In [Prz1] and [ABW] a semantical characterization of the stratified programs is given: the perfect model semantics.

Let $\mathcal{P}$ and $\mathcal{N}$ be models of a logic program $P$. We say that $\mathcal{N}$ is preferable to $\mathcal{P}$ iff for every ground atom $A$ in $\mathcal{N}\setminus\mathcal{P}$ there exists a ground atom $B$ in $\mathcal{P}\setminus\mathcal{N}$ such that $A \leftarrow B$.

A model $\mathcal{P}$ is perfect if there are no models preferable to $\mathcal{P}$.

Theorem 1 [ABW, Prz1, Lit]

A stratified program has a (unique) perfect model.

The computation of such a perfect model is given bottom-up, by iterating the immediate consequence step $\mathcal{T}_A$ on each stratum and finally by taking the set union.

Let $A$ be a $\mathcal{L}$-interpretation. We will use $\mathcal{B}_{R@D, A}$ to denote the set $\{p_0, \ldots, p_n\}: p$ is a $\mathcal{L}$-ary predicate symbol and $n_0, \ldots, n_n$ in the domain of $A$.

Theorem 2 [ABW]

Let $P$ be a program, $A$ a $\mathcal{L}$-interpretation, $I \subseteq \mathcal{B}_{R@D, A}$. Then $A \cup I$, together with an interpretation of "-" that satisfies the equality theory of comp($P$), is a model of comp($P$) iff $I$ is a closed point for the map $\mathcal{T}_A$, defined as follows:

$\mathcal{T}_A(I) = \{D : D \in B_1 \cup \ldots \cup B_n \cup C_1 \cup \ldots \cup C_n\}$. A closed instance on $A$ of a clause in $P$ with $B_1$ and $C_i \neq \emptyset$, for every $i \in S$, is $I[S]$.

Corollary: The perfect model for a stratified program $P$ is also a model for the completion of $P$.

The computation of a perfect model for a stratified program follows the global dependency graph. It attempts the minimality of the extension of each predicate as much as it is possible by taking fixed all previous predicates and by taking free all following predicates w.r.t. the dependency order, as shown by this theorem:

Theorem 3 [Prz1]

The perfect model for a stratified program $P$ satisfies the prioritized circumscription formula $\cup_{S \subseteq \text{Succ}(S')} \mathcal{C}_S(S') : \{q : q \in Q\}$.

A more general class of nonlocal programs which consistent completion was defined [Sat, Carv]: the call-consistent programs.

A program $P$ is call-consistent iff for no predicate symbol $p$, $p <_P p$.

It is easy to prove that the following conditions are equivalent:

i) $P$ is call-consistent.

ii) For all $p, q \in \text{R@D}$ it never holds: $p <_P q$, $p <_P q$, $q <_P p$.

iii) There exists a mapping $\gamma$ such that for every predicate $p, q \in \text{R@D}$, if $p$ depends on $q$ then $l(e(p)) > l(e(q))$ and if $p$ depends both positively and negatively on $q$ then $l(e(p)) > l(e(q))$.

Then the graph characterization of the call-consistent programs is the following: in every strong connected component there is no cycle with an odd number of negatives edges.

Remark: All stratified programs are call-consistent.

The inverse is false, as shown by the following program, which is call-consistent but not stratified:

\[
\begin{align*}
p & \leftarrow -q. \\
q & \leftarrow -p. 
\end{align*}
\]

The computation of a model for the completion of a call-consistent program was shown by Kuni [Kun] with a 3-valued logic.

Theorem 4 [Kun, Bar]

Let $P$ be a call-consistent logic program. There exists a model for the Clark's completion of $P$, i.e. $\text{comp}(P)$ is consistent.

Here we consider the proof given in [Bar], that relates Kun's approach and Fitting's notion of partial model [Fit]. The main idea is to construct the model by induction on the dependency order, by expanding step by step the model to an upper minimal strongly connected component. At each step two subsets of By are computed: all instances that have to be true in the model (I) and all instances that have to be false (J), for every predicate in the given equivalence class; moreover, to decide the truth value of all predicative instances still "undecided", select a class representative $p$ for: for every $q$ in the class and for every $b$-tuple $\langle i_1, \ldots, i_n \rangle$ in the Herbrand Universe, set:

$\mathcal{g}(\langle i_1, \ldots, i_n \rangle)$ is true iff $\mathcal{g}(\langle i_1, \ldots, i_n \rangle) \in I$ or $\mathcal{g}(\langle i_1, \ldots, i_n \rangle) \in J$ and $q <_P p$.

These models are named KB models. Moreover if $\mathcal{KB} = \mathcal{KB} \ldots \mathcal{KB}$, the selected representatives of all the equivalence classes, say $\mathcal{KB} = \mathcal{KB} \ldots \mathcal{KB}$, the KB-model will be denoted by $\mathcal{KB} = \mathcal{KB} \ldots \mathcal{KB}$.

Theorem 5 [CF]

Let $P$ be a stratified program. The (unique) KB-model for $\text{comp}(P)$ is perfect.

Remark: Generally the KB-models for call-consistent programs are not perfect, as shown by example 2: this program has two different KB-models, one for every selected representative in the unique equivalence class: $M = \{p\}$ and $N = \{q\}$. The first one is preferable to the second one and viceversa. Then no one is perfect.

Theorem 2 creates a link between the perfect model for a stratified program $P$ and the satisfiability of the prioritized circumscription formula on all predicates in $P$.

A similar result can be shown for call-consistent programs by considering the prioritized circumscription w.r.t. a set of predicates $S$ such that $S$ contains exactly one representative for each equivalence class.

Theorem 6 [CF]

Let $P$ be a call-consistent program. The model $\mathcal{KB} = \mathcal{KB} \ldots \mathcal{KB}$ for the Clark's completion of $P$ satisfies the prioritized circumscription formula restricted on $\mathcal{KB} = \mathcal{KB} \ldots \mathcal{KB}$, that is:

$\cup_{S \subseteq \text{Succ}(S')} \mathcal{C}_S(S') : \{q : q \in Q\}$.
Note that every equivalence class \([\mathcal{W}]\) is separable on two disjoint subsets \([\mathcal{W}]^+, [\mathcal{W}]^-\) such that \(i \in [\mathcal{W}]^+\) \(\iff\) \(i \in \{1, \ldots, \mathcal{q}\}\) \(\mathcal{q} \in [\mathcal{W}]^-\) \(\iff\) \(j \leq \mathcal{q}\) \(\land\) \(i > \mathcal{q}\). Then it is easy to see that every representative in \([\mathcal{W}]^+\) (or in \([\mathcal{W}]^-\)) gives the same expansion, and so it holds the following theorem:

**Theorem 7** [CF]  
Let \(P\) be a call-consistent program. Let \(n\) be the number of the equivalence classes w.r.t. the equivalence relation \(=_{\mathcal{W}}\). Let \(m \leq \mathcal{q}\) be the number of the classes without negation. The KB-models for the Clark's completion of \(P\) are exactly \(2^m\).

**example 3:**  
\[\begin{align*}
q(i) & \leftarrow q(i), p(i). \\
r(i) & \leftarrow q(i). \\
p(i) & \leftarrow q(i). \\
p(0) & \leftarrow r(0). \\
q(0) & \leftarrow q(0). \\
q(0) & \leftarrow r(0). \\
p(0) & \leftarrow q(0).
\end{align*}\]

The equivalence classes for \(P\) are \([r]\) and \([p,q,s]\). Let us select \(r\) and \(p\). \(KB_{p,q,s} = (\mathcal{K}(r), p(0), \mathcal{W}(q), \mathcal{W}(s)).\) The circumscription formula \(\mathcal{C}(P; \{p,q,s\})\) \(\cup \mathcal{C}(P; \{p,q,s\})\) satisfies the circumscription formula \(\mathcal{C}(P; \{p,q,s\})\) \(\cup \mathcal{C}(P; \{p,q,s\})\). Moreover we can see that the KB-models for \(P\) are exactly \(2^{\mathcal{q}} = 2\), i.e., \(KB_{p,q,s} = KB_{q,s}\).

3. Local properties.

Till now we have seen two classes of programs that are guaranteed to have consistent completion: the stratified and the call-consistent programs.

The proofs of the consistency are based on two facts:

1. the existence of a well-founded partial order on the dependency graph
2. the use of this order for constructing KB-models.

Other extended classes that have consistent completion have been defined [Car, Pir, Pir2, Sat]. They also use appropriate extensions of the above facts. In what follows, the definitions of these extended classes will be presented in a systematic way, based on the above facts. In order to clarify the relationship among them.

The first extension consists in requiring the existence of a well-founded order on the predicate instance level, called local level. The main problem here is that while at the global level the dependency graph was finite, at the local level we can meet infinite dependency chains. Then a strong condition on the order has to be imposed to ensure the computability of a completion model.

A logic program \(P\) is locally stratified if it is possible to decompose the Herbrand basis of \(P\) into disjoint sets, called strata \(H_0, H_1, \ldots, H_n\), where each stratum \(H_t\) is a countable ordinal, so that for every ground class \(A = \{B_0, \ldots, B_n, \ldots\}\) \(\iff\) \(A\) belongs to \(H_t\) then

1. all positive premises \(B_i\) belong to \(\bigcup H_{i+j}\)
2. all negative premises \(D_j\) belong to \(\bigcup H_i\).

**example 4:**  
\[\begin{align*}
P & \leftarrow \text{even}(0), \text{even}(x+2), \text{even}(x) \leftarrow \neg \text{even}(x). \\
\text{even}(0) & \leftarrow. \\
\text{even}(x+2) & \leftarrow \text{even}(x). \\
\text{even}(x) & \leftarrow \neg \text{even}(x).
\end{align*}\]

This program is locally stratified but not stratified.”

Prenumiski's [Pren1] shows that every locally stratified program has (exactly) one perfect model. From the construction of the model one can easily see that it's also a model of the Clark's completion. Perfect models of locally stratified programs satisfy the prioritized circumscription formula [Pren1, Pren2].

Consider a program consisting only of ground classes and let \(H_0, H_1, \ldots, H_n\) be a decomposition of the Herbrand base of \(P\). With \(f\), we denote the program obtained by taking the classes \(A = a\) of \(P\) such that \(A\) belongs to \(H_0\) and dropping from \(a\) all literals that do not belong to \(H_0\). If \(P\) is locally stratified w.r.t. the given decomposition, then \(P\) is obviously a definite program (without negation). One may wonder whether a class larger than that of locally stratified programs could be obtained if one requires for any program \(P\) the existence of a decomposition \(D\) of the Herbrand basis of \(P\) such that each program \(P_{D}\) is stratified (instead of definite as before). It is easy to see that it is not the case: the decomposition \(D\) can be refined obtaining a new decomposition w.r.t. which \(P\) is locally stratified:

**Call-consistency** also has been extended at the local level: such extension can be defined in two ways [Car, Sat, Bar]:

- A program \(P\) is locally call-consistent if it has a local mapping \(f\) such that for every atom \(A,B\) the Herbrand Basis, if \(A\) depends on \(B\) then \(lev(B) \leq lev(A)\) and if \(A\) depends positively on \(B\) then \(lev(B) < lev(A)\).

- A program \(P\) is locally semi-strict [Car] (negative cycle free in [Sat]) if for no \(A \in\) the Herbrand Basis of \(P\) \(A <_{L} A\).

Obviously, for the locally call-consistent class, one guarantees that the dependency relation on the predicate instance is well founded, whereas this is not the case for the locally semi-strict programs. In fact, one can easily see that, at the local level, these two classes are different:

**example 5** [Car]:  
\[\begin{align*}
P & \leftarrow p(0), p(s). \\
p(0) & \leftarrow p(s). \\
p(s) & \leftarrow p(0) \leftarrow p(s).
\end{align*}\]

This program is locally semi-strict but no locally call-consistent.

As shown in [Sat] the completion of a locally call-consistent program is consistent, whereas for the locally semi-strict program the question is still open [Car].
4. Weak properties.

Przymusinski in [Prz3] extended the locally stratified class to the weakly stratified programs. The main idea is to remove all "irrelevant" relations in the local dependency graph, modifying the program by the Davis Putnam rule.

Let P be a program such that the partial order < on the strongly connected components in the dependency graph is well-founded.

The bottom stratum is the union of all minimal components of P (i.e., S(P) = U(C: C is a minimal component in the graph of P)), and the bottom layer L(P) is the corresponding subprogram of P, i.e., the set of all clauses from P whose heads belong to S(P).

Given M a subset of S(P), for an atom A in S(P) we will say that A is true in M iff A is in M. By a reduction modus M we mean a new program P/M obtained from P by performing the following two reductions (Davis Putnam rule):

- removing from P all clauses already satisfied in M;
- removing from all the remaining clauses those premises which are satisfied in the model M.

Then we define a recursive operation which leads to the construction of a weakly partial perfect model: define P1 to be P itself, S1 = S(P1), L1 = L(P1) and let M1 be the least model of L1 if it exists. Having defined Pk, Sk, Lk, and Mk for some k (0 ≤ k), we define Pk+1 = Pk / Mk and proceed as follows:

- if Sk+1 is empty then the construction stops and Mk+1 = Mk ∪ M1 ∪ ... ∪ Mk is defined to be the weakly perfect model M of P,
- otherwise, let Sk+1 = S(Pk+1) and Lk+1 = L(Pk+1) and let Mk+1 be the least model of Lk+1.

A logic program P is weakly stratified if all layers Lk are positive logic programs. Thus, obviously, every weakly stratified program has a unique weakly perfect model.

A simple example may help the intuition.

**Example 6** P =

\[ s \leftarrow -r, -q. \]
\[ q \leftarrow -a, -q. \]
\[ r \leftarrow -a, q. \]
\[ p \leftarrow -r. \]

\[ S_0 = \{ s \}. \]
\[ L_0 = \{ s \}. \]
\[ M_0 = \varnothing. \]
\[ P_0 = \{ p \leftarrow -r, q \leftarrow -q. \}. \]
\[ M_0 = \varnothing. \]
\[ P_0 = \{ p \leftarrow -r, q \leftarrow -q. \}. \]
\[ M_0 = \varnothing. \]
\[ S_1 = \{ s \}. \]
\[ L_1 = \{ s \}. \]
\[ M_1 = \varnothing. \]
\[ P_1 = \{ p \leftarrow -r, q \leftarrow -q. \}. \]
\[ M_1 = \varnothing. \]
\[ S_2 = \{ r \}. \]
\[ L_2 = \{ r \}. \]
\[ M_2 = \varnothing. \]
\[ P_2 = \{ p \leftarrow -r, q \leftarrow -q. \}. \]
\[ M_2 = \varnothing. \]
\[ M_P = M_0 \cup M_1 \cup M_2 = \{ s \}. \]

This program is weakly-stratified but not locally stratified.

Additionally, perfect models of weakly stratified programs satisfy the priorized circumscription formula [Prz3].

Let us summarize. Till now we have seen two ways of assuring a well-founded order: stratification and call-consistency. Both these ways have been considered both as the predicate symbol level and at the level. Finally, Przymusinski introduced the idea of simplifying the program (by the Putnam Davis method) while iteratively constructing its model, but he applied it only to the definite approach and not to the stratification and call-consistency approach. It is easy to see, in fact, that the previous construction is sound even when all layers Lk are stratified or call-consistent, since we know the way to construct step by step a perfect or KB-model for these programs.

As observed before for the locally stratified programs, requiring that all layers are stratified logic programs instead of definite ones does not result in a larger class of programs. At the contrary the call-consistent property gives a larger class:

A logic program is weakly call-consistent if all layers are call-consistent logic programs. Clearly, every weakly call-consistent program has a (generally non unique) weakly-KB-model:

**Example 7.**

\[ p \leftarrow -q. \]
\[ q \leftarrow -p. \]

This program is weakly call-consistent but not weakly stratified:

\[ S_0 = \{ p, q \}. \]
\[ L_0 = \{ p \leftarrow -q, q \leftarrow -p \}. \]
\[ M_0 = \{ q \}. \]
\[ P_0 = \{ p \leftarrow -q, q \leftarrow -p \}. \]
\[ M_0 = \varnothing. \]
\[ M_P = \{ q \}. \]

**Theorem 8.** [CF]

Every locally call-consistent program P is weakly call-consistent.

Remark that generally the inverse is false, as shown by the following [Prz3]:

**Example 8.**

\( p(x, y) \leftarrow p(y, z), \neg q(x, y). \)

The new class satisfies the expected consistency property.

**Theorem 9.** [CF]

Let P be a weakly call-consistent program. The weakly-KB-models for P satisfy the Clark completion of P.
5. References


A Theory for Modeling the Synchronization Mechanisms of Concurrent Logic Languages

Catuscia Palamidessi
Dipartimento di Informatica, Corso Italia, 40, 56125 Pisa ITALY

1. Introduction.

Concurrent Logic Languages offer a high-level computational model that lends itself to a wide range of concurrent programming techniques. They are based on a subset of the First Order Logic, namely the Prolog Clause Logic (HCL) [vanEijnden76, Lloyd87, Apt88]. The main advantage of logic languages is that programs are sets of assertions which have a declarative reading. In other words, they can be seen as the description of the problem to be solved, and they can be understood without reference to the behavior of any particular machine. An other advantage is the possibility of representing data structures as logical terms and manipulating them by using unification. Finally, these languages seem, by their nature, quite suitable to express the parallelism implicit in certain problems. In particular, the atoms in a goal can naturally be interpreted as processes, and the shared variables as communication channels. However, pure HCL languages are not expressive enough to model the actual aspects of concurrent systems, such as real-time, synchronization, nondeterminism-control.

Most of the Concurrent Logic Languages extend HCL by adding some constructs for explicit concurrency. In particular, constructs for guarded nondeterminism and for synchronization. The proposals for controlling nondeterminism are based essentially on the introduction of the commit operator and of the guarded-clause. On the other side, there are many proposals concerning the synchronization mechanisms. The most popular ones are based on the idea of constraining the unification (input-constraints). In practice, some of the variables occurring in a process can be prevented from getting bound during the unification preceding a computation step. In this way, the process in which such a variable occur can be forced to wait (suspension rule) until other processes in the goal have made available the required bindings. In this way, the shared variables can be seen as directed channels, and the suspension rule provides a synchronization mechanism. This class of languages includes Concurrent Prolog (CP) [Shapiro83, Shapiro86, Shapiro87], Guarded Horn Clauses (GHC) [Ueda85, Ueda86] and PARLOG [Clark85, Clark86, Gregory87].

It has often been argued that the additional features of Concurrent Logic Languages heavily affect the clarity and the semantics purity of HCL. The main problem is not the commit operator. In fact, the presence of commits does not change the success set of a logic program. On the contrary, the input-constraints usually restrict the number of successful computations. Moreover, the declarative reading of a predicate definition is lost. For instance, the invariance of the arguments does not hold anymore.
2. The concurrent logic languages based on constraints on unification

In this section we introduce briefly the class of concurrent logic languages whose synchronization mechanism is based on input-mode constraints. We introduce first the common structure these languages have, and then we present in more details the particular cases of CP, GHC, and PARLOG.

The alphabet consists of a set \( P \) of predicate symbols \( p, q, r, \ldots \), a set \( C \) of constructor symbols, \( a, b, c, \ldots, f, g, h, \ldots \), and a set \( V \) of variables, \( x, y, z, \ldots \). The basic construct in the guarded clause is the guarded clause:

\[
A \leftarrow G_1, \ldots, G_m \mid B_1, \ldots, B_k
\]

where \( A, G_1 \ldots, G_m, B_1 \ldots, B_k \) are atoms, namely objects of the form \( p(t_1, \ldots, t_q) \), where \( t_1 \ldots t_q \) are terms built on \( C \) and \( V \). \( A \) is called the head of the clause, \( G_1 \ldots G_m \) is called the guard part, \( B_1 \ldots B_k \) is called the body part. The symbol \( \leftarrow \) is called commut operator. A program is a finite set of guarded clauses. A goal is a construct of the form

\[
\leftarrow B_1, \ldots, B_k
\]

where \( B_1, \ldots, B_k \) are atoms.

According to the process interpretation [Shapiro83, Levi85] a goal can be seen as a network of processes (the atoms) communicating via the shared variables (seen as channels). Some constraints (input constraints) can be specified on the clauses and goals. Their role is essentially to prevent the use of a certain clause, during the computation of a process. CP, GHC and PARLOG use different kinds of input constraints. We will consider them later.

We introduce now the computational model. The actual computational models of CP, GHC and PARLOG slightly differ from each other, and from the one we present here. However, the differences are not so relevant and they do not concern the main features of the synchronization mechanisms.

The execution of a process \( B_j \) is based on the derivation step. The clauses of the program are tried in parallel. They are renamed, in order to avoid clash of variables. The attempt to apply a clause \( A \leftarrow G_1 \ldots G_m \mid B_1 \ldots B_k \) can give one of the following results:
- fail, if \( A \) and \( B_j \) are not unifiable (i.e. they cannot become identical by substitution variables-terms),
- guard part (seen as a goal) fails (see below).
- succeed, if \( A \) and \( B_j \) are unifiable, the guard part (seen as a goal) succeeds (see below), and the input constraints are satisfied.
- suspend, if \( A \) and \( B_j \) are unifiable, the guard part is verified, but the input constraints are not satisfied.

If all the clauses fail, then \( B_j \) (and the whole goal) fails. If no clauses succeed, and some of them suspend, then \( B_j \) suspends. If some of the clauses succeeds, then one of them is nondeterministically selected, say \( A \leftarrow G_1 \ldots G_m \mid B_1 \ldots B_k, \) and all the other attempts are discharged. Then, the derivation step takes place: \( B_j \) is replaced by \( (B_1 \ldots B_k)\theta \) where \( \theta \) is the composition of the most literal unifier of \( A \) and \( B_j \), and of the substitution computed during the derivation of \( G_1 \ldots G_m \).

Moreover, \( \theta \) is applied to the other processes \( B_{j+1}, B_{j+2}, \ldots, B_k \) (bindings-sending). The derived goal is then

\[
\leftarrow (B_1 \ldots B_{j+1}, B_{j+2}, \ldots, B_k)\theta
\]
and $\emptyset$ is the answer substitution computed by this derivation step. After a derivation step, the processes in the goal that were suspended may be resumed. Indeed, the bindings-sending can eliminate the causes of suspension. Namely, some input-constraints can become satisfiable. The original goal succeeds, with computed answer substitution $\sigma$, if the empty goal is derived, and $\sigma$ is the composition of all the substitutions computed during the derivation steps.

The input-constraints are the synchronization mechanism of this class of languages. Indeed, a process can be forced to wait until other processes have produced the necessary bindings on the shared variables. In the rest of this section we discuss the various kind of constraints adopted in the specific languages.

2.1. Concurrent Prolog.

In CP the variables in the atoms can be annotated by a symbol "$\hat{}$". In this case, they are called read-only variables. The input-constraints of CP specify that the read-only variables must not get instantiated, neither during the unification with the head of a clause, nor during the evaluation of the guard. It is necessary a definition "ad hoc" for the application of a substitution. The application of a substitution $\sigma$ to a read-only variable $x\hat{}$ gives:

- $x\hat{}$ if $\emptyset$ does not bind $x$;
- $y\hat{}$ if $\emptyset$ binds $x$ to a variable $y$;
- $t$ if $\emptyset$ binds $x$ to a non-variable term $t$.

The application of a substitution to a term extends naturally.

**Example 2.1. Merging Lists.** The following program defines in CP a process that performs the merge on lists $nil$ is the empty list, and $cons(x,y)$ is the list obtained by prefixing a list $y$ with an element $x$.

1) merge($cons(x,y), y', cons(z,y)$) $\leftarrow$ merge($y,y'$, $z$).
2) merge($cons(x,y), cons(z,y)$) $\leftarrow$ merge($y,y'$, $z$).
3) merge($nil,y,y)$ $\leftarrow$ $y$.
4) merge($x,y,nil)$ $\leftarrow$ $y$.

Consider two processes $p$ and $q$ that produce lists of elements $a$'s and $b$'s respectively, as defined by the following clauses:

5) $p(cons(x,y), y', z')$ $\leftarrow$ $y$.
6) $p(nil)$ $\leftarrow$ $z$.
7) $q(cons(x,y), y'$, $z'$) $\leftarrow$ $y$.
8) $q(nil)$ $\leftarrow$ $z$.

Consider the goal:

$\leftarrow$ $p(x,y), q(y,y'), merge(y,y', z)$.

The process merge is forced to wait until some bindings are produced on the variables $y$ or $y'$, namely, all the clauses for merge suspend. Assume that $p$ performs a step by using the clause 3. Then the substitution $\{y\leftarrow cons(a,y), y'\leftarrow cons(a,z)\}$ is computed and the goal becomes

$\leftarrow$ $p(x,y), q(y', y), merge(cons(a,y), y', z)$.

The process merge may now perform a step by using the clause 1 (the computed substitution would be $\{x\leftarrow y, y'\leftarrow cons(a,z)\}$). The clauses 2 and 4 still suspend, while the clause 3 fails. Assume that $q$ first performs a step, by using the clause 7. Then the substitution $\{y\leftarrow cons(b,y), y'\leftarrow cons(b,z)\}$ is computed and the goal becomes

$\leftarrow$ $p(x,y), q(y', y), merge(cons(a,y), cons(b,y), y', z)$.

Now, both the clauses 1 and 2 can be used for the process merge. Assume that the clause 1 is selected. Then, the substitution $\{x\leftarrow y, y\leftarrow cons(b,y), y'\leftarrow cons(b,z)\}$ is computed and the goal becomes

$\leftarrow$ $p(x,y), q(y', y'), merge(cons(b,y), cons(b,z), y', z)$.

Now, only the clause 2 can be used for merge. The clauses 1 and 3 will suspend until a new binding is computed on $y$.

2.2. Guarded Horn Clauses.

The input-constraints of GHC require that the variables of the atom in the goal do not get instantiated during the derivation step (neither by the unification with the head of a clause, nor by the evaluation of the guard). The only exception concerns the unification atoms. They are atoms of the form $\tau = \tau'$ and they can be solved by unifying $\tau$ and $\tau'$. The computed substitution is the map of $\tau$ and $\tau'$. These atoms are the only ones that can produce bindings on the variables of the goal. It turns out that, since the variables in predicates defined by program clauses cannot be instantiated by head unification, the unification atoms cannot be defined in GHC.

**Example 2.2.** The merge processes. The merge processes $p$ and $q$ of example 2.1 can be defined in GHC in the following way:

1) merge($cons(x,y), y', z')$ $\leftarrow$ $y$.
2) merge($y, cons(x,y), z'\leftarrow$ $x'= cons(x,z), merge(x,y', z$).
3) merge($nil,y,y$) $\leftarrow$ $y$.
4) merge($\{x\leftarrow cons(a,y), y'\leftarrow cons(a,z)\}$).
5) $p(y,y', z')$ $\leftarrow$ $y$.
6) $q(\{x\leftarrow cons(a,y), y= cons(a,z), merge(x,y', z)$.
7) $q(\{y\leftarrow cons(b,y), y'\leftarrow cons(b,z)\}$.
8) $q(\{y\leftarrow cons(b,y), y'\leftarrow cons(b,z)\}$.

The initial goal corresponding to the one of example 2.1 is:

$\leftarrow$ $p(x,y), q(y', y), merge(cons(x,y), y', z)$.

Note that the clauses for merge could also be defined in the following way:

1) merge($y,y', z)$ $\leftarrow$ $y'$ $\leftarrow cons(a,y), y' \leftarrow cons(x,z), merge(x,y', z$).

$\leftarrow$ $p(x,y), q(y', y), merge(cons(x,y), y', z)$.