Classical Higher-order Processes

Fabrizio Montesi

University of Southern Denmark

Abstract. Classical Processes (CP) is a calculus where the proof theory of classical linear logic types processes à la π-calculus, building on a Curry-Howard correspondence between session types and linear propositions. We contribute to this research line by extending CP with process mobility, inspired by the Higher-Order π-calculus. The key to our calculus is that sequents are asymmetric: one side types sessions as in CP and the other types process variables, which can be instantiated with process values. The controlled interaction between the two sides ensures that process variables can be used at will, but always respecting the linear usage of sessions expected by the environment.

1 Introduction

Session types define protocols that discipline how concurrent processes may interact [10]. The type theory of sessions for (a variant of) the π-calculus [11] was found to be in a Curry-Howard correspondence with intuitionistic linear logic, where processes correspond to proofs, session types to propositions, and communication to cut elimination [4]. Important properties that are normally obtained through additional technical machinery on top of session types, like deadlock-freedom, come for free from the properties of linear logic, like cut elimination. Later, the correspondence was revisited for classical linear logic, yielding the calculus of Classical Processes (CP) [19]. The design of CP is guided by the logic, making the correspondence stricter at the cost of deviating some more from the standard π-calculus.

The solidity of the correspondence between session types and linear logic propositions has been confirmed repeatedly. From the initial seminal idea, different extensions have been proposed in order to capture, among others, multiparty sessions [8,6], the paradigm of choreographic programming [12,7], behavioural polymorphism [3,19,2,6], and integrations with functional programming [17].

In this paper, we begin extending this research line towards a key generalisation of the π-calculus: the Higher-Order π-calculus (HOπ) [15]. HOπ supports process mobility: communicated values can be processes, which can then be run or re-transmitted by the receiver – by using *process variables* to refer to the received processes in its program. Our main contribution is the development of CHOP (Classical Higher-Order Processes), which extends CP to process mobility in the same fashion as in HOπ. In CP, typing judgements are of the form $P \vdash \Delta$, read "process P uses its sessions according to Δ". If we ignore the P, this is the standard one-sided sequent form of classical linear logic. The key aspect of CHOP

is that it extends the one-sided sequents used in CP to two-sided sequents, which are manipulated by combining the typing of linear channels of CP (on the right) with a new discipline that types process variables (on the left). So our typing judgements are now of the form $\Theta \vdash P :: \Delta$, read "process P uses its session endpoints according to Δ, possibly using some process variables according to Θ". Why the "possibly" for the usage of process variables? In HOπ, a process variable can be used by the receiving process at will (zero or more times). This expressivity is carried over to CHOP by interpreting process variables as non-linear resources. As a result, we get a hybrid type system that consists of two fragments. The first is inherited directly from CP, used to manipulate linear resources (session communications), while the second disciplines the usage of process variables.

2 Classical Higher-Order Processes (CHOP)

We introduce the calculus of Classical Higher-Order Processes (CHOP), which extends the latest version of the calculus of Classical Processes (CP) [6].

Types There are two kinds of types in CHOP: session types, ranged over by A, B, C, D, and process types, ranged over by T. Session types are inherited directly from CP, and correspond to linear logic propositions. We range over atomic propositions in session types with X, Y. Process types are used to type the communication of processes and the use of process variables.

We start by giving the syntax of session types, in the following, along with a short explanation of their meanings.

A, B, C, D	$::= A \otimes B$	*(send A, proceed as B)*	$\mid$	$A \parr B$	*(receive A, proceed as B)*
	$\mid A \oplus B$	*(select A or B)*	$\mid$	$A \& B$	*(offer A or B)*
	$\mid 0$	*(unit for $\oplus$)*	$\mid$	$\top$	*(unit for $\&$)*
	$\mid 1$	*(unit for $\otimes$)*	$\mid$	$\bot$	*(unit for $\parr$)*
	$\mid ?A$	*(client request)*	$\mid$	$!A$	*(server accept)*
	$\mid \exists X.A$	*(existential)*	$\mid$	$\forall X.A$	*(universal)*
	$\mid X$	*(atomic propositions)*	$\mid$	$X^\perp$	*(dual of atomic proposition)*

CP uses the standard notion of duality from linear logic to check that types are compatible. Above, each type constructor on the left-hand side is dual to that used on the right-hand side. We write $A^\perp$ for the type dual to A, defined inductively in the standard way (cf. [19] for details). For example, $(A \otimes B)^\perp = A^\perp \parr B^\perp$.

Each session in CP has two endpoints (one for each process in the session). Endpoints are ranged over by x, y, z. Session environments, ranged over by Γ, Δ, associate endpoints to session types: $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$. We make the standard assumption that Γ and Δ have distinct endpoints when writing Γ, Δ.

In CHOP, we can refer to processes that we receive at runtime (via process mobility) by using process variables, ranged over by p, q, r. A process environment Θ maps process variables to process types: $\Theta = \{p_1 : T_1, \ldots, p_n : T_n\}$. There is only one form for process types: $T ::= \Theta \to \Delta$. A process type assignment $p : \Theta \to \Delta$ reads "p implements Δ provided that the process variables in Θ are available". Note that Θ may be empty, meaning that p does not need to invoke other process variables to implement its session behaviour as specified in Δ.

$$\frac{}{\Theta \vdash x \to y^A :: x : A^\perp, y : A}\ \textsc{Axiom} \qquad \frac{\Theta \vdash P :: \Gamma, x : A \quad \Theta \vdash Q :: \Delta, y : A^\perp}{\Theta \vdash (\boldsymbol{\nu} x^A y)\,(P \mid Q) :: \Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{\Theta \vdash P :: \Gamma, y : A \quad \Theta \vdash Q :: \Delta, x : B}{\Theta \vdash x[y].(P \mid Q) :: \Gamma, \Delta, x : A \otimes B}\ \otimes \qquad \frac{\Theta \vdash P :: \Gamma, y : A, x : B}{\Theta \vdash x(y).P :: \Gamma, x : A \parr B}\ \parr$$

$$\frac{\Theta \vdash P :: \Gamma, x : A}{\Theta \vdash x[\mathsf{inl}].P :: \Gamma, x : A \oplus B}\ \oplus_1 \qquad \frac{\Theta \vdash P :: \Gamma, x : B}{\Theta \vdash x[\mathsf{inr}].P :: \Gamma, x : A \oplus B}\ \oplus_2$$

$$\frac{\Theta \vdash P :: \Gamma, x : A \quad \Theta \vdash Q :: \Gamma, x : B}{\Theta \vdash x.\mathsf{case}(P, Q) :: \Gamma, x : A \,\&\, B}\ \&$$

$$\frac{}{\Theta \vdash x[] :: x : 1}\ 1 \qquad \frac{\Theta \vdash P :: \Gamma}{\Theta \vdash x().P :: \Gamma, x : \bot}\ \bot \qquad \frac{}{\Theta \vdash x.\mathsf{case}() :: \Gamma, x : \top}\ \top$$

Fig. 1. CHOP, Selected Typing Rules (Part 1, Sessions).

$$\frac{}{\Theta', \Theta, p : \Theta \to \Delta \vdash p :: \Delta}\ \text{MP} \qquad \frac{\Theta' \vdash Q :: \Delta' \quad \Theta, q : \Theta' \to \Delta' \vdash P :: \Delta}{\Theta \vdash P[q := Q] :: \Delta}\ \textsc{Chop}$$

$$\frac{\Theta \vdash P :: \Delta}{\Theta \vdash x[P] :: x : \Theta \to \Delta}\ \to\text{R} \qquad \frac{\Theta, p : T \vdash R :: \Delta}{\Theta \vdash x(p).R :: \Delta, x : T^\perp}\ {}^\perp R$$

Fig. 2. CHOP, Typing Rules (Part 2, Higher-Order Processes).

Processes and Typing Let P, Q, R range over processes, the program terms of CHOP. We explain terms together with their respective typing rules. A typing judgement $\Theta \vdash P :: \Delta$ states that P implements the communication behaviour specified in Δ, possibly using the process variables specified in Θ.

We first briefly recap the terms and typing rules that we inherit from CP, displayed in Figure 1. (We omit the rules for exponentials and quantifiers, for space reasons. Θ is carried in the same way as for the other rules.) The process terms in Figure 1 are the same as in [6]. We adopt the same convention of having sent objects always in square brackets $[\ldots]$, and, dually, in an input operation the received variable is always bound in round parentheses $(\ldots)$. The endpoint name that we output in a send $x[y].(P \mid Q)$ and in a client request $?x[y].P$ is bound, as in the internal π-calculus [16]. (This will not be the case for process variables, as we are going to see shortly.) A forwarder term $x \to y^B$ forwards communications from x to y. The restriction term $(\boldsymbol{\nu} x^A y)\,(P \mid Q)$ connects two endpoints x and y to form a session, thus x and y are now able to communicate.

We now move to the new terms and typing rules introduced in this work for the communication of process terms, given in Figure 2.

Rule MP allows us to use process variables. It states that if we invoke a process variable p, typed with $\Theta \to \Delta$, and the process environment provides all the process variables that p may in turn use according to Θ, then we obtain an implementation of the session behaviour specified by Δ.

There are two ways of instantiating a process variable. One is receiving a process – $x(p).R$ – which allows us to use p later on (Rule $^{\perp}$R). The other is by defining the body of a variable explicitly. We denote this as an *explicit substitution* (inspired by the $\lambda\sigma$-calculus [1]) $P[q{:=}Q]$, read "let q be Q in P". We formalise how substitutions are propagated and applied later on, in our semantics for CHOP. Rule Chop allows us to substitute Q for q in P, provided that Q and q have compatible typing. If you think in terms of processes, Chop stands for "Cut for Higher-Order Processes". If you think in terms of logic, Chop stands for "Cut for Higher-Order Proofs". The idea is that a variable p stands for a "hole" in a proof, which has to be filled as expected by the type for p. This idea is also the reason for which Modus Ponens (MP), which is usually admissible, is given as an axiom. Since p represents a missing part of our proof, we do not know that its type is valid (i.e., that there exists a proof for some P such that $\Theta \vdash P :: \Delta$). We delegate this responsibility to the term that instantiates p with a process.

Lastly, a term $x[P]$ sends P along x (recalling a weakened version of the right rule for implication, $\to R$).

As an example, consider the following cloud server implementation. It provides a choice between two options. In the first case, we expect the client to send us an application p to run, which requires a connection with an internally-provided database (DB). In the second case, we expect to receive both the application p and the database q that it needs to use, putting them in parallel. (Thus, we may decide to use DB or not.) We omit the types for restrictions.

$$\big(\quad x.\mathsf{case}(\ x(p).(\boldsymbol{\nu} zw)\,(p\ \mid\ d)\ ,\ x(y).x(p).y(q).(\boldsymbol{\nu} zw)\,(p\ \mid\ q)\ )\quad\big)[d{:=}DB]$$

Semantics To give a semantics to CHOP, we follow the same approach as in [19,7]: we derive term reductions and equivalences from sound proof transformations.

Communications in CHOP are still defined by cut reductions over linear propositions, as in CP. The key insight that underlies our semantics for process mobility is that we interpret process types as atomic propositions in linear logic. That is, in the eyes of linear logic, a process type T is an atomic proposition (X). This twist allows us to integrate the expressivity of CP with our new rules: the dual of T is just $T^{\perp}$. (Different typing systems are often integrated this way.) A consequence is that we can cut a process output with a process input, as below.

$$\dfrac{\dfrac{\Theta \vdash P :: \Delta}{\Theta \vdash x[P] :: x : \Theta \to \Delta}\ {\to}\mathrm{R} \qquad \dfrac{\Theta, p : \Theta \to \Delta \vdash Q :: \Gamma}{\Theta \vdash y(p).Q :: \Gamma, y : (\Theta \to \Delta)^{\perp}}\ {}^{\perp}\,\mathrm{R}}{\Theta \vdash (\boldsymbol{\nu} x^{\Theta \to \Delta} y)\,(x[P]\ \mid\ y(p).Q) :: \Gamma}\ \textsc{Cut}$$

The above cut can always be eliminated by rewriting it into a (smaller) chop:

$$\frac{\Theta \vdash P :: \Delta \quad \Theta, p : \Theta \to \Delta \vdash Q :: \Gamma}{\Theta \vdash Q[p{:=}P] :: \Gamma}\ \textsc{Chop}$$

By following this idea we can derive the key β-reductions ($\longrightarrow$) for process mobility in CHOP, given in the following. We also give some examples of equivalences ($\equiv$) that define how explicit substitutions are propagated.

$$\begin{aligned}
(\boldsymbol{\nu} x^{\Theta\to\Delta} y)\,(x[P] \mid y(p).Q) &\longrightarrow Q[p{:=}P] \\
p[p{:=}P] &\longrightarrow P \\
(x[y].(Q \mid R))[p{:=}P] &\equiv x[y].(Q[p{:=}P] \mid R[p{:=}P])) \\
(x(y).Q)[p{:=}P] &\equiv x(y).(Q[p{:=}P])
\end{aligned}$$

For space reasons, we do not include all reductions and conversions. Note that we inherit all the original ones from CP (cf. [6]), for example those given below.

$$\begin{aligned}
(\boldsymbol{\nu} x^{A} y)\,(w \to x^{A} \mid Q) &\longrightarrow Q\{w/y\} \\
(\boldsymbol{\nu} x^{A\otimes B} y)\,(x[u].(P \mid Q) \mid y(v).R) &\longrightarrow (\boldsymbol{\nu} u^{A} v)\,(P \mid (\boldsymbol{\nu} x^{B} y)\,(Q \mid R)) \\
(\boldsymbol{\nu} x^{1} y)\,(x[] \mid y().P) &\longrightarrow P \\
(\boldsymbol{\nu} x^{A\oplus B} y)\,(x[\mathsf{inl}].P \mid y.\mathsf{case}(Q,R)) &\longrightarrow (\boldsymbol{\nu} x^{A} y)\,(P \mid Q) \\
(\boldsymbol{\nu} x^{A\oplus B} y)\,(x[\mathsf{inr}].P \mid y.\mathsf{case}(Q,R)) &\longrightarrow (\boldsymbol{\nu} x^{B} y)\,(P \mid R) \\
(\boldsymbol{\nu} x^{\exists X.B} y)\,(x[A].P \mid y(X).Q) &\longrightarrow (\boldsymbol{\nu} x^{B\{A/X\}} y)\,(P \mid Q\{A/X\})
\end{aligned}$$

3 Related Work

Other session calculi include primitives for moving processes by relying on a functional layer [17,14]. Differently, CHOP is nearer to the original Higher-Order π-calculus (HOπ) [15], where the communicated values are processes, instead of functions (or values as intended in λ-calculus). A consequence is that the theory of CHOP is simpler. For example, our language of session types remains separate from process types, which are opaque atomic propositions in the session types of CHOP. As such, we do not require the additional asymmetric connectives in session types used in [17] for communicating processes ($\supset$ and $\wedge$). The "send a process and continue over channel x" primitive of [17] can be encoded in CHOP as $x[y].(y[P] \mid Q)$ (similarly for receive). However, the functional layer in [17] allows for a remarkably elegant integration of recursive types, which is missing in CHOP. A possible direction to recover this feature is the work presented in [18].

In previous works, the process values that can be sent usually have the form $\lambda\tilde{x}.P$, to enable reuse in contexts that use different channel names. In CHOP, this is not necessary since we can get the same result with the forwarder term inherited from CP, which the receiver of a process can use to manipulate its names. For example, suppose that p has a free session endpoint x that we want to rename to w. We can just write $(\boldsymbol{\nu} y^{A} x)\,(w \to y^{A} \mid p)$ to obtain the desired effect: whichever process will replace p will communicate over w instead of x.

(It is straightforward to generalise this construction to arbitrarily many names, and to offer it through syntactic sugar, e.g., $\mathsf{let}\,\tilde{x} = \tilde{w}\,\mathsf{in}\,P$.) This is often done in practice, e.g., in the setting of microservices [9]; for example, in the Jolie programming language, the constructs of aggregation and embedding are used to implement this pattern [13], but without any type safety guarantees on the usage of sessions as in CHOP.

In [8] and [6], the notion of duality found in linear logic is replaced with coherence, which allows many processes to participate in a same session. We leave an extension of CHOP to such multiparty sessions to future work.

The calculus of Linear Compositional Choreographies (LCC) [7] gives a propositions-as-types correspondence for Choreographic Programming [12] based on linear logic. CHOP may provide the basis for extending LCC with process mobility, potentially yielding the first higher-order choreography calculus.

4 Conclusions

We presented CHOP, an attempt at extending CP to higher-order process communication. This paper is meant as a first attempt at formulating its theory. The reductions supported by CHOP, derived by sound proof transformations, are promising in the sense that: they realise communication as expected; they preserve typing; and they point out how process substitutions may be implemented efficiently (applied only where they are needed), recalling explicit substitutions for the λ-calculus [1].

We have deliberately postponed presenting the metatheory for CHOP. The reason is that its main results require careful formulation, since differently from CP it makes sense for processes to get stuck (in CP, well-typed processes always progress). For example, the process $(\boldsymbol{\nu} x^{\bot \otimes 1} y)\,(x(y).y().x[] \mid p)$ cannot reduce because of the free process variable p. We conjecture that a progress result can be formulated by appropriately instantiating free process variables whenever they are needed, similarly to how catalyser processes can be used in standard session types to provide all missing communication endpoints [5].

We end this work with an open question on expressivity. One of the reasons for which the standard π-calculus does not include process mobility is that it can be simulated through channel mobility. Now that we have an extension of CP to process mobility, can we prove the same result for CHOP? This would provide additional confidence on the fact that the propositions-as-types correspondence between linear logic propositions and session types is on the right track.

Acknowledgements. The author thanks Luís Cruz-Filipe and the anonymous reviewers for their useful comments. This work was supported by the CRC project, grant no. DFF–4005-00304 from the Danish Council for Independent Research, and by the Open Data Framework project at the University of Southern Denmark.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.
2. Luís Caires and Jorge A. Pérez. Multiparty session types within a canonical binary theory, and beyond. In *FORTE*, pages 74–95. Springer, 2016.
3. Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *ESOP*, pages 330–349. Springer, 2013.
4. Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *MSCS*, 26(3):367–423, 2016. Also: Caires and Pfenning, CONCUR, pages 222–236, 2010.
5. Marco Carbone, Ornela Dardha, and Fabrizio Montesi. Progress as compositional lock-freedom. In *COORDINATION*, pages 49–64. Springer, 2014.
6. Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, volume 59 of *LIPIcs*, pages 33:1–33:15, 2016.
7. Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distributed Computing*, pages 1–17, 2017. Also: CONCUR, pages 47–62, 2014.
8. Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 2017. Also: CONCUR, pages 412–426, 2015.
9. Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present And Ulterior Software Engineering (PAUSE)*. Springer-Verlag, 2017. To appear. Available at https://arxiv.org/abs/1606.04036.
10. Kohei Honda, Vasco Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP*, pages 22–138, 1998.
11. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
12. Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. `http://www.fabriziomontesi.com/files/choreographic_programming.pdf`.
13. Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
14. Dimitris Mostrous and Nobuko Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.
15. Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *TAPSOFT*, pages 151–166. Springer, 1993.
16. Davide Sangiorgi. Pi-calculus, internal mobility, and agent-passing calculi. *TCS*, 167(1&2):235–274, 1996.
17. Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013.
18. Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *TGC*, pages 159–175. Springer, 2014.
19. Philip Wadler. Propositions as sessions. *JFP*, 24(2–3):384–418, 2014. Also: ICFP, pages 273–286, 2012.