Programming with behaviors in an ML framework – The syntax and semantics of LCS

Bernard Berthomieu[1] and Thierry Le Sergent[2]

[1] LAAS/ CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex, France.
e-mail: Bernard.Berthomieu@laas.fr
[2] LFCS, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, UK.
e-mail: tls@dcs.ed.ac.uk (*On leave from LAAS/CNRS*)

Abstract. LCS is an experimental high level asynchronous parallel programming language primarily aimed at exploring design, implementation and use of programming languages based upon the behavioral paradigms introduced by CSP and CCS. The language extends Standard ML with primitives for concurrency and communication based upon a higher order extension of the CCS formalism. Typechecking enforces consistency of communications. An abstract operational semantics of the language is given in terms of a transition system.

1. Introduction

Motivations

A number of parallel programming languages are designed by extending sequential languages. Parallel programming facilities are provided through a set of primitives with a functional interface typically including a function for creating processes with some given text (e.g. fork), functions for implementing process communications and synchronization (e.g. channel, input, output, sync), and possibly some functions combining processes together (e.g. select). Examples of such languages are Poly/ML [12] and CML [17], both extending the language Standard ML.

Though concurrency and communications are provided through the use of functions, the semantics of these languages depart from the usual functional semantics due to the complex apparatus needed for casting nondeterminism and nonterminating computations into this paradigm. They are usually given a semantics in operational terms, by some reduction relation on programs of the language.

An alternative to this functional introduction of concurrency and communication is the behavioral approach promoted by CSP, CCS [13], and subsequent formalisms. The approach has been undoubtedly fruitful in the last decade in the areas of specification and analysis of concurrent systems. The central role in this paradigm is played by communications: a behavior describes a way of communicating with other behaviors, a program is some combination of communicating entities. The approach is inherently parallel, nondeterministic, and nonterminating systems are naturally expressed. The semantics of these languages are given in terms of some observation relation characterizing in some sense their communication capabilities.

Though the behavioral paradigm is often advocated as a convenient formalism for concurrency and communication, there has been very few work in designing programming languages that would canonically implement the paradigm. Noticeable exceptions are the language LOTOS [9], the language LCS introduced here and, to a lesser extend, the FACILE language [7].

Anatomy of LCS

LCS is an high level asynchronous parallel programming language primarily aimed at exploring design, implementation and use of programming languages based upon the behavioral paradigm. The language and its implementations evolved through several versions [2] to that described here. A "sequentially running" implementation of LCS was made available to users in 1991 [4]; parallel and distributed implementations of the language are under way [10] [11].

Syntactically, LCS is designed as an extension of Standard ML [14]. Processes are described in an ML framework, but parallel programming capabilities are provided through a specific sub-language of behavior expressions rather than as a set of functions. Behavior expressions describe processes and evaluate to behavior values; behavior values may be turned into processes by specific process creation commands. A non trivial extension of the ML static polymorphic discipline enforces consistency of communications.

The language of behavior expressions is based on Robin Milner's Calculus of Communicating Systems [13], which is widely accepted as a convenient formalism for concurrency and communication. However, CCS, as a bare abstract formalism, lacks several ingredients for constituting a convenient programming language (e.g. how to express usual values) and also suffers some known deficiencies in terms of expressiveness. In particular, CCS is a first order formalism with respect to behaviors and to communication labels, and its set of behavior combinators may lack the flexibility required for comfortably programming real applications. Finally, CCS does not consider at all assignable values and side-effects. LCS proposes solutions to all these problems that, as far as possible, preserve the underlying theory of CCS and follow the SML design principles as well.

Organization of the paper

Section 2 introduces the features of LCS and shows its embedding into Standard ML. Section 3 describes its original typing technique for behaviors. The operational semantics of LCS programs is detailed in Section 4 in terms of a transition system and an observation relation. Section 5 briefly overviews implementation. The concluding section summarizes the experience and compares it with related projects. It is assumed throughout that the reader has some knowledge of the essential features of both the language ML and the CCS formalism.

2. Syntax and features of LCS

Syntax

The language of expressions extends that of Standard ML with constructs for building behaviors. The essentials of the syntax of LCS is given in Table 2.1. The set of ML types is also enriched with "behavior types"; the typing aspects will be discussed in Section 3. In addition to the capabilities of an SML implementation, LCS implementations have commands for starting processes to run in the foreground or background in parallel with all other running processes.

Behaviors must not be confused with processes. Evaluation of a behavior expression does not produce a process, but rather a closure similar to a function value. Creating a process from a behavior value by one of the process creation commands causes the recursive evaluation of the body of the behavior. The operational semantics of LCS processes is detailed in Section 4.

Behaviors as communication capabilities

Behaviors must be understood as interaction capabilities. The behavior constructs of LCS strictly include those of CCS. Expression **stop** denotes the behavior that has no communication capability; behaviors are built from **stop**, possibly recursively, by action prefixing, compositions, restrictions and relabellings.

<table>
<tr><td>exp::=</td><td><sml expression> | beh</td><td></td></tr>
<tr><td>beh::=</td><td>**stop**</td><td>null behavior</td></tr>
<tr><td>|</td><td>{**do** e} => exp</td><td>commitment action</td></tr>
<tr><td>|</td><td>port ? {pat} => exp</td><td>input action</td></tr>
<tr><td>|</td><td>port ! {exp} => exp</td><td>output action</td></tr>
<tr><td>|</td><td>**signal** exp {**with** exp}</td><td>event action{with message}</td></tr>
<tr><td>|</td><td>exp C exp</td><td>compositions</td></tr>
<tr><td>|</td><td>exp **catch** erules</td><td>event handling</td></tr>
<tr><td>|</td><td>exp "{"/lab$_1$ ... lab$_n$"}"</td><td>label restriction</td></tr>
<tr><td>|</td><td>exp "{"lab$_1$’ ... lab$_n$’/lab$_1$ ... lab$_n$"}"</td><td>label relabelling</td></tr>
<tr><td>C ::=</td><td>/\ | //\ | /\\ | //\\</td><td>parallel compositions</td></tr>
<tr><td>|</td><td>\/ | \\/ | \// | \\//</td><td>choice compositions</td></tr>
<tr><td colspan="2">erules ::=exp **with** match {|| exp **with** match}</td><td>event handler rules</td></tr>
<tr><td>port ::=</td><td>label {# exp}</td><td>communication ports</td></tr>
<tr><td colspan="2">match ::= pat => exp {| match}</td><td>SML matches</td></tr>
<tr><td>pat ::=</td><td><sml pattern></td><td>SML patterns</td></tr>
</table>

Table 2.1. LCS expressions

The construct “=>” prepends an action to a behavior. Actions may be either that of performing an internal move (the τ action of CCS), that of proposing a value on a communication port (output or signal), or that of accepting a value on a port (input or event handling).

There are three basic forms of compositions: parallel (/\), choice (\/) and **catch**; the first two are inherited from CCS. Processes composed in parallel may communicate values. The single communication and synchronization primitive is rendez-vous over named ports. Choice composition implements a selection mechanism, the first process in a choice composition that performs an action makes the alternative process(es) in the composition terminate. The last kind of composition (catch) is a particular parallel composition with the additional effect of terminating the behavior on the left side of catch as soon as a communication has occurred between the members of the composition. The signal and catch expressions implement a process-level exception mechanism that will be shortly described in more detail.

As in the CHOCS calculus [19], the class of behaviors and the class of messages in LCS are the same syntactic class. One can freely pass behavior values (not processes) as messages, or declare functions taking behaviors as arguments and/or returning behaviors as results.

Communication ports

The basic CCS formalism does not allow communication links to be passed between behaviors. Richer formalisms have been proposed that provide such a facility [6] [15]. LCS settled for a more conservative approach based on parametric channels as introduced in [1]; this permits to retain most of the underlying theory of CCS. LCS offers to compute communication ports, in some sense, instead of offering passing labels or channels between behaviors.

Ports in LCS have two components: a label in the class of identifiers, and an extension. Extensions may be any expression denoting a value of a type admitting equality (behavior values, like function values, do not admit equality; event values must). All ports have an extension, the default extension is the value “()” of type `unit`. Beside their extensions, the treatment of communication ports is that of CCS: ports do not denote values and have global scope unless otherwise mentioned, in contrast with the treatment retained in PFL [8] or FACILE [7].

Operationally, port labels are implicitly bound to communication lines (one for each possible extension of the label) by the enclosing context. The scope of labels is controlled with restrictions. The restriction of p in b (written b{/p}) has the effect of delimiting the scope of ports labelled p occurring within b to expression b (as the lambda-abstraction in λp.b delimits the scope of the inner p to expression b). Relabellings help to build connected systems of behaviors; the relabelling (b{p/q}) restricts label p and makes ports labelled q within b appear to the enclosing context as having label p. The restrictive effects of LCS relabellings make then slightly differents from CCS relabellings.

Labels may be hidden or renamed, but may not be computed nor passed as messages or parameters. On the other hand, extensions may be computed or passed, but may not appear in restrictions or relabellings; these apply to all ports with the labels involved, collectively. This treatment allows one to compute communication ports (their extensions, actually), while preserving the possibility of static typechecking.

Beside permitting a form of mobility [15], port extensions are particularly convenient for implementing systems constituted of arrays of processes in which the behavior of the individual processes can be parameterized by their index in the structure such as neural or systolic systems.

A simple example

The following behavior offers all prime numbers on a port labelled `out` by the Eratosthenes sieve method. `primes` is built from a behavior `succs` that offers all integers greater than its parameter on a port `out` (with the trivial extension), a behavior `filter` that echoes the integer it reads except those which are multiple of its parameter, an adhoc infix piping combinator (`^`) and the recursive `sieve` behavior. On output of each prime number, behavior `sieve` evolves to itself preceded by a filter behavior that absorbs all multiples of that prime number.

```
val primes = (* offers prime numbers on port out *)
    let fun a ^ b = (a {tmp/out} /\ b {tmp/inp}) {/tmp}
        fun succs n = out! n=> succs (n+1)
        fun filter p = inp? x=>
                if x mod p = 0 then filter p else out! x=> filter p
        val rec sieve = inp? x=> out! x=> (filter x ^ sieve)
    in succs 2 ^ sieve
    end;
```

Events, an exception mechanism at process level

Events, and the related event raising (**signal**) and event trapping (**catch**) expressions, implement an exception mechanism at process level. The exceptions we are interested in raising and trapping here are those related with communications such as the inability of some process to provide an offer on some communication port. Such exceptions would typically be handled by a reconfiguration of all or part of the current system of processes.

Events implement a particular kind of communication ports, obeying their own scope rules. Events are built from event constructors declared through specific declarations. Event declarations possibly make explicit a parameter type and a message type associated with the constructors. Since events must be comparable for equality, their parameter, if any, must admit equality in the SML sense. The message type associated with the constructor is the type of messages transmitted to the handlers when events built with that constructor are trapped.

Raising an event by **signal** offers the message associated with the event on a communication port identified by the event itself. Handling an event by **catch** must be understood as a communication between the process that offers the event and the handler. In addition to passing a message, this communication has the effect of terminating all processes under the catch.

Event constructors follow the same static scoping rules than other constructors, but the communication ports defined by events have particular scope rules. An event can only be caught by a handler if issued from the behavior on the left side of the catch composition. Further, the scope of an event stops at the innermost handler for that event. That way, events propagate like exceptions in SML, but event handling differs from exception handling in that signalling an event has no effect when no surrounding handler may catch it.

Beside their use as an exception mechanism, events are also useful as a clean-up mechanism to abort, on completion on some work, all the possibly awaiting processes that have been created as part of this work. The next example shows for instance how the previous prime behavior could be used to create an other behavior that offers on port `out` the nth prime number. Behavior `nthp` spawns primes, skips all primes up to the nth, and then raises a BYE event carrying that nth prime as message. The handler for event `BYE` offers its message on `out`; handling the event has the effect of aborting all processes resulting from execution of `primes`.

```
fun nthp n = (* offers nth prime on out *)
    let event BYE with int
         fun skip n = out? x=>
                 if n>1 then skip (n-1) else signal BYE with x
    in (primes /\ skip n) {/out}  catch BYE with p => out! p=> stop
    end;
```

Side effects, imperative features

Any evaluation may have a side-effect. Reference values are handled as in SML, but memory locations have restricted scopes here. Each running instance of a behavior (i.e., each process) is associated with a store; side-effects in evaluation of a process are restricted in scope to the store associated with it. That store may be private, or shared with some other process(es).

The basic store assignment mechanism for processes is inheritance. When started, a process inherits the store of the toplevel process or a copy of it depending on the process creation command used. The stores of processes created from expansions of running processes are determined by the composition combinators. The light compositions (`/\` and `\/`) create processes that both inherit the current store and share it. With left strong compositions (`//\` and `\\/`), the process on the left hand side inherits a copy of the current store, while the process on the rhs inherits the current store. The other composition operators can be derived from those (see their semantics in Chapter 4). The **catch** construct acts as a light composition; strong catch compositions can easily be simulated with the existing composition operators, if required.

An example of use of strong compositions is provided by the implementation of the LCS interactive toplevel. User environments are conveniently kept in references, updated at each toplevel declaration; the LCS toplevel itself is implemented as a system of LCS processes sharing the same store. Creating a multi-user implementation of LCS is which all users having successively logged in get the same initial environment, but do not share its further modifications is naturally implemented with strong compositions.

For reasons that will be explained in Section 4, port extensions and messages are not allowed to hold or encapsulate reference values. Current implementations, however, do not enforce this constraint; both static and dynamic methods are being investigated for that purpose.

User interface

In addition to the toplevel commands provided by SML user interfaces (declarations), LCS interactive user interfaces offer commands for process management.

Process creation commands create processes to run in parallel with all other currently running processes (including the toplevel). As processes successively started may have to communicate, it is essential that these processes agree on the types of the ports they use to communicate. This typing constraint is enforced when processes are started, rather than when declared. In other words, communication ports are not bound to actual communication lines at compile-time, but are dynamically bound when processes are started. For implementing the type constraint, the LCS toplevel maintains a type information that, at any time, holds the type of the parallel composition of all processes started from the beginning of the session or the last clean-up command for user processes. This type is updated by process creation commands and reset by the process clean-up command.

Commands **start** and **lstart**, taking a behavior expression as parameter, create processes to run in background. They return immediately. The former assigns to the process a private store consisting of a copy of the store of the toplevel process; processes started by the latter share their store with that of the toplevel process. It is occasionally useful to run processes in foreground rather than background (e.g. when the user process receives input from the same window that the toplevel). Foreground running capabilities are provided by commands **call** and **lcall**, which create processes with private or shared storage, respectively, together with a specific **return** behavior, semantically equivalent to **stop**. Evaluation of **return** makes the corresponding call command return (evaluation of the called process resumes in background).

Finally, a **kill** command allows the user to aborts all the processes started, and a **reboot** command permits to restart the LCS runtime on any user provided behavior; this last command is used for bootstrapping new versions of the compiler and building stand-alone applications.

3. Typing behavior expressions

The typing rules for behavior expressions must enforce type-safe communications between processes. A simple rule ensures this: whenever their scopes intersect, ports with the same label must transmit values of the same type and must have extensions of the same type. The rule is sufficient though not necessary; it is only necessary for pairs of matching ports which may actually be involved in a communication, but, obviously, this cannot be generally inferred at compile-time. In addition, port extensions must be restricted to values of types admitting equality in the Standard ML sense since comparing extensions for equality is needed to determine actual communications capabilities at run time.

In languages in which communication channels are values, such as PLF or CML, channels are given a type (α *chan*), in which α is the type of the values the channel may pass, and processes are given a trivial constant type. A distinct approach, suitable in languages obeying the behavioral paradigm, such as LCS, is to assign to processes types that tell, for each channel they may use for communication, the type of items that may be passed through this channel. In addition, LCS behavior types will assign to each such label the type of its extension. Now, higher order prevents to compute the exact set of ports through which some behavior may communicate; behaviors will thus be assigned types which make precise an extension and a message type for every possible communication label.

Syntactically, LCS behavior types are built from special type variables called behavior type variables and written `_a`, `_b`, etc., possibly prefixed by a finite number of fields of the form `{label: extension_type # message_type}` where `extension_type` is an equality type. Regular type variables (`'a`, `'b`, etc.) may be substituted by any type, including behavior types, but behavior type variables may only be substituted by other behavior type variables or behavior types. Behavior types thus constitute a strict subset of all types.

Semantically, behavior types may be read as total functions assigning types to labels. A behavior type variable assigns to every possible label a type `''a#'b` with variables `''a` and `'b` not occurring elsewhere (`''a` is an equality type variable). The type $\{p{:}\tau\}\rho$ denotes the function that associates type τ to label p, and type (ρ q) to labels q distinct from p. Unconstrained behavior types sharing no variables are denoted by distinct behavior type variables.

As an example, the type `{q:bool#int,r:unit#('a*bool)}_b`, abbreviated `{q.bool:int,r:'a*bool}_b`, is the most general type for behaviors that may send or receive integers through ports labelled `q` indexed by booleans and pairs of type `'a*bool` through ports labelled `r` with the default extension. Its unification with the type `{s:int, r:'a->int*'b}_c` yields the type `{q.bool:int, r:'a->int*bool, s:int}_d.`

Behavior types are conveniently formalized as a particular subset of the "Tagged Types" introduced in [3] where their theory, including canonical forms and unification algorithms is developed in depth. Unification of Tagged Types is reducible to standard unification for first order terms. Conceptually, tagged types bear strong relationships with Wand's row types [20] and Rémy's record types [18], but the logical and technical treatments of tagged types are original and yield, we beleive, a simpler and more intuitive treatment.

The typing rules for behavior expressions are summarized in Table 3.1; other expressions of the language are typed as in SML. Events receive type `(ty evt)` where `ty` is the type of the message brought by the event.

expression	is typed as	where	
stop	Stop	Stop	: _a
=> e	Tau e	Tau	: _a -> _a
p#e?x=>e'	Input_p e (λx.e')	Input_p	: ''a -> ('b -> {p.''a: 'b}_c) -> {p.''a: 'b}_c
p#e!e'=>e''	Output_p e e' e''	Output_p	: ''a -> 'b -> {p.''a: 'b}_c -> {p.''a: 'b}_c
e{/p}	Hide_p e	Hide_p	: {p.''a: 'b}_c -> {p.''d: 'e}_c
e{q/p}	$\text{Rename}_{q,p}$ e	$\text{Rename}_{q,p}$	: {p.''a: 'b, q.''c: 'd}_e -> {p.''f: 'g, q.''a: 'b}_e
e C e'	Compose (e,e')	Compose	: _a * _a -> _a

$$\frac{A \vdash e : ty\ \ evt \quad A \vdash e' : ty}{A \vdash \text{signal } e \text{ with } e' : \_b} \qquad \frac{A \vdash e_i : ty_i\ \ evt \quad A \vdash e_i' : ty_i \quad A \vdash e : \_b \quad A \vdash h_i : \_b}{A \vdash e \text{ catch } e_1 \text{ with } e_1' \Rightarrow h_1 \parallel \ldots \parallel e_n \text{ with } e_n' \Rightarrow h_n : \_b}$$

Table 3.1. Type inference for behavior expressions

As an example, the "pipe" combinator (^) defined in the body of behavior `primes` in Section 2 would receive as principal type:

^ : {out.''a: 'b}_c * {inp.''a: 'b, tmp.''d: 'e}_c ->{tmp.''f: 'g}_c

Which can also be written in another, more verbose, canonical form:

^ : {inp.''a: 'b, out.''c: 'd, tmp.''e: 'f}_g * {inp.''c: 'd, out.''h: 'i, tmp.''j: 'k}_g
-> {inp.''a: 'b, out.''h: 'i, tmp.''l: 'm}_g

It can be seen from this type that port `inp` (resp. `out`) has in the type of expression (`A^B`) the type that port `inp` has in `A` (resp. port `out` has in `B`).

4. Operational semantics

Core formalism and translation

The operational semantics of LCS is obtained from the semantics of a richer "core" language. Expressions e of this core language include lambda-expressions, with a recursion combinator, the usual constants, primitives for references, and the following behavior expressions:

$$b ::= \text{stop} \mid \tau.e \mid p(e_1)!e_2.e_3 \mid p(e_1)?x.e_2 \mid e\backslash E \mid e[S] \mid e_1|e_2 \mid e_1+e_2 \mid e_1 < e_2 \mid \langle e\rangle$$

p, q, r, etc, are labels in some denumerable set Σ.

This core language extends CCS in several ways: lambda abstraction and application are primitive; behaviors may be passed as messages between processes; communication labels are parameterized; there is a new binary combinator (written < and read "interrupt") and, finally, there is a specific construction for management of stores: < >. We shall not precisely give here a concrete syntax for restriction and relabelling expressions, but simply assume that these expressions may at least denote restriction or relabelling of either a single port or of all ports bearing some given label. Port extensions are restricted to some subset of expressions admitting canonical forms (corresponding to the expressions having equality types).

The translation of LCS expressions into core expressions is summarized in Table 4.1, $\lceil\ \rceil$ is the translation function. The translation of functional expressions is standard and has been omitted. The expression "**signal** `e` **with** `m`" is interpreted as an output offer of message `m` on a port made from a label χ, assumed not part of Σ, with extension `e`. The sequence of handlers for the different events is interpreted as the disjunction of the handlers, each guarded by an input on a port labelled ζ (with $\zeta \notin \Sigma$) with the corresponding event expression as extension. It results from the translation of the catch construct that the dynamic scope of an LCS event extends no further than its innermost handler. The effects of the other composition operators are obtained from a combined use of the core operators |, + and < >.

$$\begin{array}{rlrl}
\lceil \{\text{do } e\} \Rightarrow e'\rceil & = \tau.\lceil(\{e;\}\ e')\rceil & \lceil \textbf{signal}\ e\ \textbf{with}\ e'\rceil & = \chi(\lceil e\rceil)!\lceil e'\rceil.\ \text{stop} \\
\lceil p\#e!\ e' \Rightarrow e''\rceil & = p(\lceil e\rceil)!\lceil e'\rceil.\lceil e''\rceil & \lceil e\ \{/p\}\rceil & = \lceil e\rceil\backslash p \\
\lceil p\#e?\ x \Rightarrow e''\rceil & = p(\lceil e\rceil)?\ x.\lceil e''\rceil & \lceil e\ \{q/p\}\rceil & = (\lceil e\rceil\backslash q)\ [q/p]
\end{array}$$

$$\begin{array}{l}
\lceil e\ \textbf{catch}\ e_1\ \textbf{with}\ m_1 \Rightarrow h_1\ m_1 \parallel \ldots \parallel e_n\ \textbf{with}\ m_n \Rightarrow h_n\ m_n\rceil = \\
\qquad (\lambda\, x_1. \ldots . \lambda\, x_n.\ (\lceil e\rceil\ [\zeta(x_1)/\chi(x_1)] \ldots [\zeta(x_n)/\chi(x_n)] \\
\qquad\qquad < (\zeta(x_1)?\ m_1.\lceil h_1\ m_1\rceil + \ldots + \zeta(x_n)?\ m_n.\lceil h_n\ m_n\rceil))\backslash\zeta)\lceil e_1\rceil \ldots \lceil e_n\rceil
\end{array}$$

where x_i assumed not to occur free in h_j or e, and $\chi, \zeta \notin \Sigma$.

$$\begin{array}{rlcrl}
\lceil e \wedge e'\rceil & = \lceil e\rceil \mid \lceil e'\rceil & \textit{light compositions} & \lceil e \vee e'\rceil & = \lceil e\rceil + \lceil e'\rceil \\
\lceil e\ /\!\!\backslash\backslash\ e'\rceil & = \lceil e\rceil \mid \langle\lceil e'\rceil\rangle & \textit{right-strong compositions} & \lceil e\ \backslash\!/\!/\ e'\rceil & = \lceil e\rceil + \langle\lceil e'\rceil\rangle \\
\lceil e\ /\!/\!\backslash\ e'\rceil & = \lceil e'\ /\!\!\backslash\backslash\ e\rceil & \textit{left-strong compositions} & \lceil e\ \backslash\backslash\!/\ e'\rceil & = \lceil e'\ \backslash\!/\!/\ e\rceil \\
\lceil e\ /\!/\!\!\backslash\backslash\ e'\rceil & = \lceil(\text{stop}\ /\!\!\backslash\backslash\ e)\ /\!\!\backslash\backslash\ e'\rceil & \textit{strong compositions} & \lceil e\ \backslash\backslash\!/\!/\ e'\rceil & = \lceil(\text{stop}\ \backslash\!/\!/\ e)\ \backslash\!/\!/\ e'\rceil
\end{array}$$

Table 4.1. Translation into core formalism

The semantics discussed in the next sections is restricted to the subset of terms of the core language which can result from translation of well-typed LCS expressions.

Expressions, Values, Reduction ($\to$)

Evaluation in LCS is “applicative”. Evaluating a behavior in operand position, for instance, produces a value, but we do not want compositions or communications occurring within that behavior to be evaluated at that time. This first evaluation mechanism is called *reduction*.

The *reduction* relation, written $\to$, is the usual “evaluation” relation for functional expressions. The expressions which are irreducible by $\to$ are the *values*. Reduction is defined in Table 4.2 in which e, e’ range over expressions, x, x’ range over identifiers, v, v’, range over values and e{v/x} is the expression obtained by substituting v for the free occurrences of x in e (including within restriction and relabelling expressions).

Expressions are reduced in the context of a store, which may be updated as the result of reductions. For reasons that will shortly appear, we shall use the word *segment* here, in place of store. A segment maps *locations* (addresses) to values. In Table 4.2, s, s’ range over segments, a, a’ range over locations and s+(a,v) is the segment obtained from s by adding the new (location,value) pair (a,v).

$\dfrac{e_1,s \to e_1', s'}{e_1\ e_2,s \to e_1'\ e_2,s'}$	(r1)	$\mathrm{rec}\ x.e,s \to e\ \{\mathrm{rec}\ x.e/x\},s$	(r4)
$\dfrac{e_2,s \to e_2',s'}{v\ e_2,s \to v\ e_2',s'}$	(r2)	$\dfrac{e_1,s \to e_1',s'}{p(e_1)!e_2.e_3,s \to p(e_1')!e_2.e_3,s'}$	(r5)
$(\lambda x.e)\ v,s \to e\ \{v/x\},s$	(r3)	$\dfrac{e_1,s \to e_1',s'}{p(e_1)?x.e_2,s \to p(e_1')?x.e_2,s'}$	(r6)
$\dfrac{a \notin \mathrm{dom}(s)}{\mathrm{ref}\ v,s \to a,\ s+(a,v)}$ (r7)		$\mathrm{deref}\ a,s \to (s\ a),s$ (r8) $\quad a{:=}v,s \to (),\ s+(a,v)$	(r9)

Table 4.2. The reduction relation $\to$

Rules (r1) to (r4) are standard; they define an applicative order evaluation. In (r4), it is assumed that e is either a lambda abstraction or a behavior expression (this is enforced by syntactical constraints in LCS). Evaluation of input (r6) or output (r7) constructs consists of evaluating their communication port extensions. Constants, lambda expressions and other behavior expressions are irreducible by $\to$. Rules (r7) to (r9) concerning the imperative aspects are also standard.

Threads, Suspensions, Expansion ($\Rightarrow$)

Creating a process from a behavior should force the recursive evaluation of its content, including communications and creations of other processes. This second evaluation mechanism will be called *expansion*, since it generally results in creating processes. Expansion builds a structured system of threads from a behavior expression.

The *expansion* relation, written $\Rightarrow$, relates *states* constituted of a thread (also called a *process*, indifferently) and a *store*. Threads irreducible by expansion are called *suspensions*.

A thread is a simple thread, or a thread built from another by a thread restriction or relabelling operator {\\, //}, or a compound thread built by one of the thread composition operators {||,++,<<}.

Simple threads may operate either on private or on shared segments. The structure linking segments together is the *store*. Stores map segment identifiers to segments, each segment mapping locations to values. For some store S and segment identifier g, (S g) is a segment and ((S g) a) is the content of location a in segment (S g).

Simple threads are pairs ⟨e,g⟩, where e is an expression typing as a behavior, and g is a segment identifier. A state is a pair (b,S), where b is a (possibly compound) thread, and S is a store. A *thread context* C[ ] is a possibly compound thread expression in which a simple thread is replaced by a hole; C[t] is the thread obtained by filling the hole of C[ ] with thread t.

The expansion relation is defined in Table 4.3 in which b, b' range over threads, S, S' range over stores, s, s' range over segments, g, g' range over segment identifiers and S+(g,s) is the store obtained from S by binding the segment s to the identifier g.

$$\frac{b,S\ g \to b',s'}{\langle b,g\rangle,S \Rightarrow \langle b',g\rangle,S+(g,s')} \quad \text{(e1)}$$

$$\frac{e_1,S\ g \to e_1',s'}{\langle p\#v!e_1.e_2,g\rangle,S \Rightarrow \langle p\#v!e_1'.e_2,g\rangle,S+(g,s')} \quad \text{(e2)}$$

$$\frac{g' \notin dom(S)}{\langle <e>,g\rangle,S \Rightarrow \langle e,g'\rangle,S+(g',S\ g)} \quad \text{(e3)}$$

$$\langle e\backslash E,g\rangle,S \Rightarrow \langle e,g\rangle \backslash\backslash\ E,S \quad \text{(e4)}$$

$$\langle e\ [R],g\rangle,S \Rightarrow \langle e,g\rangle\ //\ R,S \quad \text{(e5)}$$

$$\langle e_1 \mid e_2,g\rangle,S \Rightarrow \langle e_1,g\rangle \parallel \langle e_2,g\rangle,S \quad \text{(e6)}$$

$$\langle e_1 + e_2,g\rangle,S \Rightarrow \langle e_1,g\rangle ++ \langle e_2,g\rangle,S \quad \text{(e7)}$$

$$\langle e_1 < e_2,g\rangle,S \Rightarrow \langle e_1,g\rangle << \langle e_2,g\rangle,S \quad \text{(e8)}$$

$$\frac{b,S \Rightarrow b',S'}{C[b],S \Rightarrow C[b'],S'} \quad \text{(e9)}$$

Table 4.3. The expansion relation ⇒

Rules (e1) and (e2) link the reduction and expansion relations; expanding an expression requires its reduction. Rule (e2) expresses that messages must be reduced prior to sending them. Rules (e4) to (e8) define the expansion of restriction, relabelling and compositions operators, which build structured threads. The binary composition combinators produce two simple threads from the current one, each inheriting the current segment identifier. Rule (e3) explains the segment copying effects of construction < >. Finally, rule (e9) defines the expansion of compound threads from the expansions of its constituent simple threads.

Communication and pruning (≡μ≡>)

Finally, evaluation of a process involves communications. The *communication* relation, written ≡μ≡>, also relates states. It is defined from a family of relations indexes by *actions*. The treatment below is adapted from the classical "observation" relation for CCS.

An action is either the particular action τ, or a triple (l,Δ,v), where l is a port value, or *line*, consisting of a label in $\Sigma \cup \{\chi,\zeta\}$ and a reference-free value admitting a canonical form (a port index), Δ is a *direction* (! or ?), and v is a *message* constituted of a reference-free value.

The communication relation is defined in Table 4.4 in which S, S' range over stores, b, b' range over threads and w, w' range over suspensions. Arbitrary actions are ranged over by μ, μ'. Given an action μ, $\overline{\mu}$ denotes the action with the opposite direction, but same line and message fields than μ. For conciseness, rules sharing the same premises are grouped into single rules with a multiple denominator.

Rules (c1) to (c3) define the atomic actions. The other rules, except (c15) to (c18) express observation of actions in the different possible contexts, rules (c15) and (c16) express commu-

$$\langle \tau.e,g\rangle,S \;\equiv\tau\equiv>\; \langle e,g\rangle,S \qquad \text{(c1)}$$

$$\langle p(v)!v'.e,g\rangle,S \;\equiv(p(v),!,v')\equiv>\; \langle e,g\rangle,S \qquad \text{(c2)}$$

$$\langle p(v)?x.e,g\rangle,S \;\equiv(p(v),?,v')\equiv>\; \langle e\{v'/x\},g\rangle,S \qquad \text{(c3)}$$

$$\frac{w,S \;\equiv\tau\equiv>\; b,S}{w \backslash\backslash E,S \;\equiv\tau\equiv>\; b \backslash\backslash\; p,S} \qquad \text{(c4)}$$

$$\frac{w,S \;\equiv(l,\Delta,v)\equiv>\; b,S \qquad l \notin E}{w \backslash\backslash E,S \;\equiv(l,\Delta,v)\equiv>\; b \backslash\backslash E,S} \qquad \text{(c5)}$$

$$\frac{w,S \;\equiv\tau\equiv>\; b,S}{w \;//\; R,S \;\equiv\tau\equiv>\; b \;//\; R,S} \qquad \text{(c6)}$$

$$\frac{w,S \;\equiv(l,\Delta,v)\equiv>\; b,S \qquad l \notin dom(R)}{w \;//\; R,S \;\equiv(l,\Delta,v)\equiv>\; b \;//\; R,S} \qquad \text{(c7)}$$

$$\frac{w,S \;\equiv(l,\Delta,v)\equiv>\; b,S \qquad l'=R\,l}{w \;//\; R,S \;\equiv(l',\Delta,v)\equiv>\; b \;//\; R,S} \qquad \text{(c8)}$$

$$\frac{w_1,S \;\equiv\mu\equiv>\; b_1,S}{\begin{array}{ll} w_1 \mathbin{++} w_2,S \;\equiv\mu\equiv>\; b_1,S & \text{(c9)} \\ w_2 \mathbin{++} w_1,S \;\equiv\mu\equiv>\; b_1,S & \text{(c10)} \\ w_1 \parallel w_2,S \;\equiv\mu\equiv>\; b_1 \parallel w_2,S & \text{(c11)} \\ w_2 \parallel w_1,S \;\equiv\mu\equiv>\; w_2 \parallel b_1,S & \text{(c12)} \\ w_1 \ll w_2,S \;\equiv\mu\equiv>\; b_1 \ll w_2,S & \text{(c13)} \\ w_2 \ll w_1,S \;\equiv\mu\equiv>\; b_1,S & \text{(c14)} \end{array}}$$

$$\frac{w_1,S \equiv\mu\equiv> b_1,S \qquad w_2,S \equiv\bar{\mu}\equiv> b_2,S}{\begin{array}{ll} w_1 \parallel w_2,S \;\equiv\tau\equiv>\; b_1 \parallel b_2,S & \text{(c15)} \\ w_1 \ll w_2,S \;\equiv\tau\equiv>\; b_2,S & \text{(c16)} \end{array}}$$

$$\frac{b,S \Rightarrow w,S' \qquad w,S' \equiv\mu\equiv> b,S''}{b,S \equiv\mu\equiv> b,S''} \qquad \text{(c17)}$$

$$\frac{w,S \equiv\mu\equiv> b,S' \qquad b,S' \Rightarrow b',S''}{w,S \equiv\mu\equiv> b',S''} \qquad \text{(c18)}$$

Table 4.4. The communication relation $\equiv\mu\equiv>$

nications between processes composed by || and <<, respectively. The communication rule for || is exactly that of the | combinator of CCS. Except for the interaction rule (c16), the rules for interruption << resemble those of the "disable" combinator of the language LOTOS [9]. For behaviors resulting from translation of LCS expressions, a communication involving the component on the right side of combinator << necessarily involves the component on its left side. The translation of signal and catch constructions imply that events are trapped by their innermost handler, if any is provided. The last two rules relate expansion with communication.

Operationally, a communication has two simultaneous effects. The first is that of passing a value from one thread to another (rules (c15) and (c16)). The other effect is that of pruning the current thread by removing some of its components (rules (c9), (c10), (c14) and (c16)).

These rules extend the CCS and CHOCS calculus. Compared to CCS, and besides stores and segments, the values in actions may include behavior values, ports have a richer structure (label plus extension), and the interrupt rules are added.

Reference passing

Neither port extensions, nor messages, may encapsulate reference values. The reasons for these restrictions are briefly explained here.

First, it seems natural that the locations created in some segment are not defined in the other segments. Allowing to pass locations would thus require to interpret dereferencing of a location not belonging to the segment of the process.

Next, an important question is whether a copy of a location (obtained by segment copying) should be considered an equal or a different location than the original. Considering them different (while allowing reference passing) would be semantically satisfying, but would lead to severe implementation problems since it would require to duplicate the whole environment of a thread upon a strong composition. On the other side, considering them the same would only require to copy the segment of the thread, what can be efficiently implemented by a simple copy-on-write technique, but location passing would then have a few counter-intuitive effects. In particular, the content of some location could change when passed from a process to another, and ports extensions or events could be considered equal though constituted of references with different contents (when compared for equality for determining communication capabilities).

The solution retained makes LCS a conservative extension of Standard ML with respect to the treatment of references (assuming there is a single thread running, or that only strong compositions are used), it also permit to avoid the effects mentioned and allow an efficient implementation. Finally, note that location passing would not cause any problem if all programs shared the same segment, but this was found too severe a restriction.

Semantics

Having defined an observation relation in the style of what has been done for CCS, we would like to take as semantics of LCS programs the observation congruence discussed in [13], or a congruence included in it such as the strong equivalence. Artesiano and Zucca [1] have shown that adding indexed communication ports to CCS could be easily accommodated by its observational semantics. Thomsen [19] observed that the notion of observation congruence could be easily extended to handle behavior passing. Furthermore, the adjunction of the interrupt combinator would not bring any specific difficulty. So, not taking assignments into account, a suitable concept of observation congruence can be formalized for the core calculus of LCS.

However, this concept of observation congruence would not be adequate for arbitrary LCS programs. In general, the communication actions themselves and/or the way they are organized may depend on the (non observable) expansions preceding the communications. Consider for instance the following behavior a, parameterized by some integer k>0:

```
fun a k =
   let val r = ref 0
   in (while deref r < k do r :=  deref r + 1; stop) /\
      let val x = deref r in p#x! => stop end
   end;
```

The behavior (a K), for some integer K, may offer an output on any line p#i, for i in the range 0…K-1. The port on which a value is offered only depends on the number of loops performed while reducing the do-while expression in the left side of the parallel composition before the computation of `x` on the right side has been completed. Observation congruence should not identify this program with the following:

```
fun b k = p#0! => stop  \/  ...  \/  p#(k-1)! => stop;
```

Which is clearly distinct from the former since only one possible communication offer is provided by (`a K`), while all alternatives are simultaneously offered by behavior (`b K`).

So, not surprisingly, side-effects and shared segments together bring a kind of nondeterminism which is not properly captured by the observation congruence concept. This nondeterminism was observed earlier by researchers concerned with verification of parallel programs; a frequent assumption required in their proof techniques was that programs are “interference-free” (see for instance [16]). However, observation congruence is a suitable semantics for the class of interference-free LCS programs. How to determine the property for a given program is a complex task in general, but, fortunately, there is an large class of structurally interference-free programs: those only using strong compositions. It can be shown that the programs obeying this restriction cannot exhibit the particular nondeterminism discussed above.

5. An overview of implementations

Abstract machines

The abstract machine associates a virtual processor with every active thread (corresponding to the non-suspension simple threads discussed in Section 4). These processors reduce and expand threads asynchronously and cooperate for communication.

The state (registers) of each thread include registers for reduction, a communication environment linking communication ports to “communication lines”, a “position” register for implementing process preemption and a store segment identifier. The global state information is constituted of all active threads, the content of the communication lines (the lines may be shared by several threads), a shared information called the accumulator and the content of all segments of the store, all soon to be described.

Reduction of threads requires exactly the same apparatus than reduction of any ML program, so we shall not detail these aspects longer. The functional sub-machine of the LCS abstract machine implements a customized and optimized SECD machine, it may be considered a cousin of Cardelli’s Functional Abstract Machine. We shall also omit to discuss the management of the store.

Handling process preemption

As noted in Section 4, some processes may be aborted upon actions performed by other processes. We shall say that those actions having pruning effects are *preemptive*, and that a process is *persistent* versus another it is resists a preemptive action of the latter. We shall also say that a thread is *obsolete* if it is not persistent with one of the threads that made a preemptive action since the beginning of evaluation. Considering that behaviors composed by choice or catch are here arbitrary behaviors, it would be costly in practice to take the pruning effects of preemptive actions into account as soon as the action is performed; LCS implementations rely on an original solution for implementing process preemption.

LCS abstract machines dynamically encode within the threads an information (the *position* information) from which the relative persistence of two threads can be efficiently determined. The consistency of the system of threads is enforced using a global register called the *position accumulator*. The accumulator holds an information dynamically computed from the position informations of all threads that made a preemptive action and permits to determine efficiently if some thread is obsolete or not. Every application of a reduction, expansion or communication rule to a thread should be preceded by a non-obsolescence test for that thread (in practice, only very few such tests are needed). Obsolete threads are removed when found; in addition, a separate mechanism incrementally collects and removes all obsolete threads from the system.

Local positions are determined from the inherited position and the composition operators. Seeing compound threads in Section 4 as trees, the position of a thread may be thought of as the path in this tree leading from the root to that thread. The global position accumulator, updated every time a process makes a preemptive action, may be thought of as the union of all paths to the threads that performed preemptive actions. We shall not discuss here concrete encoding for the position and accumulator registers.

Finally, note that two LCS threads may communicate if and only if one of them at least is persistent versus the other. So the information encoding persistence can also be used to determine if two particular threads may or not communicate.

The communication environment

Each thread maintains a structure that associates each possible communication port (constituted of a label and an extension) with a communication line. A line is a pair of possibly empty sets of suspensions: those offering input on that line, and those offering output on that line. Note that, due to the unrestricted choice composition, we may have both sets simultaneously nonempty while none of the threads in the input set may communicate with some in the output set. The lines are shared by all threads, while the communication environments are local.

Upon a restriction, a thread updates its communication environment so that ports with the restricted labels are all associated with "new" lines with empty suspension sets. A relabelling is handled by associating to the ports bearing the label being renamed the lines currently associated with the ports bearing the label in which it is renamed. Upon expansion of a composition operator, each thread created inherits the current communication environment.

Upon an input (resp. output) offer, a thread searches a partner for communication in the output (resp. input) suspension set of the line associated with the port on which the offer is made. A partner is a non-obsolete thread which may communicate with it, this is determined from the positions of the two threads and the accumulator. If there is a possible partner, then the message is transmitted, the accumulator is updated and both threads are resumed. Otherwise, the thread having made the offer is suspended in the line (the relevant suspension set is physically updated, so that the new offer will be seen by all threads referencing that line).

Bounded parallelism machines, scheduling

Real machines obviously cannot provide an arbitrary number of processors for running all active threads. In practice, the set of active threads has to be partitioned and the threads in each subset in the partition share a single processor. Two implementations have been investigated: a sequentially running version [4], and a multi-processor version still in development [10] [11].

In the sequential case, where all threads share a single processor, a queue R of ready processes is added to the global registers, holding the active threads currently waiting for the processor. The suspension sets in the lines are handled as queues, too. Upon an unsatisfied communication offer, the next process in queue R is resumed; upon a communication, the receiver is resumed and the sender is suspended at the end of R. In addition to this "communication-bound" scheduling, a time-slicing mechanism prevents diverging or looping threads to take the processor forever. Since there is a single processor, the non-obsolescence test is only required when a thread is resumed.

In parallel implementations [11], the queue of ready processes is dynamically partitioned among the available processors with the help of a load-balancing algorithm. The process migration strategy also prevents the processes subject to preemption (i.e. those nonpersistent with some other process) to be moved from a processor to another so that all processors may work asynchronously, each with a private accumulator register.

6. Conclusion

Related work

LCS benefited from the results of the PFL experience, an early attempt to add CCS capabilities to ML [8]. PFL required a "continuation-passing" style of programming together with channels passed as parameters to the behaviors. LCS managed to keep the original CCS combinator set and concept of port, and more closely integrates behaviors with other features of the language.

CML [17] and the language described in [5] are based on different grounds than LCS; they provide a strictly functional interface for concurrency and communication. One of the difficulties with that approach is delaying communications while keeping an applicative order evaluation. In both these languages, communications are controlled by applying a synchronization function to the suspended communications. LCS uses a layered evaluation method for achieving transparently the same effects.

As LCS, the language FACILE [7] uses behavior expressions to describe processes, though restricted there to the termination behavior and compositions. However, the FACILE treatment of concurrency is closer to that of languages offering a functional interface.

Finally, the connections with the LOTOS effort by Brinksma et al. at the University of Twente [9] aimed at providing rigorous specification techniques for Open System Interconnection should be noted. Though both languages are based upon CCS for their behavioral aspects, LCS and LOTOS differ in nearly all other areas. LOTOS has a more specification-oriented flavor, due in part to the algebraic treatment of its non-behavioral features; LCS certainly has a more computational flavor.

Conclusion and further work

On one hand, LCS is a rigorously designed language including all capabilities of CCS and inheriting most of its theory. Parallel applications can be written abstractly enough so that users may reason about their programs. On another hand, the language has all the ingredients and facilities of a modern general purpose programming language. The language, including its behavioral primitives, was designed with efficiency of implementations as an important concern. The fact that abstract enough specifications and efficient enough implementations can be written in a single framework is certainly an advantage for formal development of programs.

LCS allows one to comfortably experiment with the parallel programming concepts introduced by CCS and related behavioral formalisms, and to gain experience in using these paradigms. It is also a valuable teaching tool for these concepts. Our experience with users of the system over the last years taught us that newcomers to parallel programming adapt quickly to the set of high level programming primitives provided by its behavioral interface; the behavioral approach helps intuition and, yet, is powerful enough to write real size applications.

Among the spin-offs of the experiment is an original method for typing behaviors that, in the more general setting of [3] can be used for typing a variety of other features of programming languages. Though it could not be described in depth here, the implementation of LCS also has a number of unique features such as both strong and light processes combined with automatic memory allocation, handling of unguarded choice compositions, etc. We hope to present the details of the implementations in forthcoming documents.

It is hoped that practising with the concepts implemented will prompt desirable refinements of the language, such as, for instance, introduction of some synchrony or stronger forms of label passing. Our short term goals include parallel implementations of the language; an architecture and some important building blocks have already been defined [10] [11].

Acknowledgments

Didier Giralt† and Jean-Paul Gouyon worked intensively on the project in its early stages. We are indebted to all those who helped implement the language or improve its design by their criticisms, including all users. We are especially grateful to Chris Reade for his comments and suggestions, and to Gérard Berry and colleagues at Sophia-Antipolis for a number of stimulating discussions. This work was partially supported by CNRS Greco C-Cube.

References

1. E. Artesiano, E. Zucca, “Parametric channels via label expressions in CCS*”, Theoretical Computer Science, vol 33, 1984.
2. B. Berthomieu, “LCS, une implantation de CCS”, 3ème Colloque C-Cube, A. Arnold Ed., Angoulème, France, September 1988.
3. B. Berthomieu, “Tagged Types – A theory of order sorted types for tagged expressions”, LAAS Technical Report 93083, March 1993.
4. B. Berthomieu, D. Giralt, J-P. Gouyon, “LCS user’s manuals”, LAAS Technical Report 91226, June 1991.
5. D. Berry, R. Milner, D. N. Turner, “A semantics for ML concurrency primitives”, ACM Symposium on Principles of Programming Languages, 1992.
6. U. Engberg, M. Nielsen, “A Calculus of Communicating Systems with Label Passing”, Research Report DAIMI PB-208, Computer Science Department, U. of Aarhus, 1986.
7. A. Giacalone, P. Mishra, S. Prasad, “Facile: A symmetric integration of concurrent and functional programming”. Int. Journal of Parallel Programming, 18(2), April 1989.
8. S. Holstrom, “PFL: A Functional Language for Parallel Programming”. In *Declarative Programming Workshop*, Chalmers U. of Technology, U. of Goteborg, Sweden, 1983.
9. ISO, “ISO-LOTOS, A Formal Description Technique based on the Temporal Ordering of Observational Behaviour”, Int. Standard ISO 8807, ISO, 1989.
10. T. Le Sergent, B. Berthomieu, “Incremental multi-threaded garbage collection on virtually shared memory architectures”, Int. Workshop on Memory Management, St. Malo, France, Sept. 1992.
11. T. Le Sergent, “Méthodes d’exécution et machines virtuelles parallèles pour l’implantation distribuée du langage de programmation parallèle LCS”, Ph.D. thesis, Feb. 93.
12. D. Matthews, “A distributed concurrent implementation of Standard ML”, EurOpen Autumn 1991 Conference, Budapest, Hungary, 1991.
13. R. Milner, C*ommunication and Concurrency*, Prentice Hall international series in Computer science, C.A.R. Hoare Ed, 1989.
14. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990
15. R. Milner, J. Parrow, D. Walker, “A Calculus of Mobile Processes”, ECS-LFCS-89-85, LFCS report series, Edinburgh University, 1989.
16. S. Owicki, D. Gries, “An Axiomatic Proof Technique for Parallel Programs I”, Acta Informatica, vol 14, 1976.
17. J. H. Reppy, “CML: A higher-order concurrent language”, ACM SIGPLAN Conf. on Programming Language Design and Implementation, SIGPLAN Notices 26(6), 1991.
18. D. Rémy, “Typechecking records and variants in a natural extension of ml”, In Proceedings of the 16th ACM Symp. on Principles of Programming Languages, 1989.
19. B.Thomsen, “A calculus of higher order communicating systems”, In Proceedings of the 16th ACM Symposium on Principles of Programming Languages, 1989.
20. M. Wand. “Complete Type Inference for Simple Objects”, In Second Symposium on Logic in Computer Science, Ithaca, New York, june 1987.