Control Flow versus Logic: a denotational and a declarative model for Guarded Horn Clauses *

Frank S. de Boer[1]
Joost N. Kok[2]
Catuscia Palamidessi[3]
Jan J.M.M. Rutten[1]

[1]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

[2]Department of Computer Science, University of Utrecht,
P.O. Box 80089, 3508 TB Utrecht, The Netherlands

[3]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56100 Pisa, Italy

Abstract

The paper gives four semantic models for Guarded Horn Clauses (GHC). Two operational models are based on a transition system; the first one gives the set of computed answer substitutions (the so-called success set) and the second one takes deadlock and infinite behaviour into account. They are easily related. The main purpose of the paper is to develop compositional models for GHC that are correct with respect to the operational models. For the success set case we give a compositional declarative semantics which can be seen a an extension of models for Horn Clause Logic. Further, a metric semantics that uses tree-like structures is given, which is proved to be correct with respect to the second operational semantics.

1 Introduction

In this paper we consider several models for the concurrent logic language Guarded Horn Clauses (GHC). We have good hope that these models (with some minor changes) are also suitable for other concurrent logic languages like Concurrent Prolog ([Sha83] [Sha87]) and PARLOG ([CG86], [Rin88]). Interesting features of concurrent logic languages include synchronization mechanisms (annotated variables, rules of suspension) and operators that restrict the flow of control (commit). For an introduction to GHC consult [Ued85]. See also [Sar87a] for some remarks about the definition of GHC.

We introduce four models for GHC: Two *operational* models, a *denotational* model and a *declarative* model.

The first operational model gives the results of successfull finite computations, that is the set of computed answer substitutions. A second operational model gives more information: it also deals with deadlock and infinite behaviour. The two operational semantics $\mathcal{O}_1$ and $\mathcal{O}_2$ are based on the same transition relation (in the so-called SOS style ([HP79])). For concurrent logic programming we can find this style of semantics for example in [Sar87a, GCLS88, dBK88].

Although intuitively very clear, these operational models have one drawback: they are not compositional. In this paper, we set out to develop both for $\mathcal{O}_1$ and $\mathcal{O}_2$ a more distinctive model that is compositional and correct with respect to the corresponding operational model. For $\mathcal{O}_2$ we give a denotational model $\mathcal{D}$ that focuses on the *flow of control*, including the deadlock behavior of a GHC program, and for $\mathcal{O}_1$ a declarative model is given in the spirit of *logic* (programming).

In order to define the denotational semantics, which is compositional, we need structures that allow for interleaving and that contain some information about choice points and deadlock. We use tree-like structures labeled by functions that can be annotated. A function that is annotated is used to model the last step in the execution

*Part of this work was carried out in the context of ESPRIT 415: Parallel Architectures and Languages for Advanced Information Processing – a VLSI-directed approach.

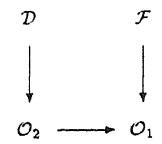

Figure 1: Overview of the models

of a guard, after which we have a new interleaving point. Note that *inside* such a guard computation, which is considered to be atomic, we do not have interleaving. Further we have operators like sequential composition, choice and merge on processes; moreover we have an operator to increase the grain size of a process. These operators allow us to give a compositional denotational semantics. We then show the correctness of the denotational semantics with respect to $\mathcal{O}_2$: there exists an operator *yield* which relates the two models. (Because the two operational models are easily related, $\mathcal{D}$ is also correct with respect to $\mathcal{O}_1$.) The proof of the correctness is rather technical. In the proof we do a step by step analysis of the denotational and operational model: The uniqueness of fixed points is exploited: we show that the operational semantics and the composition of *yield* and the denotational semantics are both fixed points of the same contraction. Other references that follow the 'flow of control' approach are [GCLS88] (for Flat Concurrent Prolog), [DM87] and [JM84]. Our denotational semantics is related to [dBK88], where a compositional semantics is presented for Concurrent Prolog. A major difference is that there the semantics is constructed for an abstract uniform programming language, onto which the language CP is mapped. Here, the semantic models are defined for GHC directly. Further, our semantic universe using annotated functions is slightly simpler than the one used there, which facilitates a more transparant correctness proof.

The compositional model for the $\mathcal{O}_1$ is a declarative semantics, which is more in the style of the traditional semantics for logic languages. References to a declarative style of models include [FL88], [Lev88], [LP85] and [LP87]. A declarative semantics of a program, say in Horn Clause Logic (not to be confused with GHC), is a set of pairs of goals and substitutions, of which the substitution verifies the goal. The situation for concurrent logic languages is more difficult. Given a substitution, we cannot check whether or not all the input mode constraints are satisfied for a certain atom. We also need to check the input mode constraints for those atoms on which it is dependent. We make use of an extended notion of Herbrand base and interpretations, enriched with variables (allowing to model the notion of *computed substitution*, [LP87, FLMP88a, FLPM88b]) and *annotations*. (Annotated variables are implicit in GHC, but are explicit in languages like Concurrent Prolog and PARLOG. They allow to model the synchronization mechanism; see [LP85] and [LP87] for similar approaches.) We extend the standard notions of the unification theory ([Ede85, LMM88]) in a formal framework. In particular, we provide an extended unification algorithm that preserves all the 'nice' properties of the standard one. Moreover, we introduce the notion of *parallel composition*, that allows to formalize the combination (plus consistency check) of the substitutions computed by subgoals run in parallel. Finally, we introduce the notion of *streams of substitutions*, that allows to overcome the difficulties presented in [LP87]. An interpretation is now a set of tuples of the form $< A, z >$, where a is an atom and z is a sequence of substitutions. We give an operator, which is called the immediate consequence operator, that, given some interpretation, gives the set of immediate consequences (i.e. that can be derived with the use of only one clause). In the definition of this operator we use annotated variables to ensure the correct semantics. We prove this declarative semantics to be correct to the first operational model.

Figure 1 gives an overview of the results. An arrow indicates that we have established an abstraction mapping between the two models.

We have omitted many proofs due to lack of space. They can be found in the full paper, of which this is an extended abstract.

Acknowledgement: We are grateful to the members of the Amsterdam Concurrency Group; while discussing a preliminary version of this paper, they detected a serious error.

2 The language GHC

We only give an informal introduction to the language Guarded Horn Clauses (GHC). For a better description we refer to [Ued85] and [Ued86].

Terms and atoms are defined as usual, except for that the set of atoms is extended with so-called unification atoms. An unification atom is of the form $t_1 = t_2$, where t_1 and t_2 are terms. (The intended meaning of $t_1 = t_2$ is that we

have to unify t_1 and t_2.) A GHC program is a finite set of clauses. Each clause is of the form $H \leftarrow \bar{G}|\bar{B}$, where H is a non unification atom and $\bar{G}$ and $\bar{B}$ are finite multisets of atoms. The atom H is called the head, $\bar{G}$ the guard and $\bar{B}$ the body of the clause. The vertical bar | is called the commit operator.

A program is invoked by a goal clause $\leftarrow C$, where C is a multiset of atoms and with current substitution the empty substitution. A goal tries to reduce itself in the following manner. Assume that the current substitution is ϑ. If there is an unification atom $t_1 = t_2$ in the goal we can remove it from the goal if the most general unifier of $t_1\vartheta$ and $t_2\vartheta$ exists. We then also update the current substitution with this unifier. We can remove a non unification atom A if there exists a clause $H \leftarrow \bar{G}|\bar{B}$ (properly renamed) in the program such that

- there exists a most general unifier γ of the atom $A\theta$ and the head of the clause H
- the goal made up of the atoms in $\bar{G}$ can in current substitution γ reduce to $\Box$ (the empty goal) yielding current substitution ϑ'
- such that no variables in the atom $A\vartheta$ are instantiated by $\gamma\vartheta'$.

If all these requirements are satisfied then we can add $\bar{B}$ to the goal (commitment to the clause $H \leftarrow \bar{G}|\bar{B}$) and update the current substitution to $\vartheta\gamma\vartheta'$.

In this description we already made some simplifications to the original semantics of GHC: unification is atomic, we impose an ordering on execution (first head unification followed by the execution of the guard and then the execution of the body) and we use an interleaving model. The second and the third restriction are very common in the literature. It is not difficult to lift the first restriction in the semantic models below.

Another well known restriction is the flatness of guards. A guard is called flat if it only consists of unification atoms and atoms which are made up of clauses with empty bodies. We do not impose this restriction.

3 Operational semantics

For the rest of the paper let W denote a fixed program. We introduce the set of substitutions $(\vartheta, \gamma \in)$ *Subst*; further we have the familiar notion of *mgu*, which is a partial function from pairs of atoms to substitutions. For X a finite set of variables, we use $\vartheta|_X$ to denote the restriction of ϑ to X. Given an atom A and term t the set of variables occurring in A and t we denote by $\mathcal{V}(A)$, $\mathcal{V}(t)$, respectively . The operational semantics will be based on the following *transition relation*:

Definition 3.1 (Transition relation)

Let $\rightarrow \subseteq (Goal \times Subst) \times (Goal \times Subst)$ be the smallest relation satisfying

1. $<\leftarrow A, \vartheta > \rightarrow < \Box, \vartheta' >$
   If $A = t_1 = t_2$ and $\vartheta' = \vartheta mgu(t_1\vartheta, t_2\vartheta)$.
2. $<\leftarrow A, \vartheta > \rightarrow <\leftarrow \bar{B}, \vartheta\vartheta' >$
   whenever $H \leftarrow \bar{G}|\bar{B} \in W$ (properly renamed), $<\leftarrow \bar{G}, mgu(A\vartheta, H) > \stackrel{*}{\rightarrow} < \Box, \vartheta' >$, and $\vartheta'|_{\mathcal{V}(A\vartheta)} = \epsilon$.
3. If $\quad <\leftarrow \bar{A}, \vartheta > \rightarrow <\leftarrow \bar{A}', \vartheta' > \mid < \Box, \vartheta' >$
   then $\quad <\leftarrow \bar{A}, \bar{B}, \vartheta > \rightarrow <\leftarrow \bar{A}', \bar{B}, \vartheta' > \mid <\leftarrow \bar{B}, \vartheta' >$
   $\qquad\quad <\leftarrow \bar{B}, \bar{A}, \vartheta > \rightarrow <\leftarrow \bar{B}, \bar{A}', \vartheta' > \mid <\leftarrow \bar{B}, \vartheta' >$

Here $\stackrel{*}{\rightarrow}$ denotes the transitive closure of the relation $\rightarrow$. In these transitions, ϑ represents the substitution that has been computed until that moment. In 1, it is stated that we can resolve an unification atom by unifying its terms. In 2, it is stated that we can resolve $\leftarrow A$ if we can find a clause in our program with a head H that can be unified with A; moreover, the refutation of the guard $\bar{G}$ of that clause must terminate successfully and must yield a substitution ϑ' that does not instantiate any variables of $A\vartheta$. A conjunction, in 3, is evaluated by the parallel execution of its conjuncts, modeled here by interleaving. In the following definition we give the operational semantics.

Definition 3.2 (Operational semantics)
We define

$$\mathcal{O}_1 : Goal \rightarrow M_1, \text{with } M_1 = \mathcal{P}(Subst), \text{ and } \mathcal{O}_2 : Goal \rightarrow M_2, \text{with } M_2 = \mathcal{P}(Subst_\delta^\infty).$$

(Here $Subst_\delta^\infty = Subst^+ \cup Subst^\omega \cup Subst^* \cdot \{\delta\}$; the symbol δ denotes deadlock.) We put $\mathcal{O}_i[\![\leftarrow true]\!] = \{\epsilon\}$, the identity substitution;

$$\mathcal{O}_1[\![\leftarrow \bar{A}]\!] = \{\vartheta|_{\mathcal{V}(\bar{A})} | \; <\leftarrow \bar{A}, \epsilon > \stackrel{*}{\rightarrow} < \square, \vartheta >\};$$

$$\begin{aligned}\mathcal{O}_2[\![\leftarrow \bar{A}]\!] &= \{(\vartheta_1 \cdots \vartheta_n)|_{\mathcal{V}(\bar{A})} | \; <\leftarrow \bar{A}, \epsilon > \rightarrow <\leftarrow \bar{A}_1, \vartheta_1 > \rightarrow \cdots \rightarrow < \square, \vartheta_n >\} \\ &\cup \{(\vartheta_1 \cdots)|_{\mathcal{V}(\bar{A})} | \; <\leftarrow \bar{A}, \epsilon > \rightarrow <\leftarrow \bar{A}_1, \vartheta_1 > \rightarrow \cdots\} \\ &\cup \{(\vartheta_1 \cdots \vartheta_n)|_{\mathcal{V}(\bar{A})} \cdot \delta | \; <\leftarrow \bar{A}, \epsilon > \rightarrow \cdots \rightarrow < \bar{A}_n, \vartheta_n > \not\rightarrow \wedge \leftarrow \bar{A}_n \neq \square\}.\end{aligned}$$

The *success set* for $\leftarrow \bar{A}$ is given by $\mathcal{O}_1[\![\leftarrow \bar{A}]\!]$: It contains all computed answer substitutions corresponding to all successfully terminating computations. The set $\mathcal{O}_2[\![\leftarrow \bar{A}]\!]$ takes in addition into account some deadlocking and infinite computations, represented by elements of $Subst^* \cdot \{\delta\}$ and $Subst^\omega$, respectively. The relation between $\mathcal{O}_1$ and $\mathcal{O}_2$ is obvious: If we set $last(X) = \{\vartheta | \exists w \in Subst^*(w \cdot \vartheta \in X)\}$ then we have: $\mathcal{O}_1 = last \circ \mathcal{O}_2$.

In the following sections, O_1 and O_2 will be related to a declarative and a denotational semantics, respectively.

We did not include all deadlocking and infinite behaviours in $\mathcal{O}_2$. In fact, we omitted so called local deadlock in guards. This can appear when a local computation in a guard commits to "wrong" clauses. It is not difficult to adapt $\mathcal{O}_2$ and the denotational model below as is shown in [KK89], but we prefer not to do so because it obscures the equivalence proof between $\mathcal{O}_2$ and the denotational model. Moreover, on the version of GHC with flat guards, which is the language that is used in the Japanese fifth generation project, the models coincide.

We end this section by noticing that our operational semantics is *not* compositional. Consider the program

$$\{p(y) \leftarrow r(y)|., \; q(y) \leftarrow |s(y)., \; r(a) \leftarrow |.\}$$

and let $\leftarrow p(x)$ and $\leftarrow q(x)$ be two goals. Operationally, they both yield failure, the former because of the constraint on the variables of the goal and the latter because of the absence of a clause for $s(y)$. However, if we extend both goals with an unification atom $x = a$, thus yielding the goals $\leftarrow p(x), x = a$ and $\leftarrow q(x), x = a$, then we get *different* operational meanings: The first goal will never fail whereas the second one always will.

4 Denotational semantics

The semantic universe M_2 of the operational semantics offers too little structure to define a compositional semantics, as we noticed at the end of the previous section. One of the reasons being that it is not able to distinguish between different kinds of deadlock. A standard solution stemming from the semantic studies of imperative languages is to use tree-like structures. Following [BZ82], we introduce a domain of such structures or a complete metric space satisfying a so-called reflexive domain equation. (We omit the proof of its existence; in [BZ82] and [AR88], it is described how to solve in general domain equations in a metric setting.)

Definition 4.1 The set $(p, q \in) \; P$ is given as the unique complete metric space satisfying

$$P \cong \{p_0\} \cup \mathcal{P}_c(\Gamma \times P).$$

where $\cong$ means "is isometric to" and $\mathcal{P}_c(\Gamma \times P)$ denotes the set of all *compact* subsets of $\Gamma \times P$. Further Γ is given by

$$\begin{aligned}&(\alpha \in) \quad \Gamma = V \cup V^{[\,]}, \text{ with} \\ &(f \in) \quad V = Subst \rightarrow Subst_\delta, \text{ and } V^{[\,]} = \{[f] : \; f \in V\}.\end{aligned}$$

Here $Subst_\delta = Subst \cup \{\delta\}$, and δ is a special element denoting deadlock.

Elements of P are called *processes*. A process p can either be p_0, which stands for termination, or a compact subset $\{< \alpha_i, p_i > : \; i \in I\}$, for some index set I. In that case, p has the choice among the steps $< \alpha_i, p_i >$. Each step consists of some action α_i, which is a state transformation, and a *resumption* p_i of this action, that is, the remaining actions to be taken after this action.

The main difference between P and M_2 is, as was already observed above, the fact that P contains tree-like structures whereas M_2 is a set of (subsets of) streams. In addition, there are two other important differences. First, we use state transforming functions rather that states (substitutions). This functionality is mandatory if we want to define a compositional semantics. Secondly, *internal* steps are visible in P, which is not the case in the operational semantics. For this purpose we distinguish between two kinds of actions: an element $f \in V$ represents an internal computation step, which in the semantics of GHC corresponds to a step in the evaluation of a guard. An action $[f] \in V^{[\,]}$ indicates an external step or to be more precise, the end of an internal computation. (In other words, external steps are modeled as internal computations of length 1.) A typical example of a process is

$$p = \{< f_1, \{< [f_2], \{< [f_3], p_0 >\} >, < f_4, \{< [f_5], p_0, < [f_6], p_0 >\} >\} >\}.$$

We shall use the following semantic operators.

Definition 4.2 We define $;, \|: P \times P \to P$ and $int: P \to P$:

1. $p_0; q = q$, $p; q = \{< \alpha, p'; q > \mid < \alpha, p' > \in p\}$. $; q = \{< \alpha, p'; q > \mid < \alpha, p' > \in p\}$.
2. $p_0 \| q = q \| p_0 = q$,
   $p \| q = p \lfloor\lfloor q \cup q \lfloor\lfloor p$,
   $p \lfloor\lfloor q = \{< \alpha, p' > \lfloor\lfloor q \mid < \alpha, p' > \in p\}$,
   $< f, p' > \lfloor\lfloor q =< f, p' \lfloor\lfloor q >$, $< [f], p' > \lfloor\lfloor q =< [f], p' \| q >$.
3. $\begin{aligned} int(p_0) &= p_0 \\ int(p) &= \{< f, int(p') > \mid (< f, p' > \in p \vee < [f], p' > \in p) \wedge p' \neq p_0\} \\ &\cup \{< [f], p_0 > \mid < f, p_0 > \in p \vee < [f], p_0 > \in p\}. \end{aligned}$

(Notice that these definitions are recursive; they can be given in a formally correct way with the use of contractions.) The definition of ; is straightforward. The parallel merge operator || models the parallel execution of two processes by the interleaving of their respective steps. In determining all possible interleavings, the notions of internal and external steps are crucial; inside an internal computation, no interleaving with other processes is allowed. Only after the last internal step, indicated by the brackets [ ], we have an interleaving point. This explains the definition of the (auxiliary) operator for the *left merge*, which is like the ordinary merge but which always start with a step from the left process: If this step is internal (but not the last step of the internal computation) then we have to continue with a next step of this left process: $< f, p' > \lfloor\lfloor q =< f, p' \lfloor\lfloor q >$. If on the other hand an interleaving point is reached then we switch back to the ordinary merge again: $< [f], p' > \lfloor\lfloor q =< [f], p' \| q >$.

The operator *int* makes a computation *internal* by removing all internal interleaving points.

Now we are ready for the definition of a denotational semantics for GHC. Let W be a fixed program.

Definition 4.3 We define $\mathcal{D} : \mathcal{P}(Var) \to Goal \to P$ as follows:

1. $\mathcal{D}[\![X]\!][\![\leftarrow t_1 = t_2]\!] = \{< [f_{(t_1,t_2,X)}], p_0 >\}$,
   with
   $$f_{(t_1,t_2,X)} = \lambda\vartheta. \begin{cases} \vartheta mgu(t_1\vartheta, t_2\vartheta) & \text{if } mgu(t_1\vartheta, t_2\vartheta) \downarrow \text{ and } mgu(t_1\vartheta, t_2\vartheta)|_{X\vartheta} = \epsilon \\ \delta & \text{otherwise} \end{cases}$$
   (Here $X\vartheta = \bigcup\{\mathcal{V}(\vartheta(x)) : x \in X\}$, and $mgu(t_1\vartheta, t_2\vartheta) \downarrow$ should be interpreted as stating the existence of the most general unifier.)
2. $\mathcal{D}[\![X]\!][\![\leftarrow A]\!] = \bigcup\{int(\{< f_{(A,H,X)}, \mathcal{D}[\![X \cup \mathcal{V}(A)]\!][\![\bar{G}]\!] >\}); \mathcal{D}[\![X]\!][\![\bar{B}]\!] : H \leftarrow \bar{G} | \bar{B} \in W\}$,
   with
   $$f_{(A,H,X)} = \lambda\vartheta. \begin{cases} \vartheta mgu(A\vartheta, H) & \text{if } mgu(A\vartheta, H) \downarrow \text{ and } mgu(A\vartheta, H)|_{X\vartheta \cup \mathcal{V}(A\vartheta)} = \epsilon \\ \delta & \text{otherwise} \end{cases}$$
3. $\mathcal{D}[\![X]\!][\![\leftarrow \bar{A}, \bar{B}]\!] = \mathcal{D}[\![X]\!][\![\leftarrow \bar{A}]\!] \| \mathcal{D}[\![X]\!][\![\leftarrow \bar{B}]\!]$.

(Notice that the definition of $\mathcal{D}$ is recursive; like the semantic operators, it can be given as the fixed point of a contraction.) Both in the clauses 1 and 2, the additional parameter of $\mathcal{D}$, the set of variables X, is used in the condition concerning the resulting new state in the definition of the state-transformation; moreover it is changed in clause 2 from X to $X \cup \mathcal{V}(A)$ because a new guard computation is entered there.

In clause 2 we have further that the computations of the unification and the guard are made internal by an application of the function *int*.

5 Correctness of $\mathcal{D}$ with respect to $\mathcal{O}_2$

We shall relate $\mathcal{O}_2$ and $\mathcal{D}$ via a function $yield - id : P \to M_2$ by showing $\mathcal{O}_2 = yield - id \circ \mathcal{D}$. This implies the correctness of $\mathcal{D}$ with respect to $\mathcal{O}_2$, that is, the fact that $\mathcal{D}$ makes at least the same distinctions that $\mathcal{O}_2$ makes. It appears technically convenient to turn M_2, the semantic universe of $\mathcal{O}_2$, into a complete metric space.

Definition 5.1 We define $M_2 = \mathcal{P}_{cl}(Subst_\delta^\infty)$, where $\mathcal{P}_{cl}$ denotes the set of all *closed* subsets. The set M_2 is a complete metric space if we supply it with the Hausdorff metric induced by the usual metric on $Subst_\delta^\infty$.

Next we define a function *yield* as follows:

Definition 5.2 Let the function $yield : P \to Subst \to M_2$ be given by

$$\begin{array}{ll} yield(p_0)(\vartheta) & = \{\vartheta\} \\ yield(p)(\vartheta) & = \bigcup_\delta \{\vartheta' \cdot yield(p_n)(\vartheta') : \ < f_1, p_1 > \in p \wedge \cdots \wedge < f_{n-1}, p_{n-1} > \in p_{n-2} \wedge \\ & \qquad\qquad < [f_n], p_n > \in p_{n-1} \wedge (f_n \circ \cdots \circ f_1)(\vartheta) = \vartheta'\} \end{array}$$

(The attentive reader might observe that the function *yield* is not well defined, because in general $yield(p)(\vartheta)$ is not closed. He is right. Fortunately, however, we are saved by the observation that the restriction of *yield* to the set $\{p : \exists \bar{A}, X (p = \mathcal{D}[\![X]\!][\![\leftarrow \bar{A}]\!])\}$ always delivers closed sets. This turns out to be everything we need.)

The function *yield* performs four abstractions at the same time. First, it turns a process (a tree-like structure) into a set of streams; secondly, it computes for every state transformation a new state (given some initial state), which is passed through to a next state transformation in the process; moreover, it performs the function composition of all functions occurring in a sequence $f_1, \ldots, f_n$ that is derived from a finite path in p like

$$< f_1, p_1 >, \ldots, < f_{n-1}, p_{n-1} >, < [f_n], p_n > .$$

Such a sequence represents an internal computation, the end of which is indicated by $[f_n]$. If we apply the resulting composition to a state ϑ then we obtain a new state ϑ' of which the substitution ϑ' is passed through to the recursive application of the function *yield*. Finally, the function *yield* removes all infinite internal computations.

A final technical comment on this definition of the function *yield* concerns the use of the operation $\bigcup_\delta$; it is defined by

$$\begin{array}{lll} \bigcup_\delta X & = \bigcup X \setminus \{\delta\} & \text{if } \bigcup X \setminus \{\delta\} \neq \emptyset \\ & = \{\delta\} & \text{otherwise.} \end{array}$$

The main result of this section is

$$\mathcal{O}_2 = yield - id \circ \mathcal{D}[\![\emptyset]\!],$$

where $yield - id : P \to M_2$ is given by $yield - id(p) = yield(p)(\epsilon)$.

The proof is rather technical and is omitted due to lack of space. It has the following structure: First we introduce an intermediate syntax *IS* such that $Goal \subseteq IS$; next we extend the definitions of $\mathcal{O}_2$ and $\mathcal{D}$ to $\mathcal{O}' : IS \to Subst \to M_2$ and $\mathcal{D}' : IS \to P$ such that $\mathcal{O}_2 = \mathcal{O}'|Goal$ (the restriction of $\mathcal{O}'$ to the set *Goal*) and $\mathcal{D}[\![\emptyset]\!] = \mathcal{D}'|Goal$; finally, we prove $\mathcal{O}' = yield \circ \mathcal{D}'$, from which the result follows. In *IS*, internal computation steps are represented explicitly; this will enable us to prove $\mathcal{O}' = yield \circ \mathcal{D}'$.

6 Declarative semantics

In this section we define the declarative semantics of GHC. In order to model the synchronization mechanism of GHC we introduce the notion of *annotated variable*. The annotation can occur on a variable in the goal, and it represents the input-mode constraint. Namely, such a variable can get bound by the execution of other atoms in the goals, but not by the execution of the atom in which it occurs (before commitment).

We will denote the set of variables, with typical elements $x, y, \ldots$, by *Var*, and the set of the annotated variables, with typical elements $x^-, y^-, \ldots$, by Var^-. ¿From a mathematical point of view, we can consider "$^-$" as a bijective mapping $^- : Var \to Var^-$. The elements of $Var \cup Var^-$ will be represented by $v, w, \ldots$. The set of terms *Term*, with typical element t, is extended on $Var \cup Var^-$. t^- is the term obtained by replacing in t every variable $x \in Var$ by x^-.

The notion of substitution extends naturally to the new set of variables and terms. Namely, a substitution ϑ is a mapping $\vartheta : Var \cup Var^- \to Term$, such that $\vartheta(v) \neq v$ for finitely many v only. ϑ will be represented by the set $\{v/t \mid v \in Var \cup Var^- \wedge \vartheta(v) = t \neq v\}$. The application of a substitution ϑ to a variable is defined by

$$x\vartheta = \vartheta(x)$$
$$x^-\vartheta = \vartheta(x^-) \quad \text{if } \vartheta(x^-) \neq x^-$$
$$x^-\vartheta = \vartheta(x)^- \quad \text{if } \vartheta(x^-) = x^-$$

The new notion of application differs from the standard one in that $\{v \in Var \cup Var^- \mid \vartheta(v) \neq v\}$ (the set of variables mapped by ϑ to a different term) is now a subset of $\{v \in Var \cup Var^- \mid v\vartheta \neq v\}$ (the set of variables bound by ϑ to a different term). An annotated variable mapped to a different term represents a violation of the associated input-mode constraint. An annotated variable bound to a different term represents the ability to receive a binding from the computation of another atom in the goal. The application of ϑ to a term (or atom, or formula) t is defined by $t\vartheta = v\vartheta$ if $t = v \in Var \cup Var^-$, and $t\vartheta = f(t_1\vartheta, \ldots, t_n\vartheta)$ if $t = f(t_1, \ldots, t_n)$ We factorize the set of substitutions with respect to the equivalence relation $\vartheta_1 \equiv \vartheta_2$ iff $\forall v \in Var \cup Var^-[v\vartheta_1 = v\vartheta_2]$. ¿From now on, a substitution ϑ will indicate its equivalence class.

The notion of composition $\vartheta_1\vartheta_2$, of two substitutions, ϑ_1 and ϑ_2 is extended as follows

$$\forall v \in Var \cup Var^-[v(\vartheta_1\vartheta_2) = (v\vartheta_1)\vartheta_2].$$

The composition is associative and the empty substitution ϵ is the neutral element. Given a set of sets of terms M, we define ϑ to be a unifier for M iff

$$\forall S \in M \forall t_1, t_2 \in S\ [t_1\vartheta = t_2\vartheta \text{ and } t_1^-\vartheta = t_2^-\vartheta].$$

The ordering on substitutions is the standard one, namely: $\vartheta_1 \leq \vartheta_2$ *iff* $\exists\vartheta_3\ [\vartheta_1\vartheta_3 = \vartheta_2]$ (ϑ_1 *is more general than* ϑ_2). The set of mgu's (most general unifiers) of a set of sets of terms M is denoted by $mgu(M)$. The unification algorithm can be extended, without modifying its *structure*, in order to deal with the new notion of application of a substitution to a term.

We need now an operation for combining the substitutions obtained by running in parallel two different atoms in the same goal. This operation can be performed in the following way: Consider the set of all the pairs corresponding to the bindings of both the substitutions. Then, compute the most general unifier of such a set. Note that the *consistency check* corresponds to a verification that such a set is unifiable. We will call this operation *parallel composition.*

Definition 6.1 Let $\mathcal{S}(\vartheta)$ denote the set of sets $\{\{v, t\} \mid v/t \in \vartheta\}$. We define

$$\vartheta_1 \,\hat{\circ}\, \vartheta_2 = mgu(\mathcal{S}(\vartheta_1) \cup \mathcal{S}(\vartheta_2)).$$

Moreover, for Θ_1, Θ_2 sets of substitutions, we define $\Theta_1 \,\hat{\circ}\, \Theta_2 = \bigcup_{\vartheta_1 \in \Theta_1, \vartheta_2 \in \Theta_2} \vartheta_1 \,\hat{\circ}\, \vartheta_2$. We will denote the sets $\{\vartheta\} \,\hat{\circ}\, \Theta$ and $\Theta \,\hat{\circ}\, \{\vartheta\}$ by $\vartheta \,\hat{\circ}\, \Theta$ and $\Theta \,\hat{\circ}\, \vartheta$ respectively.

We introduce now the notion of *sequence of substitutions.* We need it because the standard *flat* representation of the computed bindings (obtained by composing all the substitutions associated to the derivation steps), is not powerful enough to model the *effects* of the possible interleavings in the executions of the atoms in a goal. See [LP85], [LP87] and [Le2] for a discussion of this problem. Since we model declaratively the success set only, we need to consider only finite sequences.

Definition 6.2 The finite sequences of substitutions, with typical element z, are defined by the following (abstract) syntax

$$z ::= \vartheta \mid [z]_{\mathcal{V}} \mid z_1.z_2$$

The role of the squared brackets is to delimitate the *critical sections.* $\mathcal{V}$ represents a set of variables, whose annotation has to be removed when computing the result of a sequence of substitutions. Their meaning will be clarified by the definition of the *interleaving operator* and *result operator.* We introduce the following notations. If Z and Z' are sets of sequences, then $Z.Z' \stackrel{def}{=} \{z.z' \mid z \in Z, z' \in Z'\}$ and $[Z]_{\mathcal{V}} \stackrel{def}{=} \{[z]_{\mathcal{V}} \mid z \in Z\}$. If $z = \vartheta'.z'$, then $\vartheta \,\hat{\circ}\, z \stackrel{def}{=} (\vartheta \,\hat{\circ}\, \vartheta').z'$ and $\vartheta \,\hat{\circ}\, ([z]_{\mathcal{V}}.z'') \stackrel{def}{=} [(\vartheta \,\hat{\circ}\, \vartheta').z']_{\mathcal{V}}.z''$. For Θ a set of substitution we have $\Theta \,\hat{\circ}\, z \stackrel{def}{=} \bigcup_{\vartheta \in \Theta} \vartheta \,\hat{\circ}\, z$.

Definition 6.3 (Interleaving operator).

1. $z_1 \parallel z_2 = z_1 \between z_2 \cup z_2 \between z_1$
   $(\vartheta.z_1) \between z_2 = \vartheta.(z_1 \parallel z_2)$
   $([z]_{\mathcal{V}}.z_1) \between z_2 = [z]_{\mathcal{V}}.(z_1 \parallel z_2)$

2. $Z_1 \parallel Z_2 = \bigcup_{z_1 \in Z_1, z_2 \in Z_2} z_1 \parallel z_2.$

Since the interleaving operator is associative we can omit parentheses. We note that the definition of the operator $\parallel$ is similar to the one given in definition 4.2, but it works on different structures (sequences instead of trees).

The following definition introduces the notion of *result* $\mathcal{R}$ of a sequence z (or a set of sequences Z) of substitutions. Roughly, such a result is obtained by performing the parallel composition of each element of the sequence with the next one, and by checking, each time, that the partial result does not violate input-mode constraints.

Definition 6.4

1. $\mathcal{R}(\vartheta) = \begin{cases} \{\vartheta\} & \text{if } \vartheta_{|Var^-} = \epsilon \\ \emptyset & \text{otherwise} \end{cases}$
2. $\mathcal{R}([z]_{\mathcal{V}}) = disann_{\mathcal{V}}(\mathcal{R}(z))$
3. $\mathcal{R}(z_1.z_2) = \mathcal{R}(\mathcal{R}(z_1) \circ z_2)$

where $disann_{\mathcal{V}}(z)$ removes all the annotations of the variables of $\mathcal{V}$ which occur in z. Thus, rule 2. specifies that, after a critical section, the input-constraints are released. Rule 1 checks that ϑ (to be intended as the partial result) does not map annotated variables. Rule 3 specifies the order of evaluation of a sequence: from left to right. Indeed, we have $\mathcal{R}(\vartheta_1.\vartheta_2.\ldots.\vartheta_n) = \mathcal{R}(\ldots\mathcal{R}(\mathcal{R}(\vartheta_1) \circ \vartheta_2)\ldots \circ \vartheta_n)$.

For Z a set of sequences we define $\mathcal{R}(Z) = \bigcup_{z \in Z} \mathcal{R}(z)$.

Next we introduce the notion of interpretation, and a continuous mapping (associated to the program) on interpretations, whose least fixed point will be used to define the declarative semantics. Such a mapping is the extension of the *immediate consequence* operator (see [Apt87]), firstly introduced by van Emden and Kowalski [vEK76]. First we recall some basic notions. (Consult also the appendix with basic notions.) Given a program W, the *Herbrand base with variables* $\mathcal{B}_W$ associated to the program is the set of all the possible atoms that can be obtained by applying the predicates of W to elements of *Term*. *Term* consists of terms built of $Var \cup Var^-$ and of constructors of W.

Definition 6.5 An interpretation of W is a set of pairs of the form $< A, z >$, where A is an atom in $\mathcal{B}_W$ and z is a sequence of substitutions on $Var \cup Var^-$ and *Term*. $\mathcal{I}_W$ will denote the set of all the interpretations of W.

$\mathcal{I}_W$ is a complete lattice with respect to the set-inclusion, with $\emptyset$ as the minimum element, and the *set union* and *set intersection* as the *sup* and *inf* operations, respectively.

The following definition, that will be used in the least fixed point construction, is mainly introduced for technical reasons.

Definition 6.6 Let $z_1, \ldots, z_h$ be sequences of substitutions, and let $A_1, \ldots, A_k$ $(h \leq k)$ be atoms. The sequences $z_1, \ldots, z_h$ are said to be *locally independent* on $A_1, \ldots, A_k$ if and only if

$$\forall i \in \{1, \ldots, h\} \forall \vartheta \in z_i [(\mathcal{D}(\vartheta) \cup \mathcal{C}(\vartheta)) \cap \mathcal{V}(A_1 \wedge \ldots \wedge A_k) \subseteq \mathcal{V}(A_i)].$$

where $\mathcal{D}(\vartheta)$ and $\mathcal{C}(\vartheta)$ are the standard *domain* and *codomain* of ϑ, and $\mathcal{V}(F)$ denotes the set of variables of the formula F.

If X is a set of variables, then W_X will denote all the possible *variants* of W with respect to X, i.e. the programs whose clauses are variants, with respect to X, of the clauses of W. We give now the definition of our *immediate consequence* operator.

Definition 6.7 The mapping $T_W : \mathcal{I}_W \rightarrow \mathcal{I}_W$, associated to a program W, is defined as follows:

$$T_W(I) = \{< A, z > \mid \exists A' \leftarrow A_1 \wedge \ldots \wedge A_n | A_{n+1} \wedge \ldots \wedge A_m \in W_{\mathcal{V}(A)} \quad \exists z_1, \ldots, z_m \textit{ locally independent on } A, A_1, \ldots, A_m \quad [$$

$$\{< A_1, z_1 >, \ldots, < A_m, z_m >\} \subset I \wedge$$
$$z \in [mgu(A^-, A').(z_1 \parallel \ldots \parallel z_n)]_{\mathcal{V}}.(z_{n+1} \parallel \ldots \parallel z_m)$$
$$]$$
$$\}$$

$$\cup \quad \{< A, \vartheta > \mid \exists A' \in \{x = x\}_{\mathcal{V}(A)} : \vartheta \in mgu(A, A')\}$$

In this definition $\mathcal{V}$ stands for $\mathcal{V}(A, A', z_1, \ldots, z_n)$. If A is not a *unification atom,* then a possible sequence for A results from the critical section containing the *mgu* with the head of a clause, and a sequence resulting from the guard. The variables in A are annotated. The whole is followed by a sequence resulting from the body. If A is a *unification atom,* say $t_1 = t_2$, then the sequence contains only the *mgu* with an atom of the form $x = x$ (or, equivalently, the *mgu* of t_1 and t_2).

Proposition 6.8 T_W is continuous.

Corollary 6.9 The least fixed point $lfp(T_W)$ of T_W exists, and $lfp(T_W) = \bigcup_{n \geq 0} T_W^n(\emptyset)$ holds.

We define now the least fixed point semantics associated to a program W.

Definition 6.10 The least fixed point semantics $\mathcal{F}$ of a program W is the set

$$\mathcal{F}(W) = \{< A_1 \wedge \ldots \wedge A_n, \vartheta > : \exists z_1, \ldots, z_n \textit{ locally independent on } A_1, \ldots, A_n \quad [$$

$$< A_1, z_1 >, \ldots, < A_n, z_n > \in lfp(T_W)$$
$$\vartheta \in (\mathcal{R}(z_1 \parallel \ldots \parallel z_n))_{|\mathcal{V}(A_1, \ldots, A_n)}$$
$$]$$
$$\}.$$

We are able to show that the informal operational semantics (as it is given in Guarded Horn Clauses section) is sound and complete with respect to the declarative semantics $\mathcal{F}$: $\mathcal{F}(W) = \mathcal{O}_1(W)$ for any GHC program W. For the proof (which is omitted here for reasons of space) we refer to [Pal88].

The following example illustrates the necessity to use sequences of substitutions. Similar examples have been given in [LP85], [Lev88] and [Le2] to prove that a flat representation of the computed bindings (as given in [LP85] and [LP87]) is not adequate to deal with the cases of *deadlock* (it does not allow to distinguish between the two programs below).

Example

1. Consider the program $\{p(a, w_1) \leftarrow | w_1 = b.,\ q(w_2, b) \leftarrow | w_2 = a.\}$, and consider the goal $\leftarrow p(x, y), q(x, y)$. We have $< p(x, y), z_1 >, < q(x, y), z_2 > \in lfp(T_W)$, for $z_1 = [\{x^-/a, w_1/y^-\}]_{\{x,y\}}.\{w_1/b\}$ and $z_2 = [\{y^-/b, w_2/x^-\}]_{\{x,y\}}.\{w_2/a\}$.

   For all the possible interleavings $z \in z_1 \parallel z_2$, we get $\mathcal{R}(z) = \emptyset$. Indeed, no refutations are possible (*deadlock*).

2. Consider now the program $\{p(w_1, w_3) \leftarrow | r(w_1), w_3 = b.,\ r(a) \leftarrow |.,\ q(w_2, b) \leftarrow | w_2 = a.\}$. We have $< p(x, y), z_1 >, < q(x, y), z_2 > \in lfp(T_W)$, for $z_1 = [\{w_1/x^-, w_3/y^-\}]_{\{x,y\}}.\{w_3/b\}.[\{w_1^-/a\}]\{w_1\}$ and $z_2 = [\{y^-/b, w_2/x^-\}]_{\{x,y\}}.\{w_2/a\}$. We have $z = [\{w_1/x^-, w_3/y^-\}]_{\{x,y\}}.\{w_2/b\}.[\{y^-/b, w_2/x^-\}]_{\{x,y\}}.\{w_2/a\}.[\{w_1^-/a\}]_{\{w_1\}} \in z_1 \parallel z_2$. Now, we observe that $\{x/a, y/b, w_1/a, w_3/b, w_2/a\} \in \mathcal{R}(z)$. Indeed, there exists a refutation of the goal $\leftarrow p(x, y), q(x, y)$ giving the answer $\{x/a, y/b\}$.

References

[Apt87] K.R. Apt. *Introduction to logic programming.* Technical Report CS-R8741, Centre for Mathematics and Computer Science, Amsterdam, 1987. To appear as a chapter in Handbook of Theoretical Computer Science, North-Holland.

[AR88] P. America and J.J.M.M. Rutten. *Solving reflexive domain equations in a category of complete metric spaces.* Proc. of the third workshop on mathematical foundations of programming language semantics, Lecture notes in Computer Science 298, 1988, pp. 254-288.

[dBK88] J.W. de Bakker and J.N. Kok. Uniform abstraction, atomicity and contractions in the comparative semantics of concurrent prolog. In *Proc. Fifth Generation Computer Systems (FGCS 88)*, pages 347–355, Tokyo, Japan, 1988. Extended Abstract, full version available as CWI report CS-8834 and to appear in Theoretical Computer Science.

[BZ82] J.W. de Bakker and J.I. Zucker. *Processes and the denotational semantics of concurrency.* Inform. and Control 54, 1982, pp. 70-120.

[CG86] K.L. Clark, S. Gregory, *PARLOG: Parallel programming in logic,* ACM Trans. Program. Lang. Syst. Vol. 8, 1, 1986, 1-49. Res. Report DOC 84/4, Dept. of Computing, Imperial College, London,1984.

[DM87] S.K. Debray and P. Mishra. Denotational and operational semantics for prolog. In M. Wirsing, editor, *Formal Description of Programming Concepts III,* pages 245–269, North-Holland, 1987.

[Ede85] E. Eder. *Properties of substitutions and unifications.* Journal Symbolic Computation 1, 1985, pp. 31-46.

[vEK76] M.H. van Emden and R.A. Kowalski. *The semantics of predicate logic as a programming language.* Journal of the ACM 23(4), 1976, 733-742.

[FL88] M. Falaschi, G. Levi, *Finite Failures and Partial Computations in Concurrent Logic Languages,* Proc. of the FGCS'88.

[FLMP88a] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. *Declarative modeling of the operational behaviour of logic languages.* Theoretical Computer Science, 1988. To appear.

[FLPM88b] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. *A new declarative semantics for logic languages.* In Proceedings Conference and Symposium on Logic Programming, MIT press, 1988, pp. 993-1005.

[GCLS88] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. *Fully abstract denotational semantics for concurrent prolog.* In Proc. Logic In Computer Science, 1988, pp. 320-335.

[Gre87] S. Gregory. *Parallel logic programming in PARLOG.* International Series in Logic Programming, Addison-Wesley, 1987.

[HP79] M. Hennessy and G.D. Plotkin. *Full abstraction for a simple parallel programming language.* In J. Becvar, editor, Proceedings 8th MFCS, Lecture Notes in Computer Science 74, Springer Verlag, 1979, pp. 108-120.

[JM84] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for prolog. In *Proc. 1984 Int. Symp. on Logic Programming,* 1984.

[KK89] P. Knijnenburg and J.N. Kok. A compositional semantics for the finite failures of a language with atomized statements. Technical report, University of Utrecht, 1989.

[KR88] J.N. Kok and J.J.M.M. Rutten. *Contractions in comparing concurrency semantics.* In Proceedings 15th ICALP, Tampere, Lecture Notes in Computer Science 317, Springer Verlag, 1988, 317-332. To appear in Theoretical Computer Science

[Lev88] G. Levi. *A new declarative semantics of flat guarded horn clauses.* Technical Report, ICOT, Tokyo, 1988.

[Le2] G. Levi. *Models, unfolding rules and fixed point semantics.* Proc. Symp. on Logic Programming, 1988, pp. 1649-1665.

[LMM88] J.-L. Lassez, M.J. Maher, and K. Marriot. *Unification revisited.* In J. Minker, editor, Foundations of deductive databases and logic programming, Morgan Kaufmann, Los Altos, 1988.

[LP85] G. Levi and C. Palamidessi. *The declarative semantics of logical read-only variables.* In Proc. Symp. on Logic Programming, IEEE Comp. Society Press, 1985, pp. 128- 137.

[LP87] G. Levi and C. Palamidessi. *An approach to the declarative semantics of synchronization in logic languages.* In Proc. 4th Int. Conference on Logic Programming, 1987, 877-893.

[Pal88] C. Palamidessi. *A fixpoint semantics for Guarded Horn Clauses.* Technical Report CS-R8833, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[Rin88] G.A. Ringwood. Parlog 86 and the dining logicians. *Comm. ACM*, 31:10–25, 1988.

[Sar87a] V.A. Saraswat: *The concurrent logic programming language CP: definition and operational semantics*, in: Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, West Germany, January 21-23, 1987, pp. 49-62.

[Sha83] E.Y. Shapiro. *A subset of concurrent prolog and its interpreter.* Technical Report TR-003, ICOT, Tokyo, 1983.

[Sha87] E.Y. Shapiro. Concurrent prolog, a progress report. In W. Bibel and Ph. Jorrand, editors, *Fundamentals of Artificial Intelligence*, Springer Verlag, 1987. Lecture Notes in Computer Science 232.

[Sh83] E.Y. Shapiro. *A subset of concurrent prolog and its interpreter.* Tech. Report TR-003, ICOT, Tokyo, 1983.

[Ued85] K. Ueda. *Guarded Horn Clauses.* Technical Report TR-103, ICOT, 1985. Revised in 1986. A revised version is in E. Wada, editor, *Proceedings Logic Programming*, pages 168–179, Springer Verlag, 1986. LNCS 221.

[Ued86] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* Technical Report TR-208, ICOT, 1986. Revised in 1987. Also to appear in M. Nivat and K. Fuchi, editors, *Programming of Future Generation Computers*, North Holland, 1988.

7 Appendix: Basic Notation

We will use mainly the same terminology and notations of [Apt87], [LMM88], and [Ede85] to which the reader is referred also for the main properties about substitutions and unification.

Let *Var* be a set of variables, with typical elements $x, \ldots$. Let *Term* be a set of terms, with typical elements $t, \ldots$, built on *Var* and on a given sets of constructors. A substitution ϑ is a mapping from *Var* into *Term* such that $\mathcal{D}(\vartheta) = \{x \in Var : \vartheta(x) \neq x\}$ is finite. $\mathcal{D}(\vartheta)$ is called the domain of the substitution ϑ. We will use also the set-theoretic notation for ϑ: $\vartheta = \{x/t \mid x \in \mathcal{D}(\vartheta), \vartheta(x) = t\}$. Let F be a expression (term, atom or clause). The set of variables occurring in F is denoted by $\mathcal{V}(F)$. The application $F\vartheta$ of ϑ to F is defined as the expression obtained by replacing each variable x in F by $\vartheta(x)$. $\mathcal{C}(\vartheta)$ (which we will improperly call the co-domain, or range of ϑ) is the set $\bigcup_{x \in \mathcal{D}(\vartheta)} \mathcal{V}(\vartheta(x))$. A renaming ρ is any bijective substitution from *Var* to *Var*. If X is a set of variables, then $F\rho$ is a variant of a expression F with respect to X iff ρ is a renaming and $\mathcal{V}(F\rho) \cap V = \emptyset$. $F\rho$ is said to be a variant of F iff $F\rho$ is a variant of F with respect to $\mathcal{V}(F)$. The composition $\vartheta\vartheta'$ of two substitutions ϑ and ϑ' is defined in the usual way, namely $(\vartheta\vartheta')(x) = (\vartheta(x))\vartheta'$. We recall that the composition is associative, the empty substitution ϵ is the neutral element, and for each renaming ρ there exists the inverse ρ^{-1}, i.e. $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. Moreover, $F(\vartheta\vartheta') = (F\vartheta)\vartheta'$. ϑ is called idempotent iff $\vartheta\vartheta = \vartheta$ (or, equivalently, iff $\mathcal{D}(\vartheta) \cap \mathcal{C}(\vartheta) = \emptyset$). The pre-order relation $\leq$ on substitutions is defined by: $\vartheta \leq \vartheta' \Leftrightarrow \exists\vartheta_1[\vartheta\vartheta_1 = \vartheta']$. The restriction $\vartheta_{|X}$ of ϑ to a set of variables X is the substitution $\vartheta_{|X}(x) =\cdot \vartheta(x)$ for $x \in X$ and $\vartheta_{|X}(x) = x$ otherwise.

Given a set of sets of terms M, a substitution ϑ is a unifier of M iff $\forall S \in M \forall t, t' \in S[t\vartheta = t'\vartheta]$ holds. ϑ is a *most general unifier* (*mgu*) of M if it is a unifier of M and $\vartheta \leq \vartheta'$ for any other unifier ϑ' of M.

8 Appendix: Extended unification algorithm

In this appendix We give an extended version of the unification algorithm, based on the one presented in [Apt87], that works on finite sets of pairs. Given a finite set of finite sets of terms M, consider the (finite) set of pairs

$$M_{pairs} = \bigcup_{S \in M} \{< t, u > \mid t, u \in S\}.$$

The unifiers of a set $\{< t_1, u_1 >, \ldots, < t_n, u_n >\}$ are the ones of $\{\{t_1, u_1\}, \ldots, \{t_n, u_n\}\}$. Of course, M and M_{pairs} are equivalent (i.e. they have the same unifiers). A set of pairs is called *solved* if it is of the form

$$\{< x_1, t_1 >, \ldots, < x_n, t_n >\}$$

where all the x_i's are distinct elements of $Var \cup Var^-$, $x_i \notin \mathcal{V}(t_1, \ldots, t_n)$, and, if $x_i \in Var$ and $t_i \neq x_i^-$, then $x_i^- \notin \mathcal{V}(x_1, \ldots, x_n, t_1, \ldots, t_n)$. For P solved, define $\gamma_P = \{x_1/t_1, \ldots, x_n/t_n\}$, and $\delta_P = \gamma_P \gamma_P$.

The following algorithm transforms a set of pairs into an equivalent one which is solved, or halts with failure if the set has no unifiers.

Definition 8.1 (Extended unification algorithm)

- Let P, P' be sets of pairs. Define $P \Rightarrow P'$ if P' is obtained from P by choosing in P a pair of the form below and by performing the corresponding action

1. $< f(t_1, ..., t_n), f(u_1, ..., u_n) >$		replace by the pairs $< t_1, u_1 >, ..., < t_n, u_n >$
2. $< f(t_1, ..., t_n), g(u_1, ..., u_n) >$, where $f \neq g$		halt with failure
3. $< x, x >$	where $x \in Var \cup Var^-$	delete the pair
4. $< t, x >$	where $x \in Var \cup Var^-, t \notin Var \cup Var^-$	replace by the pair $< x, t >$
5. $< x, t >$	where $x \in Var, x \neq t$, $x^- \neq t$ and x or x^- occurs in other pairs	if $x \in \mathcal{V}(t)$ or $x^- \in \mathcal{V}(t)$ then halt with failure else apply the substitution $\{x/t\}$ to all the other pairs
6. $< x, x^- >$	where $x \in Var$, and x occurs in other pairs	apply the substitution $\{x/x^-\}$ to all the other pairs
7. $< x^-, t >$	where $x^- \in Var^-, x^- \neq t$ and x^- occurs in other pairs	if $x^- \in \mathcal{V}(t)$ then halt with failure else apply the substitution $\{x^-/t\}$ to all the other pairs.

  We will write $P \Rightarrow \mathit{fail}$ if a failure is detected (steps 2, 5 or 7).

- Let $\Rightarrow^*$ be the reflexive-transitive closure of the relation $\Rightarrow$, and let P_{sol} be the set $P_{sol} = \{P' \mid \mathit{symm}(P) \Rightarrow^* P'$, and P' is solved$\}$, where $\mathit{symm}(\{< t_1, u_1 >, \ldots, < t_n, u_n >\}) = \{< t_1, u_1 >, \ldots, < t_n, u_n >\} \cup \{< t_1^-, u_1^- >, \ldots, < t_n^-, u_n^- >\}$.
  The set of substitutions determined by the algorithm is $\Delta(P) = \{\delta_{P'} \mid P' \in P_{sol}\}$.

The following proposition shows that the set of the idempotent most general unifiers of M is finite and can be computed in finite time by the extended unification algorithm.

Proposition 8.2 Let P be a finite set of pairs, and M be a finite set of finite sets of terms.

1. **(finiteness)** The relation $\Rightarrow$ is *finitely-branching* and *noetherian* (i.e. *terminating*).
2. **(solved form)** If P is in *normal form* (i.e. there exist no P' such that $P \Rightarrow P'$), then P is in solved form.
3. **(soundness)** $\Delta(P) \subseteq mgu(P)$
4. **(completeness)** $mgu(M) \subseteq \Delta(M_{pairs})$.
5. $P \Rightarrow^* \mathit{fail}$ iff P is not unifiable.

This result implies that the set of the idempotent most general unifiers of M is finite and can be computed in finite time by a deterministic simulation of the extended unification algorithm (the non-determinism of the relation $\rightarrow$ can be simulated via a simple backtracking).