Parallel Searching with Multisets-as-Agents

Jean-Marc Andreoli
ECRC - Munich - Germany
E-mail: jeanmarc@ecrc.de

Paolo Ciancarini*
University of Bologna - Italy
E-mail: cianca@di.unipi.it

Remo Pareschi
ECRC - Munich - Germany
E-mail: remo@ecrc.de

Abstract

LO (for Linear Objects) is a concurrent language which allows the programmer to specify, at an abstract level, the behavior of a system of communicating agents. *LO* amalgamates two models of parallel computing: (*i*) multiset rewriting, where elements of multisets are tuples; (*ii*) "actors", where actors are agents capable of self-replication, termination and explicit message passing. We illustrate here the expressive power of *LO* for expressing parallel search algorithms.

1 Introduction

LO (for Linear Objects) [AP91, ACP92] is a concurrent language which allows the programmer to specify, at an abstract level, the behavior of a system of communicating agents. Agent states are represented as multisets, evolving in terms of transitions which can be either *transformations* (from one state to one new state), *creations* (from one state to two or more new states), or *terminations* (from one state to no state at all). Interagent communication takes the form of *broadcasting*, and is implemented in the following way: the intersection of the agents' states, called the *forum*, is left initially unspecified; the forum gets incrementally instantiated as a side effect of performing state transitions inside agents so that, whenever a step of instantiation takes place, new elements appear in the forum, which are then copied in the local area of each agent.

Thus, *LO* effectively amalgamates the two following models of parallel computing: (*i*) multiset rewriting [BCLM88, BC91, CG90, Mes92]; (*ii*) "actors" [AH87, YSH+90], where actors are agents capable of self-replication, termination and explicit message passing. The gain in expressive power achieved in this way is remarkable: *LO* agents are dynamically reconfigurable entities characterized by different levels of concurrency (internal and external) and by corresponding forms of communication. At a theoretical level, this is accounted for by a rigorously defined operational semantics given in terms of the proof theory of Linear Logic, a logic recently introduced to provide a formal framework for the study of concurrency [Gir87]. This strong theoretical foundation has the practical consequence that techniques for optimization via static analysis can be defined in the abstract and uniform domain of Linear Logic proof trees, so that the salient features of the computation can be extracted without having to filter out irrelevant implementation details (a number of powerful optimization techniques have been defined on this basis [AP92]).

LO is finding applications in distributed simulations [ACP92] and in extending object-oriented languages with mechanisms for composing and coordinating objects [BAP92]. Parallel search algorithms provide yet a further topic, and they will be used in this paper to illustrate the expressive power of the model, describing work already partially reported elsewhere [AP91, APB91]. Our claim is that *LO* offers the possibility of writing very elegant and natural parallel programs implementing sophisticated search strategies, in a way that would not be possible in a model based on multiset rewriting or actors alone. After a section on syntax and operational semantics, we shall illustrate the expressive power of the language through two examples: a massively parallel chart parser and a classical operations research algorithm.

*This author has been partially supported by CNR *Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo*, and by MURST

2 *LO*

2.1 Programs, Resources, Proofs

LO programs are built from the following operators: "par" (written @), "with" (written &), "becomes" (written <>-) and "top" (written $\top$). We assume an initial (possibly infinite) set of atomic formulae A from which we can recursively define two classes of expressions: "resource formulae" R and "program formulae" P.

$$R = A \mid R \mathbin{@} R \mid R \mathbin{\&} R \mid \top$$
$$P = A \mathbin{<>-} R \mid A \mathbin{@} P$$

A "program" is a set of program formulae and a "context" is a finite multiset of resource formulae. An *LO* sequent is a pair written as $\mathcal{P} \vdash \mathcal{C}$ where $\mathcal{P}$ is a program and $\mathcal{C}$ is a context.

A "proof" is a tree structure whose nodes are labeled with *LO* sequents. By convention, a proof tree is graphically represented with its root at the bottom and growing upward. Its branches are obtained as instances of the following inference figures.

- Decomposition inference figures

$$[\,@\,]\ \frac{\mathcal{P} \vdash \mathcal{C}, R_1, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \mathbin{@} R_2} \qquad [\top]\ \frac{}{\mathcal{P} \vdash \mathcal{C}, \top} \qquad [\,\&\,]\ \frac{\mathcal{P} \vdash \mathcal{C}, R_1 \qquad \mathcal{P} \vdash \mathcal{C}, R_2}{\mathcal{P} \vdash \mathcal{C}, R_1 \mathbin{\&} R_2}$$

- Progression inference figure

$$[\,\mathbin{<>-}\,]\ \frac{\mathcal{P} \vdash \mathcal{C}, R}{\mathcal{P} \vdash \mathcal{C}, A_1, \ldots, A_n} \ \text{if } (A_1 \mathbin{@} \cdots \mathbin{@} A_n \mathbin{<>-} R) \in \mathcal{P}$$

In these figures, $\mathcal{P}$ and $\mathcal{C}$ denote, respectively, a program and a context. R, R_1, R_2 denote resource formulae and the expression $\mathcal{C}, R$ denotes the context obtained as the multiset union of $\mathcal{C}$ and the singleton R. Notice that, by definition, the elements of a multiset are not ordered. Therefore, the order of the atoms in the left-hand side of a program formula is not relevant.

2.2 Operational Interpretation of the Inference Figures

Read bottom-up, a proof with root sequent $\mathcal{P} \vdash \mathcal{C}$ gives us a static representation (a "snapshot") of the overall evolution of a system of agents working on the initial set of resources $\mathcal{C}$ under program $\mathcal{P}$. Each branch of proof represents the evolution of one agent: the nodes are the agent states while the edges are the state transitions. The open leaves are the agents still living at the time of the snapshot.

Program formulae define the allowed state transitions. They can be thought of as composed of two parts, an *input* part (left-hand side of the symbol <>-) and an *output* part (right-hand side of the symbol <>-), implementing, respectively, the consumption and the production of resources from/to the agent's state. However, the operation of producing new resources is here more complex than in standard multiset rewriting, and may in fact involve the creation of new agents, or the termination of existing ones. Indeed, when the output part of a program formula is produced into a context by application of the progression inference figure [<>-], it is recursively decomposed by application of the decomposition inference figures, which will either terminate the agent (inference figure [$\top$]) or create a new agent by cloning (inference figure [&]) or simply add new components in the context of the agent (inference figure [@]).

We say that a proof is "normal" if the progression inference figure is never applied to a context which contains non atomic resource formulae. It is easy to show that any proof can be mapped into a normal proof by simple permutations of the application of the inference figures. In fact, in the rest of this paper, we consider only normal proofs. This means that when a program formula is selected for application by the progression inference figure, its output part is *immediately* and *fully* decomposed by recursive application of the decomposition inference figures. In other words, program formulae constitute critical sections of code which cannot be interrupted.

However, we have not yet explained how broadcast communication among agents is realized: as a matter of fact, this is not done in terms of an explicit inference figure but is instead achieved as a side

effect of how proof trees are constructed. We detail the mechanism of proof construction in the next section.

2.3 Proof Construction

A "program call" is a pair $\langle \mathcal{P}; R \rangle$ consisting of an *LO* program $\mathcal{P}$ and a resource formula R.

Definition 1 *A target proof for a program call $\langle \mathcal{P}; R \rangle$ is a proof such that its root is a sequent of the form $\mathcal{P} \vdash \mathcal{C}, R$, where $\mathcal{C}$ is a context containing only atoms.*

In other words, target proofs are searched in such a way that the context at their root node may *properly contain* the initial resource of the program call.

We consider two proof construction mechanisms. Let Π be any *LO* proof.

- Expansion:
  Let ν be an open leaf of Π whose sequent matches the lower sequent of an inference figure. Let Π' be the proof obtained by expanding Π at node ν with branches to new open leaves labeled with the upper sequent(s) of the selected inference figure. We write $\Pi \Longrightarrow_e \Pi'$
- Instantiation:
  Let Π' be the proof obtained by adding an occurrence of a given atom to the context at each node of Π. We write $\Pi \Longrightarrow_i \Pi'$.

Clearly, these proof construction mechanisms are sound in the following sense:

Theorem 1 *If Π is a target proof for a given program call, and $\Pi \Longrightarrow_e \Pi'$ or $\Pi \Longrightarrow_i \Pi'$ then Π' is also a target proof for that program call.*

Definition 2 *A proof construction sequence is a sequence of proofs $\Pi_1, \ldots, \Pi_n$ such that*

$$\forall k = 1, \ldots, n-1 \quad \begin{cases} \Pi_k \Longrightarrow_e \Pi_{k+1} \\ \text{or} \\ \Pi_k \Longrightarrow_i \Pi_{k+1} \end{cases}$$

The trivial proof Π_o reduced to the single node $\mathcal{P} \vdash R$ is obviously a target proof for the program call $\langle \mathcal{P}; R \rangle$. Hence, by application of theorem 1, so is any proof Π such that there exists a proof construction sequence leading from Π_o to Π. Furthermore, it can be shown that the proofs obtained by this method are all the possible target proofs for the program call, so that the two construction mechanisms introduced above are also complete.

In the agent-oriented computational interpretation of proof construction, an expansion step corresponds to an agent state transition whereas an instantiation step corresponds to a form of communication by broadcasting; indeed, the atom which is added to all the nodes in an instantiation step acts as a message broadcast to all the living agents in the system.

2.4 Control of Broadcast Communication

Unfortunately, the two mechanisms of expansion (i.e. state transition) and instantiation (i.e. broadcast communication) are here completely decoupled: indeed, it can be shown that any expansion step permutes with any instantiation step. In order to allow a form of synchronization between the two mechanisms, required in most applications, we introduce a pragmatic tool which gives the programmer some control over the order of execution of expansion steps and instantiation steps in proof constructions (we loose completeness here).

Let ^ be a special symbol, called the "broadcast" marker, which can be used to prefix any atom in the input part of a program formula. Consider then the following program formula:

$$p \circledast {}^\wedge a \texttt{ <>- } r$$

$$\Pi_1 \;=\; \vdash (p \,@\, q) \,\&\, r \,\&\, s$$

$$\Pi_2 \;=\; [\,\&\,]\,\dfrac{[\,@\,]\,\dfrac{\vdash p, q}{\vdash p \,@\, q} \quad [\,\&\,]\,\dfrac{\vdash r \quad \vdash s}{\vdash r \,\&\, s}}{\vdash (p \,@\, q) \,\&\, r \,\&\, s}$$

$$\Pi_3 \;=\; [\,\&\,]\,\dfrac{[\,@\,]\,\dfrac{\vdash p, q, a}{\vdash p \,@\, q, a} \quad [\,\&\,]\,\dfrac{\vdash r, a \quad [\,<>\text{-}\,]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash s, a}}{\vdash r \,\&\, s, a}}{\vdash (p \,@\, q) \,\&\, r \,\&\, s, a}$$

$$\Pi_4 \;=\; [\,\&\,]\,\dfrac{[\,@\,]\,\dfrac{[\,<>\text{-}\,]\,\dfrac{[\top]\,\dfrac{}{\vdash \top, q}}{\vdash p, q, a}}{\vdash p \,@\, q, a} \quad [\,\&\,]\,\dfrac{[\,<>\text{-}\,]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash r, a} \quad [\,<>\text{-}\,]\,\dfrac{[\top]\,\dfrac{}{\vdash \top}}{\vdash s, a}}{\vdash r \,\&\, s, a}}{\vdash (p \,@\, q) \,\&\, r \,\&\, s, a}$$

Figure 1: A subsequence of a regular proof construction sequence

This means that, to apply this program formula in an expansion step using the progression inference figure, the atom p (unmarked) must be found in the context of the selected node, while the atom a is added to this context by performing beforehand an instantiation step adding a to all the nodes of the proof. Except in this situation, no other instantiation steps are allowed. A proof construction sequence satisfying this requirement is called "regular". In the rest of this paper, we consider only regular proof construction sequences, and we take the phrase "proof construction" to mean "regular proof construction".

Definition 3 *Let Π, Π' be proofs. We write $\Pi \implies \Pi'$ if there exists a (regular) proof construction sequence from Π to Π'.*

Clearly, the relation $\implies$ is reflexive transitive. Consider for example the following *LO* program

$$\begin{array}{l} p \,@\, a \;<>\text{-}\; \top \\ q \,@\, a \;<>\text{-}\; q1 \\ r \,@\, a \;<>\text{-}\; \top \\ s \,@\, {}^\wedge a \;<>\text{-}\; \top \end{array}$$

Then, as the reader may easily verify,

$$\Pi_1 \implies \Pi_2 \implies \Pi_3 \implies \Pi_4$$

where Π_1, Π_2, Π_3 and Π_4 are the proofs in Fig. 1. (In the figure, the program is omitted from the left-hand side of the sequents.) The step between Π_2 and Π_3 consists of an instantiation step (broadcasting a), prior to an expansion step using the progression inference figure with the fourth program formula on the rightmost open leaf of Π_2. This instantiation step is indeed allowed by the presence of the broadcast marker in the program formula used.

3 Massively Parallel Chart Parsing

The example we provide here is a particularly interesting case of distributed problem solving which illustrates well the use of local resource consumption in the context of broadcast communication. The problem we address specifically is massively parallel parsing, a topic which has attracted the interest of several researchers in the concurrent object-oriented programming community [NT90, YO88], as well as

in the connectionist community [WP85]; on the other hand the problem-solving technique we employ here can be fruitfully generalized to more complex examples, like distributed expert systems operating on highly complex domains, where different experts are required to work independently on shared data, feeding back different outputs which all need to be taken in consideration for the final solution of a given problem.

The program we describe amounts to a concurrent implementation of the Earley's algorithm for context-free parsing [Ear70] and draws much in the spirit of the active chart parsing methodology [Kay80], where incomplete phrasal subtrees are viewed as agents consuming already completed elements to produce other (complete or incomplete) subtrees. However, in our case even the rules of the grammar and the entries of the lexicon act as independent units directly partaking in the computation. Moreover, as distinct from the usual sequential formulations of chart parsing, here no superimposed scheduler is in charge of the task of feeding incomplete subtrees with complete ones; instead, incomplete elements behave as truly active decentralized computational units which get their information from the forum, where finished subtrees are told as soon as they have been found. But we must preserve the fact that, once a subtree is completed, this information must be broadcast to *all* the active agents which can make use of it; indeed, in the case of ambiguous grammars, the number of such agents may be greater than one, thus leading to different parses for the same string. Local consumption neatly deals with this problem.

3.1 The Program

We view parsing as being performed by four top-level agents, a string scanner, a grammar, a dictionary and a creator of new subtrees. This is expressed by the following method, which contains in its head a single literal `parse(I,S)`, where `I` is the input string and `S` is the symbol of the grammar defining the set of strings with respect to which we want to test membership of `I`.

```
parse(Input,Symbol) <>-
        grammar & dictionary &
        scanner(Input,Symbol) & create_tree.
```

The *scanner* agent, defined in the methods in Fig. 2, performs the two following actions:

- It keeps popping words from the input and producing `pos(N)` and `word(W,N)` messages where
  - a `pos(N)` message supplies the information that position `N` has been reached in the input;
  - a `word(W,N)` message supplies the information that there is a word `W` between positions `N` and `N+1` in the input.

  Positions are encoded as integers in the "successor" notation.
- Upon reaching the end of the input string, it sends a `seek(0,S)` message, where `S` is the targeted grammar symbol, and then reduces itself into an agent whose sole task is that of retrieving answers. This is simply done by waiting for trees covering the whole input string with symbol `S` to appear in the forum; the structure `T` with which any of such trees has been represented is then explicitly added as an answer.

The *grammar* and the *dictionary* agent expand, respectively, into a set of grammatical rules and of lexical entries, each originating a different agent; a sample dictionary and grammar[1] are given in Fig. 3. Notice that the grammar is an ambiguous one. The behavior of lexical entries and rule agents is defined in terms of the methods in Fig. 4. Lexical entry agents accept as messages words with which they match and send back corresponding *complete* preterminal trees, labeling the given word with a preterminal symbol. On the other hand, rule agents consume `seek(N,S)` messages *together* with `pos(N)` messages, if the sought grammar symbol `S` corresponds to their own left-hand side symbol; in this case, they issue

[1]The symbol `==>` appearing in the grammar rules is not a primitive of *LO* but simply a convenient infix notation for a binary term constructor.

```
scanner(I,S) <>-
        scan(I,0) @ target(S).

scan([W|I],N) @ ^pos(N) @ ^word(W,N) <>-
        scan(I,s(N)).

scan([],N) @ target(S) @ ^seek(0,S) <>-
        wait(N,S).

wait(N,S) @ ctree(0,N,S,T) @ ^answer(T) <>-
        wait(N,S).
```

Figure 2: Methods for scanning

```
grammar <>-
        s ==> [np,vp] &
        np ==> [det,n] &
        np ==> [pn] &
        np ==> [np,pp] &
        vp ==> [tv,np] &
        vp ==> [vp,pp] &
        pp ==> [prep,np].
```

```
dictionary <>-
        entry(a,det) &
        entry(robot,n) &
        entry(telescope,n) &
        entry(terry,pn) &
        entry(saw,tv) &
        entry(with,prep).
```

Figure 3: A grammar and a dictionary

```
entry(W,S) @ word(W,N) @ ^ctree(N,s(N),S,S-W) <>-
        entry(W,S).

 (S ==> Ss) @ seek(N,S) @ pos(N) @ ^new(N,N,S,Ss,S) <>-
       (S ==> Ss).
```

Figure 4: Methods for lexical entries and rules

```
create_tree @ new(M,N,S,[],T) @ ^ctree(M,N,S,T) <>-
        create_tree @ ctree(M,N,S,T).

create_tree @ new(M,N,S,[S1|Ss],T) @ ^seek(N,S1) <>-
        create_tree & itree(M,N,S,S1,Ss,T).

itree(M,N,S,S1,Ss,T) @ ctree(N,P,S1,T1) @ ^new(M,P,S,Ss,T-T1) <>-
        itree(M,N,S,S1,Ss,T).
```

Figure 5: Creation and completion of trees

back a message for the creation of a new agent encoding an *incomplete* (empty) tree. Crucial is here the fact that the consumption by rule agents of `seek/2` messages must be concomitant with the consumption of matching (in the sense of being characterized by the same integer argument) `pos/1` messages; indeed, this correctly ensures that a rule agent can produce no more than one empty incomplete tree for any position of the input string, given that, for any `N`, it will be able to consume no more than one `pos(N)` message. In this way, we prevent the possibility of infinite loops of the left-recursive kind deriving from rules like the fourth and the sixth one in the grammar of Fig. 3; furthermore, we block the possibility of redundant analyses. This will be illustrated in describing a sample run of the parser further on in this section.

Creating and completing new trees is accounted for in terms of the methods in Fig. 5. The top-level `create_tree` agent consumes messages of the form `new(M,N,S,Ss,T)` where `M` and `N` are, respectively, the two string positions spanned by the new tree to be created, `S` is the root of the tree, `Ss` is a list of symbols corresponding to the roots of the complete subtrees which are still needed in order to make this tree complete, and `T` is the representation associated with the tree itself. It then deterministically chooses between the following two actions:

- in case the list `Ss` is empty, it sends a message `ctree(M,N,S,T)` to signal that a complete tree with root `S` and representation `T` has been found between positions `M` and `N`;
- in case the list `Ss` is of the form `[S1|Ss1]`, it sends a message of the form `seek(N,S1)` and then creates an incomplete tree agent of the form `itree(M,N,S,S1,Ss1,T)`.

As for incomplete tree agents of the form `itree(M,N,S,S1,Ss,T)`, they consume complete trees of the form `ctree(N,P,S1,T1)` to produce messages of the form `new(M,P,S,Ss,T-T1)`. Thus, requests for the creation of new trees can come either from rule agents as answers to `seek/2` messages, or from incomplete tree agents; in the former case such requests can be thought of as leading to the formulation of further hypotheses which need to be verified in order to satisfy a certain initial hypothesis (this is known as step of *prediction* in the usual formulations of the Earley algorithm), while in the second case they follow from having progressed "one step" in the verification of a certain hypothesis (this is known as a step of *completion*). Fig. 6 shows the flow of information among the agents. Agents are represented in square boxes and messages in round boxes (only their topmost functor is displayed). An arrow from an agent to a message (resp. from a message to an agent) means that the agent produces (resp. consumes) the message. Furthermore, we make use of a thicker arrow to explicitly connect the `create_tree` agent with the agents it creates.

3.2 A Sample Run

Let us now briefly consider a sample run of the parser. Assuming the grammar and the lexicon in Fig. 3, consider the goal

```
?- parse([terry,saw,a,robot,with,a,telescope],s)
```

After running the parser, the following two answers, corresponding to the two parses of the input sentence, will be found in the global context.

```
answer(
   s-(np-(pn-terry))
    -(vp-(tv-saw)
        -(np-(np-(det-a)-(n-robot))
            -(pp-(prep-with)
                -(np-(det-a)
                    -(n-telescope)))))).
```

```
answer(
   s-(np-(pn-terry))
    -(vp-(vp-(tv-saw)
            -(np-(det-a)-(n-robot)))
        -(pp-(prep-with)
            -(np-(det-a)
                -(n-telescope))))).
```

These two answers originate from the fact that the same complete trees can be consumed by several agents encoding different incomplete trees; specifically, the agents encoded as

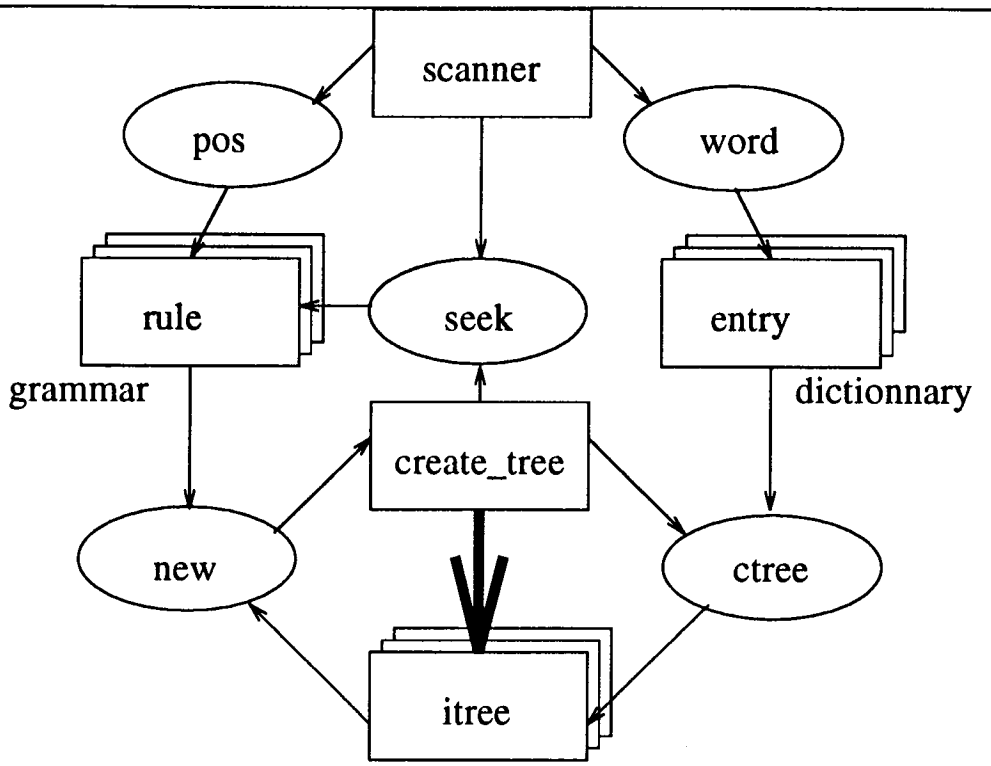

Figure 6: The flow of information

```
itree(1,2,vp,np,[],vp-(tv-saw))

itree(2,2,np,np,[pp],np)
```

will both consume the complete tree

```
ctree(2,4,np,(np-(det-a)-(n-robot)))
```

Furthermore, the agents encoded as

```
itree(1,4,vp,pp,[],(vp-(vp-(tv-saw)-(np-(det-a)-(n-robot)))))

itree(2,4,np,pp,[],(np-(np-(det-a)-(n-robot))))
```

will both consume the complete tree

```
ctree(4,7,pp,(pp-(prep-with)-(np-(det-a)-(n-telescope))))
```

As a consequence, we end up with two different analyses for the substring *saw a robot with a telescope.* On the other hand, notice that the rules whose left-hand side symbol is **np** will receive in the course of parsing more than one **seek(2,np)** message to create empty trees with root **np** and starting position **2**; however, any of such rules will never create more than one of such trees, as **seek/2** messages must be consumed together with matching **pos/1** messages, and any rule will be able to consume at most one **pos(2)** message. Thus, both redundant analyses and infinite loops deriving from left-recursion are in this way avoided. This approach to enforcing redundancy checking is quite simple and elegant and comes natural in a decentralized, agent-oriented style of programming; it can be contrasted with the more usual way of enforcing it, which is obtained by explicitly comparing newly created trees with previously existing ones. We have implemented a more refined version of the parser where our approach to redundancy checking is generalized to the case of grammars with complex grammar symbols (e.g. unification grammars).

4 An Operations Research Algorithm

4.1 The "Bagger" Problem

The example we now show illustrates the power of broadcasting combined with multiset rewriting in solving classical operations research problems. Bagger [Win84] is a problem in which a robot has to pack into bags several grocery items. The items have different weights and have to be orderly packed into the minimum number of bags. The bags are all similar: they have the same maximum capacity (in terms of weight). Items are classified in several kinds: each kind has a different weight. The problem consists of using the minimum number of bags, provided that each bag is filled in an ordered way: at the bottom the heavier items, at the top the lighter ones. A new bag is needed only when no already existing bag has a residual capacity sufficient for the remaining items of the heaviest kind.

4.2 The Program

The items are distributed over several ordered weight classes. A weight class is described by the following atoms:

```
weight(W), nb_item(I), next(WN)
item(X1), ..., item(Xn)
```

`I` is the total number of items in the class. `W` is the weight associated with the class. Its items are identified by `X1`, ..., `Xn` (and are all of weight `W`). `WN` is the highest weight class below `W`.

An atom `fill(B,C,P)` represents an activity which attempts to fill the bag `B` with residual capacity `C` at position `P` (a position in a bag is 1 at the bottom and grows upward). Several `fill/3` atoms can compete for the items within the same weight class and `nb_fill(N)` holds their number. When the capacity of a `fill/3` atom becomes lower than the weight of the class in which it is living, the atom is moved to the next class, which, by convention, has a lower weight (2nd method in Fig. 7). The `fill/3` atoms are also moved to the next class when a class becomes empty (3rd method in Fig. 7). Notice that each move of a `fill/3` atom is acknowledged (message `ack/1` in the 5th and 6th method of Fig. 7). This avoids creation of a new bag (see below) when a moved bag hasn't yet reached its destination class.

When no `fill/3` atom is present in the heaviest class (characterized by the atom `heaviest`), if this class is not empty, then the creation of a new bag is requested (message `create_bag`) and this bag starts to be filled, else, the class is terminated and the next class becomes the heaviest (Fig. 8). When a `fill/3` atom reaches the class of weight 0 (characterized by the atom `end_weight`), the corresponding bag is simply closed (message `close/2` in the last method of Fig. 7).

Other agents, not described here for simplicity purpose, are in charge of processing `create_bag(B)` requests and collecting the messages of the form `put(X,B,P)`, which requests putting the item `X` at position `P` in bag `B`. Notice that the bag identifiers `B` are here constants generated at runtime. Indeed, by convention, when applying an *LO* method, the variables of the head of the method which occurs only in broadcast messages (e.g. `B` in the 1st method of Fig. 8) are automatically instantiated with fresh constants generated dynamically. These constants may be used as specific addresses (here for bags) for implementing point to point communication using broadcast.

Finally, more methods are needed for creating and sorting the different weight classes and placing each item in the corresponding class. A typical query specifies the list of items with their weight, and the maximal capacity of the bags. The answer then appears as a list of bags with their content.

4.3 Discussion

The solution described above to the bagger problem shows the expressive power of parallel rule-based object oriented programming in *LO*. The different class weights (as well as the other objects omitted in the description above) are agents whose behavior is defined by sets of production rules. Agent are both internally and externally parallel: there are several concurrent threads inside each weight class (each `fill/3` atom can be considered a thread by itself, but also the `create_bag` request generator, and the

```
fill(B,C,P) @ weight(W) @ {C >= W} @ item(X) @ ^put(X,B,P)
@ nb_item(N) @ {C1 is C-W, P1 is P+1, N1 is N-1} <>-
  fill(B,C1,P1) @ weight(W) @ nb_item(N1).

fill(B,C,P) @ weight(W) @ {C < W} <>-
  weight(W) @ move_fill(B,C,P).

nb_item(0) @ fill(B,C,P) <>- nb_item(0) @ move_fill(B,C,P).

move_fill(B,C,P) @ next(W) @ ^start_fill(W,B,C,P) <>-
  next(W) @ wait_fill(W).

wait_fill(W) @ ack(W) @ nb_fill(N) @ {N1 is N-1} <>-
  nb_fill(N1).

start_fill(W,B,C,P) @ weight(W) @ nb_fill(N) @ {N1 is N+1}
@ ^ack(W) <>-
  weight(W) @ fill(B,C,P) @ nb_fill(N1).

start_fill(0,B,_,P) @ end_weight @ ^close(B,P) <>-
  end_weight.
```

Figure 7: Methods for filling bags

```
nb_fill(0) @ item(X) @ heaviest @ ^create_bag(B) <>-
  wait_bag(B) @ item(X) @ heaviest.

wait_bag(B) @ new_bag(B,C) <>- nb_fill(1) @ fill(B,C,1).

nb_fill(0) @ nb_item(0) @ heaviest @ next(W) @
^new_heaviest(W) <>- #t.

weight(W) @ new_heaviest(W) <>- weight(W) @ heaviest.

end_dept @ heaviest @ ^completed <>- end_dept.
```

Figure 8: Methods for the department with heaviest items

activity in charge of moving the `fill/3` atoms across weight classes) and, of course, all the weight classes run in parallel.

The bagger program above shows specific coordination mechanisms which are not present in other coordination models like Linda [Gel85]. *LO* defines computations based on multiset rewriting in a declarative way, using rules (in the sense of production rules); a C-Linda program has a much more imperative flavor. More important, Linda programs compute inside a monolithic shared tuple space; an *LO* program builds up multiple tuple spaces that can be considered different agents communicating by broadcasting.

Acknowledgement: We are grateful to S. Castellani and S. Pompei from Università di Pisa, who provided an *LO* implementation of the bagger problem.

References

[ACP92] J-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction abstract machines. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1992. To appear.

[AH87] G. Agha and C. Hewitt. Actors: a conceptual foundation for concurrent object-oriented programming. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Massachussetts, 1987.

[AP91] J-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Proc. of OOPSLA '91*, Phoenix, Az, U.S.A., 1991.

[AP92] J-M. Andreoli and R. Pareschi. Associative communication and its optimization via abstract interpretation, 1992. Submitted to the Journal of Theoretical Computer Science.

[APB91] J-M. Andreoli, R. Pareschi, and M. Bourgois. Dynamic programming as multi-agent programming. In *Proc. of the OOPSLA '90/ECOOP '91 workshop on Object-based concurrent computing*, Lecture Notes in Computer Science (612), Genève, Switzerland, 1991. Springer Verlag.

[BAP92] M. Bourgois, J-M. Andreoli, and R. Pareschi. Extending objects with rules, composition and concurrency: the *lo* experience. In *Proc. of the OOPSLA '92 workshop on Object-Oriented Programming Languages — The Next Generation*, Vancouver, Canada, 1992.

[BC91] A. Brogi and P. Ciancarini. The concurrent language shared prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[BCLM88] J-P. Banâtre, A. Coutant, and D. Le Metayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4(2):133–145, 1988.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge, Massachussetts, 1990.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2), 1970.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–113, 1985.

[Gir87] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Kay80] M. Kay. Algorithm schemata and data structure in syntactic processing. Technical report, Xerox Parc, Palo Alto, Ca, U.S.A., 1980.

[Mes92] J. Meseguer. A logical theory of concurrent objects and its realization in the MAUDE language. In G. Agha, A. Yonezawa, and P. Wegner, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1992. To appear.

[NT90] C. Numaoka and M. Tokoro. A decentralized parsing method using communicating multiple concurrent objects. In *Proc. of 2nd International Conference of Technology of Object Oriented Languages and Systems*, Paris, France, 1990.

[Win84] P. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, Massachussetts, 1984.

[WP85] D. Waltz and J. Pollack. Massively parallel parsing: a strongly interactive model of natural language interpretation. *Cognitive Science*, 9(1):51–74, 1985.

[YO88] A. Yonezawa and I. Ohsawa. Object-oriented parallel parsing for context-free gramars. In *Proc. of COLING'88*, Budapest, Hungary, 1988.

[YSH+90] A. Yonezawa, E. Shibayama, Y. Honda, T. Takada, and J-P. Briot. An object-oriented concurrent computation model ABCM/1 and its description language ABCL/1. In A. Yonezawa, editor, *ABCL, an Object-Oriented Concurrent System*, pages 13–45. MIT Press, Cambridge, Massachusetts, 1990.