Most proposals in the literature augment deductive databases with a procedural semantics to implement the control mechanisms needed to support updates (see [Mon93] as a source of references.) Often the semantics of this procedural component is accommodated in some logic capable to deal with dynamics. This is the case of transaction logic [BK94], dynamic logic [MW88] or various modal logics. For instance, in $\mathcal{LDL}$ transactions are special rules with updates, which are evaluated top-down and their semantics is given by dynamic logic. This combination of declarative and procedural semantics has as a major disadvantage the fact that the architecture of the abstract machine supporting deductive databases is deeply altered. As a consequence, the available optimization techniques are no longer directly applicable.

We propose, in a more conservative way, to leave the simple declarative framework unaltered. This is achieved by means of a transformation of updates and transactions into sets of clauses which

- reflect the intuitive meaning of state changes in a declarative way, and
- can be efficiently executed using the ordinary bottom-up, fixpoint-based evaluation of deductive databases.

We apply this transformation technique to a generalization of $\mathcal{LDL}$ transactions, i.e., clauses containing *update predicates* in their body. The subgoal preceding the updates is the precondition, and the subgoal following the updates is the postcondition of the transaction. Pre/postconditions are used to preserve database integrity: in particular, the updates are actually performed only if the postcondition is fulfilled, otherwise the transaction aborts.

The notion of $\mathcal{LDL}$ transaction provides a form of integration of query and updates, in the sense that there is a uniform notation. However, different evaluation methods are used for query and transaction clauses. The proposed compilation allows to execute transaction clauses in a bottom-up way, similarly to queries. The transformation is based on the notion of XY-stratification [AOZ93], i.e., a syntactic property of programs which properly extends ordinary stratification. A remarkable characteristic of XY-stratified programs is that they can be executed by an iterated fixpoint procedure, even if they are recursive, non monotonic programs. XY-stratification is defined in terms of *stage arguments*, i.e., predicate arguments which record the stage of the computation, and allow to control state changes. The language *Statelog*$^{+-}$, proposed in [LL93] is directly related to our work: here, a different formalization of state change leads to similar results, e.g. the capability to compute the perfect model of programs with updates. The main differences with our approach are that we do not admit states as first class values in the language, and that our focus is on compilation of updates aimed at providing efficient executions.

Section 2 gives a formal account of XY-stratification. Then sections 3 to 5 gradually introduce compilation of updates into XY-stratified programs: simple updates, composition of updates, simple and nested transactions. In all cases two different semantics are explored: parallel semantics and sequential semantics. Section 5 also presents as an example the compilation of a simple transaction. Section 6 contains a few final remarks.

2 XY-stratified programs

The basic concept of our approach is the notion of XY-stratification, i.e. a syntactic property of non-monotonic recursive $Datalog_{1s}$ programs [1]. The class of programs identified by such property, named XY-programs, captures the expressive power of the inflationary fixpoint semantics.

The basic idea is that recursive predicates have a special argument named *stage* which is an integer. There are two different ways to use the stage: in rules which do not increment the stage (X-rules), and rules that increment the stage by one (Y-rules). An XY-stratified program allows recursion only when there is an increment of the stage. If there exists a reordering of the rules of the predicates which induces a XY-stratification, then it is possible to apply an iterated fixpoint procedure which distinguishes the application of X-rules by the application of Y-rules. Such procedure computes the perfect model associated to the set of recursive predicates.

The following subsections introduce syntax and semantics of XY-stratified programs following the presentation of [AOZ93].

Syntax

Definition 2.1 Given a program P, a set of rules of P defining a maximal set of mutually recursive predicates will be called a recursive clique of P.

Definition 2.2 Given a recursive clique Q, the first arguments of the recursive predicates of a rule r in Q will be called the stage argument of r.

The usage of stage arguments is for counting as in the recursive definition of integers: *nil* stands for zero and *s(I)* stands for *I+1*.

Definition 2.3 Let Q be a recursive clique and r be a recursive rule of Q. Then r is called an

- X-rule if all stage arguments of r are equal to a simple variable, say J, which does not appear anywhere else in r;
- Y-rule if (i) some positive goal of r has as stage argument a simple variable J, (ii) the head of r has stage argument s(J), (iii) all the remaining stage arguments are either J or s(J) and (iv) J does not appear anywhere else in r.

Definition 2.4 A recursive clique Q will be said to be an XY-clique when all its recursive rules are either X-rules or Y-rules.

Priming: an atom $p'(t)$ is called the *primed version* of an atom $p(t)$.
Given an XY-clique Q, its version primed is constructed by priming certain occurrences of recursive predicates in recursive rules as follow:

- X-rules: all occurrences of recursive predicates are primed;

[1] $Datalog_{1s}$ is a simple extension of *Datalog* which admits a single unary function symbol $s(.)$. This language has been used for temporal reasoning in [CHO93].

- Y-rules: the head predicate is primed, and so is every goal with stage argument equal to that of the head.

Definition 2.5 An XY-clique Q will be said to be XY-stratified when

- the primed version of Q is non recursive
- all exit rules have as stage argument the same constant.

where a rule is an *exit*-rule if all predicates in its body are not defined in the clique.

Definition 2.6 A program is XY-stratified if every recursive rule that contains a negated recursive goal in its body belongs to an XY-stratified clique.

The dependence graph for a primed clique provides a very simple syntactic test to check whether a program is XY-stratified: it contains no cycles, thus there exists a topological sorting of the nodes of Q' which obeys stratification, and such that the unprimed predicate names precede the primed ones.

Semantics

We can partition the atoms in the Herbrand Base B_Q of the original program Q into classes according to their predicate name and their stage arguments as follow:

- there is a distinct class, say σ_0, containing all instances of non recursive predicates in Q, without a stage argument;
- all atoms with the same recursive predicate name and the same number of function symbols s in the stage argument belong to the same equivalence class $\sigma_{n,p}$, with n denoting the number of s function symbols in the stage argument of p.

The partition Σ of B_Q constructed in this way can be totally ordered, by letting σ_0 be the bottom stratum in Σ , and then letting $\sigma_{n,p} \prec \sigma_{m,q}$ if

- $n < m$, or
- if $n = m$ but p' precedes q' in the primed sorting of the clique.

The totally ordered Σ so constructed will be called *stage layering* of B_Q.

Theorem 2.1 *Each XY-stratified clique Q can be locally stratified according to a stage layering of B_Q. Then for every instance r of each rule in Q, the head of r belongs to a layer strictly higher than the layers for the goals in r (strict stratification).*

Since the stratification is strict, in computing the iterated fixpoint, the saturation for each stratum is reached in one step. Therefore, the compiler can reorder the rules according to the primed sorting of their head names; then having derived all atoms with stage value J a single pass through the rules of Q ordered according to the primed sorting computes all the atoms with stage value $s(J)$. To formalize the iterated fixpoint procedure for XY-stratified programs, we introduce the following notions.

- Let p' be the k-th predicate name in the primed sorting.

- Let T_k be the immediate consequence operator for the recursive rules in Q defining p.
- The composite consequence operator Γ_Q will be defined as follows:

$$\Gamma_Q(I) = T_n(T_{n-1} \ldots (T_1(I)) \ldots)$$

where I is an interpretation over Q's Herbrand Base B_Q, and $n \geq 1$.

- Let T_0 be the immediate consequence operator for the *exit*-rules. By the second condition of XY-stratification, all atoms in $T_0(\emptyset)$ share the same stage argument. However, additional atoms with the same stage value might be obtained by firing the X-rules. Therefore, if p_k is the k-th predicate name in the primed sorting, we define T_k^{X} the immediate consequence operator for the X-rules with head name p_k, if any such rules exists, and the identity transformation otherwise. We can define the composite consequence operator for the X-rules, Γ_Q^{X} as follow: $\Gamma_Q^{\mathrm{X}} = T_n^{\mathrm{X}}(T_{n-1}^{\mathrm{X}} \ldots (T_1^{\mathrm{X}}(I)) \ldots)$. Thus, the ground atoms with the same stage argument as the *exit*-rules are $\Gamma_Q^{\mathrm{X}}(T_0(EDB))$.

Theorem 2.2 *Let Q be a XY-stratified clique, with composite consequence operator Γ_Q and composite consequence operator for the* X*-rules Γ_Q^{X},then*

- *Q is locally stratified,*
- *the perfect model of Q is $M_Q = \Gamma_Q^{\omega}(M_{nil})$, where $M_{nil} = \Gamma_Q^{\mathrm{X}}(T_0(\emptyset))$ e T_0 is the immediate consequence operator for the exit-rules of Q.*

Thus the perfect model of an XY-stratified clique can be constructed as in the case of positive programs. Computation of XY-stratified programs proceeds similarly to that of stratified programs: all the non recursive predicates in the recursive XY-clique must be saturated before the recursive rules in the clique are computed.

3 Simple updates

In this paper we consider only updates in the body of the rules, as we are interested in extending $\mathcal{LDL}$ transactions.

Updates are often classified according to two different semantics: *weak* updates and *strong* updates. Strong updates are those which allow to delete atoms only if they are in the current database state and allow to insert atoms only if they are not. In the case of weak updates no precondition is checked. Consider, for example, the database instance { $p(a)$, $p(b)$, $q(a)$ }. Under the weak semantics the insertion of $p(b)$ or the deletion of $q(b)$ does not change the database, although they are allowed. Under the strong semantics a failure is reached.

We consider here simple updates of the form $+p(a)$, $-p(a)$, corresponding respectively to insertion and deletion of an EDB predicate. As a consequence, view updates are not considered in this paper. Also, we refer here to the weak semantics, although we briefly sketch later how strong semantics might be dealt with.

The idea is to associate with every n-ary EDB predicate p two new $(n+1)$-ary predicate symbols: p_{stage} and p_{del} where the extra argument is the *stage argument* in the first position. The stage argument in p_{stage} will model the various state transitions of the EDB predicate p performed by the updates. p_{del} will play the role of a delete list, keeping tracks of the tuples to be removed from p.

An update predicate $\pm p(a)$ is then compiled into an XY-stratified program which defines the predicates p_{stage} and p_{del}. The following definition shows the code which updates are compiled to. For generality of exposition, we deal here with updates $\pm p(a)$ where the tuple a may contain variables. The compilation is therefore parametric with respect to a query q such that $vars(a) \subseteq vars(q)$, which provides the actual tuples to be inserted in or removed from p.

Definition 3.1 Let p be an EDB predicate, a a tuple and q a query such that $vars(a) \subseteq vars(q)$. The code realizing the deletion $-p(a)$ with respect to the query q, denoted $\mathcal{T}[q](-p(a))$, is the following:

$r_1: \; p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-}rule\}$
$r_2: \; p_{del}(s(nil), a) \leftarrow q, p_{stage}(nil, \_).$ $\{deletion\text{-}rule\}$
$r_3: \; p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-}rule\}$

Definition 3.2 Let p be an EDB predicate, a a tuple and q a query such that $vars(a) \subseteq vars(q)$. The code realizing the insertion $+p(a)$ with respect to the query q, denoted $\mathcal{T}[q](+p(a))$, is the following:

$r_1: \; p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-}rule\}$
$r_2: \; p_{stage}(s(nil), a) \leftarrow q, p_{stage}(nil, \_).$ $\{insertion\text{-}rule\}$
$r_3: \; p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-}rule\}$

In both definitions, r_1 is an exit rule which initializes p_{stage}. r_2 is a Y-rule. In case of deletion r_2 records in p_{del}latex that tuple a has to be deleted; in case of insertion r_2 adds to p_{stage} the new tuple a in the next stage. Finally, in both definitions, the Y-rule r_3 is the *copy*-rule, which allows to copy to the next stage of predicate p_{stage} all tuples which have not been canceled. Notice that, r_3 acts as the *frame axiom* which states: whenever something is true in some stage and it is not explicitly deleted, then it will also be true in the next stage. In the insert and delete rules, the query q plays the role of providing the actual tuples to be inserted/removed. If a is a tuple of constants, then q is not needed.

It is simple to show that programs $\mathcal{T}[q](\pm p(a))$ are XY-stratified. It is therefore meaningful to consider their perfect model computed with the iterated fixpoint procedure.

Notice that the new extension of p after the update, denoted by p', is given by the set of tuples with the maximum stage computed by the the corresponding fragment (insert or delete) of p_{stage}:

$$r_0: \; p'(x) \leftarrow p_{stage}(s(nil), x).$$

In fact, in the case of single updates, the maximum stage is simply $s(nil)$, as the fixpoint is reached after two iterations. A more complex situation will arise when considering composition of updates.

Consider now the perfect model M of the program formed by rule r_0 and $\mathcal{T}[q](\pm p(a))$. By our construction the extension of p' in M represents the effect of the update on the

extension of p. In this sense, the above simple translation of updates into rules is a declarative reconstruction of an operational semantics based on state transitions.

It is worth noting that the fragments in definitions 3.2 and 3.1 realize weak updates. The code for strong updates differs only for the insert and delete rules, which have to check for absence (resp., presence) of the tuple to be inserted (resp., removed).

$p_{del}(s(nil), a) \leftarrow q, p_{stage}(nil, a).$ $\{delete\text{-rule}\}$
$p_{stage}(s(nil), a) \leftarrow q, \neg p_{stage}(nil, a).$ $\{insert\text{-rule}\}$

4 Composition of updates

In this section, we consider compositions of updates denoted by $u_1, \ldots, u_n$, $(n > 1)$ with reference to two different semantics: parallel and sequential evaluation of updates. According to the parallel semantics, also referred to as *non-immediate semantics*, updates are computed in two phases. During the first phase updates are collected and, in the second, they are executed all together without affecting each other.

According to the sequential semantics, also referred to as *immediate semantics*, updates are executed as soon as they are encountered. The presence of updates in a rule with immediate semantics leads to evaluate a query in a sequence of database states. Insertions and removals are immediately triggered when a body rule is satisfied, thus a single query can be evaluated on different states.

Parallel Semantics

According to this semantics, the updates $u_1, \ldots, u_n$ are evaluated concurrently without affecting each other. Therefore the code realizing parallel composition is obtained by the simple union of the programs of the single updates.

Definition 4.1 Let $u_1, \ldots, u_n$ be a composition of updates, and q a query such that $vars(u_1, \ldots, u_n) \subseteq vars(q)$. Then the code realizing the parallel semantics of the composition is the following:

$$\mathcal{T}_{par}[q](u_1, \ldots, u_n) = \mathcal{T}[q](u_1) \cup \ldots \cup \mathcal{T}[q](u_n).$$

Observe that as a consequence of the union operator, the exit rules and the copy rules in the programs of the single updates occur only once in the final program.

As an example, let us consider the update of an attribute of a tuple; it can be modelled with the parallel composition of the deletion of the old tuple and the insertion of the modified tuple. Let p be an EDB predicate and a and b tuples. The code realizing the update of tuple a into tuple b is the following:

$r_1: \; p_{stage}(nil, x) \leftarrow p(x).$ $\{exit\text{-rule}\}$
$r_2: \; p_{stage}(s(nil), b) \leftarrow q, p_{stage}(nil, \_).$ $\{insertion\text{-rule}\}$
$r_3: \; p_{del}(s(nil), a) \leftarrow q, p_{stage}(nil, \_).$ $\{deletion\text{-rule}\}$
$r_4: \; p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x).$ $\{copy\text{-rule}\}$

Observe that the parallel composition of complementary updates $+p(a), -p(a)$, according the specified semantics, results in performing the insertion $+p(a)$.

Sequential Semantics

According to this semantics, the updates $u_1, \dots, u_n$ are evaluated sequentially so that the updates on the same predicate affect each other. Therefore, the code realizing the single update in the sequential composition is now dependent on the set of updates on the same predicate which have already been performed.

Observe that the sequential composition differs from the parallel one only for updates of the same predicate. Given a composition of updates $u_1, \dots, u_n$, we can rearrange it as a parallel composition of sequential composition of updates on the same predicate, without affecting the effect of the overall composition. As a consequence, it suffices to restrict ourselves to consider only sequences of updates on the same predicate.

Definition 4.2 Let p be an EDB predicate, a a tuple and q a query such that $vars(a) \subseteq vars(q)$. Let $u_0, \dots, u_n$ be a composition of updates over the same EDB predicate p.

- The code realizing the insertion $u_i = +p(a)$ ($i \in [0, n]$) with respect to the query q, denoted $\mathcal{T}_i[q](+p(a))$ is the following:

$$\begin{array}{lll} r_1: & p_{stage}(s^{2i}(nil), x) \leftarrow p_{stage}(s^{2i-1}(nil), x). & \{\textit{exit}\text{-rule}\} \\ r_2: & p_{stage}(s^{2i+1}(nil), a) \leftarrow q, p_{stage}(s^{2i}(nil), \_). & \{\textit{insert}\text{-rule}\} \\ r_3: & p_{stage}(s^{2i+1}(nil), x) \leftarrow p_{stage}(s^{2i}(nil), x), & \\ & \qquad \neg p_{del}(s^{2i+1}(nil), x). & \{\textit{copy}\text{-rule}\} \end{array}$$

- The code realizing the deletion $u_i = -p(a)$ ($i \in [0, n]$) with respect to the query q, denoted $\mathcal{T}_i[q](-p(a))$, differs only for rule r_2:

$$\begin{array}{lll} r_2: & p_{del}(s^{2i+1}(nil), a) \leftarrow q, p_{stage}(s^{2i}(nil), \_). & \{\textit{delete}\text{-rule}\} \end{array}$$

Notice that, for $i = 0$, the clause r_1 is

$$p_{stage}(s^0(nil), x) \leftarrow p_{stage}(s^{-1}(nil), x).$$

We stipulate that $p_{stage}(s^{-1}(nil), x)$ stands for $p(x)$. Since $s^0(nil) = nil$, the clause r_1 becomes the ordinary exit-rule of definition 3.2 and 3.1:

$$p_{stage}(nil, x) \leftarrow p(x).$$

Notice that there are two differences with the simple updates of definitions 3.1 and 3.2: a different exit rule r_1 has been added; and the code is parametric with respect to the number of occurrences of updates on the same predicate. Rule r_3 records the result of the last update on the same predicate. After i updates on predicate p, $s^{2i-1}(nil)$ is the maximum stage argument of p_{stage}.

The composition of updates according to the sequential semantics is given by the following definition.

Definition 4.3 Let $u_1, \dots, u_n$ be a composition of updates over the same EDB predicate p, and q a query such that $vars(u_1, \dots, u_n) \subseteq vars(q)$. Then the code realizing the parallel semantics of the composition is the following:

$$\mathcal{T}_{seq}[q](u_1, \dots, u_n) = \mathcal{T}_1[q](u_1) \cup \dots \cup \mathcal{T}_n[q](u_n).$$

As an example, let us consider again the update of an attribute of a tuple; it can be modelled with the deletion of the old tuple and the insertion of the new tuple.

Let p be an EDB predicate and a and b tuples. The code realizing the update of tuple a into tuple b according to sequential semantics is the following:

$$\begin{array}{lll} r_1: & p_{stage}(nil, x) \leftarrow p(x).. & \{\textit{exit}\text{-rule}\} \\ r_2: & p_{del}(s(nil), a) \leftarrow q, p_{stage}(nil, \_). & \{\textit{deletion}\text{-rule}\} \\ r_3: & p_{stage}(s(nil), x) \leftarrow p_{stage}(nil, x), \neg p_{del}(s(nil), x). & \{\textit{copy}\text{-rule}\} \\ r_4: & p_{stage}(s(s(nil)), x) \leftarrow p_{stage}(s(nil), x). & \{\textit{exit}\text{-rule}\} \\ r_5: & p_{stage}(s(s(s(nil))), b) \leftarrow q, p_{stage}(s(s(nil)), \_). & \{\textit{insertion}\text{-rule}\} \\ r_6: & p_{stage}(s(s(s(nil))), x) \leftarrow p_{stage}(s(s(nil)), x), & \\ & \qquad \neg p_{del}(s(s(s(nil))), x). & \{\textit{copy}\text{-rule}\} \end{array}$$

5 Transactions

In this section we tackle the problem of integrating updates and queries. We propose how to integrate the two modalities of interacting with the deductive database in a unique framework which can be executed by a fixpoint evaluation. Such framework defines the concept of a transaction, which will be introduced gradually: simple transactions and nested transactions.

5.1 Simple Transactions

A simple transaction is a single rule of the form:

$$h \leftarrow pre, u_1, \dots, u_m, post.$$

where $u_1, \dots, u_m$ is a composition of updates, *pre* and *post* are queries, and the predicate symbol in the head h, called a *transaction* predicate, is a fresh predicate symbol, which does not occur anywhere else in the program. *pre* is called the precondition of the transaction, and *post* the postcondition. It is worth noting that preconditions and postconditions play the role of integrity constraints, and the interesting case is when the same predicate is involved both in pre/postconditions and in updates.

As in the case of multiple updates, the parallel and sequential semantics of transactions behave differently, and therefore they are considered separately.

Parallel Semantics

In this case, each single update is constrained by the success of the precondition, so that it has to be evaluated before the execution of every update. Moreover, *pre* is needed to provide the actual tuples to be inserted/removed. Therefore, we use the code $\mathcal{T}_{par}[pre](u)$ of definition 4.1 instantiated on the precondition *pre* of the transaction.

Next, we have to take into consideration the postcondition. In fact, the success of the transaction, as well as the possibility of inferring facts of the transaction predicate h, is subject to the satisfaction of the query *post*. However, the evaluation of *post* must take into account the effect of the updates on the extensional predicates. To this purpose, we use the following *derivation-rule* r_d for h:

$r_d: \quad h \leftarrow pre, post'.$ $\qquad\qquad\qquad\qquad$ $\{derivation\text{-rule}\}$

where $post'$ denotes the query $post$ evaluated with respect to the program modified by replacing every occurrence of an extensional predicate p in a rule with the predicate p', denoting the final extension of p after the updates. In the case of parallel semantics, p' can be simply defined as follows:

$$p'(x) \leftarrow p_{stage}(s(nil), x).$$

Finally, the code for a transaction $h \leftarrow pre, u_1, \ldots, u_m, post$ under a parallel semantics is obtained as follows, by cumulating the compilation of the parallel composition of updates with the derivation-rule r_d:

$$\mathcal{T}_{par}(h \leftarrow pre, u_1, \ldots, u_m, post.) = \mathcal{T}_{par}[pre](u_1, \ldots, u_m) \cup \{r_d\}.$$

As a simple example, consider the following $\mathcal{LDL}$ transaction on an EDB relation

$$emp(name, dept)$$

which transfers all employees of the toy department to the shoe department:

$$tr: \quad transf(x) \leftarrow emp(x, toy), -emp(x, toy), +emp(x, shoe).$$

According to the proposed compilation scheme, we obtain the following code for $\mathcal{T}_{par}(tr)$:

$r_1: \quad emp_{stage}(nil, x, d) \leftarrow emp(x, d).$
$r_2: \quad emp_{del}(s(nil), x, toy) \leftarrow emp(x, toy), emp_{stage}(nil, \_, \_).$
$r_3: \quad emp_{stage}(s(nil), x, d) \leftarrow emp_{stage}(nil, x, d), \neg emp_{del}(s(nil), x, d).$
$r_4: \quad emp_{stage}(s(nil), x, shoe) \leftarrow emp(x, toy), emp_{stage}(nil, \_, \_).$
$r_5: \quad transf(x) \leftarrow emp(x, toy).$

Observe that, in absence of postconditions (which is precisely the case in $\mathcal{LDL}$), there is no need to exploit the updated EBD predicates to compute the derivation rule r_5. We next add a postcondition to the transaction, by requiring that no more than 20 employees can be associated with the shoe department: $tr': \quad transf(x) \leftarrow emp(x, toy), -emp(x, toy),$ $+emp(x, shoe), count(emp(\_, shoe)) \leq 20$. According to the proposed compilation scheme, we obtain for $\mathcal{T}_{par}(tr')$ the same code as above, except from the derivation rule r_5, which now becomes:

$$r_5: \quad transf(x) \leftarrow emp(x, toy), count(emp'(\_, shoe)) \leq 20.$$

where emp' is the updated version of emp, namely:

$$emp'(x, d) \leftarrow emp_{stage}(s(nil), x, d).$$

Sequential Semantics

In this case, the precondition must be re-evaluated before each update in the transactions, in order to take into consideration the effect of the preceding updates. To this end, we adapt the compilation of transactions with parallel updates by simply modifying how

preconditions are compiled, in a way similar to postconditions in the parallel semantics. Given a precondition pre, we denote by pre'_k the query pre incrementally modified after k updates as follows. Every occurrence in pre of an extensional predicate p is replaced with the predicate p'_i, denoting the current extension of p after i updates on p itself. In the case of sequential semantics, the current stage of a relation after i updates can be retrieved using the formula $s^{2i-1}(nil)$ as stage argument. So after i updates on p, p' is defined as follows:

$$p'(x) \leftarrow p_{stage}(s^{2i-1}(nil), x).$$

Notice that, in pre'_k only those extensional predicates of pre for which one or more updates have been performed by the sequence $u_1, \ldots, u_k$ have been replaced. So the above definition of p' simply accumulates the updates on the same relation, while pre'_k accumulates the updates on all extensional predicates in pre executed by $u_1, \ldots, u_k$.

We can now redefine the compilation of definition 4.3 as follows:

$$\mathcal{T}_{seq}[pre](u_1, \ldots, u_n) = \mathcal{T}_1[pre](u_1) \cup \mathcal{T}_2[pre'_1](u_2) \cup \ldots \cup \mathcal{T}_n[pre'_{n-1}](u_n).$$

Under the above definition of the updated predicates p', the same *derivation-rule* r_d for h adopted in the case of parallel semantics can be used:

$r_d: \quad h \leftarrow pre, post'.$ $\qquad\qquad\qquad\qquad$ $\{derivation\text{-rule}\}$

Finally, the code for a transaction $h \leftarrow pre, u_1, \ldots, u_m, post$ under a sequential semantics is obtained as follows, by cumulating the translation of the sequential composition of updates with the derivation-rule r_d:

$$\mathcal{T}_{seq}(h \leftarrow pre, u_1, \ldots, u_m, post.) = \mathcal{T}_{seq}[pre](u_1, \ldots, u_m) \cup \{r_d\}.$$

5.2 Nested transactions

In general, transactions are nested in the sense that transaction predicates may occur in the pre- or postconditions, although recursive calls to transaction predicates are not allowed. This is the case in $\mathcal{LDL}$, where moreover postconditions are not allowed. We do not explain here in detail how nested transaction are compiled for limitation of space. However, the idea of the compilation is the following. A set of nested, non recursive transaction predicates can be repeatedly unfolded, until a single rule is obtained. At this stage, a transformation scheme which closely follows that for sequential composition can be directly applied.

6 Final Remarks

We proposed in this paper a compilation of updates and transactions based on their declarative reconstruction in terms of XY-stratified programs. Despite its simplicity, the proposed compilation produces code that can be efficiently executed by a machine supporting bottom-up execution of XY-stratified programs, such as that of $\mathcal{LDL}$. In particular:

- the stage arguments can be actually implemented as a single counting variable, global to the database, thus avoiding the overhead of the copy rules;

- the compilation technique directly support virtual updates, which can be actually executed after the transaction commits.

Various more general forms of transactions can be supported on the basis of the proposed technique, and are currently under investigation. These include recursive transactions (and active rules), and updates and transactions in object-oriented deductive databases. In this latter case, we refer to the object-oriented extension of deductive databases of [Zan89], which is based on a logical definition of object-identity by means of the non deterministic *choice* construct [NT88, GPSZ91, CF94]. Preliminary investigations show that the proposed compilation smoothly scales to the extended framework.

References

[AOZ93] N. Arni, K. Ong, C. Zaniolo. *Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$ ++ Approach*, In Proceeding of Deductive and Object-Oriented Databases – Third International Conference, Springer-Verlag, (Ed. S. Ceri, K. Tanaka, Shalom Tsur), LNCS, pp. 204-221 (1993)

[BK94] A.J. Bonner, M. Kifer, *An overview of Transaction Logic*, Theoretical Computer Science, Vol. 133, pp. 205-265 (1994)

[CF94] M. Carboni, V. Foddai *Aspetti dinamici delle basi di dati deduttive*, Laurea Thesis. Dipartimento di Informatica, Università di Pisa. (1994) (in Italian)

[CHO93] M. Baudinet, J.Chomicki, P. Wolper, *Temporal Deductive Databases*, in *Temporal Databases*, eds. Tansel, Clifford, Gadia, Jajodia, Sagev, Snodgrass, Benjamin and Cummings (1993)

[GPSZ91] F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo. *Non-Determinism in Deductive Databases*, In Proceeding of Deductive and Object-Oriented Databases Second International Conference, Springer-Verlag, (Ed. C. Delobel, M. Kifer, Y. Masunga), LNCS, pagg. 129-146 (1991)

[LL93] G. Lausen, B. Ludäscher. *Updates by Reasoning about States*, Second Compunet Workshop on Deductive Databases, Athens (1993)

[Mon93] D. Montesi, *A Model for Updates and Transactions in Deductive Databases*. PhD. thesis, Dipartimento di Informatica, Università di Pisa (1993)

[MW88] S. Manchanda, D.S. Warren. *A Logic Based Language for Database Updates.* In Deductive Databases and Logic Programming, Eds. J. Minker, pp. 363-394. Morgan-Kaufman (1988).

[NT88] S. Naqvi, S. Tsur. *A Logic Language for Data and Knowledge Bases*, Computer Science Press, NewYork (1988)

[Zan89] C. Zaniolo. *Object-Identity and Inheritance in Deductive Databases- an evolutionary approach*, In Proceeding of Deductive and Object-Oriented Databases Conference, Kyoto (1989)

NEGATION

An Introduction to Regular Search Spaces

Alberto Momigliano
Department of Philosophy
Carnegie Mellon University
15213 Pittsburgh PA, USA,
mobile@lcl.cmu.edu

Mario Ornaghi
Dipartimento di Scienze dell'Informazione
Universita' di Milano
Via Comelico 39/41, Milano, Italy,
ornaghi@unimi.it

Abstract

The aim of this paper is to present the proof-theoretic analysis of logic programming developed in [10, 11], and to show an application to negation-as-failure (*NF*). We define *AAR-systems*, i.e. inference systems based on Axiom Application Rules of a very general kind and we introduce the concept of *regular search space*. We show that regular *AAR*-systems enjoy the analogous of the very features that make Prolog a feasible and successful implementation of logic. Finally, we discuss our application to negation-as-failure. We contend that the notion of regularity provides a better understanding of the traditional theory of *NF*. However, *NF* is not our main concern; our aim is to show through that the adaptability and versatility of our approach.

Keywords. Foundations of logic programming, proof-theory, negation-as-failure, search spaces.

1 Introduction

In this paper we present the main features of the proof-theoretic analysis of logic programming developed in [10, 11][1]. Firstly we explain the concept of *axiom application rule* (*AAR*), which can be seen as an abstraction of an inference step of a logic programming interpreter and of *most general proof tree* (*mgpt*), which is the analog of a *SLD*-derivation. Mgpt's are in fact based on the notion of *AAR*, which easily generalizes to various definitions of clauses and goals. Finally, in the background, all is connected by the notion of *regular search space*, which plays the role of a Prolog-like search space.

Secondly we give a treatment in our proof-theoretic terms of the issue of negation-as-failure (*NF*) [4] and we analyze the incapability *NF* to providing logically justified answers to open queries.

We will contend that the notion of regularity provide us with a better understanding of the traditional theory of *NF*. In [11] it is also shown that our approach provides a firm and natural basis for a form of *constructive negation*, in the sense of [3, 13, 2]

We want to stress at this point that our interest lies here mainly in showing the versatility and the adaptability of our approach – once digested a few but simple initial definitions – rather than keeping up with the front-line of research on negation in logic programming: in particular we shall deal only marginally with new developments in the field (see for example the recent survey by Apt & Bol [1]).

The paper is organized as follows: in Section 2 and 3 we review the theory of Regular *AAR*-Systems formulated in [10, 11]. Section 4 deals with the proof-theory of *NF* and, finally,

[1]This paper is, to a great extent, a subset of those.

Section 4.3 studies the problem of evaluating open negative queries. Proofs of results stated here can be found in [10, 11].

2 Systems Based on Axiom-Application Rules

Our view of an abstract logic programming system is that of an idealized interpreter endowed with *rules* - the inference mechanism - that apply *axioms* - the program - starting from a goal and searching for a proof. We formulate this approach in all its generality and we exemplify it with a system that is related to SLD-resolution.

2.1 AAR-Systems

An AAR-system is a triple $\langle \mathcal{E}, \mathcal{A}, \mathcal{R} \rangle$ where:

1. $\mathcal{E}$ is a set of admissible goal-expressions. In this paper goal-expressions will be atoms or literals, but more general forms could be used (see [10]). Goal-expressions *should not be confused* with goals of the form $\leftarrow L_1, \ldots, L_n$ as intended in logic programming.
2. $\mathcal{A}$ is a set of admissible axioms. For example, an admissible axioms could be (the universal closure of) definite or normal clauses, the completed definitions of the predicates of a program, or more general kinds of axioms (see [10]).
3. $\mathcal{R}$ is a set of axiom application rules. A rule $R \in \mathcal{R}$ is any relation from goal-expressions and axioms to sequences of goal-expressions, including the empty sequence Λ, i.e. $R \subseteq (\mathcal{E} \times \mathcal{A}) \times \mathcal{E}^*$.

Goal-expressions will be indicated by $E, E_1, \ldots$. We shall say that $E_1; \ldots; E_n \in R(E, Ax)$ for those sequences of goal-expressions such that $\langle\langle E, Ax\rangle, E_1; \ldots; E_n\rangle \in R$ (namely $R(E, Ax)$ is a set of sequences of goals). We will draw it as

$$\frac{E_1; \cdots; E_n; \quad Ax}{E}(R)$$

When $\Lambda \in R(E, Ax)$, we write

$$\frac{Ax}{E}(R)$$

In the framework of an AAR-system, one may have *programs*, where a program $\mathcal{P}$ is a set of axioms from $\mathcal{A}$ and its computations search for proofs with rules from $\mathcal{R}$ and axioms from $\mathcal{P}$. Before considering proof-search, we complete our basic definitions and give an example of an AAR-system.

The set of proof trees $\mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$ of an AAR-system $\langle \mathcal{E}, \mathcal{A}, \mathcal{R} \rangle$ is inductively defined below, where $\Pi :: E$ is our linear notation for a pt Π with root E:

Definition 2.1 Every $E \in \mathcal{E}$ is a pt. If $\Pi_1 :: E_1, \ldots, \Pi_n :: E_n$ are pt's and $E_1; \ldots; E_n \in R(E, A)$, then the following is also a proof tree:

$$\frac{\begin{matrix}\Pi_1 & & \Pi_n \\ E_1; & \ldots & E_n;\end{matrix} \quad Ax}{E}(R)$$

We say that a goal-expression is an *assumption* of a proof tree if it is a minor premise in some leaf. The *axioms* of a proof tree are the ones appearing as major premises. The root of a proof tree is called its *consequence*.

If the axioms of a pt belong to a program $\mathcal{P} \subseteq \mathcal{A}$, we say that it is a proof tree with axioms from $\mathcal{P}$.

Definition 2.2 A proof tree is a *proof* of E iff E is its consequence and it has no assumption. If it has at least one assumption, we say that it is a *partial* proof.

Substitutions, or more properly instantiations, will turn out to be central in our treatment. Hence we restrict to goals and pt's for which a notion of substitution is sensible. Applications of a substitution θ to goal-expressions and proof-trees will be indicated by $\theta E, \theta\Pi, \ldots$. We assume, as well, that in $\theta\Pi$ axioms and rules are not affected by θ.

2.2 P-System and SLD-Resolution

In the P-system goal-expressions are literals (indicated by $L, L_1, \ldots$) and admissible axioms are of the form $\forall(\forall y(L_1 \wedge \cdots \wedge L_n) \rightarrow L)$, where the y[2] may appear in the L_i but not in L. If $n = 0$, then the axiom is $\forall(L)$.

There is a single rule $\mathtt{P}$, defined as follows. For every substitution θ renaming y with *eigenvariables*:

$$\theta L_1; \ldots; \theta L_n \in \mathtt{P}(\theta L, \forall(\forall y(L_1 \wedge \cdots \wedge L_n) \rightarrow L))$$

Recall that eigenvariables (often called *parameters*) are variables whose only possible substitutions are capture-freeing renaming.

An application of $\mathtt{P}$ with positive conclusion is for example:

$$\frac{\neg sum(v, v, X); \quad \forall x(\forall z \neg sum(z, z, x) \rightarrow odd(x))}{odd(X)}(\mathtt{P})$$

where an eigenvariable v has been introduced for the z's.

The $\mathtt{P}$ rule is admissible in minimal logic: its instances are derivable in natural deduction, as follows, where the vertical dots allude to a closing branch for the assumption $\neg sum(v, v, X)$, in a way similar to Miller's [9], the eigenvariable v has uniformly substituted z and X is a logical variable:

$$\cfrac{\cfrac{\begin{matrix}\vdots \\ \neg sum(v, v, X)\end{matrix}}{\forall z. \neg sum(z, z, X)}\forall\text{-}I \quad \cfrac{\forall x(\forall z \neg sum(z, z, x) \rightarrow odd(x))}{\forall z \neg sum(z, z, X) \rightarrow odd(X)}\forall\text{-}E}{odd(X)}\rightarrow\text{-}E$$

Now we link the admissible axioms of the P-system to logic programs. Atoms will be denoted by $A, B, A_1, \ldots$. To a clause c of the form $B :- A_1, \ldots, A_n$ we associate $Ax(c)$ of the form $\forall(A_1 \wedge \ldots \wedge A_n \rightarrow B)$, and to a program P the set $Ax(P)$ of the axioms which correspond to its clauses.

Thus SLD-resolution corresponds to the P-system, with the following restrictions: no universal quantifier occurs in the body of a clause and no negative literal is involved in the axioms.

[2] $\forall(\ldots)$ indicates universal closure and $\forall y$ a (possibly empty) list of quantified variables.

An SLD-derivation for a goal $\leftarrow A$, can be seen as a stepwise construction of a pt $\Pi :: \theta A$, as shown below for the derivation $\leftarrow sum(ss0, s0, X_0), \leftarrow sum(ss0, 0, X_1), \Box$. Notice that the formulae in the current goal correspond to the assumptions of the relative pt. We denote with $\preceq$ the relation of continuation among pt's, defined in 3.3.

$$sum(ss0, s0, sX_0) \quad \preceq \quad \dfrac{sum(ss0, 0, X_1)\ ;\ Ax_1}{sum(ss0, s0, sX_1)}\,\mathrm{P} \quad \preceq \quad \dfrac{\dfrac{Ax_0}{sum(ss0, 0, ss0)}\,\mathrm{P}\ ;\ Ax_1}{sum(ss0, s0, sss0)}\,\mathrm{P}$$

In this way every SLD-derivation can be translated into a pt and this yields the proof-theoretical equivalence between the two systems (see [10, 11]).

3 Regular AAR-Systems

The search-space of an AAR-system can be organized as a search-tree, where nodes are (partial) proof-trees and search steps are continuations. Leaves are pts without continuations. A leaf that contains a proof is a *success node*, and a leaf that contains a partial proof is a *failure node*. In general, search in the *complete* tree is untractable, and one tries to restrict it to a subspace. This relates to the idea of regular search space, as follows.

A *subspace* is obtained through a suitable equivalence relation among proof-trees, i.e. it is built by a quotientation of the (entire) search space. Regularity is a property of the equivalence classes. It ensures that a regular subspace is *success-complete*, that is the following property holds: for every successful path from a goal-expression E to a proof Π in the search space, the subspace contains at least one path from the equivalence class of E to the one of Π. Thus regularity entails that a search strategy working on representatives of the equivalence classes will not miss success nodes. One of the problems is computing the right substitutions. It can be dealt with through a quotientation by a suitable similarity relation, that mimics top-down proof-search and agrees with the subsumption ordering on proof-trees.

3.1 Search Spaces for AAR-Systems

Under the the more general version that we are developing, a pt Π may be in $\mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$, while $\theta\Pi$ is not. To ensure this, we introduce the following:

Definition 3.1 We say that an AAR-system $\langle \mathcal{E}, \mathcal{A}, \mathcal{R} \rangle$ is *closed under substitution* if $E \in \mathcal{E}$ entails $\theta E \in \mathcal{E}$ and, for all $R \in \mathcal{R}$, $A \in \mathcal{A}$ and $E \in \mathcal{E}$, $R(\theta E, A) = \theta R(E, A)$.

One easily sees that, if $\langle \mathcal{E}, \mathcal{A}, \mathcal{R} \rangle$ is closed under substitution, so is $\mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$, i.e. $\Pi \in \mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$ entails $\theta\Pi \in \mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$.

This allows us to introduce the following *pre-ordering* (intuitively to be read as Π_1 is less general or more instantiated than Π_2) and *equivalence* relation among proof-trees.

Definition 3.2

- $\Pi_1 \leq \Pi_2$ iff there is a θ such that $\Pi_1 = \theta\Pi_2$;
- $\Pi_1 \equiv \Pi_2$ iff $\Pi_1 \leq \Pi_2$ and $\Pi_2 \leq \Pi_1$;

Remark that the induced equivalence relation on proof-trees corresponds to identity of proof-trees modulo renaming of variables.

Now, let us consider how we could approach the following *search problem* in a Prolog-like way, where (finite) sets of axioms are programs and the desired outcome of the computation are answer substitutions.

Let $\mathcal{P}$ be a program and $E \in \mathcal{E}$ a goal-expression: search for a proof $\Pi :: \theta E$ with axioms from $\mathcal{P}$, for some substitution θ.

If a proof (i.e. a proof-tree *without assumptions*) $\Pi :: \theta E$ exists, we say that θ is an answer substitution for E w.r.t. $\mathcal{P}$. If, on the other hand, every proof-tree $\Pi :: \theta E$ with axioms from $\mathcal{P}$ has open assumptions, we say that E fails w.r.t. $\mathcal{P}$.

We characterize our complete search space, which contains *all* the proof-trees, through the following notion of *one-step continuation*.

Definition 3.3 Given $E \in \mathcal{E}$, $Ax \in \mathcal{P}$ and $R \in \mathcal{R}$, we say that Ax can be *applied* to E by R using θ iff there is at least one $E_1; \ldots; E_n \in R(\theta E, Ax)$. Given a proof Π with an assumption H such that $\theta E = \theta H$, the above application gives rise to a *one-step continuation with selected assumption* H, as follows:

$$\dfrac{E_1;\ \cdots\ ; E_n\ ;\ A}{\begin{array}{c}\theta H \\ \theta\Pi\end{array}}\,(\mathrm{R})$$

The *continuation* relation is the reflexive and transitive closure of one-step continuations and can be characterized as follows. Call Π' an *initial subtree* of Π iff Π' is a subtree of Π and they have the same root:

Proposition 3.1 Π_2 is a *continuation* of Π_1, denoted $\Pi_1 \preceq \Pi_2$, iff there is an initial subtree Π_3 of Π_2, s.t. $\Pi_3 \leq \Pi_1$.

One immediately sees that E has an answer substitution θ w.r.t. a program $\mathcal{P}$ iff there is a continuation $\Pi :: \theta E$ of the trivial (0-depth) proof-tree E without assumption. Then our search problem can be restated as follows:

Let $\mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$ be the set of proof-trees of a AAR-system, $\mathcal{T}(\mathcal{P}) \subseteq \mathcal{T}(\mathcal{E}, \mathcal{A}, \mathcal{R})$ be the (sub)set of the pt's with axioms from $\mathcal{P}$ and $\mathcal{T}(\mathcal{P}, E) \subseteq \mathcal{T}(\mathcal{P})$ be the (sub)set of the continuations of E. $\preceq$ is easily seen as a pre-ordering on each of those sets. To obtain a partial ordering we have to take the quotient $\mathcal{T}(\mathcal{P})/\equiv$ (i.e. consider proof-trees modulo variable renaming). Finally, take the graph (that through standard duplications can be treated as a tree with root E):

$$\langle \mathcal{T}(\mathcal{P}, E)/\equiv, \preceq \rangle$$

The leaves are (equivalences classes) of pt's which have no continuation; in particular, a *success node* is a leaf containing a proof.

$\langle \mathcal{T}(\mathcal{P}, E)/\equiv, \preceq \rangle$ is the *total* search space, which is the starting point of our analysis of regularity. It contains *all* the proof-trees (modulo renaming) and our search problem corresponds to the search of success nodes in such a tree.

For every node $[\Pi]$, where square brackets denote the equivalence class witnessed by Π, its children can be characterized as follows. $[\Pi']$ is a children of $[\Pi]$ iff there are an assumption E of $[\Pi]$, an axiom Ax from the program $\mathcal{P}$, a rule R from $\mathcal{R}$ and a substitution θ s.t. Ax can be applied to E by R using θ, yielding $[\Pi']$. We will say that $[\Pi']$ is an (E, Ax, R, θ)-children of $[\Pi]$.

Sequences $((E_0, Ax_0, R_0, \theta_0), \ldots, (E_n, Ax_n, R_n, \theta_n))$ correspond to paths in the tree. Thus, in general, we may have to backtrack on four *dimensions* (choices of $(E_k, Ax_k, R_k, \theta_k)$). Moreover, due to the presence of substitutions, even using a finite set of axioms and rules, a node may have infinitely many children. Consequently it is desirable to eliminate at least (and especially) the need to backtrack on substitutions. This is what is achieved by first-order resolution, thanks to the existence of most general unifiers. Moreover, *SLD*-resolution eliminates the choice of the selected goal E [8]. In our model this is reflected by proposition 3.5.

3.2 Regular Search Spaces

In our model the possibility of using a resolution-like method corresponds to the computation of *most general continuations*, among the (possibly infinite) *similar continuations.*

To informally motivate the notion of similarity, let us consider a path in the tree $\langle \mathcal{T}(\mathcal{P}, E_0)/\equiv, \preceq\rangle$. Let us call *similar* two paths determined by the same sequence of selected assumptions, axioms and rules, but possibly by different sequences of substitutions. Two proof-trees are *similar* if their paths from the root are similar. The idea is that an idealized interpreter, to avoid backtracking on substitutions, will follow one among similar paths by selecting a pt in a class of similar pt's. Similarity can be defined in a more abstract way, as a *structural* property of proof-trees:

Definition 3.4 An *axiom/rule-occurrence* in a pt Π is a triple $\langle p, A, R\rangle$ such that p is a path in Π from the root to a node containing an axiom A applied by a rule R. We say that two proof-trees Π_1, Π_2 are *similar* , written $\Pi_1 \sim \Pi_2$, if they have the same (non-empty) set of axiom/rule-occurrences.

One easily sees that $\sim$ is an equivalence relation; the corresponding equivalence classes, indicate by $[\Pi]_\sim$, will be called *similarity classes.*

We use similarity to curtail the subspace $\langle \mathcal{T}(\mathcal{P}, E)/\sim, \preceq\rangle$, where the continuation relation $\preceq$ has been lifted to similarity classes as follows:

$$[\Pi_1]_\sim \preceq [\Pi_2]_\sim \text{ iff there are } \Pi_1 \in [\Pi_1]_\sim, \Pi_2 \in [\Pi_2]_\sim \text{ s.t. } \Pi_1 \preceq \Pi_2 \qquad (1)$$

An interpreter that does not perform backtracking on substitutions chooses suitable representatives of the equivalence classes, and different choices correspond to different search strategies. Since (1) does not require that every representative Π of $[\Pi_1]_\sim$ has a continuation in $[\Pi_2]_\sim$, a complete search strategy has to choose a 'good representative' of $[\Pi_1]_\sim$, i.e. a $\Pi \in [\Pi_1]_\sim$ that has a 'good representative' of $[\Pi_2]_\sim$, as a continuation. Representatives could be most general proof-trees:

Definition 3.5 Let $[\Pi]_\sim$ be a similarity class. A pt Π^* is a *most general proof-tree* in $[\Pi]_\sim$ iff, for every pt $\Pi' \in [\Pi]_\sim$, $\Pi' \leq \Pi$.

If a mgpt Π^* exists, then it is unique up to renaming and all the pt's of the node $[\Pi^*]$ of the search tree $\langle \mathcal{T}(\mathcal{P}, E)/\equiv, \preceq\rangle$ are mgpt in the class $[\Pi]_\sim$. Moreover, Π^* is representative of $[\Pi]_\sim$, because it subsumes every other pt of the class. Finally, if every similarity class contains a mgpt, then mgpt's are *good representatives/*, as stated by the following proposition.

Proposition 3.2 Let Π_1^*, Π_2^* be mgpt's. Then $[\Pi_1^*]_\sim \preceq [\Pi_2^*]_\sim$ iff $\Pi_1^* \preceq \Pi_2^*$.

Unfortunately, in general, a similarity class may not contain a mgpt. *Regularity* is the basis for the existence of most general proof-trees among similar trees:

Definition 3.6 A set $\mathcal{S}$ of proof-trees is a *regular search space* iff, for every similar $\Pi_1, \Pi_2 \in \mathcal{S}$, there is a $\Pi \in \mathcal{S}$ such that $\Pi_1 \leq \Pi$ and $\Pi_2 \leq \Pi$.

Proposition 3.3 If $\mathcal{S}$ is regular, then every similarity class $[\Pi]_\sim$ contains a most general proof-tree, i.e. a proof-tree Π^* such that, for every $\Pi' \in [\Pi]$, $\Pi' \leq \Pi^*$.

Let $\mathcal{S}$ be a regular search space and $\mathcal{T}(\mathcal{S}, E)$ the set of the proof-trees $\Pi :: \theta E \in \mathcal{S}$; define $Gen(\mathcal{S})$ and $Gen(\mathcal{S}, E)$ to be the corresponding sets of mgpt's. By proposition 3.2, the subspace $\langle \mathcal{T}(\mathcal{S}, E)/\sim, \preceq\rangle$ is isomorphic to $\langle Gen(\mathcal{S}, E)/\equiv, \preceq\rangle$. To analyze the properties of this subspace, and to understand the underlying geometry, we introduce the notion of *canonical continuation* of a pt.

Definition 3.7 A continuation Π^* of a pt Π is a *most general continuation* (*mgc*) if, for every other similar continuation Δ, $\Delta \leq \Pi^*$. A continuation is *canonical* iff it is a one-step mgc.

Proposition 3.4 If Π is a mgpt, then its mgc's are mgpt's. In particular, its canonical continuations are mgpt's.

By the above proposition $\langle Gen(\mathcal{S}, E)/\equiv, \preceq\rangle$ can be built using only canonical continuations, thus avoiding even to take note of substitutions. Moreover, we can prove:

Proposition 3.5 Let Π be a mgpt of $\mathcal{S}$ and H be an assumption of Π. If there is a proof Δ that is a mgc of Π, then there is a canonical continuation Π' of Π selecting H such that Δ is a mgc of Π'.

Proposition 3.5 shows that, by using canonical (i.e. most general) continuations, the selection of the assumption E_k during search may be completely non deterministic. Thus we can further reduce the search space by using selection functions F which associate to every pt one of its assumptions. An *F-search tree* is a subtree of $\langle Gen(\mathcal{S}, E)/\equiv, \preceq\rangle$ such that, for every node $[\Pi]$, its children are the canonical continuations selecting the assumption $F(\Pi)$. This is a second reason, beyond eliminating backtracking on substitutions, for stressing the relevance of regularity in logic programming.

Now we say that an *AAR*-system is *regular* iff the set of its proof-trees is a regular search space. As one can easily see, the regularity of the *AAR*-system implies the regularity of the subspaces $\mathcal{T}(\mathcal{P})$ and $\mathcal{T}(\mathcal{P}, E)$. We can avoid backtracking on substitutions and search only for most general proof-trees, and we can use F-search trees.

The problem is then how to compute canonical continuations. We say that there is a *resolution method* when canonical continuations can be computed depending on the selected assumption and not on the whole pt.

Definition 3.8 An operator *Res* is a *resolution method* iff, for every goal-expression E, axiom Ax, rule R, $Res(E, Ax, R)$ is defined iff Ax can be applied to E by R and yields an equivalence class (w.r.t. renaming) of the form $[\langle\theta, E_1; \ldots; E_n\rangle]$ and

$$Res(E, Ax, R) = [\langle\theta, E_1; \ldots; E_n\rangle]$$

iff $E_1; \ldots; E_n \in R(\theta E, Ax)$ and the corresponding continuation is canonical.

If the search space is regular, then for every R, Ax and every pt with selected assumption E, either there is canonical continuation ($Res(E, Ax, R)$ is defined) or no continuation exists ($Res(E, Ax, R)$ is not defined). Moreover, for the same R, Ax and selected E, any two canonical continuations are equivalent; therefore $Res(E, Ax, R)$ has been defined as an operator computing equivalence classes. This operator imports all the search properties of pure Prolog, in particular the independence of the selection function.

Example 1 The P-system is regular (see [10]) and its resolution method is defined as follows. Let Ax be $\forall(\forall y(L_1 \wedge \ldots \wedge L_n) \rightarrow H)$ and L be a literal. If θ is an idempotent mgu of H and L, then $Res(L, Ax, \mathtt{P}) = [\langle\theta, \theta L_1; \ldots; \theta L_n\rangle]$. If H and L do not unify, then Res is undefined. Note the complete analogy with SLD-resolution. Observe that no substitution is attempted on the variables y. This is sound for such a simple form of axioms that involve literals only. With more general rules and axioms, there are cases where eigenvariables must be renamed by *new* names, if they occur in the current proof-tree. Analogously, to obtain most general continuations, the variables of the current proof-tree that occur free in $\forall y(L_1 \wedge \ldots \wedge L_n)$, but *not* in H, must be renamed. ◇

4 (PF)-Systems and SLDNF-Resolution

In general, failure (finite or not) simply means unprovability. However, after Clark [4], finite failure w. r. t. of logic programs is interpreted also as provability of negated formulas from the completion. To study this approach, we first introduce the F and PF-systems. Then we use them to provide a proof-theoretic interpretation of $SLDNF$-resolution.

4.1 The non-Regular F and PF-Systems

In Clark's equality theory we can derive the following *failure rule* F to apply *failure axioms* of the form: $\forall x(p(x) \rightarrow \exists y.(x = t_1 \wedge L_1) \vee \cdots \vee (x = t_n \wedge L_n))$, related to the weak completion that we will introduce later:

$$neg(\sigma_{i_1} L_{i_1}); \ldots; neg(\sigma_{i_k} L_{i_k}) \in \mathtt{F}(\neg p(a), \forall x(p(x) \rightarrow \exists y.(x = t_1 \wedge L_1) \vee \cdots \vee (x = t_n \wedge L_n)))$$

where: $neg(A) = \neg A$ and $neg(\neg A) = A$; $a, t_1, \ldots, t_n$ are terms; for $1 \leq h \leq k$, t_{i_h} and a unify with idempotent mgu σ_{i_h}, while the others do not. Moreover, if L_{i_m} contains *local* variables, i.e. variables that do not appear in t_{i_m}, those variables are substituted in $\sigma_{i_m} L_{i_m}$ by *new* (w.r.t. $p(a)$) eigenvariables.

The PF-system contains the rules P and F.

The F and PF-systems are non-regular systems. Consider for example the following proof-trees Π_1, Π_2, Π_3, where the failure axiom is $\forall x(odd(x) \rightarrow \exists y(x = s(0) \wedge true \ \vee \ x = s(s(y)) \wedge odd(y)))$:

$$\dfrac{Fax}{\neg odd(0)}(\mathtt{F}) \qquad \dfrac{\neg odd(W) \quad Fax}{\neg odd(s(s(W)))}(\mathtt{F}) \qquad \dfrac{\neg true \quad \neg odd(v) \quad Fax}{\neg odd(W)}(\mathtt{F})$$

they are similar, but there is no proof-tree Π such that $\Pi_i \leq \Pi$ for $i = 1, 2, 3$.

As far as the last proof-tree is concerned, note that, among these examples, it is the only one not closed under substitution. Second its derivation goes like this:

1. $\neg true$ is generated since W and $s(0)$ unify;
2. $\neg odd(v)$ is generated, since W unifies with $s(s(y))$ which is not substituted by the mgu $W = s(s(y))$ and v is the eigenvariable renaming the existentially quantified y.

In the absence of regularity, the notions of mgpt and of canonical continuation do not behave as expected. This means that we cannot find a success-complete resolution method unless we admit backtracking on substitutions. In particular Res has to compute many candidate substitutions and goal sequences. Thus strategies like $SLDNF$ cannot be success-complete, as we discuss below.

Notice that the F can be formulated in a way such that the $P(F)$-system is closed under substitutions, still it is not regular, i.e. regularity and closure under substitution are independent. This is also the case for higher-order Horn clauses [9], where closure but not regularity is guaranteed.

4.2 SLDNF-Refutations and PF-Proofs

We associate to a program its failure axioms, which will be applied by the F-rule. The starting point is the only-if part of the completion of a predicate $p(\ldots)$:

$$\forall x(p(x) \rightarrow \bigvee_{i=1}^{n} \exists y_i(x = t_i \wedge \bigwedge_{k=1}^{h_i} L_{i,k}))$$

For every $k_1, \ldots, k_n$ such that $1 \leq k_i \leq h_i$ we infer

$$Fax_{k_1,\ldots,k_n}(p): \quad \forall x(p(x) \rightarrow \exists y \bigvee_{i=1}^{n} (x = t_i \wedge L_{i,k_i}))$$

By convention, the failure axiom of a unit clause introduces the constant *true*, hence $h_i \geq 1$ is always fulfilled. Observe that these axioms contain *exactly* a singleton literal in every disjunct of the consequent. For a definite program P, $Fax(P)$ will indicate the set of its failure axioms in the latter sense.

Example 2 The only-if part of t (for *times*), the usual program for computing the product, is:

$$\begin{aligned}\forall(t(a,b,c) \rightarrow \ & \exists x(a = x \wedge b = 0 \wedge c = 0) \vee \\ & \exists x, y, z, w(a = x \wedge b = s(y) \wedge c = z \wedge t(x, y, w) \wedge sum(w, x, z)))\end{aligned}$$

From it we derive the failure axioms:

$$\begin{aligned}\forall(t(a,b,c) \rightarrow \exists x, y, z, w \quad & ((a = x \wedge b = 0 \wedge c = 0 \wedge true) \\ & \vee(a = x \wedge b = s(y) \wedge c = z \wedge t(x, y, w))) \\ \forall(t(a,b,c) \rightarrow \exists x, y, z, w \quad & ((a = x \wedge b = 0 \wedge c = 0 \wedge true) \\ & \vee(a = x \wedge b = s(y) \wedge c = z \wedge sum(w, x, z)))\end{aligned}$$

◇

Theorem 1 *Given a definite program P and a definite goal $\leftarrow A$, there is a finitely failed tree for $P \cup \{\leftarrow A\}$ iff there is a proof $\Pi :: \neg A$ in the F-system using only axioms from $Fax(P)$.*

To a normal program P we associate the set $WComp(P) = Ax(P) \cup Fax(P)$ of *weak completion axioms*, where $Ax(P)$, the *success* axioms, and $Fax(P)$, the *failure* axioms, are extended to normal programs in the obvious way.

The conjunction of the $Fax_{k_1,\ldots,k_n}(p)$ is notably weaker and does not imply, in general, the only-if part of $Comp(p)$. Therefore $WComp(P) \not\vdash Comp(P)$. Nevertheless this is enough to ensure the soundness of standard NF, as indicated in the next theorem.

Theorem 2 *Let P be a normal program and L a literal. If there is a finitely failed $SLDNF$-tree for $P \cup \{\leftarrow L\}$, then there is a PF-proof $\Pi :: neg(L)$ with axioms from $WComp(P)$. If there is a $SLDNF$-refutation of $P \cup \{\leftarrow L\}$ with answer substitution δ, then there is a PF-proof $\Pi :: \delta L$, with axioms from $WComp(P)$.*

4.3 On (Un)Soundness and (In)Completeness of NF

$SLDNF$-resolution has a non-logical behavior if open negative goals are selected. In our model, we can distinguish three different causes for that:

a) *Incompleteness* of the PF-system: there are literals θL such that $Comp(P) \models \theta L$, in classical logic but no PF-proof $\Pi :: \theta L$ exists.

b) *Soundness* problems: during the computation (logically unsound) substitutions on eigenvariables may occur.

c) *Success-completeness* problems: given an open goal $\leftarrow L$, $SLDNF$-resolution fails to return an answer, even if there is a PF-proof $\Pi :: \theta L$: the culprit can be found in the lack of regularity, which lies at the basis of the success-incompleteness of NF.

There are several reasons for (a). For example, failure axioms are weaker than the completion, even if they are sufficient to prove the soundness of $SLDNF$-resolution (see Theorem 2). Note that point (a) and (c) are independent: would the PF-system be complete w.r.t. classical logic, yet not regular, the success-incompleteness issue would not be solved. The discussion on point (a) is fully addressed in [11].

Soundness problems (b) are due to the fact that a non-legal substitution on the eigenvariables may be introduced in some continuation step, as shown by the following example, taken from [8].

$$\begin{aligned} Ax1 &: \quad p :- \neg q(X) \\ Ax2 &: \quad q(a) \end{aligned}$$

In standard Prolog, using an unsound selection function, the goal $\leftarrow \neg p$ succeeds (since $\leftarrow p$ fails), although it is not a logical consequence of the completion of the program. In our model, safeness (i.e. soundness) is enforced not by an external condition on the selection function, but by the usual proof-theoretic proviso on eigenvariables, i.e. that they cannot be instantiated by substitutions, as the following proof-tree shows. Once we have obtained the goal $q(u)$, with eigenvariable u, we cannot continue our proof-tree in a sound way; so we do not obtain any proof of $\neg p$

$$\frac{q(u) \quad p \rightarrow \exists x \neg q(x)}{\neg p}(\mathrm{F})$$

This shows that we have a natural way to distinguish proofs of negated goals (where such a proof has no assumptions) from partial proofs with assumptions that cannot be continued. The latter corresponds to unprovability.

We discuss (c) more extensively. We show that there are PF-proofs which are ignored by standard $SLDNF$-resolution and that the reason is the *non-regularity* of the PF-system.

Let us consider for example the following (non-stratified) program $EVEN$:

$$\begin{aligned} Ax1 &: \quad even(0). \\ Ax2 &: \quad even(s(X)) :- \neg even(X) \end{aligned}$$

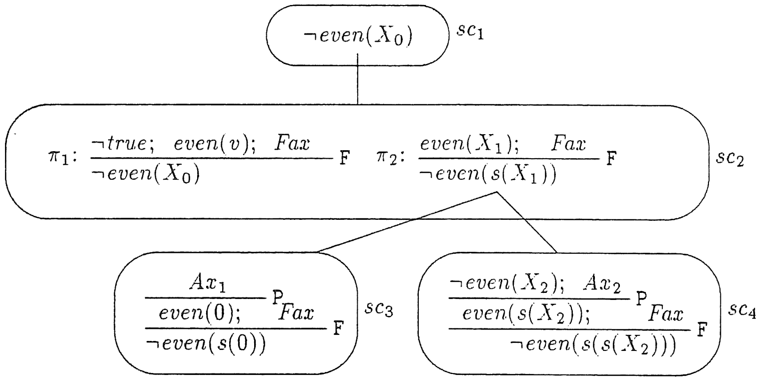
$Wcomp(EVEN)$ contains the obvious success axioms and one failure axiom:

$$Fax : \quad \forall x(even(x) \rightarrow \exists u(x = 0 \land true \ \lor \ x = s(u) \land \neg even(u)))$$

If we start from $\neg even(X)$, $SLDNF$-resolution yields a finitely failed (not safe) tree and no solution is found, even if the PF-system contains infinitely many proofs with axioms from

Ax_1, Ax_2, Fax and consequence $\neg even(\ldots)$. Success-incompleteness of $SLDNF$ is due to non-regularity, as the following picture shows.

Ovals contain similarity classes. Since $\neg even(X_0)$ is a mgpt, we can use it to generate all the one step continuations of the root class. All the continuations are similar (only Fax can be applied), and we get one children-class sc_2. The latter is non regular: it contains two maximal pt's, π_1 and π_2. Since the identical substitution is more general than $X_0/s(X_1)$, $SLDNF$ selects π_1, wich contains the unprovable assumption $\neg true$, and fails. On the other hand, π_2 has two children, and this shows that $SLDNF$-strategy *is search-incomplete.* Concerning sc_3, sc_4, the inscribed pt's are mgpt's. sc_3 is a success node, whilst sc_4 can be continued. Since the unique assumption that can be selected for continuation is $\neg even(X_2)$, sc_4 behaves as the root, i.e. it has one non-regular children-class.


$\neg even(X_0)$
sc_1
π_1: $\dfrac{\neg true; \ even(v); \ Fax}{\neg even(X_0)}$ F
π_2: $\dfrac{even(X_1); \quad Fax}{\neg even(s(X_1))}$ F
sc_2
$\dfrac{\dfrac{Ax_1}{even(0);}\mathrm{P} \quad Fax}{\neg even(s(0))}$ F
sc_3
$\dfrac{\dfrac{\neg even(X_2); \ Ax_2}{even(s(X_2));}\mathrm{P} \quad Fax}{\neg even(s(s(X_2)))}$ F
sc_4

When a similarity class contains a finite set of maximal pt's, backtracking on them provides success-completeness. An alternative solution is to split the (non-regular) failure axioms into suitable regular instances. Regular splitting is studied in [11], and it is connected to the method of constructive negation of [3, 13, 2].

5 Conclusions

Historically, the great majority of the papers on logic programming, in particular on its extension, has been carried out in a semantic way. Unfortunately these semantics, being mainly limited to term models, tend to hide proof-theoretic contents. We feel that many features that are rather cluttered in this framework instead become natural consequences of a proof-theoretic reading.

Basically two proof-theoretic approaches have been pursued in the literature, as outlined in [5]:

1. Clauses as axioms and some Gentzen calculus to infer goals [9, 12].

2. Clauses as rules [5]: programs should be seen as sets of inference rules (inductive definitions) for the derivation of (not necessarily ground) atoms.

We think of our approach as a development and enhancement of the latter: in full development, AAR-systems can be seen as a weak logical framework in the spirit of LF [7], PID [5] or hereditary Harrop formulae [9].

We have shown here that from this perspective much of the mystery of NF disappears or at least it is brought back to standard issues in basic proof-theory. This is more evidence of the fruitfulness and explicative power of the notion of *regularity* as an abstract characterization of SLD-resolution. For example, constructive negation can be seen as a side effect of regularizing normal programs.

We plan to continue this investigation, for instance concentrating on the completeness of PF-system w.r.t. the three-valued semantics of the (weak) completion.

In conclusion: while it is clear that the first steps for a new framework are the formalization of more or less well-known problems in the field, we are going to show that the theory of regular search spaces can be fruitfully used in more front-line subject such as program transformation, abduction et. al.

References

[1] Apt K.A. & Bol R. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, to appear.

[2] Barbuti R., Mancarella P., Pedreschi D. & Turini F. A Transformational Approach to Negation in Logic Programming. *Journal of Logic Programming*, pp. 201–228, 1990.

[3] Chan D. Constructive Negation Based on the Completed Database. In : *Proc. 5th Conf. Logic Programming*, Kowalski B. & Bowen K. (eds.), pp. 111 – 125, 1988.

[4] Clark K.L. Negation as Failure. In: *Logic and Data Bases*, Gallaire H. & Minker J. (eds.), Plenum Press, New York, pp. 293 – 322, 1978.

[5] Hallnäs L. & Schroeder-Heister P. A Proof-Theoretic Approach to Logic Programming: Clauses as Rules. *Journal of Logic and Computation*, v.1 no. 2. pp. 261–283, 1990, v.1 no. 5. pp. 635–660,1991.

[6] Harland J. *On Hereditary Harrop Formulae as a Basis for Logic Programming.* Ph.D. Thesis, Edinburgh, 1991.

[7] Huet G. & Plotkin G. (eds.). *Logical Frameworks* , Cambridge University Press, 1991.

[8] Lloyd J.W. *Foundations of Logic Programming.* Second Extended Edition, Springer-Verlag, Berlin, 1987.

[9] Miller D., Nadathur G., Pfenning F., Scedrov A. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51, pp. 125–157, 1991.

[10] Momigliano A. & Ornaghi M. Regular Search Spaces as a Foundation of Logic Programming. In: *Extensions of Logic Programming*, Dyckhoff R. (ed.), LNAI n. 798, Springer-Verlag, 1994.

[11] Momigliano A. & Ornaghi M. Regular Search Spaces and Constructive Negation. *Journal of Logic and Computation*, to appear.

[12] Stärk R. *The Proof-Theory of Logic Programs with Negation.* PhD. Thesis, University of Bern, 1992.

[13] Stuckey P. Constructive Negation for Constraint Logic Programming, *Proceedings of the 1991 IEEE Symposium on Logic in Computer Science*, pp. 328 – 339, 1991.

A framework for a transformational approach to negation

Josep Humet

Departament d'Informàtica i Matemàtica Aplicada
Universitat de Girona
Av. Lluís Santaló s/n., E-17071 Girona, Spain
Phone: 34 72 418417 - Fax: 34 72 418399
E-mail: humet@ima.udg.es

Abstract

In this paper we introduce *framed normal programs* as a tool for computing negation by program transformation in CLP over the Herbrand universe FT with infinite function symbols. In a framed normal program there are two kinds of predicates: those with negation already computed by some dual predicates (the frame), and those with negation still to be computed by constructive negation.

Framed normal programs are used as an intermediate step of a process for transforming normal programs into positive programs. In particular the aim is to compute a positive program called a *t-completion* which has the full semantics (positive and negative) of the original normal program.

Keywords: Constraints; Constructive Negation; Finite_Trees; Program Transformation.

1 Introduction

Negation as failure rule is the most classical way to handle negative subgoals, but it only works when they are ground. Constructive negation is an extension of the negation as failure rule to handle non-ground negative subgoals. Both, in essence involve taking a negative goal, running the positive version, and negating the answers. In Chan's [3] approach of constructive negation this is done "at the end" when all the answers for the positive version have been computed, and in Stuckey's [10] approach this is done "meanwhile" the answers are being computed. This difference makes Stuckey's approach being "more complete" than Chan's.

In [2] a transformational approach to negation is given, assuming the domain closure axiom. This assumption allows one to compute the complement of a term as a finite disjunctions of terms. (The complement of a term t is the set of all ground terms which are not instances of t.) When trying to satisfy a negative goal candidates for negation to succeed are computed, then by negation as failure they are checked and rejected if it is the case. The main reason for that work was to allow a symmetric treatment of positive and negative knowledge, because negation as failure (the paper was written in 1988) was too restrictive.

Efficiency is another reason for studying a transformational approach to negation. In both techniques, negation as failure and constructive negation, computing negation is done "indirectly" and one could ask whether would it be possible to handle negation in general by only positive means, that is to solve negative subgoals "directly". A program with negation been already computed is presumable to be more efficient than another (equivalent) but with negation still to be computed.

As said above, the restriction imposed in [2] of assuming the domain closure axiom is a consequence of the need to compute term complements. We believe that this restriction is not needed if we are allowed to write disequations in our clauses, i.e. if working in the framework of CLP(FT), where constraints are equations and disequations over the set of finite terms. In particular, let X be a set of variables and $t[X]$ be a term upon these variables. Then, the complement of the term $t[X]$ may be described by the disequation $\forall X(T \neq t[X])$ where T is a free variable. Because computing negation is ultimately related to computing the complement of a term and because disequations describe it, then positive programs enlarged with these constraints have a enough expressive power. In some sense we want to show how sufficient they are to express negation.

In our approach we introduce *framed normal programs* as an intermediate step of a process for transforming normal programs into positive programs. In a framed normal program there are two kinds of predicates: those with negation already computed by some dual predicates (the frame), and those with negation still to be computed by constructive negation. Then given a framed normal program P we compute by transformation techniques [8] a new framed normal program P' such that $\text{SEM}(P) = \text{SEM}(P')$ for some suitable semantic function SEM and such that the frame of P' includes that of P. Our aim is to compute, after some finite number of transformations, a program $\widehat{P}$ having all its predicates in the frame. Then $\widehat{P}$ will be a positive program (with disequations), which we will call a *t-completion* of P, satisfying that its semantics is equivalent to the full normal semantics of P.

The main problem comes from the existence of (local) variables in the body of a program clause which do not occur in the head. For this reason termination can not be currently guaranteed, although in many cases it may be overcamed. Abstract interpretation [4] could be a proper tool to analyze what could be done for reducing that kind of incompleteness.

In section 2 we introduce *framed normal programs*. First, in subsection 2.1 we define normal and positive programs in CLP(FT). Next, in subsection 2.2 we define a *frame* as a one-to-one correspondence between two disjoint sets of predicates listing those predicates whose negation have already been computed. Finally, in subsection 2.3 we define the semantic function SEM that has to be preserved under program transformation and define what we mean by a *t-completion* of a normal program.

In section 3 we describe a transformation procedure. Subsections 3.1 and 3.2 give the rules that are applied in the transformation distinguishing *deniable* predicates (those which do not have any local variable in the bodies of its definition) from non-deniable ones. Subsection 3.3 deals with termination and gives a rule to "complete" the program in a particular case of non-termination of the transformation procedure.

In section 4 we give three examples.

Finally, in section 5 we draw some conclusions.

2 Framed Normal Programs

2.1 Normal Programs

Let L be a language containing infinite function symbols and infinite predicate symbols. We assume that all our programs are written using symbols taken from L. Hence, the Herbrand Universe and the Herbrand Base are constructed from L, independently of specific programs. From now on, all predicates and all functions are those from L.

An *equation* is a formula of the form $s = t$, also $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$. Let E be an equation. We denote with $\text{VAR}(E)$ the set of all variables of E.

A *disequation* is a formula of the form $\forall X(s \neq t)$, also $\forall X(s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n)$, where X is the set of quantified variables of the disequation. We assume that quantified variables of a disequation do not appear elsewhere. Let $\forall XD$ be a disequation. We denote with $\text{VAR}(\forall XD) = \text{VAR}(D) \backslash X$ the set of free variables of the disequation.

A *constraint* will be either an equation or a disequation. A *system* (of constraints) is any conjunction of equations and disequations. A *solution* of a system $C_1 \wedge .. : \wedge C_r$ is a substitution σ with domain $\text{DOM}(\sigma) \subseteq \text{VAR}(C_1) \cup \ldots \cup \text{VAR}(C_r)$ such that for every equation $C_i \equiv s = t$ is $s\sigma = t\sigma$ and for every disequation $C_j \equiv \forall X(s \neq t)$ and for every substitution τ with $X \subseteq \text{DOM}(\tau)$ holds $s\tau\sigma \neq t\tau\sigma$.

An *atom* is a formula of the form $p(t)$ where p is a predicate and t is a term or a tuple of terms accordingly to the arity of p. A *literal* is an atom $p(t)$ or its negation $\neg p(t)$. The first one is called *positive* and the second one is called *negative*.

A *normal goal* is a formula of the form $\leftarrow B_1, \ldots, B_n$ where each B_i is either a literal or a constraint. A normal goal is *positive* if all its literals are positive.

A *normal clause* is a formula of the form $A \leftarrow B_1, \ldots, B_n$ where (the head) A is an atom and each B_i (in the body) is either a literal or a constraint. A normal clause is *positive* if all the literals in its body are positive.

A *normal program* is a set of normal clauses. A normal program is *positive* if all its clauses are positive.

The semantics for normal programs (see e.g. [6]) we consider here are three-valued Clark's completion of a program as the declarative semantics, and constructive negation [10] as the operational semantics. In particular we consider the success set $SS(P)$ and the finite failure set $FF(P)$ for a normal program P. The success set of P collects the answer constraints $p(T) \leftarrow c$ to simple goals $p(T)$. The finite failure set collects the set of simple goals $p(T) \leftarrow c$ which are finitely failed. Also, $GSS(P)$ and $GFF(P)$ will be the ground success set and the ground finite failure set, respectively. The first one is the set of all ground atoms $p(t)$ for which there is some answer $p(T) \leftarrow c$ in $SS(P)$ and $T = t \wedge c$ is satisfiable. The second one is the set of all ground atoms $p(t)$ for which there is some finitely failed simple goal $p(T) \leftarrow c$ in $FF(P)$ and $T = t \wedge c$ is satisfiable.

2.2 Frames

Let Γ and $\overline{\Gamma}$ be two disjoint finite sets of predicates. A $(\Gamma,\overline{\Gamma})$-*frame* is a one-to-one correspondence between Γ and $\overline{\Gamma}$. Let Φ be a $(\Gamma,\overline{\Gamma})$-frame and let $(p,\overline{p}) \in \Phi$. Then we say that $\overline{p}$ is the *dual* of p (in Φ).

Let P be a normal program and Φ be a $(\Gamma,\overline{\Gamma})$-frame. We say that P and Φ are *compatible* iff for every negative literal $\neg q(t)$ occurring in the body of any clause in P then q is not in Γ nor in $\overline{\Gamma}$. Let G be a normal goal and let Φ be a $(\Gamma,\overline{\Gamma})$-frame. We say that G and Φ are *compatible* iff for every negative literal $\neg q(t)$ occurring in G then q is not in Γ nor in $\overline{\Gamma}$.

Let P be a normal program and Φ be a frame, compatible with P. We say that Φ is a *correct* frame for P iff for every predicate p such that $(p,\overline{p}) \in \Phi$ and every t it holds $P \models \exists(\neg p(t))$ iff $P \models \exists(\overline{p}(t))$ and $P \models \exists(\neg\overline{p}(t))$ iff $P \models \exists(p(t))$. (Or if t is a ground term then it holds $p(t) \in GFF(P)$ iff $\overline{p}(t) \in GSS(P)$ and $\overline{p}(t) \in GFF(P)$ iff $p(t) \in GSS(P)$.)

A *framed normal program* is a pair (Φ, P) where Φ is a frame, P is a normal program, and Φ is correct for P.

Example. The frame $\{(p,\overline{p})\}$ is correct for the program $\{(p(X) \leftarrow \neg q(X)), (\overline{p}(X) \leftarrow q(X)), (q(X) \leftarrow X \neq a)\}$ but it is not for the program $\{(p(X) \leftarrow \neg q(X)), (\overline{p}(X) \leftarrow q(X), \neg q(X)), (q(X) \leftarrow X \neq a)\}$. Also the goal $(\leftarrow \overline{p}(f(X)), \neg q(X))$ is compatible with the frame above.

Let P be a normal program and let Φ be a correct $(\Gamma,\overline{\Gamma})$-frame for P. Then we say that all predicates in $\Gamma \cup \overline{\Gamma}$ are *fully defined* (syntactically) and those not in Γ nor in $\overline{\Gamma}$ are only *semidefined* (in this frame). We call *explicit* all those predicates defined by P which are not in $\overline{\Gamma}$ and we call *implicit* all predicates in $\overline{\Gamma}$. Let us observe that $\emptyset$ is a correct frame for every normal program and that all predicates are then semidefined and explicit.

2.3 The SEM function

Let P be a normal program and Φ a correct frame for P. Let Π be a subset of the explicit predicates. Then we define

$$\text{SEM}_\Pi(\Phi, P) = \{p(t)/p(t) \in GSS(P)_{|\Pi}\} \cup \{\neg p(t)/p(t) \in GFF(P)_{|\Pi}\}$$

where $X_{|\Pi}$ means the restriction of X to the atoms constructed only with predicates from Π.

Let p be a predicate from Π. If p is fully defined then $\neg p(t) \in \text{SEM}_\Pi(\Phi, P)$ holds iff $p(t) \in GFF(P)$ and this holds iff $\overline{p}(t) \in GSS(P)$. (But this is not true when p is only semidefined.) Then, if all predicates in Π are fully defined we have

$$\text{SEM}_\Pi(\Phi, P) = \{p(t)/p(t) \in GSS(P)_{|\Pi}\} \cup \{\neg p(t)/\overline{p}(t) \in GSS(P)_{|\overline{\Pi}}\}$$

where $\overline{\Pi} = \{\overline{p}/p \in \Pi\}$. If, in addition, all explicit predicates in P are fully defined, then program P is positive and the whole set $\text{SEM}_\Pi(\Phi, P)$ may be computed only iteratively.

Let P be a normal program and let Φ be a correct frame for P. Let Π be the set of all explicit predicates of P. A *t-completion* of program (Φ, P) is a framed normal program $(\widehat{\Phi}, \widehat{P})$ defining at least all predicates in Π, such that $\text{SEM}_\Pi(\Phi, P) = \text{SEM}_\Pi(\widehat{\Phi}, \widehat{P})$, and with all its predicates fully defined.

3 A Transformation Procedure

The transformation technique proposed here is a non-deterministic stepwise procedure to compute a t-completion for a given framed normal program (in particular usual normal programs, those framed by $\emptyset$). It is based upon fold/unfold rules (see, e.g. [8]). Specifically we use the following rules: clause replacement, reversible goal replacement, definition introduction, reversible folding, and reversible unfolding.

The *definition* of a predicate p in a program is the set of all clauses whose head is an atom constructed with p and will be denoted by $\text{DEF}(p)$.

A *local variable of a clause* is any variable not occurring in the head of the clause but just occurring in its body as a variable in a literal, or in an equation or among the non-quantified variables of a disequation. A *deniable clause* is a clause without any local variable. An explicit semidefined predicate is *deniable* if its definition has only deniable clauses. Otherwise is non-deniable.

Let P be a program and Φ a correct frame for P. Let Π be the set of all explicit predicates of P. A transformation step applied to P will give a program P' and a frame Φ' correct for P' such that Π is a subset of (Π') the explicit predicates of P' and such that $\text{SEM}_\Pi(\Phi, P) = \text{SEM}_\Pi(\Phi', P')$. In some cases a step transformation will leave unchanged both the set Π of explicit predicates and the frame Φ. But in other cases one of them may be enlarged. Schematically there will be three cases:

1. $\Phi' = \Phi \cup \{(p,\overline{p})\}$ where p is an explicit semidefined deniable predicate of P, and $\Pi' = \Pi$,

2. $\Pi' = \Pi$ and $\Phi' = \Phi$,

3. $\Pi' = \Pi \cup \{r\}$ where r is a new predicate not explicit nor implicit in P, and $\Phi' = \Phi$.

The first case corresponds to the computation of the definition of $\overline{p}$. It is obtained by the product of the negation of each clause in the definition of p. Then the pair $(p, \overline{p})$ is added to the frame and all occurrences in the program of a negative literal on p are replaced by the corresponding positive literal on $\overline{p}$.

The second and third cases correspond to the application of fold/unfold rules to non-deniable clauses in order to *deniabilize* them without changing the frame. In the second case the definition of an already explicit semidefined predicate is modified. In the third case a definition for a new explicit predicate is added to the program.

Finally, let us observe that in some of the transformation rules given in section 3.2 a definition of *solved form* for disequations is required; but the correctness of the transformation procedure do not rely on it (see e.g. [5,9] for such a definition).

3.1 Deniable Predicates

Let Φ be a correct frame for a program P and let p be an explicit semidefined deniable predicate of P. Let C be a clause in the definition of p of the form

$$p(t[X]) \leftarrow B_1[X], \ldots, B_n[X]$$

where $t[X]$ denotes a tuple of terms accordingly to the arity of p, X the set of variables which occurs in $t[X]$, and $B_i[X]$ are literals or constraints (equations or disequations). Then we define a set of clauses NC(C) as the following:

$$\begin{array}{l} \overline{p}(T) \leftarrow \forall X(T \neq t[X]) \\ \overline{p}(T) \leftarrow T = t[X], neg\ B_1[X] \\ \vdots \\ \overline{p}(T) \leftarrow T = t[X], neg\ B_n[X] \end{array}$$

where T is a new variable —or an unrestricted n-tuple of variables if p has arity n— and negations $neg\ B_i[X]$ have to be interpreted in the following way:

1. $neg(s[X] = s'[X]) \equiv s[X] \neq s'[X]$,

2. $neg(\forall Y(s[X,Y] \neq s'[X,Y])) \equiv s[X,Y] = s'[X,Y]$,

3. $neg(q(t)) \equiv \neg q(t)$, and $neg(\neg q(t)) \equiv q(t)$,
   when q is an explicit semidefined predicate,

4. $neg(q(t)) \equiv \overline{q}(t)$, and $neg(\overline{q}(t)) \equiv q(t)$,
   when $(q, \overline{q}) \in \Phi$.

Example. The application of NC to the clause

$$p(X,X) \leftarrow \forall Y(X \neq f(Y)), q(X), r(X),$$

within the frame $\Phi = \{(r, \overline{r})\}$ gives the set of clauses

$$\begin{array}{l} \overline{p}(T_1,T_2) \leftarrow \forall X((T_1,T_2) \neq (X,X)) \\ \overline{p}(T_1,T_2) \leftarrow (T_1,T_2) = (X,X), X = f(Y) \\ \overline{p}(T_1,T_2) \leftarrow (T_1,T_2) = (X,X), \neg q(X) \\ \overline{p}(T_1,T_2) \leftarrow (T_1,T_2) = (X,X), \overline{r}(X). \end{array}$$

Let us now introduce a product operator of clause sets with variant heads. Let S_1 and S_2 be clause sets such that heads of all clauses of S_1 and S_2 are variants of an atom $q(T)$ where T is a variable or an unrestricted tuple of variables. Then we define the *product* of S_1 and S_2 by

$$S_1 \times S_2 = \{(q(T) \leftarrow B_1[T], B_2[T]) \;/\; \begin{array}{l}(q(T_1) \leftarrow B_1[T_1]) \in S_1, \\ (q(T_2) \leftarrow B_2[T_2]) \in S_2\}\end{array}$$

which is commutative and associative.

Let P be a normal program and Φ a correct frame for P. Let p be an explicit semidefined deniable predicate and $\text{DEF}(p) = \{C_1, \ldots, C_n\}$ (set of clauses) be its definition. Now we define $\text{DEF}(\overline{p}) = \text{NC}(C_1) \times \ldots \times \text{NC}(C_n)$ when $n > 0$ and $\text{DEF}(\overline{p}) = \{(\overline{p}(X))\}$ when $n = 0$. Then we can apply the following transformation rule which preserves semantics.

Negation rule. The framed normal program (Φ, P) may be transformed onto the framed normal program (Φ', P') where $\Phi' = \Phi \cup \{(p, \overline{p})\}$ and $P' = (P \cup \text{DEF}(\overline{p}))_{\neg p/\overline{p}}$ where $X_{\neg p/\overline{p}}$ means that every negative literal in X of the form $\neg p(t)$ has to be replaced by the positive literal $\overline{p}(t)$.

Actually the negation rule is a combination of the definition introduction rule and the goal replacement rule in all program clauses. From Clark's completion of a program and from the fact that in a framed program every pair of predicates $(s, \overline{s})$ in the frame are both the negation of each other it holds the following theorem which states the correctness of the above negation rule.

THEOREM. Φ' is a correct frame for P' and if Π is the set of explicit predicates defined by P then $\text{SEM}_\Pi(\Phi, P) = \text{SEM}_\Pi(\Phi', P')$.

3.2 Deniabilization

By *deniabilization* we mean a transformation procedure with the purpose of obtaining deniable definitions. Moreover, the following rules may be combined with the negation rule above so if we have in the program a deniable predicate p and also a non-deniable predicate q we do not have to wait for having a deniable definition of q; but we may compute the definition of $\overline{p}$. In some cases it will be necessary when definition of q involves $\neg p$.

Let P be a normal program, let Φ be a correct frame for P, let p be a predicate, and let C be a clause in $\mathrm{DEF}(p)$. We can transform C then yielding a new program P' by any of the following ways.

1. By *solving equations.* Let us suppose that in the body of clause C there is an equation constraint. Then we solve it and get a solved form for unification. Then we apply the resulting most general unifier to all the clause after eliminating the constraint. If the constraint is unsatisfiable then C is deleted.

2. By *'solving' disequations.* Let us suppose that in the body of clause C there is a disequation constraint which is not in solved form. If the disequation is valid it is eliminated from the clause, if it is unsatisfiable then C is deleted, otherwise the disequation is replaced by its solved form.

3. By *removing redundant disequations.* If in the body of clause C there is a disequation in solved form having at least one variable which does not occur in the head of C nor in any literal of the body neither in any equation of the body, then the disequation is redundant and may be removed from the clause.

4. By *deletion of subsumed clauses.* A clause $H \leftarrow D, B$ (where D are disequations) *subsumes* another clause $H' \leftarrow D', B', S$ (where D' are disequations) if there exists a substitution θ such that $H' \equiv H\theta$ and $B' \equiv B\theta$, and all disequations in $D\theta$ are entailed by some in D'. If there are clauses C and C' such that C subsumes C' then C' may be deleted.

5. By *unfolding.* Let us suppose p be an explicit semidefined and non-deniable predicate, and C is of the form $H \leftarrow A, B$, where A is an atom not constructed with p, and B is a (possibly empty) set of literals or constraints. Then we can unfold C with respect to A using program P as follows. If A is an atom of the form $q(T)$, then for each clause in the definition of q in the program of the form $q(T') \leftarrow Q$, we get a new clause $H \leftarrow T = T', Q, B$. Then we replace C by the set of all clauses obtained by this way.

   The restriction for A to be an atom not constructed with p is to prevent the unfolding to be not self-unfolding.

6. By *definition & folding for independent subgoals.* Let us suppose p be an explicit semidefined and non-deniable predicate, and C is of the form $H \leftarrow A, B$ where A and B are (possibly empty) sets of literals or constraints. The *linking variables* of subgoal A in C are the variables in $\mathrm{VAR}(A) \cap \mathrm{VAR}(H, B)$. If the set of linking variables of A in C is included in $\mathrm{VAR}(H)$, that is, all the variables in A which are not in the head of the clause do not appear elsewhere, then we say A is an *independent subgoal* in C. Let A be an independent subgoal in C and let $X = \mathrm{VAR}(H) \cap \mathrm{VAR}(A)$ be the set of variables of H which also occur in this independent subgoal. Then we add (definition introduction) a new predicate, let us call it a, whose definition is the only clause $a(X) \leftarrow A$. Then we replace (folding) C by the new clause $H \leftarrow a(X), B$.

Reversible unfolding, reversible folding, definition introduction, reversible goal replacement, and clause replacement preserve normal semantics. All the above rules fall into these categories. In particular, solving equations, solving disequations, removing redundant disequations, and deletion of subsumed clauses are instances of clause replacement rule or of reversible goal replacement rule. Unfolding as used here is actually reversible unfolding, and definition & folding for independent subgoals is a combination of definition introduction and reversible folding. Then we have the following

THEOREM. Let P be a normal program and let Φ be a correct frame for P. Let Π be the set of all explicit predicates defined by P. Let P' be a new normal program obtained from P by using one of the above rules. Then the same frame Φ is also correct for P' and $\mathrm{SEM}_{\Pi}(\Phi, P) = \mathrm{SEM}_{\Pi}(\Phi, P')$.

3.3 Termination

Let P_0 be a normal program, Φ_0 a correct frame for P_0, and Π the set of explicit predicates in P_0. Let us apply to P_0 the procedure given by the rules in previous subsections. Then we will get a sequence $(\Phi_0, P_0), (\Phi_1, P_1), \ldots, (\Phi_n, P_n)$ of framed normal programs. If all predicates in (Φ_n, P_n) are fully defined then this is a t-completion of program (Φ_0, P_0), thus the transformation process finishes. But termination may not always succeed.

Example. Let P be the two clause program $\{(p(X) \leftarrow p(X)), (r \leftarrow p(X))\}$. Predicate p is deniable but r is not. If we apply the unfolding rule to r the program will remain the same. So in that case we would go into an infinite process and never obtain a t-completion. On the other hand, if we apply the definition & folding for independent subgoals rule we deniabilize r but then we introduce a new non-deniable predicate and thus the problem remains: $\{(p(X) \leftarrow p(X)), (r \leftarrow s), (s \leftarrow p(X))\}$. If we would notice of the non-terminating process and the undefinedness of predicate r we would have to stop the computation of the t-completion. But another strategy could be to "define" r as undefined, and so $\overline{r}$. For that purpose we replace clause $r \leftarrow p(X)$ by $r \leftarrow r$. Then program P transforms into the program $\{(p(X) \leftarrow p(X)), (r \leftarrow r), (\overline{p}(X) \leftarrow \overline{p}(X)), (\overline{r} \leftarrow \overline{r})\}$ which is a t-completion of P.

A predicate p is *entirely undefined* (semantically) with respect to a program P iff both $SS(P)$ and $FF(P)$ do not contain any literal constructed with p. Now we introduce a new rule for overcoming some cases of non-termination.

U rule. Let P be a normal program and Φ a correct frame for P, such that for all explicit semidefined predicate p all the following conditions hold simultaneously.

1. DEF(p) has only one clause and it is of the form $p(X_1, \ldots, X_n) \leftarrow B$ where $X_1, \ldots, X_n$ are n different variables and all of them occur in B.

2. There is at least one local variable in B which does not occur among (X_i) those in the head.

3. The only rules that apply are definition & folding rule (this always may be applied) and perhaps unfolding. Other rules do not apply: solving equations, solving disequations, removing redundant disequations, and deletion of subsumed clauses.

4. If unfolding rule applies to DEF(p) then it always leaves DEF(p) unchanged after its application.

5. Definition & folding rule may be applied to $p(X_1, \ldots, X_n) \leftarrow B$ only if we fold it with respect the whole body B, thus introducing a new n-ary predicate q with the definition $q(X_1, \ldots, X_n) \leftarrow B$ and replacing the only clause in DEF(p) by the new clause $p(X_1, \ldots, X_n) \leftarrow q(X_1, \ldots, X_n)$.

Then for every explicit semidefined n-ary predicate p we replace the only clause in DEF(p) by the new clause $p(X_1, \ldots, X_n) \leftarrow p(X_1, \ldots, X_n)$ which is deniable, with negation $\overline{p}(X_1, \ldots, X_n) \leftarrow \overline{p}(X_1, \ldots, X_n)$.

If all the conditions required by the U rule hold then for every predicate p still semidefined there is no answer of the form $p(T) \leftarrow c$ in the success set nor finitely failed goal of the form $p(T) \leftarrow c$ in the finite failure set. So it holds the following

THEOREM. If the U rule applies to the framed program (Φ, P) giving (Φ', P') then $\text{SEM}_\Pi(\Phi, P) = \text{SEM}_\Pi(\Phi, P')$.

This new rule is not applicable in every case, so the question of how to know, in general, when a predicate p is entirely undefined is still open.

4 Examples

Example 1. Let P_1 be the following program framed by $\emptyset$:

$$q(a) \leftarrow$$
$$q(f(X)) \leftarrow X \neq Y, q(Y).$$

Then, the following is a t-completion of P_1 with frame $\{(q, \overline{q}), (r, \overline{r}), (s, \overline{s})\}$:

$$\begin{array}{ll}
q(a) \leftarrow & \overline{q}(T) \leftarrow T \neq a, \forall X(T \neq f(X)) \\
q(f(X)) \leftarrow r(X) & \overline{q}(f(X)) \leftarrow \overline{r}(X) \\
r(X) \leftarrow X \neq a & \overline{r}(a) \leftarrow \overline{s}(a) \\
r(X) \leftarrow s(X) & \\
s(X) \leftarrow . &
\end{array}$$

Example 2. Let P_2 be the program

$$p(f(X, X)) \leftarrow X \neq a$$
$$p(X) \leftarrow \neg p(f(X, Y)).$$

After some step transformations we can get from P_2 the new program

$$\begin{array}{ll}
p(f(X, X)) \leftarrow X \neq a & \overline{p}(T) \leftarrow \forall X(T \neq f(X, X)), \overline{q}(T) \\
p(X) \leftarrow q(X) & \overline{p}(f(a, a)) \leftarrow \overline{q}(f(a, a)) \\
q(a) \leftarrow \overline{q}(f(a, a)) & \overline{q}(X) \leftarrow X \neq a, \neg r(X) \\
q(X) \leftarrow r(X) & \overline{q}(a) \leftarrow \neg r(a), q(f(a, a)) \\
r(X) \leftarrow X \neq Y, \neg r(f(X, Y)) &
\end{array}$$

with frame $\{(p, \overline{p}), (q, \overline{q})\}$. The only semidefined predicate is r and all U-rule conditions hold. Then, we replace the only clause $r(X) \leftarrow X \neq Y, \neg r(f(X, Y))$ defining r by the the new one $r(X) \leftarrow r(X)$ and its negation $\overline{r}(X) \leftarrow \overline{r}(X)$ in order to get a t-completion of P_2 with frame $\{(p, \overline{p}), (q, \overline{q}), (r, \overline{r})\}$.

Example 3. It can be proved that the given transformation procedure is unable to compute, in a finite number of steps, a t-completion of the following program

$$p(X) \leftarrow p(f(X, Y))$$
$$p(X) \leftarrow \neg p(f(X, Y)).$$

5 Conclusion

We have introduced frames to qualify normal programs and define a function SEM based on them to be preserved under program transformation. Framed normal programs allows us to combine a partial computation of negation by a transformation technique in compilation-time with constructive negation in execution-time. We have developed a sound procedure for that purpose.

We believe that, given a normal program P, the method produces in most cases a positive program $\hat{P}$, called a t-completion of P, whose success set in some sense includes both success set and finite failure set of P. Also, further research in order to improve the transformation procedure and to know when a predicate is entirely undefined is needed. Nevertheless, if a t-completion of a program can not be computed, still the procedure is useful because of being able to compute it partially in compilation-time.

A user-guided transformation system based on the operations given in this paper have been implemented by the author.

6 References

1. Apt, K.R., and Bol, R.N., Logic Programming and Negation: A Survey, *J. Logic Programming*, 19,20:9-71 (1994).

2. Barbuti, R., Mancarella, P., Pedreschi, D., and Turini, F., A Transformational Approach to Negation in Logic Programming, *J. Logic Programming*, 8:201-228 (1990).

3. Chan, D., Constructive Negation Based On The Completed Database, *Proceedings of the 5th International Conference on Logic Programming*, 111-125 (1988).

4. Cousot, P., and Cousot, R., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points, *Proc. of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, pp. 238-252 (1977).

5. Humet, J., Implementation of a constraint solver for finite trees with infinite function symbols, Internal Report (7 pages), Departament d'Informàtica i Matemàtica Aplicada, Universitat de Girona, Spain, 1995.

6. Jaffar, J., and Maher, M.J., Constraint Logic Programming: A Survey, *J. Logic Programming*, 19,20:503-581 (1994).

7. Kirchner, C., and Lescanne, P., Solving Disequations, *Proceedings Second IEEE Symp. on Logic in Computer Science*, 347-352 (1987).

8. Pettorossi, A., and Proietti, M., Transformation of Logic Programs: Foundations and Techniques, *J. Logic Programming*, 19,20:261-320 (1994).

9. Smith,D.A., Constraint Operations for CLP(FT), *Proceedings of the 8th International Conference on Logic Programming*, MIT, Cambridge, pp. 766-774 (1991).

10. Stuckey, P.J., Constructive Negation for Constraint Logic Programming, *Proc. Logic in Computer Science Conference*, 328-339 (1991).

Ordered Logic and its Relationships to other Logic Programming Formalisms

Francesco Buccafurri
ISI – CNR e D.E.I.S., Universita' della Calabria
87036 Rende, Italy
email : bucca@si.deis.unical.it

Abstract

Ordered Logic ($\mathcal{OL}$) is a nonmonotonic logic programming language for defeasible and default reasoning.

The aim of this paper is to provide a better understanding of ordered logic by means of a comparison with other logic programming languages. The relationships beetween $\mathcal{OL}$ and traditional logic programming are investigated. It is formally proven that the stable model semantics for traditional logic programming can be simulated by a simple $\mathcal{OL}$ program. $\mathcal{OL}$ is then compared to other extensions of logic programming with classical negation proposed by Gelfond and Lifschitz and by Kowalsky and Sadri. A number of examples show to which extent $\mathcal{OL}$ provides support to model some form of nonmonotonic reasoning, compared with the above mentioned logic programming formalism.

Keywords: Knowledge Representation, Nonmonotonic Reasoning, Negation, Semantics, Stable Models.

1 Introduction

Considerable research has been conducted in the last few years in the area of nonmonotonic reasoning [2], with the objective of providing a precise mathematical theory of human commonsense reasoning (which is usually not monotonic). As a result, various different nonmonotonic logics have been developed [4, 16, 17, 18, 19, 20, 21]. The peculiarity of a nonmonotonic logic is that additional information may invalidate old conclusions, unlike classical logic which is inherently monotonic.

A recent proposal in this area is represented by Ordered Logic ($\mathcal{OL}$) [3, 10, 11, 12, 13], where nonmonotonicity is obtained by allowing *true* negation (i.e., the possibility of explicitly stating the falsity of atoms) in the context of an inheritance hierarchy (which assigns different levels of reliability to rules).

Informally, an ordered logic program ("$\mathcal{OL}$ program") is a set of *components* organized into an inheritance hierarchy. Each component consists of a set of rules

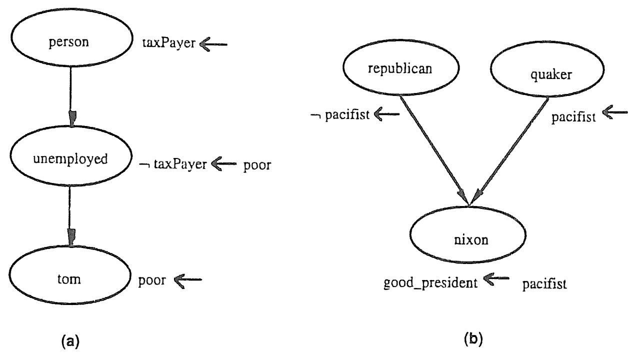


Figure 1: (a) the $\mathcal{OL}$ program $\mathcal{P}_{tom}$; (b) the $\mathcal{OL}$ program $\mathcal{P}_{nixon}$

which may have negative heads. As an example, consider the $\mathcal{OL}$ program shown in Figure 1.(a). Here, we have three components, each consisting of an *identifier* and a set of rules. The identifiers are *person*, *unemployed* and *tom*. Components are organized into a hierarchical structure, where *tom* is "lower" than *unemployed* which, in turn, is "lower" than *person*. Unlike traditional logic programming, rules may have negated heads. Thus, negation is considered as *true* negation, as a negative fact is true only if explicitly derived from a rule of the $\mathcal{OL}$ program (in other terms, a negative information is considered as valuable as a positive one). Like in the object-oriented approach, properties defined for the "higher" components in the hierarchy flow down to the "lower" ones. Hence, the contradicting conclusion *tom both pays and does not pay taxes* should hold in our case (as both rules $\neg taxPayer \leftarrow poor$ and $taxPayer \leftarrow$ hold for *tom*). However, this is not the case. Indeed, the "lower" rule $\neg taxPayer \leftarrow poor$ is considered as a sort of refinement to the first general rule, and thus the meaning of the $\mathcal{OL}$ program is rather clear: *tom does not pay taxes*, as he is both poor and unemployed. That is, $\neg taxPayer \leftarrow poor$ is preferred to the default rule $taxPayer \leftarrow$ as the hierarchy explicitly states the specificity of the former (with respect to the object *tom*). Intuitively, there is no doubt that $M = \{\neg taxPayer, poor\}$ is the only reasonable conclusion.

A property of the above $\mathcal{OL}$ program is that it provides sufficient information to choose exactly one from among all possible conflicting conclusions (indeed, one of them is more specific and, hence, can be considered preferable to any other). However, there are $\mathcal{OL}$ programs for which the way of solving conflicts is not unique (it may even not exist at all). As an example, consider the $\mathcal{OL}$ program $\mathcal{P}_{nix}$ reported in Figure 1.(b) which is a reformulation of the well-known "Nixon Diamond" [22]. Here, neither *pacifist* nor $\neg pacifist$ can be trusted more, as they are conflicts arising from rules none of which is preferable to the other. Thus, there are two (perfectly symmetric) ways of solving this conflict, either in favor of *pacifist* or of its contrary $\neg pacifist$. That is, any of the two possible conclusions $M_1 = \{pacifist, goodpresident\}$ and $M_2 = \{\neg pacifist\}$ can (nondeterministically)

be accepted as a possible meaning to be assigned to the $\mathcal{OL}$ program.

This paper focuses on the Stable Model Semantics for $\mathcal{OL}$ programs ($SMS^{\mathcal{OL}}$, for short) [3]. The stable model semantics assigns each $\mathcal{OL}$ program with a number of models (possibly zero), one for each way conflicts can be solved. Clearly, M_1 and M_2 are the two stable models of $\mathcal{P}_{nix}$, while M is the (unique) stable model of $\mathcal{P}_{tom}$.

The aim of the paper is to provide a better understanding of ordered logic (under $\mathcal{OL}$ stable model semantics) by means of a comparison with other logic programming languages. In particular, $\mathcal{OL}$ is compared to traditional logic programming (under stable model semantics) [5] and to extended logic programming [6, 9], which enrichs logic programming with classical negation.

The main result of the paper is the theorem proving that $\mathcal{OL}$ subsumes traditional logic programming [1] under stable model semantics. Informally, to see this point we note that, although $\mathcal{OL}$ interprets negation as true negation, the Closed World Assumption [2, 14, 21] can be easily mimicked: it is sufficient to make it explicit by introducing a component, say CWA, which consists of the negation of all possible facts, and such that any other component is "lower" (in the inheritance hierarchy) than CWA. Intuitively, CWA states that a fact is false unless it is explicitly contradicted. In this way, the user is free to choose the predicates where the closed world assumption is to be applied (by selecting the literals to be included in the CWA component). Thus, the $\mathcal{OL}$ approach to negation is more flexible compared to that of logic programming (where the closed world assumption is applied to *all* predicates) and closely mirrors some recent interesting proposals to closed world reasoning [7, 8], where the effects of closing the world can be restricted by specifying the predicates which may be affected by the CWA. This schema based on CWA, suggests a simple way for the representation of a logic program D by an $\mathcal{OL}$ program. In fact, D can be represented as a two-level $\mathcal{OL}$ program (called the $\mathcal{OL}$ *version* of D) where the upper component is CWA (consisting of *all* negative ground literals) and the lower one coincides with D. For an instance, consider the logic program D whose rules are $off \leftarrow \neg on$ and $on \leftarrow \neg off$. This program can be represented by an $\mathcal{OL}$ program consisting of the two components CWA and *program*, such that *program* is "lower" than CWA. The component *program* consists of the rules of D, while CWA consists of the (negative) literals $\neg on$ and $\neg off$. Intuitively, CWA expresses the closed world assumption on the world $\{on, off\}$. This $\mathcal{OL}$ program has two stable models, namely, $\{on, \neg off\}$ and $\{\neg on, off\}$, which coincide with the stable models of D. In this paper we show that, in general, given a logic program D and its $\mathcal{OL}$ version, say $\mathcal{P}_{\mathcal{D}}$, the stable models for D coincide with the $\mathcal{OL}$ stable models of $\mathcal{P}_{\mathcal{D}}$. Hence, $SMS^{\mathcal{OL}}$ can be seen as a generalization of logic programming stable model semantics.

The second formalism compared to $\mathcal{OL}$ in the paper is *Extended logic programming* [6, 9] – an extension of logic programming where two negation symbols may occur: *not* for negation as failure and $\neg$ for classical negation. Since classical negation is already featured by $\mathcal{OL}$, and negation as failure can be mimicked as outlined

[1]In this paper we consider function-free logic programs

above, it will be asily shown that extended logic programs have a direct counterpart within the $\mathcal{OL}$ formalism. Then, we shall discuss the differences between the two languages w.r.t. the suitability of representing commonsense knowledge: a number of examples will show that $\mathcal{OL}$ can better model some form of nonmonotonic reasoning.

2 Ordered Logic Programming

In this section we briefly present the syntax and the stable model semantics of Ordered Logic. We refer the reader to [3] for a thourough description of $\mathcal{OL}$.

A *term* is either a constant or a variable. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* $\neg\ p$, where p is an atom. We use an upper-case letter, say L, to denote either a positive or a negative literal. Two literals are *complementary* if they are of the form p and $\neg p$, for some atom p. Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$. A *rule* r is a statement of the form $H\ \leftarrow B$, where H is a literal (*head* of the rule) and B is a set of literals (*body* of the rule) (note that the head of a rule may be a negative literal). Given a rule r, we shall denote by $H(r)$ and $B(r)$ the head and the body of r, respectively. If $B(r)$ is empty, then r is called a *fact*. A term, an atom, a literal or a rule is *ground* if no variable appears in it.

Let $(C, \leq)$ be a finite partially ordered set of symbols, called *identifiers*.

A *component* is a pair $(c, D(c))$, where $c \in C$ and $D(c)$, the *definition of c*, is a finite set of rules. A *knowledge base* on C is a set of components, one for each element of C. In the following we shall denote by $<$ the reflexive-reduction of $\leq$ (i.e., $a < b$ iff $a \leq b$ and $a \neq b$).

Definition 1 Given a knowledge base $\mathcal{K}$ and an identifier $c \in C$, the *ordered logic program for c* ($\mathcal{OL}$ program, for short) is the set of components $\mathcal{P} = \{(c', D(c')) \in \mathcal{K} \mid c \leq c'\}$. ▮

Let $\mathcal{P}$ be an $\mathcal{OL}$ program. The *Universe* $U_\mathcal{P}$ of $\mathcal{P}$ is the set of all constants appearing in the rules in the components of $\mathcal{P}$. The *Base* $B_\mathcal{P}$ of $\mathcal{P}$ is the set of all possible ground literals constructible from the predicates appearing in the rules of $\mathcal{P}$ and the constants occurring in $U_\mathcal{P}$ (clearly, both $U_\mathcal{P}$ and $B_\mathcal{P}$ are finite). Notice that, unlike traditional logic programming, the Base of an ordered logic program contains also negative literals (this is because both negative and positive literals are treated in a uniform way in ordered logic).

Given a rule r occurring in a component of $\mathcal{P}$, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a mapping from the variables occurring in r to the constants in $U_\mathcal{P}$. We denote by *ground*($\mathcal{P}$) the (finite) multiset of all possible ground instances of the rules from the components of $\mathcal{P}$. The reason why *ground*($\mathcal{P}$) is a multiset is that a rule may appear in several different components of $\mathcal{P}$, and we require the respective ground instances be

distinct. Hence, we can define a function *compOf* from *ground*($\mathcal{P}$) onto the set C of the identifiers, associating with a ground instance $\overline{r}$ of r the (unique) component of r.

Given $I \subseteq B_\mathcal{P}$, a ground literal L is *true* (resp., *false*) w.r.t. I if $L \in I$ (resp., $L \in \neg.I$); if L is neither true nor false w.r.t. I, then it is *undefined* w.r.t. I. By I, $\neg.I$ and $\overline{I}$ we denote the sets of true, false and undefined literals w.r.t. I, respectively (note that $\overline{I} = B_\mathcal{P} - (I \cup \neg.I)$).

A set X of ground literals is *true* (resp., *false*) w.r.t. I if, $\forall L \in X$, L is true w.r.t. I (resp., $\exists L \in X$ such that L is false w.r.t. I). A rule $r \in$ *ground*($\mathcal{P}$) is *satisfied* in I if either the head of r is true w.r.t. I or the body of r not true w.r.t. I. Note that the given definition of rule satisfaction coincides with the classical one on total interpretations (as, if L is not true, then it is false).

Given an $\mathcal{OL}$ program $\mathcal{P}$, an *interpretation* for $\mathcal{P}$ is $I \subseteq B_\mathcal{P}$ such that $I \cap \neg.I\ =\ \emptyset$ (i.e., it is a *consistent* set of literals, in the sense that no two complementary literals belong to I). An interpretation I is *total* if $I \cup \neg.I = B_\mathcal{P}$ (i.e. each literal is either true of false w.r.t. I); otherwise I is *partial*. Note that, given a ground literal Q and $I \subseteq B_\mathcal{P}$, then Q can be at the same time true and false w.r.t. I; but, if I is an interpretation, then Q cannot be at the same time true and false w.r.t. I.

Next we introduce the concept of *model* for an ordered logic program. Unlike traditional logic programming, the notion of satisfiability of a rule is not sufficient at this aim, as it does not take into account the presence of explicit contradictions. Hence, we first present some preliminary definitions.

Definition 2 Let an $\mathcal{OL}$ program $\mathcal{P}$ and a set $I \subseteq B_\mathcal{P}$ be given. A rule $r \in$ *ground*($\mathcal{P}$) is *overruled* in I if there exists a rule $r' \in$ *ground*($\mathcal{P}$) such that all the following conditions hold: 1) $compOf(r') < compOf(r)$; 2) $H(r) = \neg.H(r')$, i.e., the heads of r and r' are complementary literals; 3) $B(r') \subseteq I$. ▮

Definition 3 Let an $\mathcal{OL}$ program $\mathcal{P}$ and a set $I \subseteq B_\mathcal{P}$ be given. We say that a rule $r \in$ *ground*($\mathcal{P}$) is *defeated* in I if there exists a rule r' in *ground*($\mathcal{P}$) such that all the following conditions hold: 1) neither $compOf(r) < compOf(r')$ nor $compOf(r') < compOf(r)$, 2) $H(r') = \neg.H(r)$, i.e, the heads of r and r' are complementary literals, 3) $B(r') \subseteq I$ and $H(r') \in I$. ▮

Definition 4 Let I be an interpretation for an $\mathcal{OL}$ program $\mathcal{P}$. A rule $r \in$ *ground*($\mathcal{P}$) is *effective* w.r.t. I if it is neither overruled nor defeated in I and, further, $B(r) \subseteq I$. ▮

Definition 5 Let I be an interpretation for the $\mathcal{OL}$ program $\mathcal{P}$. We say that I is a *model* for $\mathcal{P}$ if every rule in *ground*($\mathcal{P}$) which is effective w.r.t. I is satisfied in I. ▮

Observe that a model is not necessarily a total interpretation, thus, in general, a model leaves undefined a number of literals.

Now we describe the Stable Model Semantics for $\mathcal{OL}$ programs ($SMS^{\mathcal{OL}}$, for short) which is a natural generalization of the (2-valued) stable model semantics given for traditional logic programs [5].

Before giving new concepts and definitions, we introduce the *immediate consequence operator* $T_\mathcal{P}$ for an $\mathcal{OL}$ program $\mathcal{P}$ defined as follows:

$$T_\mathcal{P} : 2^{B_\mathcal{P}} \longrightarrow 2^{B_\mathcal{P}}$$
$$T_\mathcal{P}(I) = \{\ L \in 2^{B_\mathcal{P}} \mid \exists r \in ground(\mathcal{P})\ such\ that\ H(r) = L \wedge B(r) \subseteq I\}.$$

It is easy to see that $T_\mathcal{P}$ is a monotonic transformation in the complete lattice $(2^{B_\mathcal{P}}, \subseteq)$ and, therefore, it admits a least fixpoint. Consider now the sequence $\{T^n\}$, where T^n is inductively defined as follows: $T^0 = \emptyset$, $T^n = T_\mathcal{P}(T^{n-1})$. Clearly, $\{T^n\}$ is monotonically increasing. Since the Base of $\mathcal{P}$ is finite, $\{T^n\}$ is finitely bound, so that there exists a natural i such that, for each $j \geq i$, $T^j = T^i$. Clearly, T^i coincides with the least fixpoint of $T_\mathcal{P}$, which we denote by $T_\mathcal{P}^\infty$. It is worth noting that $T_\mathcal{P}(I)$ contains a (possibly negative) literal L if and only if it is the head of a rule such that every literal in its body, even if negative, is in I. That is, negation is considered as true negation.

Definition 6 Let the $\mathcal{OL}$ program $\mathcal{P}$ and the interpretation I be given. The *reduction of* $\mathcal{P}$ *w.r.t.* I is $P^I = \{r \in ground(\mathcal{P}) \mid r$ is effective w.r.t. $I\}$ ▮

Example 1 For the $\mathcal{OL}$ program $\mathcal{P}_{nixon}$ and the set $I = \{pacifist, goodpresident\}$, $\mathcal{P}^I_{nixon}$ is the set consisting of the following rules: $pacifist \leftarrow$ and $goodpresident \leftarrow pacifist$.
▮

In analogy with traditional logic programming, we give the following definition of stable model.

Definition 7 Let M be a model for an $\mathcal{OL}$ program $\mathcal{P}$. M is an *($\mathcal{OL}$) stable model* for $\mathcal{P}$ if $M = T^\infty_{\mathcal{P}^M}(\emptyset)$.
▮

3 Relation to Logic Programming

In this section we show that classical logic programming (without function symbols) under total-stable semantics [5] (simply t-stable, hereafter) can be considered as a fragment of ordered logic programming.

To prove the claim, we represent a logic program D (without function symbols) as a two level ordered logic program $\mathcal{P}_\mathcal{D}$, called the *ordered version of* D. $\mathcal{P}_\mathcal{D}$ consists of two objects, namely *program* and CWA, related by the relationship *program* $< CWA$. The definition of *program* is made of the rules in D, while the definition of CWA contains a rule of the form $\neg q(X_1, ..., X_n)$ for each predicate q appearing in D, where n is the arity of q and $X_1, ..., X_n$ are distinct variables (the above rule is called the *closed world assumption* for q). The intuition behind such a representation is the following: the higher object CWA corresponds to an explicit closed world declaration establishing that a fact is false unless explicitly contradicted.

We next show that the notion of stable model semantics for ordered logic programs is a generalization of the stable model semantics of logic programs. To this end, we first recall some basic notions.

Let $\mathcal{D}$ be a logic program and $B_\mathcal{D}$ its Herbrand base [15]. The immediate consequence operator $\overline{T}_{\mathcal{D}_M} : 2^{B_\mathcal{D}} \longrightarrow 2^{B_\mathcal{D}}$ is defined as: $\overline{T}_{\mathcal{D}_M}(I) = \{\ p \mid \exists r \in ground(\mathcal{D})\ s.t.\ H(r) = p \wedge B^+(r) \subseteq I\}$, where $B^+(r)$ is the set of positive literals occurring in the body of the rule r (likewise, we shall denote by $B^-(r)$ the set of negative literals in$B(r)$). Now, given a total interpretation $M \subseteq B_\mathcal{D} \cup \neg.B_\mathcal{D}$, (i.e., an interpretation such that for each literal $L \in B_\mathcal{D}$ either one of L and $\neg.L$ belong to M), the *reduction of* $\mathcal{D}$ *w.r.t* M is the (ground) logic program $\mathcal{D}_M$ obtained from $\mathcal{D}$ as follows: for each $L \in M$, if there is a ground rule $r \in ground(\mathcal{D})$ such that $\neg.L \in B(r)$, then discard r from $ground(\mathcal{D})$. Consider the sequence $\{\overline{T}^n\}$, where $\overline{T}^n$ is inductively defined as follows: $\overline{T}^0 = \emptyset$, $\overline{T}^n = \overline{T}_{\mathcal{D}_M}(\overline{T}^{n-1})$. M is a *t-stable* model for $\mathcal{D}$ if $\overline{T}^\infty_{\mathcal{D}_M}(\emptyset) = M^+$, where M^+ is the subset of M consisting of all positive literals of M (likewise, we denote by M^- the set of all negative literals in M, i.e., $M^- = \neg.(B_\mathcal{D} - M^+)$). Note that each rule r in $\mathcal{D}_M$ is such that $B(r) \subseteq M$.

Consider now the ordered version $\mathcal{P}_\mathcal{D}$ of D. Clearly, the base $B_{\mathcal{P}_\mathcal{D}}$ coincides $B_\mathcal{D} \cup \neg.B_\mathcal{D}$, so that M is an interpretation also for $\mathcal{P}_\mathcal{D}$ (as M is a consistent subset of $B_{\mathcal{P}_\mathcal{D}}$). By definition 6, the reduction P^M_D of $\mathcal{P}_\mathcal{D}$ w.r.t. M is obtained from $ground(\mathcal{P}_\mathcal{D})$ by selecting all rules that are effective w.r.t. M. In particular, P^M_D consists of (1) the set X of all ground rules of the form $\neg p(t_1, ..., t_n)$, coming from the component CWA of $\mathcal{P}_\mathcal{D}$, that are not overruled in M, and (2) the set Y of all ground instantiations, with a true body in M, of the rules in the component *program* of $\mathcal{P}_\mathcal{D}$. It is immediate to realize that Y coincides with $\mathcal{D}_M$. Further, we point out that no conflicting rules, i.e., rules with complementary head, are in P^M_D. Indeed, if a rule with positive head, say p, is in P^M_D, its body is true (w.r.t. M) and, hence, $\neg p \leftarrow$ is not in P^M_D, as it is overruled.

Lemma 1 *Let M be either a t-stable model for D or a stable model for $\mathcal{P}_\mathcal{D}$. Then, M^- coincides with the set $X = \{\ \neg p \mid \neg p \leftarrow \in P^M_D\}$.*

Proof. If M is a stable model for $\mathcal{P}_\mathcal{D}$, the statement is clearly true. If M is a t-stable model for D, $\neg p \in M^-$ iff there not exists a rule $r \in \mathcal{D}_M$ (with true body) iff $r \notin P^M_D$ iff $\neg p \leftarrow$ belongs to P^M_D iff $\neg p \in X$. ▮

Proposition 1 *Let $\mathcal{D}$ be a logic program and $\mathcal{P}_\mathcal{D}$ be the ordered logic version of $\mathcal{D}$. Then M is a t-stable model for $\mathcal{D}$ if and only if M is a stable model for the $\mathcal{OL}$ program $\mathcal{P}_\mathcal{D}$.*

Proof. Let M be either a t-stable for D or a stable for $\mathcal{P}_\mathcal{D}$. We define the sequences $\{\overline{T}^n\}$ and $\{T^n\}$ inductively as follows: $\overline{T}^0 = \overline{T}_{\mathcal{D}_M}(\emptyset)$, $\overline{T}^n = \overline{T}_{\mathcal{D}_M}(\overline{T}^{n-1})$ and $T^0 =$

$T_{P_D^M}(\emptyset)$, $T^n = T_{P_D^M}(T^{n-1})$, respectively. We note that the negative part of T^i, for each $i \geq 0$, is $\{\neg p | \exists \neg p \leftarrow \in P_D^M\}$, which coincides with M^- by fact 1.
Claim1. Let $\overline{T}^k = \overline{T}^{k+1}$, for some natural k. Then, $T^i \subseteq \overline{T}^k \cup M^-$, for each natural i.
Proof. We proceed by induction on the number i of applications of the operator $T_{P_D^M}$. *Basis (i=0)* Let X be the set of the positive literals in T^0, i.e., $X = \{p | \exists r \in P_D^M s.t. H(r) = p \wedge B(r) = \emptyset\}$. Since the rules with positive head are the same in P_D^M and $\mathcal{D}_M$, $X \subseteq \overline{T}^k$ holds. The set of negative literals in T^0 coincides with M^-. Thus, $T^0 \subseteq \overline{T}^k \cup M^-$ follows. *Induction* $(i > 0)$. Let p be a positive literal in T^{i+1}. Then $\exists r \in P_D^M$ such that $H(r) = p$ and $B(r) \subseteq T^i$. Hence, by the inductive hypothesis, $B(r) \subseteq (\overline{T}^k \cup M^-)$ only if $B^+(r) \subseteq \overline{T}^k$. Further, since $r \in P_D^M$ implies $r \in \mathcal{D}_M$, $p \in \overline{T}^k$ follows (as $\overline{T}^k$ is a fixpoint). To conclude the proof, we just need to observe that the negative part of T^{i+1} coincides with M^-.
Claim 2. $T^j \subset \overline{T}^j \cup M^-$ implies T^j is not a fixpoint of $T_{P_D^M}$.
Proof. Let i be the largest natural such that $\overline{T}^i \cup M^- \subseteq T^j$ (note that such a natural there exists, as $M^- \subseteq T^i$, for each $i \geq 0$). Clearly, $\overline{T}^i$ is not a fixpoint of $\overline{T}_{\mathcal{D}_M}$ (as $\overline{T}^i \subset \overline{T}^j$ and $\{\overline{T}^n\}$ is monotonic). If $\overline{T}^i \cup M^- = T^j$ holds, then clearly T^j is not a fixpoint of $T_{P_D^M}$. Otherwise, i.e., $\overline{T}^i \cup M^- \subset T^j$, there exists $r \in \mathcal{D}_M$ such that $B(r) \subseteq \overline{T}^i$ and $H(r)$ is not in T^j (if such a rule would not exist then $T^j = \overline{T}^{i+1} \cup M^-$ would hold, contradicting the assumption that i is the largest natural such that $\overline{T}^i \cup M^- \subseteq T^j$). But, $B(r) \subseteq \overline{T}^i$ implies $B(r) \subseteq T^j$ (as the negative part of T^j is M^-) and $r \in \mathcal{D}_M$ implies $r \in RedPD$. Thus, T^j is not a fixpoint of $T_{P_D^M}$.
By claims 1 and 2 above we have that if T^K is a fixpoint (in particular, the least fixpoint) of $T_{P_D^M}$, and $\overline{T}^k$ is a fixpoint of $\overline{T}_{\mathcal{D}_M}$, then $T^k = \overline{T}^k \cup M^-$ holds. Let $\overline{T}^\infty_{P_D^M}(\emptyset)$ and $T^\infty_{PM}(\emptyset)$ be the least fixpoints of T_{PM} and $\overline{T}_{\mathcal{D}_M}$, respectively. Clearly, $\exists k$ such that $\overline{T}^k = \overline{T}^\infty_{\mathcal{D}_M}(\emptyset)$ and $T^k = T^\infty_{PM}(\emptyset)$. Thus, we have that $\overline{T}^\infty_{P_D^M}(\emptyset) = \overline{T}^\infty_{\mathcal{D}_M}(\emptyset) \cup M^-$ holds.
We are finally ready to conclude the proof. Let M be a t-stable for D; then $M = \overline{T}^\infty_{\mathcal{D}_M}(\emptyset) \cup M^- = \overline{T}^\infty_{P_D^M}(\emptyset)$ and, hence, M is a stable model for $\mathcal{P}_\mathcal{D}$. Conversely, let M be a stable model for $\mathcal{P}_\mathcal{D}$; then $M = \overline{T}^\infty_{P_D^M}(\emptyset) = \overline{T}^\infty_{\mathcal{D}_M}(\emptyset) \cup M^-$, from which $M^+ = \overline{T}^\infty_{\mathcal{D}_M}(\emptyset)$. Hence, M is a t-stable for D. ▮

4 Relation to Extended Logic Programs

In [6], Gelfond and Lifschitz propose to include classical negation in logic programs in order to deal also with incomplete information. They define *extended logic programs* as logic programs where two negation symbols may occur: *not* for negation as failure and $\neg$ for classical negation. The semantics of an extended program is defined by a number of *answer sets*, which generalize the notion of stable model to this framework.

It can be easily recognized that extended logic programs have a direct counterpart within the $\mathcal{OL}$ formalism. Indeed, classical negation is directly featured by $\mathcal{OL}$, and negation as failure can be mimicked as shown in the preceding section. This correspondence can be formally proven: In [6], it is proven that every extended logic program can be equivalently represented by a normal logic program under stable model semantics; it turns out that, by virtue of theorem 1, every extended logic program can be represented in $\mathcal{OL}$. Thus, we do not need to exhibit a direct translation from extended logic programs into $\mathcal{OL}$ programs. Rather, we like to discuss the differences between the two languages w.r.t. the suitability of representing commonsense knowledge.

Extended logic programming is an useful extension of traditional logic programming, since several facts of commonsense reasoning can be represented by logic programs more easily when classical negation is available (cf. [6, 1]). However, a limit of this extension is that it ducks the issue of resolving conflicts - these often occur when negated heads are allowed in logic programs. Indeed, in commonsense reasoning it is often the case that support for a can be eliminated by (more reliable) support for $\neg a$, thus resolving the conflict.

Ordered logic overcomes this drawback, as the ordering between the components allows one to specify the more reliable rules in favour of which possible conflicts are to be resolved.

We next show a couple of examples where the explicit resolution of conflicts do allow $\mathcal{OL}$ to represent real world situations more naturally than extended logic programming.

Consider the following (part of the) classification of animals: birds and mammals are animals, penguins and albatross are birds, bats and elephants are mammals. The commonsense knowledge about the "can fly" property on the above classes is: (a) most classes of animals do not fly, (b) birds usually fly, (c) mammals do not fly in general, (d) even if penguin is a bird, penguin can not fly, (e) even if bat is a mammal, bat can fly.

The above knowledge can be easily modelled in $\mathcal{OL}$ by the $\mathcal{OL}$ knowledge base $\mathcal{K}_{animal}$ illustrated in figure 2. Intuitively, the knowledge base is built as follows. To each class of animals is associated a component in $\mathcal{K}_{animal}$. Following the classical scheme of default reasoning, we have specified the "can fly" property on the higher classes of the hierarchy; then, this property has been re-stated only on the subclasses which do not agree with the general rule.

It is easy to see that $\mathcal{K}_{animal}$ correctly models the above described commonense knowledge. Indeed, the $\mathcal{OL}$ program of each component (i.e. animal) in the hierarchy has a unique stable model where the "can fly" property is defined to be either true or false, according to the real situation. For instance, the program $\mathcal{P}_{penguin}$ for *penguin* contains the following rules: $s_1 : \neg flies \leftarrow$ (inherited from the component *Animal*), $s_2 : flies \leftarrow$ (inherited from *Birds*), $s_3 : \neg flies \leftarrow$ (defined in its own component *penguin*). $\mathcal{P}_{penguin}$ has a unique stable model $M_{penguin} = \{\neg flies\}$ (note that the more specific rule s_3 overrules s_2). Similarly, program $\mathcal{P}_{bat} = \{\neg flies \leftarrow$

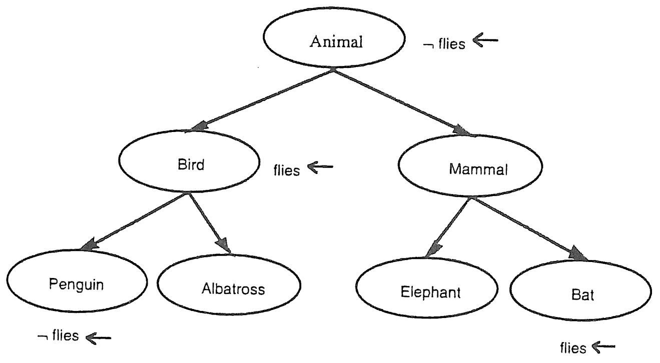


Figure 2: The $\mathcal{OL}$ Knowledge Base $\mathcal{K}_{animal}$

$$\begin{array}{ll} animal(X) \leftarrow bird(X) & r_1: \neg flies(X) \leftarrow animal(X),\ not\ ab(r_1, X) \\ animal(X) \leftarrow mammal(X) & r_2: flies(X) \leftarrow bird(X),\ not\ ab(r_2, X) \\ bird(X) \leftarrow penguin(X) & \quad flies(X) \leftarrow bat(X) \\ bird(X) \leftarrow albatross(X) & \quad ab(r_1, X) \leftarrow bird(X) \\ mammal(X) \leftarrow elephant(X) & \quad ab(r_1, X) \leftarrow bat(X) \\ mammal(X) \leftarrow bat(X). & \quad ab(r_2, X) \leftarrow penguin(X) \\ & \\ (a): ISA - hierarchy & (b): the\ "can\ fly"\ rules \end{array}$$

Figure 3: the extended logic program for the animal classification

, $flies \leftarrow\}$ has a unique stable model $M_{bat} = \{flies\}$, as the rule $flies \leftarrow$ defined on *bat* overrules the default rule $\neg flies \leftarrow$ inherited from *Animal*. The unique stable models for *elephant* and *albatross* are $M_{albatross} = \{flies\}$ and $M_{elephant} = \{\neg flies\}$, respectively. Thus, $\mathcal{K}_{animal}$ correctly models the "can fly" property of the animal hierarchy in a simple and natural fashion.

To represent this situation by extended logic programs we have to model the IS-A hierarchy first. To this end, we specify the rules showed in Figure 3.(a). Second, we have now to define the animals able to fly. To this end, we can not simply "reason by default" as in $\mathcal{OL}$, since we can not specify the different levels of reliability of the rules (which would allow to solve conflicts). Thus, we have to include some additional predicates, notably *ab*, which allow to avoid the derivation of conflicting literals. We obtain the rules in Figure 3.(b).

It is easily seen that the extended logic program in Figure 3 correctly models the real situation. Nevertheless, from the above example it should be evident that, thanks to the possibility to assign different reliability levels to the rules, $\mathcal{OL}$ is able represent the same situation in a more compact and natural way (the semantics of the $\mathcal{K}_{animal}$ knowledge base is immediately understood).

$$\begin{array}{ll} goodpresident \leftarrow pacifist & goodpresident \leftarrow pacifist \\ pacifist \leftarrow hawk & pacifist \leftarrow hawk,\ not\ \neg pacifist \\ \neg pacifist \leftarrow republican & \neg pacifist \leftarrow republican,\ not\ pacifist \\ hawk \leftarrow nixon & hawk \leftarrow nixon \\ republican \leftarrow nixon & republican \leftarrow nixon \\ nixon \leftarrow & nixon \leftarrow \\ (a) & (b) \end{array}$$

Figure 4: the extended logic program for "nixon diamond"

In the above case, there is a clear evidence of how conflicts must be resolved, and a unique reasonable conclusion can be entailed. However, several real world situations are intrinsically ambiguous, in these cases both ways of resolving the conflict (i.e., preferring either an atom or its negation) can be reasonably assumed as possible conclusions. Such situations are easily dealt with in $\mathcal{OL}$ by assigning uncomparable reliability levels to conflicting rules. Thus the mechanism of defeating resolves the conflict in favour of either one of the conflicting rules, yielding two stable models which can be chosen as alternative meaning to be assigned to the program. An example of this is the "nixon diamond" described in Figure 1.(b).

The natural encoding of the "nixon diamond" in extended logic programming would be that specified in Figure 4.(a), where the former rules define the "pacifist" and "goodpresident" properties, and the latter ones specify the IS-A hierarchy. However, the extended program in Figure 4.(a) does not capture the meaning of the "nixon diamond", as it has a unique contradicting answer set. To correctly model the "nixon diamond", we have to introduce some additional predicates in the bodies of the former rules, which block the derivation of *pacifist* and $\neg$*pacifist* if their complement is assumed.(see Figure 4.(b)) The semantics of the program in Figure 4 coincides with the semantics of the $\mathcal{OL}$ program $\mathcal{P}_{nix}$ (the program has two answer sets: $\{pacifist, goodpresident\}$ and $\{\neg pacifist\}$) and correctly represents the "nixon diamond". However, also in this case the meaning of the encoding of the knowledge offered by $\mathcal{OL}$ is more easily understood.

Before concluding, we briefly discuss an alternative semantics proposed for extended logic programs by Kowalsky and Sadri in [9]. According with this semantics, conflicts are always solved in favour of negative literals which are considered exceptions. This approach allows some form of default reasoning and overcomes some drawbacks of extended logic programming highlighted above. Comparing this work with ordered logic, we note that $\mathcal{OL}$ permits to easily represent multi-level exceptions (i.e., exceptions to exceptions, etc.) while [9] can explicitly deal only with one level of exceptions (the representation of further levels of exceptions requires to introduce additional predicates as for [6]).

Acknowledgement

Work partially supported by the EC-US project "DEUS EX MACHINA" and by a MURST grant (40% share) under the project "Sistemi formali e strumenti per basi di dati evolute". The author is grateful to Nicola Leone for his suggestions.

References

[1] Baral, C., Gelfond, M. Logic Programming and Knowledge Representation *Journal of Logic Programming*, Vol. 19/20, May/July 1994, pp. 73–148.
[2] Brewka, G., Nonmonotonic Reasoning: Logical Foundations of Commonsense, Cambridge Univ. Press, 1991.
[3] Buccafurri, F., Leone, N., Rullo, P., Stable Models and their Computation for Logic Programming with Inheritance and True Negation, *Journal of Logic Programming*, to appear.
[4] Chellas, B. F., Modal Logic, Cambridge University Press, 1980.
[5] Gelfond, M., Lifschitz, V., The Stable Model Semantics for Logic Programming, *Proc. 5th Int. Conf. on Logic Progr.*, (MIT Press, Cambridge, Ma, 1988), pp. 1070-80.
[6] Gelfond, M., Lifschitz, V., Logic Programs with Classical Negation, *Proc.of 7th ICLP*, Jerusalem, 1990, pp 579-597.
[7] Gelfond, M., Przymusinska, H., Negation as Failure: Careful Closure Procedure, *AI Journ.*, 30, 1986, pp. 273-287.
[8] Gelfond, M., Przymusinska, H., Przymusinski, T.C., On the Relation between Circumscription and Negation as Failure, *AI Journ.*, 38, 1989, pp. 75-94.
[9] Kowlaski, R.A., Sadri, F., Logic Programs with Exceptions, *Proc. of 7th ICLP*, Jerusalem, 1990, pp. 598-616.
[10] Laenens, E., Saccá, D., Vermeir, D., Extending Logic Programming, *Proc. of ACM SIGMOD*, May 1990.
[11] Laenens, E., Vermeir, D., A Fixpoint Semantics for Ordered Logic, *Jour. of Logic and Comp.*, vol.1, n.2, December, 1990, pp. 159-185.
[12] Leone, N., Rossi, G., Well-founded Semantics and Stratification for Ordered Logic Programs, *New Generation Comp.*, Vol. 12, 1, Springer-Verlag, Nov. 1993, pp. 91-121.
[13] Leone, N., Rullo, P., Ordered Logic Programming with Sets, *Journal of Logic and Computation*, Vol. 3, N. 6, Oxford University Press, December 1993, pp. 621-642.
[14] Lifschitz, V., On the Declarative Semantics of Logic Programs with Negation, In *Foundation of Deductive Database and Logic Programming* Minker, J. (ed.), Morgan Kaufman, Los Altos, 1987, pp. 89-148.
[15] Lloyd, J. W., *Foundations of logic programming*, Springer-Verlag, 1987.
[16] Marek, W., Truszczyński, M., Autoepistemic Logic, *Jour. of the ACM*, 38, 3, 1991, pp. 518-619.
[17] McCarthy, J., Circumscription - a Form of Nonmonotonic Reasoning, *AI*, 13, pp. 27-39, 1980.
[18] McDermott, D., Doyle, J., Non-Monotonic Logic I, *AI*, 13, pp. 41-72, 1980.
[19] Moore, R.C., Semantics Considerations on Non-Monotonic Logic, *AI*, 25, 1985.
[20] Poole, D., A Logical Framework for Default Reasoning, *AI*, Vol. 36, pp.27-47, 1988.
[21] Reiter, R., A Logic for Default Reasoning, *AI*, vol.13, pp. 81-132, 1980.
[22] Touretzky, D.S., *The Mathematics of Inheritance Systems*, Pitman London, 1986.

Analysis of SLDNF for Local CLP

Alberto Bottoni
Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova, Italia
`Bottoni@zenone.math.unipd.it` *Fax:* +39 49 8758596

Abstract

Local CLP is a syntactic variation of pure CLP characterized by the explicit quantification of the local variables of the clauses. It allows the definition of the SLD resolution steps as simple *local* rewritings of the atoms in the goals. We outline a formal framework for the definition and the analysis of the SLDNF resolution for local CLP. A subsidiary SLDNF tree T is represented within the goals by the expression $\neg T$. Some natural operations on derivations trees (e.g. and-compositions) are naturally defined. As an application of the framework we provide the bottom-up fix-point construction of a correct and fully abstract denotations of the parallel trees.

Keywords: *SLDNF-resolution, formal semantics, and-compositionality*

1 Introduction

A 'pure', or 'most general' atom has the form $p(\overline{x})$, where p is a predicate symbol of some arity n, and $\overline{x}$ is a n-tuple of distinct variables. Pure CLP is an equivalent form of standard CLP where only pure atoms are allowed to appear in programs and goals. 'Local' CLP is a syntactic variation of pure CLP. It is characterized by the *explicit* quantification of the local variables of the clauses, and therefore of the variables introduced by the derivation steps in the goal. Explicit existential quantification permits to model the SLD resolution as a *local* step, concerning the selected atom only, and independent from the rest of the goal. The description of resolution as a simple local rewriting may help the definition of concepts and the proof of properties in a 'modular' fashion, and by induction on the syntax. However, local CLP can be seen as a formal tool to talk of pure (or even standard) CLP, since the only difference is that existential quantifiers are used to keep an explicit note, within the goal, of the local variables.

In this paper, we use local CLP to outline a new framework for the definition and the analysis of the SLDNF resolution. The aim is to provide a formally clean and convenient base for the study of semantics issues and of run-time properties of SLDNF.

Due to its involved recursive nature, the definition of SLDNF-resolution is not plain [7, 8, 1]. The resolution of a negated atom within a goal must be defined in terms of a (subsidiary) derivation tree which, in general, is constructed through the resolutions of other negative literals. In [1] the selected negative literal points (through a function *'subs'*) at a subsidiary SLDNF tree. The main goal is then blocked until failure or success is reached by the pointed tree. In [8] the relevant part of the subsidiary tree (its frontier) is kept *within* the main goal. Differently from [1], the selection rule can suspend the 'exploration' of the subsidiary tree and continue with other parts of the goal. This makes possible the modeling of deterministic fair rules.

The approach we follow is to define the SLDNF-resolution by extending, as in [8], the syntax of the goals so as to include a representation of the subsidiary tree. However we find that a more immediate representation of a subsidiary tree is the tree itself. Therefore the expression of a negative subgoal pointing to a subsidiary tree T is simply $\neg T$. Both the explicit exhibition of the recursive dependency between goals and SLDNF trees, and the use of local CLP, contribute to the clearness of the construction. Building over the definitions, some natural operations on derivations trees (e.g. and-compositions) can be neatly defined, and their properties easily exploited. As an example of application of the definitions and of the operations, we sketch the bottom-up fix-point construction of a correct and fully abstract denotation of the parallel trees, a class of derivation trees in which all the computed answers and finite failures of SLDNF are represented.

The paper is organized as follows. The syntax of local CLP goals is presented in subsection 2.1. Subsection 2.2 completes the mutual recursive definitions of goals and derivation trees. The parallel and sequential trees are defined in subsection 2.3 and the equivalence between pure and local CLP is discussed in subsection 2.4. The operation on trees are defined in section 3 and subsection 3.1 rephrases, for local CLP, the usual notions of success, failure and floundering. A correct and fully abstract denotation of the parallel trees is defined in section 4. After the introduction of some peculiarities of parallel trees in subsection 4.1, the denotation is presented in subsection 4.2. Its bottom-up, fix-point construction is sketched in subsection 4.3. Due to the lack of space all the proofs are omitted; they can be found in [3].

1.1 Notations

We use overlining to denote (possibly empty) tuples of homogeneous objects. $\overline{x}$ denotes a tuple of *distinct* variables. The free variables of formula S are denoted by $\mathtt{FV}(S)$. The variables in a formula which are not free are said to be 'local'. Formulae are identified modulo consistent renamings of local variables. We say that two formulae S_1 and S_2 are 'variant', $S_1 \approx S_2$, if S_2 can be obtained from S_1 by a consistent renaming of its (free and local) variables.

$\mathcal{C}$ denotes the set of all the constraints. We assume that $\mathcal{C}$ contains the equations among terms, axiomatized by the Clark's Equality Theory [4], and the logical constants for the truth values $\mathtt{t}$ and $\mathtt{f}$. Moreover $\mathcal{C}$ is closed by conjunction and existential quantification, and the satisfiability of the constraints must be decidable. *sat*(c) denotes that the constraint c is satisfiable. We say that the variables $\overline{y}$ are *grounded* by the constraint c to the ground terms $\overline{g}$ iff every solution of the constraint c binds the variables $\overline{y}$ to (the interpretation of) $\overline{g}$. A formula is grounded by c (to $\overline{g}$) if so are its free variables. $\mathtt{t}(\overline{x}) \stackrel{\text{def}}{=} \overline{x} = \overline{x}$ ($\mathtt{f}(\overline{x}) \stackrel{\text{def}}{=} \mathtt{f}, \mathtt{t}(\overline{x})$) denotes a fixed always true (false) constraint in the variables $\overline{x}$. A most general atom (*mga*) is an atom, of the form $p(\overline{x})$, having a tuple of distinct variables as arguments. $\mathtt{MGA}$ denote the set of all the most general atoms. The comma in a formula means conjunction.

2 SLDNF for Local CLP

The notions of 'l-goal', 'tree' and 'extension of a tree' are defined by mutual recursion. A tree is a (well-formed) finite tree of l-goals. An l-goal is an expression possibly containing a well-formed tree as a subexpression. A well-formed tree is either a single-node tree, labeled by some l-goal, or the extension of a well-formed tree. The base cases in the definition of 'l-goal' are constraints and most general atoms. The base step in the 'extension' of a tree is the replacement of an atom with the bodies of the clauses that define that atom in the program. The other notions are defined by recursion over these base cases. Although presented separately, the definitions of l-goal, tree and extension of a tree are three parts of a single unit. The defined objects are therefore the closure of the all base cases w.r.t. the inductive ones.

2.1 L-goals and L-Programs

The class of the l-goals is defined as the closure, by conjunction and existential quantification, of constraints, *mga*'s and negations $\neg T$ of certain trees of l-goals. The trees of l-goals are defined in subsection 2.2.

DEFINITION 2.1 *An l-goal is a formula that can be generated by the following grammar*

$$G ::= c \mid p(\overline{x}) \mid G, G \mid \exists y G \mid \neg T$$

where c is any constraint, $p(\overline{x})$ is any mga and T is any tree rooted at some mga.

An l-goal $\neg T$ is said to be a '*negative l-goal pointing to T*'. Most general atoms and negative l-goals are called *and-atomic* l-goals. An l-goal is said to be 'definite' if it does not contain any negative subgoal. A negative l-goal $\neg T$ is said to be 'top-level' if T is a single-node tree (labeled by some *mga*). An l-goal is said to be 'top-level' if so are its negative subgoals. The free variables of a negative l-goal $\neg T$ are defined as the free variables of the root of T, and its local variables are the local variables of the l-goals labeling T. The set of the l-goals is denoted by `Goals`. The (existential) quantification of a set $\{G_i\}_{i\in I}$ of l-goals is defined by $\exists y\{G_i\}_{i\in I} \overset{\text{def}}{=} \{\exists y G_i\}_{i\in I}$. The cross-product of two sets of l-goals $\{G_i\}_{i\in I}$ and $\{G_j\}_{j\in J}$ is an associative operation defined by $\{G_i\}_{i\in I} \otimes \{G_j\}_{j\in J} \overset{\text{def}}{=} \{(G_i, G_j)\}_{(i,j)\in I\times J}$. An 'l-clause' has the form $p(\overline{x}) \leftarrow G$ where the head, $p(\overline{x})$, is an *mga*, and the body, G, is a *top-level* l-goal with the same free variables as the head ($\mathtt{FV}(G) = \overline{x}$). An l-program is a finite set of l-clauses, not containing variants. The SLD derivation step for local CLP can be defined, by induction on the syntax, as follows. Fixed an l-program P, let $p(\overline{y})$ be an *mga* and let $\{p(\overline{y}) \leftarrow G_i\}_{i\in I}$ be a set of variants of clauses of P containing one variant for each clause for the predicate p in P, renamed so that their head coincides with $p(\overline{y})$. If this set is not empty then we define

$$p(\overline{y}) \mapsto \{\, G_i \,\}_{i\in I} \tag{1}$$

If there are no clauses for p in the program, then $p(\overline{y}) \mapsto \{\mathtt{f}(\overline{y})\}$. The local-to-context extension of relation $\mapsto$ is straightforward:

DEFINITION 2.2 *Extension of relation $\mapsto$ from and-atomic l-goals to l-goals:*

$$(\mathrm{l})\,\frac{G \mapsto \{G_i\}_{i\in I}}{G, G' \mapsto \{G_i\}_{i\in I} \otimes \{G'\}} \qquad (\mathrm{r})\,\frac{G \mapsto \{G_i\}_{i\in I}}{G', G \mapsto \{G'\} \otimes \{G_i\}_{i\in I}}$$

$$(\exists)\,\frac{G \mapsto \{G_i\}_{i\in I}}{\exists y G \mapsto \exists y\{G_i\}_{i\in I}} \qquad (\mathrm{par})\,\frac{G_1 \mapsto \{G_i\}_{i\in I} \quad G_2 \mapsto \{G_j\}_{j\in J}}{G_1, G_2 \mapsto \{G_i\}_{i\in I} \otimes \{G_j\}_{j\in J}}$$

2.2 Trees

This subsection defines a class of finite trees of nodes labeled with l-goals. The single-node tree rooted at the l-goal G is represented by the expression $\langle G, \emptyset\rangle$. In general, a tree expression has the form $\langle G, \{T_i\}_{i\in I}\rangle$, where G is an l-goal, and $\{T_i\}_{i\in I}$ is a set of tree expressions. The trees, i.e. the well-formed tree expressions, are defined by augmenting the single-node trees via the extension relation '$\rightarrow$'.

DEFINITION 2.3 *Trees, extensions of trees and relation $\mapsto$ for negative subgoals:*

1. $$\frac{G \in \mathtt{Goals}}{\langle G, \emptyset\rangle \in \mathtt{Trees}} \qquad \frac{T \in \mathtt{Trees} \quad T \rightarrow T'}{T' \in \mathtt{Trees}}$$

2. $$\frac{\langle G, \{T_i\}_{i\in I}\rangle \in \mathtt{Trees} \quad \forall i \in I\ \ T_i = T_i' \text{ or } T_i \rightarrow T_i' \quad \exists i \in I\ \ T_i \rightarrow T_i'}{\langle G, \{T_i\}_{i\in I}\rangle \rightarrow \langle G, \{T_i'\}_{i\in I}\rangle}$$

3. $$\frac{G \mapsto \{G_i\}_{i\in I}}{\langle G, \emptyset\rangle \rightarrow \langle G, \{\langle G_i, \emptyset\rangle\}_{i\in I}\rangle}$$

4. $$\frac{T_1 \rightarrow T_2}{\neg T_1 \mapsto \{\neg T_2\}}$$

The set of all the trees, `Trees`, is defined by the rules at point 1. Rule 2 states that the extension of a tree $T = \langle G, \{T_i\}_{i\in I}\rangle$, with $I \neq \emptyset$, consists in the extension of *at least* one of its subtrees T_i. One verse of the dependency between relations $\mapsto$ and $\rightarrow$ is shown in rule 3, where a 'leaf' tree $\langle G, \emptyset\rangle$ is expanded with a set of l-goals $\{G_i\}_{i\in I}$ such that $G \mapsto \{G_i\}_{i\in I}$. The other one is exhibited by inference rule 4, which describes how the evolution of a negative subgoal reflects that of the tree it points to.

The set of the l-goals, the set of the trees and the relations '$\rightarrow$' and '$\mapsto$' are simultaneously defined by equation (1) and by definitions 2.1, 2.2 and 2.3, as the closure[1] of their base cases w.r.t. all the inductive ones.

Roughly speaking, relation $\rightarrow$ describes the 'top-down' growth of a tree by the extension of some of its 'leaves'. To every leaf G we can attach any set of l-goals which is *consequent* (w.r.t. $\mapsto$) to G. However, a tree can also be constructed 'bottom-up' by 'joining-up' an opportune set of trees with some *antecedent* (w.r.t. $\mapsto$) of their roots:

DEFINITION 2.4 *For every set of trees $\{T_j\}_{j\in J}$ we define*

$$\mathcal{U}(\{T_j\}_{j\in J}) \overset{\text{def}}{=} \{\langle G, \{T_j\}_{j\in J}\rangle \mid G \in \mathtt{Goals},\ G \mapsto \{\mathtt{r}(T_j)\}_{j\in J}\}$$

It is easy to prove that the $\mathcal{U}$ operator is a mapping between sets of trees. Moreover, by the analysis of the definitions of $\mapsto$ and $\rightarrow$, it follows that $\mathcal{U}$ is computable. A restriction of this basic operator to trees rooted at most general atoms is applied in subsection 4.3 to the bottom-up construction of the 'parallel' trees.

[1]It can be shown that all these definitions (say, collectively, '$\mathcal{F}$') are monotonic and finitary. Therefore, being $\mathcal{F}$ continuous, the defined objects, that is the closure $\mathcal{F} \uparrow \omega$, coincide with the l.f.p. $\mathcal{F} \uparrow \infty$ of $\mathcal{F}$, that is with the smaller sets satisfying the definitions.

2.3 Sequential and Parallel Trees

Some interesting subclasses of l-goals and trees may be identified by modifying, in a restrictive sense, definitions 2.2 and 2.3. Let $\overset{s}{\mapsto}$ denote the restriction of relation $\mapsto$ obtained by removing inference (par) from def. 2.2, and let $\overset{s}{\rightarrow}$ denote the resulting restriction of $\rightarrow$. The l-goals and trees that can be defined according to this restriction are called 'sequential'. By forbidding the parallel rule, sequential trees model computation rules that, as it is most common, select *just one* and-atomic subgoal of the l-goal to be expanded.

We say that an l-goal G (a tree T) is 'expandable' if $G \mapsto \{G_i\}_{i\in I}$ for some l-goals $\{G_i\}_{i\in I}$ (if $T \rightarrow T'$ for some tree T'). To be expandable is a decidable property. Indeed, G is expandable if it contains some *mga*, or if some negative subgoal points to an expandable tree, and a tree is expandable if so is some of its leaves.

Consider the restriction of relation $\mapsto$ (and therefore of $\rightarrow$) obtained by adding the condition *"and G' is not expandable"* to the premises of inferences (l) and (r) of def. 2.2. This restriction corresponds to the computation rule that always selects every expandable and-atomic subgoal of the l-goal. Consider the further restriction of relation $\rightarrow$ obtained by adding the condition *"and $T_i \rightarrow T_i'$ for every $i \in I$ such that T_i is expandable"* to the premises of inference 2. Under this restriction, $T \rightarrow T'$ if *every* expandable leaf of T is extended by getting T'. The l-goals and trees that can be defined by imposing both the two last conditions are called 'parallel', and the resulting relations are denoted by $\overset{p}{\mapsto}$ and $\overset{p}{\rightarrow}$. Some nice properties of parallel trees are discussed in subsection 4.1.

2.4 Local CLP is Equivalent to Pure CLP

A (bidirectional) formal translation from l-programs and l-goals to pure-CLP normal programs and goals is defined in [3]. Besides mapping negation of trees to negative literals, the translation arranges for explicit vs. implicit existential quantifications over the local variables (possibly applying renamings). Moreover, as it is usual in CLP notation, the constraints are gathered to the left of the goals. In one direction the translation defines the 'pure-CLP forms' of an l-goal. In the other one it states what the 'l-form' of a (pure-CLP) normal goal is. Sequential l-goals and trees are then compared to SLDNF-resolution for pure-CLP as defined by Kunen's inductive definition [6] rewritten to conform with the syntax of pure-CLP. It is proved that sequential trees of top-level l-goals compute the same answers as their pure-CLP forms (modulo the translation). Moreover, a top-level l-goal has a failed sequential tree iff its pure-CLP forms finitely fail. (The notions of computed answer and finite failure for local CLP are defined in subsection 3.1). This proves that sequential l-trees have the soundness and completeness of Kunen's definition of SLDNF for pure-CLP. Again in [3] it is shown that both the sequential and the parallel trees subclasses are fully representative w.r.t. successes and failures. All the successes and failures of SLDNF can then be observed by looking at parallel trees of top-level l-goals only. In section 4 we show how, for any l-program, this class of tree is completely coded by the parallel trees of most general atoms only.

3 Some Operations on Trees

In this section we introduce some general operations on trees and discuss some of their properties. To fix some notation, we begin with the trivial ones. The root of a tree $T = \langle G, \{T_i\}_{i\in I}\rangle$, denoted by $\mathtt{r}(T)$, is the l-goal G. The set of the subtrees of a tree $\langle G, \{T_i\}_{i\in I}\rangle$ is the set of tree $\{T_i,\}_{i\in I}$. We let $\mathcal{T}(G)$ to be a shorthand for $\langle G, \emptyset\rangle$. The depth of a tree is a natural number computable by the function $\mathtt{d} : \mathtt{Trees} \rightarrow \omega$ defined as follows: $\mathtt{d}(\mathcal{T}(G)) \overset{\text{def}}{=} 0$; $\mathtt{d}(\langle G, \{T_i\}_{i\in I}\rangle) \overset{\text{def}}{=} 1 + \max\{\mathtt{d}(T_i)\}_{i\in I}$ if $I \neq \emptyset$. The leaves $\mathcal{L}(T)$ of a tree T are defined by: $\mathcal{L}(\langle G, \{T_i\}_{i\in I}\rangle) \overset{\text{def}}{=} \bigcup_{i\in I} \mathcal{L}(T_i)$ if $I \neq \emptyset$, $\mathcal{L}(\langle G, \emptyset\rangle) \overset{\text{def}}{=} \{G\}$. Since $G \mapsto \{G_i\}_{i\in I}$ implies $\mathtt{FV}(G) = \mathtt{FV}(G_i)$ for every $i \in I$, then all the l-goals labeling a tree have the same free variables.

Next definition introduces a quite general operation on trees, called 'and-composition'. By and-composing T_1 and T_2 we obtain a set of trees having different structures. However, the root of these trees is always the conjunction $\mathtt{r}(T_1), \mathtt{r}(T_2)$, while the leaves are always the cross-product $\mathcal{L}(T_1) \otimes \mathcal{L}(T_2)$.

DEFINITION 3.1 *The set of the* and-compositions *of two trees* $T_1 = \langle G_1, \{T_i\}_{i\in I}\rangle$ *and* $T_2 = \langle G_2, \{T_j\}_{j\in J}\rangle$ *is a set of tree defined as the union*

$$T_1 \circ T_2 \overset{\text{def}}{=} (T_1 \circ_l T_2) \cup (T_1 \circ_r T_2) \cup (T_1 \circ_{par} T_2) \qquad \textit{where, recursively}$$

1. $T_1 \circ_l T_2 \overset{\text{def}}{=} \{\ \langle (G_1, G_2), \{T_i'\}_{i\in I}\rangle \mid T_i' \in T_i \circ T_2 \ \ \forall i \in I \neq \emptyset\ \}$

2. $T_1 \circ_r T_2 \overset{\text{def}}{=} \{\ \langle (G_1, G_2), \{T_j'\}_{j\in J}\rangle \mid T_j' \in T_1 \circ T_j \ \ \forall j \in J \neq \emptyset\ \}$

3. $T_1 \circ_{par} T_2 \overset{\text{def}}{=} \{\ \langle (G_1, G_2), \{T_{ij}\}_{(i,j)\in I\times J}\rangle \mid T_{ij} \in T_i \circ T_j \ \ \forall i \in I \neq \emptyset, \forall j \in J \neq \emptyset\ \}$, *if* $I \neq \emptyset$ *or* $J \neq \emptyset$

4. $T_1 \circ_{par} T_2 \overset{\text{def}}{=} \{\ \langle (G_1, G_2), \emptyset\rangle\ \}$, *if* $I = J = \emptyset$.

EXAMPLE. Let $T_1 \overset{\text{def}}{=} \langle a, \{\mathcal{T}(b), \mathcal{T}(c)\}\rangle$ and $T_2 \overset{\text{def}}{=} \langle d, \{\mathcal{T}(e)\}\rangle$. Then

$T_1 \circ_l T_2 = \{\ \langle ad, \{\langle bd, \{\mathcal{T}(be)\}\rangle, \langle cd, \{\mathcal{T}(ce)\}\rangle\}\rangle\ \}$
$T_1 \circ_r T_2 = \{\ \langle ad, \{\ \langle ae, \{\mathcal{T}(be), \mathcal{T}(ce)\}\rangle\ \}\rangle\ \}$
$T_1 \circ_{par} T_2 = \{\ \langle ad,\ \{\mathcal{T}(be), \mathcal{T}(ce)\}\ \rangle\ \}$

Other useful operations can be derived as special cases of and-compositions. For example, a limit case of and-composition can be obtained by restricting (as much as possible) the and-compositions to their 'parallel subset' given by $\circ_{par}$.

DEFINITION 3.2 *The cross-product '$\otimes$' of two trees is defined by*

- $\langle G_1, \{T_i\}_{i\in I}\rangle \otimes \langle G_2, \{T_j\}_{j\in J}\rangle \overset{\text{def}}{=} \langle (G_1, G_2),\ \{T_i \otimes T_j\}_{(i,j)\in I\times J}\rangle$ *if* $I \neq \emptyset$ *and* $J \neq \emptyset$
- $\langle G_1, \emptyset\rangle \otimes \langle G_2, \{T_j\}_{j\in J}\rangle \overset{\text{def}}{=} \langle (G_1, G_2), \{\langle G_1, \emptyset\rangle \otimes T_j\}_{j\in J}\rangle$
- $\langle G_1, \{T_i\}_{i\in I}\rangle \otimes \langle G_2, \emptyset\rangle \overset{\text{def}}{=} \langle (G_1, G_2), \{T_i \otimes \langle G_2, \emptyset\rangle\}_{i\in I}\rangle$

Since $T_1 \otimes T_2 \in T_1 \circ T_2$, then $\otimes$ is a binary operation on trees. The (left) instantiation $G * T$ of a tree T with an l-goal G is defined by $G * T \overset{\text{def}}{=} \mathcal{T}(G) \otimes T$. The definition of $T * G$ is analogous. The existential quantification of a tree is naturally defined by $\exists y \langle G, \{T_i\}_{i\in I}\rangle \overset{\text{def}}{=} \langle \exists y G, \{\exists y T_i\}_{i\in I}\rangle$.

3.1 Success, Failure and Floundering

In this subsection we redefine for l-goals and trees the usual notions of success, failure, floundering and computed answer.

To determine if an l-goal is successful or failed we must look at the constraints it contains. The 'constraint part' $\mathtt{c}(G)$ of an l-goal G is the constraint obtained by replacing in G every and-atomic subgoal in some free variables $\overline{y}$, with $\mathtt{t}(\overline{y})$. Note that $\mathtt{FV}(G) = \mathtt{FV}(\mathtt{c}(G))$. Let G be an l-goal, let $\{M_i\}_{i\in I}$ denote the set of its subgoals which are *mga* and let $\{\neg T_j\}_{j\in J}$ denote the set of its negative subgoals. Let T be a tree and let $\{G_i\}_{i\in I}$ be a set l-goals. Then

1. G is *successful* if $\mathtt{c}(G)$ is satisfiable, $I = \emptyset$ and, for every $j \in J$, T_j is grounded by $\mathtt{c}(G)$ and $\mathtt{c}(G) * T_j$ is failed
2. G is *failed* if $\mathtt{c}(G)$ is unsatisfiable or, for some $j \in J$, T_j is grounded by $\mathtt{c}(G)$ and $\mathtt{c}(G) * T_j$ is successful
3. G is *floundered* if $\mathtt{c}(G)$ is satisfiable, $I = \emptyset$ and there exists a partition of the (indexes of) the negative subgoals $J = S_{ucc} \cup N_g \cup F_l$ such that: $j \in S_{ucc}$ iff T_j is grounded by $\mathtt{c}(G)$ and $\mathtt{c}(G) * T_j$ is failed, $j \in N_g$ iff T_j is *not* grounded by $\mathtt{c}(G)$, $j \in F_l$ iff T_j is grounded by $\mathtt{c}(G)$ but $\mathtt{c}(G) * T_j$ is *floundered*, and $N_g \cup F_l \neq \emptyset$ (some negative subgoal is not grounded or floundered)
4. G is *open* if it is not failed and, if $I = \emptyset$, then there exists $j \in J$ such that T_j is grounded by $\mathtt{c}(G)$ and $\mathtt{c}(G) * T_j$ is open
5. T is successful, failed, floundered or open if so is $\mathcal{L}(T)$
6. $\{G_i\}_{i\in I}$ is *successful* if it contains some successful l-goal, *failed* if G_i is failed for every $i \in I$, *floundered* if it contains at least one floundered l-goal and every G_i which is not floundered is failed, *open* if it contains some open l-goal

Note that every l-goal has one and only one type. While failure or floundering exclude other possibilities, a tree or a set of l-goals can be both successful and open.

The constraint part of a successful l-goal G is called the answer computed by G. The answers computed by a tree are defined as the answers computed by its successful leaves.

An interesting distinction can be pointed out about the type of floundering. A goal G may be floundered because some of its negative subgoals is not grounded by $\mathtt{c}(G)$ ($N_g \neq \emptyset$). We call this situation 'recoverable' floundering, since the groundness could be provided by some instantiation of the goal (resulting for example by an and-composition of a tree containing G). The type of the l-goal could then change from floundered to open (or successful, or failed). A completely different situation is when G is floundered because it contains some floundered negative subgoal $\neg T$. Then T is grounded by $\mathtt{c}(G)$ but $\mathtt{c}(G) * T$ is floundered ($F_l \neq \emptyset$). In this case, the free variables of T (and of $\mathcal{L}(T)$) are already grounded by $\mathtt{c}(G)$. We call this 'nested' or 'unrecoverable' floundering, since any further instantiation of $\mathtt{c}(G)$ may only turn G into a *failed* goal.

4 A Denotation of the Parallel Trees

We know, by subsection 2.4, that all the successes and failures of SLDNF can be observed by looking at parallel trees of top-level l-goals only. In this section we show how, for any l-program, the parallel trees of top-level l-goals are completely coded by the parallel trees of most general atoms only. Moreover, this compact representation is shown to admit a continuous bottom-up construction. Parallel trees of *mga*'s are therefore a correct and fully abstract denotation [2, 5] of the parallel trees of top-level l-goals.

4.1 Parallel Trees

Parallel trees have some interesting properties. For example, if T is a parallel tree then, modulo the implicit renamings of the local variables, there exists at most *one* parallel tree T' such that $T \xrightarrow{p} T'$. Therefore, relations $\overset{p}{\mapsto}$ and $\xrightarrow{p}$ reduce to *partial functions* for parallel l-goals and trees. Then, for every parallel tree T let $\mathcal{S}(T) \overset{\text{def}}{=} T'$ iff $T \xrightarrow{p} T'$. By iterating $\mathcal{S}$ we define, for every parallel l-goal G and every $n \in \omega$,

$$\mathcal{T}^0(G) \overset{\text{def}}{=} \mathcal{T}(G) \qquad\qquad \mathcal{T}^{n+1}(G) \overset{\text{def}}{=} \begin{cases} \mathcal{S}(\mathcal{T}^n(G)) & \text{if defined} \\ \mathcal{T}^n(G) & \text{otherwise} \end{cases}$$

It can be shown that if T is a parallel tree then $T = \mathcal{T}^d(G)$, where $d = \mathtt{d}(T)$ and $G = \mathtt{r}(T)$. Therefore the set $I_p \overset{\text{def}}{=} \{\mathcal{T}^n(G) \mid G \textit{ is a parallel l-goal}, n \in \omega\}$, is the set of *all* the parallel trees.

A notable property of parallel trees, called 'and-compositionality' is stated by next proposition. According to it, a parallel tree of a conjunctive l-goal G_1, G_2

having depth n can be completely reconstructed by looking just at two parallel trees of G_1 and G_2. These trees are the deepest among the parallel trees of, respectively, G_1 and G_2, having depth $\leq n$.

PROPOSITION 4.1 *For every parallel l-goal G_1 and G_2, and every $n \in \omega$*

$$\mathcal{T}^n(\,(G_1, G_2)\,) \;=\; \mathcal{T}^n(G_1) \otimes \mathcal{T}^n(G_2)$$

Two parallel trees T_1 and T_2 are said to be 'compatible w.r.t. parallel-product' if there exists some $n \in \omega$ such that $T_1 = \mathcal{T}^n(\mathtt{r}(T_1))$ and $T_2 = \mathcal{T}^n(\mathtt{r}(T_2))$. By definition of $\mathcal{T}^n$ this is equivalent to say that T_1 and T_2 have the same depth, or the one with smaller depth is not expandable. Compatibility w.r.t. parallel-product is therefore decidable. A set of parallel trees is compatible w.r.t. parallel-product if so are every two trees in it. $\mathtt{cpp}(\{T_i\}_{i \in I})$ denotes that $\{T_i\}_{i \in I}$ is a set of trees compatible w.r.t. parallel-product.

The 'parallel-product' of trees is defined as the restriction of the cross-product to trees which are compatible w.r.t. parallel-product. The parallel-product is denoted by $\otimes_p$. The cross-product of two parallel trees $T_1 \otimes T_2$ is not guaranteed to be a parallel tree. For example, in general $G_1 * \mathcal{T}^n(G_2) = \mathcal{T}^0(G_1) \otimes \mathcal{T}^n(G_2)$ is different from $\mathcal{T}^n(G_1) \otimes \mathcal{T}^n(G_2) = \mathcal{T}^n(\,(G_1, G_2)\,)$. However, if T_1 and T_2 are compatible w.r.t. $\otimes_p$, by and-compositionality it is $T_1 \otimes T_2 = T_1 \otimes_p T_2 = \mathcal{T}^n(\,(\mathtt{r}(T_1), \mathtt{r}(T_2))\,)$ for some $n \in \omega$, and therefore their product is a parallel tree.

4.2 Parallel Trees of Most General Atoms

Let $\mathcal{T}^n(G)$ be a parallel tree of some parallel l-goal G. From the definition of $\xrightarrow{p}$ we may observe that, for every $m \leq n$, $\mathcal{T}^m(G)$ can be obtained by cutting at depth m all the branches of $\mathcal{T}^n(G)$ longer than m. That is, by looking just at the parallel tree of depth n, we may reconstruct all the parallel trees of G having smaller depth.

Rule 4 of def. 2.3 implies that a tree rooted at a negative l-goal has only one branch, labeled with negative l-goals. Let T' denote the parallel tree of some top-level negative l-goal $\neg\langle p(\overline{x}), \emptyset\rangle$, let n be its depth and let $T_0, \ldots, T_n$ be the trees pointed by the successive negative l-goals in the unique branch of T. By the same rule it follows $T_i \xrightarrow{p} T_{i+1}$ for $0 \leq i < n$, that is, being $T_0 = \langle p(\overline{x}), \emptyset\rangle$, $T_n = \mathcal{T}^n(p(\overline{x}))$. By the above observation, every T_i, and thus T', can be completely reconstructed by looking at T_n only. When T is a parallel tree rooted at some *mga* $p(\overline{x})$ let $\mathcal{N}(T)$ denote the parallel tree, rooted at $\neg\langle p(\overline{x}), \emptyset\rangle$, having the same depth than T. Then $\mathcal{N}$ is a function of its argument only (i.e. it does not depend on the program). When I is a set of parallel trees rooted at *mga*'s let $\mathcal{N}(I) \stackrel{\text{def}}{=} \{\mathcal{N}(T) \mid T \in I\}$.

It is easy to see that if G_1 and G_2 are parallel l-goals, $G_1 \approx G_2$ implies $\mathcal{T}^n(G_1) \approx \mathcal{T}^n(G_2)$ for every $n \in \omega$. Therefore, many unessential details can be hidden by switching to equivalence classes w.r.t. variance.

For every $n \in \omega$ let $\mathtt{PT}_n \stackrel{\text{def}}{=} \{\mathcal{T}^n(p(\overline{x})) \mid p(\overline{x}) \in \mathtt{MGA}\}_{/\approx}$. Since all the *mga*'s on the same predicate symbol are variants, by the previous observation it follows that

$\mathtt{PT}_n$ contains just *one* (parallel) tree for every predicate symbol. Since every parallel tree can be 'generated' by $\mathcal{T}^n$, with $n \in \omega$, then $\mathtt{PT}_\omega \stackrel{\text{def}}{=} \bigcup_{n \in \omega} \mathtt{PT}_n$ is the set of all the parallel trees of most general atoms, modulo variance.

For every set of trees I we define $\mathtt{pp}(I)$ as the closure of I w.r.t. variance, existential quantifications and parallel-products $\otimes_p$:

$$\boxed{\frac{T \in I}{T \in \mathtt{pp}(I)} \quad \frac{T_1 \in \mathtt{pp}(I) \;\; T_1 \approx T_2}{T_2 \in \mathtt{pp}(I)} \quad \frac{T \in \mathtt{pp}(I)}{\exists y T \in \mathtt{pp}(I)} \quad \frac{T_1 \in \mathtt{pp}(I) \;\; T_2 \in \mathtt{pp}(I) \;\; \mathtt{cpp}(\{T_1, T_2\})}{T_1 \otimes_p T_2 \in \mathtt{pp}(I)}}$$

Let $\mathtt{TC} \stackrel{\text{def}}{=} \{\mathcal{T}(c) \mid c \in \mathcal{C}\}$ denote the set of the (parallel) trees rooted at some constraint and let $\mathtt{Prods}(I) \stackrel{\text{def}}{=} \mathtt{pp}(I \cup \mathcal{N}(I) \cup \mathtt{TC})$. By applying the and-compositionality of parallel trees (proposition 4.1), by induction on the syntax of l-goals and by definition of $\mathtt{Prods}$ it follows

PROPOSITION 4.2 *For every $n \in \omega$,*

$$\mathtt{Prods}(\mathtt{PT}_n) = \{\mathcal{T}^n(G) \mid G \textit{ is a top-level parallel l-goal}\}$$

As a corollary we obtain that $\mathtt{Prods}(\mathtt{PT}_\omega)$ is the set of all the parallel trees of top-level l-goals.

The set $\mathtt{PT}_\omega$ of all the parallel trees of most general atoms modulo variance, is a function of the intended l-program (through equation 1, the base of relations $\mapsto$ and $\rightarrow$). Let the $\mathtt{PT}_\omega$-set of a generic l-program P be denoted by $\mathtt{PT}(P)$. Clearly, most general atoms (modulo variance) are particular cases of top-level l-goals. Moreover, by the corollary of prop. 4.2, the parallel trees in P of top-level l-goals are $\mathtt{Prods}(\mathtt{PT}(P))$, a function of $\mathtt{PT}(P)$. Therefore $\mathtt{PT}(P)$ is a *correct* and *fully abstract* [2, 5] denotation of the parallel trees in P of top-level l-goals.

4.3 Bottom-Up Construction of $\mathtt{PT}_\omega$

The bottom-up operator of def. 2.4 can be restricted to parallel trees of *mga*'s (modulo variance) as follows:

DEFINITION 4.3 *Let $\mathcal{U}_{pt}$ be the following mapping $\mathcal{U}_{pt} : \mathcal{P}(\mathtt{PT}_\omega) \rightarrow \mathcal{P}(\mathtt{PT}_\omega)$*

$$\mathcal{U}_{pt}(I) \stackrel{\text{def}}{=} \mathtt{PT}_0 \cup \{\langle p(\overline{x}), \{T_j\}_{j \in J}\rangle \mid p(\overline{x}) \in \mathtt{MGA},$$
$$p(\overline{x}) \mapsto \{\mathtt{r}(T_j)\}_{j \in J},\; \{T_j\}_{j \in J} \subseteq \mathtt{Prods}(I),\; \mathtt{cpp}(\{T_j\}_{j \in J})\}_{/\approx}$$

$\mathcal{U}_{pt}(I)$ contains (modulo variance) $\mathtt{PT}_0$ and every parallel tree of *mga* such that its subtrees belongs to I. The inclusion, by definition, of $\mathtt{PT}_0$ in every $\mathcal{U}_{pt}(I)$ agrees with the view of the single-node trees as being the join-up of an empty set of subtrees (which is contained in every I).

$\mathcal{U}_{pt}(\mathtt{PT}_n)$ can be finitely constructed, for every $n \in \omega$, as follows. For every predicate symbol p, if $\{G_j\}_{j \in J}$ are the (renamed) bodies of the l-clauses for p in the l-program, it must be verified if there are parallel trees in I, compatible w.r.t. $\otimes_p$, for (variants of) all the *mga*'s occurring in $\{G_i\}_{i \in I}$ (both as and-atomic subgoals or as roots of negative subgoals). If this is true, these trees can be combined according to the definition of Prods to obtain trees $\{T_j\}_{j \in J}$ rooted at the G_j's and compatible w.r.t. $\otimes_p$. Then $\langle p(\overline{x}), \{T_j\}_{j \in J} \rangle$, the join-up of these trees, is a parallel tree, and its equivalence class w.r.t. variance belongs to $\mathcal{U}_{pt}(\mathrm{PT}_n)$.

Being monotonic and finitary, $\mathcal{U}_{pt}$ is continuous. Moreover, by the above discussion, it easily follows that $\mathrm{PT}_{n+1} \subseteq \mathcal{U}_{pt}(\mathrm{PT}_n)$ holds for every $n \in \omega$. Hence,

COROLLARY 4.4 $\mathrm{PT}_\omega = \mathcal{U}_{pt} \uparrow \omega = lfp(\mathcal{U}_{pt})$

References

[1] K.R. Apt and K. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18:177–190, 1994.

[2] A. Bossi, M. Gabrielli, G. Levi and M. Martelli. The s-semantics approach: theory and applications. *Journal of Logic Programming*, 19,20:149–197, 1994.

[3] A. Bottoni. Analysis of SLDNF for Local CLP. Technical report, Dipartimento di Matematica Pura ed Applicata, Università di Padova, 1995.

[4] K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[5] M. Comini and G. Levi. An Algebraic Theory of Observables. In M. Bruynooghe, editor, *Proceedings of the 1994 Int'l Symposium on Logic Programming ISLP'94*, pages 172–186. The MIT Press, Cambridge, Mass., 1994.

[6] K. Kunen. Signed Data Dependencies in Logic Programs. *Journal of Logic Programming*, 7(3):231–245, 1989.

[7] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987.

[8] M. Martelli and C. Tricomi A new SLDNF-tree. *Information Processing Letters*, 43(2):57–62, 1992.

A semantics for the Kakas-Mancarella procedure for abductive logic programming

Francesca Toni
Department of Computing
Imperial College
180 Queen's Gate, London SW7 2BZ, UK
ft@doc.ic.ac.uk Tel: +44 171 594 8228 Fax: +44 171 589 1552

Abstract

The paper presents a soundness result for the Kakas-Mancarella proof procedure for abductive logic programming with respect to an argumentation-theoretic semantics. Furthermore, it discusses the relationship of the Kakas-Mancarella procedure and its semantics with other proof procedures and semantics for abductive logic programming.

Keywords: Abduction, Argumentation.

1 Introduction

Abductive logic programming (ALP) is the extension of normal logic programming (NLP) to incorporate abducibles and integrity constraints. Abducibles are atoms that can be added to programs, provided their addition does not violate the integrity constraints.

Various forms of ALP have been presented in the literature (see [6] for a survey). Among those, the form of ALP presented by Kakas and Mancarella [8, 7] has played an important role in the development of the field. Kakas and Mancarella propose both a semantics for ALP [8], by generalising the stable model semantics for NLP, and a proof procedure [7], by generalising the abductive proof procedure for NLP by Eshghi and Kowalski [5]. However, the Kakas-Mancarella (KM) procedure is not sound with respect to the generalised stable model semantics, in the same way as the Eshghi-Kowalski (EK) procedure is not sound with respect to the stable model semantics. Kakas and Mancarella, though, prove soundness for a special class of abductive logic programs, namely for abductive logic programs that admit at least a generalised stable model [1] and whose normal logic program is "well-behaved" [7].

[1]This condition is not always explicitly stated in the works of Kakas and Mancarella.

This paper presents an alternative semantics for ALP and shows that the KM procedure is sound (but not complete) with respect to this semantics, for any abductive logic program. This semantics is based upon argumentation-theoretic notions presented in [9], that are variations of those presented in [6, 4, 2, 10, 1]. The suggested semantics for ALP can be seen as an extension of the argumentation-theoretic formulation of the NLP semantics proposed by Dung [3] for the EK procedure. Therefore, the soundness result presented in this paper for the KM procedure can be seen as an extension of the soundness result given by Dung [3] for the EK procedure.

Toni and Kowalski [10] propose a transformation to reduce abductive logic programs into normal logic programs. Then, proof procedures for ALP can be obtained by applying existing procedures for NLP to the result of the transformation. In this paper, we compare the behaviour of the KM procedure applied to the original abductive logic programs with the behaviour of the EK procedure applied to the transformed normal logic programs, and we argue that the behaviour obtained in the second case is preferable.

The paper is organised as follows. In section 2 we revise the KM procedure and the generalised stable model semantics. In section 3 we present the argumentation-theoretic semantics that is the basis for providing the ALP semantics and the soundness result for the KM procedure, given in section 4 and proved in the appendix. Section 5 compares the behaviour of the KM procedure with the behaviour of the EK procedure applied to the result of reducing abductive logic programs to normal logic programs as in [10].

2 Kakas and Mancarella ALP

An abductive logic program [8, 7] is a triple $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, where

- $\mathcal{P}$ is a **normal logic program**,
- $\mathcal{AB}_{KM}$ is a set of predicate symbols, called **abducible predicates**,
- $\mathcal{I}_{KM}$ is a set of closed first-order formulas, called **integrity constraints**.

Without loss of generality, abducible predicates in $\mathcal{AB}_{KM}$ are assumed to be undefined in $\mathcal{P}$, namely, for every clause $H \leftarrow L_1, \ldots, L_n$ in $\mathcal{P}$, $H \neq a(t)$ where $a \in \mathcal{AB}_{KM}$ and t is a tuple of terms. In the sequel, with an abuse of notation, we will use $\mathcal{AB}_{KM}$ to indicate also the set of all the variable-free instances of predicates in $\mathcal{AB}_{KM}$.

The semantics of an abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$ is given by its **generalised stable models** [8]. A set M of atoms in the Herbrand base of $\mathcal{P}$ [2] is a generalised stable model of $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$ iff M is a stable model of $\mathcal{P} \cup \Delta$, for

[2] We assume that the Herbrand base of $\mathcal{P}$ includes all predicates, function and constant symbols occurring in $\mathcal{AB}_{KM}$ and $\mathcal{I}_{KM}$.

some $\Delta \subseteq \mathcal{AB}_{KM}$ (abducibles in Δ are interpreted as unitary clauses), and $M \models \phi$, for every integrity constraint $\phi \in \mathcal{I}_{KM}$.

In [7], Kakas and Mancarella define a proof procedure for the computation of the generalised stable model semantics. While defining the procedure they make the further assumptions that the integrity constraints are expressed in the form of denials of literals, and that all literals in integrity constraints are either abducible atoms or the negation of abducible atoms. In this paper, until section 5, we relax the second condition by assuming that every (denial) integrity constraint contains at least an abducible atom or the negation of an abducible atom.

Following [5], the procedure treats negative literals as additional abducibles, and, for any query Q, generates a set Δ of abducibles and negative literals as answer. We will refer to literals that are either abducibles or negative literals as **assumptions**. We will assume a safe selection rule $\Re$, namely one that selects an assumption only if it is variable-free. Moreover, for any literal L, let $\overline{L}$ be defined as follows: if $L = p(t)$ then $\overline{L} = not\ p(t)$, and if $L = not\ p(t)$ then $\overline{L} = p(t)$.

An **abductive derivation** from (G_1, Δ_1) to (G_n, Δ_n) with respect to $\Re$ is a sequence $(G_1, \Delta_1), \ldots, (G_n, \Delta_n)$ such that for all $i = 1, \ldots, n-1$, G_i is a goal of the form $\leftarrow L, Q$, where $\Re$ selects L in G_i and Q is a possibly empty conjunction of literals; for all $i = 1, \ldots, n$, Δ_i is a set of assumptions; and (G_{i+1}, Δ_{i+1}) is obtained according to one of the following rules:

(A1) if L is not an assumption, then $G_{i+1} = C$ and $\Delta_{i+1} = \Delta_i$,
where C is the resolvent of some clause in $\mathcal{P}$ with G_i on L;

(A2) if L is an assumption and $L \in \Delta_i$, then $G_{i+1} = \leftarrow Q$ and $\Delta_{i+1} = \Delta_i$;

(A3) if L is an assumption, $L \notin \Delta_i$, $\overline{L} \notin \Delta_i$,
and there exists a *successful consistency derivation* $(L, \Delta_i \cup \{L\}), \ldots, (\emptyset, \Delta')$
then $G_{i+1} = \leftarrow Q$ and $\Delta_{i+1} = \Delta'$.

A **successful abductive derivation** is an abductive derivation of the kind (G_1, Δ_1), $\ldots$, $(\Box, \Delta_n)$, with $n \geq 1$.

A **consistency derivation** from (L, Δ_1) to (S_n, Δ_n) is a sequence (L, Δ_1), $(S_1, \Delta_1), \ldots, (S_n, \Delta_n)$ such that L is an assumption; S_1 is the set of all goals of the form $\leftarrow \phi$ obtained by resolving the assumption L with the integrity constraints in $\mathcal{I}_{KM} \cup \{\neg[Q \wedge not\ Q] \mid Q$ is an atom in $\mathcal{P}\}$, with $\Box \notin S_1$; for all $i = 1, \ldots, n-1$, S_i is a set of goals of the form $\{\leftarrow L, Q\} \cup S'$; for all $i = 1, \ldots, n$, Δ_i is a set of assumptions; and (S_{i+1}, Δ_{i+1}) is obtained according to one of the following rules:

(C1) if L is not an assumption, then $S_{i+1} = C' \cup S'$ and $\Delta_{i+1} = \Delta_i$,
where C' is the set of all resolvents of rules in $\mathcal{P}$ with $\leftarrow L, Q$ on L and $\Box \notin C'$;

(C2) if L is an assumption, $L \in \Delta_i$ and $Q \neq \Box$, then $S_{i+1} = \{\leftarrow Q\} \cup S'$ and $\Delta_{i+1} = \Delta_i$;

(C3) if L is an assumption and $\overline{L} \in \Delta_i$, then $S_{i+1} = S'$ and $\Delta_{i+1} = \Delta_i$;

(C4) if L is an assumption, $L \notin \Delta_i$ and $\overline{L} \notin \Delta_i$, then
(i) if there exists a *successful abductive derivation* $(\leftarrow \overline{L}, \Delta_i), \ldots, (\Box, \Delta')$

then $S_{i+1} = S'$ and $\Delta_{i+1} = \Delta'$,

(ii) otherwise, if $\mathcal{Q} \neq \Box$, then $S_{i+1} = \{\leftarrow \mathcal{Q}\} \cup S'$ and $\Delta_{i+1} = \Delta_i$.

A **successful consistency derivation** is a consistency derivation of the kind $(L, \Delta_1), \ldots, (\emptyset, \Delta_n)$, for $n \geq 1$.

In general, this procedure is not sound with respect to the generalised stable model semantics, namely, given an abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, a query $\mathcal{Q}$ and a successful abductive derivation from $(\leftarrow \mathcal{Q}, \emptyset)$ to $(\Box, \Delta)$, there might exist no generalised stable model M of $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$ such that $M \models \mathcal{Q}$ and $M \cap \mathcal{AB}_{KM} = \Delta \cap \mathcal{AB}_{KM}$. We illustrate this unsoundness problem by means of two examples. The first example is adapted from the one given in [5] to show that the EK procedure is not sound with respect to the stable model semantics for NLP.

Example 2.1 Consider the abductive logic program $\langle \mathcal{P}, \emptyset, \emptyset, \rangle$ with $\mathcal{P}$

$$\begin{array}{lcl} r & \leftarrow & not\, r \\ r & \leftarrow & q \\ p & \leftarrow & not\, q \\ q & \leftarrow & not\, p \end{array}$$

There is a successful abductive derivation from $(\leftarrow p, \emptyset)$ to $(\Box, \{not\ q\})$. However, the only generalised stable model of $\langle \mathcal{P}, \emptyset, \emptyset, \rangle$ is $\{q, r\}$, and $\{q, r\} \not\models p$.

Example 2.2 Consider the abductive program $\langle \{p \leftarrow b\}, \{a, b\}, \{\neg a, \ \neg not\ a\} \rangle$. There is a successful abductive derivation from $(\leftarrow p, \emptyset)$ to $(\Box, \{b\})$. However, there is no generalised stable model of the given abductive program.

The soundness result has been proved to hold for abductive logic programs that admit at least a generalised stable model and whose normal logic program component is "well-behaved" [7]. (The logic program in the first example above is not "well-behaved".) To obtain a more general soundness result one might modify the procedure or adopt a new semantics. In this paper we follow the second alternative. The new semantics, defined in argumentation-theoretic terms, is based upon notions presented in the next section.

3 Argumentation semantics

The semantics underlying the KM procedure is expressed in the context of the instance for ALP of an abstract assumption-based framework [9] which is a variant of those proposed in [4, 2, 1] as a semantics for non-monotonic reasoning in general. An **assumption-based framework** is a tuple $\langle \mathcal{T}, \vdash, \mathcal{AB}, \mathcal{IC} \rangle$ where

- $\mathcal{T}$ is a **theory** in some formal language,
- $\vdash$ is a notion of **monotonic derivability** for the given language,

- $\mathcal{AB}$ is a set of **assumptions**, which are sentences of the language, and
- $\mathcal{IC}$ is a set of **integrity constraints**, which are denials of sentences of the language.

In such an (abstract) framework a sentence is a non-monotonic consequence if it follows monotonically from the theory extended by means of an "acceptable" set of assumptions. Various notions of "acceptability" can be defined, based upon a single notion of "attack" between sets of assumptions. Intuitively, one set of assumptions "attacks" another if the two sets together with the theory violate an integrity constraint (i.e. the two sets together with the theory derive, via $\vdash$, all sentences in the denial integrity constraint), and the second set is deemed responsible for the violation. We assume that some of the sentences in the integrity constraints in $\mathcal{IC}$ are explicitly indicated as **retractibles**, with the intended meaning that if a violation takes place, then integrity satisfaction should be restored by "retracting" the sentences explicitly indicated as retractible. Then, the derivation of some of the retractibles sanctions the responsibility in the integrity violation.

Definition 3.1 Given an argumentation framework $\langle \mathcal{T}, \vdash, \mathcal{AB}, \mathcal{IC} \rangle$, where $\mathcal{IC}$ is a set of denial integrity constraints, each one with at least one retractible:

- a set of assumptions $\mathcal{A} \subseteq \mathcal{AB}$ **attacks** another set $\Delta \subseteq \mathcal{AB}$ iff for some integrity constraint $\neg[L_1 \wedge \ldots \wedge L_i \wedge \ldots \wedge L_n] \in \mathcal{IC}$ with L_i retractible,

  $\mathcal{T} \cup \mathcal{A} \vdash \ L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$, and

  $\mathcal{T} \cup \Delta \vdash L_i$.

Various notions of "acceptability" can be defined in terms of the same notion of attack. Here we mention only some of the notions presented in [4, 2, 1, 9]: A set of assumptions which does not attack itself is called **stable**, iff it attacks all assumptions it does not contain ($\mathcal{A} \subseteq \mathcal{AB}$ attacks $\delta \in \mathcal{AB}$ iff $\mathcal{A}$ attacks $\{\delta\}$); **admissible**, iff it attacks all sets of assumptions that attack it; **preferred**, iff it is maximally (with respect to set inclusion) admissible.

ALP can be given a semantics by appropriately instantiating any of these abstract semantics. Given an abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, the corresponding assumption-based framework is $\langle \mathcal{T}, \vdash, \mathcal{AB}, \mathcal{IC} \rangle$ where

- $\mathcal{T}$ is the set of all variable-free instances of clauses in $\mathcal{P}$;
- $\vdash$ is modus ponens for the clause implication symbol $\leftarrow$;
- $\mathcal{AB}$ is $\mathcal{AB}_{KM}$ together with the set of all variable-free negative literals;
- $\mathcal{IC}$ is the set consisting of (A is a variable-free atom)
  (1) all denials of the form $\neg[A \wedge not\ A]$ with $not\ A$ retractible,
  (2) all denials of the form $\neg[A \wedge not\ A]$ with A abducible and retractible, and
  (3) all variable-free instances of the integrity constraints in $\mathcal{I}_{KM}$, with all assumptions (i.e. abducibles or negative literals) retractible.

Note that, in the domain specific integrity constraints, literals that are not assumptions are not considered as retractible. This choice is made to conform to the intended meaning of retractibles in integrity constraints, as sentences that can be retracted to restore integrity satisfaction. In fact, it might not be possible to retract literals that are not assumptions, if these literals can be derived from the program without the addition of any assumption. In general, however, it might be convenient to allow the user to choose directly retractibles in the integrity constraints when defining the domain-specific abductive logic program (e.g. see [10]). In this case, the user should take care of choosing retractibles so that the choice conforms to their intended meaning.

Note also that, in the instance of the assumption-based framework for ALP, negative literals are treated as syntactic object, whose derivation solely depends on their assumption. Therefore, a theory $\mathcal{P}\cup\Delta$, for some set of assumptions Δ in the assumption-based framework for $\langle\mathcal{P},\mathcal{AB}_{KM},\mathcal{I}_{KM}\rangle$, can be seen as a Horn program. This feature is made explicit in [5] and [8, 7] by explicitly renaming negative literals as positive atoms.

The instance for ALP of the notion of attack between sets of assumptions is:

- a set of assumptions $\mathcal{A}$ attacks another set Δ iff
  - (i) there exists an atom Q such that $\mathcal{P}\cup\mathcal{A}\vdash Q$ and *not* $Q\in\Delta$, or
  - (ii) there exists an abducible atom Q such that *not* $Q\in\mathcal{A}$ and $Q\in\Delta$, or
  - (iii) there exists a variable-free instance of an integrity constraint $\neg[L_1\wedge\ldots\wedge L_i\wedge\ldots\wedge L_n]\in\mathcal{I}_{KM}$ with L_i retractible such that $\mathcal{P}\cup\mathcal{A}\vdash L_1,\ldots,L_{i-1},L_{i+1},\ldots,L_n$ and $L_i\in\Delta$ [3].

In this concrete assumption-based framework, stable sets of assumptions correspond to generalised stable models (see [9]). Note that, since NLP is a special case of ALP, the semantics of NLP is a special instance of the semantics of ALP given by the assumption-based framework above. In the assumption-based framework for NLP, stable sets of assumptions correspond to stable models and preferred sets of assumptions correspond to partial stable models/preferred extensions [4, 2, 1].

4 Soundness result

Theorem 4.1 Given an abductive logic program $\langle\mathcal{P},\mathcal{AB}_{KM},\mathcal{I}_{KM}\rangle$ and a conjunction of literals $\mathcal{Q}$, if there is an abductive derivation from $(\leftarrow\mathcal{Q},\emptyset)$ to $(\Box,\Delta)$, then, in the assumption-based framework for $\langle\mathcal{P},\mathcal{AB}_{KM},\mathcal{I}_{KM}\rangle$,

1. $\mathcal{P}\cup\Delta\vdash\mathcal{Q}$, and

[3]This condition replaces the condition $\mathcal{P}\cup\Delta\vdash L_i$ which is equivalent to it due to the fact that all retractibles in $\mathcal{I}_{KM}$ are assumptions and assumptions are not defined in $\mathcal{P}$.

2. for each set of assumptions $\mathcal{A}$, if $\mathcal{A}$ attacks Δ (by cases (i) or (ii) or (iii)), then Δ attacks $\mathcal{A}-\Delta$ by case (i) or by case (ii).

The proof of this theorem can be found in the appendix.

The procedure is not complete with respect to the semantics expressed in theorem 4.1, due to the possible non-termination of the procedure for some queries.

Directly from the soundness result, the KM procedure allows domain-specific integrity constraints to be used to attack Δ but not to counter attack the attacks against Δ. The following example illustrates the disadvantages of this asymmetric use of domain-specific integrity constraints.

Example 4.1 Consider the propositional abductive logic program

$$\begin{aligned}\mathcal{P} &= \{q\leftarrow b,\ p\leftarrow a\}\\ \mathcal{AB}_{KM} &= \{a, b\}\\ \mathcal{I}_{KM} &= \{\neg[p\wedge b],\ \neg a,\ \neg not\ a\}\end{aligned}$$

There is no successful abductive derivation from $(\leftarrow q,\emptyset)$. In fact, in the corresponding assumption-based framework, the set $\{b\}$ (needed to prove q) is attacked by the set $\{a\}$, but $\{b\}$ does not counter attack the attack $\{a\}$ via case (i) or (ii). Moreover, the set $\{b, not\ a\}$, which counter attacks the attack $\{a\}$ via case (i) or (ii), is attacked by $\emptyset$, which can not be counter attacked via case (i) or (ii). However, q should intuitively hold, since p can not possibly hold. By allowing a symmetric use of domain-specific integrity constraints to counter attack, $\{b\}$ can be shown to be "acceptable", since $\emptyset$, and therefore $\{b\}$, counter attacks the attack $\{a\}$ against it via case (iii), and thus q can be shown to hold.

One way to obtain the behaviour suggested in the example is to modify the KM procedure to allow a symmetric use of domain-specific integrity constraints. The following section illustrates an alternative way of obtaining the same behaviour, by transforming the given abductive logic program into a normal logic program first [10] and then applying an existing proof procedure for NLP, the EK procedure.

5 Comparison with the EK procedure

New proof procedures for ALP can be obtained by applying existing proof procedures for NLP to the normal logic programs $\mathcal{P}'=\mathcal{P}^{NAF}\cup\mathcal{P}^{\mathcal{I}_{KM}}\cup\mathcal{P}^{\mathcal{AB}_{KM}}$ obtained by applying the transformation of [10] to abductive logic programs $\langle\mathcal{P},\mathcal{AB}_{KM},\mathcal{I}_{KM}\rangle$, where, if for every abducible predicate a in $\mathcal{AB}_{KM}$, a' stands for the "complement" of a:

- $\mathcal{P}^{NAF}$ is $\mathcal{P}$ with all abducible atoms $a(t)$ replaced by a negative literal *not* $a'(t)$;

- for every $\neg[L_1 \wedge \ldots \wedge L_i \wedge \ldots \wedge L_n] \in \mathcal{I}_{KM}$ and every retractible L_i in it, $\mathcal{P}^{I_{KM}}$ contains a clause

$$\alpha \leftarrow L'_1, \ldots, L'_{i-1}, L'_{i+1}, \ldots, L'_n$$

where for each $j = 1, \ldots, n$, $j \neq i$, if $L_j = a(t)$ with $a \in \mathcal{AB}_{KM}$ then $L'_j =$ *not* $a'(t)$, otherwise $L'_j = L_j$, and
if $L_i =$ *not* $a(t)$ then $\alpha = a(t)$, otherwise, if $L_i = a(t)$, then $\alpha = a'(t)$;

- for every abducible predicate a in $\mathcal{AB}_{KM}$, $\mathcal{P}^{AB_{KM}}$ contains a pair of clauses

$$a(X) \leftarrow not\ a'(X)$$
$$a'(X) \leftarrow not\ a(X)$$

If we adopt Kakas and Mancarella's restriction, that all literals in domain-specific integrity constraints are either abducible atoms or the negation of abducible atoms, then it can be shown that the original abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$ and the transformed normal logic program $\mathcal{P}'$ are equivalent with respect to all semantics defined in the underlying assumption-based frameworks. This result is a corollary of the basic result that attacks before and after the transformation are preserved (see [9, 10] for more details). As a consequence, any proof procedure for NLP, sound with respect to an argumentation-theoretic semantics expressed in the assumption-based framework for NLP, provides a proof procedure for ALP, sound with respect to the same semantics but expressed in the assumption-based framework for ALP. In particular, since the EK procedure is sound with respect to the preferred extension semantics [3], by applying the EK procedure to the normal logic program $\mathcal{P}'$ corresponding to an abductive logic program $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, we obtain a procedure for ALP computing preferred sets of assumptions, which therefore treats symmetrically integrity constraints to attack and counter attack. This is illustrated by the following example.

Example 5.1 The abductive logic program in example 4.1 is reduced to the normal logic program $\mathcal{P}'$

$$q \leftarrow not\ b' \qquad b' \leftarrow p \qquad a \leftarrow not\ a'$$
$$p \leftarrow not\ a' \qquad a' \qquad a' \leftarrow not\ a$$
$$a \qquad b \leftarrow not\ b'$$
$$b' \leftarrow not\ b$$

where a' and b' are new atoms standing for the "complement" of the abducibles a and b, respectively. By applying the EK procedure to the program $\mathcal{P}'$, the query $\leftarrow q$ succeeds to be proved, with underlying negative assumption *not* b', corresponding to the positive abducible b with respect to $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$. In the proof of $\leftarrow q$, the attacks $\{not\ a'\}$ and $\{not\ b\}$ against $\{not\ b'\}$, corresponding to the attacks $\{a\}$ and $\{not\ b\}$ against $\{b\}$ with respect to $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$, are counter attacked by $\emptyset$ and $\{not\ b'\}$, respectively. The attack $\emptyset$ against $\{not\ a'\}$ corresponds to the attack $\emptyset$ against $\{a\}$ via case (iii) with respect to $\langle \mathcal{P}, \mathcal{AB}_{KM}, \mathcal{I}_{KM} \rangle$.

6 Conclusions

We have presented a soundness result for the Kakas-Mancarella proof procedure for ALP with respect to an argumentation-theoretic semantics. We have shown other semantics for ALP and pointed to proof procedures for their computation. These procedures are based upon transforming abductive logic programs into normal logic programs first, and then applying existing semantics for NLP to the result of the transformation. We have argued that the behaviour of the Eshghi-Kowalski procedure applied the transformed normal programs is preferable to the behaviour of the Kakas-Mancarella proof procedure applied to the original logic programs.

Acknowledgements

This research was supported by the Fujitsu Research Laboratories. The author is grateful to Robert A. Kowalski for helpful suggestions and to Tony Kakas and Paolo Mancarella for helpful discussions.

References

[1] A. Bondarenko, P.M. Dung, R.A. Kowalski, and F. Toni. An abstract, argumentation-theoretic framework for default reasoning. In preparation, 1995.

[2] A. Bondarenko, F. Toni, and R. A. Kowalski. An assumption-based framework for non-monotonic reasoning. In L.M. Pereira and A. Nerode, editors, *Proc. International Workshop on Logic Programming and Nonmonotonic Reasoning*, pages 171–189, Lisbon, 1993. MIT press.

[3] P.M. Dung. Negation as hypothesis: an abductive foundation for logic programming. In K. Furukawa, editor, *Proc. ICLP*, pages 3–17, Paris, 1991. MIT Press.

[4] P.M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning and logic programming. In R. Bajcsy, editor, *Proc. IJCAI*, pages 852–857, Chambery, France, 1993. Morgan Kaufmann.

[5] K. Eshghi and R.A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. ICLP*, pages 234–255, Lisbon, 1989. MIT Press.

[6] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, 1993.

[7] A.C. Kakas and P. Mancarella. Abductive logic programming. In W. Marek, A. Nerode, D. Pedreschi, and V.S. Subrahmanian, editors, *Proc. NACLP Workshop on Non-Monotonic Reasoning and Logic Programming*, Austin, Texas, 1990.

[8] A.C. Kakas and P. Mancarella. Generalised stable models: a semantics for abduction. In L. Carlucci Aiello, editor, *Proc. ECAI*, pages 385–391, Stockolm, 1990. Pitman.