Full Abstraction for Linda

Cinzia Di Giusto **Maurizio Gabbrielli**

Technical Report UBLCS-

October 2007

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2007-02 *Towards Cooperative, Self-Organised Replica Management*, Hales, D., Marcozzi, A., Cortese, G., February 2007.

2007-03 *A Model and an Algebra for Semi-Structured and Full-Text Queries (PhD Thesis)*, Buratti, G., March 2007.

2007-04 *Data and Behavioral Contracts for Web Services (PhD Thesis)*, Carpineti, S., March 2007.

2007-05 *Pattern-Based Segmentation of Digital Documents: Model and Implementation (PhD Thesis)*, Di Iorio, A., March 2007.

2007-06 *A Communication Infrastructure to Support Knowledge Level Agents on the Web (PhD Thesis)*, Guidi, D., March 2007.

2007-07 *Formalizing Languages for Service Oriented Computing (PhD Thesis)*, Guidi, C., March 2007.

2007-08 *Secure Gossiping Techniques and Components (PhD Thesis)*, Jesi, G., March 2007.

2007-09 *Rich Media Content Adaptation in E-Learning Systems (PhD Thesis)*, Mirri, S., March 2007.

2007-10 *User Interaction Widgets for Interactive Theorem Proving (PhD Thesis)*, Zacchiroli, S., March 2007.

2007-11 *An Ontology-based Approach to Define and Manage B2B Interoperability (PhD Thesis)*, Gessa, N., March 2007.

2007-12 *Decidable and Computational Properties of Cellular Automata (PhD Thesis)*, Di Lena, P., March 2007.

2007-13 *Patterns for Descriptive Documents: a Formal Analysis*, Dattolo, A., Di Iorio, A., Duca, S., Feliziani, A. A., Vitali, F., April 2007.

2007-14 *BPM + DM = BPDM*, Magnani, M., Montesi, D., May 2007.

2007-15 *A Study on Company Name Matching for Database Integration*, Magnani, M., Montesi, D., May 2007.

2007-16 *Fault Tolerance for Large Scale Protein 3D Reconstruction from Contact Maps*, Vassura, M., Margara, L., di Lena, P., Medri, F., Fariselli, P., Casadio, R., May 2007.

2007-17 *Computing the Cost of BPMN Diagrams*, Magnani, M., Montesi, D., June 2007.

2007-18 *Expressing Priorities, External Probabilities and Time in Process Algebra via Mixed Open/Closed Systems*, Bravetti, M., June 2007.

2007-19 *Design and Evaluation of a Wide-Area Distributed Shared Memory Middleware*, Mazzucco, M., Morgan, G., Panzieri, F., July 2007.

2007-20 *An Object-based Fault-Tolerant Distributed Shared Memory Middleware*, Lodi, G., Ghini, V., Panzieri, F., Carloni, F., July 2007.

2007-21 *Templating Wiki Content for Fun and Profit*, Di Iorio, A., Zacchiroli, S., Vitali, F., August 2007.

2007-22 *EPML: Executable Process Modeling Language*, Rossi, D., Turrini, E., September 2007.

2007-23 *Stream Processing of XML Documents Made Easy with LALR(1) Parser Generators*, Padovani, L., Zacchiroli, S., September 2007.

Full Abstraction for Linda

Cinzia Di Giusto [1] **Maurizio Gabbrielli** [1]

Technical Report UBLCS-

October 2007

Abstract

This paper investigates full abstraction of a trace semantics for two Linda-like languages. The first language provides primitives for adding and removing messages from a shared memory, local choice, parallel composition and recursion. The second one adds the possibility of checking for the absence of a message in the store. After having defined a denotational semantics based on traces, we obtain fully abstract semantics for both languages by using suitable abstractions in order to identify different traces which do not correspond to different operational behaviours.

1. Dip. Scienze dell'Informazione, Università di Bologna, Bologna, Italy.

1 Introduction

One of the fundamental purposes of a semantics is to provide a rigorous mean for proving the correctness of programs w.r.t. some behavioural specification. Several different tools (operational, denotational, algebraic and logic) can be used to this aim and ideally one would like to have a compositional and fully abstract semantics.

Compositionality is of course an important feature since it is the foundation for managing large systems' complexity when considering program verification, analysis and (modular) design. Most of the above mentioned tools indeed allow to obtain rather easily a compositional semantics.

Full abstraction is also a desirable feature since it allows to simplify and "economize" as much as possible a semantics while preserving its correctness, however in general it is a rather difficult target to achieve. To be more precise and to set the ground for the content of this paper, following [4, 9, 12] we can summarize the terms of the problem as follows. Given a language L define a semantics that associates to each process (or program) P in L a set of observable properties $\mathcal{O}(P)$. This is usually done in operational terms by using a transition system and a suitable definition of $\mathcal{O}(P)$ which identifies computational aspects relevant for a specific class of applications. In case such semantics is compositional, i.e. if we can reconstruct $\mathcal{O}(P\ op\ Q)$ from $\mathcal{O}(P)$ and $\mathcal{O}(Q)$ for any operator op of the language L, we have a satisfactory semantics, since the observational equivalence on processes induced by $\mathcal{O}(P)$ is preserved by contexts. More precisely, we have that $\mathcal{O}(P) = \mathcal{O}(Q)$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[P]) = \mathcal{O}(C[Q])$.

However often this is not the case and in order to obtain a compositional semantics some richer semantic structures than those used in $\mathcal{O}(P)$ need to be considered. For example, as we will see in Section 4, typically pairs representing the input/output behaviour of a process are not sufficient to obtain compositionality and one has to use traces. It can happen that these richer semantic structures "add too much" in the sense that the semantics $[\![\cdot]\!]$ based on them allows to distinguish processes which have the same behaviour w.r.t. $\mathcal{O}(P)$, under any possible context. In this case suitable abstractions must be used in $[\![\cdot]\!]$ in order to obtain a fully abstract result which, in general, can be stated as follows: $[\![P]\!] = [\![Q]\!]$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[P]) = \mathcal{O}(C[Q])$ holds.

As we discuss in Section 6, fully abstract semantics based on traces for input/output observables have been studied many years ago for several concurrent languages. However, no one has addressed this problem for a Linda-like language, so far.

In this paper we then investigate the full abstraction problem, as described above, for two variants of Linda. Linda is a programming paradigm [11] which allows interprocess communication through a shared data space, also called tuple space, where processes can post and retrieve messages (also called tuples). The shared memory paradigm offers some advantages since it decouples communication between processes: communication is in fact asynchronous and processes do not need to be aware of each other identity or location. Indeed, the Linda paradigm has received also a commercial interest, mainly due to the applications which use the Java Spaces from Sun Microsystems [10] and TSpaces from IBM [13] models, both based on Linda (a more detailed comparison of Linda implementations can be found in [19]). Distributed Linda-like languages have also been investigated. Notably, Klaim [17] is an implemented language based on the Linda paradigm where the central store is replaced by several distributed local stores and processes' mobility among different locations is supported.

Many different formalizations and variants of Linda have been defined. Here we use essentially the process-algebraic formalization of Linda introduced in [6, 7] and we consider the two very basic Linda dialects. The first one is the core language which contains only the two Linda primitives, in and out which allow to remove and add messages to the store, respectively, plus some usual operators in process algebra: choice, parallel composition and recursion. For this language, called Linda-core, we define a compositional, fixpoint trace semantics which is correct but not fully abstract when considering the input/output pairs. Hence we introduce a suitable abstraction on traces and show that this allows us to obtain a fully abstract semantics. The second dialect enriches the syntax of Linda-core by allowing also a construct (inp) which allows to check

the absence of information in the store. We prove that in this case a much simpler abstraction on traces is sufficient to obtain a full abstraction result. This accounts for the augmented expressive power of the language with inp, which can be formally proven by using the techniques in [6, 20].

The remaining of the paper is organized as follows. Section 2 introduces the Linda languages under consideration while Section 3 defines their denotational semantics. We then provide the fully abstract semantics for the core language in Section 4. Section 5 contains the main theorem on the full abstraction for the language extended with the inp primitive. Finally, Section 6 concludes by discussing some related works.

2 Preliminaries

In this section, following the process algebraic view of Linda proposed in [6] we recall the syntax of the Linda languages that we consider and their operational semantics.

2.1 Linda-core

As previously mentioned, Linda is a paradigm which provides a simple model to describe communication between processes. The central notion in Linda is the one of *tuple space*. A tuple space is a shared data space (i.e. a common store) where all the *tuples* representing the information to be exchanged are stored. Here we shall abstract from the specific nature of tuples assuming that these are elementary messages. Communication is represented by the concurrent and asynchronous activity of several processes which add or remove messages from the common store. I.e. the sender dispatches a message through a non-blocking operation which adds the tuple in the tuple space. Then the message has an independent existence until a receiver retrieves and removes it from the shared space. Such kind of communication is called *generative* (see [11]).

Processes of the language Linda-core, denoted by $P, Q, \ldots$, are then given by the following grammar:

$$P ::= \mathbf{0} \mid out(a).P \mid in(a).P \mid P \mid P \mid P + P \mid recX.P \tag{1}$$

where we assume that $a \in Msg$ and Msg denotes the set of all possible messages (or tuples), ranged over by $a, b, \ldots$.

Intuitively $\mathbf{0}$ represents the process that does nothing. Then the process $out(a).P$ adds the message a to the store and then behaves as P. The message a which has been added to the store will be visible to other processes only after the completion of the $out(a)$ action, however note that other interpretations are possible for this primitive (see [5]). If a is present in the tuple space, $in(a).P$ removes the message and then behaves as P. Otherwise if a is not present, the process $in(a).P$ is suspended until a becomes available in the store. The parallel construct $P \mid Q$ is interpreted in terms of interleaving. The process $P + Q$ can non-deterministically choose to behave either as P or as Q (hence we have a form of local choice). Finally we have the recursion operator where we assume that guarded recursion is used.

The operational semantics of Linda-core is described by means of a transition system $T = (Conf, \rightarrow)$. Configurations $Conf$ are pairs of the form $\langle P, \mathcal{M} \rangle$ where P is a process and $\mathcal{M}$ is a multiset containing tuples, also called tuple space or store. The transition relation $\rightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 1, which should be self-explaining, provided we introduce the following notation.

Notation 1. *To describe updates in the store we will use $\oplus$ and $\ominus$ to denote multisets union and difference, respectively. So $\mathcal{M} \oplus \{a\}$ means that a message (a tuple) ' a' has been added to the store while $\mathcal{M} \ominus \{a\}$ indicates that a copy of ' a' has been removed.*

A transition $\langle P, \mathcal{M} \rangle \rightarrow \langle Q, \mathcal{M}' \rangle$ then means that the process P reduces to Q, possibly by producing some changes in the store which evolves from $\mathcal{M}$ to $\mathcal{M}'$. A sequence of configurations is called run or computation. The reflexive transitive closure of $\rightarrow$ is denoted by $\Rightarrow$.

By using the transition system described above we can characterize several different notions of observables. The ones we are interested here consider simply the input/output behaviour of a process in terms of the tuple space. The input is therefore the initial tuple space, while the output

R1 $\langle out(a).P, \mathcal{M}\rangle \rightarrow \langle P, \mathcal{M} \oplus \{a\}\rangle$

R2 $\langle in(a).P, \mathcal{M} \oplus \{a\}\rangle \rightarrow \langle P, \mathcal{M}\rangle$

R3 $\dfrac{\langle P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}{\langle P \mid Q, \mathcal{M}\rangle \rightarrow \langle P' \mid Q, \mathcal{M}'\rangle}$ and $\dfrac{\langle Q, \mathcal{M}\rangle \rightarrow \langle Q', \mathcal{M}'\rangle}{\langle P \mid Q, \mathcal{M}\rangle \rightarrow \langle P \mid Q', \mathcal{M}'\rangle}$

R4 $\langle P + Q, \mathcal{M}\rangle \rightarrow \langle P, \mathcal{M}\rangle$ and $\langle P + Q, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}\rangle$

R5 $\dfrac{\langle P[recX.P/X], \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}{\langle recX.P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}'\rangle}$

Table 1. An operational semantics for Linda-core.

R6 $\langle inp(a)?P : Q, \mathcal{M} \oplus \{a\}\rangle \rightarrow \langle P, \mathcal{M}\rangle$
$\langle inp(a)?P : Q, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}\rangle$ provided $a \notin \mathcal{M}$

Table 2. The rule for inp.

is the final store produced by a process which cannot further proceed in the computation (either because it is suspended on an in operation or because it has consumed all the actions). More precisely we define the observables as follows.

Definition 2 (Observables $\mathcal{O}(P)$)**.** *Let P be a Linda process. We define*

$$\mathcal{O}(P) = \{(\mathcal{M}_1, \mathcal{M}_n) \mid \langle P, \mathcal{M}_1\rangle \Rightarrow \langle P_n, \mathcal{M}_n\rangle \not\rightarrow\} \tag{2}$$

2.2 Linda-inp

We will now introduce a slightly different variant of Linda, called Linda-inp, obtained by adding a new operator $inp(a)?P : Q$ which allows also to check whether a message is not present in the store. More precisely, the previous construct checks whether the store contains the message a: if the message is present in the store then the process continues with P, otherwise with Q.

Therefore we will add to the grammar in (1) the following primitive:

$$P ::= inp(a)?P : P \tag{3}$$

The operational semantics for Linda-inp is obtained by (a transition system defined by) adding to the rules of Table 1 the rules contained in Table 2. The observables can be defined as before.

3 Denotational semantics

It is easy to see that the operational semantics which associates to a process P its observables $\mathcal{O}(P)$ is not compositional. For example consider the processes $P = out(a).in(a).out(b)$ and $Q = out(b)$. Then $\mathcal{O}(P) = \mathcal{O}(Q)$ holds, however, considering the process $R = in(a).out(ok)$ we have that $(\emptyset, \{ok\}) \in \mathcal{O}(P \mid R) \setminus \mathcal{O}(Q \mid R)$ which means that the observables of a parallel composition cannot be obtained from the observables of the two processes being composed (in parallel). This problem is in general well known, in fact in order to obtain a compositional model more informative structures than input/output pairs have been used. In particular, models based on traces (or sequences) have been used for many concurrent languages, starting from the early works on dataflow languages [16], imperative ones [4] and concurrent constraint programming [9].

In the following we will define a compositional semantics which correctly models the $\mathcal{O}(P)$ observables and which is based on traces. This semantics is similar to those used for timed Linda in [8] (and therefore to that one of [9]), even though the technical treatment is different. In fact in

D1 $[\![\mathbf{0}]\!] = \{\epsilon\}$

D2 $[\![out(a).P]\!] = \{out(a) \cdot s \mid s \in [\![P]\!]\}$

D3 $[\![in(a).P]\!] = \{in(a) \cdot s \mid s \in [\![P]\!]\} \cup \{\overline{in}(a)\}$

D4 $[\![P \mid Q]\!] = [\![P]\!] \tilde{|} [\![Q]\!]$
where the operator $\tilde{|}$ is inductively defined as follow:

$$(x \cdot s) \tilde{|} y = y \tilde{|} (x \cdot s) = \{(x \cdot t) \mid t \in s \tilde{|} y\} \cup \{y \cdot x \cdot s\}$$
$$(x \cdot s) \tilde{|} (y \cdot t) = (y \cdot t) \tilde{|} (x \cdot s) =$$
$$\{(x \cdot u) \mid u \in s \tilde{|} (y \cdot t)\} \cup \{(y \cdot u) \mid u \in (x \cdot s) \tilde{|} t\}$$

with $x, y \in \mathcal{A}$ and $s, t, u \in \mathcal{S}$.

D5 $[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!]$

D6 $[\![recX.P]\!] = [\![P[recX.P/X]]\!]$

Table 3. A denotational semantics for Linda-core.

[8], where maximal parallelism was assumed, the denotational model was used traces of pairs of tuple spaces, representing the input and the output at each step of the computation. Here, due to the interleaving semantics and to local choice, this kind of sequences is not sufficient to obtain a correct model. Essentially the problem is that we have to distinguish the processes $out(a) \mid in(a)$ and $out(a).in(a) + in(a).out(a)$ (because when starting with a empty store the second process can produce an empty store as aresult) and this cannot be done by using simply input/output pairs. Hence, here we consider a denotational model which associates to a process a set of sequences of the form $\alpha_1, \ldots, \alpha_n$ where each α_i is an element of the set

$$\mathcal{A} = \{in(a), out(a), \overline{in}(a), \overline{inp}(a) \mid a \in Msg\} \tag{4}$$

(where Msg denotes all the possible messages, as previously mentioned). The first two kinds of actions in $\mathcal{A}$ are obvious as they represent the corresponding operations on the store, $\overline{in}(a)$ and $\overline{inp}(a)$ are used to express absence of information. We denote with $\mathcal{S}$ the set of all possible sequences defined in this way.

We introduce now two denotational semantics (one for each language we are considering) based on traces which are compositional by construction. Such semantics are the least functions $[\![\cdot]\!] : Processes \to 2^{\mathcal{S}}$, which satisfy the equations in Table 3 for Linda-core and the equations in Table 3 plus that in Table 4 for Linda-inp. The order on functions here is the one induced by set inclusion on the co-domain. Well known fixpoint results allow to obtain the semantics as the least fixpoints of the operators defined implicitly by the equations in the Tables.

3.1 Denotational semantics for Linda-core

The equations should be self-explanatory apart from a few details. The denotation of the **0** process is the empty sequence, while the equations D2 and D3 show the expected behaviour for the basic primitives. Note that in equation D3 we have two cases: the first one corresponds to the case in which a is present in the store, thus the computation can proceed (with the sequence s) after the in action. On the other hand, the $\overline{in}(a)$ action represents the absence of a in the store, in which case the computation terminates (the process is suspended). The parallel operator is interpreted in terms of interleaving as usual, while since the choice is local, it can be modeled by a simple set union. Recursion is treated in the usual way.

In order to show that the denotational semantics is correct w.r.t. our notion of observables we will introduce a fixpoint characterization of the operational semantics which will be used as an

intermediate step towards $[\![\cdot]\!]$.

Definition 3. *Let $\Phi_1 \in (Processes \to 2^{\mathcal{S}}) \to Processes \to 2^{\mathcal{S}}$ be defined as follows:*

$$\begin{aligned}\Phi_1(I)(P) = &\{out(a) \cdot t \mid \langle P, \mathcal{M}\rangle \to \langle P', \mathcal{M} \oplus a\rangle \text{ and } t \in I(P')\} \cup \\ &\{in(a) \cdot t \mid \langle P, \mathcal{M}\rangle \to \langle P', \mathcal{M} \ominus a\rangle \text{ and } t \in I(P')\} \cup \\ &\{\overline{in}(a) \mid \langle P, \mathcal{M}\rangle \nrightarrow \text{ and } P \neq \mathbf{0}\} \cup \\ &\{t \mid \langle P + Q, \mathcal{M}\rangle \to \langle P, \mathcal{M}\rangle \text{ and } t \in I(P)\} \cup \\ &\{t \mid \langle P + Q, \mathcal{M}\rangle \to \langle Q, \mathcal{M}\rangle \text{ and } t \in I(Q)\} \cup \\ &\{\epsilon \mid P = \mathbf{0}\}.\end{aligned}$$

We define $R(P)$ as the least fixed point of Φ_1 (the ordering is the one induced by set inclusion).

Using previous definition we can prove that:

Lemma 4. *Given a process P: $[\![P]\!] \setminus \{s \cdot \overline{in}(a) \cdot t \mid t \neq \epsilon\} = R(P)$*

Proof. Let $G = \{s \cdot \overline{in}(a) \cdot t \mid t \neq \epsilon\}$. We will proceed using a double induction on the structure of the process and on the length of a sequence s: if $P = \mathbf{0}$ the case is trivial.

If $P = out(a).P'$ then by definition $[\![P]\!]$ satisfies the following equation: $[\![P]\!] = \{out(a) \cdot t \mid t \in [\![P']\!]\}$ and since the suffixes remain the same in $[\![P]\!]$ and $[\![P']\!]$ it can be easily shown that $[\![P]\!] \setminus G = \{out(a) \cdot t \mid t \in [\![P']\!] \setminus G\}$. By inductive hypothesis this is equal to $\{out(a) \cdot t \mid t \in R(P')\}$. By definition 3 and observing that the first evolution of P is $\langle P, \mathcal{M}\rangle \to \langle P', \mathcal{M} \oplus a\rangle$ for every possible $\mathcal{M}$ then $\{out(a) \cdot t \mid t \in R(P')\} = \Phi_1(R)(P)$ and since R is a fixed point of $\Phi_1(R)(P) = R(P)$.

If $P = in(a).P'$ then by definition $[\![P]\!]$ satisfies the following equation: $[\![P]\!] = \{in(a) \cdot t \mid t \in [\![P']\!]\} \cup \{\overline{in}(a)\}$ and since the suffixes remain the same in $[\![P]\!]$ and $[\![P']\!]$ it can be easily shown that $[\![P]\!] \setminus G = \{in(a) \cdot t \mid t \in [\![P']\!] \setminus G\} \cup \{\overline{in}(a)\}$. By inductive hypothesis this is equal to $\{in(a) \cdot t \mid t \in R(P')\} \cup \{\overline{in}(a)\}$. By definition 3 and observing that depending on the store the first evolution of P can be either $\langle P, \mathcal{M}\rangle \to \langle P', \mathcal{M} \ominus a\rangle$ or $\langle P, \mathcal{M}\rangle \nrightarrow$ then $\{in(a) \cdot t \mid t \in R(P')\} \cup \{\overline{in}(a)\} = \Phi_1(R)(P)$ and since R is a fixed point of $\Phi_1(R)(P) = R(P)$.

If $P = P_1 \mid P_2$ then following from definition 3 and since the interleaving does not change action's order within the sequence it can be proved that $[\![P]\!] \setminus G = ([\![P_1]\!] \setminus G \mathbin{\tilde{|}} [\![P_2]\!] \setminus G) \setminus G$. Therefore applying the inductive hypothesis we obtain $(R(P_1) \mathbin{\tilde{|}} R(P_2)) \setminus G$. We will now prove that $(R(P_1) \mathbin{\tilde{|}} R(P_2)) \setminus G = R(P)$.

- $\subseteq$ Let $s \in (R(P_1) \mathbin{\tilde{|}} R(P_2)) \setminus G$ therefore there exist two sequences $s_1 \in R(P_1)$ and $s_2 \in R(P_2)$ such that $s = s_1 \mathbin{\tilde{|}} s_2$. From the definition of $R(P)$ we can associate every action in s with a step of the form $\langle P, \mathcal{M}\rangle \to \langle P', \mathcal{M}'\rangle$. Hence we will have a sequence $\langle P_1, \mathcal{M}_0\rangle \to \langle P_1', \mathcal{M}_0'\rangle$, $\langle P_1', \mathcal{M}_1\rangle \to \langle P_1'', \mathcal{M}_1'\rangle, \dots \langle P_1^n, \mathcal{M}_n\rangle \nrightarrow$ for s_1 and $\langle P_2, \mathcal{N}_0\rangle \to \langle P_2', \mathcal{N}_0'\rangle, \langle P_2', \mathcal{N}_1\rangle \to \langle P_2'', \mathcal{N}_1'\rangle$, $\dots \langle P_2^n, \mathcal{N}_n\rangle \nrightarrow$ for s_2. The interleaving of these two traces is obviously a sequence in $\Phi_1(R)(P)$ and therefore $s \in R(P)$.

- $\supseteq$ Conversely let $s \in (R(P)$ then we can show the existence of a sequence of steps $\langle P, \mathcal{M}_0\rangle \to \langle P', \mathcal{M}_0'\rangle, \langle P', \mathcal{M}_1\rangle \to \langle P'', \mathcal{M}_1'\rangle \dots \langle P^n, \mathcal{M}_n\rangle \nrightarrow$. Following from 1 we can isolate actions performed by P_1 and P_2. Hence we will obtain two sequences of actions s_1 and s_2 that belong respectively to $R(P_1)$ and $R(P_2)$ whose interleaving is s, thus concluding the proof.

If $P = P_1 + P_2$ then $[\![P]\!] \setminus G = ([\![P_1]\!] \setminus G) \cup ([\![P_2]\!] \setminus G)$ From the inductive hypothesis we can conclude $[\![P]\!] \setminus G = R(P_1) \cup R(P_2)$. We will now prove that $(R(P_1) \cup R(P_2)) = R(P)$

- $\subseteq$ Let $s \in (R(P_1) \cup R(P_2)) \setminus G$ therefore let $s \in R(P_1) \setminus G$. From the definition of $R(P)$ as before we can associate every action in s with a step of computation. Hence we will have a sequence $\langle P_1, \mathcal{M}_0\rangle \to \langle P_1', \mathcal{M}_0'\rangle, \langle P_1', \mathcal{M}_1\rangle \to \langle P_1'', \mathcal{M}_1'\rangle, \dots \langle P_1^n, \mathcal{M}_n\rangle \nrightarrow$. But these sequence satisfies of steps will also be present in $\Phi_1(R)(P)$ (follows from the operational semantics) and therefore $s \in R(P)$.

- $\supseteq$ Conversely let $s \in (R(P)$ then we can show the existence of a sequence of steps $\langle P, \mathcal{M}_0\rangle \to \langle P', \mathcal{M}_0'\rangle, \langle P', \mathcal{M}_1\rangle \to \langle P'', \mathcal{M}_1'\rangle, \dots \langle P^n, \mathcal{M}_n\rangle \nrightarrow$. Following from 1 those steps belong ei-

D7 $[\![inp(a)?P:Q]\!] = \{in(a)\cdot s \mid s \in [\![P]\!]\} \cup \{\overline{inp}(a)\cdot s \mid s \in [\![Q]\!]\}$

Table 4. The equations for Linda-inp.

ther to the evolution of P_1 or to P_2. Therefore $s \in \Phi_1(R)(P_1)$ (or $s \in \Phi_1(R)(P_2)$) concluding the proof.

If $P = recX.P_1$ then $R(P)$ is the least fixed point of $\Phi_1(R)(P)$, from definition 3 and from the operational semantics $\Phi_1(R)(P) = \Phi_1(R)(P[recX.P_1/X])$. Since we are dealing with guarded recursion $P[recX.P_1/X]$ is a process of the following form: $\alpha(a).Q(recX.P_1)$, $Q(recX.P_1)|S(recX.P_1)$ etc. therefore we can reduce to one of the previous cases and thus $R(P[recX.P_1/X]) = [\![P[recX.P_1/X]]\!] \setminus G = [\![P]\!] \setminus G$. □ □

We can now show that the denotational semantics is correct w.r.t. our notion of observables. To this aim we need to define the evaluation of a trace as follows (↑ means undefined).

Definition 5. *Given a trace $s \in \mathcal{S}$ and a store $\mathcal{M}$, the function $eval_1(s, \mathcal{M})$ is defined by the following cases:*

$$
\begin{aligned}
&eval_1(\epsilon, \mathcal{M}) = \mathcal{M}\\
&eval_1(out(x)\cdot t, \mathcal{M}) = eval_1(t, \mathcal{M} \oplus \{x\})\\
&eval_1(in(x)\cdot t, \mathcal{M}) = \begin{cases} eval_1(t, \mathcal{M} \ominus \{x\}) & \text{if } x \in \mathcal{M}\\ \uparrow & \text{otherwise}\end{cases}\\
&eval_1(\overline{in}(x)\cdot t, \mathcal{M}) = \begin{cases} \mathcal{M} & \text{if } x \notin \mathcal{M} \text{ and } t = \epsilon\\ \uparrow & \text{otherwise}\end{cases}
\end{aligned}
$$

Proposition 6 (Correctness)**.** *Given a Linda-core process P,*
$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!]$ and $eval_1(s, \mathcal{M}_0) \neq \uparrow\}$ holds.

Proof. Let $(\mathcal{M}, \mathcal{M}') \in \mathcal{O}(P)$ thus there exists a run $\langle P, \mathcal{M}\rangle \Rightarrow \langle P_n, \mathcal{M}'\rangle \nrightarrow$, by definition 3 there exists a sequence $t \in R(P)$ associated to the previous run. . Following from Lemma 4 $t \in [\![P]\!]$. Moreover by construction we have the following property

$$
\begin{array}{ccc}
\langle P, \mathcal{M}\rangle \rightarrow \langle P', \mathcal{M}_1\rangle & & \text{Observables}\\
\updownarrow & & \\
\alpha(x) & & R(P)\\
\updownarrow & & \\
eval_1(\alpha(x)), \mathcal{M}) = \mathcal{M}_1 & & eval_1
\end{array}
$$

Hence by induction on the length of the sequence t it can be easily proved that $eval_1(t, \mathcal{M}) = \mathcal{M}'$.

Conversely let $(\mathcal{M}, \mathcal{M}') \in \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!] \ldots\}$ therefore there exists $s \in [\![P]\!]$ such that $eval_1(s, \mathcal{M}) = \mathcal{M}'$ and $eval_1(s, \mathcal{M}) \neq \uparrow$. Hence $s \neq s_1 \cdot \overline{in}(a) \cdot s_2$ with $s_2 \neq \epsilon$ and $s \in R(P)$. Again observing the previous property it can be easily proved that there exists a sequence of configurations $\langle P, \mathcal{M}\rangle \Rightarrow \langle P_n, \mathcal{M}'\rangle \nrightarrow$. Thus $(\mathcal{M}, \mathcal{M}') \in \mathcal{O}(P)$ concluding the proof. □ □

3.2 Denotational semantics for Linda-inp

When considering the Linda-inp language the denotational semantics can be obtained from Table 3 by adding the equation in Table 4. This difference w.r.t. the case of Linda core is due to the presence of the inp, which is described by Equation D7: since when a is present both the $inp(a)$ and the $in(a)$ construct are modeled in the same way, when a is not present we have to distinguish the two cases (by using $\overline{in}(a)$ and $\overline{inp}(a)$) since it would not be correct to use the evaluation given in Definition 5 for the $in(a)$.

Following the same pattern we use in the previous section, in order to prove the correcteness of the new operator we will add to definition 3 how to treat the new operator.

Definition 7.

$$\Phi_2(I)(P) = \Phi_1(I)(P) \cup \{\overline{inp}(a) \mid \langle inp(a)?P : Q, \mathcal{M}\rangle \rightarrow \langle Q, \mathcal{M}\rangle \text{ and } t \in I(Q)\}$$

And the corresponding lemma:
Lemma 8. *Given a process P: $[\![P]\!] \setminus \{s \cdot \overline{in}(a) \cdot t \mid t \neq \epsilon\} = R(P)$*

Proof. Let $G = \{s \cdot \overline{in}(a) \cdot t \mid t \neq \epsilon\}$.

If $P = inp(a)?P_1 : P_2$ then by definition $[\![P]\!]$ satisfies the following equation: $[\![P]\!] = \{in(a) \cdot s \mid s \in [\![P_1]\!]\} \cup \{\overline{inp}(a) \cdot s \mid s \in [\![P_2]\!]\}$ and since the suffixes remain the same in $[\![P]\!]$, $[\![P_1]\!]$ and $[\![P_2]\!]$ it can be easily shown that $[\![P]\!] \setminus G = \{in(a) \cdot t \mid t \in [\![P_1]\!] \setminus G\} \cup \{\overline{inp}(a) \cdot s \mid s \in [\![P_2]\!] \setminus G\}$. By inductive hypothesis this is equal to $\{in(a) \cdot t \mid t \in R(P_1)\} \cup \{\overline{inp}(a) \cdot s \mid s \in R(P_2)\}$. By definition 3 and observing that depending on the store the first evolution of P can be either $\langle P, \mathcal{M}\rangle \rightarrow \langle P_1, \mathcal{M} \ominus a\rangle$ or $\langle P, \mathcal{M}\rangle \rightarrow \langle P_2, \mathcal{M}\rangle$ then $\{in(a) \cdot t \mid t \in R(P_1)\} \cup \{\overline{inp}(a) \cdot s \mid s \in R(P_2)\} = \Phi_2(R)(P)$ and since R is a fixed point of $\Phi_2(R)(P) = R(P)$. □

In order to prove the correctness of the model introduced above we need to add to $eval_1$ the new cases obtaining the evaluation function $eval_2$:
Definition 9. *Given a trace $s \in \mathcal{S}$ and a store $\mathcal{M}$, the function $eval_2(s, \mathcal{M})$ is defined by the following cases:*

$$eval_2(\overline{inp}(x) \cdot t, \mathcal{M}) = \begin{cases} eval_2(t, \mathcal{M}) & \text{if } x \notin \mathcal{M} \\ \uparrow & \text{otherwise} \end{cases}$$

$$eval_2(\alpha(x) \cdot t, \mathcal{M}) = eval_1(\alpha(x) \cdot t, \mathcal{M}) \text{ for } \alpha \neq \overline{inp}$$

Using the same technique of Proposition 6 it can be easily proved the following theorem that states the correctness of the denotational model:
Proposition 10 (Correctness)**.** *Given a Linda-inp process P,*
$\mathcal{O}(P) = \{(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) \mid s \in [\![P]\!]\}$ holds.
=

4 Full Abstraction for Linda-core

The aim of this section is to obtain a fully abstract semantics for the Linda-core language. The semantics introduced in the previous section represents a too fine description of the actions that affect the store, since it records all the possible changes while the observables capture only the initial and the final state. It is therefore immediate to find processes which have a different denotation, while having the same input/output behaviour under any possible context.

In order to obtain full abstraction we saturate the denotational semantics by adding all those traces which, intuitively, represent a computation whose input/output behaviour, in any possible context, can be simulated by a trace which is already in the semantics. The formal definition is as follows.
Definition 11 (Saturation)**.** *Let $T \subseteq \mathcal{S}$ be a set of traces. We define the saturation of T as the minimal set $Sat(T)$ which satisfies the following rules:*

i) if $s \in T$ then $s \in Sat(T)$

ii) if $s \cdot out(a) \cdot t \cdot in(a) \cdot v \in Sat(T)$ then $s \cdot t \cdot v \in Sat(T)$

iii) if $s \cdot out(a) \cdot t \cdot in(a) \cdot v \in Sat(T)$ then $s \cdot out(a) \cdot t \cdot in(a) \cdot out(a) \cdot in(a) \cdot v \in Sat(T)$

iv) $s \in Sat(T)$ iff $s \cdot in(a) \cdot out(a) \in Sat(T)$

v) if $s \cdot out(a) \cdot t \in Sat(T)$ then $s \cdot t' \in Sat(T)$ where $t' \in \{out(a) \mathbin{\tilde{|}} t\}$

vi) all the traces in T of the form $t \cdot \overline{in}(a) \cdot u$ with $u \neq \epsilon$ are removed;

According to the previous definition in $Sat(T)$ we add all the traces which (i) are derived (inductively) from the traces in T by performing the following operations: (ii) Removing complementary actions $out(a)$ and $in(a)$ which appear, in this order, in different places of the sequence; it is rather clear that this does not change the operational behaviour described by the original sequence. (iii) Adding a "stuttering step" represented by a sequence $out(a) \cdot in(a)$ of two complementary actions is also allowed, provided that both these actions occur before (in this order) in the sequence. Intuitively, if the $out(a)$ action does not appear before in the sequence we cannot add it, since the presence of a could trigger some new computation; moreover, since the multiplicity of a message is relevant, also in case the sequence contains $out(a)$ and not $in(a)$ we cannot add the sequence $out(a) \cdot in(a)$ because after the added $out(a)$ we would have one more a than in the original sequence, which, again, could trigger new computations. (iv) Stuttering steps of the form $in(a) \cdot out(a)$ can be safely added and removed only at the end of a sequence. (v) As stated in [5] an output prefix $out(a).P$ is observably equivalent to $out \mid P$, note that from this rule follows that the core-language cannot observe the order of appearance of messages. (vi) Finally, as one can guess from Lemma 4, $\overline{in}(a)$ represents a process suspended because the message a is not present in the store, hence it is not correct to assume that other actions could take place afterwards. Clearly this is not anymore true (apart from rule (vi)) in presence of a construct which allows to check for absence of information, as we will see in the next section.

The fully abstract semantics is obtained by applying the saturation defined above to the semantics $[\![\cdot]\!]$. In order to prove the fully abstraction result we proceed by steps. First we prove that the abstraction introduced by Sat is correct (under any context) w.r.t. $\mathcal{O}(P)$. This result is obtained by first showing that the construction of $Sat([\![P]\!])$ does not add any trace that does not respect the observables of P. This is the content of the following Proposition, whose proof is in the appendix.

Proposition 12. *Given a process P, $\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) | s \in Sat([\![P]\!])\}$.*

Proof. For Proposition 6, $\mathcal{O}(P) = \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in [\![P]\!]\}$ and since by definition $[\![P]\!] \subseteq Sat([\![P]\!])$, $\mathcal{O}(P) \subseteq \{(\mathcal{M}_0, eval_1(s, \mathcal{M}_0)) \mid s \in Sat([\![P]\!])\}$.

For the other set-inclusion we shall analyze all the possible traces in $Sat([\![P]\!])$. Thus suppose that $s = s_1 \cdot out(x) \cdot s_2 \cdot in(x) \cdot s_3 \in [\![P]\!]$. Then there exists $t = s_1 \cdot s_2 \cdot s_3$ and $u = s_1 \cdot out(x) \cdot s_2 \cdot in(x) \cdot out(x) \cdot in(x) \cdot s_3 \in Sat([\![P]\!])$. It can be easily shown that $eval_1(t, \mathcal{M}_0)$ and $eval_1(u, \mathcal{M}_0)$ are equal to $eval_1(s, \mathcal{M}_0)$ thus we are not adding anything in $\mathcal{O}(P)$. We can proceed similarly for all the other traces obtained applying the rules in definition 11. □ □

Now we are ready to state that the abstract (saturated) semantics is correct under any context w.r.t. the chosen observation criteria (a proof of the following theorem is provided in the appendix). A context $C[\bullet]$ is defined as a process with a hole, that is, a process where a subprocess is left unspecified. $C[P]$ is then the process obtained from $C[\bullet]$ by replacing $\bullet$ for the process P.

Theorem 13 (Correctness for Linda-core)**.** *Given two Linda-core process A and B, if $Sat([\![A]\!]) = Sat([\![B]\!])$ then, for every context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

Proof. We will first prove $\mathcal{O}(C[A]) \subseteq \mathcal{O}(C[B])$. Let $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[A])$ then following from Proposition 6 there exists $s \in [\![C[A]]\!]$ such that $\mathcal{M}_1 = eval_1(s, \mathcal{M}_0)$. Since the denotational semantics we provide is compositional $s = c \,\tilde{\circ}\, t$ for some suitable $\tilde{\circ}$, where $c \in [\![C[\bullet]]\!]$ and $t \in [\![A]\!]$.

Since $[\![A]\!] \subset Sat([\![A]\!]) = Sat([\![B]\!])$ then $t \in Sat([\![B]\!])$ therefore two cases could arise: (1) $t \in [\![B]\!]$ hence $s \in [\![C[B]]\!]$ and $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$. (2) $t \notin [\![B]\!]$ then there exists $u \in [\![B]\!]$ such that u is derived from t following the rules in definition 11 and $eval_1(t, \mathcal{M}_0) = eval_1(u, \mathcal{M}_0)$. Hence by induction on the structure of c it can be easily proved that $eval_1(c \,\tilde{\circ}\, u, \mathcal{M}_0) = \mathcal{M}_1$ and therefore $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$.

The other set inclusion $\mathcal{O}(C[B]) \subseteq \mathcal{O}(C[A])$ is symmetrical. □

To obtain full abstraction we need now to prove the converse of the above theorem. This is the central result of this section and is the content of the following. The details of the proof are in the appendix.

Theorem 14. *Given two Linda-core processes A and B, if $Sat([\![A]\!]) \neq Sat([\![B]\!])$ then there exists a context $C[\bullet]$ such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$.*

Proof. Suppose that there exists $t \in Sat([\![A]\!]) \setminus Sat([\![B]\!])$ and consider a generic $s \in Sat([\![B]\!])$ (thus $t \neq s$). From the definition of Sat it follows that we can choose s and t as the shortest sequences such that: (i) they do not contain sub-sequences of the form $out(x) \cdot u \cdot in(x) \cdot out(x) \cdot in(x)$, (ii) they do not contain suffixes of the form $in(x) \cdot out(x)$, (iii) every output appears as soon as possible and (iv) between two consecutive inputs the outputs are ordered in lexicographic order.

Then assume that t and s have the following form

$$t = r \cdot \alpha(x) \cdot t_1 \qquad s = r \cdot \beta(y) \cdot s_1$$

where the common prefix r can also be empty and $\alpha, \beta \in \mathcal{A}$ with $\alpha \neq \beta$.

The proof is by cases, where we analyze the first couple of different actions α and β. In each case we will construct a context $C[\bullet]$ which allows to distinguish A and B (that is, a context such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$). In the proof we will use the following notation: if $in(a_1), in(a_2), \ldots, in(a_n)$ are all the input actions which appear, in this order, in the sequence r (which can also contain other out actions), then $InComp(r)$ denotes the sequence $out(a_1) \cdot out(a_2) \cdots out(a_n)$: intuitively this sequence is a sort of complement (w.r.t. in actions) of r which allows to proceed in the computation when composed in parallel with r. Furthermore, in order to further simplify the notation, in the following we will use these assumptions:

$c_1 = InComp(r)$

c_2 is a sequence consisting of as many $in(x)$ as the $out(x)$ in r

c_3 is a sequence consisting of as many $in(y)$ as the $out(y)$ in r

We have then the following cases:

1. let $\beta(y) \cdot s_1 = \epsilon$, thus $t = r \cdot \alpha(x) \cdot t_1$ and $s = r$. Depending on t we can construct the following distinguishing contexts $C[\bullet]$:
   (a) if $t = r \cdot out(x) \cdot t_1$ then $C[\bullet] = \bullet \mid c_1.c_2.in(x).out(ok)$;
   (b) if $t = r \cdot in(x) \cdot t_1$ noticing that $t_1 \neq out(x)$, the following context can be provided $C[\bullet] = \bullet \mid c_1.out(x).InComp(t_1)$.

   The symmetric case is completely analogous.

2. $\alpha(x) = in(x)$ and $\beta(y) = in(y)$ (with $x \neq y$) then in order to distinguish the two processes we need to make further distinctions (note that by construction $t_1 \neq out(x)$):
   (a) if $eval_1(t_1, \emptyset) \neq \{x\}$ then $C[\bullet] = \bullet \mid c_1.c_3.out(x)$
   (b) if $eval_1(t_1, \emptyset) = \{x\}$ and the actions $out(y)$, $in(y)$ do not appear in t_1 then $C[\bullet] = \bullet \mid c_1.c_3.out(x).InComp(t_2)$
   (c) otherwise since $out(y)$ appears in t_1, it can be provided the following context $C[\bullet] = \bullet \mid c_1.c_3.out(x).in(y).out(y).out(y)$.

3. $\alpha(x) = out(x)$ and $\beta(y) = in(y)$ or vice versa: then it can be easily shown that the context $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(x).out(ok)$ allows to distinguish A and B.

4. $\alpha(x) = out(x)$ and $\beta(y) = out(y)$ (with $x \neq y$). By hypothesis we can choose $t = r \cdot out(x) \ldots in(v) \ldots$ and $s = r \cdot out(y) \ldots in(w) \ldots$ where $in(v)$ and $in(w)$ are the first input actions after $out(x)$ and $out(y)$ respectively. Moreover $out(x)$ does not appear before $in(w)$ in s. Then two cases could arise if $v \neq x$ then the context $C[\bullet] = \bullet \mid c_1.c_4.c_5$ where c_4 and c_5 are sequences of as many $in(v)$ and $in(w)$ as the $out(v)$ and $out(w)$ that precedes the two input actions respectively. Instead if $v = x$ then we can safely assume $in(w)$ does not appear in s and the context $C[\bullet] = \bullet \mid c_1.c_5.InComp(t_1)$ can distinguish the two processes.

5. There are some remaining cases, where the two sequences are different because of a $\overline{in}$ action. However, due to the construction of our semantics, $r \cdot \overline{in}(x) \in Sat_2([\![A]\!])$ iff $r \cdot in(x) \cdot s \in Sat_2([\![A]\!])$. Therefore we can omit to consider the sequence $r \cdot \overline{in}(x)$ and just consider the case of the sequence $r \cdot in(x) \cdot s$, which is included above.

This completes the proof. □ □

The previous two theorems can be summarized in the following immediate corollary.
Corollary 15 (Full Abstraction for Linda-core)**.** *Given two Linda-core processes A and B, $Sat([\![A]\!]) = Sat([\![B]\!])$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

5 Full abstraction for Linda-inp

Now we move to consider the language Linda-inp where we can test for the absence of a message in the store by using the primitive inp. As underlined in the introduction , such a possibility augments the expressive power of the language. In semantic terms this means that we can construct more powerful contexts, thus allowing to discriminate processes which were identified by Linda-core contexts. As a simple example, consider the two processes $A = out(a).out(b)$ and $B = out(b).out(a)$. These processes cannot be distinguished (w.r.t. the observables $\mathcal{O}$) by any Linda-core contexts, indeed the corresponding traces $out(a) \cdot out(b)$ and $out(b) \cdot out(a)$ are identified by the saturation operation. However, the context $C[\bullet] = \bullet \mid in(a).(inp(b)?out(nok) : out(ok))$ allows to distinguish them, since it allows to check that a is present and b is absent in the store. Indeed we have that $(\emptyset, ok \in \mathcal{O}(C[A]) \backslash \mathcal{O}(C[B]))$. This example shows that a fully abstract semantics for Linda-inp must induce a finer equivalence on processes than Sat or, in other terms, that a less abstract operation has to be used to saturate sequences. However the Denotational semantics provided in Section 3.2 is not fully abstract. In fact, consider the two processes $A = inp(a)?\mathbf{0} : \mathbf{0}$ and $B = in(a) + A$: these two processes cannot be distinguish by any context, yet they have a different denotational semantics. Thus we need the following definition.
Definition 16 (Saturation for Linda-inp)**.** *Let $T \subseteq \mathcal{S}$ be a set of traces. We define the inp-saturation of T as the set $Sat_2(T)$ which is obtained by performing the following steps (in this order) on T:*

1. *all the traces in T of the form $t \cdot \overline{in}(a) \cdot u$ with $u \neq \epsilon$ are removed;*
2. *all the $\overline{in}(x)$ actions in all traces are replaced by $\overline{inp}(x)$ (for any x).*

Condition 1 ensures that we obtain correct traces once we have performed the transformation in 2. In fact, $\overline{in}(a)$ comes from the evaluation of $in(a)$, when a is not present. Since such an evaluation is suspended, it is not correct to assume that some action can be performed later. Thus, before transforming $\overline{in}(a)$ into $\overline{inp}(a)$ (and therefore moving from the $eval_1$ of Definition 5 to $eval_2$ of Definition 9) we have to delete these traces. The correctness of the saturation is stated by the following proposition which can be easily proven.
Proposition 17. *Given a process P, $\mathcal{O}(P) = \{(\mathcal{M}_0, eval_2(s, \mathcal{M}_0)) | s \in Sat_2([\![P]\!])\}$*
Theorem 18. *Given two Linda-inp processes A and B, if $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$ then, for every context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

Proof. We will first prove $\mathcal{O}(C[A]) \subseteq \mathcal{O}(C[B])$. Let $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[A])$ then following from Proposition 10 there exists $s \in [\![C[A]]\!]$ such that $\mathcal{M}_1 = eval_2(s, \mathcal{M}_0)$. Since the denotational semantics we provide is compositional $s = c \,\tilde{\circ}\, t$ for some suitable $\tilde{\circ}$, where $c \in [\![C[\bullet]]\!]$ and $t \in [\![A]\!]$.

Applying the rules in definition 16 we can construct a trace t' such that $eval_2(t, \mathcal{M}_0) = eval_2(t', \mathcal{M}_0)$. Hence $t' \in Sat_2([\![A]\!])$ and since $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$, $t' \in Sat_2([\![B]\!])$ therefore two cases could arise: (1) $t' \in [\![B]\!]$ hence $s \in [\![C[B]]\!]$ and $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$. Or (2) $t' \notin [\![B]\!]$ therefore there exists $u \in [\![B]\!]$ where some of the actions $\overline{in}$ have been replaced with $\overline{inp}$ and $eval_2(t, \mathcal{M}_0) = eval_2(u, \mathcal{M}_0)$. Hence by induction on the structure of c it can be easily proved that $eval_2(c \,\tilde{\circ}\, u, \mathcal{M}_0) = \mathcal{M}_1$ and therefore $(\mathcal{M}_0, \mathcal{M}_1) \in \mathcal{O}(C[B])$.

The other set inclusion $\mathcal{O}(C[B]) \subseteq \mathcal{O}(C[A])$ is symmetrical. □ □

Theorem 19. *Given two Linda-inp processes A and B, if $Sat_2([\![A]\!]) \neq Sat_2([\![B]\!])$ then there exists a context $C[\bullet]$ such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$.*

Proof. Suppose that there exists $t \in Sat_2([\![A]\!]) \setminus Sat_2([\![B]\!])$ and consider a generic $s \in Sat_2([\![B]\!])$. Since $s \neq t$ by hypothesis, we can assume that t and s have the following form:

Let

$$t = r \cdot \alpha_1(x_1) \cdots \alpha_n(x_n)$$
$$s = r \cdot \beta_1(y_1) \cdots \beta_m(y_m)$$

where the common prefix r can also be empty and $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \in \mathcal{A}$ with $\alpha_1 \neq \beta_1$.

The proof is by cases, where we analyze the first different actions α_1 and β_1 in the sequences t and s. In each case we will construct a context $C[\bullet]$ which allows to distinguish A and B (that is, a context such that $\mathcal{O}(C[A]) \neq \mathcal{O}(C[B])$). As in the proof of Theorem 14, if $in(a_1), in(a_2), \ldots, in(a_n)$ are all the input actions which appear, in this order, in the sequence r then $InComp(r)$ denotes the sequence $out(a_1) \cdot out(a_2) \cdots out(a_n)$. Furthermore, in order to further simplify the notation, in the following we will use these assumptions:

$c_1 = InComp(r)$
c_2 is a sequence consisting of as many $in(a)$ as the $out(a)$ in r
c_3 is a sequence consisting of as many $in(b)$ as the $out(b)$ in r

We have then the following cases:

1. $t = r \cdot out(a) \cdot t_1$ and $s = r$; In this case $C[\bullet] = \bullet \mid c_1.c_2.in(a).out(ok)$ allows to distinguish A and B.

2. $t = r \cdot in(a) \cdot t_1$ and $s = r$; then $C[\bullet] = out(a).\bullet \mid c_1.c_2.inp(a)?out(ok) : out(no)$ is the distinguishing context.

3. $t = r \cdot out(a) \cdot t_1$ and $s = r \cdot out(b) \cdot s_1$. We have the following sub-cases:
   (a) If the number of $out(a)$ in t is different from the number of $out(a)$ in s then it can be easily proved that there is a context that distinguishes the two programs (essentially it is a context that *counts* the occurrences of the $out(a)$). Similarly if we are considering the b's. The following is an example.
   Example 20. If $t = out(a) \cdot in(a) \cdot out(a) \cdot out(b)$ and $s = out(b) \cdot out(a)$ then we can build the distinguishing context
   $$C[\bullet] = \bullet \mid in(a).out(a).inp(a)?out(ok) : out(no)$$
   (b) Now suppose that the number of $out(a)$ (or $out(b)$) is the same in t and s. If in t_1 or in s_1 there is an input action again it is easy to provide a distinguishing context, either by blocking the execution of the rest of the trace after the input or by querying the store for the presence/absence of messages in the store. The following provide an example.
   Example 21. If $t = out(a) \cdot in(b) \cdot out(b)$ and $s = out(b) \cdot in(b) \cdot out(a)$ then we can consider the distinguishing context
   $$C[\bullet] = \bullet \mid in(a).out(b).inp(b)?out(ok) : out(no)$$
   (c) If in t_1 and in s_1 there are only outputs then either there is an output action that it is not present in one of the two traces, and in this case it is straightforward to build a distinguishing context, or the output actions of a sequence are a permutation of output actions of the other sequence; also in this case it is easy to construct a context that distinguishes the two processes by checking the presence of a message and the absence of the other one, as shown by the following.
   Example 22. If $t = out(a) \cdot out(b)$ and $s = out(b) \cdot out(a)$ then the distinguishing context $C[\bullet] = \bullet \mid in(a).inp(b)?out(ok) : out(no)$ (as seen in the initial part of this Section).

4. $t = r \cdot out(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \in Sat_2([\![B]\!])$. It suffices to consider $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(a).out(ok)$.

5. $t = r \cdot out(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \cdot s_2 \in Sat_2([\![B]\!])$. The following situations may arise:
   (a) if $out(a) \notin s_2$ then $C[\bullet] = \bullet \mid c_1.c_2.c_3.in(a).out(ok)$;
   (b) if $in(b) \notin t_1$ then $C[\bullet] = out(b).\bullet \mid c_1.inp(b)?out(ok) : out(no)$;
   (c) otherwise the only significant case is when $s' = r \cdot \overline{inp}(b) \cdot out(a) \cdot t_1$ and therefore a suitable context can be constructed observing that the order of the actions is different (i.e. b is consumed in two different positions). This is shown in the following.
   Example 23. If $t = out(a)$ and $s = in(b) \cdot out(b) \cdot out(a)$ recalling that $s' = \overline{inp}(b) \cdot out(a)$ we can build the distinguishing context $C[\bullet] = out(b).\bullet \mid in(b).out(b).inp(b)?out(ok) : out(no)$

6. $t = r \cdot in(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s = r \cdot \overline{inp}(b) \in Sat_2(B)$. In this case $C[\bullet] = out(a).\bullet \mid c_1.c_2.c_3.inp(a)?out(ok) : out(no)$.

7. $t = r \cdot in(a) \cdot t_1$, $s = r \cdot in(b) \cdot s_1$ and $s' = r \cdot \overline{inp}(b) \cdot s_2 \in Sat_2([\![B]\!])$. We should here distinguish between the following further cases
   (a) if $out(a) \notin t_1$ and $in(a) \notin s_2$ then $C[\bullet] = out(a).\bullet \mid c_1.c_3$;
   (b) otherwise the worst possible scenario happens when $s_2 = in(a) \cdot t_1$ and t_1 and s_1 are "symmetrical" in a and b. As already shown in some preceding cases, when the order of the actions changes it is always possible to find a distinguishing context. This is shown in the following, last example.
   Example 24. Given $A = inp(a)?(out(a).in(b).out(b)) : (in(b).out(b))$, and $B = inp(b)?(out(b).in(a).out(a)) : (in(a).out(a))$ thus $Sat_2([\![A]\!]) = \{in(a) \cdot out(a) \cdot in(b) \cdot out(b), \overline{inp}(a) \cdot in(b) \cdot out(b), \dots\}$ and $Sat_2([\![B]\!]) = \{in(b) \cdot out(b) \cdot in(a) \cdot out(a), \overline{inp}(b) \cdot in(a) \cdot out(a), \dots\}$ and the following context can distinguish between the two programs: $C[\bullet] = \bullet \mid inp(a)?out(ok1) : (inp(b)?out(ok2) : out(no))$.

8. There are some remaining cases, where the two sequences are different because of a $\overline{inp}$ action. However, due to the construction of our semantics, $r \cdot \overline{inp}(x) \in Sat_2([\![A]\!])$ iff $r \cdot in(x) \cdot s \in Sat_2([\![A]\!])$. Therefore we can omit to consider the sequence $r \cdot \overline{inp}(x)$ and just consider the case of the sequence $r \cdot in(x) \cdot s$, which is included above.

This completes the proof. □ □

The main result of this section is the following one, which follows from two theorems analogous to those of the previous Section (see the appendix for the proofs).
Corollary 25 (Full Abstraction for Linda-inp)**.** *Given two Linda-inp processes A and B, $Sat_2([\![A]\!]) = Sat_2([\![B]\!])$ iff, for any context $C[\bullet]$, $\mathcal{O}(C[A]) = \mathcal{O}(C[B])$ holds.*

6 Conclusions and Related work

We have studied the full abstraction problem for two variants of the Linda paradigm. For the first one, the core Linda language, we have provided a trace semantics which is fully abstract w.r.t. the input/output notion of observables. This has been obtained by using a suitable abstraction in order to identify different traces which do not represent meaningful operational differences. The second language, Linda-inp, allows also to check for the absence of information. The augmented expressive power of this language allows us to obtain a full abstraction result by using a much simpler abstraction.

In the specific context of Linda, full abstraction has been previously investigated by [3] which used also techniques inspired by [12]. The results in [3] are completely different from ours, since in such a paper a semantics based on sequences is shown to be fully abstract with respect to a notion of observable which consider traces of computations. We prefer to consider a coarser notion of observables, consisting in the input/output behaviour, which accounts for a "black box" use of processes. Clearly our notion of observables leads to a more difficult full abstraction result, being the denotational model based on traces.

Results similar to ours have been obtained in the context of concurrent constraint programming (CCP) by De Boer and Palamidessi [9], however this language differs from Linda since it does not allow to remove information from the store. This monotonic nature of CCP makes its semantic treatment simpler, hence the results in [9] cannot be applied directly to the languages we consider here. Also Brookes [4] provides a trace model and a full abstraction result for a shared variable parallel language which is substantially different from Linda. The same applies to the results in [12].

More generally, full abstraction results have been provided for many concurrent languages and in quite various settings, which however are different from the case we consider here. In fact, even though our core Linda language can be seen as a fragment of asynchronous CCS (and therefore of asynchronous π-calculus), all the full abstraction results available for these languages consider different observational equivalences from ours. Probably the closer work in this sense is [2], where full abstraction of a trace semantics w.r.t. may testing equivalence has been studied. Note however that may testing is different from the observational equivalence that we consider (which is based on the input-output behaviour). For example, the processes $in(a).in(b)$ and $in(b).in(a)$ are may testing equivalent (see [2]) while they are not equivalent in our case, since they can be distinguished by the context $out(a)$.

Several other papers consider barbed equivalences and their relations with bisimulation, (notably [1] for asynchronous π-calculus and [6] for Linda-like process algebras) which, as previously mentioned, are completely different from the equivalence we consider. It is also worth noticing that the construct inp, which is not available in π-calculus and in CCS, change considerably the semantics of the language, thus for Linda(inp) one cannot use existing results for CCS or π-calculus. For example, [6] shows that in presence of inp the coarse congruence contained in barbed equivalence is a new, specific congruence called inp-bisimulation (while for the core language it is the usual bisimulation).

Recently, full abstraction results for π-calculus with contextual equivalence [18] and for Java-like languages with testing equivalence have been obtained in [15] (by considering weak bisimulation) and in [14] (by using a model based on traces). Also in these cases the considered equivalences are different from ours.

Our results can be extended along several lines. First of all we could investigate some other of the (many) dialects of Linda which exist in the literature. In particular we are planning to investigate the case of the language Klaim [17], which supports distribution and mobility and is currently quite attractive. Furthermore it would be interesting to consider full abstraction results for other notions of observational equivalences.

References

[1] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations of the asynchronous π-calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998.

[2] M. Boreale, R. D. Nicola, and R. Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002.

[3] A. Brogi and J.-M. Jaquet. Modeling coordination via asynchronous communication. In *Proceedings of the Second Int.l Conference on Coordination Languages and Models*, pages 238–255, London, UK, 1997. Springer-Verlag.

[4] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996.

[5] N. Busi, R. Gorrieri, and G. Zavattaro. Three semantics of the output operation for generative communication. In *Proceedings of the Second Int.l Conference on Coordination Languages and Models*, pages 205–219, London, UK, 1997. Springer-Verlag.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of linda coordination primitives. *Theor. Comput. Sci.*, 192(2):167–199, 1998.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. On the turing equivalence of linda coordination primitives. *Theor. Comput. Sci.*, 230(1-2):260–261, 2000.

[8] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed linda language and its denotational semantics. *Fundamenta Informaticae*, 63(4), 2004.

[9] F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *Proceedings TAPSOFT/CAAP '91: vol 1*, pages 296–319, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[10] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[11] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[12] E. Horita, J. W. de Bakker, and J. J. M. M. Rutten. Fully abstract denotational models for nonuniform concurrent languages. *Inf. Comput.*, 115(1):125–178, 1994.

[13] IBM. Tspaces.

[14] A. Jeffrey and J. Rathke. Java jr. : Fully abstract trace semantics for a core java language. volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

[15] A. S. A. Jeffrey and J. Rathke. Full abstraction for polymorphic pi-calculus. *Theoretical Computer Science*, 2007. To appear.

[16] B. Jonsson. A model and proof system for asynchronous networks. In *Proceedings of the fourth annual ACM symposium on Principles of distributed computing*, pages 49–58, New York, NY, USA, 1985. ACM Press.

[17] R. D. Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[18] B. C. Pierce and D. Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 242–255, New York, NY, USA, 1997. ACM Press.

[19] G. Wells, P. Clayton, and A. G. Chalmers. A Comparison of Linda Implementations in Java. In P. H. Welch and A. W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 63–76, sep 2000.

[20] G. Zavattaro. Towards a hierarchy of negative test operators for generative communication. *Electr. Notes Theor. Comput. Sci.*, 16(2), 1998.