Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

F.S. de Boer, C. Palamidessi

A fully abstract model for concurrent logic languages

Computer Science/Department of Software Technology Report CS-R9046 December

Bibliotheek
Centrum voor Wiskunde en Informatica
msterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

A Fully Abstract Model for Concurrent Logic Languages

Frank S. de Boer[1] and Catuscia Palamidessi[2]

[1]Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
email: wsinfdb@tuewsd.win.tue.nl

[2]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy
email: katuscia@dipisa.di.unipi.it

[2]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl

Abstract

One of the main aims of this paper is to show that the nature of the communication mechanism of concurrent logic languages is essentially different from the classical paradigms of CCS and TCSP. We define indeed a compositional semantics based on linear sequences, whilst more complicated structures, like trees and failure sets, are needed to model compositionally CCS and TCSP. Moreover, we prove that this semantics is fully abstract, namely that the information encoded by these sequences is necessary.

Our observation criterium consists of all the finite results, i.e. the computed constraint together with the termination mode (success, failure, or deadlock). The operations we consider are the parallel composition of goals and the union of neatly intersecting programs. We define a compositional operational model delivering sequences of input-output constraints, and we obtain a fully abstract denotational semantics by requiring additionally some closure conditions, that model the monotonic nature of communication in concurrent constraint languages.

C.R. Categories: D.1.3, D.3.1, F.1.2, F.3.2, F.4.1.

Key Words and Phrases: concurrent logic languages, constraints, operational semantics, denotational semantics, compositionality, full abstraction.

Note: Part of this work was carried out in the context of the ESPRIT Basic Research Action (3020) Integration. The research of Frank S. de Boer was partially supported by the Dutch REX (Research and Education in Concurrent Systems) project. The stay of Catuscia Palamidessi at the Centre for Mathematics and Computer Science was partially supported by the Italian CNR (Consiglio Nazionale delle Ricerche).

1 Introduction

This paper addresses the problem of a compositional and fully abstract semantics for concurrent logic languages. Compositionality is considered one of the most desirable characteristics of a formal semantics, since it provides a basis for program verification and modular design. The difficulty in obtaining this property depends upon the *operators* of the language, the behaviour we want to describe (*observables*), and the degree of *abstraction* we want to reach. A compositional model is

Report CS-R9046
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

called *fully abstract* (with respect to some operators and observables) if it identifies programs that behave in the same way under all the possible contexts. A fully abstract model can be considered to be *the* semantics of a language since all the other compositional semantics can be reduced to it by abstracting from the redundant information. Full abstraction is important, for instance, to decide correctness of program transformation techniques. If a fully abstract model distinguishes the transformed program from the original one then the transformation is not correct (in the sense that it does not preserve the same behaviour under composition).

The basic operators of a logic language are the conjunction of goals and the union of clauses. The observables usually consist of the termination mode (success or failure), and the computed answer substitution. For concurrent logic languages compositionality has been studied mainly with respect to the conjunction of goals, whilst union of clauses has been considered only in the simple case of *neatly intersecting* programs [9]. This is rather natural since in a concurrent framework the main operation is the parallel composition of processes. On the other hand, the class of observables has to be enriched by *deadlock*.

The compositional description of deadlock is one of the main semantic problems of concurrent languages. For languages like CCS and TCSP it is well-known that (linear) sequences are not sufficient. On the other hand, trees encode redundant branching information. In order to abstract from it two main approaches have been proposed. One is based on equivalence relations on trees (for instance, bisimulation [16]), and the other on grouping the *branching information* in sets (for instance, *failure sets* [2]). In general, failure set semantics is more abstract than bisimulation and it is proved to be fully abstract for TCSP and CCS.

With respect to compositionality, concurrent logic languages have been regarded just as a particular case of the classic paradigms. Therefore, the problem has been approached by the standard methods. De Bakker and Kok [3, 14] and De Boer et al. [4, 5] use tree-like structures labeled with functions on substitutions. More simple tree-like structures, labeled by constraints, are used by Gabbrielli and Levi [10] and by Saraswat and Rinard [19], who also define an equivalence based on bisimulation. Gerth et al. [9] and Gaifman et al. [11] approach the problem of full abstraction by refining the failure set semantics of TCSP.

We think that concurrent logic languages require a different approach. In this paper we study the language defined in [11], that can be considered as a special case of concurrent constraint programming [18, 19]. This paradigm represents a considerable improvement with respect to "classical" concurrent logic languages. The notion of constraint [13] allows on one side to increase the expressiveness, and, on the other side, to model in a logical manner the synchronization mechanism [15, 18]. As discussed in [18], the notion of *store* in constraint programming leads naturally to a new paradigm for concurrent programming. All processes share a common store, that represents the constraint established until that moment. Communication is modeled by adding (telling) consistently some constraint to the store. Synchronization is achieved by checking (asking) if the store entails (implies) a given constraint, if not, the process suspends.

It is interesting to compare this logic paradigm with CCS. We can translate CCS by interpreting the action a as telling the constraint $x = a$, and the complementary action $\bar{a}$ as asking if $x = a$ is entailed by the store. The main difference is that complementary actions do not synchronize anymore. Indeed, telling a constraint will never suspend. In other words, the communication mechanism of concurrent logic languages is intrinsically *asynchronous*. The following example shows that this leads to an essentially different deadlock behaviour.

Example 1.1 *Let* $p_1 = \bar{a}\bar{b} + \bar{a}\bar{c} + \bar{a}\bar{d}$ *and* $p_2 = \bar{a}\bar{b} + \bar{a}(\bar{c} + \bar{d})$. *In any compositional semantics for CCS these two processes must be distinguished. Indeed, they behave differently under the context* $p = a(b + c)$. *The process* p_1 *can deadlock, by choosing the third branch, whilst* p_2 *cannot. In the formalism of [11], this example can be translated as follows.*

$$\{\ p_1(x,y) \leftarrow ask(x=a) \mid ask(y=b).$$
$$p_1(x,y) \leftarrow ask(x=a) \mid ask(y=c).$$
$$p_1(x,y) \leftarrow ask(x=a) \mid ask(y=d).\ \}$$

$$
\begin{array}{ll}
\{\ p_2(x,y) \leftarrow ask(x=a) \mid ask(y=b). & \\
\ \ p_2(x,y) \leftarrow ask(x=a) \mid p_3(y). & \\
\ \ p_3(y) \leftarrow ask(y=c) \mid . & \\
\ \ p_3(y) \leftarrow ask(y=d) \mid . & \}
\end{array}
$$

$$
\begin{array}{ll}
\{\ p(x,y) \leftarrow tell(x=a) \mid p'(y). & \\
\ \ p'(y) \leftarrow tell(y=b) \mid . & \\
\ \ p'(y) \leftarrow tell(y=c) \mid . & \}
\end{array}
$$

In this translation, both p_1 and p_2 have the same behaviour. The process p_2 can deadlock by choosing the second clause, because p can independently decide to produce $y = b$ (after $x = a$). Figure 1 illustrates this example.

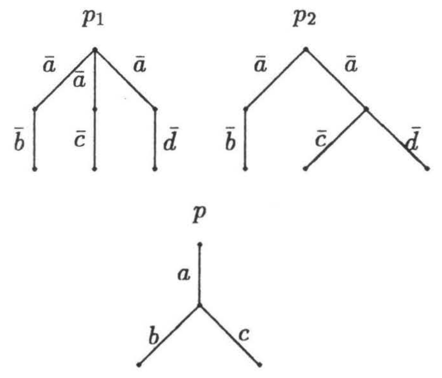

Figure 1: In logic programming p_1 and p_2 cannot be distinguished by p.

Actually, in concurrent logic languages we cannot express a context "strong" enough to distinguish between p_1 and p_2 above. The reason is that, due to the asynchronous nature of *tell*, the choice guarded by tell is a *local choice*. This is shown by the following example.

Example 1.2 *In an asynchronous reading of CCS along the line of the translation given above, $p_1 = a(b + c)$ is equivalent to $p_2 = ab + ac$ under every context. After the production of a, p_1 can proceed to produce either b or c in the same way as p_2 does.*

This example may induce to believe that simple sequences of constraints are sufficient for obtaining compositionality. This is not the case, because the choice guarded by ask is a *global choice*.

Example 1.3 *Even in the asynchronous case, the process $p_1 = \bar{a}(\bar{b} + \bar{c})$ is not equivalent to $p_2 = \bar{a}\bar{b} + \bar{a}\bar{c}$. They are distinguished by the context $p = ab$ (p_2 can deadlock whilst p_1 cannot).*

However, the nature of the global choice in concurrent logic languages is essentially different from the one of CCS. Indeed, it only depends upon the result of the past behaviour of the system, i.e. upon the constraint contained in the store.

This remark indicates a possible way to solve the problem of compositionality. Given a sequence of constraints representing the computation of a process with respect to an arbitrary environment, we add the information about who is the producer of each constraint, either the process or the environment. If the store determined by such a sequence does not provide the process with the necessary information to proceed then the process will deadlock, assuming that the environment does not produce any constraint anymore. The composition of different processes then simply amounts to verifying that the assumptions made by one process about its environment are indeed validated by the other processes.

We define the compositional semantics of a process by means of a transition system. The configurations consist of a process and a store, represented as a sequence of constraints. A transition of the environment is modeled by adding an input constraint to the store. This kind of transition, that does not occur in the usual description of CCS, allows here to obtain a compositional operational semantics based on (sets of) sequences ended by a termination mode. These sequences are essentially different from the *scenarios* of [17], where input substitutions correspond solely to assumptions about the environment which are necessary for the process to proceed. As a consequence, compositionality is there obtained only for the success set. The input-output sequences we use have been introduced in [9] as one component of the domain of the denotational semantics, the other ingredient being the suspension set. Because of what is stated above, this suspension set could have been reduced to a simple termination mode.

The language described in [9] contains non-monotonic test predicates in the guards. Non monotonic means that a predicate can be true on a certain store and false in a bigger store. For instance, the non monotonic predicate $var(x)$ is true in the empty store and false in the store $x = a$. The language we consider is monotonic, in the sense that an *ask* being enabled depends monotonically upon the store. With respect to the problem of full abstraction, this feature causes the sequences to contain too much information [1]. Indeed, they encode the order and the granularity in which constraints have been produced, details that cannot be sensed by monotonic contexts. This is mainly due to the fact that monotonic contexts cannot be specified to ask (only) a specific constraint, they can always proceed when stronger constraints are provided. Therefore a process producing first $x = a$ and then $y = b$ cannot be distinguished from a process that is also able to produce $x = a$ and $y = b$ at the same time: all contexts that are enabled by the store $x = a$ will also be enabled by the store $x = a \wedge y = b$. More in general, the reaction of any context is invariant with respect to the logical equivalence of the conjunction of the constraints produced by the process. Therefore, the final step to achieve full abstraction will consist of some closure conditions that represent this equivalence.

To our knowledge, the first proposal of a compositional semantics based on linear sequences for concurrent logic languages has been given in [7, 8]. Those papers, however, deal only with languages based on substitutions. Moreover, the model we present here is more elegant, since the hiding of local variables is formalized in terms of existential quantifiers. In this framework, the closure conditions are more easy to formulate and have a clear logical intuition. As a consequence, a transparent and structured proof of the correctness and the full abstraction of the denotational model can be given. As far as we know, this is the first time that the proof of a full abstraction result for concurrent logic languages is presented.

This paper is organized as follows. In the next section we present the language. In section three we define a compositional operational semantics based on a transition system. In section four we introduce a more abstract denotational model, the correctness of which is proved in section five. In the last section we prove that this denotational model is fully abstract. In order not to interrupt the main flow of the paper we have delegated some delving into underworldly technicalities to the appendices.

2 The language

A constraint system is any system of partial information that supports the notions of *consistency* and *entailment*. For the sake of simplicity we consider here constraint systems based on first-order languages, however our results can be extended in a straightforward way to arbitrary constraint sytems which support existantial quantification. Let V be a set of variables with typical elements $x, y, \ldots$, let F be a set of function symbols $a, b, \ldots, f, g, \ldots$, and let P be a set of predicate symbols. Furthermore, let $\Sigma = (V, F, P)$. A constraint system Γ is a first-order theory in Σ. Given the formulas ϕ, ϕ_1, and ϕ_2, we say that ϕ is *consistent* if $\Gamma \models \exists\phi$, where $\exists\phi$ denotes the existantial closure of ϕ, and that ϕ_1 *entails* ϕ_2 if $\Gamma \models \phi_1 \Rightarrow \phi_2$. A *simple constraint* ϑ is a quantifier-free

[1] We believe that sequences are not fully abstract even in the non-monotonic case. However this is out of the scope of this paper.

formula in Σ. The set *Con* of constraints, with typical element c, consists of formulas of the form $\exists X\vartheta$, where X is a set of variables. The constraints of the form $\exists\{x\}\vartheta$ and $\exists\emptyset\vartheta$ will be denoted by $\exists x\vartheta$, ϑ, respectively. For any formula ϕ we define $FV(\phi)$ to be the set of the free variables of ϕ, and $BV(\phi)$ to be the set of the bound variables of ϕ. We assume Γ to be fixed, so we will omit references to Γ.

We now describe the concurrent logic language based on Γ along the lines of the one introduced in [11]. Let *Pred* be a set of predicate symbols disjoint from P. The set of atoms *Atom* in V, F, *Pred*, with typical element A, B, is defined as usual. *Tell* and *Ask* are the sets of constructs of the form $tell(\vartheta)$, $ask(\vartheta)$, respectively. A *program* is a finite set of clauses of the form

$$p(\vec{x}) \leftarrow \exists Y g_1 : g_2 | \bar{B}.$$

where $p \in Pred$, $\vec{x}$ is a sequence of variables, $g_1 \in Ask$, $g_2 \in Tell$, $\bar{B}$ is a multiset of atoms, and Y is the set of variables occurring in $g_1, g_2, \bar{B}$ and disjoint from $\vec{x}$. The atom $p(\vec{x})$ is the *head* of the clause, g_1 and g_2 together form the *guard*, and $\bar{B}$ is called the *body* [2]. The variables of Y are the *local* variables of the clause. We will omit the symbol ":" when either g_1 or g_2 is not present, and omit $\exists Y$ when Y is $\emptyset$. The set of programs will be denoted by *Prog*.

A *goal* is an object of the form $\leftarrow \bar{A}$, where $\bar{A}$ is a multiset of atoms. The set of goals will be denoted by *Goal*. The union of the multisets $\bar{A}$ and $\bar{B}$ will be represented by $\bar{A}, \bar{B}$. An ***instantiation*** of a clause $p(\vec{x}) \leftarrow \exists Y g_1 : g_2 | \bar{B}$ is a clause of the form $p(\vec{t}) \leftarrow \exists Z g_1' : g_2' | \bar{B}'$, where g_1', g_2' and $\bar{B}'$ are obtained from g_1, g_2 and $\bar{B}$ by simultaneously replacing every occurrence of a variable of $\vec{x}$ by its corresponding term of the sequence $\vec{t}$ and every occurrence of a variable of Y by its corresponding variable of Z. The set of new local variables Z is assumed to be disjoint from the set of variables of $\vec{t}$. Given a program W the set of all the instantiations of its clauses we denote by $Inst(W)$.

Given a program W the operational semantics of a goal $\leftarrow \bar{A}$ can be described as follows. The basic computation step is defined with respect to the store, the accumulated simple constraint, and consists of checking if the store entails a certain constraint and then adding a constraint. More specifically, given a store s, a computation step consists of a selection of an atom A of $\leftarrow \bar{A}$ and a clause $A \leftarrow \exists X ask(\vartheta_1) : tell(\vartheta_2) | \bar{B}$ of $Inst(W)$, where X has no variables in common with s and A, such that

1. s entails $\exists X \vartheta_1$,

2. $s \wedge \vartheta_1 \wedge \vartheta_2$ is consistent.

Then A is replaced by $\bar{B}$ in the goal $\leftarrow \bar{A}$ and $\vartheta_1 \wedge \vartheta_2$ is added to the store.

An atom A *fails* if for every clause of $Inst(W)$ with head A the second condition does not hold. If the second condition holds but the first fails and for no other clauses the two conditions are satisfied then the atom *suspends*.

A goal fails if it contains an atom which fails and it deadlocks if all its atoms suspend.

The result of a terminating computation consists of the final store where all the variables not occurring in the initial goal are existantially quantified, together with the termination mode: *success*, *failure*, or *deadlock*, if the final goal is empty, fails, or deadlocks, respectively.

3 A compositional operational semantics

In this section we enrich the informal model of the previous section to obtain a compositional semantics, namely, the definition of the meaning of a goal in terms of its subgoals. To this purpose we describe the behaviour of a (sub-)goal as a sequence of *interactions* with its environment (the other subgoals). Interactions are modeled as *input/output* constraints. An input constraint is provided by the environment, whereas an output constraint is produced by the goal itself.

Definition 3.1

[2] Usually a body (as well as a goal) is defined as a sequence of atoms. For our purposes, however, it will be sufficient to represent it as a multiset.

- *The set of input constraints is $Con_I = \{c^I : c \in Con\}$.*
- *The set of output constraint is $Con_O = \{c^O : c \in Con\}$.*
- *The set of input/output constraints, with typical element c^ℓ, is $Con_{IO} = Con_I \cup Con_O$.*

Given a program W, the operational semantics we define is based on a transition system $T = (Conf, \longrightarrow_W)$. We will omit the symbol W when no confusion is possible. The *configurations* $Conf$ are pairs consisting of a goal

(for the sake of convenience we drop the symbol $\leftarrow$) and a finite sequence of input/output constraints s ($s \in Con^*_{IO}$). We will associate a store with such a sequence in the following way:

Definition 3.2 *We define*

$$\begin{array}{lcl} Store(\lambda) & = & true \\ Store((\exists X \vartheta)^\ell.s) & = & \vartheta \wedge Store(s) \end{array}$$

Here λ denotes the empty sequence.

Furthermore we will make extensively use of the following definition of the constraint we obtain from a sequence when abstracting from the labels, interpreting the sequencing operator as conjunction and taking into account the scope of the quantifiers:

Definition 3.3 *We define*

$$\begin{array}{lcl} Estore(\lambda) & = & true \\ Estore((\exists X \vartheta)^\ell.s) & = & \exists X(\vartheta \wedge Estore(s)) \end{array}$$

The representation of a store as a sequence of (existentially quantified) constraints, as well as definitions similar to those of *Store* and *Estore*, have also been used in [12] to model Andorra.

It will turn out to be technically convenient to introduce the following notions:

Definition 3.4 *Given a sequence s we define*

- *$FV(s)$, the free variables of s (the global variables),*
- *$BV(s)$, the bound variables of s (the local variables),*
- *$BV^\ell(s)$, $\ell \in \{I, O\}$, the bound variables of s occurring in constraints labeled by ℓ. So the local variables introduced by the process itself are given by $BV^O(s)$ and those introduced by the environment by $BV^I(s)$,*
- *$var(s)$, the variables of s.*

We can now define the transition system. Table 1 describes the rules for T relative to the "successfull" computation steps. We call them *computation rules*. The first rule models a transition by the process, whilst the second rule models a transition by the environment. The condition $FV(c) \cap BV^O(s) = \emptyset$ formalizes the requirement that the local variables of a process are hidden from the environment. Note that we allow $\bar{A}$ to be the empty goal $\Box$. This models the possibility that the process has terminated whilst the environment still continues to produce constraints. The last rule describes the behaviour of a goal as the interleaving of its subgoals.

Table 2 and table 3 illustrate the rules for failure and suspension respectively. We need to introduce in our configurations the symbols *fail* and *susp*, with the obvious meaning.

Note that if we drop from T the rule **C2**, we obtain a transition system that essentially formalizes the informal model of the previous section.

The reason why we represent a store in T as a sequence of (existentially quantified) constraints is that this allows to express in an elegant way the appropriate closure conditions for full abstraction (see section 4). If we were only interested in compositionality, then sequences of simple constraints would have been sufficient.

Table 1: The Transition System T. Computation Rules

C1 $\langle A; s\rangle \longrightarrow \langle \bar{B}; s.(\exists X(\vartheta_1 \wedge \vartheta_2))^O\rangle$ **if**
$\exists A \leftarrow \exists X\, \mathit{ask}(\vartheta_1) : \mathit{tell}(\vartheta_2) | \bar{B} \in \mathit{Inst}(W)$ such that $X \cap var(s) = \emptyset$ and
A1 : $\models \mathit{Store}(s) \Rightarrow \exists X \vartheta_1$
A2 : $\models \exists(\mathit{Store}(s) \wedge \vartheta_1 \wedge \vartheta_2)$

C2 $\langle \bar{A}; s\rangle \longrightarrow \langle \bar{A}; s.c^I\rangle$ **if**
$\models \exists \mathit{Store}(s.c^I)$, $BV(c) \cap (var(\bar{A}) \cup var(s)) = \emptyset$ and $FV(c) \cap BV^O(s) = \emptyset$

C3 $$\frac{\langle \bar{A}; s\rangle \longrightarrow \langle \bar{A}'; s.c^O\rangle}{\langle \bar{A}, \bar{B}; s\rangle \longrightarrow \langle \bar{A}', \bar{B}; s.c^O\rangle} \quad \text{if } var(\bar{B}) \cap BV(c) = \emptyset$$

Table 2: The Transition System T. Failure Rules

F1 $\langle A; s\rangle \longrightarrow \langle \mathit{fail}; s\rangle$ **if**
there exists no clause with head A in $\mathit{Inst}(W)$ such that **A2** holds.

F2 $$\frac{\langle \bar{A}; s\rangle \longrightarrow \langle \mathit{fail}; s\rangle}{\langle \bar{A}, \bar{B}; s\rangle \longrightarrow \langle \mathit{fail}; s\rangle}$$

The operational semantics $\mathcal{O}$ based on this transition system T delivers sets of sequences s of input/output constraints, ended by a *termination mode*. We denote the set of these sequences as $Seq = Con^*_{IO}.\{ss, \mathit{ff}, dd, \perp\}$. The set Con^*_{IO} denotes the sequences of constraints generated during the computation, whilst the symbols ss, ff, and dd represent the possible ways in wich a process can terminate: *success*, *failure* and *deadlock*, respectively. Sequences ending in $\perp$ denote *unfinished* computations. Such sequences are introduced in order to obtain a non-empty semantics for non-terminating programs. This is necessary to describe failure compositionally. The symbol α will denote an element ranging over the set $\{ss, \mathit{ff}, dd, \perp\}$.

We can now define the operational semantics. In the sequel, $\mathcal{P}$ will denote the powerset operation.

Table 3: The Transition System T. Suspension Rules

S1 $\langle A; s\rangle \longrightarrow \langle \mathit{susp}; s\rangle$ **if**
for all clauses with head A in $\mathit{Inst}(W)$ either **A1** or **A2** does not hold
and there exists such a clause for which **A1** does not hold and **A2** does

S2 $$\frac{\langle \bar{A}; s\rangle \longrightarrow \langle \mathit{susp}; s\rangle \quad \langle \bar{B}; s\rangle \longrightarrow \langle \mathit{susp}; s\rangle}{\langle \bar{A}, \bar{B}; s\rangle \longrightarrow \langle \mathit{susp}; s\rangle}$$

Definition 3.5 (The operational semantics) *The operational semantics $\mathcal{O} : Prog \times Goal \rightarrow \mathcal{P}(Seq)$ is given by*

$$\begin{aligned} \mathcal{O}[\![W;\bar{A}]\!] &= \{s.ss : \langle \bar{A}, \lambda \rangle \longrightarrow^* \langle \Box; s \rangle\} \\ &\cup \{s.ff : \langle \bar{A}, \lambda \rangle \longrightarrow^* \langle fail; s \rangle\} \\ &\cup \{s.dd : \langle \bar{A}, \lambda \rangle \longrightarrow^* \langle susp; s \rangle\} \\ &\cup \{s.\perp : \langle \bar{A}, \lambda \rangle \longrightarrow^* \langle \bar{B}; s \rangle\} \end{aligned}$$

The transition relation $\longrightarrow$ is assumed with respect to the program W.

¿From this operational semantics we obtain our observation criterium as follows:

Definition 3.6 (The observables) *The observables $Obs : Prog \times Goal \rightarrow \mathcal{P}(Con \times \{ss, ff, dd\})$ are defined as*

$$Obs[\![W;\bar{A}]\!] = Result(\mathcal{O}[\![W;\bar{A}]\!])_{/\Leftrightarrow}$$

where $Result(S) = \{\langle Estore(s), \alpha \rangle : s.\alpha \in S$ contains only output constraints and $\alpha \neq \perp\}$, and, given a set C of constraints, $C_{/\Leftrightarrow}$ denotes the closure of C under logical equivalence.

Note that we select sequences containing only output constraints thus modeling a computation that the initial goal is able to carry out on its own.

To show the compositionality of the operational semantics we define the *parallel composition* $\|$. This operator, first introduced in [9], allows to combine sequences of input/output constraints that are equal at each point, apart from the labels, so modeling the interaction of a process with its environment.

Definition 3.7 (The parallel composition operator) *The* partial *operator* $\|: Seq \times Seq \rightarrow Seq$ *is defined by*

- $s_1.\alpha_1 \parallel s_2.\alpha_2 = s_2.\alpha_2 \parallel s_1.\alpha_1$
- $c^I.s_1.\alpha_1 \parallel c^\ell.s_2.\alpha_2 = c^\ell.(s_1.\alpha_1 \parallel s_2.\alpha_2)$
- $\alpha \parallel ss = \alpha$
- $\alpha \parallel ff = ff$
- $dd \parallel dd = dd$
- $\perp \parallel \perp = \perp$

Sometimes we will use the parallel composition on sequences of input/output constraints, without the termination mode (notation $s_1 \parallel s_2$). The extension of $\|$ to sets is defined in the obvious way. The following result shows the compositionality of our operational semantics with respect to goal conjunction.

Theorem 3.8 (Compositionality of $\mathcal{O}$)

$$\mathcal{O}[\![W;\bar{A},\bar{B}]\!] = \mathcal{O}[\![W;\bar{A}]\!] \parallel \mathcal{O}[\![W;\bar{B}]\!]$$

Proof (Sketch) The inclusion $\mathcal{O}[\![W;\bar{A},\bar{B}]\!] \subseteq \mathcal{O}[\![W;\bar{A}]\!] \parallel \mathcal{O}[\![W;\bar{B}]\!]$ can be proved by showing that for every computation $\langle \bar{A}, \bar{B}, \lambda \rangle \longrightarrow^* \langle \bar{A}', s \rangle$ there exist computations $\langle \bar{A}, \lambda \rangle \longrightarrow^* \langle \bar{A}_1', s_1 \rangle$, $\langle \bar{B}, \lambda \rangle \longrightarrow^* \langle \bar{A}_2', s_2 \rangle$, where $\bar{A}_1', \bar{A}_2' = \bar{A}'$, and $s_1 \parallel s_2 = s$. The proof proceeds by induction on the length of s.

The other inclusion $\mathcal{O}[\![W;\bar{A}]\!] \parallel \mathcal{O}[\![W;\bar{B}]\!] \subseteq \mathcal{O}[\![W;\bar{A},\bar{B}]\!]$ is proved by showing that computations $\langle \bar{A}, \lambda \rangle \longrightarrow^* \langle \bar{A}_1', s_1 \rangle$, $\langle \bar{B}, \lambda \rangle \longrightarrow^* \langle \bar{A}_2', s_2 \rangle$, such that $s_1 \parallel s_2$ is defined, can be composed into a computation $\langle \bar{A}, \bar{B}, \lambda \rangle \longrightarrow^* \langle \bar{A}'_0, \bar{A}'_1, s_0 \parallel s_1 \rangle$. The proof proceeds by induction on the length of $s_1 \parallel s_2$. □

The compositionality result can be easily generalized to the union of *neatly intersecting* programs. Two programs W_1 and W_2 are neatly intersecting [9] iff every predicate $p \in \mathit{Pred}$ that occurs in both programs is defined in the same way. Namely, the clauses with head p are exactly the same both in W_1 and W_2.

More in general, two pairs $W_1; \bar{A}_1$ and $W_2; \bar{A}_2$ are neatly intersecting iff

- W_1 and W_2 are neatly intersecting, with shared predicates $p_1, \dots p_k$,
- A_1 does not share predicates with W_2, apart from $p_1, \dots p_k$, and
- A_2 does not share predicates with W_1, apart from $p_1, \dots p_k$.

Corollary 3.9 *Let $W_1; \bar{A}_1$ and $W_2; \bar{A}_2$ be neatly intersecting. Then*

$$\mathcal{O}[\![W_1 \cup W_2; \bar{A}_1, \bar{A}_2]\!] = \mathcal{O}[\![W_1; \bar{A}_1]\!] \parallel \mathcal{O}[\![W_2; \bar{A}_2]\!]$$

Proof By theorem 3.8 we have

$$\mathcal{O}[\![W_1 \cup W_2; \bar{A}_1, \bar{A}_2]\!] = \mathcal{O}[\![W_1 \cup W_2; \bar{A}_1]\!] \parallel \mathcal{O}[\![W_1 \cup W_2; \bar{A}_2]\!]$$

By the restriction upon the predicates of $\bar{A}_1$ and $\bar{A}_2$ we have

$$\mathcal{O}[\![W_1 \cup W_2; \bar{A}_1]\!] = \mathcal{O}[\![W_1; \bar{A}_1]\!] \text{ and } \mathcal{O}[\![W_1 \cup W_2; \bar{A}_2]\!] = \mathcal{O}[\![W_2; \bar{A}_2]\!]$$

□

In the following examples, we assume the constraint system to support the usual equality theory on the Herbrand universe.

Example 3.10 *Consider the following program*

$$W_1 = \{p(x) \leftarrow ask(x = a) \mid .\}$$

We have

$$\mathcal{O}[\![W_1; p(x)]\!] = \{ \quad dd\,, \quad (x = a)^I.(x = a)^O.ss\,, \quad (x = b)^I.f\!f\,, \quad \dots \quad \}$$

$$Obs[\![W_1; p(x)]\!] = \{\langle true, dd \rangle\}.$$

Consider now the program

$$W_2 = \{q(x) \leftarrow tell(x = a) \mid .\}$$

We have

$$\mathcal{O}[\![W_2; q(x)]\!] = \{ \quad (x = a)^O.ss\,, \quad (x = a)^O.(x = a)^I.ss\,, \quad (x = b)^I.f\!f\,, \quad \dots \quad \}$$

$$Obs[\![W_2; q(x)]\!] = \{\langle x = a, ss \rangle\}$$

We consider now the union of the two programs and the goal $\leftarrow p(x), q(x)$. Since $W_1; p(x)$ and $W_2; q(x)$ are neatly intersecting, we get

$$\begin{aligned} \mathcal{O}[\![W_1 \cup W_2; p(x), q(x)]\!] &= \mathcal{O}[\![W_1; p(x)]\!] \parallel \mathcal{O}[\![W_2; q(x)]\!] \\ &= \{ \ (x = a)^O.(x = a)^O.ss\,, \quad (x = b)^I.f\!f\,, \quad \dots \ \} \end{aligned}$$

$$Obs[\![W_1 \cup W_2; p(x), q(x)]\!] = \{\langle x = a, ss \rangle\}.$$

The following example corresponds to example 1.3 in the introduction.

Example 3.11 *Consider the two programs*

$$\begin{array}{lll} W_1 = & \{ & p_1(x,y,z) \leftarrow ask(x=a) \mid q_1(y,z). \\ & & q_1(y,z) \leftarrow ask(y=b) \mid . \\ & & q_1(y,z) \leftarrow ask(z=c) \mid . \qquad \} \end{array}$$

$$\begin{array}{lll} W_2 = & \{ & p_2(x,y,z) \leftarrow ask(x=a) \mid q_2(y). \\ & & p_2(x,y,z) \leftarrow ask(x=a) \mid r_2(z). \\ & & q_2(y) \leftarrow ask(y=b) \mid . \\ & & r_2(z) \leftarrow ask(z=c) \mid . \qquad \} \end{array}$$

We have:

$$\begin{array}{lll} \mathcal{O}[\![W_1;p_1(x,y,z)]\!] = & \{ & (x=a)^I.(x=a)^O.(y=b)^I.(y=b)^O.ss\ , \\ & & (x=a)^I.(x=a)^O.(z=c)^I.(z=c)^O.ss\ , \\ & & \ldots \qquad \} \end{array}$$

$$\begin{array}{lll} \mathcal{O}[\![W_2;p_2(x,y,z)]\!] = & \{ & (x=a)^I.(x=a)^O.(y=b)^I.(y=b)^O.ss\ , \\ & & (x=a)^I.(x=a)^O.(z=c)^I.(z=c)^O.ss\ , \\ & & (x=a)^I.(x=a)^O.(y=b)^I.dd\ , \\ & & (x=a)^I.(x=a)^O.(z=c)^I.dd\ , \\ & & \ldots \qquad \} \end{array}$$

Consider now the program

$$W = \{ \ p(x,y) \leftarrow tell(x=a) \mid tell(y=b). \ \}$$

we have

$$\begin{array}{lll} \mathcal{O}[\![W;p(x,y)]\!] = & \{ & (x=a)^O.(x=a)^I.(y=b)^O.ss\ , \\ & & (x=a)^O.(x=a)^I.(y=b)^O.(y=b)^I.ss\ , \\ & & \ldots \qquad \} \end{array}$$

Therefore, since $W_1;p_1(x,y,z)$, $W_2;p_2(x,y,z)$, *and* $W;p(x,y)$ *are neatly intersecting, we have*

$$\begin{array}{lll} \mathcal{O}[\![W \cup W_1;p(x,y),p_1(x,y,z)]\!] & = & \mathcal{O}[\![W;p(x,y)]\!] \parallel \mathcal{O}[\![W_1;p_1(x,y,z)]\!] \\ & = & \{ \ (x=a)^O.(x=a)^O.(y=b)^O.(y=b)^O.ss\ , \ \ldots \ \} \end{array}$$

$$Obs[\![W \cup W_1;p(x,y),p_1(x,y,z)]\!] = \{\langle x=a \ \wedge \ y=b, ss\rangle\}.$$

whilst

$$\begin{array}{lll} \mathcal{O}[\![W \cup W_2;p(x,y),p_2(x,y,z)]\!] & = & \mathcal{O}[\![W;p(x,y)]\!] \parallel \mathcal{O}[\![W_2;p_2(x,y,z)]\!] \\ & = & \{ \ (x=a)^O.(x=a)^O.(y=b)^O.(y=b)^O.ss\ , \\ & & \ \ (x=a)^O.(x=a)^O.(y=b)^O.dd\ , \\ & & \qquad \ldots \ \} \end{array}$$

$$Obs[\![W \cup W_1;p(x,y),p_2(x,y,z)]\!] = \{\langle x=a \ \wedge \ y=b, ss\rangle, \langle x=a \ \wedge \ y=b, dd\rangle\}.$$

4 The fully abstract semantics $\mathcal{D}$

The operational semantics $\mathcal{O}$ defined in the previous section is not fully abstract. The reason is that the way in which sequences are generated reflects the synctactical structure of the program. Namely, the sequences encode the syntactical form, the order and the granularity in which constraints are produced. These informations cannot be sensed by any context. Indeed, after any (logically) equivalent sequence of constraints produced by the process, the reaction of the context will be the same.

The fully abstract semantics is obtained by applying to the operational semantics some closure conditions that eliminate at the set level the distinctions due to these unobservable informations.

For a concise description of the closure conditions, we *modularize* the sequences of $\mathcal{O}$. The notion of *modular sequence* has been introduced in [11]. Intuitively, a sequence is modular if *Estore*(s) is equivalent to the conjunction of the constraints in s:

Definition 4.1 *A sequence of constraints is called modular if and only if for arbitrary two distinct constraints c and c' occurring in s we have $FV(c) \cap BV(c') = \emptyset$.*

Example 4.2 *The sequence $(\exists y\ x = f(y))^{\ell}.(y = a)^{\ell'}$ is not modular, whilst $(\exists y\ x = f(y))^{\ell}.(x = f(a))^{\ell'}$ is.*

In order to transform non modular sequences into modular ones without changing the *meaning* and the *structure*, we define a notion of equivalence $\equiv$.

Definition 4.3 *We define $s_1 \sim s_2$ iff*

- $s_1 = s.c^{\ell}.s'$ *and* $s_2 = s.c'^{\ell}.s'$
- $\models Estore(s.c^{\ell}) \Leftrightarrow Estore(s.c'^{\ell})$
- $\models Estore(s_1) \Leftrightarrow Estore(s_2)$

Let $\equiv$ be the reflexive, transitive closure of $\sim$.

It is easy to see that, for every sequence s, there exists a modular sequence s' such that $s \equiv s'$.

Example 4.4 *Consider the following sequences*

$$\begin{aligned}
s_1 &= (\exists y\ x = f(y))^{\ell}.(\exists z\ y = g(z))^{\ell'}.(y = g(a))^{\ell''} \\
s_2 &= (\exists y\ x = f(y))^{\ell}.(\exists\{y,z\}(x = f(y) \wedge y = g(z)))^{\ell'}.(y = g(a))^{\ell''} \\
s_3 &= (\exists y\ x = f(y))^{\ell}.(\exists\{y,z\}(x = f(y) \wedge y = g(z)))^{\ell'}.(x = f(g(a)))^{\ell''} \\
s_4 &= (\exists y\ x = f(y))^{\ell}.(\exists z\ x = f(g(z)))^{\ell'}.(x = f(g(a)))^{\ell''}
\end{aligned}$$

We have $s_1 \sim s_2 \sim s_3 \sim s_4$ (and therefore $s_1 \equiv s_4$). Note that s_1 and s_2 are not modular whilst s_3 and s_4 are.

In order to structure the presentation of the fully abstract semantics $\mathcal{D}$, we first introduce an intermediate semantics $\mathcal{O}'$ obtained by modularizing the sequences of $\mathcal{O}$.

Definition 4.5 (The operational semantics $\mathcal{O}'$)

$$\mathcal{O}'[\![W;\bar{A}]\!] = \{s.\alpha : \ s \text{ is modular and there exists } s'.\alpha \in \mathcal{O}[\![W;\bar{A}]\!] \text{ such that } s \equiv s'\}$$

Obviously we have

Proposition 4.6 (Correctness of $\mathcal{O}'$)

$$Obs[\![W;\bar{A}]\!] = Result(\mathcal{O}'[\![W;\bar{A}]\!])_{/\Leftrightarrow}$$

Note that $\mathcal{O}'$ is closed under the logical equivalence of the components of sequences. Therefore it already eliminates one of the unobservable distinctions: the syntactical form of constraints.

The semantics $\mathcal{O}'$ is more abstract than $\mathcal{O}$, but still compositional:

Proposition 4.7 (Compositionality of $\mathcal{O}'$)

$$\mathcal{O}'[\![W;\bar{A},\bar{B}]\!] = \mathcal{O}'[\![W;\bar{A}]\!] \parallel \mathcal{O}'[\![W;\bar{B}]\!]$$

Proof See appendix A. $\square$

The following corollary extends the previous result to neatly intersecting programs. It follows from proposition 4.7 in the same way as corollary 3.9 follows from theorem 3.8.

Corollary 4.8 *Let $W_1;\bar{A}_1$ and $W_2;\bar{A}_2$ be neatly intersecting. Then*

$$\mathcal{O}'[\![W_1 \cup W_2;\bar{A}_1,\bar{A}_2]\!] = \mathcal{O}'[\![W_1;\bar{A}_1]\!] \parallel \mathcal{O}'[\![W_2;\bar{A}_2]\!]$$

We define now the closure conditions that will induce some additional identifications necessary for full abstraction.

Definition 4.9 *Given a set $S \subseteq Seq$, we define $Closure(S)$ to be the minimal set containing S and satisfying the conditions* **P1** *and* **P2** *of table 4. In this table we assume all sequences to be modular, and R to be an arbitrary set of modular sequences.*

Sometimes we will use the notation $Closure(S)$ also for a set S of sequences of input/output constraints (without the termination mode), with the obvious meaning.

Table 4: The closure conditions

P1	$s_1.c_1^\ell.\ldots.c_n^\ell.s_2.\alpha \in R \Rightarrow s_1.{c'}_1^\ell.\ldots.{c'}_m^\ell.s_2.\alpha \in R$ **if** $\models Estore(s_1.c_1^\ell.\ldots.c_n^\ell) \Leftrightarrow Estore(s_1.{c'}_1^\ell.\ldots.{c'}_m^\ell)$
P2	$s_1.s_2.\alpha \in R \Rightarrow s_1.c^I.s_2.\alpha \in R$ **if** $\models Estore(s_1.s_2) \Leftrightarrow Estore(s_1.c^I.s_2)$

The condition **P1** represents the abstraction with respect to order and granularity. Two sequences that only differ for some logically equivalent subsequences of constraints produced by the same agent (either the process or the environment) must be identified. The condition **P2** completes this identification at the set level. The arbitrary input constraints (given automatically by the transition system in the original sequences of $\mathcal{O}'$) must also be added to the new sequences generated by **P1**, in order to eliminate the remaining distinctions.

Remark 4.10 *The closure conditions preserve the meaning of a sequence. Namely, if $s.\alpha \in Closure(\{s'.\alpha\})$ (abbrev. $Closure(s'.\alpha)$), then $\models Estore(s) \Leftrightarrow Estore(s')$ holds.*

Remark 4.11 *The closure operator is idempotent, namely*

$$Closure(Closure(S)) = Closure(S)$$

We define the fully abstract semantics $\mathcal{D}$ in a denotational (i.e. compositional) style, and the basic cases (empty and unit goals) we obtain by applying the closure operator to $\mathcal{O}'$.

Definition 4.12 (The fully abstract semantics $\mathcal{D}$) *We define $\mathcal{D} : Prog \times Goal \rightarrow \mathcal{P}(Seq)$ as follows*

$$\begin{array}{lcl} \mathcal{D}[\![W;\Box]\!] & = & Closure(\mathcal{O}'[\![W;\Box]\!]) \\ \mathcal{D}[\![W;A]\!] & = & Closure(\mathcal{O}'[\![W;A]\!]) \\ \mathcal{D}[\![W;\bar{A},\bar{B}]\!] & = & Closure(\mathcal{D}[\![W;\bar{A}]\!] \parallel \mathcal{D}[\![W;\bar{B}]\!]) \end{array}$$

The semantics $\mathcal{D}$ is strictly more abstract than $\mathcal{O}$ and $\mathcal{O}'$.

Example 4.13 *Consider the two programs*

$$\begin{array}{lll} W_1 = & \{ & p_1(x) \leftarrow \exists y\; tell(x = f(y)) \mid q(y). \\ & & q(y) \leftarrow tell(y = a) \mid . \qquad \} \end{array}$$

$$\begin{array}{lll} W_2 = & \{ & p_2(x) \leftarrow tell(x = f(a)) \mid . \\ & & p_2(x) \leftarrow \exists y\; tell(x = f(y)) \mid q(y). \\ & & q(y) \leftarrow tell(y = a) \mid . \qquad \} \end{array}$$

We have that $W_1;p_1(x)$ and $W_2;p_2(x)$ behave in the same way under every context (i.e. they are observationally equivalent), namely, for every $W;\bar{A}$, neatly intersecting with $W_1;p_1(x)$ and $W_2;p_2(x)$:

$$Obs[\![W \cup W_1; \bar{A}, p_1(x)]\!] = Obs[\![W \cup W_2; \bar{A}, p_2(x)]\!]$$

This is because $W_1 \subseteq W_2$ and the behaviour of the first clause of W_2 can be simulated by the two clauses of W_1. The operational semantics of these programs is however different (i.e. $\mathcal{O}$ is not fully abstract). In fact

$$(x = f(a))^O.ss \in \mathcal{O}[\![W_2;p_2(x)]\!]$$

whilst

$$(x = f(a))^O.ss \notin \mathcal{O}[\![W_1;p_1(x)]\!]$$

The same applies to $\mathcal{O}'$. On the other side, this difference in the operational semantics disappears in the denotational one. In fact, we have

$$(\exists y\; x = f(y))^O.(y = a)^O.ss \in \mathcal{O}[\![W_1;p_1(x)]\!], \text{ and}$$

$$(\exists y\; x = f(y))^O.(x = f(a))^O.ss \in \mathcal{O}'[\![W_1;p_1(x)]\!]$$

therefore, by an application of **P1**, *we obtain*

$$(x = f(a))^O.ss \in \mathcal{D}[\![W_1;p_1(x)]\!].$$

This example shows the use of **P1** to abstract from granularity of constraint production. In this case, **P1** has been used to *group* a sequence of output constraints (to *enlarge* the granularity). Examples in which **P1** is needed to *split* an output constraint (to *reduce* the granularity) are a bit more complicated, but not difficult to imagine.

Concerning **P2**, its use can be understood to complete the abstraction made by **P1**. Indeed, when we split a constraint into a sequence, we have also to allow additional interleaving points within this sequence. This is modeled by inserting arbitrary (consistent) input constraints (**P2**).

By definition, $\mathcal{D}$ is compositional. As usual, compositionality can be extended to neatly intersecting programs:

Proposition 4.14 *Let $W_1;\bar{A}_1$ and $W_2;\bar{A}_2$ be neatly intersecting. Then*

$$\mathcal{D}[\![W_1 \cup W_2; \bar{A}_1, \bar{A}_2]\!] = Closure(\mathcal{D}[\![W_1;\bar{A}_1]\!] \parallel \mathcal{D}[\![W_2;\bar{A}_2]\!])$$

The next sections will show that $\mathcal{D}$ is correct with respect to the observables and that it is fully abstract.

5 The correctness of $\mathcal{D}$

In this section we prove the correctness of $\mathcal{D}$ with respect to the observables. The structure of the proof is the following: first we prove that $\mathcal{D}$ is the closure of $\mathcal{O}'$ for arbitrary goals (not only for the empty and unit ones), then we use the correctness of $\mathcal{O}'$.

Lemma 5.1 *For arbitrary sets S_1 and S_2 of modular sequences which are closed under* **P2** *and the following version of* **P1**

$$\textbf{P1-I} \quad s_1.c_1^I \dots c_n^I.s_2.\alpha \in R \Rightarrow s_1.c'^I_1 \dots c'^I_m.s_2.\alpha \in R$$
$$\textbf{if} \models Estore(s_1.c_1^I \dots c_n^I) \Leftrightarrow Estore(s_1.c'^I_1 \dots c'^I_m)$$

we have

$$Closure(Closure(S_1) \parallel Closure(S_2)) = Closure(S_1 \parallel S_2)$$

Proof See appendix B. □

Proposition 5.2 $\quad \mathcal{D}[\![W;\bar{A}]\!] = Closure(\mathcal{O}'[\![W;\bar{A}]\!])$

Proof Straightforward induction on the length of the goal, using lemma 5.1 (it is easy to verify that $\mathcal{O}'$ is closed under **P1-I** and **P2**) and the compositionality of $\mathcal{O}'$. □

We can now prove the correctness of $\mathcal{D}$:

Theorem 5.3 (Correctness of $\mathcal{D}$) $\quad Result(\mathcal{D}[\![W;\bar{A}]\!])_{/\Leftrightarrow} = Obs[\![W;\bar{A}]\!]$

Proof

$$\begin{array}{lll} Result(\mathcal{D}[\![W;\bar{A}]\!])_{/\Leftrightarrow} & = & \text{(by proposition 5.2)} \\ Result(Closure(\mathcal{O}'[\![W;\bar{A}]\!]))_{/\Leftrightarrow} & = & \text{(by remark 4.10)} \\ Result(\mathcal{O}'[\![W;\bar{A}]\!])_{/\Leftrightarrow} & = & \text{(by proposition 4.6)} \\ Obs[\![W;\bar{A}]\!] & & \end{array}$$

□

6 The full abstraction of $\mathcal{D}$

In this section we prove the full abstraction of $\mathcal{D}$ with respect to our observation criterium. The basic lines of the proof are the following. Given two goals $\leftarrow \bar{A}_1$, $\leftarrow \bar{A}_2$ with a different semantics $\mathcal{D}$, we build a context that is able to "detect" this difference at the observational level. The definition of this context is *uniform*, in the following sense: given a modular sequence $s.\alpha$ we define a *context* $C(s.\alpha)$ as a pair *program;goal* which "recognizes" $s.\alpha$. Next we prove that every other sequence $s'.\alpha$ recognized by $C(s.\alpha)$ that gives the same result must generate $s.\alpha$ by application of the closure operator. Then we reason by contradiction: given a sequence $s.\alpha$ in the semantic difference ($s.\alpha \in \mathcal{D}[\![W;\bar{A}_1]\!] \setminus \mathcal{D}[\![W;\bar{A}_2]\!]$), if the context $C(s.\alpha)$ doesn't induce a difference in the observables, then there exists an other sequence $s'.\alpha$ ($s'.\alpha \in \mathcal{D}[\![W;\bar{A}_2]\!]$) recognized by the context that produces the same result. But, since $\mathcal{D}$ is closed, the presence of $s'.\alpha$ implies the presence of $s.\alpha$, and this contradicts the assumption.

Definition 6.1 *Let $s.\alpha$ be a modular sequence, and $\vec{x}$ be the free variables of s. We define the context $C(s.\alpha)$ by induction on the length n of s. We assume given a set of new predicate symbols $\{p_0, \dots p_n\}$ disjoint from Pred.*

- $C(ss) = C(ff) = C(dd) = \{p_0(\vec{x}) \leftarrow\mid .\}; p_0(\vec{x})$, *and* $C(\perp) = \{\}; p_0(\vec{x})$,
- $C((\exists Y\vartheta)^I.s.\alpha) = \{p_n(\vec{x}) \leftarrow \exists Y tell(\vartheta) | p_{n-1}(\vec{x}).\} \cup W; p_n(\vec{x})$, *where* $W; p_{n-1}(\vec{x}) = C(s.\alpha)$,

- $C((\exists Y\vartheta)^O.s.\alpha) = \{p_n(\vec{x}) \leftarrow \exists Y\, ask(\vartheta) | p_{n-1}(\vec{x}).\} \cup W; p_n(\vec{x})$, *where* $W; p_{n-1}(\vec{x}) = C(s.\alpha)$.

Note that the goal $\leftarrow p_0(\vec{x})$ *gives rise to failure in* $C(\perp)$, *since the predicate* p_0 *is undefined. In the other cases, it succeeds.*

The following proposition states that a context $C(s.\alpha)$ recognizes the sequence $s.\alpha$. Namely, $C(s.\alpha)$ generates $\tilde{s}.\tilde{\alpha}$, where $\tilde{s}$ denotes the "mirror" of s, i.e. $\widetilde{c^I.s} = c^O.\tilde{s}$ and $\widetilde{c^O.s} = c^I.\tilde{s}$, and

$$\tilde{\alpha} = \begin{cases} ss & \textit{if } \alpha \in \{ss, ff, dd\} \\ ff & \textit{otherwise.} \end{cases}$$

Proposition 6.2 *For any modular sequence* $s.\alpha$ *we have* $\tilde{s}.\tilde{\alpha} \in \mathcal{D}[\![C(s.\alpha)]\!]$.

Proof Let $\tilde{s}'$ be obtained by $\tilde{s}$ by adding an output constraint c^O after any input constraint c^I. It is not difficult to see that $\tilde{s}'.\tilde{\alpha} \in \mathcal{O}[\![C(s.\alpha)]\!]$. Furthermore, since s is modular, also $\tilde{s}$ and $\tilde{s}'$ are modular. Hence

$$\tilde{s}'.\tilde{\alpha} \in \mathcal{O}'[\![C(s.\alpha)]\!] \tag{1}$$

We conclude

$$\begin{array}{lll} \tilde{s}.\tilde{\alpha} & \in & \text{(by } \mathbf{P1}\text{)} \\ \mathit{Closure}(\tilde{s}'.\tilde{\alpha}) & \subseteq & \text{(by 1)} \\ \mathit{Closure}(\mathcal{O}'[\![C(s.\alpha)]\!]) & = & \text{(by proposition 5.2)} \\ \mathcal{D}[\![C(s.\alpha)]\!] & & \end{array}$$

□

The next two lemmas together imply that the context $C(s.\alpha)$ recognizes *only* $s.\alpha$, in the sense that if $C(s.\alpha)$ interacts with a sequence s' (i.e., for some α' we have $\tilde{s}'.\alpha' \in \mathcal{D}[\![C(s.\alpha)]\!]$), which gives the same result as s, i.e., $\models \mathit{Estore}(s) \Leftrightarrow \mathit{Estore}(s')$, then s can be obtained from s' by applying the closure operator.

The next lemma actually shows that $\tilde{s}'$ is in the closure of $\tilde{s}$. The final step is then made by the *mirroring lemma* (see page 18).

Lemma 6.3 *Given a modular sequence* $s.\alpha$, *for every* $s'.\alpha' \in \mathcal{D}[\![C(s.\alpha)]\!]$ *such that* $\models \mathit{Estore}(s') \Leftrightarrow \mathit{Estore}(\tilde{s})$ *we have* $s' \in \mathit{Closure}(\tilde{s})$.

Proof By proposition 5.2, $\mathcal{D}[\![C(s.\alpha)]\!] = \mathit{Closure}(\mathcal{O}'[\![C(s.\alpha)]\!])$ holds. Therefore, by remarks 4.10 and 4.11, it is sufficient to prove that, for $s'.\alpha' \in \mathcal{O}'[\![C(s.\alpha)]\!]$ with $\models \mathit{Estore}(s') \Leftrightarrow \mathit{Estore}(\tilde{s})$, we have $s' \in \mathit{Closure}(\tilde{s})$.

Let $s'.\alpha' \in \mathcal{O}'[\![C(s.\alpha)]\!]$ such that $\models \mathit{Estore}(s') \Leftrightarrow \mathit{Estore}(\tilde{s})$. Let n be the length of s. It is not so difficult to see that there exists a computation

$$\langle p_n(\vec{x}), \lambda\rangle \longrightarrow^* \ldots \longrightarrow^* \langle p_{n-i}(\vec{x}), s'_i\rangle \longrightarrow^* \ldots \longrightarrow^* \langle p_0(\vec{x}), s'_n\rangle$$

such that $s'_n \equiv s'$.

Let $s'_{(i)}$ denote the prefix of s' such that $s'_{(i)} \equiv s'_i$, and let $\tilde{s}^{(i)}$ denote the suffix of $\tilde{s}$ starting from its $(i+1)^{th}$ element. We prove that $s'_{(i)}.\tilde{s}^{(i)} \in \mathit{Closure}(\tilde{s})$ for $0 \leq i \leq n$. We proceed by induction on i.

$i = 0$) Obvious, since $s'_{(0)} = \lambda$ and $\tilde{s}^{(0)} = \tilde{s}$.

$i + 1$) By the induction hypothesis we have

$$s'_{(i)}.\tilde{s}^{(i)} \in \mathit{Closure}(\tilde{s}). \tag{2}$$

There are two cases

Case 1: $\tilde{s}^{(i)} = c^I.\tilde{s}^{(i+1)})$ In this case p_{n-i} is defined in $C(s.\alpha)$ as

$$p_{n-i}(\vec{x}) \leftarrow \exists Y\, ask(\vartheta)|p_{n-i-1}(\vec{x}).,$$

where $c = \exists Y \vartheta$. So we have

$$\langle p_{n-i}(\vec{x}), s'_i\rangle \longrightarrow^* \langle p_{n-i-1}(\vec{x}), s'_{i+1}\rangle$$

where

$$s'_{i+1} = s'_i.c^I_1.\ldots.c^I_k.c^O$$

with

$$\models Store(s'_i.c^I_1.\ldots.c^I_k) \Rightarrow c. \tag{3}$$

Since $s' \equiv s'_n$, we have

$$s'_{(i+1)} = s'_{(i)}.c'^I_1\ldots c'^I_k.c'^O.$$

We note that

$$\begin{array}{lll} \models & Estore(s') & \Leftrightarrow \text{ (by hypothesis)} \\ & Estore(\tilde{s}) & \Leftrightarrow \text{ (by 2 and remark 4.10)} \\ & Estore(s'_{(i)}.\tilde{s}^{(i)}) & \end{array}$$

from which we derive

$$\models Estore(s'_{(i)}.c'^I_1\ldots c'^I_k.c^I.\tilde{s}^{(i+1)}) \Leftrightarrow Estore(s'_{(i)}.c'^I_1\ldots c'^I_k.\tilde{s}^{(i)}) \Leftrightarrow Estore(s'_{(i)}.\tilde{s}^{(i)}) \tag{4}$$

Therefore, since $s'_{(i)}.c'^I_1\ldots c'^I_k.c^I.\tilde{s}^{(i+1)} \in Closure(s'_{(i)}.\tilde{s}^{(i)})$ (by 4 and **P2**), and $Closure(s'_{(i)}.\tilde{s}^{(i)}) \subseteq Closure(\tilde{s})$ (by 2 and remark 4.11), we obtain

$$s'_{(i)}.c'^I_1\ldots c'^I_k.c^I.\tilde{s}^{(i+1)} \in Closure(\tilde{s}) \tag{5}$$

Now we have to "transform" c^I into c'^O. We do so by two applications of **P1**. The first deletes c^I, the second adds c'^O. For the first application we need the following:

$$\begin{array}{lll} \models & Estore(s'_{(i)}.c'^I_1\ldots c'^I_k) & \Leftrightarrow \text{ (since } s' \equiv s'_n) \\ & Estore(s'_i.c^I_1\ldots c^I_k) & \Leftrightarrow \text{ (by 3)} \\ & Estore(s'_i.c^I_1\ldots c^I_k.c^I) & \Leftrightarrow \text{ (since } FV(c) \subseteq FV(s), \text{as } s \text{ is modular)} \\ & Estore(s'_i.c^I_1\ldots c^I_k) \wedge c & \Leftrightarrow \text{ (since } s' \equiv s'_n) \\ & Estore(s'_{(i)}.c'^I_1\ldots c'^I_k) \wedge c & \Leftrightarrow \text{ (since } FV(c) \subseteq FV(s)) \\ & Estore(s'_{(i)}.c'^I_1\ldots c'^I_k.c^I). & \end{array}$$

The last equation holds under the assumption $BV(s') \cap FV(s) = \emptyset$. We can assume this without loss of generality, since $\models Estore(s') \Leftrightarrow Estore(\tilde{s}) \Leftrightarrow Estore(s)$.

Now we can apply **P1**, thus obtaining

$$s'_{(i)}.c'^I_1\ldots c'^I_k.\tilde{s}^{(i+1)} \in Closure(s'_{(i)}.c'^I_1\ldots c'^I_k.c^I.\tilde{s}^{(i+1)}). \tag{6}$$

For the second application we need

$$\begin{array}{lll} \models & Estore(s'_{(i)}.c'^I_1\ldots c'^I_k) & \Leftrightarrow \text{ (since } s' \equiv s'_n) \\ & Estore(s'_i.c^I_1\ldots c^I_k) & \Leftrightarrow \text{ (by 3)} \\ & Estore(s'_i.c^I_1\ldots c^I_k.c^O) & \Leftrightarrow \text{ (since } s' \equiv s'_n) \\ & Estore(s'_{(i)}.c'^I_1\ldots c'^I_k.c'^O) & \end{array}$$

Then, we can apply **P1** again, thus obtaining

$$s'_{(i)}.c'^I_1.\dots.c'^I_k.c'^O.\tilde{s}^{(i+1)} \in \mathit{Closure}(s'_{(i)}.c'^I_1.\dots.c'^I_k.\tilde{s}^{(i+1)}). \tag{7}$$

We conclude

$$\begin{array}{lll}
s'_{(i+1)}.\tilde{s}^{(i+1)} & = & \\
s'_{(i)}.c'^I_1.\dots.c'^I_k.c'^O.\tilde{s}^{(i+1)} & \in & \text{(by 7)} \\
\mathit{Closure}(s'_{(i)}.c'^I_1.\dots.c'^I_k.\tilde{s}^{(i+1)}) & \subseteq & \text{(by 6)} \\
\mathit{Closure}(s'_{(i)}.c'^I_1.\dots.c'^I_k.c^I.\tilde{s}^{(i+1)}) & \subseteq & \text{(by 5)} \\
\mathit{Closure}(\tilde{s}) & &
\end{array}$$

Case 2: $\tilde{s}^{(i)} = c^O.\tilde{s}^{(i+1)})$ In this case p_{n-i} is defined in $C(s.\alpha)$ as

$$p_{n-i}(\vec{x}) \leftarrow \exists Y\, \mathit{tell}(\vartheta) | p_{n-i-1}(\vec{x}).,$$

where $c = \exists Y \vartheta$. So we have

$$\langle p_{n-i}(\vec{x}), s'_i\rangle \longrightarrow^* \langle p_{n-i-1}(\vec{x}), s'_{i+1}\rangle$$

where

$$s'_{i+1} = s'_i.c^I_1.\dots.c^I_k.c^O$$

Since $s' \equiv s'_n$, we have

$$s'_{(i+1)} = s'_{(i)}.c'^I_1.\dots.c'^I_k.c'^O.$$

Similarly to the previous case, we have

$$\models \mathit{Estore}(s'_{(i)}.c'^I_1.\dots.c'^I_k.c^O.\tilde{s}^{(i+1)}) \Leftrightarrow \mathit{Estore}(s'_{(i)}.c'^I_1.\dots.c'^I_k.\tilde{s}^{(i)}) \Leftrightarrow \mathit{Estore}(s'_{(i)}.\tilde{s}^{(i)}) \tag{8}$$

therefore, since $s'_{(i)}.c'^I_1.\dots.c'^I_k.c^O.\tilde{s}^{(i+1)} \in \mathit{Closure}(s'_{(i)}.\tilde{s}^{(i)})$ (by 8 and **P2**), and $\mathit{Closure}(s'_{(i)}.\tilde{s}^{(i)}) \subseteq \mathit{Closure}(\tilde{s})$ (by 2 and remark 4.11), we obtain

$$s'_{(i)}.c'^I_1.\dots.c'^I_k.c^O.\tilde{s}^{(i+1)} \in \mathit{Closure}(\tilde{s}) \tag{9}$$

We have now to "transform" c^O into c'^O. We need the following

$$\begin{array}{llll}
\models & \mathit{Estore}(s'_{(i)}.c'^I_1\dots c'^I_k.c^O) & \Leftrightarrow & \text{(since } FV(c) \subseteq FV(s)) \\
& \mathit{Estore}(s'_{(i)}.c'^I_1\dots c'^I_k) \wedge c & \Leftrightarrow & \text{(since } s' \equiv s'_n) \\
& \mathit{Estore}(s'_i.c^I_1\dots c^I_k) \wedge c & \Leftrightarrow & \text{(since } FV(c) \subseteq FV(s)) \\
& \mathit{Estore}(s'_i.c^I_1\dots c^I_k.c^O) & \Leftrightarrow & \text{(since } s' \equiv s'_n) \\
& \mathit{Estore}(s'_{(i)}.c'^I_1\dots c'^I_k.c'^O). & &
\end{array}$$

Thus an application of **P1** yields

$$s'_{(i)}.c'^I_1.\dots.c'^I_k.c'^O.\tilde{s}^{(i+1)} \in \mathit{Closure}(s'_{(i)}.c'^I_1.\dots.c'^I_k.c^O.\tilde{s}^{(i+1)}). \tag{10}$$

Therefore we can conclude

$$\begin{array}{lll}
s'_{(i+1)}.\tilde{s}^{(i+1)} & = & \\
s'_{(i)}.c'^I_1.\dots.c'^I_k.c'^O.\tilde{s}^{(i+1)} & \in & \text{(by 10)} \\
\mathit{Closure}(s'_{(i)}.c'^I_1.\dots.c'^I_k.c^O.\tilde{s}^{(i+1)}) & \in & \text{(by 9)} \\
\mathit{Closure}(\tilde{s}). & &
\end{array}$$

□

Lemma 6.4 (Mirroring lemma) *If $s' \in Closure(s)$, then $\tilde{s} \in Closure(\tilde{s}')$. (As usual, $\tilde{s}$ and $\tilde{s}'$ denote the mirror of s, s', respectively.)*

Proof It is sufficient to show that for any set of sequences S, if $Closure(S) = S$, then S satisfies the following property:

P3 $\quad s_1.c^O.s_2.\alpha \in S \Rightarrow s_1.s_2.\alpha \in S$
if $\models Estore(s_1.s_2) \Leftrightarrow Estore(s_1.c^O.s_2)$.

We then can proceed by induction on the number of applications of the closure conditions **P1** and **P2**, making use of the fact that **P1** mirrors itself, in the following sense: if s' is derived from s by one application of **P1** then $\tilde{s}$ can be derived from $\tilde{s}'$ using **P1** again. In same sense an application of **P2** can be mirrored by **P3**.

We prove **P3** by induction on the length of s_2:

$s_2 = \lambda$) In this case we just apply **P1**.

$s_2 = c'^{\ell}.s_2'$) We consider the cases $\ell = I$ and $\ell = O$ separately.

$\ell = O$) By **P1** we have

$$s_1.c^O.c'^O.s_2'.\alpha \in S \Rightarrow s_1.c'^O.c^O.s_2'.\alpha \in S$$

The induction hypothesis then gives us

$$s_1.c'^O.s_2'.\alpha \in S$$

$\ell = I$) By **P2** we have

$$s_1.c^O.c'^I.s_2'.\alpha \in S \Rightarrow s_1.c'^I.c^O.c'^I.s_2'.\alpha \in S$$

An application of **P1** then gives us

$$s_1.c'^I.c^O.s_2'.\alpha \in S$$

By induction hypothesis we obtain

$$s_1.c'^I.s_2'.\alpha \in S.$$

□

Now we are ready to prove the main theorem

Theorem 6.5 (Full abstraction of $\mathcal{D}$) *For arbitrary $W_1; \bar{A}_1$, $W_2; \bar{A}_2$ such that $\mathcal{D}[\![W_1; \bar{A}_1]\!] \neq \mathcal{D}[\![W_2; \bar{A}_2]\!]$ there exists $W; \bar{A}$, neatly intersecting with $W_1; \bar{A}_1$ and $W_2; \bar{A}_2$, such that $Obs[\![W \cup W_1; \bar{A}, \bar{A}_1]\!] \neq Obs[\![W \cup W_2; \bar{A}, \bar{A}_2]\!]$.*

Proof Assume $s.\alpha \in \mathcal{D}[\![W_1; \bar{A}_1]\!] \setminus \mathcal{D}[\![W_2; \bar{A}_2]\!]$. Let $W; \bar{A} = C(s.\alpha)$, and let

$$\bar{\alpha} = \begin{cases} ss & \text{if } \alpha \in \{ss, ff, dd\} \\ ff & \text{otherwise.} \end{cases}$$

By proposition 6.2 we have $\tilde{s}.\bar{\alpha} \in \mathcal{D}[\![W; \bar{A}]\!]$, therefore

$$Result(s.\alpha \parallel \tilde{s}.\bar{\alpha}) \in Result(\mathcal{D}[\![W \cup W_1; \bar{A}, \bar{A}_1]\!])_{/\Leftrightarrow} = Obs[\![W \cup W_1; \bar{A}, \bar{A}_1]\!].$$

Assume now that

$$Result(s.\alpha \parallel \tilde{s}.\bar{\alpha}) \in Obs[\![W \cup W_2; \bar{A}, \bar{A}_2]\!] = Result(\mathcal{D}[\![W \cup W_2; \bar{A}, \bar{A}_2]\!])_{/\Leftrightarrow}$$

By the compositionality of $\mathcal{D}$ and remark 4.10 it follows that there exist $s'.\alpha' \in \mathcal{D}[\![W_2; \bar{A}_2]\!]$ and $\tilde{s}'.\alpha'' \in \mathcal{D}[\![W; \bar{A}]\!]$ such that

$$Result(s'.\alpha' \parallel \tilde{s}'.\alpha'') = Result(s.\alpha \parallel \tilde{s}.\bar{\alpha}). \tag{11}$$

Thus we have

$$\models Estore(\tilde{s}') \Leftrightarrow Estore(s' \parallel \tilde{s}') \Leftrightarrow Estore(s \parallel \tilde{s}) \Leftrightarrow Estore(\tilde{s}),$$

so by lemma 6.3 we have $\tilde{s}' \in Closure(\tilde{s})$. An application of lemma 6.4 then yields $s \in Closure(s')$. Therefore $s.\alpha \in \mathcal{D}[\![W_2; \bar{A}_2]\!]$ holds. By definition of $\mathcal{O}$ and $\mathcal{O}'$, and proposition 5.2, it follows $s.\perp\in \mathcal{D}[\![W_2; \bar{A}_2]\!]$. Furthermore we observe that, by definition of $\bar{\alpha}$, of the context $C(s.\alpha)$, and by 11, we have $\alpha' = \alpha$ if $\alpha \neq \perp$. So we conclude $s.\alpha \in \mathcal{D}[\![W_2; \bar{A}_2]\!]$, and this contradicts our initial assumption. □

7 Conclusions and future work

We have studied in this paper the asynchronous nature of the communication in concurrent logic languages. We have shown that the fully abstract semantics for these languages requires an approach quite different from the standard ones for languages like CCS. One of the main differences consists in the description of the deadlock behaviour. In CCS the deadlock of a process depends upon the *current* state of the system as described by the failure sets, whereas in concurrent logic languages the deadlock essentially depends upon the result of the *past* behaviour of the system.

A future research topic is the investigation of the non-monotonic case, namely when non-monotonic test predicates (like the *var* of Prolog) occur in the guards.

An other promising subject is the extension of our approach to more general concurrent logic paradigms, like concurrent constraint programming [18], which include an explicit choice operator.

An even more ambitious project is to show that the construction of the fully abstract model we have presented can be extended to asynchronous languages in general. This conjecture asks for the definition of a general paradigm which subsumes all possible asynchronous communication mechanisms. For such a paradigm we think it is possible to define a compositional semantics based upon linear sequences with input assumptions, thus providing an uniform deadlock analysis for asynchronous communication. The fully abstract model for a particular asynchronous language then will consist of the application of some appropriate closure conditions on the general compositional model. These closure conditions will model the specific features of the asynchronous communication of that particular language. Actually, part of this project is already under development in [6].

Acknowledgements Part of the motivations for this work originated from the joint work with Joost Kok and Jan Rutten on the semantics of PARLOG and GHC. We thank the members of the C.W.I. concurrency group, J.W. de Bakker, F. Breughel, A de Bruin, E. Horita, P. Knijnenburg, J. Kok, J. Rutten, E. de Vink and J. Warmerdam for their comments on preliminary versions of this paper. We also thank Seif Haridi and Vijay Saraswat for helpfull and stimulating discussions.

References

[1] K.R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Volume B, Elsevier Science Publishers, 1990. Also available as Technical Report CS-R8826, Centre for Mathematics and Computer Science, Amsterdam, 1988.

[2] S.D. Brookes, C.A.R. Hoare, and W. Roscoe. A Theory of Communicating Sequential Processes. *JACM*, 31:499–560, 1984.

[3] J.W. de Bakker and J.N. Kok. Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 347–355, Tokyo, Japan, 1988. OHMSHA, LTD. Extended Abstract, full version available as CWI report CS-8834.

[4] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Control flow versus logic: a denotational and a declarative model for Guarded Horn Clauses. In A. Kreczmar and G. Mirkowska, editors, *Proc. of the Symposium on Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 1989.

[5] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Semantic models for a version of PARLOG. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Series in Logic Programming, pages 621–636, Lisboa, 1989. The MIT Press. Extended version to appear in *Theoretical Computer Science.*

[6] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The Failure of Failures: Towards a Paradigm for Asynchronous Communication. Technical Report, Utrecht University, Dept. of Computer Science, 1990.

[7] F.S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proc. of the North American Conference on Logic Programming*, Series in Logic Programming, pages 175–194. The MIT Press, 1990. Full version available as Technical Report TR 6/90, Dipartimento di Informatica, Università di Pisa.

[8] F.S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of Concur 90*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114, The Netherlands, 1990. Springer-Verlag. Full version available as report at the Technische Universiteit Eindhoven.

[9] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully abstract denotational semantics for Concurrent Prolog. In *Proc. of the Third IEEE Symposium on Logic In Computer Science*, pages 320–335, 1988.

[10] M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics for concurrent constraint logic programs. In Springer-Verlag, editor, *Proc. of the Second Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science, Nancy, France, 1990.

[11] H. Gaifman, M.J. Maher, and E. Shapiro. Reactive behaviour semantics for concurrent constraint logic languages. In *Proc. of the North American Conference on Logic Programming*, 1989.

[12] S. Haridi and C. Palamidessi. Structural transformational semantics for Kernel Andorra Prolog. Technical report, SICS, Sweden, 1990.

[13] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 111–119, 1987.

[14] J.N. Kok. A compositional semantics for Concurrent Prolog. In R. Cori and M. Wirsing, editors, *Proc. Fifth Symposium on Theoretical Aspects of Computer Science*, volume 294 of *Lecture Notes in Computer Science*, pages 373–388. Springer-Verlag, 1988.

[15] M. J. Maher. Logic semantics for a class of committed choice programs. In J.-L. Lassez, editor, *Proc. of the Fourth Int. Conference on Logic Programming*, pages 877–893, Melbourne, 1987. MTI Press.

[16] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of LNCS. Springer Verlag, New York, 1980.

[17] V.A. Saraswat. Partial correctness semantics for cp($\downarrow, |, \&$). In *Proc. of the Conf. on Foundations of Software Computing and Theoretical Computer Science*. Volume 206 of LNCS, pages 347–368, 1985.

[18] V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, january 1989. To be published by MTI Press.

[19] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages*. ACM, New York, 1990.

Appendix

A The compositionality of $\mathcal{O}'$

In this appendix we prove:

Proposition 4.7

$$\mathcal{O}'[\![W;\bar{A},\bar{B}]\!] = \mathcal{O}'[\![W;\bar{A}]\!] \parallel \mathcal{O}'[\![W;\bar{B}]\!]$$

Proof We prove $\mathcal{O}'[\![W;\bar{A}]\!] \parallel \mathcal{O}'[\![W;\bar{B}]\!] \subseteq \mathcal{O}'[\![W;\bar{A},\bar{B}]\!]$, the other inclusion follows immediately from the compositionality of $\mathcal{O}$. Let $r_1.\alpha_1 \parallel r_2.\alpha_2 \in \mathcal{O}'[\![W;\bar{A}]\!] \parallel \mathcal{O}'[\![W;\bar{B}]\!]$, say, $r_1 = e_1^{\ell_1}\dots e_n^{\ell_n}$, and $r_2 = e_1^{\ell'_1}\dots e_n^{\ell'_n}$. So there exist

- $s_1 = c_1^{\ell_1}\dots c_n^{\ell_n}$, $s_1.\alpha_1 \in \mathcal{O}[\![W;\bar{A}]\!]$
- $s_2 = d_1^{\ell'_1}\dots d_n^{\ell'_n}$, $s_2.\alpha_2 \in \mathcal{O}[\![W;\bar{B}]\!]$,

such that $r_1 \equiv s_1$, and $r_2 \equiv s_2$.

We may assume without loss of generality that the sets $BV(r_1), BV(s_1)$, and $BV(s_2)$ have no variables in common. Define $s'_1 = f_1^{\ell_n}\dots f_n^{\ell_n}$ and $s'_2 = g_1^{\ell'_1}\dots g_n^{\ell'_n}$, where, for $1 \le i \le n$,

$$f_i = g_i = \begin{cases} c_i & \text{if } \ell_i = O \\ d_i & \text{if } \ell'_i = O \\ e_i & \text{otherwise} \end{cases}$$

Note that $s'_1 \parallel s'_2$ is defined.

It remains to be shown that $s'_1.\alpha_1 \in \mathcal{O}[\![W;\bar{A}]\!]$, $s'_2.\alpha_2 \in \mathcal{O}[\![W;\bar{B}]\!]$, that is, $s'_1.\alpha_1 \parallel s'_2.\alpha_2 \in \mathcal{O}[\![W;\bar{A},\bar{B}]\!]$, and $s'_1 \parallel s'_2 \equiv s_1 \parallel s_2$. For this purpose we introduce the following

Lemma A.1 *For $1 \le i \le n$ we have*

$$\models Estore(f_1^{\ell_1}\dots f_i^{\ell_i}) \Leftrightarrow Estore(c_1^{\ell_1}\dots c_i^{\ell_i})$$

and

$$\models Estore(g_1^{\ell'_1}\dots g_i^{\ell'_i}) \Leftrightarrow Estore(d_1^{\ell'_1}\dots d_i^{\ell'_i}).$$

Proof of lemma A.1 We prove the first equivalence by induction on i (the second is treated similarly): suppose the proposition holds for i. We consider the cases $\ell_{i+1} = I$, $\ell_{i+1} = O$ separately.

Let $\ell_{i+1} = O$, so $f_{i+1} = c_{i+1}$. Furthermore, let $out(s)$ $[in(s)]$ be the subsequence of s consisting of all the output [input] constraints. We have

$$\begin{array}{lll} \models & Estore(f_1^{\ell_1}\dots f_{i+1}^{\ell_{i+1}}) & \Leftrightarrow \text{ (since } BV(s_1)\cap BV(r_1) = BV(s_1)\cap BV(s_2) = \emptyset) \\ & Estore(out(f_1^{\ell_1}\dots f_{i+1}^{\ell_{i+1}})) \wedge Estore(f_1^{\ell_1}\dots f_i^{\ell_i}) & \Leftrightarrow \\ & Estore(out(c_1^{\ell_1}\dots c_{i+1}^{\ell_{i+1}})) \wedge Estore(c_1^{\ell_1}\dots c_i^{\ell_i}) & \Leftrightarrow \text{ (since } BV^I(s_1)\cap BV^O(s_1) = \emptyset) \\ & Estore(c_1^{\ell_1}\dots c_{i+1}^{\ell_{i+1}}). & \end{array}$$

Now let $\ell_{i+1} = \ell'_{i+1} = I$, so $f_{i+1} = e_{i+1}$. We have

$$\begin{array}{lll} \models & Estore(f_1^{\ell_1}\dots f_{i+1}^{\ell_{i+1}}) & \Leftrightarrow \text{ (since } BV(r_1)\cap BV(s_1) = BV(r_1)\cap BV(s_2) = \emptyset) \\ & Estore(in(e_1^{\ell_1}\dots e_n^{\ell_n})) \wedge Estore(f_1^{\ell_1}\dots f_i^{\ell_i}) & \Leftrightarrow \\ & Estore(in(e_1^{\ell_1}\dots e_n^{\ell_n})) \wedge Estore(c_1^{\ell_1}\dots c_i^{\ell_i}) & \Leftrightarrow \text{ (since } r_1 \equiv s_1) \\ & Estore(in(e_1^{\ell_1}\dots e_n^{\ell_n})) \wedge Estore(e_1^{\ell_1}\dots e_i^{\ell_i}) & \Leftrightarrow \text{ (since } BV^I(r_1)\cap BV^O(r_1) = \emptyset) \\ & Estore(e_1^{\ell_1}\dots e_{i+1}^{\ell_{i+1}}) & \Leftrightarrow \text{ (since } r_1 \equiv s_1) \\ & Estore(c_1^{\ell_1}\dots c_{i+1}^{\ell_{i+1}}) & \end{array}$$

Finally, let $\ell_{i+1} = I$ and $\ell'_{i+1} = O$. We have

$$\begin{array}{lll}
\models & Estore(f_1^{\ell_1} \dots f_{i+1}^{\ell_{i+1}}) & \Leftrightarrow \text{(since } BV(r_1) \cap BV(s_2) = \\
 & & \qquad BV(s_1) \cap BV(s_2) = \emptyset) \\
 & Estore(out(d_1^{\ell'_1} \dots d_{i+1}^{\ell'_{i+1}})) \wedge Estore(f_1^{\ell_1} \dots f_i^{\ell_i}) & \Leftrightarrow \\
 & Estore(out(d_1^{\ell'_1} \dots d_{i+1}^{\ell'_{i+1}})) \wedge Estore(c_1^{\ell_1} \dots c_i^{\ell_i}) & \Leftrightarrow \text{(since } r_1 \equiv s_1 \text{ and } r_2 \equiv s_2) \\
 & Estore(out(d_1^{\ell'_1} \dots d_{i+1}^{\ell'_{i+1}})) \wedge Estore(d_1^{\ell'_1} \dots d_i^{\ell'_i}) & \Leftrightarrow \text{(since } BV^I(s_2) \cap BV^O(s_2) = \emptyset) \\
 & Estore(d_1^{\ell'_1} \dots d_{i+1}^{\ell'_{i+1}}) & \Leftrightarrow \text{(since } r_2 \equiv s_2) \\
 & Estore(c_1^{\ell_1} \dots c_{i+1}^{\ell_{i+1}}) &
\end{array}$$

□

¿From lemma A.1 it follows that $s'_1 \parallel s'_2 \equiv r_1 \parallel r_2$.

Furthermore, $s'_1.\alpha_1 \in \mathcal{O}[\![W; \bar{A}]\!]$ can easily be seen to follow from the observation that if $\models Store(c_1^{\ell_1} \dots c_i^{\ell_i}) \Rightarrow c$, with $FV(c) \cap BV^I(c_1^{\ell_1} \dots c_i^{\ell_i}) = \emptyset$, then $\models Store(f_1^{\ell_1} \dots f_i^{\ell_i}) \Rightarrow c$. In fact, given some first-order model, let σ be an assignment of values of that model to the variables. Then we have, by lemma A.1, that

$$\sigma \models Store(f_1^{\ell_1} \dots f_i^{\ell_i})$$

implies

$$\sigma \models Estore(c_1^{\ell_1} \dots c_i^{\ell_i})$$

Since $BV^I(s_1) \cap BV^O(s_1) = \emptyset$, we have

$$\models Estore(c_1^{\ell_1} \dots c_i^{\ell_i}) \Leftrightarrow Estore(out(c_1^{\ell_1} \dots c_i^{\ell_i})) \wedge Estore(in(c_1^{\ell_1} \dots c_i^{\ell_i}))$$

Furthermore, by definition,

$$out(f_1^{\ell_1} \dots f_i^{\ell_i}) = out(c_1^{\ell_1} \dots c_i^{\ell_i}),$$

therefore we have

$$\sigma \models Store(out(c_1^{\ell_1} \dots c_i^{\ell_i})) \wedge Estore(in(c_1^{\ell_1} \dots c_i^{\ell_i}))$$

¿From which we conclude, by the validity of $Store(c_1^{\ell_1} \dots c_i^{\ell_i}) \Rightarrow c$ and the restriction $FV(c) \cap BV^I(c_1^{\ell_1} \dots c_i^{\ell_i}) = \emptyset$, that $\sigma \models c$. In a similar way it follows that $s'_2.\alpha_2 \in \mathcal{O}[\![W; \bar{B}]\!]$. □

B A property of the closure conditions

In this appendix we prove:

Lemma 5.1 *For arbitrary sets S_1 and S_2 of modular sequences which are closed under* **P2** *and the following version of* **P1**

P1-I $s_1.c_1^I \ldots c_n^I.s_2.\alpha \in R \Rightarrow s_1.c'^I_1 \ldots c'^I_m.s_2.\alpha \in R$
if $\models Estore(s_1.c_1^I \ldots c_n^I) \Leftrightarrow Estore(s_1.c'^I_1 \ldots c'^I_m)$

we have

$$Closure(Closure(S_1) \parallel Closure(S_2)) = Closure(S_1 \parallel S_2)$$

Proof It suffices to prove that $Closure(S_1) \parallel Closure(S_2) \subseteq Closure(S_1 \parallel S_2)$. Let $s_1.\alpha_1 \in S_1$ and $s_2.\alpha_2 \in S_2$. Furthermore let $s'_1.\alpha_1 \in Closure(s_1.\alpha_1)$ and $s'_2.\alpha_2 \in Closure(s_2.\alpha_2)$ such that $s'_1.\alpha_1 \parallel s'_2.\alpha_2$ is defined. We prove by induction on the number of applications of **P1-O**, the closure condition **P1** for $\ell = O$, that $s'_1.\alpha_1 \parallel s'_2.\alpha_2 \in Closure(S_1 \parallel S_2)$.

The case that the number of applications of **P1-O** equals zero follows immediately from the closedness of S_1 and S_2 under **P1-I** and **P2**.

For the sake of a smooth presentation let's introduce the notation $s \Rightarrow s'$ for the derivability of s' from s by one application of an arbitrary closure condition.

Now let

$$s_1 \Rightarrow^* s_{11}.c_1^O \ldots c_n^O.s_{12} \Rightarrow s_{11}.c'^O_1 \ldots c'^O_m.s_{12} \Rightarrow^* s'_1$$

where $s_{11}.c'^O_1 \ldots c'^O_m.s_{12} \Rightarrow^* s'_1$ consists only of applications of **P1-I** and **P2**. We have

$$s'_1 = s'_{11}.c'^O_1.d_1^I.c'^O_2 \ldots d_{m-1}^I.c'^O_m.s'_{12}$$

where the input constraints are introduced by **P2**, $s_{11} \Rightarrow^* s'_{11}$ and $s_{12} \Rightarrow s'_{12}$, both derivations using only **P1-I** and **P2**. (The slightly more general case that some of the d_i's introduced are actually sequences of input constraints can be treated in the same way, but requires a more elaborate notation which might obscure the underlying idea.)

So we have

$$s'_2 = s'_{21}c'^I_1.d_1^{\ell_1}.c'^I_2 \ldots d_{m-1}^{\ell_{m-1}}.c'^I_m.s'_{22}$$

such that $s'_{11} \parallel s'_{21}$ and $s'_{12} \parallel s'_{22}$ are defined.

Define

$$r_1 = s'_{11}.c_1^O \ldots c_n^O.d_1^I \ldots d_{m-1}^I.s'_{12}$$

It is not difficult to see that $s_1 \Rightarrow^* r_1$, where the number of applications of **P1-O** is one less than that in $s_1 \Rightarrow^* s'_1$: $s_1 \Rightarrow^* s_{11}.c_1^O \ldots c_n^O.s_{12} \Rightarrow^* s'_{11}.c_1^O \ldots c_n^O.s'_{12} \Rightarrow^* s'_{11}.c_1^O \ldots c_m^O.d_1^I \ldots d_{m-1}^I.s'_{12}$.

Furthermore let

$$r_2 = s'_{21}.c_1^I \ldots c_n^I.d_1^{\ell_1} \ldots d_{m-1}^{\ell_{m-1}}.s'_{22}$$

Again it is not difficult to check that $s_2 \Rightarrow^* s'_2 \Rightarrow^* r_2$, with the same number of applications of **P1-O** as in the derivation $s_2 \Rightarrow^* s'_2$.

So we are now in a position which allows us to apply the induction hypothesis: $r_1.\alpha_1 \parallel r_2.\alpha_2 \in Closure(S_1 \parallel S_2)$.

Finally, we have $r_1.\alpha_1 \parallel r_2.\alpha_2 \Rightarrow^* s'_1.\alpha_1 \parallel s'_2.\alpha_2$, which we leave the reader to verify. □