URL: http://www.elsevier.nl/locate/entcs/volume2.html 8 pages

Linear Types for Higher Order Processes with First Class Directed Channels

Georg Schied

Institut für Informatik
Universität Stuttgart
D-70565 Stuttgart, Germany

Klaus Barthelmann

Institut für Informatik
Universität Mainz
D-55099 Mainz, Germany

Abstract

We present a small programming language for distributed systems based on message passing processes. In contrast to similar languages, channels are one-to-one connections between a unique sender and a unique receiver process. Process definitions and channels are first class values and the topology of process systems can change dynamically. The operational semantics of the language is defined by means of graph rewriting rules. A static type system based on the notion of linear types ensures that channels are always used as one-to-one connections.

Keywords: distributed programming, process algebras, linear types, operational semantics, graph rewriting

1 Introduction

Since the beginning of the eighties, process algebras have been successfully used for specifying and verifying concurrent systems. In the past years, there have been several attempts to integrate the concepts of process algebras into programming languages, mostly extending functional languages, e.g. Facile [3,11], CML [8] or LCS [2]. These languages inherit from process algebras the notion of parallel processes communicating via channel names. Channels are undirected communication links. If a process knows a channel name, it can interact with any other process that knows the same name (maybe restricted by some scope constructs).

Implementing this kind of communication efficiently on a distributed system is rather difficult. Therefore it is advantageous to use channels as *directed one-to-one* communication links between exactly one sender and one receiver

process. However, we would like to change dynamically the process topology, i.e. passing channels from one process to another. How can we ensure then that a channel is not used by several processes? Using runtime checks is unsatisfactory. Instead, we propose a type system inspired by the notion of *linear types*[12] that restricts the use of channels in an appropriate way. We incorporate this idea into a small language called DHOP (Distributed Higher Order Processes) with the following features: Channels are one-to-one connections between process instances. Channel ends and process definitions are first class values (similar to the π-calculus [7] and CHOCS [10], respectively), hence they can be sent to other processes, be used as arguments for process instantiation, or be used as components of data structures. This allows powerful programming techniques similar to the use of higher order functions in functional languages. DHOP is strongly typed, using some kind of linear types for channel values. The operational semantics of DHOP is formally defined. Usual methods like the structural operational semantics (SOS) technique or denotational semantics are not well suited to express structural conditions concerning the topology of process nets. Therefore we use graphs and graph rewriting systems, that allow to model the connection relation between channels and processes explicitly. An operational semantics of a precursor of DHOP has already been presented in [1] but typing aspects have not been addressed there at all.

Section 2 contains an introduction to DHOP. Section 3 presents a type system for DHOP. In the following section we sketch the operational semantics of DHOP and explain some relations between the typing system and the operational semantics.

2 Distributed Higher Order Processes

DHOP is (the core of) a programming language based on processes communicating by synchronous message passing via channels. Channels are *directed* communication links between exactly two processes. This stands in contrast to most process calculi (CCS [6], CSP [5], etc.) and related programming languages (e.g. CML [8], Facile [3]), where channels (sometimes called *ports*) are just names and every process that knows a channel name can potentially communicate with every other process that knows the same name.

The syntax of DHOP is depicted in Fig. 1. Statements S describe the behaviour of processes. `STOP` denotes a process that immediately terminates. $C;S$ means that the process can execute a communication action C and then continue according to the statement S. Communication actions are send actions $c\texttt{!}e$ and receive actions $c\texttt{?}(x_1, \ldots, x_n)$. The statement `SELECT` C_1 `=>` S_1 $[]\ldots[]C_n$ `=>` S_n `END` corresponds to the external choice operator of process algebras. Depending on which communication C_i can be performed first, the process continues as specified by the statement S_i. $C;S$ is just an abbreviation for a select statement `SELECT` C `=>` S `END` with only one alternative. Calls for process definitions are written `CALL` e_1 e_2. Expression e_1 denotes

```
S  ::=  STOP   |   C ; S
      | SELECT C1 => S1 [] ... [] Cn => Sn END
      | CALL e1 e2   |   [ S1 PAR S2 ]
      | CHANNEL x1? x2!:t; S   |   IF e THEN S1 ELSE S2
      | LET p = e; S   |   REC x1 = e1, ..., xn = en; S
C  ::=  x!e   |   x?p
e  ::=  True   |   False   |   n   |   x
      | e1 op e2   |   (e1, ..., en)   |   PROCESS p:t.S
p  ::=  x   |   (p1, ..., pn)
t  ::=  Bool   |   Int   |   (t1, ..., tn)
      | PROC t   |   ?t   |   !t
```

Fig. 1. Syntax of DHOP

a process definition of type `PROC` t and e_2 denotes the argument of type t. The process executing the call statement continues according to the process definition.

Channels and processes can be generated dynamically. `CHANNEL` x_i`?` x_o`!:` t`;` S creates a new channel transmitting values of type t. The identifiers x_i and x_o are different names for the receiving and the sending end of the channel, respectively. This allows to model directed communication links. The sending end x_o has type `!`t and the receiving end x_i has type `?`t. Channel ends are first class values. Hence, the topology of a system can be changed dynamically passing channel ends between processes. The statement `[` S_1 `PAR` S_2 `]` splits a process into two processes executing S_1 and S_2 in parallel.

Local declarations `LET` p `=` e`;` S are used for two purposes. First, they introduce local identifiers for values. Second, they provide pattern matching as a means to extract the components of tuples. A pattern p is either a single identifier or tuple of patterns. `REC` x_1 `=` e_1`, ...,` x_n `=` e_n`;` S enables to define recursive processes. Here, all expressions e_i must be process definitions. Process definitions `PROCESS` x`:`t`.`S denote values of type `PROC` t. They are first class values, i.e. they can be sent to other processes or be parameters for calls of process definitions. Last, there is a conditional statement `IF` e `THEN` S_1 `ELSE` S_2.

3 Static Semantics of DHOP

In DHOP any channel is connected with a unique sender and a unique receiver process. On the one hand, we need channel ends as first class values, e.g. for using input/output devices from several processes. On the other hand, we cannot allow unrestricted use of channel ends as values. In the following example we suppose `out` to be a predefined channel of type `!Int`.

```
LET p1 = PROCESS (c):(!Int). c ! 42; STOP;
[ CALL p1(out) PAR CALL p1(out) ]
```

Here, two instances of the process definition `p1` are generated and the output channel `out` would be passed to both instances. Therefore this program has to be rejected by the compiler.

We use a type system inspired by the notion of *linear types* [12] to fulfill both requirements. Values of linear types can be used only once in contrast to values of conventional nonlinear types. Channel ends are basic linear values. If a tuple contains a component of linear type, then the tuple itself must become linear. Otherwise the following program would erroneously pass the type check (we suppose `out` to be a predefined output channel of type `!Int`).

```
LET p2  = PROCESS (c, n):(!Int, Int) ...;
LET pair = (out, 13);
[ CALL p2(pair)  PAR  CALL p2(pair) ]
```

Process definitions with free linear identifiers might lead to similar difficulties.

```
LET p3 = PROCESS (n):(Int). out ! n; STOP;
[ CALL p3(13) PAR CALL p3(42)]
```

As `out` occurs free in the definition of `p3`, both instances of `p3` would try to use this output channel at the same time. In order to keep things simple, any process definition must not contain free linear identifiers. Then all process definitions `PROCESS` $p{:}t\,.\,S$ can be treated as nonlinear values.

The typing rules for DHOP are shown in Fig. 2. An assertion $E \vdash e : t$ means: given type environment E, expression e has type t. Similarily, $E \vdash S$ states that statement S is well typed, given type environment E. A type environment is a multiset (!) of pairs. $E_1 + E_2$ denotes the (disjoint) union of multisets. $E[x{:}t]$ means first deleting all pairs with first component x from E and then adding the pair $x{:}t$. $\mathit{linear}(t)$ is the coarsest predicate over types respecting the following conditions:

(i) $\mathit{linear}(?t)$,
(ii) $\mathit{linear}(!\,t)$,
(iii) $\mathit{linear}(t_1) \vee \ldots \vee \mathit{linear}(t_n) \Rightarrow \mathit{linear}((t_1,\ \ldots,\ t_n))$.

We define $\mathit{nonlinear}(t) \Leftrightarrow \neg\mathit{linear}(t)$ for types t and $\mathit{nonlinear}(E) \Leftrightarrow \forall x{:}t \in E.\ \mathit{nonlinear}(t)$ for environments E.

An occurrence of a pair $x{:}t$ in an environment means that identifier x has type t and it constitutes one exclusive access right for x. Every occurrence of x in an expression consumes one access right (see rule (3)). Rule (17) shows that rights for nonlinear values can be duplicated and hence identifiers with nonlinear types can be used arbitrarily often. Please note that using a channel for communication does not consume its "access right"(see (8), (9)).

4 Operational Semantics of DHOP

The operational semantics of DHOP is defined by means of graph rewriting rules. The state of a system is represented as a configuration graph and the

$$(1)\ \{\} \vdash b : \texttt{Bool} \qquad (2)\ \{\} \vdash n : \texttt{Int} \qquad (3)\ \{x : t\} \vdash x : t$$

$$(4)\ \frac{E_1 \vdash e_1 : t_1 \qquad E_2 \vdash e_2 : t_2}{E_1 + E_2 \vdash e_1\, op\ e_2 : t} \quad \mathit{if}\ op : (t_1, t_2) \to t$$

$$(5)\ \frac{E_1 \vdash e_1 : t_1 \qquad \ldots \qquad E_n \vdash e_n : t_n}{E_1 + \ldots + E_n \vdash (e_1,\ \ldots,\ e_n) : (t_1,\ \ldots,\ t_n)}$$

$$(6)\ \frac{E[x_1 : t_1, \ldots, x_n : t_n] \vdash S \qquad \mathit{nonlinear}(E)}{E \vdash \texttt{PROCESS}\ (x_1, \ldots, x_n){:}(t_1, \ldots, t_n).S : \texttt{PROC}\ (t_1, \ldots, t_n)}$$

$$(7)\ \{\} \vdash \texttt{STOP} \qquad (8)\ \frac{E \vdash e : t \qquad E' + \{x{:}!\,t\} \vdash S}{E + E' + \{x{:}!\,t\} \vdash x\,!\,e\,;S}$$

$$(9)\ \frac{(E + \{x{:}?(t_1, \ldots, t_n)\})[x_1{:}t_1, \ldots, x_n{:}t_n] \vdash S}{E + \{x{:}?(t_1, \ldots, t_n)\} \vdash x?(x_1, \ldots, x_n);S}$$

$$(10)\ \frac{E \vdash C_1\,;S_1 \qquad \ldots \qquad E \vdash C_n\,;S_n}{E \vdash \texttt{SELECT}\ C_1\ \texttt{=>}\ S_1 [] \ldots [] C_n\ \texttt{=>}\ S_n\ \texttt{END}}$$

$$(11)\ \frac{E_1 \vdash S_1 \qquad E_2 \vdash S_2}{E_1 + E_2 \vdash \texttt{[}\ S_1\ \texttt{PAR}\ S_2\ \texttt{]}} \qquad (12)\ \frac{E_1 \vdash e_1 : \texttt{PROC}\ t \qquad E_2 \vdash e_2 : t}{E_1 + E_2 \vdash \texttt{CALL}\ e_1\ e_2}$$

$$(13)\ \frac{E + \{x_1{:}?t, x_2{:}!\,t\} \vdash S}{E \vdash \texttt{CHANNEL}\ x_1\texttt{?}\ x_2\texttt{!} : t; S} \qquad (14)\ \frac{E \vdash e : \texttt{Bool} \quad E' \vdash S_1 \quad E' \vdash S_2}{E + E' \vdash \texttt{IF}\ e\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2}$$

$$(15)\ \frac{E \vdash e : (t_1, \ldots, t_n) \qquad E'[x_1{:}t_1, \ldots, x_n{:}t_n] \vdash S}{E + E' \vdash \texttt{LET}\ (x_1,\ \ldots,\ x_n)\ \texttt{=}\ e\,;S}$$

$$(16)\ \frac{E' \vdash e_1 : t_1 \qquad \ldots \qquad E' \vdash e_n : t_n \qquad E' \vdash S}{E \vdash \texttt{REC}\ x_1 = e_1, \ldots, x_n = e_n\,;S}$$

where t_i are process types $\texttt{PROC}\ t'_i$ and
e_i are process definitions (for $i = 1, \ldots, n$)
$E' = E[x_1{:}t_1, \ldots, x_n{:}t_n]$

$$(17)\ \frac{E + \{x{:}t, x{:}t\} \vdash S \qquad \mathit{nonlinear}(t)}{E + \{x{:}t\} \vdash S} \qquad (18)\ \frac{E \vdash S}{E + \{x{:}t\} \vdash S}$$

Fig. 2. Typing rules for DHOP

dynamic evolution is modelled with graph rewriting rules. The graph rewriting semantics is similar to that given in [1]. The only significant difference is that we include typing information into the labels of the configuration graphs. Here we omit all the technical details and rely on the readers intuition. A configuration graph contains process nodes (ovals), that represent process instances and channel nodes (squares), that represent communication channels. The edges of the configuration graph describe the connection relation between processes and channels. A channel node is labelled with a channel identifier

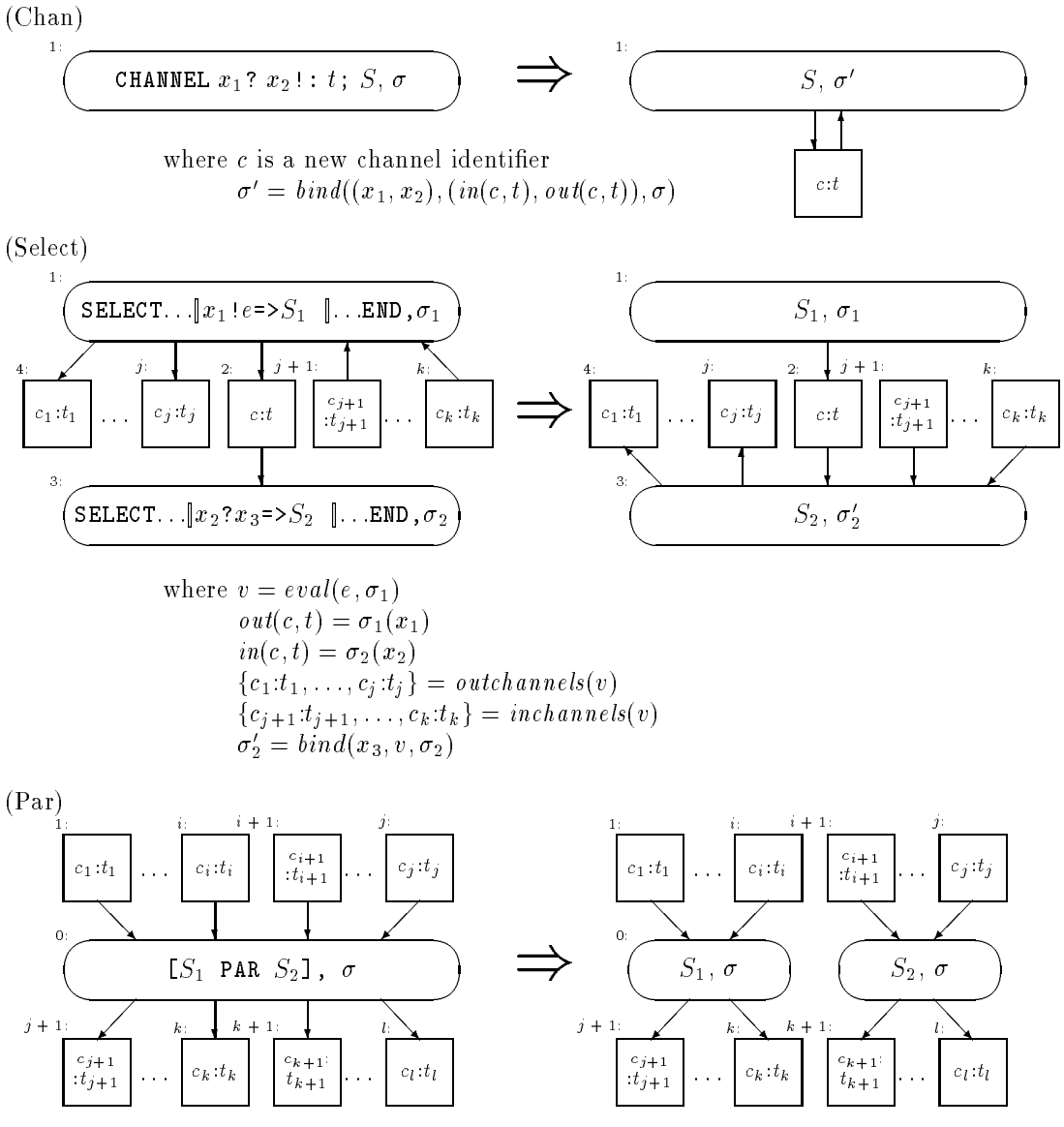


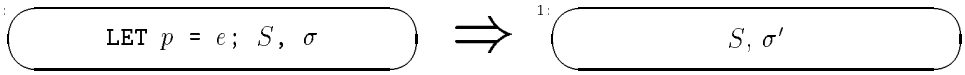
Fig. 3. Operational semantics of DHOP – part I

c and a type t. Process nodes are labelled with a statement S that describes the future behaviour of the process and a store $\sigma : Id \to Value$ that assigns values to identifiers. The start graph for the execution of a DHOP program S consists of one process node that is labelled with the statement S and the empty value environment. Fig. 3 and Fig. 4 depict the graph rewriting rules describing the execution of DHOP programs.

We can infer from the rewriting rules that there is a unique sender and a unique receiver process assigned to every channel:

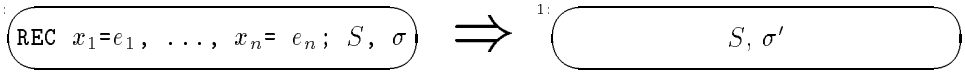
Proposition 4.1 (Structural correctness) *All rewriting rules of the operational semantics preserve the structural condition that any channel node has*

(Let)



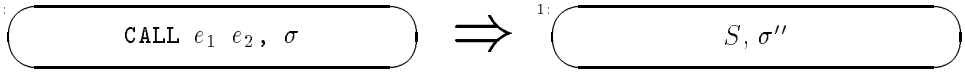
where $v = eval(e, \sigma)$ and $\sigma' = bind(p, v, \sigma)$

(Rec)



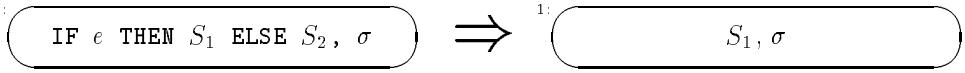
where $E' = recbind((x_1, \ldots, x_n), (e_1, \ldots, e_n), \sigma)$

(Call)



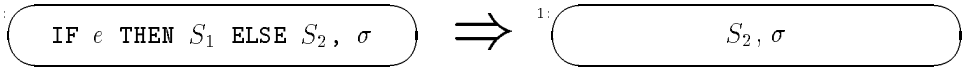
where $\langle \texttt{PROCESS}\ p : t.S, \sigma' \rangle = eval(e_1, \sigma)$
$v = eval(e_2, \sigma)$
$\sigma'' = bind(p, v, \sigma')$

(If1)



if $\texttt{True} = eval(e, \sigma)$

(If2)



if $\texttt{False} = eval(e, \sigma)$

Fig. 4. Operational semantics of DHOP – part II

exactly one ingoing and one outgoing edge from/to a process node.

Proposition 4.2 (Absence of dynamic type errors) *Starting with a statement S such that $\{\} \vdash S$ can be derived, no dynamic type errors can occur during execution of S.*

We subsume the following situations under dynamic type errors : (1) A process tries to perform a communication action, but the corresponding channel node is not correctly connected to the process node. (2) Sender and/or receiver process assume a type of a channel (stored in the store component of a process state) that does not agree with the type of the corresponding channel node.

5 Conclusion

Linear types and related concepts like uniqueness types [9] or the single-threaded lambda calculus [4] have been proposed to support referentially transparent I/O, efficient array handling, and mutable data structures in functional programming languages. In this paper we showed another application of linear types in the context of communicating processes. We considered only channels as basic linear values. If desired, other linear types or constructors could be introduced as well, i.e. arrays as linear values in order to allow ef-

ficient update operations [12]. Polymorphism and type inference for higher order processes with linear types will be considered in a forthcoming paper.

References

[1] K. Barthelmann and G. Schied. Graph grammar semantics of a higher-order programming language for distributed systems. *Graph Transformations in Computer Science*, *LNCS* 776, pages 71–85. Springer, 1994.

[2] B. Berthomieu and T. Le Sergent. Programming with behaviors in an ML framework – the syntax and semantics of LCS. *ESOP'94*, *LNCS* 788, pages 89–104. Springer, 1994.

[3] A. Giacalone, P. Mishra, and S. Prasad. Facile: a symmetric integration of concurrent and functional programming. *Int. Journal of Parallel Programming*, 18(2):121–160, 1989.

[4] Juan C. Guzman and Paul Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symp. Logic in Computer Science*, pages 333–343, 1990.

[5] C.A.R. Hoare. *Communicating sequential processes.* Prentice Hall International, 1985.

[6] R. Milner. *A calculus of communicating systems*, *Lecture Notes in Computer Science* 92. Springer-Verlag, Berlin, 1980.

[7] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation* 100:1–77, 1992.

[8] John H. Reppy. CML: A higher-order concurrent language. In *SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[9] S. Smetsers, E. Barendsen, M. Eekelen, and R. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. *Graph transformations in computer science*, *LNCS* 766, pages 358–379. Springer, 1994.

[10] B. Thomsen. Plain CHOCS – a second generation calculus for higher order processes. *Acta Informatica*, 30:1–59, 1993.

[11] B. Thomson, L. Leth, S. Prasad, T.-M. Kuo, A. Kamer, F. Knabe, and A. Giacalone. Facile antigua release programming guide. Technical Report ECRC-93-20, European Computer-Industry Research Center (ECRC), Munich, 1993.

[12] P. Wadler. Linear types can change the world! In *Proc. of working conference on programming concepts and methods*, pages 385–407. North Holland, 1990.