

On the Expressiveness of Coordination via Shared Dataspaces

Antonio Brogi

Department of Computer Science, University of Pisa, Italy,
brogi@di.unipi.it

Jean-Marie Jacquet

Department of Computer Science, University of Namur, Belgium,
jnj@info.fundp.ac.be

Abstract

A number of different coordination models for specifying inter-process communication and synchronisation rely on a notion of shared dataspace. Many of these models are extensions of the Linda coordination model, which includes operations for adding, deleting and testing the presence/absence of data in a shared dataspace.

We compare the expressive power of three classes of coordination models based on shared dataspaces. The first class relies on Linda's communication primitives, while a second class relies on the more general notion of multi-set rewriting (e.g., like Bauhaus Linda or Gamma). Finally, we consider a third class of models featuring communication transactions that consist of sequences of Linda-like operations to be executed atomically (e.g., like in Shared Prolog or PoliS).

1 Introduction

1.1 Motivations

As motivated by the constant expansion of computer networks and illustrated by the development of distributed applications, the design of modern software systems centers on re-using and integrating software components. The corresponding paradigm shift from stand-alone applications to interacting distributed systems calls for well-defined methodologies and tools for integrating heterogeneous software components.

One of the key issues in this perspective is a clear separation between the *interaction* and the *computation* aspects of software components. Such a separation was advocated by Gelernter and Carriero in [18] as a promising approach to master the complexity of large applications, to enhance software reusability and to ease global analysis. The importance of separating interaction and computation aspects may be also summarised by Wegner’s provocative argument that “interaction is more important than algorithms” [26].

Accordingly, the last decade has seen an increasing attention towards models and languages which support a neat separation of the design of individual software components from their interaction. Such models and languages are often referred to as *coordination models and languages*, respectively [12,17].

Linda [9] was the first coordination language, originally presented as a set of inter-agent communication primitives which may be added to virtually any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other coordination models have been proposed after Linda. Some of them extend Linda in different ways, for instance by introducing multiple dataspaces and meta-level control rules (e.g., Bauhaus Linda [10], Bonita [22], μ Log ([19]), PoliS [11], Shared Prolog [5]), by addressing open distributed systems (e.g., Laura [25]), middleware web-based environments (e.g., Jada [13]), or mobility (e.g., KLAIM [15]). A number of other coordination models rely on a notion of shared dataspace, e.g., Concurrent Constraint Programming [23], Gamma [2], and Linear Objects [1], to cite only a few. A comprehensive survey of these and other coordination models and languages has been recently reported in [21].

The availability of a considerable number of coordination models and languages stimulates a natural question:

Which is the best language or model for expressing coordination issues?

Of course the answer depends on what we mean by the “best” model. A formal way of specifying this question is to reformulate it in terms of the expressive power of models and languages.

1.2 Comparing the expressive power of coordination languages

As pointed out in [14], from a computational point of view all “reasonable” sequential programming languages are equivalent, as they express the same class of functions. Still it is common practice to speak about the “power” of

a language on the basis of the expressibility or non-expressibility of programming constructs. In general, a sequential language L is considered to be more expressive than another sequential language L' if the constructs of L' can be translated in L without requiring a “global reorganisation of the program” [16], that is, in a compositional way. Of course the translation must preserve the meaning, at least in the weak sense of preserving termination.

When considering concurrent languages, the notion of termination must be reconsidered as each possible computation represents a possible different evolution of a system of interacting processes. Moreover *deadlock* represents an additional case of termination. De Boer and Palamidessi introduced in [14] the notion of *modular embedding* as a method to compare the expressive power of concurrent languages.

In this paper we use the notion of modular embedding to compare the relative expressive power of three classes of coordination languages that employ data-driven communication primitives. The first family, denoted by \mathcal{L}_L , is based on a set of communication primitives à la Linda: *tell*, *get*, *ask*, and *nask* for respectively adding, deleting, and checking the presence and the absence of data in a shared dataspace. The second family, denoted by \mathcal{L}_{MR} , adopts an alternative view of these primitives by considering them as the rewriting of pre- and post-conditions on a shared data space, namely as multi-sets of *tell*, *get*, *ask*, and *nask* operations. The third family, denoted by \mathcal{L}_{CS} , imposes an order on the evaluation of the primitives, hence introducing communication sequences to be evaluated atomically as “all-or-nothing” transactions.

All the languages considered contain sequential, parallel and choice operators. For each family (viz., \mathcal{L}_L , \mathcal{L}_{MR} , \mathcal{L}_{CS}) we consider three different languages that differ from one another in the set \mathcal{X} of communication primitives used, syntactically denoted by a set parameter. For instance, if \mathcal{X} is the set $\{ask, tell\}$ then the language $\mathcal{L}_L(\mathcal{X})$ is Linda restricted to *ask* and *tell* operations, and it corresponds to a basic form of concurrent constraint programming [23]. Analogously, $\mathcal{L}_L(ask, get, tell)$ ¹ corresponds to non-monotonic concurrent constraint programming [3], while $\mathcal{L}_L(ask, nask, get, tell)$ corresponds to Linda without process creation. Moreover, $\mathcal{L}_{MR}(ask, get, tell)$ corresponds to Gamma [2], $\mathcal{L}_{MR}(ask, nask, get, tell)$ extends Gamma with negative (non-local) pre-conditions, while $\mathcal{L}_{CS}(ask, nask, get, tell)$ generalises the communication transactions introduced in Shared Prolog [5].

As just suggested, the families \mathcal{L}_L , \mathcal{L}_{MR} , and \mathcal{L}_{CS} are thus representatives of a substantial number of coordination languages. We turn in this paper to an exhaustive pair-wise comparison of the expressive power of the languages obtained by taking \mathcal{X} as $\{ask, tell\}$, $\{ask, get, tell\}$, and $\{ask, nask, get, tell\}$,

¹ Set brackets will be omitted for the ease of reading.

for each of the three classes.

1.3 Results of the comparisons

It is easy to see that a number of (modular) embeddings can be trivially established by considering sub-languages. For instance, for any considered class of languages (viz., for any possible subscript of \mathcal{L}):

$$\mathcal{L}(ask, tell) \leq \mathcal{L}(ask, get, tell) \leq \mathcal{L}(ask, nask, get, tell)$$

holds, where $L' \leq L$ denotes that L' can be (modularly) embedded by L . However, the most interesting results are *separation* results, where a language is shown to be strictly more powerful than another language, and *equivalence* results, where two languages are shown to have the same expressive power.

An expected result proved in the paper is that the above disequalities are strict in the sense that, on the one hand, it is not possible to simulate the destructive *get* primitives via *ask* and *tell* operations and, on the other hand, it is not possible to reduce *nask* tests to *ask*, *get*, and *tell* primitives. Hence for instance concurrent constraint programming languages are strictly less expressive than their non-monotonic versions, which are in turn strictly less expressive than Linda.

Another interesting result is that, for any subset \mathcal{X} of communication primitives, $\mathcal{L}_L(\mathcal{X}) < \mathcal{L}_{MR}(\mathcal{X})$. This establishes that Linda without *nask* operations is strictly less expressive than Gamma. Similarly, for each \mathcal{X} , $\mathcal{L}_L(\mathcal{X}) < \mathcal{L}_{CS}(\mathcal{X})$, which shows that the introduction of communication transactions strictly increases the expressive power of languages. However, $\mathcal{L}_L(ask, nask, get, tell)$ and $\mathcal{L}_{MR}(ask, get, tell)$ are incomparable, which proves that full Linda and Gamma are incomparable.

It is interesting to observe that communication transactions get more and more expressiveness as they are enriched with primitives, as evidenced by the following relations:

$$\begin{aligned} \mathcal{L}_{CS}(ask, tell) &< \mathcal{L}_{MR}(ask, tell) \\ \mathcal{L}_{CS}(ask, get, tell) &= \mathcal{L}_{MR}(ask, get, tell) \\ \mathcal{L}_{CS}(ask, nask, get, tell) &> \mathcal{L}_{MR}(ask, nask, get, tell) \end{aligned}$$

Finally, it is worth observing that $\mathcal{L}_{CS}(ask, nask, get, tell)$ is the most expressive language of the nine languages under study.

Our study of the languages is complete in the sense that all possible relations between pairs of languages have been analysed. For each pair of languages we have established whether they have the same expressive power ($L = L'$), or one is strictly more powerful than the other ($L < L'$), or none of the above two cases holds (i.e., L and L' are incomparable).

This study provides useful insights for both the theory and the practice of coordination-based approaches. Indeed, the resulting hierarchy depicted in Figure 4 shows the equivalence of different models and indicates which extensions may be worth considering because of their additional expressive power.

1.4 Related work

The specificities of our work may be highlighted by contrasting it with related work. The closest pieces of work are [28] and [27].

The expressiveness of four coordination languages is analysed in [28]. Using our terminology, they are obtained by enriching the language $L_0 = \mathcal{L}_L(\text{get}, \text{tell})$ with three forms of negative tests: $\text{nask}(a)$ which tests for the absence of a , $\text{t\&e}(a)$ which instantaneously produces a after having tested that a is not present, and $\text{t\&p}(a, b)$ which atomically tests for the absence of a and produces an instance of b . Consequently, the first extension L_1 is $\mathcal{L}_L(\text{nask}, \text{get}, \text{tell})$, which is proved equivalent in [6] to $\mathcal{L}_L(\text{ask}, \text{nask}, \text{get}, \text{tell})$. The second extension L_2 is a restricted version of the language $\mathcal{L}_{CS}(\text{nask}, \text{get}, \text{tell})$ reduced by considering as communication primitives operations of the form $[\text{get}(t)]$, $[\text{tell}(t)]$, and $[\text{nask}(t); \text{tell}(t)]$, where the $[\dots]$ construct denotes a communication transaction. Finally, the third extension L_3 is obtained by allowing communication transactions of the form $[\text{nask}(t); \text{tell}(u)]$ for possibly different data t and u . In [28] the languages are compared on the basis of three properties: compositionality of the encoding with respect to parallel composition, preservation of divergence and deadlock, and a symmetry condition. It is worth noting that the resulting hierarchy $L_0 < L_1 < L_2 < L_3$ is consistent with our results. Similar properties are used in [27] to establish the incomparability of Linda and Gamma.

Compared to our work, we shall use compositionality of the encoding with respect to sequential composition, choice, and the parallel composition operator. We will use the preservation of termination marks too, and require an element-wise decoding of the set of observables. However, in contrast to [28] and [27], we shall be more liberal with respect to the preservation of termination marks in requiring these preservations on the store resulting from the execution from the empty store of the coded versions of the considered agents and not on the same store. In particular, these ending stores are not required

to be of the form $\sigma \cup \sigma$ (where \cup denotes multi-set union) if this is so for the stores resulting from the agents themselves. Moreover, as the reader may appreciate, this paper presents a wider comparison of a larger class of languages, which requires new proof techniques at the technical level.

The paper [4] compares nine variants of the $\mathcal{L}_L(ask, nask, get, tell)$ language. They are obtained by varying both the nature of the shared data space and its structure. On the one hand, one distributed model and two centralised models, preserving or not the order in which data values are produced, are proposed. On the other hand, a multi-set structure, a set structure, and a list structure of the dataspace are considered. Rephrased in the [14] setting, this amounts to considering different operational semantics. In contrast, we fix an operational semantics and compare different languages on the basis of this semantics. The goals are thus different, and call for completely different treatments and results.

In [8], a process algebraic treatment of a family of Linda-like concurrent languages is presented. A lattice of eight languages is obtained by considering different sets of primitives out of *ask*, *get*, *tell* primitives, cited above, and conditional *ask* and *get* variants. The authors also show that this lattice collapses to a smaller four-points lattice of different bisimulation-based semantics. Again, compared to our work, different semantics are considered whereas we shall stick to one semantics and compare languages on this basis.

Busi, Gorrieri and Zavattaro also recently studied in [7] the issue of Turing-completeness in Linda-like concurrent languages. They define a process algebra containing Linda's communication primitives and compare two possible semantics for the *tell* primitive: an ordered one, with respect to which the execution of *tell* is considered to be finished when the data has reached the dataspace, and an *unordered* one for which *tell* terminates just after having sent the insertion request to the dataspace. The main result presented in [7] is that the process algebra is not Turing-complete under the second interpretation of *tell*, while it is so under the first interpretation. Besides the fact that we tackle in this paper a broader class of languages, including among others the \mathcal{L}_{MR} and \mathcal{L}_{CS} family, the work [7] and ours are somehow orthogonal. While [7] studies the *absolute* expressive power of different variants of Linda-like languages (using Turing-completeness as a yard-stick), we study the *relative* expressive power of different variants of such languages (using modular embedding as a yard-stick and the ordered interpretation of *tell*).

Finally, this paper extends the exhaustive comparison of the languages in \mathcal{L}_L , that was reported in [6].

<u>GENERAL RULE</u>	<u>\mathcal{L}_L RULE</u>
$A ::= C \mid A ; A \mid A \parallel A \mid A + A$	$C ::= tell(t) \mid ask(t) \mid get(t) \mid nask(t)$
<u>\mathcal{L}_{MR} RULES</u>	<u>\mathcal{L}_{CS} RULES</u>
$C ::= (\{M\}, \{M\})$	$C ::= [T]$
$M ::= \lambda \mid +t \mid -t \mid M, M$	$T ::= tell(t) \mid ask(t) \mid get(t) \mid nask(t) \mid T; T$

Fig. 1. Comparative syntax of the languages.

1.5 Plan of the paper

The remainder of the paper is organised as follows. Section 2 formally defines the syntax of the three classes of concurrent languages considered, while Section 3 defines their operational semantics. Section 4 introduces the notion of modular embedding proposed in [14]. Section 5 contains the exhaustive comparison of the expressive power of the languages. The presentation of the propositions (and of the corresponding proof sketches) establishing the results of the comparisons is preceded by an informal analysis of the results from a programming point of view. The results presented in this section are summarised in Figure 4. Finally Section 6 contains a discussion of related work and some concluding remarks.

2 Three families of coordination languages

2.1 Common syntax and rules

We shall consider a set of languages $\mathcal{L}(\mathcal{X})$, parameterised with respect to the set of communication primitives \mathcal{X} . Such a set \mathcal{X} is in turn a subset of a general set of communication primitives, depending on the family under consideration. Assuming this general set, all the languages use sequential, parallel, and choice operators (see “General rule” in Figure 1), whose meaning is defined by the usual rules (S), (P), and (C) in Figure 2.

<u>GENERAL RULES</u>		<u>\mathcal{L}_L RULES</u>	
$(S) \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A ; B \mid \sigma \rangle \longrightarrow \langle A' ; B \mid \sigma' \rangle}$		$(T) \quad \langle tell(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$	
$(P) \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \parallel B \mid \sigma \rangle \longrightarrow \langle A' \parallel B \mid \sigma' \rangle}$ $\langle B \parallel A \mid \sigma \rangle \longrightarrow \langle B \parallel A' \mid \sigma' \rangle$		$(A) \quad \langle ask(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$	
$(C) \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A + B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}$ $\langle B + A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle$		$(N) \quad \frac{t \notin \sigma}{\langle nask(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}$	
$(CM) \frac{\sigma = pre^+ \cup \sigma', \quad pre^- \cap \sigma = \emptyset, \quad \sigma'' = (\sigma - post^-) \cup post^+}{\langle (pre, post) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma'' \rangle}$		$(G) \quad \langle get(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \rangle$	
		<u>\mathcal{L}_{CS} RULE</u>	
		$(CS) \frac{\langle cs \mid \sigma \rangle \longrightarrow^* \langle E \mid \sigma' \rangle}{\langle [cs] \mid \sigma \rangle \longrightarrow \langle E \mid \sigma' \rangle}$	

Fig. 2. Comparative semantics of the languages.

2.2 \mathcal{L}_L : Linda

The first family of languages is the Linda-like family of languages.

Definition 1 Define the set of communication primitives $Slcom$ as the set of C 's generated by the \mathcal{L}_L rule of Figure 1. Moreover, for any subset \mathcal{X} of $Slcom$, define the language $\mathcal{L}_L(\mathcal{X})$ as the set of agents A generated by the general rule of Figure 1. The transition rules for these agents are the general ones of Figure 2 together with rules (T), (A), (N), (G) of that figure, where E denotes the empty agent.

Rule (T) states that an atomic agent $tell(t)$ can be executed in any store σ , and that its execution results in adding the token t to the store σ . Rules (A) and (N) state respectively that the atomic agents $ask(t)$ and $nask(t)$ can be executed in any store containing the token t and not containing t , and that their execution does not modify the current store. Rule (G) also states that an atomic agent $get(t)$ can be executed in any store containing an occurrence of t , but in the resulting store the occurrence of t has been deleted. Note that

the symbol \cup actually denotes multiset union.

Note that to meet the intuition, we shall subsequently always rewrite agents of the form $(E ; A)$, $(E \parallel A)$, and $(A \parallel E)$ as A . This is technically achieved by imposing that, for any language $\mathcal{L}(\mathcal{X})$, the structure $(\mathcal{L}(\mathcal{X}), E, ;, \parallel)$ is a bimonoid.

2.3 \mathcal{L}_{MR} : Multi-set rewriting

The transition rules (T) , (A) , (N) , and (G) suggest an alternative view of Linda-like communication primitives in terms of which conditions the current store should obey to allow the transitions to occur and which modifications these transitions make on the store.

A natural dual view of communication primitives is then to consider them as the rewriting of pre-conditions into post-conditions. We shall consequently examine, as a second family, languages based on multi-set rewriting. It is here worth noting that this approach has already been taken in [2,10,20].

Each communication primitive thus consists of a multi-set of pre-conditions and of a multi-set of post-conditions. Pre- and post-conditions are (possibly empty) multi-sets of positive and negative tuples. Intuitively speaking, the operational effect of a multi-set rewriting $(pre, post)$ is to insert all positive post-conditions and to delete all negative post-conditions from the current dataspace σ , provided that σ contains all positive pre-conditions and does not contain any of the negative pre-conditions. For instance, the operational effect of the multi-set rewriting $(\{+a, -b, +d\}, \{+c, -d\})$ is to add c and delete d from the current dataspace σ provided that σ contains a and d and does not contain b .

Given a multi-set rewriting $(pre, post)$ we shall denote by pre^+ the multi-set $\{t \mid +t \in pre\}$ and by pre^- the multi-set $\{t \mid -t \in pre\}$. The denotations $post^+$ and $post^-$ are defined analogously.

A multi-set rewriting $(pre, post)$ is *consistent* iff $pre^+ \cap pre^- = \emptyset$. A multi-set rewriting $(pre, post)$ is *valid* if $post^- \subseteq pre^+$, where \subseteq denotes multi-set inclusion.

Definition 2 *Define the set of multi-set communication primitives $Smcom$ as the set of C 's engendered by the \mathcal{L}_{MR} rules of Figure 1. Given a subset \mathcal{X} of $Smcom$, define the language $\mathcal{L}_{MR}(\mathcal{X})$ as the set of A 's generated by the general rule of Figure 1.*

As a result of restricting to consistent and valid multi-set communication prim-

itives, four basic pairs of pre and post-conditions are only possible: $(\{+t\}, \{\})$, $(\{-t\}, \{\})$, $(\{\}, \{+t\})$, $(\{+t\}, \{-t\})$. We shall respectively identify them to $ask(t)$, $nask(t)$, $tell(t)$, and $get(t)$.

For our comparison purposes, given \mathcal{X} a subset of communication primitives of $Slcom$, we shall abuse notations and denote by $\mathcal{L}_{MR}(\mathcal{X})$ the language obtained by restricting multi-set rewriting pairs to component-wise multi-set unions of pairs associated with the communication primitives of \mathcal{X} . For instance, if $\mathcal{X} = \{ask, nask\}$, then the language $\mathcal{L}_{MR}(\mathcal{X})$ only involves pairs of the form $(Pre, \{\})$ where Pre may contain positive and negative tokens. Similarly, if $\mathcal{X} = \{tell, get\}$ then $\mathcal{L}_{MR}(\mathcal{X})$ includes only pairs of the form $(Pre, Post)$ where Pre contain positive tokens only provided that each one is associated with one negative counterpart in $Post$ and $Post$ contain negative tokens provided each one is associated to one positive token in Pre as well as positive tokens (without restriction). Note that these notations fully agree with the one introduced in Definition 2.

Definition 3 *Define the transition rules for the \mathcal{L}_{MR} family of languages as the general rules of Figure 2 together with rule (CM) of that figure.*

Rule (CM) states that a multi-set rewriting $(pre, post)$ can be executed in a store σ if the multi-set pre^+ is included in σ and if no negative pre-condition occurs in σ . If such conditions hold then the execution of the rewriting deletes from σ all the negative post-conditions, and adds to σ all the positive post-conditions.

2.4 \mathcal{L}_{CS} : Communication transactions

A natural further refinement is to impose an order on the test of pre-conditions and the evaluation of post-conditions, possibly mixing pre- and post-conditions. We are thus led to sequences of elementary actions, which we will take, for clarity purposes, in the Linda form instead of the $+t$ and $-t$ of the \mathcal{L}_{MR} family. These sequences will be called *communication transactions*, with the intuition that they are to be executed as single “all-or-nothing” transactions. They have been employed in Shared-Prolog ([5]) and in PoliS [11]).

Definition 4 *Define the set of communication transactions $Stcom$ as the set of C 's engendered by the \mathcal{L}_{CS} rules of Figure 1. Moreover, for any subset \mathcal{X} of $Stcom$, define the language $\mathcal{L}_L(\mathcal{X})$ as the set of agents A generated by the general rule of Figure 1. The transition rules for these agents are the general ones of Figure 2 together with rule (CS).*

3 Operational semantics

3.1 Observables

Definition 5

- (1) Let $Stoken$ be a denumerable set, the elements of which are subsequently called tokens and are typically represented by the letters t and u . Define the set of stores $Sstore$ as the set of finite multisets with elements from $Stoken$.
- (2) Let δ^+ and δ^- be two fresh symbols denoting respectively success and failure. Define the set of histories $Shist$ as the set $Sstore \times \{\delta^+, \delta^-\}$.
- (3) Define the operational semantics $\mathcal{O} : \mathcal{L}(\mathcal{X}) \rightarrow \mathcal{P}(Shist)$ as the following function: For any agent $A \in \mathcal{L}(\mathcal{X})$

$$\begin{aligned} \mathcal{O}(A) = & \{(\sigma, \delta^+) : \langle A \mid \emptyset \rangle \rightarrow^* \langle E \mid \sigma \rangle\} \\ & \cup \\ & \{(\sigma, \delta^-) : \langle A \mid \emptyset \rangle \rightarrow^* \langle B \mid \sigma \rangle \not\rightarrow, B \neq E\} \end{aligned}$$

- (4) Define, for any agent $A \in \mathcal{L}(\mathcal{X})$, a derivation sequence as a complete finite sequence of computation steps induced by the transition system and starting from the empty multiset of tokens:

$$\langle A \mid \emptyset \rangle \rightarrow \cdots \rightarrow \langle T \mid \sigma \rangle$$

with $T = E$ or $\langle T \mid \sigma \rangle \not\rightarrow$.

3.2 Normal form

A classical result of concurrency theory is that modelling parallel composition by interleaving, as we do, allows agents to be considered in a normal form. We first define what we actually mean, and then state the proposition that agents and their normal forms are equivalent in the sense that they yield the same computations.

Definition 6 Given a subset \mathcal{X} of $Slcom$, $Smcom$, or $Stcom$, the set $Snagent$ of agents in normal form is defined by the following rule, where N is an agent in normal form, c denotes a communication action of \mathcal{X} , and A denotes an agent of $\mathcal{L}(\mathcal{X})$:

$$N ::= c \mid c ; A \mid N + N.$$

Proposition 7 *For any agent A , there is an agent N in normal form which has the same derivation sequences as A .*

PROOF. Indeed, it is possible to associate to any agent A an agent $\tau(A)$ in normal form by using the following translation defined inductively on the structure of A :

$$\begin{aligned}
\tau(c) &= c \\
\tau(X;Y) &= \tau(X);Y \\
\tau(X + Y) &= \tau(X) + \tau(Y) \\
\tau(X \parallel Y) &= \tau(X) \parallel \tau(Y) \\
c \parallel Z &= c; Z \\
(c; A) \parallel Z &= c; (A \parallel Z) \\
(N_1 + N_2) \parallel Z &= N_1 \parallel Z + N_2 \parallel Z
\end{aligned}$$

It is easy to verify that, for any agent A , the agent $\tau(A)$ is in normal form. Moreover, it is straightforward to verify that A and $\tau(A)$ share the same derivation sequences, namely that if $\langle A \mid \emptyset \rangle \rightarrow \langle B \mid \sigma \rangle \rightarrow \dots \rightarrow \langle T \mid \phi \rangle$ is a derivation sequence for A , then $\langle \tau(A) \mid \emptyset \rangle \rightarrow \langle B \mid \sigma \rangle \rightarrow \dots \rightarrow \langle T \mid \phi \rangle$ is a derivation sequence for $\tau(A)$ and vice-versa.

4 Modular embedding

A natural way to compare the expressive power of two languages is to see whether all programs written in one language can be “easily” and “equivalently” translated into the other language, where equivalent is intended in the sense of the same observable behaviour.

The basic definition of embedding, given by Shapiro [24] is the following. Consider two languages L and L' . Assume given the semantics mappings (*observation criteria*) $\mathcal{S} : L \rightarrow \mathcal{O}$ and $\mathcal{S}' : L' \rightarrow \mathcal{O}'$, where \mathcal{O} and \mathcal{O}' are some suitable domains. Then L can *embed* L' if there exists a mapping \mathcal{C} (*coder*) from the statements of L' to the statements of L , and a mapping \mathcal{D} (*decoder*) from \mathcal{O} to \mathcal{O}' , such that the diagram of Figure 3 commutes, namely such that for every statement $A \in L'$: $\mathcal{D}(\mathcal{S}(\mathcal{C}(A))) = \mathcal{S}'(A)$.

The basic notion of embedding is too weak since, for instance, the above equation is satisfied by any pair of Turing-complete languages. De Boer and Palamidessi hence proposed in [14] to add three constraints on the coder \mathcal{C}

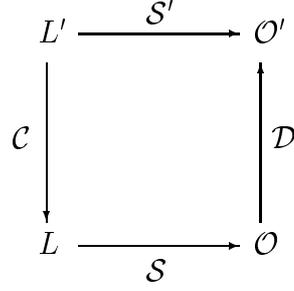


Fig. 3. Basic embedding.

and on the decoder \mathcal{D} in order to obtain a notion of *modular* embedding usable for concurrent languages:

- (1) \mathcal{D} should be defined in an element-wise way with respect to \mathcal{O} :

$$\forall X \in \mathcal{O} : \mathcal{D}(X) = \{\mathcal{D}_{el}(x) \mid x \in X\} \quad (P_1)$$

for some appropriate mapping \mathcal{D}_{el} ;

- (2) the coder \mathcal{C} should be defined in a compositional way with respect to the sequential, parallel and choice operators²:

$$\mathcal{C}(A ; B) = \mathcal{C}(A) ; \mathcal{C}(B)$$

$$\mathcal{C}(A \parallel B) = \mathcal{C}(A) \parallel \mathcal{C}(B) \quad (P_2)$$

$$\mathcal{C}(A + B) = \mathcal{C}(A) + \mathcal{C}(B)$$

- (3) the embedding should preserve the behaviour of the original processes with respect to deadlock, failure and success (*termination invariance*):

$$\forall X \in \mathcal{O}, \forall x \in X : tm'(\mathcal{D}_{el}(x)) = tm(x) \quad (P_3)$$

where tm and tm' extract the information on termination from the observables of L and L' , respectively.

An embedding is then called *modular* if it satisfies properties P_1 , P_2 , and P_3 .

The existence of a modular embedding from L' into L will be denoted by $L' \leq L$. It is easy to see that \leq is a pre-order relation. Moreover if $L' \subseteq L$ then $L' \leq L$ that is, any language embeds all its sublanguages. This property descends immediately from the definition of embedding, by setting \mathcal{C} and \mathcal{D} equal to the identity function.

² Actually, this is not required for the sequential operator in [14] since it does not occur in that work.

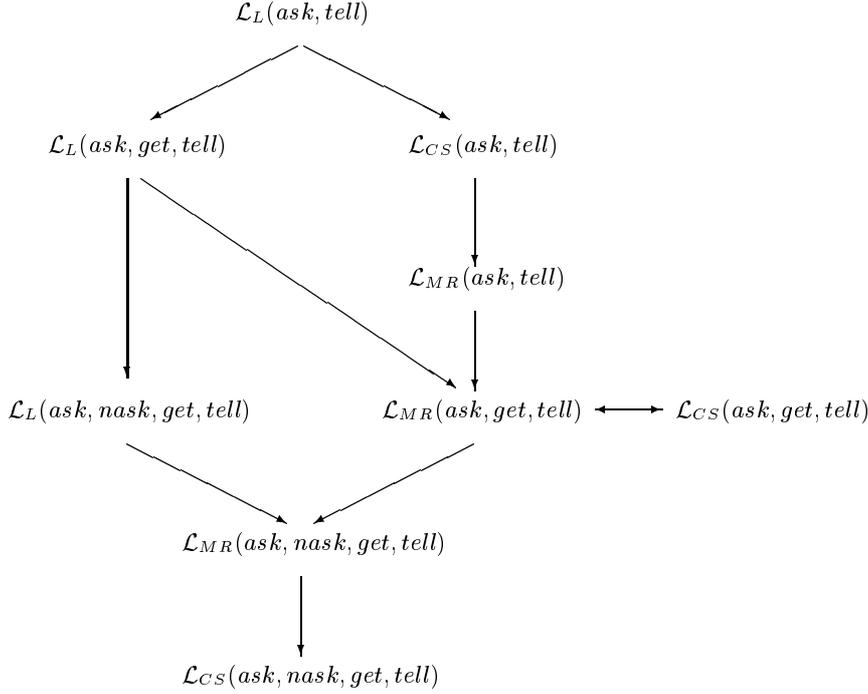


Fig. 4. Main results of the comparisons.

5 Comparisons

We now turn to an exhaustive comparison of the relative expressive power of the languages introduced in Section 2.

We will consider nine different languages which are obtained by considering three different sets of communication primitives, namely $\mathcal{X} = \{ask, tell\}$, $\mathcal{X} = \{ask, get, tell\}$, and $\mathcal{X} = \{ask, nask, get, tell\}$, for each of the three parameterised languages $\mathcal{L}_L(\mathcal{X})$, $\mathcal{L}_{CS}(\mathcal{X})$, and $\mathcal{L}_{MR}(\mathcal{X})$.

The whole set of separation and equivalence results are summarised in Figure 4, where an arrow from a language \mathcal{L}_1 to a language \mathcal{L}_2 means that \mathcal{L}_2 embeds \mathcal{L}_1 , that is $\mathcal{L}_1 \leq \mathcal{L}_2$. Notice that, thanks to the transitivity of embedding, the figure contains only a minimal amount of arrows. However, apart from these induced relations, no other relation holds. In particular, when there is one arrow from \mathcal{L}_1 to \mathcal{L}_2 but there is no arrow from \mathcal{L}_2 to \mathcal{L}_1 , then \mathcal{L}_1 is strictly less expressive than \mathcal{L}_2 , that is $\mathcal{L}_1 < \mathcal{L}_2$.

The separation and equivalence results are presented in two steps. Section 5.1 first presents the intuition for these results whereas their formal proofs are given in Section 5.2.

5.1 Intuitive analysis of the results

Before presenting the proofs of the results illustrated in Figure 4, we will try to analyse their intuitive meaning. More precisely, we shall try here to show informally how such formal separation results confirm the intuitive expectations from a programming viewpoint. Of course the intuitive explanation of a separation (or equivalence) result does not formally prove the validity of the result itself. One may indeed argue that even if there is no obvious encoding between the two languages, there may well be a non-trivial encoding that may yield the embedding. The nonexistence of such embeddings will be formally established by the propositions proved in Section 5.2.

5.1.1 Analysis for $\mathcal{X} = \{ask, tell\}$

Let us first consider the case in which $\mathcal{X} = \{ask, tell\}$. It is easy to see that $\mathcal{L}_L(ask, tell)$ does not support a straightforward way of *atomically* testing the simultaneous presence of two resources a and b in the dataspace. Indeed the obvious coding $(ask(a); ask(b))$ will not be executed atomically and may not produce the desired behaviour for instance in: $(ask(a); ask(b); P) + (ask(a); ask(c); Q)$. The language $\mathcal{L}_{CS}(ask, tell)$ instead supports a straightforward way of atomically testing the presence of two resources in the dataspace, via the communication transaction $[ask(a); ask(b)]$, thus intuitively confirming the separation result $\mathcal{L}_L(ask, tell) < \mathcal{L}_{CS}(ask, tell)$.

It is easy to observe that the same kind of test can be naturally expressed also in $\mathcal{L}_{MR}(ask, tell)$ via the rewriting $(\{+a, +b\}, \{\dots\})$. Moreover the language $\mathcal{L}_{MR}(ask, tell)$ permits to express also tests of the form “if there are at least n copies of a resource a then”. For instance the rewriting $(\{+a, +a\}, \{+b\})$ states that if there are at least two copies of resource a then resource b will be added to the dataspace. The same test cannot be easily expressed in $\mathcal{L}_{CS}(ask, tell)$ with $[ask(a); ask(a); tell(b)]$, since the two ask operations may match the same instance of a in the dataspace. The inability of $\mathcal{L}_{CS}(ask, tell)$ to atomically test the presence of multiple copies of the same resource confirms intuitively the separation result $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{MR}(ask, tell)$.

5.1.2 Analysis for $\mathcal{X} = \{ask, get, tell\}$

The addition of the *get* primitive to the set \mathcal{X} gives to each of the former three languages the ability of deleting tuples, hence yielding a non-monotonic evolution of the dataspace. The three separation results $\mathcal{L}_L(ask, tell) < \mathcal{L}_L(ask, get, tell)$, $\mathcal{L}_{MR}(ask, tell) < \mathcal{L}_{MR}(ask, get, tell)$, and $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{CS}(ask, get, tell)$ follow such intuition.

The separation result between the basic Linda calculus and the multi-set rewriting calculus continues to hold also after introducing the *get* operation, that is, $\mathcal{L}_L(ask, get, tell) < \mathcal{L}_{MR}(ask, get, tell)$. Indeed the addition of *get* still does not allow $\mathcal{L}_L(ask, get, tell)$ to atomically test the simultaneous presence of two resources *a* and *b* in the dataspace.

On the other hand, the introduction of *get* removes the gap between communication sequences and multi-set rewriting, which have in this case the same expressive power, that is, $\mathcal{L}_{MR}(ask, get, tell) = \mathcal{L}_{CS}(ask, get, tell)$. For instance $\mathcal{L}_{CS}(ask, get, tell)$ can now express tests of the form “if there are at least two copies of resource *a* then” via the transaction $[get(a); ask(a); tell(a)]$.

5.1.3 Analysis for $\mathcal{X} = \{ask, nask, get, tell\}$

The introduction of the *nask* primitive into the set \mathcal{X} gives to each language the ability of testing the *absence* of data from the dataspace, and hence to express *if-then-else* conditions of the form “if resource *a* belongs to the dataspace then do *P* else do *Q*”. For instance such a test can be expressed in $\mathcal{L}_L(ask, nask, get, tell)$ as $(ask(a); P) + (nask(a); Q)$. The additional expressive power given by *nask* intuitively explains the separations result $\mathcal{L}(ask, get, tell) < \mathcal{L}(ask, nask, get, tell)$, which holds for \mathcal{L} being either \mathcal{L}_L , \mathcal{L}_{MR} or \mathcal{L}_{CS} .

Even after introducing *nask*, the basic Linda calculus is less expressive than both communication transactions and multi-set rewriting. Indeed $\mathcal{L}_L(ask, nask, get, tell)$ is still not able to atomically test the simultaneous presence of two resources *a* and *b* in the dataspace.

The introduction of negative tests instead reverses the relation between $\mathcal{L}_{MR}(ask, nask, get, tell)$ and $\mathcal{L}_{CS}(ask, nask, get, tell)$. Indeed the availability of *nask* allows $\mathcal{L}_{CS}(ask, nask, get, tell)$ to “count” the number of copies of a resource available in the dataspace. For instance $\mathcal{L}_{CS}(ask, nask, get, tell)$ can express tests of the form “if there are *exactly* two copies of a resource *a* then do *P*” via the communication sequence $[get(a); get(a); nask(a); tell(a); tell(a)]; P$ while $\mathcal{L}_{MR}(ask, nask, get, tell)$ can only express test of the form “if there *at least* *n* copies of resource then”. This intuitively explains the last separation result $\mathcal{L}_{MR}(ask, nask, get, tell) < \mathcal{L}_{CS}(ask, nask, get, tell)$.

5.2 Formal propositions and proofs

5.2.1 Basic results

The following propositions and their proofs give an insight on how to proceed to compare the families of languages.

Proposition 8 For any set of communication primitives \mathcal{X} , $\mathcal{L}_L(\mathcal{X}) \leq \mathcal{L}_{MR}(\mathcal{X})$.

PROOF. Immediate by defining the coder as follows:

$$\begin{aligned} \mathcal{C}(\text{tell}(t)) &= (\{\}, \{+t\}) & \mathcal{C}(\text{get}(t)) &= (\{+t\}, \{-t\}) \\ \mathcal{C}(\text{ask}(t)) &= (\{+t\}, \{\}) & \mathcal{C}(\text{nask}(t)) &= (\{-t\}, \{\}) \end{aligned}$$

□

Proposition 9

- i) $\mathcal{L}_{MR}(\text{tell}, \text{ask}, \text{get}) \leq \mathcal{L}_{CS}(\text{tell}, \text{ask}, \text{get})$
- ii) $\mathcal{L}_{MR}(\text{tell}, \text{ask}, \text{nask}, \text{get}) \leq \mathcal{L}_{CS}(\text{tell}, \text{ask}, \text{nask}, \text{get})$.

PROOF. Indeed, the non-redundancy of multiple ask queries in the \mathcal{L}_{MR} family of languages can be taken into account by first getting the tokens and then telling them back. Consequently, it is sufficient to code

$$(\{+g_1, \dots, +g_p, +a_1, \dots, +a_q, -n_1, \dots, -n_r\}, \{+t_1, \dots, +t_s, -g_1, \dots, -g_p\})$$

into

$$\left[\begin{array}{l} \text{get}(g_1), \dots, \text{get}(g_p), \text{get}(a_1), \dots, \text{get}(a_q), \text{tell}(a_1), \dots, \text{tell}(a_q), \\ \text{nask}(n_1), \dots, \text{nask}(n_r), \text{tell}(t_1), \dots, \text{tell}(t_s) \end{array} \right]$$

□

In contrast however, $\mathcal{L}_{MR}(\text{ask}, \text{tell})$ cannot be embedded in $\mathcal{L}_{CS}(\text{ask}, \text{tell})$.

Proposition 10 $\mathcal{L}_{MR}(\text{tell}, \text{ask}) \not\leq \mathcal{L}_{CS}(\text{tell}, \text{ask})$.

PROOF. By contradiction, assume that there is a coder \mathcal{C} . Obviously, for any token t , the computation of $(\{\}, \{+t\})$ succeeds and so should that of

$\mathcal{C}(\{\{\}, \{+t\}\})$ by P_3 . Let us call σ the state resulting from one computation. As $\mathcal{L}_{CS}(tell, ask)$ contains no destructive operations and no negative tests, $\mathcal{C}(\{\{\}, \{+t\}\}) ; \mathcal{C}(\{\{\}, \{+t\}\})$ has a successful computation resulting in the store $\sigma \cup \sigma$. Now consider $\mathcal{C}(\{\{+t, +t\}, \{\}\})$ in its normal form: $a_1 ; A_1 + \dots + a_m ; A_m$. Since $(\{\{\}, \{+t\}\}) ; (\{\{\}, \{+t\}\}) ; (\{\{+t, +t\}, \{\}\})$ succeeds, by (P_3) , there should exist $i \in \{1, \dots, m\}$ such that $\langle \mathcal{C}(\{\{+t, +t\}, \{\}\}) \mid \sigma \cup \sigma \rangle \longrightarrow \langle A_i \mid \sigma \cup \sigma \cup \tau \rangle$, for some store τ , Moreover A_i computed from $\sigma \cup \sigma \cup \tau$ should only lead to success and thus, as $\mathcal{L}_{CS}(tell, ask)$ does not contain any destructive operation, A_i started on $\sigma \cup \tau$ has only successful computations. It follows that

$$\begin{aligned} \langle \mathcal{C}(\{\{\}, \{+t\}\}) ; \mathcal{C}(\{\{+t, +t\}, \{\}\}) \mid \emptyset \rangle &\longrightarrow^* \langle \mathcal{C}(\{\{+t, +t\}, \{\}\}) \mid \sigma \rangle \\ &\longrightarrow \langle A_i \mid \sigma \cup \tau \rangle \end{aligned}$$

is a valid computation prefix for $\mathcal{C}(\{\{\}, \{+t\}\}) ; (\{\{+t, +t\}, \{\}\})$ which can only be continued by successful computations. This contradicts by P_3 the fact that $(\{\{\}, \{+t\}\}) ; (\{\{+t, +t\}, \{\}\})$ has only one failing computation. \square

5.2.2 Embedding the \mathcal{L}_{MR} family into \mathcal{L}_L

As the \mathcal{L}_L family of languages can all be embedded in the corresponding languages of the \mathcal{L}_{MR} family of languages, the natural next properties to investigate are whether the converse holds. This is not true as established by the following propositions.

Proposition 11 $\mathcal{L}_{MR}(tell, ask) \not\subseteq \mathcal{L}_L(tell, ask)$

PROOF. Let us proceed by contradiction and assume the existence of a coder \mathcal{C} and a decoder \mathcal{D} . Let a, b be two distinct tokens. Since $\mathcal{O}(\{\{\}, \{+a\}\}) = \{\{\{a\}, \delta^+\}\}$ any computation of $\mathcal{C}(\{\{\}, \{+a\}\})$ starting in the empty store succeeds by property (P_3) . Let

$$\langle \mathcal{C}(\{\{\}, \{+a\}\}) \mid \emptyset \rangle \longrightarrow \dots \longrightarrow \langle E \mid \{a_1, \dots, a_m\} \rangle$$

be one computation of $\mathcal{C}(\{\{\}, \{+a\}\})$. Similarly, any computation of $\mathcal{C}(\{\{\}, \{+b\}\})$ starting on the empty store succeeds. Let

$$\langle \mathcal{C}(\{\{\}, \{+b\}\}) \mid \emptyset \rangle \longrightarrow \dots \longrightarrow \langle E \mid \{b_1, \dots, b_n\} \rangle$$

be one computation of $\mathcal{C}(\{\{\}, \{+b\}\})$.

Consider now $AB = (\{+a, +b\}, \{\})$. As it is in $\mathcal{L}_L(\text{tell}, \text{ask})$, $\mathcal{C}(AB)$ can be regarded in its normal form, which, in its more general form, is of the form

$$\text{tell}(t_1) ; A_1 + \cdots + \text{tell}(t_p) ; A_p + \text{ask}(u_1) ; B_1 + \cdots + \text{ask}(u_q) ; B_q$$

Let us first establish that there is no alternative guarded by a $\text{tell}(t_i)$ operation. Indeed, if this was the case, then

$$D = \langle \mathcal{C}(AB) \mid \emptyset \rangle \longrightarrow \langle A_i \mid \{t_i\} \rangle$$

would be a valid computation prefix of $\mathcal{C}(AB)$ which should deadlock afterwards since $\mathcal{O}(AB) = \{(\emptyset, \delta^-)\}$. However D is also a valid computation prefix of $\mathcal{C}(AB + (\{\}, \{+a\}))$. Hence, $\mathcal{C}(AB + (\{\}, \{+a\}))$ admits a failing computation which contradicts property P_3 and the fact that $\mathcal{O}(AB + (\{\}, \{+a\})) = \{(\{a\}, \delta^+)\}$.

Let us now establish that none of the u_i 's belong to $\{a_1, \dots, a_m\} \cup \{b_1, \dots, b_n\}$. Indeed, if $u_j \in \{a_1, \dots, a_m\}$ for some $j \in \{1, \dots, q\}$, then

$$\begin{aligned} D' = \langle \mathcal{C}(\{\}, \{+a\}) ; AB \mid \emptyset \rangle &\longrightarrow \cdots \longrightarrow \langle AB \mid \{a_1, \dots, a_m\} \rangle \\ &\longrightarrow \langle B_j \mid \{a_1, \dots, a_m\} \rangle \end{aligned}$$

is a valid computation prefix of $\mathcal{C}(\{\}, \{+a\}) ; AB$ which can only be continued by failing suffixes. However, D' induces the following computation prefix D'' for $(\{\}, \{+a\}) ; (AB + (\{+a\}, \{\}))$ which as just seen admits only successful computations:

$$\begin{aligned} D'' = \langle \mathcal{C}(\{\}, \{+a\}) ; (AB + (\{+a\}, \{\})) \mid \emptyset \rangle &\longrightarrow \cdots \\ &\longrightarrow \langle AB + (\{+a\}, \{\}) \mid \{a_1, \dots, a_m\} \rangle \\ &\longrightarrow \langle B_j \mid \{a_1, \dots, a_m\} \rangle. \end{aligned}$$

The proof proceeds similarly in the case $u_j \in \{b_1, \dots, b_n\}$ for some $j \in \{1, \dots, q\}$ by then considering $(\{\}, \{+b\}) ; AB$ and $(\{\}, \{+b\}) ; (AB + (\{+b\}, \{\}))$.

The u_i 's are thus forced not to belong to $\{a_1, \dots, a_m\} \cup \{b_1, \dots, b_n\}$. However, this induces a contradiction as well. Indeed, if this is the case then

$$\begin{aligned} & \langle \mathcal{C}(\{\}, \{+a\}) ; (\{\}, \{+b\}) ; AB \mid \emptyset \rangle \\ & \quad \longrightarrow \dots \longrightarrow \langle (\{\}, \{+b\}) ; AB \mid \{a_1, \dots, a_m\} \rangle \\ & \quad \longrightarrow \dots \longrightarrow \langle AB \mid \{a_1, \dots, a_m, b_1, \dots, b_n\} \rangle \not\rightarrow \end{aligned}$$

is a valid failing computation of $\mathcal{C}(\{\}, \{+a\}) ; (\{\}, \{+b\}) ; AB$. However, $(\{\}, \{+a\}) ; (\{\}, \{+b\}) ; AB$ has only one successful computation. \square

Proposition 12 $\mathcal{L}_{MR}(\text{tell}, \text{ask}, \text{get}) \not\subseteq \mathcal{L}_L(\text{tell}, \text{ask}, \text{get})$

PROOF. The proof proceeds as for Proposition 11 the only difference being that the normal form of $\mathcal{C}(AB)$ may contain get primitives. Technically this amounts to considering some of the $\text{ask}(u_i)$'s to be $\text{get}(u_i)$ but does not affect more the proof. \square

Let us now introduce the nask primitive in the languages. At first sight, one may think of transposing the proofs given for the previous proposition. However, since $\mathcal{L}_L(\text{ask}, \text{nask}, \text{get}, \text{tell})$ can perform negative tests, $\mathcal{C}(\{\}, \{+b\})$ could in principle check whether the $\mathcal{C}(\{\}, \{+a\})$ has taken place before and then place or remove suitable tokens thereby making clear to $\mathcal{C}(AB)$ that both a and b have been told or not. The same reasoning would hold for telling the same token several times. However, as all the agents of the languages are finitely branching and contain a finite number of communication primitives, this is not possible for ever, as stated in the following lemmas.

Lemma 13 *For any agent $A \in \mathcal{L}_L(\text{tell}, \text{ask}, \text{get}, \text{nask})$, if $(\sigma, \delta^+) \in \mathcal{O}(A)$ then the parallel composition $B = A \parallel \dots \parallel A$ of n copies of A has a successful computation resulting in the store $\tau = \sigma \cup \dots \cup \sigma$ consisting of the multi-set union of n copies of σ : $(\tau, \delta^+) \in \mathcal{O}(A)$.*

PROOF. The lemma is proved by induction on the length of the computations and by establishing by induction on the size of the agent A that: if $\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle$ then $\langle A \parallel \dots \parallel A \mid \sigma \cup \dots \cup \sigma \rangle \longrightarrow \langle A' \parallel \dots \parallel A' \mid \sigma' \cup \dots \cup \sigma' \rangle$, with the \dots dots indicating n copies. \square

Lemma 14 *Let $A \in \mathcal{L}_L(\text{tell}, \text{get}, \text{ask}, \text{nask})$ be such that $(\sigma, \delta^+) \in \mathcal{O}(A)$. Let B be an agent of $\mathcal{L}_L(\text{tell}, \text{get}, \text{nask}, \text{ask})$ composed of n ask, nask and get primitives and such that $\langle B \mid \bigcup_{i=1}^{n+1} \sigma \rangle \longrightarrow^* \langle E \mid \tau \rangle$. Then, for any $p > 0$ $((\bigcup_{i=1}^p \sigma) \cup \tau, \delta^+) \in \mathcal{O}(\bigparallel_{i=1}^{n+1+p} A ; B)$.*

PROOF. Indeed, by Lemma 13, $\langle \parallel_{i=1}^{n+1+p} A \mid \emptyset \rangle \longrightarrow^* \langle E \mid \cup_{i=1}^{n+1+p} \sigma \rangle$. As B is composed of n ask, get, and nask primitives and since $\langle B \mid \cup_{i=1}^{n+1} \sigma \rangle \longrightarrow^* \langle E \mid \tau \rangle$, it is easy to verify by induction on the length of the computations that adding p copies of σ does not alter the computations and moreover is unchanged by them. \square

Lemma 15 *Let S be a finite set of tokens and $f : \text{Stoken} \rightarrow \mathcal{P}_f(\text{Stoken})$ be a function associating to each token a finite set of tokens. Assume there is a token a such that for any other token b either $f(a) \cap f(b) \neq \emptyset$ or $f(b) \cap S \neq \emptyset$. Then there is a denumerable series of token $(x_i)_i$ and an integer N such that $\bigcap_{i=1}^N f(x_i) \neq \emptyset$ and $\bigcap_{i=1}^N f(x_i) = \bigcap_{i=1}^N f(x_i) \cap f(x_n)$, for any $n > N$.*

PROOF. Let us first note that, under the hypothesis of the proposition, there is a token y and a denumerable set of tokens x such that $f(y) \cap f(x) \neq \emptyset$. Indeed, either a plays this role or, there is an infinite number of tokens z such that $S \cap f(z) \neq \emptyset$. Since S is finite, there is thus a token $s \in S$ and a denumerable set of tokens z_i such that $s \in f(z_i)$. Taking one of them as y then establishes the claim.

Let $f(y) = \{a_1, \dots, a_m\}$. Since $f(y) \cap f(z_i) \neq \emptyset$ for any z_i and since $f(y)$ is finite, some of the a_i 's appear infinitely often in the sets $f(z_j)$. For any of them, let $z_{\nu_j}^i$ be the subseries of $(z_j)_j$ in which a_i appear. Take $(x_k)_k = (z_{\mu_k}^k)_k$ as the subseries common to all these subseries. It satisfies the thesis. \square

Proposition 16 $\mathcal{L}_{MR}(\text{ask}, \text{nask}, \text{get}, \text{tell}) \not\subseteq \mathcal{L}_L(\text{ask}, \text{nask}, \text{get}, \text{tell})$

PROOF. Fix a token a . Let n be the number of get and nask primitives in $\mathcal{C}(\{\{\}, \{+a\}\})$. Let for any communication primitive c and any integer p , c^p denote the parallel composition of p copies of c : $c^p = \parallel_{i=1}^p c$.

As the computation of $(\{\}, \{+b\})^{n+2}$; $(\{\}, \{+a\})$ succeeds, for any token $b \neq a$, let S'_b be one store of one successful derivation of $\mathcal{C}(\{\{\}, \{+b\}\})^{n+2}$; $\mathcal{C}(\{\{\}, \{+a\}\})$. Moreover, take as S_x the store resulting from one successful derivation of $\mathcal{C}(\{\{\}, \{+x\}\})$. Note that, thanks to Lemma 14, one may assume that $S_x \subseteq S'_x$, for any token x .

Two cases need to be considered.

Case I. Assume first there is a token b such that $S_a \cap S'_b = \emptyset$. Then consider ABs consisting of removing a together with $n + 3$ copies of b : $ABs = (\{+a, +b, \dots, +b\}, \{-a, -b, \dots, -b\})$. The normal form of $\mathcal{C}(ABs)$

can be rewritten in $\mathcal{L}_{MR}(tell, get, nask)$ as:

$$\begin{aligned}
& tell(t_1) ; A_1 + \cdots + tell(t_p) ; A_p \\
& + ask(u_1) ; B_1 + \cdots + ask(u_q) ; B_q \\
& + get(v_1) ; C_1 + \cdots + get(v_r) ; C_r \\
& + nask(w_1) ; D_1 + \cdots + nask(w_s) ; D_s
\end{aligned}$$

Using arguments analogous to those of Proposition 11, it is possible to prove that there are no alternatives guarded by a $tell(t_i)$ primitive.

Moreover, let us now establish that $\{w_1, \dots, w_s\} \subseteq S_a \cap S'_b$. which, in view of the hypothesis on a and b , amounts to expressing that there are no alternatives guarded by a $nask(w_j)$ primitive. Indeed, assume that $w_j \notin S_a$, for some $j \in \{1, \dots, s\}$. Then

$$E = \langle \mathcal{C}(\{\}, \{+a\}) ; ABs \mid \emptyset \rangle \longrightarrow \cdots \longrightarrow \langle ABs \mid S_a \rangle \longrightarrow \langle D_j \mid S_a \rangle$$

is a valid computation prefix for $\mathcal{C}(\{\}, \{+a\}) ; ABs$ which can only be continued by failing suffixes since $\mathcal{O}(\{\}, \{+a\}) ; ABs = \{(\{a\}, \delta^-)\}$. However, E' induces the following computation prefix E'' for $(\{\}, \{+a\}) ; (ABs + (\{\}, \{+b\}))$ which only admits successful computations:

$$\begin{aligned}
E'' &= \langle \mathcal{C}(\{\}, \{+a\}) ; (ABs + (\{\}, \{+b\})) \mid \emptyset \rangle \\
&\longrightarrow \cdots \\
&\longrightarrow \langle ABs + (\{\}, \{+b\}) \mid S_a \rangle \\
&\longrightarrow \langle D_j \mid S_a \rangle.
\end{aligned}$$

The proof proceeds similarly for $w_j \notin S'_b$ since then $w_j \notin S_b$.

Finally, using the same reasoning as in Proposition 11, it is possible to establish that $\{u_1, \dots, u_q, v_1, \dots, v_r\} \cap (S_b \cup S'_b) = \emptyset$. Therefore $\langle ABs \mid S_b \cup S'_b \rangle \not\rightarrow$. The contradiction then comes from Lemma 14 which validates the following derivation: $\langle \mathcal{C}(\{\}, \{+b\})^{n+3} ; \mathcal{C}(\{\}, \{+a\}) ; ABs \mid \emptyset \rangle \xrightarrow{*} \langle ABs \mid S_b \cup S'_b \rangle$. The agent $\mathcal{C}(\{\}, \{+b\})^{n+3} ; \mathcal{C}(\{\}, \{+a\}) ; ABs$ then admits a failing computation whereas $\mathcal{C}(\{\}, \{+b\})^{n+3} ; \mathcal{C}(\{\}, \{+a\}) ; ABs$ only has one successful computation.

Case II. Assume now that $S_a \cap S'_b \neq \emptyset$ for any token b distinct from a . Then, thanks to Lemma 15, there is a denumerable set of distinct tokens x_i , also distinct from a and an integer m , such that $\cap_{i=1}^m (S_a \cap S'_{x_i}) \neq \emptyset$ and $[\cap_{i=1}^m (S_a \cap$

$S'_{x_i}] \cap (S_a \cap S'_{x_j}) \neq \emptyset$, for $j > m$. Consider $NT = (\{-a, -x_1, \dots, -x_n\}, \{\})$ and $\mathcal{C}(NT)$ in the following normal form.

$$\begin{aligned} & tell(t_1) ; A_1 + \dots + tell(t_p) ; A_p \\ & + ask(u_1) ; B_1 + \dots + ask(u_q) ; B_q \\ & + get(v_1) ; C_1 + \dots + get(v_r) ; C_r \\ & + nask(w_1) ; D_1 + \dots + nask(w_s) ; D_s. \end{aligned}$$

Let us first establish that there are no alternative guarded by a $tell(t_j)$ primitive. Indeed, if this was not the case then

$$\begin{aligned} F' = \langle \mathcal{C}(\{\}, \{+x_1\}) ; \mathcal{C}(NT) \mid \emptyset \rangle &\longrightarrow \dots \longrightarrow \langle \mathcal{C}(NT) \mid S_{x_1} \rangle \\ &\longrightarrow \langle A_j \mid S_{x_1} \cup \{t_j\} \rangle \end{aligned}$$

would be, by properties (P_2) , a valid computation prefix for $\mathcal{C}(\{\}, \{+x_1\}) ; NT$ which, by (P_3) , can only be continued by failing suffixes since $\mathcal{O}(\{\}, \{+x_1\}) ; NT = \{(\{x_1\}, \delta^-)\}$. However F' induces the following computation prefix F'' for $\mathcal{C}(\{\}, \{+x_1\}) ; (NT + (\{+x_1\}, \{\}))$ and thus a failing computation for it, which, by P_3 is absurd since the computation of $(\{\}, \{+x_1\}) ; (NT + (\{+x_1\}, \{\}))$ succeeds:

$$\begin{aligned} F'' = \langle \mathcal{C}(\{\}, \{+x_1\}) ; (\mathcal{C}(NT) + (\{+x_1\}, \{\})) \mid \emptyset \rangle &\longrightarrow \dots \\ &\longrightarrow \langle \mathcal{C}(NT + (\{+x_1\}, \{\})) \mid S_{x_1} \rangle \longrightarrow \langle A_j \mid S_{x_1} \cup \{t_j\} \rangle \end{aligned}$$

Let us now note prove that

$$\{w_1, \dots, w_s\} \subseteq (S_{x_1} \cap \dots \cap S_{x_n})$$

Indeed, if there was some $w_k \notin S_{x_i}$, then using similar reasonings for $(\{\}, \{+x_i\}) ; NT$ and $(\{\}, \{+x_i\}) ; (NT + (\{+x_i\}, \{\}))$ and employing the $nask(w_k) ; D_k$ alternative of $\mathcal{C}(NT)$ would lead to a contradiction.

To establish the final contradiction, consider $(\{\}, \{+x_{m+1}\}) ; NT$. A possible computation prefix for $\mathcal{C}(\{\}, \{+x_{m+1}\}) ; NT$ is, by P_2 , as follows: $\langle \mathcal{C}(\{\}, \{+x_{m+1}\}) ; \mathcal{C}(NT) \mid \emptyset \rangle \xrightarrow{*} \langle \mathcal{C}(NT) \mid S_{x_{m+1}} \rangle$. Since $(\{\}, \{+x_{m+1}\}) ; NT$ has a successful computation, and since $\{w_1, \dots, w_s\} \subseteq S_a \cap S_{x_1} \cap \dots \cap S_{x_m} \subseteq S_{x_{m+1}}$ there should exist j such that $u_j \in S_{x_{m+1}}$ or $v_j \in S_{x_{m+1}}$. Let us assume the first case hold, the other being treated simi-

larly. In these conditions, as $S_{x_{m+1}} \subseteq S'_{x_{m+1}}$, the following derivation is valid:

$$\begin{aligned} H &= \langle \mathcal{C}(\{\{\}, \{+x_{m+1}\}\})^{n+2} ; \mathcal{C}(\{\{\}, \{+a\}\}) ; \mathcal{C}(NT) \mid \emptyset \rangle \\ &\longrightarrow^* \langle \mathcal{C}(NT) \mid S'_{x_{m+1}} \rangle \longrightarrow \langle B_j \mid S'_{x_{m+1}} \setminus \{u_j\} \rangle \end{aligned}$$

Moreover, as $(\{\{\}, \{+x_{m+1}\}\})^{n+2} ; (\{\{\}, \{+a\}\}) ; NT$ has only one failing computation, H should be continued by failing suffixes only. It follows that H introduces a failing computations for $(\{\{\}, \{+x_{m+1}\}\})^{n+2} ; (\{\{\}, \{+a\}\}) ; (NT + (\{+a\}, \{-a\}))$ which is impossible in view of (P_3) since the latter agent has only one successful computation. \square

5.2.3 Embedding the \mathcal{L}_{CS} family into the \mathcal{L}_{MR} class

Let us now turn to the embeddings of the \mathcal{L}_{CS} family of languages in the languages of \mathcal{L}_{MR} .

Proposition 17 $\mathcal{L}_{CS}(tell, ask) \leq \mathcal{L}_{MR}(tell, ask)$.

PROOF. Indeed tell and ask primitives can be performed in any order under the observation that an $ask(t)$ primitive is redundant if it is executed after a $tell(t)$ primitive. In particular, (under this observation) all the ask primitives can be executed before all the tell primitives, as operated in $\mathcal{L}_{MR}(tell, ask)$. Intuitively speaking, the translation of any agent $A \in \mathcal{L}_{CS}(tell, ask)$ is then obtained from A by applying the following rewriting rules, where t and u denotes distinct tokens:

$$\begin{aligned} tell(t), ask(t) &\longmapsto tell(t) \\ tell(t), ask(u) &\longmapsto ask(u), tell(t) \end{aligned}$$

The corresponding coder is formally defined as:

$$\mathcal{C}(S) = f(S, (\{\}, \{\}))$$

where S is a communication transaction and where f is defined as follows:

$$f([\], (pre, post)) = (pre, post)$$

$$f([tell(t) :: S], (pre, post)) = f(S, (pre, post \cup \{+t\}))$$

$$f([ask(t) :: S], (pre, post)) = \begin{cases} f(S, (pre, post)) & \text{if } t \in (pre^+ \cup post^+) \\ f(S, (pre \cup \{+t\}, post)) & \text{otherwise} \end{cases}$$

□

Proposition 18 $\mathcal{L}_{CS}(tell, ask, get) \leq \mathcal{L}_{MR}(tell, ask, get)$.

PROOF. The proof proceeds as for Proposition 17 but with further care for the non-commutativity of ask and get primitives: $ask(t), get(t)$ requires only one copy of t whereas $get(t), ask(t)$ requires two copies. Intuitively speaking, the resulting rewriting system is as follows:

$$\begin{aligned}
tell(t), get(t) &\mapsto \lambda \\
tell(t), get(u) &\mapsto get(u), tell(t) \\
ask(t), get(t) &\mapsto get(t) \\
ask(u), get(t) &\mapsto get(t), ask(u) \\
tell(t), ask(t) &\mapsto tell(t) \\
tell(t), ask(u) &\mapsto ask(u), tell(t)
\end{aligned}$$

Note that the reduced form has the property of first listing the get operations, then the ask operations and finally the tell operation while preserving the operational semantics of the agents.

The corresponding coder is formally defined as:

$$\mathcal{C}(S) = f(S, (\{\}, \{\}))$$

where S is a communication transaction and where f is defined as follows:

$$f([\], (pre, post)) = (pre, post)$$

$$f([tell(t) :: S], (pre, post)) = \begin{cases} f(S, (pre, post \setminus \{-t\})) & \text{if } t \in post^- \\ f(S, (pre, post \cup \{+t\})) & \text{if } t \notin post^- \end{cases}$$

$$f([ask(t) :: S], (pre, post)) = \begin{cases} f(S, (pre, post)) & \text{if } (t \in post^+) \text{ or} \\ & (t \notin post \text{ and } t \in pre^+) \\ f(S, (pre \cup \{+t\}, post)) & \text{otherwise} \end{cases}$$

$$f([get(t) :: S], (pre, post)) = \begin{cases} f(S, (pre, post \setminus \{+t\})) & \text{if } t \in post^+ \text{ and } t \notin post^- \\ f(S, (pre, post \cup \{-t\})) & \text{if } t \notin post \text{ and } t \in pre^+ \\ f(S, (pre \cup \{+t\}, post \cup \{-t\})) & \text{otherwise} \end{cases}$$

□

As for Proposition 16, the separation result for $\mathcal{L}_{CS}(\text{tell}, \text{get}, \text{nask})$ and $\mathcal{L}_{MR}(\text{tell}, \text{get}, \text{nask})$ requires a saturation lemma.

Notation 19 For any agent $A \in \mathcal{L}_{MR}(\text{tell}, \text{ask}, \text{nask}, \text{get})$ and any integer n , denote by \vec{A}^n the agent obtained as the sequential composition of n copies of A . Moreover, extend this notation to $\mathcal{L}_{CS}(\text{tell}, \text{ask}, \text{nask}, \text{get})$ by defining \vec{A}^n to mean the communication transaction obtained by concatenating n times the list of communication primitives of A .

Lemma 20 Let $A \in \mathcal{L}_{MR}(\text{tell}, \text{ask}, \text{nask}, \text{get})$ be an agent such that, for any integer n , the agent \vec{A}^n has a successful computation, say resulting in store S_n . Then there is p and $q > p$ such that $S_p \subseteq S_q$ and such that $\text{set}(S_p) = \text{set}(S_q)$, where $\text{set}(M)$ denotes the set associated with the multiset M .

PROOF. For the ease of the proof, given an element x of the multiset E , let $\#(x, E)$ denote the number of occurrences of x in E .

Let us proceed by contradiction. Assume thus that the following property (P_4) holds: for any p and any $q > p$ either $S_p \not\subseteq S_q$ or $\text{set}(S_p) \neq \text{set}(S_q)$, namely, using the above notation, that either there is $x \in S_p$ such that $\#(x, S_p) > \#(x, S_q)$ or that $\text{set}(S_p) \subset \text{set}(S_q)$.³

In these conditions, let us first establish that there is a subsequence $(\nu_i)_i$ such that the following property (P_5) holds: for any p and any $q > p$, there is $x \in S_{\nu_p}$ such that $\#(x, S_{\nu_p}) > \#(x, S_{\nu_q})$. Indeed, A can only tell a finite number of tokens. Call \mathcal{T} the set of these tokens. It follows that the sets $\text{set}(S_i)$'s are members of $\text{power set}(\mathcal{T})$, of a finite cardinality. Consequently, one of the sets $\text{set}(S_i)$ is necessarily repeated in the sequence $(S_i)_i$. The corresponding subsequence $(S_{\mu_i})_i$ should verify property (P_1) but is such that $\text{set}(S_{\mu_i}) \subset \text{set}(S_{\mu_j})$ cannot hold for $i \neq j$. It must thus satisfy property (P_5).

Given property (P_5), for any $i > 1$, there is thus $x_i \in S_{\nu_1}$ such that $\#(x_i, S_{\nu_1}) > \#(x_i, S_{\nu_i})$. Consider the series of these x_i 's. In view of the finite choice in $\mathcal{P}(\mathcal{T})$ for the sets S_{ν_i} , one of the x_i should occur an infinite number of times. Take s_1 to be such an x_i and let us focus on the subsequence S_{α_i} of the S_{ν_i} for which $x_i = s_1$ with S_{ν_1} as first element. Again for any $i > \alpha_2$, there is $y_i \in S_{\alpha_1}$ such that $\#(y_i, S_{\alpha_1}) > \#(y_i, S_{\alpha_i})$. Take s_2 to be one of y_i 's occurring infinitely and continue the reasoning on the corresponding subsequence of S_{α_i} . This in the end produces a series $(s_i)_i$ of tokens and a subseries $(S_{\tau_i})_i$ such that

$$\forall i \forall j > i \#(s_i, S_{\tau_i}) > \#(s_i, S_{\tau_j})$$

³ We use here $A \subset B$ in a strict sense.

However, the tokens s_i 's are members of \mathcal{T} and thus at least one of them occurs an infinite number of times. Let s be such a token and (S_{ξ_i}) be the subseries corresponding to $s_i = t$. It verifies

$$\#(s, S_{\xi_1}) > \#(s, S_{\xi_2}) > \cdots > \#(s, S_{\xi_i}) > \cdots$$

which induces that s occurs an infinite number of times in S_{ξ_1} . However, in view of the transition rules, the set S_{ξ_1} can only be finite.

□

Proposition 21 $\mathcal{L}_{CS}(tell, ask, get, nask) \not\subseteq \mathcal{L}_{MR}(tell, ask, get, nask)$.

PROOF. By contradiction, assume the existence of a coder \mathcal{C} . Consider $\mathcal{C}([tell(t)])$ for some token t . By Lemma 20, there is p and $q > p$ such that $S_p \subseteq S_q$ and $set(S_p) = set(S_q)$, where S_p and S_q denotes the store resulting from one successful computation of $\overline{\mathcal{C}([tell(t)])}^p$ and $\overline{\mathcal{C}([tell(t)])}^q$, respectively.

Consider now $Tp = \overline{[tell(t)]}^p$, $Tq = \overline{[tell(t)]}^q$, and $T = \overline{[get(t)]}^p, nask(t)$. Let $m_1 ; M_1 + \cdots + m_r ; M_r$ be the normal form of $\mathcal{C}(T)$, with the m_i 's being atomic communication actions of $\mathcal{L}_{MR}(tell, ask, get, nask)$.

As $Tp ; T$ has one successful computation, it follows that $\langle \mathcal{C}(T) \mid S_p \rangle \longrightarrow \langle M_i \mid \sigma \rangle$ for some $i \in \{1, \dots, r\}$, and some store σ and consequently, thanks to the relation between S_p and S_q , that $\langle \mathcal{C}(T) \mid S_q \rangle \longrightarrow \langle M_i \mid \sigma' \rangle$ for some store σ . As $Tq ; T$ fails, it follows from (P_3) that the computation of M_i starting from σ' is failing. Therefore, although $Tq ; (T + [get(t)])$ has only one successful computation, the following derivation D is a valid computation prefix at the coded level which leads to failure, which is absurd by (P_3) :

$$\begin{aligned} D = \langle \mathcal{C}(Tq) ; (\mathcal{C}(T) + \mathcal{C}([get(t)])) \mid \emptyset \rangle &\longrightarrow^* \langle \mathcal{C}(T) + \mathcal{C}([get(t)]) \mid S_q \rangle \\ &\longrightarrow \langle M_i \mid \sigma' \rangle \end{aligned}$$

□

5.2.4 Hierarchies in the \mathcal{L}_L , \mathcal{L}_{MR} and \mathcal{L}_{CS} families

The coding \mathcal{C} in the proof of Proposition 8 translates any of the basic primitives of \mathcal{L}_L into an equivalent form in \mathcal{L}_{MR} . With respect to the class \mathcal{L}_{CS} , the translation is even more straightforward since any communication primitive c can be directly coded as $[c]$.

As a result, any combination of basic primitives which has established a separation result in the family of languages \mathcal{L}_L (see [6]) can be re-employed to prove a corresponding result in the family of languages \mathcal{L}_{MR} and \mathcal{L}_{CS} . This fact combined with embedding induced by language inclusion establish the following proposition.

Proposition 22

- i)* $\mathcal{L}_{MR}(ask, tell) < \mathcal{L}_{MR}(ask, get, tell) < \mathcal{L}_{MR}(ask, nask, get, tell)$
- ii)* $\mathcal{L}_{CS}(ask, tell) < \mathcal{L}_{CS}(ask, get, tell) < \mathcal{L}_{CS}(ask, nask, get, tell)$

Finally, for the sake of completeness, we recall here two relations on the \mathcal{L}_L family which are exploited in Figure 4 and which were established in [6]:

- i)* $\mathcal{L}_L(ask, get, tell) \not\leq \mathcal{L}_L(ask, tell)$
- ii)* $\mathcal{L}_L(ask, nask, get, tell) \not\leq \mathcal{L}_L(ask, get, tell)$

5.2.5 Relating the families \mathcal{L}_L and \mathcal{L}_{CS}

A few further propositions are required to complete the comparison. We start by relations between the \mathcal{L}_L and \mathcal{L}_{CS} families of languages.

Proposition 23 $\mathcal{L}_L(ask, tell) \leq \mathcal{L}_{CS}(ask, tell)$

PROOF. Immediate by coding any $ask(t)$, $nask(t)$, $tell(t)$ primitive as $[ask(t)]$, $[nask(t)]$, $[tell(t)]$, respectively. \square

Proposition 24 $\mathcal{L}_{CS}(ask, tell) \not\leq \mathcal{L}_L(ask, tell)$

PROOF. Similar to the proof of Proposition 11. \square

Proposition 25

- i)* $\mathcal{L}_L(ask, get, nask, tell) \not\leq \mathcal{L}_{CS}(ask, tell)$
- ii)* $\mathcal{L}_{CS}(ask, tell) \not\leq \mathcal{L}_L(ask, get, nask, tell)$

PROOF. (i) By contradiction, suppose that $\mathcal{L}_L(ask, nask, get, tell) \leq \mathcal{L}_{CS}(ask, tell)$. Let us then establish that, for the considered coder and for any token t , $\mathcal{C}(tell(t) ; nask(t))$ has only successful computations, which by P_3 , contradicts the fact that $\mathcal{O}(tell(t) ; nask(t)) = \{(\{t\}, \delta^-)\}$. Indeed, since $\mathcal{O}(tell(t)) = \{(\{t\}, \delta^+)\}$, any computation of $\mathcal{C}(tell(t))$ is successful. Similarly,

it follows from $\mathcal{O}(nask(t)) = \{(\emptyset, \delta^+)\}$ that any computation of $\mathcal{C}(nask(t))$ is successful and, consequently, so is any computation of $\mathcal{C}(nask(t))$ starting on any store. It follows that any computation of $\mathcal{C}(tell(t))$ can be followed by a successful computation of $\mathcal{C}(nask(t))$ and thus, by P_2 , that $\mathcal{C}(tell(t) ; nask(t))$ has only successful computations.

(ii) By contradiction, assume that there is a coder \mathcal{C} which translates agents of $\mathcal{L}_{CS}(ask, tell)$ into agents of $\mathcal{L}_L(ask, nask, get, tell)$. Let us first observe that, for any token t , there are tokens $x_1, \dots, x_m, y_1, \dots, y_n$ such that if σ is the store resulting from one computation C of $\mathcal{C}([tell(x_1)] ; \dots ; [tell(x_m)])$ and if $\sigma \cup \alpha \setminus \beta$ is the store resulting from the continuation of C by the computation of $\mathcal{C}([tell(t)])$ then there is a computation of $\mathcal{C}([tell(x_1)] ; \dots ; [tell(x_m)] ; [tell(y_1)] ; \dots ; [tell(y_n)])$ ending in a store τ such that the computation of $\mathcal{C}([tell(x_1)] ; \dots ; [tell(x_m)] ; [tell(y_1)] ; \dots ; [tell(y_n)] ; [tell(t)])$ ends in the store $\tau \cup \alpha \setminus \beta$. Indeed, any computation of $\mathcal{C}([tell(t)])$ can be viewed as a sequence of ask, nask, tell and get operations. Since $\mathcal{C}([tell(t)])$ is finite, there is only a finite set of such sequences. Moreover, for any set of distinct tokens z_1, \dots, z_p , any computation of $\mathcal{C}([tell(z_1)] ; \dots ; [tell(z_p)] ; [tell(t)])$, which is necessarily successful by P_3 , necessarily uses, by P_2 such a sequence. In these conditions, progressively increasing the set of tokens z_i necessarily results in repeating a sequence, which establishes the claim.

To conclude, let us consider the normal form of $\mathcal{C}(XYT)$ for

$$XYT = [ask(x_1), \dots, ask(x_m), ask(y_1), \dots, ask(y_n), ask(t)]$$

which, in its most general form, is written as

$$\begin{aligned} & tell(t_1) ; A_1 + \dots + tell(t_p) ; A_p + nask(a_1) ; B_1 + \dots + nask(a_q) ; B_q \\ & + ask(v_1) ; C_1 + \dots + ask(v_r) ; C_r + get(w_1) ; D_1 + \dots + get(w_s) ; D_s \end{aligned}$$

As for Proposition 11, it is possible to prove there are actually no alternatives guarded by a $tell(t_i)$ operation. Moreover, let us establish that there are no alternatives guarded by a $nask(a_j)$ primitive. Indeed, otherwise, the transition $\langle \mathcal{C}(XYT) \mid \emptyset \rangle \longrightarrow \langle B_j \mid \emptyset \rangle$ would be valid, which, as XYT has only one failing computation, can only be continued by failing computations. However, this transition then induces a failing computation for $\mathcal{C}(XYT) + \mathcal{C}([tell(t)])$, which is absurd by P_2 and P_3 .

Now, observe that the three agents

$$\begin{aligned} & [tell(x_1)] ; \cdots ; [tell(x_m)] ; XYT \\ & [tell(x_1)] ; \cdots ; [tell(x_m)] ; [tell(t)] ; XYT \\ & [tell(x_1)] ; \cdots ; [tell(x_m)] ; [tell(y_1)] ; \cdots ; [tell(y_n)] ; XYT \end{aligned}$$

all have just one failing computation. Using the same argument as above to prove the absence of $nask(a_j)$ operations, it follows that $\{v_1, \dots, v_r, w_1, \dots, w_s\} \cap (\sigma \cup \tau \cup \alpha) = \emptyset$. However, the following computation prefix is then valid:

$$\begin{aligned} & \langle \mathcal{C}([tell(x_1)] ; \cdots ; [tell(x_m)] ; [tell(y_1)] ; \cdots ; [tell(y_n)] ; [tell(t)] ; XYT) \mid \emptyset \rangle \\ & \longrightarrow^* \langle \mathcal{C}([tell(t)] ; XYT) \mid \tau \rangle \longrightarrow \langle \mathcal{C}(XYT) \mid \tau \cup \alpha \setminus \beta \rangle \not\rightarrow . \end{aligned}$$

It yields a failing computation for $\mathcal{C}([tell(x_1)] ; \cdots ; [tell(x_m)] ; [tell(y_1)] ; \cdots ; [tell(y_n)] ; XYT)$, which is absurd by P_3 . \square

Proposition 26

- i) $\mathcal{L}_L(ask, get, tell) \not\leq \mathcal{L}_{CS}(ask, tell)$
- ii) $\mathcal{L}_{CS}(ask, tell) \not\leq \mathcal{L}_L(ask, get, tell)$

PROOF. (i) By contradiction, assume that $\mathcal{L}_L(ask, get, tell) \leq \mathcal{L}_{CS}(ask, tell)$. Consider $tell(a) ; get(a)$ for some arbitrary token a . As $\mathcal{O}(tell(a) ; get(a)) = \{(\emptyset, \delta^+)\}$, it follows that any computation of $\mathcal{C}(tell(a) ; get(a))$ is successful. However such a computation is, by P_2 , a computation of $\mathcal{C}(tell(a))$ followed by a computation of $\mathcal{C}(get(a))$. Since $\mathcal{C}(get(a))$ is composed of ask and tell operations only, this latter computation can be repeated, which induces a successful computation for $\mathcal{C}(tell(a) ; get(a) ; get(a))$ and hence, by P_3 , for $tell(a) ; get(a) ; get(a)$. This is obviously not possible.

(ii) The proof of point (ii) consists of rephrasing the proof of Proposition 11 by taking $AB = [ask(a), ask(b)]$ and by replacing the operations $ask(u_i)$ by $get(u_i)$, the destructive character of the get operations playing no role in the proof. \square

5.2.6 Further separation results between the \mathcal{L}_L and \mathcal{L}_{MR} families

Proposition 27

- i) $\mathcal{L}_L(ask, get, tell) \not\leq \mathcal{L}_{MR}(ask, tell)$
- ii) $\mathcal{L}_{MR}(ask, tell) \not\leq \mathcal{L}_L(ask, get, tell)$

PROOF. (i) The proof of point (i) consists of a simple adaptation of the proof of point (i) of Proposition 26.

(ii) Otherwise, by Proposition 17, $\mathcal{L}_{CS}(ask, tell) \leq \mathcal{L}_L(get, tell)$, which contradicts Proposition 26. \square

Proposition 28

- i) $\mathcal{L}_{MR}(ask, tell) \not\leq \mathcal{L}_L(ask, nask, get, tell)$
- ii) $\mathcal{L}_L(ask, nask, get, tell) \not\leq \mathcal{L}_{MR}(ask, tell)$

PROOF. (i) Otherwise, by Proposition 17, the embedding $\mathcal{L}_{CS}(ask, tell) \leq \mathcal{L}_L(ask, nask, get, tell)$ would hold, which contradicts Proposition 25.

(ii) Otherwise, by the embedding $\mathcal{L}_L(ask, get, tell) \leq \mathcal{L}_L(ask, nask, get, tell)$, induced by sublanguage inclusion, the embedding $\mathcal{L}_L(ask, get, tell) \leq \mathcal{L}_{MR}(ask, tell)$ would hold, which contradicts Proposition 27. \square

Proposition 29

- i) $\mathcal{L}_{MR}(ask, get, tell) \not\leq \mathcal{L}_L(ask, nask, get, tell)$
- ii) $\mathcal{L}_L(ask, nask, get, tell) \not\leq \mathcal{L}_{MR}(ask, get, tell)$

PROOF. (i) Otherwise, the embedding $\mathcal{L}_{MR}(ask, tell) \leq \mathcal{L}_L(ask, nask, get, tell)$ holds, which contradicts Proposition 28.

(ii) The proof proceeds as for part (i) of Proposition 25. \square

6 Concluding remarks

We have compared the expressive power of three families of coordination models based on shared dataspace. The first class \mathcal{L}_L relies on Linda's communication primitives, the second class \mathcal{L}_{MR} relies on the more general notion of

multi-set rewriting, while the third class \mathcal{L}_{CS} features communication transactions that consist of sequences of Linda-like operations to be executed atomically. For each family we have considered three different languages that differ from one another in the set \mathcal{X} of communication primitives used, for \mathcal{X} equal respectively to $\{ask, tell\}$, $\{ask, get, tell\}$ and $\{ask, nask, get, tell\}$.

It is worth mentioning that we have exploited the main proof techniques reported in this paper to perform a wider comparison of the languages, by considering also other sets \mathcal{X} of communication primitives with $\mathcal{X} \subseteq \{ask, nask, get, tell\}$. We decided to report only the main comparisons in this paper, because of lack of space.

As pointed out in the Introduction, the families \mathcal{L}_L , \mathcal{L}_{MR} , and \mathcal{L}_{CS} are representative of a substantial amount of coordination languages. We believe that the comparison of the expressive power of different classes of coordination models provides useful insights for both the theory and the practice of coordination-based approaches. The resulting hierarchy highlights the equivalence of different models and indicates which extensions may be worth considering because of their additional expressive power.

Acknowledgments. We would like to thank the anonymous referees for their valuable comments and suggestions on a previous version of this article. The first author was partly supported by the M.U.R.S.T. project “Theory of Concurrency, Higher Order and Types (TOSCA)”.

References

- [1] J.M. Andreoli and R. Pareschi. Linear Objects: logical processes with builtin inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.
- [2] J. Banatre and D. LeMetayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1991.
- [3] E. Best, F.S. de Boer and C. Palamidessi. Partial Order and SOS Semantics for Linear Constraint Programs. In D. Garlan and D. Le Metayer, editors, *Coordination '97*, LNCS, 1997.
- [4] M. Bonsangue, J. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In proceeding of the ACM Symposium on Applied Computing, SAC'99. Pag. 156-165, ACM Press, February 1999.
- [5] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

- [6] A. Brogi and J.-M. Jacquet. On the Expressiveness of Linda-like Concurrent Languages. *Electronic Notes in Theoretical Computer Science*, 16(2), 1998.
- [7] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing Equivalence of Linda Coordination Primitives. *Electronic Notes in Theoretical Computer Science*, 7, 1997.
- [8] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.
- [9] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
- [10] N. Carriero, D. Gelernter and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz and A. Yonezawa (editors) *Object based models and languages for concurrent systems*, LNCS 924, pages 66–76, Springer-Verlag, 1994.
- [11] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, 1994.
- [12] P. Ciancarini and C. Hankin (editors). Coordination'96: Proceedings of The First International Conference on Coordination Models and Languages. Number 1061 in LNCS. Springer-Verlag, 1996.
- [13] P. Ciancarini and D. Rossi. Jada: coordination and communication for Java agents. In *Second Int. Workshop on mobile object systems*, LNCS 1222, pages 213–228, Springer-Verlag, 1996.
- [14] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 108(1):128–157, 1994.
- [15] R. De Nicola, G. Ferrari and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, Special Issue on Mobility and Network Aware Computing (Catalin Roman and Ghezzi Eds), 1998.
- [16] M. Felleisen. On the Expressive Power of Programming Languages. In N. Jones, editor, *Proceedings European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 134–151. Springer-Verlag, 1990.
- [17] D. Garlan and D. Le Metayer (editors). Coordination'97: Proceedings of The Second International Conference on Coordination Models and Languages. LNCS. Springer-Verlag, 1997.
- [18] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
- [19] J.M. Jacquet and K. De Bosschere. On the Semantics of muLog. *Future Generation Computer Systems Journal*:10, pages 93–135, Elsevier, 1994.
- [20] J.-M. Jacquet and L. Monteiro. Towards Resource Handling in Logic Programming: the PPL Framework and its Semantics. *Computer Languages*, 22(2/3):51–77, 1996.

- [21] G.A. Papadopolous and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, Academic-Press, 1998.
- [22] A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In *30th Hawaii Int. Conf. on System Sciences*, IEEE Press, Vol. 1, pages 379–388, 1997.
- [23] V.A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.
- [24] E.Y. Shapiro. Embeddings among Concurrent Programming Languages. In W.R. Cleaveland, editor, *Proceedings of CONCUR'92*, pages 486–503. Springer-Verlag, 1992.
- [25] R. Tolksdorf. Coordinating services in open distributed systems with LAURA. In [12], pages 386–402.
- [26] P. Wegner. Why Interaction Is More Powerful Than Algorithms. *Communications of the ACM*, 1997.
- [27] G. Zavattaro. On the incomparability of Gamma and Linda. Technical Report SEN-R9827, Department of Software Engineering, Centrum voor Wiskunde en Informatica (CWI), 1998.
- [28] G. Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. *Electronic Notes in Theoretical Computer Science*, 16(2), 1998.