Targeted Communication in LINEAR OBJECTS*

Vladimir Alexiev†

`<vladimir@cs.ualberta.ca>`

November 1994

Abstract

LINEAR OBJECTS (LO) of Andreoli and Pareschi is the first proposal to integrate object-oriented programming into logic programming based on Girard's Linear Logic (LL). In LO each object is represented by a separate open node of a proof tree. This "insulates" objects from one another which allows the attributes of an object to be represented as a multiset of atoms and thus facilitates easy retrieval and update of attributes. However this separation hinders communication between objects. Communication in LO is achieved through broadcasting to all objects which in our opinion is infeasible from a computational viewpoint.

This paper proposes a refined communication mechanism for LO which uses explicit communication channels specified by the programmer. We name it TCLO which stands for "Targeted Communication in LO". Although channel specification puts some burden on the programmer, we demonstrate that the language is expressive enough by redoing some of the examples given for LO. Broadcasting can be done in a controlled manner. LO can be seen as a special case of TCLO where only one global channel (the *forum*) is used.

Keywords LINEAR OBJECTS, communication, broadcasting, objects and logic, linear logic.

1 Introduction

The integration of Object-Oriented Programming (OOP) and Logic Programming (LP) is a long-standing goal of both the OOP and the LP research communities. The benefits of such an integration for both paradigms are apparent: OOP brings to LP the techniques of modern software engineering, while LP brings to OOP declarativity to replace the prevalent procedural languages of today. Seemingly the LP community has been more active in this effort. For a survey see [Dav92].

One of the hardest obstacles to an integration of OOP and LP is *state change* which is inherent in OOP but (up to until recently) was hard to express logically [Ale93]. Probably the most promising development in this area is Girard's Linear Logic (LL) [Gir87]. LL has proved useful in many areas of computation [Ale94], most notably concurrency and resource-based reasoning. (Another promising approach to state representation is Meseguer and Martí-Oliet's Rewriting Logic [Mes92, MOM93].)

LINEAR OBJECTS (LO) of Andreoli and Pareschi is the first proposal for integration of OOP and LP based on LL. Technically LL bears similarity to an earlier proposal [Con88] in that both approaches allow multiple atoms in the head of the clauses. The novelty of LL is its firm foundation in LL, which provides both a standard model-theoretic semantics based on phase spaces [AP91a, Section 2.4] (see also [CC94]) and proof-theoretical insights for the design of the language. As an example of the latter, the logic connective & is interpreted as object cloning (see Section 2 for details).

LO represents objects as separate open leaf nodes of the proof tree being constructed. The attributes of the object are atoms in the multiset context of the node. This allows easy access to the attributes for

---

consumption and addition by the methods of the object and makes possible compositional ("built-in") inheritance. However the separation of objects hinders communication. Indeed, the only communication mode in LO is *broadcasting*.[1] Broadcasting is a powerful communication abstraction, but we deem it computationally infeasible and not scalable (see Section 2.1 for discussion). We propose a refined communication mechanism for LO which we name TCLO.

This paper is organized as follows. Section 2 gives an overview of LO and explains the disadvantages of broadcasting. Section 3 describes the proposed extension of LO. Section 4 re-does some of the example programs given by Andreoli and Pareschi to demonstrate that the TCLO style of programming does not suffer a loss of expressive power. Section 5 concludes the paper.

2 Linear Objects

The syntax of LO is defined over a set of atomic formulas A constructed from predicate names $P = \{a, b, \ldots\}$ and terms in the usual way.

$$
\begin{array}{llll}
A & ::= & P(\langle \text{term} \rangle, \ldots) & \text{Atomic formulas} \\
G & ::= & A \mid \top \mid G_1 \,\mathfrak{N}\, G_2 \mid G_1 \& G_2 & \text{Goals} \\
V & ::= & A \mid {}^{\wedge}A \mid V_1 \,\mathfrak{N}\, V_2 & \text{Views (clause heads)} \\
M & ::= & V \circ\!\!-\, G & \text{Methods (clauses)}
\end{array}
$$

A context Γ is a multiset of goals. A program $\mathcal{P}$ is a set of methods. Since the program remains the same in every node of a proof tree, we abbreviate the LL sequent $!\&\mathcal{P} \vdash \Gamma$ to simply Γ (here $!\&\mathcal{P}$ denotes the modalized additive conjunction of the elements of $\mathcal{P}$).

The operational semantics of LO corresponds almost completely to bottom-up proof construction in Linear Logic [Gir87] (LL) (with the exception of the broadcast operator ${}^{\wedge}A$):

$$
\frac{}{\top, \Gamma} \;\; \top, \text{ Termination (success)}
$$

$$
\frac{G_1, G_2, \Gamma}{G_1 \,\mathfrak{N}\, G_2, \Gamma} \;\; \mathfrak{N}, \text{ Decomposition} \qquad
\frac{G_1, \Gamma \quad G_2, \Gamma}{G_1 \& G_2, \Gamma} \;\; \&, \text{ Cloning}
$$

$$
\frac{G, \Gamma}{V, \Gamma} \;\; \text{If } (V \circ\!\!-\, G) \in \mathcal{P} \quad \text{Resolution (propagation)}
$$

The root node in the proof tree of a LO execution is $\Gamma, \mathcal{C}$ where Γ is the initial state of the system and $\mathcal{C}$ is an unspecified context called the *forum* and used for communication. The proof tree grows from the root up and it can contain open (not completed) leaf nodes. Computation proceeds according to the following rules:

1. Pick an open node (or work in parallel on more than one open node).

2. *Termination*. If the context of the node contains $\top$, close the node according to the $\top$ rule.

3. *Decomposition* and *Cloning*. If the context of the node contains formulas with main connectives $\mathfrak{N}$ or $\&$ decompose them to atoms using repeatedly rules $\mathfrak{N}$ and $\&$. This may generate more than one successors of the current node.

4. *Resolution*. When the context contains only atoms and if the multiset of atoms is a superset of a clause head V, rewrite the occurrence of V in the context with the clause body G.

5. *Broadcasting*. If in the previous case V contained some broadcast atoms ${}^{\wedge}A$ then add these atoms to all non-closed (open or internal) nodes but the current one. In other words, replace the forum $\mathcal{C}$ with $A, \mathcal{C}'$, thus reducing the indeterminacy of the forum.

---

[1] It is also possible to do term-level (as opposed to formula-level) stream-based communication as in the earlier proposals based on concurrent PROLOGs (see [ST83]), but the inconveniences of this approach are well-known.

All of the above cases 1–4 correspond directly to inference rules of LL and expand the proof tree by changing an open node locally. However case 5 (the broadcasting rule) changes the proof tree globally by instantiating $\mathcal{C}$ and thus cannot correspond to an inference rule since these always act locally.

2.1 The Shortcomings of Broadcasting

The initial version of LO [AP91b] did not include broadcasting and the authors used stream-based (term-level) communication similar to the one in concurrent PROLOG implementations of OOP [ST83]. Andreoli and Pareschi introduced broadcasting in [AP91a] and described in great detail its expressive power for modeling inter-object communication. However it is our opinion that broadcasting is not a plausible concept for the modeling of large distributed systems and it does not scale well:

- Broadcasting is expensive. Its naive implementation leads to slow execution, and an efficient implementation is hard to achieve (see [BAP92, BAP94] for a discussion).

- Broadcasting destroys the locality of interaction (*e.g.* [BAP94, Sec. 9] uses a central entity to implement broadcasting), which is a very important principle for large systems of independent agents.

- Broadcasting leads to message name clashes. Since every agent sees all messages, one has to use different names for different messages, even if the messages are semantically similar (*e.g.* "get" sent to a list and "get" sent to a stack).

- Broadcasting makes it hard to partition the program because every method is potentially applicable to every object: even if a method is not currently applicable to an object, the object may receive in the future enough messages to make the method applicable. In fact not only LO, but all LL-based OOLP proposals that we know of, treat the program as a monolithic entity. We see this problem as a promising research direction.

- Although only a small percent of the broadcasted messages are relevant to any given agent, all agents receive all messages and have to store them in their contexts. This leads to "context saturation" (maybe more aptly named "context garbaging"). Andreoli and Pareschi propose to solve this problem using abstract interpretation [AP92, APC93], but it is better not to allow it in the first place.

3 Targeted Communication in LO

We propose to extend LO with explicit channels in order to solve the problems described above. We call the proposed extension TCLO.

In addition to the set of predicate names P, TCLO features a set of channel variables $\mathbf{X} = \{\mathbf{x}, \mathbf{y}, \ldots\}$ (denoted by bold face). These variables are similar to (but not the same as, see Section 3.3) second-order logic variables. We allow atomic formulas to include channels at the term level, *e.g.* $a(\mathbf{x})$ is a valid formula. Unlike predicate names, channels cannot bear arguments (*e.g.* $a(i)$ is a valid formula, but $\mathbf{x}(i)$ is not). Since any atomic formula can be sent along a channel, it would be superfluous to allow channels to bear arguments. We modify the definitions of goals and views (Section 2) as follows:

$$
\begin{array}{llll}
A & ::= & P(\langle \text{term} \rangle \mid \mathbf{X}, \ldots) & \text{Atomic formulas} \\
G & ::= & A \mid \mathbf{X} \mid \top \mid G_1 \,\mathbin{\rotimes}\, G_2 \mid G_1 \& G_2 & \text{Goals} \\
S & ::= & A \mid \mathbf{X} \mid S_1 \,\mathbin{\rotimes}\, S_2 & \text{Simple views (no sending)} \\
V & ::= & A \mid \mathbf{X}\hat{\ }S \mid V_1 \,\mathbin{\rotimes}\, V_2 & \text{Views} \\
M & ::= & V \circ\!\!-\, G & \text{Methods (clauses)}
\end{array}
$$

Thus we replace broadcasting $\hat{\ }A$ with message sending along a channel, $\mathbf{X}\hat{\ }S$. The extension of sending simple views and not plain atoms is needed to allow renewed communication (see the next section).

The operational semantics of TCLO is defined by modifying the *Resolution* and *Broadcasting* rules of LO.

- *Resolution*. If the context is flat (contains no LL connectives) and a clause head matches V, rewrite the occurrence of V in the context with the clause body G. A message send $\mathbf{x}\hat{\ }m$ in V can be matched either if a channel variable appears in the context at the outer level (the variable name need not be the same, *e.g.* $\mathbf{y}$ would match $\mathbf{x}$), or else if the channel is specified explicitly using unification, *e.g.* $V = c(\mathbf{x}) \,\mathcal{R}\, \mathbf{x}\hat{\ }a$ will match a context containing $c(\mathbf{y})$ (and identify $\mathbf{x}$ and $\mathbf{y}$) but not one containing $\mathbf{y}$ only at the outer level. Resolution does not look inside the messages that V sends, *e.g.* $V = c(\mathbf{x}) \,\mathcal{R}\, \mathbf{x}\hat{\ }(m\mathcal{R}\mathbf{y})$ requires the presence in the context of a channel $c(\mathbf{z})$ but does not care about m nor $\mathbf{y}$.

- *Message Send*. If in the previous case V contained some message sends $\mathbf{X}\hat{\ }S$, replace $\mathbf{X}$ everywhere in the proof tree by S. The occurrence of $\mathbf{X}$ in the current node was consumed by the *Resolution* rule, so the current node will not receive a copy of the message it sends. However, if the node contained more than one occurrence of $\mathbf{X}$, the remaining occurrences will be replaced by the message S.

The occurrence of a channel $\mathbf{X}$ in a context can either be consumed by a message-sending Resolution or replaced with the message body S during a send initiated by another agent.

3.1 Communication Patterns in TCLO

Here we describe various communication patterns that occur commonly in TCLO.

Multidirectionality TCLO channels are multidirectional: any agent having an occurrence of the channel variable can write to it. This can be either useful or inconvenient depending on th setting.

Many-to-One Communication If an agent contains more than one channel at the top level of its context, it can receive messages along these channels without any interference between them. Thus unlike OOLP based on concurrent Prolog languages [ST83], no message merging elements are needed. This is due to the fact that channels are at the logic level, not at the term level.

One-to-Many (Group) Communication All the agents connected to a channel will receive the data placed on it. Arranging various communication topologies is as simple as passing the channel name to another agent. During *Cloning* the channels known to the parent are passed to both children automatically (or if *Cloning* splits a formula containing channels, they are distributed between the two children). If we regard the children open nodes of a given node as a "process group" and if the parent node has a channel shared among all children, we can easily achieve group communication by writing to this channel.

Named vs Unnamed Channels If an agent needs only one channel (that is, it communicates with only one agent group), it can use the channel at outer level in its context. However, if an agent has more than one outgoing channels, it needs to label them with distinct names. For example, an agent may have two named channels $c(\mathbf{x}), d(\mathbf{y})$ and send messages using clauses like

$$c(\mathbf{x}) \,\mathcal{R}\, \mathbf{x}\hat{\ }m \multimap \langle \text{new state} \rangle$$
$$d(\mathbf{y}) \,\mathcal{R}\, \mathbf{y}\hat{\ }n \multimap \langle \text{new state} \rangle$$

Incoming channels are usually unnamed to allow easy merging of message streams.

Renewable Communication Often a channel is needed to deliver more than one message. While the clause

$$s \,\mathcal{R}\, \mathbf{x}\hat{\ }m \multimap s'$$

corresponds to a sender s using channel x for a one-shot communication m, the channel can be renewed using

$$s \,\mathcal{R}\, \mathbf{x}\hat{\ }(m\mathcal{R}\mathbf{y}) \multimap s'\mathcal{R}\mathbf{y}.$$

Responses Usually no separate channel is needed to carry the response to a request. Since the channel is bidirectional, the response can be sent along the renewed channel, similar to the use of first-order logic variables to return answers from function-like predicates.

3.2 LO as a special case of TCLO

It is easy to see that LO is a special case of TCLO that uses only one channel, the forum. A LO broadcasting clause

$$V \bindnasrepma {}^\wedge A \multimap G$$

is replaced by the TCLO clause

$$V \bindnasrepma \mathbf{x}^\wedge (A \bindnasrepma \mathbf{y}) \multimap G \bindnasrepma \mathbf{y}.$$

The initial state of the LO program Γ is replaced by $\Gamma \bindnasrepma \mathbf{x}$. Thus, what Andreoli and Pareschi call "the unspecified part of the context", we represent explicitly as an unnamed channel.

In terms of the "group communication" described in the previous section, LO can be seen to consider the whole concurrent system as consisting of one communication group, generated by the root node.

Although TCLO is an extension of LO, from a practical viewpoint its programming style is a restriction of the one of LO since the programmer is required to specify explicitly the communication channels and to take care of distributing them among agents. Section 4 suggests that this restriction is not a severe one.

3.3 TCLO and Second-Order LL

Although TCLO channel variables are similar to second-order variables in linear logic, the two are not the same. We tried to describe TCLO communication as second-order unification (that is to say, existential introduction under the bottom-up reading), but found it impossible to do so. Unfortunately TCLO suffers from the same impossibility to ascribe true logical rules to its communication as LO, and for the same reason: while every logical rule acts locally on an open node, TCLO communication involves "distant action" in separate branches of the proof tree.

On the other hand, channel variables are used rather restrictively compared to the possible uses of second-order variables. This obviates the need for full second-order unification and simplifies implementation.

4 Programming Examples

In this section we redo some of the examples given for LO in order to demonstrate that TCLO-style programming suffers no loss of expressive power. In fact we find that the non-directed nature of broadcasting is sometimes confusing to the programmer (Won't my message cause any adverse interference in other objects?) and to the reader of a program (Where is this message going to?).

4.1 The Optimal Path Problem

This example is taken from [APB91, Section 3]. The problem is

> Given a weighted-edge acyclic directed graph with a distinguished source s and target t (a *network*), find a path from s to t with the smallest possible weight.

The solution is based on dynamic programming and is easily adapted from [APB91]. For simplicity, we restrict the solution to use only one representation of the network (every node stores the set of its *outgoing* edges), we implement only "forward chaining" algorithm, and we destructively modify the network representation during the path search. Also, our example network is smaller than the one in [APB91] (see Figure 1). Each edge is labeled with its weight and the channel variable corresponding to the edge (see below). The optimal path is drawn bold.

We represent the network as follows:

```
network ○—
   (node(s) ⅋ in(0) ⅋ e(5,A1) ⅋ e(6,B1) ⅋ e(4,C1)) &
   (node(a) ⅋ in(1) ⅋ A1 ⅋ e(2,D1) ⅋ e(5,E1)) &
```
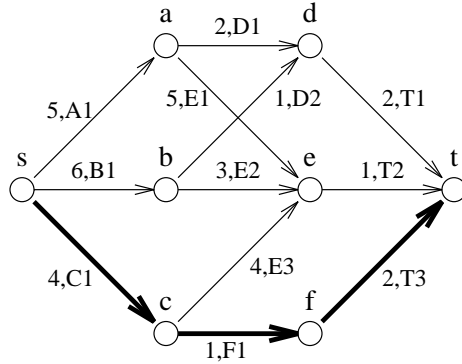
Figure 1: Example of a Weighted-Edge Network

```
(node(b) ⅋ in(1) ⅋ B1 ⅋ e(1,D2) ⅋ e(3,E2)) &
(node(c) ⅋ in(1) ⅋ C1 ⅋ e(4,E3) ⅋ e(1,F1)) &
(node(d) ⅋ in(2) ⅋ D1 ⅋ D2 ⅋ e(2,T1)) &
(node(e) ⅋ in(3) ⅋ E1 ⅋ E2 ⅋ E3 ⅋ e(1,T2)) &
(node(f) ⅋ in(1) ⅋ F1 ⅋ e(2,T3)) &
(node(t) ⅋ in(1) ⅋ T1 ⅋ T2 ⅋ T3).
```

Every node is represented as a separate agent (expressed by & above). For every node we store its name node(_). Unlike the LO solution, these names are included only for the purpose of printing the found path in readable form, and not for communication between nodes. We record the node's input arity and for every incoming edge create a separate channel (denoted by uppercase). The outgoing edges are represented by the atoms e(W,C) storing their weights and a channel to the destination node. Please note that the incoming channels are represented at the higher (formula) level, while outgoing channels are hidden at the lower (term) level. This allows an agent to have easy uniform access to all its incoming messages yet to differentiate between its outgoing channels.

The agents work through two phases: selection and spreading. These phases occur asynchronously in the network, forming a "wave front" moving from s to t.

1. During the select phase, the agent receives best-path offers from each of its predecessors in the form of atoms path(C,P) where C is the cost of the path and P is a list of the nodes comprising the path. The agent discards all but the best offer.

```
select ⅋ in(I) ⅋ path(C1,P1) ⅋ path (C2,P2) ⅋ {C1<=C2} ∘—
  select ⅋ in(I-1) ⅋ path(C1,P1).          ; discard suboptimal offers
select ⅋ in(1) ∘— spread.                   ; move to the next phase
```

2. During the spread phase the agent propagates this best path to all its successors, adding its name and the weight of the outgoing edge to the path.

```
spread ⅋ path(C,P) ⅋ node(M) ⅋ e(W,N) ⅋ N^path(C+W,P::M) ∘—
  spread ⅋ path(C,P) ⅋ node(M).        ; send message. :: is append
```

The program is started by a goal of the form

```
∘— network ⅋ source(s) ⅋ target(t) ⅋ output(O) ⅋ select.
```

All of the atoms `source`, `target`, `output` and `select` are delivered to all agents (&-components of `network`), although only the source and target nodes make use of the first three. The special clauses for the source and target nodes are

```
node(S) ⅋ source(S) ⅋ select ∘— node(S) ⅋ path(0,[]) ⅋ spread.
node(T) ⅋ target(T) ⅋ path(C,P) ⅋ spread ⅋ output(O)
  O^path(C,P::T) ∘— ⊤.                      ; output result and terminate
```

4.2 Distributed Active Parsing

In [AP91a, Section 3.2] Andreoli and Pareschi describe an LO implementation of active parsing for context-free (and possibly ambiguous) grammars. The term "active" refers to the feature that incomplete phrasal trees are considered active agents which look for and consume complete trees. The parser is composed of the agents shown on Figure 2, adapted from [AP91a]. We renamed the agent `create_tree` to `dispatcher`
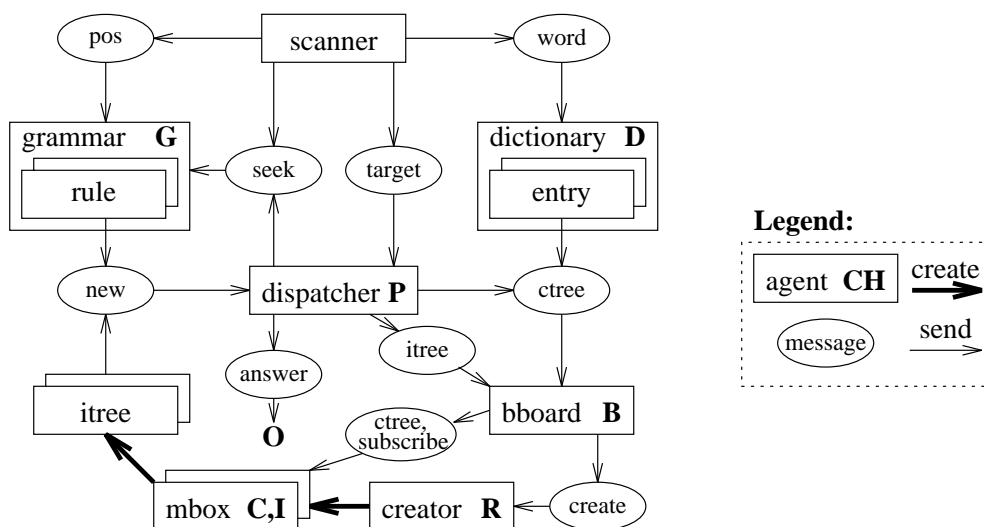

Figure 2: Distributed Active Parser: Agents and Flow of Information

and added the agents `bboard`, `creator`, `mbox`, and some messages. The bold letters denote typical channel variables used to talk to a particular agent.

The distinction between agents and messages is mainly a conceptual one, because both are represented by atoms. However while every method referencing an agent atom in its head reinstates it in its body, message atoms are not reinstated and are thus consumed. Thus agents are relatively long-lived and messages are short-lived.

This example is more involved both for the reader to understand (it is more complex), and for the programmer to create (it uses generative communication [Gel85] which we have to represent using targeted communication). Our program is shown below.

```
1: parse(Input,Symbol,O) ∘—
2:    (scanner(Input,0,Symbol,O,P2,G2,D)) &
3:    (grammar(P1)⅋G1⅋G2) &
4:    (dictionary(B2)⅋D) &
5:    (dispatcher(G1,B1)⅋P1⅋P2) &
6:    (bboard(R)⅋B1⅋B2) &
```

```
 7:    (creator⅋R).
 8: scanner([],N,S,O,P,G,D) ⅋
 9:    G^seek(O,S) ⅋
10:    P^target(O,N,S) ∘— ⊤.
11: scanner([W|I],N,S,O,P,G,D) ⅋
12:    G^(pos(N)⅋G1) ⅋
13:    D^(word(W,N)⅋D1) ∘—
14:    scanner(I,N+1,S,O,P1,G1,D1).
15: grammar(P) ∘— replicate(P) ⅋
16:    rule(s,[np,vp]) ⅋ rule(np,[det,n]) ⅋ rule(np,[pn]) ⅋
17:    rule(np,[np,pp]) ⅋ rule(vp,[tv,np]) ⅋ rule(vp,[vp,pp]) ⅋
18:    rule(pp,[prep,np]).
19: replicate(P) ⅋ rule(S,Ss) ⅋
20:    P^(P1⅋P2) ∘—
21:    replicate(P1) & rule(S,Ss,P2).
22: rule(S,Ss,P) ⅋ seek(N,S) ⅋ pos(N) ⅋
23:    P^(new(N,N,S,Ss,S)⅋P1) ∘— rule(S,Ss,P1).
24: dictionary(B) ∘— replicate(B) ⅋
25:    entry(a,det) ⅋ entry(robot,n) ⅋ entry(telescope,n) ⅋
26:    entry(terry,pn) ⅋ entry(saw,tv) ⅋ entry(with,prep).
27: replicate(B) ⅋ entry(W,S) ⅋
28:    B^(B1⅋B2) ∘—
29:    replicate(B1) & entry(W,S,B2).
30: entry(W,S,B) ⅋ word(W,N) ⅋
31:    B^(message(N,S,ctree(N,N+1,S,S-W)⅋B1)) ∘— entry(W,S,B1).
32: dispatcher(G,B) ⅋ new(O,N,S,[],T) ⅋ target(O,N,S) ⅋
33:    O^(answer(T)⅋O1) ∘—
34:    dispatcher(G,B) ⅋ target(O1,N,S).
35: dispatcher(G,B) ⅋ new(M,N,S,[],T) ⅋
36:    B^(message(M,S,ctree(M,N,S,T))⅋B1) ∘—
37:    dispatcher(G,B1).
38: dispatcher(G,B) ⅋ new(M,N,S,[S1|Ss],T) ⅋
39:    G^(seek(N,S1)⅋G1) ⅋
40:    B^(message(N,S1,itree(M,N,S,S1,Ss,T,P))⅋B1) ∘—
41:    dispatcher(G1,B1)⅋P.
42: itree(M,N,S,S1,Ss,T,P) ⅋ ctree(N,P,S1,T1) ⅋
43:    P^(new(M,P,S,Ss,T-T1)⅋P1) ∘—
44:    itree(M,N,S,S1,Ss,T,P1).
45: bboard(R) ⅋ ctree(M,N,S,T) ⅋ mbox(M,S,C,I) ⅋
46:    C^(ctree(M,N,S,T)⅋C1) ∘—
47:    bboard(R) ⅋ mbox(M,S,C1,I).
48: bboard(R) ⅋ itree(M,N,S,S1,Ss,T,P) ⅋ mbox(N,S1,C,I) ⅋
49:    I^subscribe(itree(M,N,S,S1,Ss,T,P),I1) ∘—
50:    bboard(R) ⅋ mbox(N,S,C,I1).
51: bboard(R) ⅋ message(N,S,Msg) ⅋ mbox(N,S,C,I) ∘—
52:    bboard(R) ⅋ Msg ⅋ mbox(N,S,C,I).
53: bboard(R) ⅋ message(N,S,Msg) ⅋
54:    R^create(C,I,R1) ∘—
55:    bboard(R) ⅋ Msg ⅋ mbox(N,S,C,I).
56: creator ⅋ create(C,I,R) ∘—
57:    (creator⅋R) & (mbox⅋C⅋I).
```

```
58: mbox ⅋ subscribe(ITree,I) ∘—
59:    (mbox⅋I) & ITree.
```

We describe the computation and communication aspects of the parser separately in the following two
sections. The section on computation is sketchy since the algorithm is essentially the same as the one in
[AP91a]. We put the emphasis on interconnecting the agents, thus the section on communication is more
detailed.

4.2.1 Computation

The agents **bboard**, **creator** and **mbox** have only communication functions, so we describe them in the next
section. In this section we generally omit channel variables. Numbers in parentheses refer to program lines.

parse The program is started by a goal of the form **parse(Input,Symbol,O)** specifying the input list (*e.g.*
   [terry, saw, a, robot, with, a, telescope]), the target (initial) non-literal symbol (*e.g.* **s** for
   "sentence") and the output channel. **Parse** (1) creates the "static" agents of the parser (2-7) and
   interconnects them.

scanner The agent **scanner** is of the form **scanner(I,N,S)** where **I** is the remainder of the input list, **N**
   is the current input position and **S** is the initial non-terminal. **Scanner** decomposes (11) the input
   list to a series of **pos(N)** (12) and **word(W,N)** (13) messages which it sends to the **grammar** and the
   dictionary respectively. When the input list is exhausted (8), **scanner** sends a "seeding" message
   seek(0,S) to the **grammar** (9) meaning "Try to parse non-terminal **S** starting at position 0", and
   informs the **dispatcher** about the **target** channel and non-terminal and the final position (10).

grammar The **grammar** (15) is a set of **rules(S,Ss)** (16-18) specifying that the non-terminal **S** can be
   replaced with the list **Ss**. Every **rule** responds to **seek(N,S)** messages (22) by requesting the creation
   of a new incomplete tree using the **new(N,N,S,Ss,S)** message (23). The "side condition" **pos(N)**
   ensures that the rule will fire no more than once for every position.

dictionary The **dictionary** (24) is a set of **entries(W,S)** (25-26) specifying that word **W** is of category **S**.
   Every **entry** responds to **word(W,N)** messages (30) by producing a complete (although very simple)
   one-node phrasal tree over that word and then submitting that **ctree(N,N+1,S,S-W)** to the **bboard**
   (31) for further processing. (For a reason explained later, this message has to be wrapped in a **message**
   atom.)

dispatcher The **dispatcher** (32-41) handles tree creation requests of the form **new(M,N,S,Ss,T)** where
   the tree **T** is rooted at non-terminal **S**, the known part of the tree spans input positions **M...N**, and the
   unknown part of the tree starts at position **N** and consists of the "concatenation" of trees rooted at the
   elements of the list **Ss**. The **dispatcher** distinguishes between three cases:

   1. If the unknown part is not empty (38, the pattern **[S1|Ss]**), an *incomplete* tree **itree(M,N,S,S1,Ss,T)**
      is produced (40). Also, **seek(N,S1)** is sent to the **grammar** (39) to initiate parsing the incomplete
      part.

   2. If the unknown part is empty (35, **[]**) then a *complete* tree **ctree(M,N,S,T)** is produced (36). The
      arguments mean that there is a phrasal tree **T** rooted at the non-terminal symbol **S** and spanning
      input positions **M...N**. Together with the "primitive" **ctrees** produced by the **dictionary** (31),
      these play the role of building blocks that are put together by **itrees** (see below).

   3. If the unknown part is empty *and* the tree spans the whole input *and* the tree is rooted at the
      target non-terminal **S** (32), then **dispatcher** puts the tree at the output channel (33). Since
      this case is a specialization of the previous one, it is possible that method (35) will fire instead
      of method (32). Aside from being inefficient, this will do no harm because the **ctree** will travel
      the loop **dispatcher - bboard - mbox - itree** and come back to **dispatcher** (see Figure 2).

9

Under some fairness assumptions, (32) will eventually fire and spit the `ctree` out, thus breaking the loop.

itree If we compared `ctrees` to building blocks, we should compare `itrees` to masons. An `itree(M,N,S,S1,Ss,T)` (42) consumes `ctrees` that "mesh" with it, meaning that they start in the input position where the `itree` ends, and are rooted at the required non-terminal `S1`, *i.e.* are of the form `ctree(N,P,S1,T1)`. Then the `itree` attaches this `ctree` to itself (43) by joining the ranges `M...N` and `N...P` to `M...P`, and connecting the trees `T` and `T1` using the constructor '-'[2], to form `T-T1`. The result is sent back to `dispatcher` for further dispatching.

4.2.2 Communication

We refine the algorithm in [AP91a] by establishing channels between agents, *i.e.* making explicit the flow of information depicted on Figure 2. Some of the interconnection techniques we use are "standard" and we already described them in Section 3.1.

One-to-One Communication For example, program lines (2-7) interconnect the static agents using the same technique as in Section 4.1, namely incoming channels at the formula level and outgoing channels at the term level.

One-to-Many Communication All `rules` (resp. `entries`) receive messages through their parent, `grammar` (resp. `dictionary`). This is achieved by including the channel in the context cloned by & (line 21, resp. 29).

Many-to-One Communication Every `rule` needs to talk to the `dispatcher` (resp. every `entry` needs to talk to the `bboard`). To avoid contentions, the channel must be `replicated` (lines (20), resp. (28)).

In addition to these "standard" optimizations, we exploit the fact that every `itree` only cares about `ctrees` that "mesh" with it (see the end of the previous subsection and line (42)). We refine communication to `itree` so that every `itree` receives only relevant `ctrees`.[3] The agents `bboard`, `mbox` and `creator` came into existence due to this optimization. We describe them below.

mbox There is a "mail box" `mbox`[4] (58) for every combination (`M,S`) of starting position `M` and root symbol `S` (these being the `ctree`-`itree` "meshing" parameters) that occurs during parsing. To each `mbox` are attached a number of `itrees` that we call `subscribed` to the `mbox`. In fact, the `mbox` creates[5] these `itrees` using cloning (59). Every one of these `itrees` sees every `ctree` message sent to the `mbox`, including the ones sent after its creation. Conversely, a newly created `itree` will inherit all the `ctree` messages already sent to the `mbox`.

bboard The agent `bboard` (45-55) ("bulletin board", or "billboard", or "blackboard") manages the access to and creation of `mboxes`.

> **Managing Mbox Access** (45-50) The `bboard` holds channels to every `mbox` in atoms `mbox/4` and its context looks like this at runtime:
>
> `bboard(R), mbox(N1,S1,C1,I1), ..., mbox(Nn,Sn,Cn,In)`
>
> where `n` is the current number of `mboxes`, **R** is a channel to the `creator` (see below), (`N,S`) is the "key" of the `mbox`, channel **I** is used to send `itree` messages to the `mbox` (49), and channel **C** is used for `ctree` messages (46). We use two separate channels because we want all children `itrees`

---

[2] This is a convenient graphical notation for "cons", it is not arithmetic minus.

[3] A similar consideration concerning `rule` and `seek` (22) could be exploited. We do not pursue it here, the case of `itree` and `ctree` being more involved.

[4] Initially we called `mboxes` "channels" (being inspired by Internet's Inter Relay Chat), but then we decided to change the name in order to avoid conflict with the predominant use of the word "channel" in this paper.

[5] More precisely, emancipates to the rank of agents.

of a `mbox` to see all `ctree` messages sent to it, but none of the `itree` messages. To this end the `mbox` lets `ctree` messages go through transparently (it even does not have a method for handling such messages), but appropriates the **I** channel before doing cloning (59), so the `ITree` does not get access to that channel. The renewal of **C** and **I** is also being done differently: "externally" for **C** (46) and "by mutual consent" for **I** (49).

Managing Mbox Creation (51-55) When a `message(N,S,Msg)`[6] is sent to `bboard`, it has to create a `(N,S)`-indexed `mbox` iff no such exists already. The method (51) works when the `mbox` exists by simply unwrapping the `message` that will then be picked by either (45) or (48). The method (53) works when the `mbox` does not exist by asking `creator` to create it (54) and then unwrapping the `message` as in the previous case. Unfortunately (53) can fire even if the `mbox` already exists, unless the language implementation utilises a Most-Specific-First scheduling strategy. Unlike the conflict (32)–(35) (see the description of `dispatcher` above), this conflict (51)–(53) is not harmless because it creates spurious `mboxes`. Thus we assume that the implementation supports such a scheduling strategy.

creator Finally, `creator` (56) is a very simple agent that creates `mboxes` with nothing in the context but two channels **C** and **I** (57). We cannot create `mboxes` directly from `bboard` because `bboard` carries many atoms in its context that do not belong to the `mbox` being created.

5 Concluding Remarks

We described TCLO, a refinement of LO with explicit channels. We have shown that LO is a specialization of TCLO that uses only one channel. Our work can be seen as a less-ambitious approach to the problem of efficiency of LO since we provide language means to achieve what Andreoli and Pareschi purport to achieve using compilation techniques (in particular, abstract interpretation and partial evaluation).

5.1 Future Work

A CHAM-based semantics of TCLO (adapted from the one for LO, [ALPT93][CC94]) is under development.

The exact relation of our channel variables to second-order variables, and the non-linear properties of channels should be investigated. For example, channel replication calls for the use of contraction, thus exponentials.

We wonder what are the limits of abstract interpretation for the optimization of communication. For example we doubt that the manual optimization we did with `bboard/mbox` can be achieved using abstract interpretation.

All the LL-based OOLP approaches we are aware of deal with a constant program (the left sides of all sequents are the same). Modularization of the program into class-based units and OOP phenomena such as method overriding should be investigated.

5.2 Related Work

TCLO is similar to Hewitt and Agha's ACTORS [Agh86] in that communication is capability (acquaintance) based. Unlike ACTORS, in TCLO the channels are not interpreted as identities (mailboxes), since an agent can have more than one input channel and can create and drop channels dynamically.

To date, there are only a few proposals for integration of OOP and LP based on Linear Logic. LO was the first one. Saraswat and Lincoln proposed `lcc` [SL92][Sar93] and independently Kobayashi and Yonezawa proposed ACL [KY94a][KY94b]. They recast the earlier renditions of Milner's π-calculus as LL theories [BS92, Mil92] into the LP paradigm of "computation as proof search" (*e.g.* ACL is dubbed "process calculus in logical form"). These languages contain two crucial ingredients of objects: state change (provided for by

---

[6] Here `Msg` is either `ctree` or `itree`.

linear logic) and message-based communication. However they are not specifically object-oriented because they lack object attributes.

References

[Agh86] Gul Agha. ACTORS: A model of concurrent computation. In *Distributed Systems*. MIT Press, 1986.

[Ale93] V. Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report TR93-15, University of Alberta, August 1993. Available from `ftp.cs.ualberta.ca: pub/TechReports/TR93-15`, file `TR93-15.ps.Z` or `TR93-15.a4.ps.Z`.

[Ale94] V. Alexiev. Applications of linear logic to computation: An overview. *Bulletin of the IGPL*, 2(1):77–107, March 1994. Available from `theory.doc.ic.ac.uk: theory/forum/igpl/Bulletin/V2-1/ Alexiev.ps.gz`. Also University of Alberta TR93-18, December 1993.

[ALPT93] J.-M. Andreoli, L. Leth, R. Pareschi, and B. Thomsen. True concurrency semantics for a linear logic programming language with broadcast communication. In *Theory and Practice of Software Development (TAPSOFT'93)*, pages 182–198, 1993.

[AP91a] J.-M. Andreoli and R. Pareschi. Communication as fair distribution of knowledge. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'91)*, pages 212–229, November 1991. *ACM SIGPLAN Notices*, 26(11).

[AP91b] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9(3-4):445–473, 1991. Shorter version appeared in D.H.D. Warren and P. Szeredi (eds), *Intl. Conf. on Logic Programming (ICLP'90)*, Jerusalem, Israel, June 1990, pages 495–510.

[AP92]   J.-M. Andreoli and R. Pareschi. Associative communication and its optimization via abstract interpretation. Submitted to TCS (but seems to never have appeared), 1992.

[APB91] J.-M. Andreoli, R. Pareschi, and M. Bourgois. Dynamic programming as multiagent programming. Technical Report ECRC-91-13, ECRC, München, 1991.

[APC93] J.-M. Andreoli, R. Pareschi, and T. Castagnetti. Abstract interpretation of linear logic programming. In *International Logic Programming Symposium (ILPS'93)*, pages 295–314. MIT Press, 1993.

[BAP92] M. Bourgois, J.-M. Andreoli, and R. Pareschi. Extending objects with rules, composition and concurrency: the LO experience. In *OOPSLA'92 Workshop "Object-Oriented Languages: The Next Generation"*, 1992. Also technical report ECRC-92-26.

[BAP94] M. Bourgois, J.-M. Andreoli, and R. Pareschi. Concurrency and communication: Choices in implementing the coordination language LO. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proc. of the ECOOP'93 Workshop on Object-Based Distributed Programming*, number 791 in LNCS, pages 73–92, 1994.

[BS92]   G. Bellin and P.J. Scott. On the π-calculus and linear logic. Manuscript to be submitted to Proc. MFPS 8, Oxford, November 1992.

[CC94]   S. Castellani and P. Ciancarini. Comparative semantics of LO. Technical Report UBLCS-94-7, University of Bologna, April 1994. Availabile by anonymous FTP from `ftp.cs.unibo.it: /pub/TR/UBLCS/SemanticsOfLO.ps.gz`.

[Con88]   John S. Conery. Logical objects. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Intl. Conf. and Symp. on Logic Programming (ICLP'88)*, pages 420–434, 1988.

[Dav92]   A. Davison. A survey of logic programming-based object-oriented languages. Technical Report 92/3, University of Melbourne, January 1992. Fourth revision; first published April 1989.

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

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

[KY94a]   N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, (3), 1994. Short version appeared in *Joint Intl. Conf. and Symp. on Logic Programming (JICSLP'92)*, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.

[KY94b]   N. Kobayashi and A. Yonezawa. Typed higher-order concurrent linear logic programming. Technical Report 94-12, University of Tokyo, July 1994. Available by FTP from `camille.is.s.u-tokyo.ac.jp /pub/papers/ TR94-12-hacl-a4.ps.Z`.

[Mes92]   J. Meseguer. Multiparadigm logic programming. In G. Levi and H. Kirchner, editors, *Third Intl. Conf. on Algebraic and Logic Programming (ALP'92)*, number 632 in LNCS, pages 158–200, Volterra, Italy, September 1992.

[Mil92]   D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming (ELP'92)*, 1992. Also University of Pennsylvania technical report MS-CIS-92-48, available from `ftp.cis.upenn.edu: pub/papers/miller/pic.dvi.Z`.

[MOM93]   N. Martí-Oliet and J. Meseguer. Action and change in rewriting logic. Technical Report (to appear), Computer Science Laboratory, SRI International, 1993.

[Sar93]   V. Saraswat. A brief introduction to linear concurrent constraint programming, April 1993. Available from `parcftp.xerox.com: pub/ccp/lcc/lcc-intro.dvi.Z`.

[SL92]   V. Saraswat and Patrick Lincoln. Higher-order, linear, concurrent constraint programming, July 1992. Available from `parcftp.xerox.com: pub/ccp/lcc/hlcc.dvi.Z`.

[ST83]   Ehud Shapiro and Akikazu Takeuchi. Object-oriented programming in CONCURRENT PROLOG. *New Generation Computing*, 1:25–48, 1983.