A Concurrent Extension of Functional Logic Programming Languages

(extended abstract)

Rachid Echahed and Wendelin Serwe

Laboratoire Leibniz – Institut IMAG, CNRS
46, av. Felix Viallet, F-38031 Grenoble – France
Tel: (+33) 4 76 57 48 91; Fax: (+33) 4 76 57 46 02
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 labeled 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 [19]. Since then, a plethora of such declarative languages has been proposed, see [9] for a survey, and [11, 15] 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 in parallel 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. 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 in the same way as functions or predicates.

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

– We propose a framework of concurrent functional logic programming languages where concurrency is expressed by means of processes (or agents). The latter are described in process algebra style [1], 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) 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).

The rest of this extended abstract is organized as follows: In the next section, we outline an abstract syntax for a concurrent extension of functional logic languages. In Sect. 3, we sketch briefly a compositional semantics (based on labeled sequences) for the proposed languages. Finally Sect. 4 gives a brief comparison of our approach to related work. The details omitted in the Sect. 2 and 3 can be found in the full paper [6].

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., [8, 2]. Our approach of concurrent functional logic programs can be generalized easily to other frameworks such as Curry [11] or Escher [15]. 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 and SLDE-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 modeling 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:

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

In fact, the derivation (1) rather corresponds to a *run* (or behavior) 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 near 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 (suppose that the volume of the room is fixed), 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. Roughly speaking, a concurrent functional logic program $\mathcal{P}$ will consist of five components, $\mathcal{P} = \langle \Sigma^Q, \mathcal{P}rocs, cl_0, \mathfrak{a}_0, \sigma_0 \rangle$, where Σ^Q is a set of declarations (signature) of different entities used in $\mathcal{P}$, 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 the 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. More details about the proposed framework can be found in the full paper [6].

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* sorts, Ω *is a set of* operators *(or* functions*) and* Π *is a set of predicates including equality.*
- Q *is a set of* procedures *(or* processes*). We assume that* success *is an element of* Q_ε.
- Ξ *is a set of* "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 *(resp.,* V_O*,* V_L*,* V_C*) is a set of* internal variables *(resp.,* external variables*,* queue-variables*,* channel-variables*).*

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 internal variables (V_I) are similar to variables known in imperative programming languages: they are initialized before program-execution and are modifiable via affectation. 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. The channel-variables (V_C) are used to define a part of the communication with external machines. In this paper we do not consider how external machines implement the reception of messages. Whenever an external machine is modeled 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:

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

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 memorize 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.

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 $\mathcal{A}^{\mathrm{e}}(\Sigma^Q, V_{\mathrm{p}}, X)$ *is obtained by instantiating the parameters of the following five primitives:*

- Nop
- Tell(Cl)*, where* $Cl \in \mathcal{H}orn(\Sigma, X)$
- Del(Cl)*, where* $Cl \in \mathcal{H}orn(\Sigma, X)$
- $(v := t)$*, where* $v \in V_I$ *and* t *is a term* $(\in T(\Sigma, V_{\mathrm{p}}))$ *of the same sort as* v
- Send(ξ, A)*, where* ξ *is a machine* $(\in \Xi)$ *and* $A \in A(\Sigma, X)$ *an atom*
- Send($\xi, (v{:=}t)$)*, where* $\xi \in \Xi$*,* t *is a term and* v *is a channel-variable of the machine* ξ

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. ":=" affects the value[1] 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.

We call *action* a $(\in \mathcal{A}(\Sigma^Q, V_{\mathrm{p}}, X))$ a set of elementary actions which contains no "contradictions", e.g., a Tell and a Del of the same clause or a double affectation of an internal variable.

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

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).

In reference to process algebras [1], we call ; the operator of sequential composition, $\parallel$ the operator of parallelism and $+$ the operator of indeterministic 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 indeterminism is inadmissible. 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) $g_i \in \mathcal{G}(\Sigma, \{x_1; \ldots; x_n\})$, $a_i \in \mathcal{A}(\Sigma^Q, \{x_1; \ldots; x_n\}, X)$ *and* $\mathfrak{a}_i \in RPT(\Sigma^Q, \{x_1; \ldots; x_n\}, X)$. *The* x_i *are the (local) variables of the procedure* $\underline{\mathrm{p}}$.

Example 2. Consider the problem of the "Dining Philosophers." We model the situation with two predicates: stick(x) and is_eating(y). The former represents the fact that stick x is lying on the table, and the latter affirms that philosopher y is eating. Then we can model the behavior of a philosopher by the procedures shown in Fig. 1. Note that we do not use low-level synchronization (like semaphores) and auxiliary constructions (as placing the table in a separate room or asymmetric philosophers).

$$\underline{\mathrm{thinks}}(x,n) \Leftarrow \left[\begin{array}{l} \mathrm{stick}(x) \quad \wedge \\ \mathrm{stick}(x+1 \bmod n) \end{array} \Rightarrow \begin{array}{l} 8 \\ < \mathsf{Del}(\mathrm{stick}(x)) \\ \mathsf{Del}(\mathrm{stick}(x+1 \bmod n)) \\ \mathsf{Tell}(\mathrm{is\_eating}(x)) \end{array} \begin{array}{l} 9 \\ = \\ \\ , \end{array} \right]; \underline{\mathrm{eats}}(x,n)$$

$$\underline{\mathrm{eats}}(x,n) \Leftarrow [\mathrm{TRUE} \Rightarrow \begin{array}{l} 8 \\ < \mathsf{Del}(\mathrm{is\_eating}(x)) \\ \mathsf{Tell}(\mathrm{stick}(x)) \\ \mathsf{Tell}(\mathrm{stick}(x+1 \bmod n)) \end{array} \begin{array}{l} 9 \\ = \\ \\ , \end{array}]; \underline{\mathrm{thinks}}(x,n)$$

Fig. 1. Procedures for the dining philosophers example

Now we can define a concurrent functional logic program.

Definition 5. *A* concurrent functional logic program *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* initialization *(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 initialization. The former will not 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. $(\sigma_I)_0$ initializes the internal variables; the external and queue-variables are initialized by the environment. This distinction reflects the fact that the programmer cannot know 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 are defined by Fig. 2. The initialization 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 $\mathfrak{a}_0 = \underline{\text{stop}}$ and the initial value for the internal variable $(\sigma_I)_0 = \{T_{iavg} \mapsto 20\}$.

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

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

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

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

3 Compositional Semantics

The execution of a concurrent functional logic program $P = \langle \Sigma^Q, \mathcal{P}rocs, cl_0, \mathfrak{a}_0, (\sigma_I)_0\rangle$ informally consists, on the one hand, in running the procedures in $\mathcal{P}rocs$ starting from the initial agent $\mathfrak{a}_0$ and, on the other hand, in solving goals in the possibly different reachable theory presentations (e.g., search for the currently eating philosophers). The complete set of rules defining the operational semantics may be found in the full paper [6].

In this section we discuss briefly the basic ideas for constructing a compositional denotational semantics for our language-proposal. 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. We can define [6] a transition system $\mathfrak{Tr}(\mathcal{P})$ corresponding to $\mathcal{P}$ by using the inference rules describing the operational semantics. A first semantics for $\mathcal{P}$, denoted by $\mathcal{T}r(\mathcal{P})$, may be defined as the set of possible executions or traces of $\mathfrak{Tr}(\mathcal{P})$, such as in (1). An element in $\mathcal{T}r(\mathcal{P})$ 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 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: $\mathcal{T}r(\mathfrak{a}_1) = \mathcal{T}r(\mathfrak{a}_2) = \varnothing$, whereas $\mathcal{T}r(\mathfrak{a}_1 \parallel \mathfrak{a}_3) \neq \mathcal{T}r(\mathfrak{a}_2 \parallel \mathfrak{a}_3)$

The source of the non-compositionality is that sequences as (2) represent only the behavior 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 $\mathfrak{a}_3$. To overcome this defect, the semantics of a process $\mathfrak{a}$ has to take into account the possible contexts within which it may run.

Thus the new compositional semantics of $\mathcal{P}$, denoted by $\mathcal{T}r'(\mathcal{P})$, is the set of possible action sequences interleaved with assumptions about the behavior of parallel processes, i.e., the actions they may execute. Furthermore, the actions in the traces are labeled by their "author", in order to distinguish between the actions executed by the process and the assumptions made. In extension to [5] an action in our framework can be executed by the process itself (p), another process (executing in parallel) (o) or the environment, e.g., the reception of a message or the modification of a value captured by a sensor (e). The distinction between other processes and the environment is necessary because the actions of another process could be executed by the process itself, whereas the actions of the environment cannot. A run as (2) now becomes for example:

$$P_0 \xrightarrow{a_1^{\mathsf{p}}} P_1 \xrightarrow{a_2^{\mathsf{p}}} P_1' \xrightarrow{(a_2')^{\mathsf{o}}} P_2 \xrightarrow{a_3^{\mathsf{p}}} P_3 \xrightarrow{a_4^{\mathsf{p}}} P_4 \xrightarrow{(a_4')^{\mathsf{e}}} P_4' \cdots \tag{3}$$

Having defined the labels of the actions, it is possible to define semantic operators [6] corresponding to the operators for composing processes, in order to obtain compositionality. Remark that at a first sight, the traces in $\mathcal{T}r'(\mathcal{P})$ may seem to incorporate "too much" information; nevertheless this is the price to pay for compositionality.

4 Related Work

Due to lack of space we do not survey all the propositions in the area, but focus on some of those which are close to our proposal.

Concurrent extensions of functional programming languages as CML [18] and Concurrent Haskell (CH) [16] add to ML, resp., to Haskell, new *primitives* which can be used to "encode" concurrent systems. 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 operational semantics in order to allow interactive goal-solving. Also, there is no direct support for building atomic actions as we used in our modeling of the dining philosophers program.

LOTOS [12] 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. In addition, in LOTOS the function-definitions (theory presentations) cannot be modified.

Concurrent Constraint Programming (ccp) [20] integrates ideas from concurrent logic programming [22] and constraint logic programming [13]. 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 indeterministic choice with *priority* ($\oplus$) very useful for some examples. Unfortunately this operator does not exist in ccp. However, in a recent extension of ccp, [3], 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 due to the possibility of deleting some clauses (see e.g., Del). Some variants of ccp with non-monotonic stores have been proposed, see e.g., [4, 21, 7].
  In [4] 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 [4] the first

solution for the dining philosophers can only be understood with some knowledge about the constraint system.

Another different approach to handle non-monotonic stores is presented in [21, 7]. This approach is based on linear logic and allows the *implicit* deletion of information thanks to the semantics of the *linear* ask.

- In our proposition, we distinguish clearly between the behavior 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 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.

Prolog III offers the possibility of adding and removing clauses by predefined "predicates", as for example `assert` and `retract` [17]. However, these predicates are not declarative: their use implies the knowledge of the operational details of the language interpreter. In our opinion, these extensions are intended to ease interactive program development, and not programming concurrent systems.

AKL [14] introduces the notion of *ports* as a communication medium for processes described in concurrent contraint logic programming framework. Primitives on ports such as open_port(P, S) or send(P, m) have been introduced to describe message passing. It is argued in [14] that the introduced port primitives have a logical reading and preserve the monotonicity of the contraint 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 [10] in order to cope with distributed applications.

References

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

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

[3] F. S. de Boer and M. Gabbrielli. Modeling real-time in concurrent constraint programming. In *Proc. of the Int. Logic Programming Symposium*, 1995.

[4] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In *Proc. of the Int. Symp. on Logic Programming*, pages 315 – 334. The MIT Press, 1993.

[5] 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 the Int. Joint Conf. on Theory and Practice of Software Development, Volume 1, Colloquium on Trees in Algebra and Programming*, volume 493 of *LNCS*, pages 296 – 319, Brighton, UK, April 1991. Springer.

[6] R. Echahed and W. Serwe. A concurrent extension of functional logic programming languages. available at `ftp://ftp.imag.fr/pub/LEIBNIZ/PMP/conc_ext_flp.ps.gz`.

[7] F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proc. of LICS*, 1998.

[8] 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.

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

[10] M. Hanus. Distributed programming in Curry. In R. Echahed, editor, *Proc. of the* 8^{th} *Int. Workshop on Functional and Logic Programming*, pages 195 – 208, Grenoble, June 1999. Institut IMAG, RR-1021-1-.

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

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

[13] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of the* 14^{th} *Annual ACM Symp. on Principles of Programming Languages*, pages 111 – 119, München, January 1987. ACM.

[14] 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.

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

[16] S. L. Peyton Jones, A. D. Gorden, and S. Finne. Concurrent Haskell. In *Proc. of ACM Symp. on Principles of Programming Languages*, pages 295 – 308, St Petersburg Beach, Florida, January 1996.

[17] PrologIA. *PrologIII, Reference-Manual*.

[18] J. H. Reppy. CML: A higher-order concurrent language. In *Proc. of the Conf. on Programming Language Design and Implementation*, ACM SIGPLAN Notices, pages 293 – 305, Toronto, June 1991. ACM Press.

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

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

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

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