A Concurrent Extension of Functional Logic Programming Languages*

draft of December 16, 1999

Rachid Echahed and Wendelin Serwe

LEIBNIZ–IMAG / CNRS, 46, av. F. Viallet, 38031 Grenoble, FRANCE
Rachid.Echahed@imag.fr Wendelin.Serwe@imag.fr

Abstract. We present a concurrent extension of functional logic programming languages together with a compositional semantics based on labelled sequences, which takes into account the environment of the program. This framework allows to specify, at a very high level, applications that need concurrency and interaction with the environment. For that, we introduce the possibility of defining processes (agents) which specify the dynamics (evolution) of a classical functional logic program, including its communication with the environment. The resulting formalism integrates in a uniform way the main features of functional, logic and concurrent programming.

1 Introduction

The aim of a functional logic programming language is to combine, in a uniform way, the main advantages of both functional programming (e.g., efficient reduction strategies, higher order facilities etc.) and (constraint) logic programming (e.g., goal solving, computation with partial information (logical variables) etc.). The integration of functional and logic programming languages has been launched by Robinson and Sibert in [25]. Since then, a plethora of such declarative languages has been proposed, see [14] for a survey, and [16, 20] for recent contributions.

However, pure declarative languages (either functional or logic) fail to provide the possibility of high-level description of concurrent real-world applications; by concurrent applications we mean programs describing several processes which may be executed concurrently and cooperate together to achieve their (different) specific tasks (via communication or sharing critical resources), as for example an application involving several UNIX processes.

Thus, many concurrent extensions of declarative languages have been proposed. However, these extensions *encode* processes, whereas we consider processes as *basic concepts* in the same way as functions or predicates.

The main contributions of this paper can be summarised as follows:

* This work has been partially supported by the PROCOPE programme under grant number 99093.

– We propose a framework of concurrent functional logic programming languages where concurrency is expressed by means of processes (or agents). These processes are described in process algebra style, whereas functions and predicates are defined in classical functional logic formalisms.
– Our operational semantics distinguishes clearly between the *implicit* parallelism that may be used to evaluate functional logic expressions (e.g., parallel evaluation of function arguments or different kinds of and-parallelism, as for instance [21, 23]) from concurrency which is *explicitly* specified by a programmer to meet the intrinsic concurrency of a system (e.g., UNIX processes).
– Our framework provides some elementary actions which allow programs to interact with their environments. We found these primitives very useful in some realistic case studies.
– We give a semantics for terminating or nonterminating programs. This semantics is compositional and takes into account the possible interactions of a program with its environment (which is not part of the program itself).

This paper reviews and extends a previous version [5] and provides a compositional semantics.

The rest of this paper is organised as follows: In the next section, we introduce an abstract syntax for a concurrent extension of functional logic languages. In Sect. 3, we describe an operational semantics of the proposed languages. A compositional semantics based on labelled sequences is subject of Sect. 4. In Sect. 5 we compare briefly our proposal to some related existing languages. Section 6 concludes the paper and gives some directions of future work.

2 Syntax

Many frameworks have been used to define the integration of functional and logic programming paradigms. Without loss of generality, we consider here Horn clause logic with equality (HCLE) as the basis of a functional logic language, e.g., [12, 3]. Our approach of concurrent functional logic programs can be generalised easily to other frameworks such as Curry [16] or Escher [20]. Hereafter, we assume that the reader is familiar with the classical notions about HCLE.

Let $P = (\Sigma, Cl)$ be a HCLE-theory presentation where $\Sigma = (S, \Omega, \Pi)$ is a many-sorted first order signature with equality and Cl a finite set of clauses defining the operator symbols in Ω as well as the predicate symbols in Π, S being the set of sorts. Roughly speaking, P constitutes a functional logic program. The operational semantics of such programs, which is classically based on rewriting, narrowing or (extended) SLD-resolution, allows the evaluation of functional expressions as well as goal-solving.

The theory presentation (or program) P describes a concrete system. But as the latter is a part of the real world, and the world changes, the theory modelling a part of the world has to change too, i.e., to be modified. So in this paper, we suggest to extend functional logic languages with *processes* controlling the evolution of an initial program P_0 through a finite or infinite sequence of

changes as it is shown below:

$$P_0 = (\Sigma, Cl_0) \longrightarrow P_1 = (\Sigma, Cl_1) \longrightarrow P_2 = (\Sigma, Cl_2) \cdots \quad (1)$$

In fact, the derivation (1) rather corresponds to a *run* (or behaviour) of an extended automaton. To motivate the need for systems having runs such as (1), we consider two examples.

The first one is a system keeping the temperature of a room about a given value (T_{avg}). It has a sensor for the current temperature (T_{env}) in the room and can command an air conditioning device (AC) to heat up or cool down the room. Then a possible instance of (1) may be: P_0 describes an "initial" state, e.g., $Cl_0 = \{T_{avg} == 18;\ T_{env} == 18\}$; P_1 is obtained from P_0 by a modification of the desired temperature of the room, e.g., $T_{avg} == 20$; and P_2 reflects the effect of starting the heating, e.g., T_{env} has raised to 19. As a second example, consider the well known dining philosophers problem. In that case, each P_i can be seen as a description of the situation of the philosophers, i.e., who is thinking and who is eating; P_1 may be obtained from P_0 after a philosopher stops eating and P_2 may be obtained from P_1 after a philosopher starts eating.

Note that each P_i in (1) constitutes a functional logic program and that its standard operational semantics can still be used. So we can for example calculate the pressure in the room, or ask for the philosophers who are eating, etc.

In the rest of this section we sketch an abstract syntax of a concurrent extension of functional logic programming languages. Informally, a concurrent functional logic program $\mathcal{P}$ will consist of five components, $\langle \Sigma^Q, \mathcal{P}rocs, Cl_0, \mathfrak{a}_0, \sigma_0 \rangle$. Σ^Q is a set of declarations (signature) of different entities used in $\mathcal{P}$ and Cl_0 is the initial functional logic program which can be modified by the processes defined in $\mathcal{P}rocs$. $\mathfrak{a}_0$ stands for the initial process (agent) to be run and σ_0 completes Cl_0 by giving initial values to "changing constants" (variables). Thus we introduce first the notion of a concurrent functional logic signature. Then we define a set of elementary actions, i.e., the primitives by means of which a process can modify a theory presentation (i.e., a functional logic program). Thereafter we introduce the notion of a process term, the operators on process terms and the "clauses" defining processes, namely procedures. Finally, we give the definition of a concurrent functional logic program.

2.1 Concurrent Functional Logic Signature

Definition 1. *A* concurrent functional logic signature Σ^Q *is a tuple* $\langle S, \Omega, \Pi, Q, \Xi, V \rangle$ *such that:*

- $\langle S, \Omega, \Pi \rangle$ *is a first-order signature with equality, i.e., S is a set of symbols (of sorts), Ω is a (S^+-indexed) family of operator-symbol-sets and Π is a (S^*-indexed) family of predicate-symbol-sets including equality.*
- $Q = \bigcup_{u \in S^*} Q_u$ *is a (S^*-indexed) family of symbol-sets, called* procedures *or* processes. *We assume that* success *is an element of* Q_ε[1].

[1] where Π_ε denotes the set of parameterless procedures (ε is the empty sequence).

– Ξ *is a set of symbols for* "external machines" *representing the external programs or physical devices with which the program can communicate.*
– $V = V_I \cup V_O \cup V_L \cup V_C$ *is a set of variables, where:*
  - $V_I = \bigcup_{s \in S}(V_I)_s$ *is a (S-indexed) family of symbol-sets, called* internal variables.
  - $V_O = \bigcup_{s \in S}(V_O)_s$ *is a (S-indexed) family of symbol-sets, called* external variables.
  - $V_L = \bigcup_{s \in S}(V_L)_s$ *is a (S-indexed) family of symbol-sets, called* queue-variables.
  - $\forall v \in V_L, empty_v \in \Pi_\varepsilon$.
  - $V_C = \bigcup_{\xi \in \Xi}(V_C)^\xi$ *is a (Ξ-indexed) family of (S-indexed) families of symbol-sets, called* channel-variables.

The internal variables (V_I) are similar to variables known in imperative programming languages: they are initialised before program-execution and are modifiable via assignment. The external variables (V_O) are similar to sensors: their value is determined by something outside the system. The queue-variables (V_L) are used to model the reception of values from an external machine via a FIFO-channel: arriving messages are implicitly buffered in the order of arrival. For each channel-variable v, we suppose the existence of the predicate $empty_v$ indicating if the associated channel is empty. In the sequel, the set of all internal, external and queue variables, i.e., $V_{st} = V_I \cup V_O \cup V_L$ will be called *state variables* (V_{st}). The channel-variables (V_C) denote a part of the communication interface with external machines, i.e., they represent the channels by means of which we can send messages to external machines. In this paper we do not consider how external machines implement the reception of messages. Nevertheless, whenever an external machine is modelled by a concurrent functional logic program, the reception of messages may be buffered (via queue-variables) or not (e.g., via external variables).

Example 1. According to the notations of Definition 1, we define the following concurrent functional logic signature for the temperature-controller example:

$$S = \{Nat;\ Mode\}$$
$$\Omega = \{0 :\rightarrow Nat;\ succ\colon Nat \rightarrow Nat;\ pred\colon Nat \rightarrow Nat;\ T_{min} :\rightarrow Nat;\ T_{max} :\rightarrow Nat\}$$
$$\Pi = \{\text{stop};\ \text{heat};\ \text{cool}\}$$
$$Q = \{\underline{\text{stop}};\ \underline{\text{heat}};\ \underline{\text{cool}};\ \underline{\text{success}}\}$$
$$V_I = \{T_{iavg}\}, \qquad V_O = \{T_{env}\}, \qquad V_L = \{T_{avg}\}, \qquad V_C = \varnothing$$
$$\Xi = \{\mathsf{AC}\}$$

We use external variables for a sensor (T_{env}) of the current temperature and a queue-variable for a control device (T_{avg}) through which a user may communicate to the system the desired average temperature. T_{iavg} is an internal variable used to memorise the current value of T_{avg}. AC stands for "Air Conditioner" and represents the physical machine that will receive the orders "stop", "cool" or "heat" from the temperature controller. The program for the controller is subject of Example 3.

2.2 Process Terms, Procedures and Programs

In the following, we suppose given a concurrent functional logic signature Σ^Q and denote by $T(\Sigma, X)$ (resp., $A(\Sigma, X)$, $\mathcal{G}(\Sigma, X)$ and $\mathcal{H}orn(\Sigma, X)$) the set of terms (resp., atoms, (finite) conjunctions of atoms and (definite) Horn clauses) built from the (first-order) signature $\Sigma = \langle S, \Omega, \Pi \rangle$ with variables X.

The following definition introduces the notion of elementary actions, that is to say the kind of operations which, when executed, make a theory presentation (i.e., a functional logic program) P_i evolve to P_{i+1} as in (1).

Definition 2. *Let V_p be the set of local variables needed to define the process* p*, and X a set of variables. Then the set of* elementary actions $\mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$ *over the variable-sets V_p and X is the smallest set (for inclusion) such that:*

1. $\mathsf{Nop} \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$
2. $\forall cl \in \mathcal{H}orn(\Sigma, X)$, $\mathsf{Tell}(cl) \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$
3. $\forall cl \in \mathcal{H}orn(\Sigma, X)$, $\mathsf{Del}(cl) \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$
4. $\forall x \in (V_I)_s, \forall t \in T_s(\Sigma, V_\mathrm{p})$[2], $(x := t) \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$
5. $\forall \xi \in \Xi, \forall A \in A(\Sigma, X)$, $\mathsf{Send}(\xi, A) \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$
6. $\forall \xi \in \Xi, \forall x \in (V_C)^\xi_s, \forall t \in T_s(\Sigma, V_\mathrm{p})$, $\mathsf{Send}(\xi, (x := t)) \in \mathcal{A}^\mathrm{e}(\Sigma^Q, V_\mathrm{p}, X)$

Nop is the invisible action without any effect. The elementary actions Tell and Del are dual: they are used to add and remove a clause from the current theory. ":=" assigns the value[3] of the term t to an internal variable v. Finally, Send allows the communication with the external machines via message-passing. These messages can be of two forms: an atomic formula or an affectation of a channel-variable. There are no elementary actions associated with the external and the queue-variables: their management is supposed to be external.

An *action* a ($\in \mathcal{A}(\Sigma^Q, V_\mathrm{p}, X)$) is a set of elementary actions satisfying:

1. $\forall \mathsf{Tell}(cl), \mathsf{Del}(cl') \in a$, cl is not a variant (equal up to renaming) of cl',
2. $\forall (x := e), (y := f) \in a$, $x \neq y$

These conditions imply that an action is consistent: so a double affectation or the simultaneous addition and removal of a same clause (modulo renaming of variables) are not allowed. Therefore the order of elementary actions in an action is insignificant, and an action can be considered as "atomic".

Definition 3. *A* process term *(or an* agent*)* $\mathfrak{a}$ *(containing the set V_p of local variables) is described by the following grammar:*

$$\mathfrak{a} ::= \underline{\mathrm{q}}(t_1, \ldots, t_n) \mid [g \Rightarrow a] \mid (\mathfrak{a}; \mathfrak{a}) \mid (\mathfrak{a} \parallel \mathfrak{a}) \mid (\mathfrak{a} + \mathfrak{a}) \mid (\mathfrak{a} \oplus \mathfrak{a})$$

$\underline{\mathrm{q}}(t_1, \ldots, t_n)$ is a procedure-call $\underline{\mathrm{q}}$ ($\in Q$) with parameters $t_i \in T(\Sigma, V_\mathrm{p})$. The process term $[g \Rightarrow a]$ describes the atomic execution of the action a, given the validity of the guard g ($\in \mathcal{G}(\Sigma, V_\mathrm{p})$) in the current store (i.e., the current theory).

[2] $T_s(\Sigma, X)$ denotes the set of terms of sort s ($T(\Sigma, X) = \bigcup_{s \in S} T_s(\Sigma, X)$).

[3] note that t's free variables have to be local in the procedure p.

In reference to process algebras [2], we call ; the operator of sequential composition, $\|$ the operator of parallelism and + the operator of non-deterministic choice. The operator $\oplus$ is not very common: we call it *operator of choice with priority*. It is necessary to model certain (critical) applications, where non-determinism is inadmissible [1]. The intended meaning of $\mathfrak{a}_1 \oplus \mathfrak{a}_2$ is to execute $\mathfrak{a}_2$ iff $\mathfrak{a}_1$ cannot be executed.

In the sequel, we will write $PT(\Sigma^Q, V_\mathrm{p}, X)$ for the set of process terms, and $RPT(\Sigma^Q, V_\mathrm{p}, X)$ for the set of process terms containing neither $[g \Rightarrow a]$ nor $\oplus$.

Definition 4. *A* procedure *is defined by a formula of the following form:*

$$\underline{\mathrm{p}}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{i=1}^{m} ([g_i \Rightarrow a_i]; \mathfrak{a}_i)$$

where (for each i) g_i is a guard (i.e., in $\mathcal{G}(\Sigma, \{x_1; \ldots; x_n\})$), a_i is an action (i.e., in $\mathcal{A}(\Sigma^Q, \{x_1; \ldots; x_n\}, X)$) and $\mathfrak{a}_i$ is a process term containing neither $\oplus$ nor $[g \Rightarrow a]$ (i.e., in $RPT(\Sigma^Q, \{x_1; \ldots; x_n\}, X)$), with the x_i as (local) variables of the procedure $\underline{\mathrm{p}}$.

Example 2. Consider the problem of the "Dining Philosophers." We model the situation with two predicates: fork(x) and is_eating(y). The former represents the fact that fork x is lying on the table, and the latter affirms that philosopher y is eating. Then we can model the behaviour of a philosopher by the two procedures shown in Fig. 1, corresponding to the two different behaviours of a philosopher: either a philosopher $\underline{\text{thinks}}$ or $\underline{\text{eats}}$. Note that we do not use low-level synchronisation (like semaphores) and auxiliary constructions (as placing the table in a separate room or asymmetric philosophers).

$$\underline{\text{thinks}}(x, n) \Leftarrow [\begin{pmatrix} \text{fork}(x) & \wedge \\ \text{fork}(x+1 \bmod n) & \end{pmatrix} \Rightarrow \left\{ \begin{array}{l} \mathsf{Del}(\text{fork}(x)) \\ \mathsf{Del}(\text{fork}(x+1 \bmod n)) \\ \mathsf{Tell}(\text{is\_eating}(x)) \end{array} \right\}]; \underline{\text{eats}}(x, n)$$

$$\underline{\text{eats}}(x, n) \Leftarrow [\text{TRUE} \Rightarrow \left\{ \begin{array}{l} \mathsf{Del}(\text{is\_eating}(x)) \\ \mathsf{Tell}(\text{fork}(x)) \\ \mathsf{Tell}(\text{fork}(x+1 \bmod n)) \end{array} \right\}]; \underline{\text{thinks}}(x, n)$$

Fig. 1. Procedures for the dining philosophers example

Now we can define a concurrent functional logic program.

Definition 5. *A* concurrent functional logic program $\mathcal{P}$ *is a tuple* $\langle \Sigma^Q, \mathcal{P}rocs, Cl_0, \mathfrak{a}_0, (\sigma_I)_0 \rangle$ *such that:*

- Σ^Q *is a concurrent functional logic signature,*
- $\mathcal{P}rocs$ *is the set of procedures of the program,*

– Cl_0 *is a set of Horn clauses with equality, called the* initial clause set*,*
– $\mathfrak{a}_0 \in PT(\Sigma^Q, \varnothing, X)$ *is the* initial process term*, containing no free variable,*
– $(\sigma_I)_0$ *is a valuation (i.e., a total substitution) over* V_I*, called* initialisation *(for the internal variables).*

According to Definition 5, a concurrent functional logic program has two parts: a static one (Σ^Q and $\mathcal{P}rocs$) and an initialisation. The former will be modifiable neither by the actions nor by the environment. The latter describes everything that is modifiable.

$\langle\langle S, (\Omega \cup V_{st}), \Pi\rangle, Cl_0 \cup (\sigma_I)_0\rangle$ is the initial theory which can be modified by the execution of actions, for example via addition and removal of clauses. $\mathfrak{a}_0$ is the initial process which will modify the state of the system via the application of transition rules. $(\sigma_I)_0$ initialises the internal variables; the external and queue-variables are initialised by the environment. This distinction reflects the fact that the programmer may ignore the (initial) environment.

Example 3. We give now the program for the temperature-controller mentioned at the beginning of this section. We take for Σ^Q the concurrent functional logic signature of Example 1. The procedures corresponding to the three states of the controller, e.g., to heat, to cool or to do nothing (stop), are defined by Fig. 2. The initialisation is defined by the initial theory $Cl_0 = \{pred(succ(x)) == x;\ T_{min} == pred(T_{iavg});\ T_{max} == succ(T_{iavg})\}$, the initial process $\mathfrak{a}_0 =$ stop and the initial value for the internal variable $(\sigma_I)_0 = \{T_{iavg} \mapsto 20\}$.

heat $\Leftarrow$	cool $\Leftarrow$
$\quad [T_{env} > T_{max} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{cool})]$; cool	$\quad [T_{env} < T_{min} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{heat})]$; heat
$\oplus\ [T_{env} \geq T_{iavg} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{stop})]$; stop	$\oplus\ [T_{env} \leq T_{iavg} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{stop})]$; stop
$\oplus\ [\text{TRUE} \Rightarrow (T_{iavg} := T_{avg})]$; heat	$\oplus\ [\text{TRUE} \Rightarrow (T_{iavg} := T_{avg})]$; cool
stop $\Leftarrow$	
$\quad [T_{env} < T_{min} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{heat})]$; heat	
$\oplus\ [T_{env} > T_{max} \Rightarrow \mathsf{Send}(\mathsf{AC}, \text{cool})]$; cool	
$\oplus\ [\text{TRUE} \Rightarrow (T_{iavg} := T_{avg})]$; stop	

Fig. 2. Procedures for the example of a temperature controller

3 Operational Semantics

In this section we give the operational semantics for the concurrent functional logic programming languages sketched before. The operational semantics of a concurrent functional logic program $\mathcal{P}$ is defined by a transition system $\mathfrak{T}\mathfrak{r}(\mathcal{P})$ (presented in Sect. 3.2) integrating two types of transitions corresponding to the two orthogonal aspects of concurrent functional logic languages: those describing

the evolution of the theory description (i.e., the execution of processes) and those dedicated to goal-solving (i.e., the use of the current theory as a functional logic program).

3.1 Describing the evolution of the theory

We first define a transition system $\mathsf{Tr}(\mathcal{P})$ which models (only) the evolution of the theory description (in its environment), i.e., the execution of processes of $\mathcal{P}$, without taking care of goal-solving. Therefore, we introduce first the notion of configurations which will serve as the states of $\mathsf{Tr}(\mathcal{P})$. Then we specify the operational semantics of the (elementary) actions, and finally present the transition system $\mathsf{Tr}(\mathcal{P})$.

Definition 6. *A* configuration *is a tuple* $\langle \mathfrak{a}, \langle Cl, \sigma_I \rangle, \sigma_O, L \rangle$ *such that:*

- $\mathfrak{a}$ *is a process term, i.e., in* $PT(\Sigma^Q, \varnothing, X)$,
- Cl *is a set of Horn clauses with equality,*
- σ_I *(resp.,* σ_O*) is a valuation for the internal (resp., external) variables and*
- L *is a mapping defining the content of the queues corresponding to the queue-variables* V_L*.*

A configuration represents the current state of the system: $\mathfrak{a}$ is the current process and the pair $(\langle Cl, \sigma_I \rangle, \sigma_O, L)$ represents the current theory or *store*. The distinction between Cl and σ_I on the one hand and L and σ_O on the other hand reflects the fact that (elementary) actions modify the set of clauses and the internal variables directly, whereas the handling of the queues and external variables is implicit. In the sequel, $\mathcal{C}onfig(\Sigma^Q, X)$ will denote the set of configurations.

For an atom A and a configuration $\mathcal{C} = \langle \mathfrak{a}, \langle Cl, \sigma_I \rangle, \sigma_O, L \rangle$, we say:

- If $A = empty_v$, then A is called *valid* in $\mathcal{C}$ iff $L(v) = nil$.
- If $A \neq empty_v$, then A is called *valid* in $\mathcal{C}$ iff $\sigma(Cl) \models \sigma(A)$, with $\sigma = \sigma_I \cup \sigma_O \cup \sigma_L$, where σ_L is a substitution which associates a queue-variable with the first element of its corresponding FIFO-channel, and $\models$ is the standard validity of first-order logic.

A guard g is valid in a configuration $\mathcal{C}$ iff all of its conjuncts are valid in $\mathcal{C}$. We note the validity of g in $\mathcal{C}$ by $\mathcal{C} \models g$.

The execution of actions is expressed by means of the auxiliary function *do* describing the transformations of theories (stores).

Definition 7. *Let* σ_L *be a substitution for the queue-variables as before, and* σ_l *a substitution for the local variables of a procedure. Let further* $\sigma = \sigma_I \cup \sigma_O \cup \sigma_L \cup \sigma_l$*. The (partial) function do is defined by:*

- $do(\mathsf{Nop}, \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I \rangle) = \langle Cl, \sigma_I \rangle$
- $do(\mathsf{Tell}(cl), \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I \rangle) = \langle Cl \cup \{\overline{\sigma_L(\sigma_l(cl))}\}, \sigma_I \rangle$
- $do(\mathsf{Del}(cl), \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I \rangle) = \langle Cl \backslash \{\overline{\sigma_L(\sigma_l(cl))}\}, \sigma_I \rangle$
  where $\overline{Cl}$ *stands for the clause-set obtained from* Cl *via variable renaming*
- $do((x := e), \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I \rangle) = \langle Cl, \sigma_I \lhd \{x \mapsto (\sigma(e))!\} \rangle$[4]
  where $t!$ *denotes a normal (simplified) form of the term* t

– $do(\mathsf{Send}(\xi, A), \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I\rangle) = \langle Cl, \sigma_I\rangle$
– $do(\mathsf{Send}(\xi, (x := e)), \sigma_O, L, \sigma_l)(\langle Cl, \sigma_I\rangle) = \langle Cl, \sigma_I\rangle$

The function *do* gives the following semantics to the elementary actions:

– Nop is the identity,
– Tell allows to add a formula in which all local and queue-variables have been substituted,
– Del removes a formula (after replacing the local and queue-variables) modulo renaming of bounded variables (we remove the corresponding equivalence class),
– ":=" changes the value of an internal variable (the new value is the normal form (symbol "!") of the expression e, where all state and local variables have been replaced by their current value),
– Send does not alter the configuration – it is nevertheless different from Nop, because its implicit semantics is to send a message to an *external* machine.

We will note $\tilde{do}$ the straightforward extension of *do* over actions.

The execution of an action has no effect on external variables. But all the queue-variables used in an action (denoted by $\mathcal{V}_L([g \Rightarrow a]))$ are modified: the first message in the associated queues is removed after the execution. We describe this by the function *newL*:

$$newL([g \Rightarrow a], L) = L \lhd \{v \mapsto tail(l) \mid v \in \mathcal{V}_L([g \Rightarrow a]) \text{ and } l = L(v)\}$$

The transition system $\mathsf{Tr}(\mathcal{P})$ modelling the evolution of a process (in a system described by $\mathcal{P}$) is a triple $\langle \mathcal{C}onfig(\Sigma^Q, X), \longrightarrow, \mathcal{C}_0\rangle$, where $\mathcal{C}_0$ is an initial configuration (i.e., $\mathcal{C}_0 = \langle \mathfrak{a}_0, \langle Cl_0, (\sigma_I)_0\rangle, (\sigma_O)_0, \varnothing\rangle$ ($(\sigma_O)_0$ being a valuation of the external variables)), and the transition relation is defined by a set of inference rules given in Fig. 3. In this section we will not consider the transition-labels. In fact, they are used in Sect. 4 to construct a compositional semantics.

According to rule (R1), the process success is always executable and its execution yields the special symbol SS. The latter witnesses the successful termination of processes. Rule (R2) concerns the execution of a process $[g \Rightarrow a]$, where $a \in \mathcal{A}(\Sigma^Q, \varnothing, X)$ and $g \in \mathcal{G}(\Sigma, \varnothing)$, i.e., $[g \Rightarrow a]$ contains no free variables. The action a is executed, when the guard g holds in the current configuration and no queue used in the guard or the elementary actions is empty. The transformation of the configuration is described using the functions $\tilde{do}$ and *newL* defined above. The substitution σ_l in rule (R3) replaces the formal parameters of the procedure p, i.e., $x_1, \ldots, x_n$ by the actual arguments, i.e., $t_1, \ldots, t_n$, thus $\sigma_l = \{x_i \mapsto t_i \mid i \in \{1; \ldots; n\}\}$. So rule (R3) tells us that if the body of p (after applying σ_l) can make a transitions, then the call $\mathrm{p}(t_1, \ldots, t_n)$ can do alike.

The rules concerning the operators of sequential composition, of parallelism and of choice are as usual. Due to space limitations we do not show the symmetric version of the rules (R5), (R5′) and (R6).

[4] $(\sigma \lhd \{x \mapsto v\})(x) = v$ and $(\sigma \lhd \{x \mapsto v\})(y) = \sigma(y)$ when $x \neq y$.

(R1) $$\frac{}{\langle \underline{\text{success}}, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{Success}^{\mathsf{p}}} \langle \text{SS}, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle}$$

(R2) $$\frac{\langle [g \Rightarrow a], \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \models g \qquad \forall x \in \mathcal{V}_L([g \Rightarrow a]),\ L(x) \neq nil}{\langle [g \Rightarrow a], \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{[g \Rightarrow a]^{\mathsf{p}}} \langle \text{SS}, \tilde{do}(a, \sigma_O, L, \varnothing)(\langle Cl, \sigma_I\rangle), \sigma_O, newL([g \Rightarrow a], L)\rangle}$$

(R3) $$\frac{\underline{\text{p}}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{j=1}^{m}([g_j \Rightarrow a_j]; \mathfrak{a}_j) \in \mathcal{P}rocs \quad \langle \bigoplus_{j=1}^{m} \sigma_l([g_j \Rightarrow a_j]; \mathfrak{a}_j), \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \underline{\text{p}}(t_1, \ldots, t_n), \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}$$

(R4) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1; \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1'; \mathfrak{a}_2, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle} \text{ if } \mathfrak{a}_1' \neq \text{SS}$$

(R4′) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \text{SS}, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1; \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_2, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}$$

(R5) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1 \parallel \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1' \parallel \mathfrak{a}_2, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle} \text{ if } \mathfrak{a}_1' \neq \text{SS}$$

(R5′) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \text{SS}, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1 \parallel \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_2, \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}$$

(R6) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1 + \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}$$

(R7l) $$\frac{\langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}{\langle \mathfrak{a}_1 \oplus \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{\mathsf{a}^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle}$$

(R7r) $$\frac{\langle \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{(\mathsf{a}'')^{\mathsf{p}}} \langle \mathfrak{a}_2'', \langle Cl'', \sigma_I''\rangle, \sigma_O, L''\rangle}{\langle \mathfrak{a}_1 \oplus \mathfrak{a}_2, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{(\mathsf{a}'')^{\mathsf{p}}} \langle \mathfrak{a}_2'', \langle Cl'', \sigma_I''\rangle, \sigma_O, L''\rangle}$$
if $\nexists\, \mathfrak{a}_1'(\neq \mathfrak{a}_1), \mathsf{a}', Cl', L' : \langle \mathfrak{a}_1, \langle Cl, \sigma_I\rangle, \sigma_O, L\rangle \xrightarrow{(\mathsf{a}')^{\mathsf{p}}} \langle \mathfrak{a}_1', \langle Cl', \sigma_I'\rangle, \sigma_O, L'\rangle$

Fig. 3. Transition rules

3.2 Operational Semantics of a Program

For a concurrent functional logic program $\mathcal{P} = \langle \Sigma^{\mathcal{Q}}, \mathcal{P}rocs, Cl_0, \mathfrak{a}_0, (\sigma_I)_0 \rangle$, the operational semantics is composed of two orthogonal parts: the transition system describing the state changes (given in Sect. 3.1) and the goal-solving, based on a complete system for functional logic programs such as (extended) SLD-resolution, rewriting, narrowing, etc. Thus, the operational semantics of a concurrent functional logic program $\mathcal{P}$ is defined by a new transition system $\mathfrak{T}\mathfrak{r}(\mathcal{P})$ that combines dynamic rules (which change the theory) and "static" inference rules (that solve goals). This transition system is a triple $\mathfrak{T}\mathfrak{r}(\mathcal{P}) = \langle \mathcal{Q}, \longmapsto, q_0 \rangle$ where $\mathcal{Q} = (\mathcal{C}onfig(\Sigma^{\mathcal{Q}}, X)) \times (\mathcal{G}oal \times \mathcal{G}oal \times \mathcal{S}ubst)$ with $\mathcal{S}ubst$ is the set of substitutions over terms $T(\Sigma, X)$, $q_0 \in \mathcal{Q}$ is the initial state of the system of the form $q_0 = \langle \mathcal{C}_0, \langle G, G, \varnothing \rangle \rangle$, where $\mathcal{C}_0$ is an initial configuration for P and G is a goal to solve. The transition relation $\longmapsto$ is defined by the two following "abstract" rules.

Rule (STA) concerns the goal resolution. $G_1 \rightsquigarrow_\sigma G_2$ means that the new goal G_2 is deduced by the inference rules of a functional logic language.

$$\text{(STA)}\ \frac{G_1 \rightsquigarrow_\sigma G_2}{\langle \mathcal{C}, \langle G, G_1, \vartheta \rangle \rangle \ \longmapsto\ \langle \mathcal{C}, \langle G, G_2, \sigma \circ \vartheta \rangle \rangle}$$

As usual in the description of functional logic languages, (STA) describes only one possible step; in fact, an implementation has to take into account the set of *all* possible applications of (STA), by for example depth-first (backtracking) or breadth-first traversal strategy of the search space.

The second rule describes the states changes of the system. $\mathcal{C} \longrightarrow \mathcal{C}'$ witnesses the change of configuration according to the rules in Sect. 3.1.

$$\text{(DYN)}\ \frac{\mathcal{C} \ \longrightarrow\ \mathcal{C}'}{\langle \mathcal{C}, \langle G, G', \vartheta \rangle \rangle \ \longmapsto\ \langle \mathcal{C}', \langle G, G, \varnothing \rangle \rangle}$$

Thanks to this operational semantics, we can solve goals while the processes are running. So we might ask in the dining philosophers example the question is_eating(Z) in order to know which of the philosophers are currently eating. As the philosophers continue to run, we will get different answers to this question, depending on the time when we ask.

4 A Compositional Semantics Based on Labelled Sequences

In this section we present a compositional denotational semantics for the languages presented before. Compositionality allows the inference of the semantics of a program from that of its components. Thus compositionality eases reusability as well as program analysis and validation and so it is most desirable for building large systems.

Let $\mathcal{P}$ be a concurrent functional logic program. A first semantics for $\mathcal{P}$, denoted by $\mathcal{T}(\mathcal{P})$, may be defined as the set of possible executions or traces of $\mathfrak{Tr}(\mathcal{P})$, such as in (1). An element in $\mathcal{T}(\mathcal{P})$ is a sequence of actions which can be observed by executing $\mathcal{P}$. For example, the sequence of actions $a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdots$ is a trace of the run

$$P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 \xrightarrow{a_3} P_3 \xrightarrow{a_4} P_4 \ \cdots \tag{2}$$

It turns out that this semantics is not compositional. Indeed consider the following counter-example:

Example 4. Consider the following three processes:

$$\mathfrak{a}_1 = Q \Rightarrow \underline{\text{success}}, \quad \mathfrak{a}_2 = R \Rightarrow \underline{\text{success}} \quad \text{and} \quad \mathfrak{a}_3 = \mathsf{Tell}(Q); \underline{\text{success}}$$

Then, starting with an empty store (theory), we have: $\mathcal{T}(\mathfrak{a}_1) = \mathcal{T}(\mathfrak{a}_2) = \varnothing$, whereas $\mathcal{T}(\mathfrak{a}_1 \parallel \mathfrak{a}_3) \neq \mathcal{T}(\mathfrak{a}_2 \parallel \mathfrak{a}_3)$

The source of the non-compositionality is that sequences as (2) represent only the behaviour of the process under consideration without taking into account their context, that is to say possible actions of other parallel processes or the environment. Thus two processes, e.g., $\mathfrak{a}_1$ and $\mathfrak{a}_2$, though having the same semantics, may behave differently in some contexts, e.g., when executed in parallel with $\mathfrak{a}_3$.

4.1 Semantics $\mathcal{T}'$

To overcome the defect of non-compositionality, the semantics of a process $\mathfrak{a}$ has to take into account the possible contexts within which it may run. Thus we provide a new transition rule ((R8), see Fig. 4) which allows a process to make assumptions about the behaviour of other processes executing concurrently on the same store. This new rule mimics exactly rule (R2), the only difference is the transition-label (and the unmodified agent $\mathfrak{a}$). Figure 4 gives also the rules concerning the modelling of the environment: rule (R9) models the changes of external variables by the environment, rule (R10) describes the arrival of new values at the end of the queues associated with the queue-variables[5], and rule (R11) the reception of an atomic formula which is immediately added to the store. In practice, the predicate building this atomic formula has to be declared as a communication predicate in order to be acceptable in the receiving store.

Furthermore, to distinguish between the actions actually executed by the process (via the rule (R2)) and assumptions about actions executed by someone else (i.e., applications of rule (R8)), we label the actions in the traces. A labelled sequence is a sequence of labelled actions, as it can be obtained as an execution trace of the transition system $\mathsf{Tr}(\mathcal{P})$ (presented in Sect. 3.1 plus rule (R8), but now including the transition labels). These labels ($\{\mathsf{p}, \mathsf{o}, \mathsf{e}\}$) are used as indicators for the "author" of the corresponding action. In extension to [10], we have

[5] $\ll$ denotes the enqueue-operation.

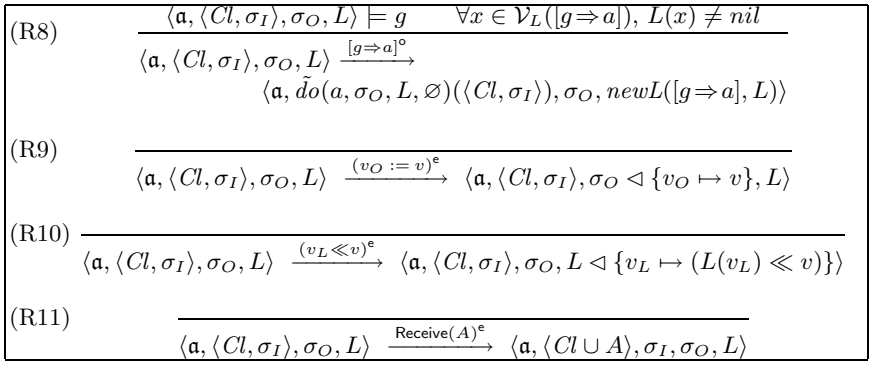

Fig. 4. Rule for modelling modifications by concurrent processes on the same store

three possibilities: the process itself, p, another process, o, (which is executed concurrently on the same store) or the environment, e, (an external machine) which modifies the external variables or sends values to the queue-variables. It is necessary to distinguish between other processes and external machines, because the actions of other processes could be executed by the process itself (the rules (R8) and (R2) are identical except the label), but the actions of external machines cannot.

Definition 8. *The* semantics $\mathcal{T}'$ *is a function associating to a process term (for a program P and an initial configuration $\mathcal{C}_0$) the set of (finite and infinite) traces that can be produced by an execution via the transition system, starting with $\mathcal{C}_0$.*

4.2 Semantic Operators

In this paragraph, we will define the semantic operators corresponding to those defined on processes, i.e., $\tilde{;}$, $\tilde{\|}$, $\tilde{+}$ and $\tilde{\oplus}$. The goal is to define the right operators on the semantical level that allow the definition of a compositional semantics.

For $\tilde{;}$ and $\tilde{\|}$, we define first the corresponding partial operators on sequences (represented equally by $\tilde{;}$ and $\tilde{\|}$); the extensions to sets (of sequences) are straightforward and given later.

Sequential Composition. To execute the processes $\mathfrak{a}_1$ and $\mathfrak{a}_2$ one after another means to execute first $\mathfrak{a}_1$ and then $\mathfrak{a}_2$; in addition, to make $\mathfrak{a}_1$ executable, it may be necessary to make some assumptions (i.e., transitions using rules (R9) to (R8)) about the behaviour of other processes and the environment. The same holds for $\mathfrak{a}_2$ once $\mathfrak{a}_1$ has terminated (if ever). So a sequence s can only be in $\mathcal{T}'(\mathfrak{a}_1; \mathfrak{a}_2)$ if (by only changing the labels) s can be seen as both, an execution trace of $\mathfrak{a}_1$ followed by an assumption of $\mathfrak{a}_2$ and an execution trace of $\mathfrak{a}_2$ preceded by an assumption of $\mathfrak{a}_1$.

Let $s_1 = (\mathsf{a}_i^{x_i^1})_{i\in\mathbb{N}}$ and $s_2 = (\mathsf{a}_i^{x_i^2})_{i\in\mathbb{N}}$ be two labelled sequences such that there are $m \in \mathbb{N}$ and n, $\mu \in \mathbb{N} \cup \{\infty\}$ satisfying the following conditions:

- $m + \mu < n$ ($\mathfrak{a}_1$ has finished before $\mathfrak{a}_2$ starts),
- $\mu > 0$ ($\mathfrak{a}_1$ "does something"),
- $\forall j$ such that $0 \leq j < m$ and $m + \mu < j$, $x_j^1 \in \{\mathsf{e}; \mathsf{o}\}$ ($\mathfrak{a}_1$ "does not do anything" before step m and after step $m + \mu$),
- $\forall j$ s.t. $0 \leq j < n$, $x_j^2 \in \{\mathsf{e}; \mathsf{o}\}$ ($\mathfrak{a}_2$ "does not do anything" before step n), and
- $\forall j$, $x_j^1 = \mathsf{e}$ iff $x_j^2 = \mathsf{e}$ (in both sequences, the interactions with the environment take place in the same steps).

Then we define $s_1 \tilde{;} s_2 = (\mathsf{a}_i^{y_i})_{i\in\mathbb{N}}$ where $y_i = \begin{cases} x_i^1 & \text{if } i < n \\ x_i^2 & \text{otherwise} \end{cases}$.

Parallelism. To execute two processes in parallel, both processes have to make assumptions about the execution of the other. If this is the case, the actions executed by the parallel composition are the actions executed by the processes. Obviously, both processes cannot execute an action at the same time.

Let $s_1 = (\mathsf{a}_i^{x_i^1})_{i\in\mathbb{N}}$ and $s_2 = (\mathsf{a}_i^{x_i^2})_{i\in\mathbb{N}}$ be two labelled sequences.

Then define $s_1 \tilde{\|} s_2 = (\mathsf{a}_i^{y_i})_{i\in\mathbb{N}}$ where $y_i = \begin{cases} \mathsf{p} & \text{if } x_i^1 = \mathsf{p} \text{ and } x_i^2 = \mathsf{o} \\ \mathsf{p} & \text{if } x_i^1 = \mathsf{o} \text{ and } x_i^2 = \mathsf{p} \\ \mathsf{o} & \text{if } x_i^1 = \mathsf{o} \text{ and } x_i^2 = \mathsf{o} \\ \mathsf{e} & \text{if } x_i^1 = \mathsf{e} \text{ and } x_i^2 = \mathsf{e} \end{cases}$.

Extension of these Operators. Since arbitrary assumptions can always be made, it is sufficient to consider the cases listed above, and to leave $\tilde{;}$ and $\tilde{\|}$ undefined on all sequences of different kind. The extension of those operators to sets of sequences is defined in the obvious way:

$$A \tilde{;} B = \{s \mid \exists s_1 \in A, \exists s_2 \in B \text{ such that } s_1 \tilde{;} s_2 = s\}$$
$$A \tilde{\|} B = \{s \mid \exists s_1 \in A, \exists s_2 \in B \text{ such that } s_1 \tilde{\|} s_2 = s\}$$

Non-Deterministic Choice. It is possible to execute both processes. Therefore we simply have $\tilde{+} = \cup$.

Choice with Priority. When executing $\mathfrak{a}_1 \oplus \mathfrak{a}_2$, the process $\mathfrak{a}_2$ is executed iff in the configuration in which rule (R7r) is applied, rule (R7l) cannot be applied. This means in terms of sequences, that there is no sequence for $\mathfrak{a}_1$ that makes the same assumptions as the sequence for $\mathfrak{a}_2$ and has its first action labelled p. To put this formally, we introduce the notion of a *hypothetical prefix*.

Definition 9. *The* hypothetical prefix *of a labelled sequence* $s = \mathsf{a}_1^{\ell_1} \cdot \cdots \cdot \mathsf{a}_{n-1}^{\ell_{n-1}} \cdot \mathsf{a}_n^{\mathsf{p}} \cdot s'$*, where all* $\ell_i \in \{\mathsf{o}, \mathsf{e}\}$ *is the (finite) labelled sequence* $\mathsf{prh}(s) = \mathsf{a}_1^{\ell_1} \cdot \cdots \cdot \mathsf{a}_{n-1}^{\ell_{n-1}}$.

We define $S_1 \tilde{\oplus} S_2 = S_1 \cup \{s \mid s \in S_2 \text{ and } \forall s' \in S_1 : \mathsf{prh}(s) \neq \mathsf{prh}(s')\}$.

From the definitions above, we can prove the following equalities which express the compositionality of the semantics $\mathcal{T}'$.

Theorem 1 (Compositionality of $\mathcal{T}'$). *Let $\mathfrak{a}_1, \mathfrak{a}_2 \in PT(\Sigma^Q, \varnothing, X)$. Then the following equations hold:*

$$\begin{aligned}
\mathcal{T}'(\mathfrak{a}_1 \parallel \mathfrak{a}_2) &= \mathcal{T}'(\mathfrak{a}_1) \mathbin{\tilde{\parallel}} \mathcal{T}'(\mathfrak{a}_2) \\
\mathcal{T}'(\mathfrak{a}_1; \mathfrak{a}_2) &= \mathcal{T}'(\mathfrak{a}_1) \mathbin{\tilde{;}} \mathcal{T}'(\mathfrak{a}_2) \\
\mathcal{T}'(\mathfrak{a}_1 + \mathfrak{a}_2) &= \mathcal{T}'(\mathfrak{a}_1) \mathbin{\tilde{+}} \mathcal{T}'(\mathfrak{a}_2) \\
\mathcal{T}'(\mathfrak{a}_1 \oplus \mathfrak{a}_2) &= \mathcal{T}'(\mathfrak{a}_1) \mathbin{\tilde{\oplus}} \mathcal{T}'(\mathfrak{a}_2)
\end{aligned}$$

5 Related Work

As mentioned in the introduction, many authors noticed the need of adding concurrency to declarative programming languages. We do not survey all the propositions in the area, but focus on some of those which are close to our proposal. Our proposal departs radically from the classical concurrent extensions of declarative languages in the sense that it does not encode processes but considers them as first class citizens.

Concurrent Functional Programming. Both CML [24] and Concurrent Haskell (CH) [22] propose concurrent extensions for functional languages by adding to ML resp., to Haskell new primitives which can be used to "encode" concurrent systems. In CH, the authors defined new primitive operations based on monadic IO. These primitives are `forkIO` and those forming the data type `MVar`, namely, `newMVar`, `takeMVar` and `putMVar`. The `forkIO` primitive allows to start a new parallel process, whereas the `MVar` data type is used to describe atomically-mutable state. Thanks to this data type "more friendly abstractions" [22] for communication and cooperation of processes (as for example quantity semaphores `QSem`) can be implemented. A CH-program for the dining philosophers is given in Fig. 5. This program tries to simulate the behaviour of the program given in Fig. 1, i.e., a philosopher takes the two forks atomically. Thus, contrary to Fig. 1, the programmer has to encode the currently available forks as well as the associated operations (e.g., `getForks`, `putForks`, `possible` and `newForks`).

As for CML, parallel processes are implemented by "a number of threads, which use message passing on typed channels". The implementation of threads and their communication is made by set of primitives: `spawn` (resp., `channel`) allows to create new threads (resp., channels). The primitives `sync`, `transmit`, `receive`, `choose` and `wrap` are used to achieve synchronisation and communication between threads.

Though these primitives of CML or CH look very interesting for concurrent programming, they are still low level: We find it more abstract to use *new language constructs* to express concurrency. Indeed, in our framework processes are specified independently from functions and predicates. Furthermore the proposed primitives of (CML and CH) do not distinguish between parallel evaluation of functional expressions and concurrent execution of processes. As a consequence of this last point, there is no hope to extend, in a straightforward manner, the

```
phil s i = think >> getForks s i >> eat >> putForks s i >> phil s i

getForks s i = takeMVar s >>= \ fl ->
               if (possible fl i)
               then putMVar s (newForks fl i)
               else (putMVar s fl >> threadDelay 1000 >> getForks s i)

putForks s i = takeMVar s >>= \ forks ->
               putMVar s (i : (((i+1) `mod` n) : forks))

possible l i = and [i `elem` l, ((i+1) `mod` n) `elem` l]

newForks []     i = []
newForks (f:fl) i = if (or [f == i, f == ((i + 1) `mod` n)])
                       then (newForks fl i)
                       else (f : (newForks fl i))

think = ...                          eat = ...
```

Fig. 5. Dining Philosophers in Concurrent Haskell

operational semantics in order to allow interactive goal-solving. Also, there is no direct support for building atomic actions as we used in our modelling of the dining philosophers program.

LOTOS [17] is a language which integrates process definitions with functions. It allows the definition of processes using classical process algebra operators. LOTOS also provides the possibility to define functions in equational logic. However, communication between processes in LOTOS is performed via connected gates in contrast to our framework, where processes communicate via the store. Our state variables (V_{st}) play almost the same rôle as LOTOS gates. But one of the main advantages in using state variables is the possibility to broadcast a message in one shot, just by instantiating a variable in the store by the right message. The use of a store as a medium of communication allows sophisticated message passing via logic formulas (or constraints) which is impossible in LOTOS. Also, in LOTOS the function-definitions (theory presentations) cannot be modified.

Concurrent (Constraint) Logic Programming. Concurrent Constraint Programming (ccp) [27] integrates ideas from concurrent logic programming [30] and constraint logic programming [18]. In ccp a set of agents (or processes) communicates via a common store. These agents are defined according to the following grammar:

$$A ::= \mathsf{stop} \mid \mathsf{fail} \mid \mathsf{ask}(c) \rightarrow A \mid \mathsf{tell}(c) \rightarrow A \mid A + A \mid A \parallel A \mid \exists X.A \mid p(\vec{t})$$

where $p(\vec{t})$ corresponds to a procedure call. Procedure-definitions are sentences of the form $p(\vec{x})$:- A; obviously they play the same rôle as our procedures. Nevertheless, there are noteworthy differences between the two frameworks, e.g.,

– We found the operator of choice with *priority* ($\oplus$) very useful for some examples. Unfortunately this operator does not exist in ccp. However, in a recent extension of ccp, [8], a now-then-else operator has been introduced which can simulate the operator $\oplus$.
– A store in ccp is monotonic. This is not the case in our approach as we allow the deletion of clauses (see e.g., Del). Some variants of ccp with non-monotonic stores have been proposed, see e.g., [29, 9, 28, 4, 11].
In [9] an operator $update_x$ is used to hide the name of a variable. Our Del operator is more precise and allows us to describe for instance the example of philosophers without special requirements on the structure of the constraint system. In fact, the program:

$$\begin{array}{ll} phil(x) & \text{:- } eat(x) \\ eat(x) & \text{:- } atell(use(x, leftfork) \wedge use(x, rightfork)); think(x) \\ think(x) & \text{:- } update_x(true); eat(x) \end{array}$$

of [9] needs the following condition on the underlying constraint system (where the x_i represent the philosophers and the basic constraint are of the form $use(x_i, leftfork)$ or $use(x_i, rightfork)$):

$$\begin{aligned} use(x_1, leftfork) \wedge use(x_2, rightfork) &= use(x_2, leftfork) \wedge use(x_3, rightfork) \\ &\vdots \\ &= use(x_n, leftfork) \wedge use(x_1, rightfork) \\ &= false \end{aligned}$$

Note that this condition in combination with the "atomic tell" (*atell*) is needed to model the atomic request for two forks.
Another different approach to handle non-monotonic stores is presented in [29, 4, 11]. This approach is based on linear logic and allows the *implicit* deletion of information thanks to the semantics of the *linear* ask.
[28] is based on default logic. Thus, if an action is taken based on the absence of information, it is assumed that this information will not be present in the final result. To model non-terminating systems, the framework is extended over time in the style of synchronous languages, i.e., a given set of processes is executed (until termination) at every time instant.
– In our proposition, we distinguish clearly between the behaviour of processes and goal-solving. The latter may be performed sequentially or concurrently à la parallel-Prolog. However, this distinction is not present in ccp (nor in the extensions we know of) where the resolution of a goal corresponds surprisingly to the run of a process.
– State variables are necessary for many real-word applications. Unfortunately, there is no counterpart of state variables in ccp and its extensions we are aware of.

The logic programming language Prolog provides two "predicates" that allow to modify the logic program by adding and deleting clauses, namely `retract` and `assert` [26]. Obviously, they destroy the declarativity of the language: it is

necessary to know implementation details to use them correctly (consider, e.g., the difference between `asserta` and `assertz`).

In Shared Prolog (SP, [6]), the "extra-logical operators" `assert` and `retract` are interpreted as send and receive on a blackboard built of atomic formulas (so-called "blackboard interpretation" of logic programs). An SP-program is built from several "theories" that communicate via a multiset of atoms, called the "blackboard". Each theory is provided with a logic program and a set of (guarded) rules. Whenever the guard of a rule holds, the rule is activated, i.e., retracts some atoms from the blackboard, starts the solving of a goal using the logic program of its associated theory, and on successful termination adds a set of new atoms to the blackboard. Our approach is more general, as our actions modify *stores*, i.e., more than only a multiset of atoms.

Recently, another extension of Prolog close to the "blackboard interpretation" has been proposed [7] – without the need of large modifications of the underlying execution model, namely the WAM. It provides (a family of) new primitives for launching the concurrent solving of goals (in a (local) copy of the current logic program). Additional primitives allow to control this new goal-solving process, e.g., to force backtracking or termination, or to synchronise on its result. Communication between these processes is achieved by `assert`ing and `retract`ing (marked) atoms. This model has the same restrictions as Shared Prolog. Meanwhile, [7] proposes good implementation techniques which can be extended to our general framework.

AKL [19] introduces the notion of *ports* as a communication medium for processes described in concurrent constraint logic programming framework. Primitives on ports such as $\mathsf{open\_port}(P, S)$ or $\mathsf{send}(P, m)$ have been introduced to describe message passing. It is argued in [19] that the introduced port primitives have a logical reading and preserve the monotonicity of the constraint store. In our case, message passing can be achieved by the modification of state variables, or the addition or deletion of clauses of the (constraint) store. Since in our framework monotonicity is not a concern, we do not need ports; they can be implemented via state variables. Recently, the idea of ports has been extended and integrated in the functional logic programming language Curry [15] in order to cope with distributed applications.

Synchronous Programming. Our processes communicate in an asynchronous manner. They can also synchronise through a rendezvous. Nevertheless, hard real time applications do need more sophisticated synchronisation primitives that can be found in some domain specific languages [13]. The real-time extensions of our framework is under work.

6 Conclusion

We have sketched a concurrent extension of functional logic programming languages which allows to model concrete applications where concurrency is necessary. In this paper, we have used many-sorted Horn clause logic with equality as

a means to define abstract data types, functions and predicates. Of course, the choice of this logic is not mandatory; any formalism allowing the integration of functional and logic languages (with constraints) can be considered. Our operational semantics distinguishes clearly the rules that implement the behaviour of processes from the rules of a traditional functional logic language. In addition, we presented a compositional semantics that takes into account the behaviour of the environment.

Future work includes the axiomatisation of the equivalence defined by our compositional semantics over processes and the investigation of new proof techniques for temporal logics such as (extended) LTL. Other topics are the integration of different kind of synchronisations which are needed in some real-time applications. For these applications the integration of temporal aspects in our framework will be mandatory. Finally, the extension of the framework to consider systems of several stores, possibly in different languages and the generalisation of the notion of elementary actions provide promising fields of research.

The implementation of a prototype is under progress.

References

[1] J.-R. Abrial. *Formal Methods for Industrial Applications*, *LNCS* 1165, chapter Steam-Boiler Control Specification Problem, pages 500–510. Springer, 1996.

[2] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[3] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proc. of ESOP '86*, *LNCS* 213, pages 119–132, Saarbrücken, March 1986. Springer.

[4] E. Best, F. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In D. Garlan and D. L. Métayer, editors, *Proc. of COORDINATION '97*, *LNCS* 1282, pages 256–273, Berlin, September 1997. Springer.

[5] J. Blanc, R. Echahed, and W. Serwe. Towards reactive functional logic programming languages. In H. Kuchen, editor, *Proc. of WFLP '98*. Institut für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, April 1998.

[6] A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM TOPLAS*, 13(1):99–123, January 1991.

[7] M. Carro and M. Hermenegildo. Concurrency in prolog using threads and a shared database. In *Proc. of ICLP '99*, Las Cruces, November 1999. MIT Press.

[8] F. S. de Boer and M. Gabbrielli. Modeling real-time in concurrent constraint programming. In J. W. Lloyd, editor, *Proc. of ILPS '95*, pages 528–542, Portland, December 1995.

[9] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *Proc. of ILPS '93*, pages 315–334. The MIT Press, 1993.

[10] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proc. of TAPSOFT '91, Volume 1: CAAP '91*, *LNCS* 493, pages 296–319, Brighton, UK, April 1991. Springer.

[11] F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proc. of the* 13^{th} *Annual IEEE Symp. on Logic in Computer Science (LICS '98)*, 1998.

[12] J. A. Goguen and J. Meseguer. EQLOG: Equality, types and generic modules for logic programming. In DeGroot and Lindstrom, editors, *Functional and Logic Programming*. Prentice Hall, 1986.

[13] N. Halbwachs. Synchronous programming of reactive systems: A tutorial and commented bibliography. In A. J. Hu and M. Y. Vardi, editors, *Proc. of CAV '98*, *LNCS* 1427, pages 1–16, Vancouver, June/July 1998. Springer.

[14] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19 & 20:583–628, 1994.

[15] M. Hanus. Distributed programming in a multi-paradigm declarative language. In G. Nadathur, editor, *Proc. of PPDP '99*, *LNCS* 1702, pages 188–205, Paris, 1999. Springer.

[16] M. Hanus (Ed.). Curry: An integrated functional logic language. available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/report.html`, 1999.

[17] ISO/IEC JTC1/SC21 WG7. *Final Committee Draft on Enhancements to* LOTOS, May 1998. Project: WI 1.21.20.2.3.

[18] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of POPL '87*, pages 111–119, München, January 1987. ACM.

[19] S. Janson, J. Montelius, and S. Haridi. Ports for objects in concurrent logic programs. In Agha, Wegner, and Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, 1993.

[20] J. W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Departement of Computer Science, University of Bristol, June 1995.

[21] M. Olmedilla, F. Bueno, and M. Hermenegildo. Automatic exploitation of non-determinate independent and-parallelism in the basic andorra model. In Y. Deville, editor, *Proc. of LOPSTR '93*, Workshops in Computing, pages 177–165. Springer, 1993.

[22] S. L. Peyton Jones, A. D. Gorden, and S. Finne. Concurrent Haskell. In *Proc. of POPL '96*, pages 295–308, St Petersburg Beach, Florida, January 1996.

[23] G. Puebla and M. Hermenegildo. Abstract specialization and its application to program parallelization. In J. Gallagher, editor, *Proc. of LOPSTR'96*, *LNCS* 1207, pages 169–186. Springer, 1997.

[24] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. of PLDI '91*, ACM SIGPLAN Notices, pages 293–305, Toronto, June 1991. ACM Press.

[25] J. A. Robinson and E. E. Sibert. The LOGLISP user's manual. Technical Report 12/81, Syracuse University, New York, 1981.

[26] P. H. Salus, editor. *Handbook of Programming Languages: Functional and Logic Programming Languages*, volume 4, chapter Prolog: Programming in Logic. Macmillan Technical Publishing, 1998.

[27] V. A. Saraswat. *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[28] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996.

[29] V. A. Saraswat and P. Lincoln. Higher-order, linear, concurrent constraint programming. Technical report, Xerox PARC, 1992.

[30] E. Shapiro. The family of concurrent logic programmming languages. *ACM Computing surveys*, 21(3):412–510, 1989.