A Theoretical Framework for Consistency Techniques in Logic Programming.

Fascal VAN HENTENRYCK

ECRC, Arabellastr. 17, 8000 Muenchen 81, West Germany

Abstract.

While being a powerful paradigm for solving Constraint Satisfaction Problems (CSPs), Consistency Techniques (CTs) have never been taken into account during the design of declarative programming languages. This paper defines a theoretical framework for using CTs inside logic programming. Three inference rules are introduced and their formal properties are investigated. Also, computation rules are defined which are worth considering wrt the inference rules. As practical results, the programmer can write "generate & test" programs while the interpreter/compiler will use CTs for solving them (e.g forward checking or arc consistency). This makes logic programming not only a good language for stating CSPs but also an efficient tool for solving them as confirmed by our first experiences.

1. Motivation

Our work aims at the integration of CTs inside a declarative programming language in order to solve Constraint Satisfaction Problems (CSPs).

The class of CSPs is of great importance in AI as, for instance, graph colouring, graph isomorphisms and homomorphisms, boolean satisfiability, scene and edge labeling and logical pussies can be seen as particular cases of it. A CSP can be defined in the following way. Assume the existence of a finite set 1 of variables which take respectively their values from finite domains and a set of constraints. A constraint between k variables from 1 is a subset of the cartesian product of the respective variable domains which specifies which values of the variables are compatible with each other. A solution to a CSP is an assignment of values to all variables which satisfies all the constraints and the task is to find one or all the solutions.

CTs are a powerful paradigm for solving CSPs. They should be contrasted to "generate & test", standard backtracking (depth-first search with chronological backtracking) and dependency-directed backtracking. Clearly, "generate & test" is an unfeasible search procedure as soon as the sise of a problem makes it- interesting to consider. "Generate & test" corresponds to the naive way of writing logic programs for CSPs. For instance, a "generate & test" for the 8-queens problems generates a possible configuration for the queens and then test the constraints. Standard backtracking (ST), while being a substantial improvement over "generate & test", leads to a pathological behaviour known as thrashing. In logic programming, ST corresponds generally to a "generate & test" program with coroutining informations or to a program written for testing the constraints as soon as possible. Dependency-directed backtracking [14] has been introduced in order to avoid one of the thrashing symptoms but it is more a remedy to a symptom of the malady and not to the malady itself. Indeed, It is better to prevent failures than to react intelligently to them. More generally, the drawback of standard and dependency-directed backtracking lies in *the way they reduce the search space,* only in an *"a posteriori'* way after having discovered a failure. Contrarily, CTs [4, 11, 13] *prevent JaUuret and reduce the search space* in *an* "a *prion' way* before discovering a failure by removing combinations of values which cannot appear together in a solution. This leads to an early detection of failures and reduces both the amount of backtracking and the number of constraint checks.

The paradigm behind CTs has been the basis for some problem-solvers like HEF-ARF |3j and ALICE [9| but it has never been taken into account in the design of (high-level) programming languages although its importance in this context has been stressed elsewhere |Ilj. *However, they are some inherent interest for using CTs m the design of declarative languages and not to restrict our attention to problem solvers.* Programming languages, contrary to problem-solvers, should lead to a greater *flexibility* with respect to the set of constraints (used to define a CSP) and the strategy (used to solve it). Among declarative programming languages, *logic programming* is very appropriate for integrating CTs due to *its relational form* which makes it an adequate tool for expressing CSPs and *its freedom of control* which makes it adequate for integrating different paradigms.

Our work atrm *at a declarative logic programming language integrating CTs.* The objective is to preserve the expressiveness of logic programming while using the efficiency of CTs. This allows the programmer to write a "generate & test" program for stating his problem while the interpreter/compiler will use CTs for solving it. For this purpose, three new inference rules are introduced, say the forward checking inference rule (the FCIR), the lookahead inference rule (the LAIR) and the partial lookahead inference rule (the PLA1R), which are general mechanisms for using CTs in logic programming. Examples of their uses have been described in |16, 17]. In the second paper, performance measures on several problems have been given which show the feasibility and the importance of the approach. Its main advantage lies in the *duality generality/specialization* and the *total freedom wrt the strategy.* The duality *generality/specialization* comes from the fact that, on one hand, the inference rules are not restricted to a particular set of predefined constraints but can be used for logic programs and, on the other hand, they can be specialised (i.e. built-in) for some constraints by taking into account their particular properties leading to a very efficient handling of these constraints. *The freedom wrt the strategy* comes from the fact that these inference rules can be combined inside the same program (i.e different strategies can be used for different kinds of constraints) This is important as some constraints are more appropriate for a forward-checking use while others are best-suited for a lookahead use. Also, they can be combined either with domain-splitting or instantiation.

The present paper describes the theoretical framework for our approach. We show how SLD-resolution can be extended in

order to use CTs in logic programming. For this purpose, the next section introduces the domain concept and the unification algorithm is extended to handle it. Next, the inference rules are defined and their formal properties are investigated.

Our approach should be related in philosophy to the works on constraints in logic programming (e.g |7, 2]). It differs from them as we are not restricted to a set of built-in constraints. Genera) mechanisms are provided which can be used for any constraint to solve problems in a well-defined class.

2. Domains in logic programming.

2.1. The domain concept.

Domains provides the basis for using CTs in logic programming. It is often the case that variables range over a finite domain but this information cannot be expressed clearly in logic programming languages. *Domain declarations* have been introduced for taking this fact into account (16).

Definition 1: A *domain declaration* for predicate symbol p of arity n is an expression of the following form.

domain $p(a_1,...,a_n)$ where a_i is either h or d..

When a_i is equal to h, this means that the i argument of p ranges over the Herbrand universe. Otherwise, it means that the i argument is a list of variables which ranges over d_1 In the following, the domains di are finite and explicit sets of values (i.e constants).

Definition 2: Let dl,...,dn the domains appearing in the domain declarations of a logic program PR and different from the Herbrand universe. We note D(PR) the set $\{ d \mid d \neq \emptyset \text{ and } d \in 2^{di} \ (1 \leq i \leq n)\}$. We call it the domain set of the logic program. The domain set of a logic program contains all domains we possibly need during the computations.

The resulting language is a first-order language with aggregate variables (6). This means that the domains must be seen as unary relations and an aggregate variable as a variable which ranges over this unary relation. In the following, we refer aggregate variables as d-variables (domain variables), we note x^d a variable x ranging over d and we use $a \in d$ to denote d(a) where a is a constant. Also, we use $\exists P$ and $\forall P$ to denote the existential and universal closures of P. Two rules are added to the usual first-order validity rules. We assume an interpretation 1 and a variable assignment A. A(x/y) is A with x assigned to y and |d| is the unary relation d defines in I. The terms are constructed as usual except that variables can now be usual variables and d-variables.

1. If the formula has the form $\exists x^d F$, then the truth value of the formula is true if there exists $d \in |d|^I$ such that F has truth value true wrt 1 and A(x |d); otherwise its truth value is false.
2. If the formula has the form $\forall x^d F$, then the truth value of the formula is true if for all $d \in |d|^I$ we have that F has truth value true wrt I and A(x |d); otherwise its truth value is false.

The unification algorithm must be extended to take the domains into account. Informally, a d-variable and a constant can only been unified if the constant is in the domain of the d-variable. Also, when unifying a variable and a d-variable, the variable is bound to the d-variable. Finally, two d-variables can only be unified if the intersection of their domains is not empty, in which case they are bound to a new variable whose domain is this intersection.

2.2. Properties of the domain concept.

We now define formally the algorithm and study its properties. We first need some notions.

Definition 3: We say that *the range of t is included in a domain d*, noted $|t| \in d$, if t is a constant $\in d$ or a d-variable x^{d1} such that $d1 \subseteq d$.

Definition 4: An d-substitution θ is a finite set of the form $\{v_1/t_1,...,v_n/t_n\}$, where

1. each v_i is either a variable or a d-variable.
2. t_i is a term distinct from v_i.
3. $v_1,...,v_n$ are all distinct.
4. if v_i is a d-variable v^{d1}, $|t_i| \in d1$.

Definition 5: We say that two d-substitutions θ and λ *agree* on a set V of variables and d-variables, denoted $\theta = \lambda|V|$ iff $x\theta = x\lambda$ for each $x \in V$ where = denotes the syntactic equality.

Definition 6: θ is a d-instance of λ in V, denoted $\lambda \leq \theta$, iff $\theta = \delta \circ \lambda|V|$ for some d-substitution δ. In the following, we use $\lambda\delta$ instead of $\delta \circ \lambda$.

Definition 7: A d-substitution σ is an d-unifier of some non-empty and finite subset $S=\{t_1,...,t_n\}$ where t_i is a literal or a term iff $t_1\sigma =...= t_n\sigma$; we also say that σ unifies S. UNI(S) is the set of all d-unifiers of S. σ is called a *most general d-unifier or d-mgu of S* iff for each $\theta \in$ UNI(S), $\theta \leq \sigma|vars(S)|$ implies $\sigma \leq \theta|vars(S)|$ where vars(S) is the set of all variable or d-variable symbols in S.

In the following, we use substitution and mgu instead of d-substitution and d-mgu. We now present the unification algorithm. In this algorithm, S denotes a finite set of predicates or terms, ϵ the empty substitution and the disagreement set is defined as usual (for instance [10]).

UNIFICATION ALGORITHM.

1. Put k= 0 and $\sigma_0=\epsilon$.
2. If $S\sigma_k$ is a singleton, then stop; σ_k is the substitution returned by the algorithm. Otherwise, find the disagreement set of D_k of $S\sigma_k$.
3. If there exist v and t in D_k such that v is a variable that does not occur in t, then put $\sigma_{k+1}=\sigma_k\{v/t\}$, increment k and go to 2.
4. If there exist v^d and a constant c in D_k such that $c \in d$, then put $\sigma_{k+1}=\sigma_k\{v^d/c\}$, increment k and go to 2.
5. If there exist v^{d1} and w^{d2} in D_k such that $d1 \supseteq d2$ then put $\sigma_{k+1}=\sigma_k\{v^{d1}/w^{d2}\}$, increment k and go to 2.
6. If there exist v^{d1} and w^{d2} in D_k such that $d3 = d1 \cap D2 \neq \emptyset$ then put $\sigma_{k+1}=\sigma_k\{v^{d1}/z^{d3},w^{d2}/z^{d3}\}$ where z^{d3} is a new variable ranging over d3, increment k and go to 2.
7. Otherwise, stop; S is not unifiable.

Clearly, this algorithm terminates because S contains only

finitely many variables and each application of steps 3,..,6 decreases by one the number of variables. We now prove that the unification algorithm does indeed find a tngu of a unifiable set of terms or predicates. It is a generalisation of the usual unification theorem (see for instance (10)).

Theorem S: (Unification theorem).

Let S a finite set of terms or predicates. If 8 is unifiable, then the unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the unification algorithm terminates and reports this fact.

Proof We have already noted that the algorithm always terminates. It suffices to show that if S is unifiable, then the algorithm finds an mgu. In fact, if S is not unifiable, then the algorithm cannot terminate at step 2 and, since It does terminate, it must terminate at step 7. Thus it reports the fact that S is not unifiable.

Assume then that S is unifiable and let *9* be any unifier for S. We prove first that, for $k \geq 0$, if σ_k is the substitution given at the kth iteration of the algorithm, then there exists a substitution γ_k such that $\theta = \sigma_k\gamma_k|vars(S)|$.

Suppose first that $k=0$. Then wt can put $\gamma_0 = \theta$, since $\theta = \varepsilon\theta$. Next, suppose for some $k \geq 0$, there exists σ_k such that $\theta = \sigma_k\gamma_k|vars(S)|$. If $S\sigma_k$ is a singleton, then the algorithm terminates at step 2. Hence we can confine attention to the case when $S\sigma_k$ is not a singleton. We want to show that the algorithm will produce a further substitution σ_{k+1} and that there exists a substitution γ_{k+1} such that $\theta = \sigma_{k+1}\gamma_{k+1}|vars(S)|$. Since $S\sigma_k$ is not a singleton, the algorithm will determine the disagreement set D_k of Sff. and go to Step 3. Since $\theta = \sigma_k\gamma_k|vars(S)|$ and *6* unifies S, it follows that γ_k unifies D_k. Thu D_k ust contain either a variable or a d-variab)e.

Suppose first that D_k contains a variable v and let t another term of D_k. Then v cannot occur In t because $v\gamma_k = t\gamma_k$. We can suppose that $\{v/t\}$ is indeed the substitution chosen at step 3. Thus $\sigma_{k+1} = \sigma_k\{v/t\}$. We now d $\epsilon\gamma_{k+1} = \gamma_k\backslash\{v/v\gamma_k\}$. If γ_k s a binding for v, then

$$\gamma_k = \{v/v\gamma_k\} \cup \gamma_{k+1}$$
$$\gamma_k = \{v/t\gamma_k\} \cup \gamma_{k+1}$$
$$\gamma_k = \{v/t\gamma_{k+1}\} \cup \gamma_{k+1}$$
$$\gamma_k = \{v/t\}\gamma_{k+1} \text{ (by definition of composition)}$$

If γ_k does not have a binding for v, than $\gamma_{k+1} = \gamma_k$, each element of D_k is a variable and $\gamma_k = \{v/t\}\gamma_{k+1}$. Thus $\theta = \sigma_k\gamma_k|vars(S)| = \sigma_k\{v/t\}\gamma_{k+1}|vars(S)| = \sigma_{k+1}\gamma_{k+1}$(vari(S)) , as required.

Now, if D_k docs not contain a variable, it contains a d-variable. The steps 4 and 5 can be handled in the same way as the step 3.

We now consider step 6. Let $v1^{d1}$ and $v2^{d2}$ be the two d-variables in D_k and we suppose that the substitution chosen in this step is $\{v1^{d1}/z^{d3}, v2^{d2}/z^{d3}\}$ where z^{d3} is a new variable. We first note that $v1^{d1}$ and $v2^{d2}$ have bindings in γ_k. Also, since $|v1^{d1}\gamma_k| \in d1$, $|v1^{d2}\gamma_k| \in d2$ and $v1^{d1}\gamma_k = v2^{d2}\gamma_k$, we have that $|v1^{d1}\gamma_k| \in d3$ and $|v2^{d2}\gamma_k| \in d3$. It follows that $\{z^{d3}/v1^{d1}\gamma_k\}$ is a d-substitution. We now define $\delta = \gamma_k \cup \{z^{d3}/v1^{d1}\gamma_k\}$. We have that

1. $z^{d3}\delta = v1^{d1}\delta = v2^{d2}\delta$.
2. $\theta = \sigma_k\delta|vars(S)|$ since s does not occur in σ_k and S.

We finally define $\gamma_{k+1} = \delta\backslash\{v1^{d1}/v1^{d1}\delta, v2^{d2}/v2^{d2}\delta\}$. It follows that

$$\delta = \{v1^{d1}/v1^{d1}\delta, v2^{d2}/v2^{d2}\delta\} \cup \gamma_{k+1}$$
$$\delta = \{v1^{d1}/z^{d3}\delta, v2^{d2}/z^{d3}\delta\} \cup \gamma_{k+1}$$
$$\delta = \{v1^{d1}/z^{d3}\gamma_{k+1}, v2^{d2}/z^{d3}\gamma_{k+1}\} \cup \gamma_{k+1}$$
$$\delta = \{v1^{d1}/z^{d3}, v2^{d2}/z^{d3}\}\gamma_{k+1}$$

Thus, $\theta = \sigma_k\delta\ |vars(S)| = \sigma_k\{v1^{d1}/z^{d3}, v2^{d2}/z^{d3}\}\gamma_{k+1}$ $|vars(S)| = \sigma_{k+1}\gamma_{k+1}\ |vars(S)|$.

Now we can complete the proof. If S is unifiable, then we have shown that the algorithm must terminate at step 2 and, if It terminates at the kth iteration, then $\theta = \sigma_k\gamma_k|vars(S)|$. Since a_k is a unifier of S, this equality shows that it is Indeed an mgu for S. ∇

We refer SLD-resolution with this extended unification algorithm as SLDD-resolution. The notions of SLDD-refutatlon and SLDD-answer substitutions are defined by analogy to the SLD case. The reader can verify easily that the mgu and lifting lemmas as well as theorems 7.1 and 8.4 in |10] hold for SLDD-resolution. Thus, SLDD-resolution is both sound and complete for definite clauses. This result corresponds to the one of [6] which has shown that resolution is sound and complete when extended for handling d-varlablee Also, the same result can be proved if we switch to a many-sorted logic since the domain set augmented by the herbrand universe and the empty set can be organised as a meet-semilattice [18]. The domain concept Is a necessary extension for consistency techniques to be applied but it not in it-self as interesting as other extensions like, for instance, LOGIN [1]. Before presenting the inference rules, we define the constraints we considered.

Definition 9s Let p be a n-ary predicate symbol, p Is a *conttraint* iff for any ground t e r m $t_1,...,t_n$, h e r $p(t_1,...,t_n)$ has a successful refutation or $p(t_1,...,t_n)$ has only finitely failed derivations.

S. Forward checking in logic programming.

Forward checking is often considered as one of the most efficient procedures for solving CSPs. Intuitively, a constraint can be used in forward checking as soon as at most one variable occurs In it. In this case, the set of possible values for the variable Is reduced to the set of values which satisfy the constraint. Thus, a program based on forward checking gives a value to a variable, uses all constraints which contains

at moat one variable, chooses a value for Che next variable, uses the constraint! and to on until all variables have received valuet. If, during the search, a constraint cannot be satisfied (it reduces the set of possible values for a variable to the empty set), the search procedure gives another value to the previously assigned variable. This section defines the FCIR and its use for a general control mechanism, forward declarations, and for the implementation of some built-in predicates. They enable programs previously based on a ST or "generate & test" search to use a forward checking strategy.

3.I. The inference rule.

Definition 10: A literal $p(x_1,...,x_n)$ in the resolvent is *forward checkable* iff p is a constraint and all its arguments are ground but one which is a d-variable. This d-variable is called the forward-variable.

Definition 11: The forward checking inference rule (FCIR).

Let G_i be the goal '$\leftarrow A_1,...,A_{m-1},P,A_{m+1},...,A_k$' and PR be a logic program. G_{i+1} is derived from G_i using mgu θ_{i+1} via PR if the following conditions hold

1. **P is forward checkable and x^d is the forward-variable.**
2. **dnew = {a ∈ d | PR |= P{x/a)} and dnew ≠ {}.**
3. **θ_{i+1} is**
   - **$\{x^d/e\}$ if dnew = {e}.**
   - **$\{x^d/z^{dnew}\}$ where z^{dnew} is a new variable otherwise.**
4. **G_{i+1} is the goal '$\leftarrow(A_1,...,A_{m-1},A_{m+1},...,A_k)\theta_{i+1}$'.**

Since a ground instance of a constraint either succeeds *or* finitely fails, the set dnew in point 3 of the definition can be computed easily (for instance by using SLDD-resolution) . The PCIR can be seen as a general mechanism for enforcing node-consistency [11].

The FCIR provides a theoretical foundation (1) for a general control mechanism, forward declarations, that can be used whatever the kind of constraints to be satisfied and (2) for the implementation of some built-in constraints.

Forward declarations [17] provide a general method for using forward checking inside logic programming. A forward declaration for a predicate symbol p of arity n is an expression of the following form.

forward $p(a_1,...,a_n)$ where a_i is either g either d.

This declaration, which is unique for a particular predicate symbol, specifies that a literal $p(t_1,...,t_n)$ in the resolvent can be selected only when all its arguments corresponding to a 'g' in the declaration are ground and when it is either ground or forward checkable. When it is forward checkable, the FCIR must be used to resolve it; otherwise, normal derivation is applied.

Forward declarations are best combined with a special-purpose computation rule.

Definition 13s A computation rule is efficient wrt the forward declarations, if it selects only a predicate submitted to forward declaration when it is ground or forward checkable and if, whenever the resolvent contains literals submitted to a forward declaration which are either forward-checkable or ground, it selects one of them.

The main interests of a efficient computation rule wrt the forward declarations are for expressiveness and efficiency. Constraints can be stated before the generators and the interpreter/compiler is responsible to select them at an appropriate computation step. Thus, forward declarations will act as preconditions to the selection of a predicate and can be selected as soon as possible for reducing the search space. This Introduces a 'dolo drteen' computation as, for instance, In the constraint language of (15) and a generalised form of forward checking.

Example i Consider the problem of colouring a map in four colors. A logic program for solving this problem will include a constraint different(X,Y) which holds if X and Y are different colors. It can be defined as a finite set of assertions of the form

different(color1,color2) ←
different(color1,color3) ←
different(color1,color4) ←

where color1, color2, color3 and color4 are the constants for the possible colors. Also, the forward declaration 'forward different(d,d)' specifies we want a forward checking use of this predicate. Now, if the resolvent contains a literal 'different(color1,x^d)' with d = {color1,...,color4}, the FCIR will return a binding $\{x^d/z^{d1}\}$ where d1 = d \ {color1}. This prunes the search space in a 'a priori' way by reducing the set of possible values which can be assigned to x^d.

The FCIR provides also a theoretical foundation for the implementation of built-in constraints which are the primitives of the logic language (e.g arithmetic constraints). In usual logic languages, these constraints can only be used for testing values and thus only reduce the search space in an "a posteriori way". However, the FCIR can be specialised for these constraints which now can not only test values but can also prune the search space when only one d-variable is left uninstantiated. Consider a non-equality constraint between integers (i.e $x \neq y$ which holds If x and y are different integers). In usual logic languages, the non-equality predicate is implemented by mean of the "negation as failure" rule and thus can only be selected when both arguments are ground. By redefining this constraint as specialisation of the FCIR, the same pruning as the above different predicate is achieved but In a more efficient way.

3.2. Properties of the FCIR.

We now prove the soundness and completeness of the FCIR. The first two lemmas allow us to "remove" a value from the domain of a d-variable if this value does not satisfy a constraint.

Lemma 13: Let F a conjunction of predicates containing $x_1,...,x_n$ as variables. Let PR a logic program and Mpr its *minimal model*. Then $(\models_{Mpr} \exists F) \rightarrow (PR \models \exists F)$.
Proof: **Let M be a model of PR. We show that ∃F is true in M. If M is a model, we can define a Herbrand model Mh of PR as follows**

$$Mh = \{p(t_1,...,t_n) | p(t_1,...,t_n) \text{ is included in the Herbrand base and is true in M}\}$$

Since $\models_{Mpr} \exists F$, then it is true in all Herbrand models and $\models_{Mh} \exists F$. It follows that there exist term $t_1,...,t_n$ in the Herbrand universe such that $\models_{Mh} F\{x_1/t_1,...,x_n/t_n\}$. By definition of Mh and since $F\{x_1/t_1,...,x_n/t_n\}$ is a conjunction of ground predicates, $F\{x_1/t_1,...,x_n/t_n\}$ is true in M. Thus, ∃F is true in M. ∇
A similar construct has been used in [10].

Lemma 14: Let PR be a logic program, d and dnew be non-empty domains such that d = $\{a_1,...,a_p\}$ and dnew = d\$\{a_1\}$, let P, $P_1,...,P_m$ be positive literals such that P contains a free variable x^d and $\{x^d,x_1,...,x_n\}$ are the set of free variables occurring in P, $P_1,...,P_m$. We note Q the wff $P_1 \wedge ... \wedge P_m$ and F the wff $\exists x^d \exists x_1 ... \exists x_n (P \wedge Q)$. Then

PR $\not\models \exists x_1 ... \exists x_n P\{x^d/a_1\}$ implies $F \equiv_{PR} F\{x^d/x^{dnew}\}$

Proof: Since PR is a set of definite clauses, it admits a minimal model, noted Mpr, which is the intersection of all models of PR. Since we know that PR $\not\models \exists x_1 ... \exists x_n P\{x^d/a_1\}$, it follows that

- PR $\not\models \exists x_1 ... \exists x_n (P\{x^d/a_1\} \wedge Q\{x^d/a_1\})$
- $\not\models_{Mpr} \exists x_1 ... \exists x_n (P\{x^d/a_1\} \wedge Q\{x^d/a_1\})$ (lemma 12)
- $\not\models_{Mpr} F\{x^d/a_1\}$

Hence, the following are equivalent.

- PR $\models$ F
- $\models_{Mpr}$ F
- $\models_{Mpr} F\{x^d/a_1\} \vee ... \vee F\{x^d/a_p\}$
- $\models_{Mpr} F\{x^d/a_1\}$ or ... or $\models_{Mpr} F\{x^d/a_p\}$
- $\models_{Mpr} F\{x^d/a_2\}$ or ... or $\models_{Mpr} F\{x^d/a_p\})$
- $\models_{Mpr} F\{x^d/a_2\} \vee ... \vee F\{x^d/a_p\}$
- $\models_{Mpr} F\{x^d/x^{dnew}\}$
- PR $\models F\{x^d/x^{dnew}\}$ (by lemma 13) ∇

Note that this lemma is no longer true if PR is an arbitrary set of clauses.

Lemma 15: $(\{\neg\exists(Q \wedge P)) \wedge \forall P) \equiv ((\neg\exists Q) \wedge \forall P)$.

Theorem 16: Soundness of the FCIR.

Consider PR be a logic program, G_i a goal '$\leftarrow A_1,...,A_{m-1},P,A_{m+1},...,A_k$' where P is a forward checkable literal and x^d is the forward variable. We assume that d = $\{a_1,..., a_n, b_1,..., b_m\}$ and dnew = $\{b_1,..., b_m\}$ both being not empty. We also assume that $\{x^d,x_1,...,x_l\}$ is the set of all free variables of G_i. We note Q as $A_1 \wedge ... \wedge A_{m-1} \wedge A_{m+1} \wedge ... \wedge A_k$ and restate G_i as $\neg\exists x^d \exists x_1 ... \exists x_l (Q \wedge P)$. We prove that $G_i \equiv_{PR} G_{i+1}$.

Proof:

From point 3 of the definition of the FCIR, we know

1. PR $\not\models P\{x^d/a_1) \wedge ... \wedge PR \not\models P\{x^d/a_n\}$.
2. PR $\models \forall x^{dnew} P\{x^d/x^{dnew}\}$.

Then, by (1) and successive applications of lemma 13, $\neg\exists x^d \exists x_1 ... \exists x_l (Q \wedge P)$ is equivalent in PR to

$$\neg\exists x^{dnew} \exists x_1 ... \exists x_l (Q\{x^d/x^{dnew}\} \wedge P\{x^d/x^{dnew}\}).$$

and thus by (2) and lemma 14, it is equivalent to

- $\neg\exists x^{dnew} \exists x_1 ... \exists x_l ((P \wedge Q)\{x^d/x^{dnew}\}) \wedge \forall x^{dnew} P\{x^d/x^{dnew}\}$
- $\neg\exists x^{dnew} \exists x_1 ... \exists x_l Q\{x^d/x^{dnew}\}$
- $Q\theta_{i+1}$
- G_{i+1} ∇

We now turn to the completeness of the FCIR. We first define an SLDFC-resolution as a proof procedure which uses the FCIR for forward checkable predicates and SLDD-resolution otherwise. We also speak about SLDFC-refutation and SLDFC-aniwer substitution.

Theorem 17: Completeness of the FCIR. Let PR a logic program and G a goal. If there exists a SLDD-refutation of PR$\cup${G} with a SLDD-answer substitution θ, then there exist a SLDFC-refutation of PR$\cup${G} with a SLDFC-answer substitution σ such that $\sigma \leq \theta$.

Proof We prove it by induction on the length of the SLDD-refutation.

Suppose 1 = 1 and G is forward-checkable. Then, let x be the forward checkable variable and *O* the substitution resulting from the application of the FCIR. Then, given any SLDD-answer substitution 0, by definition of the FCIR, there exists a SLDFC-refutation with SLDFC-answer substitution *O* such that $a \leq \theta$.

Suppose the result holds for $l \leq l$ and we show that it holds for l = i+1. Let G = '$\leftarrow A_1 \wedge ... \wedge A_n \wedge P$'. Suppose P is forward checkable, x is the forward variable and $\{x/x^a\}$ is the substitution resulting from the application of the FCIR. Consider a SLDD-refutation of PR$\cup${G} with SLDD-answer substitution θ and suppose θ have a binding for x. Let θ be $\{x_1/t_1,...,x_n/t_n,x/t\}$. Since, by definition of the FCIR, $|t| \in d$, θ can be restated as $\delta = \delta'\delta''$ where $\delta' = \{x/x^d\}$ and $\delta'' = \{x_1/t_1,...,x_n/t_n,x^d/t\}$. It follows that $\theta = \delta|var(G)|$ and thus θ can be viewed as a restriction of δ to the variables of G. Also, PR$\cup\{\leftarrow(A_1 \wedge ... \wedge A_n \wedge P)\delta\}$ has a SLDD-refutation (the same as the one which returns θ except for the unifiers and the initial goal) of the same length with SLDD-answer substitution ϵ. Now, we can construct a SLDD-refutation of PR$\cup\{\leftarrow(A_1 \wedge ... \wedge A_n)\delta\}$(by removing the steps concerning the refutation of Pδ) with SLDD-answer substitution ϵ. By the lifting lemma, PR$\cup\{(A_1 \wedge ... \wedge A_n)\delta'\}$ has a SLDD-refutation with SLDD-answer substitution ψ such that $\psi \leq \delta''$. By induction hypothesis, there exist a SLDFC-refutation of PR$\cup\{\leftarrow(A_1 \wedge ... \wedge A_n)\delta'\}$ with SLDFC-answer substitution σ'' such that $\sigma'' \leq \psi$. It follows that

$$\delta'\sigma'' \leq \delta'\psi \leq \delta'\delta'' = \theta|var(G)|$$

Thus, it exist a SLDFC-refutation of PR$\cup${G} with a SLDFC-answer substitution σ which is a restriction of $\delta'\sigma''$ to the variables of G such that $\sigma \leq \theta$. ∇

4. Lookahead in logic programming.

There exists other CTs which use constraints even when several variables appear in them. Typical examples are lookahead [5] and the arc and path-consistency algorithms [11,13,19]. For instance, given a constraint "C{x,y}", a lookahead use of it consists in eliminating from the domain of x all the values which are not consistent with any value of the domain of y and vice-versa for y. The reduction of the search space is done earlier than in forward checking but its computation tost is higher. Also, the constraint C{x,y} is not

necessary solved by this treatment and mutt be reconsidered later on. The LAIR and lookahead declarations are introduced in logic programming in order to use lookahead whatever the kind of constraints used in the program. Therefore, programs previously based on a ST or "generate & test" search now use Walts filtering-like algorithm or lookahead. This is especially important in areas like vision and qualitative reasoning (8|.

4.1. The inference pule.

Definition 18: A literal $p(t_1,...,t_n)$ in the resolvent is *lookahead checkable* iff p is a constraint and there exists at least one t_i which is a domain-variable and each of the other arguments is either ground or a domain-variable. The domain-variables in $t_1,...,t_n$ are called the *lookahead variables.*

Definition 19: The lookahead inference rule (LAIR). Let G_i be the goal '$\leftarrow A_1,...,A_{m-1},P,A_{m+1},...,A_k$' and PR be a logic program.

G_{i+1} is derived from G_i using mgu θ_{i+1} via PR if the following conditions hold

1. P is lookahead checkable and $x_1,...,x_n$ are the lookahead variables whose domains are dx1,...,dxn.
2. For each x_j^{dxj}, let
   a. $dxj = \{ y_j \in dxj \mid \exists\, y_1 \in dx1,..., \exists\, y_{j-1} \in dxj\text{-}1, \exists\, y_{j+1} \in dxj\text{+}1,..., \exists\, y_n \in dxn$ such that PR $\models P\theta$ with $\theta = \{x_1/y_1,...,x_n/y_n\}\}$ and $dzj \neq \{\}$.
   b. z_j as
      - a new variable of domain dzj if $dzj = \{e_1,...,e_l\}$! > 1.
      - the constant e if $dzj = \{e\}$.
3. $\theta_{i+1} = \{x_1/z_1, ...,x_n/z_n\}$
4. G_{i+1} is the goal
   - '$\leftarrow (A_1,...,A_{m-1},A_{m+1},...,A_k)\theta_{i+1}$' if at most one z_i is a d-variable.
   - '$\leftarrow (A_1,...,A_{m-1},P,A_{m+1},...,A_k)\theta_{i+1}$' otherwise.

The LAIR can be seen as a general mechanism for enforcing a k-consistency between the k lookahead variables. The LAIR reduces the search space in 'a priori way* and earlier than a forward checking use of the constraint. However, it takes also more computation time to produce this reduction. Note also, that when only one variable occurs in P, the LAIR reduces to the FCIR.

The LAIR is the theoretical foundation for a control mechanism called lookahead declarations. A lookahead declaration for a predicate symbol p of arity n is an expression of the following form.

lookahead $p(a_1,...,a_n)$ where a. is either g either d.

This declaration, which is unique for a particular predicate symbol, specifies that a l i t $p(t_1,...,t_n)$ n the resolvent can be selected only when all its arguments corresponding to a 'g' in the declaration are ground and when it is either ground or lookahead checkable. When it is lookahead checkable, the LAIR must be used to resolve it; otherwise, normal derivation is applied.

The computation rule is even more important for the LAIR than for the FCIR. We first define a efficient computation rule wrt lookahead declarations.

Definition 20: A computation rule is efficient wrt the lookahead declarations, if a l i t $p(x_1,...,x_n)$ n the resolvent submitted to a lookahead declaration is only selected if either it is lookahead checkable or all its arguments are ground.

A efficient computation rule wrt lookahead declarations gives UB few informations about when to select a lookahead constraint. It is clear that selecting it too early can induce some unproductive work (no new informations are inferred) and that a late selection reduces the pruning of the search space. It is not difficult to define efficient computation rules which select only lookahead constraints which are likely to produce new informations. The definition of the LAIR should not be seen as suggesting a particular implementation. Actual implementations should be based, for instance, on generalisation* of AC-3 |II) or AC-4 |12).

Example: Consider a vision problem in a three-faced vertex world. $d = \{>,<,+,-\}$ is the set of possible labels for the vertex. Constraints in the problem are given by the "so-called" fork, L, T and arrow junctions. For instance, the fork junction can be defined as follows.

fork(+,+,+) ←
fork(-,-,-) ←
fork(>,-,>) ←

Also, a lookahead declaration 'lookahead fork(d,d,d)' can be specified. If the resolvent contains a literal 'fork($x1^d$,$x2^d$,$x3^d$)', an application of the LAIR returns the substitution $\{x1^d/z1^{d1},x2^d/z2^{d2},x3^d/z3^{d1}\}$ where d1 = $\{>,+,-\}$ and d2 = $\{+,-\}$.

Finally, the LAIR can be specialised for some constraints (e.g an inequality constraint between two integers). This will achieve the pruning defined by the LAIR in a very efficient way.

4.2. Properties of the LAIR.

Theorem 21: Soundness of the LAIR.

The proof of this theorem is a simple generalisation of the soundness proof of the FCIR.

There is no equivalent for the LAIR to theorem 16. Thus, a proof procedure using the LAIR for lookahead checkable predicates and SLDD-resolution otherwise will not be complete. The reason is that the LAIR is used only to remove inconsistent values and not for making choices. A sufficient condition to ensure completeness is to provide generators of values for the variables occurring in predicates submitted to lookahead declarations. This result was expected as the LAIR can be seen as a general mechanism for enforcing a k-conttistency between the k lookahead variables and it is well-known that enforcing a k-consistency throughout a network of constraints is not generally sufficient for solving arbitrary problems.

5. Partial lookahead in logic programming.

This section provides a theoretical basis for building-in some constraints in such a way that there are neither a specialisation of the FCIR or a specialisation of the LAIR. It is motivated by the existence of some constraints for which, on one hand, forward checking is not appropriate (the reduction of the search space occurs too late in the computation) and lookahead use is too costly (in computation time) while, on the other hand, it is possible to use this constraint for reducing drastically the search space with a small amount of computation. Examples of such constraints

are linear equation! and inequations on natural numbers which can be handled by a reasoning about variation intervals |9). For instance, given

$$R + E + 1 = 10 + T$$
$$R \in \{0,1\}$$
$$E, T \in \{0,2,3,4,5,6,7,8,9\}$$

we can infer $T = 0$ and $E \in \{8,9\}$ since $1 \leq R + E + 1 \leq 11$ and $10 \leq 10 + T \leq 19$ which implies $10 + T \leq 11$ and $R + E + 1 \geq 10$. Such kind of reasoning is very interesting since it reduces drastically the search space while inducing almost no overhead. Since it can happen that not all the inconsistent values are removed, they are a special case of the PLAIR. These constraints have been applied successfully in areas ranging from crypt-arithmetic to integer linear programming. We now define an inference rule we call the partial lookahead inference rule which is of no use per se but provides a theoretical basis for building in some particular class of constraints. The LAIR can be seen as a particular case of It. It consists in replacing the point 3 and S in the LAIR definition by the following two points.

1. For each x_j, let
   a. $dsj \supseteq \{ y_j \in dxj \mid \exists y_1 \in dx1, ..., \exists y_{j-1} \in dxj\text{-}1, \exists y_{j+1} \in dxj+1, ..., \exists y_n \in dxn$ such that $PR \models P\theta$ with $\theta = \{x_1/y_1, ..., x_n/y_n\}\}$ and $dsj \neq \{\}$.
   b. $dsj \subseteq dxj$.
2. G_{i+1} is '$\leftarrow (A_1,...,A_{m-1},P,A_{m+1},...,A_k)\theta_{i+1}$'

The set dzj is not defined in this inference rule and is dependent of each particular constraint. What we have defined Is a theoretical framework for justifying certain kinds of specialisation. The soundness of the PLAIR can easily be proved from the soundness of LAIR.

6. Conclusion.

CTs are a powerful paradigm for solving CSPs. While being the basis for some successful problem-solvers, this paradigm has not been taken into account during the design of programming languages. However, there exists a inherent Interest to build a declarative language based on this paradigm as it increases both flexibility for stating and solving the problem.

This paper has presented a theoretical framework for integrating CTs inside logic programming. Several new inference rules have been defined and their formal properties have been proved. Also, the interest of some classes of computation rules wrt the expressiveness and the efficiency have been stressed in this context.

This makes logic programming not only a good language for slating CSPs but also an efficient tool for solving them as confirmed by our first experiments.

Acknowledgements.

1 gracefully acknowledge many helpful discussions concerning this research with M. Dincbas, H. Gallaire and H. Simonis. In addition, L. Vieille provided a careful criticism on a first version of this paper.

References

I. Alt-kacl, H. and Nasr, R. "LOGIN: A Logic Programming Language with built-in Inheritance". Journal of Logic Programming 5, 3 (October 1986), 185-215.

3. Colmerauer, A., Kanoui, H. and Van Caneghem, M. "Prolog, bases theoriques et developpements actuals.". T.SJ. (Techniques at Sciences /nformatiques). 2, 4 (1083), 271-311.

3. Flkes, R.E. A heuristic program for solving problem* stated a$ non deterministic procedures.. Ph.D. Th., Com put. scl. dept. Carnegie-Mellon Univ. Pittsburgh, PA, 1968.

4. Freuder, E.C. "Synthesislng constraint expressions". Communications of the ACM 21 (November 1978), 058-066.

5. Haralick, R.M. and Elliot, G.L. "Increasing tree search efficiency for constraint satisfaction problems.". Artificial intelligence 14 (1080), 263-313.

6. Irani, K.B. and Shin, D.G. A Many-Sorted Resolution baaed on an Extension of a First-Order language. IJCA1-85, Los Angeles, August, 1085.

7. Jaffar, J. and Lasses, J-L. Constraint Logic Programming. Proceedings of the 14th ACM POPL Symposium, Munich, Wast Germany, January, 1087.

8. Kulpers, B.J. "Qualitative simulation". Artificial InteUgenee 29, 3 (September 1086), 280-338.

9. Lauriere, J-L. "A Language and a Program for Stating . and Solving Combinatorial Problems". Artificial InteUgenee 10, 1 (1078), 20-127.

10. Lloyd, J. W.. Foundations of Logic Programming. Springer-Verlag, Berlin Heidelberg New York Tokyo, 1084.

11. Mackworth, A.K. . "Consistency in network of relations". Artificial Intelligence 8, 1 (1077), 90-116.

12. Mohr, R and Henderson, T.C. "Arc and Path Consistency Revisited". Artificial Intelligence 28 (1086), 225-233.

13. Montanari, U. "Networks of constraints : fundamental properties and applications to picture processing". Information Science 7, 2 (1074), 05132.

14. Stallman, R.M. and Sussman, G.J. "Forward reasoning and depency-directed backtracking In a system for computer-aided circuit analysis". Artificial InteUgenee 0 (1077), 135106.

15. Sussman, G.J. and Steele, G.L. "CONSTRA1NTS-A Language for Expressing Almost-Hierarchical Descriptions". Artificial InteUgenee 14, 1 (1080), 1-30.

16. Van Hentenryck, P. and Dincbas, M. Domains in Logic Programming. Proceedings of the National Conference on Artificial Intelligence (AAA1-86), Philadelphia, USA, August, 1986, pp. 750-765.

17. Van Hentenryck, P. and Dincbas, M. Forward Checking in Logic Programming. Fourth International Conference on Logic Programming, Melbourne, Australia, May, 1087.

18. Walther, C. Unification in many-sorted theories. Proceedings of the 6 European Conference on Artificial Intelligence, Pisa (Italia), September, 1084.

19. Walts, D. Generating semantic descriptions from drawings of scenes with shadows. A1271, MIT, Massachusetts, November, 1072.