Objects in Parallel Label-Selective λ-calculus (LCEPOO)*

Carlos Herrero§ Javier Oliver§

§ DSIC, UPV, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain.
Teléfono: 96 387 7350. Fax: 96 387 7359.
E-mail: cherrero,fjoliver@dsic.upv.es.

Abstract

LCEP is a calculus for modelling concurrent systems. The efforts to use it to represent object oriented features have been successfully treated in this paper. We present an operational semantics for a parallel object-oriented programming language by means of a phrase-by-phrase translation from the language into an extension of LCEP in which only a few changes from the original LCEP are made.

1 Introduction

We can find many studies in computation to produce an elegant semantics which correctly defines languages with concurrent features. The role played by the λ-calculus in computation theory is well-known but while it was adopted by D.S. Scott and C. Strachey as a semantic basis for programming languages, it cannot be successfully used with concurrence features. In 1980, R. Milner proposed CCS, [7], which is a calculus for modelling concurrent systems. His contribution was to view computational entities as agents whose interaction is the basic behavioural unit. This calculus was the basis for the π-calculus [8] which emerged as a refinement of CCS with two primitive notions as foundations: the *name* and the *agent*. A *name* is used to provide access to agents. Specifically, two agents that share a name can interact by using it. In addition, an agent can send a name in an interaction, and therefore one agent can transfer its ability to interact with other agents to the other. On the other hand, with the label-selective λ-calculus of H. Aït-Kaci [1] the distinction between terms and channels appears. A term plays the role of an agent of the π-calculus and a channel (numeric or symbolic labels) represents the way through which terms can send information. With the extension of this calculus to LCEP [6] (Parallel Label-Selective λ-calculus) we have a complete calculus for modelling concurrency by using numeric and/or symbolic labels and parallel operators, and also by representing polyadic functions.

Among the proposed approaches to parallel programming, the disciplines which are characteristic of parallel object-oriented programming have many desirable features. However, providing adequate semantics for these languages is very complex. The family of

*This work was partially supported by CICYT, TIC 95-0433-C03-03.

POOL languages is outstanding among the works in this field. This is the starting point for this work. In particular, the operational semantics of a member of this family of languages appeared in [2] and was later expanded by the denotacional semantics [3] based on metric spaces. Moreover, there are other semantics available which are associated to the family of POOL languages using process algebras [10]. The nature of the language suggests that correct models are complicated. More precisely, attempts to discern a clear vision of the central concept (i.e. the object) is very complex, although already important advances have been made in this direction. In this article, we show a semantics in LCEP of a slightly modified version of POOL, via a phrase-by-phrase translation. Each syntactic entity is represented as a parameterized agent. The representation of composite entities is constructed by using the operators of the calculus. In particular, we are interested in modelling the communication, the invocation of methods, and the transfer of results between objects. An object is represented as a composed term which is able to interact with others by asking for the execution of a certain method.

The remainder of the work is organized as follows. In Section 2, LCEP is briefly defined. In Section 3, the POOL language is described. In Section 4, we show the translation function of the different instructions from POOL into LCEP. In Section 5, we present an example of a POOL program translation. Finally, Section 6 presents some concluding remarks.

2 The Parallel Label-Selective λ-calculus

The Parallel Label-Selective λ-calculus (LCEP) [9] is the calculus that we use for modelling the LCEPOO language, which originated from a previous proposal by H. Aït-Kaci (the label-selective λ-calculus). This calculus is an extension of the λ-calculus in which function arguments are selected by labels. Symbolic and numeric labels can be used to name the communication channels. This has not been possible in other proposals and it allows for the use of currying and the labeled specification of parameters. It makes the ordering of the actual parameters in a function call independent of the presentation order of the formal parameters in the definition.

Let $V = \{x, y, z, \ldots\}$ be a set of variables, and $C = \{a, b, c, \ldots\}$ be a set of constants. P represents a channel name from a set of labels $\mathcal{L} = \mathbb{N} \cup \mathcal{S}$, where $m, n, \ldots$ denote numerical labels taken from $\mathbb{N}$. $\mathcal{S} = \{p, q, \ldots\}$ is a set of symbolic labels. $\mathbb{N} \cap \mathcal{S} = \emptyset$. 0 represents the null process.

The language of the formal system is $\mathcal{M}$. It is defined as follows:

$$\begin{array}{lll} \mathcal{M} & ::= & 0 \mid a \mid x \mid \lambda_P x.M \mid \widehat{P}\ M \mid M \parallel M \mid M \circ M \mid (M + M) \mid !M \\ P & ::= & \text{number} \mid \text{symbol} \end{array}$$

Usually, the terms of this signature are called λ_{EP}-terms.

In concurrent programming, the terms of $\mathcal{M}$ represent processes. Operation symbols $\circ, \parallel, +, !$ are considered as process constructors. The communication $\lambda_P x.$ (abstraction or input) and $\widehat{P}$ (parameter passing or output) are also considered as process constructors, whose effect over a process M is to define processes ($\lambda_P x.M$ y $\widehat{P}M$, respectively) which are involved in functional applications under certain given conditions. Constants, variables and the null process form the atomic symbols of the language.

The interpretation of the process constructors is as follows: sequential composition ($\circ$): it defines a process $M \circ N$ from the processes M y N. The process N is executed after M; parallel composition ($\|$): it defines a new process $M \parallel N$ from the processes M and N. Both processes are executed simultaneously; non-deterministic choice ($+$): $M+N$ shows the capability of M or N (either one or the other) to take part in a communication. A communication can be made by one of them and the other disappears; replicator (!): $!M$ defines a process which produces the multiple generation of M. It acts as a warehouse, where it is possible to take an infinite number of the process M.

Operators have the subsequent priority order: $! \; > \; \widehat{P} \; > \; \circ \; > \; \lambda_P x. \; > \; + \; > \; \|$.

An occurrence of the variable x is *bound* in a process M iff it is in a term $\lambda_P x.M$ (for all $P \in \mathcal{L}$). In any other case, the occurrence of x is *free*. If x has, at least one free occurrence in M, we say that x is a *free* variable of M. We call $FV(M)$ the set of free variables of M. A process is *closed* if it does not have free variables.

The relations between the processes are described in terms of communications. The constructors $\lambda_P x.$ and $\widehat{P}$ provide the processes M and N with the possibility of communicating through a channel labelled with $P \in \mathcal{L}$ as follows: constructor of input ($\lambda_P x.$): through channel P process $\lambda_P x.M$ can receive a process which replaces the free occurrences of x in M by the incoming process through this channel; constructor of output ($\widehat{P}$): process $\widehat{P}M$ can send a process N through channel P. It then becomes inactive.

In addition, a partial order relation is defined in the set of labels which we denote as $\preceq_L$. It is required as a condition for the relation that the numerical label 0 be the minimum of the ordered set. We use the partial order relation in the set of labels (among others) to represent the application order of the real parameters to the formal parameters of a function, which is analogous to how it is done in the λ-calculus in order to treat the problem of parameterization in function calls.

For more complete information about the calculus, see [9].

3 Parallel Object-Oriented Language (POOL)

When we work with the object-oriented programming style, we always describe a computational system as a collection of self-contained entities possessing data and methods. These entities are called *objects*. Objects interact by sending messages. There are two kinds of messages: a request by the sender for the receiver to execute one of its methods with its parameters (client to server), and a reply (server to client) in response to such a method invocation. The parameters and the reply are object references. The system evolution depends on the communications, and the creation/destruction of objects is made in computing time. In the variant of POOL considered [2], a program is a sequence of class declarations together with an indicator of which class constitutes the object root.

The declaration of a class consists of a sequence of variable and method declarations together with a sentence, the body of the class. Each object is an instance of a class and its creation executes a copy of the body of the class in parallel with all the other existing objects using its own copies of the variables and class methods. The sentences are built by means of the sequential, conditional and iterative operators of the expressions, instructions of assignment and the answer instruction, which means that an object accepts a request of execution of one of its methods. Among the expressions, we can find *new(B)*, whose evaluation creates a new object of the class B and returns the reference to this object; *self*, whose evaluation returns a reference to the object in which it happens; and $E!M(E_1, \ldots, E_n)$, whose evaluation implies the left to right evaluation of the $E, E_1, \ldots, E_n$, expressions, the sending to the object for which the value of E is a request to execute method M with these parameters. The value of the expression is the reference which has been returned by the object owner of the invoked method. The activity of the transmitter is interrupted until this value is received.

Therefore, the *expressions* in POOL, the *declaration of variables*, and the *sentences* are:

$$\begin{array}{lcl} E & ::= & b \mid new(B) \mid self \mid X \mid E!M(E_1,\ldots,E_n) \mid M(E_1,\ldots,E_n) \mid Vdecs \text{ in } E \mid S;E \\ Vdecs & ::= & var\ X_1 : A_1,\ldots,X_n : A_n \\ S & ::= & X := E \mid answer \mid E \mid S1;S2 \mid \text{if } E \text{ then } S_1 \text{ else } S_2 \mid \text{while } E \text{ do } S \end{array}$$

These sentences are *assignment*, *answer* (an object accepts the request), *expression*, *sequential composition*, *choice*, and *iteration*, respectively. Note that an expression is considered as a sentence.

The declaration of *methods* (*Mdec*), *sequences of method declarations* (*Mdecs*), *declarations of a class* (*Cdec*), and *units* (*Udec*) are given by:

$$\begin{array}{lcl} Mdec & ::= & \text{method } M(Y_1 : C_1,\ldots,Y_p : C_p)\ C \text{ is } E \\ Mdecs & ::= & Mdec_1,\ldots,Mdec_q \\ Cdec & ::= & class\ A\ is\ Vdecs, Mdecs\ with\ body\ S \\ Udec & ::= & unit\ U\ is\ Cdec_1,\ldots,Cdec_r\ with\ root\ A \end{array}$$

where C is the resulting method class, $Y_1,\ldots,Y_p$ are the formal parameters, and E is the body.

The language is strongly typed, and several conditions are imposed. For example, in a unit declaration, A must be the name of one of the classes declared in $Cdec_1,\ldots,Cdec_r$ and if an expression $E!M(E_1,\ldots,E_n)$ appears in one of the class definitions and the type of E is C, then it must have a class C and a called method M of the appropriate type. The classes are only considered if they fulfill all the imposed conditions. See [2] for further details.

The language differs from POOL in some small syntactic details. The significant differences are that there is a general sentence *answer* (any method can be invoked) and not a conditional one (*answer + select*), which allows the exclusion of certain methods, and that the standard class *integer* does not appear. These differences simplify the proposal language and it can be easily translated to LCEP.

For further details about the operational semantics of POOL and of the current version of POOL see [2] and [11], respectively.

4 The Translation

It is obvious that each syntactic entity is represented as a λ_{EP}-term and composite entities are represented by their components. LCEP cannot use multiple parameters as channels. However, we can communicate through numeric channels (using currying we can model polyadic functions) or by sending some processes in parallel. This problem has an easy treatment in π-calculus, because we can send names of channels through the channels.

In LCEP we have a numeric or symbolic label P, which represents a channel through which we send a process N from an output $\widehat{p}N$ into an input $\lambda_p x.M$, where the occurrences of x in M are replaced by the process N, i.e. every valid λ_{EP}-term (variables, constants, processes, etc. but not channels). The idea of conversion is to use the name of the channel to substitute the name of the agent. This channel is then used to send the parameters and a λ_{EP}-term, which works as *output* of the result instead of the name of the method and the associated parameters. Therefore, the sender process is like an *input* which is waiting for the reply through the same channel. When we need to send a name or channel, we actually send an *output* process through this channel. Consider an expression as a variable value or an *output* process which communicates or starts the expression from the represented channel. In this paper, the process perspective is presented. Therefore, the definition of classes comes from the definition of the objects'behaviour. To simplify the notation, construction blocks are used:

$$
\begin{array}{lcl}
Value(l,v) & \equiv & \widehat{\ }_{l}\ v \\
Null(l) & \equiv & \widehat{\ }_{l}\ 0 \\
Wait(l,l') & \equiv & \lambda_{l'} v.Value(l,v) \\
Return(l,l',l'') & \equiv & \lambda_{l''} v.\widehat{\ }_{l'}\ v \circ Null(l)
\end{array}
$$

Value(l,v) represents the evaluation of an expression or variable. *Null(l)* is translated as an *output* through the channel l of the null term. *Wait(l,l′)* is an abstraction (a waiting) and a later evaluation. *Return(l,l′,l″)* is its complement.

Simple POOL expressions are translated into LCEP as follows:

$$
\begin{array}{lcl}
[new(B)](l,k) & \equiv & \widehat{\ }_{k}(\lambda_{ret} x.\widehat{\ }_{l}\ x) \\
[self](l,v) & \equiv & Value(l,v) \\
[Y](l,y) & \equiv & \widehat{\ }_{y}(\widehat{\ }_{r}\ 0) \circ \lambda_y v.Value(l,v) + \widehat{\ }_{y}(\widehat{\ }_{u}\ v') \circ \lambda_y v.Value(l,v) \\
[Var\ Y](y,v) & \equiv & Var(y,v) \\
Var(y,v) & \equiv & \lambda_y x_1.(x_1 \parallel \lambda_r x.(\widehat{\ }_{y}\ v \circ Var(y,v)) \parallel \lambda_u v'.(\widehat{\ }_{y}\ v' \circ Var(y,v')))
\end{array}
$$

The creation of a new object is modelled as a call to the class and the later reception of the new object identifier (*Channel*). This is the only exception in which we can see a label as a variable. Using a variable is equivalent to updating it or to consulting it. In any case, the declaration of a variable is the parallel composition of two answers.

POOL classes'declaration in LCEP is:

$$
\begin{array}{lcl}
[Cdec](k) & \equiv & !\ \lambda_k x.((x \parallel \widehat{\ }_{ret}\ c_i) \circ Newobject(c_i)) \\
Newobject(c_i) & \equiv & ([S] \parallel [Vdecs](\tilde{\mathrm{y}},\tilde{\mathrm{v}}) \parallel [Mdecs](\tilde{\mathrm{m}})) \\
[Mdecs](\tilde{\mathrm{m}}) & \equiv & [Mdec_1](m_1) \parallel \ldots \parallel [Mdec_i](m_i)
\end{array}
$$

In addition, we are going to make the request for the execution of a remote method into LCEP. First, let us see the translation of an object c_0 which requests the execution of a remote method m with only one evaluated parameter.

$$
[\beta_0!M(\beta_1)](l,c_0,m) \quad \equiv \quad \widehat{\ }_{c_0}(\widehat{\ }_{m}\ \beta_1 \circ \widehat{\ }_{m}(\lambda_m x.\widehat{\ }_{ret}\ x) \circ \lambda_{ret} x.\widehat{\ }_{l'}\ x) \circ Wait(l,l')
$$

In this case, a term is sent to the object (received by an *answer* into the object body), and then the object waits using *Wait(l,l′)*. The term which has been sent will execute three actions: (1) send the parameter to the method; (2) send a term which produces the answer of the method; (3) make a term which stays in the *answer* and which really sends the result to the *Wait(l,l′)*.

The declaration of a one-parameter method is:

$$
\begin{array}{lcl}
[Mdec](m) & \equiv & !\ Mmethod(m) \\
Mmethod(m) & \equiv & Mhandle(m) \parallel [Var\ Y_1](y_1,v_1) \parallel \lambda_g x.[E](m) \\
Mhandle(m) & \equiv & \lambda_m x.\widehat{\ }_{y_1}(\widehat{\ }_{u}\ x) \circ \lambda_{y_1} x.0 \circ \lambda_m x.(x \parallel \widehat{\ }_{g}\ 0)
\end{array}
$$

As you can see, the receipt of a parameter is translated by updating the variable Y_1, joined to the parallel composition with the answer sender term and with another that synchronizes with the evaluation of the method body.

Now let us see a similar process, but using multiple parameters:

$$[\beta_0!M(\beta_1,\ldots,\beta_n)](l,c_0,m) \equiv \widehat{c_0}(\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{ret}\ x)) \circ (\lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l,l')$$

The method declaration into the server object is:

$$\begin{array}{lcl}
[Mdec](m) & \equiv & !\ Mmethod(m) \\
Mmethod(m) & \equiv & Mhandle(m) \parallel [Var\ Y_1](y_1,v_1) \parallel \ldots \parallel [Var\ Y_n](y_n,v_n) \parallel \lambda_g x[E](m) \\
Mhandle(m) & \equiv & \lambda_m x.\widehat{y_1}(\widehat{u}\ x) \circ \lambda_{y_1} x.0 \circ \lambda_m x.(\widehat{u}\ x) \circ \widehat{y_n}\ x.0 \circ \widehat{m}\ x(x \parallel \widehat{g}\ 0)
\end{array}$$

If, instead of the evaluated parameters, we have expressions (knowing that a POOL expression must be left-to-right evaluated), we require that the evaluation of an expression E_i not start, since the evaluation of the expresion E_{i-1} has finished. This effect is successfully represented by:

$$\begin{array}{lcl}
[\beta_0!M(\beta_1,\ldots,\beta_{i-1},E_i,\ldots,E_n)](l,c_0,m) & \equiv & \widehat{c_0}(\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_{i-1} \circ \lambda_{l_i}\beta_i.\widehat{m}\ \beta_i \circ \widehat{g_{i+1}}\ 0 \circ \ldots \circ \\
& & \circ \lambda_{l_n}\beta_n.\widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{ret}\ x)) \circ \lambda_{ret} x.\widehat{l'}\ x \parallel \\
& & \parallel [E_i](l_1) \parallel \lambda_{g_{i+1}} x.[E_{i+1}](l_{i+1}) \parallel \ldots \parallel \\
& & \parallel \lambda_{g_n} x.[E_n](l_n) \circ Wait(l,l')
\end{array}$$

It is easy to see that the treatment of this expression may be similar to the treatment of a local call to execution of a method. Actually, it only differs in one feature. In a remote call, we need to generate a λ_{EP}-term that really executes the sending of the parameters, and makes the answer to the original object that is waiting in a *Wait(l,l')* instruction. In a local call, we do not have an answer because the purpose of a local invocation is execution, and not reply. This is the reason why the λ_{EP}-term, which results after the evaluation, is the sending of the expression. Therefore, the translation of a local call of one evaluated parameter method is:

$$[M(\beta_1)](m,l) \equiv \widehat{m}\ \beta_1 \circ \widehat{m}(\lambda_m x.\widehat{l}\ x)$$

A multiple evaluated arguments method is as follows:

$$[M(\beta_1,\ldots,\beta_n)](m,l) \equiv \widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{l}\ x)$$

If we have not evaluated parameters and what we really have are expressions (as in the remote call), then the translation into LCEP is:

$$\begin{array}{lcl}
[M(\beta_1,\ldots,\beta_{i-1},E_i,\ldots,E_n)](m,l) & \equiv & (\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_{i-1} \circ \lambda_{l_i}\beta_i.\widehat{m}\ \beta_i \circ \\
& & \circ \widehat{g_{i+1}}\ 0 \circ \ldots \circ \lambda_{l_n}\beta_n.\widehat{m}\ \beta_n \circ \widehat{m}(\lambda_m x.\widehat{l}\ x) \parallel \\
& & \parallel [E_i](l_i) \parallel \lambda_{g_{i+1}} x.[E_{i+1}](l_{i+1}) \parallel \ldots \parallel \lambda_{g_n} x.[E_n](l_n))
\end{array}$$

For a local declaration of variables with $Vdecs \equiv var\ \tilde{\mathrm{Y}}$ into an expression, we can model into LCEP as:

$$[Vdecs\ in\ E](l,\ldots,\tilde{\mathrm{y}},\tilde{\mathrm{v}}) \equiv [E](l,\ldots) \parallel [Var\ \tilde{\mathrm{Y}}](\tilde{\mathrm{y}},\tilde{\mathrm{v}})$$

where $[Var\ \tilde{\mathrm{Y}}](\tilde{\mathrm{y}},\tilde{\mathrm{v}}) \equiv [Var\ Y_1](y_1,v_1) \parallel \ldots \parallel [Var\ Y_v](y_v,v_v)$.

Now, let us see the translation into LCEP of the auxiliary expressions whose definitions were postponed:

$$\begin{array}{lcl}
[Wait(\beta)](l,l') & \equiv & Wait(l,l') \\
[E \Rightarrow \beta](l,\ldots) & \equiv & [E](l',\ldots) \parallel Return(l,l',l'') \\
[E,\rho](l,\ldots) & \equiv & [E](l,\ldots) \parallel [\rho](\tilde{\mathrm{y}},\tilde{\mathrm{v}}) \\
[\rho](\tilde{\mathrm{y}},\tilde{\mathrm{v}}) & \equiv & Var(y_1,v_1) \parallel \ldots \parallel Var(y_n,v_n) \\
[nil](l) & \equiv & Null(l) \\
[Y := E](l,\ldots,y) & \equiv & [E](l',\ldots) \parallel \lambda_{l'} v'.(\widehat{y}(\widehat{u}\ v') \circ \lambda_y v.Null(l)) \\
[answer](a,l) & \equiv & \lambda_a x.x \circ Null(l)
\end{array}$$

We express the assignment as the parallel composition of two λ_{EP}-terms: one which evaluates the expression and sends the result through the auxiliary channel l', and the other which receives the result of the expression from this channel and updates the variable Y by assigning the value of the expression.

With regard to the *answer* which replies to the request of remote invocation of methods, $[E_0!M(E_1,\ldots,E_n)]$, we can see that it accepts any request of any ready method, allowing to the replicator of the method to execute a copy of itself. It returns the result (to activate the client process which is in a *wait* instruction) and later returns a *null* through its own channel l to finish. Although only the last channel is really defined in the *answer*, all the others are sent from the client, as a λ_{EP}-term like $\widehat{m}\ \beta_1 \circ \ldots \circ \widehat{m}\ \beta_n \circ n\widehat{m}(\lambda_m x.\widehat{ret}\ x) \circ \lambda_{ret}x.\widehat{l'}\ x$ if there is no unevaluated expression.

The fundamental part of the λ_{EP}-term are the last two terms of the sequence. The second to the last term is a sender through the channel *ret* of the reply in a method execution. The last one takes that answer and re-sends it to the *Wait(l,l')* in the other process and it then becomes inactive.

The basic expression of the standard class *Bool* is as follows:

$$
\begin{array}{lcl}
[true](l,b_k) & \equiv & \widehat{b_k}(\widehat{t}(\lambda_{ret}x.\widehat{l'}\ x) \circ \lambda_l x.Value(l,x)) \\
[false](l,b_k) & \equiv & \widehat{b_k}(\widehat{f}(\lambda_{ret}x.\widehat{l'}\ x) \circ \lambda_l x.Value(l,x))
\end{array}
$$

And the standard class *Boolean* is:

$$
\begin{array}{lcl}
[Boolean](b_k) & \equiv & (BoolClass(b_k) \parallel Bool(b,b_k)) \\
BoolClass(b_k) & \equiv & !\ \lambda_{b_k}x.(x \parallel (\lambda_t x.(x \parallel \widehat{ret}\ \widehat{true}\ 0) \parallel \lambda_f x.(x \parallel \widehat{ret}\ \widehat{false}\ 0))) \\
Bool(b,b_k) & \equiv & (!\ BoolBody(b) \parallel !\ BoolVal(m_1) \parallel !\ BoolNot(m_2,m_1) \parallel \\
 & & \parallel !\ BoolAnd(m_3,m_1)) \\
BoolBody(b) & \equiv & \lambda_b x.x \\
BoolVal(m_1) & \equiv & \lambda_{m_1}x.(\widehat{m_1'}\ x \parallel \lambda_{m_1}y.(\lambda_{m_1'}z.(y \parallel \widehat{m_1}\ z))) \\
BoolNot(m_2,m_1) & \equiv & \lambda_{m_2}x.(\widehat{m_1}\ x \circ \widehat{m_1}(\lambda_{m_1}x.\widehat{m_2}x))\circ \\
 & & \circ\lambda_{m_2}x.(x \parallel \lambda_{true}y.\widehat{m_2}\ \widehat{false}\ 0 \parallel \lambda_{false}y.\widehat{m_2}\ \widehat{true}\ 0) \\
BoolAnd(m_3,m_1) & \equiv & \lambda_{m_3}x.(\widehat{m_1}\ x \circ \widehat{m_1}(\lambda_{m_1}x.\widehat{m_3'}x)) \circ \lambda_{m_3}x.(x \parallel \lambda_{true}y.(\lambda_{m_3}x.(\widehat{m_1}x\circ \\
 & & \widehat{m_1}(\lambda_{m_1}x.\widehat{m_3'}x)) \circ \lambda_{m_3'}x.(x \parallel \lambda_{true}y.(\widehat{m_3}\ \widehat{true}\ 0) \parallel \\
 & & \parallel \lambda_{false}y.(\widehat{m_3}\ \widehat{false}\ 0)) \parallel \lambda_{false}y.(\lambda_{m_3}z.\widehat{m_3}\ \widehat{false}\ 0))
\end{array}
$$

To model the conditional instruction we have:

$$
\begin{array}{lcl}
[if\ E\ then\ S_1\ else\ S_2](l) & \equiv & BoolEval(l',l_1,l_2) \parallel [E](l') \parallel \lambda_{l_1}x.[S_1](l) \parallel \lambda_{l_2}x.[S_2](l) \\
BoolEval(l',l_1,l_2) & \equiv & \lambda_{l'}x.\widehat{b}(\widehat{m_1}\ x \circ (\widehat{m_1}(\lambda_{m_1}x.\widehat{ret}\ x)))\circ \\
 & & \circ(\lambda_{ret}x.x \parallel \lambda_{true}x.\widehat{l_1}\ 0 \parallel \lambda_{false}x.\widehat{l_2}\ 0)
\end{array}
$$

With regard to the sequence of sentences, if we have the sentence $[S_1;S_2]$, and we suppose that the sentence S_1 is an expression, then:

$$
[S_1;S_2](l,\ldots) \ \equiv\ [S_1](l',\ldots) \parallel \lambda_{l'}v.[S_2](l,\ldots)
$$

where v is a variable which appears in S_2 and is a bound variable.

On the contrary, if the sentence S_1 is not an expression, therefore, v is not bound in S_2, and the behaviour of the process is like $[S_1](l',\ldots) \circ [S_2](l,\ldots)$, with $l' : Null$. Knowing this and using the dual behaviour of expressions and non-expression sentences in the translation to LCEP of iterative instructions, we can find a new example of the use of the standard class *Bool* by *BoolEval*.

$$
[while\ E\ do\ S](l,\ldots) \ \equiv\ Loop(l,l',\ldots) \parallel !\ \lambda_{l'}v.Loop(l,l',\ldots)
$$

```
CLASS PHILOSOPHERS
   VAR
      eat : boolean
      sig : boolean
      chopl : CHOPS
      chopr : CHOPS
   METHOD ToEat( ) : PHILOSOPHERS
      BEGIN
         WHILE not Eat DO
            sig := true;
            WHILE sig DO
               sig := chopl!Takechop( )
            OD;
            sig := chopr!Takechop();
            IF sig
               THEN eat := false;
                  chopl!Leavechop( )
               ELSE eat := true
            FI
         OD
         RESULT SELF
      END;

   METHOD Tothink( ) : PHILOSOPHERS
      BEGIN
         chopl!Leavechop( );
         chopr!Leavechop( );
         eat := false;
         RESULT SELF
      END;
   BODY
      eat := false;
      WHILE true DO
         Toeat( )
         Tothink( )
      OD
   YDOB

CLASS CHOPS
   VAR
      freechop : boolean
   METHOD Takechop ( ) : boolean
      BEGIN
         IF freechop
            THEN
               freechop := false;
               RETURN false
            ELSE RETURN true;
         FI
      END;
   METHOD Leavechop ( ) : CHOP
      BEGIN
         freechop := true;
         RESULT nil
      END;
   BODY
      freechop := true;
      WHILE true DO
         answer
      OD
   YDOB
```

Figure 1: The POOL philosophers problem.

If S is an expression then c is a bound variable and therefore:

$$Loop(l, l', \ldots) \equiv (BoolEval(l'', l_1, l_2) \parallel [E](l'') \parallel \lambda_{l_1} x.[S](l', \ldots) \parallel \lambda_{l_2} x.Value(l, v))$$

If S is not an expression, then v is a free variable in S and then:

$$Loop(l, l', \ldots) \equiv (BoolEval(l'', l_1, l_2) \parallel [E](l'') \parallel \lambda_{l_1} x.[S](l', \ldots) \parallel \lambda_{l_2} x.Null(l))$$

5 Example of translation

To reflect the translation of a POOL program into LCEP, we use an easy variant of the well-known philosophers problem [5], in which we pay attention to the communication between the different objects. To model that behaviour in POOL, we consider two object classes: the *philosopher* class and the *chop* class.

We are basically interested in communication and method invocation. Therefore, we center our attention on the implementation of the classes, and we do not show *units* or specifications in our solution. We are not looking for an efficient solution to the concurrence problem. Therefore, we do not pay attention to aspects such as the non-deterministic choice between left and right chop (we necessarily take the left one before the right one). We can see the POOL example in Fig. 1.

From lack of space, we only show some parts of the translation following a similar scheme as in Section 4.

The translation of the philosophers'class declaration to LCEP is as follows:

$$\begin{aligned} [Cphils] &\equiv\ !\, \lambda_{phils} x.((x \parallel \widehat{ret}\ phil_i) \circ Newobject(phil_i)) \\ Newobject(phil_i) &\equiv (BODYphil(phil_i)) \parallel [Vdecs](eat, sig, chopl, chopr) \parallel \\ &\quad \parallel [Mdecs](toeat, tothink)) \end{aligned}$$

An example of variable declaration is:

$$\begin{aligned} Var(eat, v_b) \equiv\ & \lambda_{eat} x_1.(x_1 \parallel \lambda_r x.(\widehat{eat}\ v_b \circ Var(eat, v_b)) \parallel \\ & \parallel \lambda_u v'.(\lambda_{eat} v_{b'} \circ Var(eat, v_{b'}))) \end{aligned}$$

The invocation of the method *takechop* of an object *chop* from a *philosopher* is modelled in LCEP as:

$$\begin{array}{lcl} [chopl\ !\ takechop(\ )](l, chopl, tchop) & \equiv & \widehat{chopl}(\widehat{tchop}(\lambda_{tchop} x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l, l') \\ [chopr\ !\ takechop(\ )](l, chopr, tchop) & \equiv & \widehat{chopr}(\widehat{tchop}(\lambda_{tchop} x.\widehat{ret}\ x) \circ \lambda_{ret} x.\widehat{l'}\ x) \circ Wait(l, l') \end{array}$$

Furthermore, the definition of the method *takechop( )* is:

$$\begin{array}{lcl} [Mtakechopdec](tchop) & \equiv & !\ Mtakechop(tchop) \\ Mtakechop(tchop) & \equiv & Mhtakechop(tchop) \parallel \lambda_g x.[E\_takechop](tchop) \\ Mhtakechop(tchop) & \equiv & \lambda_{tchop} x.(x \parallel \widehat{g}\ 0) \end{array}$$

An example of a local call appears at the method *toeat( )* in a philosopher body:

$$[toeat()](toeat, l') \equiv \widehat{toeat}(\lambda_{toeat} x.\widehat{l'}\ x)$$

And the declaration of method *toeat( )* is:

$$\begin{array}{lcl} [Mtoeatdec](toeat) & \equiv & !\ Mtoeat(toeat) \\ Mtoeat(toeat) & \equiv & Mhtoeat(toeat) \parallel \lambda_g x.[E\_toeat](toeat) \\ Mhtoeat(toeat) & \equiv & \lambda_{toeat} x.(x \parallel \widehat{g}\ 0) \end{array}$$

Our example contains a conditional instruction into the body of method *toeat( )* in the class *chop*:

```
IF sig
    THEN eat := false;
        chopl ! Leavechop()
    ELSE eat := true
FI
```

Therefore:

$$\begin{array}{lcl} [if\ SIG\ then\ S_1\ else\ S_2](l) & \equiv & (BoolEval(l', l_1, l_2) \parallel (\widehat{sig}(\widehat{r}\ 0) \circ \lambda_{sig} v.Value(l', v)) \parallel \\ & & \parallel \lambda_{l_1} x.[S_1](l) \parallel \lambda_{l_2} x.[S_2](l) \end{array}$$

The following is a practical example of the translation of the loop into the body of object class *philosopher*:

```
WHILE true
    DO
        Toeat( )
        Tothink( )
    OD
```

Therefore:

$$\begin{array}{lcl} While_0(l) & \equiv & [while\ true\ do\ S_3](l) \equiv Loop_0(l, l') \parallel !\ \lambda_{l'} c.Loop_0(l, l') \\ Loop_0(l, l') & \equiv & (BoolEval(l'', l_1, l_2) \parallel [true](l'', b_k) \parallel \lambda_{l_1} x.[S_3](l') \parallel \lambda_{l_2} x.null(l)) \end{array}$$

S_3 is the sequential composition of procedures *Toeat()* and *Tothink()*, which can be translated as:

$$\begin{array}{lcl} [S_3](l') & \equiv & [toeat(\ )](toeat, l') \parallel \lambda_{l'} v.[tothink(\ )](tothink, l) \\ [toeat(\ )](toeat, l') & \equiv & \widehat{toeat}\ \lambda_{toeat} x.\widehat{l'}\ x) \\ [tothink(\ )](tothink, l) & \equiv & \widehat{tothink}\ \lambda_{tothink} x.\widehat{l'}\ x) \end{array}$$

where v is a free variable in $[tothink(\ )](tothink, l)$.

6 Conclusions and future work

An extension of LCEP has been presented. Under a process perspective, this extension can model a simple version of a parallel object-oriented language, LCEPOO. The phrase-by-phrase translation process has been made by modelling object definition, management, communication, and evolution features. The object activities are synchronized by sending messages which contain references to other system objects. Among the characteristics of LCEP, we can not find the sending of communicating channels. The absence of this feature has not been an obstacle for the modelling of the behaviour of the chosen POOL variant. We have incorporated the possibility of sending variables (channels) only for the modelling of the object generation.

As we have shown, LCEP is an effective tool which has sucessfully been used to model parallel and concurrent processes as well as to model object-oriented language features. A translator system from a high level language (ALEPH) into LCEP is already available [4]. By transferring the proposal extension to that system, we can obtain an explicitly parallel high-level language with object-oriented features.

References

1. H. Aït-Kaci and J. Garrigue. Label-Selective λ-Calculus: Syntax and Confluence. In *Proc. of the 13th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1993.
2. P. America, J. de Bakker, J. Kok, and J. Rutten. Operational Semantics of a Parallel Object-Oriented language. *In Proc. of the 13^{th} Symposium on Principles of Programming Languages*, pages 194–208, 1986.
3. P. America, J. de Bakker, J. Kok, and J. Rutten. Denotational semantics of a parallel object-oriented language. In *Information and Computation*, volume 83, pages 152–205. 1989.
4. L. Climent, M.L. Llorens, and J. Oliver. LCEP as an Abstract Machine. Technical Report DSIC-II/38/97, UPV, 1997.
5. E.W. Dijkstra. Cooperating Sequential Processes. In F. Genus, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.
6. S. Lucas and J. Oliver. Parallel label-selective λ-calculus (LCEP). In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proc. of 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, pages 125–139, 1994.
7. R. Milner. A Calculus of Communicating Systems. *LNCS 92*, 1980. Springer Verlag.
8. R. Milner. The polyadic π-calculus: A tutorial. In F.L. Brauer, W. Bauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specifications*. Springer-Verlag, Berlin, 1993.
9. J. Oliver. *Extensions of the λ-calculus for Modelling Concurrent Processes*. PhD thesis, DSIC (UPV), 1996.
10. F. Vaandrager. A process algebra semantics in POOL. *Applications of Process Algebra, Tracts in Theoretical Computer Science*, 17:173–236, 1990.
11. D. Walker. Objects in the π-calculus. In *Information and Computation*, volume 116, pages 253–271. 1995.