Locality Based Linda: Programming with Explicit Localities*

Rocco De Nicola[1] GianLuigi Ferrari[2] Rosario Pugliese[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze
e-mail: {denicola,pugliese}@dsi2.ing.unifi.it

[2]Dipartimento di Informatica, Università di Pisa
e-mail: giangi@di.unipi.it

Abstract. In this paper we investigate the issue of defining a programming calculus which supports programming with explicit localities. We introduce a language which embeds the asynchronous Linda communication paradigm extended with explicit localities in a process calculus. We consider multiple tuple spaces that are distributed over a collections of sites and use localities to distribute/retrieve tuples and processes over/from these sites. The operational semantics of the language turns out to be useful for discussing the language design, e.g. the effects of scoping disciplines over mobile agents which maintain their connections to the located tuple spaces while moving along sites. The flexibility of the language is illustrated by a few examples.

1 Introduction

The World–Wide Web (WWW) is the best known example of an application geographically distributed over a collections of processors and networks. Recently, the names of *global information structures* and *global computers* have been used to identify such applications [8] and their underlying architecture. Another example of global information structure is given by the `Telnet` protocol which provides a global multiprocessor system.

Global structures are rapidly evolving towards programmability. Again, an illustrative example is provided by the WWW. One could easily imagine applications where programs running at different sites need continuous interactions or applications where decisions are taking according to information retrieved from the global environment. This has called for *new* programming languages and paradigms that supports migratory (mobile) applications. As an example the Java language [16] permits local executions of self–contained programs downloaded from other sites. Similarly, the Facile language [14] supports mobility of programs by allowing processes to be transmitted in communications. Obliq [7]

* Work partially supported by EEC: HCM project EXPRESS and Esprit Working Group *CONFER2*, and by CNR: Progetto Speciale "Modelli e Metodi per la Matematica e l'Ingegneria".

is an example of a programming language with a static scoping discipline where mobile processes maintain their connections when they move from one site to the other.

¿From a theoretical perspective, several research efforts have been devoted to address mobility starting from the definition of the π–calculus [19], that has been used as a design tool for the development of the concurrent object oriented programming language PICT [21]. Indeed, an abstract semantic framework to formalize and understand how global programming languages operate is clearly required. Such semantic framework may provide the formal basis to discuss and motivate controversial design/implementation issues (e.g. the scoping discipline of mobile processes) and the support for reasoning about global programs.

In global programming one has to face the problem of developing applications which need to access data or computational–resources distributed over a set of sites. A simple example is provided by "distribute and print" applications where, after a request, a server spawns a print job over certain sites (the sites where printed data are needed) and delegates the control of the actual printing activities to each site.

In this paper we investigate the issue of defining a programming notation which *directly* supports programming with explicit localities. We concentrate on the formal definition of the core language to clarify the critical design decisions. Simulation and prototyping activities based on the formal definition are in progress.

Our proposal embeds the Linda paradigm [12, 9] extended with an explicit notion of locality within a CCS–like [18] process calculus. The new language will be named Locality Linda, `LLinda` for short. The Linda asynchronous communication model, known as *Generative Communication*, allows programmers to explicit control interactions among processes via shared data and to use the same set of primitives both for data manipulation and for process synchronization. This has the advantage of rendering explicit all the interactions of a program with its environment. The original Linda primitives are however not completely adequate for programming distributed systems. For example, data protection and security, that are key features of a distributed programming environment, are problematic because the Linda communication model cannot guarantee data privacy. Also, modular programming disciplines are awkward to follow in practice as there is no mean to guarantee that tuples coming from different contexts are not mixed up when two modules are put together. Multiple tuple spaces [13] are a first step toward the solution of these problems. `LLinda`, that takes multiple tuple spaces as the starting point, can be seen as the formalization of that idea, that had never thoroughly pursued.

`LLinda` can be seen as an asynchronous value–passing process calculus whose basic actions are the original Linda primitives enriched with explicit information about the location of the nodes where processes and tuples are allocated. This allows programmers to distribute (retrieve) data and processes over (from) different nodes directly from the language. Localities permit splitting the tuple space into multiple, located spaces and to view groups of processes and their

data as distinct entities. Moreover, since localities are treated as first–order data, that can be exchanged in communications and dynamically created, they become a powerful programming device. For example, encapsulation can be easily obtained: an encapsulated module can be realized via a tuple space located at a private locality, thus ensuring a controlled access to data. However, programmers have to share with a coordinator their control. This sharing is obtained by providing abstractions over geographical distribution and by separating the operation of logical distribution of processes and data from the mechanism which maps a logically distributed program into a physically distributed application. To this purpose a new class of values to represent logical distribution is introduced. *Logical localities* provide an abstraction mechanism that allows programmers to structure mobile agents by controlling the location of computation while ignoring the precise allocation of processes and data.

The handling of logical localities — the mapping on processors and nets, the visibility of specific localities from each node — is done at another level; we refer to it as the *coordination* level. When applications migrate, all the issues related to the scope discipline are dealt with at the coordination level; this is somehow in the spirit of [3].

The two syntactic levels of our programming framework are reflected at the semantic level. The operational semantics of `LLinda` follows the SOS style [22] and proceeds in two steps. The first step defines the *symbolic semantics* where process commitments, i.e. the control on localities and the effects on the tuple spaces, are only partially evaluated. The full evaluation of process commitments is the main concern of the second step, the one at the coordination level.

In this paper we show that the separation between logical and physical localities is a clean abstraction for global programming languages. Moreover, the coordination level turns out to be essential to study migratory applications and to understand configuration decisions before carrying out an implementation. This will be illustrated in the present paper by analyzing the effects of choosing specific scoping disciplines for accessing tuple spaces. The usage of the language is illustrated by presenting some examples of distribution and mobility.

2 LLinda

The language `LLinda` is an attempt to amalgamate the Linda paradigm [12, 9] with an explicit notion of *locality* to support a programming paradigm where applications can migrate from one computing environment to another.

2.1 A Linda Outline

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called Tuple Space (TS), a multiset of tuples. A tuple is a sequence of actual fields, i.e. expressions or values, and formal fields, i.e. variables. *Pattern–matching* is used to select tuples in TS: two tuples match if they have the same number of fields

and corresponding fields have matching values or variables; variables match any value of the same type and two values match only if identical. Linda has just four primitives for manipulating tuples. Two (non–blocking) operations, **out**(t) and **eval**(t), permit to add tuples to TS. The operation **out**(t) adds the tuple resulting from the evaluation of t to TS. The operation **eval**(t) differs from **out**(t) because t is firstly added to TS and then a new concurrent process is created for evaluating the tuple; this will not be available for matching until its evaluation is completed. Two (possibly blocking) operations, **in**(t) and **read**(t), permit accessing tuples in TS. The operation **in**(t) evaluates t and looks for a matching tuple t' in TS. Whenever t' is found, it is removed from TS; then, the corresponding values of t' are assigned to the variables of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The operation **read**(t) differs from **in**(t) because the tuple t' selected by pattern–matching is not withdrawn from TS.

In the original proposal of [12] two predicative (non–blocking) forms, **inp** and **readp**, were part of the language. We do not include them in LLinda because they appear to us as functional duplicates of their non–predicative counterparts and as statements about the global state of a distributed program, thus requiring expensive global synchronizations (see [17]).

2.2 Syntax

The LLinda language consists of a core Linda language with multiple tuple spaces, where tuple spaces and operations over tuples are located, and of a set of operators, borrowed from Milner's CCS [18], for building processes.

Localities are taken from a set *Loc* of localities. A locality ℓ can be considered as the address of the node where processes and tuple spaces are allocated. To provide an abstraction mechanism that allows to structure programs over a distributed environment while hiding the precise allocation of processes and data, also the set $\underline{Loc}$ of *logical localities* is introduced. A logical locality may be thought of as the symbolic name or alias for a physical site. We assume a distinguished logical locality `self` (`self`$\in \underline{Loc}$), that processes may use for denoting the physical locality at which they are executed. We shall use l to range over logical localities.

An assignment of logical localities is a (partial) function γ from $\underline{Loc}$ to Loc. In what follows Γ will denote the set of assignments, ϕ the empty assignment and $[\ell/l]$ the assignment which maps the logical locality l to ℓ. Finally, if $\gamma_1, \gamma_2 \in \Gamma$, we will use the notation $\gamma_1 \bullet \gamma_2$ for the function defined by:

$$\gamma_1 \bullet \gamma_2\,(l) = \begin{cases} \gamma_1(l) & \text{if } l \in dom(\gamma_1) \\ \gamma_2(l) & \text{otherwise} \end{cases}$$

One of the syntactic categories of LLinda is that of *expressions*. We assume existence of a set of variable symbols, Var, whose typical elements are $x, y, \cdots$, and a non–empty countable set of basic values $v \in Val$, together with a set of operators. This yields Exp, ranged over by e, the category of *value expressions*.

Furthermore, we assume existence of the syntactic category *LExp* (ranged over by le) of *locality expressions* built out of locality variables (ranged over by u) and operators over them, that will not be explicited here. We also assume a set of *process variables* (ranged over by X) and a set of *process constants* (ranged over by A), each with a fixed *arity*. We will use z for denoting a value variable or a locality variable and w for denoting a value or a logical locality.

Substitution works as expected and we will use the standard notation $e[e'/x]$ to indicate the substitution of the value expression e' for the variable x in e. A similar notation will be adopted for denoting the substitution of (actual) parameters and that of data tuples inside processes.

The LLinda process expressions (terms) are given by the abstract syntax below:

$$
\begin{array}{lll}
P & ::= & \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid P_1{+}P_2 \mid A(z_1,\ldots,z_n) \\
a & ::= & \mathbf{out}(t)@le \mid \mathbf{in}(t)@le \mid \mathbf{read}(t)@le \mid \mathbf{eval}(P)@le \mid \mathbf{newloc}(u) \\
t & ::= & e \mid le \mid P \mid !\,x \mid !\,u \mid !\,X \mid t_1, t_2
\end{array}
$$

The basic operators for building processes are **nil** (*inaction*), $a.P$ (*prefixing*), $P_1|P_2$ (*parallel composition*) and $P_1{+}P_2$ (*choice*). **nil** stands for the process that cannot perform any action. $a.P$ denotes the process that first executes action a and then behaves like P. $P_1|P_2$ denotes the parallel composition of P_1 and P_2. Finally, $P_1{+}P_2$ denotes the nondeterministic composition of P_1 and P_2.

The Linda operations to generate tuples (**out**), to spawn a new process (**eval**), to read tuples (**read**), and to remove tuples (**in**) are located. Hence, LLinda permits multiple, distributed tuple spaces, accessible via the evaluation of locality expressions. We have a modified **eval** primitive that permits processes as arguments rather than tuples. As it will be clarified later, action $\mathbf{eval}(\mathbf{out}(t)@le.\mathbf{nil})@le$ can be used to simulate the "expected" behaviour of action $\mathbf{eval}(t)@le$. New physical localities are created through the prefix $\mathbf{newloc}(u)$. This operation creates a fresh physical locality that can be accessed via locality variable u. We shall assume that locations are garbage collected, and thus that no explicit deletion is necessary.

Variables occurring in a LLinda process expression can be bound by prefixes. More precisely, prefixes $\mathbf{in}(t)@le.\_$ and $\mathbf{read}(t)@le.\_$ act as binders for variables in the formal fields of t. Formal fields of tuples are denoted by "$!\,var$" where var is a generic variable. Prefix $\mathbf{newloc}(u).\_$ binds the locality variable u.

Process constants are used in recursive process definitions, and it is assumed that each process constant A with arity n has a *single* defining equation $A(z_1,\ldots,z_n) \stackrel{def}{=} P$, where all free (value and locality) variables in P are contained in $\{z_1,\ldots,z_n\}$ and all occurrences of process constants in P are *guarded* (i.e. each occurrence is within the scope of a prefix $a.\_$).

A *process* is a process expression without free variables. Observe that processes and localities are first-class data and can be manipulated and generated as any other data occurring in tuples.

To simplify notation, in the following, we often shall write a instead of $a.\mathbf{nil}$, moreover we shall use $\equiv$ for denoting syntactic identity of terms.

2.3 The Symbolic Semantics

We shall present the two–level operational semantics of LLinda in the SOS style [22]. The first level consists of the definition of a *symbolic semantics*. The second one packages processes and data over a distributed environment.

The labelled transition system of the symbolic semantics describes abstractly the possible evolutions of LLinda processes without providing the actual allocation of processes and tuple spaces. For this reason, the corresponding operational semantics is called *symbolic* in that neither value and locality expressions nor tuples are evaluated. Our use of allocation environments as part of the labels of transitions is similar to the use of boolean expressions in the operational framework of [15].

To describe the effects of processes over the different localities, we introduce the auxiliary process expression $P\{\gamma\}$ that indicates the process P packaged with the allocation of logical localities specified by γ. Intuitively, the mapping γ is a sort of environment and $P\{\gamma\}$ is a *closure*. For the sake of simplicity we will use P to range also over closures.

The structural rules of the symbolic semantics are displayed in Table 1. The *transition*

$$P \xrightarrow[\gamma]{\mu} P'$$

describes the evolution of a process. Labels of transitions are pairs $\langle \mu, \gamma \rangle$ which provide an abstract description of the activities performed in process evolution. For instance, $\mu = o(t)@le$ describes the output of tuple t in the tuple space specified by le. Similarly, $\mu = \nu(u)@\mathtt{self}$ can be thought of as the request of binding a fresh locality to the variable u. The function γ records the local allocation environment that must be used for evaluating μ. The interpretation of the structural rules of Table 1 is straightforward. We have already remarked that these rules do not evaluate expressions and location expressions, they only describe symbolic evaluation of processes. There is one exception, namely the rule for process closures $P\{\gamma\}$; in this case the evaluation of process P is determined also by the allocation requirements specified by γ.

2.4 The Coordination Level

As in [10, 23], we model tuples as processes but find it convenient to introduce a new process for denoting evaluated tuples that have been placed in one of the tuple spaces. Thus, we extend the syntax of processes with the construct $\mathbf{out}(et)$ (that is different from the prefix operator $\mathbf{out}(t)@le.\_$), where the set of *evaluated tuples* is generated by the following syntax:

$$et ::= v \mid \ell \mid P \mid !\,x \mid !\,u \mid !\,X \mid et_1, et_2$$

The structural rule of the symbolic semantic of process $\mathbf{out}(et)$ is

$$\mathbf{out}(et) \xrightarrow[\phi]{o(et)@\mathtt{self}} \mathbf{nil}$$

$$\mathbf{out}(t)@le.P \xrightarrow{s(t)@le} P \qquad \mathbf{eval}(Q)@le.P \xrightarrow{e(Q)@le} P$$

$$\mathbf{in}(t)@le.P \xrightarrow{i(t)@le} P \qquad \mathbf{read}(t)@le.P \xrightarrow{r(t)@le} P$$

$$\mathbf{newloc}(u).P \xrightarrow{\nu(u)@\mathtt{self}} P$$

$$\frac{P \xrightarrow[\gamma]{\mu} P'}{P+Q \xrightarrow[\gamma]{\mu} P'} \qquad \frac{P \xrightarrow[\gamma]{\mu} P'}{Q+P \xrightarrow[\gamma]{\mu} P'}$$

$$\frac{P \xrightarrow[\gamma]{\mu} P'}{P \mid Q \xrightarrow[\gamma]{\mu} P' \mid Q} \qquad \frac{P \xrightarrow[\gamma]{\mu} P'}{Q \mid P \xrightarrow[\gamma]{\mu} Q \mid P'}$$

$$\frac{P \xrightarrow[\gamma']{\mu} P'}{P\{\gamma\} \xrightarrow[\gamma' \bullet \gamma]{\mu} P'\{\gamma\}} \qquad \frac{P[w_1/z_1, \ldots, w_n/z_n] \xrightarrow[\gamma]{\mu} P'}{A(w_1, \ldots, w_n) \xrightarrow[\gamma]{\mu} P'} \text{ if } A(z_1, \ldots, z_n) \stackrel{def}{=} P$$

Table 1. The Structural Rules of Symbolic Semantics

In the semantics rules, we shall permit using physical localities alike locality expressions within processes. Thus, the operational semantics is defined for terms generated by this extended syntax.

Given a finite set of physical localities, a *net* of processes with multiple, distributed tuple spaces is a map that associates a *node* to each physical locality. A node is a pair (P, γ) where P encompasses both processes and the local tuple space, and γ is the local allocation environment. $\mathcal{S}$ will be used to indicate the set of nodes.

Let L be a finite subset of Loc; a *net over* L is a map $N_L : Loc \to \mathcal{S}$ such that

- $N_L(\ell)$ is defined if and only if $\ell \in L$,
- $N_L(\ell) = (P, \gamma)$ implies $range(\gamma) \subseteq L$ and $\gamma(\mathtt{self}) = \ell$.

A net provides a mechanism for coordinating the allocation of processes which interacts via multiple tuple spaces distributed over the localities of L. Processes at each locality can potentially access any other locality of the net; however locality visibility is controlled (*locally*) by the local allocation environment. A locality ℓ is *visible* at the node (P, γ) only if $\ell \in range(\gamma)$.

The operational semantics of nets makes use of evaluation mechanisms for value and locality expressions. We let them be the evaluation functions below, that are defined in the obvious way.

$$\mathcal{E}[\![\cdot]\!] : Exp \longrightarrow \Gamma \longrightarrow Val \qquad \mathcal{L}[\![\cdot]\!] : LExp \longrightarrow \Gamma \longrightarrow Loc$$

We will use $\mathcal{E}[\![e]\!]\gamma$ and $\mathcal{L}[\![le]\!]\gamma$ for denoting the value of the expression e and of the locality expression le when evaluated in γ (we implicitly assume that they have no variables). Similarly, the evaluation of tuples depends on the allocation environment: $\mathcal{T}[\![t]\!]\gamma$ is the tuple obtained by evaluating the tuple t in the allocation environment γ. The mapping $\mathcal{T}[\![\cdot]\!]$ is inductively defined over the definition of tuples. There is only one non-trivial case, namely the evaluation of a process, say $\mathcal{T}[\![P]\!]\gamma$, which yields a process closure, i.e. $P\{\gamma\}$. Finally, the pattern matching predicate is defined in Table 2.

$match(v, v)$	$match(\ell, \ell)$
$match(P, P)$	$match(!\, x, v)$
$match(!\, u, \ell)$	$match(!\, X, P)$
$match(et_1, et_2)$	$match(et_1, et_2) \quad match(et_3, et_4)$
$match(et_2, et_1)$	$match((et_1, et_3), (et_2, et_4))$

Table 2. The Matching Rules

The operational semantics of nets is presented in Table 3. Each node in a net has a unique physical locality, thus we can consider a net just as a set. We write $\ell ::_\gamma P$ for an element of a net, and $N_L, \ell ::_\gamma P$ for the net given by $N_L \cup \{\ell ::_\gamma P\}$ (with the implicit side condition that $\ell \notin L$). Basically, a node can be thought of as a *located process* in the style of [20]. The structural rules of the operational semantics specify the outcome of both local and remote operations performed by located processes. Thus, for each Linda primitive, we have two structural rules.

The evaluation of an **out** operation modifies a tuple space. Rule (1) adds a new tuple to the local tuple space of the process. Rule (2), instead, adds a new tuple to the remote tuple space located at ℓ_2. Notice that in the latter rule, the evaluation of the tuple t depends on the allocation environment $\gamma \bullet \gamma_1$. This corresponds to having a *static scope* discipline for the remote generation of tuples. Moreover, if the tuple t contains a field with a process, the corresponding field of the evaluated tuple et contains a closure. Hence, processes in a tuple are transmitted together with their local allocation environment.

A *dynamic scoping* strategy is adopted for the **eval** operation, described by rules (3) and (4). In this case the process spawned in the remote node is transmitted *without* the local allocation environment, and its execution is influenced by the remote allocation environment γ_2.

For the communication operations **in** and **read** we have to spell out that **in** modifies the tuple space (see rules (5) and (6)) while **read** does not (in the conclusions of rules (7) and (8) the tuple space encompassed within process P_2 is left unchanged by process evolution). Obviously, we have to distinguish between local, rules (5) and (7), and remote, rules (6) and (8), accesses.

Let us consider rule (5) (rules (6), (7) and (8) can be interpreted similarly).

$$\frac{P \xrightarrow[\gamma']{s(t)@le} P' \quad \ell = \mathcal{L}[\![le]\!]_{\gamma' \bullet \gamma} \quad et = \mathcal{T}[\![t]\!]_{\gamma' \bullet \gamma}}{N_L, \ell ::_\gamma P \rightarrowtail N_L, \ell ::_\gamma P' \mid \mathbf{out}(et)} \tag{1}$$

$$\frac{P_1 \xrightarrow[\gamma]{s(t)@le} P_1' \quad \ell_2 = \mathcal{L}[\![le]\!]_{\gamma \bullet \gamma_1} \quad et = \mathcal{T}[\![t]\!]_{\gamma \bullet \gamma_1}}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1', \ell_2 ::_{\gamma_2} P_2 \mid \mathbf{out}(et)} \tag{2}$$

$$\frac{P \xrightarrow[\gamma']{e(Q)@le} P' \quad \ell = \mathcal{L}[\![le]\!]_{\gamma' \bullet \gamma}}{N_L, \ell ::_\gamma P \rightarrowtail N_L, \ell ::_\gamma Q \mid P'} \tag{3}$$

$$\frac{P_1 \xrightarrow[\gamma]{e(Q)@le} P_1' \quad \ell_2 = \mathcal{L}[\![le]\!]_{\gamma \bullet \gamma_1}}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1', \ell_2 ::_{\gamma_2} Q \mid P_2} \tag{4}$$

$$\frac{P_1 \xrightarrow[\gamma']{i(t)@le} P_1' \quad \ell = \mathcal{L}[\![le]\!]_{\gamma' \bullet \gamma} \quad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \quad match(\mathcal{T}[\![t]\!]_{\gamma' \bullet \gamma}, et)}{N_L, \ell ::_\gamma P_1 | P_2 \rightarrowtail N_L, \ell ::_\gamma P_1'[et/\mathcal{T}[\![t]\!]_{\gamma' \bullet \gamma}] | P_2'} \tag{5}$$

$$\frac{P_1 \xrightarrow[\gamma]{i(t)@le} P_1' \quad \ell_2 = \mathcal{L}[\![le]\!]_{\gamma \bullet \gamma_1} \quad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \quad match(\mathcal{T}[\![t]\!]_{\gamma \bullet \gamma_1}, et)}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1'[et/\mathcal{T}[\![t]\!]_{\gamma \bullet \gamma_1}], \ell_2 ::_{\gamma_2} P_2'} \tag{6}$$

$$\frac{P_1 \xrightarrow[\gamma']{r(t)@le} P_1' \quad \ell = \mathcal{L}[\![le]\!]_{\gamma' \bullet \gamma} \quad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \quad match(\mathcal{T}[\![t]\!]_{\gamma' \bullet \gamma}, et)}{N_L, \ell ::_\gamma P_1 | P_2 \rightarrowtail N_L, \ell ::_\gamma P_1'[et/\mathcal{T}[\![t]\!]_{\gamma' \bullet \gamma}] | P_2} \tag{7}$$

$$\frac{P_1 \xrightarrow[\gamma]{r(t)@le} P_1' \quad \ell_2 = \mathcal{L}[\![le]\!]_{\gamma \bullet \gamma_1} \quad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \quad match(\mathcal{T}[\![t]\!]_{\gamma \bullet \gamma_1}, et)}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1'[et/\mathcal{T}[\![t]\!]_{\gamma \bullet \gamma_1}], \ell_2 ::_{\gamma_2} P_2} \tag{8}$$

$$\frac{N_L, \ell ::_\gamma P_1 \rightarrowtail N_L, \ell ::_\gamma P_1'}{N_L, \ell ::_\gamma P_1 | P_2 \rightarrowtail N_L, \ell ::_\gamma P_1' | P_2} \tag{9}$$

$$\frac{P \xrightarrow[\gamma']{\nu(u)@\mathtt{self}} P' \quad \ell \notin L \cup \{\ell'\}}{N_L, \ell' ::_\gamma P \rightarrowtail N_L, \ell' ::_\gamma P'[\ell/u], \ell ::_{[\ell/\mathtt{self}] \bullet \gamma} \mathbf{nil}} \tag{10}$$

plus the symmetric of rules (5), (7) and (9)

Table 3. The Structural Rules of Nets Operational Semantics

It says that a process can perform an **in** action at the local tuple space by synchronizing with a process which represents a matching tuple. The result of this synchronization is that the tuple is consumed, i.e. the corresponding process becomes **nil**, and its values are used to replace the corresponding (free) variables of the process which has performed the **in** operation.

Rule (9) models the asynchronous evolution of subcomponents of a node. Such a rule is necessary because, due to the syntax of nodes, rules (5) and (7) might not be applicable. For an example consider the case $P_2 \equiv \mathbf{in}(!\,x)@\mathtt{self}.Q|\mathbf{out}(1)$.

Rules (1)–(9) may modify the structure of the nodes of the net but they cannot introduce new localities. The creation of a new node is described by rule (10). The environment of a new node is obtained from that of the creating one (with the obvious update for the `self` locality). The underlying idea is that the new node inherits all the knowledge about localities of the creating node; obviously, other choices could have been taken. An alternative formulation is:

$$\frac{P \xrightarrow[\gamma']{\nu(u)@\mathtt{self}} P' \quad \ell \notin L \cup \{\ell'\}}{N_L, \ell' ::_\gamma P \rightarrowtail N_L, \ell' ::_\gamma P'[\ell/u], \ell ::_{[\ell/\mathtt{self}]\bullet\phi} \mathbf{nil}}$$

The rationale behind this choice (adopted in [25]) is that any new node has no knowledge of the previously existing net.

We would like to remark that the introduction of rule (9) and its symmetric and of the symmetric of rules (5) and (7) could be avoided by assuming a *structural congruence* in the style of [4] that would imply commutativity and associativity of "|". We did not make this choice because we would like our operational semantics be a guide for future implementations.

2.5 Static vs. dynamic binding

Our operational semantics of nets adopts a static binding discipline for the evaluation of **out** operations. Instead, a dynamic scope discipline is adopted for remote **eval** operations: the meaning of logical localities used by a process spawned at a remote locality depends on the remote allocation environment.

Indeed, whenever a process P located at the locality ℓ_1 wishes to insert a tuple t into the remote tuple space located at ℓ_2, the local environment of P, namely γ_1, is used for evaluating t. A dynamic binding discipline for **out** can be obtained by replacing rule (2) in Table 3 with the following:

$$\frac{P_1 \xrightarrow[\gamma]{s(t)@le} P_1' \quad \ell_2 = \mathcal{L}[\![\, le \,]\!]_{\gamma\bullet\gamma_1} \quad et = \mathcal{T}[\![\, t \,]\!]_{\gamma\bullet\gamma_2}}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1', \ell_2 ::_{\gamma_2} P_2 \mid \mathbf{out}(et)}$$

where the local environment γ_2 is used for evaluating t.

Dynamic binding for **out** can be also simulated within our proposed semantics (without any modification of the operational rules for nets) by writing **eval**(**out**(t)@`self`)@$le.P$ instead of **out**(t)@$le.P$. The execution of **eval** spawns process **out**(t)@`self` at locality ℓ_2 (resulting from the evaluation of le) and, therefore, t is evaluated by using the local environment at ℓ_2.

When process P located at ℓ_1 wants to spawn a process Q at the remote locality ℓ_2, a dynamic binding discipline is followed. The local environment at γ_2 is used for giving meaning to the logical localities which may be referred in Q. A static binding discipline for **eval** can be obtained by spawning $Q\{\gamma_1\}$ rather than Q. More precisely, rule (4) in Table 3 could be replaced by the following:

$$\frac{P_1 \xrightarrow[\gamma]{e(Q)@le} P_1' \quad \ell_2 = \mathcal{L}[\![le]\!]_{\gamma \bullet \gamma_1} \quad Q' = Q\{\gamma_1\}}{N_L, \ell_1 ::_{\gamma_1} P_1, \ell_2 ::_{\gamma_2} P_2 \rightarrowtail N_L, \ell_1 ::_{\gamma_1} P_1', \ell_2 ::_{\gamma_2} Q' \mid P_2}$$

In this case the remote spawning of process Q consists of transmitting Q packaged with its allocation environment γ_1.

Again, **eval** with static scoping can be simulated (without modifying the operational semantics of nets) via the primitives of the language, in particular, by passing processes (and then closures) as fields of tuples and using private localities for storing intermediate results. With this in mind, we can write **newloc**(u).**out**(Q)@u.**in**(!X)@u.**eval**(X)@$le.P$ instead of **eval**(Q)@$le.P$. When **eval**(X) is executed at ℓ_2, X is bound to the process Q packaged with γ_1. Hence, a closure instead of a plain process is activated at ℓ_2, differently from the case of **eval**(Q).

3 Programming Examples

In this section we shall present three small examples which are useful for illustrating how mobile computations can be expressed in `LLinda`. Here, we assume that natural numbers and identifiers are basic values, i.e. belong to the set Val.

3.1 Remote Procedure Call

Our first example shows how remote procedure call can be encoded in our language.

A caller process, *caller*, sends a request to the callee, *callee*, and waits for a response. The request, together with the name of the procedure and its actual parameters, contains the *caller*'s private locality where the response is delivered.

$$\begin{aligned} caller = \mathbf{newloc}(u).\ & \mathbf{out}(\mathrm{proc} - -\mathrm{id}, e_1, \ldots, e_n, u)@l_{callee} \cdot \\ & \mathbf{in}(!\, y_1, \ldots, !\, y_k)@u.\ \langle \text{next behaviour} \rangle. \end{aligned}$$

Process *callee* waits for an invocation, executes the related procedure and sends back the results using the locality, which has been passed together with the service request, while ready to accept other requests.

$$callee = \mathbf{in}(!\,pid, !\,x_1, \ldots, !\,x_n, !\,u)@\mathtt{self}.(callee \mid \langle\, pid(x_1, \ldots, x_n)\rangle.\mathbf{out}(r_1, \ldots, r_k)@u.\mathbf{nil}).$$

When processes are allocated in a net, the local environment of *caller* assigns to the logical locality l_{callee} the physical locality where *callee* is allocated. Hence, we have:

$$net \equiv \{\ell_1 ::_{\{\ell_1/\mathtt{self}, \ell_2/l_{callee}\}} caller, \ell_2 ::_{\{\ell_2/\mathtt{self}\}} callee\}$$

This example points out the use of the operation $\mathbf{newloc}(u)$ to create a private data space accessible only via the variable u.

3.2 Remote Server

Here we tackle the problem of client–server programming. LLinda procedures (i.e. processes) can be invoked, but also transmitted, over the nodes of the net. Due to the static binding discipline for evaluating the arguments (tuples) of **out**, processes passed as fields of tuples have a locality–independent meaning. This, in practice, means that the environment of the originating node is used in the evaluation of the logical localities of the transmitted processes. This (lexical) scoping discipline is similar to that used in Obliq [7]. This is clarified in our next example: a remote server.

Suppose that a client process, *client*, needs to call a server, *server*, to execute a procedure *proc*, incrementing the value of a local integer variable x represented via a two–field tuple (x, v), with v being its actual value. After calling *server*, *client* will wait for an acknowledgment signalling that its request has been serviced.

$$proc = \mathbf{in}(\mathrm{x}, !\,x)@\mathtt{self}.\mathbf{out}(\mathrm{x}, x+1)@\mathtt{self}.\mathbf{nil}$$

$$client = \mathbf{newloc}(u).\mathbf{out}(\mathrm{exec}, u, proc)@l_{server}.\mathbf{in}(\mathrm{ack})@u.\langle \text{ next behaviour}\rangle$$

Process *server* waits for the request of services, manages the incoming request and then sends an acknowledgment back to the client.

$$\begin{aligned} server = \quad & \mathbf{in}(\mathrm{exec}, !\,u, !\,X)@\mathtt{self}.\mathbf{eval}(X)@\mathtt{self}.\mathbf{out}(\mathrm{ack})@u.server \\ + \ & \mathbf{in}(\mathrm{other}--\mathrm{service}, !\,u, \ldots)@\mathtt{self}.\ldots \\ + \ & \ldots \end{aligned}$$

When the *server* and *client* are coordinated into a net, *client* local environment refers the physical allocation of the *server*:

$$net \equiv \{\ell_1 ::_{\{\ell_1/\mathtt{self}, \ell_2/l_{server}\}} client | P_1, \ell_2 ::_{\{\ell_2/\mathtt{self}\}} server\}$$

where we have used P_1 for denoting the local tuple space at node ℓ_1. Therefore, if $P_1 \equiv \mathbf{out}(\mathrm{x}, 0)$, i.e. the value of variable x of the client is 0, the execution of procedure *proc* at locality ℓ_2 assigns 1 to x.

3.3 Dynamic Newsgatherer

Here we illustrate how LLinda can be used for *remote programming*. This kind of programming discipline allows the user to write agents which can dynamically move along the network and can interact locally with other agents. In this way, an agent placed by a user at the server's location can be decoupled from the user and can interact with the server without using the net.

Consider the following scenario. User P needs additional information on a piece of data represented by *item* (e.g. *item* could be the title of a book of which P wants to know the price). Part of the behaviour of P depends on this information; however, there are some activities which are independent of it. P can look for the required information in a database distributed over the network. The starting point of the search, say locality l_{item}, can be chosen according to the search key *item*. We assume that at each node of the database reachable from l_{item}, it is present either a tuple of the form $(item, v)$, containing the desired information, or a tuple of the form $(item, l_{next})$, containing information about the next node to search for the additional information.

The user process P calls for the execution at l_{item} of the agent *gatherer*, which dynamically travels between nodes looking for a tuple that contains information on *item*. This agent takes as parameters the research key *item* and a fresh locality u, which provides the address of the user's private tuple space where the result of the search has to be placed. Once *gatherer* has been spawned, P splits its behaviour into two parallel components: one waits for the additional information and the other proceeds. Thus, those activities which do not need the additional information are decoupled from the search activity, which might be complex and expensive.

$$P = \mathbf{newloc}(u).\mathbf{eval}(gatherer(item, u))@l_{item}.((\mathbf{in}(!\,x)@u.P_1)|P_2)$$

Process *gatherer* can match two alternative tuples. The first one captures the additional information on *item* (e.g. the price); if this is found then it is placed at locality u and *gatherer* terminates. The second tuple is used for obtaining the address of the node where the search has to be repeated.

$$\begin{aligned} gatherer(item, u) = \quad & \mathbf{read}(item, !\,x)@\mathtt{self}.\mathbf{out}(x)@u.\mathbf{nil} \\ & + \mathbf{read}(item, !\,u')@\mathtt{self}.\mathbf{eval}(gatherer(item, u))@u'.\mathbf{nil} \end{aligned}$$

Our assumption about the distributed database guarantees that *gatherer* never deadlocks (because either the associated information or a location where the search can be repeated are surely found) but it does not ensure that the search activity will successful terminate: *gatherer* might loop indefinitely. This could happen if its second tuple, that with location information, always finds a match in the tuple spaces.

4 Concluding Remarks and Related Work

In this paper we have presented a programming notation that supports mobile applications. Our proposal embeds Linda enriched with explicit locality in a

CCS–like calculus. An operational semantics, which focuses on the coordination of mobile agents, is provided. Examples are presented that illustrate how mobile applications and remote programming can be expressed in LLinda. We plan to develop observational semantics as foundation for programming logics and verification techniques. To this purpose, our starting point will be the testing framework developed for a process calculus based on Linda in [10, 23].

Differently, from other distributed programming paradigms (e.g. CML [24], Facile [14] and Telescript [26]), our basic communication mechanism is asynchronous. We consider this kind of communication as more practical. When implemented, communication takes time and its distributed implementation has to face with delays and synchronization overheads. Asynchronous communication is then simpler to implement and indeed many distributed systems and programming languages, such as data flow, concurrent logic and concurrent constraint languages, offer it as basic primitive. Synchronous communications can be implemented by means of a more complex protocol where the sender waits for the reception of acknowledgments. Asynchronous communications has also the advantage of decoupling the behaviours of sender and receiver and of avoiding propagation of failures.

Several theoretical works in non–interleaving semantics of process calculi have adopted the notion of locality to capture logical distribution of processes (see e.g. [5], [6] and the references therein). The basic idea of these approaches is to allow the external observer to see an action together with the location (access path) where it takes place. In our approach, localities are not used as a tool for observing distribution of processes but rather as a programming device to structure and control distribution of processes and data. The formal models presented in [2, 11] are closely related to the work presented here. These approaches deal with mobility much like the π–calculus (channel and locality names can be passed in interactions). Remarkably, localities in LLinda can be used for simulating the private name passing and the scope extrusion mechanisms of the π–calculus, so that a natural encoding of the asynchronous π–calculus (see e.g. [1]) in LLinda can be easily programmed.

Acknowledgments We are grateful to Luca Cardelli for stimulating discussions about global programming.

References

1. R. Amadio, I. Castellani, D. Sangiorgi. On Bisimulation for the Asynchronous π–calculus. In Proc. of CONCUR'96, *LNCS* 1119, 1996.
2. R. Amadio, S. Prasad. Localities and Failures. In Proc. of FCT&TCS 14, *LNCS* 880, 1994.
3. E. Astesiano, G. Reggio. SMoLCS Driven Concurrent Calculi. In Proc. of TAPSOFT'87, *LNCS* 249, 1987.
4. G. Berry, G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217-248, 1992.

5. G. Boudol, I. Castellani, M. Hennessy, A. Kiehn. Observing Localities. *Theoretical Computer Science*, 114, 1993.
6. F. Corradini, R. De Nicola. Locality Based Semantics for Process Algebras. Report DSI-94-05, Univ. Roma, La Sapienza, 1994 (to appear in *Acta Informatica*).
7. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27-59, MIT Press, 1995.
8. L. Cardelli. Global Computation. Manuscript, 1996.
9. N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, 1989.
10. R. De Nicola, R. Pugliese. A Process Algebra based on Linda. Proc. COORDINATION'96, *LNCS* 1061, 1996.
11. C. Fournet, G. Gonthier, J.-L. Lévy, L. Maranget, D. Rémy. A Calculus of Mobile Agents. Proc. CONCUR'96, *LNCS* 1119, 1996.
12. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, 1985.
13. D. Gelernter. Multiple Tuple Spaces in Linda. PARLE'89, LNCS 365, 1989.
14. A. Giacalone, P. Mishra, S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2), 1989.
15. M. Hennessy, H. Lin. Symbolic Bisimulations, *Theoretical Computer Science*, 138:353-389, 1995.
16. Sun Microsystems. The Java Language: A white paper. Sun Microsystems White Paper, 1994.
17. J. Leitcher. Shared Memories, Buses and LANs — Linda Implementations Across the Spectrum of Connectivity. Dep. of Computer Science, Yale Univ., Research Report YALEU/DCS/TR-714, 1989.
18. R. Milner. *Communication and Concurrency*. Prentice Hall Int., 1989.
19. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes, (Part I and II). *Information and Computation*, 100:1-77, 1992.
20. D. Murphy. Observing Located Concurrency. MFCS'93, LNCS 711, 1993.
21. B. Pierce, D. Turner. Concurrent Objects in a Process Calculus. In "Theory and Practice of Parallel Programming", LNCS 907, 1994.
22. G.D. Plotkin. A Structural Approach to Operational Semantics. Tech.Rep. DAIMI FN-19, Aarhus University, Dep. of Computer Science, 1981.
23. R. Pugliese. Semantic Theories for Asynchronous Languages. Ph.D. Thesis VIII-96-6, Univ. di Roma "La Sapienza", Dip. Scienze dell'Informazione, 1996.
24. J. Reppy. Higher Order Concurrency. Ph.D. Thesis, Cornell University, Tr-92-1285, 1992.
25. B. Thomsen, L. Leth, A. Giacalone. Some Issues in the Semantics of Facile Distributed Programming. REX Workshop "Semantics: Foundations and Applications", LNCS 666, Springer, 1992.
26. J.E. White. Telescript Technology: The Foundation for the Electronic Market Place. General Magic White Paper, 1994.