Semantics and Analysis of Linda-based languages

Régis Cridlig & Eric Goubault
Laboratoire d'Informatique de l'Ecole Normale Supérieure

Address: Ecole Normale Supérieure - 45 rue d'Ulm, 75230 Paris Cedex 05, France.
Electronic mail: {cridlig,goubault}@dmi.ens.fr

Abstract. In this paper we define a process algebra abstracting relevant features of the Linda paradigm to parallel computation and show how to give it a semantics based on higher-dimensional automata which is more expressive than interleaving transition systems. In particular, it is a truly concurrent operational semantics, compositional in nature.
Furthermore this semantics leads us to new kinds of abstract interpretations useful for the static analysis of concurrency. One of these addresses the correctness of implementations of Linda programs on real computers (which have a finite number of processors).

1 Introduction

Parallel languages are difficult to design and implement. On the one hand, the task of actually using at the same time several processors should be taken care of in a transparent manner for the user. On the other hand, the multiplicity of architectures and paradigms for parallel machines and languages makes it difficult to find a unified way of speaking about semantics and about efficiency of implementations. In this article, we choose a paradigm for concurrency, exemplified by the Linda based languages, which does not make any assumption on the architecture of the machine it will be implemented on. It does not assume a shared memory nor a channel based mechanism for implementing inter-process communication.

As a first step towards the analysis of such languages, we introduce a process algebra that models their basic features, abstracting the constructs dealing with concurrency and communication. We then use a classical interleaving operational semantics to define the process algebra. Unfortunately, interleaving causes a dramatic combinatorial increase in complexity and the possible schedulings of actions on a given number of processors are out of reach. Therefore, we introduce a generalised operational semantics, based on higher-dimensional automata (see [GJ92]) which expresses the truly concurrent execution of actions, thus enabling us to speak of n actions scheduled on m processors (the "mapping problem"). It has also nice properties borrowed from the denotational world, that is being compositional.

Then we develop two kinds of abstract interpretations (see [CC92]) which link the semantics to non-standard ones of interest. The first one aims at speaking about the mapping problem, whereas the second one is more classical, and can be used for instance for abstracting the actual values of tuples to types. It may also be used

to shrink the domain of the semantics to a finite state one. The article ends by putting these abstract interpretations to work with two examples, for determining the possible communications and the "best" schedulings given a few constraints.

2 Overview of Linda

2.1 The Linda paradigm

How to write parallel programs? This is the subject of N. Carriero and D. Gelernter's book [CG90] that presents and uses Linda, a language they developed as a way to coordinate multiple parallel processes to achieve a given common task.

There are three different basic models for coordination:

1. The first one is still in use in most parallel extensions of imperative languages and is based upon shared memory and variables. To avoid conflicts and race conditions between processes, it usually comprises some basic forms of synchronisation like Djikstra's semaphores. It is considered as a difficult model to program in and debug, and is restricted to shared-memory architectures.
2. The best known one today is message-passing and has been widespread by the influence of Hoare's CSP [Hoa78] that inspired the Occam language [May83]. Communication is modelled by synchronous operations in the process of sending and receiving messages. Remote Procedure Calls and buffered streams are other instances of this paradigm that is quite simple and powerful, allowing distributed computing. But it can lead to cumbersome programs when message passing does not fit well with the concurrent algorithm to implement.
3. Distributed data structures are less frequently encountered, but in some ways generalise both preceding models. Linda offers the programmer a "tuple space" model, which is orthogonal to sequential processes: the data were created by a Linda *eval* operation and are merely "active" tuples that deposit their results as "passive" ones when they exit. Another way to create one (passive) tuple is via the *put* operation. Processes can only access passive tuples, reading them with *read* and also removing them with *get*. These two atomic operations specify a pattern that can only match certain kinds of tuples, thus creating potential non-determinism and possibly blocking until some well-matching object is found.

2.2 C-Linda

In Carriero and Gelernter's C-Linda realisation, there is one global tuple space. Pattern matching is usually done by giving some key that denotes a particular distributed data structure. For instance, pattern $("A", i, j, ?val)$ is used to return element $A(i, j)$ of a distributed matrix A into variable val. This language, while being well-suited for many kinds of parallel algorithms, does not support processes (or functions) as values and lacks object hiding.

2.3 Example

Here we give a merge-sort routine written in ML-Linda, our own combination of the Linda model and the ML functional language. In ML-Linda, multiple tuple spaces

called "bags" can be created and each bag only contains objects of a particular type and is only accessible through its dynamic scope:

```
let rec merge_sort lf rg A =
  if lf < rg then
    let wait = new_bag()
    and middle = (lf+rg)/2
    in eval wait (merge_sort lf middle A);
       eval wait (merge_sort middle rg A);
       get wait () in get wait ()
       in << merge classical algorithm >>;
  ();;
```

Notice that:

- the third parameter `A` is a bag containing the array to be sorted; its elements are of type int $\times \alpha$.
- `wait` is a bag local to each invocation of `merge_sort`; its only purpose is synchronisation and it can hold nothing but objects of type unit.

3 A Linda-calculus

In this paper we are solely interested in modelling and analysing coordination through the tuple space; we shall thus ignore the details of the computation inside of sequential processes. Similarly, we do not want to take into account the values themselves carried by a given tuple.

Consequently, we shall use a kind of Linda-calculus, only describing relevant features of coordination between processes. In that way, we follow [CJY92] but define and use an even simpler form of calculus.

3.1 Syntax

Let X denote a variable for defining recursion (we shall use upper-case characters for them) and t a passive tuple or tuple pattern. Then we define a process as follows:

$$P ::= t \mid \mathrm{out}(P).P' \mid \mathrm{read}(t).P \mid \mathrm{get}(t).P \mid P \| P' \mid X \mid \mathrm{rec}\ X.P$$

'out' subsumes both Linda's eval and put operations, this last one being obtained by the construct $\mathrm{out}(t).P$. We write $\mathcal{L}$ for the set of terms defined by this grammar.

3.2 Semantics by interleaving and multisets

An operational semantics for the Linda-calculus will be given by an Interleaving Transition System. It will be based on a pattern-matching relation between tuples: let us say for instance that tuples a,b,c,... are values (ground objects) while tuples x,y,z,... are formal variables. Then t matches t' iff $t = t'$ or $t' \in \{x, y, z, \ldots\}$.

For each matching pair t, t' a substitution σ upon tuples must be chosen that satisfies the property $t = \sigma t'$. We shall note the matching relation together with its associated substitution by: t σ-matches t'. Substitutions lift to terms of $\mathcal{L}$ in a straightforward manner.

3.3 SOS rules

We choose to model the tuple space by a global multiset M. We shall write $\oplus$ for both multiset construction and union.

$$\begin{array}{rl}
\text{(out)} & M \oplus \text{out}(P).P' \rightarrow M \oplus P \oplus P' \\
\text{(read)} & M \oplus \text{read}(p).P \oplus t \rightarrow M \oplus \sigma(P) \oplus t \text{ if } t \ \sigma\text{-matches } p \\
\text{(get)} & M \oplus \text{get}(p).P \oplus t \rightarrow M \oplus \sigma(P) \quad \text{if } t \ \sigma\text{-matches } p \\
\text{(left choice)} & M \oplus P[\!]P' \rightarrow M \oplus P \\
\text{(right choice)} & M \oplus P[\!]P' \rightarrow M \oplus P' \\
\text{(rec)} & M \oplus \text{rec } X.P \rightarrow M \oplus P' \quad \text{if } M \oplus P[\text{rec } X.P/X] \rightarrow M \oplus P'
\end{array}$$

By this semantics, all actions are synchronous even if we can add a rule mimicking true parallelism[1]:

$$\frac{M_1 \rightarrow M_1' \quad M_2 \rightarrow M_2'}{M_1 \oplus M_2 \rightarrow M_1' \oplus M_2'}$$

This is clearly unrealistic for distributed implementations of the Linda concept. Furthermore, this interleaving semantics creates a lot of undue non-determinism that can only complicate the analysis of Linda programs.

3.4 Example

Let us try to "abstract" our merge-sort program to a Linda-calculus term. The function `merge_sort` becomes a process variable S that is bound by the rec construct to a term describing the internal behaviour of the process. We shall call v the tuple corresponding to ML's () value:

$$\text{rec } S.(v[\!]\text{out}(S).\mathit{out}(S).\text{get}(v).\text{get}(v).v)$$

But this term is not really faithful to the original algorithm, because it can lead for example to the following execution, where instances of process S cannot distinguish between the return tuples of their own children and other ones:

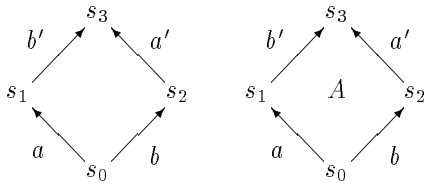
$$\begin{array}{ll}
S & \xrightarrow{*} S \oplus S \oplus \text{get}(v).\text{get}(v).v \\
& \xrightarrow{*} \text{out}(S).\mathit{out}(S).\text{get}(v).\text{get}(v).v \oplus v \oplus \text{get}(v).\text{get}(v).v \\
& \xrightarrow{*} S \oplus S \oplus \text{get}(v).\text{get}(v).v \oplus \text{get}(v).v \\
& \xrightarrow{*} v \oplus v \oplus \text{get}(v).\text{get}(v).v \oplus \text{get}(v).v \\
& \xrightarrow{*} \text{get}(v).v \oplus v \\
& \rightarrow v
\end{array}$$

As exemplified, our calculus can only give an approximate view of Linda-based programs. It can be enriched with a more precise treatment of variables, tuples and substitution, thus leading to a calculus we call λ-linda.

[1] In the light of next section's formalism we can see this rule as allowing cartesian product of transitions $M_1 \rightarrow M_1'$ and $M_2 \rightarrow M_2'$, which represents their synchronised execution.

4 Higher Dimensional Automata

In [Pra91] and [Gla91] Pratt and Glabbeek advocate a model of concurrency based on geometry and in particular on the notion of a higher dimensional automaton. HDA are a generalisation of the usual non–deterministic finite automata as described in *e.g.* [HU79]. The basic idea is to use the higher dimensions to represent the concurrent execution of processes. Thus for two processes, a and b, we model the mutually exclusive execution of a and b by the automaton (1):



(1) **(2)**

whereas their concurrent execution is modeled by including the two–dimensional surface delineated by the (one–dimensional) a– and b–transitions as a transition in the automaton. This is pictured as (2).

HDA are built as sets of states and transitions between states, but also as sets of 2-transitions between transitions, and more generally n-transitions between $(n-1)$-transitions. Transitions (or 1-transitions) are usually depicted as segments, that is one-dimensional objects, whereas states are just points, i.e. 0-dimensional objects. It is therefore natural to represent n-transitions as n-dimensional objects, whose boundaries are the $(n-1)$-transitions from which they can be fired, and to which they end up. n-transitions represent the concurrent execution of n sequential processes. For instance, in automaton (2), the 2-dimensional transition A represents the concurrent execution of a and b. This 2-transition can be fired from a or from b at any time, thus the beginning of A is in some way a and b. Similarly, the end of A is a' and b'. One may want also to add coefficients (like integers) to transitions to keep track of the number of times we go through them. This motivates the introduction of vector spaces[2] generated by states and transitions and source and target boundary operators acting on them.

Definition 1. A (unlabelled) *higher dimensional automaton* (HDA) is a vector space M with two boundary operators ∂_0 and ∂_1, such that:

- there is a decomposition: $M = \sum_{p,q \in \mathbb{Z}} M_{p,q}$, verifying:
  $\forall p,q,\ M_{p,q} \cap (\sum_{r+s \neq p+q} M_{r,s}) = 0$.
- the two boundary operators are compatible with the decomposition and give M a structure of bicomplex:

$$\partial_0 : M_{p,q} \longrightarrow M_{p-1,q}$$

$$\partial_1 : M_{p,q} \longrightarrow M_{p,q-1}$$

$$\partial_0 \circ \partial_0 = 0, \quad \partial_1 \circ \partial_1 = 0, \quad \partial_0 \circ \partial_1 + \partial_1 \circ \partial_0 = 0$$

[2] or, more generally, free modules.

The dimension of an element of $M_{p,q}$ is $p+q$. Such an element is called a $(p+q)$-transition and we will sometimes write M_{p+q} for the sub-vector space of M generated by the $p+q$ transitions. For instance, automaton (2) is defined as,

$$\begin{array}{ccccc}
M_{1,1}=(A) & \xrightarrow{\partial_0} & M_{0,1}=(a)\oplus(b) & \xrightarrow{\partial_0} & M_{-1,1}=(s_0) \\
\downarrow{\scriptstyle\partial_1} & & \downarrow{\scriptstyle\partial_1} & & \downarrow{\scriptstyle\partial_1} \\
M_{1,0}=(a')\oplus(b') & \xrightarrow{\partial_0} & M_{0,0}=(s_1)\oplus(s_2) & \xrightarrow{\partial_0} & M_{-1,0}=0 \\
\downarrow{\scriptstyle\partial_1} & & \downarrow{\scriptstyle\partial_1} & & \downarrow{\scriptstyle\partial_1} \\
M_{1,-1}=(s_3) & \xrightarrow{\partial_0} & M_{0,-1}=0 & \xrightarrow{\partial_0} & M_{-1,-1}=0
\end{array}$$

with $\partial_0(A)=a-b$, $\partial_1(A)=a'-b'$, $\partial_0(a)=\partial_0(b)=s_0$, $\partial_1(a)=\partial_0(b')=s_1$, $\partial_1(b)=\partial_0(a')=s_2$ and $\partial_1(a')=\partial_1(b')=s_3$.

Let Υ be the category of HDA, whose objects are HDA and whose morphisms are linear functions $f : P \to Q$ such that for all $i,j,k, f(P_{i,j}) \subseteq Q_{i,j}$ and $f \circ \partial_k = \partial_k \circ f$. Υ is (small-) complete and co-complete, has a tensor product $\otimes$ and a Hom object such that $\mathrm{Hom}(P \otimes Q, R) \equiv \mathrm{Hom}(P, \mathrm{Hom}(Q,R))$ (see [Gou93]). Moreover, a few properties of the shape of the transition system (branchings, mergings) can be computed algebraically, and inductively for most of the constructs defined on HDA. This is done via the application of suitable n-dimensional homology functors $H_n(.,\partial_k)$ defined on objects X as being the quotient of the kernel, in X_n, of ∂_k by $\partial_k(X_{n+1})$.

Now, we have to define what we mean by a HDA semantics. In ordinary denotational semantics, we just consider the relation between input states and output states of a given program. Therefore semantic domains are made of sets of states, suitably ordered. Now, if we want to be able to observe the whole dynamics of a program, we also need all transitions between these states, and even all higher-dimensional transitions between these transitions. Then a HDA-domain (or in short, a domain) is a huge HDA which contains all possible traces and branchings. Elements of such a domain are just its sub-HDA.

A 1-transition between two states x and y is constructed as an *homotopy* between x and y. This can be coded by means of two special 1-transitions t and v defined by $\partial_0(t)=1$, $\partial_1(t)=0$, $\partial_0(v)=0$, $\partial_1(v)=1$, where 1 is a 0-dimensional element, neutral for the tensor product.

Then a 1-transition[3] going from x to y is $x \rightsquigarrow y = t \otimes x + v \otimes y$. The same formula generalises to higher dimensions, and for instance, $(x \rightsquigarrow y) \rightsquigarrow (z \rightsquigarrow t)$ is a filled-in square whose vertices are x, y, z and t.

5 A truly concurrent semantics of the Linda-calculus

Consider a denumerable family of copies of t and v, denoted by (t_i) and (v_i). Let W be the HDA defined by $W_0=(term)$ for $term$ varying in $\mathcal{L}$, $W_1=(t_i)\oplus(v_i)$. We

[3] Labelling can also be defined (as an element of a slice category Υ/L, see [Gou93]) and can be useful when performing program analysis.

construct a semantic domain D of HDA by the amalgamated sum (noted $+$)

$$D = \sum_{n \in \mathbb{N}} W^{\otimes^n}$$

This domain is easily seen to contain as sub-HDA all sub-HDA of W, and to be stable under the tensor product [4].

The semantic function $[\![.]\!] \in Hom(D, D)$ takes a term x of the Linda-calculus together with a context, the HDA describing the evaluation of the other members of Linda's tuple space, and constructs the HDA representing the possible transitions of x. The semantics of the Linda-calculus is now given by:

$$\begin{aligned}
[\![t]\!]\rho &= t \otimes \rho \\
[\![X]\!]\rho &= X \otimes \rho \\
[\![\mathrm{read}(t).P]\!]\rho &= (\mathrm{read}(t).P) \otimes \rho \rightsquigarrow r^t_P(\rho) + R^t_P(\rho) \\
[\![\mathrm{get}(t).P]\!]\rho &= (\mathrm{get}(t).P) \otimes \rho \rightsquigarrow g^t_P(\rho) + G^t_P(\rho) \\
[\![\mathrm{out}(e_1); e_2]\!]\rho &= (\mathrm{out}(e_1); e_2) \otimes \rho \rightsquigarrow e_1 \otimes e_2 \otimes \rho + [\![e_1]\!]([\![e_2]\!]\rho) + [\![e_2]\!]([\![e_1]\!]\rho) \\
[\![e_1 \| e_2]\!]\rho &= (e_1 \| e_2) \otimes \rho \rightsquigarrow e_1 \otimes \rho + [\![e_1]\!]\rho + (e_1 \| e_2) \otimes \rho \rightsquigarrow e_2 \otimes \rho + [\![e_2]\!]\rho \\
[\![\mathrm{rec}\ X.P]\!]\rho &= \varinjlim [\![P^n(X)]\!]\rho
\end{aligned}$$

where ($s_1 \hat{s_k} s_n$ meaning the product of all s_j except s_k),

$$\begin{aligned}
r^t_P(P_1 \otimes \ldots \otimes P_m) &= \sum_{P_i\ \sigma\text{-matching } t} \sigma(P) \otimes P_1 \otimes \ldots \otimes P_m \\
R^t_P(P_1 \otimes \ldots \otimes P_m) &= \sum_{P_i\ \sigma\text{-matching } t} [\![\sigma(P)]\!](P_1 \otimes \ldots \otimes P_m) \\
g^t_P(P_1 \otimes \ldots \otimes P_m) &= \sum_{P_i\ \sigma\text{-matching } t} \sigma(P) \otimes P_1 \otimes \ldots \otimes \hat{P_i} \otimes \ldots \otimes P_m \\
G^t_P(P_1 \otimes \ldots \otimes P_m) &= \sum_{P_i\ \sigma\text{-matching } t} [\![\sigma(P)]\!](P_1 \otimes \ldots \otimes \hat{P_i} \otimes \ldots \otimes P_m)
\end{aligned}$$

First equation states that the action t is to push the value t in the tuple space. Second one is trivially the same. Then the third and fourth ones mean that a read action (resp. get) is a sum of potential 1-transitions from $\mathrm{read}(t).P$ (resp. $\mathrm{get}(t).P$) to the substitutions of P induced by the matchings of pattern t with elements of context ρ, and then carries on by the evaluation of these substitutions in the appropriate context. out first executes a 1-transition to represent the spawning and then concurrently evaluates its first argument in the context of the execution of the second one, and vice-versa. The choice operator $\|$ is just the union operator, that is the amalgamated sum $+$, between 1-transitions to the beginnings of the execution of its two arguments. Finally recursion is obtained by a direct limit of HDA: the

[4] From now on, we assume that, by a classical argument, we have abelianised — not taking the signs into account — the tensor product by quotienting D by $\{a \otimes b = (-1)^{(\dim a)(\dim b)} b \otimes a\}$.

colimit is taken on the diagram whose objects are the different steps of unfolding $[\![P^n(X)]\!] \overset{Id}{\hookrightarrow} D$ and whose arrows are the inclusion morphisms between them.

Let us now list a few properties of interest, that one can read from this denotational semantics. These are called *geometric* since they are related to the shape of the transition system. They are extracted by using functors built from the homology functors (as defined in [Gou93]).

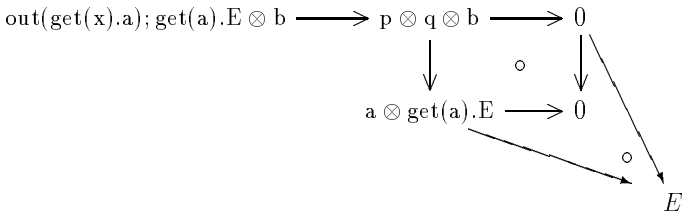
Deadlocks A 1-transition a leads to a deadlock, or simply is a 1-deadlock, if and only if one cannot fire any other transition b in the sequential composition *à la* CSP a;b. This leads to define it in an abstract manner, as a 1-transition a such that $\partial_1(a) = 0$, i.e. a transition of which no information whatsoever can be retrieved as soon as it has been fired. This is typically the case for the t_i's. Therefore, an elementary 1-transition which is also a generator of $\mathrm{Ker}\partial_1$ defines a 1-deadlock. In fact, we just need its representant modulo ∂_1 of 2-transitions. Then finding the 1-deadlocks amounts to computing the generators of $H_1(D, \partial_1)$ that are elementary 1-transitions. This generalises to what we call n-deadlocks, which are n-transitions generators of $H_n(D, \partial_1)$. These may be seen as those n-transitions that deadlock n processors simultaneously.

Serialisation A concurrent program is serialisable if it "gives the same result" as a sequential execution of it. This is a highly geometric property for HDA: this means that all paths can be deformed continuously into another. For instance, branchings of dimension one, given by the computation of the homology group of dimension one for ∂_0, are obstructions to such deformations.

Example 1. In this example, we assume that x is a variable and matches any possible value, while a and b are constant tuples that can only match themselves. Then, denoting $\mathrm{get}(x).a$ by p and $\mathrm{get}(a).E$ by q:

$$
\begin{aligned}
[\![\mathrm{out}(\mathrm{get}(x).a); \mathrm{get}(a).E]\!]b &= (\mathrm{out}(p); q) \otimes b \rightsquigarrow p \otimes q \otimes b + [\![p]\!]([\![q]\!]b) + [\![q]\!]([\![p]\!]b) \\
[\![p]\!]b = p \otimes b \rightsquigarrow g_a^x(b) + G_a^x(b) &= p \otimes b \rightsquigarrow a \\
[\![q]\!]b = q \otimes b \rightsquigarrow g_E^a(b) + G_E^a(b) &= q \otimes b \rightsquigarrow 0 \\
[\![p]\!]([\![q]\!]b) &= (p \otimes q \otimes b \rightsquigarrow 0) \rightsquigarrow (q \otimes a \rightsquigarrow 0) \\
[\![q]\!]([\![p]\!]b) &= (p \otimes q \otimes b \rightsquigarrow q \otimes a) \rightsquigarrow (0 \rightsquigarrow E)
\end{aligned}
$$

Therefore, in D we have the following HDA for the program considered (where the symbol $\circ$ marks 2-transitions) :



In the front square, we have two deadlocks. The only way not to deadlock the system is to schedule the execution of p with higher priority than the evaluation of q, thus creating the value a that enables the matching in q.

6 Static Analyses

6.1 Motivation

The main motivation for static analysis of Linda-based programs is to provide sufficient information for enabling efficient support of fine-grained parallelism on stock hardware. There are two sides to this goal:

1. Constructing efficient implementations of shared data structures. One could infer patterns of usage and sharing for concurrent data structures, as is proposed in [Jag91]. In some cases one can even optimise away Linda tuples (when they only carry synchronisation information for instance).
2. Managing process threads through process allocation and task scheduling. This can be done at runtime, with help from static analysis information about process causality, priority and inter-process communication.

Software validation constitutes another challenge for concurrent programming where static analysis tools can surely help. We will give a method to calculate the homology groups of HDA in a subsequent paper, thus computing relevant branching and merging properties of the associated Linda-calculus term possible executions. These properties can be directly used to show that a program is not bisimulation-equivalent (see [Gou93]) to its specification, for instance.

6.2 Abstract Interpretation

Having given a denotational semantics on a specified domain of transitions, one can interpret the semantic rules in a non-standard domain, related to the full one by a pair of adjoint functors. Let us make this a bit more precise. Let D_c be the domain on which we have given our semantics. Let D_a be another domain. We will say that D_a is an abstraction of D_c if and only if there exists a pair of adjoints functors (α,γ), α being left adjoint to γ, with $\alpha : \Upsilon/D_c \longrightarrow \Upsilon/D_a$ and $\gamma : \Upsilon/D_a \longrightarrow \Upsilon/D_c$. The slice categories Υ/D_c and Υ/D_a (see [FS90]) have as objects "generalized elements", that is, morphisms with value in D_c (resp. D_a). In particular, monomorphisms are just inclusions (in the geometric sense) of HDA in D_c (resp. D_a), that is, correspond to sub-HDA. Notice that these adjoint pairs do not always induce a Galois connection between the lattices of sub-objects (seen as sub-categories of the corresponding slice categories), for instance the relation with ordinary denotational semantics needs more morphisms than just the inclusion morphisms defining the ordering on sub-objects.

In the following, we build several such abstract interpretations.

The truncation functors The truly concurrent semantics we have given for the Linda-calculus assumes an infinite number of processors. The mapping problem is concerned with the possible implementations of such a semantics on a real machine with only n processors. We introduce first an abstract interpretation whose abstraction maps a program onto n processors. Any scheduler can then be proven correct with respect to this abstract interpretation. Questions of efficiency of the scheduling may then be asked. Notice that as a particular case, we obtain the correctness of the interleaving

operational semantics (given in section 3.2) with respect to the truly concurrent one by setting $n = 1$.

Let $D_{a,n} = \sum_{0 \leq k \leq n} W'^{\otimes^k}$ where $W' = (t_i) + (v_i) + \sum_{k \in \mathbb{N}} (term^{\otimes^k})$. It is the domain of processes of dimension at most n. Let now $x : X \to D_c$ be an element of Υ/D_c. Let X' be the sub-HDA of X consisting of transitions up to dimension n ("truncation" of X of order n). We define $T_n(x)$ to be the induced morphism from X' to D_a. For f a morphism between $x : X \to D_c$ and $y : Y \to D_c$, we define $T_n(f)$ to be the induced morphism between the truncations of X and Y of order n. This defines the abstraction functor.

Take A in $\Upsilon/D_{a,n}$. Let $Y(A)$ be the diagram in Υ/D_c, whose objects are all elements x of Υ/D_c such that $T_n(x)$ is isomorphic to A, and whose arrows are all possible morphisms in Υ/D_c between these objects. We define a functor $G_n : \Upsilon/D_{a,n} \longrightarrow \Upsilon/D_c$ to be $G_n = \varinjlim Y(.)$. Then,

Lemma 2. *(T_n, G_n) is a pair of adjoint functors.*

This pair of adjoint functors induces a Galois connection between the lattices of sub-HDA of D_c and $D_{a,n}$ (viewed as a sub-category of Υ/D_c and $\Upsilon/D_{a,n}$ respectively). We apply this result for $n = 1$ to prove:

Proposition 3. *The interleaving operational semantics is correct with respect to the HDA semantics.*

As a matter of fact, T_1 maps any sub-HDA of D to the interleaved 1-transitions on its boundary. For instance automaton (2) of section 4 is mapped onto automaton (1). To prove the correctness, we just have to forget all explicitly coded deadlocks in the HDA semantics. This is also part of an adjunction we will not describe now.

The folding functors Let $\equiv$ be a given equivalence on terms, and let $p : \mathcal{L} \to \mathcal{L}/\equiv$ be the associated canonical projection. We define an abstract domain by $D_a = \sum_{n \in \mathbb{N}} W''^{\otimes^n}$ where $W'' = (t_i) + (v_i) + (term_{\equiv})$. Now, p extends to a multiplicative morphism[5] from D_c to D_a, by, $p(x \otimes y) = p(x) \otimes p(y)$ and $p(t_i) = t_i$, $p(v_i) = v_i$. More generally, we can assume that we are given an epimorphism p from D_c to a domain D_a. Then the abstraction functor is:

$$M_p(x : X \to D_c) = p \circ x : X \to D_a$$

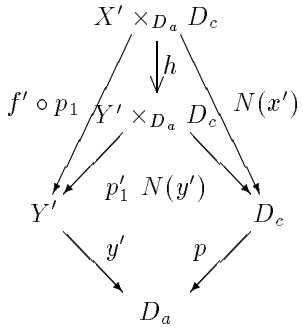
$$M_p(f : (x : X \to D_c) \to (y : Y \to D_c)) = f : (M_p(x) : X \to D_a) \to (M_p(y) : Y \to D_a)$$

Let now N_p be the functor from Υ/D_a to Υ/D_c defined by:

- for $x' : X' \to D_a$, $N_p(x')$ is the pullback of x' along p, i.e. is the "greatest" morphism $N_p(x') : X' \times_{D_a} D_c \to D_c$ such that $p \circ N(x') = x' \circ p_1$ where $p_1 : X' \times_{D_a} D_c \to X'$ is given by the pullback diagram (see [Mac71]).

[5] i.e. a morphism which commutes with the tensor product.

– and for $f' : (x' : X' \rightharpoonup D_a) \longrightarrow (y' : Y' \rightharpoonup D_a)$, $N_p(f') : X' \times_{D_a} D_c \rightharpoonup Y' \times_{D_a} D_c$ is the unique morphism h in the following pullback diagram:



Then,

Lemma 4. *(M_p, N_p) is a pair of adjoint functors.*

We need in the following to compute the abstract operators, i.e. the abstract counterparts of $+$, $\otimes$ and $\varinjlim$. This will not be possible in general, and we may have to use safe approximations of them. For H any endofunctor on Υ/D_c, we say that G, endofunctor on Υ/D_a, is a safe approximation of H if and only if there exists a natural transformation from $\alpha H \gamma$ to G. Notice that it reduces to the usual definition when (α, γ) is a Galois connection. The fact that (α, γ) is a pair of adjoint functors implies that colimits in Υ/D_a are safe approximations of colimits in Υ/D_c. For instance, we can take as abstraction of $+$ and $\varinjlim$, $+$ and $\varinjlim$ respectively. This does not hold for $\otimes$ and its abstract version $\otimes_a$. But we can prove the following:

– For the adjunction (T_n, G_n), there is an "expansion law", $x \otimes_a y = x \otimes T_0(y) + T_0(x) \otimes y + \Sigma_{0<k<n} T_k(x) \otimes T_{n-k}(y)$.
– For the adjunction (M_p, N_p), if p is a multiplicative morphism then $x \otimes_a y = x \otimes y$.

6.3 Example: communication analysis

We would like to trace the communications occurring during the execution of Linda programs by collecting input operations (read and get) and tuples at each point of execution.

Annotated syntax In order to distinguish operations up to their syntactic context, we must add "control points" to Linda programs. A control point is essentially a token annotating each syntactical construct, for instance integers which are given in the textual order. As an example, $\mathrm{out}(a); \mathrm{read}(x).E$ becomes $\mathrm{out}_1(a_2); \mathrm{read}_3(x_4).E_5$. We will not define this operation formally, but assume from now on that the concrete domain is changed into $D_c = \sum_k W^{\otimes^k}$ with $W = (t_i) + (v_i) + (\textit{annotated-term})$.

Adjunction For our abstract domain D_m, we take the domain of HDA generated by all terms $\mathrm{read}_k(t)$, $\mathrm{get}_k(t)$ and all tuples s_k *with the restriction that there will be at most* $m > 0$ *copies of the same annotated tuple* s_k *in each state.* Each 0-transition of these HDA describes the state of the tuple space during the evaluation of the program and the possible synchronisations occurring after it.

Let us define an abstraction morphism $u_m : D_{a,0} \to D_m$ by induction on the following equations:

$$\begin{aligned}
u_m(\otimes^{m+k} s_j) &= \otimes^m s_j \\
u_m(\otimes^n s_j) &= \otimes^n s_j \qquad \text{if } n < m \\
u_m(\mathrm{read}_k(t).P) &= \mathrm{read}_k(t) \\
u_m(\mathrm{get}_k(t).P) &= \mathrm{get}_k(t) \\
u_m(term) &= 1 \qquad \text{otherwise}
\end{aligned}$$

It is clear that u_m is the canonical projection of an equivalence relation on annotated terms. So, by the result of the last section, u_m defines a pair of adjoint functors (M_{u_m}, N_{u_m}).

Abstract semantics The non-standard semantic equations are thence given by the same equations as in the truncation domain $D_{a,0}$. These equations can be further simplified when one remembers that the tensor product works in the abstract domain here, thus calculating in the equivalence classes:

$$\begin{aligned}
[\![t]\!]\rho &= t \otimes \rho \\
[\![X]\!]\rho &= \rho \\
[\![\mathrm{read}(t).P]\!]\rho &= \mathrm{read}(t) \otimes \rho + R^t_P(\rho) \\
[\![\mathrm{get}(t).P]\!]\rho &= \mathrm{get}(t) \otimes \rho + G^t_P(\rho) \\
[\![\mathrm{out}(e_1); e_2]\!]\rho &= \rho + [\![e_1]\!]([\![e_2]\!]\rho) + [\![e_2]\!]([\![e_1]\!]\rho) \\
[\![e_1 \| e_2]\!]\rho &= \rho + [\![e_1]\!]\rho + [\![e_2]\!]\rho \\
[\![\mathrm{rec}\ X.P]\!]\rho &= \lim_{\rightarrow} [\![P^n(X)]\!]\rho
\end{aligned}$$

The law for R^t_P remains the same, while the one for G^t_P becomes:

$$G^t_P(P_1^{\otimes^{k_1}} \otimes \ldots \otimes (P_l^{\otimes^{k_l}} = \sum_{P_i\ \sigma\text{-matching } t} [\![\sigma(P)]\!](P_1 \otimes \ldots \otimes P_i^{\hat{\otimes}^{k_i}} \otimes \ldots \otimes P_l)$$

$$\text{where} \quad P_i^{\hat{\otimes}^m} = P_i^{\otimes^m} + P_i^{\otimes^{m-1}}$$

$$\text{and} \quad P_i^{\hat{\otimes}^{n<m}} = P_i^{\otimes^{n-1}}$$

Example Let $m = 1$: an abstract tuple represents any number of concrete ones. We can calculate the abstract semantics of the program of the first example:

$$\begin{aligned}
&[\![\mathrm{out}(\mathrm{get}(x).a); \mathrm{get}(a).E]\!]b = b + [\![p]\!]([\![q]\!]b) + [\![q]\!]([\![p]\!]b)\\
&[\![p]\!]b = u(p) \otimes b + G^x_a(b) \quad = u(p) \otimes b + a \otimes b + a\\
&[\![q]\!]b = u(q) \otimes b + G^a_1(b) \quad = u(q) \otimes b\\
&[\![p]\!]([\![q]\!]b) \qquad\qquad\quad = u(p) \otimes u(q) \otimes b + u(q) \otimes a + u(q) \otimes a \otimes b\\
&[\![q]\!]([\![p]\!]b) \qquad\qquad\quad = u(q) \otimes (u(p) \otimes b + a \otimes b + a) + a \otimes b + b + a\\
&\quad \text{So, } [\![\mathrm{out}(p); q]\!]b = a + b + a \otimes b + u(q) \otimes u(p) \otimes b + u(q) \otimes a \otimes b + u(q) \otimes a
\end{aligned}$$

The result shows an upper approximation of the potential matchings between the tuples a and b and the two syntactic get operations; with $m = 2$, we would have obtained an exact result.

6.4 Example: the scheduling of constrained Linda programs

When we want to compile Linda programs, we have to keep in mind that there exists a number of different constraints due to the target machine. In this article, we will focus on two such constraints. The first one is that the machine has only n processors. The second one is that the machine has a limited amount of memory. We wish to give the compiler static information about which scheduling of the different actions is the "best" given these constraints. In the limited model we consider, "best" means that we do not want to create too many resources of some type A_i (as it may overflow the capacity of some distributed memory) but sufficiently many ones of type A_j (since they are output data we do not want to wait for). It means also that we want sufficiently many resources of type A_k for not blocking read and get operations, and not making the program deadlock, since we have only a limited number of processors that could all be waiting for a resource.

Now, we show how to abstract the semantics to have just the authorised transitions, given that we are constrained to n processors and that we do not want to have more than β_i and less than α_i tuples A_i. Let us define a morphism p from D_c to D_c by:

$$\begin{aligned}
p(A_i^{\otimes^j}) &= A_i^{\otimes^j} && \text{if } \alpha_i \leq j \leq \beta_i\\
p(A_i^{\otimes^j}) &= 0 && \text{if } j < \alpha_i \text{ or } \beta_i < j\\
p(t_i) &= t_i\\
p(v_i) &= v_i\\
p(term) &= term
\end{aligned}$$

Let then $D'_a = p(D_{a,n})$. It is the domain of traces of execution on an n-processor machine, deadlocking (thus measuring in a very definite way the ineffectiveness of the implementation) if it uses more or less resources than specified. Then we know that $(M_p \circ T_n, N_p \circ G_n)$ is a pair of adjoint functors between Υ/D_c and Υ/D'_a.

Finally, we want to collect the set of states which lead to deadlocks, as well as the deadlocking transitions. This is the information we need to tell the scheduler, "when you are in such and such states, try to delay such and such transitions".

Let $D_a = D'_a/(v_i)_i$. It is an abstract domain where transitions are deadlocks. The abstraction functor is essentially the homology functor for ∂_1 which computes deadlocks. But we do not want in the first place to bother with equivalence classes. Therefore, we set for any HDA M, $F(M) = \oplus_{i\geq 1}(\mathrm{Ker}_{|M_i}\partial_1)$ (the HDA in D_a generated by the kernel of ∂_1 on objects of dimension greater or equal than one). If $f : M \to N$ is a morphism in Υ, then f induces a morphism $f^* : F(M) \to F(N)$, since f commutes with ∂_1 (and ∂_0). It therefore defines a functor from Υ/D'_a to Υ/D_a by:

$$F(X \xrightarrow{x} D'_a) = F(X) \xrightarrow{x^*} D_a$$
$$F(f : (X \xrightarrow{x} D'_a) \longrightarrow (Y \xrightarrow{y} D'_a)) = f^* : F(x) \to F(y)$$

Let G be the functor $G(x) = \varinjlim Z(x)$, where $Z(x)$ is, for $x \in \Upsilon/D_a$, the diagram in Υ/D'_a, whose objects are the y such that $F(y) = x$ and whose morphisms are all possible morphisms between them. It is easy to see that (F, G) is a pair of adjoint functors.

Therefore, $(F \circ M_p \circ T_n, G \circ N_p \circ G_n)$ is a pair of adjoint functors. Representing objects $x^* : F(X) \to D_a$ by $x^*(F(X))$, we get the abstract semantic equations for $n = 1$, $\alpha_i = 0$ and $\beta_i = \infty$:

$$\begin{aligned}
[\![t]\!]\rho &= t \otimes \rho \\
[\![X]\!]\rho &= X \otimes \rho \\
[\![\mathrm{read}(t).P]\!]\rho &= (\mathrm{read}(t).P) \otimes T_0(\rho) \rightsquigarrow 0 + \mathrm{read}(t).P \otimes \rho && \text{if } r^t_P(T_0(\rho)) = 0 \\
[\![\mathrm{read}(t).P]\!]\rho &= \mathrm{read}(t).P \otimes \rho + R^t_P(\rho) && \text{otherwise} \\
[\![\mathrm{get}(t).P]\!]\rho &= (\mathrm{get}(t).P) \otimes T_0(\rho) \rightsquigarrow 0 + \mathrm{get}(t).P \otimes \rho && \text{if } g^t_P(T_0(\rho)) = 0 \\
[\![\mathrm{get}(t).P]\!]\rho &= \mathrm{get}(t).P \otimes \rho + G^t_P(\rho) && \text{otherwise} \\
[\![\mathrm{out}(e_1); e_2]\!]\rho &= (\mathrm{out}(e_1); e_2) \otimes \rho + [\![e_1]\!]([\![e_2]\!]\rho) + [\![e_2]\!]([\![e_1]\!]\rho) \\
[\![e_1 \| e_2]\!]\rho &= (e_1 \| e_2) \otimes \rho + [\![e_1]\!]\rho + [\![e_2]\!]\rho \\
[\![\mathrm{rec}\ X.P]\!]\rho &= \lim_{\rightarrow} [\![P^n(X)]\!]\rho
\end{aligned}$$

Example 2. We get back again to our first example.

$$\begin{aligned}
[\![q]\!]b &= q \otimes b \rightsquigarrow 0 \\
[\![p]\!]b &= p \otimes b + a \\
[\![p]\!]([\![q]\!]b) &= p \otimes q \otimes b \rightsquigarrow 0 + a \otimes q \rightsquigarrow 0 \\
[\![q]\!]([\![p]\!]b) &= E + p \otimes q \otimes b + a \otimes q \\
[\![\mathrm{out}(p); q]\!]b &= \mathrm{out}(p); q \otimes b + [\![p]\!]([\![q]\!]b) + [\![q]\!]([\![p]\!]b) \\
&= \mathrm{out}(p); q \otimes b + p \otimes q \otimes b \rightsquigarrow 0 + a \otimes q \rightsquigarrow 0 + E
\end{aligned}$$

Using the labelling, the result is to be interpreted as follows: the scheduler must try to delay the execution of q in favor of p.

7 Conclusion

In this article, we have presented a process algebra which is the core of the Linda based languages. Using higher dimensional automata for giving its semantics, we have been able to introduce new kinds of abstract interpretations. In particular, we have been able to formalise the link between the idealistic truly concurrent semantics, assuming an infinite number of processors, and more realistic ones, constrained to use a finite number of processors only. It is noteworthy that the approach promoted here is not restricted to Linda-based languages. Future work will have to combine our abstractions with more classical analyses like numerical ones (see [Mas92]). They could in particular be used for abstracting states or relations between states in our semantic domains.

Acknowledgements We wish to thank Chris Hankin and Lindsay Errington for the many discussions we had about the static analysis of Linda- and Gamma- like languages. We would like to thank Klaus Havelund for many valuable discussions as well, about concurrency in general and the Linda model in particular.

References

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), 1992.

[CG90] N. Carriero and D. Gelernter. *How to write parallel programs A first course*. MIT Press, Cambridge, Mass., 1990.

[CJY92] P. Ciancarini, K. K. Jensen, and D. Yankelevitch. The semantics of a parallel language based on a shared data space. Technical Report 26, DIUP, 1992.

[FS90] P. J. Freyd and A. Scedrov. Categories, allegories. In *North-Holland Mathematical Library*, volume 39. North-Holland, 1990.

[GJ92] E. Goubault and T.P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*, Stonybrook, New York, August 1992. Springer-Verlag.

[Gla91] R. van Glabbeek. Bisimulation semantics for higher dimensional automata. Technical report, Stanford University, 1991.

[Gou93] E. Goubault. Towards a theory of higher-dimensional automata. Technical report, Ecole Normale Supérieure, to appear 1993.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):667–677, August 1978.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley, 1979.

[Jag91] S. Jagannathan. Expressing fine-grained parallelism using concurrent data structures. In J. B. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 77–92, June 1991.

[Mac71] S. Mac Lane. *Categories for the working mathematician*. Springer-Verlag, 1971.

[Mas92] F. Masdupuy. Array operations abstraction using semantic analysis of trapezoid congruences. In *International Conference on Supercomputing*, July 1992.

[May83] D. May. OCCAM. *ACM SIGPLAN Notices*, 18(4):69–79, April 1983.

[Pra91] V. Pratt. Modeling concurrency with geometry. In *Proc. 18th ACM Symposium on Principles of Programming Languages*. ACM Press, 1991.