appeared in Proc. of 8^{th} Workshop on Functional and Logic Programming, Herbert Kuchen, editor, 1998

Towards Reactive Functional Logic Programming Languages

Jean-Jérémie Blanc Rachid Echahed Wendelin Serwe

Laboratoire LEIBNIZ – Institut IMAG, CNRS
46, avenue Felix Viallet, F-38031 Grenoble, France
Email: `<First-name>.<Last-name>@imag.fr`

Abstract

In this paper we present a new extension of functional logic programming languages which allows to specify, at a very high level, applications that need parallelism and reactivity with the environment. For that, we introduce the possibility of defining procedures (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 (algebraic), logic and parallel programming.

1 Introduction

Many functional logic languages have been proposed in the literature, see [8] for a survey and [10, 9] for recent language propositions. The aim of functional logic languages is to integrate, in a uniform way, the main advantages of functional languages and (constraint) logic languages. Roughly speaking, a functional logic program is a static entity. It consists of a presentation theory, defining data types, functions and possibly predicates, with which one may evaluate expressions or solve goals, concurrently or sequentially. So, a major limitation of functional logic programming languages is the impossibility to describe reactive systems, i.e. systems in (continuous) interaction with their environment.

Let us consider an (toy) example of such a reactive system which is a temperature-controller of a room. The goal of this system is to keep the temperature in a room, T_{env}, close to some given average temperature, T_{avg}, by giving orders to an external machine or physical device (PD) of the real world in order to cool the room down (if it is too hot, i.e., $T_{env} > T_{max}$ where $T_{max} = T_{avg} + 1$) or to heat (if it is too cold, i.e., $T_{env} < T_{min}$ where $T_{min} = T_{avg} - 1$). In addition, it should also be possible to modify the desired temperature, T_{avg}, of the room and to use the values of T_{avg} and T_{env} like any constant of a functional logic programming language. For instance, one may compute at any time the pressure in the room by evaluating the expression $p = \frac{mRT}{V}$.

A sample of an evolution of such a control system is given in Fig. 1: External events provoke (re-)actions of the controller and changes of the theory.

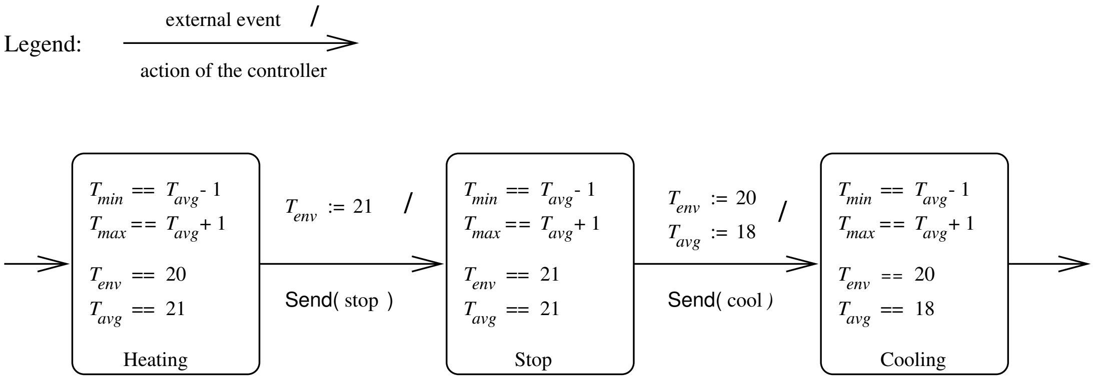

Figure 1: a sample of an evolution of the temperature control system

Why this system cannot be described by a classical functional logic program ? The answer is quite obvious: First, because a functional logic program (as a static entity) is not able to access the dynamically changing value of the temperature and second, because there is no possibility of communication with external machines.

In order to make functional logic programs reactive, our suggestion consists first in making possible the change of theories (sentences that define a program). Thus the evolution of a program can be described by means of an extended automaton or rather an extended PA process [3] when parallel operators are needed. Second, we also allow programs to exchange "messages" with their environments. The resulting language is an integration of functional logic languages and reactive and parallel languages. Our proposal is the result of several case studies such as the steam boiler [1], lift control system [2, page 358] and an aircraft sequencing problem [4].

The rest of the paper is organized as follows. In the next section, we introduce our abstract syntax for the integration of functional logic languages and reactivity. In Section 3, we describe an operational semantics of the proposed languages. Section 4 concludes the paper by a brief comparison of our proposal to the most related existing languages and gives some directions of future work.

2 Abstract Syntax

In this section we present the abstract syntax for reactive functional logic programming languages. Therefore, we introduce first the notion of a reactive signature, give some technical definitions and present finally the notion of a reactive functional logic program. We assume that the reader is familiar with the classical notions about Horn clause logic with equality (HCLE).

2.1 Reactive Signature

Definition 1 (Reactive signature)
We call reactive signature *a tuple* $\Sigma^Q = < S, \Omega, \Pi, Q, V_I, V_O, V_L, V_C, \Xi >$ *such that:*

- $< S, \Omega, \Pi >$ *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 with equality.*
- $V_I = \bigcup_{s \in S} (V_I)_s$ *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*,*
- $\forall v \in V_L, empty_v \in \Pi_\varepsilon$*,*
- $Q = \bigcup_{u \in S^*} Q_u$ *is a (*S^**-indexed) family of symbol-sets, called* processes*,* procedures *or* agents*,*
- <u>success</u> *is an element of* Q_ε*,*
- Ξ *is a set of symbols for "external machines" and*
- $V_C = \bigcup_{\xi \in \Xi} (V_C)^\xi$ *is a (*Ξ*-indexed) family of (*S*-indexed) families of symbol-sets, called* channel-variables*.*

We call *state variables* (V_{st}) the set of all internal, external and queue variables, i.e. $V_{st} = V_I \cup V_O \cup V_L$. The internal variables are similar to variables known in imperative programming languages: they are initialized at the beginning and modified only via affectation. The external variables are similar to sensors: they have a value which is determined by something outside the system. The queue-variables are used to model the possibility of receiving messages via a FIFO channel. The family $(V_C)^\xi$ of channel-variables is used for the communication with the external machine ξ (which might be a program).

Example 1 *According to the notations of Def. 1, we define the following reactive signature for the example of a temperature-controller:*

- $S = \{Nat; Mode\}$
- $\Omega = \{\ 0 :\longrightarrow Nat; succ : Nat \longrightarrow Nat; pred : Nat \longrightarrow Nat;$
  $T_{min} :\longrightarrow Nat; T_{max} :\longrightarrow Nat\}$
- $\Pi = \Pi_\varepsilon = \{\mathrm{heat}; \mathrm{stop}; \mathrm{cool}\}$
- $Q = Q_\varepsilon = \{\underline{\mathrm{Stop}}; \underline{\mathrm{Heating}}; \underline{\mathrm{Cooling}}\}$
- $V_I = (V_I)_{Nat} = \{T_{iavg}\}$
- $V_O = (V_O)_{Nat} = \{T_{env}\}$
- $V_L = (V_L)_{Nat} = \{T_{avg}\}$
- $V_C = \emptyset$
- $\Xi = \{\mathsf{PD}\}$

In Example 1 we can see the use of external variables for a sensor (T_{env}) and 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}. PD stands for Physical Device and represents the machine that will receive the orders stop, cool or heat from the temperature controller. Instead of those predicates the use channel-variables would have been also possible; but we found it less readable.

2.2 Guards, Actions, Process Terms and Nodes

In this subsection we give some technical definitions necessary to define the syntax of a reactive functional logic program (see Sect. 2.3). Throughout this subsection, we suppose given a reactive signature. In the following, we denote by $T(\Sigma, X)$ (respectively, $A(\Sigma, X)$ and $\mathcal{H}orn(\Sigma, X)$) the set of terms (respectively, atoms and (definite) Horn clauses) build from the (first-order) signature $\Sigma = < S, (\Omega \cup V_{st}), \Pi >$ with variables X.

Definition 2 (guard) *We call* guard *a (finite) conjunction of atoms. We denote by $\mathcal{G}(\Sigma, V_l)$ the set of guards (over state variables V_{st}, which are part of Σ, and a set of (local) variables V_l).*

Definition 3 (elementary action) *We define the set $\Delta(\Sigma^Q, V_C, V_l, X)$ of* elementary actions *over the set of variables V_l and X, by:*

1. $\mathsf{Nop} \in \Delta(\Sigma^Q, V_C, V_l, X)$
2. $\forall Cl \in \mathcal{H}orn(\Sigma, X)$, $\mathsf{Tell}(Cl) \in \Delta(\Sigma^Q, V_C, V_l, X)$
3. $\forall Cl \in \mathcal{H}orn(\Sigma, X)$, $\mathsf{Del}(Cl) \in \Delta(\Sigma^Q, V_C, V_l, X)$
4. $\forall x \in V_I, \forall e \in T(\Sigma, V_l)$, $(x := e) \in \Delta(\Sigma^Q, V_C, V_l, X)$
5. $\forall \xi \in \Xi, \forall A \in A(\Sigma, X)$, $\mathsf{Send}(\xi, A) \in \Delta(\Sigma^Q, V_C, V_l, X)$
6. $\forall \xi \in \Xi, \forall x \in (V_C)^\xi, \forall e \in T(\Sigma, V_l)$, $\mathsf{Send}(\xi, (x := e)) \in \Delta(\Sigma^Q, V_C, V_l, X)$

$\Delta(\Sigma^Q, V_C, V_l, X)$ is the smallest set (for inclusion) satisfying these six properties.

Nop is the invisible action without any effect. The elementary actions Tell and Del are dual and are used to add and delete a clause from the current theory. ":=" represents the affectation of internal variables. Finally, Send allows the communication with the external machines via message-passing. There is no elementary action dedicated to updates of external and queue-variables: their management is supposed to be external (or on a meta-level).

Definition 4 (action) *A (guarded)* action *a is a tuple $(g \Rightarrow \alpha)$ consisting of a guard g and a set α of elementary actions satisfying the following two conditions:*

- *for all $\mathsf{Tell}(Cl), \mathsf{Del}(Cl') \in \alpha$, Cl is not a variant (equal up to renaming) of Cl'*
- *for all $(x := e)$, $(y := f) \in a$, $x \neq y$.*

We denote by $\Delta^*(\Sigma^Q, V_C, V_l, X)$ the set of actions.

These two conditions imply that the set of elementary actions of an action is not contradictory: so we do not allow a double affectation, or the simultaneous addition and removal of a same formula (modulo renaming of bounded variables). Therefore the order of the elementary actions in an action is not significant, and one can imagine an action as "atomic."

In the sequel, we will abbreviate actions of the type $(true \Rightarrow \alpha)$ by just α.

Definition 5 (atomic process) *The set $AP(\Sigma^Q, X)$ of* atomic processes *is defined by:*

$$AP(\Sigma^Q, X) = \{\underline{\mathrm{q}}(t_1, \ldots, t_n) \mid t_1 \in T_{s_1}(\Sigma, X), \ldots, t_n \in T_{s_n}(\Sigma, X) \ \textit{and}\ \underline{\mathrm{q}} \in Q_{s_1 \ldots s_n}\}$$

Definition 6 (process term) *The set $PT(\Sigma^Q, V_l, X)$ of* process terms *is defined inductively by:*

1. *If $\underline{\mathrm{q}}(t_1, \ldots, t_n) \in AP(\Sigma^Q, V_l)$ then $\underline{\mathrm{q}}(t_1, \ldots, t_n) \in PT(\Sigma^Q, V_l, X)$,*
2. *If $a = (g \Rightarrow \alpha) \in \Delta^*(\Sigma^Q, V_C, V_l, X)$ then $a \in PT(\Sigma^Q, V_l, X)$,*
3. *If $t_{E1}, t_{E2} \in PT(\Sigma^Q, V_l, X)$ then $(t_{E1}; t_{E2}) \in PT(\Sigma^Q, V_l, X)$,*
4. *If $t_{E1}, t_{E2} \in PT(\Sigma^Q, V_l, X)$ then $(t_{E1} \parallel t_{E2}) \in PT(\Sigma^Q, V_l, X)$,*
5. *If $t_{E1}, t_{E2} \in PT(\Sigma^Q, V_l, X)$ then $(t_{E1} + t_{E2}) \in PT(\Sigma^Q, V_l, X)$,*
6. *If $t_{E1}, t_{E2} \in PT(\Sigma^Q, V_l, X)$ then $(t_{E1} \oplus t_{E2}) \in PT(\Sigma^Q, V_l, X)$.*

In reference to the Process Algebras, we call ; the operator of sequential composition, $\parallel$ the operator of parallelism (or merge operator) and $+$ the operator of indeterministic choice (or operator of alternative composition). 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 not admissible[1].

Notations

- *The set $PT(\Sigma^Q)$ of* closed atomic processes *is denoted by $AP(\Sigma^Q) = AP(\Sigma^Q, \emptyset)$*
- *The set of* restricted process terms, *denoted by $RPT(\Sigma^Q, V_l, X)$, represents the set of process terms constructed only with atomic processes and the operators ;, $\parallel$ and $+$.*

Nodes describe the behavior of processes (or agents).

Definition 7 (node) *A* node *is the defining expression for a process, i.e.*

$$\underline{\mathrm{p}}(x_1, \ldots, x_n) \Leftarrow \bigoplus_{i=1}^{m} (a_i; t_{Ei}),$$

where (for each i) the action $a_i = (g_i \Rightarrow \alpha_i) \in \Delta^*(\Sigma^Q, V_C, \{x_1; \ldots; x_n\}, X)$ and the process terms $t_{Ei} \in RPT(\Sigma^Q, \{x_1; \ldots; x_n\}, X)$. The x_i's are the (local) variables of the node $\underline{\mathrm{p}}$.

[1]Nevertheless, we will keep $+$, because it is needed to model the interleaving semantics of $\parallel$

We remark that in Def. 7 the operator $\oplus$ is applied only to "guarded process terms" ; the operational semantics (see section 3.2) for $\oplus$ will therefore take in account only such process terms.

In the sequel we will use the following notation for nodes:

$$\boxed{\begin{array}{l} \textsf{Node } \underline{\mathrm{p}}(x_1,\ldots,x_n) \\ \quad (g_1 \Rightarrow a_1);\ t_{E1} \\ \quad\quad \vdots \\ \quad (g_m \Rightarrow a_m);\ t_{Em} \\ \textsf{End } \underline{\mathrm{p}} \end{array}}$$

2.3 Program

With all these definitions of the preceeding subsection, we are now able to define a reactive functional logic program.

Definition 8 (reactive functional logic program) *A* reactive functional logic program *is a tuple* $< \Sigma^Q, \mathcal{N}odes, (cl^*)_0, (t_E)_0, \sigma_0 >$ *such that:*

- Σ^Q *is a reactive signature,*
- $\mathcal{N}odes$ *is the set of all nodes of the program,*
- $(cl^*)_0$ *is a set of Horn clauses with equality, called the* initial clause set*,*
- $(t_E)_0 \in PT(\Sigma^Q)$ *is the* initial process term*,*
- σ_0 *is a total substitution (i.e. valuation) over* V_I*, called* initialization *of the internal variables.*

This definition distinguishes two parts of a reactive functional logic program: a static one (Σ^Q and $\mathcal{N}odes$) and an initialization. The former will not be modifiable neither by the actions nor by the environment. The latter describes everything which is modifiable. $< \Sigma = < S, (\Omega \cup V_{st}), \Pi >, (cl^*)_0 >$ is the initial theory which can be changed via the addition and the removal of clauses. σ_0 initializes the internal variables; the external and queue-variables are initialized by the initial configuration. This distinction reflects that the programmer cannot know the initial environment. $(t_E)_0$ is the initial process which will modify the state of the system via the application of transition rules.

2.4 Examples

Temperature Controller.

Example 2 *We give here a program corresponding to the example mentioned in the introduction. We take for* Σ^Q *the reactive signature defined in Example 1. The set of nodes is defined by Fig. 2.*

The initializations of the program are defined as follows:

- $(cl^*)_0 = \{pred(succ(x)) == x; T_{min} == pred(T_{iavg}); T_{max} == succ(T_{iavg})\}$

- $(t_E)_0 = \underline{\text{Stop}}$
- $\sigma_0 = \{T_{iavg} \mapsto 20\}$

$$
\begin{array}{|l|}
\hline
\textsf{Node } \underline{\text{Stop}} \\
\qquad (T_{env} < T_{min} \Rightarrow \textsf{Send}(\textsf{PD}, \text{heat})); \underline{\text{Heating}} \\
\qquad (T_{env} > T_{max} \Rightarrow \textsf{Send}(\textsf{PD}, \text{cool})); \underline{\text{Cooling}} \\
\qquad (true \Rightarrow (T_{iavg} := T_{avg})); \underline{\text{Stop}} \\
\textsf{End } \underline{\text{Stop}} \\
\\
\textsf{Node } \underline{\text{Heating}} \\
\qquad (T_{env} > T_{max} \Rightarrow \textsf{Send}(\textsf{PD}, \text{cool})); \underline{\text{Cooling}} \\
\qquad (T_{env} \geq T_{iavg} \Rightarrow \textsf{Send}(\textsf{PD}, \text{stop})); \underline{\text{Stop}} \\
\qquad (true \Rightarrow (T_{iavg} := T_{avg})); \underline{\text{Heating}} \\
\textsf{End } \underline{\text{Heating}} \\
\\
\textsf{Node } \underline{\text{Cooling}} \\
\qquad (T_{env} < T_{min} \Rightarrow \textsf{Send}(\textsf{PD}, \text{heat})); \underline{\text{Heating}} \\
\qquad (T_{env} \leq T_{iavg} \Rightarrow \textsf{Send}(\textsf{PD}, \text{stop})); \underline{\text{Stop}} \\
\qquad (true \Rightarrow (T_{iavg} := T_{avg})); \underline{\text{Cooling}} \\
\textsf{End } \underline{\text{Cooling}} \\
\hline
\end{array}
$$

Figure 2: set of nodes for the example of a temperature controller

Dining Philosophers.

Another notion not covered by classical functional logic programming languages is parallelism. This notion is common in reactive systems, but does not appear in the simple example of a temperature-controller. To keep the examples small, we have chosen the classical example of the dining philosophers to illustrate the facilities of our extension for parallelism.

The problem is as follows: n (Chinese) philosophers are sitting round a table. Their only occupations are to eat and to think. If they want to eat, they need the two sticks from both sides of their plate, but there are only n sticks on the table, so that a philosopher has to share the sticks with his neighbors.

Example 3 *We do not present the complete definition of this example, but only the definitions of the different nodes (see Fig. 3).*

We model the situation with two predicates: stick(x) and is_eating(x). The former represents the fact that stick x is lying on the table, and the latter affirms that philosopher x is eating. The three nodes model the starting off (Init) and the two states of a philosopher (thinks and eats). So, the initial theory contains all the definitions about mod, $+$, *etc. and the initial process term is* $\underline{\text{Init}}(1, n)$.

This example shows also the usefulness of the atomicity of an action: It is possible to solve the dining philosophers problem without an arbiter or the use of semaphores.

```
Node thinks(x, n)
    (stick(x) ∧ stick(x + 1 mod n) ⇒
        {Del(stick(x)); Del(stick(x + 1 mod n)); Tell(is_eating(x))}); eats(x, n)
End thinks

Node eats(x, n)
    (true ⇒
        {Del(is_eating(x)); Tell(stick(x)); Tell(stick(x + 1 mod n))}); thinks(x, n)
End eats

Node init(x, n)
    (x == n ⇒ Nop); success
    (x ≤ n ⇒ Tell(stick(x))); (init(x + 1, n) ∥ thinks(x, n))
End init
```

Figure 3: set of nodes for the dining philosophers example

3 Operational Semantics

In this section we give the operational semantics of the reactive functional logic language sketched in the previous section. The ultimate operational semantics is defined by a transition system which integrates the rules describing theory changes and the rules dedicated to resolution of goals.

Thus, we first define in subsection 3.2 a transition system which models only the evolution of a system in its environment without taking care of resolution of goals. For this purpose, we begin by some preliminary definitions in the next subsection, such as the notion of a configuration, and the semantics of the actions.

3.1 Configuration and Interpretation of Actions

Definition 9 (configuration) *A* configuration *is a tuple* $< t_E, cl^*, \sigma_I, \sigma_O, L >$ *such that:*

- $t_E \in PT(\Sigma^Q)$
- cl^* *is a set of definite Horn clauses with equality*
- σ_I *is a valuation for* V_I
- σ_O *is a valuation for* V_O
- L *is an application defining the content of the queues corresponding to the queue-variables* V_L

An atomic configuration *is configuration* $< t_E, cl^*, \sigma_I, \sigma_O, L >$ *such that* $t_E \in AP(\Sigma^Q)$.

A configuration represents the state of the system: t_E is the current process and $< cl^*, \sigma_I, \sigma_O, L >$ is the current theory.

Definition 10 (validity in a configuration) *Let $\mathcal{C} = < t_E, cl^*, \sigma_I, \sigma_O, L >$ be a configuration and A an atom.*

- *If $A = empty_v$, A is valid in $\mathcal{C}$ iff $L(v) = nil$.*
- *If $A \neq empty_v$, A is 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 classical validity relation of HCLE.*

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

An action $a = (g \Rightarrow \alpha)$ is called *executable* in a configuration $\mathcal{C}$, if $\mathcal{C} \models g$ and for all queue-variables x appearing in a, $L(x) \neq nil$. To be able to define the interpretation of an action, we introduce the notion of a store.

Definition 11 (store) *A* store *is a quintuple $< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >$ such that:*

- *cl^* is a set of definite Horn clauses with equality,*
- *$\sigma_I, \sigma_O, \sigma_l$ are valuations for the internal, external and local variables,*
- *σ_L is a substitution for the queue-variables.*

In the sequel, $\mathcal{S}tore$ will represent the set of stores.

In the sequel we will use the following notation for functions: Let $f_1, f_2 : E \longrightarrow D$ be two partial functions having the same profile. We define the partial function $f_1 \lhd f_2 : E \longrightarrow D$ by:

$$(f_1 \lhd f_2)(x) = \begin{cases} f_2(x) \text{ if } f_2(x) \text{ is defined} \\ f_1(x) \text{ otherwise} \end{cases}$$

Definition 12 (interpretation of elementary actions) *Let $\sigma = \sigma_I \cup \sigma_O \cup \sigma_L \cup \sigma_l$. We define the partial function $do : \Delta(\Sigma^Q, V_C, V_l, X) \longrightarrow (\mathcal{S}tore \longrightarrow \mathcal{S}tore)$ by:*

- $do(\mathsf{Nop})(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >$
- $do(\mathsf{Tell}(Cl))(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^* \cup \sigma_L(\sigma_l(Cl)), \sigma_I, \sigma_O, \sigma_L, \sigma_l >$
- $do(\mathsf{Del}(Cl))(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^* \backslash \overline{\sigma_L(\sigma_l(Cl))}, \sigma_I, \sigma_O, \sigma_L, \sigma_l >$ *where $\overline{cl}$ stands for the set of clauses obtained from cl via variable renaming.*
- $do((x := e))(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^*, \sigma_I \lhd \{x \mapsto (\sigma(e))!\}, \sigma_O, \sigma_L, \sigma_l >$ *where $t!$ stands for a normal (simplified) form of the term t.*
- $do(\mathsf{Send}(\xi, A))(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >$
- $do(\mathsf{Send}(\xi, (x := e)))(< cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >) = < cl^*, \sigma_I, \sigma_O, \sigma_L, \sigma_l >$

The extension $\tilde{do}$ of the function do to sets of elementary actions is straightforward.

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.

3.2 Transition System

With these definitions we can give now the transition system describing the part of the operational semantics concerning the changes of the theory and the communication with the environment.

The first rule expresses the call of an atomic process which is immediately replaced by its defining expression.

$$\text{(TP1)}\ \frac{}{< \underline{\mathrm{p}}(t_1,\ldots,t_n), cl^*, \sigma_I, \sigma_O, L > \longrightarrow < \bigoplus_{i=1}^{m} \hat{\sigma}_l(a_i;t_{Ei}), cl^*, \sigma_I, \sigma_O, L >}$$

if: 1. $\underline{\mathrm{p}}(x_1,\ldots,x_n) \Leftarrow \bigoplus_{i=1}^{m}(a_i;t_{Ei}) \in \mathcal{N}odes$

2. $\sigma_l = \{x_k \mapsto t_k \mid k \in \{1;\ldots;n\}\}$

The next rule describes the execution of an action: the store is changed via the function $\tilde{do}$.

$$\text{(TP2)}\ \frac{}{< (g \Rightarrow \alpha), cl^*, \sigma_I, \sigma_O, L > \longrightarrow < \underline{\mathrm{success}}, cl^{*\prime}, \sigma_I', \sigma_O, L' >}$$

if: 1. $(g \Rightarrow \alpha) \in \Delta^*(\Sigma^Q, V_C, \emptyset, X)$

2. $< (g \Rightarrow \alpha), cl^*, \sigma_I, \sigma_O, L > \models g$

3. $\forall x \in \mathcal{V}_L(\alpha) \cup \mathcal{V}_L(g),\ L(x) \neq nil$

4. $\tilde{do}(\alpha)(< cl^*, \sigma_I, \sigma_O, \sigma_L, \emptyset >) = < cl^{*\prime}, \sigma_I', \sigma_O, \sigma_L, \emptyset >$

5. $L' = L \lhd \{v \mapsto tail(l) \mid v \in \mathcal{V}_L(\alpha) \cup \mathcal{V}_L(g)\ and\ l = L(v)\}$

For the following inference rules, we suppose that the processes t_E and t_E' are different.

The two following rules describe the sequential execution (in its classical sense).

$$\text{(TP3)}\ \frac{< t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}', cl^{*\prime}, \sigma_I', \sigma_O, L' >}{< t_{E1};t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}';t_{E2}, cl^{*\prime}, \sigma_I', \sigma_O, L' >}\ if\ t_{E1}' \neq \underline{\mathrm{success}}$$

$$(\mathrm{TP3}')\ \frac{< t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < \underline{\mathrm{success}}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1}; t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

The four next rules describe the parallelism as interleaving.

$$(\mathrm{TP4l})\ \frac{< t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} \parallel t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E1} \parallel t_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}\ \textit{if } t'_{E1} \neq \underline{\mathrm{success}}$$

$$(\mathrm{TP4r})\ \frac{< t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} \parallel t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1} \parallel t'_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}\ \textit{if } t'_{E2} \neq \underline{\mathrm{success}}$$

$$(\mathrm{TP4l}')\ \frac{< t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < \underline{\mathrm{success}}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} \parallel t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

$$(\mathrm{TP4r}')\ \frac{< t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < \underline{\mathrm{success}}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} \parallel t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

The choice operator $+$ is associative and commutative.

$$(\mathrm{TP5l})\ \frac{< t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} + t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

$$(\mathrm{TP5r})\ \frac{< t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< t_{E1} + t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t'_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

The (intuitive) semantics of the process term $a_1; t_{E1} \oplus a_2; t_{E2}$ is: "execute $a_2; t_{E2}$ iff it is impossible to execute $a_1; t_{E1}$." $(a_1, a_2 \in \Delta^*(\Sigma^Q, V_C, V_l, X))$

$$(\mathrm{TP6l})\ \frac{< a_1; t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< a_1; t_{E1} \oplus a_2; t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

$$(\mathrm{TP6r})\ \frac{< a_2; t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}{< a_1; t_{E1} \oplus a_2; t_{E2}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E2}, cl^{*\prime}, \sigma'_I, \sigma_O, L' >}$$

if there are no $cl^{*\prime\prime}$, σ''_I and L'' such that
$< a_1; t_{E1}, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_{E1}, cl^{*\prime\prime}, \sigma''_I, \sigma_O, L'' >$

Rule (TP7) models the changes of external variables by the environment.

$$(\mathrm{TP7})\ \frac{}{< t_E, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_E, cl^*, \sigma_I, \sigma'_O, L >}$$

for an external variable v_O and a value v such that $\sigma'_O = \sigma_O \lhd \{v_O \mapsto v\}$

The arrival of new values at the end of the queues associated with the queue-variables is described by rule (TP8).

$$\text{(TP8)}\ \frac{}{< t_E, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_E, cl^*, \sigma_I, \sigma_O, L' >}$$

for a queue-variable v_L *and a value* v *such that* $L' = L \lhd \{v_L \mapsto L(v_L) \ll v\}$

The last rule (TP9) describes the receipt of an atom, which is supposed to be sent by a Send action.

$$\text{(TP9)}\ \frac{}{< t_E, cl^*, \sigma_I, \sigma_O, L > \longrightarrow < t_E, cl^{*\prime}, \sigma_I, \sigma_O, L >}$$

where $cl^{*\prime}$ *is build from* cl^* *by adding an atom* A

3.3 Operational Semantics of a Program

For a reactive functional logic program $P = < \Sigma^Q, \mathcal{N}odes, (cl^*)_0, (t_E)_0, (\sigma_I)_0 >$, the operational semantics is composed of two parts: the transition system describing the state changes (given in Sect. 3.2) and the resolution of goals, based a complete system for functional logic programs such as SLDE-resolution, Narrowing, Constrained SLD, etc. Thus, the operational semantics of a reactive functional logic program, P, is defined by a new transition system that combines the dynamic rules which change the theories with the "static" inference rules that solve goals. The resulting transition system is a triple $< \mathcal{Q}, \longmapsto, q_0 >$ where $\mathcal{Q} = (\mathcal{C}onfig(\Sigma^Q, X)) \times (\mathcal{G}oal \times \mathcal{G}oal \times \mathcal{S}ubst)$ with $\mathcal{S}ubst$ the set of substitutions over $T(\Sigma, X)$, $q_0 \in \mathcal{Q}$ is the initial state of the system of the form $q_0 = < \mathcal{C}_0, < G, G, \emptyset >>$, where $\mathcal{C}_0$ is an initial configuration for P, i.e. $\mathcal{C}_0 = < (t_E)_0, (cl^*)_0, (\sigma_I)_0, (\sigma_O)_0, \emptyset >$ ($(\sigma_O)_0$ being a valuation of the external variables) 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. $G_1 \leadsto_\sigma G_2$ means that the new goal G_2 is deduced by the inference rules of a functional logic language.

$$\text{(STA)}\ \frac{G_1 \leadsto_\sigma G_2}{< \mathcal{C}, < G, G_1, \vartheta >> \longmapsto < \mathcal{C}, < G, G_2, \sigma \circ \vartheta >>}$$

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. 3.2.

$$\text{(DYN)}\ \frac{\mathcal{C} \longrightarrow \mathcal{C}'}{< \mathcal{C}, < G, G', \vartheta >> \longmapsto < \mathcal{C}', < G, G, \emptyset >>}$$

The rule (DYN) suggests to restart solving a goal from scratch whenever a change in the current theory occurs (because the change can invalidate the deductions already done). It is obvious that rule (DYN) is a simple approach. Nevertheless, the amount of redundant calculus could probably be limited by a more sophisticated modification of the "resolution tree", like in [7].

4 Conclusion

We have presented an extension of functional logic languages which allows to model concrete applications where parallelism and reactivity are necessary. In our approach, we have used the 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.

Several extensions of declarative languages allowing the description of parallel or reactive programs exist in the literature. Our approach is very close to Concurrent Constraint Programming (CCP) [11] and Reactive Constraint Logic Programming (CLP) [7].

CCP integrates ideas from concurrent logic programming and constraint logic programming. Within the CCP framework, it is possible to define processes (agents) communicating via a common store. These processes are defined according to the following grammar

$$A ::= \mathbf{Success} \mid \mathbf{Fail} \mid \mathbf{ask}(c) \rightarrow A \mid \mathbf{tell}(c) \rightarrow A \mid A + A \mid A \parallel A \mid \exists X.A \mid p(\vec{t})$$

where $p(\vec{t})$ corresponds to a procedure call. Definitions of procedures is given by sentences of the form $p(\vec{x})$:- A. Obviously our nodes play the same role as CCP procedures. The guards within the definition of nodes correspond to **ask**. Nevertheless, there are some noteworthy differences between them, we quote for instance:

- We found the operator of indeterministic choice with priority very useful to describe some examples. Unfortunately this operator does not exits in CCP. Hence, the semantics of CCP processes and ours are different.
- A store in CCP is monotonic, whereas in our approach this is not the case due to the possibility of deleting some clauses (see the actions Del and :=). Some variants of CCP with non-monotonic stores exist. We quote for instance [5] where an operator $update_x$ is used to hide the name of a variable. Our Del operator is more precise and thus allows us to describe for instance the example of philosophers without special requirements on the structure of the constraint system: in [5] the first solution for the dining philosophers can be understood only with some knowledge about the constraint system.
- State variables are necessary for many real-word applications. Unfortunately, there is no counterpart of state variables in CCP.

In [7] a reactive approach of constraint logic programs is proposed. The authors give some hypothesis under which the Constrained SLD tree could be modified whenever some changes occur in the initial goal. Clearly, the approach in [7] corresponds to an enhancement of our rule (DYN). Indeed, the rule (DYN) concerns the resolution of goals (sequentially or concurrently) in a dynamic environment.

In this paper, due to lack of space, we did not present a denotational semantics. We did not either introduce the quantitative notion of time which is needed in some real-time applications.

Furthermore, the trace-semantics induced directly by the transition system presented in Sect. 3.2 is not compositional. This can easily be seen via the following example.

Example 4 *Let us assume that the initial store is empty, and consider the following three processes:*

$$t_{E1} = P \Rightarrow \underline{\text{success}}, \quad t_{E2} = R \Rightarrow \underline{\text{success}} \quad \textit{and} \quad t_{E3} = \mathsf{Tell}(P); \underline{\text{success}}$$

Then we have: $\mathcal{T}races(t_{E1}) = \mathcal{T}races(t_{E2})$, *but* $\mathcal{T}races(t_{E1} \parallel t_{E3}) \neq \mathcal{T}races(t_{E2} \parallel t_{E3})$

A possibility to overcome this default described in the context of concurrent constraint programming ([6]) is to introduce the notion of labeled sequences, and to provide a new transition rule which allows a process to make assumptions about the behavior of other processes executed in parallel. We are working on such a compositional semantics based on labeled sequences.

References

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

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

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

[4] L. Bianco, G. Renaldi, and A. Sassano. A combinatorial optimization approach to aircraft sequencing problem. Technical Report R. 182, Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185 Roma, Italy, April 1987.

[5] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. Non-monotonic concurrent constraint programming. In M. Press, editor, *Proceedings of the International Symposium on Logic Programming*, pages 315 – 334, 1993.

[6] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 1, Colloquium on Trees in Algebra and Programming*, number 493 in Lecture Notes in Computer Science, pages 296 – 319, Brighton, UK, April 1991. Springer Verlag.

[7] F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. to appear in The Journal of Logic Programming 1998.

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

[9] M. Hanus, S. Antoy, H. Kuchen, F. J. López-Fraguas, and F. Steiner. Curry: An integrated functional logic language. available at `http://www-i2.informatik.rwth-aachen.de/~hanus/curry/report.html`, 1998.

[10] J. W. Lloyd. Programming in an integrated functional and logic language. to appear in The Journal of Functional an Logic Programming.

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