Electronic Notes in Theoretical Computer Science 16 No. 2 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume16.html 22 pages

# On the Expressiveness of Linda-like Concurrent Languages

Antonio Brogi [1]

*Dipartimento di Informatica*
*Università di Pisa*
*Pisa, Italy*

Jean-Marie Jacquet [1]

*Institut d'Informatique*
*Facultés Universitaires de Namur*
*Namur, Belgium*

**Abstract**

We compare the expressiveness of a class of concurrent languages that employ asynchronous communication primitives à la Linda. All the languages considered contain sequential, parallel and choice operators, and they differ from one another in the set of communication primitives used. These primitives include *tell*, *get* and *ask* operations for adding, deleting, and checking for the presence of data in a dataspace shared by a number of concurrent processes, as well as a *nask* (*n*egative *ask*) operation for checking for the absence of data in the shared dataspace. We use the notion of modular embedding introduced by De Boer and Palamidessi in [3] to compare the relative expressive power of the languages. A first result is the formalisation of the intuitive separation result stating that the language with *get* and *tell* is strictly more expressive than the language with *ask* and *tell* operations. An interesting result is that the ability to check for the presence of information (*ask*) does not increase the power of a language containing *get* and *tell* operations, whereas the ability to check for the absence of information (*nask*) does increase the power of such a language. Another interesting result shown is that the language containing all the communication primitives considered is strictly more expressive than each of its sub-languages, except for the redundancy of *ask*.

[1] This work has been partially supported by the INTAS Project 93-172 and by the Belgium-Italy cooperation project no. 95.12.

# 1 Introduction

Most modern computing systems consist of large numbers of software components that interact with one another. The constant expansion of computer networks is promoting the development of distributed applications based on the interaction of heterogeneous software components, which are generally distributed on the net.

The paradigm shift from stand-alone to distributed computer systems has made the issue of *interaction* [16] one of the central issues both in the theory and in the practice of Computer Science. For instance, Wegner [20] recently supported the provocative argument that "interaction is more important than algorithms", namely that the way in which a software component interacts with the environment is by far more important than what the component alone is able to compute.

Increasing attention is being paid to the related issues of the *coordination* of software components [8,11]. Terms like "coordination" and "interaction" have a wide meaning and are used to label a number of different activities. For instance, the term "coordination" is often used to denote the issues related to the specific problem of how heterogeneous, existing software components can be integrated together so as to coordinate their activities and to let them interact with one other.

If on the one hand renewed attention is being paid to the issues of interaction and coordination, on the other hand a substantial body of research has been already devoted to study the issues of communication and synchronisation of concurrent processes. A lot of work has been done in the design of concurrent languages (e.g., CSP, concurrent constraint programming, Linda) as well as in process algebras (e.g., CCS, $\pi$-calculus, ACP), just to mention some examples from two large research bodies.

One of the intriguing questions that is still somehow open to the debate in the scientific community is the following: *Which is the "best" model for expressing the coordination or the communication of concurrent components?* Of course the question depends on what we mean by the "best" model. A formal way of making precise this question is to reformulate it in terms of the expressive power of models and languages.

As pointed out in [3], 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 [13], 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", 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 [3] 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 a class of concurrent languages that employ asynchronous communication primitives à la Linda [7]. All the languages considered contain sequential, parallel and choice operators, and they differ from one another in the set of communication primitives used. The set of communication primitives includes *tell*, *get* and *ask* primitives for adding, deleting, and checking for the presence of data in a dataspace shared by a number of concurrent processes. A primitive *nask* (negative *ask*) is also considered, whose meaning is the dual of *ask*, namely checking for the absence of data in the shared dataspace.

We then compare the relative expressive power of languages using different sets of communication primitives from the set $\{ask, nask, get, tell\}$. We shall denote by $\mathcal{L}(\mathcal{X})$ the language containing the set $\mathcal{X} \subseteq \{ask, nask, get, tell\}$ of communication primitives (plus of course the sequential, parallel and choice operators).

It is easy to see that a number of (modular) embeddings can be trivially established by considering sub-languages. For instance:

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

where $L' \leq L$ denotes that $L'$ can be embedded into $L$.

The most interesting results are however *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. Consider for instance two languages $\mathcal{L}(\mathcal{X})$ and $\mathcal{L}(\mathcal{X}')$, where the set of primitives $\mathcal{X}'$ extends the set $\mathcal{X}$ with another primitive $c$ (viz., $\mathcal{X}' = \mathcal{X} \cup \{c\}$.) A separation result or an equivalence result on $\mathcal{L}(\mathcal{X})$ and $\mathcal{L}(\mathcal{X}')$ indicates whether the inclusion of the new primitive $c$ really increases (separation) the expressive power of a language $\mathcal{L}(\mathcal{X})$, or whether $c$ is instead just "syntactic sugar" added to $\mathcal{L}(\mathcal{X})$ (equivalence).

Our study of this class of languages is complete in the sense that all possible relations between pairs of languages in the class 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). One of the separation results is for instance that:

$$\mathcal{L}(\text{ask, tell}) \;<\; \mathcal{L}(\text{ask, get, tell})$$

that is, the ability of deleting information from a dataspace (in addition to adding and checking for the presence of information) strictly increases the expressive power of the language. This separation result can be for instance used

for comparing concurrent constraint programming [18] with its non-monotonic extensions [2].

Another interesting result is that once we have the ability of adding (*tell*) and deleting (*get*) information, the availability of primitives for checking for the presence of information (*ask*) is only "syntactic sugar", formally:

$$\mathcal{L}(\text{get, tell}) \;=\; \mathcal{L}(\text{ask, get, tell}).$$

On the other hand, the ability of checking for the *absence* of information (*nask*) does increase the expressive power in this context, namely:

$$\mathcal{L}(\text{get, tell}) \;<\; \mathcal{L}(\text{nask, get, tell}).$$

A consequence of these results is the formalisation of the intuitive separation result stating that the language with *get* and *tell* is strictly more expressive than the language with *ask* and *tell* operations, namely:

$$\mathcal{L}(\text{ask, tell}) \;<\; \mathcal{L}(\text{get, tell}).$$

Finally we show that the availability of all the above communication primitives makes a language strictly more expressive than each of its sub-languages, except for the redundancy of *ask* with respect to *get*, namely:

$$\mathcal{L}(\text{nask, get, tell}) \;=\; \mathcal{L}(\text{ask, nask, get, tell})$$

and

$$\mathcal{L}(\mathcal{X}) < \mathcal{L}(\text{ask, nask, get, tell})$$

for any subset $\mathcal{X}$ of $\{ask, nask, get, tell\}$ different from $\{nask, get, tell\}$.

The results presented in this paper provide a formal ground for reasoning on the family of concurrent languages based on Linda-like communication primitives. From a practical point of view, these results can be exploited when evaluating whether it is worth or not to include a communication primitive (e.g., like *nask*) in a Linda-like language.

The paper is organised as follows. Section 2 introduces the notion of modular embedding proposed in [3]. Section 3 formally defines syntax and operational semantics of the class of concurrent languages considered. Section 4 contains an exhaustive comparison of the expressive power of the non-trivial languages of the class, namely of all languages containing *tell*. This section contains a large number of propositions, with a proof sketch associated with each proposition. Figure 3 summarises the results presented in this section. Finally section 5 contains a discussion of related work and some concluding remarks.

For the sake of completeness, the analysis of the other languages in the class (not containing *tell*) is reported in the Appendix, where figure 4 summarises the whole set of results.

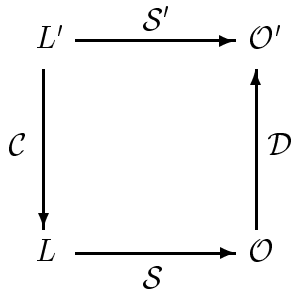


Fig. 1. Basic embedding.

## 2 Modular embedding

We summarise here the method for language comparison, called *modular embedding*, proposed by De Boer and Palamidessi in [3].

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 [19] is the following. Consider two languages $L$ and $L'$. Assume given the semantics mappings (*observation criteria*) $\mathcal{S} : L \to \mathcal{O}$ and $\mathcal{S}' : L' \to \mathcal{O}'$, where $\mathcal{O}$ and $\mathcal{O}'$ are some suitable domains. Then $L$ can *embed* $L'$ if there exists a mapping $\mathcal{C}$ (*compiler*) 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 1 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 [3] to add three constraints on the coder $\mathcal{C}$ and on the decoder $\mathcal{D}$ in order to obtain a notion of *modular* embedding usable for concurrent languages. Namely $\mathcal{D}$ should be defined in an element-wise way w.r.t. $\mathcal{O}$:

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

for some appropriate mapping $\mathcal{D}_{el}$. Moreover the compiler $\mathcal{C}$ should be defined in a compositional way w.r.t. the sequential, parallel and choice operators: [2]

$$\begin{aligned} \mathcal{C}(A \ ; \ B) &= \mathcal{C}(A) \ ; \ \mathcal{C}(B) \\ \mathcal{C}(A \mid\mid B) &= \mathcal{C}(A) \mid\mid \mathcal{C}(B) \\ \mathcal{C}(A \ + \ B) &= \mathcal{C}(A) \ + \ \mathcal{C}(B) \end{aligned} \qquad (P_2)$$

Finally, the embedding should preserve the behaviour of the original processes

[2] Actually, this is only required for the parallel and choice operators in [3].

w.r.t. deadlock, failure and success (*termination invariance*):

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

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.

## 3 The family of Linda-like concurrent languages

### *3.1 Syntax*

We shall consider a family of languages $\mathcal{L}(\mathcal{X})$, parameterised on the set of communication primitives $\mathcal{X}$. The set $\mathcal{X}$ consists of the basic Linda primitives `out`, `in`, and `rd` primitives, for putting an object in a shared dataspace, getting it and checking for its presence, respectively, together with a primitive testing the absence of an object from the dataspace. The languages $\mathcal{L}(\mathcal{X})$ also include sequential and parallel composition operators as well as a choice operator in the style of CCS [15]. However, for simplicity purposes, only finite processes are treated here, under the observation that infinite processes can be handled by extending the results of this paper in the classical way, as exemplified for instance in [14].

The family of languages $\mathcal{L}(\mathcal{X})$ is formally defined by the following grammar.

**Definition 3.1**

(i) *Let Stoken be a denumerable set, the elements of which are subsequently called* tokens *and are typically represented by the letters $t$ and $u$.*

(ii) *Define the set of communication actions Scom as*

$$c ::= tell(t) \mid ask(t) \mid get(t) \mid nask(t)$$

*where $t$ is a token and $c$ is a communication action.*

(iii) *Given a subset $\mathcal{X}$ of Scom, define the set of agents Sagent by the following rule, where $A$ is an agent and $c$ denotes a communication action of $\mathcal{X}$.*

$$A ::= c \mid A \ ; \ A \mid A \parallel A \mid A \ + \ A$$

*Define then the language $\mathcal{L}(\mathcal{X})$ as the above set Sagent.*

### *3.2 Operational semantics*

#### *3.2.1 Configurations.*

For any $\mathcal{X}$, computations in $\mathcal{L}(\mathcal{X})$ may be modelled by the following transition system written in Plotkin's style. Following the intuition, most of the configurations consist of an agent together with a multi-set of tokens denoting the tokens currently available for the computation. To easily express termination, we shall introduce particular configurations composed of a special terminating symbol $E$ together with a multi-set of tokens. For uniformity purposes, we shall abuse language and qualify $E$ as an agent. However, to meet the intuition, we shall always rewrite agents of the form $(E\ ;\ A)$, $(E\ ||\ A)$, and $(A\ ||\ E)$ as $A$. This is technically achieved by defining the extended set of agents as follows, and by operating simplifications by imposing a bimonoid structure.

**Definition 3.2** *Define the extended set of agents Seagent by the following grammar*

$$Ae ::= E \mid c \mid A\ ;\ A \mid A\ ||\ A \mid A\ +\ A$$

*Moreover, we shall subsequently assert that the structure* $(Seagent, E, \ ;\ ,\ ||\ )$ *is a bimonoid and simplify elements of* $\mathcal{L}(\mathcal{X})$ *accordingly.*

**Definition 3.3** *Define the set of stores Sstore as the set of finite multisets with elements from Stoken.*

**Definition 3.4** *Define the set of configurations Sconf as* $Seagent \times Sstore$*. Configurations are denoted as* $\langle A \mid \sigma \rangle$*, where A is an (extended) agent and* $\sigma$ *is a multi-set of tokens.*

#### *3.2.2 Transition rules.*

The transition rules defining the operational semantics of the language are reported in Figure 2.

Rule **(T)** states that an atomic agent $tell(t)$ can be executed in any store $\sigma$, and that its execution results in adding the token $t$ to the store $\sigma$. 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 this token, 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. Rules **(S)**, **(P)**, and **(C)** describe the operational meaning of sequential, parallel and choice operators in the standard way [15]. Note that, in the first four rules, the symbol $\cup$ actually denotes multiset union.

#### *3.2.3 Observables.*

We are now in a position to define the operational semantics.

$$\textbf{(T)} \qquad \langle tell(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$$

$$\textbf{(A)} \qquad \langle ask(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \cup \{t\} \rangle$$

$$\textbf{(N)} \qquad \frac{t \notin \sigma}{\langle nask(t) \mid \sigma \rangle \longrightarrow \langle E \mid \sigma \rangle}$$

$$\textbf{(G)} \qquad \langle get(t) \mid \sigma \cup \{t\} \rangle \longrightarrow \langle E \mid \sigma \rangle$$

$$\textbf{(S)} \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\langle A \ ; \ B \mid \sigma \rangle \longrightarrow \langle A' \ ; \ B \mid \sigma' \rangle}$$

$$\textbf{(P)} \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{l} \langle A \mid\mid B \mid \sigma \rangle \longrightarrow \langle A' \mid\mid B \mid \sigma' \rangle \\ \langle B \mid\mid A \mid \sigma \rangle \longrightarrow \langle B \mid\mid A' \mid \sigma' \rangle \end{array}}$$

$$\textbf{(C)} \qquad \frac{\langle A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle}{\begin{array}{l} \langle A \ + \ B \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \\ \langle B \ + \ A \mid \sigma \rangle \longrightarrow \langle A' \mid \sigma' \rangle \end{array}}$$

Fig. 2. Transition rules.

**Definition 3.5**

(i) *Let $\delta^+$ and $\delta^-$ be two fresh symbols denoting respectively success and failure. Define the set of histories Shist as the set Sstore $\times \{\delta^+, \delta^-\}$.*

(ii) *Define the* operational semantics *$\mathcal{O} : Sagent \to \mathcal{P}(Shist)$ as the following function: For any agent A,*

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

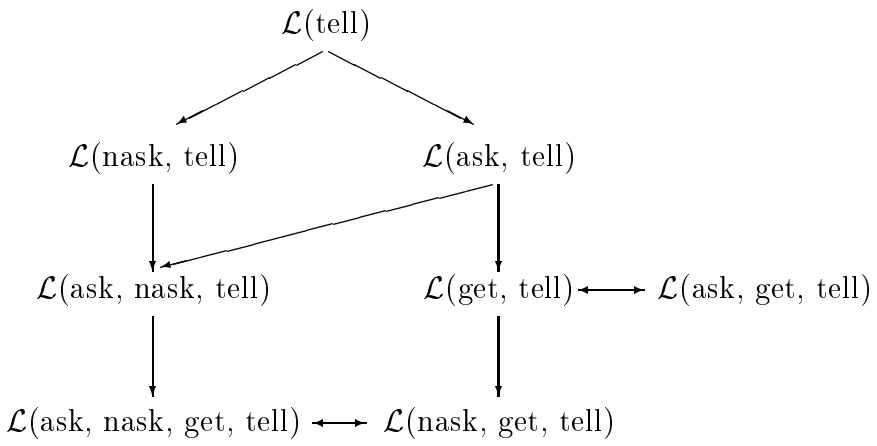


Fig. 3. The hierarchy of languages.

*3.3 Normal form*

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

**Definition 3.6** *Given a subset $\mathcal{X}$ of Scom, the set Snagent of agents in normal form is defined by the following rule, where $N$ is an agent in normal form and $c$ denotes a communication action of $\mathcal{X}$.*

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

**Proposition 3.7** *For any agent $A$, there is an agent $N$ in normal form such that $\mathcal{O}(A) = \mathcal{O}(N)$.*

## 4 Comparisons

We now perform an exhaustive comparison of the relative expressive power of the family of Linda-like concurrent languages introduced in the previous section.

We shall focus here on the non-trivial languages of the class, namely of the languages containing the *tell* operation. The whole set of separation and equivalence results are summarised in figure 3, 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 results of figure 3 are substantiated by the following propositions.

Let us first consider the languages $\mathcal{L}$(ask,tell) and $\mathcal{L}$(nask,tell) obtained by extending $\mathcal{L}$(tell) with the ability of checking for the presence and for the absence of data, respectively, in the dataspace. It is easy to show that both $\mathcal{L}$(ask,tell) and $\mathcal{L}$(nask,tell) are strictly more expressive than $\mathcal{L}$(tell).

**Proposition 4.1**

$(i)$ $\quad \mathcal{L}(tell) \leq \mathcal{L}(ask,tell)$

$(ii)$ $\quad \mathcal{L}(tell) \leq \mathcal{L}(nask,tell)$

$(iii)$ $\quad \mathcal{L}(ask,tell) \not\leq \mathcal{L}(tell)$

$(iv)$ $\quad \mathcal{L}(nask,tell) \not\leq \mathcal{L}(tell)$

**Proof.** (i) and (ii): Immediate since $\mathcal{L}$(tell) $\subseteq$ $\mathcal{L}$(ask,tell) and $\mathcal{L}$(tell) $\subseteq$ $\mathcal{L}$(nask,tell). (iii): By contradiction suppose that $\mathcal{L}$(ask,tell) $\leq$ $\mathcal{L}$(tell) and consider the agent $ask(a)$. It is easy to see that $\mathcal{O}(ask(a)) = \{(\emptyset, \delta^-)\}$ while any agent in $\mathcal{L}$(tell) has only successful computations. Hence we have a contradiction by $P_3$. (iv): The proof is analogous to the proof of (iii), by considering the agent $tell(a); nask(a)$. □

We observe that while $\mathcal{L}$(ask,tell) and $\mathcal{L}$(nask,tell) are both strictly more powerful than $\mathcal{L}$(tell), they are not comparable with each other.

**Proposition 4.2**

$(i)$ $\quad \mathcal{L}(ask,tell) \not\leq \mathcal{L}(nask,tell)$

$(ii)$ $\quad \mathcal{L}(nask,tell) \not\leq \mathcal{L}(ask,tell)$

**Proof.** (i): By contradiction suppose that $\mathcal{L}$(ask,tell) $\leq$ $\mathcal{L}$(nask,tell) and consider the agent $tell(a); ask(a)$. We show that while $\mathcal{O}(tell(a); ask(a)) = \{(\{a\}, \delta^+)\}$ the agent $\mathcal{C}(tell(a)\ ;\ ask(a))$ — which is equivalent by $P_2$ to $\mathcal{C}(tell(a))\ ;\ \mathcal{C}(ask(a))$ — has only failing computations, thus contradicting $P_3$. Indeed, any successful computation for $\mathcal{C}(tell(a))\ ;\ \mathcal{C}(ask(a))$ should start with a successful computation for $\mathcal{C}(tell(a))$ which should be followed by a successful computation for $\mathcal{C}(ask(a))$. However, as $\mathcal{O}(ask(a)) = \{(\emptyset, \delta^-)\}$, any computation of $\mathcal{C}(ask(a))$ starting on the empty store fails. It follows that any computation starting on any (arbitrary) store should fail since $\mathcal{C}(ask(a))$ is composed of tell and nask primitives only. Hence, even if $\mathcal{C}(tell(a))$ has a successful computation, this computation cannot be continued by a successful computation of $\mathcal{C}(ask(a))$. (ii): Let us again proceed by contradiction. Otherwise, $\mathcal{C}(tell(a))\ ;\ \mathcal{C}(nask(a))$ has only successful computations, which, by $P_3$ contradicts the fact that $\mathcal{O}(tell(a)\ ;\ nask(a)) = \{(\{a\}, \delta^-)\}$. Indeed, since $\mathcal{O}(tell(a)) = \{(\{a\}, \delta^+)\}$, by $P_3$ any computation of $\mathcal{C}(tell(a))$ (starting on the empty store) is successful. Similarly, it follows from $\mathcal{O}(nask(a)) = \{(\emptyset, \delta^+)\}$ that any computation starting on the empty store is successful, and consequently, so does any computation starting from any store, since $\mathcal{C}(nask(a))$ is composed of ask and tell primitives. Summing up, any (successful) computa-

tion of $\mathcal{C}(tell(a))$ starting on the empty store can be continued by a (successful) computation of $\mathcal{C}(nask(a))$. □

Let us now include the *get* primitive in the language. We first prove the intuitive separation result that $\mathcal{L}$(get,tell) is strictly more expressive than $\mathcal{L}$(ask,tell).

**Proposition 4.3**

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

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

**Proof.** (i): Indeed, it suffices to code any $ask(t)$ primitive into $get(t)$ ; $tell(t)$. (ii): Assume that $\mathcal{L}$(get, tell) $\leq$ $\mathcal{L}$(ask, tell) and consider $tell(a)$ ; $get(a)$. Since $\mathcal{C}$ is compositional and since $\mathcal{O}(tell(a)\ ;\ get(a)) = \{(\emptyset, \delta^+)\}$, the termination mark of any element of $\mathcal{O}(\mathcal{C}(tell(a))\ ;\ \mathcal{C}(get(a)))$ is successful. As $\mathcal{C}(get(a))$ is composed of ask and tell primitives only and since ask and tell primitives do not destroy elements, it follows that any element of $\mathcal{O}(\mathcal{C}(tell(a))\ ;\ \mathcal{C}(get(a))\ ;\ \mathcal{C}(get(a)))$ has a successful termination mark. However, $\mathcal{O}(tell(a)\ ;\ get(a)\ ;\ get(a)) = \{(\emptyset, \delta^-)\}$ which contradicts property $P_3$.□

We observe that $\mathcal{L}$(get,tell) and $\mathcal{L}$(nask,tell) are not comparable with each other.

**Proposition 4.4**

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

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

**Proof.** (i): The proof proceeds as for proposition 4.2(ii). (ii): Otherwise, by proposition 4.3(i), $\mathcal{L}$(ask, tell) $\leq$ $\mathcal{L}$(nask, tell), which contradicts proposition 4.2(i). □

We now show that *ask* is redundant when the language contains *get* and *tell*.

**Proposition 4.5**

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

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

**Proof.** (i): Immediate since $\mathcal{L}$(get,tell) $\subseteq$ $\mathcal{L}$(ask,get,tell). (ii): Immediate by translating each get and tell primitive to itself and each $ask(t)$ primitive in $get(t)$ ; $tell(t)$. □

It is worth observing that while adding *ask* to $\mathcal{L}$(get,tell) does not increase the expressive power of the language, adding *nask* does.

**Proposition 4.6**

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

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

**Proof.** (i): Immediate since $\mathcal{L}$(get,tell) $\subseteq$ $\mathcal{L}$(nask,get,tell). (ii): Otherwise, it is possible to prove that any computation of $\mathcal{C}(tell(t))$ ; $\mathcal{C}(nask(t))$ starting in the empty store is successful, which contradicts by $P_3$ the fact that $\mathcal{O}(tell(t)\ ;\ nask(t)) = \{(\{t\}, \delta^-)\}$. Indeed, any computation of $\mathcal{C}(tell(t))$ starting on the empty store succeeds since $\mathcal{O}(tell(t)) = \{(\{t\}, \delta^+)\}$. Similarly, any computation of $\mathcal{C}(nask(t))$ starting on the empty store succeeds and consequently so does any general computation starting on any store since $\mathcal{C}(nask(t))$ is composed of get and tell primitives only. It follows that any computation of $\mathcal{C}(tell(t))$ starting on the empty store can be continued by (successful) computations of $\mathcal{C}(nask(t))$. □

Let us now consider the language $\mathcal{L}$(ask,nask,tell). We first observe that $\mathcal{L}$(ask,nask,tell) is strictly more expressive of both $\mathcal{L}$(nask,tell) and $\mathcal{L}$(ask,tell).

**Proposition 4.7**

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

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

(*iii*) $\mathcal{L}(ask,nask,tell) \not\leq \mathcal{L}(nask,tell)$

(*iv*) $\mathcal{L}(ask,nask,tell) \not\leq \mathcal{L}(ask,tell)$

**Proof.** (i) and (ii): Immediate since $\mathcal{L}$(nask,tell) $\subseteq$ $\mathcal{L}$(ask,nask,tell) and $\mathcal{L}$(ask,tell) $\subseteq$ $\mathcal{L}$(ask,nask,tell). (iii): Indeed if $\mathcal{L}$(ask,nask,tell) $\leq$ $\mathcal{L}$(nask,tell) then, by (ii) and by the transitivity of embedding, $\mathcal{L}$(ask,tell) $\leq$ $\mathcal{L}$(nask,tell) would hold, but this would contradict proposition 4.2(i). (iv): Similarly if $\mathcal{L}$(ask,nask,tell) $\leq$ $\mathcal{L}$(ask,tell) then, by (i) and by the transitivity of embedding, we would have that $\mathcal{L}$(nask,tell) $\leq$ $\mathcal{L}$(ask,tell), which contradicts proposition 4.2(ii). □

Moreover the languages $\mathcal{L}$(ask,nask,tell) and $\mathcal{L}$(get,tell) are incomparable.

**Proposition 4.8**

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

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

**Proof.** (i): Indeed if $\mathcal{L}$(ask,nask,tell) $\leq$ $\mathcal{L}$(get,tell) then, since $\mathcal{L}$(nask,tell) $\leq$ $\mathcal{L}$(ask,nask,tell), we would have that $\mathcal{L}$(nask,tell) $\leq$ $\mathcal{L}$(get,tell) by the transitivity of embedding, which contradicts proposition 4.4(i). (ii): Let us proceed by contradiction and assume that $\mathcal{L}$(get, tell) $\leq$ $\mathcal{L}$(ask, nask, tell). Then, as $\mathcal{O}(tell(t)\ ;\ get(t)) = \{(\emptyset, \delta^+)\}$ any computation of $A = \mathcal{C}(tell(t))\ ;\ \mathcal{C}(get(t))$

starting in the empty store is successful by $P_3$. Our claim is that, as a consequence, any computation of $B = \mathcal{C}(tell(t)) \ ; \ \mathcal{C}(get(t)) \ ; \ \mathcal{C}(get(t))$ starting in the empty store is successful, which contradicts, by $P_2$ and $P_3$, the fact that $\mathcal{O}(tell(t) \ ; \ get(t) \ ; \ get(t)) = \{(\emptyset, \delta^-)\}$. Indeed, assume $B$ has a failing computation from the empty store. This computation should start with a computation of $A$ which, as seen above is successful. Moreover, since $\mathcal{L}$(ask, nask, tell) contains no destructive operation, the contents of the store can only increase at each computation step. It follows that there are stores $\sigma$, $\tau$, $\gamma$ and an agent $C$ such that

$$\begin{aligned}
&\langle \mathcal{C}(tell(t)) \ ; \ \mathcal{C}(get(t)) \ ; \ \mathcal{C}(get(t)) \mid \emptyset \rangle \\
&\longrightarrow^* \langle \mathcal{C}(get(t)) \ ; \ \mathcal{C}(get(t)) \mid \sigma \rangle \\
&\longrightarrow^* \langle \mathcal{C}(get(t)) \mid \sigma \cup \tau \rangle \\
&\longrightarrow^* \langle C \mid \sigma \cup \tau \cup \gamma \rangle \not\rightarrow
\end{aligned}$$

Moreover, one may assume that $C$ is in normal form. Since $C$ is blocked, it cannot contain alternative lead by tell primitives and thus rewrites as $C = ask(u \ ) \ ; \ C \ + \cdots + ask(u_m) \ ; \ C_m + nask(v \ ) \ ; \ D \ + \cdots + nask(v_n) \ ; \ D_n$. Again, as $C$ is blocked then, for $i = 1, \cdots, m$, one has $u_i \notin \sigma \cup \tau \cup \gamma$ and for $j = 1, \cdots, n$, $v_j \in \sigma \cup \tau \cup \gamma$. As, $\mathcal{C}$ is compositional, the following computation steps are valid:

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

Moreover, as $u_i \notin \sigma \cup \sigma \cup \tau \cup \gamma$, for $i = 1, \cdots, m$, and $v_j \in \sigma \cup \sigma \cup \tau \cup \gamma$, for $j = 1, \cdots, n$, $\langle C \mid \sigma \cup \sigma \cup \tau \rangle \not\rightarrow$. Then $\mathcal{C}(tell(t)) \ ; \ \mathcal{C}(tell(t)) \ ; \ \mathcal{C}(get(t)) \ ; \ \mathcal{C}(get(t))$ has a failing computation starting in the empty store, which contradicts by $P_3$, the fact that $\mathcal{O}(tell(t) \ ; \ tell(t) \ ; \ get(t) \ ; \ get(t)) = \{(\emptyset, \delta^+)\}$. □

Let us now consider the language containing all primitives. We prove that *ask* is redundant in the full language.

**Proposition 4.9**

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

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

**Proof.** (i): Immediate since $\mathcal{L}$(nask,get,tell) $\subseteq$ $\mathcal{L}$(ask,nask,get,tell). (ii): To establish the second inequality, let us first code any token $t$ by a pair of tokens which we denote $(t \ ,t_2)$. This can be done because *Stoken* is denumerable: for instance, it suffices to associate the token associated with the integer $n$ to the tokens associated with the integers $2n$ and $2(n+1)$. Given such a coding of tokens, we define the coder $\mathcal{C}$ as follows.

$$\mathcal{C}(ask(t)) = get(t_2) \ ; \ tell(t_2)$$

$$\mathcal{C}(nask(t)) = nask(t_1)$$
$$\mathcal{C}(get(t)) = get(t_2)\ ;\ get(t_1)$$
$$\mathcal{C}(tell(t)) = tell(t_1)\ ;\ tell(t_2)$$

Moreover, the decoder $\mathcal{D}$ is defined as follows:

$$\mathcal{D}_{el}((\sigma, \delta)) = (\overline{\sigma}, \delta)$$

where $\overline{\sigma}$ is composed of the tokens $t$ for which $t_1$ and $t_2$ are in $\sigma$, the multiplicity of occurrences of $t$ being that of pairs $(t_1, t_2)$ in $\sigma$. To conclude, it remains to establish that $\mathcal{O}(A) = \mathcal{D}(\mathcal{O}(\mathcal{C}(A)))$, for any agent $A$ of $\mathcal{L}$(ask, nask, get, tell). The key point for this proof consists of first establishing that if, $A'$ denotes $\mathcal{C}(A)$, for any agent $A$, and, if $\sigma'$ denotes the store obtained by coding the tokens of $\sigma$, for any store $\sigma$, then $\langle A \mid \sigma \rangle \longrightarrow \langle B \mid \tau \rangle$ if and only if $\langle A' \mid \sigma' \rangle \longrightarrow^* \langle B' \mid \tau' \rangle$, for any agents $A$, $B$ and any stores $\sigma$, $\tau$. This in turn is proved by inductively reasoning on the structure of the agent $A$ and for parallelly composed agents by reasoning on their normal forms. □

Finally, we prove that $\mathcal{L}$(nask, get, tell) and $\mathcal{L}$(ask, nask, get, tell) are strictly more expressive than any other sublanguage.

**Proposition 4.10**

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

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

**Proof.** (i): Immediate since $\mathcal{L}$(ask,nask,tell) $\subseteq \mathcal{L}$(ask,nask,get,tell).
(ii): Indeed if $\mathcal{L}$(ask,nask,get,tell) $\leq \mathcal{L}$(ask,nask,tell) then, since $\mathcal{L}$(get,tell) $\leq$ $\mathcal{L}$(nask,get,tell) and since $\mathcal{L}$(nask,get,tell) $\leq \mathcal{L}$(ask,nask,get,tell) by proposition 4.9(i), we would have that $\mathcal{L}$(get,tell) $\leq \mathcal{L}$(ask,nask,tell) by the transitivity of embedding. But this contradicts proposition 4.8(ii). □

## 5 Concluding remarks

The notion of modular embedding was introduced and used by De Boer and Palamidessi in [3] to prove separation results for the class of concurrent logic languages, namely to prove that Flat CP cannot be embedded into Flat GHC, and that Flat GHC cannot be embedded into a language without communication primitives in the guards, while the converse results hold. De Boer and Palamidessi also showed in [4] a similar separation result for CSP with input guards w.r.t. full CSP. We have used this notion of modular embedding to compare the expressive power of a family of Linda-like concurrent languages. However, the difference of languages has required novel proofs and treatments, as will be appreciated by the reader.

The notion of modular embedding is not of course the only means for comparing the relative expressive power of formalisms. As we already mentioned, the notion of Turing-completeness is for instance a classical yard-stick for comparing the expressive power of sequential languages. In the field of

process algebras, for instance, Vaandrager [22] classified various notions of expressiveness, ranging from the capability of simulating Turing machines, to the capability of expressing effective operations on graphs. A comparison of different methodologies for comparing the relative expressive power of formalisms is however outside the scope of this paper, and can be found for instance in [3].

Busi, Gorrieri and Zavattaro have recently presented in [6] a process algebraic treatment of a family of Linda-like concurrent languages. Besides *ask*, *get* and *tell*, they consider two other primitives, *getp* and *askp* (using our naming), for conditional *get* and conditional *ask*. Intuitively speaking, the agent $getp(t)?P\_Q$ tests whether the tuple $t$ is present in the dataspace. If $t$ belongs to the dataspace then $t$ is removed and the agent chooses $P$ to continue; otherwise the agent directly evolves in $Q$. The definition of *askp* is analogous, with the difference that if $t$ belongs to the dataspace it is not removed. It is worth noting that the primitives *getp* and *askp* can be expressed in terms of the *nask* operation by rewriting $getp(t)?P\_Q$ as $(get(t);P)+(nask(t);Q)$, and $askp(t)?P\_Q$ as $(ask(t);P)+(nask(t);Q)$. Busi, Gorrieri and Zavattaro build in [6] a lattice of eight languages obtained by considering different sets of primitives and by taking as observational semantics the coarsest congruence contained in the barbed semantics. They also show that the lattice of eight languages collapses to a smaller four-points lattice of different bisimulation-based semantics. It is interesting to observe for instance that in [6] $\mathcal{L}$(get,tell) is distinguished from $\mathcal{L}$(ask,get,tell), as well as $\mathcal{L}$(getp,get,tell) is distinguished from $\mathcal{L}$(askp,getp,get,tell). While the point of view of [6] is different from ours, it would be very interesting to further investigate the relations between the two approaches. Such a future work may also provide some insights on the comparison between bisimulation and modular embedding as yard-sticks for measuring the expressive power of languages.

Busi, Gorrieri and Zavattaro also recently studied in [5] the issue of Turing-completeness in Linda-like concurrent languages. They defined a process algebra containing Linda's communication primitives and compared two possible semantics for the output operation (*tell*). They considered an *ordered* semantics for *tell*, where the operation returns only when the data have reached the dataspace, and an *unordered* semantics for *tell*, where the operation returns just after sending the insertion request to the dataspace. The main result presented in [5] is that the process algebra is not Turing-complete under the second interpretation of *tell*, while it is so under the first interpretation. The work [5] and ours are somehow orthogonal: While in [5] they studied the *absolute* expressive power of different variants of Linda-like languages (using Turing-completeness as a yard-stick), we studied the *relative* expressive power of different variants of such languages (using modular embedding as a yard-stick).

One might however argue that Turing-completeness is not probably the "right" yard-stick for measuring the (absolute) expressive power of coordi-

nation languages, like Linda-like languages. Indeed coordination languages [12] are intended to be used for expressing the interaction or coordination of computational components, rather than for expressing computations in the Turing-completeness sense. An intriguing question in this perspective might then be how to formally characterise the absolute expressive power of a coordination language.

Finally, it is worth observing that the so-called *generative* communication of Linda bears strong similarities with other well-known interaction models, like the blackboard model of problem solving in A.I. [17], the GAMMA coordination model [1], or the concurrent constraint paradigm [18]. All these models rely on a notion of abstract shared store via which concurrent agents synchronise and communicate by performing forms of *ask*, *tell*, and *get* operations. We plan to devote future work to better analyse the relation between these models so as to try to get a clearer understanding of the relations between different coordination models based on (multi-)set rewriting operations. In this direction, ongoing work by Zavattaro [21] is aimed at showing that the expressive powers of Gamma and Linda are not comparable with each other, as there seems to be no encoding of one language into the other that preserves program distribution properties.

Another interesting direction for future work is to investigate the expressiveness of Linda-based languages that have been recently proposed by De Nicola, Ferrari and Pugliese [9,10] for programming agents capable of migrating across different computing environments.

## References

[1] J. Banatre and D. LeMetayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111. 1991.

[2] 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.

[3] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison. *Information and Computation*, 108(1):128-157, 1994.

[4] F.S. de Boer and C. Palamidessi. Embedding as a tool for language comparison: On the CSP hierarchy. In J.C.M. Baeten and J.F. Groote, editors, *Proc. of CONCUR 91*, volume 527 of Lecture Notes in Computer Science, pages 127-141. Springer-Verlag, 1991.

[5] N. Busi, R. Gorrieri and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In C. Palamidessi and J. Parrow (editors) *EXPRESS 97: Proceedings of the 4th workshop on Expressiveness in Concurrency.* Volume 7 of Electronic Notes in Theoretical Computer Science, 1997.

[6] N. Busi, R. Gorrieri and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[7] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[8] 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.

[9] R. De Nicola, G. Ferrari and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In D. Garlan and D. Le Metayer (editors). *Coordination'97: Proceedings of The Second International Conference on Coordination Models and Languages*, pages 220—237. LNCS. Springer-Verlag, 1997.

[10] R. De Nicola, G. Ferrari and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. To appear in *IEEE Trans. on Software Engineering*, Special Issue on Mobility and Network Aware Computing (Catalin Roman and Ghezzi Eds), 1998.

[11] D. Garlan and D. Le Metayer (editors). Coordination'97: Proceedings of The Second International Conference on Coordination Models and Languages. LNCS. Springer-Verlag, 1997.

[12] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[13] M. Felleisen. On the expressive power of programming languages. In N. Jones (editor) "Proceedings European Symposium on Programming", LNCS 432, pages 134–151. Springer-Verlag, 1990.

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

[15] R. Milner. *A Calculus of communicating systems.* Springer-Verlag, 1989.

[16] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, 1993.

[17] H. P. Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architecture, Part 1. *AI Magazine*, 7:2, pages 38–53, 1986.

[18] V.A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, 1993.

[19] E.Y. Shapiro. Embeddings among concurrent programming languages. In W.R. Cleaveland (editor), "Proceedings of CONCUR'92", LNCS 630, pages 486–5-3. Springer-Verlag, 1992.

[20] P. Wegner. Why Interaction Is More Powerful Than Algorithms. *Communications of the ACM*, May 1997.

[21] G. Zavattaro. On the incomparability of Gamma and Linda. Personal communication, March 1998.

[22] F. Vaandrager. Expressiveness results for process algebras. In "Proceedings REX workshop on 'Semantics: Foundations and applications"', LNCS. Springer-Verlag, 1992.

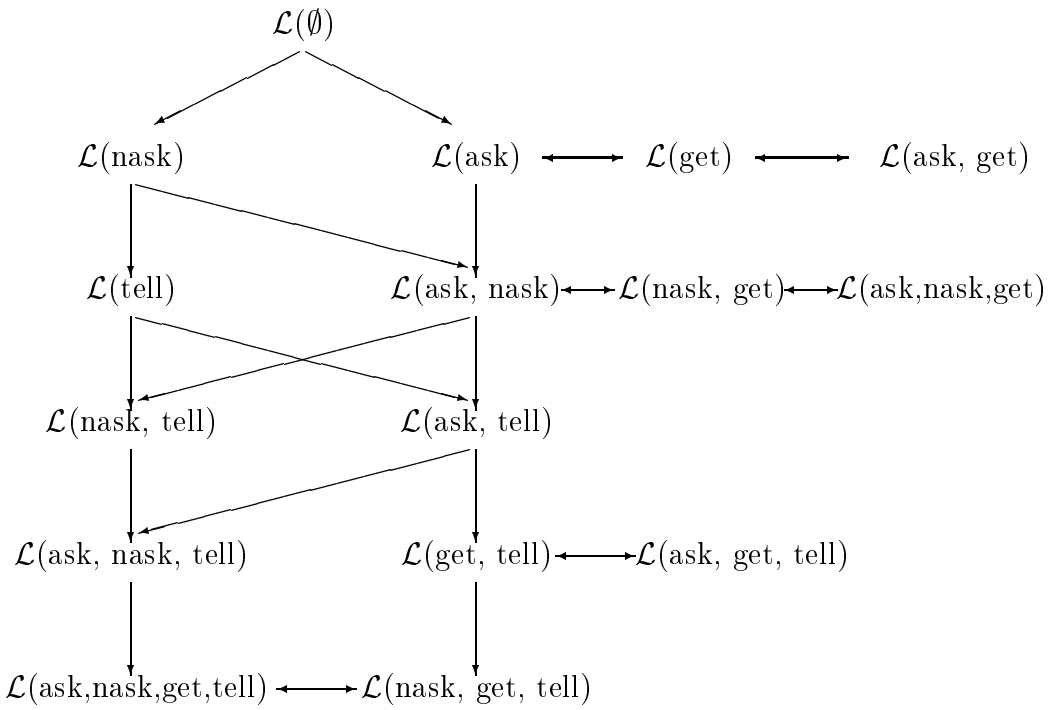


Fig. 4. The complete hierarchy of languages.

## Appendix

For the sake of completeness, we report here also the analysis of the languages in the class not containing *tell*. Figure 4 summarises the complete set of results for the whole class of languages. The presented results are substantiated by the following propositions.

**Proposition 5.1**

$(i)$ $\mathcal{L}(\emptyset) \leq \mathcal{L}(nask)$

$(ii)$ $\mathcal{L}(\emptyset) \leq \mathcal{L}(ask)$

$(iii)$ $\mathcal{L}(nask) \not\leq \mathcal{L}(\emptyset)$

$(iv)$ $\mathcal{L}(ask) \not\leq \mathcal{L}(\emptyset)$

**Proof.** (i) and (ii): Immediate since $\mathcal{L}(\emptyset) \subseteq \mathcal{L}$(nask) and $\mathcal{L}(\emptyset) \subseteq \mathcal{L}$(ask).
(iii) and (iv): Descend from the fact that, by $P$ , $\mathcal{D}$ is defined elementwise. □

**Proposition 5.2**

$(i)$ $\mathcal{L}(ask) \not\leq \mathcal{L}(nask)$

$(ii)$ $\mathcal{L}(nask) \not\leq \mathcal{L}(ask)$

**Proof.** Indeed, by property $P_3$, termination marks should be preserved. However, $\mathcal{O}(A) = \{(\emptyset, \delta^+)\}$ for any agent $A$ of $\mathcal{L}$(nask) and $\mathcal{O}(B) = \{(\emptyset, \delta^-)\}$ for any agent $B$ of $\mathcal{L}$(ask). □

**Proposition 5.3**

$(i)$ $\mathcal{L}(nask) \leq \mathcal{L}(tell)$

$(ii)$ $\mathcal{L}(tell) \not\leq \mathcal{L}(nask)$

**Proof.** (i): Indeed, it is sufficient to code any primitive $nask(t)$ as $tell(t)$ and to use as decoder the function which preserves termination marks and which transforms any store in the empty store. (ii): Assume by contradiction that $\mathcal{L}$(tell) $\leq \mathcal{L}$(nask). Consider $tell(a)$ and $tell(b)$. Obviously $\mathcal{O}(tell(a)) = \{(\{a\}, \delta^+)\}$ and $\mathcal{O}(tell(b)) = \{(\{b\}, \delta^+)\}$. Moreover, since $\mathcal{C}(tell(a))$ and $\mathcal{C}(tell(b))$ are composed of $nask$ primitives only, $\mathcal{O}(\mathcal{C}(tell(a))) = \{(\emptyset, \delta^+)\}$ and $\mathcal{O}(\mathcal{C}(tell(b))) = \{(\emptyset, \delta^+)\}$. Therefore, since $\mathcal{O}(tell(a)) = \mathcal{D}(\mathcal{O}(\mathcal{C}(tell(a))))$ and $\mathcal{O}(tell(b)) = \mathcal{D}(\mathcal{O}(\mathcal{C}(tell(b))))$, we have that $\mathcal{D}((\emptyset, \delta^+)) = (\{a\}, \delta^+)$ and $\mathcal{D}((\emptyset, \delta^+)) = (\{b\}, \delta^+)$ which contradicts the fact that $\mathcal{D}$ is a function. □

**Proposition 5.4**

$(i)$ $\mathcal{L}(ask) \not\leq \mathcal{L}(tell)$

$(ii)$ $\mathcal{L}(tell) \not\leq \mathcal{L}(ask)$

**Proof.** Indeed, by property $P_3$, termination marks should be preserved. However, $\mathcal{O}(A) = \{(\emptyset, \delta^+)\}$ for any agent $A$ of $\mathcal{L}$(tell) and $\mathcal{O}(B) = \{(\emptyset, \delta^-)\}$ for any agent $B$ of $\mathcal{L}$(ask). □

**Proposition 5.5**

$(i)$ $\mathcal{L}(ask) \leq \mathcal{L}(get)$

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

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

**Proof.** Indeed, since for any token $t$, $\mathcal{O}(ask(t)) = \{(\emptyset, \delta^-)\} = \mathcal{O}(get(t))$, it is sufficient to code any $ask(t)$ primitive in the $get(t)$ primitive and vice-versa and to use the identity as decoding function. □

**Proposition 5.6**

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

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

$(iii)$ $\mathcal{L}(ask,nask) \not\leq \mathcal{L}(nask)$

$(iv)$ $\mathcal{L}(ask,nask) \not\leq \mathcal{L}(ask)$

**Proof.** (i) and (ii): Immediate since $\mathcal{L}$(nask) $\subseteq \mathcal{L}$(ask,nask) and $\mathcal{L}$(ask) $\subseteq \mathcal{L}$(ask,nask). (iii): By contradiction, if $\mathcal{L}$(ask,nask) $\leq \mathcal{L}$(nask) then by (ii) and by transitivity of embedding we would have that $\mathcal{L}$(ask) $\leq \mathcal{L}$(nask), which contradicts proposition 5.2(i). (iv): By contradiction, if $\mathcal{L}$(ask,nask) $\leq \mathcal{L}$(ask) then by (i) and by transitivity of embedding we would have that $\mathcal{L}$(nask) $\leq \mathcal{L}$(ask), which contradicts proposition 5.2(ii). □

**Proposition 5.7**

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

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

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

**Proof.** Indeed, as for proposition 5.5, it suffices to code any $ask(t)$ into $get(t)$ and vice-versa. □

**Proposition 5.8**

$(i)$ $\mathcal{L}(tell) \not\leq \mathcal{L}(ask,nask)$

$(ii)$ $\mathcal{L}(ask,nask) \not\leq \mathcal{L}(tell)$

**Proof.** (i): The proof proceeds as for proposition 5.3(ii) but by stating that $\mathcal{O}(\mathcal{C}(tell(x))) = \{(\emptyset, \delta^+)\}$ because, on the one hand, $ask$ and $nask$ primitives cannot update the initial store $\emptyset$ and, on the other hand, termination marks are preserved by $\mathcal{D}$. (ii): Indeed, otherwise, $\mathcal{L}$(ask) $\leq \mathcal{L}$(tell), which contradicts proposition 5.4(i). □

**Proposition 5.9**

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

$(ii)$ $\mathcal{L}(nask,tell) \not\leq \mathcal{L}(ask,nask)$

**Proof.** (i): Indeed, since ask and nask primitives do not update the store, $ask$ and $nask$ primitives may be coded as follows:

$$\mathcal{C}(ask(t)) = tell(t)\ ;\ nask(t)$$
$$\mathcal{C}(nask(t)) = nask(t)$$

for any token $t$. The decoding function to be used is then defined by

$$\mathcal{D}_{el}((\sigma, \delta)) = (\emptyset, \delta).$$

(ii): Indeed, otherwise since $\mathcal{L}$(tell) $\leq \mathcal{L}$(nask,tell) then $\mathcal{L}$(tell) $\leq \mathcal{L}$(ask, nask) which contradicts proposition 5.8(i). □

**Proposition 5.10**

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

$(ii)$ $\mathcal{L}(ask,tell) \not\leq \mathcal{L}(ask,nask)$

**Proof.** (i): Let us first observe that, no $A \in \mathcal{L}$(ask, nask) updates the initial store $\emptyset$. Moreover, in these conditions, any $ask$ primitive always fails whereas any $nask$ primitive always succeeds. Since the set of tokens is denumerable, let us code any token $t$ by its successor, say $t'$, with respect to some numbering function. Furthermore, let $t_0$ be the first token in this numbering. The coder $\mathcal{C}$ is then defined as follows:

$$\mathcal{C}(ask(t)) = ask(t')$$
$$\mathcal{C}(nask(t)) = tell(t_0)$$

The decoder is the function which preserves termination marks and which transforms any store into the empty store. (ii): Assume by contradiction that $\mathcal{L}(\text{ask, tell}) \leq \mathcal{L}(\text{ask, nask})$. Consider $tell(a)$ and $tell(b)$. Obviously $\mathcal{O}(tell(a)) = \{(\{a\}, \delta^+)\}$ and $\mathcal{O}(tell(b)) = \{(\{b\}, \delta^+)\}$. Since $\mathcal{C}(tell(a))$ and $\mathcal{C}(tell(b))$ are composed of $ask$ and $nask$ primitives only and since the decoder should preserve termination marks by property $P_3$, $\mathcal{O}(\mathcal{C}(tell(a))) = \{(\emptyset, \delta^+)\}$ and $\mathcal{O}(\mathcal{C}(tell(b))) = \{(\emptyset, \delta^+)\}$. Therefore, since $\mathcal{O}(tell(a)) = \mathcal{D}(\mathcal{O}(\mathcal{C}(tell(a))))$ and $\mathcal{O}(tell(b)) = \mathcal{D}(\mathcal{O}(\mathcal{C}(tell(b))))$, we have that $\mathcal{D}((\emptyset, \delta^+)) = (\{a\}, \delta^+)$ and $\mathcal{D}((\emptyset, \delta^+)) = (\{b\}, \delta^+)$ which contradicts the fact that $\mathcal{D}$ is a function. □