A typed parallel λ-calculus via 1-depth intermediate proofs[*]

Federico Aschieri[1], Agata Ciabattoni[1], and Francesco A. Genco[2]

[1] TU Wien, Vienna, Austria
`federico.aschieri@tuwien.ac.at, agata@logic.at`
[2] IHPST, Université Paris 1 and CNRS, Paris, France
`frgenco@gmail.com`

Abstract

We introduce a Curry–Howard correspondence for a large class of intermediate logics characterized by intuitionistic proofs with non-nested applications of rules for classical disjunctive tautologies (1-depth intermediate proofs). The resulting calculus, we call it $\lambda_{\parallel}$, is a strongly normalizing parallel extension of the simply typed λ-calculus. Although simple, the $\lambda_{\parallel}$ reduction rules can model arbitrary process network topologies, and encode interesting parallel programs ranging from numeric computation to algorithms on graphs.

1 Introduction

The fundamental connection between logic and computation, known as Curry–Howard correspondence, relates logics and computational systems. Originally introduced for intuitionistic logic **IL** and simply typed λ-calculus, it has been extended to many different logics (classical [26], linear [9, 5], modal [25]...) and notions of computation, see, e.g., [30] for an overview.

A recent addition to them is the discovery in [4] of the connection between propositional logics intermediate between classical logic and **IL**, and concurrent extensions of the simply typed λ-calculus. More precisely, the considered logics extend **IL** with the classical disjunctive tautologies interpreted as synchronization schemata in [14], i.e. axiom schemata of the form $(F_1 \to G_1) \vee \ldots \vee (F_m \to G_m)$. This general result was preceeded by Curry–Howard correspondences for the *particular cases* of Gödel-Dummett logic (**IL** with LIN $= (A \to B) \vee (B \to A)$) and classical logic (**IL** with EM $= A \vee \neg A$) in [2, 3]. All these logics possess cut-free hypersequent calculi – a generalization of sequent calculi consisting of parallel compositions of sequents that can "communicate"; our results confirmed Avron's conjecture [7] of the connection between (intermediate logics characterized by cut-free) hypersequent calculi and concurrency. Although the resulting typed concurrent λ-calculi provided an adequate computational interpretation of these logics, they are quite sophisticated and not easy to use as programming languages. Their main reductions are: *intuitionistic reductions* – which are the usual computational rules for the simply typed λ-calculus [16], *permutation reductions* – needed to prove weak normalization, and the reductions implementing communications among simply typed λ-terms; these are divided into *basic cross reductions* – simple reductions for the natural deduction version of the characteristic hypersequent rules, and *full cross reductions* – needed for the subformula property – which enable the transmission of messages that depend on their computational environment by using code mobility concepts [15] such as that of closure [20].

The aim of this work is to present a version of the calculi in [4, 2, 3] suitable for programming. We show that a simplified version of these calculi is expressive enough to model arbitrary process network topologies and to encode interesting parallel algorithms taken from [22]. Inspired

[*]Funded by the FWF grants Y544-N23 and P32080-N31, and project IBS (ANR-18-CE27-0012-01).

by [12, 13] we base our Curry–Howard correspondence on 1-depth[1] intermediate proofs, i.e., **IL** proofs with *non-nested* applications of the rules for classical disjunctive tautologies. This leads to $\lambda_\parallel$, an easy-to-use and yet expressive parallel extension of the simply typed λ-calculus. Consisting only of *intuitionistic reductions* and (a simplified version of) *basic cross reductions*, the reduction rules of $\lambda_\parallel$ always terminate, regardless of the reduction strategy.

Motivated by the tool Grace[2] [19] which allows programmers in the parallel functional language Eden to specify a network of processes as a directed graph and provides special constructs to generate the actual network topology, we also provide an automatic procedure to extract $\lambda_\parallel$ typing rules from any communication topology in such a way that the typed terms can only communicate according to the topology. The idea of enforcing network topologies with types is also present in [10] in the different context of the π-calculus [24] – the most widespread formalism for *modeling* concurrent systems.

We consider classical disjunctive tautologies in the (intuitionistically equivalent) form

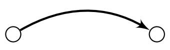
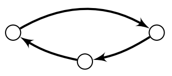
$$(\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{B}_1) \vee \ldots \vee (\mathbb{A}_m \to \mathbb{A}_m \wedge \mathbb{B}_m) \qquad (1)$$

such that all $\mathbb{B}_i$ are either $\bot$ or a conjunction of some $\mathbb{A}_1, \ldots, \mathbb{A}_m$. These axioms can indeed encode any reflexive directed graphs as follows: $\mathbb{A}_i$ represents a process in the network specified by the graph, and $\mathbb{B}_i$ is the list of processes that are connected to $\mathbb{A}_i$ by an outgoing arc, or $\bot$ if there are no such processes. The intuitive reading of $(\mathbb{A}_i \to \mathbb{A}_i \wedge \mathbb{B}_i)$ is that all conjuncts in $\mathbb{B}_i$, as well as $\mathbb{A}_i$ itself, can send messages to $\mathbb{A}_i$. We allow $\mathbb{A}_i$ to send a message to itself in case $\mathbb{A}_i$ wants to save it for later use, thus simulating a memory mechanism.

We establish a Curry–Howard correspondence between $\lambda_\parallel$ terms and (fragments of) the intermediate logics obtained from the intuitionistic natural deduction calculus **NJ** by allowing a non-nested use of the rules corresponding to axioms of the form (1). The terms typed using these natural deduction rules contain as many parallel processes as the number of premises. For instance, the decorated version of the rules for the axiom schema $em = (\mathbb{A} \to \mathbb{A}\wedge\bot)\vee(\mathbb{B} \to \mathbb{B}\wedge\mathbb{A})$ and $C_3 = (\mathbb{A} \to \mathbb{A}\wedge\mathbb{B}) \vee (\mathbb{B} \to \mathbb{B}\wedge\mathbb{C}) \vee (\mathbb{C} \to \mathbb{C}\wedge\mathbb{A})$, intuitionistically equivalent to the excluded middle law and C_3 [23] are

$$\dfrac{\begin{matrix}[a : A \to A \wedge \bot] \\ \vdots \\ u : C\end{matrix} \qquad \begin{matrix}[a : B \to B \wedge A] \\ \vdots \\ v : C\end{matrix}}{{}_a(u \parallel v) : C}\,(em) \qquad \dfrac{\begin{matrix}[a : A \to A \wedge B] \\ \vdots \\ t : D\end{matrix} \qquad \begin{matrix}[a : B \to B \wedge C] \\ \vdots \\ u : D\end{matrix} \qquad \begin{matrix}[a : C \to C \wedge A] \\ \vdots \\ v : D\end{matrix}}{{}_a(t \parallel u \parallel v) : D}\,(C_3)$$

with the (1-depth) restriction: t, u and v are a parallel composition of simply-typed λ-terms that cannot communicate with each other. The variable a represents a private communication channel that behaves similarly to the π-calculus operator ν. The typing rules establish how the communication channels connect the terms. For example, (em) encodes the fact that the process v can receive a message of type A from the process u, while the rule (C_3) that t can receive a message from u, u from v and v from t. The behavior of these channels during the actual communications is defined by the reduction rules of the calculus. If we omit reflexive edges, the communication topologies corresponding to the reductions for the above rules are

(em)  (C_3) 

In particular, (em) implements the simplest message-passing mechanism, similar to that of the asynchronous π-calculus [18], and (C_3) cyclic communication among three processes.

The restriction (1-depth) enables us to define simple and yet expressive reduction rules and to prove the strong normalization of $\lambda_\parallel$. The resulting calculus is strictly more expressive than

[1] [12, 13] Introduced the concept of *bounded depth proofs* to approximate classical logic with nested applications of the structural rule expressing the excluded middle axiom EM limited by a fixed natural number.

[2]Grace stands for GRAph-based Communication in Eden.

$$x^A : A \qquad \frac{u : \bot}{u\,\mathsf{efq}_P : P} \text{ with } P \text{ atomic, } P \neq \bot \qquad \frac{\begin{array}{c}[x^A : A]\\ \vdots \\ u : B\end{array}}{\lambda x^A u : A \to B} \qquad \frac{t : A \to B \quad u : A}{tu : B}$$

$$\frac{u : A \quad t : B}{\langle u, t\rangle : A \wedge B} \qquad \frac{u : A \wedge B}{u\,\pi_0 : A} \qquad \frac{u : A \wedge B}{u\,\pi_1 : B}$$

Table 1: Type assignments for the simply typed λ-calculus.

the simply typed λ-calculus, and can be used for interesting computational tasks (Sec. 6).

2 The Typing System of $\lambda_{\|}$

$\lambda_{\|}$ extends the simply typed λ-calculus with channels for multi-party communication and reduction rules for message exchange. Table 1 contains the type assignments for λ-terms, see e.g. [16] for details. Terms typed by such rules are called **simply typed λ-terms** and are denoted here by $t, u, v \dots$ Terms may contain variables $x_0^A, x_1^A, x_2^A, \dots$ of type A for every formula A. Free and bound variables of a proof term are defined as usual. We assume the standard renaming rules and α-equivalences that avoid the capture of variables during reductions.

The typing rules of simply typed λ-calculus, stripped of λ-terms, are the inference rules of Gentzen's natural deduction system **NJ** for **IL**. Actually, if $\Gamma = x_1 : A_1,\dots, x_n : A_n$, and all free variables of a term $t : A$ are in $x_1,\dots,x_n$, from the logical point of view, t represents an **NJ** derivation of A from the hypotheses $A_1,\dots, A_n$. We will thus write $\Gamma \vdash t : A$.
Notation. $\to$ and $\wedge$ associate to the right. $\langle t_1, t_2, \dots, t_n\rangle$ stands for $\langle t_1, \langle t_2, \dots \langle t_{n-1}, t_n\rangle \dots\rangle\rangle$, and $\langle t_1, t_2, \dots, t_n\rangle\,\pi_i$, $(i = 0, \dots, n)$ for $\langle t_1, t_2, \dots, t_n\rangle\,\pi_1 \dots \pi_1\pi_0$ containing the projections that select the $(i + 1)$th element of the sequence. $\neg A$ is $A \to \bot$ and $\top$ is $\bot \to \bot$.

To type parallel terms that interact according to possibly complex communication mechanisms, we build on ideas from [4, 12, 14, 11] and base the Curry–Howard correspondence for $\lambda_{\|}$ on a fragment of the axiomatic extensions of **IL** with (intuitionistically equivalent versions of) classical disjunctive tautologies [14]. These axioms can be transformed into rules involving parallel communicating sequents (i.e., hypersequents [7]), and as shown in [4] they lead to various communication schemata. The fragment considered in this paper consists of 1-depth proofs, whose notion is adapted from [12, 13]. These are **NJ** proofs with *non-nested* applications of the natural deduction version of the corresponding hypersequent rules, see [11]. The use of this fragment drastically simplifies the reduction rules of $\lambda_{\|}$ w.r.t. the calculi in [4, 2, 3] and enables us to type channels with input/output directions, as in the π calculus.

The class Ax of axiom schemata that we consided here, that will be shown in Sec. 4 to encode all communication topologies represented as reflexive directed graphs, is

$$(\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{B}_1) \vee \dots \vee (\mathbb{A}_m \to \mathbb{A}_m \wedge \mathbb{B}_m)$$

where $\mathbb{A}_i \neq \mathbb{A}_j$ whenever $i \neq j$; and for any $i \in \{1, \dots, m\}$, either $\mathbb{B}_i = \bot$ or $\mathbb{B}_i = \mathbb{A}_{k_1} \wedge \dots \wedge \mathbb{A}_{k_p}$, with $k_1 < \dots < k_p$.

Remark 1. *Each disjunct $\mathbb{A}_i \to \mathbb{A}_i \wedge \mathbb{B}_i$ is logically equivalent to $\mathbb{A}_i \to \mathbb{B}_i$. The redundant occurrence of $\mathbb{A}_i$ is kept to type a memory mechanism for input channels.*

As usual, an instance of an axiom schema $\mathcal{A} \in \mathsf{Ax}$ is a formula obtained from $\mathcal{A}$ by uniformly replacing each propositional variable with an actual formula. $\lambda_{\|}$ is obtained by the Curry–Howard correspondence applied to **NJ** extended with non-nested applications of the natural deduction rules for the axioms in Ax. The type assignment for $\lambda_{\|}$ terms comprises the rules for the simply typed λ-calculus in Table 1 and those for the parallel operators in Table 2. We denote the variables of simply typed λ-calculus as x^A, y^A, $z^A, \dots$, $a^A, b^A, c^A, \dots$,

$$\frac{t_1 : A \quad \dots \quad t_n : A}{t_1 \parallel \dots \parallel t_n : A}\ (\mathit{contr}) \qquad \text{where } t_1, \dots, t_n \text{ are simply typed } \lambda\text{-terms}$$

$$\dfrac{\begin{array}{c}[\overline{a}^{\,A_1 \to A_1 \wedge B_1}, \underline{a}^{\,A_1 \to A_1 \wedge B_1} : A_1 \to A_1 \wedge B_1] \\ \vdots \\ u_1 \parallel \dots \parallel u_n : B\end{array} \qquad \dots \qquad \begin{array}{c}[\overline{a}^{\,A_m \to A_m \wedge B_m}, \underline{a}^{\,A_m \to A_m \wedge B_m} : A_m \to A_m \wedge B_m] \\ \vdots \\ u_p \parallel \dots \parallel u_q : B\end{array}}{{}^{\mathcal{A}}_{a}((u_1 \parallel \dots \parallel u_n) \parallel \dots \parallel (u_p \parallel \dots \parallel u_q)) : B}\ (\mathcal{A})$$

where $(A_1 \to A_1 \wedge B_1) \vee \dots \vee (A_m \to A_m \wedge B_m)$ is an instance of $\mathcal{A} \in \mathsf{Ax}$

Table 2: Type assignments for $\lambda_{\parallel}$.

$\overline{a}^{\,A}, \overline{b}^{\,A}, \overline{c}^{\,A} \dots, \underline{a}^{\,A}, \underline{b}^{\,A}, \underline{c}^{\,A} \dots$ and, whenever the type is not important, as $x, y, z, \dots, a, b, c \dots$ We call **intuitionistic variables** the variables $x, y, z, \dots$, which stand for terms; they are bound by the λ operator. The variables $a, \overline{a}, \underline{a}, \dots$ are called **channels** or **communication variables** and represent communication channels between parallel processes: $a, b, c, \dots$ are used as channel binders, $\overline{a}, \overline{b}, \overline{c}, \dots$ represent **output channels** that can transmit messages, while $\underline{a}, \underline{b}, \underline{c}, \dots$ **input channels** that can receive messages. We denote ${}^{\mathcal{A}}_{a}(u_1 \parallel \dots \parallel u_m)$ by ${}_{a}(u_1 \parallel \dots \parallel u_m)$ when $\mathcal{A}$ is clear from the context or irrelevant. All free occurrences of $\overline{a}$ and $\underline{a}$ in $u_1, \dots, u_m$ are bound in ${}_{a}(u_1 \parallel \dots \parallel u_m)$ and must have the types indicated by the inference rule $(\mathcal{A})$.

The rule (contr) is useful for representing parallelism without communication. It is logically redundant, though, since it is an instance of $(\mathcal{A})$ with no channel occurrence.

From a computational perspective the rules $(\mathcal{A})$ produce terms of the shape ${}^{\mathcal{A}}_{a}(v_1 \parallel \dots \parallel v_m)$ that put in parallel $v_1, \dots, v_m$, which we call the **processes** of this term; each v_i in turn has the shape $u_1 \parallel \dots \parallel u_k$, where $u_1, \dots, u_k$ are simply typed λ-terms called the **threads** of v_i. Processes can communicate through the channel a, whereas their threads represent parallel independent subprograms that cannot interact with each other. Informally, in order to establish a communication channel connecting two terms v_i and v_j, we require that $\overline{a}^{\,A_i \to A_i \wedge B_i}$ occurs in v_i, $\underline{a}^{\,A_j \to A_j \wedge B_j}$ occurs in v_j and A_i is in B_j.

On one hand, the argument w of a channel application $\overline{a}^{\,A_i \to A_i \wedge B_i}\, w$ will be interpreted as a message of type A_i that must be *transmitted*; on the other hand, the channel application $\underline{a}^{\,A_j \to A_j \wedge B_j}\, t$ will *receive* a batch of messages of type B_j containing w that will replace the whole channel application $\underline{a}^{\,A_j \to A_j \wedge B_j}\, t$ upon reception. This is the reason why the direction of the communication and the direction of $\to$ are reversed. In general, each channel application $\overline{a}\, v$ first transmits v and, immediately after that, starts to listen on the same channel by reducing to $\underline{a}\, v$ and waiting for a message that will replace the whole term $\underline{a}\, v$. To formalize the relation between a process v_i and all the processes v_j such that v_i can send messages to v_j, we need to look at the structure of the axiom schema $\mathcal{A}$, because its instances may lose information about its general shape. For this purpose, we introduce the concept of outlink.

Definition 2 (Outlinks)**.** *Let* ${}^{\mathcal{A}}_{a}(v_1 \parallel \dots \parallel v_m)$ *be a term, where* $\mathcal{A} = (\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{B}_1) \vee \dots \vee (\mathbb{A}_m \to \mathbb{A}_m \wedge \mathbb{B}_m)$*. For any* $i, j \in \{1, \dots, m\}$ *and* $i \neq j$*, we say that the term* v_i *is* ***outlinked*** *to the term* v_j *if* $\mathbb{B}_j = \mathbb{A}_{k_1} \wedge \dots \wedge \mathbb{A}_i \wedge \dots \wedge \mathbb{A}_{k_p}$.

Remark 3. *The restriction of (1-depth) proofs forces the derivations of the premises of the rule* $(\mathcal{A})$ *to be* **IL** *proofs; by removing this restriction, the rule is equivalent to an (unrestricted) instance of the axiom* $\mathcal{A}$*, see [11].*

3 Communications in $\lambda_{\parallel}$

In Sec. 2 we showed how to install communication channels connecting processes, we present now the reduction rules of $\lambda_{\parallel}$ that implement the actual communications. Since these communications are higher-order, they can transmit arbitrary simply typed λ-terms as messages, provided their free variables are not bound in the surrounding context. Unlike in [4], messages are thus not restricted to values and communication channels have directions. The reduction rules of $\lambda_{\|}$ comprise two groups of rules: those for the simply typed λ-calculus (*intuitionistic reductions*), and those that deal with process communication (*cross reductions*).

Intuitionistic Reductions. The usual computational rules for the simply typed λ-calculus represent the operations of applying a function to an argument and extracting a component from a pair [16]. From the logical point of view, they are the standard Prawitz' reductions [27] of the natural deduction calculus **NJ** for **IL**.

Cross Reductions. Their goal is to implement communication, namely to transmit programs in the form of simply typed λ-terms. A very simple example of cross reduction for the (em) topology shown in Section 1 is the following:

$$_a(\mathcal{C}[\overline{a}\, v] \parallel \mathcal{D}[\underline{a}\, w]) \;\mapsto\; _a(\mathcal{C}[\underline{a}\, v] \parallel \mathcal{D}[\langle w, v\rangle])$$

where $\mathcal{C}[\,], \mathcal{D}[\,], v$ and w contain no channels. In general, since $\lambda_{\|}$ terms may contain more than one output channel application, we first have to choose which application will transmit the next message. For example, here we have two communicating processes, each containing two threads:

$$(*) \qquad _a(\;(\underline{a}\; r \parallel \overline{a}\,(x(\overline{a}\; s))) \;\;\parallel\;\; (\underline{a}\; u \parallel \underline{a}\,(y(\underline{a}\, w)))\;)$$

where r, s, u, w are simply typed λ-terms not containing a. The first process $\underline{a}\; r \parallel \overline{a}\,(x(\overline{a}\; s))$ contains two occurrences of the output channel $\overline{a}$. Let us focus on its second thread $\overline{a}\,(x(\overline{a}\; s))$. The channel application $\overline{a}\,(x(\overline{a}\; s))$ cannot transmit the message $x(\overline{a}\; s)$, because the channel a might be used with a different type in the second process, so type preservation after reduction would fail. Hence the only possibility here is to choose the second channel application $\overline{a}\; s$ as the one containing the output message, in this case s. In general, to make sure that the message does not contain the channel a, it is enough to choose as application occurrence that contains the output message the *rightmost occurrence* of the channel a in the whole process, provided it occurs as an actual output channel $\overline{a}$. Hence, in any process, the rightmost thread that contains the channel $\overline{a}$ may contain the message. If instead the *rightmost occurrence* of a is of the form $\underline{a}$, the process has no message to send.

The second choice is which occurrence of an input channel application should receive the current message. For example, in the term $(*)$ above, the second process $\underline{a}\; u \parallel \underline{a}\,(y(\underline{a}\, w))$ contains three occurrences of $\underline{a}$. Since a channel can both send and receive, and has usually sent something before receiving, we are led to choose again the rightmost occurrence of a channel application as receiver, provided it is an input channel. Nonetheless, since the threads of a single process do not communicate with each other, it is best to let all of them receive the message in correspondence of their *locally rightmost* channel application. Thus in the example, both $\underline{a}\; u$ and $\underline{a}\; w$ will receive messages.

The third choice is what to do with the arguments of a receiving channel. After a few programming examples, as those in Section 6, it is natural to convince oneself that it is better to keep the arguments, a feature that we call *memory.* In the previous term, when $\underline{a}\; u$ and $\underline{a}\; w$ receive s, they will be replaced respectively, with $\langle u, s\rangle$ and $\langle w, s\rangle$.

Continuing our example and summing up, we will have the following reduction of $(*)$:

$$_a(\;(\underline{a}\; r \parallel \overline{a}\,(x(\overline{a}\; s))) \;\;\parallel\;\; (\underline{a}\; u \parallel \underline{a}\,(y(\underline{a}\, w)))\;) \quad\mapsto\quad _a(\;(\underline{a}\; r \parallel \overline{a}\,(x(\underline{a}\; s))) \;\parallel\; (\langle u, s\rangle \parallel \underline{a}\,(y\langle w, s\rangle))\;)$$

As we can see, the message s in correspondence of the rightmost occurrence of $\overline{a}$ in $\overline{a}\,(x(\overline{a}\; s))$ is transmitted by the first process to the rightmost application of $\underline{a}$ in each one of the two threads $\underline{a}\; u$ and $\underline{a}\,(y(\underline{a}\, w))$ of the second process; at the same time, the output channel application $\overline{a}\; s$ has turned into the input channel application $\underline{a}\; s$.

The fourth choice to make is which processes should receive messages and which processes should send them. As anticipated in the previous section, we are guided by the typing. Let us consider for instance the term ${}_{a}(\ x\,(\underline{a}^{\,A\to A\wedge B}\ s) \parallel\ y\,(\overline{a}^{\,B\to B\wedge A}\ t)\ )$, where s and t are specific simply typed λ-terms not containing a, while $x : A \wedge B \to C$ and $y : B \wedge A \to C$. A cross reduction rule corresponding to this typing rule admits communication in two directions, from left to right and from right to left, because, according to the types, the second process can receive messages from the first and viceversa. The actual direction of the message is determined however by the rightmost occurrences of the channel a in the two processes. In this case, the rightmost occurrence of a in the first process is an input channel, while the rightmost occurrence of a in second process is an output channel. Hence the reduction is from right to left and is ${}_{a}(\ x\,(\underline{a}^{\,A\to A\wedge B}\ s) \parallel\ y\,(\overline{a}^{\,B\to B\wedge A}\ t)\ ) \ \mapsto\ {}_{a}(\ x\,\langle s,t\rangle \parallel y\,(\underline{a}^{\,B\to B\wedge A}\ t)\ \ )$

To define communication reductions, we need to introduce two kinds of contexts: one for terms that can communicate, one for terms which are in parallel but cannot communicate.

Definition 4 (Simple Parallel Term)**.** *A* ***simple parallel term*** *is a* $\lambda_{\parallel}$ *term* $t_1 \parallel \ldots \parallel t_n$*, where each* t_i*, for* $1 \le i \le n$*, is a simply typed λ-term.*

Definition 5. *A* ***context*** $\mathcal{C}[\ ]$ *is a* $\lambda_{\parallel}$ *term with some fixed variable* $[\ ]$ *occurring*

- *A* ***simple context*** *is a context which is a simply typed λ-term.*
- *A* ***simple parallel context*** *is a context which is a simple parallel term.*

For any $\lambda_{\parallel}$ *term* u *of the same type of* $[\ ]$*,* $\mathcal{C}[u]$ *denotes the term obtained replacing* $[\ ]$ *with* u *in* $\mathcal{C}[\ ]$*,* without renaming bound variables.

We explain the general case of the cross reduction. The rule identifies a single process as the receiver and possibly several processes as senders. Once the receiving process is fixed, the senders are determined by the axiom schema $\mathcal{A}$, that is instantiated by the type of the communication channel occurring in the receiving process. In particular, in the term

$$(\star)\qquad {}^{\mathcal{A}}_{a}(\ldots\parallel \mathcal{C}_1[\overline{a}\,w_1] \parallel\ldots\parallel (\ldots \parallel \mathcal{D}_1[\underline{a}\ v_1] \parallel\ldots\parallel \mathcal{D}_n[\underline{a}\ v_n] \parallel \ldots) \parallel\ldots\parallel \mathcal{C}_p[\overline{a}\,w_p] \parallel\ldots)$$

the processes $\mathcal{C}_1[\overline{a}\,w_1\ ], \ldots, \mathcal{C}_p[\overline{a}\,w_p\ ]$ are the senders and $(\ldots \parallel \mathcal{D}_1[\underline{a}\ v_1] \parallel\ldots\parallel \mathcal{D}_n[\underline{a}\ v_n] \parallel \ldots)$ in its entirety is the receiver. Formally, $\mathcal{C}_1[\overline{a}\,w_1\ ], \ldots, \mathcal{C}_p[\overline{a}\,w_p\ ]$ are all the process *outlinked* to the process $(\ldots \parallel \mathcal{D}_1[\underline{a}\ v_1] \parallel\ldots\parallel \mathcal{D}_n[\underline{a}\ v_n] \parallel \ldots)$, see Definition 2. In this latter process, we have displayed the threads that actually contain the channel a: all of them will receive the messages. Consistently with our choices, the displayed occurrences of a are rightmost in each $\mathcal{D}_j[\underline{a}\ v_j]$ and in each $\mathcal{C}_j[\overline{a}\ w_j]$. The processes containing $w_1, \ldots, w_p$ send them to all the rightmost occurrences of $\underline{a}\ v_1, \ldots, \underline{a}\ v_n$ in the processes $\mathcal{D}_1, \ldots, \mathcal{D}_n$:

$$(\star)\quad \mapsto\quad {}^{\mathcal{A}}_{a}(\ldots\parallel \mathcal{C}_1[a w_1] \parallel\ldots\parallel(\ldots\parallel \mathcal{D}_1[\langle v_1, w_1,\ldots, w_p\rangle] \parallel\ldots\parallel \mathcal{D}_n[\langle v_n, w_1,\ldots, w_p\rangle] \parallel\ldots)\parallel\ldots\parallel \mathcal{C}_p[a w_p] \parallel\ldots)$$

provided that, for each w_j, its free variables are free in $\mathcal{C}_j[a\,w_j]$: this condition is needed to avoid that bound variables become free, violating the Subject Reduction. Whenever w_j is a closed term – executable code – the condition is automatically satisfied. As we can see, the reduction retains all terms $v_1, \ldots, v_n$ occurring in $\mathcal{D}_1, \ldots, \mathcal{D}_n$ before the communication.

Remark 6. *Unlike in π-calculus, channel* occurrences *that send messages are not immediately consumed in* $\lambda_{\parallel}$*, because cross reductions adopt the perspective of the receiver rather than that of the senders. However, after transmission, the output channel occurrence will be turned immediately into an input channel occurrence, which will be consumed when the process is selected as current receiver.*

The last choice is what to do with the threads or processes that do not contain any communication channel. The idea is that whenever a term contains no channel occurrence, it has already reached a result, as it does not need to interact with the context. Hence at the

Intuitionistic Reductions $(\lambda x^A\, u)t \mapsto u[t/x^A]$ $\langle u_0, u_1\rangle\, \pi_i \mapsto u_i$ for $i = 0, 1$

Cross Reductions: Communication

$$\begin{array}{c} {}^{\mathcal{A}}_{a}(\ldots \parallel \mathcal{C}_1[\overline{a}\, w_1] \parallel\ldots\parallel(\ldots\parallel \mathcal{D}_1[\underline{a}\ v_1] \parallel\ldots\parallel \mathcal{D}_n[\underline{a}\ v_n] \parallel\ldots)\parallel\ldots\parallel \mathcal{C}_p[\overline{a}\, w_p] \parallel\ldots) \quad \mapsto \\ {}^{\mathcal{A}}_{a}(\ldots\parallel \mathcal{C}_1[\underline{a}\, w_1] \parallel\ldots\parallel(\ldots\parallel \mathcal{D}_1[\langle v_1, w_1,\ldots, w_p\rangle] \parallel\ldots\parallel \mathcal{D}_n[\langle v_n, w_1,\ldots, w_p\rangle] \parallel\ldots)\parallel\ldots\parallel \mathcal{C}_p[\underline{a}\, w_p] \parallel\ldots) \end{array}$$

where $\mathcal{C}_1[\overline{a}\ w_1], \ldots, \mathcal{C}_p[\overline{a}\ w_p]$ are all the processes outlinked to the process $(\ldots \parallel \mathcal{D}_1[\underline{a}\ v_1] \parallel \ldots \parallel \mathcal{D}_n[\underline{a}\ v_n] \parallel \ldots)$; the displayed occurrences of a are rightmost in each $\mathcal{D}_j[\underline{a}\ v_j]$ and in each $\mathcal{C}_j[\overline{a}\ w_j]$; each $\mathcal{D}_j$ is a simple context and each $\mathcal{C}_j$ is a simple parallel context; the free variables of each w_j are free in $\mathcal{C}_j[\overline{a}\ w_j]$;

Cross Reductions: Simplification ${}^{\mathcal{A}}_{a}((u_1 \parallel\ldots\parallel u_n) \parallel\ldots\parallel (u_m \parallel\ldots\parallel u_p)) \mapsto u_{i_1} \parallel\ldots\parallel u_{i_q}$
whenever $u_{i_1},\ldots, u_{i_q}$ do not contain a and $1 \le i_1 <\ldots< i_q \le p$.

Table 3: Reduction Rules for $\lambda_{\parallel}$.

end of the computation we select some of the processes that have reached their own results and consider them all together the global result of the computation. Thus we introduce the simplification reduction in Table 3, which also displays all the reduction rules of $\lambda_{\parallel}$. As usual, we adopt the reduction schema: $\mathcal{C}[t] \mapsto \mathcal{C}[u]$ whenever $t \mapsto u$ and for any context $\mathcal{C}$. We denote by $\mapsto^*$ the reflexive and transitive closure of the one-step reduction $\mapsto$.

Theorem 7 (Subject Reduction)**.** *If $t : A$ and $t \mapsto u$, then $u : A$ and all the free variables of u appear among those of t.*

Proof. The only not trivial case is that of cross reductions. Assume the step is as follows:

$$\begin{array}{c} {}^{\mathcal{A}}_{a}(\ldots\parallel \mathcal{C}_1[\overline{a}\, w_1] \parallel\ldots\parallel(\ldots\parallel \mathcal{D}_1[\underline{a}^{\,A_i\to A_i\wedge B_i}\ v_1] \parallel\ldots\parallel \mathcal{D}_n[\underline{a}^{\,A_i\to A_i\wedge B_i}\ v_n] \parallel\ldots)\parallel\ldots\parallel \mathcal{C}_p[\overline{a}\, w_p] \parallel\ldots) \\ \mapsto \\ {}^{\mathcal{A}}_{a}(\ldots\parallel \mathcal{C}_1[\underline{a}\, w_1] \parallel\ldots\parallel(\ldots\parallel \mathcal{D}_1[\langle v_1, w_1,\ldots, w_p\rangle] \parallel\ldots\parallel \mathcal{D}_n[\langle v_n, w_1,\ldots, w_p\rangle] \parallel\ldots)\parallel\ldots\parallel \mathcal{C}_p[\underline{a}\, w_p] \parallel\ldots) \end{array}$$

As the type of the channel a is an instance $(A_1 \to A_1 \wedge B_1) \vee\ldots\vee (A_m \to A_m \wedge B_m)$ of the schema $\mathcal{A} = (\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{B}_1) \vee\ldots\vee (\mathbb{A}_m \to \mathbb{A}_m \wedge \mathbb{B}_m)$ where $\mathbb{B}_i = \mathbb{A}_{k_1} \wedge\ldots\wedge \mathbb{A}_{k_p}$, and as $w_1 : A_{k_1},\ldots, w_p : A_{k_p}$ where $A_{k_1},\ldots, A_{k_p}$ instantiate $\mathbb{A}_{k_1},\ldots, \mathbb{A}_{k_p}$, then $(\ldots\parallel \mathcal{D}_1[\langle v_1, w_1,\ldots, w_p\rangle] \parallel\ldots\parallel \mathcal{D}_n[\langle v_n, w_1,\ldots, w_p\rangle] \parallel\ldots)$ is well defined and its type is the same as that of $(\ldots\parallel \mathcal{D}_1[\underline{a}^{\,A_i\to A_i\wedge B_i}\ v_1] \parallel \ldots\parallel \mathcal{D}_n[\underline{a}^{\,A_i\to A_i\wedge B_i}\ v_n] \parallel\ldots)$. Hence the term type term does not change.

Since the displayed occurrences of a are rightmost in each $\mathcal{C}_j[\overline{a}\ w_j]$ and thus a does not occur in w_j, no occurrence of a with type different from $A_i \to A_i \wedge B_i$ occurs in the terms $\mathcal{D}_1[\langle v_1, w_1,\ldots, w_p\rangle],\ldots, \mathcal{D}_n[\langle v_n, w_1,\ldots, w_p\rangle]$. Finally, the free variables of each w_j are free also in $\mathcal{C}_j[\overline{a}\ w_j]$, and thus no new free variable is created by the reduction.

□

Remark 8. *The communication reductions of $\lambda_{\parallel}$ allow processes to communicate independently of their order, and hence the parallel composition needs not to be commutative.*

4 From Communication Topologies to Programs

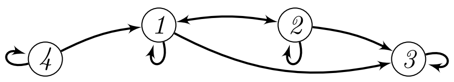
We present a method for automatically extracting $\lambda_{\parallel}$ typing rules from graph-specified communication topologies. Given a directed, reflexive graph G whose nodes and edges represent respectively processes and communication channels, we describe how to transform it into an axiom schema $\mathcal{A} \in \mathsf{Ax}$ corresponding to a typing $(\mathcal{A})$ rule for $\lambda_{\parallel}$ terms. We will show that this typing rule encodes a process topology which exactly mirrors the graph G: two processes may communicate if and only the corresponding graph nodes are connected by an edge and the direction of communication follows the edge. In other words, the edges of the graph correspond to the *outlinked* relation between processes of Definition 2.

Procedure 1. *Given a directed reflexive graph $G = (V, E)$ with $k = |V|$, the axiom schema $\mathcal{A}$ encoding G is $\mathbb{C}_1 \vee \cdots \vee \mathbb{C}_k$ such that for each $n \in \{1, \ldots, k\}$:*

- $\mathbb{C}_n = \mathbb{A}_n \to \mathbb{A}_n \wedge \mathbb{A}_{i_1} \wedge \ldots \wedge \mathbb{A}_{i_m}$*, if the n-th node in G has incoming edges from the non-empty list of pairwise distinct nodes $i_1, \ldots, i_m \neq n$;*
- $\mathbb{C}_n = \mathbb{A}_n \to \mathbb{A}_n \wedge \bot$*, if the n-th node in G has only one incoming edge.*

We show an example and then state the correspondence between graphs and $\lambda_{\|}$ reductions.

Example 9. *Consider the axiom below corresponding to the graph*



$$(\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{A}_2 \wedge \mathbb{A}_4) \vee (\mathbb{A}_2 \to \mathbb{A}_2 \wedge \mathbb{A}_1) \vee (\mathbb{A}_3 \to \mathbb{A}_3 \wedge \mathbb{A}_1 \wedge \mathbb{A}_2) \vee (\mathbb{A}_4 \to \mathbb{A}_4 \wedge \bot)$$

The rule extracted from a graph forces communications to happen as indicated by the edges of the graph. Indeed the following holds.

Proposition 10 (Topology Correspondence)**.** *For any directed reflexive graph G and term $t := {}_a(u_1 \parallel \ldots \parallel u_m)$ typed using the rule corresponding to the axiom extracted from G by Procedure 1, there is a cross reduction for t that transmits a term w from u_x to u_y if and only if G contains an edge from x to y.*

5 The Strong Normalization Theorem

We prove the strong normalization theorem for $\lambda_{\|}$: any reduction of every $\lambda_{\|}$ term ends in a finite number of steps into a normal form. This means that the computation of any typed $\lambda_{\|}$ term always terminates independently of the chosen reduction strategy. Instead of struggling directly with the complicated combinatorial properties of process communication, as done for the weak normalization in [3, 2], we reduce the strong normalization of $\lambda_{\|}$ to the strong normalization of a non-deterministic reduction relation over simply typed λ-terms. The idea is to simulate communication by non-determinism, a technique inspired by [6].

Definition 11 (Normal Forms and Strongly Normalizable Terms)**.**

- *A **redex** is a term u such that $u \mapsto v$ for some v and reduction in Table 3. A term t is called a **normal form** or, simply, **normal**, if t is not a redex.*
- *A finite or infinite sequence of terms $u_1, u_2, \ldots, u_n, \ldots$ is said to be a **reduction** of t, if $t = u_1$, and for all i, $u_i \mapsto u_{i+1}$. A term u of $\lambda_{\|}$ is **normalizable** if there is a finite reduction of u whose last term is normal and is **strong normalizable** if every reduction of u is finite. With SN we denote the set of the strongly normalizable terms of $\lambda_{\|}$.*

5.1 Non-deterministic Reductions

Strong normalization for parallel λ-calculi is an intricate problem. Decreasing complexity measures are quite hard to find and indeed those introduced in [3, 2] for proving normalization of fragments of $\lambda_{\|}$ fail in case of strong normalization. Given a term ${}_a(u_1 \parallel \ldots \parallel u_n)$, one would like to measure its complexity as a function of the complexities of the terms u_i, for example taking into account the number of channel occurrences and the length of the longest reduction of λ-redexes in u_i. However, when u_i receives a message its code may drastically change and both those numbers may increase. Moreover, there is a potential circularity to address: channels send messages and may generate new λ-calculus redexes in the receivers; λ-calculus redexes may duplicate channel occurrences, generating even more communications.

To break this circularity, we use a radically different complexity measure. The complexity of a process u_i should take into account all the possible messages that u_i may receive. If $u_i = \mathcal{D}[\underline{a}\ t]$, after receiving a message it becomes $\mathcal{D}[\langle t, s\rangle]$, where s is an *arbitrary* simply typed λ-term from the point of view of u_i. We thus create a reduction relation over simply typed λ-terms that simulates the reception "out of the blue" of this kind of messages. Namely, we extend the reduction relation of λ-calculus by two new rules: i) $\overline{a}^{\,T} \rightsquigarrow \underline{a}^{\,T}$; ii) $\underline{a}^{\,T} \rightsquigarrow t$ for every channel a^T and simply typed λ-term $t : T$ that does not contain channels. In order to simulate the reception of an arbitrary batch w of messages, t will be instantiated as $\lambda x\, \langle x, w\rangle$ in the proof of Th. 24. With these reductions, λ-terms that do not contain channels are the usual deterministic ones.

Definition 12 (Deterministic simply typed λ-terms)**.** *A simply typed λ-term t is called* ***deterministic****, if t does not contain any channel occurrence.*

Definition 13 (The non-deterministic reduction relation $\rightsquigarrow$)**.** *The reduction relation $\rightsquigarrow$ over simply typed λ-terms is defined as extension of the relation $\mapsto$ as follows:*

$$(\lambda x^A\, u)\, t \rightsquigarrow u[t/x^A] \qquad \langle u_0, u_1\rangle\, \pi_i \rightsquigarrow u_i,\ \textit{for } i = 0, 1 \qquad \overline{a}^{\,T} \rightsquigarrow \underline{a}^{\,T}$$

$$\underline{a}^{\,T} \rightsquigarrow t, \quad \textit{for every channel } a^T \textit{ and deterministic simply typed } \lambda\textit{-term } t : T$$

and as usual we close by contexts: $\mathcal{C}[t] \rightsquigarrow \mathcal{C}[u]$ whenever $t \rightsquigarrow u$ and $\mathcal{C}[\,]$ is a simple context. We denote by $\rightsquigarrow^*$ the reflexive and transitive closure of the one-step reduction $\rightsquigarrow$.

The plan of our proof will be to prove the strong normalization of simply typed λ-calculus with respect to the reduction $\rightsquigarrow$ (Corollary 23) and then derive the strong normalization of $\lambda_{\|}$ using $\rightsquigarrow$ as the source of the complexity measure (Theorem 22). The first result will be proved by the standard Tait–Girard reducibility technique (Definition 14).

We define $\mathsf{SN}^\star$ to be the set of strongly normalizing simply typed λ-terms with respect to the non-deterministic reduction $\rightsquigarrow$. The reduction tree of a strongly normalizable term with respect to $\rightsquigarrow$ is no more finite, but still well-founded. It is well-known that it is possible to assign to each node of a well-founded tree an ordinal number, in such a way that it decreases passing from a node to any of its children. We will call the *ordinal size* of a term $t \in \mathsf{SN}^\star$ the ordinal number assigned to the root of its reduction tree and we denote it by $h(t)$; thus, if $t \rightsquigarrow u$, then $h(t) > h(u)$. To fix ideas, one may define $h(t) := \mathsf{sup}\{h(u) + 1 \mid t \rightsquigarrow u\}$.

5.2 Reducibility and Properties of Reducible Terms

We define a notion of reducibility for simply typed λ-terms with respect to the reduction $\rightsquigarrow$. As usual, we prove that every reducible term is strongly normalizable w.r.t. $\rightsquigarrow$ and afterwards that all simply typed λ-terms are reducible. The difference with the usual reducibility proof is that we first prove that deterministic simply typed λ-terms are reducible (Th. 19), which makes it possible to prove that channels are reducible (Prop. 21) and finally that all terms are reducible (Th. 22). This amounts to prove the usual Adequacy Theorem twice. This stratification of the proof reminds of the reducibility technique employed in [1] for proving weak normalization of a concurrent λ-calculus with shared memory.

Definition 14 (Reducibility)**.** *Assume $t : C$ is a simply typed λ-term. We define the relation $t\ \mathsf{r}\ C$ ("t is reducible of type C") by induction and by cases according to the form of C:*

1. *$t \mathrel{\mathsf{r}} \mathsf{P}$, with P atomic, if and only if $t \in \mathsf{SN}^\star$*
2. *$t \mathrel{\mathsf{r}} A \wedge B$ if and only if $t\pi_0 \mathrel{\mathsf{r}} A$ and $t\pi_1 \mathrel{\mathsf{r}} B$*
3. *$t \mathrel{\mathsf{r}} A \to B$ if and only if for all u, if $u \mathrel{\mathsf{r}} A$, then $tu \mathrel{\mathsf{r}} B$*

We show that the set of reducible terms for a formula C is a reducibility candidate [16]. Neutral terms are defined as terms that are not "values" and need to be further reduced.

Definition 15. *A simply typed λ-term not of the form $\lambda x\, u$ or $\langle u, t\rangle$ is neutral.*

Definition 16 (Reducibility Candidates). *Extending the approach of [16], we define three properties of reducible terms t: **(CR1)** If $t \mathrel{\mathsf{r}} A$, then $t \in \mathsf{SN}^\star$, **(CR2)** If $t \mathrel{\mathsf{r}} A$ and $t \leadsto^* t'$, then $t' \mathrel{\mathsf{r}} A$, and **(CR3)** If t is neutral and for every t', $t \leadsto t'$ implies $t' \mathrel{\mathsf{r}} A$, then $t \mathrel{\mathsf{r}} A$.*

We show that every term t possesses the reducibility candidate properties. The arguments for **(CR1)**, **(CR2)**, **(CR3)** are standard (see [16]).

Proposition 17 ([16]). *Every simply typed λ-term satisfies **(CR1)**, **(CR2)**, **(CR3)**.*

The next task is to prove that all introduction rules of simply typed λ-calculus define a reducible term from a list of reducible terms for all premises.

Proposition 18. *(1) If for every $t \mathrel{\mathsf{r}} A$, $u[t/x] \mathrel{\mathsf{r}} B$, then $\lambda x\, u \mathrel{\mathsf{r}} A \to B$ and (2) If $u \mathrel{\mathsf{r}} A$ and $v \mathrel{\mathsf{r}} B$, then $\langle u, v\rangle \mathrel{\mathsf{r}} A \wedge B$.*

Proof. As in [16], using **(CR1)**, **(CR2)** and **(CR3)**. □

5.3 The Mini Adequacy Theorem

We prove that simply typed λ-terms that do not contain channels are reducible.

Theorem 19 (Mini Adequacy Theorem). *Suppose that $w : A$ is a deterministic simply typed λ-term, with intuitionistic free variables among $x_1 : A_1, \dots, x_n : A_n$. For all terms $t_1, \dots, t_n$ such that for $i = 1, \dots, n$, $t_i \mathrel{\mathsf{r}} A_i$ we have $w[t_1/x_1 \cdots t_n/x_n] \mathrel{\mathsf{r}} A$*

Proof. As the proof of Th. 22 without case 2., which concerns channels. □

Corollary 20 (Mini Strong Normalization of $\leadsto$). *If $t : A$ is a deterministic simply typed λ term, then $t \mathrel{\mathsf{r}} A$ and $t \in \mathsf{SN}^\star$.*

Proof. Assume $x_1 : A_1, \dots, x_n : A_n$ are all the intuitionistic free variables of t and thus all its free variables. By **(CR3)**, one has $x_i \mathrel{\mathsf{r}} A_i$, for $i = 1, \dots, n$. From Theorem 19, we derive $t \mathrel{\mathsf{r}} A$. From **(CR1)**, we conclude that $t \in \mathsf{SN}^\star$. □

By the Mini Adequacy Theorem we can prove that channels are reducible.

Proposition 21 (Reducibility of Channels). *i) For every input channel $\underline{a} : T$, $\underline{a} \mathrel{\mathsf{r}} T$, and ii) For every output channel $\overline{a} : T$, $\overline{a} \mathrel{\mathsf{r}} T$*

Proof. i) Since $\underline{a}$ is neutral, by **(CR3)** it is enough to show that for all u such that $\underline{a} \leadsto u$, it holds that $u \mathrel{\mathsf{r}} T$. Indeed, let us consider any u such that $\underline{a} \leadsto u$; since u must be a deterministic simply typed λ-term, by Corollary 20 $u \mathrel{\mathsf{r}} T$.

ii) Since $\overline{a}$ neutral, by **(CR3)** it is enough to show that for all u such that $\overline{a} \leadsto u$, it holds $u \mathrel{\mathsf{r}} T$. Indeed, let us consider any u such that $\overline{a} \leadsto u$; since $u = \underline{a}$, by i) we have $u \mathrel{\mathsf{r}} T$. □

We can finally prove that all simply typed λ-terms are reducible.

Theorem 22 (Adequacy Theorem)**.** *Suppose that $w : A$ is any simply typed λ-term, with intuitionistic free variables among $x_1 : A_1, \ldots, x_n : A_n$. For all terms $t_1, \ldots, t_n$ such that for $i = 1, \ldots, n$, $t_i \Vdash A_i$ we have $w[t_1/x_1 \cdots t_n/x_n] \Vdash A$*

Corollary 23 (Strong Normalization of $\rightsquigarrow$)**.** *For any simply typed λ-term $t : A$, then $t \in \mathsf{SN}^\star$.*

Proof. Let $x_1 : A_1, \ldots, x_n : A_n$ be all the intuitionistic free variables of t. **(CR3)** leads to $x_i \Vdash A_i$. From Th. 22, we derive $t \Vdash A$. $t \in \mathsf{SN}^\star$ follows from **(CR1)**. □

Theorem 24 (Strong Normalization of $\lambda_{\|}$)**.** *For any $\lambda_{\|}$ term t, $t \in \mathsf{SN}$.*

Proof. Assume $t = {}_a(u_1 \parallel \ldots \parallel u_m)$ and $u_1 = v_1 \parallel \ldots \parallel v_n, \ldots, u_m = v_p \parallel \ldots \parallel v_q$. Define ρ_i, for $i \in \{1, \ldots, q\}$, as the ordinal size $h(v_i)$ of v_i with respect to the reduction relation $\rightsquigarrow$. We proceed by lexicographic induction on the sequence $\rho = (\rho_1, \ldots, \rho_q)$, which we call the complexity of t. Let t' be any term such that $t \mapsto t'$: to prove the thesis, it is enough to show that $t' \in \mathsf{SN}$. Let ρ' the complexity of t'. We must consider three cases.

1. $t' = {}_a(u_1 \parallel \ldots \parallel u'_k \parallel \ldots \parallel u_m)$ and in turn $u'_k = v_i \parallel \ldots \parallel v'_r \parallel \ldots \parallel v_j$ with $v_r \mapsto v_{r'}$ by contraction of an intuitionistic redex. Then also $v_r \rightsquigarrow v'_r$. Hence ρ' is lexicographically strictly smaller than ρ and we conclude by the i.h. that $t' \in \mathsf{SN}$.

2. $t' = {}_a(u'_1 \parallel \ldots \parallel u'_k \parallel \ldots \parallel u'_m) \quad u_k = \ldots \parallel v_i \parallel \ldots \parallel v_j \parallel \ldots \quad u'_k = \ldots \parallel v'_i \parallel \ldots \parallel v'_j \parallel \ldots$
$v_i = \mathcal{D}_1[\underline{a}\ w_1], \ldots, v_j = \mathcal{D}_n[\underline{a}\ w_n] \qquad v'_i = \mathcal{D}_1[\langle w_1, w\rangle], \ldots v_j = \mathcal{D}_n[\langle w_n, w\rangle]$
where w is the sequence of messages transmitted by cross reduction. Being for each l

$$\mathcal{D}_l[\underline{a}\ w_l] \rightsquigarrow \mathcal{D}_l[(\lambda x\, \langle x, w\rangle)\, w_l] \rightsquigarrow \mathcal{D}_l[\langle w_l, w\rangle]$$

we obtain $v_i \rightsquigarrow^* v'_i, \ldots v_j \rightsquigarrow^* v'_j$. Moreover, for each $l \neq k$, either $u_l = u'_l$ or
$u_l = \ldots \parallel s_i \parallel \ldots \parallel s_j \parallel \ldots \quad u'_k = \ldots \parallel s_i \parallel \ldots \parallel s'_j \parallel \ldots \quad s_j = \mathcal{C}[\overline{a}\ r] \rightsquigarrow s'_j = \mathcal{C}[\underline{a}\ r]$

Hence ρ' is lexicographically strictly smaller than ρ and by i.h. $t' \in \mathsf{SN}$.

3. $t' = v_{i_1} \parallel \ldots \parallel v_{i_k}$. As $v_{i_1}, \ldots, v_{i_k}$ all belong to SN, we easily obtain $t' \in \mathsf{SN}$. □

6 Computing with $\lambda_{\|}$

We illustrate the expressive power of $\lambda_{\|}$ with examples of parallel programs from [21].

Reductions are performed according to the principles (a)–(c) below, that make programming in $\lambda_{\|}$ as efficient and deterministic as possible. Indeed, as put by Harper [17], whereas concurrency is concerned with nondeterministic composition of programs, parallelism is concerned with asymptotic *efficiency* of programs with *deterministic behavior*.

(a) Messages should be normalized before being sent. This property is fundamental for efficient parallel computation. Indeed, for instance, a repeatedly forwarded message can be duplicated many times inside a process network; hence if the message is not normal, each process may have to normalize it, wasting resources. This implies in particular that message transmission cannot be triggered according to a call-by-value strategy.

(b) The senders and the receiver of the message should be normalized before the communication. Indeed, the communication behaviour depends on the syntactic shape of a process: it is the rightmost occurrence of the channel in a $\lambda_{\|}$ term to determine what message is sent or received. Hence, a term might send a message, whereas its normal form another. For example $(\lambda x\, y\, x\, (an))(am)$ would transmit m, while its normal form $y\,(am)\,(an)$ would transmit n. We want to avoid this kind of unpredictable behaviour due to the non-confluence of the calculus and make clear what channel is supposed to be the active one, which sends or receives a message.

(c) When no communication nor intuitionistic reduction is possible, we extract the desired result of the computation by a suitable simplification reduction.

Informally, our normalization strategy consists in iterating the basic reduction relation $\succ$ defined below, that takes a term $t = {}_a(u_1 \parallel \ldots \parallel u_m)$ and performs the following operations:

1) We select all the threads of $u_1, \ldots, u_m$ that will send or receive the next message and we let them complete their internal computations.

2) We let the selected threads transmit their message.

3) While executing 1) and 2), to avoid inactivity we let the other threads perform some other independent calculations that may be carried out in parallel.

4) If the previous operations are not possible, we extract the results, if any.

Definition 25 (Reduction Strategy $\succ$)**.** *Let $t = {}_a(u_1 \parallel \ldots \parallel u_m)$ be a $\lambda_{\parallel}$ term. We write $t \succ t'$ whenever t' has been obtained from t by applying on of the following:*

1. *We select a receiver u_i for the next communication. We normalize, among the threads that contain a, those that are rightmost in u_i or in a process outlinked to u_i. If now it is possible, we apply a cross reduction*
$$ {}_a(u_1 \parallel \ldots \parallel u_i \parallel \ldots \parallel u_m) \mapsto {}_a(u_1 \parallel \ldots \parallel u'_i \parallel \ldots \parallel u_m)$$
followed by some intuitionistic reductions.

2. *Provided that by 1. we can only obtain the trivial reduction $t \mapsto^* t$, we apply, if possible, a simplification reduction. We then normalize the remaining simply typed λ-terms.*

We can reduce every $\lambda_{\parallel}$-term in normal form just by iterating the reduction relation $\succ$. Indeed, if a communication is possible, by 1. we can select a suitable receiver and apply a cross reduction. If no communication is possible but a simplification can be done, 2. applies and we can simplify the term. Otherwise, we can just normalize the simply typed λ-terms by 1. or 2.

This normalization strategy leaves some room for non-determinism: it prescribes when a communication reduction should be fired, but does not select a process out of those that can potentially receive messages, thus leaving a number of possible ways of actually performing the communication. To limit this non-determinism, before the beginning of the computation we select one process and we impose that only the selected process can receive messages. Immediately after the reception of the messages, we select the next process to the right, if any, or the first process from left otherwise. Another source of non-determinism is due to cross reductions of the form
$$ {}^{\mathcal{A}}_{a}((u_{m_1} \parallel \ldots \parallel u_{n_1}) \parallel \ldots \parallel (u_{m_p} \parallel \ldots \parallel u_{n_p})) \mapsto u_{j_1} \parallel \ldots \parallel u_{j_q}$$
In this case, we impose that $u_{j_1} \parallel \ldots \parallel u_{j_q}$ results by selecting from each $(u_{m_i} \parallel \ldots \parallel u_{n_i})$, with $1 \leq i \leq p$, the leftmost thread not containing a.

Before presenting the parallel programs for computing π and the all-pair-shortest-path problem, we show that $\lambda_{\parallel}$ is more expressive than simply typed λ calculus.

Example 26 (Parallel OR)**.** *Berry's sequentiality theorem [8] implies that there is no simply typed λ-term O : $\mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Bool}$ such that $O\,\mathbf{ff}\,\mathbf{ff} \mapsto^* \mathbf{ff}$, $O\,u\,\mathbf{tt} \mapsto^* \mathbf{tt}$, $O\,\mathbf{tt}\,u \mapsto^* \mathbf{tt}$ for every λ-term u, where $\mathbf{tt}, \mathbf{ff}$ are the boolean constants. As a consequence, there cannot be a λ-term O such that $\mathsf{O}[\mathbf{ff}/x][\mathbf{ff}/y] \mapsto^* \mathbf{ff}$, $\mathsf{O}[u/x][\mathbf{tt}/y] \mapsto^* \mathbf{tt}$, $\mathsf{O}[\mathbf{tt}/x][u/y] \mapsto^* \mathbf{tt}$ for every λ-term u. To implement a $\lambda_{\parallel}$-term with the above property we process the two inputs in parallel. If both inputs evaluate to $\mathbf{ff}$, though, at least one process needs to have all the information in order to output the result $\mathbf{ff}$. Hence the simple topology $\circ\!\longrightarrow\!\circ$ is enough. Proc. 1 extracts from this graph the axiom schema $(\mathbb{A} \to \mathbb{A} \wedge \bot) \vee (\mathbb{B} \to \mathbb{B} \wedge \mathbb{A})$ with reduction*
$$ {}_a(\mathcal{C}[\overline{a}\,w] \parallel (\mathcal{D}_1[\underline{a}\,v_1] \parallel \ldots \parallel \mathcal{D}_n[\underline{a}\,v_n])) \;\mapsto\; {}_a(\mathcal{C}[\underline{a}\,w] \parallel (\mathcal{D}_1[\langle v_1, w\rangle] \parallel \ldots \parallel \mathcal{D}_n[\langle v_n, w\rangle]))$$
We add to $\lambda_{\parallel}$ the boolean type, $\mathbf{tt}, \mathbf{ff}$ and an if_then_else_ *construct [16]. We define in $\lambda_{\parallel}$*

$$\mathsf{O} := {}_a(\mathsf{if}\ x\ \mathsf{then}\ \mathbf{tt}\ \mathsf{else}\ \overline{a}\ \mathbf{ff}\ \pi_0 \parallel \mathsf{if}\ y\ \mathsf{then}\ \mathbf{tt}\ \mathsf{else}\ \underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1)$$

where we assume $\overline{a} : \mathsf{Bool} \to \mathsf{Bool} \wedge \perp$ *in the first process and that* $\underline{a} : \top \to \top \wedge \mathsf{Bool}$ *in the second one. Now, on one hand*

$$\mathsf{O}[u/x][\mathbf{tt}/y] = {}_a(\mathsf{if}\ u\ \mathsf{then}\ \mathbf{tt}\ \mathsf{else}\ \overline{a}\ \mathbf{ff}\pi_0 \parallel \mathsf{if}\ \mathbf{tt}\ \mathsf{then}\ \mathbf{tt}\ \mathsf{else}\ \underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1) \mapsto^* {}_a(\mathsf{if}\ u\ \mathsf{then}\ \mathbf{tt}\ \mathsf{else}\ \overline{a}\ \mathbf{ff}\pi_0 \parallel \mathbf{tt}) \mapsto \mathbf{tt}$$

and symmetrically $\mathsf{O}[\mathbf{tt}/x][u/y] \mapsto^* \mathbf{tt}$. *On the other hand,* $\mathsf{O}[\mathbf{ff}/x][\mathbf{ff}/y] \mapsto^* \mathbf{ff}$.

Example 27 (A Parallel Program for Computing π)**.** *We implement in* $\lambda_{\parallel}$ *a parallel program for computing arbitrary precise approximations of* π*. As is well known,* π *can be computed as follows:* $\pi = \lim_{l\to\infty} \frac{1}{l}\sum_{i=1}^{l} f(\frac{i-\frac{1}{2}}{l})$ *where* $f(x) = \frac{4}{1+x^2}$*. For any given* l*, instead of calculating sequentially the whole sum* $\sum_{i=1}^{l} f(\frac{i-1/2}{l})$ *and then dividing it by* l*, it is more efficient to distribute different parts of the sum to* p *parallel processes. When the processes terminate, they can send the results* $m_1, \ldots, m_p$*, as shown below, to a process which computes the final result* $\frac{1}{l}(m_1 + \ldots + m_p)$ *(see also [21]). The graph in the following figure, in which we omit reflexive edges, is encoded by the axiom* $\mathcal{A} = (\mathbb{A}_1 \to \mathbb{A}_1 \wedge \perp) \vee \ldots \vee (\mathbb{A}_p \to \mathbb{A}_p \wedge \perp) \vee \mathbb{B} \to (\mathbb{B} \wedge \mathbb{A}_1 \wedge \ldots \wedge \mathbb{A}_p)$.

$$\begin{tikzcd}
\sum_{i=1}^{l/p} f(\frac{i-\frac{1}{2}}{l}) \arrow[drr, "m_1"] & \sum_{i=l/p+1}^{2(l/p)} f(\frac{i-\frac{1}{2}}{l}) \arrow[dr, "m_2"] & \sum_{i=2(l/p)+1}^{3(l/p)} f(\frac{i-\frac{1}{2}}{l}) \arrow[d, "m_3"] & \cdots & \sum_{i=(p-1)(l/p)+1}^{l} f(\frac{i-\frac{1}{2}}{l}) \arrow[dll, bend left, "m_p"] \\
 & & \frac{1}{l}(m_1 + \ldots + m_p) & &
\end{tikzcd}$$

To write the program, we add to $\lambda_{\parallel}$ *types and constants for rational numbers, together with function constants* $\mathsf{f}_1, \ldots, \mathsf{f}_p, \mathsf{sum}$ *such that*

$$\mathsf{f}_k\, l \mapsto^* \sum_{i=(k-1)l/p+1}^{(kl/p)} f(\frac{i-\frac{1}{2}}{l}) \quad \textit{and} \quad \mathsf{sum}\,\langle n_1, \ldots, n_i\rangle\, l \mapsto^* \frac{1}{l}(n_1 + \ldots + n_i)$$

thus computing respectively the p *partial sums and the final result. The* $\lambda_{\parallel}$ *term that takes as input the length* l *of the summation to be carried out and yields the corresponding approximation of* π *is:* ${}_a(\ \overline{a}\,(\mathsf{f}_1\, l)\pi_0 \parallel \ldots \parallel \overline{a}\,(\mathsf{f}_p\, l)\pi_0 \parallel \mathsf{sum}\,(\underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1)\, l\ )$ *. By instantiating* $\mathbb{A}_1, \ldots, \mathbb{A}_p$ *in the extracted* $\mathcal{A}$ *with the type* Q *of rational numbers, we type all displayed occurrences of* a *in the first* p *threads by* $\mathsf{Q} \to \mathsf{Q} \wedge \perp$*, and the last by* $\top \to (\top \wedge \mathsf{Q} \wedge \ldots \wedge \mathsf{Q})$*. Given any multiple* n *of* p*, we have*

$$({}_a(\ \overline{a}\,(\mathsf{f}_1\, l)\, \pi_0 \parallel \ldots \parallel \overline{a}\,(\mathsf{f}_p\, l)\, \pi_0 \parallel \mathsf{sum}\,(\underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1)\, l\ ))[n/l] = {}_a(\ \overline{a}\,(\mathsf{f}_1\, n)\, \pi_0 \parallel \ldots \parallel \overline{a}\,(\mathsf{f}_p\, n)\, \pi_0 \parallel \mathsf{sum}\,(\underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1)\, n\ )$$

$$\mapsto {}_a(\ \overline{a}\,(\sum_{i=1}^{(n/p)} f(\frac{i-\frac{1}{2}}{l}))\, \pi_0 \parallel \ldots \parallel \overline{a}\,(\sum_{i=(p-1)n/p+1}^{n} f(\frac{i-\frac{1}{2}}{l}))\, \pi_0 \parallel \mathsf{sum}\,(\underline{a}\ (\lambda x^{\perp}\, x)\, \pi_1)\, n\ )$$

$$\mapsto^* \mathsf{sum}\,(\langle \sum_{i=1}^{(n/p)} f(\frac{i-\frac{1}{2}}{l}))), \ldots, \sum_{i=(p-1)n/p+1}^{n} f(\frac{i-\frac{1}{2}}{l})))\rangle\, n \quad \mapsto^* \frac{1}{n}\sum_{i=1}^{n} f(\frac{i-\frac{1}{2}}{n})$$

Example 28 (A Parallel Floyd–Warshall Algorithm)**.** *We define a* $\lambda_{\parallel}$ *term that implements a parallel version of the Floyd–Warshall algorithm. The algorithm takes as input a directed graph and outputs a matrix containing the length of the shortest path between each pair of nodes. Formally, the input graph is coded as a matrix* $I(0)$ *and the nodes of the graphs are labeled as* $1, 2, \ldots, n$*. Then the sequential Floyd–Warshall algorithm computes a sequence of matrixes* $I(1), \ldots, I(n)$*, representing closer and closer approximations of the desired output matrix. In particular, the entry* (i, j) *of* $I(k)$ *is the length of the shortest path connecting* i *and* j *such that every node of the path, except for the endpoints, is among the nodes* $1, 2, \ldots k$*. Now, each matrix* $I(k)$ *can be easily computed from* $I(k-1)$*. The idea is that passing through the node* k *might* *be better than passing only through the first $k-1$ nodes, or not. Hence in order to compute $I(k+1)$ from $I(k)$, one only needs to evaluate the right-hand side of the following equation:*

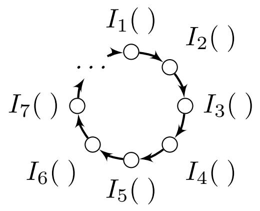
$$I_{i,j}(k) = \mathsf{min}(I_{i,j}(k-1), I_{i,k}(k-1) + I_{k,j}(k-1)) \tag{1}$$

Indeed, $I_{i,j}(k-1)$ represents the shortest path between i, j passing only through the first $k-1$ nodes, while $I_{i,k}(k-1) + I_{k,j}(k-1)$ computes the shortest path between i, j that passes through k and the first $k-1$ nodes. To speed-up this computation, we create a parallel process for each row $I(k)$ of the matrix and put the process in charge of computing the row. We say that the i-th process has to compute the i-th row $I_i(k)$. Which information does it need? Actually, only two rows: $I_i(k-1)$ and $I_k(k-1)$. Hence at each round k the process-i only lacks the row $I_k(k-1)$ to perform its computation. This row can be communicated to process-i by the process-k, in charge to compute that row. These considerations lead to a well-known parallel algorithm.

The ring Floyd–Warshall algorithm

1. Take the input $n \times n$-matrix $I(0)$ and distribute the i-th row $I_i(0)$ to process-i. Organize the n processes in a ring structure such as the following (see [21]), omitting reflexive edges:

(∗)



2. For $k = 1$ to n, starting from process-k, let all the processes forward the row $I_k(k-1)$ to their successors in the ring, until the process $k+1$ receives again the same row. After the row has circulated, let the processes compute in parallel the rows of the matrix $I(k)$.
3. Let $I(n)$ be the output.

In this algorithm, at any stage k, communicating the required row $I_k(k-1)$ through the ring structure requires less time compared to computing each row of the matrix, therefore the overhead of the communication is compensated by the speed-up in the matrix computation.

A $\lambda_{\|}$ program for the Floyd–Warshall algorithm

To write a $\lambda_{\|}$ program that computes the ring Floyd–Warshall Algorithm, we add integers to $\lambda_{\|}$, as usual, and we denote with A the function type corresponding to the rows of the matrixes $I(k)$ computed by the algorithm. The expression $\mathrm{I}_x(y)$ represents the function computing the x^{th} line of the matrix at the y^{th} stage of the algorithm. We assume that the value of $I_x(y)$ contains information about x and y. We also add the constant function $f : A \wedge A \rightarrow A$ such that: $f\langle \mathrm{I}_i(k-1), \mathrm{I}_k(k-1)\rangle = \mathrm{I}_i(k)$ and $f\langle \mathrm{I}_z(l), \mathrm{I}_m(n)\rangle = \mathrm{I}_z(l)$.

The first equation implements the calculations needed for equation (1). The second comes into play at the end of each iteration of the algorithm, when the process that receives twice the same row discards it, starts sending its own row, and begins the next iteration of the algorithm.

As handy notation, for any three terms u, v, s we define $(u,v)^1 s$ as $u(vs)$, and $(u,v)^{n+1}s$ for $n > 0$ as $u(v((u,v)^n s))$. Moreover, for any two terms u, s we define $(u,\pi_i)^1 s$ as $u(s\,\pi_i)$, and $(u,\pi_i)^{n+1}s$ for $n > 0$ as $u(((u,\pi_i)^n s)\pi_i)$. Intuitively, $(u,v)^n s$ represents what we obtain if we take a term s and then apply alternately v and u to the term resulting from the last operation.

We obtain, ultimately, a term of the form $u(v(u(v(\dots u(vs)\dots))))$ in which and u and v occur n times each. The notation $(u,\pi_i)^n s$ is analogous, but instead of applying v we apply the projection π_i. The n processes that run in parallel during the execution of the algorithm are ($1 < i < n$):
Process p_1*:* $\quad (f,\underline{a})^n\, I_1(0) \parallel ((\overline{a}\,,\pi_1)^n\, \underline{a}\, I_1(0))\pi_0 \parallel (\overline{a}\, I_1(0))\pi_0$
Process p_i*:* $(f,\underline{a})^{n+1}\, I_i(0) \parallel ((\overline{a}\,,\pi_1)^{n+1-i}\, (\underline{a}\,,\pi_1)^i\, \underline{a}\; I_i(0))\pi_0 \parallel (\overline{a}\; (f,\underline{a})^i\; I_i(0))\pi_0 \parallel ((\overline{a}\,,\pi_1)^{i-1}\, \underline{a}\, I_i(0))\pi_0$
Process p_n*:* $(f,\underline{a})^{n+1}\, I_n(0) \parallel (\overline{a}\; ((f,\underline{a})^n\; I_n(0)))\pi_0 \parallel ((\overline{a}\,,\pi_1)^{n-1}\, \underline{a}\, I_n(0))\pi_0$

For an intuitive reading of the parts of these terms, consider the notation $(f,\underline{a})^m\, t$. This notation represents a term of the form $f(\underline{a}\,(\dots f(\underline{a}\,t)\dots))$. Only one operation can be immediately performed with a term like this: using the innermost application of a and consume at to receive a message. The received message will be the argument of the innermost f. The value that f computes, in turn, will be the argument of the next communication channel. Terms of this form alternate two phases: one in which they receive, and one in which they apply f to the received message. The terms $(\overline{a}\,,\pi_1)^m\, t$ have the form $\overline{a}\,((\dots\overline{a}\,(t\pi_1)\dots)\pi_1)$. These terms project, send and receive; and then start over. The projections are used to select messages from the tuple of received messages. The selected messages are not used, but forwarded to another process.

Each process p_i ($i > 1$) is a parallel composition of four threads: the first and the third compute the row $I_i(k)$, and the second and the fourth send and forward rows. The process p_1 behaves in the same way, but has three threads: the first computes the rows $I_1(k)$, the second receives and forwards rows, and the third sends its own row. The term implementing the algorithm is $_a(p_1 \parallel \dots \parallel p_n)$. The axiom extracted from the ring structure above by Proc. 1 is $\mathcal{A} = (\mathbb{A}_1 \to \mathbb{A}_1 \wedge \mathbb{A}_n) \vee (\mathbb{A}_2 \to \mathbb{A}_2 \wedge \mathbb{A}_1) \vee \dots \vee (\mathbb{A}_{n-1} \to \mathbb{A}_{n-1} \wedge \mathbb{A}_{n-2}) \vee (\mathbb{A}_n \to \mathbb{A}_n \wedge \mathbb{A}_{n-1})$. We instantiate $\mathbb{A}_1,\dots,\mathbb{A}_n$ in $\mathcal{A}$ with A and type a by $A \to A \wedge A$.

We present some steps of the execution of the instance $_a(p_1 \parallel p_2 \parallel p_3)$. According to our normalization principles, we normalize the threads sending messages right before they communicate; then, we normalize the rightmost threads receiving the messages. All other **IL** *reductions are performed in-between communications. Finally, we make sure to contract all intuitionistic redexes before the beginning of a new iteration of the algorithm. We start with:*

$$\begin{aligned}
&\mapsto^*{}_a\Big(\big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_1(0)))))) \parallel (\overline{a}\,((\overline{a}\,((\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_1))\pi_0 \parallel (\overline{a}\,I_1(0))\pi_0\big)\\
&\quad \parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_2(0)))))))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,((\underline{a}\,((\underline{a}\,I_2(0))\pi_1))\pi_1))\pi_1))\pi_1))\pi_0\\
&\qquad \parallel (\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_2(0))))))\pi_0 \parallel (\overline{a}\,((\underline{a}\,I_2(0))\pi_1))\pi_0\big)\\
&\quad \parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0)))))))) \parallel (\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0))))))))\pi_0 \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,I_3(0))\pi_1))\pi_1))\pi_0\big)\Big)
\end{aligned}$$

We first transmit the value $I_1(0)$ from the first process to the second process and we move the focus to the next term. As an aid to the reader, we display between * * the occurrences of the message that are involved in the reduction:

$$\begin{aligned}
&{}_a\Big(\big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_1(0)))))) \parallel (\overline{a}\,((\overline{a}\,((\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_1))\pi_0 \parallel ({}^\star\underline{a}\,I_1(0)^\star)\pi_0\big)\\
&\parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f\langle I_2(0),{}^\star\, I_1(0)^\star\rangle))))))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,((\underline{a}\,(\langle I_2(0),{}^\star\, I_1(0)^\star\rangle\pi_1))\pi_1))\pi_1))\pi_1))\pi_0\\
&\quad \parallel (\overline{a}\,(f(\underline{a}\,(f\langle I_2(0),{}^\star\, I_1(0)^\star\rangle)))))\pi_0 \parallel (\overline{a}\,(\langle I_2(0),{}^\star\, I_1(0)^\star\rangle\pi_1))\pi_0\big)\\
&\parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0))))))))(\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0))))))))\pi_0 \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,(\langle 0, I_3(0)\rangle\pi_1))\pi_1))\pi_1))\pi_0\big)\Big)
\end{aligned}$$

Remark 29. *$I_2(0)$ is not destroyed by the communication but saved using the memorization mechanism. This term is needed, indeed, by the function f in order to compute $I_2(1)$.*

We normalize the receiver of the previous communication and keep normalizing the other redexes in parallel, thus the rightmost thread of the second process become ready to forward the value $I_1(0)$ to the third thread:

$$\begin{aligned}
&\mapsto^*{}_a\Big(\big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_1(0)))))) \parallel (\overline{a}\,((\overline{a}\,((\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_1))\pi_0 \parallel (\underline{a}\,(I_1(0))\big)\big)\pi_0\\
&\quad \parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f\langle I_2(0), I_1(0)\rangle))))))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_1))\pi_0\\
&\qquad \parallel (\overline{a}\,(f(\underline{a}\,(f\langle I_2(0), I_1(0)\rangle)))))\pi_0 \parallel (\;{}^\star\overline{a}\,I_1(0)^\star\;)\pi_0\big)\\
&\quad \parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0)))))))) \parallel (\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(0))))))))\pi_0 \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,I_3(0))\pi_1))\pi_1))\pi_0\big)\Big)
\end{aligned}$$

The third thread receives $I_1(0)$ and the computation continues: $\mapsto \dots$
At the end of the first of the three cycles, the second process receives $I_1(0)$ again:

$$\begin{aligned}\mapsto^*{}_a\big(&(f(\underline{a}\,(f(\underline{a}\,f\langle I_1(0), I_1(0)\rangle)))) \parallel (\overline{a}\,((\overline{a}\,((\,^\star\underline{a}\,I_1(0)^\star\,)\pi_1))\pi_1))\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(f(\underline{a}\,(f(\underline{a}\,(f\langle I_2(0),\,^\star I_1(0)^\star\,\rangle)))))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,(\langle I_1(0),\,^\star I_1(0)^\star\,\rangle\pi_1)\pi_1))\pi_1))\pi_0\\ &\quad \parallel (\overline{a}\,(f\langle I_2(0),\,^\star I_1(0)^\star\,\rangle)))\pi_0 \parallel \langle I_1(0),\,^\star I_1(0)^\star\,\rangle\pi_0\big)\\ &\parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,f\langle I_3(0), I_1(0)\rangle)))))) \parallel (\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,f\langle I_3(0), I_1(0)\rangle))))))\pi_0 \parallel (\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_0\big)\big)\end{aligned}$$

The reception of $I_1(0)$ exhausts all forwarding channels of the second process in the rightmost thread and triggers the communication of the $I_2(1)$ just computed, thus starting the second cycle:

$$\begin{aligned}\mapsto^*{}_a\big(&(f(\underline{a}\,(f(\underline{a}\,I_1(1)))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_0 \parallel (\langle I_1(0), I_1(0)\rangle)\pi_0\big)\\ &\parallel \big(f(\underline{a}\,(f(\underline{a}\,I_2(1)))) \parallel (\overline{a}\,((\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_1))\pi_0 \parallel (\,^\star\overline{a}\,I_2(1)^\star\,)\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(f(\underline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(1)))))) \parallel (\overline{a}\,(f(\underline{a}\,(f(\underline{a}\,I_3(1))))))\pi_0 \parallel (\overline{a}\,((\underline{a}\,I_1(0))\pi_1))\pi_0\big)\big) \quad \mapsto \dots\end{aligned}$$

The reduction continues in the same fashion for two other cycles, until the third process does not contain any occurrence of a anymore:

$$\begin{aligned}\mapsto^*{}_a\big(&(f\langle I_1(2), I_3(2)\rangle \parallel (\overline{a}\,I_3(2))\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(f\langle I_2(2), I_3(2)\rangle \parallel (\underline{a}\,I_3(2))\pi_0 \parallel \langle I_2(1), I_2(1)\rangle\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(f\langle I_3(2), I_3(2)\rangle \parallel \langle I_3(2), I_3(2)\rangle\pi_0 \parallel \langle I_2(1), I_2(1)\rangle\pi_0\big)\big)\end{aligned}$$

We then compute all occurrences of f in parallel:

$$\begin{aligned}\mapsto^*{}_a\big(&(I_1(3) \parallel (\overline{a}\,I_3(2))\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(I_2(3) \parallel (\underline{a}\,I_3(2))\pi_0 \parallel \langle I_2(1), I_2(1)\rangle\pi_0 \parallel \langle I_1(0), I_1(0)\rangle\pi_0\big)\\ &\parallel \big(I_3(3) \parallel \langle I_3(2), I_3(2)\rangle\pi_0 \parallel \langle I_2(1), I_2(1)\rangle\pi_0\big)\big)\end{aligned}$$

We use a simplification reduction to remove the intermediate results no more needed and only keep the threads containing the result of the computation: $\mapsto^* I_1(3) \parallel I_2(3) \parallel I_3(3)$ *.*

Conclusions and Related Work

We introduced $\lambda_\parallel$, a simple and yet expressive typed parallel λ-calculus based on (suitable fragments of) propositional intermediate logics. $\lambda_\parallel$ is based on and greatly simplifies the calculi in [4, 3, 2]. The basic communication reductions in the calculi λ_{CL} [3] and λ_G [2] implement *particular* $\lambda_\parallel$ communications: the message passing mechanisms based on classical logic and on Gödel logic, respectively. λ_{CL} and λ_G also contain additional reductions (*permutations* and *full cross*), needed for proving weak normalization and the subformula property. These calculi were generalized in [4], that introduces a Curry–Howard correspondence for all propositional intermediate logics characterized by classical disjunctive tautologies, the axioms for classical and Gödel logic being particular cases. The reductions in [4] are the same as for λ_{CL} and λ_G but their activation procedure is based on transmitting values and thus it is logic independent. The simple(r) $\lambda_\parallel$ reduction allows us to encode interesting parallel programs in Eden – an extension of Haskell [21, 22], to prove strong normalization and to specify process networks in an automated way. The price to pay is the lack of the subformula property and the fact that well-typed $\lambda_\parallel$-terms might contain deadlock. In a model of parallel computation the latter is an unwanted feature, especially in absence[3] of recursion or fixed points. Unlike $\lambda_\parallel$, the typed versions of the π-calculus, starting with the seminal work in [9], are usually deadlock free, e.g., [29, 28, 10].

[3]Note that as the number of $\lambda_\parallel$ communications is not type-determined, the system is compatible with the addition of a recursion operator.

References

[1] R.M. Amadio. On stratified regions. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Proceedings, LNCS*, number 5904, pages 210–225, 2009.

[2] F. Aschieri, A. Ciabattoni, and F.A. Genco. Gödel logic: From natural deduction to parallel computation. In *LICS 2017*, pages 1–12, 2017.

[3] F. Aschieri, A. Ciabattoni, and F.A. Genco. Classical proofs as parallel programs. In *Proceedings of GandALF 2018*, pages 43–57, 2018.

[4] F. Aschieri, A. Ciabattoni, and F.A. Genco. On the concurrent computational content of intermediate logics. *Theoretical Computer Science. https://www.logic.at/staff/agata/lambda-logics.pdf*, 2020.

[5] F. Aschieri and F.A. Genco. Par means parallel: Multiplicative linear logic proofs as concurrent functional programs. *POPL 2020*, 2020.

[6] F. Aschieri and M. Zorzi. On natural deduction in classical first-order logic: Curry–howard correspondence, strong normalization and herbrand's theorem. *Theoret. Comput. Sci.*, 625:125–146, 2016.

[7] A. Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Annals of Mathematics and Artificial Intelligence*, 4(3):225–248, 1991.

[8] H.P. Barendregt. *The lambda calculus.* North-Holland Amsterdam, 1984.

[9] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, pages 222–236, 2010.

[10] M. Carbone, F. Montesi, and C. Schümann. Choreographies, logically. *Distributed Computing*, 31(1):51–67, 2018.

[11] A. Ciabattoni and F.A. Genco. Hypersequents and systems of rules: Embeddings and applications. *ACM TOCL*, 19:1–27, 2018.

[12] M. D'Agostino. An informational view of classical logic. *Theor. Comp. Sci.*, 606:79–97, 2015.

[13] M. D'Agostino, M. Finger, and Dov M. Gabbay. Semantics and proof-theory of depth bounded boolean logics. *Theor. Comp. Sci.*, 480:43–68, 2013.

[14] V. Danos and J.-L. Krivine. Disjunctive tautologies as synchronisation schemes. In *CSL 2000*, pages 292–301, 2000.

[15] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding code mobility. In *IEEE Transactions on Software Engineering*, volume 24, pages 342–361, 1998.

[16] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[17] R. Harper. Parallelism is not concurrency. Available at `https://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency`, 2011.

[18] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP 1991*, pages 133–147, 1991.

[19] T. Horstmeyer and R. Loogen. Graph-based communication in Eden. *Higher-order and symbolic computation*, 26(1):3–28, 2013.

[20] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[21] R. Loogen. Eden – parallel functional programming with Haskell. In *CEFP 2011*, pages 142–206, 2011.

[22] R. Loogen, Y. Ortega-Mallèn, and R. Pena. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[23] E. G. K. López-Escobar. Implicational logics in natural deduction systems. *Journal of Symbolic Logic*, 47(1):184–186, 1982.

[24] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[25] T. Murphy VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS 2004*, pages 286–295, 2004.

[26] M. Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR 1992*, pages 190–201, 1992.

[27] D. Prawitz. Ideas and results in proof theory. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 237–309. North-Holland, 1971.

[28] B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP 2013*, pages 350–369, 2013.

[29] P. Wadler. Propositions as sessions. *ICFP 2012*, 24:384–418, 2012.

[30] P. Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.