce any negative literal, say not(Q),

(i) by reducing Q to an normalized conjunctive formula F involving equalities, inequalities and unreduced subgoals

(ii) and by negating F in a backtracking spirit.

This rule has first been presented in a sequential framework and has then been extended in [Chan, 1989] in an incremental version of constructive negation applied to co-routining.

A first difference with our work is that, although equations and inequations take a great place and although normalized formulas are introduced, there is, in [Chan, 1988] and [Chan, 1989], no counterpart for our theory of n-substitutions and extended terms, unification and instantiation. As a symptom, negative information plays no role during the reductions whereas it does in ours. Also, inequalities are treated in a more uniform way in our model : there is, in our work, no need to look for satisfiability or validity. Finally, no concurrent model is treated in [Chan, 1988] and the model presented in [Chan, 1989] does not tackle and-parallelism and or-parallelism. There is thus no concept of reconciliation and of communication of bindings between subtrees. It is here furthermore worth noting that incremental constructive negation can also be achieved in our Conclog model by means of the construction slice by slice of the and/or/not search tree and that co-routining can also be simulated in the same way by fixing the depth of the slices to 1.

The SLSC-resolution rule ([Przymusinski, 1989]) is a theoretical adaptation of the SLD-CNF resolution rule associating infinite derivation with failed derivation and making reference to the perfect model semantics instead of the completion interpretation of the programs. These two latter points make already two differences with the Conclog model. Other comparison points result from the above comparison with the SLD-CNF-resolution.

Negation by failure substitutions ([Maluszynski and Näslund, 1989]) is an extension of the negation-as-failure principle based on the idea that if γ is a substitution such that the query $\leftarrow G\gamma$ has a finitely failed SLD-tree then, by soundness of the negation as failure rule, $\forall \neg G\gamma$ is a logical consequence of the completion of the program under consideration. Reducing negative literals can thus be achieved by finding all such substitutions γ, called failed substitutions. Only the maximal substitutions need, of course, to be found. This is indeed performed in [Maluszynski and Näslund, 1989]. Concepts of constrained substitutions and constrained terms, adding inequalities of terms to substitutions and terms, respectively, are introduced for this purpose. Our n-substitutions and extended terms share some similarities with them. The major differences is that n-substitutions and extended terms are always presented in a normalized form and that n-substitutions may have the empty substitution as positive part θ. Furthermore, our theory of n-substitutions and extended terms is more elaborated than that of [Maluszynski and Näslund, 1989], where the concept of composition of substitutions is just extended.

Another difference with our work is that negation by failed substitutions only tackles the sequential compositions. It does not provide any counterpart for our concept of reconciliation of n-substitutions, our parallel form of negation or our concept of publication of n-substitutions.

Another way of handling negation consists of deriving from a Horn clause program P, a new set of predicates computing the negation of the predicates of P. This approach has been first proposed in [Sato and Tamaki, 1984] and has been improved in [Barbuti et al., 1987] and [Barbuti et al., 1990]. One attractive feature of those pieces of work is certainly that negation is tackled without extending the terms and the substitutions with negative information while negative literals can compute substitutions as the positive literals do. Another advantadge is that the resulting negation is complete. The drawbacks are that the derived procedures may be quite verbose and that the computed substitutions may suffer from conciseness problems. Furthermore, the treatment of negation requires both a new form of negation introduced by the synthesized predicates and the negation as failure rule, used to compute s-goals of the form $\forall Y$ $r(X,Y)$.

The ENF deduction procedure ([Lugiez, 1989]) is a deduction procedure for first order programs which subsumes SLDNF-resolution and which improves the handling of the latter goals. It presents several similarities with our work in that negative information is also combined with the positive one to sum up solutions. In fact, the employed substitutions are quite close to our n-substitutions. However, no theory of such substitutions is proposed and negative information is not employed to constrain the reduction process. Moreover, as parallel executions are not tackled, there is also no counterpart for our reconciliation concept. Another major departure with our work is that first order formulae are handled by the ENF procedure. It is thus computationally more expensive than the Conclog one. Finally, the verbosity and conciseness problems of the above transformational approach to negation have not been solved in [Lugiez, 1989].

The Conclog execution model can be seen as an extension to incorporate negation of parallel execution models of Horn clauses ([Pollard, 1981], [Conery, 1983], [Li and Martin, 1986], [Kalé, 1987]). Besides the introduction of negation, the last three pieces of work take another approach to and-parallelism. Given a conjunction of subgoals that share variables, they do not allow those subgoals to be reduced concurrently but avoid the generation of conflicting bindings by designating a producer process for each shared variable. Unfortunately, the designation of the right producer is undecidable. Moreover, we strongly believe that it is impossible to find a sufficiently general class of programs for which it would be decidable. The practical consequence is that any algorithm computing this determination does not work properly on some goals. In view of this, we prefer to tackle the determination of dataflows at a programming level by means of annotations. Another difference with the Conclog model is that the models of [Conery, 1983] and [Li and Martin, 1986] are not complete whereas the Conclog model is when it is restricted to Horn clauses. Finally, all the three models suffer from the rigidity of the dataflow they determine : the unproductive derivations of the producer processes cannot be removed in view of the consumer ones and back communication cannot be performed. In contrast, this is possible in Conclog.

The Conclog model share the reconciliation approach with that of [Pollard, 1981] but differs in the way it manages the computed and/or search trees. In Conclog, every aspect is treated in a simpler and more uniform framework based on equation manipulation whereas a lot of concepts and principles are introduced in Pollard's approach. Moreover, this approach does not publish bindings, which prevents the Conclog model from developing useless branches.

As a final comparative remark, it is worth noting that, although systems of equations and inequations have already been studied and although extended substitutions have already been proposed (see e.g. [Khabaza, 1984], [Maluszynski and Näslund, 1989], [Turi, 1991]), no work has elaborated, to our extent, a theory generalizing, to a negative constructive framework, that of the basic notions of logic programming, namely terms and substitutions.

The remainder of this paper is organized in 5 Sections. Section 2 sketches our theory of n-substitutions, extended terms, extended instantiation and extended unification and introduces the basic mechanisms of our constructive negation. Section 3 presents the Conclog execution model of general Horn clauses including our constructive negation and sketches its main properties. Section 4 gives our conclusions. Finally, Sections 5 and 6 present our acknowledgments and references.

Lack of space prevents us to give proofs of the claimed results. All of them have however been established in [Jacquet, 1989], to which we refer the reader for more information. He is also assumed to be familiar with logic programming. We shall essentially use the conventional Edinburgh syntax (see e.g. [Clocksin and Mellish, 1981]) and the conventional logic programming terminology (see e.g. [Lloyd, 1987] and [Apt, 1990]). Our only particular notation is to denote the identity substitution by $\{\}$. We shall furthermore use the abbreviation s-goal to denote any subgoal (either positive or negative literal) of a conjunction and the qualification ps-goal and ns-goal to denote the positive and the negative s-goals, respectively. Finally, from now on, we assume that the language under consideration has an infinite number of variables, constants and function symbols and define the Herbrand universe with respect to the language rather than the programs under consideration.

2 Towards a theory of n-substitutions and extended terms

2.1 Intuition

In order to provide the reader with an intuitive support, it is worth presenting the underlying ideas having lead to our subsequent theory of n-substitutions and extended terms.

The ideal of concurrent logic programming is to use both or-parallelism and and-parallelism. In this ideal world, the sequential reduction of subgoals of goals is thus replaced by the concurrent reduction of the subgoals, even if they do share variables. It follows that the traditional composition of substitutions returned by the successive reductions of subgoals needs to be replaced by a concurrent composition of substitutions, devoted to combining the (possibly conflicting) substitutions issued from the concurrent reductions of the subgoals. It has been provided, independently, in [Jacquet, 1989] and [Palamidessi, 1990] under the name of reconciliation of substitutions and parallel composition of substitutions, respectively. In both cases, the basic idea is to interpret substitutions equationally and to combine them by solving the system of equations associated with the substitutions. This equational interpretation has been shown, in both references, to be quite intuitive and very simple. Nevertheless, it induces one restriction : unifiers and mgus should be taken as idempotent. Curiously, however, this restriction turns out to simplify the theory rather than to complicate it. It is furthermore worth noting that the restriction to idempotent substitutions is not of importance both from a theoretical and a practical point of view. Indeed, on the one hand, any pair of unifiable terms have an idempotent mgu . On the other hand, most unification algorithms report an idempotent mgu. Finally, the classical objection that the composition of two idempotent substitutions is not necessarily an idempotent substitution cannot be retained in our parallel context : reconciliation of idempotent substitutions delivers an idempotent substitution and the only compositions that are needed may be proved to deliver idempotent substitutions.

The equational interpretation of substitutions has another merit of suggesting a very simple way of dealing with negation. Let, for any substitution $\theta=\{X_1/t_1,\ldots,X_m/t_m\}$, eq($\theta$) be the formula

$X_1=t_1 \wedge \ldots \wedge X_m=t_m$

Let not(G) be the negative literal under consideration and $\theta_1, \ldots, \theta_p$ be all the substitutions returned by the reduction of G. Then, the substitutions to be returned by the reduction of not(G) could be the substitutions $\mu_1, \ldots, \mu_q$ such that the formula $\neg(eq(\theta_1)\vee \ldots \vee eq(\theta_p))$ is equivalent to the formula $eq(\mu_1) \vee \ldots \vee eq(\mu_q)$. This basic idea needs however to be refined in two ways.

1° First, substitutions are, in general, not sufficient to ensure the above equivalence. In fact, keeping some negative information is necessary. This fact has lead us to generalize substitutions in so-called n-substitutions. As an additional consequence, reconciliation has to be extended to n-substitutions, or, restated in other terms, systems of equations and inequations have to be manipulated instead of just systems of equations.

2° Second, care must be taken to the quantification of the variables. This has lead us to introduce the concept of consistency of n-substitutions.

These notions have suggested us to extend other classical notions (such as composition, idempotence, ...) to n-substitutions. In particular, a natural notion to generalize is that of the instantiation of a term by an n-substitution. The concepts of extended term, and related ones of extended unification and instantiation, have followed.

All these extensions have been conflated in a theory of n-substitutions and extended terms. We will only describe here the part necessary to make the paper as self-contained as possible. The interested reader is referred to [Jacquet, 1989] for more details.

The remainder of this section is organized in three subsections. The first one discusses the systems of equations and inequations, introduces the concepts of unifier, mgu for such systems and the related notion of n-substitution. The second subsection generalizes, to n-substitutions, the classical concepts of composition, reconciliation, restriction of substitutions and the partial order $\leq$ on substitutions. It also introduces the concepts of negation and consistency of n-substitutions. Finally, the third subsection discusses the extended terms, extended unification and extended instantiation.

2.2 Systems of equations and inequations

For the ease of discussion, let us first adopt the following terminology.

Definition 2.1 We subsequently call *h-system* any system of equations and inequations over terms of the Herbrand universe. Given an h-system S, the system of the equations (resp. of the inequations) of S is subsequently denoted by S^+ (resp. S^-). A *solution* of S is a grounding substitution α verifying the three following properties :

(i) its domain, dom(α), consists of the variables of S,

(ii) for any equation t=u of S^+, $t\theta$ and $u\theta$ are syntactically identical

(iii) for any inequation $v \neq w$ of S^-, $v\theta$ and $w\theta$ are syntactically distinct.
The set of solutions of S is denoted by *Sol(S)*. The h-system S is said to be *solvable* iff the set Sol(S) is not empty. ♦

Domain restriction of solutions makes a solution of a system not necessarily a solution of an intuitively equivalent system. For instance, the solution {X/a,Y/b} of the h-system

$$\begin{cases} Y=Y \\ X \neq b \end{cases}$$

is not a solution of the h-system composed of the only inequation

$X \neq b$.

It is however desirable to define, as equivalent, h-systems that are intuitively equivalent. This is achieved through the notion of solution-weaker systems.

Definition 2.2 The h-system S is *solution-weaker* than the h-system T iff for any solution σ of S, for any grounding substitution γ for the variables of var(T)\var(S), the restriction of $\sigma\gamma$ to the variables of T is a solution of T. This denoted by $T \supset_{sol} S$. The h-systems S and T are *equivalent* iff any of them is solution-weaker than the other. This is denoted by $S \approx T$. ♦

Notation 2.3 It will be convenient to extend the operand of the $\supset_{sol}$ and $\approx$ relations to "disjunctions" of h-systems. We write

$(T_1 \vee \ldots \vee T_n) \supset_{sol} (S_1 \vee \ldots \vee S_m)$

to denote the following property : for any $i \in \{1,\ldots,m\}$, for any solution σ of S_i, there is a $j \in \{1,\ldots,n\}$ such that for any grounding substitution α for the variables of $var(S_i) \setminus var(T_j)$, the restriction of $\sigma\alpha$ to the variables of T_j is a solution of T_j. The notation

$(T_1 \vee \ldots \vee T_n) \approx (S_1 \vee \ldots \vee S_m)$

is then employed to sum up the two inclusions

$(T_1 \vee \ldots \vee T_n) \supset_{sol} (S_1 \vee \ldots \vee S_m)$
$(S_1 \vee \ldots \vee S_m) \supset_{sol} (T_1 \vee \ldots \vee T_n)$. ♦

We are now in position to substantiate our need for explicitly coupling negative information to substitutions and, consequently, to generalize substitutions to n-substitutions. The following proposition, due to [Lassez et al., 1988], proves that, in general, one substitution and even a finite number of them are not sufficient to represent the set of solutions of a h-system. The above need results therefrom.

Proposition 2.4 ([Lassez et al., 1988]) Let S be a solvable h-system. Suppose S^- is not redundant with S^+ that is $S^+ \approx S$ does not hold. Then, there is no finite set of substitutions $\{\theta_1,\ldots,\theta_m\}$, such that for any solution α of S, one has $\theta_i \leq \alpha$ for some $i \in \{1,\ldots,m\}$.

Substitutions are generalized to n-substitutions as follows.

Definition 2.5 An *n-substitution* ν is a pair of the form

$(\{X_1/t_1,\ldots,X_m/t_m\}, not\{Y_1/u_1,\ldots,Y_n/u_n\})$

where $\{X_1/t_1,\ldots,X_m/t_m\}$ is a substitution and $\{Y_1/u_1,\ldots,Y_n/u_n\}$ is a set of bindings such that

(i) $var(\{t_1,\ldots,t_m\}) \supset \{X_1,\ldots,X_m\} \cap \{Y_1,\ldots,Y_n\}$,
(ii) $var(\{t_1,\ldots,t_m\}) \supset \{X_1,\ldots,X_m\} \cap var(\{u_1,\ldots,u_n\})$,
(iii) $u_i \notin \{X_1,\ldots,X_m\}$, for all $i \in \{1,\ldots,n\}$.

The two components are called the *positive part* and the *negative part* of ν, respectively. They are denoted by ν^+ and ν^-, respectively. An n-substitution whose positive part is empty is called an *en-substitution*. It is subsequently represented by its negative part, namely as not$\{Y_1/u_1,\ldots,Y_n/u_n\}$. ♦

The interpretation of the n-substitutions is as follows : $X_1, \ldots, X_m$ have $t_1, \ldots, t_m$ as respective values with the constraint that each Y_i must differ from u_i ($1 \leq i \leq n$). Conditions (i), (ii) and (iii) further force the n-substitutions to be presented in a normal form. Basically, conditions (i) and (ii) express the fact that the inequations cannot directly constrain the X_i's but must do this indirectly through their bindings. This has the interesting consequence that no X_i occurs in the negative part of n-substitutions whose positive part is idempotent. Condition (iii) states a weaker property in the general case: the u_i's cannot be used to negate the X_i's.

Given the equational interpretation of n-substitutions, it will be useful to associate a h-system to any n-substitution. This is achieved as follows.

Definition 2.6 Let $v = (\{X_1/t_1,\ldots,X_m/t_m\},\text{not}\{Y_1/u_1,\ldots,Y_n/u_n\})$ be a n-substitution. The *h-system associated with v*, denoted by *hsyst(v)*, is the h-system composed of the following equations and inequations : $X_1 = t_1, \ldots, X_m = t_m, Y_1 \neq u_1, \ldots, Y_n \neq u_n$. The h-system associated with the n-substitutions $v_1, \ldots, v_m$ is the h-system composed of the equations and inequations of the h-systems hsyst(v_1), ..., hsyst(v_m). It is denoted by *hsyst($v_1,\ldots,v_m$)*. ♦

The concepts of unifier and mgu are extended to h-systems by means of solutions. It is here worth noting that one n-substitution is, in general, not sufficient to sum up all the solutions of a h-system, as proved by the following h-system

$$\begin{cases} X = f(Y,Z) \\ X \neq f(a,b) \end{cases} .$$

However, solving the positive part S^+, injecting the values in the negative part S^- and simplifying the resulting inequations make possible to represent the solutions of any h-system S by a finite number of n-substitutions. As a consequence, n-mgu's are not defined subsequently as n-substitutions but instead as finite sets of n-substitutions

Definition 2.7 An *n-unifier* of a h-system is an n-substitution v that verifies $S \supset_{sol}$ hsyst(v). An *n-mgu* of S is a set of n-substitutions $\{v_1,\ldots,v_m\}$ that verifies $S \approx \text{hsyst}(v_1) \vee \ldots \vee \text{hsyst}(v_m)$. ♦

Definition 2.8 Some n-mgu's have the property that their n-substitutions share the same positive part. They are called *elementary n-mgu* or *en-mgu*, for short. They are denoted as $\theta\oplus\{\omega_1,\ldots,\omega_m\}$ where θ is the common positive part and $\omega_1, \ldots, \omega_m$ are the negative parts of the n-substitutions. Sets of this form are, more generally, called *elementary sets of n-substitutions* or *es-nsubst*, for short.♦

2.3 The space of n-substitutions

A. The partial order ≤

We now show how the theory of substitutions can be extended to n-substitutions. As a first extension, the partial ordering ≤ on substitutions is extended by means of their associated system.

Definition 2.9

1) The n-substitution ν is *more general than* the n-substitution μ iff the inclusion $hsyst(\nu) \supset_{sol} hsyst(\mu)$ holds. This is denoted by $\nu \leq \mu$. Similarly, the set of n-substitutions $M=\{\nu_1,\ldots,\nu_m\}$ is more general than the set of n-substitutions $N=\{\mu_1,\ldots,\mu_n\}$ iff $hsyst(\nu_1) \vee \ldots \vee hsyst(\nu_m) \supset_{sol} hsyst(\mu_1) \vee \ldots \vee hsyst(\mu_n)$ holds. This is denoted by $M \leq N$, too.
2) The n-substitutions ν and μ are *variants* iff they verify the inequalities $\nu \leq \mu$ and $\mu \leq \nu$. Similarly, the sets of n-substitutions $M=\{\nu_1,\ldots,\nu_m\}$ and $N=\{\mu_1,\ldots,\mu_n\}$ are variants iff they verify the inequalities $M \leq N$ and $N \leq M$. ♦

Note that, as a direct consequence of this definition, n-mgus of h-systems are still variants from one another.

B. Composition of n-substitutions

N-substitutions are composed in the following way.

Definition 2.10 The composition of the n-substitution
$\nu=(\{X_1/t_1,\ldots,X_m/t_m\},not\{Y_1/u_1,\ldots,Y_n/u_n\})$
by the n-substitution
$\mu=(\{Z_1/v_1,\ldots,Z_p/v_p\},not\{T_1/w_1,\ldots,T_q/w_q\})$
is the set of n-mgu's of the h-system obtained from the equalities and inequalitites
$X_1 = t_1\mu^+, \ldots, X_m = t_m\mu^+, Y_1\mu^+ \neq u_1\mu^+, \ldots, Y_n\mu^+ \neq u_n\mu^+,$
$Z_1 = v_1, \ldots, Z_p = v_p, T_1 \neq w_1, \ldots, T_q \neq w_q$
by removing any equality $Z_i = v_i$ for which $Z_i \in \{X_1,\ldots,X_m\}$. It is subsequently referred to as $Ncomp(\nu,\mu)$. The notation $\nu \mathbin{\bar{o}} \mu$ is furthermore used to refer to an arbitrary n-mgu of $Ncomp(\nu,\mu)$.♦

C. Idempotence

The notion of the composition of n-substitutions provides us with the possibility of introducing the notion of idempotent n-substitution.

Definition 2.11 An n-substitution ν is idempotent iff it verifies $\{\nu\} \in Ncomp(\nu,\nu)$. ♦

The following proposition gives a worth noting characterization of idempotent n-substitutions in terms of their positive part.

Proposition 2.12 ([Jacquet, 1989]) An n-substitution is idempotent iff its positive part is idempotent. ♦

Given a set of n-substitutions, we subsequently call it idempotent if all its n-substitutions are idempotent. The algorithm sketched for introducing n-mgus (see Definition 2.7) allows us to claim that any solvable h-system has an idempotent en-mgu. Thanks to this property, all subsequent notions defined from n-mgu's of h-systems have an idempotent instance. For example, n-substitutions admit an idempotent n-reconciliation when they are n-reconcilable; extended terms admit an idempotent n-mgu if they are unifiable in an extended sense.

D. Reconciliation of n-substitutions

Following our equational interpretation, reconciliation is generalized to n-substitutions as follows.

Definition 2.13 The n-substitutions $v_1, \ldots, v_m$ are *n-reconcilable* iff the h-system $hsyst(v_1,\ldots,v_m)$ is solvable. In this case, any n-mgu of this h-system is called an *n-reconciliation* of $v_1, \ldots, v_m$. It is called an *en-reconciliation* if it is of the es-nsubst form.

Reconciliation is extended to sets of n-substitutions as follows.

Definition 2.14 The sets of n-substitutions $\Theta=\{\theta_1,\ldots,\theta_m\}$, $\Psi=\{\psi_1,\ldots,\psi_n\}$, ..., $\Omega=\{\omega_1,\ldots,\omega_p\}$ are n-reconcilable iff at least one of the h-systems

$hsyst(\theta_i,\psi_j,\ldots,\omega_k) \quad (1\leq i\leq m,\ 1\leq j\leq n,\ \ldots,\ 1\leq k\leq p)$

is solvable. In this case, let S be the set of tuples $(\theta,\psi,\ldots,\omega)$ of n-reconcilable n-substitutions of $\Theta\times\Psi\times\ldots\times\Omega$. An n-reconciliation of Θ, Ψ, ..., Ω consists of one union

$$\bigcup_{(\theta,\psi,\ldots,\omega)\in S} \rho_N(\theta,\psi,\ldots,\omega)$$

where, for any $(\theta,\psi,\ldots,\omega)\in S$, $\rho_N(\theta,\psi,\ldots,\omega)$ denotes an n-reconciliation of θ, ψ, ..., ω.

E. Consistency

Reconciliation of n-substitutions is quite intuitive but is still too weak to handle negation correctly, as shown by the following example.

Example 2.1 Consider the query ←not(p(X)),q(X) and the program

```
p(f(Y)).
q(f(3)).
```

Reduce the s-goals not(p(X)) and q(X) independently. The n-substitutions ({},not{X/f(Y)}) and ({X/f(3)},not{}) are produced respectively. They n-reconcile - with n-reconciliation { ({X/f(3)},not{Y/3}) } - although the query is manifestly not satisfiable! ♦

In fact, simple resolution of h-systems does not take into account the way in which variables are implicitly quantified. In the above example, the n-substitution ({},not{X/f(Y)}) reports the fact that X must be different from the term f(Y) whatever Y stands for. However, the reconciliation with ({X/f(3)},not{}) only tests whether there is some value of Y such that $X=f(3)$ and $X\neq f(Y)$. Safe introduction of negation thus requires to fix some way of quantifying variables in n-substitutions. We adopt the following one.

Definition 2.15 Let $v=(\{X_1/t_1,\ldots,X_m/t_m\},not\{Y_1/u_1,\ldots,Y_n/u_n\})$ be a n-substitution. Let furthermore Svars be a set of variables. The notation

Form(v;Svars)

denotes the formula

$\exists Z_1\ldots\exists Z_p\ \forall V_1\ldots\forall V_q : (X_1=t_1 \wedge \ldots \wedge X_m=t_m \wedge Y_1\neq u_1 \wedge \ldots \wedge Y_n\neq u_n)$

where

- $Z_1, \ldots, Z_p$ are the variables occurring in v^+ and not in Svars
- $V_1, \ldots, V_q$ are the variables occurring in v^- but neither in v^+ nor in Svars.

In particular, it is reduced to the empty conjunction (interpreted as true) if v is ({},not{}). ♦

The set of variables Svars is used to further postpone the quantification of some variables. This allows the Form interpretation to be used with different quantifications of those variables. For instance, assuming v is the only n-substitution computed by p(X), the question

Is there X such that $\leftarrow$p(X) ?

can be answered by answering the equivalent question using the interpretation Form(v;{X}):

Is there X such that $Ax_= \models \exists X$ Form(v;{X}) holds

where $Ax_=$ denotes the usual axioms of equality. Moreover, the request

Find all X such that p(X)

can be equivalently treated by handling the query

Find all terms t for X such that $Ax_= \models \forall($ Form(v;{X}){X/t}). [1]

Safety of the quantification of variables in n-substitutions is handled through the notion of consistency.

Definition 2.16 An n-substitution v is *consistent with respect to (wrt) a set of variables* Svars iff any binding Y/u of v^- verifies the following properties :

(i) Y is a variable of Svars or of v^+;

(ii) if u is a variable then u is a variable of Svars or of v^+. ♦

The following theorem relates the consistency notion and the Form interpretation.

Proposition 2.17 ([Jacquet, 1989]) Let Svars be a set of variables and v be an idempotent n-substitution. Then v is consistent wrt Svars iff the relation $Ax_= \models \exists_{Svars}$(Form(v;Svars)) holds. ♦

One desired property is that consistency is simultaneously achieved by variant n-substitutions. This is indeed the case for idempotent n-substitutions.

Proposition 2.18 ([Jacquet, 1989]) Let Svars be a set of variables. All variant idempotent n-substitutions are simultaneously consistent wrt Svars.

As n-reconciliation manipulate sets of n-substitutions, it is interesting to extend the above Form interpretation and consistency notions to sets of n-substitutions. This is achieved as follows.

Definition 2.19 A set of n-substitutions $\{v_1,\ldots,v_m\}$ is *consistent wrt a set of variables* Svars iff, at least, one v_i is consistent wrt Svars. ♦

Proposition 2.20 ([Jacquet, 1989]) Let Svars be a set of variables and $\theta\oplus\{\omega_1,\ldots,\omega_m\}$ be an idempotent es-nsubst. Then $\theta\oplus\{\omega_1,\ldots,\omega_m\}$ is consistent wrt Svars iff the relation

$$Ax_= \models \exists_{Svars} [\, \text{Form}((\theta,\omega_1);\text{Svars}) \vee \ldots \vee \text{Form}((\theta,\omega_m);\text{Svars} \,]$$

holds. ♦

Proposition 2.21 ([Jacquet, 1989]) Let Svars be a set of variables. All variants idempotent en-subst's are simultaneously consistent wrt Svars. ♦

1 The notation $\exists_{Set}$ (F) (resp. $\forall_{Set}$ (F)) is used as a shorthand to denote the formula $(\exists X_1 \ldots \exists X_m)$ (F) (resp. $(\forall X_1 \ldots \forall X_m)$ (F)), where $X_1, \ldots, X_m$ are the variables of Set. The notation $\exists$(F) (resp. $\forall$(F)) is used to denote the existential (resp. universal) closure of the formula F.

F. Quantified reconciliation

Example 2.1 shows that quantification of the variables must be introduced in the reconciliation process. This is achieved as follows.

Definition 2.22 Let Svars be a set of variables. The n-substitutions $v_1, \ldots, v_m$ are *qn-reconcilable wrt Svars* iff

(i) they are n-reconcilable

(ii) any idempotent en-reconciliation is consistent wrt Svars.

In this case, any such en-reconciliation, simplified from the n-substitutions that are not consistent with respect to Svars, is called a *qn-reconciliation* of $v_1, \ldots, v_m$ wrt Svars. Qn-reconcilation is extended to sets of n-substitutions by analogy to the n-reconciliation of Definition 2.14 ♦

It is worth noting that Proposition 2.21 ensures that all idempotent en-reconciliations simultaneously verify condition (ii) if one of them does. The consistency test can thus only be performed on one of them.

Example 2.2 Returning to the Example 2.1, it is worth noting that the n-substitutions ({},not{X/f(Y)}) and ({X/f(3)},not{}) are not qn-reconcilable wrt {X}. ♦

G. Negation of n-substitutions

Reducing negative literals constructively requires to negate n-substitutions in some way. This is performed according to their equational interpretation.

Definition 2.23 The *negation of the n-substitution*

$v=(\{X_1/t_1,\ldots,X_m/t_m\},\text{not}\{Y_1/u_1,\ldots,Y_n/u_n\})$

is the set of n-substitutions obtained by associating

- the n-substitution $(\{\},\text{not}\{X_i/t_i\})$ with each binding X_i/t_i, $1\leq i\leq m$,
- the n-substitution $(\{X_1/t_1,\ldots,X_m/t_m\}\circ\{Y_j/u_j\},\text{not}\{\})$ with each binding Y_j/u_j, $1\leq j\leq n$.

It is denoted by *neg(v)*. ♦

Appealing to our equational interpretation of n-substitutions, it might seem quite natural to define the negation of the n-substitution $(\{X_1/t_1,\ldots,X_m/t_m\},\text{not}\{Y_1/u_1,\ldots,Y_n/u_n\})$ as the set of n-substitutions

$\{ (\{\},\text{not}\{X_1/t_1\}), \ldots, (\{\},\text{not}\{X_m/t_m\}), (\{Y_1/u_1\},\text{not}\{\}), \ldots, (\{Y_n/u_n\},\text{not}\{\}) \}.$

Once again, this does completely ignore the quantification of variables. Consider, for instance, the n-substitution $v=(\{X/f(Y)\},\text{not}\{Y/3\})$ of the first query $\leftarrow p(X)$ of Example 2.1. Its associated Form(v;{X}) interpretation is

$\exists Y : X=f(Y) \wedge Y\neq 3.$

Negating this n-substitution as above would lead to the two n-substitutions ({},not{X/f(Y)}) and ({Y/3},not{}) with the interpretation

$\forall Y : X\neq f(Y),$

$\exists Y : Y=3,$

respectively. Assigning the value f(2) to X would thus verify the negation of

$\neg\ (\exists Y : X=f(Y) \wedge Y\neq 3)$!

The problem is that the variable Y has lost its relation with X in the second n-substitution. This is circumvented by composing the positive part with each binding of the negative part (as achieved in Definition 2.23).

Following this remark, it may then be strange to find out that the variables $X_1, ..., X_m$ receive no special treatment. This is not necessary for our purposes since we can manage (and we do in fact) so that the quantification of the X_i's is always out of the scope of the considered n-substitutions.

Ns-goals will generally have to negate sets of n-substitutions rather than single n-substitutions. Negation should thus be extended to sets of n-substitutions. This is achieved by considering n-substitutions of sets as as many alternatives. Any set of n-substitutions is then negated by negating each n-substitution separately and by combining the resulting n-substitutions. The last operation implies reconciliation. This induces two possible acceptations according as quantification is taken into account or not. We could then have defined two negations, one involving quantification and the other not. The latter is however useless for our purposes. We will thus only describe the former.

Definition 2.24 A set of n-substitutions $\{\nu_1, ..., \nu_m\}$ is *negatable wrt the set of variables Svars* iff one of the two following conditions holds :

- the set is empty
- the set is not empty and the sets of n-substitutions $neg(\nu_1)$, ..., $neg(\nu_m)$ are qn-reconcilable wrt Svars.

In the first case, the n-substitution ({},not{}) is called the *negation* of the empty set. In the last case, any qn-reconciliation is called a *negation of* $\{\nu_1, ..., \nu_m\}$ *wrt Svars.* ♦

H. Restrictions of n-substitutions

Restriction of substitutions is finally extended to the n-substitutions. This is achieved subsequently in two ways. A first restriction, called equational restriction, rests on our equational interpretation of n-substitutions. It is however too strong to subsume the usual restriction of substitutions. To that end, another restriction, named functional restriction, is proposed. It rests on a functional perception of the substitutions (from which the usual restriction of substitutions issues).

Definition 2.25 Let $\nu=(\{X_1/t_1,...,X_m/t_m\},not\{Y_1/u_1,...,Y_n/u_n\})$ be an n-substitution and Svars be a set of variables.

1) The equational restriction $\nu /_e Svars$ of ν to Svars is the n-substitution μ

- whose positive part μ^+ is obtained from $\{X_1/t_1,...,X_m/t_m\}$
  1. by removing any binding X_i/t_i such that $X_i \notin$ Svars,
  2. by removing any binding X_i/t_i such that $X_i \in$ Svars and t_i is a variable not in Svars,
  3. by replacing any occurrence of any t_i pointed out in 2 by one of the X_j such that X_j/t_i is of the form pointed out in 2.

- whose negative part μ^- is obtained from the version of $\{Y_1/u_1,\ldots,Y_n/u_n\}$, updated as indicated in 3. above, by removing any binding Y_i/u_i that verifies one of the two following conditions [1]:
  (i) $Y_i \notin$ Svars, $Y_i \notin$ varcod(μ^+)
  (ii) u_i is a variable, $u_i \notin$ Svars, $u_i \notin$ varcod(μ^+).

2) The *functional restriction* $v_{|_f Svars}$ of v to Svars is the n-substitution μ
- whose positive part μ^+ is obtained from $\{X_1/t_1,\ldots,X_m/t_m\}$ by removing any binding X_i/t_i such that $X_i \notin$ Svars;
- whose negative part μ^- is obtained from $\{Y_1/u_1,\ldots,Y_n/u_n\}$ by removing any binding Y_i/u_i that verifies one of the two following conditions :
  (i) $Y_i \notin$ Svars, $Y_i \notin$ varcod(μ^+)
  (ii) u_i is a variable, $u_i \notin$ Svars, $u_i \notin$ varcod(μ^+). ♦

Example 2.3 The strength of the equational restriction over the functional one can be illustrated by the following example. Let v be

$(\{X_1/f(Z), X_2/Y_1, X_3/g(Y_1),X_4/h(Y_2)\},not\{Y_1/1,Y_2/2\})$

and Svars be $\{X_2,X_3,X_4,Z\}$. The equational restriction $v_{|_e Svars}$ is

$(\{X_3/g(X_2),X_4/h(Y_2)\},not\{X_2/1,Y_2/2\})$.

whereas the functional restriction $v_{|_f Svars}$ is

$(\{X_2/Y_1,X_3/g(Y_1),X_4/h(Y_2)\},not\{Y_1/1,Y_2/2\})$.

The former results manifestly from a stronger restriction than the latter. Discarding the negative part, this example also shows that the equational restriction is too strong to be a generalization of the usual restriction of substitution. The functional restriction corresponds in fact to that restriction. ♦

Definition 2.25 is extended to sets of n-substitutions in a straightforward manner.

Definition 2.26 The equational (resp. functional) restriction of the set of n-substitutions Θ to the set of variables Svars is the set of the equational (resp. functional) restriction of the n-substitutions of Θ to Svars. ♦

2.4 Extending terms, unification and instantiation

The instantiation of a term by a n-substitution could be defined in two ways :
- by ignoring the negative part of the n-substitution and by calling instantiation of the term its usual instantiation by the positive part of the n-substitution;
- by keeping the negative part and by calling instantiation of the term the above instantiation coupled with the negative part of the n-substitution.

The second solution is adopted hereafter in order to conserve as much information as possible. It will be useful, in the execution model, in order to detect useless reductions as early as possible. It has two consequences :
- any expression must be coupled with negative information;
- unification and instantiation must be extended to such a generalized expression.

[1] Given a substitution θ, the notation varcod(θ) is used to denote the set of the variables of the codomain of θ.

Definition 2.27 On a point of terminology, the association (E,Ω) of an expression E with a set of en-substitutions Ω verifying the following property P is called an *extended expression* :

P : any Y/u of any en-substitution of Ω verifies the following conditions

i) Y occurs in E,

ii) if u is a variable then it should appear in E ♦

The intuition behind extended expressions $(E,\{\omega_1,\ldots,\omega_m\})$ is to constrain E to at least one ω_i, with the variables of ω_i not in E universally quantified. For instance, (h(X,Y),{not{X/f(Y)},not(X/g(Z)}) represents the term h(X,Y) restricted by one of the following constraints :

i) X differs from f(Y) (i.e. $X \neq f(Y)$)

ii) X differs from a 1-ary term which functor is f (i.e. $X \neq f(Z)$, $\forall Z$)

It is furthermore worth noting that, as for n-substitutions, the condition P is used to force the extended expressions to be presented in a normalized form.

Unification is generalized to extended expression by means of the classical unification and the qn-reconciliation.

Definition 2.28 Two extended expressions (E,Θ) and (F,Ψ) are said to be unifiable iff the two following conditions hold :

- E and F are unifiable, say with the idempotent mgu θ,
- the sets of n-substitutions $\{(\theta,\text{not}\{\})\}$, Θ and Ψ are qn-reconciliable wrt the variables of E and F.

Any resulting qn-reconciliation, if any, is called an *n-mgu* of (E,Θ) and (F,Ψ). As Θ and Ψ are composed of en-substitutions, one of them can be expressed in the form $\theta\oplus\Omega$ where Ω is a finite set of en-substitutions. Let Ω_r be the functional restriction of Ω to the variables of $E\theta$. The term $(E\theta,\Omega_r)$ is defined as the *most general common instance* of (E,Θ) and (F,Ψ) wrt the n-mgu $\theta\oplus\Omega$. Finally, this unification is called *extended unification*. The extended unifiability character can be proved independent from the choice of the idempotent mgu θ. ♦

Finally, the instantiation concept is defined for extended expressions and n-substitutions by means of qn-reconciliation. Some auxiliary set of variables is here taken into account in the aim of allowing some variables to be considered as existentially quantified. Note that, in contrast with the usual instantiation, the extended instantiation does not always succeed. Note also that the result, when it exists, is an extended expression. There is thus no need to introduce new expressions generalizing, in their turn, the extended expressions, and, consequently, there is no need to generalize once more the extended instantiation and unification.

Definition 2.29 An extended expression (E,Ω) is said to be *instantiable by an n-substitution* ν wrt a set of variables Svars iff the sets Ω and $\{\nu\}$ are qn-reconcilable wrt the variables of E and Svars. In this case, any resulting qn-reconciliation is called an instantiation n-substitution. Let $\psi\oplus\Psi$ be one of them and let Ψ_r be the set of the following restriction φ_r of the en-substitution φ of Ψ : for any $\varphi\in\Psi$, the corresponding restriction φ_r is defined from φ by removing any binding Y/u that verifies one of the following properties :

i) Y is not a variable of $E\psi$

ii) Y is a variable of $E\psi$, u contains one variable of Svars not occurring in $E\psi$.

Then, the extended expression $(E\psi, \Psi_r)$ is defined as *the instance of (E,Ω) by v wrt the instantiation n-substitution $\psi \oplus \Psi$ and the set of variables Svars.* Given, some set Ψ of en-substitutions, some (non-extended) expression E and some set of variables Svars, the set Ψ_r determined as above, but by taking E instead of $E\psi$, is called the term restriction of Ψ to E wrt to Svars.♦

3 A parallel execution model with constructive negation

We are now in a position to show how the concepts of Section 2 can be used to design a parallel execution model of general Horn clauses. This model has, in its turn, been employed in the design of a concurrent logic programming language, named Conclog ([Jacquet, 1989]). To understand our motivations and the resulting model, it is worth spending a few words on it.

The Conclog language has been created with the aim of expressing concurrent executions in the conventional logic programming framework while standing as close as possible to the ideal of logic programming. Soundness and completeness properties have been ensured as much as possible. Multi-directional and multi-solution procedures are also supported. Nevertheless, efficient procedures can be coded thanks to the introduction of control annotations. To get such properties, the Conclog language has been designed in three steps. A parallel execution model of Horn clauses has first been conceived. It is sound and complete. Negation has then been integrated so as to preserve these properties as much as possible. Finally, annotations and built-in primitives have been introduced for purposes of optimization and practicability, respectively. We report hereafter the result of the second design phase. For the sake of space, the conceptual features are only presented. The reader is referred to [Jacquet, 1989] for more implementation details as well as for more information about the language.

3.1 Overview

As a snapshot, the main characteristics of the Conclog model are as follows. It uses both or-parallelism and and-parallelism in essentially an unrestricted way. Hence, all the clauses unifiable with a ps-goal are used simultaneously to reduce the ps-goal. Furthermore, conjoined s-goals are evaluated in parallel even if they do share variables. Quantified reconciliation is then employed to combine the produced n-substitutions.

The reconciliation calculus is also used intermittently to restore consistency in the deductions as well as to propagate bindings from subtrees to others. In operational terms, this means that the generation induced by the or- and and-parallelism is stopped after some amount of reduction steps and resumed on a purified version of these reductions. The purpose of this operation is to prevent the computation from useless reductions as soon as possible.

Negation is introduced by means of the not(.) predicate. Its argument is defined as a goal in order to conserve the alternation of goals and s-goals in the reductions. Its evaluation is performed according to Definition 2.24 and with respect to the variables of the ns-goal under consideration and the set of n-substitutions returned by the reduction of the associated goal.

As a final major characteristic, extended literals, extended unification and extended instantiation are used to avoid useless computations as early as possible.

The Conclog execution model of general Horn clauses is most easily further explained by using two complementary views, called the tree view and the process view. They are complementary in the sense that the former depicts the computation from a global perspective in terms of trees whereas the latter gives a more detailed and more dynamic view of the computation in terms of the behavior of processes.

3.2 The tree view

In the tree view, any computation is seen as the progressive construction, slice by slice, of the and/or/not search tree induced by the query and the program under consideration. This is achieved by means of a sequence of cycles, each one composed of one generation phase followed by one reconciliation phase. The aim of the generation phase is to extend the already constructed part from one slice. The aim of the reconciliation phase is twofold :

- to produce newly constructed solutions (i.e. n-substitutions corresponding to successful derivations of the general model, newly ended),
- to prevent the execution from useless computations by
  - cutting off branches that are detected to participate to no solution subtree (i.e. subtrees corresponding to the successful derivation of the general model),
  - communicating bindings (including negative ones) from one subtree to another.

The last operation is called *binding publication.*

The computation then consists of starting this sequence of cycles with the and/or/not search tree reduced to its query-node and of ending it when the and/or/not search tree is completely constructed.

The depth of the slices constitutes a parameter of the model. Precisely, it indicates the number of reduction steps that the reduction of any node of the slice can engender. (In this number, the reduction of an ns-goal to its positive goal is counted as one derivation step). A family of execution models is thus in fact defined. All of them have been proved sound and as complete as possible ([Jacquet, 1989]).

3.3 The process view

The real computation is far more dynamic. It is captured more closely by the process perception of the computations. According to it, the computation is described in terms of the behavior of processes, associated with nodes of the and/or/not search tree in a one-to-one mapping. The life of a process basically consists of creating its children as concurrent processes, of waiting for them to report sets of n-substitutions[1], of performing some reconciliation procedure (based on the qn-reconciliation) and of sending incrementally the set of the resulting substitutions to its father process or, for the process associated with the query, delivering them as computed answer substitutions.

[1] By abuse of language, we also say that the processes send n-substitutions instead of sets of n-substitutions.

Processes associated with the tips of one slice or of the whole tree make exception. They do not create children but directly report the following result :

i) processes associated with the last step of a failed derivation report failure by sending the empty set of n-substitutions;

ii) processes associated with other tip nodes report (the set composed of) the restriction, to the variables of their father, of their associated en-mgu.

Tip processes associated with non completed reductions are re-activated to resume their process creation once the process associated with the query has completely performed its reconciliation procedure.

To complete the scheme, let us briefly comment on process creation, process reconciliation, process killing and binding publication. As a point of terminology, we will, from now on, call ps-goal-processes, ns-goal-processes, s-goal-processes, goal-processes and query-process the processes associated with ps-goals, ns-goals, s-goals, goals and the query, respectively.

A. Process creation

Process creation is performed in order to achieve and-parallelism, or-parallelism and the constructive form of negation. It reflects rules (E_1) to (C_2) above. Any goal-process thus creates a process for each of its subgoals, as restricted by rule (C_1). All of them are launched as concurrent processes. Ps-goal-processes search for unifiable clauses (in the extended sense) and create, for each of them, a process for the induced instance of the body. All these processes behave also concurrently. Furthermore, they register the associated en-mgu and the variables introduced at this point in the execution. Finally, any ns-goal-process, say associated with the ns-goal not(G), creates a goal-process for its argument G.

B. The reconciliation procedure

The reconciliation procedure of tip processes is performed according to points i) and ii) above. The reconciliation procedure of non-tip processes is as follows. It essentially rephrase rules (E_1) to (C_2) but deviates in making the restriction and composition in a slightly different way, of ordering the killing of some process and of publishing some bindings. N-substitutions are furthermore not transmitted alone but in so-called R-triplets. Such a R-triplet consist of a triplet composed of an n-substitution, of a label of value either "completed" or "incompleted", stating that the n-substitution is associated with a completed or incompleted derivation, and of a set of the tip-processes associated with the derivation. The label and set of processes information associated with n-substitutions issued from tip-processes is determined straightforwardly from this characterization. With respect to other processes, it is determined as indicated below.

1) Reconciliation procedure of ps-goal-processes

The reconciliation procedure of a ps-goal-process is performed with the intuition that its children represent alternative ways of reducing its associated ps-goal. It thus simply transmit the R-triplet sent by those children.

2) Reconciliation procedure of ns-goal-processes

The reconciliation procedure of an ns-goal is performed according to its intuitive understanding of negator. Any ns-goal-process first collects all the n-substitutions sent by its goal-process child and then negate the set of those corresponding to completely constructed subtrees (the negation is performed wrt the variables of its associated ns-goal). Two particular cases are worth noting :

1) if one of the reported n-substitutions is ({},not{}) then the empty set of n-substitutions is reported. In this case, failure is thus reported by the ns-goal-process.
2) if no n-substitution is reported (i.e. if the child goal-process fails) then the only ({},not{}) n-substitution is reported. In this case, the ns-goal-process succeeds.

N-substitutions are also sent in R-triplets. The auxiliary information attached to the μ_i's is as follows :

- the set of process identifiers part reduces to the ns-goal-process under consideration, say Proc.
- the label part is "completed" if the subtree engendered by Proc is completely constructed. It is "incompleted" otherwise.

Note that the set of n-substitutions sent by the ns-goal cannot be incrementally constructed as the n-substitutions are received from its goal-process child. Those latter n-substitutions are thus in fact collected by the ns-goal-process before being negated. However, in case the ({},not{}) n-substitution is received together with the complete label, failure can be reported directly without waiting for other n-substitutions. This is indeed achieved as an optimization in Conclog. Finally, partial information cannot be taken into account and is eliminated from the negation process. As a convincing argument of this rejection, consider a derivation with partial result ({},not{}) for one prefix that fails in a subsequent prefix.

3) Reconciliation procedure of a goal-process not associated with the query

Goal-processes not associated with the query form (incrementally) the cartesian product of the sets of n-substitutions sent by their children and for each tuple try to qn-reconcile them wrt to the variables of the goal. For any successful qn-reconciliation, the following n-substitution is sent. Let T be the considered tuple, ν be the substitution resulting from the reconciliation, $\theta \oplus \Theta$ be the en-mgu associated with the treated goal-process and Vars be the set of variables of the s-goal associated with the father process[1]. Then the equational restriction of the composition $\theta \mathbin{\bar{\circ}} \nu$ to the variables of Vars is sent. The label and set of nodes accompanying it are as follows. The label is "completed" if the n-substitution of T are associated with a "completed" label; it is "incompleted", otherwise. The set of nodes is just the union of the set of nodes appearing in T.

For the ease of the discussion, the n-substitutions ν resulting from the reconciliation of tuples are subsequently called n-reconciliation-substitutions.

1 This set can be determined thanks to the set of variables associated with the goal-processes or more simply by the a suitable naming of variables.

N-reconciliation of distinct tuples may deliver the same results. Repetitions are avoided by eliminating those duplicates as follows : any goal-process records the R-triplets sent by its children. R-triplets are registered only if they do not correspond to an already sent R-triplet.

Finally, when all n-substitutions sent by all children have been registered, the goal-process order the killing of the children processes that are registered in one received R-triplet but participate to no sent R-triplet. It also orders the publication of the n-substitution

- whose positive part collects the bindings X_i/t_i that verify the following properties :
  - they are common to the positive part of all the n-reconciliation-substitutions
  - their LHS variable X_i is a variable introduced in the reduction at the goal-process or afterwards
  - their RHS term t_i is a non-variable term.
- whose negative part collects the bindings Y_j/u_j that verify the following properties :
  - they or their inversion u_j/Y_j are common to the negative part of the n-reconciliation-substitutions
  - their LHS variable Y_j is a variable introduced in the reduction at the goal-process or afterwards
  - they are not registered with their inversion i.e. there are no bindings Y_p/u_p and Y_q/u_q such that $Y_p=u_q$ and $u_p=Y_q$.

For the correctness of subsequent instantiation, the published n-substitution is sent together with the set of variables of the goal under consideration. The binding publication order is progressively transmitted from father processes to their children processes. What this induced is made precise in a moment.

4) Reconciliation procedure of the query-process

The query-process reconciles the n-substitutions from their children in the same way but delivers n-substitutions in a slightly different way. N-reconciliation-substitutions whose associated label are all "completed" only engender answer n-substitutions. They consist of their equational restriction to the variables of the query. Duplicates due to reconciliation are here avoided directly by memorizing the corresponding R-triplets and by discarding newly computed R-triplets already registered.

Binding publication and process killing are furthermore ordered in the same way as goal-processes by the query-process once it has completely treated all the R-triplets sent by its children. Finally, tip-processes corresponding to unreduced goals are woken up for a new generation phase.

C. Binding publication

Binding publication consists essentially of transmitting, in a given subtree, bindings (even negated ones) that are known to be verified by any answer n-substitution issued from the subtree. It acts in two ways : by constraining subsequent reductions that are incompatible with the published bindings and by killing some reduction whose prefix is incompatible with the published bindings. This incompatibility may have two sources :

- the extended instantiation of the goal or s-goal by the published n-substitution does not succeed;

- the published n-substitution is not consistent with the associated en-mgu.

Precisely, binding publication is operated as follows. Let v be the n-substitution made public, Svars be its associated set of variables and P be the process receiving the publication message.

- If P is a(n) (extended) goal-process then the two following tests are operated. Let G be the goal associated with P.
    - The n-substitution v is tested for qn-reconciliation with the en-mgu of P wrt to the variables of G and of Svars,
    - Each s-goal of G is tested for instantiability with v with respect to the variables of Svars.

  Two different behaviors arise from the issues of the tests.
    - In case the two tests succeeds then G is replaced by the induced instantiation. The publication is passed thereafter to the child processes of P.
    - In case one of the tests fails, then a failure message is reported to the father process of P. Process P then orders the killing of its child processes and commits suicide.
- If P is a(n) (extended) s-goal-process then instantiation of its (extended) s-goal with v is ensured by the previous instantiation of its father process. The s-goal is then replaced by the corresponding instantiation and the instantiation message is transmitted to the child processes of P.

D. Handling process killing and failure messages

Lack of space prevents us from describing the handling of failure messages and of process killing in all details. However the following rough description should be sufficient. Roughly speaking, process killing implies the real killing of the process as well as the following actions. There are, in fact, two types of killing. One involves the descendants and ancestors of the process to kill; the other involves the descendants only. The former is called all_killing (or a_killing, for short) and the other is called desc_killing (or d_killing, for short). Killing, ordered as a consequence of reconciliation (see point 3 above), is of the first kind whereas killing induced by failure messages or binding publication is of the second kind.

All_killing is handled as follows :

- the query-process reports failure of the computation if one of its children is killed and d_kills other children,
- when it is told of the killing of one of its children, any other goal-process commits suicide, d_kills its other children and reports its killing to its father,
- any ps-goal-process collects the killing reports of its children; when all children have been killed, it commits suicide and reports this killing to its father,
- report of a killing message has not to be defined for ns-goal-processes since it can be proved that, thanks to their reconciliation procedure, ns-goal-processes never receive them.

Desc_killing is handled as follows : when it is ordered to d_kill, the process just commits suicide and orders its children to d_kill. Finally, failure messages are treated as follows :

- any ns-goal-process treats failure report by reporting definite success,

- the query-process reports failure of the computation as a report of the failure of one child,
- any other goal-process reports failure in answer to the report of failure and d_kills all child processes,
- any ps-goal-process collects the failure reports and just reports failure when all its children did,
- any ns-goal-process treats failure report by reporting definite success.

To conclude, it is worth pointing out that because of process killing, some subtree may move from the incompletely constructed state to the completely constructed ones. Transmission of special messages are provided in Conclog for that purpose. Lack of space prevents us from detailing that point here. We refer the interested reader to [Jacquet, 1989] for more information.

3.4 Properties

The Conclog parallel execution model just presented has been proved sound with respect to the completion understanding of the programs in [Jacquet, 1989]. Assuming that all the derivations issued from the involved negative literals are finite, it has also been proved complete there. As this hypothesis on the negative literals is the more general situation where the negation as failure rule can be proved complete, the Conclog model can thus be said as complete as possible when the negation as failure rule and the resolution rule are used. Retricted to Horn clause programs, it has also been proved sound and complete (with respect to the classical first order models of the program).

Constructiveness of the negation is another major characteristic. In the Conclog model, the negative literals compute n-substitutions and thus act symmetrically to the positive literals. Consequently, the floundering problem is of no concern in Conclog.

It is also worth pointing out that, although they have been introduced for theoretical purposes (see Proposition 2.4), n-substitutions turn out to be very elegant and very intuitive. As an illustration, let us consider the efface procedure ([Deville, 1990]) :

```
efface(X,[X|L],L).
efface(X,[H|L],[H|L_eff]) ← not(X=H), efface(X,L,L_eff).
```

The relation it computes is defined as follows : efface(X,L,L_eff) holds iff X occurs in L and L_eff is L where the first occurrence of X has been removed. Consider the query efface(X,L,[1,2]). Intuitively, the answers are

L=[X,1,2],

L=[1,X,2] with X≠1,

L=[1,2,X] with X≠1, X≠2.

They are indeed found through the n-substitutions :

({L/[X,1,2]},not{}),

({L/[1,X,2]},not{X/1}),

({L/[1,2,X]},not{X/1,X/2}).

Note that, in contrast, Prolog systems have a very poor behavior for such a query. They indeed only produce the first answer. Most of them - Prolog, in particular - fails in evaluating the negative literal not(X=H) because X is a variable. Then, the reduction of X=H succeeds and implies the failure of not(X=H). Other dialects such that Nu-Prolog ([Naish, 1985]) or Prolog II ([Giannessini et al., 1986]) suspends infinitely that reduction until X becomes non-variable.

The interest of the slice by slice construction of the and/or/not search tree and its parameterization in the Conclog model should finally be stressed. Fixing this parameter to 1 allows to simulate co-routining. Fixing it to a finite value allows to make an incremental form of constructive negation. Fixing it to the infinite value delivers a model where no intermittent restoring of consistency take place. Besides this modelling quality, the essential advantage of the slice by slice construction is to lighten the execution from the computation of useless branches. Its drawback is however to add some extra computation, issued from the reconciliation phases. An interpreter has been made for the Conclog model which allows to perform some tests. However, the optimal value of the depth parameter remains to be determined.

4 Conclusion and future work

A new constructive form of the negation-failure-rule related to concurrent logic programming has been presented. Based on it, a parallel execution model of general Horn clauses has also been exposed. It is sound wrt the completion understanding of the programs and as complete as possible when the resolution rule and the negation-as-failure rule are used. Restricted to Horn clauses, it is both sound and complete. Thanks to its constructiveness character, negative literals are reduced even if they are not ground. Furthermore, their reduction computes bindings for variables as the reduction of positive literals does. Hence, computations cannot flounder.

This new form of negation and this execution model take profit of a reconciliation-approach to concurrency and are based on an equational interpretation of substitutions. A generalization of substitutions, called n-substitutions, has resulted from the natural requirement of representing solutions of systems of equations and inequations in finite terms. They consist of coupling negative information with the substitutions. Despite their theoretical introduction, they have been argued to be quite intuitive and to provide a quite elegant representation of answers to queries. A generalization of the theory of substitutions to n-substitutions has been sketched in this paper.

In a symmetric way, negative information has been coupled with terms. The related notions of unification and instantiation have then also been extended.

The work presented in this paper issued from the second step of the design of a concurrent logic programming language, named Conclog. In addition to the language design purpose, we believe that the execution model has also a theoretical interest arising from their truly concurrent and constructive nature. Our future research will reflect these dual aspects of practicability and theory. They will include further developments of the theory sketched above, semantics of concurrent logic programming languages including this new constructive form of negation and practical issues of implementation of the Conclog language, in particular of the parallel execution model of general Horn clauses.

We refer the reader to the introductory Section 1 for a comparison of our work with related one.

5 Acknowledgements

I wish to acknowledge A. van Lamsweerde for having initiated my research in concurrent logic programming and for his valuable remarks on earlier versions of this work. I also thank the so many people at the University of Namur and at the Centre for Mathematics and Computer Science as well as the anonymous referees for their interest and comments on it. Finally, I like also to thank the Belgian National Fund for Scientific Research and the Centre for Mathematics and Computer Science for having supported this research.

6 References

[Apt, 1990]
APT K.P., *Introduction to Logic Programming*, in : J. van Leuwen (editor), Handbook of Theoretical Computer Science, volume B : Formal Models and Semantics, Elsevier and The MIT Press, 1990, pp. 493-574.

[Barbuti et al., 1987]
BARBUTI R., MANCARELLA P., PEDRESCHI D., TURINI F., *Intensional Negation in Logic Programs : Examples and Implementation Techniques*, Proc. TAPSOFT '87, LNCS 250, 1987, pp. 96-110.

[Barbuti et al., 1990]
BARBUTI R., MANCARELLA P., PEDRESCHI D., TURINI F., *A Transformational Approach to Negation in Logic Programming*, Journal of Logic Programming 8, 1990, pp. 201-228.

[Chan, 1988]
CHAN D., *Constructive Negation Based on the Completed Database*, Proc. 5th Conf. on Logic Programming, 1988, pp. 111-125.

[Chan, 1989]
CHAN D., *An Extension of Constructive Negation and its Application in Coroutining*, Proc. of the North American Conference on Logic Programming, 1989, pp. 477-496.

[Clocksin and Mellish, 1981]
CLOCKSIN W.F., MELLISH C.S., *Programming in Prolog*, Springer Verlag, 1981.

[Conery, 1983]
CONERY J.S., *The And/Or Process Model for Parallel Interpretation of Logic Programs*, Ph.D. thesis, University of California, 1983.

[Deville, 1990]
DEVILLE Y., *Logic Programming : Systematic Program Development*, Addison-Wesley, 1990.

[Eder, 1985]
EDER E., *Properties of Substitutions and Unifications*, Journal of Symbolic Computation, 1, 1985, pp. 31-46.

[Giannesini et al., 1986]
GIANNESINI F., KANOUI H., PASSERO R., VAN CANEGHEM M., *Prolog*, Intereditions, 1986.

[Jacquet, 1989]
JACQUET J.-M., *Conclog : a Methodological Approach to Concurrent Logic Programming*, Ph.D. thesis, University of Namur, Belgium, November 1989, to appear as Lecture Notes in Computer Science, Springer-Verlag.

[Kalé, 1987]
KALE L.V., *Parallel Execution of Logic Programs : the REDUCE-OR Process Model*, Proc. 4th Int. Conf. on Logic Programming, May 1987, pp. 616-632.

[Khabaza, 1984]
KHABAZA T., *Negation as Failure and Parallelism*, Proc. Int. Conf. on Logic Programming, 1984, pp. 70-75.

[Lassez et al., 1988]
LASSEZ J.L., MAHER M.J., MARRIOT K., *Unification revisited*, In Minker J. (editor), Foundations of deductive databses and logic programming, Morgan Kaufmann, Los Altos, 1988, pp. 587-626.

[Li and Martin, 1986]
LI P.P., MARTIN A.J., *The Sync Model : A Parallel Execution Method for Logic Programming*, Proc. Symp. on Logic Programming, 1986, pp 223-235.

[Lloyd, 1987]
LLOYD J.W., *Foundation of Logic Programming*, Springer Verlag, 1987.

[Lugiez, 1989]
LUGIEZ D., *A Deduction Procedure for First Order Programs*, Proc. 6th Int. Conf. on Logic Programming, 1989, pp. 585-599.

[Maluszynski and Näslund, 1989]
MALUSZINSKI J., NASLUND T., *Fail Substitutions for Negation as Failure*, Proc. of the North American Conference on Logic Programming, 1989, pp. 461-476.

[Martelli and Montanari, 1982]
MARTELLI A., MONTANARI U., *An Efficient Unification Algorithm*, TOPLAS, vol. 4, No. 2, April 1982, pp. 258-282.

[Naish, 1985]
NAISH L., *Negation and Control in Prolog*, Ph.D. Thesis, University of Melbourne, Australia, 1985.

[Palamidessi, 1990]
PALAMIDESSI C., *Algebraic Properties of Idempotent Substitutions*, Proc. of the 17th ICALP, 1990, pp. 386-399.

[Pollard, 1981]
POLLARD G.H., *Parallel Execution of Horn Clause Programs*, Ph. D. thesis, Dept. of Computing, Imperial College, London, 1981.

[Przymusinski, 1989]
PRZYMUSINSKI T., *On Constructive Negation in Logic Programming*, Proc. of the North American Conference on Logic Programming, 1989.

[Sato and Tamaki, 1984]
SATO T., TAMAKI H., *Transformational Logic Program Synthesis*, Proc. of FGCS, 1984, pp. 195-201.

[Tarnlund, 1977]
TARNLUND S.A., *Horn clause computability*, BIT 17, pp. 215-226, 1977.

[Turi, 1991]
TURI D., *Extending S-Models to Logic Programs with Negation*, to appear in Proc. 8th Int. Conf. on Logic Programming, 1991.

[Wallace, 1987]
WALLACE M., *Negation By Constraints : a Sound and Efficient Implementation of Negation in Deductive Databases*, Proc. Int. Symp. on Logic Programming, 1987, pp. 253-263.