Proving the correctness of optimising destructive and non-destructive reads over tuple spaces

Rocco De Nicola[1], Rosario Pugliese[1], and Antony Rowstron[2]

[1] Dipartimento di Sistemi ed Informatica, Università di Firenze,
Via C. Lombroso, 6/17 50135 Firenze, Italy.
{denicola, pugliese}@dsi.unifi.it
[2] Microsoft Research Ltd, St. George House,
1 Guildhall Street, Cambridge, CB2 3NH, UK.
antr@microsoft.com

Abstract. In this paper we describe the proof of an optimisation that can be applied to tuple space based run-time systems (as used in Linda). The optimisation allows, under certain circumstances, for a tuple that has been destructively removed from a shared tuple space (for example, by a Linda in) to be returned as the result for a non-destructive read (for example, a Linda rd) for a different process. The optimisation has been successfully used in a prototype run-time system.

1 Introduction

In this paper we present the proof of an optimisation that can be applied to tuple space based run-time systems, which was first presented in Rowstron [1]. Examples of tuple space based systems are JavaSpaces [2], KLAIM [3], Linda [4], PageSpace [5], TSpaces [6], TuCSoN [7] and WCL [8] to name just a few.

Throughout this paper we will just use the three standard Linda tuple space access primitives:

out(tuple) Insert a tuple into a tuple space.
in(template) If a tuple exists that matches the template then remove the tuple and return it to the process performing the `in`. If no matching tuple is available then the process blocks until a matching tuple is available.
rd(template) If a tuple exists that matches the template then return a copy of the tuple to the process that performed the `rd`. If there is no matching tuple then the process blocks until a matching tuple is available.

Moreover, we shall assume that a single global tuple space is being used by all processes.

The optimisation proved in this paper is referred to as *tuple ghosting*. The (informal) semantics of the `in` primitive leads implementers to remove the tuple that is returned to a process from the tuple space as soon as the `in` primitive is completed. Tuple ghosting allows the tuple to potentially remain as a valid result tuple for a non-destructive read performed by another process whilst a set of assumptions holds.

Studying the soundness of the optimisation has been highly valuable; it showed that the original algorithm [1] was too optimistic and allowed the result to remain visible for too long. In certain circumstances, the original rules used for the optimisation altered the semantics of the access primitives. We are confident now that the actual optimisation, modified to use the semantics given in Section 3, is sound.

1.1 Motivation for the optimisation

Optimisation of tuple removal is useful because often tuples are used to store shared state between processes. For instance, a list is usually stored in a tuple space so that the items of the list are stored in separate tuples, with each tuple containing a unique number as the first field, representing its position in the list. A single tuple is required that contains a shared counter indicating the number of the next element that can be added. In order to add an element to the list, the shared counter is removed using an `in`, the value of the counter increased and the tuple is re-inserted, and then a new tuple is inserted containing the number of the counter and the data as an element in the list. This is a common operation and there have been proposals for the addition of new primitives to help performing the update of the shared counter (see e.g. Eilean [9]), and, when using compile-time analysis, to convert the counter updating into a single operation [10]. The proposals were made with the intention of increasing concurrency. Additionally, in high performance servers the cost of managing a primitive blocked waiting for a matching tuple is greater than finding a matching tuple and not blocking. One of the new challenges for tuple space implementers is to create large-scale high throughput servers, and therefore optimisations that reduce the server load are important.

1.2 Implementation

Tuple ghosting has been implemented in a Java run-time environment and it has proved to be clean and efficient. To provide tuple ghosting the implementation uses the following informal rules. When a tuple is returned as the result of an `in`:

1. the same tuple cannot be returned as a result of another `in`;
2. the process which performed the `in` cannot access the tuple anymore;
3. whenever the process which performed the `in` performs any tuple space access or terminates, the tuple is removed.

The kernel works by marking tuples as ghosted once they have been returned by an `in` primitive. Every process using the kernel has a Globally Unique Identifier (GUID) created dynamically as it starts to execute. When the process registers with the run-time system, the process GUID is passed to the run-time system that creates a primitive counter associated with the process. Each time a process performs a tuple space access, the counter associated with the process is incremented by one (before the primitive is performed). When a process requests a tuple using an `in`, the matched tuple is marked as "ghosted" and tagged with the GUID of the process that removed the tuple and with the current value of the primitive counter associated with the process. Any other process can then perform a `rd` and have this tuple as the result. However, whenever the tuple is matched, the system compares on-the-fly the current value of the primitive counter associated with the GUID attached to the tuple with the counter value attached to the tuple. If the primitive counters differ or if the process has terminated then the tuple is discarded, and not used as a result for the `rd`.

All communication between the processes must occur through the shared tuple space. Hidden communication between the processes would allow the processes to determine that one had read a tuple after it had been destructively removed by another. Process termination is an example of hidden communication (where, for example, one process is started after another process terminates). The starting process can deduce that any tuples removed by the terminated process should not exist. Therefore, accounting for termination is important.

The rules that the kernel uses are described in detail in Section 3.

1.3 Performance

Table 1 shows some experimental results which, by means of the example of a list stored in a tuple space, demonstrate the advantages of using tuple ghosting using. In our scenario, the list is accessed by two reader processes that read the counter 20 times, and a writer process that appends 40 elements to the end of the list (updates the counter and adds new element).

The experimental run-time was written in Java, with the reader and writer processes running as Java threads. The results were gathered on a Pentium II 400 MHz PC. The results shown in Table 1 are the average times of 20 executions with tuple ghosting both enabled and disabled. The execution times (with standard deviations) are shown for the three processes. For the reader processes, the number of blocked and ghosted `rd` primitives are also shown. A ghosted `rd` is one that would have blocked if tuple ghosting was not enabled.

The results show (as expected) that no `rd` primitive leads to blocking when tuple ghosting is enabled, but when ghosting is disabled we have that 70% of the `rd` primitives do lead to a block. Tuple ghosting has therefore increased the level of concurrency achieved in the system. In addition, the execution times are reduced when the tuple ghosting is enabled. This is due to the overhead associated with managing a `rd` that is blocked because no tuple is available.

The rest of the paper is structured as follows. In the next section the structural operational semantics for a traditional Linda implementation is outlined, then in Section 3 the optimisation is outlined in more detail, and the structural operational semantics for the optimised Linda implementation is presented. The proof of correctness of the optimised version is then given in Section 4.

		Ghosting disabled		Ghosting enabled	
		Value	St. Dev.	Value	St. Dev.
Reader 1	Time (ms)	**4367**	566	**185**	39
	No. of blocking `rd`	**15.75**	1.37	**0**	0
	No. of ghosted `rd`	0	0	9.75	0.64
Reader 2	Time (ms)	**4281**	743	**194**	37
	No. of blocking `rd`	**15.35**	1.81	**0**	0
	No. of ghosted `rd`	0	0	9.9	0.85
Writer	Time (ms)	**4886**	670	**590**	71

Table 1. Performance of the implementation with tuple ghosting enabled and disabled.

2 Structural Operational Semantics for a Linda Kernel

2.1 Syntax

We assume the existence of some predefined syntactic categories that processes can use. EXP, the category of *value expressions*, which is ranged over by e, contains a set of *variable symbols*, VAR, ranged over by x, y and z, and a non-empty countable set of value symbols, VAL, ranged over by v.

The three standard Linda tuple space primitives are the elementary actions that processes can perform. Processes are constructed by using three composition operators: the *null* process nil is a process constant that denotes a terminated process, the *action prefix* operator $a.\_$ is a unary operator that denotes a process that first executes action a and then behaves as its process argument, and the *parallel composition* operator $\_ \parallel \_$ is a binary operator that denotes the concurrent execution of its two arguments. Processes can also consist of *evaluated* tuples (they are a separate syntactic category), that represent tuples that have been added to the tuple space (as in [11]). An evaluated tuple is denoted by $\underline{out}(v)$ with $v \in VAL$.

To give a simpler presentation of our formal framework, we make a few simplifying assumptions. We assume that tuples and templates consists of just one field. The only difference between tuples and templates is that the formers can only contain expressions (or values) while the latters can also contain formal parameters (i.e. variables to be assigned). A parameter x is denoted by $\underline{x}$, the set of all parameters $\{\underline{x} \mid x \in VAR\}$ is denoted by $\underline{VAR}$.

By summarizing, the syntax of the language is

$$P, Q \ ::= \ nil \mid a.P \mid P \parallel Q \mid P \parallel O \ \mid \ O \parallel P$$
$$O \ ::= \ \underline{out}(v) \ \mid \ O_1 \parallel O_2$$
$$a \ ::= \ out(e) \mid rd(t) \mid in(t)$$
$$t \ ::= \ e \mid \underline{x}$$

Variables which occur in formal parameters of a template t are *bound* by $rd(t).\_$ and $in(t).\_$. If P is a process, we let $bv(P)$ denote the set of bound variables in P and $fv(P)$ denote that of free variables in P. If $bv(P) = \emptyset$ we say that process P is *closed*. Sets $bv(\_)$ and $fv(\_)$ can be inductively defined as follows:

$$\begin{aligned} fv(nil) &\overset{\text{def}}{=} \emptyset & bv(nil) &\overset{\text{def}}{=} \emptyset \\ fv(a.P) &\overset{\text{def}}{=} fv(P) \setminus bv(a) & bv(a.P) &\overset{\text{def}}{=} bv(P) \cup bv(a) \\ fv(P \parallel Q) &\overset{\text{def}}{=} fv(P) \cup fv(Q) & bv(P \parallel Q) &\overset{\text{def}}{=} bv(P) \cup bv(Q) \\ fv(P \parallel O) &\overset{\text{def}}{=} fv(P) & bv(P \parallel O) &\overset{\text{def}}{=} bv(P) \\ fv(O \parallel P) &\overset{\text{def}}{=} fv(P) & bv(O \parallel P) &\overset{\text{def}}{=} bv(P) \end{aligned}$$

$$\begin{aligned} fv(out(e)) &\overset{\text{def}}{=} \begin{cases} \{e\} \text{ if } e \in VAR \\ \emptyset \quad \text{otherwise} \end{cases} & bv(out(e)) &\overset{\text{def}}{=} \emptyset \\ fv(in(t)) &\overset{\text{def}}{=} \begin{cases} \{t\} \text{ if } t \in VAR \\ \emptyset \quad \text{otherwise} \end{cases} & bv(in(t)) &\overset{\text{def}}{=} \begin{cases} \{x\} \text{ if } t = \underline{x} \\ \emptyset \quad \text{otherwise} \end{cases} \\ fv(rd(t)) &\overset{\text{def}}{=} \begin{cases} \{t\} \text{ if } t \in VAR \\ \emptyset \quad \text{otherwise} \end{cases} & bv(rd(t)) &\overset{\text{def}}{=} \begin{cases} \{x\} \text{ if } t = \underline{x} \\ \emptyset \quad \text{otherwise} \end{cases} \end{aligned}$$

As usual, we write $P[v/t]$ to denote the term obtained by substituting each free occurrence of t in P with v, whenever $t \in VAR$, and to denote P, otherwise.

2.2 Operational semantics

The operational semantics assumes the existence of a function for evaluating value expressions; $[\![\cdot]\!] : EXP \longrightarrow VAL$. So, $[\![e]\!]$ will denote the value of expression e, provided that it does not contain variables. $[\![\cdot]\!]$ is extended to templates in the obvious way (i.e. $[\![\underline{x}]\!] = \underline{x}$).

The operational semantics of the language is defined in the SOS style [12] by means of a *Labelled Transition System* (LTS). This LTS is the triple $(\mathcal{P}_1, \mathcal{L}_1, \longrightarrow_1)$ where:

- $\mathcal{P}_1$, ranged over by P and Q, is the set of closed processes generated by the syntax given in Section 2.1.
- $\mathcal{L}_1 \overset{\text{def}}{=} \{out(v), rd(v), in(v) | v \in VAL\}$ is the set of *labels* (we shall use a to range over $\mathcal{L}_1$ and s over $\mathcal{L}_1^*$).
- $\longrightarrow_1 \subseteq \mathcal{P}_1 \times \mathcal{L}_1 \times \mathcal{P}_1$, called the *transition relation*, is the least relation induced by the operational rules in Table 2 (to give a simpler presentation of the rules, we rely on a *structural relation* defined as the least equivalence relation closed under parallel composition that satisfies the structural rules in Table 2). We shall write $P \xrightarrow{a} Q$ instead of $(P, a, Q) \in \longrightarrow_1$.

For $s \in \mathcal{L}_1^*$ and $P, Q \in \mathcal{P}_1$, we shall write $P \xrightarrow{s} Q$ to denote that $P = Q$, if $s = \epsilon$, and that $\exists P_1, \ldots, P_{n-1} \in \mathcal{P}_1 : P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \ldots P_{n-1} \xrightarrow{a_n} Q$, if $s = a_1 a_2 \ldots a_n$.

Let us briefly comment on the rules in Table 2. The structural laws simply say that, as expected, parallel composition is commutative, associative and has *nil* as the identity element. The operational rules S1-S5 should be self-explanatory. Rules S1 and S2 just account for the intentions of processes to perform operations. They define an auxiliary transition relation whose states are (not necessarily closed) processes and whose set of labels is $\{\underline{out(e)}, \underline{rd(t)}, \underline{in(t)} \,|\, e \in EXP, t \in$

Structural Rules

$$P \parallel nil \equiv P \qquad P \parallel Q \equiv Q \parallel P \qquad P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$$

Operational Rules

S1 $a.P \xrightarrow{a} P$

S2 $\dfrac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}$

S3 $\dfrac{P \xrightarrow{out(e)} P' \quad \wedge \quad [\![e]\!] = v}{P \xrightarrow{out(v)} P' \parallel \underline{out}(v)}$

S4 $\dfrac{P \xrightarrow{rd(t)} P' \quad \wedge \quad ([\![t]\!] = v \vee t \in \underline{VAR})}{P \parallel \underline{out}(v) \xrightarrow{rd(v)} P'[v/t] \parallel \underline{out}(v)}$

S5 $\dfrac{P \xrightarrow{in(t)} P' \quad \wedge \quad ([\![t]\!] = v \vee t \in \underline{VAR})}{P \parallel \underline{out}(v) \xrightarrow{in(v)} P'[v/t]}$

S6 $\dfrac{P \equiv Q \quad \wedge \quad Q \xrightarrow{a} Q' \quad \wedge \quad Q' \equiv P'}{P \xrightarrow{a} P'}$

Table 2. Linda Operational Semantics

$EXP \cup \underline{VAR}\}$. The remaining rules build upon them. Rule S3 says that an output operation is always non blocking, provided that the argument tuple can be evaluated. Rule S4 says that a read operation can be performed only if there is a tuple matching the template used by the operation. To check pattern-matching, condition "$[\![t]\!] = v \vee t \in \underline{VAR}$" is used; it is satisfied when either t is an expression that evaluates to v (the value stored in the tuple) or t is a parameter (a parameter matches any value). Rule S5 differs from S4 just for the management of the accessed tuple: indeed, in S5, the tuple is consumed, while, in S4, the tuple is left untouched. Finally, rule S6 ensures that the structural relation does not modify the behaviour of processes.

3 Structural Semantics for Optimised Linda

Having described the basic Linda structural semantics, we now consider the structural semantics for the optimised Linda implementation that uses tuple ghosting. In order to illustrate tuple ghosting in more detail, let us consider two very simple processes that interact through the tuple space. Their actions are shown in Table 3.

Process A	Process B
A_1 out(a)	B_1 in(a)
A_2 rd(a)	B_2 out(b)
A_3 rd(b)	B_3 out(a)

Table 3. Two simple example processes.

We shall use Petri Nets and their unfoldings as case graphs to describe the difference between the "classical" and the "optimized" semantics. In a Petri net the circles represent places, and the squares represent transitions. A transition can fire only when all the places that are preconditions for that transition contain tokens. When a transition fires it consumes the tokens in its preconditions and places a token in each of the output places that are linked to it by arcs.

The Petri net and the case graph showing the parallel composition of our two processes can be seen in Figure 1. If one ignores the dotted links in the figure, then the Petri net and the case graph are those created according to the semantics for the primitives as given in the previous section.

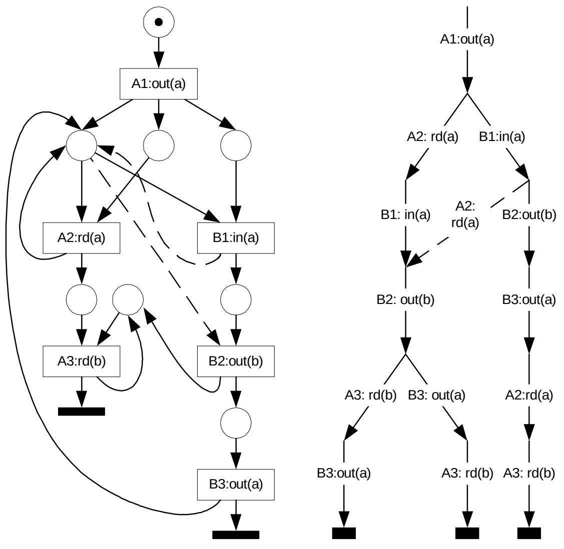

Fig. 1. A Petri Net and case graph for processes A and B.

In Figure 1 the token starts in the initial place, and the only transition that can fire is `A1:out(a)`. When this fires, a token is placed in the three output places connected to the transition. This means that either the transitions `A2:rd(a)` or `B1:in(a)` can fire. If `B1:in(a)` fires then the other cannot fire, because the token is removed from one of its preconditions. This token is replaced when the transition `B3:out(a)` is fired. If `A2:rd(a)` fires, then the precondition tokens are consumed, but the transition is linked to one of its own preconditions. So, a token is reinserted in that place. However, the same rule cannot re-fire because the other precondition does not contain a token any longer. This means that the transition `B1:in(a)` is the only one that can fire, as it is the only transition that has all precondition places filled with a token. The case graph shown in the same figure shows the different ordering of the possible transition firings (of course, the dotted arc has to be ignored).

In Figure 1 the dotted arcs represent the tuple ghosting optimisation. We allow the transition `A2:rd(a)` to fire after the transition `B1:in(a)` fires. This means that the manipulation of a tuple has been suspended in the middle of the operation; Process B has performed the `in` operation and has received the tuple and can continue, but the tuple is not actually removed whilst Process A cannot know that process B has received the tuple. This only occurs when there is the possibility of a synchronisation between the two processes, which happens using the tuple `b`, when Process B inserts it.

From the global perspective, this appears to be incorrect; it allows the reading of a tuple that should have been removed. We will now present the formal semantics of the optimised version, and then show the proof that the two semantics are equivalent.

3.1 Optimized operational semantics

The *optimized* operational semantics of the language is defined by means of another LTS. To this aim, we assume the existence of a set of process *locations*, *Loc*, ranged over by ℓ, where the parallel components of processes can be allocated, and of a distinct location, τ, where evaluated tuples are placed. We denote by $\underline{Loc}$ a disjoint set of *ghost* locations (where *ghost* tuples can be placed) which is in bijection with *Loc* via the operation $\underline{\cdot}$. Finally, we let $LOC = Loc \cup \underline{Loc} \cup \{\tau\}$, ranged over by λ, be the set of all locations. Locations shall be used to model the GUID assigned to processes in the implementation.

The idea is that Linda processes are statically allocated, e.g. distributed over a net of processors, once and for all. The names of locations and the distribution of processes over locations can be arbitrarily chosen. Hence, for any given process P, its distribution is determined by the number of its parallel components, i.e. by the number of occurrences of the parallel operator which are not guarded by any action. For instance, the process $\underline{out}(1) \parallel \underline{out}(2).(\underline{out}(3) \parallel \underline{out}(4))$ has initially two parallel components (although, after the execution of the $\underline{out}(2)$ operation, it is composed of three parallel processes) and can be allocated over, at most, two processors. This means that, as far as distribution is concerned, we have conceptually two different parallel operators and it is convenient to use different notations for them: we shall use $|$ to denote the occurrences of the parallel operator that do not cause distribution of their components, e.g.

those occurrences guarded by some action, and shall still use $||$ for the other occurrences, e.g. (some of) the unguarded occurrences. Obviously, the semantics of $|$ is defined by rules analogues to S2 and to the structural ones.

To manage locations we introduce two new operators: an *allocator* operator $\lambda :: P$, that says that process P is allocated at location λ, and a *location remover* operator $P \setminus \lambda$, that says that location λ (and the process located there) must be removed from P.

The optimized LTS is the triple $(\mathcal{P}_2, \mathcal{L}_2, \longrightarrow_2)$ where:

- $\mathcal{P}_2$, ranged over by P and Q, is the set of closed processes generated by the syntax given in Section 2.1 extended with the following productions

  $P, Q \ ::= \ \ldots \mid P \mid Q \mid \lambda :: P \mid P \setminus \lambda$

  Hence, $\mathcal{P}_2$ also contains the *distributed* versions of processes from $\mathcal{P}_1$.
- $\mathcal{L}_2 \stackrel{\text{def}}{=} \{out(v)@\lambda, rd(v)@\lambda, in(v)@\lambda, stop@\lambda | v \in VAL, \lambda \in LOC\}$ is the set of *labels* (we shall use $\alpha@\lambda$ to range over $\mathcal{L}_2$ and σ over $\mathcal{L}_2^*$).
- $\longrightarrow_2 \subseteq \mathcal{P}_2 \times \mathcal{L}_2 \times \mathcal{P}_2$, called the *transition relation*, is the least relation closed under parallel composition that satisfies the operational rules in Table 4 (again, to give a simpler presentation of the rules, we rely on a *structural relation* defined as the least equivalence relation closed under parallel composition that satisfies the structural rules in Table 4). We shall write $P \xrightarrow{\alpha@\lambda} Q$ instead of $(P, \alpha@\lambda, Q) \in \longrightarrow_2$.

For $\sigma \in \mathcal{L}_2^*$ and $P, Q \in \mathcal{P}_2$, we shall write $P \xrightarrow{\sigma} Q$ to denote that $P = Q$, if $\sigma = \epsilon$, and that $\exists P_1, \ldots, P_{n-1} \in \mathcal{P}_2 : P \xrightarrow{\alpha_1@\lambda_1} P_1 \xrightarrow{\alpha_2@\lambda_2} \ldots P_{n-1} \xrightarrow{\alpha_n@\lambda_n} Q$, if $\sigma = \alpha_1@\lambda_1 \cdot \alpha_2@\lambda_2 \cdot \ldots \cdot \alpha_n@\lambda_n$.

Let us briefly comment on the rules in Table 4. The additional structural laws say that the location remover distributes with respect to parallel composition and that the removal just concerns the location (and the process located there) explicitely named by the operator. The operational rules should be quite explicative. The general idea is as follows. Tuples are initially allocated at location τ. When a tuple located at τ is accessed by an `in` action performed by a process located at ℓ, the tuple becomes a ghost tuple and is relocated at the ghost location $\underline{\ell}$. Whenever a process located at ℓ performs an action or terminates, removal of the ghost tuple that could have been allocated at $\underline{\ell}$ takes place. In particular, rules OS1-OS3 just account for the intentions of processes to perform operations. They define an auxiliary transition relation whose states are (not necessarily closed) processes generated by the extended syntax and whose set of labels is $\{\underline{out(e)@\ell}, \underline{rd(t)@\ell}, \underline{in(t)@\ell}, \underline{stop@\ell} \mid e \in EXP, t \in EXP \cup \underline{VAR}, \ell \in Loc\}$. The remaining rules build upon them. Process termination is modelled by letting $\ell :: nil$ perform the action $\underline{stop@\ell}$ (rule OS2), and, in the presence of a $\underline{stop@\ell}$ action, requiring the removal of ghost tuples at $\underline{\ell}$ (rule OS4). Rule OS5 deals with addition of tuples to the tuple space (located at τ). Rule OS6 says that a `rd` operation can access both tuples in the tuple space and ghost tuples that are not allocated at the location of the process that performs the operation. Rule OS7 says that an `in` operation can access just tuples in the tuple space

Structural Rules

$$P \parallel nil \equiv P \qquad\qquad P \parallel Q \equiv Q \parallel P$$

$$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \qquad\qquad (P \parallel Q) \setminus \underline{\ell} \equiv P \setminus \underline{\ell} \parallel Q \setminus \underline{\ell}$$

$$(\underline{\ell} :: P) \setminus \underline{\ell} \equiv nil \qquad\qquad (\lambda :: P) \setminus \underline{\ell} \equiv \lambda :: P \quad \text{if } \lambda \neq \underline{\ell}$$

Operational Rules

$$\text{OS1}\ \frac{P \xrightarrow{\underline{a}} P'}{\ell :: P \xrightarrow{\underline{a@\ell}} \ell :: P'}$$

$$\text{OS2}\ \ell :: nil \xrightarrow{\underline{stop@\ell}} nil$$

$$\text{OS3}\ \frac{P \xrightarrow{\underline{\alpha@\ell}} P'}{P \parallel Q \xrightarrow{\underline{\alpha@\ell}} P' \parallel Q}$$

$$\text{OS4}\ \frac{P \xrightarrow{\underline{stop@\ell}} P'}{P \xrightarrow{stop@\ell} P' \setminus \underline{\ell}}$$

$$\text{OS5}\ \frac{P \xrightarrow{\underline{out(e)@\ell}} P' \quad \wedge \quad [\![\, e \,]\!] = v}{P \xrightarrow{out(v)@\ell} P' \setminus \underline{\ell} \parallel \tau :: \underline{out}(v)}$$

$$\text{OS6}\ \frac{P \xrightarrow{\underline{rd(t)@\ell}} P' \quad \wedge \quad ([\![\, t \,]\!] = v \vee t \in \underline{VAR}) \quad \wedge \quad \lambda \neq \underline{\ell}}{P \parallel \lambda :: \underline{out}(v) \xrightarrow{rd(v)@\lambda'} P'[v/t] \setminus \underline{\ell} \parallel \lambda :: \underline{out}(v)} \quad \text{where } \lambda' = \begin{cases} \ell \text{ if } \lambda = \tau \\ \lambda \text{ otherwise} \end{cases}$$

$$\text{OS7}\ \frac{P \xrightarrow{\underline{in(t)@\ell}} P' \quad \wedge \quad ([\![\, t \,]\!] = v \vee t \in \underline{VAR})}{P \parallel \tau :: \underline{out}(v) \xrightarrow{in(v)@\ell} P'[v/t] \setminus \underline{\ell} \parallel \underline{\ell} :: \underline{out}(v)}$$

$$\text{OS8}\ \frac{P \equiv Q \quad \wedge \quad Q \xrightarrow{\alpha@\lambda} Q' \quad \wedge \quad Q' \equiv P'}{P \xrightarrow{\alpha@\lambda} P'}$$

Table 4. Optimized Linda Operational Semantics

(i.e., it cannot access ghost tuples). Location removal is actually performed into two steps: first, a location restriction is put and, then, when applying rule OS8, the removal actually takes place by means of the structural relation. The transition labels always refer the location of the process that performs the operation, apart for the label in the conclusion of rule OS6 that, whenever a ghost tuple is accessed, refers the location of such a tuple.

4 Proof of correctness

The actual proof is technically involved, although not conceptually difficult, and can be found in the full paper [13]. In this section, we only provide a sketch.

The two main results can be informally stated as follows:

– each computation from a distributed version of a process P allowed by the optimized semantics can be simulated by a computation from P within the original semantics (Theorem 1);
– each computation from a process P allowed by the original semantics can be simulated by a computation from a distributed version of P within the optimized semantics (Theorem 2).

To simplify the statement of properties, in the rest of this section we shall use P, Q and R to range over $\mathcal{P}_1$ and P_o, Q_o and R_o to range over $\mathcal{P}_2$.

First, it is convenient to fix the allocation function used to distribute the parallel components of processes. To this aim, we assume that $\{l,r\}^* \subseteq \mathit{Loc}$ and use ρ to range over $\{l,r\}^*$. Hence, strings of the form llr and $rllrl$ are valid locations. Now, by relying on locations of the form $\{l,r\}^*$ that can be easily "duplicated" (given a ρ, ρl and ρr are two new different locations), we define an allocation function that, intuitively, for any process $P \in \mathcal{P}_1$ returns its "maximal" distribution: each parallel component is allocated over a different location.

Definition 1. The *allocation function* $\mathcal{L}_\rho : \mathcal{P}_1 \longrightarrow \mathcal{P}_2$ is defined as follows:

$$\begin{array}{rlrl}
\mathcal{L}_\rho(\mathit{nil}) & \stackrel{\text{def}}{=} \rho :: \mathit{nil} & \mathcal{L}_\rho(a.P) & \stackrel{\text{def}}{=} \rho :: a.P \\
\mathcal{L}_\rho(P \mid Q) & \stackrel{\text{def}}{=} \rho :: (P \mid Q) & \mathcal{L}_\rho(P \parallel Q) & \stackrel{\text{def}}{=} \mathcal{L}_{\rho l}(P) \parallel \mathcal{L}_{\rho r}(Q) \\
\mathcal{L}_\rho(P \parallel O) & \stackrel{\text{def}}{=} \mathcal{L}_\rho(P) \parallel \mathcal{T}(O) & \mathcal{L}_\rho(O \parallel P) & \stackrel{\text{def}}{=} \mathcal{T}(O) \parallel \mathcal{L}_\rho(P) \\
\mathcal{T}(O_1 \parallel O_2) & \stackrel{\text{def}}{=} \mathcal{T}(O_1) \parallel \mathcal{T}(O_2) & \mathcal{T}(\underline{out}(v)) & \stackrel{\text{def}}{=} \tau :: \underline{out}(v)
\end{array}$$

where function $\mathcal{T}$ separately allocates all evaluated tuples at location τ.

Correctness will be sketched in the case function $\mathcal{L}_\rho$ (hence, maximal distribution) is used for allocating processes. The proof would proceed similarly also if a different allocation function was used to initially allocate processes from $\mathcal{P}_1$.

We will also use an "inverse" function $\mathcal{C}$ that relates the states of $\mathcal{P}_2$ to those of $\mathcal{P}_1$.

Definition 2. The *cleaning function* $\mathcal{C} : \mathcal{P}_2 \longrightarrow \mathcal{P}_1$ is defined as follows:

$$\begin{array}{rcl} \mathcal{C}(\ell :: P) \overset{\text{def}}{=} P & & \mathcal{C}(\tau :: \underline{out}(v)) \overset{\text{def}}{=} \underline{out}(v) \\ \mathcal{C}(\underline{\ell} :: P) \overset{\text{def}}{=} nil & & \mathcal{C}(P \mid Q) \overset{\text{def}}{=} P \mid Q \\ \mathcal{C}(P_o \parallel Q_o) \overset{\text{def}}{=} \mathcal{C}(P_o) \parallel \mathcal{C}(Q_o) & & \end{array}$$

With abuse of notation, given a label $\alpha@\lambda \in \mathcal{L}_2$ we write $\mathcal{C}(\alpha@\lambda)$ to denote the action part α whenever $\alpha \in \mathcal{L}_1$, and the empty action ϵ otherwise (i.e. whenever $\alpha = stop$). A similar notation shall be used for sequences of labels from $\mathcal{L}_2^*$.

We shall use $\Lambda(P)$ to denote the set of locations occurring in P_o. Formally, function $\Lambda : \mathcal{P}_2 \longrightarrow LOC$ is defined inductively as follows:

$$\begin{array}{ll} \Lambda(nil) = \Lambda(a.P) = \Lambda(\underline{out}(v)) = \Lambda(P_o \mid Q_o) = \emptyset, & \Lambda(P_o \parallel Q_o) = \Lambda(P_o) \cup \Lambda(Q_o), \\ \Lambda(\lambda :: P) = \{\lambda\}, & \Lambda(P_o \setminus \lambda) = \Lambda(P_o) \setminus \{\lambda\}. \end{array}$$

As a matter of notation, we shall use $P_o[\ell'/\ell]$ to denote the term obtained by substituting each occurrence of ℓ in P_o with ℓ'. Finally, we use the notation $\Pi_{\ell_i \in L}\underline{\ell_i} :: \underline{out}(v_i)$ as a shorthand for $\underline{\ell_1} :: \underline{out}(v_1) \parallel \ldots \parallel \underline{\ell_n} :: \underline{out}(v_n)$ (the order in which the operands $\underline{\ell_i} :: \underline{out}(v_i)$ are arranged is unimportant, as $\parallel$ is associative and commutative in both the two operational semantics considered in the paper); and when $n = 0$, this term will by convention indicate nil.

We first outline the proof that the original semantics can simulate the optimized one. To this aim, we introduce the following preorder over traces (i.e. sequences of actions) in $\mathcal{L}_2^*$.

Definition 3. Let $\prec$ be the least preorder relation over $\mathcal{L}_2^*$ induced by the two following laws:

$$\begin{array}{lll} \text{TP1} & \sigma' \cdot rd(v)@\tau \cdot in(v)@\ell \cdot \sigma \prec \sigma' \cdot in(v)@\ell \cdot rd(v)@\underline{\ell} \cdot \sigma & \\ \text{TP2} & \sigma' \cdot \alpha@\lambda \cdot in(v)@\ell \cdot \sigma \prec \sigma' \cdot in(v)@\ell \cdot \alpha@\lambda \cdot \sigma & \text{if } \lambda \neq \ell, \underline{\ell} \end{array}$$

The intuition behind the trace preorder $\prec$ is that if $P_o \xrightarrow{\sigma} Q_o$ and $\sigma' \prec \sigma$ then it also holds that $P_o \xrightarrow{\sigma'} Q_o$, hence σ' can simulate σ. Law TP1 permits exchanging the execution order of two operations accessing the same evaluated tuple in order to avoid accessing ghost tuples. Law TP2 permits exchanging the execution order of operations that are not causally related. Its simple presentation relies on the observation that there cannot be two ghost tuples at the same location, hence if $\mathcal{L}_\rho(P) \xrightarrow{\sigma}\xrightarrow{in(v)@\ell}\xrightarrow{\alpha@\underline{\ell}} Q_o$ then it should be $\alpha = rd(v)$ and we would fall in the case dealt with by law TP1. Note that operations that take place at the same location can never be swapped because there is no way to ascertain when they are causally independent.

Let us now introduce some useful notations. We shall write $a \notin s$ to denote that there are not s_1, s_2 such that $s = s_1 a s_2$ ($\alpha@\lambda \notin \sigma$ has a similar meaning). Moreover, we write $g(\sigma)$ to denote the number of occurrences in σ of locations of $\underline{Loc}$ ('g' stands for 'ghost').

Intuitively, sequences of labels $\sigma \in \mathcal{L}_2^*$ such that $g(\sigma) > 0$ are obtained from sequences of operations that also access ghost tuples and, hence, cannot be mimicked in the original semantics. We will show that, however, for each σ with $g(\sigma) > 0$ it is possible to find a σ' such that (*i*) $g(\sigma') = 0$, hence σ' is obtained from a sequence of operations that can also be performed according to the original semantics, and (*ii*) $\sigma' \prec \sigma$, hence σ' simulates σ according to the optimized semantics.

The laws of the trace preorder can be used to reduce the number of ghost tuples accessed during a computation. The following crucial property gives a method for transforming a generic computation in an equivalent one (i.e. with the same final state) that corresponds to a sequence of operations that never access ghost tuples.

Proposition 1. $\mathcal{L}_\rho(P) \xrightarrow{\sigma} Q_o \xrightarrow{rd(v)@\underline{\ell}} R_o$ implies that there are σ_1 and σ_2 such that $\sigma = \sigma_1 \cdot in(v)@\ell \cdot \sigma_2$ and $\alpha@\ell \notin \sigma_2$. Moreover, if $\alpha@\underline{\ell} \notin \sigma_2$ then there is ℓ' such that $\mathcal{L}_\rho(P) \xrightarrow{\sigma_1}\xrightarrow{\sigma_2}\xrightarrow{rd(v)@\ell'}\xrightarrow{in(v)@\ell} R_o$.

By repeatedly applying the previous property we have that

Proposition 2. $\mathcal{L}_\rho(P) \xrightarrow{\sigma} Q_o$ implies that there is $\sigma' \prec \sigma$ such that $\mathcal{L}_\rho(P) \xrightarrow{\sigma'} Q_o$ and $g(\sigma') = 0$.

We now relate the single transitions of the optimized semantics that do not access ghost tuples to the transitions of the original semantics. Notice that the states of the optimized semantics may contain ghost tuples.

Proposition 3. For all $L \subseteq \Lambda(\mathcal{L}_\rho(P))$, $\mathcal{L}_\rho(P) \parallel \Pi_{\ell_i \in L} \underline{\ell_i} :: \underline{out}(v_i) \xrightarrow{a@\ell} Q_o$ implies that there are R, ρ' and $L' \subseteq \Lambda(\mathcal{L}_{\rho'}(R))$ such that $P \xrightarrow{a} R$ and $Q_o \equiv \mathcal{L}_{\rho'}(R) \parallel \Pi_{\ell_i \in L'} \underline{\ell_i} :: \underline{out}(v_i)$.

Proposition 4. For all $L \subseteq \Lambda(\mathcal{L}_\rho(P))$, $\mathcal{L}_\rho(P) \parallel \Pi_{\ell_i \in L} \underline{\ell_i} :: \underline{out}(v_i) \xrightarrow{stop@\ell} Q_o$ implies that there is R such that $P \equiv R$ and $Q_o \equiv \mathcal{L}_\rho(R) \parallel \Pi_{\ell_i \in L \setminus \{\ell\}} \underline{\ell_i} :: \underline{out}(v_i)$.

We can generalize the previous two properties to sequences of transitions.

Proposition 5. $\mathcal{L}_\rho(P) \xrightarrow{\sigma} Q_o$ and $g(\sigma) = 0$ imply that there are P', ρ', $L \subseteq \Lambda(\mathcal{L}_{\rho'}(P'))$ and v_i such that $P \xrightarrow{\mathcal{C}(\sigma)} P'$ and $Q_o \equiv \mathcal{L}_{\rho'}(P') \parallel \Pi_{\ell_i \in L} \underline{\ell_i} :: \underline{out}(v_i)$.

Finally, from Propositions 2 and 5, we get that the original semantics can simulate the optimized one. Formally,

Theorem 1. $\mathcal{L}_\rho(P) \xrightarrow{\sigma} Q_o$ implies that there are $\sigma' \prec \sigma$ and P' such that $P \xrightarrow{\mathcal{C}(\sigma')} P'$ and $\mathcal{C}(Q_o) \equiv P'$.

Now, we outline the proof that the optimized semantics can simulate the original one (Theorem 2). First, we need to formalize the idea that locations can be arbitrarily chosen and that distributed processes that only differ for the names of their locations behave similarly. The important point here is that the allocation function does not preserve the structural equivalence. Indeed, the allocation of two structurally equivalent processes gives rise to two new processes that are not structurally equivalent. However, structural equivalence can be recovered by appropriately renaming the locations of one of the two processes by means of a one-to-one function. The crucial properties are

Proposition 6. If $P \equiv P'$ can be proved without using the first structural law, then there is a one-to-one function $\phi : \Lambda(\mathcal{L}_{\rho'}(P')) \longrightarrow \Lambda(\mathcal{L}_\rho(P))$ such that $\mathcal{L}_\rho(P) \equiv \phi(\mathcal{L}_{\rho'}(P'))$.

Proposition 7. Let $\phi : \Lambda(\mathcal{L}_{\rho'}(P')) \longrightarrow \Lambda(\mathcal{L}_\rho(P))$ be a one-to-one function. If $P \equiv P'$ can be proved without using the first structural law, $\phi(\mathcal{L}_{\rho'}(P')) \equiv \mathcal{L}_\rho(P)$ and $\Lambda(\mathcal{L}_{\rho'}(P')) = \Lambda(\mathcal{L}_{\rho'}(Q'))$, then there is Q such that $Q \equiv Q'$ and $\phi(\mathcal{L}_{\rho'}(Q')) \equiv \mathcal{L}_\rho(Q)$.

Now, by exploiting the above properties, we are able to relate the transitions of the original semantics to those of the optimized one. Notice that the states of the optimized semantics may contain ghost tuples.

Proposition 8. For all $L \subseteq \Lambda(\mathcal{L}_\rho(P))$, $P \xrightarrow{a} Q$ implies that there are R, ℓ and $L' \subseteq \Lambda(\mathcal{L}_\rho(R))$ such that $\mathcal{L}_\rho(P) \parallel \Pi_{\ell_i \in L} \underline{\ell_i} :: \underline{out}(v_i) \xrightarrow{a@\ell} \mathcal{L}_\rho(R) \parallel \Pi_{\ell_i \in L'} \underline{\ell_i} :: \underline{out}(v_i)$ and $R \equiv Q$.

By generalizing the previous property to nonempty sequences of transitions, we get that the optimized semantics can simulate the original one. Formally,

Theorem 2. $P \xrightarrow{s} Q$ implies that there are R_o and σ such that $\mathcal{L}_\rho(P) \xrightarrow{\sigma} R_o$, $s = \mathcal{C}(\sigma)$ and $\mathcal{C}(R_o) \equiv Q$.

5 Conclusion

We have described a tuple ghosting optimisation that allows tuples to be still used as the results of non-destructive tuple space accesses once they have been destructively removed. The motivation for tuple ghosting has been briefly outlined, as have some practical results from a prototype system demonstrating the advantage of the approach.

The operational semantics of the original Linda and of the version with the optimisation are illustrated. Using these operational semantics, we have presented a sketch of the formal proof of the tuple ghosting optimisation, and shown that the optimisation does not alter the semantics of the primitives from a programmers' perspective. This has been achieved by proving that the optimised semantics can simulate the original semantics, and that a sequence of transitions from the optimised semantics can be mimicked by a sequence of transitions from the original semantics.

Acknowledgements This work has been partially supported by MURST projects SALADIN and TOSCA.

References

1. A. Rowstron. Optimising the Linda `in` primitive: Understanding tuple-space runtimes. In J. Carroll, E. Damiani, H. Haddad, and D. Oppenheim, editors, *Proceedings of the 2000 ACM Symposium on Applied Computing*, volume 1, pages 227–232. ACM Press, March 2000.
2. Sun Microsystems. Javaspace specification. available at: `http://java.sun.com/`, 1999.
3. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
4. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
5. P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–366, 1998.
6. P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.
7. A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999. Special Issue on Coordination Mechanisms and Patterns for Web Agents.
8. A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.
9. J. Carreria, L. Silva, and J. Silva. On the design of Eilean: A Linda-like library for MPI. Technical report, Universidade de Coimbra, 1994.
10. N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 114–125. MIT Press, 1990.
11. R. De Nicola and R. Pugliese. Linda based applicative and imperative process algebras. *Theoretical Computer Science*, 238(1-2):389–437, 2000.
12. G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Dep. of Computer Science, Aarhus University, Denmark, 1981.
13. R. De Nicola, R. Pugliese, and A. Rowstron. Proving the correctness of optimising destructive and non-destructive reads over tuple spaces. Technical Report available at: `http://music.dsi.unifi.it/`, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2000.